The OpenCL™ Specification

Khronos® OpenCL Working Group

Version v3.1.1, Fri, 22 May 2026 12:00:00 +0000: from git branch: main commit: 9f68efb5d80e77a4e437c5b8ee67d581666d044c

Table of Contents

1. Introduction . . . 2
   1.1. Version Numbers . . . 3
   1.2. Unified Specification . . . 3
   1.3. Document Conventions . . . 3
      1.3.1. Normative Requirements . . . 3
      1.3.2. Error Codes . . . 4
      1.3.3. String Representation . . . 5
2. Glossary . . . 6
3. The OpenCL Architecture . . . 20
   3.1. Platform Model . . . 20
   3.2. Execution Model . . . 21
      3.2.1. Mapping Work-items Onto an ND-range . . . 25
      3.2.2. Execution of Kernel-instances . . . 27
      3.2.3. Device-Side Enqueue . . . 28
      3.2.4. Synchronization . . . 29
      3.2.5. Categories of Kernels . . . 30
   3.3. Memory Model . . . 31
      3.3.1. Fundamental Memory Regions . . . 32
      3.3.2. Memory Objects . . . 34
      3.3.3. Lifetime of Shared Direct3D Memory Objects . . . 36
      3.3.4. Lifetime of Shared OpenCL/OpenGL Memory Objects . . . 36
      3.3.5. Shared Virtual Memory . . . 37
      3.3.6. Memory Consistency Model for OpenCL 1.x . . . 38
      3.3.7. Memory Consistency Model for OpenCL 2.x . . . 38
      3.3.8. Overview of Atomic and Fence Operations . . . 41
      3.3.9. Memory Ordering Rules . . . 42
   3.4. The OpenCL Framework . . . 53
      3.4.1. Mixed Version Support . . . 53
      3.4.2. Backwards Compatibility . . . 54
      3.4.3. Versioning . . . 55
      3.4.4. Valid Usage and Undefined Behavior . . . 57
4. The OpenCL Platform Layer . . . 58
   4.1. Querying Platform Info . . . 58
   4.2. Querying Devices . . . 64
      4.2.1. Required OpenCL Extensions . . . 105
      4.2.2. Sharing DirectX9 Media Surfaces With OpenCL Images . . . 105
      4.2.3. Sharing Direct3D 10 Resources With OpenCL Memory Objects . . . 108
      4.2.4. Sharing Direct3D 11 Resources With OpenCL Memory Objects . . . 110

4.3. Partitioning a Device . . . 112
4.4. Contexts . . . 117
4.5. Extension Function Pointers . . . 131
5. The OpenCL Runtime . . . 133
5.1. Command-Queues . . . 133
5.2. Buffer Objects . . . 143
5.2.1. Creating Buffer Objects . . . 143
5.2.2. Reading, Writing and Copying Buffer Objects . . . 152
5.2.3. Filling Buffer Objects . . . 165
5.2.4. Mapping Buffer Objects . . . 167
5.2.5. Creating Buffer Objects From Direct3D Buffer Resources . . . 170
5.2.6. Creating Buffer Objects From OpenGL Buffer Objects . . . 172
5.3. Image Objects . . . 174
5.3.1. Creating Image Objects . . . 174
5.3.2. Querying List of Supported Image Formats . . . 191
5.3.3. Mapping to External Image Formats . . . 195
5.3.4. Reading, Writing and Copying Image Objects . . . 200
5.3.5. Filling Image Objects . . . 206
5.3.6. Copying Between Image and Buffer Objects . . . 209
5.3.7. Mapping Image Objects . . . 214
5.3.8. Specifying Mipmap Levels to Image Operations . . . 218
5.3.9. Image Object Queries . . . 218
5.3.10. Creating Image Objects From DirectX 9 Media Resources . . . 221
5.3.11. Creating Image Objects From Direct3D Textures and Resources . . . 223
5.3.12. Creating Image Objects From EGL Images . . . 227
5.3.13. Creating Image Objects From OpenGL Textures and Renderbuffers . . . 229
5.3.14. Querying Image Requirements . . . 233
5.4. Pipes . . . 238
5.4.1. Creating Pipe Objects . . . 238
5.4.2. Pipe Object Queries . . . 239
5.5. Memory Objects . . . 241
5.5.1. Retaining and Releasing Memory Objects . . . 241
5.5.2. Descriptions of External Memory Handle Types . . . 248
5.5.3. Unmapping Mapped Memory Objects . . . 249
5.5.4. Accessing Mapped Regions of a Memory Object . . . 251
5.5.5. Migrating Memory Objects . . . 252
5.5.6. Memory Object Queries . . . 254
5.5.7. Querying Media Surface Properties of Memory Objects Created From DirectX 9 Media Surfaces . . . 259
5.5.8. Querying Direct3D Properties of Memory Objects Created From Direct3D Resources . . . 259
5.5.9. Querying OpenGL Object Information From an OpenCL Memory Object . . . 259

5.5.10. Sharing Memory Objects Created From Media Surfaces Between a Media Adapter and OpenCL . . . 261
5.5.11. Sharing Memory Objects Created From Direct3D Resources Between Direct3D and OpenCL Contexts . . . 264
5.5.12. Sharing Memory Objects Created From EGL Resources Between EGL and OpenCL Contexts . . . 270
5.5.13. Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects . . . 273
5.6. Shared Virtual Memory . . . 277
5.6.1. SVM Sharing Granularity: Coarse- and Fine- Grained Sharing . . . 278
5.6.2. Memory Consistency for SVM Allocations . . . 292
5.7. Sampler Objects . . . 292
5.7.1. Creating Sampler Objects . . . 292
5.7.2. Sampler Object Queries . . . 297
5.8. Program Objects . . . 298
5.8.1. Creating Program Objects . . . 299
5.8.2. Retaining and Releasing Program Objects . . . 303
5.8.3. Setting SPIR-V Specialization Constants . . . 305
5.8.4. Building Program Executables . . . 306
5.8.5. Separate Compilation and Linking of Programs . . . 308
5.8.6. Compiler Options . . . 314
5.8.7. Linker Options . . . 318
5.8.8. Unloading the OpenCL Compiler . . . 320
5.8.9. Program Object Queries . . . 320
5.9. Kernel Objects . . . 328
5.9.1. Creating Kernel Objects . . . 328
5.9.2. Setting Kernel Arguments . . . 331
5.9.3. Copying Kernel Objects . . . 339
5.9.4. Kernel Object Queries . . . 340
5.10. Executing Kernels . . . 354
5.11. Event Objects . . . 362
5.11.1. Creating, Waiting for, and Releasing Event Objects . . . 363
5.12. Markers, Barriers and Waiting for Events . . . 377
5.13. Semaphores . . . 381
5.13.1. Semaphore Types . . . 381
5.13.2. Creating Semaphores . . . 381
5.13.3. Exporting Semaphore External Handles . . . 383
5.13.4. Importing Semaphore External Handles . . . 384
5.13.5. Descriptions of External Semaphore Handle Types . . . 384
5.13.6. Waiting On and Signaling Semaphores . . . 388
5.13.7. Retaining and Releasing Semaphores . . . 391

5.13.8. Semaphore Queries . . . 392
5.14. Out-of-Order Execution of Kernels and Memory Object Commands . . . 394
5.15. Profiling Operations on Memory Objects and Kernels . . . 395
5.16. Flush and Finish . . . 398
5.17. Command-Buffers . . . 399
5.17.1. Command-Buffers and Multiple Devices . . . 401
5.17.2. Command-Buffer Lifecycle . . . 401
5.17.3. Creating Command-Buffer Objects . . . 402
5.17.4. Enqueuing a Command-Buffer . . . 408
5.17.5. Recording Commands to a Command-Buffer . . . 410
5.17.6. Remapping Command-Buffers . . . 434
5.17.7. Mutable Commands . . . 436
5.17.8. Command-Buffer Queries . . . 440
5.18. Querying Devices That Support Sharing With OpenGL . . . 447
6. Associated OpenCL Specifications . . . 450
6.1. The OpenCL C Kernel Language . . . 450
6.2. SPIR-V Intermediate Language . . . 450
6.3. Extensions to OpenCL . . . 450
7. OpenCL Embedded Profile . . . 451
Appendix A: Host environment and thread safety . . . 458
Shared OpenCL Objects . . . 458
Multiple Host Threads . . . 458
Global Constructors and Destructors . . . 459
Appendix B: Portability . . . 460
Appendix C: Application Data Types . . . 465
Supported Application Scalar Data Types . . . 465
Supported Application Vector Data Types . . . 465
Alignment of Application Data Types . . . 466
Vector Literals . . . 466
Vector Components . . . 466
Named Vector Components Notation . . . 467
High/Low Vector Component Notation . . . 467
Native Vector Type Notation . . . 468
Implicit Conversions . . . 468
Explicit Casts . . . 468
Other Operators and Functions . . . 468
Application Constant Definitions . . . 469
Appendix D: Checking for Memory Copy Overlap . . . 472
Appendix E: Changes to OpenCL . . . 474
Summary of Changes from OpenCL 1.0 to OpenCL 1.1 . . . 474
Summary of Changes from OpenCL 1.1 to OpenCL 1.2 . . . 475

Summary of Changes from OpenCL 1.2 to OpenCL 2.0 . . . 477
Summary of Changes from OpenCL 2.0 to OpenCL 2.1 . . . 479
Summary of Changes from OpenCL 2.1 to OpenCL 2.2 . . . 479
Summary of Changes from OpenCL 2.2 to OpenCL 3.0 . . . 480
Summary of Changes from OpenCL 3.0 to OpenCL 3.1 . . . 489
Appendix F: Error Codes . . . 491
Appendix G: Other Miscellaneous Enums . . . 498
Appendix H: OpenCL 3.0 Backwards Compatibility . . . 499
Shared Virtual Memory . . . 499
Memory Consistency Model . . . 499
Device-Side Enqueue . . . 501
Pipes . . . 502
Program Scope Global Variables . . . 502
Non-Uniform Work-groups . . . 503
Read-Write Images . . . 503
Creating 2D Images From Buffers . . . 503
sRGB Images . . . 504
Depth Images . . . 504
Device and Host Timer Synchronization . . . 504
Intermediate Language Programs . . . 504
Sub-groups . . . 505
Program Initialization and Clean-Up Kernels . . . 505
3D Image Writes . . . 506
Work-group Collective Functions . . . 506
Generic Address Space . . . 506
Language Features That Were Already Optional . . . 507
Appendix I: OpenCL Extensions (Informative) . . . 508
Naming Convention for Optional Extensions . . . 508
Header Conventions for Optional Extensions . . . 509
Experimental Extensions . . . 510
Extension Dependencies . . . 510
List of Current Extensions . . . 510
cl_khr_3d_image_writes . . . 514
cl_khr_async_work_group_copy_fence . . . 514
cl_khr_byte_addressable_store . . . 515
cl_khr_create_command_queue . . . 516
cl_khr_d3d10_sharing . . . 517
cl_khr_d3d11_sharing . . . 519
cl_khr_depth_images . . . 521
cl_khr_device_enqueue_local_arg_types . . . 522
cl_khr_device_uuid . . . 522

cl_khr_dx9_media_sharing . . . 523
cl_khr_egl_event . . . 525
cl_khr_egl_image . . . 527
cl_khr_expect_assume . . . 529
cl_khr_extended_async_copies . . . 530
cl_khr_extended_bit_ops . . . 531
cl_khr_extended_versioning . . . 532
cl_khr_external_memory . . . 535
cl_khr_external_memory_dma_buf . . . 541
cl_khr_external_memory_opaque_fd . . . 542
cl_khr_external_memory_win32 . . . 544
cl_khr_external_semaphore . . . 546
cl_khr_external_semaphore_opaque_fd . . . 552
cl_khr_external_semaphore_sync_fd . . . 553
cl_khr_fp16 . . . 555
cl_khr_fp64 . . . 556
cl_khr_gl_depth_images . . . 556
cl_khr_gl_event . . . 557
cl_khr_gl_msaa_sharing . . . 559
cl_khr_gl_sharing . . . 560
cl_khr_global_int32_base_atomics . . . 564
cl_khr_global_int32_extended_atomics . . . 565
cl_khr_icd . . . 566
cl_khr_icd_unloadable . . . 571
cl_khr_il_program . . . 572
cl_khr_image2d_from_buffer . . . 573
cl_khr_initialize_memory . . . 574
cl_khr_int64_base_atomics . . . 575
cl_khr_int64_extended_atomics . . . 576
cl_khr_integer_dot_product . . . 576
cl_khr_kernel_clock . . . 578
cl_khr_local_int32_base_atomics . . . 579
cl_khr_local_int32_extended_atomics . . . 580
cl_khr_mipmap_image . . . 581
cl_khr_mipmap_image_writes . . . 582
cl_khr_pci_bus_info . . . 582
cl_khr_priority_hints . . . 583
cl_khr_select_fprounding_mode . . . 584
cl_khr_semaphore . . . 585
cl_khr_spirv_extended_debug_info . . . 591
cl_khr_spirv_linkonce_odr . . . 591

cl_khr_spirv_no_integer_wrap_decoration . . . 592
cl_khr_spirv_queries . . . 593
cl_khr_srgb_image_writes . . . 593
cl_khr_subgroup_ballot . . . 594
cl_khr_subgroup_clustered_reduce . . . 595
cl_khr_subgroup_extended_types . . . 596
cl_khr_subgroup_named_barrier . . . 597
cl_khr_subgroup_non_uniform_arithmetic . . . 598
cl_khr_subgroup_non_uniform_vote . . . 600
cl_khr_subgroup_rotate . . . 601
cl_khr_subgroup_shuffle . . . 602
cl_khr_subgroup_shuffle_relative . . . 602
cl_khr_subgroups . . . 603
cl_khr_suggested_local_work_size . . . 605
cl_khr_terminate_context . . . 605
cl_khr_throttle_hints . . . 607
cl_khr_work_group_uniform_arithmetic . . . 608
cl_ext_buffer_device_address . . . 609
cl_ext_cxx_for_opencl . . . 611
cl_ext_device_fission . . . 612
cl_ext_float_atomics . . . 613
cl_ext_image_from_buffer . . . 613
cl_ext_image_raw10_raw12 . . . 615
cl_ext_image_requirements_info . . . 616
cl_ext_image_unorm_int_2_101010 . . . 619
cl_ext_image_unsigned_10x6_12x4_14x2 . . . 620
cl_ext_immutable_memory_objects . . . 622
cl_ext_migrate_memobject . . . 623
List of Experimental Extensions . . . 623
cl_khr_command_buffer . . . 624
cl_khr_command_buffer_multi_device . . . 633
cl_khr_command_buffer_mutable_dispatch . . . 639
cl_khr_external_memory_android_hardware_buffer . . . 647
cl_khr_external_semaphore_dx_fence . . . 650
cl_khr_external_semaphore_win32 . . . 651
cl_khr_unified_svm . . . 653
List of Deprecated Extensions . . . 656
cl_khr_spir . . . 657
Acknowledgements . . . 658

Copyright 2008-2026 The Khronos Group Inc.

This Specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos.

This Specification has been created under the Khronos Intellectual Property Rights Policy, which is Attachment A of the Khronos Group Membership Agreement available at www.khronos.org/files/member_agreement.pdf and defines the terms 'Scope', 'Compliant Portion', and 'Necessary Patent Claims'.

Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms. Parties desiring to implement the Specification and make use of Khronos trademarks in relation to that implementation, and receive reciprocal patent license protection under the Khronos Intellectual Property Rights Policy must become Adopters and confirm the implementation as conformant under the process defined by Khronos for this Specification; see https://www.khronos.org/adopters.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this Specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

Where this Specification identifies specific sections of external references, only those specifically identified sections define normative functionality. The Khronos Intellectual Property Rights Policy excludes external references to materials and associated enabling technology not created by Khronos from the Scope of this specification, and any licenses that may be required to implement such referenced materials and associated technologies must be obtained separately and may involve royalty payments.

Khronos® and Vulkan® are registered trademarks, and SPIR™, SPIR-V™, and SYCL™ are trademarks of The Khronos Group Inc. OpenCL™ is a trademark of Apple Inc. used under license by Khronos. OpenGL® is a registered trademark and the OpenGL ES™ and OpenGL SC™ logos are trademarks of Hewlett Packard Enterprise used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Chapter 1. Introduction

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Facing technical challenges with higher clock speeds in a fixed power envelope, Central Processing Units (CPUs) now improve performance by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into programmable parallel processors. As today's computer systems often include highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take full advantage of these heterogeneous processing platforms.

Creating applications for heterogeneous parallel processing platforms is challenging as traditional programming approaches for multi-core CPUs and GPUs are very different. CPU-based parallel programming models are typically based on standards but usually assume a shared address space and do not encompass vector operations. General purpose GPU programming models address complex memory hierarchies and vector operations but are traditionally platform-, vendor- or hardware-specific. These limitations make it difficult for a developer to access the compute power of heterogeneous CPUs, GPUs and other types of processors from a single, multi-platform source code base. More than ever, there is a need to enable software developers to effectively take full advantage of heterogeneous processing platforms from high performance compute servers, through desktop computer systems to handheld devices - that include a diverse mix of parallel CPUs, GPUs and other processors such as DSPs and the Cell/B.E. processor.

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications. OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines.

OpenCL consists of an API for coordinating parallel computation across heterogeneous processors, a cross-platform programming language, and a cross-platform intermediate language with a well-specified computation environment. The OpenCL standard:

- Supports both data- and task-based parallel programming models
- Supports kernels written using a subset of ISO C99 with extensions for parallel execution
- Supports kernels represented by a portable and self-contained intermediate language (e.g. SPIR-V) with support for parallel execution
- Defines consistent numerical requirements based on IEEE 754
- Defines a configuration profile for handheld and embedded devices
- Supports efficient interop with OpenGL, OpenGL ES and other APIs

This document begins with an overview of basic concepts and the architecture of OpenCL, followed by a detailed description of its execution model, memory model and synchronization support. It

then discusses the OpenCL platform and runtime API. Some examples are given that describe sample compute use-cases and how they would be written in OpenCL. The specification is divided into a core specification that any OpenCL compliant implementation must support; a handheld/embedded profile which relaxes the OpenCL compliance requirements for handheld and embedded devices; and a set of optional extensions that are likely to move into the core specification in later revisions of the OpenCL specification.

1.1. Version Numbers

The OpenCL version number follows a *major.minor-revision* scheme. When this version number is used within the API it generally only includes the *major.minor* components of the version number.

A difference in the *major* or *minor* version number indicates that some amount of new functionality has been added to the specification, and may also include behavior changes and bug fixes. Functionality may also be deprecated when the *major* or *minor* version changes or removed when the *major* version changes.

A difference in the *revision* number indicates small changes to the specification, typically to fix a bug or to clarify language. When the *revision* number changes there may be an impact on the behavior of existing functionality, but this should not affect backwards compatibility. Functionality should not be added or removed when the *revision* number changes.

1.2. Unified Specification

This document specifies all versions of the OpenCL API.

There are three ways that an OpenCL feature may be described in terms of what versions of OpenCL support that feature.

- Missing before *major.minor*: Features that were introduced in version *major.minor*. Implementations of an earlier version of OpenCL will not provide these features.
- Deprecated by *major.minor*: Features that were deprecated in version *major.minor*, see the definition of deprecation in the glossary.
- Universal: Features that have no mention of what version they are missing before or deprecated by are available in all versions of OpenCL.

1.3. Document Conventions

The OpenCL specification is intended for use by both implementors of the API and application developers seeking to make use of the API, forming a contract between these parties. The OpenCL specification text may address either party; typically the intended audience can be inferred from context, though some sections address only one of these parties.

1.3.1. Normative Requirements

The OpenCL Specification uses a combination of normative terminology and normative descriptions to express the requirements that it imposes on applications and implementations. An

application which complies with all normative requirements imposed on applications is said to make **valid use** of the API; failing to comply with such requirements may result in undefined behavior, as described in the Valid Usage section. In the context of this document, an implementation which complies with all normative requirements imposed on implementations is said to be **conformant**.

1.3.1.1. Normative Terminology

Within this specification, the key words must, required, should, may, and optional are to be interpreted as described in RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels (https://www.ietf.org/rfc/rfc2119.txt). The additional key word optionally is an alternate form of optional, for use where grammatically appropriate.

1.3.1.2. Normative Descriptions

The normative term must is primarily used to describe **application** behavior, and in particular to constrain what inputs or commands issued by the application to the implementation are considered valid.

To constrain **implementation** behavior, the specification sometimes uses must, but more often simply describes the behavior of the implementation in response to specified commands and inputs. Unless explicitly stated otherwise, such references to implementation behavior describe the behavior of **conformant** implementations, and express normative requirements which an implementation must satisfy in order to conform to the specification.

When the normative terms may, should, or optional are used to describe implementation behavior, they define alternative or optional behaviors which a conformant implementation may or may not exhibit. Such statements are also normative.

1.3.1.3. Normative References

References to external documents are considered normative references if the OpenCL specification uses normative terminology or normative descriptions to refer to them or their requirements, either as a whole or in part.

The following documents are referenced by normative sections of the specification:

ISO/IEC 9899:2011 - Information technology - Programming languages - C, https://www.iso.org/standard/57853.html (final specification), http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1570.pdf (last public draft)

SPIR-V Specification, Version 1.6, Unified. https://registry.khronos.org/spir-v/

1.3.2. Error Codes

In many cases, the OpenCL specification describes conditions when an implementation must return an error code due to invalid application usage. If multiple conditions are described for an error code, the error code must be returned if any of the conditions are met. If error conditions are met for multiple error codes, the implementation may return any of the error conditions.

1.3.3. String Representation

Unless specified otherwise, strings passed to or returned from OpenCL API functions are defined to be null-terminated and UTF-8 encoded.

Chapter 2. Glossary

Application

The combination of the program running on the host and OpenCL devices.

Acquire semantics

One of the memory order semantics defined for synchronization operations. Acquire semantics apply to atomic operations that load from memory. Given two units of execution, **A** and **B**, acting on a shared atomic object **M**, if **A** uses an atomic load of **M** with acquire semantics to synchronize-with an atomic store to **M** by **B** that used release semantics, then **A**'s atomic load will occur before any subsequent operations by **A**. Note that the memory orders *release*, *sequentially consistent*, and *acquire_release* all include *release semantics* and effectively pair with a load using acquire semantics.

Acquire release semantics

A memory order semantics for synchronization operations (such as atomic operations) that has the properties of both acquire and release memory orders. It is used with read-modify-write operations.

Atomic operations

Operations that at any point, and from any perspective, have either occurred completely, or not at all. Memory orders associated with atomic operations may constrain the visibility of loads and stores with respect to the atomic operations (see *relaxed semantics*, *acquire semantics*, *release semantics* or *acquire release semantics*).

Blocking and Non-Blocking Enqueue API calls

A *non-blocking enqueue API call* places a *command* on a *command-queue* and returns immediately to the host. The *blocking-mode enqueue API calls* do not return to the host until the command has completed.

Barrier

There are three types of *barriers* a command-queue barrier, a work-group barrier, and a sub-group barrier.

- The OpenCL API provides a function to enqueue a *command-queue barrier* command. This *barrier* command ensures that all previously enqueued commands to a command-queue have finished execution before any following *commands* enqueued in the *command-queue* can begin execution.
- The OpenCL kernel execution model provides built-in *work-group barrier* functionality. This *barrier* built-in function can be used by a *kernel* executing on a *device* to perform synchronization between *work-items* in a *work-group* executing the *kernel*. All the *work-items* of a *work-group* must execute the *barrier* construct before any are allowed to continue execution beyond the *barrier*.
- The OpenCL kernel execution model provides built-in *sub-group barrier* functionality. This *barrier* built-in function can be used by a *kernel* executing on a *device* to perform synchronization between *work-items* in a *sub-group* executing the *kernel*. All the *work-items* of a *sub-group* must execute the *barrier* construct before any are allowed to continue

execution beyond the *barrier*.

Buffer Object

A memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a *kernel* executing on a *device*. Buffer objects can be manipulated by the host using OpenCL API calls. A *buffer object* encapsulates the following information:

- Size in bytes.
- Properties that describe usage information and which region to allocate from.
- Buffer data.

Built-in Kernel

A *built-in kernel* is a *kernel* that is provided by an OpenCL implementation. A *built-in kernel* is enqueued for execution like other *kernels*, but may execute on specialized hardware that is unavailable to non-built-in kernels. *Applications* can query the *built-in kernels* supported by a *device*.

Child kernel

See *Device-side enqueue*.

Command

The OpenCL operations that are submitted to a *command-queue* for execution. For example, OpenCL commands issue kernels for execution on a compute device, manipulate memory objects, etc.

Command-queue

An object that holds *commands* that will be executed on a specific *device*. The *command-queue* is created on a specific *device* in a *context*. *Commands* to a *command-queue* are queued in-order but may be executed in-order or out-of-order. Refer to *In-order Execution* and *Out-of-order Execution*.

Command-queue Barrier

See *Barrier*.

Command synchronization

Constraints on the order that commands are launched for execution on a device defined in terms of the synchronization points that occur between commands in host command-queues and between commands in device-side command-queues. See *synchronization points*.

Complete

The final state in the six state model for the execution of a command. The transition into this state occurs is signaled through event objects or callback functions associated with a command.

Compute Device Memory

This refers to one or more memories attached to the compute device.

Compute Unit

An OpenCL *device* has one or more *compute units*. A *work-group* executes on a single *compute unit*. A *compute unit* is composed of one or more *processing elements* and *local memory*. A

compute unit may also include dedicated texture filter units that can be accessed by its processing elements.

Concurrency

A property of a system in which a set of tasks in a system can remain active and make progress at the same time. To utilize concurrent execution when running a program, a programmer must identify the concurrency in their problem, expose it within the source code, and then exploit it using a notation that supports concurrency.

Constant Memory

A region of *global memory* that remains constant during the execution of a *kernel*. The *host* allocates and initializes memory objects placed into *constant memory*.

Context

The environment within which the kernels execute and the domain in which synchronization and memory management is defined. The *context* includes a set of *devices*, the memory accessible to those *devices*, the corresponding memory properties and one or more *command-queues* used to schedule execution of a *kernel(s)* or operations on *memory objects*.

Control flow

The flow of instructions executed by a work-item. Multiple logically related work-items may or may not execute the same control flow. The control flow is said to be *converged* if all the work-items in the set execution the same stream of instructions. In a *diverged* control flow, the work-items in the set execute different instructions. At a later point, if a diverged control flow becomes converged, it is said to be a re-converged control flow.

Converged control flow

See *Control flow*.

Custom Device

A *custom device* is a specialized *device* that supports a subset of the OpenCL runtime APIs for directed tasks but is not OpenCL conformant. A *custom device* must implement all of the OpenCL runtime APIs, but may not support all of the required minimum device capabilities and may return implementation-defined error codes for unsupported functionality. See also *Device*.

Data Parallel Programming Model

Traditionally, this term refers to a programming model where concurrency is expressed as instructions from a single program applied to multiple elements within a set of data structures. The term has been generalized in OpenCL to refer to a model wherein a set of instructions from a single program are applied concurrently to each point within an abstract domain of indices.

Data race

The execution of a program contains a data race if it contains two actions in different work-items or host threads where (1) one action modifies a memory location and the other action reads or modifies the same memory location, and (2) at least one of these actions is not atomic, or the corresponding memory scopes are not inclusive, and (3) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens-before relation.

Deprecation

Existing features are marked as deprecated if their usage is not recommended as that feature is being de-emphasized, superseded and may be removed from a future version of the specification.

Device

A *device* is a collection of *compute units*. A *command-queue* is used to queue *commands* to a *device*. Examples of *commands* include executing *kernels*, or reading and writing *memory objects*. OpenCL devices typically correspond to a GPU, a multi-core CPU, and other processors such as DSPs and the Cell/B.E. processor.

Device-side enqueue

A mechanism whereby a kernel-instance is enqueued by a kernel-instance running on a device without direct involvement by the host program. This produces *nested parallelism*; i.e. additional levels of concurrency are nested inside a running kernel-instance. The kernel-instance executing on a device (the *parent kernel*) enqueues a kernel-instance (the *child kernel*) to a device-side command-queue. Child and parent kernels execute asynchronously though a parent kernel does not complete until all of its child-kernels have completed.

Diverged control flow

See *Control flow*.

Ended

The fifth state in the six state model for the execution of a command. The transition into this state occurs when execution of a command has ended. When a Kernel-enqueue command ends, all of the work-groups associated with that command have finished their execution.

Event Object

An *event object* encapsulates the status of an operation such as a *command*. It can be used to synchronize operations in a context.

Event Wait List

An *event wait list* is a list of *event objects* that can be used to control when a particular *command* begins execution.

Fence

A memory ordering operation without an associated atomic object. A fence can use the *acquire semantics, release semantics*, or *acquire release semantics*.

Framework

A software system that contains the set of components to support software development and execution. A *framework* typically includes libraries, APIs, runtime systems, compilers, etc.

Generic address space

An address space that include the *private*, *local*, and *global* address spaces available to a device. The generic address space supports conversion of pointers to and from private, local and global address spaces, and hence lets a programmer write a single function that at compile time can take arguments from any of the three named address spaces.

Global-happens-before

See *Happens-before*.

Global ID

A *global ID* is used to uniquely identify a *work-item* and is derived from the number of *global work-items* specified when executing a *kernel*. The *global ID* is a N-dimensional value that starts at (0, 0, … 0). See also *Local ID*.

Global Memory

A memory region accessible to all *work-items* executing in a *context*. It is accessible to the *host* using *commands* such as read, write and map. *Global memory* is included within the *generic address space* that includes the private and local address spaces.

GL share group

A *GL share group* object manages shared OpenGL or OpenGL ES resources such as textures, buffers, framebuffers, and renderbuffers and is associated with one or more GL context objects. The *GL share group* is typically an opaque object and not directly accessible.

Handle

An opaque type that references an *object* allocated by OpenCL. Any operation on an *object* occurs by reference to that object's handle. Each object must have a unique handle value during the course of its lifetime. Handle values may be, but are not required to be, re-used by an implementation.

Happens-before

An ordering relationship between operations that execute on multiple units of execution. If an operation A happens-before operation B then A must occur before B; in particular, any value written by A will be visible to B. We define two separate happens-before relations: *global-happens-before* and *local-happens-before*. These are defined in Memory Ordering Rules.

Host

The *host* interacts with the *context* using the OpenCL API.

Host-thread

The unit of execution that executes the statements in the host program.

Host pointer

A pointer to memory that is in the virtual address space on the *host*.

Illegal

Behavior of a system that is explicitly not allowed and will be reported as an error when encountered by OpenCL.

Image Object

A *memory object* that stores a two- or three-dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a *sampler*.

The *image object* encapsulates the following information:

- Dimensions of the image.
- Description of each element in the image.
- Properties that describe usage information and which region to allocate from.
- Image data.

The elements of an image are selected from a list of predefined image formats.

Immutable Memory Object

A *memory object* whose contents cannot be altered by the implementation after creation. External entities may change the contents of *immutable memory objects* created from external handles.

Implementation-Defined

Behavior that is explicitly allowed to vary between conforming implementations of OpenCL. An OpenCL implementor is required to document the implementation-defined behavior.

Independent Forward Progress

If an entity supports independent forward progress, then if it is otherwise not dependent on any actions due to be performed by any other entity (for example it does not wait on a lock held by, and thus that must be released by, any other entity), then its execution cannot be blocked by the execution of any other entity in the system (it will not be starved). Work-items in a sub-group, for example, typically do not support independent forward progress, so one work-item in a sub-group may be completely blocked (starved) if a different work-item in the same sub-group enters a spin loop.

In-order Execution

A model of execution in OpenCL where the *commands* in a *command-queue* are executed in order of submission with each *command* running to completion before the next one begins. See *Out-of-order Execution*.

Intermediate Language

A lower-level language that may be used to create programs. SPIR-V is a required intermediate language (IL) for OpenCL 2.1, OpenCL 2.2, and OpenCL 3.1 or newer devices. Other OpenCL devices may optionally support SPIR-V or other ILs.

Kernel

A *kernel* is a function declared in a *program* and executed on an OpenCL *device*. A *kernel* is identified by the `__kernel` or `kernel` qualifier applied to any function defined in a *program*.

Kernel-instance

The work carried out by an OpenCL program occurs through the execution of kernel-instances on devices. The kernel-instance is the *kernel object*, the values associated with the arguments to the kernel, and the parameters that define the *ND-range* index space.

Kernel Object

A *kernel object* encapsulates a specific *kernel* function declared in a *program* and the argument values to be used when executing this *kernel* function.

Kernel Language

A language that is used to represent source code for kernel. Kernels may be directly created from OpenCL C kernel language source strings. Other kernel languages may be supported by compiling to SPIR-V, another supported Intermediate Language, or to a device-specific program binary format.

Launch

The transition of a command from the *submitted* state to the *ready* state. See *Ready*.

Local ID

A *local ID* specifies a unique *work-item ID* within a given *work-group* that is executing a *kernel*. The *local ID* is a N-dimensional value that starts at (0, 0, … 0). See also *Global ID*.

Local Memory

A memory region associated with a *work-group* and accessible only by *work-items* in that *work-group*. *Local memory* is included within the *generic address space* that includes the private and global address spaces.

Marker

A *command* queued in a *command-queue* that can be used to tag all *commands* queued before the *marker* in the *command-queue*. The *marker* command returns an *event* which can be used by the *application* to queue a wait on the marker event i.e. wait for all commands queued before the *marker* command to complete.

Memory Consistency Model

Rules that define which values are observed when multiple units of execution load data from any shared memory plus the synchronization operations that constrain the order of memory operations and define synchronization relationships. The memory consistency model in OpenCL is based on the memory model from the ISO C11 programming language.

Memory Objects

A *memory object* is a handle to a reference counted region of *Global Memory*. Also see *Buffer Object* and *Image Object*.

Memory Regions (or Pools)

A distinct address space in OpenCL. *Memory regions* may overlap in physical memory though OpenCL will treat them as logically distinct. The *memory regions* are denoted as *private*, *local*, *constant*, and *global*.

Memory Scopes

These memory scopes define a hierarchy of visibilities when analyzing the ordering constraints of memory operations. They are defined by the values of the **memory_scope** enumeration constant. Current values are **memory_scope_work_item** (memory constraints only apply to a single work-item and in practice apply only to image operations), **memory_scope_sub_group** (memory-ordering constraints only apply to work-items executing in a sub-group), **memory_scope_work_group** (memory-ordering constraints only apply to work-items executing in a work-group), **memory_scope_device** (memory-ordering constraints only apply to work-items executing on a single device) and **memory_scope_all_svm_devices** or equivalently

memory_scope_all_devices (memory-ordering constraints only apply to work-items executing across multiple devices and when using shared virtual memory).

Modification Order

All modifications to a particular atomic object M occur in some particular *total order*, called the *modification order* of M. If A and B are modifications of an atomic object M, and A happens-before B, then A shall precede B in the modification order of M. Note that the modification order of an atomic object M is independent of whether M is in local or global memory.

Nested Parallelism

See *device-side enqueue*.

Object

Objects are abstract representation of the resources that can be manipulated by the OpenCL API. Examples include *program objects*, *kernel objects*, and *memory objects*.

Out-of-order Execution

A model of execution in which *commands* placed in the *work queue* may begin and complete execution in any order consistent with constraints imposed by *event wait lists_and_command-queue barrier*. See *In-order Execution*.

Parent device

The OpenCL *device* which is partitioned to create *sub-devices*. Not all *parent devices* are *root devices*. A *root device* might be partitioned and the *sub-devices* partitioned again. In this case, the first set of *sub-devices* would be *parent devices* of the second set, but not the *root devices*. Also see *Device*, *parent device* and *root device*.

Parent kernel

see *Device-side enqueue*.

Pipe

The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel-instance may write into a pipe, and only one kernel-instance may read from a pipe. To support the producer consumer design pattern, one kernel-instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer).

Platform

The *host* plus a collection of *devices* managed by the OpenCL *framework* that allow an application to share *resources* and execute *kernels* on *devices* in the *platform*.

Private Memory

A region of memory private to a *work-item*. Variables defined in one *work-items private memory* are not visible to another *work-item*.

Processing Element

A virtual scalar processor. A work-item may execute on one or more processing elements.

Program

An OpenCL *program* consists of a set of *kernels*. *Programs* may also contain auxiliary functions called by the *kernel* functions and constant data.

Program Object

A *program object* encapsulates the following information:

- A reference to an associated *context*.
- A *program* source or binary.
- The latest successfully built program executable, the list of *devices* for which the program executable is built, the build options used and a build log.
- The number of *kernel objects* currently attached.

Queued

The first state in the six state model for the execution of a command. The transition into this state occurs when the command is enqueued into a command-queue.

Ready

The third state in the six state model for the execution of a command. The transition into this state occurs when pre-requisites constraining execution of a command have been met; i.e. the command has been launched. When a kernel-enqueue command is launched, work-groups associated with the command are placed in a devices work-pool from which they are scheduled for execution.

Re-converged Control Flow

see *Control flow*.

Reference Count

The life span of an OpenCL object is determined by its *reference count*, an internal count of the number of references to the object. When you create an object in OpenCL, its *reference count* is set to one. Subsequent calls to the appropriate *retain* API (such as **clRetainContext**, **clRetainCommandQueue**) increment the *reference count*. Calls to the appropriate *release* API (such as **clReleaseContext**, **clReleaseCommandQueue**) decrement the *reference count*. Implementations may also modify the *reference count*, e.g. to track attached objects or to ensure correct operation of in-progress or scheduled activities. The object becomes inaccessible to host code when the number of *release* operations performed matches the number of *retain* operations plus the allocation of the object. At this point the reference count may be zero but this is not guaranteed.

Relaxed Consistency

A memory consistency model in which the contents of memory visible to different *work-items* or *commands* may be different except at a *barrier* or other explicit synchronization points.

Relaxed Semantics

A memory order semantics for atomic operations that implies no order constraints. The operation is *atomic* but it has no impact on the order of memory operations.

Release Semantics

One of the memory order semantics defined for synchronization operations. Release semantics apply to atomic operations that store to memory. Given two units of execution, **A** and **B**, acting on a shared atomic object **M**, if **A** uses an atomic store of **M** with release semantics to synchronize-with an atomic load to **M** by **B** that used acquire semantics, then **A**'s atomic store will occur *after* any prior operations by **A**. Note that the memory orders *acquire*, *sequentially consistent*, and *acquire_release* all include *acquire semantics* and effectively pair with a store using release semantics.

Remainder work-groups

When the work-groups associated with a kernel-instance are defined, the sizes of a work-group in each dimension may not evenly divide the size of the ND-range in the corresponding dimensions. The result is a collection of work-groups on the boundaries of the ND-range that are smaller than the base work-group size. These are known as *remainder work-groups*.

Running

The fourth state in the six state model for the execution of a command. The transition into this state occurs when the execution of the command starts. When a Kernel-enqueue command starts, one or more work-groups associated with the command start to execute.

Root device

A *root device* is an OpenCL *device* that has not been partitioned. Also see *Device*, *Parent device* and *Root device*.

Resource

A class of *objects* defined by OpenCL. An instance of a *resource* is an *object*. The most common *resources* are the *context*, *command-queue*, *program objects*, *kernel objects*, and *memory objects*. Computational resources are hardware elements that participate in the action of advancing a program counter. Examples include the *host*, *devices*, *compute units* and *processing elements*.

Retain, Release

The action of incrementing (retain) and decrementing (release) the reference count using an OpenCL *object*. This is a book keeping functionality to make sure the system doesn't remove an *object* before all instances that use this *object* have finished. Refer to *Reference Count*.

Sampler

An *object* that describes how to sample an image when the image is read in the *kernel*. The image read functions take a *sampler* as an argument. The *sampler* specifies the image addressing-mode i.e. how out-of-range image coordinates are handled, the filter mode, and whether the input image coordinate is a normalized or unnormalized value.

Scope inclusion

Two actions **A** and **B** have an inclusive scope if the work-item or host thread performing **A** is in the memory scope of **B** and the work-item or host thread performing **B** is in the memory scope of **A**. Prior to OpenCL 3.1, the memory scope of **A** and **B** additionally must be the same for the actions to have an inclusive scope. For OpenCL 3.1 and newer, the memory scope of **A** and **B** may be different.

Sequenced before

A relation between evaluations executed by a single unit of execution. Sequenced-before is an asymmetric, transitive, pair-wise relation that induces a partial order between evaluations. Given any two evaluations A and B, if A is sequenced-before B, then the execution of A shall precede the execution of B.

Sequential consistency

Sequential consistency interleaves the steps executed by each unit of execution. Each access to a memory location sees the last assignment to that location in that interleaving.

Sequentially consistent semantics

One of the memory order semantics defined for synchronization operations. When using sequentially-consistent synchronization operations, the loads and stores within one unit of execution appear to execute in program order (i.e., the sequenced-before order), and loads and stores from different units of execution appear to be simply interleaved.

Shared Virtual Memory (SVM)

An address space exposed to both the host and the devices within a context. SVM causes addresses to be meaningful between the host and all of the devices within a context and therefore supports the use of pointer based data structures in OpenCL kernels. It logically extends a portion of the global memory into the host address space therefore giving work-items access to the host address space. There are three types of SVM in OpenCL:

Coarse-Grained buffer SVM

Sharing occurs at the granularity of regions of OpenCL buffer memory objects.

Fine-Grained buffer SVM

Sharing occurs at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects.

Fine-Grained system SVM

Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory.

SIMD

Single Instruction Multiple Data. A programming model where a *kernel* is executed concurrently on multiple *processing elements* each with its own data and a shared program counter. All *processing elements* execute a strictly identical set of instructions.

Specialization constants

Specialization constants are special constant objects that do not have known constant values in an intermediate language (e.g. SPIR-V). Applications may provide updated values for the specialization constants before a program is built. Specialization constants that do not receive a value from an application shall use the default specialization constant value.

SPMD

Single Program Multiple Data. A programming model where a *kernel* is executed concurrently on multiple *processing elements* each with its own data and its own program counter. Hence,

while all computational resources run the same *kernel* they maintain their own instruction counter and due to branches in a *kernel*, the actual sequence of instructions can be quite different across the set of *processing elements*.

Sub-device

An OpenCL *device* can be partitioned into multiple *sub-devices*. The new *sub-devices* alias specific collections of compute units within the parent *device*, according to a partition scheme. The *sub-devices* may be used in any situation that their parent *device* may be used. Partitioning a *device* does not destroy the parent *device*, which may continue to be used along side and intermingled with its child *sub-devices*. Also see *Device*, *Parent device* and *Root device*.

Sub-group

Sub-groups are an implementation-dependent grouping of work-items within a work-group. The size and number of sub-groups is implementation-defined.

Sub-group Barrier

See *Barrier*.

Submitted

The second state in the six state model for the execution of a command. The transition into this state occurs when the command is flushed from the command-queue and submitted for execution on the device. Once submitted, a programmer can assume a command will execute once its prerequisites have been met.

SVM Buffer

A memory allocation enabled to work with *Shared Virtual Memory (SVM)*. Depending on how the SVM buffer is created, it can be a coarse-grained or fine-grained SVM buffer. Optionally it may be wrapped by a *Buffer Object*. See *Shared Virtual Memory (SVM)*.

Synchronization

Synchronization refers to mechanisms that constrain the order of execution and the visibility of memory operations between two or more units of execution.

Synchronization operations

Operations that define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. Synchronization operations in OpenCL include *atomic operations* and *fences*.

Synchronization point

A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched (i.e. enters the ready state) .

Synchronizes with

A relation between operations in two different units of execution that defines a memory order constraint in global memory (*global-synchronizes-with*) or local memory (*local-synchronizes-with*).

Task Parallel Programming Model

A programming model in which computations are expressed in terms of multiple concurrent tasks executing in one or more *command-queues*. The concurrent tasks can be running different *kernels*.

Thread-safe

An OpenCL API call is considered to be *thread-safe* if the internal state as managed by OpenCL remains consistent when called simultaneously by multiple *host* threads. OpenCL API calls that are *thread-safe* allow an application to call these functions in multiple *host* threads without having to implement mutual exclusion across these *host* threads.

Undefined

The behavior of an OpenCL API call, built-in function used inside a *kernel* or execution of a *kernel* that is explicitly not defined by OpenCL. A conforming implementation is not required to specify what occurs when an undefined construct is encountered in OpenCL.

Unit of execution

A generic term for a process, OS managed thread running on the host (a host-thread), kernel-instance, host program, work-item or any other executable agent that advances the work associated with a program.

Valid Object

An OpenCL object is considered valid if it meets all of the following criteria:

- The object was created by a successful call to an OpenCL API function.
- The object has a strictly positive application-owned reference count.
- The object has not had its backing memory changed outside of normal usage by the OpenCL implementation (e.g. corrupted by the application, a library it uses, the implementation itself, or any other agent that can access the object's backing memory).

An object is only valid in the platform where it was created.

An OpenCL implementation must check for a `NULL` object to determine if an object is valid. The behavior for all other invalid objects is implementation-defined.

Work-group

A collection of related *work-items* that execute on a single *compute unit*. The *work-items* in the group execute the same *kernel-instance* and share *local memory* and *work-group functions*.

Work-group Barrier

See *Barrier*.

Work-group Function

A function that carries out collective operations across all the work-items in a work-group. Available collective operations are a barrier, reduction, broadcast, prefix sum, and evaluation of a predicate. A work-group function must occur within a *converged control flow*; i.e. all work-items in the work-group must encounter precisely the same work-group function.

Work-group Synchronization

Constraints on the order of execution for work-items in a single work-group.

Work-pool

A logical pool associated with a device that holds commands and work-groups from kernel-instances that are ready to execute. OpenCL does not constrain the order that commands and work-groups are scheduled for execution from the work-pool; i.e. a programmer must assume that they could be interleaved. There is one work-pool per device used by all command-queues associated with that device. The work-pool may be implemented in any manner as long as it assures that work-groups placed in the pool will eventually execute.

Work-item

One of a collection of parallel executions of a *kernel* invoked on a *device* by a *command*. A *work-item* is executed by one or more *processing elements* as part of a *work-group* executing on a *compute unit*. A *work-item* is distinguished from other work-items by its *global ID* or the combination of its *work-group* ID and its *local ID* within a *work-group*.

Chapter 3. The OpenCL Architecture

OpenCL is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. It is more than a language. OpenCL is a framework for parallel programming and includes a language, API, libraries and a runtime system to support software development. Using OpenCL, for example, a programmer can write general purpose programs that execute on GPUs without the need to map their algorithms onto a 3D graphics API such as OpenGL or DirectX.

The target of OpenCL is expert programmers wanting to write portable yet efficient code. This includes library writers, middleware vendors, and performance oriented application programmers. Therefore OpenCL provides a low-level hardware abstraction plus a framework to support programming and many details of the underlying hardware are exposed.

To describe the core ideas behind OpenCL, we will use a hierarchy of models:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

3.1. Platform Model

The Platform model for OpenCL is defined below. The model consists of a **host** connected to one or more **OpenCL devices**. An OpenCL device is divided into one or more **compute units** (CUs) which are further divided into one or more **processing elements** (PEs). Computations on a device occur within the processing elements.

An OpenCL application is implemented as both host code and device kernel code. The host code portion of an OpenCL application runs on a host processor according to the models native to the host platform. The OpenCL application host code submits the kernel code as commands from the host to OpenCL devices. An OpenCL device executes the commands on the processing elements within the device.

An OpenCL device has considerable latitude on how computations are mapped onto the device's processing elements. When processing elements within a compute unit execute the same sequence of statements across the processing elements, the control flow is said to be *converged*. Hardware optimized for executing a single stream of instructions over multiple processing elements is well suited to converged control flows. When the control flow varies from one processing element to another, it is said to be *diverged*. While a kernel always begins execution with a converged control flow, due to branching statements within a kernel, converged and diverged control flows may occur within a single kernel. This provides a great deal of flexibility in the algorithms that can be implemented with OpenCL.

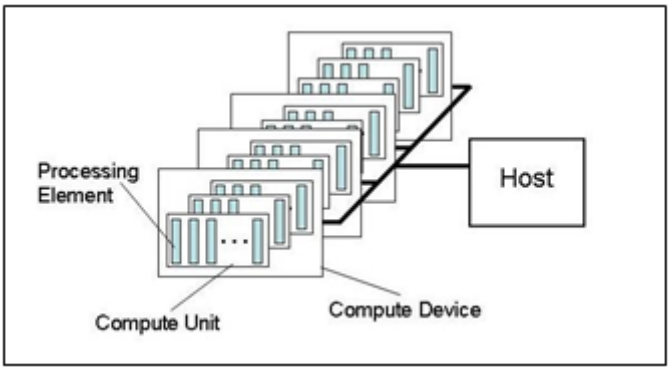

Figure 1. Platform Model ... one host plus one or more compute devices each with one or more compute units composed of one or more processing elements.

Programmers may provide programs in the form of OpenCL C source strings, the SPIR-V intermediate language, or as implementation-defined binary objects. An OpenCL platform provides a compiler to translate programs of these forms into executable program objects. The device code compiler may be *online* or *offline*. An *online compiler* is available during host program execution using standard APIs. An *offline compiler* is invoked outside of host program control, using platform-specific methods. The OpenCL runtime allows developers to get a previously compiled device program executable and be able to load and execute a previously compiled device program executable.

OpenCL defines two kinds of platform profiles: a *full profile* and a reduced-functionality *embedded profile*. A full profile platform must provide an online compiler for all its devices. An embedded profile platform may provide an online compiler, but is not required to do so.

A device may expose special purpose functionality as a *built-in kernel*. The platform provides APIs for enumerating and invoking the built-in kernels offered by a device, but otherwise does not define their construction or semantics.

Built-in kernels are missing before version 1.2.

All device types support the OpenCL execution model, the OpenCL memory model, and the APIs used in OpenCL to manage devices.

The platform model is an abstraction describing how OpenCL views the hardware. The relationship between the elements of the platform model and the hardware in a system may be a fixed property of a device or it may be a dynamic feature of a program dependent on how a compiler optimizes code to best utilize physical hardware.

3.2. Execution Model

The OpenCL execution model is defined in terms of two distinct units of execution: **kernels** that execute on one or more OpenCL devices and a **host program** that executes on the host. With regard to OpenCL, the kernels are where the "work" associated with a computation occurs. This work occurs through **work-items** that execute in groups (**work-groups**).

A kernel executes within a well-defined context managed by the host. The context defines the environment within which kernels execute. It includes the following resources:

- **Devices**: One or more devices exposed by the OpenCL platform.
- **Kernel Objects**: The OpenCL functions with their associated argument values that run on OpenCL devices.
- **Program Objects**: The program source and executable that implement the kernels.
- **Memory Objects**: Variables visible to the host and the OpenCL devices. Instances of kernels operate on these objects as they execute.

The host program uses the OpenCL API to create and manage the context. Functions from the OpenCL API enable the host to interact with a device through a *command-queue*. Each command-queue is associated with a single device. The commands placed into the command-queue fall into one of three types:

- **Kernel-enqueue commands**: Enqueue a kernel for execution on a device.
- **Memory commands**: Transfer data between the host and device memory, between memory objects, or map and unmap memory objects from the host address space.
- **Synchronization commands**: Explicit synchronization points that define order constraints between commands.

In addition to commands submitted from the host command-queue, a kernel running on a device can enqueue commands to a device-side command-queue. This results in *child kernels* enqueued by a kernel executing on a device (the *parent kernel*). Regardless of whether the command-queue resides on the host or a device, each command passes through six states.

1. **Queued**: The command is enqueued to a command-queue. A command may reside in the queue until it is flushed either explicitly (a call to **clFlush**) or implicitly by some other command.
2. **Submitted**: The command is flushed from the command-queue and submitted for execution on the device. Once flushed from the command-queue, a command will execute after any prerequisites for execution are met.
3. **Ready**: All prerequisites constraining execution of a command have been met. The command, or for a kernel-enqueue command the collection of work-groups associated with a command, is placed in a device work-pool from which it is scheduled for execution.
4. **Running**: Execution of the command starts. For the case of a kernel-enqueue command, one or more work-groups associated with the command start to execute.
5. **Ended**: Execution of a command ends. When a kernel-enqueue command ends, all of the work-groups associated with that command have finished their execution. *Immediate side effects*, i.e. those associated with the kernel but not necessarily with its child kernels, are visible to other units of execution. These side effects include updates to values in global memory.
6. **Complete**: The command and its child commands have finished execution and the status of the event object, if any, associated with the command is set to `CL_COMPLETE`.

The execution states and the transitions between them are summarized below. These states and the concept of a device work-pool are conceptual elements of the execution model. An implementation of OpenCL has considerable freedom in how these are exposed to a program. Five of the transitions, however, are directly observable through a profiling interface. These profiled states are shown below.

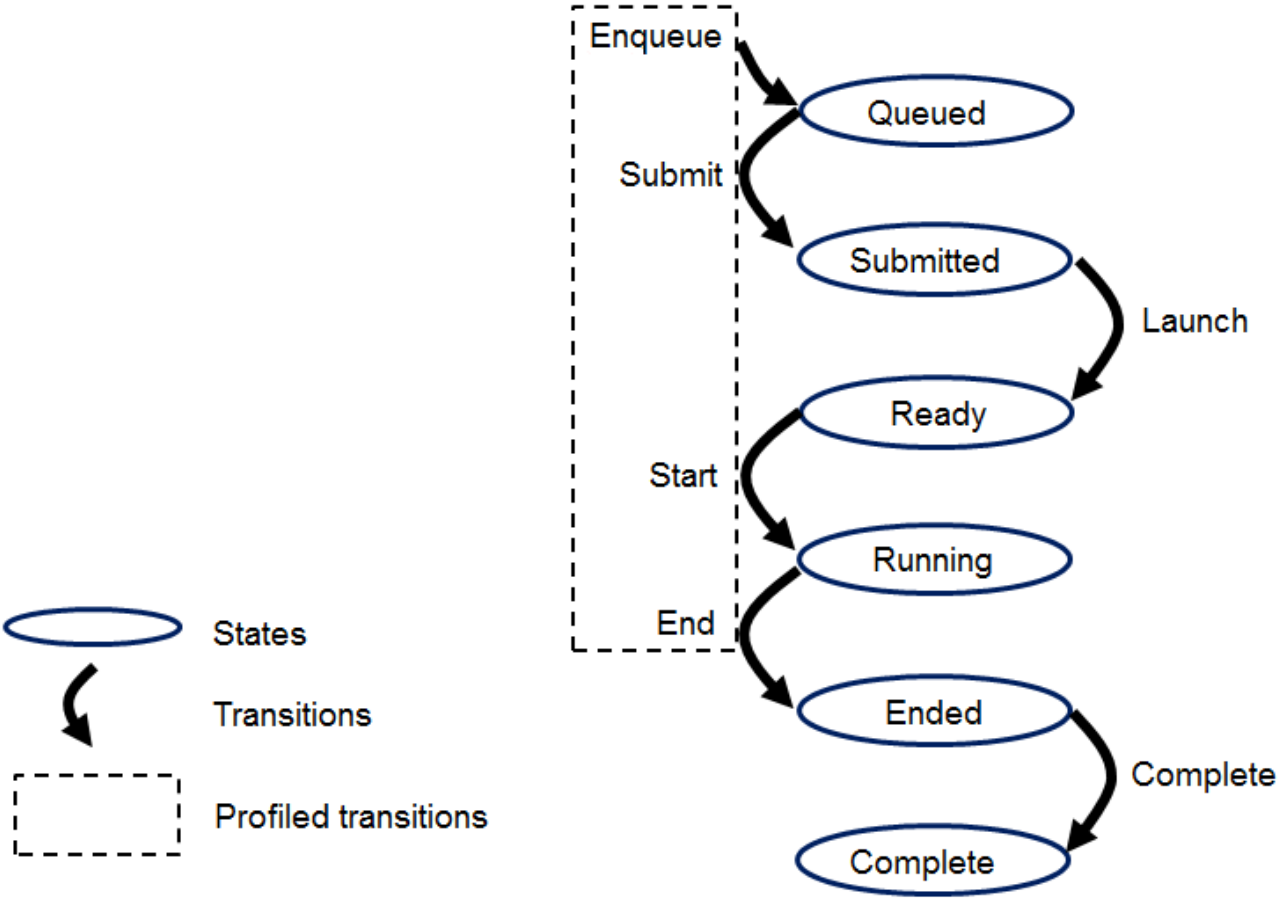

Figure 2. The states and transitions between states defined in the OpenCL execution model. A subset of these transitions is exposed through the profiling interface.

Commands communicate their status through *event objects*. Successful completion is indicated by setting the event status associated with a command to `CL_COMPLETE`. Unsuccessful completion results in abnormal termination of the command which is indicated by setting the event status to a negative value. In this case, the command-queue associated with the abnormally terminated command and all other command-queues in the same context may no longer be available and their behavior is implementation-defined.

A command submitted to a device will not launch until prerequisites that constrain the order of commands have been resolved. These prerequisites have three sources:

- The first source of prerequisites is implicit dependencies between commands enqueued to the same command-queue which arise as follows:
  - Commands enqueued after a command-queue barrier have the preceding barrier command as a prerequisite.
  - Commands enqueued in an in-order command-queue have the command enqueued before them as a prerequisite.
- The second source of prerequisites is dependencies between commands expressed through events. A command may include an optional list of events. The command will wait and not launch until all the events in the list are in the state `CL_COMPLETE`. By this mechanism, event objects define order constraints between commands and coordinate execution between the host and one or more devices.
- The third source of prerequisites can be the presence of non-trivial C initializers or C++

constructors for program scope global variables. In this case, the OpenCL C/C++ compiler must generate program initialization kernels that perform C initialization or C++ construction. These kernels must be executed by OpenCL runtime on a device before any kernel from the same program can be executed on the same device. The ND-range for any program initialization kernel is (1,1,1). When multiple programs are linked together, the order of execution of program initialization kernels that belong to different programs is undefined.

Program clean up may result in the execution of one or more program clean up kernels by the OpenCL runtime. This is due to the presence of non-trivial C++ destructors for program scope variables. The ND-range for executing any program clean up kernel is (1,1,1). The order of execution of clean up kernels from different programs (that are linked together) is undefined.

Program initialization and clean-up kernels are missing before version 2.2.

Note that C initializers, C++ constructors, or C++ destructors for program scope variables cannot use pointers to coarse grain and fine grain SVM allocations.

A command may be submitted to a device and yet have no visible side effects outside of waiting on and satisfying event dependences. Examples include markers, kernels executed over ranges of no work-items or copy operations with zero sizes. Such commands may pass directly from the *ready* state to the *ended* state.

Command execution can be blocking or non-blocking. Consider a sequence of OpenCL commands. For blocking commands, the OpenCL API functions that enqueue commands don't return until the command has completed. Alternatively, OpenCL functions that enqueue non-blocking commands return immediately and require that a programmer defines dependencies between enqueued commands to ensure that enqueued commands are not launched before needed resources are available. In both cases, the actual execution of the command may occur asynchronously with execution of the host program.

Commands within a single command-queue execute relative to each other in one of two modes:

- **In-order Execution**: Commands and any side effects associated with commands appear to the OpenCL application as if they execute in the same order they are enqueued to a command-queue.
- **Out-of-order Execution**: Commands execute in any order constrained only by explicit synchronization points (e.g. through command-queue barriers) or explicit dependencies on events.

Multiple command-queues can be present within a single context. Multiple command-queues execute commands independently. Event objects visible to the host program can be used to define synchronization points between commands in multiple command-queues. If such synchronization points are established between commands in multiple command-queues, an implementation must assure that the command-queues progress concurrently and correctly account for the dependencies established by the synchronization points. For a detailed explanation of synchronization points, see the execution model Synchronization section.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel-enqueue command submits a kernel for execution, an index space is defined. The kernel, the

argument values associated with the arguments to the kernel, and the parameters that define the index space define a *kernel-instance*. When a kernel-instance executes on a device, the kernel function executes for each point in the defined index space. Each of these executing kernel functions is called a *work-item*. The work-items associated with a given kernel-instance are managed by the device in groups called *work-groups*. These work-groups define a coarse-grained decomposition of the index space. Work-groups are further divided into *sub-groups*, which provide an additional level of control over execution.

Sub-groups are missing before version 2.1.

Work-items have a global ID based on their coordinates within the index space. They can also be defined in terms of their work-group and the local ID within a work-group. The details of this mapping are described in the following section.

3.2.1. Mapping Work-items Onto an ND-range

The index space supported by OpenCL is called an ND-range. An ND-range is an N-dimensional index space, where N is one, two or three. The ND-range is decomposed into work-groups forming blocks that cover the index space. An ND-range is defined by three integer arrays of length N:

- The extent of the index space (or global size) in each dimension.
- An offset index F indicating the initial value of the indices in each dimension (zero by default).
- The size of a work-group (local size) in each dimension.

Each work-item's global ID is an N-dimensional tuple. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

Unless a kernel comes from a source that disallows it, e.g. OpenCL C 1.x or using `-cl-uniform-work-group-size`, the size of work-groups in an ND-range (the local size) need not be the same for all work-groups. In this case, any single dimension for which the global size is not divisible by the local size will be partitioned into two regions. One region will have work-groups that have the same number of work-items as was specified for that dimension by the programmer (the local size). The other region will have work-groups with less than the number of work items specified by the local size parameter in that dimension (the *remainder work-groups*). Work-group sizes could be non-uniform in multiple dimensions, potentially producing work-groups of up to 4 different sizes in a 2D range and 8 different sizes in a 3D range.

Non-uniform work-group sizes are missing before version 2.0.

Each work-item is assigned to a work-group and given a local ID to represent its position within the work-group. A work-item's local ID is an N-dimensional tuple with components in the range from zero to the size of the work-group in that dimension minus one.

Work-groups are assigned IDs similarly. The number of work-groups in each dimension is not directly defined but is inferred from the local and global ND-ranges provided when a kernel-instance is enqueued. A work-group's ID is an N-dimensional tuple with components in the range 0 to the ceiling of the global size in that dimension divided by the local size in the same dimension. As a result, the combination of a work-group ID and the local-ID within a work-group uniquely defines

a work-item. Each work-item is identifiable in two ways; in terms of a global index, and in terms of a work-group index plus a local index within a work-group.

For example, consider the 2-dimensional index space shown below. We input the index space for the work-items (G_x, G_y), the size of each work-group (S_x, S_y) and the global ID offset (F_x, F_y). The global indices define a $G_x$by G_y index space where the total number of work-items is the product of G_x and G_y. The local indices define an S_x by S_y index space where the number of work-items in a single work-group is the product of S_x and S_y. Given the size of each work-group and the total number of work-items we can compute the number of work-groups. A 2-dimensional index space is used to uniquely identify a work-group. Each work-item is identified by its global ID (g_x, g_y) or by the combination of the work-group ID (w_x, w_y), the size of each work-group (S_x,S_y) and the local ID (s_x, s_y) inside the work-group such that

$$(g_x, g_y) = (w_x \times S_x + s_x + F_x, w_y \times S_y + s_y + F_y)$$

The number of work-groups can be computed as:

$$(W_x, W_y) = (\text{ceil}(G_x / S_x), \text{ceil}(G_y / S_y))$$

Given a global ID and the work-group size, the work-group ID for a work-item is computed as:

$$(w_x, w_y) = ((g_x - s_x - F_x) / S_x, (g_y - s_y - F_y) / S_y)$$

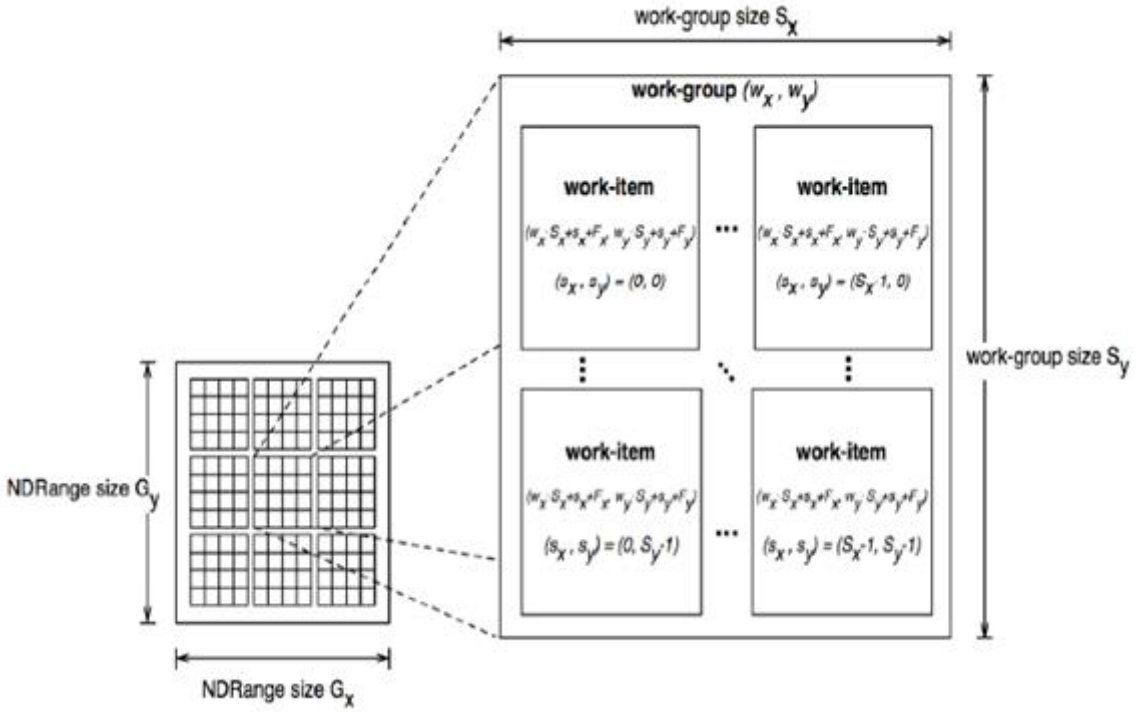

Figure 3. An example of an ND-range index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs. In this case, we assume that in each dimension, the size of the work-group evenly divides the global ND-range size (i.e. all work-groups have the same size) and that the offset is equal to zero.

Within a work-group work-items may be divided into sub-groups. The mapping of work-items to sub-groups is implementation-defined, and may be queried at runtime. While sub-groups may be used in multi-dimensional work-groups, each sub-group is 1-dimensional and any given work-item may query which sub-group it is a member of.

Sub-groups are missing before version 2.1.

Work-items are mapped into sub-groups through a combination of compile-time decisions and the parameters of the dispatch. The mapping to sub-groups is invariant for the duration of a kernel's execution, across dispatches of a given kernel with the same work-group dimensions, between dispatches and query operations consistent with the dispatch parameterization, and from one work-group to another within the dispatch (excluding the trailing edge work-groups in the presence of non-uniform work-group sizes). In addition, all sub-groups within a work-group will be the same size, apart from the sub-group with the maximum index which may be smaller if the size of the work-group is not evenly divisible by the size of the sub-groups.

In the degenerate case, a single sub-group must be supported for each work-group. In this degenerate case all sub-group scope functions are equivalent to their work-group level equivalents.

3.2.2. Execution of Kernel-instances

The work carried out by an OpenCL program occurs through the execution of kernel-instances on compute devices. To understand the details of OpenCL's execution model, we need to consider how a kernel object moves from the kernel-enqueue command, into a command-queue, executes on a device, and completes.

A kernel object is defined as a function within the program object and a collection of arguments connecting the kernel to a set of argument values. The host program enqueues a kernel object to the command-queue along with the ND-range and the work-group decomposition. These define a *kernel-instance*. In addition, an optional set of events may be defined when the kernel is enqueued. The events associated with a particular kernel-instance are used to constrain when the kernel-instance is launched with respect to other commands in the queue or to commands in other queues within the same context.

A kernel-instance is submitted to a device. For an in-order command-queue, the kernel-instances appear to launch and then execute in that same order; where we use the term appear to emphasize that when there are no dependencies between commands and hence differences in the order that commands execute cannot be observed in a program, an implementation can reorder commands even in an in-order command-queue. For an out-of-order command-queue, kernel-instances wait to be launched until:

- Synchronization commands enqueued prior to the kernel-instance are satisfied.
- Each of the events in an optional event list defined when the kernel-instance was enqueued are set to `CL_COMPLETE`.

Once these conditions are met, the kernel-instance is launched and the work-groups associated with the kernel-instance are placed into a pool of ready to execute work-groups. This pool is called a *work-pool*. The work-pool may be implemented in any manner as long as it assures that work-groups placed in the pool will eventually execute. The device schedules work-groups from the work-pool for execution on the compute units of the device. The kernel-enqueue command is complete when all work-groups associated with the kernel-instance end their execution, updates to global memory associated with a command are visible globally, and the device signals successful completion by setting the event associated with the kernel-enqueue command to `CL_COMPLETE`.

While a command-queue is associated with only one device, a single device may be associated with multiple command-queues all feeding into the single work-pool. A device may also be associated with command-queues associated with different contexts within the same platform, again all feeding into the single work-pool. The device will pull work-groups from the work-pool and execute them on one or several compute units in any order; possibly interleaving execution of work-groups from multiple commands. A conforming implementation may choose to serialize the work-groups so a correct algorithm cannot assume that work-groups will execute in parallel. There is no safe and portable way to synchronize across the independent execution of work-groups since once in the work-pool, they can execute in any order.

The work-items within a single sub-group execute concurrently but not necessarily in parallel (i.e. they are not guaranteed to make independent forward progress). Therefore, only high-level synchronization constructs (e.g. sub-group functions such as barriers) that apply to all the work-items in a sub-group are well defined and included in OpenCL.

Sub-groups are missing before version 2.1.

Sub-groups execute concurrently within a given work-group and with appropriate device support (see Querying Devices), may make independent forward progress with respect to each other, with respect to host threads and with respect to any entities external to the OpenCL system but running on an OpenCL device, even in the absence of work-group barrier operations. In this situation, sub-groups are able to internally synchronize using barrier operations without synchronizing with each other and may perform operations that rely on runtime dependencies on operations other sub-groups perform.

The work-items within a single work-group execute concurrently but are only guaranteed to make independent progress in the presence of sub-groups and device support. In the absence of this capability, only high-level synchronization constructs (e.g. work-group functions such as barriers) that apply to all the work-items in a work-group are well defined and included in OpenCL for synchronization within the work-group.

In the absence of synchronization functions (e.g. a barrier), work-items within a sub-group may be serialized. In the presence of sub-group functions, work-items within a sub-group may be serialized before any given sub-group function, between dynamically encountered pairs of sub-group functions and between a work-group function and the end of the kernel.

In the absence of independent forward progress of constituent sub-groups, work-items within a work-group may be serialized before, after or between work-group synchronization functions.

3.2.3. Device-Side Enqueue

Device-side enqueue is missing before version 2.0.

Algorithms may need to generate additional work as they execute. In many cases, this additional work cannot be determined statically; so the work associated with a kernel only emerges at runtime as the kernel-instance executes. This capability could be implemented in logic running within the host program, but involvement of the host may add significant overhead and/or complexity to the application control flow. A more efficient approach would be to nest kernel-

enqueue commands from inside other kernels. This **nested parallelism** can be realized by supporting the enqueuing of kernels on a device without direct involvement by the host program; so-called **device-side enqueue**.

Device-side kernel-enqueue commands are similar to host-side kernel-enqueue commands. The kernel executing on a device (the **parent kernel**) enqueues a kernel-instance (the **child kernel**) to a device-side command-queue. This is an out-of-order command-queue and follows the same behavior as the out-of-order command-queues exposed to the host program. Commands enqueued to a device-side command-queue generate and use events to enforce order constraints just as for the command-queue on the host. These events, however, are only visible to the parent kernel running on the device. When these prerequisite events take on the value `CL_COMPLETE`, the work-groups associated with the child kernel are launched into the device's work pool. The device then schedules them for execution on the compute units of the device. Child and parent kernels execute asynchronously. However, a parent will not indicate that it is complete by setting its event to `CL_COMPLETE` until all child kernels have ended execution and have signaled completion by setting any associated events to the value `CL_COMPLETE`. Should any child kernel complete with an event status set to a negative value (i.e. abnormally terminate), the parent kernel will abnormally terminate and propagate the child's negative event value as the value of the parents event. If there are multiple children that have an event status set to a negative value, the selection of which child's negative event value is propagated is implementation-defined.

3.2.4. Synchronization

Synchronization refers to mechanisms that constrain the order of execution between two or more units of execution. Consider the following three domains of synchronization in OpenCL:

- Work-group synchronization: Constraints on the order of execution for work-items in a single work-group.
- Sub-group synchronization: Constraints on the order of execution for work-items in a single sub-group. Note: Sub-groups are missing before version 2.1
- Command synchronization: Constraints on the order of commands launched for execution.

Synchronization across work-items within a work-group is done by using a *work-group function*. These functions perform collective operations across the work-items in a work-group. Available collective operations include barriers, reductions, broadcast, prefix sums, and evaluation of a predicate. A work-group function must occur within a converged control flow; i.e. all work-items in the work-group must encounter precisely the same work-group function. For example, if a work-group function occurs within a loop, the work-items must encounter the same work-group function in the same loop iterations. All the work-items of a work-group must execute the work-group function and complete reads and writes to memory before any are allowed to continue execution beyond the work-group function. Work-group functions that apply across work-groups are not provided in OpenCL since OpenCL does not define forward-progress or ordering relations between work-groups, hence collective synchronization operations are not well defined.

Synchronization across work-items within a sub-group is done by using a *sub-group function*. These functions perform collective operations across the work-items in a sub-group. Like work-group functions, sub-group functions must also occur within a converged control flow; i.e. all work-items in the sub-group must encounter precisely the same sub-group function. Using memory operations

for sub-group synchronization should be used carefully as forward progress of sub-groups relative to each other is only supported optionally by OpenCL implementations.

Command synchronization is defined in terms of distinct **synchronization points**. The synchronization points occur between commands in host command-queues and between commands in device-side command-queues. The synchronization points defined in OpenCL include:

- **Launching a command:** A kernel-instance is launched onto a device after all events that kernel is waiting-on have been set to `CL_COMPLETE`.
- **Ending a command:** Child kernels may be enqueued such that they wait for the parent kernel to reach the *end* state before they can be launched. In this case, the ending of the parent command defines a synchronization point.
- **Completion of a command:** A kernel-instance is complete after all of the work-groups in the kernel and all of its child kernels have completed. This is signaled to the host, a parent kernel or other kernels within command-queues by setting the value of the event associated with a kernel to `CL_COMPLETE`.
- **Blocking Commands:** A blocking command defines a synchronization point between the unit of execution that calls the blocking API function and the enqueued command reaching the complete state.
- **Command-queue barrier:** The command-queue barrier ensures that all previously enqueued commands have completed before subsequently enqueued commands can be launched.
- **clFinish**: This function blocks until all previously enqueued commands in the command-queue have completed after which **clFinish** defines a synchronization point and the **clFinish** function returns.

A synchronization point between a pair of commands (A and B) assures that results of command A happens-before command B is launched. This requires that any updates to memory from command A complete and are made available to other commands before the synchronization point completes. Likewise, this requires that command B waits until after the synchronization point before loading values from global memory. The concept of a synchronization point works in a similar fashion for commands such as a barrier that apply to two sets of commands. All the commands prior to the barrier must complete and make their results available to following commands. Furthermore, any commands following the barrier must wait for the commands prior to the barrier before loading values and continuing their execution.

These *happens-before* relationships are a fundamental part of the OpenCL 2.x memory model. When applied at the level of commands, they are straightforward to define at a language level in terms of ordering relationships between different commands. Ordering memory operations inside different commands, however, requires rules more complex than can be captured by the high level concept of a synchronization point. These rules are described in detail in Memory Ordering Rules.

3.2.5. Categories of Kernels

The OpenCL execution model supports three types of kernels:

- **OpenCL kernels** are managed by the OpenCL API as kernel objects associated with kernel

functions within program objects. OpenCL program objects are created and built using OpenCL APIs. The OpenCL API includes functions to query the kernel languages and intermediate languages that may be used to create OpenCL program objects for a device.

- **Native kernels** are accessed through a host function pointer. Native kernels are queued for execution along with OpenCL kernels on a device and share memory objects with OpenCL kernels. For example, these native kernels could be functions defined in application code or exported from a library. The ability to execute native kernels is optional within OpenCL and the semantics of native kernels are implementation-defined. The OpenCL API includes functions to query capabilities of a device to determine if this capability is supported.
- **Built-in kernels** are tied to a particular device and are not built at runtime from source code in a program object. The common use of built-in kernels is to expose fixed-function hardware or firmware associated with a particular OpenCL device. The semantics of a built-in kernel are defined outside of OpenCL and hence are implementation-defined.

All three types of kernels are manipulated through the OpenCL command-queues and must conform to the synchronization points defined in the OpenCL execution model.

3.3. Memory Model

The OpenCL memory model describes the structure, contents, and behavior of the memory exposed by an OpenCL platform as an OpenCL program runs. The model allows a programmer to reason about values in memory as the host program and multiple kernel-instances execute.

An OpenCL program defines a context that includes a host, one or more devices, command-queues, and memory exposed within the context. Consider the units of execution involved with such a program. The host program runs as one or more host threads managed by the operating system running on the host (the details of which are defined outside of OpenCL). There may be multiple devices in a single context which all have access to memory objects defined by OpenCL. On a single device, multiple work-groups may execute in parallel with potentially overlapping updates to memory. Finally, within a single work-group, multiple work-items concurrently execute, once again with potentially overlapping updates to memory.

The memory model must precisely define how the values in memory as seen from each of these units of execution interact so a programmer can reason about the correctness of OpenCL programs. We define the memory model in four parts.

- Memory regions: The distinct memories visible to the host and the devices that share a context.
- Memory objects: The objects defined by the OpenCL API and their management by the host and devices.
- Shared Virtual Memory: A virtual address space exposed to both the host and the devices within a context. Note: SVM is missing before version 2.0.
- Consistency Model: Rules that define which values are observed when multiple units of execution load data from memory plus the atomic/fence operations that constrain the order of memory operations and define synchronization relationships.

3.3.1. Fundamental Memory Regions

Memory in OpenCL is divided into two parts.

- **Host Memory:** The memory directly available to the host. The detailed behavior of host memory is defined outside of OpenCL. Memory objects move between the Host and the devices through functions within the OpenCL API or through a shared virtual memory interface.
- **Device Memory:** Memory directly available to kernels executing on OpenCL devices.

Device memory consists of four named address spaces or *memory regions*:

- **Global Memory:** This memory region permits read/write access to all work-items in all work-groups running on any device within a context. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel-instance. The host allocates and initializes memory objects placed into constant memory.
- **Local Memory**: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group.
- **Private Memory**: A region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

The memory regions and their relationship to the OpenCL Platform model are summarized below. Local and private memories are always associated with a particular device. The global and constant memories, however, are shared between all devices within a given context. An OpenCL device may include a cache to support efficient access to these shared memories.

To understand memory in OpenCL, it is important to appreciate the relationships between these named address spaces. The four named address spaces available to a device are disjoint meaning they do not overlap. This is a logical relationship, however, and an implementation may choose to let these disjoint named address spaces share physical memory.

Programmers often need functions callable from kernels where the pointers manipulated by those functions can point to multiple named address spaces. This saves a programmer from the error-prone and wasteful practice of creating multiple copies of functions; one for each named address space. Therefore the global, local and private address spaces belong to a single *generic address space*. This is closely modeled after the concept of a generic address space used in the embedded C standard (ISO/IEC 9899:1999). Since they all belong to a single generic address space, the following properties are supported for pointers to named address spaces in device memory:

- A pointer to the generic address space can be cast to a pointer to a global, local or private address space
- A pointer to a global, local or private address space can be cast to a pointer to the generic address space.
- A pointer to a global, local or private address space can be implicitly converted to a pointer to the generic address space, but the converse is not allowed.

The constant address space is disjoint from the generic address space.

The generic address space is missing before version 2.0.

The addresses of memory associated with memory objects in Global memory are not preserved between kernel-instances, between a device and the host, and between devices. In this regard global memory acts as a global pool of memory objects rather than an address space. This restriction is relaxed when shared virtual memory (SVM) is used.

Shared virtual memory is missing before version 2.0.

SVM causes addresses to be meaningful between the host and all of the devices within a context hence supporting the use of pointer based data structures in OpenCL kernels. It logically extends a portion of the global memory into the host address space giving work-items access to the host address space. On platforms with hardware support for a shared address space between the host and one or more devices, SVM may also provide a more efficient way to share data between devices and the host. Details about SVM are presented in Shared Virtual Memory.

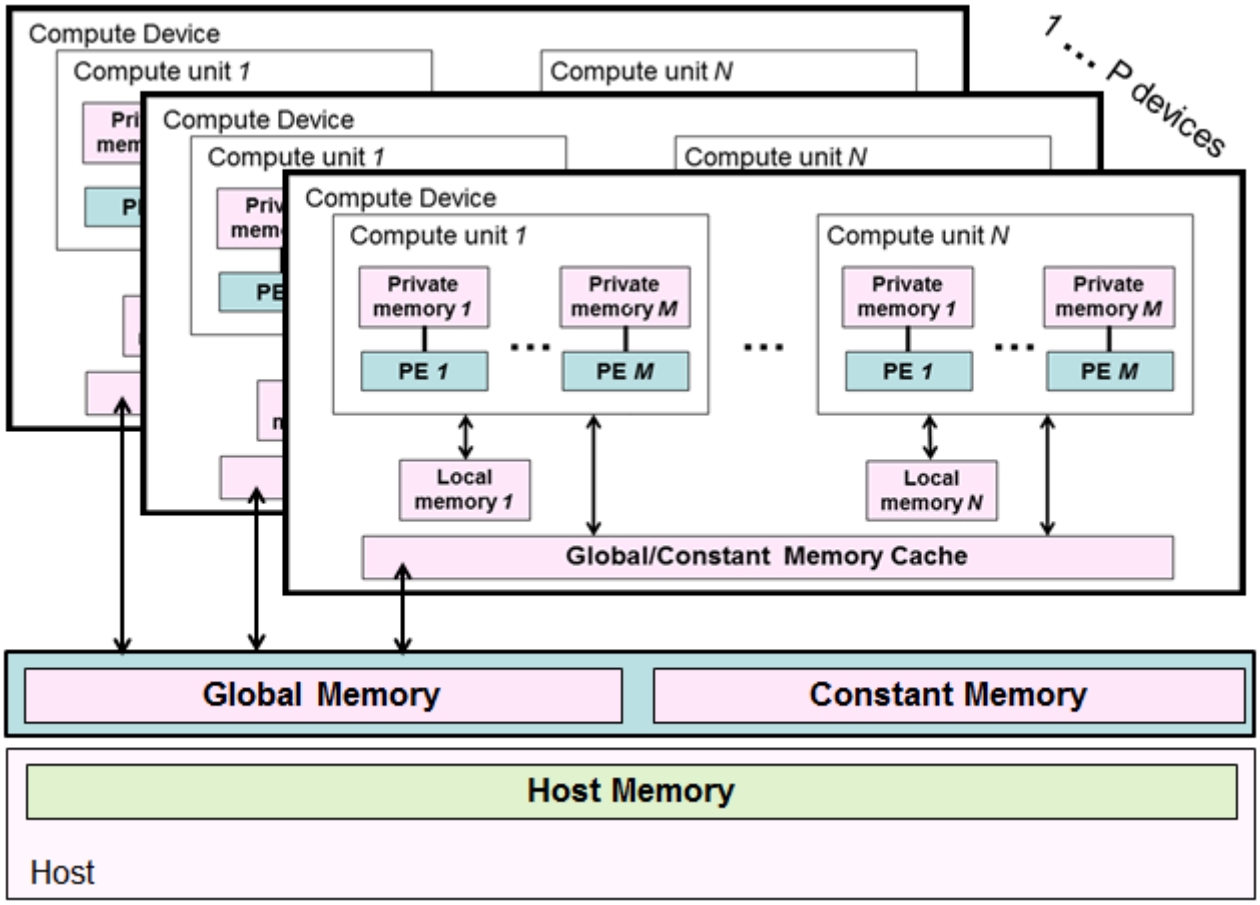

Figure 4. The named address spaces exposed in an OpenCL Platform. Global and Constant memories are shared between the one or more devices within a context, while local and private memories are associated with a single device. Each device may include an optional cache to support efficient access to their view of the global and constant address spaces.

A programmer may use the features of the memory consistency model to manage safe access to global memory from multiple work-items potentially running on one or more devices. In addition, when using shared virtual memory (SVM), the memory consistency model may also be used to

ensure that host threads safely access memory locations in the shared memory region.

3.3.2. Memory Objects

The contents of global memory are *memory objects*. A memory object is a handle to a reference counted region of global memory. Memory objects use the OpenCL type *cl_mem* and fall into three distinct classes.

- **Buffer**: A memory object stored as a block of contiguous memory and used as a general purpose object to hold data used in an OpenCL program. The types of the values within a buffer may be any of the built-in types (such as int, float), vector types, or user-defined structures. The buffer can be manipulated through pointers much as one would with any block of memory in C.
- **Image**: An image memory object holds one, two or three dimensional images. The formats are based on the standard image formats used in graphics applications. An image is an opaque data structure managed by functions defined in the OpenCL API. To optimize the manipulation of images stored in the texture memories found in many GPUs, OpenCL kernels have traditionally been disallowed from both reading and writing a single image. In OpenCL 2.0, however, we have relaxed this restriction by providing synchronization and fence operations that let programmers properly synchronize their code to safely allow a kernel to read and write a single image.
- **Pipe**: The *pipe* memory object conceptually is an ordered sequence of data items. A pipe has two endpoints: a write endpoint into which data items are inserted, and a read endpoint from which data items are removed. At any one time, only one kernel-instance may write into a pipe, and only one kernel-instance may read from a pipe. To support the producer consumer design pattern, one kernel-instance connects to the write endpoint (the producer) while another kernel instance connects to the reading endpoint (the consumer). Note: The *pipe* memory object is missing before version 2.0.

Memory objects are allocated by host APIs. The host program can provide the runtime with a pointer to a block of continuous memory to hold the memory object when the object is created (`CL_MEM_USE_HOST_PTR`). Alternatively, the physical memory can be managed by the OpenCL runtime and not be directly accessible to the host program.

Allocation and access to memory objects within the different memory regions varies between the host and work-items running on a device. This is summarized in the Memory Regions table, which describes whether the kernel or the host can allocate from a memory region, the type of allocation (static at compile time vs. dynamic at runtime) and the type of access allowed (i.e. whether the kernel or the host can read and/or write to a memory region).

Table 1. Memory Regions

		Global	**Constant**	**Local**	**Private**
Host	Allocation	Dynamic	Dynamic	Dynamic	None
	Access	Read/Write to Buffers and Images, but not Pipes	Read/Write	None	None

		Global	Constant	Local	Private
Kernel	Allocation	Static (program scope variables)	Static (program scope variables)	Static for parent kernel, Dynamic for child kernels	Static
	Access	Read/Write	Read-only	Read/Write, No access to child kernel memory	Read/Write

The Memory Regions table shows the different memory regions in OpenCL and how memory objects are allocated and accessed by the host and by an executing instance of a kernel. For kernels, we distinguish between the behavior of local memory for a parent kernel and its child kernels.

Once allocated, a memory object is made available to kernel-instances running on one or more devices. In addition to Shared Virtual Memory, there are three basic ways to manage the contents of buffers between the host and devices.

- **Read/Write/Fill commands**: The data associated with a memory object is explicitly read and written between the host and global memory regions using commands enqueued to an OpenCL command-queue. Note: Fill commands are missing before version 1.2.
- **Map/Unmap commands**: Data from the memory object is mapped into a contiguous block of memory accessed through a host accessible pointer. The host program enqueues a *map* command on block of a memory object before it can be safely manipulated by the host program. When the host program is finished working with the block of memory, the host program enqueues an *unmap* command to allow a kernel-instance to safely read and/or write the buffer.
- **Copy commands:** The data associated with a memory object is copied between two buffers, each of which may reside either on the host or on the device.

With Read/Write/Map, the commands can be blocking or non-blocking operations. The OpenCL function call for a blocking memory transfer returns once the command (memory transfer) has completed. At this point the associated memory resources on the host can be safely reused, and following operations on the host are guaranteed that the transfer has already completed. For a non-blocking memory transfer, the OpenCL function call returns as soon as the command is enqueued.

Memory objects are bound to a context and hence can appear in multiple kernel-instances running on more than one physical device. The OpenCL platform must support a large range of hardware platforms including systems that do not support a single shared address space in hardware; hence the ways memory objects can be shared between kernel-instances is restricted. The basic principle is that multiple read operations on memory objects from multiple kernel-instances that overlap in time are allowed, but mixing overlapping reads and writes into the same memory objects from different kernel-instances is only allowed when fine grained synchronization is used with Shared Virtual Memory.

When global memory is manipulated by multiple kernel-instances running on multiple devices, the OpenCL runtime system must manage the association of memory objects with a given device. In most cases the OpenCL runtime will implicitly associate a memory object with a device. A kernel-instance is naturally associated with the command-queue to which the kernel was submitted. Since a command-queue can only access a single device, the queue uniquely defines which device is

involved with any given kernel-instance; hence defining a clear association between memory objects, kernel-instances and devices. Programmers may anticipate these associations in their programs and explicitly manage association of memory objects with devices in order to improve performance.

3.3.3. Lifetime of Shared Direct3D Memory Objects

This section refers to similar Direct3D 10 and Direct3D 11 objects and concepts such as *resources*, *reference counts*, and *devices*.

Sharing is accomplished by creating an OpenCL context via the context create parameters `CL_CONTEXT_D3D10_DEVICE_KHR` (for Direct3D 10, if the `cl_khr_d3d10_sharing` extension is supported) or `CL_CONTEXT_D3D11_DEVICE_KHR` (for Direct3D 11, if the `cl_khr_d3d11_sharing` extension is supported.

An OpenCL memory object created from a Direct3D resource remains valid as long as the corresponding Direct3D resource has not been deleted. If the Direct3D resource is deleted through the Direct3D API, subsequent use of the OpenCL memory object will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

The successful creation of a `cl_context` against a Direct3D device will increment the internal Direct3D reference count on the specified device. The internal Direct3D reference count on that Direct3D device will be decremented when the OpenCL reference count on the returned OpenCL context drops to zero.

The OpenCL context and corresponding command-queues are dependent on the existence of the Direct3D device from which the OpenCL context was created. If the Direct3D device is deleted through the Direct3D API, subsequent use of the OpenCL context will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

3.3.3.1. Lifetime of Shared EGLImage Objects

An OpenCL memory object created from an EGL `EGLImage` object remains valid according to the lifetime behavior as described in the `EGL_KHR_image_base` extension.

Any `EGLImage` siblings exist in any client API context

For OpenCL this means that while the application retains a reference on the `cl_mem` (the EGL sibling), the image remains valid.

3.3.4. Lifetime of Shared OpenCL/OpenGL Memory Objects

An OpenCL memory object created from an OpenGL object (hereinafter referred to as a “shared OpenCL/OpenGL object”) remains valid as long as the corresponding OpenGL object has not been deleted. If the OpenGL object is deleted through the OpenGL API (e.g. `glDeleteBuffers`, `glDeleteTextures`, or `glDeleteRenderbuffers`), subsequent use of the OpenCL buffer or image object will result in undefined behavior, including but not limited to possible OpenCL errors and data corruption, but may not result in program termination.

The OpenCL context and corresponding command-queues are dependent on the existence of the OpenGL share group object, or the share group associated with the OpenGL context from which the

OpenCL context is created. If the OpenGL share group object or all OpenGL contexts in the share group are destroyed, any use of the OpenCL context or command-queue(s) will result in undefined behavior, which may include program termination. Applications should destroy the OpenCL command-queue(s) and OpenCL context before destroying the corresponding OpenGL share group or contexts

3.3.5. Shared Virtual Memory

Shared virtual memory is missing before version 2.0.

OpenCL extends the global memory region into the host memory region through a shared virtual memory (SVM) mechanism. There are three types of SVM in OpenCL

- **Coarse-Grained buffer SVM**: Sharing occurs at the granularity of regions of OpenCL buffer memory objects. Consistency is enforced at synchronization points and with map/unmap commands to drive updates between the host and the device. This form of SVM is similar to non-SVM use of memory; however, it lets kernel-instances share pointer-based data structures (such as linked-lists) with the host program. Program scope global variables are treated as per-device coarse-grained SVM for addressing and sharing purposes.
- **Fine-Grained buffer SVM**: Sharing occurs at the granularity of individual loads/stores into bytes within OpenCL buffer memory objects. Loads and stores may be cached. This means consistency is guaranteed at synchronization points. If the optional OpenCL atomics are supported, they can be used to provide fine-grained control of memory consistency.
- **Fine-Grained system SVM**: Sharing occurs at the granularity of individual loads/stores into bytes occurring anywhere within the host memory. Loads and stores may be cached so consistency is guaranteed at synchronization points. If the optional OpenCL atomics are supported, they can be used to provide fine-grained control of memory consistency.

Table 2. A summary of shared virtual memory (SVM) options in OpenCL

	Granularity of sharing	Memory Allocation	Mechanisms to enforce Consistency	Explicit updates between host and device
Non-SVM buffers	OpenCL Memory objects(buffer)	**clCreateBuffer clCreateBufferWithProperties**	Host synchronization points on the same or between devices.	yes, through Map and Unmap commands.
Coarse-Grained buffer SVM	OpenCL Memory objects (buffer)	**clSVMAlloc**	Host synchronization points between devices	yes, through Map and Unmap commands.
Fine-Grained buffer SVM	Bytes within OpenCL Memory objects (buffer)	**clSVMAlloc**	Synchronization points plus atomics (if supported)	No

	Granularity of sharing	Memory Allocation	Mechanisms to enforce Consistency	Explicit updates between host and device
Fine-Grained system SVM	Bytes within Host memory (system)	Host memory allocation mechanisms (e.g. malloc)	Synchronization points plus atomics (if supported)	No

Coarse-Grained buffer SVM is a required feature for OpenCL 2.0, 2.1, or 2.2 devices and an optional feature for OpenCL 3.0 or newer devices. Fine-Grained SVM is an optional feature for all OpenCL devices. The various SVM mechanisms to access host memory from the work-items associated with a kernel-instance are summarized above.

3.3.6. Memory Consistency Model for OpenCL 1.x

This memory consistency model is deprecated by version 2.0.

OpenCL 1.x uses a relaxed consistency memory model; i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load / store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier. Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for memory objects shared between enqueued commands is enforced at a synchronization point.

3.3.7. Memory Consistency Model for OpenCL 2.x

This memory consistency model is missing before version 2.0.

The OpenCL 2.x memory consistency model tells programmers what they can expect from an OpenCL 2.x or newer implementation; which memory operations are guaranteed to happen in which order and which memory values each read operation will return. The memory model tells compiler writers which restrictions they must follow when implementing compiler optimizations; which variables they can cache in registers and when they can move reads or writes around a barrier or atomic operation. The memory model also tells hardware designers about limitations on hardware optimizations; for example, when they must flush or invalidate hardware caches.

The OpenCL 2.x memory consistency model is based on the memory model from the ISO C11 programming language. To help make the presentation more precise and self-contained, we include modified paragraphs taken verbatim from the ISO C11 international standard. When a paragraph is taken or modified from the C11 standard, it is identified as such along with its original location in the C11 standard.

For programmers, the most intuitive model is the *sequential consistency* memory model. Sequential consistency interleaves the steps executed by each of the units of execution. Each access to a

memory location sees the last assignment to that location in that interleaving. While sequential consistency is relatively straightforward for a programmer to reason about, implementing sequential consistency is expensive. Therefore, the OpenCL 2.x memory consistency model is a relaxed memory consistency model; i.e. it is possible to write programs where the loads from memory violate sequential consistency. Fortunately, if a program does not contain any races and if the program only uses atomic operations that utilize the sequentially consistent memory order (the default memory ordering for OpenCL C 2.x), OpenCL programs appear to execute with sequential consistency.

Programmers can to some degree control how the memory model is relaxed by choosing the memory order for synchronization operations. The precise semantics of synchronization and the memory orders are formally defined in Memory Ordering Rules. Here, we give a high level description of how these memory orders apply to atomic operations on atomic objects shared between units of execution. The OpenCL 2.x memory orders are based on those from the ISO C11 standard memory model. They are specified in certain OpenCL C functions through the following **memory_order** enumeration constants:

- **memory_order_relaxed**: implies no order constraints. This memory order can be used safely to increment counters that are concurrently incremented, but it doesn't guarantee anything about the ordering with respect to operations to other memory locations. It can also be used, for example, to do ticket allocation and by expert programmers implementing lock-free algorithms.
- **memory_order_acquire**: A synchronization operation (fence or atomic) that has acquire semantics "acquires" side-effects from a release operation that synchronises with it: if an acquire synchronises with a release, the acquiring unit of execution will see all side-effects preceding that release (and possibly subsequent side-effects.) As part of carefully-designed protocols, programmers can use an "acquire" to safely observe the work of another unit of execution.
- **memory_order_release**: A synchronization operation (fence or atomic operation) that has release semantics "releases" side effects to an acquire operation that synchronises with it. All side effects that precede the release are included in the release. As part of carefully-designed protocols, programmers can use a "release" to make changes made in one unit of execution visible to other units of execution.

In general, no acquire must *always* synchronise with any particular release. However, synchronisation can be forced by certain executions. See the description of Fence Operations for detailed rules for when synchronisation must occur.

- **memory_order_acq_rel**: A synchronization operation with acquire-release semantics has the properties of both the acquire and release memory orders. It is typically used to order read-modify-write operations.
- **memory_order_seq_cst**: The loads and stores of each unit of execution appear to execute in program (i.e., sequenced-before) order, and the loads and stores from different units of execution appear to be simply interleaved.

Regardless of which memory order is specified, resolving constraints on memory operations across a heterogeneous platform adds considerable overhead to the execution of a program. An OpenCL platform may be able to optimize certain operations that depend on the features of the memory

consistency model by restricting the scope of the memory operations. Distinct memory scopes are defined by the values of the **memory_scope** enumeration constant:

- **memory_scope_work_item**: memory-ordering constraints only apply within the work-item [1].
- **memory_scope_sub_group**: memory-ordering constraints only apply within the sub-group.
- **memory_scope_work_group**: memory-ordering constraints only apply to work-items executing within a single work-group.
- **memory_scope_device:** memory-ordering constraints only apply to work-items executing on a single device
- **memory_scope_all_svm_devices**: memory-ordering constraints apply to work-items executing across multiple devices and (when using SVM) the host. A release performed with **memory_scope_all_svm_devices** to a buffer that does not have the `CL_MEM_SVM_ATOMICS` flag set will commit to at least **memory_scope_device** visibility, with full synchronization of the buffer at a queue synchronization point (e.g. an OpenCL event).
- **memory_scope_all_devices**: an alias for **memory_scope_all_svm_devices**.

These memory scopes define a hierarchy of visibilities when analyzing the ordering constraints of memory operations. For example if a programmer knows that a sequence of memory operations will only be associated with a collection of work-items from a single work-group (and hence will run on a single device), the implementation is spared the overhead of managing the memory orders across other devices within the same context. This can substantially reduce overhead in a program. All memory scopes are valid when used on global memory or local memory. For local memory, all visibility is constrained to within a given work-group and scopes wider than **memory_scope_work_group** carry no additional meaning.

In the following subsections (leading up to OpenCL Framework), we will explain the synchronization constructs and detailed rules needed to use OpenCL's 2.x relaxed memory models. It is important to appreciate, however, that many programs do not benefit from relaxed memory models. Even expert programmers have a difficult time using atomics and fences to write correct programs with relaxed memory models. A large number of OpenCL programs can be written using a simplified memory model. This is accomplished by following these guidelines.

- Write programs that manage safe sharing of global memory objects through the synchronization points defined by the command-queues.
- Restrict low level synchronization inside work-groups to the work-group functions such as barrier.
- If you want sequential consistency behavior with system allocations or fine-grain SVM buffers with atomics support, use only **memory_order_seq_cst** operations with the scope **memory_scope_all_svm_devices**.
- If you want sequential consistency behavior when not using system allocations or fine-grain SVM buffers with atomics support, use only **memory_order_seq_cst** operations with the scope **memory_scope_device** or **memory_scope_all_svm_devices**.
- Ensure your program has no races.

If these guidelines are followed in your OpenCL programs, you can skip the detailed rules behind

the relaxed memory models and go directly to OpenCL Framework.

3.3.8. Overview of Atomic and Fence Operations

OpenCL C 2.x has a number of *synchronization operations* that are used to define memory order constraints in a program. They play a special role in controlling how memory operations in one unit of execution (such as work-items or, when using SVM a host thread) are made visible to another. There are two types of synchronization operations in OpenCL; *atomic operations* and *fences*.

Atomic operations are indivisible. They either occur completely or not at all. These operations are used to order memory operations between units of execution and hence they are parameterized with the memory order and memory scope parameters defined by the OpenCL memory consistency model. The atomic operations for OpenCL kernel languages are similar to the corresponding operations defined by the C11 standard.

The OpenCL C 2.x atomic operations apply to variables of an atomic type (a subset of those in the C11 standard).

An atomic operation on one or more memory locations is either an acquire operation, a release operation, or both an acquire and release operation. An atomic operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which do not have synchronization properties, and atomic read-modify-write operations, which have special characteristics. [C11 standard, Section 5.1.2.4, paragraph 5, modified.]

The orders **memory_order_acquire** (used for reads), **memory_order_release** (used for writes), and **memory_order_acq_rel** (used for read-modify-write operations) are used for simple communication between units of execution using shared variables. Informally, executing a **memory_order_release** on an atomic object **A** makes all previous side effects visible to any unit of execution that later executes a **memory_order_acquire** on **A**. The orders **memory_order_acquire**, **memory_order_release**, and **memory_order_acq_rel** do not provide sequential consistency for race-free programs because they will not ensure that atomic stores followed by atomic loads become visible to other threads in that order.

The fence operation is **atomic_work_item_fence**, which includes a memory order argument as well as memory scope and memory flag arguments. Depending on the memory order argument, this operation:

- has no effects, if the memory order is **memory_order_relaxed**;
- is an acquire fence, if the memory order is **memory_order_acquire**;
- is a release fence, if the memory order is **memory_order_release**;
- is both an acquire fence and a release fence, if the memory order is **memory_order_acq_rel**;
- is a sequentially-consistent fence with both acquire and release semantics, if the memory order is **memory_order_seq_cst**.

If specified, the cl_mem_fence_flags argument must be `CLK_IMAGE_MEM_FENCE`, `CLK_GLOBAL_MEM_FENCE`, `CLK_LOCAL_MEM_FENCE`, or `CLK_GLOBAL_MEM_FENCE | CLK_LOCAL_MEM_FENCE`.

The **atomic_work_item_fence** built-in function must be used with `CLK_IMAGE_MEM_FENCE` to make sure that sampler-less writes are visible to later reads by the same work-item. Without use of the **atomic_work_item_fence** function, write-read coherence on image objects is not guaranteed: if a work-item reads from an image to which it has previously written without an intervening **atomic_work_item_fence**, it is not guaranteed that those previous writes are visible to the work-item.

The synchronization operations in OpenCL C 2.x can be parameterized by a memory scope. Memory scopes control the extent that an atomic operation or fence is visible with respect to the memory model. These memory scopes may be used when performing atomic operations and fences on global memory and local memory. When used on global memory visibility is bounded by the capabilities of that memory. When used on a fine-grained non-atomic SVM buffer, a coarse-grained SVM buffer, or a non-SVM buffer, operations parameterized with **memory_scope_all_svm_devices** will behave as if they were parameterized with **memory_scope_device**. When used on local memory, visibility is bounded by the work-group and, as a result, memory_scope with wider visibility than **memory_scope_work_group** will be reduced to **memory_scope_work_group**.

Two actions **A** and **B** are defined to have an inclusive scope if:

- **A** and **B** are executed by work-items within the same sub-group and the memory scope of **A** and **B** is **memory_scope_sub_group** or wider.
- **A** and **B** are executed by work-items within the same work-group and the memory scope of **A** and **B** is **memory_scope_work_group** or wider.
- **A** and **B** are executed by work-items in the same kernel-instance or a child-kernel of the kernel-instance, and the memory scope of **A** and **B** is **memory_scope_device** or wider.
- **A** and **B** are executed by work-items on the same device, **A** and **B** operate on an SVM allocation, and the memory scope of **A** and **B** is **memory_scope_device** or wider.
- **A** and **B** are executed by a host thread or by work-items on one or more devices that support SVM atomic operations, **A** and **B** operate on an SVM allocation, and the memory scope of **A** and **B** is **memory_scope_all_svm_devices** or **memory_scope_all_devices**.

Prior to OpenCL 3.1, the memory scope of **A** and **B** must additionally be the same for the actions to have an inclusive scope. For OpenCL 3.1 and newer, the memory scope of **A** and **B** may be different.

3.3.9. Memory Ordering Rules

Fundamentally, the issue in a memory model is to understand the orderings in time of modifications to objects in memory. Modifying an object or calling a function that modifies an object are side effects, i.e. changes in the state of the execution environment. Evaluation of an expression in general includes both value computations and initiation of side effects. Value computation for an lvalue expression includes determining the identity of the designated object. [C11 standard, Section 5.1.2.3, paragraph 2, modified.]

We assume that the OpenCL kernel language and host programming languages have a sequenced-before relation between the evaluations executed by a single unit of execution. This sequenced-before relation is an asymmetric, transitive, pair-wise relation between those evaluations, which induces a partial order among them. Given any two evaluations **A** and **B**, if **A** is sequenced-before **B**,

then the execution of **A** shall precede the execution of **B**. (Conversely, if **A** is sequenced-before **B**, then **B** is sequenced-after **A**.) If **A** is not sequenced-before or sequenced-after **B**, then **A** and **B** are unsequenced. Evaluations **A** and **B** are indeterminately sequenced when **A** is either sequenced-before or sequenced-after **B**, but it is unspecified which. [C11 standard, Section 5.1.2.3, paragraph 3, modified.]

Sequenced-before is a partial order of the operations executed by a single unit of execution (e.g. a host thread or work-item). It generally corresponds to the source program order of those operations, and is partial because of the undefined argument evaluation order of the OpenCL C kernel language.

In an OpenCL kernel language, the value of an object visible to a work-item W at a particular point is the initial value of the object, a value stored in the object by W, or a value stored in the object by another work-item or host thread, according to the rules below. Depending on details of the host programming language, the value of an object visible to a host thread may also be the value stored in that object by another work-item or host thread. [C11 standard, Section 5.1.2.4, paragraph 2, modified.]

Two expression evaluations conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. [C11 standard, Section 5.1.2.4, paragraph 4.]

All modifications to a particular atomic object **M** occur in some particular total order, called the modification order of **M**. If **A** and **B** are modifications of an atomic object **M**, and **A** happens-before **B**, then **A** shall precede **B** in the modification order of **M**, which is defined below. Note that the modification order of an atomic object **M** is independent of whether **M** is in local or global memory. [C11 standard, Section 5.1.2.4, paragraph 7, modified.]

A release sequence begins with a release operation **A** on an atomic object **M** and is the maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first operation is **A** and every subsequent operation either is performed by the same work-item or host thread that performed the release or is an atomic read-modify-write operation. [C11 standard, Section 5.1.2.4, paragraph 10, modified.]

OpenCL's local and global memories are disjoint. Kernels may access both kinds of memory while host threads may only access global memory. Furthermore, the *flags* argument of OpenCL's work_group_barrier function specifies which memory operations the function will make visible: these memory operations can be, for example, just the ones to local memory, or the ones to global memory, or both. Since the visibility of memory operations can be specified for local memory separately from global memory, we define two related but independent relations, *global-synchronizes-with* and *local-synchronizes-with*. Certain operations on global memory may global-synchronize-with other operations performed by another work-item or host thread. An example is a release atomic operation in one work- item that global-synchronizes-with an acquire atomic operation in a second work-item. Similarly, certain atomic operations on local objects in kernels can local-synchronize- with other atomic operations on those local objects. [C11 standard, Section 5.1.2.4, paragraph 11, modified.]

We define two separate happens-before relations: global-happens-before and local-happens-before.

A global memory action **A** global-happens-before a global memory action **B** if

- **A** is sequenced before **B**, or
- **A** global-synchronizes-with **B**, or
- For some global memory action **C**, **A** global-happens-before **C** and **C** global-happens-before **B**.

A local memory action **A** local-happens-before a local memory action **B** if

- **A** is sequenced before **B**, or
- **A** local-synchronizes-with **B**, or
- For some local memory action **C**, **A** local-happens-before **C** and **C** local-happens-before **B**.

An implementation of the OpenCL 2.x memory consistency model shall ensure that no program execution demonstrates a cycle in either the local-happens-before relation or the global-happens-before relation.

The global- and local-happens-before relations are critical to defining what values are read and when data races occur. The global-happens-before relation, for example, defines what global memory operations definitely happen before what other global memory operations. If an operation **A** global-happens-before operation **B** then **A** must occur before **B**; in particular, any write done by **A** will be visible to **B**. The local-happens-before relation has similar properties for local memory. Programmers can use the local- and global-happens-before relations to reason about the order of program actions.

A visible side effect **A** on a global object **M** with respect to a value computation **B** of **M** satisfies the conditions:

- **A** global-happens-before **B**, and
- there is no other side effect **X** to **M** such that **A** global-happens-before **X** and **X** global-happens-before **B**.

We define visible side effects for local objects **M** similarly. The value of a non-atomic scalar object **M**, as determined by evaluation **B**, shall be the value stored by the visible side effect **A**. [C11 standard, Section 5.1.2.4, paragraph 19, modified.]

The execution of a program contains a data race if it contains two conflicting actions **A** and **B** in different units of execution, and

- (1) at least one of **A** or **B** is not atomic, or **A** and **B** do not have inclusive memory scope, and
- (2) the actions are global actions unordered by the global-happens-before relation or are local actions unordered by the local-happens-before relation.

Any such data race results in undefined behavior. [C11 standard, Section 5.1.2.4, paragraph 25, modified.]

We also define the visible sequence of side effects on local and global atomic objects. The remaining paragraphs of this subsection define this sequence for a global atomic object **M**; the visible sequence of side effects for a local atomic object is defined similarly by using the local-happens-

before relation.

The visible sequence of side effects on a global atomic object **M**, with respect to a value computation **B** of **M**, is a maximal contiguous sub-sequence of side effects in the modification order of **M**, where the first side effect is visible with respect to **B**, and for every side effect, it is not the case that **B** global-happens-before it. The value of **M**, as determined by evaluation **B**, shall be the value stored by some operation in the visible sequence of **M** with respect to **B**. [C11 standard, Section 5.1.2.4, paragraph 22, modified.]

If an operation **A** that modifies an atomic object **M** global-happens-before an operation **B** that modifies **M**, then **A** shall be earlier than **B** in the modification order of **M**. This requirement is known as write-write coherence.

If a value computation **A** of an atomic object **M** global-happens-before a value computation **B** of **M**, and **A** takes its value from a side effect **X** on **M**, then the value computed by **B** shall either equal the value stored by **X**, or be the value stored by a side effect **Y** on **M**, where **Y** follows **X** in the modification order of **M**. This requirement is known as read-read coherence. [C11 standard, Section 5.1.2.4, paragraph 22, modified.]

If a value computation **A** of an atomic object **M** global-happens-before an operation **B** on **M**, then **A** shall take its value from a side effect **X** on **M**, where **X** precedes **B** in the modification order of **M**. This requirement is known as read-write coherence.

If a side effect **X** on an atomic object **M** global-happens-before a value computation **B** of **M**, then the evaluation **B** shall take its value from **X** or from a side effect **Y** that follows **X** in the modification order of **M**. This requirement is known as write-read coherence.

3.3.9.1. Atomic Operations

This and following sections describe how different program actions in kernel C code and the host program contribute to the local- and global-happens-before relations. This section discusses ordering rules for OpenCL C 2.x atomic operations.

The Memory Consistency Model for OpenCL 2.x section defines the enumerated type **memory_order**.

- For **memory_order_relaxed**, there is no memory ordering.
- For **memory_order_release**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a store operation performs a release operation on the affected memory location.
- For **memory_order_acquire**, **memory_order_acq_rel**, and **memory_order_seq_cst**, a load operation performs an acquire operation on the affected memory location. [C11 standard, Section 7.17.3, paragraphs 2-4, modified.]

Certain built-in functions synchronize with other built-in functions performed by another unit of execution. This is true for pairs of release and acquire operations under specific circumstances. An atomic operation **A** that performs a release operation on a global object **M** global-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. A similar rule holds for atomic operations on objects in local memory: an atomic operation **A** that performs a release operation on a local object

M local-synchronizes-with an atomic operation **B** that performs an acquire operation on **M** and reads a value written by any side effect in the release sequence headed by **A**. [C11 standard, Section 5.1.2.4, paragraph 11, modified.]

Atomic operations specifying **memory_order_relaxed** are relaxed only with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object.

There shall exist a single total order **S** for all **memory_order_seq_cst** operations that is consistent with the modification orders for all affected locations, as well as the appropriate global-happens-before and local-happens-before orders for those locations, such that each **memory_order_seq_cst** operation **B** that loads a value from an atomic object **M** in global or local memory observes one of the following values:

- the result of the last modification **A** of **M** that precedes **B** in **S**, if it exists, or
- if **A** exists, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not **memory_order_seq_cst** and that does not happen before **A**, or
- if **A** does not exist, the result of some modification of **M** in the visible sequence of side effects with respect to **B** that is not **memory_order_seq_cst**. [C11 standard, Section 7.17.3, paragraph 6, modified.]

Let X and Y be two **memory_order_seq_cst** operations. If X local-synchronizes-with or global-synchronizes-with Y then X both local-synchronizes-with Y and global-synchronizes-with Y.

If the total order **S** exists, the following rules hold:

- For an atomic operation **B** that reads the value of an atomic object **M**, if there is a **memory_order_seq_cst** fence **X** sequenced-before **B**, then **B** observes either the last **memory_order_seq_cst** modification of **M** preceding **X** in the total order **S** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 9.]
- For atomic operations **A** and **B** on an atomic object **M**, where **A** modifies **M** and **B** takes its value, if there is a **memory_order_seq_cst** fence **X** such that **A** is sequenced-before **X** and **B** follows **X** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 10.]
- For atomic operations **A** and **B** on an atomic object **M**, where **A** modifies **M** and **B** takes its value, if there are **memory_order_seq_cst** fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** observes either the effects of **A** or a later modification of **M** in its modification order. [C11 standard, Section 7.17.3, paragraph 11.]
- For atomic operations **A** and **B** on an atomic object **M**, if there are **memory_order_seq_cst** fences **X** and **Y** such that **A** is sequenced-before **X**, **Y** is sequenced-before **B**, and **X** precedes **Y** in **S**, then **B** occurs later than **A** in the modification order of **M**.

memory_order_seq_cst ensures sequential consistency only for a program that is (1) free of data races, and (2) exclusively uses **memory_order_seq_cst** synchronization operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, **memory_order_seq_cst**

> fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications.

Atomic read-modify-write operations should always read the last value (in the modification order) stored before the write associated with the read-modify-write operation. [C11 standard, Section 7.17.3, paragraph 12.]

Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.

Note: Under the rules described above, and independent to the previously footnoted C++ issue, it is known that *x == y == 42* is a valid final state in the following problematic example:

```
global atomic_int x = ATOMIC_VAR_INIT(0);
local atomic_int y = ATOMIC_VAR_INIT(0);

unit_of_execution_1:
... [execution not reading or writing x or y, leading up to:]
int t = atomic_load_explicit(&y, memory_order_acquire);
atomic_store_explicit(&x, t, memory_order_release);

unit_of_execution_2:
... [execution not reading or writing x or y, leading up to:]
int t = atomic_load_explicit(&x, memory_order_acquire);
atomic_store_explicit(&y, t, memory_order_release);
```

This is not useful behavior and implementations should not exploit this phenomenon. It should be expected that in the future this may be disallowed by appropriate updates to the memory model description by the OpenCL committee.

Implementations should make atomic stores visible to atomic loads within a reasonable amount of time. [C11 standard, Section 7.17.3, paragraph 16.]

As long as the following conditions are met, a host program sharing SVM memory with a kernel executing on one or more OpenCL 2.x or newer devices may use atomic and synchronization operations to ensure that its assignments, and those of the kernel, are visible to each other:

1. Either fine-grained buffer or fine-grained system SVM must be used to share memory. While coarse-grained buffer SVM allocations may support atomic operations, visibility on these allocations is not guaranteed except at map and unmap operations.
2. The optional OpenCL SVM atomic-controlled visibility specified by provision of the `CL_MEM_SVM_ATOMICS` flag must be supported by the device and the flag provided to the SVM buffer on allocation.
3. The host atomic and synchronization operations must be compatible with those of an OpenCL kernel language. This requires that the size and representation of the data types that the host atomic operations act on be consistent with the OpenCL kernel language atomic types.

If these conditions are met, the host operations will apply at all_svm_devices scope.

3.3.9.2. Fence Operations

This section describes how the OpenCL C 2.x fence operations contribute to the local- and global-happens-before relations.

Earlier, we introduced synchronization primitives called fences. Fences can utilize the acquire memory order, release memory order, or both. A fence with acquire semantics is called an acquire fence; a fence with release semantics is called a release fence. The overview of atomic and fence operations section describes the memory orders that result in acquire and release fences.

A global release fence **A** global-synchronizes-with a global acquire fence **B** if there exist atomic operations **X** and **Y**, both operating on some global atomic object **M**, such that **A** is sequenced-before **X**, **X** modifies **M**, **Y** is sequenced-before **B**, **Y** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and that the scopes of **A**, **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A global release fence **A** global-synchronizes-with an atomic operation **B** that performs an acquire operation on a global atomic object **M** if there exists an atomic operation **X** such that **A** is sequenced-before **X**, **X** modifies **M**, **B** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a global atomic object **M** global-synchronizes-with a global acquire fence **B** if there exists some atomic operation **X** on **M** such that **X** is sequenced-before **B** and reads the value written by **A** or a value written by any side effect in the release sequence headed by **A**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

A local release fence **A** local-synchronizes-with a local acquire fence **B** if there exist atomic operations **X** and **Y**, both operating on some local atomic object **M**, such that **A** is sequenced-before **X**, **X** modifies **M**, **Y** is sequenced-before **B**, and **Y** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 2, modified.]

A local release fence **A** local-synchronizes-with an atomic operation **B** that performs an acquire operation on a local atomic object **M** if there exists an atomic operation **X** such that **A** is sequenced-before **X**, **X** modifies **M**, and **B** reads the value written by **X** or a value written by any side effect in the hypothetical release sequence **X** would head if it were a release operation, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 3, modified.]

An atomic operation **A** that is a release operation on a local atomic object **M** local-synchronizes-with a local acquire fence **B** if there exists some atomic operation **X** on **M** such that **X** is sequenced-before **B** and reads the value written by **A** or a value written by any side effect in the release sequence headed by **A**, and the scopes of **A** and **B** are inclusive. [C11 standard, Section 7.17.4, paragraph 4, modified.]

Let **X** and **Y** be two work-item fences that each have both the `CLK_GLOBAL_MEM_FENCE` and `CLK_LOCAL_MEM_FENCE` flags set. **X** global-synchronizes-with **Y** and **X** local synchronizes with **Y** if the

conditions required for **X** to global-synchronize with **Y** are met, the conditions required for **X** to local-synchronize-with **Y** are met, or both sets of conditions are met.

3.3.9.3. Work-Group Functions

The OpenCL kernel execution model includes collective operations across the work-items within a single work-group. These are called work-group functions, and include functions such as barriers, scans, reductions, and broadcasts. We will first discuss the work-group barrier function. Other work-group functions are discussed afterwards.

The barrier function provides a mechanism for a kernel to synchronize the work-items within a single work-group: informally, each work-item of the work-group must execute the barrier before any are allowed to proceed. It also orders memory operations to a specified combination of one or more address spaces such as local memory or global memory, in a similar manner to a fence.

To precisely specify the memory ordering semantics for barrier, we need to distinguish between a dynamic and a static instance of the call to a barrier. A call to a barrier can appear in a loop, for example, and each execution of the same static barrier call results in a new dynamic instance of the barrier that will independently synchronize a work-groups work-items.

A work-item executing a dynamic instance of a barrier results in two operations, both fences, that are called the entry and exit fences. These fences obey all the rules for fences specified elsewhere in this chapter as well as the following:

- The entry fence is a release fence with the same flags and scope as requested for the barrier.
- The exit fence is an acquire fence with the same flags and scope as requested for the barrier.
- For each work-item the entry fence is sequenced before the exit fence.
- If the flags have `CLK_GLOBAL_MEM_FENCE` set then for each work-item the entry fence global-synchronizes-with the exit fence of all other work-items in the same work-group.
- If the flags have `CLK_LOCAL_MEM_FENCE` set then for each work-item the entry fence local-synchronizes-with the exit fence of all other work-items in the same work-group.

Other work-group functions include such functions as scans, reductions, and broadcasts, and are described in the kernel language and IL specifications. The use of these work-group functions implies sequenced-before relationships between statements within the execution of a single work-item in order to satisfy data dependencies. For example, a work-item that provides a value to a work-group function must behave as if it generates that value before beginning execution of that work-group function. Furthermore, the programmer must ensure that all work-items in a work-group must execute the same work-group function call site, or dynamic work-group function instance.

3.3.9.4. Sub-Group Functions

Sub-group functions are missing before version 2.1. Also see `cl_khr_subgroups`.

The OpenCL kernel execution model includes collective operations across the work-items within a single sub-group. These are called sub-group functions. We will first discuss the sub-group barrier. Other sub-group functions are discussed afterwards.

The barrier function provides a mechanism for a kernel to synchronize the work-items within a single sub-group: informally, each work-item of the sub-group must execute the barrier before any are allowed to proceed. It also orders memory operations to a specified combination of one or more address spaces such as local memory or global memory, in a similar manner to a fence.

To precisely specify the memory ordering semantics for barrier, we need to distinguish between a dynamic and a static instance of the call to a barrier. A call to a barrier can appear in a loop, for example, and each execution of the same static barrier call results in a new dynamic instance of the barrier that will independently synchronize the work-items in a sub-group.

A work-item executing a dynamic instance of a barrier results in two operations, both fences, that are called the entry and exit fences. These fences obey all the rules for fences specified elsewhere in this chapter as well as the following:

- The entry fence is a release fence with the same flags and scope as requested for the barrier.
- The exit fence is an acquire fence with the same flags and scope as requested for the barrier.
- For each work-item the entry fence is sequenced before the exit fence.
- If the flags have `CLK_GLOBAL_MEM_FENCE` set then for each work-item the entry fence global-synchronizes-with the exit fence of all other work-items in the same sub-group.
- If the flags have `CLK_LOCAL_MEM_FENCE` set then for each work-item the entry fence local-synchronizes-with the exit fence of all other work-items in the same sub-group.

Other sub-group functions include such functions as scans, reductions, and broadcasts, and are described in the kernel languages and IL specifications. The use of these sub-group functions implies sequenced-before relationships between statements within the execution of a single work-item in order to satisfy data dependencies. For example, a work-item that provides a value to a sub-group function must behave as if it generates that value before beginning execution of that sub-group function. Furthermore, the programmer must ensure that all work-items in a sub-group must execute the same sub-group function call site, or dynamic sub-group function instance.

3.3.9.5. Host-Side and Device-Side Commands

This section describes how the OpenCL API functions associated with command-queues contribute to happens-before relations. There are two types of command-queues and associated API functions in OpenCL 2.x; *host command-queues* and *device command-queues*. The interaction of these command-queues with the memory model are for the most part equivalent. In a few cases, the rules only applies to the host command-queue. We will indicate these special cases by specifically denoting the host command-queue in the memory ordering rule. SVM memory consistency in such instances is implied only with respect to synchronizing host commands.

Memory ordering rules in this section apply to all memory objects (buffers, images and pipes) as well as to SVM allocations where no earlier, and more fine-grained, rules apply.

In the remainder of this section, we assume that each command **C** enqueued onto a command-queue has an associated event object **E** that signals its execution status, regardless of whether **E** was returned to the unit of execution that enqueued **C**. We also distinguish between the API function call that enqueues a command **C** and creates an event **E**, the execution of **C**, and the completion of **C**(which marks the event **E** as complete).

The ordering and synchronization rules for API commands are defined as following:

1. If an API function call **X** enqueues a command **C**, then **X** global-synchronizes-with **C**. For example, a host API function to enqueue a kernel global-synchronizes-with the start of that kernel-instances execution, so that memory updates sequenced-before the enqueue kernel function call will global-happen-before any kernel reads or writes to those same memory locations. For a device-side enqueue, global memory updates sequenced before **X** happens-before **C** reads or writes to those memory locations only in the case of fine-grained SVM.
2. If **E** is an event upon which a command **C** waits, then **E** global-synchronizes-with **C**. In particular, if **C** waits on an event **E** that is tracking the execution status of the command **C1**, then memory operations done by **C1** will global-happen-before memory operations done by **C**. As an example, assume we have an OpenCL program using coarse-grain SVM sharing that enqueues a kernel to a host command-queue to manipulate the contents of a region of a buffer that the host thread then accesses after the kernel completes. To do this, the host thread can call **clEnqueueMapBuffer** to enqueue a blocking-mode map command to map that buffer region, specifying that the map command must wait on an event signaling the kernels completion. When **clEnqueueMapBuffer** returns, any memory operations performed by the kernel to that buffer region will global- happen-before subsequent memory operations made by the host thread.
3. If a command **C** has an event **E** that signals its completion, then **C** global-synchronizes-with **E**.
4. For a command **C** enqueued to a host-side command-queue, if **C** has an event **E** that signals its completion, then **E** global-synchronizes-with an API call **X** that waits on **E**. For example, if a host thread or kernel-instance calls the wait-for-events function on **E** (e.g. the **clWaitForEvents** function called from a host thread), then **E** global-synchronizes-with that wait-for-events function call.
5. If commands **C** and **C1** are enqueued in that sequence onto an in-order command-queue, then the event (including the event implied between **C** and **C1** due to the in-order queue) signaling **C**'s completion global-synchronizes-with **C1**.
6. If an API call enqueues a marker command **C** with an empty list of events upon which **C** should wait, then the events of all commands enqueued prior to **C** in the command-queue global-synchronize-with **C**.
7. If a host API call enqueues a command-queue barrier command **C** with an empty list of events on which **C** should wait, then the events of all commands enqueued prior to **C** in the command-queue global-synchronize-with **C**. In addition, the event signaling the completion of **C** global-synchronizes-with all commands enqueued after **C** in the command-queue.
8. If a host thread executes a **clFinish** call **X**, then the events of all commands enqueued prior to **X** in the command-queue global-synchronizes-with **X**.
9. The start of a kernel-instance **K** global-synchronizes-with all operations in the work-items of **K**. Note that this includes the execution of any atomic operations by the work-items in a program using fine-grain SVM.
10. All operations of all work-items of a kernel-instance **K** global-synchronizes-with the event signaling the completion of **K**. Note that this also includes the execution of any atomic operations by the work-items in a program using fine-grain SVM.
11. If a callback procedure **P** is registered on an event **E**, then **E** global-synchronizes-with all

operations of **P**. Note that callback procedures are only defined for commands within host command-queues.

12. If **C** is a command that waits for an event **E**'s completion, and API function call **X** sets user event **E**'s status to `CL_COMPLETE` (for example, from a host thread using a **clSetUserEventStatus** function), then **X** global-synchronizes-with **C**.
13. If a device enqueues a command **C** with the `CLK_ENQUEUE_FLAGS_WAIT_KERNEL` flag, then the end state of the parent kernel-instance global-synchronizes with **C**.
14. If a work-group enqueues a command **C** with the `CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP` flag, then the end state of the work-group global-synchronizes with **C**.

When using an out-of-order command-queue, a wait on an event or a marker or command-queue barrier command can be used to ensure the correct ordering of dependent commands. In those cases, the wait for the event or the marker or barrier command will provide the necessary global-synchronizes-with relation.

In this situation:

- access to shared locations or disjoint locations in a single `cl_mem` object when using atomic operations from different kernel-instances enqueued from the host such that one or more of the atomic operations is a write is implementation-defined and correct behavior is not guaranteed except at synchronization points.
- access to shared locations or disjoint locations in a single `cl_mem` object when using atomic operations from different kernel-instances consisting of a parent kernel and any number of child kernels enqueued by that kernel is guaranteed under the memory ordering rules described earlier in this section.
- access to shared locations or disjoint locations in a single program scope global variable, coarse-grained SVM allocation or fine-grained SVM allocation when using atomic operations from different kernel-instances enqueued from the host to a single device is guaranteed under the memory ordering rules described earlier in this section.

If fine-grain SVM is used but without support for SVM atomic operations, then the host and devices can concurrently read the same memory locations and can concurrently update non-overlapping memory regions, but attempts to update the same memory locations are undefined. Memory consistency is guaranteed at the OpenCL synchronization points without the need for calls to **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject**. For fine-grained SVM buffers it is guaranteed that at synchronization points only values written by the kernel will be updated. No writes to fine-grained SVM buffers can be introduced that were not in the original program.

In the remainder of this section, we discuss a few points regarding the ordering rules for commands with a host command-queue.

In an OpenCL 1.x implementation a synchronization point is a kernel-instance or host program location where the contents of memory visible to different work-items or command-queue commands are the same. It also says that waiting on an event and a command-queue barrier are synchronization points between commands in command-queues. Four of the rules listed above (2, 4, 7, and 8) cover these OpenCL synchronization points.

A map operation (**clEnqueueMapBuffer** or **clEnqueueMapImage**) performed on a non-SVM buffer or a coarse-grained SVM buffer is allowed to overwrite the entire target region with the latest runtime view of the data as seen by the command with which the map operation synchronizes, whether the values were written by the executing kernels or not. Any values that were changed within this region by another kernel or host thread while the kernel synchronizing with the map operation was executing may be overwritten by the map operation.

Access to non-SVM `cl_mem` buffers and coarse-grained SVM allocations is ordered at synchronization points between host commands. In the presence of an out-of-order command-queue or a set of command-queues mapped to the same device, multiple kernel-instances may execute concurrently on the same device.

3.4. The OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- **OpenCL Platform layer**: The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.
- **OpenCL Runtime**: The runtime allows the host program to manipulate contexts once they have been created.
- **OpenCL Compiler**: The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL compiler may build program executables from OpenCL C source strings, the SPIR-V intermediate language, or device-specific program binary objects, depending on the capabilities of a device. Other kernel languages or intermediate languages may be supported by some implementations.

3.4.1. Mixed Version Support

Mixed version support is missing before version 1.1.

OpenCL supports devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification. There are three version identifiers to consider for an OpenCL system: the platform version, the version of a device, and the version(s) of the kernel language or IL supported on a device.

The platform version indicates the version of the OpenCL runtime that is supported. This includes all of the APIs that the host can use to interact with resources exposed by the OpenCL runtime; including contexts, memory objects, devices, and command-queues.

The device version is an indication of the device's capabilities separate from the runtime and compiler as represented by the device info returned by **clGetDeviceInfo**. Examples of attributes associated with the device version are resource limits (e.g., minimum size of local memory per compute unit) and extended functionality (e.g., list of supported KHR extensions). The version returned corresponds to the highest version of the OpenCL specification for which the device is conformant, but is not higher than the platform version.

The language version for a device represents the OpenCL programming language features a developer can assume are supported on a given device. The version reported is the highest version of the language supported.

3.4.2. Backwards Compatibility

Backwards compatibility is an important goal for the OpenCL standard. Backwards compatibility is expected such that a device will consume earlier versions of the OpenCL C programming languages and the SPIR-V intermediate language with the following minimum requirements:

- An OpenCL 1.x device must support at least one 1.x version of the OpenCL C programming language.
- An OpenCL 2.0 device must support all the requirements of an OpenCL 1.2 device in addition to the OpenCL C 2.0 programming language. If multiple language versions are supported, the compiler defaults to using the OpenCL C 1.2 language version. To utilize the OpenCL 2.0 Kernel programming language, a programmer must specifically pass the appropriate compiler build option (`-cl-std=CL2.0`). The language version must not be higher than the platform version, but may exceed the device version.
- An OpenCL 2.1 device must support all the requirements of an OpenCL 2.0 device in addition to the SPIR-V intermediate language at version 1.0 or above. Intermediate language versioning is encoded as part of the binary object and no flags are required to be passed to the compiler.
- An OpenCL 2.2 device must support all the requirements of an OpenCL 2.0 device in addition to the SPIR-V intermediate language at version 1.2 or above. Intermediate language versioning is encoded as a part of the binary object and no flags are required to be passed to the compiler.
- OpenCL 3.0 is designed to enable any OpenCL implementation supporting OpenCL 1.2 or newer to easily support and transition to OpenCL 3.0, by making many features in OpenCL 2.0, 2.1, or 2.2 optional. This means that OpenCL 3.0 is backwards compatible with OpenCL 1.2, but is not necessarily backwards compatible with OpenCL 2.0, 2.1, or 2.2.

  An OpenCL 3.0 platform must implement all OpenCL 3.0 APIs, but some APIs may return an error code unconditionally when a feature is not supported by any devices in the platform. Whenever a feature is optional, it will be paired with a query to determine whether the feature is supported. The queries will enable correctly written applications to selectively use all optional features without generating any OpenCL errors, if desired.

  OpenCL 3.0 also adds a new version of the OpenCL C programming language, which makes many features in OpenCL C 2.0 optional. The new version of OpenCL C is backwards compatible with OpenCL C 1.2, but is not backwards compatible with OpenCL C 2.0. The new version of OpenCL C must be explicitly requested via the `-cl-std=` build option, otherwise a program will continue to be compiled using the highest OpenCL C 1.x language version supported for the device.

  Whenever an OpenCL C feature is optional in the new version of the OpenCL C programming language, it will be paired with a feature macro, such as `__opencl_c_<feature_name>`, and a corresponding API query. If a feature macro is defined then the feature is supported by the OpenCL C compiler, otherwise the optional feature is not supported.

- An OpenCL 3.1 device must support all the requirements of an OpenCL 3.0 device in addition to the OpenCL C 3.1 programming language and the SPIR-V 1.0, SPIR-V 1.1, SPIR-V 1.2, SPIR-V 1.3, and SPIR-V 1.4 intermediate languages (ILs).

In order to allow future versions of OpenCL to support new types of devices, minor releases of OpenCL may add new profiles where some features that are currently required for all OpenCL devices become optional. All features that are required for an OpenCL profile will also be required for that profile in subsequent minor releases of OpenCL, thereby guaranteeing backwards compatibility for applications targeting specific profiles. It is therefore strongly recommended that applications query the profile supported by the OpenCL device they are running on in order to remain robust to future changes.

3.4.3. Versioning

The OpenCL specification is regularly updated with bug fixes and clarifications. Occasionally, new functionality is added to the core and extensions. In order to indicate to developers how and when these changes are made to the specification, and to provide a way to identify each set of changes, the OpenCL API, C language, intermediate languages, and extensions maintain a version number. Built-in kernels are also versioned.

3.4.3.1. Version Numbers

A version number comprises three logical fields:

- The *major* version indicates a significant change. Backwards compatibility may break across major versions.
- The *minor* version indicates the addition of new functionality with backwards compatibility for any existing profiles.
- The *patch* version indicates bug fixes, clarifications and general improvements.

Version numbers are represented using the `cl_version` type that is an alias for a 32-bit integer. The fields are packed as follows:

- The *major* version is a 10-bit integer packed into bits 31-22.
- The *minor* version is a 10-bit integer packed into bits 21-12.
- The *patch* version is a 12-bit integer packed into bits 11-0.

This enables versions to be ordered using standard C/C++ operators.

A number of convenience macros are provided by the OpenCL Headers to make working with version numbers easier.

- `CL_VERSION_MAJOR` extracts the *major* version from a packed `cl_version`.
- `CL_VERSION_MINOR` extracts the *minor* version from a packed `cl_version`.
- `CL_VERSION_PATCH` extracts the *patch* version from a packed `cl_version`.
- `CL_MAKE_VERSION` returns a packed `cl_version` from a *major*, *minor* and *patch* version.
- `CL_VERSION_MAJOR_BITS`, `CL_VERSION_MINOR_BITS`, and `CL_VERSION_PATCH_BITS` are the number of bits

in the corresponding field.

- `CL_VERSION_MAJOR_MASK`, `CL_VERSION_MINOR_MASK`, and `CL_VERSION_PATCH_MASK` are bitmasks used to extract the corresponding packed fields from the version number.

```
typedef cl_uint cl_version;

#define CL_VERSION_MAJOR_BITS (10)
#define CL_VERSION_MINOR_BITS (10)
#define CL_VERSION_PATCH_BITS (12)

#define CL_VERSION_MAJOR_MASK ((1 << CL_VERSION_MAJOR_BITS) - 1)
#define CL_VERSION_MINOR_MASK ((1 << CL_VERSION_MINOR_BITS) - 1)
#define CL_VERSION_PATCH_MASK ((1 << CL_VERSION_PATCH_BITS) - 1)

#define CL_VERSION_MAJOR(version) \
  ((version) >> (CL_VERSION_MINOR_BITS + CL_VERSION_PATCH_BITS))

#define CL_VERSION_MINOR(version) \
  (((version) >> CL_VERSION_PATCH_BITS) & CL_VERSION_MINOR_MASK)

#define CL_VERSION_PATCH(version) ((version) & CL_VERSION_PATCH_MASK)

#define CL_MAKE_VERSION(major, minor, patch) \
  ((((major) & CL_VERSION_MAJOR_MASK) << \
        (CL_VERSION_MINOR_BITS + CL_VERSION_PATCH_BITS)) | \
   (((minor) & CL_VERSION_MINOR_MASK) << \
         CL_VERSION_PATCH_BITS) | \
    ((patch) & CL_VERSION_PATCH_MASK))
```

The available version of an extension is exposed to the user via a macro defined by the OpenCL Headers. This macro takes the format of the uppercase extension name followed by the `_EXTENSION_VERSION` suffix. For example, `CL_KHR_SEMAPHORE_EXTENSION_VERSION` is the macro defining the version of the `cl_khr_semaphore` extension.

The value of this macro is set to the `cl_version` of the extension using the semantic version of the extension. If no semantic version is defined for the extension, then the value of the macro is set to `0` to represent semantic version `0.0.0`.

Applications can use these version macros along with the convenience macros defined in this section to guard their code against breaking changes to the API of extensions, in particular experimental KHR extensions which have yet to finalize an API.

3.4.3.2. Version-Name Pairing

The `cl_name_version` structure describes a version number and a corresponding entity (e.g. extension or built-in kernel) name:

```
// Provided by CL_VERSION_3_0
typedef struct cl_name_version {
    cl_version    version;
    char          name[CL_NAME_VERSION_MAX_NAME_SIZE];
} cl_name_version;
```

- *version* is a Version Number.
- *name* is an array of `CL_NAME_VERSION_MAX_NAME_SIZE` characters containing a null-terminated string, whose maximum length is therefore `CL_NAME_VERSION_MAX_NAME_SIZE` minus one.

3.4.4. Valid Usage and Undefined Behavior

The OpenCL specification describes valid usage and how to use the API correctly. For some conditions where an API is used incorrectly, behavior is well-defined, such as returning an error code. For other conditions, behavior is undefined, and may include program termination. However, OpenCL implementations must always ensure that incorrect usage by an application does not affect the integrity of the operating system, the OpenCL implementation, or other OpenCL client applications in the system. In particular, any guarantees made by an operating system about whether memory from one process can be visible to another process or not must not be violated by an OpenCL implementation for any memory allocation. OpenCL implementations are not required to make additional security or integrity guarantees beyond those provided by the operating system unless explicitly directed by the application's use of a particular feature or extension.

For instance, if an operating system guarantees that data in all its memory allocations are set to zero when newly allocated, the OpenCL implementation must make the same guarantees for any allocations it controls.

Similarly, if an operating system guarantees that use-after-free of host allocations will not result in values written by another process becoming visible, the same guarantees must be made by the OpenCL implementation for memory accessible to an OpenCL device.

[1] This value for **memory_scope** can only be used with **atomic_work_item_fence** with flags set to `CLK_IMAGE_MEM_FENCE`.

Chapter 4. The OpenCL Platform Layer

This section describes the OpenCL platform layer which implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

4.1. Querying Platform Info

The list of platforms available can be obtained with the function:

```
// Provided by CL_VERSION_1_0
cl_int clGetPlatformIDs(
    cl_uint num_entries,
    cl_platform_id* platforms,
    cl_uint* num_platforms);
```

- *num_entries* is the number of `cl_platform_id` entries that can be added to *platforms*. If *platforms* is not `NULL`, *num_entries* must be greater than zero.
- *platforms* returns a list of OpenCL platforms found. The `cl_platform_id` values returned in *platforms* can be used to identify a specific OpenCL platform. If *platforms* is `NULL`, this argument is ignored. The number of OpenCL platforms returned is the minimum of the value specified by *num_entries* or the number of OpenCL platforms available.
- *num_platforms* returns the number of OpenCL platforms available. If *num_platforms* is `NULL`, this argument is ignored.

clGetPlatformIDs returns `CL_SUCCESS` if the function is executed and, if the `cl_khr_icd` extension is supported, there are a non-zero number of platforms available. Otherwise, it returns one of the following errors:

- `CL_PLATFORM_NOT_FOUND_KHR`
  - if the `cl_khr_icd` extension is supported and zero platforms are available
- `CL_INVALID_VALUE`
  - if *num_entries* is equal to zero and *platforms* is not `NULL`
  - if both *num_platforms* and *platforms* are `NULL`
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To obtain the list of platforms accessible through the Khronos ICD Loader, call the function:

```
// Provided by cl_khr_icd
cl_int clIcdGetPlatformIDsKHR(
    cl_uint num_entries,
    cl_platform_id* platforms,
    cl_uint* num_platforms);
```

clIcdGetPlatformIDsKHR is provided by the `cl_khr_icd` extension.

- *num_entries* is the number of `cl_platform_id` entries that can be added to *platforms*. If *platforms* is not `NULL`, then *num_entries* must be greater than zero.
- *platforms* returns a list of OpenCL platforms available for access through the Khronos ICD Loader. The `cl_platform_id` values returned in *platforms* are ICD compatible and can be used to identify a specific OpenCL platform. If the *platforms* argument is `NULL`, then this argument is ignored. The number of OpenCL platforms returned is the minimum of the value specified by *num_entries* or the number of OpenCL platforms available.
- *num_platforms* returns the number of OpenCL platforms available. If *num_platforms* is `NULL`, then this argument is ignored.

clIcdGetPlatformIDsKHR returns `CL_SUCCESS` if the function is executed successfully and there are a non-zero number of platforms available. Otherwise, it returns one of the following errors:

- `CL_PLATFORM_NOT_FOUND_KHR`
  - if zero platforms are available
- `CL_INVALID_VALUE`
  - if *num_entries* is equal to zero and *platforms* is not `NULL`
  - if both *num_platforms* and *platforms* are `NULL`
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The entry point of a given platform that is ICD 2 compatible can be obtained using the following function:

```
// Provided by cl_khr_icd
void* clIcdGetFunctionAddressForPlatformKHR(
    cl_platform_id platform,
    const char* func_name);
```

clIcdGetFunctionAddressForPlatformKHR is provided by the `cl_khr_icd` extension.

- *platform* refers to the platform ID returned by **clIcdGetPlatformIDsKHR**
- *func_name* name of an API entry point

clIcdGetFunctionAddressForPlatformKHR returns the address of the API entry point named by *func_name*. The pointer returned should be cast to a function pointer type matching the API entry point as defined in the specification header file.

A return value of `NULL` indicates that the specified function does not exist for *platform*, or *platform* is not a valid platform for the implementation.

The `dispatch_data` field of a platform can be set using the following function:

```
// Provided by cl_khr_icd
cl_int clIcdSetPlatformDispatchDataKHR(
    cl_platform_id platform,
    void* dispatch_data);
```

clIcdSetPlatformDispatchDataKHR is provided by the `cl_khr_icd` extension.

- *platform* refers to the platform ID returned by **clIcdGetPlatformIDsKHR**
- *dispatch_data* is the value to set the `dispatch_data` field of the structure to

clIcdSetPlatformDispatchDataKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PLATFORM`
  - if *platform* is not a valid platform
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Specific information about an OpenCL platform can be obtained with the function:

```
// Provided by CL_VERSION_1_0
cl_int clGetPlatformInfo(
    cl_platform_id platform,
    cl_platform_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *platform* refers to the platform ID returned by **clGetPlatformIDs** or can be `NULL`. If *platform* is `NULL`, the behavior is implementation-defined.
- *param_name* is an enumeration constant that identifies the platform information being queried. It can be one of the following values as specified in the Platform Queries table.
- *param_value* is a pointer to memory location where appropriate values for a given *param_name*, as specified in the Platform Queries table, will be returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size

must be greater than or equal to the size of the return type specified in the Platform Queries table. If *param_value* is `NULL`, it is ignored.

- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

The information that can be queried using **clGetPlatformInfo** is specified in the Platform Queries table.

Table 3. List of supported param_names by **clGetPlatformInfo**

Platform Info	Return Type	Description
`CL_PLATFORM_PROFILE` [1]	`char`[] [2]	OpenCL profile string. Returns the profile name supported by the implementation. The profile name returned can be one of the following strings: FULL_PROFILE - if the implementation supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported). EMBEDDED_PROFILE - if the implementation supports the OpenCL embedded profile. The embedded profile is defined to be a subset for each version of OpenCL. The embedded profile for OpenCL is described in OpenCL Embedded Profile.
`CL_PLATFORM_VERSION`	`char`[]	OpenCL version string. Returns the OpenCL version supported by the implementation. This version string has the following format: *OpenCL<space><major_version.minor_version><space><platform-specific information>* The *major_version.minor_version* value returned will be one of 1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 3.0 or 3.1.

Platform Info	Return Type	Description
CL_PLATFORM_NUMERIC_VERSION missing before version 3.0. or CL_PLATFORM_NUMERIC_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_version or cl_version_khr	Returns the detailed (major, minor, patch) version supported by the platform. The major and minor version numbers returned must match those returned via CL_PLATFORM_VERSION.
CL_PLATFORM_NAME	char[]	Platform name string.
CL_PLATFORM_VENDOR	char[]	Platform vendor string.
CL_PLATFORM_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the platform. Each extension that is supported by all devices associated with this platform must be reported here.
CL_PLATFORM_EXTENSIONS_WITH_VERSION missing before version 3.0. or CL_PLATFORM_EXTENSIONS_WITH_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_name_version[] or cl_name_version_khr[]	Returns an array of description (name and version) structures that lists all the extensions supported by the platform. The same extension name must not be reported more than once. The list of extensions reported must match the list reported via CL_PLATFORM_EXTENSIONS.
CL_PLATFORM_HOST_TIMER_RESOLUTION missing before version 2.1.	cl_ulong	Returns the resolution of the host timer in nanoseconds as used by **clGetDeviceAndHostTimer**. Support for device and host timer synchronization is required for platforms supporting OpenCL 2.1 or 2.2. This value must be 0 for platforms that do not support device and host timer synchronization.

Platform Info	Return Type	Description
CL_PLATFORM_COMMAND_BUFFER_CAPABILITIES_KHR provided by the cl_khr_command_buffer_multi_device extension.	cl_platform_command_buffer_capabilities_khr	Describes platform command-buffer capabilities, encoded as bits in a bitfield. Supported capabilities are: CL_COMMAND_BUFFER_PLATFORM_UNIVERSAL_SYNC_KHR - Platform supports the ability to synchronize all commands in a command-buffer using sync-points, irrespective of the queue the individual commands are recorded to. provided by the cl_khr_command_buffer_multi_device extension. CL_COMMAND_BUFFER_PLATFORM_REMAP_QUEUES_KHR - Platform supports the ability to create a deep copy of an existing command-buffer with the commands explicitly remapped to different queues. provided by the cl_khr_command_buffer_multi_device extension. CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR - Platform supports the ability to create a remapped command-buffer where the mapping of commands to queues is done by the OpenCL runtime in a way it determines as optimal. If CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR is reported, CL_COMMAND_BUFFER_PLATFORM_REMAP_QUEUES_KHR must also be reported. provided by the cl_khr_command_buffer_multi_device extension.
CL_PLATFORM_EXTERNAL_MEMORY_IMPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_memory extension.	cl_external_memory_handle_type_khr[]	Returns the list of importable external memory handle types supported by all devices in *platform*.
CL_PLATFORM_SEMAPHORE_TYPES_KHR provided by the cl_khr_semaphore extension.	cl_semaphore_type_khr[]	Returns the list of the semaphore types supported all devices in *platform*.

Platform Info	Return Type	Description
CL_PLATFORM_SEMAPHORE_IMPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_semaphore extension.	cl_external_semaphore_handle_type_khr[]	Returns the list of importable external semaphore handle types supported by all devices in *platform*. The size of this query may be 0 if no importable external semaphore handle types are supported by all devices in *platform*.
CL_PLATFORM_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_semaphore extension.	cl_external_semaphore_handle_type_khr[]	Returns the list of exportable external semaphore handle types supported by all devices in the platform. This size of this query may be 0 if no exportable external semaphore handle types are supported by all devices in *platform*.
CL_PLATFORM_ICD_SUFFIX_KHR provided by the cl_khr_icd extension.	char[]	The function name suffix used to identify extension functions to be directed to this platform by the ICD Loader.
CL_PLATFORM_UNLOADABLE_KHR provided by the cl_khr_icd_unloadable extension.	cl_bool	The platform can be unloaded by the ICD Loader.

clGetPlatformInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors [3].

- CL_INVALID_PLATFORM
  - if *platform* is not a valid platform
- CL_INVALID_VALUE
  - if *param_name* is not one of the supported values
  - if the size in bytes specified by *param_value_size* is less than the size of the return type specified in the Platform Queries table and *param_value* is not NULL
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.2. Querying Devices

The list of devices available on a platform can be obtained using the function [4]:

```
// Provided by CL_VERSION_1_0
cl_int clGetDeviceIDs(
    cl_platform_id platform,
    cl_device_type device_type,
    cl_uint num_entries,
    cl_device_id* devices,
    cl_uint* num_devices);
```

- *platform* refers to the platform ID returned by **clGetPlatformIDs** or can be `NULL`. If *platform* is `NULL`, the behavior is implementation-defined.
- *device_type* is a bitfield that identifies the type of OpenCL device. The *device_type* can be used to query specific OpenCL devices or all OpenCL devices available. The valid values for *device_type* are specified in the Device Types table.
- *num_entries* is the number of `cl_device_id` entries that can be added to *devices*. If *devices* is not `NULL`, the *num_entries* must be greater than zero.
- *devices* returns a list of OpenCL devices found. The `cl_device_id` values returned in *devices* can be used to identify a specific OpenCL device. If *devices* is `NULL`, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* or the number of OpenCL devices whose type matches *device_type*.
- *num_devices* returns the number of OpenCL devices available that match *device_type*. If *num_devices* is `NULL`, this argument is ignored.

Table 4. List of supported device_types by **clGetDeviceIDs**

Device Type	Description
`CL_DEVICE_TYPE_CPU`	An OpenCL device similar to a traditional CPU (Central Processing Unit). The host processor that executes OpenCL host code may also be considered a CPU OpenCL device.
`CL_DEVICE_TYPE_GPU`	An OpenCL device similar to a GPU (Graphics Processing Unit). Many systems include a dedicated processor for graphics or rendering that may be considered a GPU OpenCL device.
`CL_DEVICE_TYPE_ACCELERATOR`	Dedicated devices that may accelerate OpenCL programs, such as FPGAs (Field Programmable Gate Arrays), DSPs (Digital Signal Processors), or AI (Artificial Intelligence) processors.

Device Type	Description
CL_DEVICE_TYPE_CUSTOM missing before version 1.2.	Specialized devices that support a subset of the OpenCL runtime APIs for directed tasks but are not OpenCL conformant. A custom device must implement all of the OpenCL runtime APIs, but may not support all of the required minimum device capabilities and may return implementation-defined error codes for unsupported functionality.
CL_DEVICE_TYPE_DEFAULT	The default OpenCL device in the platform. One device in the platform must be returned as the CL_DEVICE_TYPE_DEFAULT device when passed as the *device_type* to **clGetDeviceIDs**. CL_DEVICE_TYPE_DEFAULT is only used to query OpenCL devices using **clGetDeviceIDs** or to create OpenCL contexts using **clCreateContextFromType**, and will never be returned in CL_DEVICE_TYPE for any OpenCL device. The default OpenCL device must not be a CL_DEVICE_TYPE_CUSTOM device unless it is the only device in the platform.
CL_DEVICE_TYPE_ALL	All OpenCL devices in the platform. CL_DEVICE_TYPE_ALL is only used to query OpenCL devices using **clGetDeviceIDs** or to create OpenCL contexts using **clCreateContextFromType**, and will never be returned in CL_DEVICE_TYPE for any OpenCL device.

clGetDeviceIDs returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PLATFORM
  - if *platform* is not a valid platform
- CL_INVALID_DEVICE_TYPE
  - if *device_type* is not a valid value
- CL_INVALID_VALUE
  - if *num_entries* is equal to zero and *devices* is not NULL
  - if both *num_devices* and *devices* are NULL
- CL_DEVICE_NOT_FOUND
  - if no OpenCL devices that matched *device_type* were found
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device

- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The application can query specific capabilities of the OpenCL device(s) returned by **clGetDeviceIDs**. This can be used by the application to determine which device(s) to use.

To get specific information about an OpenCL device, call the function:

```
// Provided by CL_VERSION_1_0
cl_int clGetDeviceInfo(
    cl_device_id device,
    cl_device_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *device* may be a device returned by **clGetDeviceIDs** or a sub-device created by **clCreateSubDevices**. If *device* is a sub-device, the specific information for the sub-device will be returned. The information that can be queried using **clGetDeviceInfo** is specified in the Device Queries table.
- *param_name* is an enumeration constant that identifies the device information being queried. It can be one of the following values as specified in the Device Queries table.
- *param_value* is a pointer to memory location where appropriate values for a given *param_name*, as specified in the Device Queries table, will be returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Device Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

The device queries described in the Device Queries table should return the same information for a root-level device i.e. a device returned by **clGetDeviceIDs** and any sub-devices created from this device except for the following queries:

- `CL_DEVICE_GLOBAL_MEM_CACHE_SIZE`
- `CL_DEVICE_BUILT_IN_KERNELS`
- `CL_DEVICE_PARENT_DEVICE`
- `CL_DEVICE_PARTITION_TYPE`
- `CL_DEVICE_REFERENCE_COUNT`

Table 5. List of supported param_names by **clGetDeviceInfo**

Device Info	Return Type	Description
CL_DEVICE_TYPE	cl_device_type	The type of the OpenCL device. The device type is purely informational and has no semantic meaning. The device must report a single device type, which must not be CL_DEVICE_TYPE_DEFAULT or CL_DEVICE_TYPE_ALL. Please see the Device Types table for supported device types and device type descriptions.
CL_DEVICE_VENDOR_ID [5]	cl_uint	A unique device vendor identifier. If the vendor has a PCI vendor ID, the low 16 bits must contain that PCI vendor ID, and the remaining bits must be set to zero. Otherwise, the value returned must be a valid Khronos vendor ID represented by type cl_khronos_vendor_id. Khronos vendor IDs are allocated starting at 0x10000, to distinguish them from the PCI vendor ID namespace.
CL_DEVICE_MAX_COMPUTE_UNITS	cl_uint	The number of parallel compute units on the OpenCL device. A work-group executes on a single compute unit. The minimum value is 1.
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS	cl_uint	Maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. (Refer to **clEnqueueNDRangeKernel**). The minimum value is 3.
CL_DEVICE_MAX_WORK_GROUP_SIZES missing before version 3.1.	size_t[]	The maximum number of work-items that can be specified in each dimension of a work-group to **clEnqueueNDRangeKernel**. Returns *n* size_t values, where *n* is the value returned by the query for CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS. The minimum value is (1, 1, 1).
CL_DEVICE_MAX_WORK_ITEM_SIZES deprecated by version 3.1.	size_t[]	See description of CL_DEVICE_MAX_WORK_GROUP_SIZES.

Device Info	Return Type	Description
CL_DEVICE_MAX_WORK_GROUP_SIZE	size_t	The maximum total number of work-items that can be specified for a work-group to **clEnqueueNDRangeKernel**. The returned value is an upper limit and will not necessarily maximize performance. Additionally, this maximum for the device may be larger than the maximum supported for a specific kernel, refer to the per-kernel query CL_KERNEL_WORK_GROUP_SIZE. The minimum value is 1.
CL_DEVICE_PREFERRED_VECTOR_WIDTH_CHAR CL_DEVICE_PREFERRED_VECTOR_WIDTH_SHORT CL_DEVICE_PREFERRED_VECTOR_WIDTH_INT CL_DEVICE_PREFERRED_VECTOR_WIDTH_LONG CL_DEVICE_PREFERRED_VECTOR_WIDTH_FLOAT CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF is missing before version 1.1.	cl_uint	Preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_DOUBLE must return 0. If the cl_khr_fp16 extension is not supported, CL_DEVICE_PREFERRED_VECTOR_WIDTH_HALF must return 0.
CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT CL_DEVICE_NATIVE_VECTOR_WIDTH_INT CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF missing before version 1.1.	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. If double precision is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE must return 0. If the cl_khr_fp16 extension is not supported, CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF must return 0.

Device Info	Return Type	Description
CL_DEVICE_MAX_CLOCK_FREQUENCY	cl_uint	Clock frequency of the device in MHz. The meaning of this value is implementation-defined. For devices with multiple clock domains, the clock frequency for any of the clock domains may be returned. For devices that dynamically change frequency for power or thermal reasons, the returned clock frequency may be any valid frequency. Note: This definition is missing before version 2.2. Maximum configured clock frequency of the device in MHz. Note: This definition is deprecated by version 2.2.
CL_DEVICE_ADDRESS_BITS	cl_uint	The default compute device address space size of the global address space specified as an unsigned integer value in bits. Currently supported values are 32 or 64 bits.
CL_DEVICE_MAX_MEM_ALLOC_SIZE	cl_ulong	Max size of memory object allocation in bytes. The minimum value is max(min(1024 × 1024 × 1024, 1/4th of CL_DEVICE_GLOBAL_MEM_SIZE), 32 × 1024 × 1024).
CL_DEVICE_IMAGE_SUPPORT	cl_bool	Is CL_TRUE if images are supported by the OpenCL device and CL_FALSE otherwise.
CL_DEVICE_MAX_READ_IMAGE_ARGS [6]	cl_uint	Max number of image objects arguments of a kernel declared with the read_only qualifier. The minimum value is 128 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only qualifier. The minimum value is 64 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS missing before version 2.0.	cl_uint	Max number of image objects arguments of a kernel declared with the write_only or read_write qualifier. Support for read-write image arguments is required for an OpenCL 2.0, 2.1, or 2.2 device if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE. The minimum value is 64 if the device supports read-write images arguments, and must be 0 for devices that do not support read-write images.

Device Info	Return Type	Description
CL_DEVICE_IL_VERSION missing before version 2.1. or CL_DEVICE_IL_VERSION_KHR provided by the cl_khr_il_program extension.	char[]	The intermediate languages that are supported by **clCreateProgramWithIL** for this device. Returns a space-separated list of intermediate language version strings of the form: <IL_Prefix>_<Major_Version>.<Minor_Version> An OpenCL 2.1 device or a device that supports the cl_khr_il_program extension must support at least SPIR-V 1.0. An OpenCL 2.2 device must support at least SPIR-V 1.0, SPIR-V 1.1, and SPIR-V 1.2. An OpenCL 3.1 device must support at least SPIR-V 1.0, SPIR-V 1.1, SPIR-V 1.2, SPIR-V 1.3, and SPIR-V 1.4. If the device does not support intermediate language programs, the returned value must be "" (an empty string).
CL_DEVICE_ILS_WITH_VERSION missing before version 3.0. or CL_DEVICE_ILS_WITH_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_name_version[] or cl_name_version_khr[]	Returns an array of descriptions (name and version) for all supported intermediate languages. Intermediate languages with the same name may be reported more than once but each name and major/minor version combination may only be reported once. The list of intermediate languages reported must match the list reported via CL_DEVICE_IL_VERSION.
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	Max width of 2D image or 1D image not created from a buffer object in pixels. The minimum value is 16384 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	Max height of 2D image in pixels. The minimum value is 16384 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.

Device Info	Return Type	Description
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	Max width of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	Max height of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	Max depth of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE_MAX_BUFFER_SIZE missing before version 1.2.	size_t	Max number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE_MAX_ARRAY_SIZE missing before version 1.2.	size_t	Max number of images in a 1D or 2D image array. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_SAMPLERS	cl_uint	Maximum number of samplers that can be used in a kernel. The minimum value is 16 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.

Device Info	Return Type	Description
CL_DEVICE_IMAGE_PITCH_ALIGNMENT missing before version 2.0. The equivalent CL_DEVICE_IMAGE_PITCH_ALIGNMENT_KHR may be used if the cl_khr_image2d_from_buffer extension is supported.	cl_uint	The row pitch alignment size in pixels for 2D images created from a buffer. The value returned must be a power of 2. Support for 2D images created from a buffer is required for an OpenCL 2.0, 2.1, or 2.2 device if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE. This value must be 0 for devices that do not support 2D images created from a buffer. If the cl_khr_image2d_from_buffer and cl_ext_image_requirements_info extensions are supported, the value returned by CL_DEVICE_IMAGE_PITCH_ALIGNMENT after converting in bytes for the supported format with the biggest element size (channel data type size × number of channels) must be greater than or equal to the value returned by CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT for any supported format.
CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT missing before version 2.0. The equivalent CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT_KHR may be used if the cl_khr_image2d_from_buffer extension is supported.	cl_uint	This query specifies the minimum alignment in pixels of the host_ptr specified to **clCreateBuffer** or **clCreateBufferWithProperties** when a 2D image is created from a buffer which was created using CL_MEM_USE_HOST_PTR. The value returned must be a power of 2. Support for 2D images created from a buffer is required for an OpenCL 2.0, 2.1, or 2.2 device if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE. This value must be 0 for devices that do not support 2D images created from a buffer. If the cl_khr_image2d_from_buffer and cl_ext_image_requirements_info extensions are supported, the value returned by CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT after converting in bytes for the supported format with the biggest element size (channel data type size × number of channels) must be greater than or equal to the value returned by CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT for any supported format.
CL_DEVICE_MAX_PIPE_ARGS missing before version 2.0.	cl_uint	The maximum number of pipe objects that can be passed as arguments to a kernel. The minimum value is 16 for devices supporting pipes, and must be 0 for devices that do not support pipes.

Device Info	Return Type	Description
`CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS` missing before version 2.0.	`cl_uint`	The maximum number of reservations that can be active for a pipe per work-item in a kernel. A work-group reservation is counted as one reservation per work-item. The minimum value is 1 for devices supporting pipes, and must be 0 for devices that do not support pipes.
`CL_DEVICE_PIPE_MAX_PACKET_SIZE` missing before version 2.0.	`cl_uint`	The maximum size of pipe packet in bytes. Support for pipes is required for an OpenCL 2.0, 2.1, or 2.2 device. The minimum value is 1024 bytes if the device supports pipes, and must be 0 for devices that do not support pipes.
`CL_DEVICE_MAX_PARAMETER_SIZE`	`size_t`	Max size in bytes of all arguments that can be passed to a kernel. The minimum value is 1024. For this minimum value, only a maximum of 128 arguments can be passed to a kernel. For all other values, a maximum of 255 arguments can be passed to a kernel.
`CL_DEVICE_MEM_BASE_ADDR_ALIGN`	`cl_uint`	Alignment requirement (in bits) for sub-buffer offsets. The minimum value is the size (in bits) of the largest OpenCL built-in data type supported by the device (long16 in FULL profile, long16 or int16 in EMBEDDED profile).
`CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE` deprecated by version 1.2.	`cl_uint`	The minimum value is the size (in bytes) of the largest OpenCL data type supported by the device (`long16` in FULL profile, `long16` or `int16` in EMBEDDED profile).

Device Info	Return Type	Description
CL_DEVICE_SINGLE_FP_CONFIG [7]	cl_device_fp_config	Describes single precision floating-point capability of the device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM - denorms are supported CL_FP_INF_NAN - INF and quiet NaNs are supported CL_FP_ROUND_TO_NEAREST-- round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO - round to zero rounding mode supported CL_FP_ROUND_TO_INF - round to positive and negative infinity rounding modes supported CL_FP_FMA - the fma kernel builtin is implemented using hardware-accelerated fused multiply-add as defined in IEEE754-2008. CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT - divide and sqrt are correctly rounded as defined by the IEEE754 specification CL_FP_SOFT_FLOAT - Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software For the full profile, the mandated minimum floating-point capability is: CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN. For the embedded profile, see the dedicated table.

Device Info	Return Type	Description
CL_DEVICE_DOUBLE_FP_CONFIG [7] missing before version 1.2. Also see cl_khr_fp64.	cl_device_fp_config	Describes double precision floating-point capability of the OpenCL device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM - denorms are supported CL_FP_INF_NAN - INF and NaNs are supported CL_FP_ROUND_TO_NEAREST - round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO - round to zero rounding mode supported CL_FP_ROUND_TO_INF - round to positive and negative infinity rounding modes supported CL_FP_FMA - the fma kernel builtin is implemented using hardware-accelerated fused multiply-add as defined in IEEE754-2008. CL_FP_SOFT_FLOAT - Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software Double precision is an optional feature so the mandated minimum double precision floating-point capability is 0. If double precision is supported by the device, then the minimum double precision floating-point capability for OpenCL 2.0 or newer devices is: CL_FP_FMA | CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN | CL_FP_DENORM. or for OpenCL 1.0, OpenCL 1.1 or OpenCL 1.2 devices: CL_FP_FMA | CL_FP_ROUND_TO_NEAREST | CL_FP_ROUND_TO_ZERO | CL_FP_ROUND_TO_INF | CL_FP_INF_NAN | CL_FP_DENORM.
CL_DEVICE_GLOBAL_MEM_CACHE_TYPE	cl_device_mem_cache_type	Type of global memory cache supported. Valid values are: CL_NONE, CL_READ_ONLY_CACHE, and CL_READ_WRITE_CACHE.
CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE	cl_uint	Size of global memory cache line in bytes.

Device Info	Return Type	Description
CL_DEVICE_GLOBAL_MEM_CACHE_SIZE	cl_ulong	Size of global memory cache in bytes.
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Size of global device memory in bytes.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong	Max size in bytes of a constant buffer allocation. The minimum value is 64 KB.
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 8.
CL_DEVICE_MAX_GLOBAL_VARIABLE_SIZE missing before version 2.0.	size_t	The maximum number of bytes of storage that may be allocated for any single variable in program scope or inside a function in an OpenCL kernel language declared in the global address space. Support for program scope global variables is required for an OpenCL 2.0, 2.1, or 2.2 device. The minimum value is 64 KB if the device supports program scope global variables, and must be 0 for devices that do not support program scope global variables.
CL_DEVICE_GLOBAL_VARIABLE_PREFERRED_TOTAL_SIZE missing before version 2.0.	size_t	Maximum preferred total size, in bytes, of all program variables in the global address space. This is a performance hint. An implementation may place such variables in storage with optimized device access. This query returns the capacity of such storage. The minimum value is 0.
CL_DEVICE_LOCAL_MEM_TYPE	cl_device_local_mem_type	Type of local memory supported. This can be set to CL_LOCAL implying dedicated local memory storage such as SRAM, or CL_GLOBAL. For custom devices, CL_NONE can also be returned indicating no local memory support.
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory region in bytes. The minimum value is 32 KB.
CL_DEVICE_ERROR_CORRECTION_SUPPORT	cl_bool	Is CL_TRUE if the device implements error correction for all accesses to compute device memory (global and constant). Is CL_FALSE if the device does not implement such error correction.
CL_DEVICE_HOST_UNIFIED_MEMORY missing before version 1.1.	cl_bool	Is CL_TRUE if the device and the host have a unified memory subsystem and is CL_FALSE otherwise. [8]

Device Info	Return Type	Description
CL_DEVICE_PROFILING_TIMER_RESOLUTION	size_t	Describes the resolution of device timer. This is measured in nanoseconds. Refer to Profiling Operations for details.
CL_DEVICE_ENDIAN_LITTLE	cl_bool	Is CL_TRUE if the OpenCL device is a little endian device and CL_FALSE otherwise.
CL_DEVICE_AVAILABLE	cl_bool	Is CL_TRUE if the device is available and CL_FALSE otherwise. A device is considered to be available if the device can be expected to successfully execute commands enqueued to the device.
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Is CL_TRUE if a compiler is available to compile programs created from source or IL, or CL_FALSE if the implementation does not have a compiler available. This must be CL_TRUE for devices supporting the full profile. When an online compiler is not available, OpenCL programs may still be created from binaries or built-in kernels supported by the device.
CL_DEVICE_LINKER_AVAILABLE missing before version 1.2.	cl_bool	Is CL_TRUE if a linker is available to link compiled programs, or CL_FALSE if the implementation does not have a linker available. This must be CL_TRUE for devices supporting the full profile or if CL_DEVICE_COMPILER_AVAILABLE is CL_TRUE.
CL_DEVICE_EXECUTION_CAPABILITIES	cl_device_exec_capabilities	Describes the execution capabilities of the device. This is a bit-field that describes one or more of the following values: CL_EXEC_KERNEL - The OpenCL device can execute OpenCL kernels. CL_EXEC_NATIVE_KERNEL - The OpenCL device can execute native kernels. The mandated minimum capability is: CL_EXEC_KERNEL.
CL_DEVICE_QUEUE_PROPERTIES deprecated by version 2.0.	cl_command_queue_properties	See description of CL_DEVICE_QUEUE_ON_HOST_PROPERTIES.

Device Info	Return Type	Description

`CL_DEVICE_QUEUE_ON_HOST_PROPERTIES`
 	
missing before version 2.0.
	`cl_command_queue_properties`	Describes the on host command-queue properties supported by the device. This is a bit-field that describes one or more of the following values:
		
		`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`
		`CL_QUEUE_PROFILING_ENABLE`
		
		These properties are described in the Queue Properties table.
		
		The mandated minimum capability is: `CL_QUEUE_PROFILING_ENABLE`.

`CL_DEVICE_QUEUE_ON_DEVICE_PROPERTIES`
 	
missing before version 2.0.
	`cl_command_queue_properties`	Describes the on device command-queue properties supported by the device. This is a bit-field that describes one or more of the following values:
		
		`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`
		`CL_QUEUE_PROFILING_ENABLE`
		
		These properties are described in the Queue Properties table.
		
		Support for on-device queues is required for an OpenCL 2.0, 2.1, or 2.2 device. When on-device queues are supported, the mandated minimum capability is:
		
		`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` | `CL_QUEUE_PROFILING_ENABLE`.
		
		Must be 0 for devices that do not support on-device queues.

`CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE` missing before version 2.0.	`cl_uint`	The preferred size of the device queue, in bytes. Applications should use this size for the device queue to ensure good performance. The minimum value is 16 KB for devices supporting on-device queues, and must be 0 for devices that do not support on-device queues.
`CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE` missing before version 2.0.	`cl_uint`	The maximum size of the device queue in bytes. The minimum value is 256 KB for the full profile and 64 KB for the embedded profile for devices supporting on-device queues, and must be 0 for devices that do not support on-device queues.

Device Info	Return Type	Description
CL_DEVICE_MAX_ON_DEVICE_QUEUES missing before version 2.0.	cl_uint	The maximum number of device queues that can be created for this device in a single context. The minimum value is 1 for devices supporting on-device queues, and must be 0 for devices that do not support on-device queues.
CL_DEVICE_MAX_ON_DEVICE_EVENTS missing before version 2.0.	cl_uint	The maximum number of events in use by a device queue. These refer to events returned by the enqueue_ built-in functions to a device queue or user events returned by the create_user_event built-in function that have not been released. The minimum value is 1024 for devices supporting on-device queues, and must be 0 for devices that do not support on-device queues.
CL_DEVICE_BUILT_IN_KERNELS missing before version 1.2.	char[]	A semi-colon separated list of built-in kernels supported by the device. An empty string is returned if no built-in kernels are supported by the device.
CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION missing before version 3.0. or CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_name_version[] or cl_name_version_khr[]	Returns an array of descriptions for the built-in kernels supported by the device. Each built-in kernel may only be reported once. The list of reported kernels must match the list returned via CL_DEVICE_BUILT_IN_KERNELS.
CL_DEVICE_PLATFORM	cl_platform_id	The platform associated with this device.
CL_DEVICE_NAME	char[]	Device name string.
CL_DEVICE_VENDOR	char[]	Vendor name string.
CL_DRIVER_VERSION	char[]	OpenCL software driver version string. Follows a vendor-specific format.

Device Info	Return Type	Description
CL_DEVICE_PROFILE	char[]	OpenCL profile string. Returns the profile name supported by the device. The profile name returned can be one of the following strings: FULL_PROFILE - if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported). EMBEDDED_PROFILE - if the device supports the OpenCL embedded profile.
CL_DEVICE_VERSION	char[]	OpenCL version string. Returns the OpenCL version supported by the device. This version string has the following format: *OpenCL<space><major_version.minor_version><space><vendor-specific information>* The *major_version.minor_version* value returned will be one of 1.0, 1.1, 1.2, 2.0, 2.1, 2.2, 3.0 or 3.1.
CL_DEVICE_NUMERIC_VERSION missing before version 3.0. or CL_DEVICE_NUMERIC_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_version or cl_version_khr	Returns the detailed (major, minor, patch) version supported by the device. The major and minor version numbers returned must match those returned via CL_DEVICE_VERSION.

Device Info	Return Type	Description
`CL_DEVICE_OPENCL_C_VERSION` missing before version 1.1 and deprecated by version 3.0.	`char`[]	Returns the highest fully backwards compatible OpenCL C version supported by the compiler for the device. For devices supporting compilation from OpenCL C source, this will return a version string with the following format: *OpenCL<space>C<space><major_version.minor_version><space><vendor-specific information>* For devices that support compilation from OpenCL C source: Because OpenCL 3.0 and newer is backwards compatible with OpenCL 1.2, an OpenCL 3.0 or newer device must support at least OpenCL C 1.2. An OpenCL 3.0 or newer device may return an OpenCL C version newer than OpenCL C 1.2 if and only if all optional OpenCL C features are supported by the device for the newer version. Support for OpenCL C 2.0 is required for an OpenCL 2.0, OpenCL 2.1, or OpenCL 2.2 device. Support for OpenCL C 1.2 is required for an OpenCL 1.2 device. Support for OpenCL C 1.1 is required for an OpenCL 1.1 device. Support for either OpenCL C 1.0 or OpenCL C 1.1 is required for an OpenCL 1.0 device. For devices that do not support compilation from OpenCL C source, such as when `CL_DEVICE_COMPILER_AVAILABLE` is `CL_FALSE`, this query may return an empty string. This query has been superseded by the `CL_DEVICE_OPENCL_C_ALL_VERSIONS` query, which returns a set of OpenCL C versions supported by a device.

Device Info	Return Type	Description
`CL_DEVICE_OPENCL_C_ALL_VERSIONS` missing before version 3.0.	`cl_name_version`[]	Returns an array of name, version descriptions listing all the versions of OpenCL C supported by the compiler for the device. In each returned description structure, the name field is required to be "OpenCL C". The list may include both newer non-backwards compatible OpenCL C versions, such as OpenCL C 3.0, and older OpenCL C versions with mandatory backwards compatibility. The version returned by `CL_DEVICE_OPENCL_C_VERSION` is required to be present in the list. For devices that support compilation from OpenCL C source: Because OpenCL 3.0 is backwards compatible with OpenCL 1.2, and OpenCL 1.2 is backwards compatible with OpenCL 1.1 and OpenCL 1.0, support for at least OpenCL C 3.0, OpenCL C 1.2, OpenCL C 1.1, and OpenCL C 1.0 is required for an OpenCL 3.0 device. Support for at least OpenCL C 3.1, OpenCL C 3.0, OpenCL C 1.2, OpenCL C 1.1, and OpenCL C 1.0 is required for an OpenCL 3.1 device. Support for OpenCL C 2.0, OpenCL C 1.2, OpenCL C 1.1, and OpenCL C 1.0 is required for an OpenCL 2.0, OpenCL 2.1, or OpenCL 2.2 device. Support for OpenCL C 1.2, OpenCL C 1.1, and OpenCL C 1.0 is required for an OpenCL 1.2 device. Support for OpenCL C 1.1 and OpenCL C 1.0 is required for an OpenCL 1.1 device. Support for at least OpenCL C 1.0 is required for an OpenCL 1.0 device. For devices that do not support compilation from OpenCL C source, this query may return an empty array.

Device Info	Return Type	Description
CL_DEVICE_OPENCL_C_NUMERIC_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_version_khr	Returns detailed (major, minor, patch) numeric version information. The major and minor version numbers returned must match those returned via CL_DEVICE_OPENCL_C_VERSION. This query was not promoted to core in OpenCL version 3.0, but the core query CL_DEVICE_OPENCL_C_ALL_VERSIONS can be used to obtain equivalent information.
CL_DEVICE_OPENCL_C_FEATURES missing before version 3.0.	cl_name_version[]	Returns an array of optional OpenCL C features supported by the compiler for the device alongside the OpenCL C version that introduced the feature macro. For example, if a compiler supports an OpenCL C 3.0 feature, the returned name will be the full name of the OpenCL C feature macro, and the returned version will be 3.0.0. For devices that do not support compilation from OpenCL C source, this query may return an empty array.
CL_DEVICE_EXTENSIONS	char[]	Returns a space separated list of extension names (the extension names themselves do not contain any spaces) supported by the device. The list of extension names may include Khronos approved extension names and vendor specified extension names. Please refer to Required OpenCL Extensions for a list of OpenCL extensions that must be supported for each OpenCL version. Please refer to the OpenCL Specification or vendor-provided documentation for a detailed description of these extensions.
CL_DEVICE_EXTENSIONS_WITH_VERSION missing before version 3.0. or CL_DEVICE_EXTENSIONS_WITH_VERSION_KHR provided by the cl_khr_extended_versioning extension.	cl_name_version[] or cl_name_version_khr[]	Returns an array of description (name and version) structures. The same extension name must not be reported more than once. The list of extensions reported must match the list reported via CL_DEVICE_EXTENSIONS. See CL_DEVICE_EXTENSIONS for a list of extensions that are required to be reported for a given OpenCL version.

Device Info	Return Type	Description
CL_DEVICE_PRINTF_BUFFER_SIZE missing before version 1.2.	size_t	Maximum size in bytes of the internal buffer that holds the output of printf calls from a kernel. The minimum value for the FULL profile is 1 MB.
CL_DEVICE_PREFERRED_INTEROP_USER_SYNC missing before version 1.2.	cl_bool	Is CL_TRUE if the devices preference is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, CL_FALSE if the device / implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX.
CL_DEVICE_PARENT_DEVICE missing before version 1.2.	cl_device_id	Returns the cl_device_id of the parent device to which this sub-device belongs. If *device* is a root-level device, a NULL value is returned.
CL_DEVICE_PARTITION_MAX_SUB_DEVICES missing before version 1.2.	cl_uint	Returns the maximum number of sub-devices that can be created when a device is partitioned. The value returned cannot exceed CL_DEVICE_MAX_COMPUTE_UNITS.
CL_DEVICE_PARTITION_PROPERTIES missing before version 1.2.	cl_device_partition_property[]	Returns the list of partition types supported by *device*. This is an array of cl_device_partition_property values drawn from the following list: CL_DEVICE_PARTITION_EQUALLY CL_DEVICE_PARTITION_BY_COUNTS CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN If the device cannot be partitioned (i.e. there is no partitioning scheme supported by the device that will return at least two sub-devices), a value of 0 will be returned.

Device Info	Return Type	Description
CL_DEVICE_PARTITION_AFFINITY_DOMAIN missing before version 1.2.	cl_device_affinity_domain	Returns the list of supported affinity domains for partitioning the device using CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN. This is a bit-field that describes one or more of the following values: CL_DEVICE_AFFINITY_DOMAIN_NUMA CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE If the device does not support any affinity domains, a value of 0 will be returned.
CL_DEVICE_PARTITION_TYPE missing before version 1.2.	cl_device_partition_property[]	Returns the properties argument specified in **clCreateSubDevices** if device is a sub-device. In the case where the properties argument to **clCreateSubDevices** is CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE, the affinity domain used to perform the partition will be returned. This can be one of the following values: CL_DEVICE_AFFINITY_DOMAIN_NUMA CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE Otherwise the implementation may either return a *param_value_size_ret* of 0 i.e. there is no partition type associated with device or can return a property value of 0 (where 0 is used to terminate the partition property list) in the memory that *param_value* points to.
CL_DEVICE_REFERENCE_COUNT [9] missing before version 1.2.	cl_uint	Returns the *device* reference count. If the device is a root-level device, a reference count of one is returned.

Device Info	Return Type	Description
CL_DEVICE_SVM_CAPABILITIES missing before version 2.0.	cl_device_svm_capabilities	Describes the various shared virtual memory (SVM) memory allocation types the device supports. This is a bit-field that describes a combination of the following values: CL_DEVICE_SVM_COARSE_GRAIN_BUFFER - Support for coarse-grain buffer sharing using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points and the host must use calls to **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject**. CL_DEVICE_SVM_FINE_GRAIN_BUFFER - Support for fine-grain buffer sharing using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points without need for **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject**. CL_DEVICE_SVM_FINE_GRAIN_SYSTEM - Support for sharing the host's entire virtual memory including memory allocated using **malloc**. Memory consistency is guaranteed at synchronization points. CL_DEVICE_SVM_ATOMICS - Support for the OpenCL 2.0 atomic operations that provide memory consistency across the host and all OpenCL devices supporting fine-grain SVM allocations. The mandated minimum capability for an OpenCL 2.0, 2.1, or 2.2 device is CL_DEVICE_SVM_COARSE_GRAIN_BUFFER. For other device versions there is no mandated minimum capability.
CL_DEVICE_PREFERRED_PLATFORM_ATOMIC_ALIGNMENT missing before version 2.0.	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 fine-grained SVM atomic types. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
CL_DEVICE_PREFERRED_GLOBAL_ATOMIC_ALIGNMENT missing before version 2.0.	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 atomic types to global memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.
CL_DEVICE_PREFERRED_LOCAL_ATOMIC_ALIGNMENT missing before version 2.0.	cl_uint	Returns the value representing the preferred alignment in bytes for OpenCL 2.0 atomic types to local memory. This query can return 0 which indicates that the preferred alignment is aligned to the natural size of the type.

Device Info	Return Type	Description
`CL_DEVICE_MAX_NUM_SUB_GROUPS` missing before version 2.1.	`cl_uint`	Maximum number of sub-groups in a work-group that a device is capable of executing on a single compute unit, for any given kernel-instance running on the device. The minimum value is 1 if the device supports sub-groups, and must be 0 for devices that do not support sub-groups. Support for sub-groups is required for an OpenCL 2.1, OpenCL 2.2, or OpenCL 3.1 device.
`CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS` missing before version 2.1.	`cl_bool`	Is `CL_TRUE` if this device supports independent forward progress of sub-groups, `CL_FALSE` otherwise. This query must return `CL_TRUE` for devices that support the `cl_khr_subgroups` extension, and must return `CL_FALSE` for devices that do not support sub-groups.

Device Info	Return Type	Description
CL_DEVICE_ATOMIC_MEMORY_CAPABILITIES missing before version 3.0.	cl_device_atomic_capabilities	Describes the various memory orders and scopes that the device supports for atomic memory operations. This is a bit-field that describes a combination of the following values: CL_DEVICE_ATOMIC_ORDER_RELAXED - Support for the **relaxed** memory order. CL_DEVICE_ATOMIC_ORDER_ACQ_REL - Support for the **acquire**, **release**, and **acquire-release** memory orders. CL_DEVICE_ATOMIC_ORDER_SEQ_CST - Support for the **sequentially consistent** memory order. Because atomic memory orders are hierarchical, a device that supports a strong memory order must also support all weaker memory orders. CL_DEVICE_ATOMIC_SCOPE_WORK_ITEM [10] - Support for memory ordering constraints that apply to a single work-item. CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP - Support for memory ordering constraints that apply to all work-items in a work-group. CL_DEVICE_ATOMIC_SCOPE_DEVICE - Support for memory ordering constraints that apply to all work-items executing on the device. CL_DEVICE_ATOMIC_SCOPE_ALL_DEVICES - Support for memory ordering constraints that apply to all work-items executing across all devices that can share SVM memory with each other and the host process. Because atomic scopes are hierarchical, a device that supports a wide scope must also support all narrower scopes, except for the work-item scope, which is a special case. The mandated minimum capability is: CL_DEVICE_ATOMIC_ORDER_RELAXED | CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP A device that does not support CL_DEVICE_SVM_ATOMICS (and hence does not support CL_MEM_SVM_ATOMICS) may still support CL_DEVICE_ATOMIC_SCOPE_ALL_DEVICES. On these devices, an atomic operation with **memory_scope_all_svm_devices** will behave the same as if the scope were **memory_scope_device** - refer to the memory consistency model.

Device Info	Return Type	Description
CL_DEVICE_ATOMIC_FENCE_CAPABILITIES missing before version 3.0.	cl_device_atomic_capabilities	Describes the various memory orders and scopes that the device supports for atomic fence operations. This is a bit-field that has the same set of possible values as described for CL_DEVICE_ATOMIC_MEMORY_CAPABILITIES. The mandated minimum capability is: CL_DEVICE_ATOMIC_ORDER_RELAXED | CL_DEVICE_ATOMIC_ORDER_ACQ_REL | CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP
CL_DEVICE_NON_UNIFORM_WORK_GROUP_SUPPORT missing before version 3.0.	cl_bool	Is CL_TRUE if the device supports non-uniform work-groups, and CL_FALSE otherwise.
CL_DEVICE_WORK_GROUP_COLLECTIVE_FUNCTIONS_SUPPORT missing before version 3.0.	cl_bool	Is CL_TRUE if the device supports work-group collective functions e.g. work_group_broadcast, work_group_reduce, and work_group_scan, and CL_FALSE otherwise.
CL_DEVICE_GENERIC_ADDRESS_SPACE_SUPPORT missing before version 3.0.	cl_bool	Is CL_TRUE if the device supports the generic address space and its associated built-in functions, and CL_FALSE otherwise.
CL_DEVICE_DEVICE_ENQUEUE_CAPABILITIES missing before version 3.0.	cl_device_device_enqueue_capabilities	Describes device-side enqueue capabilities of the device. This is a bit-field that describes one or more of the following values: CL_DEVICE_QUEUE_SUPPORTED - Device supports device-side enqueue and on-device queues. CL_DEVICE_QUEUE_REPLACEABLE_DEFAULT - Device supports a replaceable default on-device queue. If CL_DEVICE_QUEUE_REPLACEABLE_DEFAULT is set, CL_DEVICE_QUEUE_SUPPORTED must also be set. Devices that set CL_DEVICE_QUEUE_SUPPORTED for CL_DEVICE_DEVICE_ENQUEUE_CAPABILITIES must also return CL_TRUE for CL_DEVICE_GENERIC_ADDRESS_SPACE_SUPPORT.

Device Info	Return Type	Description
CL_DEVICE_PIPE_SUPPORT missing before version 3.0.	cl_bool	Is CL_TRUE if the device supports pipes, and CL_FALSE otherwise. Devices that return CL_TRUE for CL_DEVICE_PIPE_SUPPORT must also return CL_TRUE for CL_DEVICE_GENERIC_ADDRESS_SPACE_SUPPORT.
CL_DEVICE_PREFERRED_WORK_GROUP_SIZE_MULTIPLE missing before version 3.0.	size_t	Returns the preferred multiple of work-group size for the given device. This is a performance hint intended as a guide when specifying the local work size argument to **clEnqueueNDRangeKernel**. (Refer also to **clGetKernelWorkGroupInfo** where CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE can return a different value to CL_DEVICE_PREFERRED_WORK_GROUP_SIZE_MULTIPLE which may be more optimal.)
CL_DEVICE_LATEST_CONFORMANCE_VERSION_PASSED missing before version 3.0.	char[]	Returns the latest version of the conformance test suite that this device has fully passed in accordance with the official conformance process.
CL_DEVICE_UUID missing before version 3.1. or CL_DEVICE_UUID_KHR provided by the cl_khr_device_uuid extension.	cl_uchar[CL_UUID_SIZE] or cl_uchar[CL_UUID_SIZE_KHR]	Returns a universally unique identifier (UUID) for the device. Device UUIDs must be immutable for a given device across processes, driver APIs, driver versions, and system reboots. CL_UUID_SIZE or CL_UUID_SIZE_KHR is the size of the UUID, in bytes.
CL_DRIVER_UUID missing before version 3.1. or CL_DRIVER_UUID_KHR provided by the cl_khr_device_uuid extension.	cl_uchar[CL_UUID_SIZE] or cl_uchar[CL_UUID_SIZE_KHR]	Returns a universally unique identifier (UUID) for the software driver for the device. CL_UUID_SIZE or CL_UUID_SIZE_KHR is the size of the UUID, in bytes.

Device Info	Return Type	Description
CL_DEVICE_LUID_VALID missing before version 3.1. or CL_DEVICE_LUID_VALID_KHR provided by the cl_khr_device_uuid extension.	cl_bool	Returns CL_TRUE if the device has a valid LUID and CL_FALSE otherwise.
CL_DEVICE_LUID missing before version 3.1. or CL_DEVICE_LUID_KHR provided by the cl_khr_device_uuid extension.	cl_uchar[CL_LUID_SIZE] or cl_uchar[CL_LUID_SIZE_KHR]	Returns a locally unique identifier (LUID) for the device. It is not an error to query CL_DEVICE_LUID or CL_DEVICE_LUID_KHR when CL_DEVICE_LUID_VALID or CL_DEVICE_LUID_VALID_KHR returns CL_FALSE, but in this case the returned LUID value is undefined. When CL_DEVICE_LUID_VALID_KHR or CL_DEVICE_LUID_VALID_KHR returns CL_TRUE, and the OpenCL device is running on the Windows operating system, the returned LUID value can be cast to an LUID object and must be equal to the locally unique identifier of an IDXGIAdapter1 object that corresponds to the OpenCL device. CL_LUID_SIZE_KHR or CL_LUID_SIZE_KHR is the size of the LUID, in bytes.
CL_DEVICE_NODE_MASK missing before version 3.1. or CL_DEVICE_NODE_MASK_KHR provided by the cl_khr_device_uuid extension.	cl_uint	Returns a node mask for the device. It is not an error to query CL_DEVICE_NODE_MASK or CL_DEVICE_NODE_MASK_KHR when CL_DEVICE_LUID_VALID or CL_DEVICE_LUID_VALID_KHR returns CL_FALSE, but in this case the returned node mask is undefined. When CL_DEVICE_LUID_VALID_KHR or CL_DEVICE_LUID_VALID_KHR returns CL_TRUE, the returned node mask must contain exactly one bit. If the OpenCL device is running on an operating system that supports the Direct3D 12 API and the OpenCL device corresponds to an individual device in a linked device adapter, the returned node mask identifies the Direct3D 12 node corresponding to the OpenCL device. Otherwise, the returned node mask must be 1.

Device Info	Return Type	Description
CL_DEVICE_INTEGER_DOT_PRODUCT_CAPABILITIES missing before version 3.1. or CL_DEVICE_INTEGER_DOT_PRODUCT_CAPABILITIES_KHR provided by the cl_khr_integer_dot_product extension.	cl_device_integer_dot_product_capabilities or cl_device_integer_dot_product_capabilities_khr	Returns the integer dot product capabilities supported by the device. All implementations that support OpenCL C 3.1 or cl_khr_integer_dot_product must support CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_PACKED or CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_PACKED_KHR to indicate support for dot product built-in functions and, when SPIR-V is supported, SPIR-V instructions that take four-component vectors of 8-bit integers packed into 32-bit integers as input. CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT or CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_KHR may be set to indicate support for dot product built-in functions and, when SPIR-V is supported, SPIR-V instructions that take four-component of 8-bit elements as input are supported. CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_KHR must be set when version 2.0 or higher of the cl_khr_integer_dot_product extension is supported.
CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_8BIT missing before version 3.1. or CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_8BIT_KHR provided by the cl_khr_integer_dot_product extension.	cl_device_integer_dot_product_acceleration_properties or cl_device_integer_dot_product_acceleration_properties_khr	Returns a structure describing the exact 8-bit dot product combinations that are accelerated on the device. Each member is CL_TRUE if the combination it corresponds to is accelerated, CL_FALSE otherwise. CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_8BIT_KHR is missing before version 2.0 of the cl_khr_integer_dot_product extension.

Device Info	Return Type	Description
CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_4x8BIT_PACKED missing before version 3.1. or CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_4x8BIT_PACKED_KHR provided by the cl_khr_integer_dot_product extension.	cl_device_integer_dot_product_acceleration_properties or cl_device_integer_dot_product_acceleration_properties_khr	Returns a structure describing the exact 4x8-bit packed dot product combinations that are accelerated on the device. Each member is CL_TRUE if the combination it corresponds to is accelerated, CL_FALSE otherwise. CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_4x8BIT_PACKED_KHR is missing before version 2.0 of the cl_khr_integer_dot_product extension.
CL_DEVICE_SPIRV_EXTENDED_INSTRUCTION_SETS missing before version 3.1. or CL_DEVICE_SPIRV_EXTENDED_INSTRUCTION_SETS_KHR provided by the cl_khr_spirv_queries extension.	const char*[]	Returns an array of null-terminated strings, where each string describes a SPIR-V extended instruction set that is supported by the device.
CL_DEVICE_SPIRV_EXTENSIONS missing before version 3.1. or CL_DEVICE_SPIRV_EXTENSIONS_KHR provided by the cl_khr_spirv_queries extension.	const char*[]	Returns an array of null-terminated strings, where each string describes a SPIR-V extension that is supported by the device.

Device Info	Return Type	Description
CL_DEVICE_SPIRV_CAPABILITIES missing before version 3.1. or CL_DEVICE_SPIRV_CAPABILITIES_KHR provided by the cl_khr_spirv_queries extension.	cl_uint[]	Returns an array of SPIR-V token values, where each value describes a SPIR-V capability that is supported by the device.
CL_DEVICE_COMMAND_BUFFER_CAPABILITIES_KHR provided by the cl_khr_command_buffer extension.	cl_device_command_buffer_capabilities_khr	Describes device command-buffer capabilities, encoded as bits in a bitfield. Supported capabilities are: CL_COMMAND_BUFFER_CAPABILITY_KERNEL_PRINTF_KHR Device supports the ability to record commands that execute kernels which contain printf calls. provided by the cl_khr_command_buffer extension. CL_COMMAND_BUFFER_CAPABILITY_DEVICE_SIDE_ENQUEUE_KHR Device supports the ability to record commands that execute kernels which contain device-side kernel-enqueue calls. provided by the cl_khr_command_buffer extension. CL_COMMAND_BUFFER_CAPABILITY_SIMULTANEOUS_USE_KHR Device supports enqueueing command-buffers with a simultaneous use usage pattern. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_COMMAND_BUFFER_CAPABILITY_MULTIPLE_QUEUE_KHR Device supports the ability to record commands to more than one command-queue associated with *device* in a single command-buffer. provided by the cl_khr_command_buffer_multi_device extension.

Device Info	Return Type	Description
CL_DEVICE_COMMAND_BUFFER_REQUIRED_QUEUE_PROPERTIES_KHR provided by the cl_khr_command_buffer extension.	cl_command_queue_properties	Bitmask of the minimum properties with which a command-queue must be created to allow a command-buffer to be executed on it. It is valid for a command-queue to be created with extra properties in addition to this base requirement and still be compatible with command-buffer execution.
CL_DEVICE_COMMAND_BUFFER_SUPPORTED_QUEUE_PROPERTIES_KHR provided by the cl_khr_command_buffer extension.	cl_command_queue_properties	Bitmask of the supported properties with which a command-queue may be created to allow a command-buffer to be executed on it. It is invalid for a command-queue to be created with a property not reported and still be compatible with command-buffer execution. The mandated minimum capability is: CL_QUEUE_PROFILING_ENABLE.
CL_DEVICE_COMMAND_BUFFER_NUM_SYNC_DEVICES_KHR provided by the cl_khr_command_buffer_multi_device extension.	cl_uint	Return the number of root devices listed in CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR that *device* can use device-side synchronization with.
CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR provided by the cl_khr_command_buffer_multi_device extension.	cl_device_id[]	Return the list of root devices *device* can use device-side synchronization with. A device should list itself only if it has native support for synchronizing commands. Sub-devices are not listed to avoid non-deterministic results as sub-devices are created. Instead if a root device is listed, then any of its partitioned sub-devices can also be natively synchronized with.

Device Info	Return Type	Description
CL_DEVICE_MUTABLE_DISPATCH_CAPABILITIES_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	cl_mutable_dispatch_fields_khr	Describes device mutable-dispatch capabilities, encoded as bits in a bitfield. Supported capabilities are: CL_MUTABLE_DISPATCH_GLOBAL_OFFSET_KHR - Device supports the ability to modify the *global_work_offset* of kernel execution after command recording. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR - Device supports the ability to modify the *global_work_size* of kernel execution after command recording. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_MUTABLE_DISPATCH_LOCAL_SIZE_KHR - Device supports the ability to modify the *local_work_size* of kernel execution after command recording. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_MUTABLE_DISPATCH_ARGUMENTS_KHR - Device supports the ability to modify arguments set on a kernel after command recording. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_MUTABLE_DISPATCH_EXEC_INFO_KHR - Device supports the ability to modify execution information set on a kernel after command recording. provided by the cl_khr_command_buffer_mutable_dispatch extension.
CL_DEVICE_EXTERNAL_MEMORY_IMPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_memory extension.	cl_external_memory_handle_type_khr[]	Returns the list of importable external memory handle types supported by *device*. Must return a non-empty list of external memory handle types for at least one of the devices in the platform.

Device Info	Return Type	Description
CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR provided by the cl_khr_external_memory extension.	cl_external_memory_handle_type_khr[]	Returns the list of importable external memory handle types supported by *device*, that are assumed to apply linear layout to imported images when no other tiling information is provided. This list contains a subset of CL_DEVICE_EXTERNAL_MEMORY_IMPORT_HANDLE_TYPES_KHR. The returned list may be empty. External memory handle types not in CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR may have any memory layout. The layout interpretation of images imported with these handle types is implementation defined.
CL_DEVICE_HALF_FP_CONFIG provided by the cl_khr_fp16 extension.	cl_device_fp_config	Describes half-precision floating-point capability of the OpenCL device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM - denorms are supported CL_FP_INF_NAN - INF and NaNs are supported CL_FP_ROUND_TO_NEAREST - round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO - round to zero rounding mode supported CL_FP_ROUND_TO_INF - round to positive and negative infinity rounding modes supported CL_FP_FMA - the fma kernel builtin is implemented using hardware-accelerated fused multiply-add as defined in IEEE754-2008. CL_FP_SOFT_FLOAT - Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software If half-precision is supported by the device, then the minimum half-precision floating-point capability is either: CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST | CL_FP_INF_NAN.

Device Info	Return Type	Description
CL_DEVICE_KERNEL_CLOCK_CAPABILITIES_KHR provided by the cl_khr_kernel_clock extension.	cl_device_kernel_clock_capabilities_khr	Returns the kernel clock capabilities of the device. CL_DEVICE_KERNEL_CLOCK_SCOPE_DEVICE_KHR is set when kernels are allowed to call the clock_read_device and clock_read_hilo_device OpenCL-C functions. CL_DEVICE_KERNEL_CLOCK_SCOPE_WORK_GROUP_KHR is set when kernels are allowed to call the clock_read_work_group and clock_read_hilo_work_group OpenCL-C functions. CL_DEVICE_KERNEL_CLOCK_SCOPE_SUB_GROUP_KHR is set when kernels are allowed to call the clock_read_sub_group and clock_read_hilo_sub_group OpenCL-C functions.
CL_DEVICE_PCI_BUS_INFO_KHR provided by the cl_khr_pci_bus_info extension.	cl_device_pci_bus_info_khr	Returns PCI bus information for the device. The PCI bus information is returned as a single structure that includes the PCI bus domain, the PCI bus identifier, the PCI device identifier, and the PCI device function identifier.
CL_DEVICE_SEMAPHORE_TYPES_KHR provided by the cl_khr_semaphore extension.	cl_semaphore_type_khr[]	Returns the list of the semaphore types supported by *device*. Must return a non-empty list for at least one of the devices in the platform, meeting the minimum requirements described for cl_semaphore_type_khr.
CL_DEVICE_SEMAPHORE_IMPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_semaphore extension.	cl_external_semaphore_handle_type_khr[]	Returns the list of importable external semaphore handle types supported by *device*. This size of this query may be 0 indicating that the device does not support importing semaphores.
CL_DEVICE_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_semaphore extension.	cl_external_semaphore_handle_type_khr[]	Returns the list of exportable external semaphore handle types supported by *device*. This size of this query may be 0 indicating that the device does not support exporting semaphores.
CL_DEVICE_SPIR_VERSIONS provided by the cl_khr_spir extension.	char[]	A space separated list of SPIR versions supported by the device. For example, returning "1.2" in this query implies that SPIR version 1.2 is supported by the implementation.

Device Info	Return Type	Description
CL_DEVICE_MAX_NAMED_BARRIER_COUNT_KHR provided by the cl_khr_subgroup_named_barrier extension.	cl_uint	Maximum number of named barriers in a work-group for any given kernel-instance running on the device. The minimum value is 8.
CL_DEVICE_TERMINATE_CAPABILITY_KHR provided by the cl_khr_terminate_context extension.	cl_device_terminate_capability_khr	Describes the termination capability of the OpenCL device. This is a bit-field, where the following values are currently supported: CL_DEVICE_TERMINATE_CAPABILITY_CONTEXT_KHR - Indicates that context termination is supported.
CL_DEVICE_CXX_FOR_OPENCL_NUMERIC_VERSION_EXT provided by the cl_ext_cxx_for_opencl extension.	cl_version	Returns the version of the C++ for OpenCL language supported by the device compiler.

OpenCL 3.0 or newer devices must report the following feature macros via CL_DEVICE_OPENCL_C_FEATURES when the corresponding bit is set in the bitfield returned for CL_DEVICE_INTEGER_DOT_PRODUCT_CAPABILITIES or CL_DEVICE_INTEGER_DOT_PRODUCT_CAPABILITIES_KHR:

Feature Bit	Feature Macro
CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_PACKED or CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_PACKED_KHR	__opencl_c_integer_dot_product_input_4x8bit_packed
CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT or CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_KHR	__opencl_c_integer_dot_product_input_4x8bit

OpenCL 3.0 devices must report the following feature macros via CL_DEVICE_OPENCL_C_FEATURES when the corresponding bit is set in the bitfield returned for CL_DEVICE_KERNEL_CLOCK_CAPABILITIES_KHR:

Feature Bit	Feature Macro
CL_DEVICE_KERNEL_CLOCK_SCOPE_DEVICE_KHR	__opencl_c_kernel_clock_scope_device
CL_DEVICE_KERNEL_CLOCK_SCOPE_WORK_GROUP_KHR	__opencl_c_kernel_clock_scope_work_group
CL_DEVICE_KERNEL_CLOCK_SCOPE_SUB_GROUP_KHR	__opencl_c_kernel_clock_scope_sub_group

One of the two queries CL_DEVICE_SEMAPHORE_IMPORT_HANDLE_TYPES_KHR and CL_DEVICE_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR must return a non-empty list indicating support for at least one of the valid

semaphore handle types either for import, for export, or both.

Note

While CL_DEVICE_UUID_KHR is specified to remain consistent across driver versions and system reboots, it is not intended to be usable as a serializable persistent identifier for a device. It may change when a device is physically added to, removed from, or moved to a different connector in a system while that system is powered down. Further, there is no reasonable way to verify with conformance testing that a given device retains the same UUID in a given system across all driver versions supported in that system. While implementations should make every effort to report consistent device UUIDs across driver versions, applications should avoid relying on the persistence of this value for uses other than identifying compatible devices for external object sharing purposes.

clGetDeviceInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_DEVICE
  - if *device* is not a valid device
- CL_INVALID_VALUE
  - if *param_name* is not one of the supported values
  - if the size in bytes specified by *param_value_size* is less than the size of the return type specified in the Device Queries table and *param_value* is not NULL
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The cl_device_integer_dot_product_acceleration_properties structure:

```
// Provided by CL_VERSION_3_1
typedef struct cl_device_integer_dot_product_acceleration_properties {
    cl_bool     signed_accelerated;
    cl_bool     unsigned_accelerated;
    cl_bool     mixed_signedness_accelerated;
    cl_bool     accumulating_saturating_signed_accelerated;
    cl_bool     accumulating_saturating_unsigned_accelerated;
    cl_bool     accumulating_saturating_mixed_signedness_accelerated;
} cl_device_integer_dot_product_acceleration_properties;
```

or the equivalent cl_device_integer_dot_product_acceleration_properties_khr structure:

```
// Provided by cl_khr_integer_dot_product
typedef struct cl_device_integer_dot_product_acceleration_properties_khr {
    cl_bool    signed_accelerated;
    cl_bool    unsigned_accelerated;
    cl_bool    mixed_signedness_accelerated;
    cl_bool    accumulating_saturating_signed_accelerated;
    cl_bool    accumulating_saturating_unsigned_accelerated;
    cl_bool    accumulating_saturating_mixed_signedness_accelerated;
} cl_device_integer_dot_product_acceleration_properties_khr;
```

describes the exact dot product operations that are accelerated on the device.

- *signed_accelerated* is `CL_TRUE` when signed dot product operations are accelerated, `CL_FALSE` otherwise.
- *unsigned_accelerated* is `CL_TRUE` when unsigned dot product operations are accelerated, `CL_FALSE` otherwise.
- *mixed_signedness_accelerated* is `CL_TRUE` when mixed signedness dot product operations are accelerated, `CL_FALSE` otherwise.
- *accumulating_saturating_signed_accelerated* is `CL_TRUE` when accumulating saturating signed dot product operations are accelerated, `CL_FALSE` otherwise.
- *accumulating_saturating_unsigned_accelerated* is `CL_TRUE` when accumulating saturating unsigned dot product operations are accelerated, `CL_FALSE` otherwise.
- *accumulating_saturating_mixed_signedness_accelerated* is `CL_TRUE` when accumulating saturating mixed signedness dot product operations are accelerated, `CL_FALSE` otherwise.

A dot product operation is deemed accelerated if its implementation provides a performance advantage over application-provided code composed from elementary instructions and/or other dot product instructions, either because the implementation uses optimized machine code sequences whose generation from application-provided code cannot be guaranteed or because it uses hardware features that cannot otherwise be targeted from application-provided code.

The `cl_device_pci_bus_info_khr` structure describes PCI bus information for a device:

```
// Provided by cl_khr_pci_bus_info
typedef struct cl_device_pci_bus_info_khr {
    cl_uint    pci_domain;
    cl_uint    pci_bus;
    cl_uint    pci_device;
    cl_uint    pci_function;
} cl_device_pci_bus_info_khr;
```

- *pci_domain* is the PCI bus domain of the device.
- *pci_bus* is the PCI bus identified of the device.
- *pci_device* is the PCI device identifier of the device.

- *pci_function* is the PCI device function identifier of the device.

To query device and host timestamps, call the function:

```
// Provided by CL_VERSION_2_1
cl_int clGetDeviceAndHostTimer(
    cl_device_id device,
    cl_ulong* device_timestamp,
    cl_ulong* host_timestamp);
```

clGetDeviceAndHostTimer is missing before version 2.1.

- *device* is a device returned by **clGetDeviceIDs**.
- *device_timestamp* will be updated with the value of the device timer in nanoseconds. The resolution of the timer is the same as the device profiling timer returned by **clGetDeviceInfo** and the `CL_DEVICE_PROFILING_TIMER_RESOLUTION` query.
- *host_timestamp* will be updated with the value of the host timer in nanoseconds at the closest possible point in time to that at which *device_timer* was returned. The resolution of the timer may be queried via **clGetPlatformInfo** and the flag `CL_PLATFORM_HOST_TIMER_RESOLUTION`.

clGetDeviceAndHostTimer returns a reasonably synchronized pair of timestamps from the device timer and the host timer as seen by *device*. Implementations may need to execute this query with a high latency in order to provide reasonable synchronization of the timestamps. The host timestamp and device timestamp returned by this function and **clGetHostTimer** each have an implementation-defined timebase. The timestamps will always be in their respective timebases regardless of which query function is used. The timestamp returned from **clGetEventProfilingInfo** for an event on a device and a device timestamp queried from the same device will always be in the same timebase.

clGetDeviceAndHostTimer will return `CL_SUCCESS` with a time value in *host_timestamp* if provided. Otherwise, it returns one of the following errors:

- `CL_INVALID_DEVICE`
  - if *device* is not a valid device
- `CL_INVALID_OPERATION`
  - if the platform associated with *device* does not support device and host timer synchronization
- `CL_INVALID_VALUE`
  - if *host_timestamp* is `NULL`
  - if *device_timestamp* is `NULL`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device

- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To query the host clock, call the function:

```
// Provided by CL_VERSION_2_1
cl_int clGetHostTimer(
    cl_device_id device,
    cl_ulong* host_timestamp);
```

clGetHostTimer is missing before version 2.1.

- *device* is a device returned by **clGetDeviceIDs**.
- *host_timestamp* will be updated with the value of the current timer in nanoseconds. The resolution of the timer may be queried via **clGetPlatformInfo** and the flag `CL_PLATFORM_HOST_TIMER_RESOLUTION`.

clGetHostTimer returns the current value of the host clock as seen by *device*. This value is in the same timebase as the *host_timestamp* returned from **clGetDeviceAndHostTimer**. The implementation will return with as low a latency as possible to allow a correlation with a subsequent application sampled time. The host timestamp and device timestamp returned by this function and **clGetDeviceAndHostTimer** each have an implementation-defined timebase. The timestamps will always be in their respective timebases regardless of which query function is used. The timestamp returned from **clGetEventProfilingInfo** for an event on a device and a device timestamp queried from the same device will always be in the same timebase.

clGetHostTimer will return `CL_SUCCESS` with a time value in *host_timestamp* if provided. Otherwise, it returns one of the following errors:

- `CL_INVALID_DEVICE`
  - if *device* is not a valid device
- `CL_INVALID_OPERATION`
  - if the platform associated with *device* does not support device and host timer synchronization
- `CL_INVALID_VALUE`
  - if *host_timestamp* is `NULL`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.2.1. Required OpenCL Extensions

Some OpenCL features were originally OpenCL extensions before adoption into the core OpenCL standard. This section describes the OpenCL extensions that must be reported by a device through `CL_DEVICE_EXTENSIONS` or `CL_DEVICE_EXTENSIONS_WITH_VERSION` for the OpenCL version reported by `CL_DEVICE_VERSION`.

The following Khronos extension names must be returned by all devices that support OpenCL 1.1 or newer:

`cl_khr_byte_addressable_store`
`cl_khr_global_int32_base_atomics`
`cl_khr_global_int32_extended_atomics`
`cl_khr_local_int32_base_atomics`
`cl_khr_local_int32_extended_atomics`

The following Khronos extension names must be returned by all devices that support OpenCL 1.2 or newer when and only when the optional feature is supported:

`cl_khr_fp64`

The following Khronos extension names must be returned by all devices that support OpenCL 2.0, OpenCL 2.1, or OpenCL 2.2. For devices that support OpenCL 3.0 or newer, these extension names must be returned when and only when the optional feature is supported:

`cl_khr_3d_image_writes`
`cl_khr_depth_images`
`cl_khr_image2d_from_buffer`

The following Khronos extension names must be returned by all devices that support OpenCL 3.1 or newer:

`cl_khr_device_uuid`
`cl_khr_extended_bit_ops`
`cl_khr_integer_dot_product`
`cl_khr_spirv_queries`
`cl_khr_subgroup_extended_types`
`cl_khr_subgroup_rotate`
`cl_khr_subgroup_shuffle`
`cl_khr_subgroup_shuffle_relative`
`cl_khr_suggested_local_work_size`

4.2.2. Sharing DirectX9 Media Surfaces With OpenCL Images

This section discusses OpenCL functions that allow applications to use media surfaces as OpenCL memory objects. This allows efficient sharing of data between OpenCL and media surface APIs. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also media surfaces. An OpenCL image object may be created from a media surface. OpenCL memory objects may be created from media surfaces if and only if the OpenCL context has been created from a media adapter.

4.2.2.1. Querying OpenCL Devices Corresponding to Media Adapters

Media adapters are an abstraction associated with devices that provide media capabilities. Adapters with associated OpenCL devices can enable media surface sharing between the two.

To query a media adapter for any associated OpenCL devices, call the function

```
// Provided by cl_khr_dx9_media_sharing
cl_int clGetDeviceIDsFromDX9MediaAdapterKHR(
    cl_platform_id platform,
    cl_uint num_media_adapters,
    cl_dx9_media_adapter_type_khr* media_adapter_type,
    void* media_adapters,
    cl_dx9_media_adapter_set_khr media_adapter_set,
    cl_uint num_entries,
    cl_device_id* devices,
    cl_uint* num_devices);
```

clGetDeviceIDsFromDX9MediaAdapterKHR is provided by the `cl_khr_dx9_media_sharing` extension.

- *platform* refers to the platform ID returned by **clGetPlatformIDs**.
- *num_media_adapters* specifies the number of media adapters.
- *media_adapters_type* is an array of *num_media_adapters* entries. Each entry specifies the type of media adapter and must be one of the values described in the media adapter type table below.
- *media_adapters* is an array of *num_media_adapters* entries. Each entry specifies the actual adapter whose type is specified by *media_adapter_type*. The *media_adapters* must be one of the types described in the cl_dx9_media_adapter_type_khr values table.
- *media_adapter_set* specifies the set of adapters to return and must be one of the values described in the cl_dx9_media_adapter_set_khr values table.
- *num_entries* is the number of `cl_device_id` entries that can be added to *devices*. If *devices* is not `NULL`, the *num_entries* must be greater than zero.
- *devices* returns a list of OpenCL devices found that support the list of media adapters specified. The `cl_device_id` values returned in *devices* can be used to identify a specific OpenCL device. If *devices* argument is `NULL`, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* or the number of OpenCL devices whose type matches *device_type*.
- *num_devices* returns the number of OpenCL devices. If *num_devices* is `NULL`, this argument is ignored.

Table 6. DirectX 9 object types that may be used by **clGetDeviceIDsFromDX9MediaAdapterKHR**

cl_dx9_media_adapter_type_khr	Type of Media Adapter
CL_ADAPTER_D3D9_KHR provided by the cl_khr_dx9_media_sharing extension.	IDirect3DDevice9 *
CL_ADAPTER_D3D9EX_KHR provided by the cl_khr_dx9_media_sharing extension.	IDirect3DDevice9Ex *
CL_ADAPTER_DXVA_KHR provided by the cl_khr_dx9_media_sharing extension.	IDXVAHD_Device *

Table 7. Sets of devices queriable using **clGetDeviceIDsFromDX9MediaAdapterKHR**

cl_dx9_media_adapter_set_khr	Description
CL_PREFERRED_DEVICES_FOR_DX9_MEDIA_ADAPTER_KHR provided by the cl_khr_dx9_media_sharing extension.	The preferred OpenCL devices associated with the media adapter.
CL_ALL_DEVICES_FOR_DX9_MEDIA_ADAPTER_KHR provided by the cl_khr_dx9_media_sharing extension.	All OpenCL devices that may interoperate with the media adapter

clGetDeviceIDsFromDX9MediaAdapterKHR returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PLATFORM
  - if *platform* is not a valid platform
- CL_INVALID_VALUE
  - if *num_media_adapters* is zero
  - if *media_adapters_type* is NULL
  - if *media_adapters* is NULL
  - if any of the entries in *media_adapters_type* is not a valid value
  - if any of the entries in *media_adapters* are not valid
  - if *media_adapter_set* is not a valid value
  - if *num_entries* is equal to zero and *devices* is not NULL
  - if both *num_devices* and *devices* are NULL
- CL_DEVICE_NOT_FOUND
  - if no OpenCL devices that correspond to adapters specified in *media_adapters* and

media_adapters_type were found

- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.2.3. Sharing Direct3D 10 Resources With OpenCL Memory Objects

This section discusses OpenCL functions that allow applications to use Direct3D 10 resources as OpenCL memory objects. This allows efficient sharing of data between OpenCL and Direct3D 10. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also Direct3D 10 resources. An OpenCL image object may be created from a Direct3D 10 texture resource. An OpenCL buffer object may be created from a Direct3D 10 buffer resource. OpenCL memory objects may be created from Direct3D 10 objects if and only if the OpenCL context has been created from a Direct3D 10 device.

4.2.3.1. Querying OpenCL Devices Corresponding to Direct3D 10 Devices

The OpenCL devices corresponding to a Direct3D 10 device may be queried. The OpenCL devices corresponding to a DXGI adapter may also be queried. The OpenCL devices corresponding to a Direct3D 10 device will be a subset of the OpenCL devices corresponding to the DXGI adapter against which the Direct3D 10 device was created.

To query OpenCL devices corresponding to a Direct3D 10 device or a DXGI device, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_int clGetDeviceIDsFromD3D10KHR(
    cl_platform_id platform,
    cl_d3d10_device_source_khr d3d_device_source,
    void* d3d_object,
    cl_d3d10_device_set_khr d3d_device_set,
    cl_uint num_entries,
    cl_device_id* devices,
    cl_uint* num_devices);
```

clGetDeviceIDsFromD3D10KHR is provided by the `cl_khr_d3d10_sharing` extension.

- *platform* refers to the platform ID returned by **clGetPlatformIDs**.
- *d3d_device_source* specifies the type of *d3d_object*, and must be one of the values shown in the Direct3D 10 Object Types table.
- *d3d_object* specifies the object whose corresponding OpenCL devices are being queried. The type of *d3d_object* must be as specified in the Direct3D 10 Object Types table.
- *d3d_device_set* specifies the set of devices to return, and must be one of the values shown in the

Direct3D 10 Device Sets table.

- *num_entries* is the number of `cl_device_id` entries that can be added to *devices*. If *devices* is not `NULL` then *num_entries* must be greater than zero.
- *devices* returns a list of OpenCL devices found. The `cl_device_id` values returned in *devices* can be used to identify a specific OpenCL device. If *devices* is `NULL`, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* and the number of OpenCL devices corresponding to *d3d_object*.
- *num_devices* returns the number of OpenCL devices available that correspond to *d3d_object*. If *num_devices* is `NULL`, this argument is ignored.

Table 8. Direct3D 10 object types that may be used by **clGetDeviceIDsFromD3D10KHR**

`cl_d3d10_device_source_khr`	**Type of** *d3d_object*
`CL_D3D10_DEVICE_KHR` provided by the `cl_khr_d3d10_sharing` extension.	`ID3D10Device *`
`CL_D3D10_DXGI_ADAPTER_KHR` provided by the `cl_khr_d3d10_sharing` extension.	`IDXGIAdapter *`

Table 9. Sets of devices queriable using **clGetDeviceIDsFromD3D10KHR**

`cl_d3d10_device_set_khr`	**Devices returned in** *devices*
`CL_PREFERRED_DEVICES_FOR_D3D10_KHR` provided by the `cl_khr_d3d10_sharing` extension.	The preferred OpenCL devices associated with the specified Direct3D object.
`CL_ALL_DEVICES_FOR_D3D10_KHR` provided by the `cl_khr_d3d10_sharing` extension.	All OpenCL devices which may interoperate with the specified Direct3D object. Performance of sharing data on these devices may be considerably less than on the preferred devices.

clGetDeviceIDsFromD3D10KHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise it may return

- `CL_INVALID_PLATFORM`
  - if *platform* is not a valid platform
- `CL_INVALID_VALUE`
  - if *d3d_device_source* is not a valid value
  - if *d3d_device_set* is not a valid value
  - if *num_entries* is equal to zero and *devices* is not `NULL`
  - if both *num_devices* and *devices* are `NULL`
- `CL_DEVICE_NOT_FOUND`
  - if no OpenCL devices that correspond to *d3d_object* were found.

- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.2.4. Sharing Direct3D 11 Resources With OpenCL Memory Objects

This section discusses OpenCL functions that allow applications to use Direct3D 11 resources as OpenCL memory objects. This allows efficient sharing of data between OpenCL and Direct3D 11. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also Direct3D 11 resources. An OpenCL image object may be created from a Direct3D 11 texture resource. An OpenCL buffer object may be created from a Direct3D 11 buffer resource. OpenCL memory objects may be created from Direct3D 11 objects if and only if the OpenCL context has been created from a Direct3D 11 device.

4.2.4.1. Querying OpenCL Devices Corresponding to Direct3D 11 Devices

The OpenCL devices corresponding to a Direct3D 11 device may be queried. The OpenCL devices corresponding to a DXGI adapter may also be queried. The OpenCL devices corresponding to a Direct3D 11 device will be a subset of the OpenCL devices corresponding to the DXGI adapter against which the Direct3D 11 device was created.

To query OpenCL devices corresponding to a Direct3D 11 device or a DXGI device, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_int clGetDeviceIDsFromD3D11KHR(
    cl_platform_id platform,
    cl_d3d11_device_source_khr d3d_device_source,
    void* d3d_object,
    cl_d3d11_device_set_khr d3d_device_set,
    cl_uint num_entries,
    cl_device_id* devices,
    cl_uint* num_devices);
```

clGetDeviceIDsFromD3D11KHR is provided by the `cl_khr_d3d11_sharing` extension.

- *platform* refers to the platform ID returned by **clGetPlatformIDs**.
- *d3d_device_source* specifies the type of *d3d_object*, and must be one of the values shown in the Direct3D 11 Object Types table.
- *d3d_object* specifies the object whose corresponding OpenCL devices are being queried. The type of *d3d_object* must be as specified in the Direct3D 11 Object Types table.
- *d3d_device_set* specifies the set of devices to return, and must be one of the values shown in the Direct3D 11 Device Sets table.

- *num_entries* is the number of `cl_device_id` entries that can be added to *devices*. If *devices* is not `NULL` then *num_entries* must be greater than zero.
- *devices* returns a list of OpenCL devices found. The `cl_device_id` values returned in *devices* can be used to identify a specific OpenCL device. If *devices* is `NULL`, this argument is ignored. The number of OpenCL devices returned is the minimum of the value specified by *num_entries* and the number of OpenCL devices corresponding to *d3d_object*.
- *num_devices* returns the number of OpenCL devices available that correspond to *d3d_object*. If *num_devices* is `NULL`, this argument is ignored.

Table 10. Direct3D 11 object types that may be used by **clGetDeviceIDsFromD3D11KHR**

`cl_d3d11_device_source_khr`	**Type of** *d3d_object*
`CL_D3D11_DEVICE_KHR` provided by the `cl_khr_d3d11_sharing` extension.	`ID3D11Device *`
`CL_D3D11_DXGI_ADAPTER_KHR` provided by the `cl_khr_d3d11_sharing` extension.	`IDXGIAdapter *`

Table 11. Sets of devices queriable using **clGetDeviceIDsFromD3D11KHR**

`cl_d3d11_device_set_khr`	**Devices returned in** *devices*
`CL_PREFERRED_DEVICES_FOR_D3D11_KHR` provided by the `cl_khr_d3d11_sharing` extension.	The preferred OpenCL devices associated with the specified Direct3D object.
`CL_ALL_DEVICES_FOR_D3D11_KHR` provided by the `cl_khr_d3d11_sharing` extension.	All OpenCL devices which may interoperate with the specified Direct3D object. Performance of sharing data on these devices may be considerably less than on the preferred devices.

clGetDeviceIDsFromD3D11KHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise it may return

- `CL_INVALID_PLATFORM`
  - if *platform* is not a valid platform
- `CL_INVALID_VALUE`
  - if *d3d_device_source* is not a valid value
  - if *d3d_device_set* is not a valid value
  - if *num_entries* is equal to zero and *devices* is not `NULL`
  - if both *num_devices* and *devices* are `NULL`
- `CL_DEVICE_NOT_FOUND`
  - if no OpenCL devices that correspond to *d3d_object* were found.
- `CL_OUT_OF_RESOURCES`

- if there is a failure to allocate resources required by the OpenCL implementation on the device

- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.3. Partitioning a Device

Partitioning devices is missing before version 1.2.

To create sub-devices partitioning an OpenCL device, call the function:

```
// Provided by CL_VERSION_1_2
cl_int clCreateSubDevices(
    cl_device_id in_device,
    const cl_device_partition_property* properties,
    cl_uint num_devices,
    cl_device_id* out_devices,
    cl_uint* num_devices_ret);
```

clCreateSubDevices is missing before version 1.2.

- *in_device* is the device to be partitioned.
- *properties* specifies how *in_device* is to be partitioned, described by a partition name and its corresponding value. Each partition name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported partitioning schemes is described in the Sub-device Partition table. Only one of the listed partitioning schemes can be specified in *properties*.
- *num_devices* is the size of memory pointed to by *out_devices* specified as the number of `cl_device_id` entries.
- *out_devices* is the buffer where the OpenCL sub-devices will be returned. If *out_devices* is `NULL`, this argument is ignored. If *out_devices* is not `NULL`, *num_devices* must be greater than or equal to the number of sub-devices that *device* may be partitioned into according to the partitioning scheme specified in *properties*.
- *num_devices_ret* returns the number of sub-devices that *device* may be partitioned into according to the partitioning scheme specified in *properties*. If *num_devices_ret* is `NULL`, it is ignored.

clCreateSubDevices creates an array of sub-devices that each reference a non-intersecting set of compute units within *in_device*, according to the partition scheme given by *properties*. The output sub-devices may be used in every way that the root (or parent) device can be used, including creating contexts, building programs, further calls to **clCreateSubDevices** and creating command-queues. When a command-queue is created against a sub-device, the commands enqueued on the queue are executed only on the sub-device.

Table 12. List of supported partition schemes by **clCreateSubDevices**

Partition Property	Partition Value	Description
CL_DEVICE_PARTITION_EQUALLY missing before version 1.2.	cl_uint	Split the aggregate device into as many smaller aggregate devices as can be created, each containing *n* compute units. The value *n* is passed as the value accompanying this property. If *n* does not divide evenly into CL_DEVICE_MAX_COMPUTE_UNITS, then the remaining compute units are not used.
CL_DEVICE_PARTITION_BY_COUNTS missing before version 1.2.	cl_uint	This property is followed by a list of compute unit counts terminated with 0 or CL_DEVICE_PARTITION_BY_COUNTS_LIST_END. For each non-zero count *m* in the list, a sub-device is created with *m* compute units in it. The number of non-zero count entries in the list may not exceed CL_DEVICE_PARTITION_MAX_SUB_DEVICES. The total number of compute units specified may not exceed CL_DEVICE_MAX_COMPUTE_UNITS.

Partition Property	Partition Value	Description
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN missing before version 1.2.	cl_device_affinity_domain	Split the device into smaller aggregate devices containing one or more compute units that all share part of a cache hierarchy. The value accompanying this property may be drawn from the following list: CL_DEVICE_AFFINITY_DOMAIN_NUMA - Split the device into sub-devices comprised of compute units that share a NUMA node. CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE - Split the device into sub-devices comprised of compute units that share a level 4 data cache. CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE - Split the device into sub-devices comprised of compute units that share a level 3 data cache. CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE - Split the device into sub-devices comprised of compute units that share a level 2 data cache. CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE - Split the device into sub-devices comprised of compute units that share a level 1 data cache. CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE - Split the device along the next partitionable affinity domain. The implementation shall find the first level along which the device or sub-device may be further subdivided in the order NUMA, L4, L3, L2, L1, and partition the device into sub-devices comprised of compute units that share memory subsystems at this level. The user may determine what happened by calling **clGetDeviceInfo**(CL_DEVICE_PARTITION_TYPE) on the sub-devices.

clCreateSubDevices returns CL_SUCCESS if the partition is created successfully. Otherwise, it returns a NULL value with the following error values returned in *errcode_ret*:

- CL_INVALID_DEVICE
  - if *in_device* is not a valid device
- CL_INVALID_VALUE

- if values specified in *properties* are not valid
- if values specified in *properties* are valid but not are supported by *in_device*
- if *out_devices* is not `NULL` and *num_devices* is less than the number of sub-devices created by the partition scheme

- `CL_DEVICE_PARTITION_FAILED`
  - if the partition name is supported by the implementation but in_device could not be further partitioned
- `CL_INVALID_DEVICE_PARTITION_COUNT`
  - if the partition name specified in *properties* is `CL_DEVICE_PARTITION_BY_COUNTS` and the number of sub-devices requested exceeds `CL_DEVICE_PARTITION_MAX_SUB_DEVICES`
  - if the total number of compute units requested exceeds `CL_DEVICE_MAX_COMPUTE_UNITS` for *in_device*
  - if the number of compute units requested for one or more sub-devices is less than zero
  - if the number of sub-devices requested for one or more sub-devices exceeds `CL_DEVICE_MAX_COMPUTE_UNITS` for *in_device*
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

A few examples that describe how to specify partition properties in *properties* argument to **clCreateSubDevices** are given below:

To partition a device containing 16 compute units into two sub-devices, each containing 8 compute units, pass the following in *properties*:

```
{ CL_DEVICE_PARTITION_EQUALLY, 8,
  0 } // 0 terminates the property list
```

To partition a device with four compute units into two sub-devices with one sub-device containing 3 compute units and the other sub-device 1 compute unit, pass the following in properties argument:

```
{ CL_DEVICE_PARTITION_BY_COUNTS,
    3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END,
  0 } // 0 terminates the property list
```

To split a device along the outermost cache line (if any), pass the following in properties argument:

```
{ CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN,
```

```
    CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE,
  0 } // 0 terminates the property list
```

To retain a device, call the function:

```
// Provided by CL_VERSION_1_2
cl_int clRetainDevice(
    cl_device_id device);
```

clRetainDevice is missing before version 1.2.

- *device* is the OpenCL device to retain.

clRetainDevice increments the *device* reference count if *device* is a valid sub-device created by a call to **clCreateSubDevices**. If *device* is a root level device i.e. a `cl_device_id` returned by **clGetDeviceIDs**, the *device* reference count remains unchanged.

clRetainDevice returns `CL_SUCCESS` if the function is executed successfully or the device is a root-level device. Otherwise, it returns one of the following errors:

- `CL_INVALID_DEVICE`
  - if *device* is not a valid device
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To release a device, call the function:

```
// Provided by CL_VERSION_1_2
cl_int clReleaseDevice(
    cl_device_id device);
```

clReleaseDevice is missing before version 1.2.

- *device* is the OpenCL device to release.

clReleaseDevice decrements the *device* reference count if device is a valid sub-device created by a call to **clCreateSubDevices**. If *device* is a root level device i.e. a `cl_device_id` returned by **clGetDeviceIDs**, the *device* reference count remains unchanged.

clReleaseDevice returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_DEVICE`
  - if *device* is not a valid device
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

After the *device* reference count becomes zero and all the objects attached to *device* (such as command-queues) are released, the *device* object is deleted. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainDevice** causes undefined behavior.

4.4. Contexts

To create an OpenCL context, call the function:

```
// Provided by CL_VERSION_1_0
cl_context clCreateContext(
    const cl_context_properties* properties,
    cl_uint num_devices,
    const cl_device_id* devices,
    void (CL_CALLBACK* pfn_notify)(const char* errinfo, const void* private_info,
size_t cb, void* user_data),
    void* user_data,
    cl_int* errcode_ret);
```

- *properties* specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties, and their default values if not present in *properties*, is described in the Context Properties table. *properties* can be `NULL`, in which case all properties take on their default values.
- *num_devices* is the number of devices specified in the *devices* argument.
- *devices* is a pointer to a list of unique devices returned by **clGetDeviceIDs** or sub-devices created by **clCreateSubDevices** for a platform. [11]
- *pfn_notify* is a callback function that can be registered by the application. This callback function will be used by the OpenCL implementation to report information on errors during context creation as well as errors that occur at runtime in this context. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. If *pfn_notify* is `NULL`, no callback function is registered.
- *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be `NULL`.

- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The parameters to the callback function *pfn_notify* are:

- *errinfo* is a pointer to an error string.
- *private_info* and *cb* represent a pointer to binary data that is returned by the OpenCL implementation that can be used to log additional information helpful in debugging the error.
- *user_data* is a pointer to user supplied data.

Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context.

Table 13. List of supported context creation properties by **clCreateContext**

Context Property	Property Value	Description
`CL_CONTEXT_PLATFORM`	`cl_platform_id`	Specifies the platform to use. Defaults to an implementation-defined platform if not specified.
`CL_CONTEXT_INTEROP_USER_SYNC` missing before version 1.2.	`cl_bool`	Specifies whether the user is responsible for synchronization between OpenCL and other APIs. Please refer to the specific sections in the OpenCL Specification that describe sharing with other APIs for restrictions on using this flag. Defaults to `CL_FALSE` if not specified.
`CL_CONTEXT_ADAPTER_D3D9_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	`IDirect3DDevice9 *`	Specifies an `IDirect3DDevice9` to use for D3D9 interop.
`CL_CONTEXT_ADAPTER_D3D9EX_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	`IDirect3DDeviceEx*`	Specifies an `IDirect3DDevice9Ex` to use for D3D9 interop.
`CL_CONTEXT_ADAPTER_DXVA_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	`IDXVAHD_Device *`	Specifies an `IDXVAHD_Device` to use for DXVA interop.

Context Property	Property Value	Description
CL_CONTEXT_D3D10_DEVICE_KHR provided by the cl_khr_d3d10_sharing extension.	ID3D10Device *	Specifies the ID3D10Device * to use for Direct3D 10 interoperability. The default value is NULL.
CL_CONTEXT_D3D11_DEVICE_KHR provided by the cl_khr_d3d11_sharing extension.	ID3D11Device *	Specifies the ID3D11Device * to use for Direct3D 11 interoperability. The default value is NULL.
CL_GL_CONTEXT_KHR provided by the cl_khr_gl_sharing extension.	OpenGL context handle	OpenGL context to associate the OpenCL context with Defaults to 0 if not specified.
CL_CGL_SHAREGROUP_KHR provided by the cl_khr_gl_sharing extension.	CGL share group handle	CGL share group to associate the OpenCL context with Defaults to 0 if not specified.
CL_EGL_DISPLAY_KHR provided by the cl_khr_gl_sharing extension.	EGL EGLDisplay handle	EGLDisplay an OpenGL context was created with respect to Defaults to EGL_NO_DISPLAY if not specified.
CL_GLX_DISPLAY_KHR provided by the cl_khr_gl_sharing extension.	X handle	X Display an OpenGL context was created with respect to Defaults to None if not specified.
CL_WGL_HDC_KHR provided by the cl_khr_gl_sharing extension.	Windows HDC handle	HDC an OpenGL context was created with respect to Defaults to 0 if not specified.
CL_CONTEXT_MEMORY_INITIALIZE_KHR provided by the cl_khr_initialize_memory extension.	cl_context_memory_initialize_khr	Describes which memory types for the context must be initialized. This is a bit-field, where the following values are currently supported: CL_CONTEXT_MEMORY_INITIALIZE_LOCAL_KHR — Initialize local memory to zeros. CL_CONTEXT_MEMORY_INITIALIZE_PRIVATE_KHR — Initialize private memory to zeros.
CL_CONTEXT_TERMINATE_KHR provided by the cl_khr_terminate_context extension.	cl_bool	Specifies whether the context can be terminated. The default value is CL_FALSE.

Some of the properties specified in the Context Properties table control sharing of OpenCL memory objects with OpenGL buffer, texture, and renderbuffer objects.

Depending on the platform-specific API used to bind OpenGL contexts to the window system, the following properties may be set to identify an OpenGL context:

- When the CGL binding API is supported, the property `CL_CGL_SHAREGROUP_KHR` should be set to a CGLShareGroup handle to a CGL share group object.
- When the EGL binding API is supported, the property `CL_GL_CONTEXT_KHR` should be set to an EGLContext handle to an OpenGL ES or OpenGL context, and the property `CL_EGL_DISPLAY_KHR` should be set to the `EGLDisplay` handle of the display used to create the OpenGL ES or OpenGL context.
- When the GLX binding API is supported, the property `CL_GL_CONTEXT_KHR` should be set to a GLXContext handle to an OpenGL context, and the property `CL_GLX_DISPLAY_KHR` should be set to the `Display` handle of the X Window System display used to create the OpenGL context.
- When the WGL binding API is supported, the property `CL_GL_CONTEXT_KHR` should be set to an HGLRC handle to an OpenGL context, and the property `CL_WGL_HDC_KHR` should be set to the HDC handle of the display used to create the OpenGL context.

Memory objects created in the context so specified may be shared with the specified OpenGL or OpenGL ES context (as well as with any other OpenGL contexts on the share list of that context, according to the description of sharing in the GLX 1.4 and EGL 1.5 specifications, and the WGL documentation for OpenGL implementations on Microsoft Windows), or with the explicitly identified OpenGL share group for CGL. If no OpenGL or OpenGL ES context or share group is specified in the property list, then memory objects may not be shared, and attempts to create such objects will result in a `CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR` error.

OpenCL / OpenGL sharing does not support the `CL_CONTEXT_INTEROP_USER_SYNC` property defined in the Context Properties table. Specifying this property when creating a context with OpenCL / OpenGL sharing will return an appropriate error.

There are a number of cases where error notifications need to be delivered due to an error that occurs outside a context. Such notifications may not be delivered through the *pfn_notify* callback. Where these notifications go is implementation-defined.

clCreateContext returns a valid non-zero context and *errcode_ret* is set to `CL_SUCCESS` if the context is created successfully. Otherwise, it returns a `NULL` value with the following error values returned in *errcode_ret*:

- `CL_INVALID_PLATFORM`
  - if no platform is specified in *properties* and no platform could be selected
  - if the platform specified in *properties* is not a valid platform
- `CL_INVALID_PROPERTY`
  - if a context property name in *properties* is not a supported property name
  - if the value specified for a supported property name is not valid

    - if the same property name is specified more than once
- `CL_INVALID_VALUE`
    - if *devices* is `NULL`
    - if *num_devices* is equal to zero
    - if *pfn_notify* is `NULL` and *user_data* is not `NULL`
- `CL_INVALID_DEVICE`
    - if any device in *devices* is not a valid device
- `CL_DEVICE_NOT_AVAILABLE`
    - if a device in *devices* is currently not available
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

The following errors may be returned if the `cl_khr_dx9_media_sharing` extension is supported:

- `CL_INVALID_DX9_MEDIA_ADAPTER_KHR`
    - if any of the values of the properties `CL_CONTEXT_ADAPTER_D3D9_KHR`, `CL_CONTEXT_ADAPTER_D3D9EX_KHR` or `CL_CONTEXT_ADAPTER_DXVA_KHR` is non-`NULL` and does not specify a valid media adapter with which the *cl_device_ids* against which this context is to be created may interoperate

The following errors may be returned if the `cl_khr_d3d10_sharing` extension is supported:

- `CL_INVALID_D3D10_DEVICE_KHR`
    - if the value of the property `CL_CONTEXT_D3D10_DEVICE_KHR` is non-`NULL` and does not specify a valid Direct3D 10 device with which the *cl_device_ids* against which this context is to be created may interoperate
- `CL_INVALID_OPERATION`
    - if Direct3D 10 interoperability is specified by setting `CL_INVALID_D3D10_DEVICE_KHR` to a non-`NULL` value and interoperability with another graphics API is also specified

The following errors may be returned if the `cl_khr_d3d11_sharing` extension is supported:

- `CL_INVALID_D3D11_DEVICE_KHR`
    - if the value of the property `CL_CONTEXT_D3D11_DEVICE_KHR` is non-`NULL` and does not specify a valid Direct3D 11 device with which the *cl_device_ids* against which this context is to be created may interoperate
- `CL_INVALID_OPERATION`
    - if Direct3D 11 interoperability is specified by setting `CL_INVALID_D3D11_DEVICE_KHR` to a non-`NULL` value and interoperability with another graphics API is also specified

The following errors may be returned if the cl_khr_gl_sharing extension is supported:

- CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR
  - if a context was specified for an OpenGL or OpenGL ES implementation using the EGL, GLX, or WGL binding APIs, as described above; and any of the following conditions hold:
    - The specified display and context properties do not identify a valid OpenGL or OpenGL ES context.
    - The specified context does not support buffer and renderbuffer objects.
    - The specified context is not compatible with the OpenCL context being created (for example, it exists in a physically distinct address space, such as another hardware device; or it does not support sharing data with OpenCL due to implementation restrictions).
- CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR
  - if a share group was specified for a CGL-based OpenGL implementation by setting the property CL_CGL_SHAREGROUP_KHR, and the specified share group does not identify a valid CGL share group object
- CL_INVALID_OPERATION
  - if a context was specified as described above and any of the following conditions hold:
    - A context or share group object was specified for one of CGL, EGL, GLX, or WGL and the OpenGL implementation does not support that window-system binding API.
    - More than one of the properties CL_CGL_SHAREGROUP_KHR, CL_EGL_DISPLAY_KHR, CL_GLX_DISPLAY_KHR, and CL_WGL_HDC_KHR is set to a non-default value.
    - Both of the properties CL_CGL_SHAREGROUP_KHR and CL_GL_CONTEXT_KHR are set to non-default values.
    - Any of the devices specified in the *devices* argument cannot support OpenCL objects which share the data store of an OpenGL object.
- CL_INVALID_PROPERTY
  - if both CL_CONTEXT_INTEROP_USER_SYNC, and any of the properties defined by the cl_khr_gl_sharing extension are defined in *properties*

The following errors may be returned if the cl_khr_terminate_context extension is supported:

- CL_INVALID_PROPERTY
  - if the cl_khr_terminate_context extension is supported and CL_CONTEXT_TERMINATE_KHR is set to CL_TRUE in *properties*, but not all of the devices associated with the context support the ability to support context termination (i.e. CL_DEVICE_TERMINATE_CAPABILITY_CONTEXT_KHR is set for CL_DEVICE_TERMINATE_CAPABILITY_KHR)

It is possible that a device(s) becomes unavailable after a context and command-queues that use this device(s) have been created and commands have been queued to command-queues. In this case the behavior of OpenCL API calls that use this context (and command-queues) are considered to be implementation-defined. The user callback function, if specified when the context is created, can be used to

> record appropriate information in the *errinfo* and *private_info* arguments passed to the callback function when the device becomes unavailable.

To create an OpenCL context from a specific device type [12], call the function:

```
// Provided by CL_VERSION_1_0
cl_context clCreateContextFromType(
    const cl_context_properties* properties,
    cl_device_type device_type,
    void (CL_CALLBACK* pfn_notify)(const char* errinfo, const void* private_info,
size_t cb, void* user_data),
    void* user_data,
    cl_int* errcode_ret);
```

- *properties* specifies a list of context property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list of supported properties, and their default values if not present in *properties*, is described in the Context Properties table. *properties* can be `NULL`, in which case all properties take on their default values.
- *device_type* is a bit-field that identifies the type of device and is described in the Device Types table.
- *pfn_notify* and *user_data* are described in **clCreateContext**.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Only devices that are returned by **clGetDeviceIDs** for *device_type* are used to create the context. The context does not reference any sub-devices that may have been created from these devices.

clCreateContextFromType returns a valid non-zero context and *errcode_ret* is set to `CL_SUCCESS` if the context is created successfully. Otherwise, it returns a `NULL` value with the following error values returned in *errcode_ret*:

- `CL_INVALID_PLATFORM`
  - if no platform is specified in *properties* and no platform could be selected
  - if the platform specified in *properties* is not a valid platform
- `CL_INVALID_PROPERTY`
  - if a context property name in *properties* is not a supported property name
  - if the value specified for a supported property name is not valid
  - if the same property name is specified more than once
- `CL_INVALID_VALUE`
  - if *pfn_notify* is `NULL` and *user_data* is not `NULL`
- `CL_INVALID_DEVICE_TYPE`
  - if *device_type* is not a valid value

- `CL_DEVICE_NOT_AVAILABLE`
  - if no devices that match *device_type* and property values specified in *properties* are currently available
- `CL_DEVICE_NOT_FOUND`
  - if no devices that match *device_type* and property values specified in *properties* were found
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The following errors may be returned if the `cl_khr_dx9_media_sharing` extension is supported:

- `CL_INVALID_DX9_MEDIA_ADAPTER_KHR`
  - if any of the values of the properties `CL_CONTEXT_ADAPTER_D3D9_KHR`, `CL_CONTEXT_ADAPTER_D3D9EX_KHR` or `CL_CONTEXT_ADAPTER_DXVA_KHR` is non-`NULL` and does not specify a valid media adapter with which the *cl_device_ids* against which this context is to be created may interoperate

The following errors may be returned if the `cl_khr_d3d10_sharing` extension is supported:

- `CL_INVALID_D3D10_DEVICE_KHR`
  - if the value of the property `CL_CONTEXT_D3D10_DEVICE_KHR` is non-`NULL` and does not specify a valid Direct3D 10 device with which the *cl_device_ids* against which this context is to be created may interoperate
- `CL_INVALID_OPERATION`
  - if Direct3D 10 interoperability is specified by setting `CL_INVALID_D3D10_DEVICE_KHR` to a non-`NULL` value and interoperability with another graphics API is also specified

The following errors may be returned if the `cl_khr_d3d11_sharing` extension is supported:

- `CL_INVALID_D3D11_DEVICE_KHR`
  - if the value of the property `CL_CONTEXT_D3D11_DEVICE_KHR` is non-`NULL` and does not specify a valid Direct3D 11 device with which the *cl_device_ids* against which this context is to be created may interoperate
- `CL_INVALID_OPERATION`
  - if Direct3D 11 interoperability is specified by setting `CL_INVALID_D3D11_DEVICE_KHR` to a non-`NULL` value and interoperability with another graphics API is also specified

The following errors may be returned if the `cl_khr_gl_sharing` extension is supported:

- `CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR`
  - if a context was specified for an OpenGL or OpenGL ES implementation using the EGL, GLX, or WGL binding APIs, as described for `clCreateContext`; and any of the following conditions

hold:

  - The specified display and context properties do not identify a valid OpenGL or OpenGL ES context.
  - The specified context does not support buffer and renderbuffer objects.
  - The specified context is not compatible with the OpenCL context being created (for example, it exists in a physically distinct address space, such as another hardware device; or it does not support sharing data with OpenCL due to implementation restrictions).

- `CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR`
  - if a share group was specified for a CGL-based OpenGL implementation by setting the property `CL_CGL_SHAREGROUP_KHR`, and the specified share group does not identify a valid CGL share group object
- `CL_INVALID_OPERATION`
  - if a context was specified as described above and any of the following conditions hold:
    - A context or share group object was specified for one of CGL, EGL, GLX, or WGL and the OpenGL implementation does not support that window-system binding API.
    - More than one of the properties `CL_CGL_SHAREGROUP_KHR`, `CL_EGL_DISPLAY_KHR`, `CL_GLX_DISPLAY_KHR`, and `CL_WGL_HDC_KHR` is set to a non-default value.
    - Both of the properties `CL_CGL_SHAREGROUP_KHR` and `CL_GL_CONTEXT_KHR` are set to non-default values.
    - Any of the devices specified in the *devices* argument cannot support OpenCL objects which share the data store of an OpenGL object.
- `CL_INVALID_PROPERTY`
  - if both `CL_CONTEXT_INTEROP_USER_SYNC`, and any of the properties defined by the `cl_khr_gl_sharing` extension are defined in *properties*

To retain a context, call the function:

```
// Provided by CL_VERSION_1_0
cl_int clRetainContext(
    cl_context context);
```

- *context* specifies the OpenCL context to retain.

clRetainContext increments the *context* reference count.

clCreateContext and **clCreateContextFromType** perform an implicit retain. This is very helpful for 3rd party libraries, which typically get a context passed to them by the application. However, it is possible that the application may delete the context without informing the library. Allowing functions to attach to (i.e. retain) and release a context solves the problem of a context being used by a library no longer being valid.

clRetainContext returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns

one of the following errors:

- `CL_INVALID_CONTEXT`
  - if *context* is not a valid OpenCL context
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To release a context, call the function:

```
// Provided by CL_VERSION_1_0
cl_int clReleaseContext(
    cl_context context);
```

- *context* specifies the OpenCL context to release.

clReleaseContext decrements the *context* reference count. After the reference count becomes zero and all the objects attached to *context* (such as memory objects, command-queues) are released, the *context* is deleted. When the *context* is deleted, any commands associated with the context that have not completed may be abnormally terminated. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainContext** causes undefined behavior.

clReleaseContext returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_CONTEXT`
  - if *context* is not a valid OpenCL context
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To terminate all pending work associated with a context and render all data owned by the context invalid, call the function

```
// Provided by cl_khr_terminate_context
cl_int clTerminateContextKHR(
    cl_context context);
```

clTerminateContextKHR is provided by the `cl_khr_terminate_context` extension.

- *context* must be a valid OpenCL context.

It is the responsibility of the application to release all objects associated with the context being terminated.

When a context is terminated:

- The execution status of enqueued commands will be `CL_CONTEXT_TERMINATED_KHR`. Event objects can be queried using **clGetEventInfo**. Event callbacks can be registered and registered event callbacks will be called with *event_command_status* set to `CL_CONTEXT_TERMINATED_KHR`. **clWaitForEvents** will return as immediately for commands associated with event objects specified in event_list. The status of user events can be set. Event objects can be retained and released. **clGetEventProfilingInfo** returns `CL_PROFILING_INFO_NOT_AVAILABLE`.
- The context is considered to be terminated. A callback function registered when the context was created will be called. Only queries, retain and release operations can be performed on the context. All other APIs that use a context as an argument will return `CL_CONTEXT_TERMINATED_KHR`.
- The contents of the memory regions of the memory objects is undefined. Queries, registering a destructor callback, retain and release operations can be performed on the memory objects.
- Once a context has been terminated, all OpenCL API calls that create objects or enqueue commands will return `CL_CONTEXT_TERMINATED_KHR`. APIs that release OpenCL objects will continue to operate as though **clTerminateContextKHR** was not called.
- The behavior of callbacks will remain unchanged, and will report appropriate error, if executing after termination of context. This behavior is similar to enqueued commands, after the command-queue has become invalid.

clTerminateContextKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_CONTEXT`
  - if *context* is not a valid OpenCL context
- `CL_CONTEXT_TERMINATED_KHR`
  - if *context* has already been terminated
- `CL_INVALID_OPERATION`
  - if *context* was not created with `CL_CONTEXT_TERMINATE_KHR` set to `CL_TRUE`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

An implementation that supports this extension must be able to terminate commands currently executing on devices or queued across all command-queues associated with the context that is being terminated. The implementation cannot implement this extension by waiting for currently executing (or queued) commands to finish execution on devices associated with this context (i.e.

doing a **clFinish**).

To query information about a context, call the function:

```
// Provided by CL_VERSION_1_0
cl_int clGetContextInfo(
    cl_context context,
    cl_context_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *context* specifies the OpenCL context being queried.
- *param_name* is an enumeration constant that specifies the information to query.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Context Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

The list of supported *param_name* values and the information returned in *param_value* by **clGetContextInfo** is described in the Context Queries table.

Table 14. List of supported param_names by **clGetContextInfo**

Context Info	Return Type	Description
`CL_CONTEXT_REFERENCE_COUNT` [13]	`cl_uint`	Return the *context* reference count.
`CL_CONTEXT_NUM_DEVICES` missing before version 1.1.	`cl_uint`	Return the number of devices in *context*.
`CL_CONTEXT_DEVICES`	`cl_device_id`[]	Return the list of devices and sub-devices in *context*.

Context Info	Return Type	Description
CL_CONTEXT_PROPERTIES	cl_context_properties[]	Return the properties argument specified in **clCreateContext** or **clCreateContextFromType**. If the *properties* argument specified in **clCreateContext** or **clCreateContextFromType** used to create *context* was not NULL, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If the *properties* argument specified in **clCreateContext** or **clCreateContextFromType** used to create *context* was NULL, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.
CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR provided by the cl_khr_d3d10_sharing extension.	cl_bool	Returns CL_TRUE if Direct3D 10 resources created as shared by setting *MiscFlags* to include D3D10_RESOURCE_MISC_SHARED will perform faster when shared with OpenCL, compared with resources which have not set this flag. Otherwise returns CL_FALSE.
CL_CONTEXT_D3D11_PREFER_SHARED_RESOURCES_KHR provided by the cl_khr_d3d11_sharing extension.	cl_bool	Returns CL_TRUE if Direct3D 11 resources created as shared by setting *MiscFlags* to include D3D11_RESOURCE_MISC_SHARED will perform faster when shared with OpenCL, compared with resources which have not set this flag. Otherwise returns CL_FALSE.

clGetContextInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT
  - if *context* is not a valid context
- CL_INVALID_VALUE
  - if *param_name* is not one of the supported values
  - if the size in bytes specified by *param_value_size* is less than the size of the return type specified in the Context Queries table and *param_value* is not NULL
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY

- if there is a failure to allocate resources required by the OpenCL implementation on the host

To register a callback function with a context that is called when the context is destroyed, call the function

```
// Provided by CL_VERSION_3_0
cl_int clSetContextDestructorCallback(
    cl_context context,
    void (CL_CALLBACK* pfn_notify)(cl_context context, void* user_data),
    void* user_data);
```

clSetContextDestructorCallback is missing before version 3.0.

- *context* specifies the OpenCL context to register the callback to.
- *pfn_notify* is the callback function to register. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:
  - *context* is the OpenCL context being deleted. When the callback function is called by the implementation, this context is no longer valid. *context* is only provided for reference purposes.
  - *user_data* is a pointer to user-supplied data.
- *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be `NULL`.

Each call to **clSetContextDestructorCallback** registers the specified callback function on a destructor callback stack associated with *context*. The registered callback functions are called in the reverse order in which they were registered. If a context callback function was specified when *context* was created, it will not be called after any context destructor callback is called. Therefore, the context destructor callback provides a mechanism for an application to safely re-use or free any *user_data* specified for the context callback function when *context* was created.

clSetContextDestructorCallback returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_CONTEXT`
  - if *context* is not a valid context
- `CL_INVALID_VALUE`
  - if *pfn_notify* is `NULL`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

4.5. Extension Function Pointers

Extension function pointers for a given platform can be obtained with the function:

```
// Provided by CL_VERSION_1_2
void* clGetExtensionFunctionAddressForPlatform(
    cl_platform_id platform,
    const char* func_name);
```

clGetExtensionFunctionAddressForPlatform is missing before version 1.2.

Or, with the function:

```
// Provided by CL_VERSION_1_0
void* clGetExtensionFunctionAddress(
    const char* func_name);
```

clGetExtensionFunctionAddress is deprecated by version 1.2.

The returned function pointer should be cast to a function pointer type matching the extension function's definition defined in the appropriate extension specification and header file.

A return value of `NULL` indicates that *platform* is not a valid platform or that the specified extension function does not exist for the platform.

A non-`NULL` return value from **clGetExtensionFunctionAddressForPlatform** does not guarantee that an extension function is actually supported by the platform. The application must also query `CL_PLATFORM_EXTENSIONS` or `CL_DEVICE_EXTENSIONS` to determine if an extension is supported by an OpenCL platform or device.

Since there is no way to qualify the **clGetExtensionFunctionAddressForPlatform** query with a device, the function pointer returned must work for all implementations of and extension on different devices for a platform. The behavior of calling a device extension function on a device not supporting that extension is undefined.

clGetExtensionFunctionAddressForPlatform may not be used to query for core (non-extension) functions in OpenCL. For extension functions queried using **clGetExtensionFunctionAddressForPlatform**, implementations may also choose to export those functions statically from the object libraries implementing those functions, however, portable applications cannot rely on this behavior.

[1] The platform profile returns the profile that is implemented by the OpenCL framework. If the platform profile returned is FULL_PROFILE, the OpenCL framework will support devices that are FULL_PROFILE and may also support devices that are EMBEDDED_PROFILE. The compiler must be available for all devices i.e. `CL_DEVICE_COMPILER_AVAILABLE` is `CL_TRUE`. If the platform profile returned is EMBEDDED_PROFILE, then devices that are only EMBEDDED_PROFILE are supported.

[2] A null terminated string is returned by OpenCL query function calls if the return type of the information being queried is a `char[]`.

[3] The OpenCL specification does not describe the order of precedence for error codes returned by API calls.

[4] **clGetDeviceIDs** may return all or a subset of the actual physical devices present in the platform and that match *device_type*.

[5] OpenCL adopters must report a valid vendor ID for their implementation. If there is no valid PCI vendor ID defined for the physical device, implementations must obtain a Khronos vendor ID. This is a unique identifier greater than the largest PCI vendor ID (`0x10000`) and is representable by a `cl_uint`. Khronos vendor IDs are synchronized across APIs by utilizing Vulkan's vk.xml as the central Khronos vendor ID registry. An ID must be reserved here prior to use in OpenCL, regardless of whether a vendor implements Vulkan. Only once the ID has been allotted may it be exposed to OpenCL by proposing a merge request against cl.xml, in the `main` branch of the OpenCL-Docs project. The merge must define a new enumerant by adding an `<enum>` tag to the `cl_khronos_vendor_id <enums>` tag, with the `<value>` attribute set as the acquired Khronos vendor ID. The `<name>` attribute must identify the vendor/adopter, and be of the form `CL_KHRONOS_VENDOR_ID_<vendor>`.

[6] A kernel that uses an image argument with the write_only or read_write image qualifier may result in additional read_only images resources being created internally by an implementation. The internally created read_only image resources will count against the max supported read image arguments given by `CL_DEVICE_MAX_READ_IMAGE_ARGS`. Enqueuing a kernel that requires more images than the implementation can support will result in a `CL_OUT_OF_RESOURCES` error being returned.

[7] The optional rounding modes should be included as a device capability only if it is supported natively. All explicit conversion functions with specific rounding modes must still operate correctly.

[8] When the query for `CL_DEVICE_HOST_UNIFIED_MEMORY` is `CL_TRUE`, allocating OpenCL memory will likely reduce the amount of host memory available to the system. Likewise, allocating host memory will likely reduce the amount of memory available to OpenCL. If the memory described by `CL_DEVICE_GLOBAL_MEM_SIZE` is primarily host memory, such as for CPUs, integrated GPUs, and other devices with a relatively small amount of dedicated device memory, then the device should return `CL_TRUE` for `CL_DEVICE_HOST_UNIFIED_MEMORY`.

[9] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[10] Note that this flag does not provide meaning for atomic memory operations, but only for atomic fence operations in certain circumstances, refer to the Memory Scope section of the OpenCL C specification.

[11] Duplicate devices specified in *devices* are ignored.

[12] **clCreateContextFromType** may create a context for all or a subset of the actual physical devices present in the platform that match *device_type*.

[13] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

Chapter 5. The OpenCL Runtime

In this section we describe the API calls that manage OpenCL objects such as command-queues, memory objects, program objects, kernel objects for kernel functions in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object.

5.1. Command-Queues

OpenCL objects such as memory, program and kernel objects are created using a context. Operations on these objects are performed using a command-queue. The command-queue can be used to queue a set of operations (referred to as commands) in order. Having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization. Note that this should work as long as these objects are not being shared. Sharing of objects across multiple command-queues will require the application to perform appropriate synchronization. This is described in Shared OpenCL Objects

To create a host or device command-queue on a specific device, call the function

```
// Provided by CL_VERSION_2_0
cl_command_queue clCreateCommandQueueWithProperties(
    cl_context context,
    cl_device_id device,
    const cl_queue_properties* properties,
    cl_int* errcode_ret);
```

clCreateCommandQueueWithProperties is missing before version 2.0.

or the equivalent

```
// Provided by cl_khr_create_command_queue
cl_command_queue clCreateCommandQueueWithPropertiesKHR(
    cl_context context,
    cl_device_id device,
    const cl_queue_properties_khr* properties,
    cl_int* errcode_ret);
```

clCreateCommandQueueWithPropertiesKHR is provided by the `cl_khr_create_command_queue` extension.

- *context* must be a valid OpenCL context.
- *device* must be a device or sub-device associated with *context*. It can either be in the list of devices and sub-devices specified when *context* is created using **clCreateContext** or be a root device with the same device type as specified when *context* is created using **clCreateContextFromType**.

- *properties* specifies a list of properties for the command-queue and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in the table below. If a supported property and its value is not specified in *properties*, its default value will be used. *properties* can be `NULL` in which case the default values for supported command-queue properties will be used.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Table 15. List of supported queue creation properties by **clCreateCommandQueueWithProperties**

Queue Property	Property Value	Description
`CL_QUEUE_PROPERTIES`	`cl_command_queue_properties`	This is a bitfield and can be set to a combination of the following values: `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` - Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order. `CL_QUEUE_PROFILING_ENABLE` - Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled. `CL_QUEUE_ON_DEVICE` - Indicates that this is a device queue. If `CL_QUEUE_ON_DEVICE` is set, `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` [1] must also be set. missing before version 2.0. `CL_QUEUE_ON_DEVICE_DEFAULT` [2] - indicates that this is the default device queue. This can only be used with `CL_QUEUE_ON_DEVICE`. missing before version 2.0. If `CL_QUEUE_PROPERTIES` is not specified an in-order host command-queue is created for the specified device

Queue Property	Property Value	Description
CL_QUEUE_SIZE missing before version 2.0.	cl_uint	Specifies the size of the device queue in bytes. This can only be specified if CL_QUEUE_ON_DEVICE is set in CL_QUEUE_PROPERTIES. This must be a value ≤ CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE. For best performance, this should be ≤ CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE. If CL_QUEUE_SIZE is not specified, the device queue is created with CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE as the size of the queue.
CL_QUEUE_PRIORITY_KHR provided by the cl_khr_priority_hints extension.	cl_queue_priority_khr	Specifies a priority hint for command queues belonging to the same OpenCL context. NOTE: Refer to the user guide associated with each implementation supporting this extension for its priority behavior guarantees, if any. CL_QUEUE_PRIORITY_HIGH_KHR - Indicates command queues should have high priority. CL_QUEUE_PRIORITY_MED_KHR - Indicates command queues should have medium priority. CL_QUEUE_PRIORITY_LOW_KHR - Indicates command queues should have low priority. If CL_QUEUE_PRIORITY_KHR is not specified, the default priority CL_QUEUE_PRIORITY_MED_KHR is used.

Queue Property	Property Value	Description
CL_QUEUE_THROTTLE_KHR provided by the cl_khr_throttle_hints extension.	cl_queue_throttle_khr	Specifies a throttle hint for a command queue. NOTE: Refer to the user guide associated with each implementation supporting this extension for its throttling behavior guarantees, if any. CL_QUEUE_THROTTLE_HIGH_KHR - Indicates the queue should execute at full throttle, which may consume more energy. CL_QUEUE_THROTTLE_MED_KHR - Indicates normal throttling behavior. CL_QUEUE_THROTTLE_LOW_KHR - Indicates the queue should execute at low throttle, optimized for lowest energy consumption. If CL_QUEUE_THROTTLE_KHR is not specified, the default priority CL_QUEUE_THROTTLE_MED_KHR is used.

clCreateCommandQueueWithProperties returns a valid non-zero command-queue and *errcode_ret* is set to CL_SUCCESS if the command-queue is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT
  - if *context* is not a valid context
- CL_INVALID_DEVICE
  - if *device* is not a valid device
  - if *device* is not associated with *context*
- CL_INVALID_VALUE
  - if values specified in *properties* are not valid
- CL_INVALID_QUEUE_PROPERTIES
  - if values specified in *properties* are valid but are not supported by *device*
  - if the cl_khr_priority_hints extension is supported, the CL_QUEUE_PRIORITY_KHR property is specified, and the queue is a CL_QUEUE_ON_DEVICE
  - if the cl_khr_throttle_hints extension is supported, the CL_QUEUE_THROTTLE_KHR property is specified, and the queue is a CL_QUEUE_ON_DEVICE
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device

- CL_OUT_OF_HOST_MEMORY
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

To create a host command-queue on a specific device, call the function

```
// Provided by CL_VERSION_1_0
cl_command_queue clCreateCommandQueue(
    cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties,
    cl_int* errcode_ret);
```

clCreateCommandQueue is deprecated by version 2.0.

- *context* must be a valid OpenCL context.
- *device* must be a device or sub-device associated with *context*. It can either be in the list of devices and sub-devices specified when *context* is created using **clCreateContext** or be a root device with the same device type as specified when *context* is created using **clCreateContextFromType**.
- *properties* specifies a list of properties for the command-queue. This is a bit-field and the supported properties are described in the table below. Only command-queue properties specified in this table can be used, otherwise the value specified in *properties* is considered to be not valid. *properties* can be 0 in which case the default values for supported command-queue properties will be used.

Table 16. List of supported `cl_command_queue_property` *values by* **clCreateCommandQueue**

Command-Queue Properties	Description
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE	Determines whether the commands queued in the command-queue are executed in-order or out-of-order. If set, the commands in the command-queue are executed out-of-order. Otherwise, commands are executed in-order.
CL_QUEUE_PROFILING_ENABLE	Enable or disable profiling of commands in the command-queue. If set, the profiling of commands is enabled. Otherwise profiling of commands is disabled.

- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

clCreateCommandQueue returns a valid non-zero command-queue and *errcode_ret* is set to CL_SUCCESS if the command-queue is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT

    - if *context* is not a valid context
- `CL_INVALID_DEVICE`
    - if *device* is not a valid device
    - if *device* is not associated with *context*
- `CL_INVALID_VALUE`
    - if values specified in *properties* are not valid
- `CL_INVALID_QUEUE_PROPERTIES`
    - if values specified in *properties* are valid but are not supported by *device*
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

To replace the default command-queue on a device, call the function

```
// Provided by CL_VERSION_2_1
cl_int clSetDefaultDeviceCommandQueue(
    cl_context context,
    cl_device_id device,
    cl_command_queue command_queue);
```

clSetDefaultDeviceCommandQueue is missing before version 2.1.

- *context* is the OpenCL context used to create *command_queue*.
- *device* is a valid OpenCL device associated with *context*.
- *command_queue* specifies a command-queue object which replaces the default device command-queue

clSetDefaultDeviceCommandQueue may be used to replace a default device command-queue created with **clCreateCommandQueueWithProperties** and the `CL_QUEUE_ON_DEVICE_DEFAULT` flag.

clSetDefaultDeviceCommandQueue returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_CONTEXT`
    - if *context* is not a valid context
- `CL_INVALID_DEVICE`
    - if *device* is not a valid device
    - if *device* is not associated with *context*
- `CL_INVALID_OPERATION`

    - if *device* does not support a replaceable default on-device queue
- `CL_INVALID_COMMAND_QUEUE`
    - if *command_queue* is not a valid command-queue for *device*
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

To retain a command-queue, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainCommandQueue(
    cl_command_queue command_queue);
```

- *command_queue* specifies the command-queue to be retained.

The *command_queue* reference count is incremented.

clCreateCommandQueueWithProperties and **clCreateCommandQueue** perform an implicit retain. This is very helpful for 3rd party libraries, which typically get a command-queue passed to them by the application. However, it is possible that the application may delete the command-queue without informing the library. Allowing functions to attach to (i.e. retain) and release a command-queue solves the problem of a command-queue being used by a library no longer being valid.

clRetainCommandQueue returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
    - if *command_queue* is not a valid command-queue
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

To release a command-queue, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseCommandQueue(
    cl_command_queue command_queue);
```

- *command_queue* specifies the command-queue to be released.

The *command_queue* reference count is decremented.

After the *command_queue* reference count becomes zero and all commands queued to *command_queue* have finished (eg. kernel-instances, memory object updates etc.), the command-queue is deleted.

clReleaseCommandQueue performs an implicit flush to issue any previously queued OpenCL commands in *command_queue*. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainCommandQueue** causes undefined behavior.

clReleaseCommandQueue returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid command-queue
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To query information about a command-queue, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetCommandQueueInfo(
    cl_command_queue command_queue,
    cl_command_queue_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *command_queue* specifies the command-queue being queried.
- *param_name* specifies the information to query.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Command-Queue Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

The list of supported *param_name* values and the information returned in *param_value* by **clGetCommandQueueInfo** is described in the Command-Queue Queries table.

Table 17. List of supported param_names by **clGetCommandQueueInfo**

Queue Info	Return Type	Description
CL_QUEUE_CONTEXT	cl_context	Return the context specified when the command-queue is created.
CL_QUEUE_DEVICE	cl_device_id	Return the device specified when the command-queue is created.
CL_QUEUE_REFERENCE_COUNT [3]	cl_uint	Return the command-queue reference count.
CL_QUEUE_PROPERTIES	cl_command_queue_properties	Return the currently specified properties for the command-queue. These properties are specified by the value associated with the CL_QUEUE_PROPERTIES passed in *properties* argument in **clCreateCommandQueueWithProperties**, or the value of the *properties* argument in **clCreateCommandQueue**.
CL_QUEUE_PROPERTIES_ARRAY missing before version 3.0.	cl_queue_properties[]	Return the properties argument specified in **clCreateCommandQueueWithProperties**. If the *properties* argument specified in **clCreateCommandQueueWithProperties** used to create *command_queue* was not NULL, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If *command_queue* was created using **clCreateCommandQueue**, or if the *properties* argument specified in **clCreateCommandQueueWithProperties** was NULL, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.
CL_QUEUE_SIZE missing before version 2.0.	cl_uint	Return the size of the device command-queue. To be considered valid for this query, *command_queue* must be a device command-queue.
CL_QUEUE_DEVICE_DEFAULT missing before version 2.1.	cl_command_queue	Return the current default command-queue for the underlying device.

clGetCommandQueueInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE
  - if *command_queue* is not a valid command-queue

    - if *command_queue* is not a valid command-queue for *param_name*
- `CL_INVALID_VALUE`
    - if *param_name* is not one of the supported values
    - if the size in bytes specified by *param_value_size* is less than the size of the return type specified in the Command-Queue Queries table and *param_value* is not `NULL`
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enable or disable the properties of a command-queue, call the function

```
// Provided by CL_VERSION_1_0
cl_int clSetCommandQueueProperty(
    cl_command_queue command_queue,
    cl_command_queue_properties properties,
    cl_bool enable,
    cl_command_queue_properties* old_properties);
```

clSetCommandQueueProperty is deprecated by version 1.1.

- *command_queue* specifies the command-queue being modified.
- *properties* specifies the new list of properties for the command-queue. This is a bit-field and the supported properties are described in the Command-Queue Properties table for **clCreateCommandQueue**. Only command-queue properties specified in this table can be used, otherwise the value specified in *properties* is considered to be not valid.
- *enable* determines whether the values specified by *properties* are enabled (if *enable* is `CL_TRUE`) or disabled (if *enable* is `CL_FALSE`) for the command-queue.
- *old_properties* returns the command-queue properties before they were changed by **clSetCommandQueueProperty**. If *old_properties* is `NULL`, it is ignored.

clSetCommandQueueProperty may unconditionally return an error if no devices in the context associated with *command_queue* support modifying the properties of a command-queue. Support for modifying the properties of a command-queue is required only for OpenCL 1.0 devices.

clSetCommandQueueProperty returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
    - if *command_queue* is not a valid command-queue
- `CL_INVALID_OPERATION`
    - if no devices in the context associated with *command_queue* support modifying the

properties of a command-queue

- `CL_INVALID_VALUE`
  - if values specified in *properties* are not valid
- `CL_INVALID_QUEUE_PROPERTIES`
  - if values specified in *properties* are valid but are not supported by the device associated with *command_queue*
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.2. Buffer Objects

A *buffer* object stores a one-dimensional collection of elements. Elements of a *buffer* object can be a scalar data type (such as an int, float), vector data type, or a user-defined structure.

5.2.1. Creating Buffer Objects

A **buffer object** may be created using the function

```
// Provided by CL_VERSION_1_0
cl_mem clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void* host_ptr,
    cl_int* errcode_ret);
```

A **buffer object** may also be created with additional properties using the function

```
// Provided by CL_VERSION_3_0
cl_mem clCreateBufferWithProperties(
    cl_context context,
    const cl_mem_properties* properties,
    cl_mem_flags flags,
    size_t size,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateBufferWithProperties is missing before version 3.0.

- *context* is a valid OpenCL context used to create the buffer object.

- *properties* is an optional list of properties for the buffer object and their corresponding values. The list is terminated with the special property `0`. If no properties are required, *properties* may be `NULL`. OpenCL does not define any optional properties for buffers, but extensions may define properties as described in the List of supported buffer creation properties.
- *flags* is a bit-field that is used to specify allocation and usage information about the buffer memory object being created and is described in the supported memory flag values table.
- *size* is the size in bytes of the buffer memory object to be allocated.
- *host_ptr* is a pointer to the buffer data that may already be allocated by the application. The size of the buffer that *host_ptr* points to must be greater than or equal to *size* bytes.
- *errcode_ret* may return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The alignment requirements for data stored in buffer objects are described in Alignment of Application Data Types.

If **clCreateBuffer** or **clCreateBufferWithProperties** is called with `CL_MEM_USE_HOST_PTR` set in its *flags* argument, the contents of the memory pointed to by *host_ptr* at the time of the **clCreateBuffer** call define the initial contents of the buffer object.

If **clCreateBuffer** or **clCreateBufferWithProperties** is called with a pointer returned by **clSVMAlloc** as its *host_ptr* argument, and `CL_MEM_USE_HOST_PTR` is set in its *flags* argument, **clCreateBuffer** or **clCreateBufferWithProperties** will succeed and return a valid non-zero buffer object as long as the *size* argument is no larger than the *size* argument passed in the original **clSVMAlloc** call. The new buffer object returned has the shared memory as the underlying storage. Locations in the buffers underlying shared memory can be operated on using atomic operations to the devices level of support as defined in the memory model.

Table 18. List of supported buffer creation properties

Property	Property Value	Description
`CL_MEM_DEVICE_HANDLE_LIST_KHR` provided by the `cl_khr_external_memory` extension.	`cl_device_id`[]	Specifies the list of OpenCL devices (terminated with `CL_MEM_DEVICE_HANDLE_LIST_END_KHR`) to associate with the external memory handle.

Property	Property Value	Description
`CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT` provided by the `cl_ext_buffer_device_address` extension.	`cl_bool`	When set to `CL_TRUE`, specifies that the buffer must have a single fixed device-side address for its lifetime, and the address can be queried via **clGetMemObjectInfo**. Each device in the context can have their own (fixed) device-side address and a copy of the created buffer which are synchronized implicitly by the runtime. The flag might imply that the buffer will be "pinned" permanently to a device's memory, but might not be necessarily so, as long as the address range of the buffer remains constant. The device addresses of sub-buffers derived from `CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT` allocated buffers can be computed by adding the sub-buffer origin to the device-specific start address. If the device supports SVM and **clCreateBufferWithProperties** is called with a pointer returned by **clSVMAlloc** as its *host_ptr* argument, and `CL_MEM_USE_HOST_PTR` is set in its *flags* argument, the device-side address is guaranteed to match the *host_ptr*.

If `CL_MEM_DEVICE_HANDLE_LIST_KHR` is not specified as part of *properties*, the memory object created by **clCreateBufferWithProperties** or **clCreateImageWithProperties** is by default associated with all devices in the *context*.

The properties used to create a buffer from an external memory handle are described for the corresponding extensions. When a buffer is created from an external memory handle, the *flags* used to specify usage information for the buffer must not include `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR`, or `CL_MEM_COPY_HOST_PTR`, and the *host_ptr* argument must be `NULL`.

If `CL_MEM_IMMUTABLE_EXT` is set in *flags*, the buffer created is an *immutable memory object* that cannot be modified by the implementation after creation, and one and only one of the following conditions must be met:

- Either `CL_MEM_COPY_HOST_PTR` or `CL_MEM_USE_HOST_PTR` must be set in *flags*.
- An external memory handle must be included in *properties*.

clCreateBuffer and **clCreateBufferWithProperties** returns a valid non-zero buffer object and *errcode_ret* is set to `CL_SUCCESS` if the buffer object is created successfully. Otherwise, they return a `NULL` value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT
  - if *context* is not a valid context
- CL_INVALID_PROPERTY
  - if a property name in *properties* is not a supported property name
  - if the value specified for a supported property name is not valid
  - if the same property name is specified more than once
  - if *properties* does not include a supported external memory handle and CL_MEM_DEVICE_HANDLE_LIST_KHR is specified as part of *properties*
  - if *properties* includes more than one external memory handle
- CL_INVALID_DEVICE
  - if a device identified by the property CL_MEM_DEVICE_HANDLE_LIST_KHR is not a valid device or is not associated with *context*
  - if a device identified by property CL_MEM_DEVICE_HANDLE_LIST_KHR cannot import the requested external memory object type
  - if CL_MEM_DEVICE_HANDLE_LIST_KHR is not specified as part of *properties* and one or more devices in *context* cannot import the requested external memory object type
- CL_INVALID_VALUE
  - if values specified in *flags* are not valid as defined in the Memory Flags table
  - if *properties* includes a supported external memory handle and *flags* includes CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, or CL_MEM_COPY_HOST_PTR
  - if CL_MEM_IMMUTABLE_EXT is set in *flags* and CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, or CL_MEM_HOST_WRITE_ONLY is set in *flags*
  - if CL_MEM_IMMUTABLE_EXT is set in *flags* and none of the following conditions are met:
    - CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR is set in *flags*
    - *properties* includes an external memory handle
- CL_INVALID_BUFFER_SIZE
  - if *size* is zero and *properties* does not include an AHardwareBuffer external memory handle
  - if *size* is non-zero and *properties* includes an AHardwareBuffer external memory handle
  - if *size* is greater than CL_DEVICE_MAX_MEM_ALLOC_SIZE for all devices in *context*
  - if CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR is set in *flags*, and *host_ptr* is a pointer returned by **clSVMAlloc**, and *size* is greater than the size passed to **clSVMAlloc**
- CL_INVALID_HOST_PTR
  - if *host_ptr* is NULL, and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags*
  - if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*
  - if *properties* includes a supported external memory handle and *host_ptr* is not NULL
- CL_INVALID_OPERATION
  - if *properties* includes an AHardwareBuffer external memory handle and the

AHardwareBuffer format is not AHARDWAREBUFFER_FORMAT_BLOB

- if *properties* includes CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT and there are no devices in the context that support the cl_ext_buffer_device_address extension

- CL_MEM_OBJECT_ALLOCATION_FAILURE
  - if there is a failure to allocate memory for the buffer object
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Table 19. List of supported memory flag values

Memory Flags	Description
CL_MEM_READ_WRITE	This flag specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	This flag specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE and CL_MEM_WRITE_ONLY are mutually exclusive.
CL_MEM_READ_ONLY	This flag specifies that the memory object is a readonly memory object when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive.

Memory Flags	Description
CL_MEM_USE_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by host_ptr as the storage bits for the memory object. The contents of the memory pointed to by host_ptr at the time of the **clCreateBuffer**, **clCreateBufferWithProperties**, **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D** call define the initial contents of the memory object. OpenCL implementations are allowed to cache the contents pointed to by host_ptr in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same host_ptr or from overlapping host or SVM regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	This flag specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.

Memory Flags	Description
CL_MEM_COPY_HOST_PTR	This flag is valid only if host_ptr is not NULL. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by host_ptr. The implementation will copy the memory immediately and host_ptr is available for reuse by the application when the **clCreateBuffer**, **clCreateBufferWithProperties**, **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D** operation returns. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the cl_mem object allocated using host-accessible (e.g. PCIe) memory.
CL_MEM_HOST_WRITE_ONLY missing before version 1.2.	This flag specifies that the host will only write to the memory object (using OpenCL APIs that enqueue a write or a map for write). This can be used to optimize write access from the host (e.g. enable write-combined allocations for memory objects for devices that communicate with the host over a system bus such as PCIe).
CL_MEM_HOST_READ_ONLY missing before version 1.2.	This flag specifies that the host will only read the memory object (using OpenCL APIs that enqueue a read or a map for read). CL_MEM_HOST_WRITE_ONLY and CL_MEM_HOST_READ_ONLY are mutually exclusive.
CL_MEM_HOST_NO_ACCESS missing before version 1.2.	This flag specifies that the host will not read or write the memory object. CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_READ_ONLY and CL_MEM_HOST_NO_ACCESS are mutually exclusive.

Memory Flags	Description
`CL_MEM_KERNEL_READ_AND_WRITE` missing before version 2.0.	This flag is only used by **clGetSupportedImageFormats** to query image formats that may be both read from and written to by the same kernel instance. To create a memory object that may be read from and written to use `CL_MEM_READ_WRITE`.
`CL_MEM_IMMUTABLE_EXT` provided by the `cl_ext_immutable_memory_objects` extension.	This flag specifies that the contents of the memory object cannot be altered by the implementation after creation.

To create a new buffer object (referred to as a sub-buffer object) from an existing buffer object, call the function

```
// Provided by CL_VERSION_1_1
cl_mem clCreateSubBuffer(
    cl_mem buffer,
    cl_mem_flags flags,
    cl_buffer_create_type buffer_create_type,
    const void* buffer_create_info,
    cl_int* errcode_ret);
```

clCreateSubBuffer is missing before version 1.1.

- *buffer* must be a valid buffer object and cannot be a sub-buffer object.
- *flags* is a bit-field that is used to specify allocation and usage information about the sub-buffer memory object being created and is described in the Memory Flags table. If the `CL_MEM_READ_WRITE`, `CL_MEM_READ_ONLY`, `CL_MEM_IMMUTABLE_EXT`, or `CL_MEM_WRITE_ONLY` values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *buffer*. The `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR`, and `CL_MEM_COPY_HOST_PTR` values cannot be specified in *flags* but are inherited from the corresponding memory access qualifiers associated with *buffer*. If `CL_MEM_COPY_HOST_PTR` is specified in the memory access qualifier values associated with *buffer* it does not imply any additional copies when the sub-buffer is created from *buffer*. If the `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`, or `CL_MEM_HOST_NO_ACCESS` values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *buffer*.
- *buffer_create_type* and *buffer_create_info* describe the type of buffer object to be created. The list of supported values for *buffer_create_type* and corresponding descriptor that *buffer_create_info* points to is described in the SubBuffer Attributes table.

Table 20. List of supported buffer creation types by **clCreateSubBuffer**

Buffer Creation Type	Description
`CL_BUFFER_CREATE_TYPE_REGION` missing before version 1.1.	Create a buffer object that represents a specific region in *buffer*. *buffer_create_info* is a pointer to a `cl_buffer_region` structure specifying a region of the buffer. If *buffer* is created with `CL_MEM_USE_HOST_PTR`, the *host_ptr* associated with the buffer object returned is *host_ptr* + *origin*. The buffer object returned references the data store allocated for buffer and points to the region specified by *buffer_create_info* in this data store.

clCreateSubBuffer returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors in *errcode_ret*:

- `CL_INVALID_MEM_OBJECT`
  - if *buffer* is not a valid buffer object
  - if *buffer* is a sub-buffer object
- `CL_INVALID_VALUE`
  - if *buffer* was created with `CL_MEM_WRITE_ONLY` and *flags* specifies `CL_MEM_READ_WRITE` or `CL_MEM_READ_ONLY`
  - if *buffer* was created with `CL_MEM_READ_ONLY` and *flags* specifies `CL_MEM_READ_WRITE` or `CL_MEM_WRITE_ONLY`
  - if *buffer* was created with `CL_MEM_HOST_WRITE_ONLY` and *flags* specify `CL_MEM_HOST_READ_ONLY`
  - if *buffer* was created with `CL_MEM_HOST_READ_ONLY` and *flags* specify `CL_MEM_HOST_WRITE_ONLY`
  - if *buffer* was created with `CL_MEM_HOST_NO_ACCESS` and *flags* specify `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_WRITE_ONLY`
  - if *buffer* was created with `CL_MEM_IMMUTABLE_EXT` and *flags* specifies `CL_MEM_READ_WRITE`, `CL_MEM_WRITE_ONLY`, or `CL_MEM_HOST_WRITE_ONLY`
  - if *flags* specifies `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR` or `CL_MEM_COPY_HOST_PTR`
  - if the value specified in *buffer_create_type* is not valid
  - if *buffer_create_info* is `NULL`
  - if a value specified in *buffer_create_info* for a given *buffer_create_type* is not valid
  - if the region specified by the `cl_buffer_region` structure passed in *buffer_create_info* is out of bounds in *buffer*
- `CL_INVALID_BUFFER_SIZE`

- if the *size* field of the `cl_buffer_region` structure passed in *buffer_create_info* is zero

- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if there are no devices in *context* associated with *buffer* for which the *origin* field of the `cl_buffer_region` structure passed in *buffer_create_info* is aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value. This error code is missing before version 1.1.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with the sub-buffer object
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Concurrent reading from, writing to and copying between both a buffer object and its sub-buffer object(s) is undefined. Concurrent reading from, writing to and copying between overlapping sub-buffer objects created with the same buffer object is undefined. Only reading from both a buffer object and its sub-buffer objects or reading from multiple overlapping sub-buffer objects is defined.

The `cl_buffer_region` structure specifies a region of a buffer object:

```
// Provided by CL_VERSION_1_0
typedef struct cl_buffer_region {
    size_t    origin;
    size_t    size;
} cl_buffer_region;
```

- *origin* is the offset in bytes of the region.
- *size* is the size in bytes of the region.

Constraints on the values of *origin* and *size* are specified for the **clCreateSubBuffer** function to which this structure is passed.

5.2.2. Reading, Writing and Copying Buffer Objects

The following functions enqueue commands to read from a buffer object to host memory or write to a buffer object from host memory.

To read from a buffer object to host memory or to write to a buffer object from host memory call one of the functions

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueReadBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_read,
    size_t offset,
    size_t size,
    void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueWriteBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_write,
    size_t offset,
    size_t size,
    const void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* is a valid host command-queue in which the read / write command will be queued. *command_queue* and *buffer* must be created with the same OpenCL context.
- *buffer* refers to a valid buffer object.
- *blocking_read* and *blocking_write* indicate if the read and write operations are *blocking* or *non-blocking* (see below).
- *offset* is the offset in bytes in the buffer object to read from or write to.
- *size* is the size in bytes of data being read or written.
- *ptr* is the pointer to buffer in host memory where data is to be read into or to be written from.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this read / write command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`,

event must not refer to an element of the *event_wait_list* array.

If *blocking_read* is `CL_TRUE` i.e. the read command is blocking, **clEnqueueReadBuffer** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is `CL_FALSE` i.e. the read command is non-blocking, **clEnqueueReadBuffer** queues a non-blocking read command and returns. The contents of the buffer that *ptr* points to cannot be used until the read command has completed. The *event* argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that *ptr* points to can be used by the application.

If *blocking_write* is `CL_TRUE`, the write command is blocking and does not return until the command is complete, including transfer of the data. The memory pointed to by *ptr* can be reused by the application after the **clEnqueueWriteBuffer** call returns.

If *blocking_write* is `CL_FALSE`, the OpenCL implementation will use *ptr* to perform a non-blocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The *event* argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

clEnqueueReadBuffer and **clEnqueueWriteBuffer** return `CL_SUCCESS` if the function is executed successfully. Otherwise, they return one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *buffer* is not a valid buffer object
- `CL_INVALID_VALUE`
  - if the region being read or written specified by *offset* and *size* is out of bounds
  - if *ptr* is `NULL`
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.

- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the read or write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error. This error code is missing before version 1.1.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *buffer*
- `CL_INVALID_OPERATION`
  - if **clEnqueueReadBuffer** is called on *buffer* which has been created with `CL_MEM_HOST_WRITE_ONLY` or `CL_MEM_HOST_NO_ACCESS`
  - if **clEnqueueWriteBuffer** is called on *buffer* which has been created with `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_NO_ACCESS`
  - if **clEnqueueWriteBuffer** is called on *buffer* which has been created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The following functions enqueue commands to read a 2D or 3D rectangular region from a buffer object to host memory or write a 2D or 3D rectangular region to a buffer object from host memory.

```
// Provided by CL_VERSION_1_1
cl_int clEnqueueReadBufferRect(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_read,
    const size_t* buffer_origin,
    const size_t* host_origin,
    const size_t* region,
    size_t buffer_row_pitch,
    size_t buffer_slice_pitch,
    size_t host_row_pitch,
    size_t host_slice_pitch,
    void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReadBufferRect is missing before version 1.1.

```
// Provided by CL_VERSION_1_1
cl_int clEnqueueWriteBufferRect(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_write,
    const size_t* buffer_origin,
    const size_t* host_origin,
    const size_t* region,
    size_t buffer_row_pitch,
    size_t buffer_slice_pitch,
    size_t host_row_pitch,
    size_t host_slice_pitch,
    const void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueWriteBufferRect is missing before version 1.1.

- *command_queue* refers is a valid host command-queue in which the read / write command will be queued. *command_queue* and *buffer* must be created with the same OpenCL context.
- *buffer* refers to a valid buffer object.
- *blocking_read* and *blocking_write* indicate if the read and write operations are *blocking* or *non-blocking* (see below).
- *buffer_origin* defines the (*x*, *y*, *z*) offset in the memory region associated with *buffer*. For a 2D rectangle region, the *z* value given by *buffer_origin*[2] should be 0. The offset in bytes is computed as *buffer_origin*[2] × *buffer_slice_pitch* + *buffer_origin*[1] × *buffer_row_pitch* + *buffer_origin*[0].
- *host_origin* defines the (*x*, *y*, *z*) offset in the memory region pointed to by *ptr*. For a 2D rectangle region, the *z* value given by *host_origin*[2] should be 0. The offset in bytes is computed as *host_origin*[2] × *host_slice_pitch* + *host_origin*[1] × *host_row_pitch* + *host_origin*[0].
- *region* defines the (*width* in bytes, *height* in rows, *depth* in slices) of the 2D or 3D rectangle being read or written. For a 2D rectangle copy, the *depth* value given by *region*[2] should be 1. The values in region cannot be 0.
- *buffer_row_pitch* is the length of each row in bytes to be used for the memory region associated with *buffer*. If *buffer_row_pitch* is 0, *buffer_row_pitch* is computed as *region*[0].
- *buffer_slice_pitch* is the length of each 2D slice in bytes to be used for the memory region associated with *buffer*. If *buffer_slice_pitch* is 0, *buffer_slice_pitch* is computed as *region*[1] × *buffer_row_pitch*.
- *host_row_pitch* is the length of each row in bytes to be used for the memory region pointed to by *ptr*. If *host_row_pitch* is 0, *host_row_pitch* is computed as *region*[0].
- *host_slice_pitch* is the length of each 2D slice in bytes to be used for the memory region pointed to by *ptr*. If *host_slice_pitch* is 0, *host_slice_pitch* is computed as *region*[1] × *host_row_pitch*.

- *ptr* is the pointer to buffer in host memory where data is to be read into or to be written from.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this read / write command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If *blocking_read* is `CL_TRUE` i.e. the read command is blocking, **clEnqueueReadBufferRect** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is `CL_FALSE` i.e. the read command is non-blocking, **clEnqueueReadBufferRect** queues a non-blocking read command and returns. The contents of the buffer that *ptr* points to cannot be used until the read command has completed. The *event* argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that *ptr* points to can be used by the application.

If *blocking_write* is `CL_TRUE`, the write command is blocking and does not return until the command is complete, including transfer of the data. The memory pointed to by *ptr* can be reused by the application after the **clEnqueueWriteBufferRect** call returns.

If *blocking_write* is `CL_FALSE`, the OpenCL implementation will use *ptr* to perform a non-blocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The *event* argument returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

clEnqueueReadBufferRect and **clEnqueueWriteBufferRect** return `CL_SUCCESS` if the function is executed successfully. Otherwise, they return one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *buffer* is not a valid buffer object
- `CL_INVALID_VALUE`

- if *buffer_origin*, *host_origin*, or *region* is `NULL`
- if the region being read or written specified by *buffer_origin*, *region*, *buffer_row_pitch*, and *buffer_slice_pitch* is out of bounds
- if any *region* array element is zero
- if *buffer_row_pitch* is not equal to zero and is less than *region*[0]
- if *buffer_slice_pitch* is not equal to zero and is less than *region*[1] times *buffer_row_pitch*
- if *buffer_slice_pitch* is not equal to zero and is not a multiple of *buffer_row_pitch*
- if *host_row_pitch* is not equal to zero and is less than *region*[0]
- if *host_slice_pitch* is not equal to zero and is less than *region*[1] times *host_row_pitch*
- if *host_slice_pitch* is not equal to zero and is not a multiple of *host_row_pitch*
- if *ptr* is `NULL`

- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *buffer* is a sub-buffer object and the offset specified when the sub-buffer object is created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the read or write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error. This error code is missing before version 1.1.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *buffer*
- `CL_INVALID_OPERATION`
  - if **clEnqueueReadBufferRect** is called on *buffer* which has been created with `CL_MEM_HOST_WRITE_ONLY` or `CL_MEM_HOST_NO_ACCESS`
  - if **clEnqueueWriteBufferRect** is called on *buffer* which has been created with `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_NO_ACCESS`
  - if **clEnqueueWriteBufferRect** is called on *buffer* which has been created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Calling **clEnqueueReadBuffer** to read a region of the buffer object with the *ptr* argument value set to *host_ptr* + *offset*, where *host_ptr* is a pointer to the memory region specified when the buffer object being read is created with `CL_MEM_USE_HOST_PTR`, must meet the following requirements in order to avoid undefined behavior:

- All commands that use this buffer object or a memory object (buffer or image) created from this buffer object have finished execution before the read command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the read command has finished execution.

Calling **clEnqueueReadBufferRect** to read a region of the buffer object with the *ptr* argument value set to *host_ptr* and *host_origin*, *buffer_origin* values are the same, where *host_ptr* is a pointer to the memory region specified when the buffer object being read is created with `CL_MEM_USE_HOST_PTR`, must meet the same requirements given above for **clEnqueueReadBuffer**.

Calling **clEnqueueWriteBuffer** to update the latest bits in a region of the buffer object with the *ptr* argument value set to *host_ptr* + *offset*, where *host_ptr* is a pointer to the memory region specified when the buffer object being written is created with `CL_MEM_USE_HOST_PTR`, must meet the following requirements in order to avoid undefined behavior:

- The host memory region given by (*host_ptr* + *offset*, *cb*) contains the latest bits when the enqueued write command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

Calling **clEnqueueWriteBufferRect** to update the latest bits in a region of the buffer object with the *ptr* argument value set to *host_ptr* and *host_origin*, *buffer_origin* values are the same, where *host_ptr* is a pointer to the memory region specified when the buffer object being written is created with `CL_MEM_USE_HOST_PTR`, must meet the following requirements in order to avoid undefined behavior:

- The host memory region given by (*buffer_origin region*) contains the latest bits when the enqueued write command begins execution.
- The buffer object or memory objects created from this buffer object are not mapped.
- The buffer object or memory objects created from this buffer object are not used by any command-queue until the write command has finished execution.

To enqueue a command to copy a buffer object identified by *src_buffer* to another buffer object identified by *dst_buffer*, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueCopyBuffer(
    cl_command_queue command_queue,
    cl_mem src_buffer,
    cl_mem dst_buffer,
    size_t src_offset,
    size_t dst_offset,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* refers to a host command-queue in which the copy command will be queued. The OpenCL context associated with *command_queue*, *src_buffer* and *dst_buffer* must be the same.
- *src_offset* refers to the offset where to begin copying data from *src_buffer*.
- *dst_offset* refers to the offset where to begin copying data into *dst_buffer*.
- *size* refers to the size in bytes to copy.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this copy command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_buffer* or _dst_buffer is created is ignored by **clEnqueueCopyBuffer**.

clEnqueueCopyBuffer returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`

- if the context associated with *command_queue* and *src_buffer* are not the same
- if the context associated with *command_queue* and *dst_buffer* are not the same
- if the context associated with *command_queue* and events in *event_wait_list* are not the same

- `CL_INVALID_MEM_OBJECT`
  - if *src_buffer* is not a valid buffer object
  - if *dst_buffer* is not a valid buffer object
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *src_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
  - if *dst_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_INVALID_VALUE`
  - if the region specified by *src_offset* and *size* is out of bounds of *src_buffer*
  - if the region specified by *dst_offset* and *size* is out of bounds of *dst_buffer*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MEM_COPY_OVERLAP`
  - if *src_buffer* and *dst_buffer* are the same buffer or sub-buffer object and the source and destination regions overlap
  - if *src_buffer* and *dst_buffer* are different sub-buffers of the same associated buffer object and they overlap
  - Refer to Checking for Memory Copy Overlap for details on how to determine if source and destination regions overlap.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *src_buffer* or *dst_buffer*
- `CL_INVALID_OPERATION`
  - if *dst_buffer* was created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command to copy a 2D or 3D rectangular region from the buffer object identified by *src_buffer* to a 2D or 3D region in the buffer object identified by *dst_buffer*, call the function

```
// Provided by CL_VERSION_1_1
cl_int clEnqueueCopyBufferRect(
    cl_command_queue command_queue,
    cl_mem src_buffer,
    cl_mem dst_buffer,
    const size_t* src_origin,
    const size_t* dst_origin,
    const size_t* region,
    size_t src_row_pitch,
    size_t src_slice_pitch,
    size_t dst_row_pitch,
    size_t dst_slice_pitch,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueCopyBufferRect is missing before version 1.1.

- *command_queue* refers to the host command-queue in which the copy command will be queued. The OpenCL context associated with *command_queue*, *src_buffer* and *dst_buffer* must be the same.
- *src_origin* defines the (*x*, *y*, *z*) offset in the memory region associated with *src_buffer*. For a 2D rectangle region, the *z* value given by *src_origin*[2] should be 0. The offset in bytes is computed as *src_origin*[2] × *src_slice_pitch* + *src_origin*[1] × *src_row_pitch* + *src_origin*[0].
- *dst_origin* defines the (*x*, *y*, *z*) offset in the memory region associated with *dst_buffer*. For a 2D rectangle region, the *z* value given by *dst_origin*[2] should be 0. The offset in bytes is computed as *dst_origin*[2] × *dst_slice_pitch* + *dst_origin*[1] × *dst_row_pitch* + *dst_origin*[0].
- *region* defines the (*width* in bytes, *height* in rows, *depth* in slices) of the 2D or 3D rectangle being copied. For a 2D rectangle, the *depth* value given by *region*[2] should be 1. The values in region cannot be 0.
- *src_row_pitch* is the length of each row in bytes to be used for the memory region associated with *src_buffer*. If *src_row_pitch* is 0, *src_row_pitch* is computed as *region*[0].
- *src_slice_pitch* is the length of each 2D slice in bytes to be used for the memory region associated with *src_buffer*. If *src_slice_pitch* is 0, *src_slice_pitch* is computed as *region*[1] × *src_row_pitch*.
- *dst_row_pitch* is the length of each row in bytes to be used for the memory region associated with *dst_buffer*. If *dst_row_pitch* is 0, *dst_row_pitch* is computed as *region*[0].
- *dst_slice_pitch* is the length of each 2D slice in bytes to be used for the memory region associated with *dst_buffer*. If *dst_slice_pitch* is 0, *dst_slice_pitch* is computed as *region*[1] × *dst_row_pitch*.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must

be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this copy command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

Copying begins at the source offset and destination offset which are computed as described below in the description for *src_origin* and *dst_origin*. Each byte of the region's width is copied from the source offset to the destination offset. After copying each width, the source and destination offsets are incremented by their respective source and destination row pitches. After copying each 2D rectangle, the source and destination offsets are incremented by their respective source and destination slice pitches.

If *src_buffer* and *dst_buffer* are the same buffer object, *src_row_pitch* must equal *dst_row_pitch* and *src_slice_pitch* must equal *dst_slice_pitch*.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_buffer* or *dst_buffer* is created is ignored by **clEnqueueCopyBufferRect**.

clEnqueueCopyBufferRect returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *src_buffer* are not the same
  - if the context associated with *command_queue* and *dst_buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *src_buffer* is not a valid buffer object
  - if *dst_buffer* is not a valid buffer object
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *src_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
  - if *dst_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.

- `CL_INVALID_VALUE`
  - if *src_origin*, *dst_origin*, or *region* is `NULL`
  - if the region specified by *src_origin*, *region*, *src_row_pitch*, and *src_slice_pitch* is out of bounds of *src_buffer*
  - if the region specified by *dst_origin*, *region*, *dst_row_pitch*, and *dst_slice_pitch* is out of bounds of *dst_buffer*
  - if any *region* array element is zero
  - if *src_row_pitch* is not equal to zero and is less than *region*[0]
  - if *src_slice_pitch* is not equal to zero and is less than *region*[1] times *src_row_pitch*
  - if *src_slice_pitch* is not equal to zero and is not a multiple of *src_row_pitch*
  - if *dst_row_pitch* is not equal to zero and is less than *region*[0]
  - if *dst_slice_pitch* is not equal to zero and is less than *region*[1] times *dst_row_pitch*
  - if *dst_slice_pitch* is not equal to zero and is not a multiple of *dst_row_pitch*
  - if *src_buffer* and *dst_buffer* are the same buffer object and *src_row_pitch* is not equal to *dst_row_pitch*
  - if *src_buffer* and *dst_buffer* are the same buffer object and *src_slice_pitch* is not equal to *dst_slice_pitch*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MEM_COPY_OVERLAP`
  - if *src_buffer* and *dst_buffer* are the same buffer or sub-buffer object and the source and destination regions overlap
  - if *src_buffer* and *dst_buffer* are different sub-buffers of the same associated buffer object and they overlap
  - Refer to Checking for Memory Copy Overlap for details on how to determine if source and destination regions overlap.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *src_buffer* or *dst_buffer*
- `CL_INVALID_OPERATION`
  - if *dst_buffer* was created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`

- if there is a failure to allocate resources required by the OpenCL implementation on the host

5.2.3. Filling Buffer Objects

Filling buffer objects is missing before version 1.2.

To enqueue a command to fill a buffer object with a pattern of a given pattern size, call the function

```
// Provided by CL_VERSION_1_2
cl_int clEnqueueFillBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    const void* pattern,
    size_t pattern_size,
    size_t offset,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueFillBuffer is missing before version 1.2.

- *command_queue* refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and *buffer* must be the same.
- *buffer* is a valid buffer object.
- *pattern* is a pointer to the data pattern of size *pattern_size* in bytes. *pattern* will be used to fill a region in *buffer* starting at *offset* and is *size* bytes in size. The data pattern must be a scalar or vector integer or floating-point data type supported by OpenCL as described in Shared Application Scalar Data Types and Supported Application Vector Data Types. For example, if *buffer* is to be filled with a pattern of `float4` values, then *pattern* will be a pointer to a `cl_float4` value and *pattern_size* will be `sizeof(cl_float4)`. The maximum value of *pattern_size* is the size of the largest integer or floating-point vector data type supported by the OpenCL device. The memory associated with *pattern* can be reused or freed after the function returns.
- *offset* is the location in bytes of the region being filled in *buffer* and must be a multiple of *pattern_size*.
- *size* is the size in bytes of region being filled in *buffer* and must be a multiple of *pattern_size*.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a

wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *buffer* is created is ignored by **clEnqueueFillBuffer**.

clEnqueueFillBuffer returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *buffer* is not a valid buffer object
- `CL_INVALID_VALUE`
  - if the region being filled specified by *offset* and *size* is out of bounds
  - if *pattern* is `NULL`
  - if *pattern_size* is zero
  - if *pattern_size* is not a power of two
  - if *pattern_size* is greater than 128
  - if *offset* is not a multiple of *pattern_size*
  - if *size* is not a multiple of *pattern_size*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *buffer*
- `CL_INVALID_OPERATION`
  - if *buffer* was created with `CL_MEM_IMMUTABLE_EXT`

- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.2.4. Mapping Buffer Objects

To enqueue a command to map a region of the buffer object given by *buffer* into the host address space and returns a pointer to this mapped region, call the function

```
// Provided by CL_VERSION_1_0
void* clEnqueueMapBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_map,
    cl_map_flags map_flags,
    size_t offset,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event,
    cl_int* errcode_ret);
```

- *command_queue* must be a valid host command-queue.
- *blocking_map* indicates if the map operation is *blocking* or *non-blocking*.

If *blocking_map* is `CL_TRUE`, **clEnqueueMapBuffer** does not return until the specified region in *buffer* is mapped into the host address space and the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapBuffer**.

If *blocking_map* is `CL_FALSE` i.e. map operation is non-blocking, the pointer to the mapped region returned by **clEnqueueMapBuffer** cannot be used until the map command has completed. The *event* argument returns an event object which can be used to query the execution status of the map command. When the map command is completed, the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapBuffer**.

- *map_flags* is a bit-field and is described in the Memory Map Flags table.
- *buffer* is a valid buffer object. The OpenCL context associated with *command_queue* and *buffer* must be the same.
- *offset* and *size* are the offset in bytes and the size of the region in the buffer object that is being mapped.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid

and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

clEnqueueMapBuffer will return a pointer to the mapped region. The *errcode_ret* is set to `CL_SUCCESS`.

A `NULL` pointer is returned otherwise with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *buffer* is not a valid buffer object
- `CL_INVALID_VALUE`
  - if the region being mapped specified by *offset* and *size* is out of bounds
  - if *size* is zero
  - if values specified in *map_flags* are not valid
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_MAP_FAILURE`
  - if there is a failure to map the requested region into the host address space. This error cannot occur for buffer objects created with `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR`.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`

    - if the map operations is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error. This error code is missing before version 1.1.

- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
    - if there is a failure to allocate memory for the data store associated with *buffer*
- `CL_INVALID_OPERATION`
    - if *buffer* was created with `CL_MEM_HOST_WRITE_ONLY` or `CL_MEM_HOST_NO_ACCESS` and `CL_MAP_READ` is set in *map_flags*
    - if *buffer* was created with `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_NO_ACCESS` and `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` is set in *map_flags*
    - if *buffer* was created with `CL_MEM_IMMUTABLE_EXT` and `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` is set in *map_flags*
    - if mapping would lead to overlapping regions being mapped for writing
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

The pointer returned maps a region starting at *offset* and is at least *size* bytes in size. The result of a memory access outside this region is undefined.

If the buffer object is created with `CL_MEM_USE_HOST_PTR` set in *mem_flags*, the following will be true:

- The *host_ptr* specified in **clCreateBuffer** or **clCreateBufferWithProperties** will contain the latest bits in the region being mapped when the **clEnqueueMapBuffer** command has completed.
- The pointer value returned by **clEnqueueMapBuffer** will be derived from the *host_ptr* specified when the buffer object is created.

Mapped buffer objects are unmapped using **clEnqueueUnmapMemObject**. This is described in Unmapping Mapped Memory Objects.

Table 21. List of supported map flag values

Map Flags	Description
CL_MAP_READ	This flag specifies that the region being mapped in the memory object is being mapped for reading. The pointer returned by **clEnqueueMapBuffer** (**clEnqueueMapImage**) is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapBuffer** (**clEnqueueMapImage**) command has completed.
CL_MAP_WRITE	This flag specifies that the region being mapped in the memory object is being mapped for writing. The pointer returned by **clEnqueueMapBuffer** (**clEnqueueMapImage**) is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapBuffer** (**clEnqueueMapImage**) command has completed
CL_MAP_WRITE_INVALIDATE_REGION missing before version 1.2.	This flag specifies that the region being mapped in the memory object is being mapped for writing. The contents of the region being mapped are to be discarded. This is typically the case when the region being mapped is overwritten by the host. This flag allows the implementation to no longer guarantee that the pointer returned by **clEnqueueMapBuffer** (**clEnqueueMapImage**) contains the latest bits in the region being mapped which can be a significant performance enhancement. CL_MAP_READ or CL_MAP_WRITE and CL_MAP_WRITE_INVALIDATE_REGION are mutually exclusive.

5.2.5. Creating Buffer Objects From Direct3D Buffer Resources

To create an OpenCL buffer object from a Direct3D 10 buffer, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_mem clCreateFromD3D10BufferKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D10Buffer* resource,
    cl_int* errcode_ret);
```

clCreateFromD3D10BufferKHR is provided by the `cl_khr_d3d10_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 10 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *resource* is a pointer to the Direct3D 10 buffer to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The size of the returned OpenCL buffer object is the same as the size of *resource*. This call will increment the internal Direct3D 10 reference count on *resource*. The internal Direct3D 10 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D10BufferKHR returns a valid non-zero OpenCL buffer object and *errcode_ret* is set to `CL_SUCCESS` if the buffer object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid.
- `CL_INVALID_D3D10_RESOURCE_KHR` if *resource* is not a Direct3D 10 buffer resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if a `cl_mem` from *resource* has already been created using **clCreateFromD3D10BufferKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create an OpenCL buffer object from a Direct3D 11 buffer, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_mem clCreateFromD3D11BufferKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D11Buffer* resource,
    cl_int* errcode_ret);
```

clCreateFromD3D11BufferKHR is provided by the `cl_khr_d3d11_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 11 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *resource* is a pointer to the Direct3D 11 buffer to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The size of the returned OpenCL buffer object is the same as the size of *resource*. This call will increment the internal Direct3D 11 reference count on *resource*. The internal Direct3D 11 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D11BufferKHR returns a valid non-zero OpenCL buffer object and *errcode_ret* is set to `CL_SUCCESS` if the buffer object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid.
- `CL_INVALID_D3D11_RESOURCE_KHR` if *resource* is not a Direct3D 11 buffer resource, if *resource* was created with the D3D11_USAGE flag D3D11_USAGE_IMMUTABLE, if a `cl_mem` from *resource* has already been created using **clCreateFromD3D11BufferKHR**, or if *context* was not created against the same Direct3D 11 device from which *resource* was created.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.2.6. Creating Buffer Objects From OpenGL Buffer Objects

To create an OpenCL buffer object from an OpenGL buffer object, call the function

```
// Provided by cl_khr_gl_sharing
cl_mem clCreateFromGLBuffer(
    cl_context context,
    cl_mem_flags flags,
    cl_GLuint bufobj,
    cl_int* errcode_ret);
```

clCreateFromGLBuffer is provided by the `cl_khr_gl_sharing` extension.

- *context* is a valid OpenCL context created from an OpenGL context.
- *flags* is a bit-field that is used to specify usage information. Refer to the Memory Flags table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *bufobj* is the name of an OpenGL buffer object. The data store of the OpenGL buffer object must have been previously created by calling `glBufferData`, although its contents need not be initialized. The size of the data store will be used to determine the size of the OpenCL buffer object.
- *errcode_ret* will return an appropriate error code as described below. If *errcode_ret* is `NULL`, no error code is returned.

The size of the OpenGL buffer object data store at the time **clCreateFromGLBuffer** is called will be used as the size of buffer object returned by **clCreateFromGLBuffer**. If the state of an OpenGL buffer object is modified through the OpenGL API (e.g. `glBufferData`) while there exists a corresponding OpenCL buffer object, subsequent use of the OpenCL buffer object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the buffer object.

The OpenCL buffer object created using **clCreateFromGLBuffer** can also be used to create an OpenCL 1D image buffer object.

Refer to the Lifetime of Shared OpenCL/OpenGL Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromGLBuffer returns a valid non-zero OpenCL buffer object and *errcode_ret* is set to `CL_SUCCESS` if the buffer object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context or was not created from an OpenGL context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid.
- `CL_INVALID_GL_OBJECT` if *bufobj* is not an OpenGL buffer object or is a OpenGL buffer object but does not have an existing data store or the size of the buffer is 0.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL

implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3. Image Objects

An *image* object is used to store a one-, two- or three-dimensional texture, frame-buffer or image. The elements of an image object are selected from a list of predefined image formats. The minimum number of elements in a memory object is one.

5.3.1. Creating Image Objects

An **image object** may be created using the function

```
// Provided by CL_VERSION_1_2
cl_mem clCreateImage(
    cl_context context,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    const cl_image_desc* image_desc,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateImage is missing before version 1.2.

An **image object** may also be created with additional properties using the function

```
// Provided by CL_VERSION_3_0
cl_mem clCreateImageWithProperties(
    cl_context context,
    const cl_mem_properties* properties,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    const cl_image_desc* image_desc,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateImageWithProperties is missing before version 3.0.

- *context* is a valid OpenCL context used to create the image object.
- *properties* is an optional list of properties for the image object and their corresponding values. The list is terminated with the special property `0`. If no properties are required, *properties* may be `NULL`. OpenCL does not define any optional properties for images, but extensions may define properties as described in the List of supported image creation properties.
- *flags* is a bit-field that is used to specify allocation and usage information about the image

memory object being created and is described in the supported memory flag values table.

- *image_format* is a pointer to a structure that describes format properties of the image to be allocated. A 1D image buffer can be created from a buffer by specifying a buffer object in *image_desc→mem_object*. If the `cl_khr_image2d_from_buffer` extension is supported, a 2D image can be created from a buffer by specifying a buffer object in *image_desc→mem_object*. If the `cl_ext_image_from_buffer` extension is supported, an image of any type can be created from a buffer by specifying a buffer object in *image_desc→mem_object*. A 2D image can be created from another 2D image object by specifying an image object in the *image_desc→mem_object*. Refer to the Image Format Descriptor section for a detailed description of the image format descriptor.
- *image_desc* is a pointer to a structure that describes type and dimensions of the image to be allocated. Refer to the Image Descriptor section for a detailed description of the image descriptor.
- *host_ptr* is a pointer to the image data that may already be allocated by the application. Refer to the table below for a description of how large the buffer that *host_ptr* points to must be.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The alignment requirements for data stored in image objects are described in Alignment of Application Data Types.

For all image types except `CL_MEM_OBJECT_IMAGE1D_BUFFER`, if the value specified for *flags* is 0, the default is used which is `CL_MEM_READ_WRITE`.

For `CL_MEM_OBJECT_IMAGE1D_BUFFER` image type, or an image created from another memory object (image or buffer), if the `CL_MEM_READ_WRITE`, `CL_MEM_READ_ONLY` or `CL_MEM_WRITE_ONLY` values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *mem_object*. The `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR` and `CL_MEM_COPY_HOST_PTR` values cannot be specified in *flags* but are inherited from the corresponding memory access qualifiers associated with *mem_object*. If `CL_MEM_COPY_HOST_PTR` is specified in the memory access qualifier values associated with *mem_object* it does not imply any additional copies when the image is created from *mem_object*. If the `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_NO_ACCESS` values are not specified in *flags*, they are inherited from the corresponding memory access qualifiers associated with *mem_object*.

Mipmap Images

A mipmapped 1D image, 1D image array, 2D image, 2D image array or 3D image is created by specifying *num_mip_levels* to be a value greater than one in *image_desc*. The dimensions of a mipmapped image can be a power of two or a non-power of two. Each successively smaller mipmap level is half the size of the previous level, rounded down to the nearest integer.

The following restrictions apply when mipmapped images are created with **clCreateImage**:

- `CL_MEM_USE_HOST_PTR` or `CL_MEM_COPY_HOST_PTR` cannot be specified if a mipmapped image is created.
- The *host_ptr* argument to **clCreateImage** must be a `NULL` value.
- Mip-mapped images cannot be created for `CL_MEM_OBJECT_IMAGE1D_BUFFER` images, depth images

or multi-sampled (i.e. msaa) images.

Image Data in Host Memory

For a 3D image or 2D image array, the image data specified by *host_ptr* is stored as a linear sequence of adjacent 2D image slices or 2D images respectively. Each 2D image is a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements.

For a 2D image, the image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements.

For a 1D image array, the image data specified by *host_ptr* is stored as a linear sequence of adjacent 1D images. Each 1D image is stored as a single scanline which is a linear sequence of adjacent elements.

For 1D image or 1D image buffer, the image data specified by *host_ptr* is stored as a single scanline which is a linear sequence of adjacent elements.

Image elements are stored according to their image format as described in the Image Format Descriptor section.

Table 22. List of supported image creation properties

Property	Property Value	Description
`CL_MEM_DEVICE_HANDLE_LIST_KHR` provided by the `cl_khr_external_memory` extension.	`cl_device_id`[]	Specifies the list of OpenCL devices (terminated with `CL_MEM_DEVICE_HANDLE_LIST_END_KHR`) to associate with the external memory handle.

If `CL_MEM_DEVICE_HANDLE_LIST_KHR` is not specified as part of *properties*, the memory object created by **clCreateBufferWithProperties** or **clCreateImageWithProperties** is by default associated with all devices in the *context*.

The properties used to create an image from an external memory handle are described for the corresponding extensions. When an image is created from an external memory handle, the *flags* used to specify usage information for the image must not include `CL_MEM_USE_HOST_PTR`, `CL_MEM_ALLOC_HOST_PTR`, or `CL_MEM_COPY_HOST_PTR`, and the *host_ptr* argument must be `NULL`. When images are created from an external memory handle, implementations may acquire information about image attributes such as format and layout at the time of creation. When such information is acquired at image creation time, it is used for the lifetime of the image object.

If `CL_MEM_IMMUTABLE_EXT` is set in *flags*, the image created is an *immutable memory object* that cannot be modified by the implementation after creation, and one and only one of the following conditions must be met:

- Either `CL_MEM_COPY_HOST_PTR` or `CL_MEM_USE_HOST_PTR` must be set in *flags*.
- An external memory handle must be included in *properties*.

If the image being created is created from another memory object and that memory object is an

immutable memory object, then CL_MEM_IMMUTABLE_EXT must be set in *flags*.

clCreateImage and **clCreateImageWithProperties** returns a valid non-zero image object and *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, they return a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT
  - if *context* is not a valid context
- CL_INVALID_OPERATION
  - if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_FALSE)
- CL_INVALID_PROPERTY
  - if a property name in *properties* is not a supported property name
  - if the value specified for a supported property name is not valid
  - if the same property name is specified more than once
  - if *properties* does not include a supported external memory handle and CL_MEM_DEVICE_HANDLE_LIST_KHR is specified as part of *properties*
  - if *properties* includes more than one external memory handle
- CL_INVALID_DEVICE
  - if a device identified by the property CL_MEM_DEVICE_HANDLE_LIST_KHR is not a valid device or is not associated with *context*
  - if a device identified by property CL_MEM_DEVICE_HANDLE_LIST_KHR cannot import the requested external memory object type
  - if CL_MEM_DEVICE_HANDLE_LIST_KHR is not specified as part of *properties* and one or more devices in *context* cannot import the requested external memory object type
- CL_INVALID_VALUE
  - if values specified in *flags* are not valid as defined in the Memory Flags table
  - if an image is being created from another memory object (buffer or image) under one of the following circumstances:
    - *mem_object* was created with CL_MEM_WRITE_ONLY and *flags* specifies CL_MEM_READ_WRITE or CL_MEM_READ_ONLY
    - *mem_object* was created with CL_MEM_READ_ONLY and *flags* specifies CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY
    - *flags* specifies CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR or CL_MEM_COPY_HOST_PTR
  - if an image is being created from another memory object (buffer or image) and *mem_object* was created with CL_MEM_HOST_WRITE_ONLY and *flags* specifies CL_MEM_HOST_READ_ONLY
  - if *mem_object* was created with CL_MEM_HOST_READ_ONLY and *flags* specifies CL_MEM_HOST_WRITE_ONLY
  - if *mem_object* was created with CL_MEM_HOST_NO_ACCESS and_flags_ specifies CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_WRITE_ONLY

- if *properties* includes a supported external memory handle and *flags* includes CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, or CL_MEM_COPY_HOST_PTR
- if CL_MEM_IMMUTABLE_EXT is set in *flags* and CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, or CL_MEM_HOST_WRITE_ONLY is set in *flags*
- if CL_MEM_IMMUTABLE_EXT is not set in *flags* and *mem_object* was created with CL_MEM_IMMUTABLE_EXT
- if CL_MEM_IMMUTABLE_EXT is set in *flags* and none of the following conditions are met:
  - CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR is set in *flags*
  - the image is being created from another memory object (buffer or image)
  - *properties* includes an external memory handle

- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR
  - if *image_format* is NULL
  - if values specified in *image_format* are not valid
  - if *properties* includes an AHardwareBuffer external memory handle and *image_format* is not NULL
  - if an image is created from a buffer and the row pitch, or slice pitch, if the cl_ext_image_from_buffer extension is supported, or base address alignment do not follow the rules described for creating an image from a buffer
- CL_INVALID_IMAGE_DESCRIPTOR
  - if *properties* includes an AHardwareBuffer external memory handle and *image_desc* is not NULL
- CL_INVALID_IMAGE_SIZE
  - if the image dimensions specified in *image_desc* are not valid or exceed the maximum image dimensions described in the Device Queries table for all devices in *context*
  - if the cl_ext_image_from_buffer extension is supported and an image is created from a buffer and the buffer passed in *mem_object* is too small to be used as a data store for the image, e.g. if its size is smaller than the value returned for CL_IMAGE_REQUIREMENTS_SIZE_EXT for the parameters used to create the image
- CL_INVALID_HOST_PTR
  - if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags*
  - if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*
  - if *properties* includes a supported external memory handle and *host_ptr* is not NULL
- CL_IMAGE_FORMAT_NOT_SUPPORTED
  - if there are no devices in *context* that support *image_format*
  - if *properties* includes an AHardwareBuffer external memory handle and the AHardwareBuffer format is AHARDWAREBUFFER_FORMAT_BLOB or not supported
- CL_MEM_OBJECT_ALLOCATION_FAILURE
  - if there is a failure to allocate memory for the image object

- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Table 23. Required host_ptr buffer sizes for images

Image Type	Size of buffer that *host_ptr* points to
CL_MEM_OBJECT_IMAGE1D missing before version 1.2.	≥ image_row_pitch
CL_MEM_OBJECT_IMAGE1D_BUFFER missing before version 1.2.	≥ image_row_pitch
CL_MEM_OBJECT_IMAGE2D	≥ image_row_pitch × image_height
CL_MEM_OBJECT_IMAGE3D	≥ image_slice_pitch × image_depth
CL_MEM_OBJECT_IMAGE1D_ARRAY missing before version 1.2.	≥ image_slice_pitch × image_array_size
CL_MEM_OBJECT_IMAGE2D_ARRAY missing before version 1.2.	≥ image_slice_pitch × image_array_size

A **2D image** object can be created using the following function

```
// Provided by CL_VERSION_1_0
cl_mem clCreateImage2D(
    cl_context context,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    size_t image_width,
    size_t image_height,
    size_t image_row_pitch,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateImage2D is deprecated by version 1.2.

- *context* is a valid OpenCL context on which the image object is to be created.
- *flags* is a bit-field that is used to specify allocation and usage information about the image memory object being created and is described in the supported memory flag values table. If the value specified for *flags* is 0, the default is used which is CL_MEM_READ_WRITE.
- *image_format* is a pointer to a structure that describes format properties of the image to be

allocated. Refer to the Image Format Descriptor section for a detailed description of the image format descriptor.

- *image_width* and *image_height* are the width and height of the image in pixels. These must be values greater than or equal to 1.
- *image_row_pitch* is the scan-line pitch in bytes. This must be 0 if *host_ptr* is NULL and can be either 0 or ≥ *image_width* × size of element in bytes if *host_ptr* is not NULL. If *host_ptr* is not NULL and *image_row_pitch* is 0, *image_row_pitch* is calculated as *image_width* × size of element in bytes. If *image_row_pitch* is not 0, it must be a multiple of the image element size in bytes.
- *host_ptr* is a pointer to the image data that may already be allocated by the application. Refer to the CL_MEM_OBJECT_IMAGE2D entry in the required *host_ptr* buffer size table for a description of how large the buffer that *host_ptr* points to must be. The image data specified by *host_ptr* is stored as a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements. Image elements are stored according to their image format as described in the Image Format Descriptor section.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

If CL_MEM_IMMUTABLE_EXT is set in *flags*, the image created is an *immutable memory object* that cannot be modified by the implementation after creation, and CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR must be set in *flags*.

clCreateImage2D returns a valid non-zero image object created and the *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format* are not valid or if *image_format* is NULL.
- CL_INVALID_IMAGE_SIZE if *image_width* or *image_height* are 0 or if they exceed the maximum values specified in CL_DEVICE_IMAGE2D_MAX_WIDTH or CL_DEVICE_IMAGE2D_MAX_HEIGHT respectively for all devices in *context* or if values specified by *image_row_pitch* do not follow rules described in the argument description above.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags* or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if there are no devices in *context* that support *image_format*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL

implementation on the host.

- `CL_INVALID_VALUE` if `CL_MEM_IMMUTABLE_EXT` is set in *flags* and neither `CL_MEM_COPY_HOST_PTR` nor `CL_MEM_USE_HOST_PTR` is set in *flags*.

A **3D image** object can be created using the following function

```
// Provided by CL_VERSION_1_0
cl_mem clCreateImage3D(
    cl_context context,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    size_t image_width,
    size_t image_height,
    size_t image_depth,
    size_t image_row_pitch,
    size_t image_slice_pitch,
    void* host_ptr,
    cl_int* errcode_ret);
```

clCreateImage3D is deprecated by version 1.2.

- *context* is a valid OpenCL context on which the image object is to be created.
- *flags* is a bit-field that is used to specify allocation and usage information about the image memory object being created and is described in the supported memory flag values table. If the value specified for *flags* is 0, the default is used which is `CL_MEM_READ_WRITE`.
- *image_format* is a pointer to a structure that describes format properties of the image to be allocated. Refer to the Image Format Descriptor section for a detailed description of the image format descriptor.
- *image_width* and *image_height* are the width and height of the image in pixels. These must be values greater than or equal to 1.
- *image_depth* is the depth of the image in pixels. For **clCreateImage3D**, this must be a value > 1.
- *image_row_pitch* is the scan-line pitch in bytes. This must be 0 if *host_ptr* is `NULL` and can be either 0 or ≥ *image_width* × size of element in bytes if *host_ptr* is not `NULL`. If *host_ptr* is not `NULL` and *image_row_pitch* is 0, *image_row_pitch* is calculated as *image_width* × size of element in bytes. If *image_row_pitch* is not 0, it must be a multiple of the image element size in bytes.
- *image_slice_pitch* is the size in bytes of each 2D slice in the 3D image. This must be 0 if *host_ptr* is `NULL` and can be 0 or ≥ *image_row_pitch* × *image_height* if *host_ptr* is not `NULL`. If *host_ptr* is not `NULL` and *image_slice_pitch* is 0, *image_slice_pitch* is calculated as *image_row_pitch* × *image_height*. If *image_slice_pitch* is not 0, it must be a multiple of the *image_row_pitch*.
- *host_ptr* is a pointer to the image data that may already be allocated by the application. Refer to the `CL_MEM_OBJECT_IMAGE3D` entry in the required *host_ptr* buffer size table for a description of how large the buffer that *host_ptr* points to must be. The image data specified by *host_ptr* is stored as a linear sequence of adjacent 2D slices. Each scanline is a linear sequence of image elements. Image elements are stored according to their image format as described in the Image

Format Descriptor section.

- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

If CL_MEM_IMMUTABLE_EXT is set in *flags*, the image created is an *immutable memory object* that cannot be modified by the implementation after creation, and CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR must be set in *flags*.

clCreateImage3D returns a valid non-zero image object created and the *errcode_ret* is set to CL_SUCCESS if the image object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if values specified in *flags* are not valid.
- CL_INVALID_IMAGE_FORMAT_DESCRIPTOR if values specified in *image_format* are not valid or if *image_format* is NULL.
- CL_INVALID_IMAGE_SIZE if *image_width* or *image_height* are 0 or if *image_depth* ≤ 1, or if they exceed the maximum values specified in CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT or CL_DEVICE_IMAGE3D_MAX_DEPTH respectively for all devices in *context*, or if values specified by *image_row_pitch* and *image_slice_pitch* do not follow rules described in the argument description above.
- CL_INVALID_HOST_PTR if *host_ptr* is NULL and CL_MEM_USE_HOST_PTR or CL_MEM_COPY_HOST_PTR are set in *flags* or if *host_ptr* is not NULL but CL_MEM_COPY_HOST_PTR or CL_MEM_USE_HOST_PTR are not set in *flags*.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if there are no devices in *context* that support *image_format*.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for image object.
- CL_INVALID_OPERATION if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_INVALID_VALUE if CL_MEM_IMMUTABLE_EXT is set in *flags* and neither CL_MEM_COPY_HOST_PTR nor CL_MEM_USE_HOST_PTR is set in *flags*.

5.3.1.1. Image Format Descriptor

The cl_image_format image format descriptor structure describes an image format, and is defined as:

```
// Provided by CL_VERSION_1_0
typedef struct cl_image_format {
    cl_channel_order     image_channel_order;
    cl_channel_type      image_channel_data_type;
} cl_image_format;
```

- *image_channel_order* specifies the number of channels and the channel layout i.e. the memory layout in which channels are stored in the image. Valid values are described in the Image Channel Order table.
- *image_channel_data_type* describes the size of the channel data type. The list of supported values is described in the Image Channel Data Types table. The number of bits per element determined by the `image_channel_data_type` and `image_channel_order` must be a power of two.

Table 24. List of supported Image Channel Order Values

Image Channel Order	Description
`CL_R`, `CL_A`,	Single channel image formats where the single channel represents a `RED` or `ALPHA` component.
`CL_DEPTH` missing before version 2.0. Also supported if the `cl_khr_depth_images` extension is supported.	A single channel image format where the single channel represents a `DEPTH` component.
`CL_LUMINANCE`	A single channel image format where the single channel represents a `LUMINANCE` value. The `LUMINANCE` value is replicated into the `RED`, `GREEN`, and `BLUE` components.
`CL_INTENSITY`,	A single channel image format where the single channel represents an `INTENSITY` value. The `INTENSITY` value is replicated into the `RED`, `GREEN`, `BLUE`, and `ALPHA` components.
`CL_RG`, `CL_RA`	Two channel image formats. The first channel always represents a `RED` component. The second channel represents a `GREEN` component or an `ALPHA` component.
`CL_Rx` missing before version 1.1.	A two channel image format, where the first channel represents a `RED` component and the second channel is ignored.

Image Channel Order	Description
CL_DEPTH_STENCIL provided by the cl_khr_gl_depth_images extension.	A two channel image format, where the first channel represents a DEPTH component and the second channel represents a stencil component. This format can only be used if the image channel data type is CL_UNORM_INT24 or CL_FLOAT. See Restrictions on Depth/Stencil Images.
CL_RGB	A three channel image format, where the three channels represent RED, GREEN, and BLUE components.
CL_RGx missing before version 1.1.	A three channel image format, where the first two channels represent RED and GREEN components and the third channel is ignored.
CL_RGBA, CL_ARGB, CL_BGRA, CL_ABGR CL_ABGR is missing before version 2.0.	Four channel image formats, where the four channels represent RED, GREEN, BLUE, and ALPHA components.
CL_RGBx missing before version 1.1.	A four channel image format, where the first three channels represent RED, GREEN, and BLUE components and the fourth channel is ignored.
CL_sRGB missing before version 2.0.	A three channel image format, where the three channels represent RED, GREEN, and BLUE components in the sRGB color space.
CL_sRGBA, CL_sBGRA missing before version 2.0.	Four channel image formats, where the first three channels represent RED, GREEN, and BLUE components in the sRGB color space. The fourth channel represents an ALPHA component.
CL_sRGBx missing before version 2.0.	A four channel image format, where the three channels represent RED, GREEN, and BLUE components in the sRGB color space. The fourth channel is ignored.

Table 25. List of supported Image Channel Data Types

Image Channel Data Type	Description
CL_SNORM_INT8	Each channel component is a normalized signed 8-bit integer value
CL_SNORM_INT16	Each channel component is a normalized signed 16-bit integer value
CL_UNORM_INT8	Each channel component is a normalized unsigned 8-bit integer value

Image Channel Data Type	Description
CL_UNORM_INT16 Also supported if the cl_khr_depth_images extension is supported.	Each channel component is a normalized unsigned 16-bit integer value
CL_UNORM_SHORT_565	Represents a normalized 5-6-5 3-channel RGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_SHORT_555	Represents a normalized x-5-5-5 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_INT_101010	Represents a normalized x-10-10-10 4-channel xRGB image. The channel order must be CL_RGB or CL_RGBx.
CL_UNORM_INT_101010_2 missing before version 2.1.	Represents a normalized 10-10-10-2 four-channel RGBA image. The channel order must be CL_RGBA.
CL_SIGNED_INT8	Each channel component is an unnormalized signed 8-bit integer value
CL_UNORM_INT_2_101010_EXT provided by the cl_ext_image_unorm_int_2_101010 extension.	Represents a normalized 2-10-10-10 four-channel ABGR image. The channel order must be CL_ABGR.
CL_SIGNED_INT16	Each channel component is an unnormalized signed 16-bit integer value
CL_SIGNED_INT32	Each channel component is an unnormalized signed 32-bit integer value
CL_UNSIGNED_INT8	Each channel component is an unnormalized unsigned 8-bit integer value
CL_UNSIGNED_INT16	Each channel component is an unnormalized unsigned 16-bit integer value
CL_UNORM_INT24 provided by the cl_khr_gl_depth_images extension.	Each channel component is a normalized unsigned 24-bit integer value
CL_UNSIGNED_INT32	Each channel component is an unnormalized unsigned 32-bit integer value
CL_HALF_FLOAT	Each channel component is a 16-bit half-float value

Image Channel Data Type	Description
CL_FLOAT Also supported if the cl_khr_depth_images extension is supported.	Each channel component is a single precision floating-point value
CL_UNSIGNED_INT10X6_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is an unnormalized unsigned 10-bit integer value stored in bits 15:6 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.
CL_UNSIGNED_INT12X4_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is an unnormalized unsigned 12-bit integer value stored in bits 15:4 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.
CL_UNSIGNED_INT14X2_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is an unnormalized unsigned 14-bit integer value stored in bits 15:2 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.
CL_UNORM_INT10X6_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is a normalized unsigned 10-bit integer value stored in bits 15:6 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.
CL_UNORM_INT12X4_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is a normalized unsigned 12-bit integer value stored in bits 15:4 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.
CL_UNORM_INT14X2_EXT provided by the cl_ext_image_unsigned_10x6_12x4_14x2 extension.	Each channel component is a normalized unsigned 14-bit integer value stored in bits 15:2 of a 16-bit location. The channel order must be CL_R, CL_RG, or CL_RGBA.

For example, to specify a normalized unsigned 8-bit / channel RGBA image, image_channel_order = CL_RGBA, and image_channel_data_type = CL_UNORM_INT8. The memory layout of this image format is described below:

R	G	B	A	...

with the corresponding byte offsets

0	1	2	3	...

Similar, if image_channel_order = CL_RGBA and image_channel_data_type = CL_SIGNED_INT16, the memory layout of this image format is described below:

R	G	B	A	...

with the corresponding byte offsets

0	2	4	6	...

`image_channel_data_type` values of `CL_UNORM_SHORT_565`, `CL_UNORM_SHORT_555`, `CL_UNORM_INT_101010`, and `CL_UNORM_INT_101010_2` are special cases of packed image formats where the channels of each element are packed into a single unsigned short or unsigned int. For these special packed image formats, the channels are normally packed with the first channel in the most significant bits of the bitfield, and successive channels occupying progressively less significant locations. For `CL_UNORM_SHORT_565`, R is in bits 15:11, G is in bits 10:5 and B is in bits 4:0. For `CL_UNORM_SHORT_555`, bit 15 is undefined, R is in bits 14:10, G in bits 9:5 and B in bits 4:0. For `CL_UNORM_INT_101010`, bits 31:30 are undefined, R is in bits 29:20, G in bits 19:10 and B in bits 9:0. For `CL_UNORM_INT_101010_2`, R is in bits 31:22, G in bits 21:12, B in bits 11:2 and A in bits 1:0. For `CL_UNORM_INT_2_101010_EXT`, A is in bits 31:30, B in bits 29:20, G in bits 19:10 and R in bits 9:0.

OpenCL implementations must maintain the minimum precision specified by the number of bits in `image_channel_data_type`. If the image format specified by `image_channel_order`, and `image_channel_data_type` cannot be supported by the OpenCL implementation, then the call to **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D** will return a `NULL` memory object.

5.3.1.2. Image Descriptor

The `cl_image_desc` image descriptor structure describes the image type and dimensions of an image or image array when creating an image using **clCreateImage** or **clCreateImageWithProperties**, and is defined as:

```
// Provided by CL_VERSION_1_2
typedef struct cl_image_desc {
    cl_mem_object_type    image_type;
    size_t                image_width;
    size_t                image_height;
    size_t                image_depth;
    size_t                image_array_size;
    size_t                image_row_pitch;
    size_t                image_slice_pitch;
    cl_uint               num_mip_levels;
    cl_uint               num_samples;
    union {
        cl_mem buffer;
        cl_mem mem_object;
    };
} cl_image_desc;
```

- *image_type* describes the image type and must be either `CL_MEM_OBJECT_IMAGE1D`, `CL_MEM_OBJECT_IMAGE1D_BUFFER`, `CL_MEM_OBJECT_IMAGE1D_ARRAY`, `CL_MEM_OBJECT_IMAGE2D`, `CL_MEM_OBJECT_IMAGE2D_ARRAY`, or `CL_MEM_OBJECT_IMAGE3D`.
- *image_width* is the width of the image in pixels. For a 1D image, 1D image array, 2D image, or 2D

image array, the image width must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE2D_MAX_WIDTH for all devices in the context. For a 3D image, the image width must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE3D_MAX_WIDTH for all devices in the context. For a 1D image buffer, the image width must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE_MAX_BUFFER_SIZE for all devices in the context.

- *image_height* is the height of the image in pixels. It is only used if the image is a 2D image, 2D image array, or 3D image. For a 2D image or 2D image array, the image height must greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE2D_MAX_HEIGHT for all devices in the context. For a 3D image, the image height must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE3D_MAX_HEIGHT for all devices in the context.
- *image_depth* is the depth of the image in pixels. It is only used if the image is a 3D image. For a 3D image, the image depth must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE3D_MAX_DEPTH for all devices in the context.
- *image_array_size* [4] is the number of images in the image array. It is only used if the image is a 1D image array or 2D image array. For a 1D image array or 2D image array, the image array size must be greater than or equal to one and less than or equal to the value returned for CL_DEVICE_IMAGE_MAX_ARRAY_SIZE for all devices in the context.
- *image_row_pitch* is the scan-line pitch in bytes. The *image_row_pitch* must be zero if *host_ptr* is NULL, the image is not an image created from an external memory handle, and the image is not a 2D image created from a buffer.
  If *image_row_pitch* is zero and *host_ptr* is not NULL, then the image row pitch is calculated as *image_width* times the size of an image element in bytes.
  If *image_row_pitch* is zero and the image is created from an external memory handle, then the image row pitch is implementation-defined.
  If *image_row_pitch* is non-zero, then it must be greater than or equal to the *image_width* times the size of an image element in bytes, and must be a multiple of the size of an image element in bytes.
  For an image created from a buffer, the *image_row_pitch* must also:
  - be a multiple of the CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT value for the *image_format*, *image_type*, and *flags* used to create the image, if the cl_ext_image_requirements_info extension is supported, or
  - be a multiple of the maximum of the CL_DEVICE_IMAGE_PITCH_ALIGNMENT value for all devices in the context that support images, otherwise.
- *image_slice_pitch* is the size in bytes of each 2D slice in a 3D image, or the size in bytes of each image in a 1D or 2D image array. The *image_slice_pitch* must be zero if *host_ptr* is NULL and the image is not an image created from an external memory handle, If *image_slice_pitch* is zero and *host_ptr* is not NULL then the image slice pitch is calculated as the image row pitch × *image_height* for a 2D image array or a 3D image, and as the image row pitch for a 1D image array. If *image_slice_pitch* is zero and the image is created from an external memory handle, then the image slice pitch is implementation-defined. The image slice pitch must be ≥ the image row pitch × *image_height* for a 2D image array or a 3D image, must be ≥ the image row pitch for a 1D image array, and must be a multiple of the image row pitch. For an image created from a buffer, the image slice pitch must be a multiple of the CL_IMAGE_REQUIREMENTS_SLICE_PITCH_

ALIGNMENT_EXT value for the *image_format*, *image_type* and *flags* used to create the image, if the cl_ext_image_from_buffer extension is supported.

- *num_mip_levels* must be 0, indicating that the image has a single mipmap level, unless the cl_khr_mipmap_image extension is supported. When the cl_khr_mipmap_image extension is supported, *num_mip_levels* may additionally specify the total number of mipmap levels in the image, including the base level [5].
- *num_samples* must be 0.
- *mem_object* may refer to a valid buffer or image memory object. mem_object can be a buffer memory object if image_type is CL_MEM_OBJECT_IMAGE1D_BUFFER or CL_MEM_OBJECT_IMAGE2D , or any other image_type if the cl_ext_image_from_buffer extension is supported. [6] mem_object can be an image object if image_type is CL_MEM_OBJECT_IMAGE2D [7]. Otherwise it must be NULL. The image pixels are taken from the memory objects data store. When the contents of the specified memory objects data store are modified, those changes are reflected in the contents of the image object and vice-versa at corresponding synchronization points.

For a 1D image buffer created from a buffer object, the image_width × size of element in bytes must be ≤ size of the buffer object. The image data in the buffer object is stored as a single scanline which is a linear sequence of adjacent elements.

For a 1D image created from a buffer object, the image_width × size of element in bytes must be ≤ size of the buffer object. The image data in the buffer object is stored as a single scanline which is a linear sequence of adjacent elements.

For a 1D image array created from a buffer object, the image_slice_pitch × image_array_size must be ≤ size of the buffer object specified by mem_object. The image data in the buffer object is stored as a linear sequence of adjacent 1D slices. Each slice is a single scanline padded to image_slice_pitch bytes. Each scanline is a linear sequence of image elements.

For a 2D image created from a buffer object, the image_row_pitch × image_height must be ≤ size of the buffer object specified by mem_object. The image data in the buffer object is stored as a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements padded to image_row_pitch bytes.

For a 2D image array created from a buffer object, the image_slice_pitch × image_array_size must be ≤ size of the buffer object specified by mem_object. The image data in the buffer object is stored as a linear sequence of adjacent 2D slices. Each slice is a linear sequence of adjacent scanlines padded to image_slice_pitch bytes. Each scanline is a linear sequence of image elements padded to image_row_pitch bytes.

For a 3D image created from a buffer object, the image_slice_pitch × image_depth must be ≤ size of the buffer object specified by mem_object. The image data in the buffer object is stored as a linear sequence of adjacent 2D slices padded to image_slice_pitch bytes. Each slice is a linear sequence of adjacent scanlines. Each scanline is a linear sequence of image elements padded to image_row_pitch bytes.

For an image object created from another image object, the values specified in the image descriptor except for mem_object must match the image descriptor information associated with mem_object.

Image elements are stored according to their image format as described in Image Format Descriptor.

If the buffer object specified by `mem_object` was created with `CL_MEM_USE_HOST_PTR`, the *host_ptr* specified to **clCreateBuffer** or **clCreateBufferWithProperties** must

- Be aligned to the `CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT` value for the returned for parameters compatible with those used to create the image, if the `cl_ext_image_requirements_info` extension is supported, or
- Be aligned to the maximum of the `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT` value for all devices in the context associated with the buffer specified by `mem_object` that support images, otherwise.

Creating a 2D image object from another 2D image object creates a new 2D image object that shares the image data store with `mem_object` but views the pixels in the image with a different image channel order. Restrictions are:

- All of the values specified in *image_desc* must match the image descriptor information associated with `mem_object`, except for `mem_object`.
- The image channel data type specified in *image_format* must match the image channel data type associated with `mem_object`.
- The image channel order specified in *image_format* must be compatible with the image channel order associated with `mem_object`, as described in the Compatible Image Channel Orders table.

The image channel order compatibility constraint allows creation of a sRGB view of the image from a linear RGB view or vice-versa, i.e. the pixels stored in the image can be accessed as linear RGB or sRGB values.

Table 26. Compatible Image Channel Orders

Image Channel Order in *image_format*:	**Image Channel Order associated with** `mem_object`:
`CL_sBGRA`	`CL_BGRA`
`CL_BGRA`	`CL_sBGRA`
`CL_sRGBA`	`CL_RGBA`
`CL_RGBA`	`CL_sRGBA`
`CL_sRGB`	`CL_RGB`
`CL_RGB`	`CL_sRGB`
`CL_sRGBx`	`CL_RGBx`
`CL_RGBx`	`CL_sRGBx`
`CL_DEPTH`	`CL_R`

Concurrent reading from, writing to and copying between both a buffer object and buffer or image object associated with the buffer object is undefined. Only reading from both a buffer object and buffer or image object associated with the buffer object is defined.

> Writing to an image created from a buffer and then reading from this buffer in a kernel even if appropriate synchronization operations (such as a barrier) are performed between the writes and reads is undefined. Similarly, writing to the buffer and reading from the image created from this buffer with appropriate synchronization between the writes and reads is undefined.

5.3.2. Querying List of Supported Image Formats

To get the list of image formats supported by an OpenCL implementation for a specified context, image type, and allocation information, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetSupportedImageFormats(
    cl_context context,
    cl_mem_flags flags,
    cl_mem_object_type image_type,
    cl_uint num_entries,
    cl_image_format* image_formats,
    cl_uint* num_image_formats);
```

- *context* is a valid OpenCL context on which the image object(s) will be created.
- *flags* is a bit-field that is used to specify usage information about the image formats being queried and is described in the Memory Flags table. *flags* may be `CL_MEM_READ_WRITE` to query image formats that may be read from and written to by different kernel-instances when correctly ordered by event dependencies, or `CL_MEM_READ_ONLY` to query image formats that may be read from by a kernel, or `CL_MEM_WRITE_ONLY` to query image formats that may be written to by a kernel, or `CL_MEM_KERNEL_READ_AND_WRITE` to query image formats that may be both read from and written to by the same kernel instance. *flags* may also be `CL_MEM_IMMUTABLE_EXT` to query image formats that may be used to create immutable images. Please see Image Format Mapping for clarification.
- *image_type* describes the image type and must be either `CL_MEM_OBJECT_IMAGE1D`, `CL_MEM_OBJECT_IMAGE1D_BUFFER`, `CL_MEM_OBJECT_IMAGE2D`, `CL_MEM_OBJECT_IMAGE3D`, `CL_MEM_OBJECT_IMAGE1D_ARRAY`, or `CL_MEM_OBJECT_IMAGE2D_ARRAY`.
- *num_entries* specifies the number of entries that can be returned in the memory location given by *image_formats*.
- *image_formats* is a pointer to a memory location where the list of supported image formats are returned. Each entry describes a `cl_image_format` structure supported by the OpenCL implementation. If *image_formats* is `NULL`, it is ignored.
- *num_image_formats* is the actual number of supported image formats for a specific *context* and values specified by *flags*. If *num_image_formats* is `NULL`, it is ignored.

clGetSupportedImageFormats returns a union of image formats supported by all devices in the context.

clGetSupportedImageFormats returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_CONTEXT
  - if *context* is not a valid context
- CL_INVALID_VALUE
  - if *flags* is not valid
  - if *image_type* is not valid
  - if *num_entries* is zero and *image_formats* is not NULL
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

If CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_TRUE, the values assigned to CL_DEVICE_MAX_READ_IMAGE_ARGS, CL_DEVICE_MAX_WRITE_IMAGE_ARGS CL_DEVICE_IMAGE2D_MAX_WIDTH, CL_DEVICE_IMAGE2D_MAX_HEIGHT CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT CL_DEVICE_IMAGE3D_MAX_DEPTH, and CL_DEVICE_MAX_SAMPLERS by the implementation must be greater than or equal to the minimum values specified in the Device Queries table.

5.3.2.1. Minimum List of Supported Image Formats

The tables below describe the required minimum lists of supported image formats. To query all image formats supported by an implementation, call the function **clGetSupportedImageFormats**.

For full profile devices supporting OpenCL 2.0, 2.1, or 2.2, the minimum list of supported image formats for either reading or writing in a kernel is:

Table 27. Minimum list of supported image formats for reading or writing (OpenCL 2.0, 2.1, or 2.2)

num_channels	channel_order	channel_data_type
1	CL_R	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

num_channels	channel_order	channel_data_type
1	CL_DEPTH [8] Also supported if the cl_khr_depth_images extension is supported.	CL_UNORM_INT16 CL_FLOAT
1	CL_DEPTH_STENCIL	CL_UNORM_INT24 CL_FLOAT See Restrictions on Depth/Stencil Images.
2	CL_RG	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SNORM_INT8 CL_SNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8
4	CL_sRGBA [9]	CL_UNORM_INT8

For full profile devices supporting other OpenCL versions, such as OpenCL 1.2, OpenCL 3.0, or OpenCL 3.1, the minimum list of supported image formats for either reading or writing in a kernel is:

Table 28. Minimum list of required image formats for reading or writing

num_channels	channel_order	channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_BGRA	CL_UNORM_INT8

For full profile devices that support reading from and writing to the same image object from the same kernel-instance (see CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS), the minimum list of supported image formats for reading and writing in the same kernel-instance is:

Table 29. Minimum list of required image formats for reading and writing

num_channels	channel_order	channel_data_type
1	CL_R	CL_UNORM_INT8 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT
4	CL_RGBA	CL_UNORM_INT8 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

5.3.2.2. Image Format Mapping to OpenCL Kernel Language Image Access Qualifiers

Image arguments to kernels may have the read_only, write_only or read_write qualifier. Not all image formats supported by the device and platform are valid to be passed to all of these access qualifiers. For each access qualifier, only images whose format is in the list of formats returned by **clGetSupportedImageFormats** with the given flag arguments in the Image Format Mapping table are permitted.

Table 30. Mapping from format flags passed to clGetSupportedImageFormats to OpenCL kernel language image access qualifiers

Access Qualifier	Memory Flags
`read_only`	`CL_MEM_READ_ONLY`, `CL_MEM_READ_WRITE`, `CL_MEM_KERNEL_READ_AND_WRITE` Additionally, if the `cl_ext_immutable_memory_objects` extension is supported: `CL_MEM_IMMUTABLE_EXT`
`write_only`	`CL_MEM_WRITE_ONLY`, `CL_MEM_READ_WRITE`, `CL_MEM_KERNEL_READ_AND_WRITE`
`read_write`	`CL_MEM_KERNEL_READ_AND_WRITE`

5.3.3. Mapping to External Image Formats

OpenCL image objects can be created which share storage with image objects in external APIs such as DirectX and OpenGL when the corresponding OpenCL extensions are supported. When creating such OpenCL images, there are restrictions on the allowed formats. The tables in this section list, for each such external API, the supported image formats in that API and the corresponding OpenCL image format.

5.3.3.1. Image Formats for DirectX 9 Media Surface Sharing

When the `cl_khr_dx9_media_sharing` extension is supported, image objects sharing storage with Direct3D 9 surfaces can be created. This section describes the Direct3D 9 surface formats that are supported when the adapter type is one of the Direct 3D lineage. Using a Direct3D 9 surface format not listed here is an error. To extend the use of this extension to support media adapters beyond DirectX 9 tables similar to the ones in this section will need to be defined for the surface formats supported by the new media adapter. All implementations that support this extension are required to support the NV12 surface format. The other surface formats supported are the same surface formats that the adapter you are sharing with supports as long as they are listed in the YUV FourCC Codes and Corresponding OpenCL Image Formats or Direct3D 9 Formats and Corresponding OpenCL Image Formats tables.

Table 31. YUV FourCC Codes and Corresponding OpenCL Image Formats

FOUR CC Code	CL Image Format (Channel Order, Channel Data Type)
FOURCC('N','V','1','2'), Plane 0	`CL_R`, `CL_UNORM_INT8`
FOURCC('N','V','1','2'), Plane 1	`CL_RG`, `CL_UNORM_INT8`
FOURCC('Y','V','1','2'), Plane 0	`CL_R`, `CL_UNORM_INT8`
FOURCC('Y','V','1','2'), Plane 1	`CL_R`, `CL_UNORM_INT8`
FOURCC('Y','V','1','2'), Plane 2	`CL_R`, `CL_UNORM_INT8`

In the YUV FourCC Codes and Corresponding OpenCL Image Formats table, NV12 Plane 0 corresponds to the luminance (Y) channel and Plane 1 corresponds to the UV channels. The YV12

Plane 0 corresponds to the Y channel, Plane 1 corresponds to the V channel and Plane 2 corresponds to the U channel. Note that the YUV formats map to `CL_R` and `CL_RG` but do not perform any YUV to RGB conversion, and vice-versa.

Table 32. Direct3D 9 Formats and Corresponding OpenCL Image Formats

Direct3D 9 Format	CL Image Format (Channel Order, Channel Data Type)
`D3DFMT_R32F`	`CL_R`, `CL_FLOAT`
`D3DFMT_R16F`	`CL_R`, `CL_HALF_FLOAT`
`D3DFMT_L16`	`CL_R`, `CL_UNORM_INT16`
`D3DFMT_A8`	`CL_A`, `CL_UNORM_INT8`
`D3DFMT_L8`	`CL_R`, `CL_UNORM_INT8`
`D3DFMT_G32R32F`	`CL_RG`, `CL_FLOAT`
`D3DFMT_G16R16F`	`CL_RG`, `CL_HALF_FLOAT`
`D3DFMT_G16R16`	`CL_RG`, `CL_UNORM_INT16`
`D3DFMT_A8L8`	`CL_RG`, `CL_UNORM_INT8`
`D3DFMT_A32B32G32R32F`	`CL_RGBA`, `CL_FLOAT`
`D3DFMT_A16B16G16R16F`	`CL_RGBA`, `CL_HALF_FLOAT`
`D3DFMT_A16B16G16R16`	`CL_RGBA`, `CL_UNORM_INT16`
`D3DFMT_A8B8G8R8`	`CL_RGBA`, `CL_UNORM_INT8`
`D3DFMT_X8B8G8R8`	`CL_RGBA`, `CL_UNORM_INT8`
`D3DFMT_A8R8G8B8`	`CL_BGRA`, `CL_UNORM_INT8`
`D3DFMT_X8R8G8B8`	`CL_BGRA`, `CL_UNORM_INT8`

The Direct3D 9 format names in the table above seem to imply that the order of the color channels are switched relative to OpenCL, but this is not the case. For example, the layout of channels for each pixel for `D3DFMT_A32FB32FG32FR32F` is the same as `CL_RGBA`, `CL_FLOAT`.

5.3.3.2. Image Formats for Direct3D Texture Sharing

When the `cl_khr_d3d10_sharing` or `cl_khr_d3d11_sharing` extensions are supported, image objects sharing storage with Direct3D 10 and Direct3D 11 textures, respectively, can be created. The DXGI Formats and Corresponding OpenCL Image Formats table describes the supported DirectX Graphics Infrastructure (DXGI) texture formats.

Table 33. DXGI Formats and Corresponding OpenCL Image Formats

DXGI Format	CL Image Format (Channel Order, Channel Data Type)
DXGI_FORMAT_R32G32B32A32_FLOAT	CL_RGBA, CL_FLOAT
DXGI_FORMAT_R32G32B32A32_UINT	CL_RGBA, CL_UNSIGNED_INT32
DXGI_FORMAT_R32G32B32A32_SINT	CL_RGBA, CL_SIGNED_INT32
DXGI_FORMAT_R16G16B16A16_FLOAT	CL_RGBA, CL_HALF_FLOAT
DXGI_FORMAT_R16G16B16A16_UNORM	CL_RGBA, CL_UNORM_INT16
DXGI_FORMAT_R16G16B16A16_UINT	CL_RGBA, CL_UNSIGNED_INT16
DXGI_FORMAT_R16G16B16A16_SNORM	CL_RGBA, CL_SNORM_INT16
DXGI_FORMAT_R16G16B16A16_SINT	CL_RGBA, CL_SIGNED_INT16
DXGI_FORMAT_B8G8R8A8_UNORM	CL_BGRA, CL_UNORM_INT8
DXGI_FORMAT_R8G8B8A8_UNORM	CL_RGBA, CL_UNORM_INT8
DXGI_FORMAT_R8G8B8A8_UINT	CL_RGBA, CL_UNSIGNED_INT8
DXGI_FORMAT_R8G8B8A8_SNORM	CL_RGBA, CL_SNORM_INT8
DXGI_FORMAT_R8G8B8A8_SINT	CL_RGBA, CL_SIGNED_INT8
DXGI_FORMAT_R32G32_FLOAT	CL_RG, CL_FLOAT
DXGI_FORMAT_R32G32_UINT	CL_RG, CL_UNSIGNED_INT32
DXGI_FORMAT_R32G32_SINT	CL_RG, CL_SIGNED_INT32
DXGI_FORMAT_R16G16_FLOAT	CL_RG, CL_HALF_FLOAT
DXGI_FORMAT_R16G16_UNORM	CL_RG, CL_UNORM_INT16
DXGI_FORMAT_R16G16_UINT	CL_RG, CL_UNSIGNED_INT16
DXGI_FORMAT_R16G16_SNORM	CL_RG, CL_SNORM_INT16
DXGI_FORMAT_R16G16_SINT	CL_RG, CL_SIGNED_INT16
DXGI_FORMAT_R8G8_UNORM	CL_RG, CL_UNORM_INT8
DXGI_FORMAT_R8G8_UINT	CL_RG, CL_UNSIGNED_INT8
DXGI_FORMAT_R8G8_SNORM	CL_RG, CL_SNORM_INT8
DXGI_FORMAT_R8G8_SINT	CL_RG, CL_SIGNED_INT8
DXGI_FORMAT_R32_FLOAT	CL_R, CL_FLOAT
DXGI_FORMAT_R32_UINT	CL_R, CL_UNSIGNED_INT32
DXGI_FORMAT_R32_SINT	CL_R, CL_SIGNED_INT32
DXGI_FORMAT_R16_FLOAT	CL_R, CL_HALF_FLOAT
DXGI_FORMAT_R16_UNORM	CL_R, CL_UNORM_INT16

DXGI Format	CL Image Format (Channel Order, Channel Data Type)
DXGI_FORMAT_R16_UINT	CL_R, CL_UNSIGNED_INT16
DXGI_FORMAT_R16_SNORM	CL_R, CL_SNORM_INT16
DXGI_FORMAT_R16_SINT	CL_R, CL_SIGNED_INT16
DXGI_FORMAT_R8_UNORM	CL_R, CL_UNORM_INT8
DXGI_FORMAT_R8_UINT	CL_R, CL_UNSIGNED_INT8
DXGI_FORMAT_R8_SNORM	CL_R, CL_SNORM_INT8
DXGI_FORMAT_R8_SINT	CL_R, CL_SIGNED_INT8

5.3.3.3. Image Formats for OpenGL Texture and Renderbuffer Sharing

When the cl_khr_gl_sharing extension is supported, image objects sharing storage with OpenGL texture and renderbuffer objects can be created. The OpenGL Internal Formats and Corresponding OpenCL Internal Formats table describes the supported OpenGL image formats. If an OpenGL texture or renderbuffer object with an internal format from the table is successfully created by OpenGL, then there is guaranteed to be a mapping to one of the corresponding OpenCL image format(s) in the table. Texture and renderbuffer objects created with other OpenGL internal formats may (but are not guaranteed to) have a mapping to an OpenCL image format. If such mappings exist, they are guaranteed to preserve all color components, data types, and at least the number of bits/component actually allocated by OpenGL for that format.

Table 34. OpenGL Internal Formats and Corresponding OpenCL Internal Formats

OpenGL internal format	OpenCL Image Format (Channel Order, Channel Data Type)
GL_RGBA8	CL_RGBA, CL_UNORM_INT8 or CL_BGRA, CL_UNORM_INT8
GL_SRGB8_ALPHA8	CL_sRGBA, CL_UNORM_INT8
GL_RGBA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_RGBA, CL_UNORM_INT8
GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV	CL_BGRA, CL_UNORM_INT8
GL_RGBA8I, GL_RGBA8I_EXT	CL_RGBA, CL_SIGNED_INT8
GL_RGBA16I, GL_RGBA16I_EXT	CL_RGBA, CL_SIGNED_INT16
GL_RGBA32I, GL_RGBA32I_EXT	CL_RGBA, CL_SIGNED_INT32
GL_RGBA8UI, GL_RGBA8UI_EXT	CL_RGBA, CL_UNSIGNED_INT8
GL_RGBA16UI, GL_RGBA16UI_EXT	CL_RGBA, CL_UNSIGNED_INT16
GL_RGBA32UI, GL_RGBA32UI_EXT	CL_RGBA, CL_UNSIGNED_INT32
GL_RGBA8_SNORM	CL_RGBA, CL_SNORM_INT8
GL_RGBA16	CL_RGBA, CL_UNORM_INT16

OpenGL internal format	OpenCL Image Format (Channel Order, Channel Data Type)
GL_RGBA16_SNORM	CL_RGBA, CL_SNORM_INT16
GL_RGBA16F, GL_RGBA16F_ARB	CL_RGBA, CL_HALF_FLOAT
GL_RGBA32F, GL_RGBA32F_ARB	CL_RGBA, CL_FLOAT
GL_R8	CL_R, CL_UNORM_INT8
GL_R8_SNORM	CL_R, CL_SNORM_INT8
GL_R16	CL_R, CL_UNORM_INT16
GL_R16_SNORM	CL_R, CL_SNORM_INT16
GL_R16F	CL_R, CL_HALF_FLOAT
GL_R32F	CL_R, CL_FLOAT
GL_R8I	CL_R, CL_SIGNED_INT8
GL_R16I	CL_R, CL_SIGNED_INT16
GL_R32I	CL_R, CL_SIGNED_INT32
GL_R8UI	CL_R, CL_UNSIGNED_INT8
GL_R16UI	CL_R, CL_UNSIGNED_INT16
GL_R32UI	CL_R, CL_UNSIGNED_INT32
GL_RG8	CL_RG, CL_UNORM_INT8
GL_RG8_SNORM	CL_RG, CL_SNORM_INT8
GL_RG16	CL_RG, CL_UNORM_INT16
GL_RG16_SNORM	CL_RG, CL_SNORM_INT16
GL_RG16F	CL_RG, CL_HALF_FLOAT
GL_RG32F	CL_RG, CL_FLOAT
GL_RG8I	CL_RG, CL_SIGNED_INT8
GL_RG16I	CL_RG, CL_SIGNED_INT16
GL_RG32I	CL_RG, CL_SIGNED_INT32
GL_RG8UI	CL_RG, CL_UNSIGNED_INT8
GL_RG16UI	CL_RG, CL_UNSIGNED_INT16
GL_RG32UI	CL_RG, CL_UNSIGNED_INT32
GL_DEPTH_COMPONENT32F	CL_DEPTH, CL_FLOAT
GL_DEPTH_COMPONENT16	CL_DEPTH, CL_UNORM_INT16
GL_DEPTH24_STENCIL8	CL_DEPTH_STENCIL, CL_UNORM_INT24
GL_DEPTH32F_STENCIL8	CL_DEPTH_STENCIL, CL_FLOAT

5.3.4. Reading, Writing and Copying Image Objects

The following functions enqueue commands to read from an image or image array object to host memory or write to an image or image array object from host memory.

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueReadImage(
    cl_command_queue command_queue,
    cl_mem image,
    cl_bool blocking_read,
    const size_t* origin,
    const size_t* region,
    size_t row_pitch,
    size_t slice_pitch,
    void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueWriteImage(
    cl_command_queue command_queue,
    cl_mem image,
    cl_bool blocking_write,
    const size_t* origin,
    const size_t* region,
    size_t input_row_pitch,
    size_t input_slice_pitch,
    const void* ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* refers to the host command-queue in which the read / write command will be queued. *command_queue* and *image* must be created with the same OpenCL context.
- *image* refers to a valid image or image array object.
- *blocking_read* and *blocking_write* indicate if the read and write operations are *blocking* or *non-blocking*.
- *origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array.
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (

width) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0. If *image* is a mipmapped image, the mip level to read or write is determined from *origin* as described in Specifying Mipmap Levels to Image Operations

- *row_pitch* in **clEnqueueReadImage** and *input_row_pitch* in **clEnqueueWriteImage** is the length of each row in bytes. This value must be greater than or equal to the element size in bytes × *width*. If *row_pitch* (or *input_row_pitch*) is set to 0, the appropriate row pitch is calculated based on the size of each element in bytes multiplied by *width*.
- *slice_pitch* in **clEnqueueReadImage** and *input_slice_pitch* in **clEnqueueWriteImage** is the size in bytes of the 2D slice of the 3D region of a 3D image or each image of a 1D or 2D image array being read or written respectively. This must be 0 if *image* is a 1D or 2D image. Otherwise this value must be greater than or equal to *row_pitch* × *height*. If *slice_pitch* (or *input_slice_pitch*) is set to 0, the appropriate slice pitch is calculated based on the *row_pitch* × *height*.
- *ptr* is the pointer to a buffer in host memory where image data is to be read from or to be written to. The alignment requirements for ptr are specified in Alignment of Application Data Types.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this read / write command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If *blocking_read* is `CL_TRUE` i.e. the read command is blocking, **clEnqueueReadImage** does not return until the buffer data has been read and copied into memory pointed to by *ptr*.

If *blocking_read* is `CL_FALSE` i.e. the read command is non-blocking, **clEnqueueReadImage** queues a non-blocking read command and returns. The contents of the buffer that *ptr* points to cannot be used until the read command has completed. The *event* argument returns an event object which can be used to query the execution status of the read command. When the read command has completed, the contents of the buffer that *ptr* points to can be used by the application.

If *blocking_write* is `CL_TRUE`, the write command is blocking and does not return until the command is complete, including transfer of the data. The memory pointed to by *ptr* can be reused by the application after the **clEnqueueWriteImage** call returns.

If *blocking_write* is `CL_FALSE`, the OpenCL implementation will use *ptr* to perform a non-blocking write. As the write is non-blocking the implementation can return immediately. The memory pointed to by *ptr* cannot be reused by the application after the call returns. The *event* argument

returns an event object which can be used to query the execution status of the write command. When the write command has completed, the memory pointed to by *ptr* can then be reused by the application.

clEnqueueReadImage and **clEnqueueWriteImage** return `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *image* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *image* is not a valid image object
- `CL_INVALID_VALUE`
  - if *origin* is `NULL`
  - if *region* is `NULL`
  - if the region being read or written specified by *origin* and *region* is out of bounds
  - if values in *origin* and *region* do not follow the rules described in the argument description for *origin* and *region*
  - if *image* is a 1D or 2D image and *slice_pitch* or *input_slice_pitch* is not zero
  - if *ptr* is `NULL`
- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *origin* is not a valid level for *image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_INVALID_IMAGE_SIZE`
  - if the dimensions for *image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if the format for *image*, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the read or write operations are blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error. This error code is missing before version 1.1.

- CL_MEM_OBJECT_ALLOCATION_FAILURE
  - if there is a failure to allocate memory for the data store associated with *image*
- CL_INVALID_OPERATION
  - if the device associated with *command_queue* does not support images (the query for CL_DEVICE_IMAGE_SUPPORT returns CL_FALSE)
  - if **clEnqueueReadImage** is called on *image* which has been created with CL_MEM_HOST_WRITE_ONLY or CL_MEM_HOST_NO_ACCESS
  - if **clEnqueueWriteImage** is called on *image* which has been created with CL_MEM_HOST_READ_ONLY or CL_MEM_HOST_NO_ACCESS
  - if **clEnqueueWriteImage** is called on *image* which has been created with CL_MEM_IMMUTABLE_EXT
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

Calling **clEnqueueReadImage** to read a region of the *image* with the *ptr* argument value set to *host_ptr* + (*origin*[2] × *image slice pitch* + *origin*[1] × *image row pitch* + *origin*[0] × *bytes per pixel*), where *host_ptr* is a pointer to the memory region specified when the *image* being read is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- All commands that use this image object have finished execution before the read command begins execution.
- The *row_pitch* and *slice_pitch* argument values in **clEnqueueReadImage** must be set to the image row pitch and slice pitch.
- The image object is not mapped.
- The image object is not used by any command-queue until the read command has finished execution.

Calling **clEnqueueWriteImage** to update the latest bits in a region of the *image* with the *ptr* argument value set to *host_ptr* + (*origin*[2] × *image slice pitch* + *origin*[1] × *image row pitch* + *origin*[0] × *bytes per pixel*), where *host_ptr* is a pointer to the memory region specified when the *image* being written is created with CL_MEM_USE_HOST_PTR, must meet the following requirements in order to avoid undefined behavior:

- The host memory region being written contains the latest bits when the enqueued write command begins execution.
- The *input_row_pitch* and *input_slice_pitch* argument values in **clEnqueueWriteImage** must be set to the image row pitch and slice pitch.
- The image object is not mapped.

- The image object is not used by any command-queue until the write command has finished execution.

To enqueue a command to copy image objects, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueCopyImage(
    cl_command_queue command_queue,
    cl_mem src_image,
    cl_mem dst_image,
    const size_t* src_origin,
    const size_t* dst_origin,
    const size_t* region,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *src_image* and *dst_image* can be 1D, 2D, 3D image or a 1D, 2D image array objects. It is possible to copy subregions between any combinations of source and destination types, provided that the dimensions of the subregions are the same e.g., one can copy a rectangular region from a 2D image to a slice of a 3D image.
- *command_queue* refers to the host command-queue in which the copy command will be queued. The OpenCL context associated with *command_queue*, *src_image* and *dst_image* must be the same.
- *src_origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *src_origin*[2] must be 0. If *src_image* is a 1D image object, *src_origin*[1] and *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[1] describes the image index in the 1D image array. If *src_image* is a 2D image array object, *src_origin*[2] describes the image index in the 2D image array. If *src_image* is a mipmapped image, the mip level to read is determined from *src_origin* as described in Specifying Mipmap Levels to Image Operations
- *dst_origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *dst_image* is a 2D image object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image or 1D image buffer object, *dst_origin*[1] and *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[1] describes the image index in the 1D image array. If *dst_image* is a 2D image array object, *dst_origin*[2] describes the image index in the 2D image array. If *dst_image* is a mipmapped image, the mip level to write is determined from *dst_origin* as described in Specifying Mipmap Levels to Image Operations
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *src_image* or *dst_image* is a 2D image object, *region*[2] must be 1. If *src_image* or *dst_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *src_image* or *dst_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this copy command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

It is currently a requirement that the *src_image* and *dst_image* image memory objects for **clEnqueueCopyImage** must have the exact same image format (i.e. the `cl_image_format` descriptor specified when *src_image* and *dst_image* are created must match).

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_image* or *dst_image* is created is ignored by **clEnqueueCopyImage**.

clEnqueueCopyImage returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *src_image* are not the same
  - if the context associated with *command_queue* and *dst_image* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *src_image* is not a valid image object
  - if *dst_image* is not a valid image object
- `CL_IMAGE_FORMAT_MISMATCH`
  - if *src_image* and *dst_image* do not use the same image format
- `CL_INVALID_VALUE`
  - if *src_origin* is `NULL`
  - if *dst_origin* is `NULL`
  - if *region* is `NULL`
  - if the 2D or 3D rectangular region specified by *src_origin* and *region* refers to a region outside *src_image*

- if the 2D or 3D rectangular region specified by *dst_origin* and *region* refers to a region outside *dst_image*
- if values in *src_origin*, *dst_origin*, and *region* do not follow the rules described in the argument descriptions for *src_origin*, *dst_origin* and *region*

- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *src_origin* is not a valid level for *src_image*
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *dst_origin* is not a valid level for *dst_image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_INVALID_IMAGE_SIZE`
  - if the dimensions for *src_image* or *dst_image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if the format for *src_image* or *dst_image*, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- `CL_MEM_COPY_OVERLAP`
  - if *src_image* and *dst_image* are the same image object and the source and destination regions overlap
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *src_image* or *dst_image*
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support images (the query for `CL_DEVICE_IMAGE_SUPPORT` returns `CL_FALSE`)
  - if *dst_image* was created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.3.5. Filling Image Objects

Filling image objects is missing before version 1.2.

To enqueue a command to fill an image object with a specified color, call the function

```
// Provided by CL_VERSION_1_2
cl_int clEnqueueFillImage(
    cl_command_queue command_queue,
    cl_mem image,
    const void* fill_color,
    const size_t* origin,
    const size_t* region,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueFillImage is missing before version 1.2.

- *command_queue* refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and *image* must be the same.
- *image* is a valid image object.
- *fill_color* is the color used to fill the image. The fill color is a single floating-point value if the channel order is `CL_DEPTH`. Otherwise, the fill color is a four component RGBA floating-point color value if the *image* channel data type is not an unnormalized signed or unsigned integer type, is a four component signed integer value if the *image* channel data type is an unnormalized signed integer type and is a four component unsigned integer value if the *image* channel data type is an unnormalized unsigned integer type. The fill color will be converted to the appropriate image channel format and order associated with *image*.
- *origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array. If *image* is a mipmapped image, the mip level to fill is determined from *origin* as described in Specifying Mipmap Levels to Image Operations
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and

command_queue must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *image* is created is ignored by **clEnqueueFillImage**.

clEnqueueFillImage returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *image* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *image* is not a valid image object
- `CL_INVALID_VALUE`
  - if *fill_color* is `NULL`
  - if *origin* is `NULL`
  - if *region* is `NULL`
  - if the region being filled specified by *origin* and *region* is out of bounds
  - if values in *origin* and *region* do not follow the rules described in the argument description for *origin* and *region*
- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *origin* is not a valid level for *image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_INVALID_IMAGE_SIZE`
  - if the dimensions for *image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`

- if the format for *image*, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*

- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *image*
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support images (the query for `CL_DEVICE_IMAGE_SUPPORT` returns `CL_FALSE`)
  - if *image* was created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.3.6. Copying Between Image and Buffer Objects

To enqueue a command to copy an image object to a buffer object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueCopyImageToBuffer(
    cl_command_queue command_queue,
    cl_mem src_image,
    cl_mem dst_buffer,
    const size_t* src_origin,
    const size_t* region,
    size_t dst_offset,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* must be a valid host command-queue. The OpenCL context associated with *command_queue*, *src_image* and *dst_buffer* must be the same.
- *src_image* is a valid image object.
- *dst_buffer* is a valid buffer object.
- *src_origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *src_image* is a 2D image object, *src_origin*[2] must be 0. If *src_image* is a 1D image or 1D image buffer object, *src_origin*[1] and *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[2] must be 0. If *src_image* is a 1D image array object, *src_origin*[1] describes the image index in the 1D image array. If *src_image* is a 2D image array object, *src_origin*[2] describes the image index in the 2D image array. If *src_image* is a mipmapped image, the mip level to read is determined from *src_origin* as described in Specifying Mipmap Levels to Image Operations
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*,

height) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *src_image* is a 2D image object, *region*[2] must be 1. If *src_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *src_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.

- *dst_offset* refers to the offset where to begin copying data into *dst_buffer*. The size in bytes of the region to be copied referred to as *dst_cb* is computed as *width* × *height* × *depth* × *bytes/image element* if *src_image* is a 3D image object, is computed as *width* × *height* × *bytes/image element* if *src_image* is a 2D image, is computed as *width* × *height* × *arraysize* × *bytes/image element* if *src_image* is a 2D image array object, is computed as *width* × *bytes/image element* if *src_image* is a 1D image or 1D image buffer object and is computed as *width* × *arraysize* × *bytes/image element* if *src_image* is a 1D image array object.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this copy command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_image* or *dst_buffer* is created is ignored by **clEnqueueCopyImageToBuffer**.

clEnqueueCopyImageToBuffer returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *src_image* are not the same
  - if the context associated with *command_queue* and *dst_buffer* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *src_image* is not a valid image object
  - if *dst_buffer* is not a valid buffer object
  - if *src_image* is a 1D image buffer object created from *dst_buffer*

- `CL_INVALID_VALUE`
  - if *src_origin* is `NULL`
  - if *region* is `NULL`
  - if the 1D, 2D or 3D rectangular region specified by *src_origin* and *region* is out of bounds of *src_image*
  - if the region specified by *dst_offset* and the computed *dst_cb* is out of bounds of *dst_buffer*
  - if values in *src_origin* and *region* do not follow the rules described in the argument description for *src_origin* and *region*
- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *src_origin* is not a valid level for *src_image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if *dst_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_INVALID_IMAGE_SIZE`
  - if the dimensions for *src_image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if the format for *src_image*, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *src_image*
  - if there is a failure to allocate memory for the data store associated with *dst_buffer*
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support images (the query for `CL_DEVICE_IMAGE_SUPPORT` returns `CL_FALSE`)
  - if *dst_buffer* was created with `CL_MEM_IMMUTABLE_EXT`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command to copy a buffer object to an image object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueCopyBufferToImage(
    cl_command_queue command_queue,
    cl_mem src_buffer,
    cl_mem dst_image,
    size_t src_offset,
    const size_t* dst_origin,
    const size_t* region,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* must be a valid host command-queue. The OpenCL context associated with *command_queue*, *src_buffer* and *dst_image* must be the same.
- *src_buffer* is a valid buffer object.
- *dst_image* is a valid image object.
- *src_offset* refers to the offset where to begin copying data from *src_buffer*.
- *dst_origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *dst_image* is a 2D image object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image or 1D image buffer object, *dst_origin*[1] and *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[2] must be 0. If *dst_image* is a 1D image array object, *dst_origin*[1] describes the image index in the 1D image array. If *dst_image* is a 2D image array object, *dst_origin*[2] describes the image index in the 2D image array. If *dst_image* is a mipmapped image, the mip level to write is determined from *dst_origin* as described in Specifying Mipmap Levels to Image Operations
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *dst_image* is a 2D image object, *region*[2] must be 1. If *dst_image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *dst_image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this copy command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not

refer to an element of the *event_wait_list* array.

The size in bytes of the region to be copied from *src_buffer* referred to as *src_cb* is computed as *width* × *height* × *depth* × *bytes/image element* if *dst_image* is a 3D image object, is computed as *width* × *height* × *bytes/image element* if *dst_image* is a 2D image, is computed as *width* × *height* × *arraysize* × *bytes/image element* if *dst_image* is a 2D image array object, is computed as *width* × *bytes/image element* if *dst_image* is a 1D image or 1D image buffer object and is computed as *width* × *arraysize* × *bytes/image element* if *dst_image* is a 1D image array object.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_buffer* or *dst_image* is created is ignored by **clEnqueueCopyBufferToImage**.

clEnqueueCopyBufferToImage returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *src_buffer* are not the same
  - if the context associated with *command_queue* and *dst_image* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *src_buffer* is not a valid buffer object
  - if *dst_image* is not a valid image object
  - if *dst_image* is a 1D image buffer object created from *src_buffer*
- `CL_INVALID_VALUE`
  - if *dst_origin* is `NULL`
  - if *region* is `NULL`
  - if the region specified by *src_offset* and the computed *src_cb* is out of bounds of *src_buffer*
  - if the 1D, 2D or 3D rectangular region specified by *dst_origin* and *region* is out of bounds of *dst_image*
  - if values in *dst_origin* and *region* do not follow rules described in the argument description for *dst_origin* and *region*
- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *dst_origin* is not a valid level for *dst_image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events

- CL_MISALIGNED_SUB_BUFFER_OFFSET
  - if *src_buffer* is a sub-buffer object and the offset specified when the sub-buffer object was created is not aligned to the CL_DEVICE_MEM_BASE_ADDR_ALIGN value for the device associated with *command_queue*. This error code is missing before version 1.1.
- CL_INVALID_IMAGE_SIZE
  - if the dimensions for *dst_image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- CL_IMAGE_FORMAT_NOT_SUPPORTED
  - if the format for *dst_image*, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- CL_MEM_OBJECT_ALLOCATION_FAILURE
  - if there is a failure to allocate memory for the data store associated with *src_buffer*
  - if there is a failure to allocate memory for the data store associated with *dst_image*
- CL_INVALID_OPERATION
  - if the device associated with *command_queue* does not support images (the query for CL_DEVICE_IMAGE_SUPPORT returns CL_FALSE)
  - if *dst_image* was created with CL_MEM_IMMUTABLE_EXT
- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.3.7. Mapping Image Objects

To enqueue a command to map a region in the image object given by *image* into the host address space and returns a pointer to this mapped region, call the function

```
// Provided by CL_VERSION_1_0
void* clEnqueueMapImage(
    cl_command_queue command_queue,
    cl_mem image,
    cl_bool blocking_map,
    cl_map_flags map_flags,
    const size_t* origin,
    const size_t* region,
    size_t* image_row_pitch,
    size_t* image_slice_pitch,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event,
    cl_int* errcode_ret);
```

- *command_queue* must be a valid host command-queue.
- *image* is a valid image object. The OpenCL context associated with *command_queue* and *image* must be the same.
- *blocking_map* indicates if the map operation is *blocking* or *non-blocking*.
- *map_flags* is a bit-field and is described in the Memory Map Flags table.
- *origin* defines the (*x*, *y*, *z*) offset in pixels in the 1D, 2D or 3D image, the (*x*, *y*) offset and the image index in the 2D image array or the (*x*) offset and the image index in the 1D image array. If *image* is a 2D image object, *origin*[2] must be 0. If *image* is a 1D image or 1D image buffer object, *origin*[1] and *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[2] must be 0. If *image* is a 1D image array object, *origin*[1] describes the image index in the 1D image array. If *image* is a 2D image array object, *origin*[2] describes the image index in the 2D image array.
- *region* defines the (*width*, *height*, *depth*) in pixels of the 1D, 2D or 3D rectangle, the (*width*, *height*) in pixels of the 2D rectangle and the number of images of a 2D image array or the (*width*) in pixels of the 1D rectangle and the number of images of a 1D image array. If *image* is a 2D image object, *region*[2] must be 1. If *image* is a 1D image or 1D image buffer object, *region*[1] and *region*[2] must be 1. If *image* is a 1D image array object, *region*[2] must be 1. The values in *region* cannot be 0.
- *image_row_pitch* returns the scan-line pitch in bytes for the mapped region. This must be a non-`NULL` value.
- *image_slice_pitch* returns the size in bytes of each 2D slice of a 3D image or the size of each 1D or 2D image in a 1D or 2D image array for the mapped region. For a 1D and 2D image, zero is returned if this argument is not `NULL`. For a 3D image, 1D and 2D image array, *image_slice_pitch* must be a non-`NULL` value.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before **clEnqueueMapImage** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueMapImage** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

If *blocking_map* is `CL_TRUE`, **clEnqueueMapImage** does not return until the specified region in *image* is mapped into the host address space and the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapImage**.

If *blocking_map* is `CL_FALSE` i.e. map operation is non-blocking, the pointer to the mapped region

returned by **clEnqueueMapImage** cannot be used until the map command has completed. The *event* argument returns an event object which can be used to query the execution status of the map command. When the map command is completed, the application can access the contents of the mapped region using the pointer returned by **clEnqueueMapImage**.

clEnqueueMapImage will return a pointer to the mapped region. The *errcode_ret* is set to `CL_SUCCESS`.

A `NULL` pointer is returned otherwise with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *image* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *image* is not a valid image object
- `CL_INVALID_VALUE`
  - if *origin* is `NULL`
  - if *region* is `NULL`
  - if the region being mapped specified by *origin* and *region* is out of bounds
  - if values specified in *map_flags* are not valid
  - if values in *origin* and *region* do not follow the rules described in the argument description for *origin* and *region*
  - if *image_row_pitch* is `NULL`
  - if *image* is a 1D or 2D image array object and *image_slice_pitch* is `NULL`
  - if *image* is a 3D and *image_slice_pitch* is `NULL`
- `CL_INVALID_MIP_LEVEL`
  - if the `cl_khr_mipmap_image` extension is supported and the mip level specified in *origin* is not a valid level for *image*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_INVALID_IMAGE_SIZE`
  - if the dimensions for *image*, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if the format for *image*, such as the image channel order or image channel data type, are not

supported by the device associated with *command_queue*

- `CL_MAP_FAILURE`
  - if there is a failure to map the requested region into the host address space. This error cannot occur for image objects created with `CL_MEM_USE_HOST_PTR` or `CL_MEM_ALLOC_HOST_PTR`.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the map operations is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error. This error code is missing before version 1.1.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with *image*
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support images (the query for `CL_DEVICE_IMAGE_SUPPORT` returns `CL_FALSE`)
  - if *image* has been created with `CL_MEM_HOST_WRITE_ONLY` or `CL_MEM_HOST_NO_ACCESS` and `CL_MAP_READ` is set in *map_flags*
  - if *image* has been created with `CL_MEM_HOST_READ_ONLY` or `CL_MEM_HOST_NO_ACCESS` and `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` is set in *map_flags*
  - if *image* was created with `CL_MEM_IMMUTABLE_EXT` in *flags* and `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` is set in *map_flags*
  - if mapping would lead to overlapping regions being mapped for writing
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

The pointer returned maps a 1D, 2D or 3D region starting at *origin* and is at least *region*[0] pixels in size for a 1D image, 1D image buffer or 1D image array, (*image_row_pitch* × *region[1])* pixels in size for a 2D image or 2D image array, and (*image_slice_pitch* × *region[2])* pixels in size for a 3D image. The result of a memory access outside this region is undefined.

If the image object is created with `CL_MEM_USE_HOST_PTR` set in *mem_flags*, the following will be true:

- The *host_ptr* specified in **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D** is guaranteed to contain the latest bits in the region being mapped when the **clEnqueueMapImage** command has completed.
- The pointer value returned by **clEnqueueMapImage** will be derived from the *host_ptr* specified when the image object is created.

Mapped image objects are unmapped using **clEnqueueUnmapMemObject**. This is described in Unmapping Mapped Memory Objects.

5.3.8. Specifying Mipmap Levels to Image Operations

When the `cl_khr_mipmap_image` extension is supported, the **clEnqueueReadImage**, **clEnqueueWriteImage**, **clEnqueueMapImage**, **clEnqueueCopyImage**, **clEnqueueCopyImageToBuffer**, **clEnqueueCopyBufferToImage**, and **clEnqueueFillImage** functions described above can operate on mipmapped images.

The mipmap image level(s) to access for each command are determined from the *origin* parameter when accessing a single *image* (non-copy functions), or from the *src_origin* and *dst_origin* parameters when accessing two *src_image* and *dst_image* images (copy functions). The logic below applies to each of these parameters, with *image* and *origin* replaced by *src_image* and *src_origin*, or *dst_image* and *dst_origin* as appropriate:

- If *image* is a 1D image, *origin*[1] specifies the mip level to use.
- If *image* is a 1D image array, *origin*[2] specifies the mip level to use.
- If *image* is a 2D image, *origin*[2] specifies the mip level to use.
- If *image* is a 2D image array or a 3D image, *origin*[3] specifies the mip level to use.

5.3.9. Image Object Queries

To get information that is common to all memory objects, use the **clGetMemObjectInfo** function described in Memory Object Queries.

To get information specific to an image object created with **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D** call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetImageInfo(
    cl_mem image,
    cl_image_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *image* specifies the image object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetImageInfo** is described in the Image Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Image Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 35. List of supported param_names by **clGetImageInfo**

Image Info	Return type	Description
`CL_IMAGE_FORMAT`	`cl_image_format`	Return the image format descriptor specified when *image* is created with **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D** or **clCreateImage3D**.
`CL_IMAGE_ELEMENT_SIZE`	`size_t`	Return size of each element of the image memory object given by *image* in bytes.
`CL_IMAGE_ROW_PITCH`	`size_t`	Returns the row pitch in bytes of a row of elements of the image object given by *image*. If *image* was created with a non-zero value for `image_row_pitch`, then the value provided for `image_row_pitch` by the application is returned, otherwise the returned value is calculated as `CL_IMAGE_WIDTH` × `CL_IMAGE_ELEMENT_SIZE`.
`CL_IMAGE_SLICE_PITCH`	`size_t`	Returns the slice pitch in bytes of a 2D slice for the 3D image object or size of each image in a 1D or 2D image array given by *image*. If *image* was created with a non-zero value for `image_slice_pitch` then the value provided for `image_slice_pitch` by the application is returned, otherwise the returned value is calculated as: - `CL_IMAGE_ROW_PITCH` for 1D image arrays. - `CL_IMAGE_HEIGHT` × `CL_IMAGE_ROW_PITCH` for 3D images and 2D image arrays. For a 1D image, 1D image buffer and 2D image object return 0.
`CL_IMAGE_WIDTH`	`size_t`	Return width of the image in pixels.
`CL_IMAGE_HEIGHT`	`size_t`	Return height of the image in pixels. For a 1D image, 1D image buffer and 1D image array object, height = 0.
`CL_IMAGE_DEPTH`	`size_t`	Return depth of the image in pixels. For a 1D image, 1D image buffer, 2D image or 1D and 2D image array object, depth = 0.
`CL_IMAGE_ARRAY_SIZE` missing before version 1.2.	`size_t`	Return number of images in the image array. If *image* is not an image array, 0 is returned.
`CL_IMAGE_BUFFER` missing before version 1.2 and deprecated by version 2.0.	`cl_mem`	Return buffer object associated with *image*.

Image Info	Return type	Description
`CL_IMAGE_NUM_MIP_LEVELS` missing before version 1.2.	`cl_uint`	Return `num_mip_levels` associated with *image*.
`CL_IMAGE_NUM_SAMPLES` missing before version 1.2.	`cl_uint`	Return `num_samples` associated with *image*.
`CL_IMAGE_DX9_MEDIA_PLANE_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	`cl_uint`	If *image* was created using **clCreateFromDX9MediaSurfaceKHR**, returns the *plane* argument specified when *image* was created.
`CL_IMAGE_D3D10_SUBRESOURCE_KHR` provided by the `cl_khr_d3d10_sharing` extension.	`cl_uint`	If *image* was created using **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**, returns the *subresource* argument specified when *image* was created.
`CL_IMAGE_D3D11_SUBRESOURCE_KHR` provided by the `cl_khr_d3d11_sharing` extension.	`cl_uint`	If *image* was created using **clCreateFromD3D11Texture2DKHR**, or **clCreateFromD3D11Texture3DKHR**, returns the *subresource* argument specified when *image* was created.

clGetImageInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_MEM_OBJECT`
  - if *image* is a not a valid image object
- `CL_INVALID_VALUE`
  - if *param_name* is not one of the supported values
  - if the size in bytes specified by *param_value_size* is less than the size of the return type specified in the Image Object Queries table and *param_value* is not `NULL`
- `CL_INVALID_DX9_MEDIA_SURFACE_KHR`
  - if the `cl_khr_dx9_media_sharing` extension is supported, if *param_name* is `CL_IMAGE_DX9_MEDIA_PLANE_KHR`, and if *image* was not created by calling **clCreateFromDX9MediaSurfaceKHR**
- `CL_INVALID_D3D10_RESOURCE_KHR`
  - if the `cl_khr_d3d10_sharing` extension is supported, if *param_name* is `CL_IMAGE_D3D10_SUBRESOURCE_KHR`, and if *image* was not created by the function **clCreateFromD3D10Texture2DKHR** or **clCreateFromD3D10Texture3DKHR**
- `CL_INVALID_D3D11_RESOURCE_KHR`
  - if the `cl_khr_d3d11_sharing` extension is supported, if *param_name* is `CL_IMAGE_D3D11_`

SUBRESOURCE_KHR, and if *image* was not created by the function **clCreateFromD3D11Texture2DKHR** or **clCreateFromD3D11Texture3DKHR**

- CL_OUT_OF_RESOURCES
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- CL_OUT_OF_HOST_MEMORY
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.3.10. Creating Image Objects From DirectX 9 Media Resources

To create an OpenCL image object from a media surface, call the function

```
// Provided by cl_khr_dx9_media_sharing
cl_mem clCreateFromDX9MediaSurfaceKHR(
    cl_context context,
    cl_mem_flags flags,
    cl_dx9_media_adapter_type_khr adapter_type,
    void* surface_info,
    cl_uint plane,
    cl_int* errcode_ret);
```

clCreateFromDX9MediaSurfaceKHR is provided by the cl_khr_dx9_media_sharing extension.

- *context* is a valid OpenCL context created from a media adapter.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE flags specified in that table can be used.
- *adapter_type* is a value from enumeration of supported adapters described in the cl_dx9_media_adapter_type_khr values table. The type of *surface_info* is determined by the adapter type. The implementation does not need to support all adapter types. This approach provides flexibility to support additional adapter types in the future. Supported adapter types are CL_ADAPTER_D3D9_KHR, CL_ADAPTER_D3D9EX_KHR and CL_ADAPTER_DXVA_KHR.
- *surface_info* is a pointer to one of the structures defined in the *adapter_type* description above, passed in as a void *. If *adapter_type* is CL_ADAPTER_D3D9_KHR, CL_ADAPTER_D3D9EX_KHR and CL_ADAPTER_DXVA_KHR, *surface_info* points to a [cl_dx9_surface_info_khr] structure describing the surface.
- *plane* is the plane of resource to share for planar surface formats. For planar formats, we use the plane parameter to obtain a handle to thie specific plane (Y, U or V for example). For non-planar formats used by media, *plane* must be 0.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

The width and height of the returned OpenCL 2D image object are determined by the width and

height of the *plane* of the resource (*surface_info*→_resource_). The channel type and order of the returned image object is determined by the format and plane of the resource, and are described in the YUV FourCC Codes and Corresponding OpenCL Image Formats and Direct3D 9 Formats and Corresponding OpenCL Image Formats tables.

This call will increment the internal media surface count on the resource. The internal media surface reference count on the resource will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

clCreateFromDX9MediaSurfaceKHR returns a valid non-zero 2D image object and *errcode_ret* is set to `CL_SUCCESS` if the 2D image object is created successfully. Otherwise it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if *plane* is not a valid plane of *resource* specified in *surface_info*.
- `CL_INVALID_DX9_MEDIA_SURFACE_KHR` if *resource* specified in *surface_info* is not a valid resource or is not associated with *adapter_type* (e.g., *adapter_type* is set to `CL_ADAPTER_D3D9_KHR` and *resource* is not a Direct3D 9 surface created in D3DPOOL_DEFAULT).
- `CL_INVALID_DX9_MEDIA_SURFACE_KHR` if *shared_handle* specified in *surface_info* is not `NULL` or a valid handle value.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the texture format of *resource* is not listed in the YUV FourCC Codes and Corresponding OpenCL Image Formats or Direct3D 9 Formats and Corresponding OpenCL Image Formats tables.
- `CL_INVALID_OPERATION` if there are no devices in *context* that support *adapter_type*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

The `cl_dx9_surface_info_khr` structure is passed to **clCreateFromDX9MediaSurfaceKHR** to describe a DX9 surface, and is defined as:

```
// Provided by cl_khr_dx9_media_sharing
typedef struct cl_dx9_surface_info_khr {
    IDirect3DSurface9*    resource;
    HANDLE                shared_handle;
} cl_dx9_surface_info_khr;
```

- *resource* is a pointer to a `IDirect3DSurface9` surface interface.
- *shared_handle* is a `HANDLE` to the resource.

For DX9 surfaces, we need both the handle to the resource and the resource itself to have a sufficient amount of information to eliminate a copy of the surface for sharing in cases where this is possible. Elimination of the copy is driver dependent. *shared_handle* may be `NULL` and this may

result in sub-optimal performance.

5.3.11. Creating Image Objects From Direct3D Textures and Resources

To create an OpenCL 2D image object from a subresource of a Direct3D 10 2D texture, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_mem clCreateFromD3D10Texture2DKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D10Texture2D* resource,
    UINT subresource,
    cl_int* errcode_ret);
```

clCreateFromD3D10Texture2DKHR is provided by the `cl_khr_d3d10_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 10 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *resource* is a pointer to the Direct3D 10 2D texture to share.
- *subresource* is the subresource of *resource* to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The width and height of the returned OpenCL 2D image object are determined by the width and height of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 2D image object is determined by the format of *resource* and the DXGI Formats and Corresponding OpenCL Image Formats table.

This call will increment the internal Direct3D 10 reference count on *resource*. The internal Direct3D 10 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D10Texture2DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if *subresource* is not a valid

subresource index for *resource*.

- `CL_INVALID_D3D10_RESOURCE_KHR` if *resource* is not a Direct3D 10 texture resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if *resource* is a multisampled texture, if a `cl_mem` from subresource *subresource* of *resource* has already been created using **clCreateFromD3D10Texture2DKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the Direct3D 10 texture format of *resource* is not listed in the DXGI Formats and Corresponding OpenCL Image Formats table or if the Direct3D 10 texture format of *resource* does not map to a supported OpenCL image format.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create an OpenCL 3D image object from a subresource of a Direct3D 10 3D texture, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_mem clCreateFromD3D10Texture3DKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D10Texture3D* resource,
    UINT subresource,
    cl_int* errcode_ret);
```

clCreateFromD3D10Texture3DKHR is provided by the `cl_khr_d3d10_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 10 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *resource* is a pointer to the Direct3D 10 3D texture to share.
- *subresource* is the subresource of *resource* to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The width, height and depth of the returned OpenCL 3D image object are determined by the width, height and depth of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 3D image object is determined by the format of *resource* and the DXGI Formats and Corresponding OpenCL Image Formats table.

This call will increment the internal Direct3D 10 reference count on *resource*. The internal Direct3D 10 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing,

> and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D10Texture3DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if *subresource* is not a valid subresource index for *resource*.
- `CL_INVALID_D3D10_RESOURCE_KHR` if *resource* is not a Direct3D 10 texture resource, if *resource* was created with the D3D10_USAGE flag D3D10_USAGE_IMMUTABLE, if a `cl_mem` from subresource *subresource* of *resource* has already been created using **clCreateFromD3D10Texture3DKHR**, or if *context* was not created against the same Direct3D 10 device from which *resource* was created.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the Direct3D 10 texture format of *resource* is not listed in the DXGI Formats and Corresponding OpenCL Image Formats table or if the Direct3D 10 texture format of *resource* does not map to a supported OpenCL image format.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create an OpenCL 2D image object from a subresource of a Direct3D 11 2D texture, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_mem clCreateFromD3D11Texture2DKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D11Texture2D* resource,
    UINT subresource,
    cl_int* errcode_ret);
```

> **clCreateFromD3D11Texture2DKHR** is provided by the `cl_khr_d3d11_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 11 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *resource* is a pointer to the Direct3D 11 2D texture to share.
- *subresource* is the subresource of *resource* to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The width and height of the returned OpenCL 2D image object are determined by the width and

height of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 2D image object is determined by the format of *resource* and the DXGI Formats and Corresponding OpenCL Image Formats table.

This call will increment the internal Direct3D 11 reference count on *resource*. The internal Direct3D 11 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D11Texture2DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if *subresource* is not a valid subresource index for *resource*.
- `CL_INVALID_D3D11_RESOURCE_KHR` if *resource* is not a Direct3D 11 texture resource, if *resource* was created with the D3D11_USAGE flag D3D11_USAGE_IMMUTABLE, if *resource* is a multisampled texture, if a `cl_mem` from subresource *subresource* of *resource* has already been created using **clCreateFromD3D11Texture2DKHR**, or if *context* was not created against the same Direct3D 11 device from which *resource* was created.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the Direct3D 11 texture format of *resource* is not listed in the DXGI Formats and Corresponding OpenCL Image Formats table or if the Direct3D 11 texture format of *resource* does not map to a supported OpenCL image format.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create an OpenCL 3D image object from a subresource of a Direct3D 11 3D texture, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_mem clCreateFromD3D11Texture3DKHR(
    cl_context context,
    cl_mem_flags flags,
    ID3D11Texture3D* resource,
    UINT subresource,
    cl_int* errcode_ret);
```

clCreateFromD3D11Texture3DKHR is provided by the `cl_khr_d3d11_sharing` extension.

- *context* is a valid OpenCL context created from a Direct3D 11 device.
- *flags* is a bit-field that is used to specify usage information. Refer to the List of supported

memory flag values table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.

- *resource* is a pointer to the Direct3D 11 3D texture to share.
- *subresource* is the subresource of *resource* to share.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The width, height and depth of the returned OpenCL 3D image object are determined by the width, height and depth of subresource *subresource* of *resource*. The channel type and order of the returned OpenCL 3D image object is determined by the format of *resource* and the DXGI Formats and Corresponding OpenCL Image Formats table.

This call will increment the internal Direct3D 11 reference count on *resource*. The internal Direct3D 11 reference count on *resource* will be decremented when the OpenCL reference count on the returned OpenCL memory object drops to zero.

Refer to the Lifetime of Shared Direct3D Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromD3D11Texture3DKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if *subresource* is not a valid subresource index for *resource*.
- `CL_INVALID_D3D11_RESOURCE_KHR` if *resource* is not a Direct3D 11 texture resource, if *resource* was created with the D3D11_USAGE flag D3D11_USAGE_IMMUTABLE, if a `cl_mem` from subresource *subresource* of *resource* has already been created using **clCreateFromD3D11Texture3DKHR**, or if *context* was not created against the same Direct3D 11 device from which *resource* was created.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the Direct3D 11 texture format of *resource* is not listed in the DXGI Formats and Corresponding OpenCL Image Formats table or if the Direct3D 11 texture format of *resource* does not map to a supported OpenCL image format.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.12. Creating Image Objects From EGL Images

To create an `EGLImage` target of type `cl_mem` from the `EGLImage` source provided as *image*, call the function

```
// Provided by cl_khr_egl_image
cl_mem clCreateFromEGLImageKHR(
    cl_context context,
    CLeglDisplayKHR egldisplay,
    CLeglImageKHR eglimage,
    cl_mem_flags flags,
    const cl_egl_image_properties_khr* properties,
    cl_int* errcode_ret);
```

clCreateFromEGLImageKHR is provided by the `cl_khr_egl_image` extension.

- *display* should be of type `EGLDisplay`, cast into the type `CLeglDisplayKHR`.
- *image* should be of type `EGLImageKHR`, cast into the type `CLeglImageKHR`. Assuming no errors are generated in this function, the resulting image object will be an `EGLImage` target of the specified `EGLImage` *image*. The resulting `cl_mem` is an image object which may be used normally by all OpenCL operations. This maps to an `image2d_t` type in OpenCL kernel code.
- *flags* is a bit-field that is used to specify usage information about the memory object being created. Refer to the Memory Flags table for a description of *flags*. Accepted values in *flags* are described below.
- *properties* specifies a list of property names and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. No properties are currently supported with this version of the extension. *properties* can be `NULL`.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Accepted for *flags* are `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY`, `CL_MEM_IMMUTABLE_EXT`, and `CL_MEM_READ_WRITE`. If OpenCL 1.2 is supported, *flags* also accepts `CL_MEM_HOST_WRITE_ONLY`, `CL_MEM_HOST_READ_ONLY`, and `CL_MEM_HOST_NO_ACCESS`.

`cl_khr_egl_image` only requires support for `CL_MEM_READ_ONLY`, and for `CL_MEM_HOST_NO_ACCESS` if OpenCL 1.2 or later is supported. For OpenCL 1.1, a `CL_INVALID_OPERATION` will be returned for images which do not support host mapping.

If the value passed in *flags* is not supported by the OpenCL implementation, it will return `CL_INVALID_VALUE`. The accepted *flags* may be dependent upon the texture format used.

If `CL_MEM_IMMUTABLE_EXT` is set in *flags*, the image created is an *immutable memory object* that cannot be modified by the implementation after creation.

clCreateFromEGLImageKHR returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid OpenCL context.
- `CL_INVALID_VALUE` if *properties* contains invalid values, if *display* is not a valid display object or if *flags* are not in the set defined above.

- CL_INVALID_EGL_OBJECT_KHR if *image* is not a valid EGLImage object.
- CL_IMAGE_FORMAT_NOT_SUPPORTED if the OpenCL implementation is not able to create a cl_mem compatible with the provided CLeglImageKHR for an implementation-dependent reason (this could be caused by, but not limited to, reasons such as unsupported texture formats, etc).
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_INVALID_OPERATION if there are no devices in *context* that support images (i.e. CL_DEVICE_IMAGE_SUPPORT specified in table 4.3 is CL_FALSE) or if the flags passed are not supported for that image type.
- CL_INVALID_VALUE if CL_MEM_IMMUTABLE_EXT is set in *flags* and CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, or CL_MEM_HOST_WRITE_ONLY is set in *flags*.

5.3.13. Creating Image Objects From OpenGL Textures and Renderbuffers

To create an OpenCL image object from an OpenGL texture object, call the function

```
// Provided by cl_khr_gl_sharing
cl_mem clCreateFromGLTexture(
    cl_context context,
    cl_mem_flags flags,
    cl_GLenum target,
    cl_GLint miplevel,
    cl_GLuint texture,
    cl_int* errcode_ret);
```

clCreateFromGLTexture is provided by the cl_khr_gl_sharing extension.

- *context* is a valid OpenCL context created from an OpenGL context.
- *flags* is a bit-field that is used to specify usage information. Refer to the Memory Flags table for a description of *flags*. Only the CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY and CL_MEM_READ_WRITE flags specified in that table can be used.
- *texture_target* must be one of GL_TEXTURE_1D, GL_TEXTURE_1D_ARRAY, GL_TEXTURE_BUFFER, GL_TEXTURE_2D, GL_TEXTURE_2D_ARRAY, GL_TEXTURE_3D, GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z. GL_TEXTURE_RECTANGLE or the equivalent GL_TEXTURE_RECTANGLE_ARB may be specified if an OpenGL implementation supporting rectangular textures is supported. GL_TEXTURE_2D_MULTISAMPLE and GL_TEXTURE_2D_MULTISAMPLE_ARRAY may be specified if an OpenGL implementation supporting multi-sample two-dimensional textures is supported, and the cl_khr_gl_msaa_sharing extension is supported. Refer to the Restrictions on Multi-Sample Images section for more information on multi-sample images. *texture_target* is used only to define the image type of *texture*. No

reference to a bound OpenGL texture object is made or implied by this parameter.

- *miplevel* is the mipmap level to be used. If *texture_target* is `GL_TEXTURE_BUFFER`, *miplevel* must be 0. Note: Implementations may return `CL_INVALID_OPERATION` for miplevel values > 0.
- *texture* is the name of an OpenGL 1D, 2D, 3D, 1D array, 2D array, cubemap, rectangle or buffer texture object. The texture object must be a complete texture as per OpenGL rules on texture completeness. The *texture* format and dimensions defined by OpenGL for the specified *miplevel* of the texture will be used to create the OpenCL image memory object. Only OpenGL texture objects with an internal format that maps to an appropriate Image Channel Order and Image Channel Data Type may be used to create the OpenCL image memory object.
- *errcode_ret* will return an appropriate error code as described below. If *errcode_ret* is `NULL`, no error code is returned.

clCreateFromGLTexture may create any of the following:

- an OpenCL 2D image object from an OpenGL 2D texture object or a single face of an OpenGL cubemap texture object,
- an OpenCL 2D image array object from an OpenGL 2D texture array object,
- an OpenCL 2D multi-sample image object from an OpenGL 2D multi-sample texture.
- an OpenCL 2D multi-sample array image object from an OpenGL 2D multi-sample texture.
- an OpenCL 1D image object from an OpenGL 1D texture object,
- an OpenCL 1D image buffer object from an OpenGL texture buffer object,
- an OpenCL 1D image array object from an OpenGL 1D texture array object,
- an OpenCL 3D image object from an OpenGL 3D texture object.

If both the `cl_khr_mipmap_image` and `cl_khr_gl_sharing` extensions are supported by the OpenCL device, **clCreateFromGLTexture** may also be used to create a mipmapped OpenCL image from a mipmapped OpenGL texture by specify a negative value for *miplevel*. In this case, then an OpenCL mipmapped image object is created from a mipmapped OpenGL texture object, instead of an OpenCL image object for a specific miplevel of the OpenGL texture.

For a detailed description of how the level of detail is computed, please refer to the “Scale Factor and Level-of-Detail” section of the OpenGL 4.6 Specification.

If the state of an OpenGL texture object is modified through the OpenGL API (e.g. `glTexImage2D`, `glTexImage3D` or the values of the texture parameters `GL_TEXTURE_BASE_LEVEL` or `GL_TEXTURE_MAX_LEVEL` are modified) while there exists a corresponding OpenCL image object, subsequent use of the OpenCL image object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image objects.

Refer to the Lifetime of Shared OpenCL/OpenGL Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromGLTexture returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context or was not created from an OpenGL context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid or if value specified in *texture_target* is not one of the values specified in the description of *texture_target*.
- `CL_INVALID_MIP_LEVEL` if *miplevel* is less than the value of $level_{base}$ (for OpenGL implementations) or zero (for OpenGL ES implementations); or greater than the value of q (for both OpenGL and OpenGL ES). $level_{base}$ and q are defined for the texture in *section 3.8.10* (Texture Completeness) of the OpenGL 2.1 Specification and *section 3.7.10* of the OpenGL ES 2.0 Specification.
- `CL_INVALID_MIP_LEVEL` if *miplevel* is greather than zero and the OpenGL implementation does not support creating from non-zero mipmap levels.
- `CL_INVALID_GL_OBJECT` if *texture* is not an OpenGL texture object whose type matches *texture_target*, if the specified *miplevel* of *texture* is not defined, or if the width or height of the specified *miplevel* is zero or if the OpenGL texture object is incomplete.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the internal format of *texture* is not listed in the OpenGL Internal Formats and Corresponding OpenCL Internal Formats table.
- `CL_INVALID_OPERATION` if *texture* is an OpenGL texture object created with a border width value greater than zero.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.13.1. Restrictions on Depth/Stencil Images

Depth images with an image channel order of `CL_DEPTH_STENCIL` can only be created using the **clCreateFromGLTexture** API, and only when the `cl_khr_gl_depth_images` extension is supported.

For the image format given by channel order of `CL_DEPTH_STENCIL` and channel data type of `CL_UNORM_INT24`, the depth is stored as an unsigned normalized 24-bit value.

For the image format given by channel order of `CL_DEPTH_STENCIL` and channel data type of `CL_FLOAT`, each pixel is two 32-bit values. The depth is stored as a single precision floating-point value followed by the stencil which is stored as a 8-bit integer value.

Such images appear in the] Minimum List of Supported Image Formats, but only require read support, not write support.

The stencil value cannot be read or written using the **read_imagef** and **write_imagef** built-in functions in an OpenCL kernel.

Depth image objects with an image channel order of `CL_DEPTH_STENCIL` cannot be used as arguments to **clEnqueueReadImage**, **clEnqueueWriteImage**, **clEnqueueCopyImage**, **clEnqueueCopyImageToBuffer**, **clEnqueueCopyBufferToImage**, **clEnqueueMapImage**, and

clEnqueueFillImage. Such use will return a `CL_INVALID_OPERATION` error.

5.3.13.2. Restrictions on Multi-Sample Images

The formats described in the Image Channel Order Values and Image Channel Data Types tables of the OpenCL specification also support OpenCL images created from a OpenGL multi-sampled color or depth texture.

Multi-sample OpenCL image objects can only be read from a kernel. Multi-sample OpenCL image objects cannot be used as arguments to **clEnqueueReadImage**, **clEnqueueWriteImage**, **clEnqueueCopyImage**, **clEnqueueCopyImageToBuffer**, **clEnqueueCopyBufferToImage**, **clEnqueueMapImage**, and **clEnqueueFillImage**. Such use will return a `CL_INVALID_OPERATION` error.

To create an OpenCL 2D image object from an OpenGL renderbuffer object, call the function

```
// Provided by cl_khr_gl_sharing
cl_mem clCreateFromGLRenderbuffer(
    cl_context context,
    cl_mem_flags flags,
    cl_GLuint renderbuffer,
    cl_int* errcode_ret);
```

clCreateFromGLRenderbuffer is provided by the `cl_khr_gl_sharing` extension.

- *context* is a valid OpenCL context created from an OpenGL context.
- *flags* is a bit-field that is used to specify usage information. Refer to the Memory Flags table for a description of *flags*. Only the `CL_MEM_READ_ONLY`, `CL_MEM_WRITE_ONLY` and `CL_MEM_READ_WRITE` flags specified in that table can be used.
- *renderbuffer* is the name of an OpenGL renderbuffer object. The renderbuffer storage must be specified before the image object can be created. The *renderbuffer* format and dimensions defined by OpenGL will be used to create the 2D image object. Only OpenGL renderbuffers with an internal format that maps to an appropriate Image Channel Order and Image Channel Data Type may be used to create the 2D image object.
- *errcode_ret* will return an appropriate error code as described below. If *errcode_ret* is `NULL`, no error code is returned.

If the state of an OpenGL renderbuffer object is modified through the OpenGL API (i.e. changes to the dimensions or format used to represent pixels of the OpenGL renderbuffer using appropriate OpenGL API calls such as `glRenderbufferStorage`) while there exists a corresponding OpenCL image object, subsequent use of the OpenCL image object will result in undefined behavior.

The **clRetainMemObject** and **clReleaseMemObject** functions can be used to retain and release the image objects.

The OpenGL Internal Formats and Corresponding OpenCL Internal Formats table describes the list of OpenGL renderbuffer internal formats and the Corresponding OpenCL Image Formats. If an

OpenGL renderbuffer object with an internal format from the table is successfully created by OpenGL, then there is guaranteed to be a mapping to one of the corresponding OpenCL image format(s) in that table. Renderbuffer objects created with other OpenGL internal formats may (but are not guaranteed to) have a mapping to an OpenCL image format; if such mappings exist, they are guaranteed to preserve all color components, data types, and at least the number of bits/component actually allocated by OpenGL for that format.

Refer to the Lifetime of Shared OpenCL/OpenGL Memory Objects and Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects sections for more information.

clCreateFromGLRenderbuffer returns a valid non-zero OpenCL image object and *errcode_ret* is set to `CL_SUCCESS` if the image object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context or was not created from an OpenGL context.
- `CL_INVALID_VALUE` if values specified in *flags* are not valid.
- `CL_INVALID_GL_OBJECT` if *renderbuffer* is not an OpenGL renderbuffer object, or if the width or height of *renderbuffer* is zero.
- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` if the internal format of *renderbuffer* is not listed in the OpenGL Internal Formats and Corresponding OpenCL Internal Formats table.
- `CL_INVALID_OPERATION` if *renderbuffer* is a multi-sample OpenGL renderbuffer object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.3.14. Querying Image Requirements

To get information specific to the requirements of an image before creating it call the function

```
// Provided by cl_ext_image_requirements_info
cl_int clGetImageRequirementsInfoEXT(
    cl_context context,
    const cl_mem_properties* properties,
    cl_mem_flags flags,
    const cl_image_format* image_format,
    const cl_image_desc* image_desc,
    cl_image_requirements_info_ext param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetImageRequirementsInfoEXT is provided by the `cl_ext_image_requirements_info` extension.

- *context* is the OpenCL context in which the query will be performed.
- *properties* is an optional list of properties for the image object and their corresponding values. The list is terminated with the special property 0. If no properties are required, properties may be `NULL`.
- *flags* is a bit-field that is used to specify allocation and usage information about the image format being queried and is described in the Memory Flags table. *flags* may be `CL_MEM_READ_WRITE` to specialize the query for images that may be read from and written to by different kernel-instances when correctly ordered by event dependencies, or `CL_MEM_READ_ONLY` to specialize the query for images that may be read from by a kernel, or `CL_MEM_WRITE_ONLY` to specialiaze the query for images that may be written to by a kernel, or `CL_MEM_KERNEL_READ_AND_WRITE` to specialize the query for images that may be both read from and written to by the same kernel-instance. When *flags* is `0` the value returned for the query must be correct for all possible values of *flags*.
- *image_format* is a pointer to a structure describing the format of the image for which requirements are being queried. Refer to the Image Format Descriptor section for a detailed description.
- *image_desc* is a pointer to a structure that describes type and dimensions of the image for which requirements are being queried. Refer to the Image Descriptor section for a detailed description of the image descriptor.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetImageRequirementsInfoEXT** is described in the Image Requirements Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* is used to specify the size in bytes of memory pointed to by *param_value*. This size must be ≥ size of return type as described in the Image Requirements Queries table.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 36. List of supported param_names by **clGetImageRequirementsInfoEXT**

Image Format Info	**Return type**	**Info. returned in** *param_value*
`CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT` provided by the `cl_ext_image_requirements_info` extension.	`size_t`	Returns the minimum alignment in bytes required for the data store backing an image created using the parameters passed to **clGetImageRequirementsInfoEXT**. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the implementation will return an alignment that would be sufficient for all possible values of the missing argument. The value returned is a power of two.

Image Format Info	Return type	Info. returned in *param_value*
`CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT` provided by the `cl_ext_image_requirements_info` extension.	`size_t`	Returns the row pitch alignment required in bytes for images created from a buffer with the parameters passed to **clGetImageRequirementsInfoEXT**. The value returned is a power of two. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the value returned is the minimum row pitch alignment that works for all possible values of the missing argument(s).
`CL_IMAGE_REQUIREMENTS_SIZE_EXT` provided by the `cl_ext_image_requirements_info` extension.	`size_t`	Returns the minimal size in bytes that a buffer would need to be to back an image created using the parameters passed to **clGetImageRequirementsInfoEXT**. Both *image_format* and *image_desc* must be non-`NULL`, otherwise `CL_INVALID_VALUE` is returned.
`CL_IMAGE_REQUIREMENTS_MAX_WIDTH_EXT` provided by the `cl_ext_image_requirements_info` extension.	`cl_uint`	Returns the max width supported for creating images with the parameters passed to **clGetImageRequirementsInfoEXT**. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the implementation will return a max width that would be supported for all possible values of the missing argument(s). When *image_desc* is not `NULL`, the value of its *image_width* member is ignored and has no effect on the value returned. The value of all other members, except `mem_object` may be `0` to require that the value returned be supported for all possible values of the members that are set to `0`.

Image Format Info	Return type	Info. returned in *param_value*
`CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT` provided by the `cl_ext_image_requirements_info` extension.	`cl_uint`	Returns the max height supported for creating images with the parameters passed to **clGetImageRequirementsInfoEXT**. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the implementation will return a max height that would be supported for all possible values of the missing argument(s). When *image_desc* is not `NULL`, the value of its *image_height* member is ignored and has no effect on the value returned. The value of all other members, except `mem_object` may be `0` to require that the value returned be supported for all possible values of the members that are set to `0`. If *image_desc* is not `NULL`, then *image_type* must be either `0`, `CL_MEM_OBJECT_IMAGE2D`, `CL_MEM_OBJECT_IMAGE2D_ARRAY`, or `CL_MEM_OBJECT_IMAGE3D`, otherwise `CL_INVALID_IMAGE_DESCRIPTOR` is returned.
`CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT` provided by the `cl_ext_image_requirements_info` extension.	`cl_uint`	Returns the max depth supported for creating images with the parameters passed to **clGetImageRequirementsInfoEXT**. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the implementation will return a max depth that would be supported for all possible values of the missing argument(s). When *image_desc* is not `NULL`, the value of its *image_depth* member is ignored and has no effect on the value returned. The value of all other members, except `mem_object` may be `0` to require that the value returned be supported for all possible values of the members that are set to `0`. If *image_desc* is not `NULL`, then *image_type* must be either `0` or `CL_MEM_OBJECT_IMAGE3D`, otherwise `CL_INVALID_IMAGE_DESCRIPTOR` is returned.

Image Format Info	Return type	Info. returned in *param_value*
`CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT` provided by the `cl_ext_image_requirements_info` extension.	`cl_uint`	Returns the max array size supported for creating images with the parameters passed to **clGetImageRequirementsInfoEXT**. *image_format* or *image_desc* are allowed to be `NULL`. When either or both is `NULL` the implementation will return a max array size that would be supported for all possible values of the missing argument(s). When *image_desc* is not `NULL`, the value of its *image_array_size* member is ignored and has no effect on the value returned. The value of all other members, except `mem_object` may be `0` to require that the value returned be supported for all possible values of the members that are set to `0`. If *image_desc* is not `NULL`, then *image_type* must be either `0`, `CL_MEM_OBJECT_IMAGE1D_ARRAY` or `CL_MEM_OBJECT_IMAGE2D_ARRAY`, otherwise `CL_INVALID_IMAGE_DESCRIPTOR` is returned.
`CL_IMAGE_REQUIREMENTS_SLICE_PITCH_ALIGNMENT_EXT` provided by the `cl_ext_image_from_buffer` extension.	`size_t`	Returns the slice pitch alignment required in bytes for images created from a buffer with the parameters passed to **clGetImageRequirementsInfoEXT**. The value returned is a power of two. *image_format* and *image_desc* are allowed to be `NULL`. When either or both is `NULL` the value returned is the minimum slice pitch alignment that is supported for all possible values of the missing argument(s).

clGetImageRequirementsInfoEXT returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_CONTEXT`
  - if *context* if not a valid context
- `CL_INVALID_PROPERTY`
  - if a property name in *properties* is not a supported property name
  - if the value specified for a supported property name is not valid
  - if the same property name is specified more than once
- `CL_INVALID_VALUE`

- if values specified in *flags* are not valid

- `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR`
  - if values specified in *image_format* are not valid
- `CL_INVALID_IMAGE_DESCRIPTOR`
  - if values specified in *image_desc* are not valid
- `CL_INVALID_VALUE`
  - if *param_name* is not valid
  - if size in bytes specified by *param_value_size* is less than the size of the return type specified in the Image Requirements Queries table and *param_value* is not `NULL`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.4. Pipes

Pipes are missing before version 2.0.

A *pipe* is a memory object that stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. A pipe object encapsulates the following information:

- Packet size in bytes
- Maximum capacity in packets
- Information about the number of packets currently in the pipe
- Data packets

5.4.1. Creating Pipe Objects

To create a **pipe object**, call the function

```
// Provided by CL_VERSION_2_0
cl_mem clCreatePipe(
    cl_context context,
    cl_mem_flags flags,
    cl_uint pipe_packet_size,
    cl_uint pipe_max_packets,
    const cl_pipe_properties* properties,
    cl_int* errcode_ret);
```

clCreatePipe is missing before version 2.0.

- *context* is a valid OpenCL context used to create the pipe object.
- *flags* is a bit-field that is used to specify allocation and usage information such as the memory arena that should be used to allocate the pipe object and how it will be used. The Memory Flags table describes the possible values for *flags*. Only `CL_MEM_READ_WRITE` and `CL_MEM_HOST_NO_ACCESS` can be specified when creating a pipe object. If the value specified for *flags* is 0, the default is used which is `CL_MEM_READ_WRITE` | `CL_MEM_HOST_NO_ACCESS`.
- *pipe_packet_size* is the size in bytes of a pipe packet.
- *pipe_max_packets* specifies the pipe capacity by specifying the maximum number of packets the pipe can hold.
- *properties* specifies a list of properties for the pipe and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. Currently, in all OpenCL versions, *properties* must be `NULL`.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

clCreatePipe returns a valid non-zero pipe object and *errcode_ret* is set to `CL_SUCCESS` if the pipe object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_OPERATION` if no devices in *context* support pipes.
- `CL_INVALID_VALUE` if values specified in *flags* are not as defined above.
- `CL_INVALID_VALUE` if *properties* is not `NULL`.
- `CL_INVALID_PIPE_SIZE` if *pipe_packet_size* is 0 or the *pipe_packet_size* exceeds `CL_DEVICE_PIPE_MAX_PACKET_SIZE` value specified in the Device Queries table for all devices in *context* or if *pipe_max_packets* is 0.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE` if there is a failure to allocate memory for the pipe object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.
- `CL_INVALID_VALUE` if `CL_MEM_IMMUTABLE_EXT` is set in *flags*.

Pipes follow the same memory consistency model as defined for buffer and image objects. The pipe state i.e. contents of the pipe across kernel-instances (on the same or different devices) is enforced at a synchronization point.

5.4.2. Pipe Object Queries

To get information that is common to all memory objects, use the **clGetMemObjectInfo** function described in Memory Object Queries.

To get information specific to a pipe object created with **clCreatePipe**, call the function

```
// Provided by CL_VERSION_2_0
cl_int clGetPipeInfo(
    cl_mem pipe,
    cl_pipe_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetPipeInfo is missing before version 2.0.

- *pipe* specifies the pipe object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetPipeInfo** is described in the Pipe Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Pipe Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 37. List of supported param_names by **clGetPipeInfo**

Pipe Info	Return type	Description
`CL_PIPE_PACKET_SIZE` missing before version 2.0.	`cl_uint`	Return pipe packet size specified when *pipe* is created with **clCreatePipe**.
`CL_PIPE_MAX_PACKETS` missing before version 2.0.	`cl_uint`	Return max. number of packets specified when *pipe* is created with **clCreatePipe**.

Pipe Info	Return type	Description
CL_PIPE_PROPERTIES missing before version 3.0.	cl_pipe_properties[]	Return the properties argument specified in **clCreatePipe**. If the *properties* argument specified in **clCreatePipe** used to create *pipe* was not NULL, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If the *properties* argument specified in **clCreatePipe** used to create *pipe* was NULL, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.

clGetPipeInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_MEM_OBJECT if *pipe* is a not a valid pipe object.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Pipe Object Queries table and *param_value* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5. Memory Objects

5.5.1. Retaining and Releasing Memory Objects

To retain a memory object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainMemObject(
    cl_mem memobj);
```

- *memobj* specifies the memory object to be retained.

The *memobj* reference count is incremented.

clRetainMemObject returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_MEM_OBJECT` if *memobj* is not a valid memory object (buffer or image object).
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

clCreateBuffer, **clCreateBufferWithProperties**, **clCreateSubBuffer**, **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, **clCreateImage3D** and **clCreatePipe** perform an implicit retain.

To release a memory object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseMemObject(
    cl_mem memobj);
```

- *memobj* specifies the memory object to be released.

The *memobj* reference count is decremented.

clReleaseMemObject returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_MEM_OBJECT` if *memobj* is not a valid memory object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

After the *memobj* reference count becomes zero and commands queued for execution on a command-queue(s) that use *memobj* have finished, the memory object is deleted. If *memobj* is a buffer object, *memobj* cannot be deleted until all sub-buffer objects associated with *memobj* are deleted. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainMemObject** causes undefined behavior.

To register a callback function with a memory object that is called when the memory object is destroyed, call the function

```
// Provided by CL_VERSION_1_1
cl_int clSetMemObjectDestructorCallback(
    cl_mem memobj,
    void (CL_CALLBACK* pfn_notify)(cl_mem memobj, void* user_data),
    void* user_data);
```

clSetMemObjectDestructorCallback is missing before version 1.1.

- *memobj* specifies the memory object to register the callback to.
- *pfn_notify* is the callback function to register. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:
  - *memobj* is the memory object being deleted. When the callback function is called by the implementation, this memory object is not longer valid. *memobj* is only provided for reference purposes.
  - *user_data* is a pointer to user-supplied data.
- *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be `NULL`.

Each call to **clSetMemObjectDestructorCallback** registers the specified callback function on a destructor callback stack associated with *memobj*. The registered callback functions are called in the reverse order in which they were registered. The registered callback functions are called and then the memory object's resources are freed and the memory object is deleted. Therefore, the memory object destructor callback provides a mechanism for an application to safely re-use or free a *host_ptr* that was specified when *memobj* was created and used as the storage bits for the memory object.

clSetMemObjectDestructorCallback returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_MEM_OBJECT` if *memobj* is not a valid memory object.
- `CL_INVALID_VALUE` if *pfn_notify* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

When the user callback function is called by the implementation, the contents of the memory region pointed to by *host_ptr* (if the memory object is created with `CL_MEM_USE_HOST_PTR`) are undefined. The callback function is typically used by the application to either free or reuse the memory region pointed to by *host_ptr*.

The behavior of calling expensive system routines, OpenCL API calls to create contexts or command-queues, or blocking OpenCL operations from the following list below, in a callback is undefined.

- **clFinish**,
- **clWaitForEvents**,
- blocking calls to **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**, **clEnqueueWriteBuffer**, **clEnqueueWriteBufferRect**,
- blocking calls to **clEnqueueReadImage** and **clEnqueueWriteImage**,
- blocking calls to **clEnqueueMapBuffer**, **clEnqueueMapImage**,

- blocking calls to **clBuildProgram**, **clCompileProgram** or **clLinkProgram**

If an application needs to wait for completion of a routine from the above list in a callback, please use the non-blocking form of the function, and assign a completion callback to it to do the remainder of your work. Note that when a callback (or other code) enqueues commands to a command-queue, the commands are not required to begin execution until the queue is flushed. In standard usage, blocking enqueue calls serve this role by implicitly flushing the queue. Since blocking calls are not permitted in callbacks, those callbacks that enqueue commands on a command-queue should either call **clFlush** on the queue before returning or arrange for **clFlush** to be called later on another thread.

The user callback function may not call OpenCL APIs with the memory object for which the callback function is invoked and for such cases the behavior of OpenCL APIs is considered to be undefined.

5.5.1.1. Acquiring and Releasing External Memory Objects

To enqueue a command to acquire OpenCL memory objects created from external memory handles, call the function

```
// Provided by cl_khr_external_memory
cl_int clEnqueueAcquireExternalMemObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_mem_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireExternalMemObjectsKHR is provided by the `cl_khr_external_memory` extension.

- *command_queue* specifies a valid command-queue.
- *num_mem_objects* specifies the number of memory objects to acquire.
- *mem_objects* points to a list of valid memory objects.
- *num_events_in_wait_list* specifies the number of events in *event_wait_list*.
- *event_wait_list* points to the list of events that need to complete before **clEnqueueAcquireExternalMemObjectsKHR** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueAcquireExternalMemObjectsKHR** does not explicitly wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and that of *command_queue* must be the same.
- *event* returns an event object that identifies this particular command and can be used to query

or queue a wait for this particular command to complete. *event* can be NULL in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

Applications must acquire the memory objects that are created using external handles before they can be used by any OpenCL commands queued to a command-queue. Behavior is undefined if a memory object created from an external memory handle is used by an OpenCL command queued to a command-queue without being acquired. This is to guarantee that the state of the memory objects is up-to-date and they are accessible to OpenCL.

The following restrictions shall apply:

- Each memory object must be acquired only once. Acquiring a memory object multiple times without releasing it results in implementation-defined behavior.
- The acquire must be performed on a command-queue associated with a device that was one of the devices specified via CL_MEM_DEVICE_HANDLE_LIST_KHR when the memory object was imported using **clCreateBufferWithProperties** or **clCreateImageWithProperties**. If CL_MEM_DEVICE_HANDLE_LIST_KHR was not specified, the acquire can be performed on a command-queue associated with any device in the context.
- The memory object will be acquired for all devices specified via CL_MEM_DEVICE_HANDLE_LIST_KHR when the memory object was imported using **clCreateBufferWithProperties** or **clCreateImageWithProperties**. If CL_MEM_DEVICE_HANDLE_LIST_KHR was not specified, the memory object will be acquired for all devices in the context.

See "Example with Acquire / Release" for more details on how to use this API.

If *num_mem_objects* is 0 and *mem_objects* is NULL, the command will trivially succeed after its event dependencies are satisfied and will update its completion event.

clEnqueueAcquireExternalMemObjectsKHR returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_VALUE if *num_mem_objects* is zero and *mem_objects* is not a NULL value, or if *num_mem_objects* is greater than 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if any of the memory objects in *mem_objects* is not a valid OpenCL memory object created using an external memory handle.
- CL_INVALID_COMMAND_QUEUE
  - if *command_queue* is not a valid command-queue, or
  - if device associated with *command_queue* is not one of the devices specified by CL_MEM_DEVICE_HANDLE_LIST_KHR at the time of creating one or more of *mem_objects*, or
  - if one or more of *mem_objects* belong to a context that does not contain a device associated with *command_queue*.
- CL_INVALID_EVENT_WAIT_LIST
  - if *event_wait_list* is NULL and *num_events_in_wait_list* is not 0, or
  - if *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or

    - if event objects in *event_wait_list* are not valid events.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST` if the execution status of any of the events in *event_wait_list* is a negative integer value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a command to release OpenCL memory objects created from external memory handles, call the function

```
// Provided by cl_khr_external_memory
cl_int clEnqueueReleaseExternalMemObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_mem_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* specifies a valid command-queue.
- *num_mem_objects* specifies the number of memory objects to release.
- *mem_objects* points to a list of valid memory objects.
- *num_events_in_wait_list* specifies the number of events in *event_wait_list*.
- *event_wait_list* points to the list of events that need to complete before **clEnqueueReleaseExternalMemObjectsKHR** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueReleaseExternalMemObjectsKHR** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and that of *command_queue* must be the same.
- *event* returns an event object that identifies this particular command and can be used to query or queue a wait for this particular command to complete. *event* can be `NULL` in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

Applications must release memory objects that were acquired using **clEnqueueReleaseExternalMemObjectsKHR** before using any external memory handles that were specified when the memory objects were created or accessing the external memory that these handles represent outside of OpenCL, otherwise behavior is undefined. This is to guarantee that the state of memory objects is up-to-date and that they are accessible outside of OpenCL.

The following restrictions shall apply:

- Each memory object must be released only once. Releasing a memory object multiple times

without acquiring it results in implementation-defined behavior.

- The release must be performed on a command-queue associated with a device that was one of the devices specified via `CL_MEM_DEVICE_HANDLE_LIST_KHR` when the memory object was imported using **clCreateBufferWithProperties** or **clCreateImageWithProperties**. If `CL_MEM_DEVICE_HANDLE_LIST_KHR` was not specified, the release can be performed on a command-queue associated with any device in the context.
- The memory object will be released for all devices specified via `CL_MEM_DEVICE_HANDLE_LIST_KHR` when the memory object was imported using **clCreateBufferWithProperties** or **clCreateImageWithProperties**. If `CL_MEM_DEVICE_HANDLE_LIST_KHR` was not specified, the memory object will be released for all devices in the context.

See "Example with Acquire / Release" provided in Sample Code for more details on how to use this API.

If *num_mem_objects* is 0 and *mem_objects* is `NULL`, the command will trivially succeed after its event dependencies are satisfied and will update its completion event.

clEnqueueReleaseExternalMemObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_mem_objects* is zero and *mem_objects* is not a `NULL` value, or if *num_mem_objects* is greater than 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if any of the memory objects in *mem_objects* is not a valid OpenCL memory object created using an external memory handle.
- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid command-queue, or
  - if device associated with *command_queue* is not one of the devices specified by `CL_MEM_DEVICE_HANDLE_LIST_KHR` at the time of creating one or more of *mem_objects*, or
  - if one or more of *mem_objects* belong to a context that does not contain a device associated with *command_queue*.
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is not 0, or
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or
  - if event objects in *event_wait_list* are not valid events.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST` if the execution status of any of the events in *event_wait_list* is a negative integer value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.2. Descriptions of External Memory Handle Types

This section describes external memory handle types that are added by extensions.

Applications can import the same payload into multiple OpenCL contexts and multiple times into a given OpenCL context. In all cases, each import operation must create a distinct memory object.

5.5.2.1. File Descriptor Handle Types

The `cl_khr_external_memory_opaque_fd` extension extends `cl_external_memory_handle_type_khr` to support the following new types of handles, and adds as a property that may be specified when creating a buffer or an image memory object from an external handle:

- `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_FD_KHR` specifies a POSIX file descriptor handle that has only limited valid usage outside of OpenCL and other compatible APIs. It must be compatible with the POSIX system calls `dup`, `dup2`, `close`, and the non-standard system call `dup3`. Additionally, it must be transportable over a socket using a `SCM_RIGHTS` control message. It owns a reference to the underlying memory resource represented by its memory object.

The `cl_khr_external_memory_dma_buf` extension extends `cl_external_memory_handle_type_khr` to support the following types of handles, and adds as a property that may be specified when creating a buffer or an image memory object from an external handle:

- `CL_EXTERNAL_MEMORY_HANDLE_DMA_BUF_KHR` is a file descriptor for a Linux dma_buf. Once a memory object created from a dma_buf has been acquired with **clEnqueueAcquireExternalMemObjectsKHR**, all access, including host access should be done through OpenCL, until a call to **clEnqueueReleaseExternalMemObjectsKHR** has completed. Any operations external to OpenCL on the dma_buf while it is acquired may result in undefined behavior.

  ```
  It owns a reference to the underlying memory resource represented by its
  memory object.
  ```

The `cl_khr_external_memory_android_hardware_buffer` extension extends `cl_external_memory_handle_type_khr` to support the following new handle type, and adds as a property that may be specified when creating a buffer or an image memory object from an external handle:

- `CL_EXTERNAL_MEMORY_HANDLE_ANDROID_HARDWARE_BUFFER_KHR` specifies an Android AHardwareBuffer object. If importing as a buffer, the implementation will infer the size of the buffer. If importing as an image, the implementation will infer the image format, dimensions, and data layout. The application is responsible for verifying that the AHardwareBuffer_UsageFlags are compatible with the OpenCL device, and *flags* passed to **clCreateImageWithProperties** and **clCreateBufferWithProperties**. The memory object created by importing an AHardwareBuffer will increment the reference count on the AHardwareBuffer, and decrement the reference count once the memory object is deleted.

For these extensions, importing memory from a file descriptor transfers ownership of the file descriptor from the application to the OpenCL implementation. The application must not perform any operations on the file descriptor after a successful import. The imported memory object holds a

reference to its payload.

5.5.2.2. NT Handle Types

The `cl_khr_external_memory_win32` extension extends `cl_external_memory_handle_type_khr` to support the following new types of handles, and adds as a property that may be specified when creating a buffer or an image memory object from an external handle:

- `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_KHR` specifies an NT handle that has only limited valid usage outside of OpenCL and other compatible APIs. It must be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying memory resource represented by its memory object.
- `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_KMT_KHR` specifies a global share handle that has only limited valid usage outside of OpenCL and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying memory resource represented by its memory object, and will therefore become invalid when all memory objects associated with it are destroyed.
- `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_NAME_KHR` specifies an NT handle name that has only limited valid usage outside of OpenCL and other compatible APIs. NT handle name is a null-terminated UTF-16 string naming the payload to import. It must be compatible with the functions `DuplicateHandle`, `CloseHandle`, `CompareObjectHandles`, `GetHandleInformation`, and `SetHandleInformation`. It owns a reference to the underlying memory resource represented by its memory object.

For these extensions, importing memory object payloads from Windows handles does not transfer ownership of the handle to the OpenCL implementation. For handle types defined as NT handles, the application must release handle ownership using the CloseHandle system call when the handle is no longer needed. For handle types defined as NT handles, the imported memory object holds a reference to its payload.

Note: Non-NT handle import operations do not add a reference to their associated payload. If the original object owning the payload is destroyed, all resources and handles sharing that payload will become invalid.

5.5.3. Unmapping Mapped Memory Objects

To enqueue a command to unmap a previously mapped region of a memory object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueUnmapMemObject(
    cl_command_queue command_queue,
    cl_mem memobj,
    void* mapped_ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* must be a valid host command-queue.
- *memobj* is a valid memory (buffer or image) object. The OpenCL context associated with *command_queue* and *memobj* must be the same.
- *mapped_ptr* is the host address returned by a previous call to **clEnqueueMapBuffer**, or **clEnqueueMapImage** for *memobj*.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before **clEnqueueUnmapMemObject** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueUnmapMemObject** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

Reads or writes from the host using the pointer returned by **clEnqueueMapBuffer** or **clEnqueueMapImage** are considered to be complete.

clEnqueueMapBuffer and **clEnqueueMapImage** increment the mapped count of the memory object. The initial mapped count value of the memory object is zero. Multiple calls to **clEnqueueMapBuffer**, or **clEnqueueMapImage** on the same memory object will increment this mapped count by appropriate number of calls. **clEnqueueUnmapMemObject** decrements the mapped count of the memory object.

clEnqueueMapBuffer, and **clEnqueueMapImage** act as synchronization points for a region of the buffer object being mapped.

clEnqueueUnmapMemObject returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_MEM_OBJECT` if *memobj* is not a valid memory object or is a pipe object.
- `CL_INVALID_VALUE` if *mapped_ptr* is not a valid pointer returned by **clEnqueueMapBuffer** or

clEnqueueMapImage for *memobj*.

- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* and *memobj* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

5.5.4. Accessing Mapped Regions of a Memory Object

This section describes the behavior of OpenCL commands that access mapped regions of a memory object.

The contents of the region of a memory object and associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) mapped for writing (i.e. `CL_MAP_WRITE` or `CL_MAP_WRITE_INVALIDATE_REGION` is set in *map_flags* argument to **clEnqueueMapBuffer**, or **clEnqueueMapImage**) are considered to be undefined until this region is unmapped.

Multiple commands in command-queues can map a region or overlapping regions of a memory object and associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) for reading (i.e. *map_flags* = `CL_MAP_READ`). The contents of the regions of a memory object mapped for reading can also be read by kernels and other OpenCL commands (such as **clEnqueueCopyBuffer**) executing on a device(s).

Mapping (and unmapping) overlapped regions in a memory object and/or associated memory objects (sub-buffer objects or 1D image buffer objects that overlap this region) for writing is an error and will result in `CL_INVALID_OPERATION` error returned by **clEnqueueMapBuffer**, or **clEnqueueMapImage**.

If a memory object is currently mapped for writing, the application must ensure that the memory object is unmapped before any enqueued kernels or commands that read from or write to this memory object or any of its associated memory objects (sub-buffer or 1D image buffer objects) or its parent object (if the memory object is a sub-buffer or 1D image buffer object) begin execution; otherwise the behavior is undefined.

If a memory object is currently mapped for reading, the application must ensure that the memory object is unmapped before any enqueued kernels or commands that write to this memory object or any of its associated memory objects (sub-buffer or 1D image buffer objects) or its parent object (if the memory object is a sub-buffer or 1D image buffer object) begin execution; otherwise the behavior is undefined.

A memory object is considered as mapped if there are one or more active mappings for the memory object irrespective of whether the mapped regions span the entire memory object.

Accessing the contents of the memory region referred to by the mapped pointer that has been

unmapped is undefined.

The mapped pointer returned by **clEnqueueMapBuffer** or **clEnqueueMapImage** can be used as the *ptr* argument value to **clEnqueueReadBuffer**, **clEnqueueWriteBuffer**, **clEnqueueReadBufferRect**, **clEnqueueWriteBufferRect**, **clEnqueueReadImage**, or **clEnqueueWriteImage** provided the rules described above are adhered to.

5.5.5. Migrating Memory Objects

Migrating memory objects is missing before version 1.2.

This section describes a mechanism for assigning which device an OpenCL memory object resides. A user may wish to have more explicit control over the location of their memory objects on creation. This could be used to:

- Ensure that an object is allocated on a specific device prior to usage.
- Preemptively migrate an object from one device to another.

To enqueue a command to indicate which device a set of memory objects should be associated with, call the function

```
// Provided by CL_VERSION_1_2
cl_int clEnqueueMigrateMemObjects(
    cl_command_queue command_queue,
    cl_uint num_mem_objects,
    const cl_mem* mem_objects,
    cl_mem_migration_flags flags,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueMigrateMemObjects is missing before version 1.2.

- *command_queue* is a valid host command-queue. The specified set of memory objects in *mem_objects* will be migrated to the OpenCL device associated with *command_queue* or to the host if the `CL_MIGRATE_MEM_OBJECT_HOST` has been specified.
- *num_mem_objects* is the number of memory objects specified in *mem_objects*.
- *mem_objects* is a pointer to a list of memory objects.
- *flags* is a bit-field that is used to specify migration options. The Memory Migration Flags describes the possible values for flags.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and

command_queue must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

Table 38. List of supported migration flags by **clEnqueueMigrateMemObjects**

Memory Migration Flags	Description
`CL_MIGRATE_MEM_OBJECT_HOST` missing before version 1.2.	This flag indicates that the specified set of memory objects are to be migrated to the host, regardless of the target command-queue.
`CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED` missing before version 1.2.	This flag indicates that the contents of the set of memory objects are undefined after migration. The specified set of memory objects are migrated to the device associated with *command_queue* without incurring the overhead of migrating their contents.

Typically, memory objects are implicitly migrated to a device for which enqueued commands, using the memory object, are targeted. **clEnqueueMigrateMemObjects** allows this migration to be explicitly performed ahead of the dependent commands. This allows a user to preemptively change the association of a memory object, through regular command-queue scheduling, in order to prepare for another upcoming command. This also permits an application to overlap the placement of memory objects with other unrelated operations before these memory objects are needed potentially hiding transfer latencies. Once the event, returned from **clEnqueueMigrateMemObjects**, has been marked `CL_COMPLETE` the memory objects specified in *mem_objects* have been successfully migrated to the device associated with *command_queue*. The migrated memory object shall remain resident on the device until another command is enqueued that either implicitly or explicitly migrates it away.

clEnqueueMigrateMemObjects can also be used to direct the initial placement of a memory object, after creation, possibly avoiding the initial overhead of instantiating the object on the first enqueued command to use it.

The user is responsible for managing the event dependencies, associated with this command, in order to avoid overlapping access to memory objects. Improperly specified event dependencies passed to **clEnqueueMigrateMemObjects** could result in undefined results.

clEnqueueMigrateMemObjects return `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_CONTEXT` if the context associated with *command_queue* and memory objects in *mem_objects* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.

- `CL_INVALID_MEM_OBJECT` if any of the memory objects in *mem_objects* is not a valid memory object.
- `CL_INVALID_VALUE` if *num_mem_objects* is zero or if *mem_objects* is `NULL`.
- `CL_INVALID_VALUE` if *flags* is not 0 or is not any of the values described in the table above.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE` if there is a failure to allocate memory for the specified set of memory objects in *mem_objects*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.6. Memory Object Queries

To get information that is common to all memory objects (buffer and image objects), call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetMemObjectInfo(
    cl_mem memobj,
    cl_mem_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *memobj* specifies the memory object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetMemObjectInfo** is described in the Memory Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Memory Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 39. List of supported param_names by **clGetMemObjectInfo**

Memory Object Info	Return type	Description
`CL_MEM_TYPE`	`cl_mem_object_type`	Returns one of the following values: `CL_MEM_OBJECT_BUFFER` if *memobj* is created with **clCreateBuffer**, **clCreateBufferWithProperties**, or **clCreateSubBuffer**. `CL_MEM_OBJECT_IMAGE2D` if *memobj* is created with **clCreateImage2D**. `CL_MEM_OBJECT_IMAGE3D` if *memobj* is created with **clCreateImage3D**. The value of *image_desc→image_type* if *memobj* is created with **clCreateImage** or **clCreateImageWithProperties**. `CL_MEM_OBJECT_PIPE` if *memobj* is created with **clCreatePipe**.
`CL_MEM_FLAGS`	`cl_mem_flags`	Return the flags argument value specified when *memobj* is created with **clCreateBuffer**, **clCreateBufferWithProperties**, **clCreateSubBuffer**, **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, **clCreateImage3D**, or **clCreatePipe**. If *memobj* is a sub-buffer the memory access qualifiers inherited from parent buffer is also returned.
`CL_MEM_SIZE`	`size_t`	Return actual size of the data store associated with *memobj* in bytes.

Memory Object Info	Return type	Description
CL_MEM_HOST_PTR	void*	If *memobj* is created with **clCreateBuffer**, **clCreateBufferWithProperties**, **clCreateImage**, **clCreateImageWithProperties**, **clCreateImage2D**, or **clCreateImage3D**, and CL_MEM_USE_HOST_PTR is specified in mem_flags, return the host_ptr argument value specified when *memobj* is created. Otherwise, if *memobj* is created with **clCreateSubBuffer**, and *memobj* is created from a buffer that was created with CL_MEM_USE_HOST_PTR specified in mem_flags, return the host_ptr passed to **clCreateBuffer** or **clCreateBufferWithProperties**, plus the origin value specified in buffer_create_info when *memobj* is created. Otherwise, returns NULL.
CL_MEM_MAP_COUNT [10]	cl_uint	Map count.
CL_MEM_REFERENCE_COUNT [11]	cl_uint	Return *memobj* reference count.
CL_MEM_CONTEXT	cl_context	Return context specified when memory object is created. If *memobj* is created using **clCreateSubBuffer**, the context associated with the memory object specified as the *buffer* argument to **clCreateSubBuffer** is returned.
CL_MEM_ASSOCIATED_MEMOBJECT missing before version 1.1.	cl_mem	Return memory object from which *memobj* is created. This returns the memory object specified as buffer argument to **clCreateSubBuffer** if *memobj* is a subbuffer object created using **clCreateSubBuffer**. This returns *image_desc→mem_object* if *memobj* is an image object created using **clCreateImage** or **clCreateImageWithProperties**. Otherwise, returns NULL.
CL_MEM_OFFSET missing before version 1.1.	size_t	Return offset if *memobj* is a sub-buffer object created using **clCreateSubBuffer**. This return 0 if *memobj* is not a subbuffer object.

Memory Object Info	Return type	Description
CL_MEM_USES_SVM_POINTER missing before version 2.0.	cl_bool	Return CL_TRUE if *memobj* is a buffer object that was created with CL_MEM_USE_HOST_PTR or is a sub-buffer object of a buffer object that was created with CL_MEM_USE_HOST_PTR and the *host_ptr* specified when the buffer object was created is a SVM pointer; otherwise returns CL_FALSE.
CL_MEM_PROPERTIES missing before version 3.0.	cl_mem_properties[]	Return the properties argument specified in **clCreateBufferWithProperties** or **clCreateImageWithProperties**. If the *properties* argument specified in **clCreateBufferWithProperties** or **clCreateImageWithProperties** used to create *memobj* was not NULL, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If *memobj* was created using **clCreateBuffer**, **clCreateSubBuffer**, **clCreateImage**, **clCreateImage2D**, or **clCreateImage3D**, or if the *properties* argument specified in **clCreateBufferWithProperties** or **clCreateImageWithProperties** was NULL, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.
CL_MEM_DX9_MEDIA_ADAPTER_TYPE_KHR provided by the cl_khr_dx9_media_sharing extension.	cl_dx9_media_adapter_type_khr	If *memobj* was created using **clCreateFromDX9MediaSurfaceKHR**, returns the *adapter_type* argument specified when *memobj* was created.
CL_MEM_DX9_MEDIA_SURFACE_INFO_KHR provided by the cl_khr_dx9_media_sharing extension.	cl_dx9_surface_info_khr	If *memobj* was created using **clCreateFromDX9MediaSurfaceKHR**, returns the *surface_info* argument specified when *memobj* was created.

Memory Object Info	Return type	Description
CL_MEM_D3D10_RESOURCE_KHR provided by the cl_khr_d3d10_sharing extension.	ID3D10Resource *	If *memobj* was created using **clCreateFromD3D10BufferKHR**, **clCreateFromD3D10Texture2DKHR**, or **clCreateFromD3D10Texture3DKHR**, returns the *resource* argument specified when *memobj* was created.
CL_MEM_D3D11_RESOURCE_KHR provided by the cl_khr_d3d11_sharing extension.	ID3D11Resource *	If *memobj* was created using **clCreateFromD3D11BufferKHR**, **clCreateFromD3D11Texture2DKHR**, or **clCreateFromD3D11Texture3DKHR**, returns the *resource* argument specified when *memobj* was created.
CL_MEM_DEVICE_ADDRESS_EXT provided by the cl_ext_buffer_device_address extension.	cl_mem_device_address_ext[]	If *memobj* was created using **clCreateBufferWithProperties** with the CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT property set to CL_TRUE, returns a list of device addresses for the buffer, one for each device in the context in the same order as the list of devices passed to **clCreateContext**.

clGetMemObjectInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_MEM_OBJECT if *memobj* is a not a valid memory object.
- CL_INVALID_OPERATION is returned for the CL_MEM_DEVICE_ADDRESS_EXT query if the cl_ext_buffer_device_address is not supported or if the buffer was not allocated with CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Memory Object Queries table and *param_value* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The following errors may be returned if the cl_khr_dx9_media_sharing extension is supported:

- CL_INVALID_DX9_MEDIA_SURFACE_KHR if *param_name* is CL_MEM_DX9_MEDIA_SURFACE_INFO_KHR and *memobj* was not created by calling **clCreateFromDX9MediaSurfaceKHR** from a Direct3D9 surface.

The following errors may be returned if the cl_khr_d3d10_sharing extension is supported:

- CL_INVALID_D3D10_RESOURCE_KHR if *param_name* is CL_MEM_D3D10_RESOURCE_KHR and *memobj* was not created by calling **clCreateFromD3D10BufferKHR**, **clCreateFromD3D10Texture2DKHR**, or

clCreateFromD3D10Texture3DKHR.

The following errors may be returned if the `cl_khr_d3d11_sharing` extension is supported:

- `CL_INVALID_D3D11_RESOURCE_KHR` if *param_name* is `CL_MEM_D3D11_RESOURCE_KHR` and *memobj* was not created by calling **clCreateFromD3D11BufferKHR**, **clCreateFromD3D11Texture2DKHR**, or **clCreateFromD3D11Texture3DKHR**.

5.5.7. Querying Media Surface Properties of Memory Objects Created From DirectX 9 Media Surfaces

Properties of media surface objects may be queried using **clGetMemObjectInfo** and **clGetImageInfo** with *param_name* `CL_MEM_DX9_MEDIA_ADAPTER_TYPE_KHR`, `CL_MEM_DX9_MEDIA_SURFACE_INFO_KHR` and `CL_IMAGE_DX9_MEDIA_PLANE_KHR`.

5.5.8. Querying Direct3D Properties of Memory Objects Created From Direct3D Resources

Properties of Direct3D 10 objects may be queried using **clGetMemObjectInfo** and **clGetImageInfo** with *param_name* `CL_MEM_D3D10_RESOURCE_KHR` and `CL_IMAGE_D3D10_SUBRESOURCE_KHR` respectively.

Properties of Direct3D 11 objects may be queried using **clGetMemObjectInfo** and **clGetImageInfo** with *param_name* `CL_MEM_D3D11_RESOURCE_KHR` and `CL_IMAGE_D3D11_SUBRESOURCE_KHR` respectively.

5.5.9. Querying OpenGL Object Information From an OpenCL Memory Object

To query the OpenGL object and object type used to create an OpenCL memory object, call the function

```
// Provided by cl_khr_gl_sharing
cl_int clGetGLObjectInfo(
    cl_mem memobj,
    cl_gl_object_type* gl_object_type,
    cl_GLuint* gl_object_name);
```

clGetGLObjectInfo is provided by the `cl_khr_gl_sharing` extension.

- *memobj* is the memory object to query.
- *gl_object_type* returns the type of OpenGL object attached to *memobj* and can be `CL_GL_OBJECT_BUFFER`, `CL_GL_OBJECT_TEXTURE2D`, `CL_GL_OBJECT_TEXTURE3D`, `CL_GL_OBJECT_TEXTURE2D_ARRAY`, `CL_GL_OBJECT_TEXTURE1D`, `CL_GL_OBJECT_TEXTURE1D_ARRAY`, `CL_GL_OBJECT_TEXTURE_BUFFER`, or `CL_GL_OBJECT_RENDERBUFFER`. If *gl_object_type* is `NULL`, it is ignored
- *gl_object_name* returns the OpenGL object name used to create *memobj*. If *gl_object_name* is `NULL`, it is ignored.

clGetGLObjectInfo returns `CL_SUCCESS` if the call was executed successfully. Otherwise, it returns

one of the following errors:

- `CL_INVALID_MEM_OBJECT` if *memobj* is not a valid OpenCL memory object.
- `CL_INVALID_GL_OBJECT` if there is no OpenGL object associated with *memobj*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To query additional information about the OpenGL texture object associated with an OpenCL memory object, call the function

```
// Provided by cl_khr_gl_sharing
cl_int clGetGLTextureInfo(
    cl_mem memobj,
    cl_gl_texture_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetGLTextureInfo is provided by the `cl_khr_gl_sharing` extension.

- *memobj* is the memory object to query.
- *param_name* specifies what additional information about the OpenGL texture object associated with *memobj* to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetGLTextureInfo** is described in the table below.
- *param_value* is a pointer to memory where the result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the OpenGL Texture Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data copied to *param_value*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 40. OpenGL texture info that may be queried with **clGetGLTextureInfo**

`cl_gl_texture_info`	**Return Type**	**Info. Returned in** *param_value*
`CL_GL_TEXTURE_TARGET` provided by the `cl_khr_gl_sharing` extension.	`GLenum`	The *texture_target* argument specified in **clCreateFromGLTexture**.
`CL_GL_MIPMAP_LEVEL` provided by the `cl_khr_gl_sharing` extension.	`GLint`	The *miplevel* argument specified in **clCreateFromGLTexture**.

cl_gl_texture_info	Return Type	Info. Returned in *param_value*
CL_GL_NUM_SAMPLES provided by the cl_khr_gl_msaa_sharing extension.	GLsizei	The *samples* argument passed to glTexImage2DMultisample or glTexImage3DMultisample. If *image* is not a MSAA texture, 1 is returned.

clGetGLTextureInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_MEM_OBJECT if *memobj* is not a valid OpenCL memory object.
- CL_INVALID_GL_OBJECT if there is no OpenGL texture object associated with *memobj*.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the OpenGL Texture Queries table and *param_value* is not NULL.
- CL_INVALID_VALUE if *param_value* and *param_value_size_ret* are NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.10. Sharing Memory Objects Created From Media Surfaces Between a Media Adapter and OpenCL

To acquire OpenCL memory objects that have been created from a media surface, call the function

```
// Provided by cl_khr_dx9_media_sharing
cl_int clEnqueueAcquireDX9MediaSurfacesKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireDX9MediaSurfacesKHR is provided by the cl_khr_dx9_media_sharing extension.

- *command_queue* is a valid command-queue.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from media surfaces.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this

particular command can be executed. If *event_wait_list* is NULL, then this particular command does not wait on any event to complete. If *event_wait_list* is NULL, *num_events_in_wait_list* must be 0. If *event_wait_list* is not NULL, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.

- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is NULL or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not NULL, *event* must not refer to an element of the *event_wait_list* array.

The media surfaces are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

OpenCL memory objects created from media surfaces must be acquired before they can be used by any OpenCL commands queued to a command-queue. If an OpenCL memory object created from a media surface is used while it is not currently acquired by OpenCL, the call attempting to use that OpenCL memory object will return CL_DX9_MEDIA_SURFACE_NOT_ACQUIRED_KHR.

If CL_CONTEXT_INTEROP_USER_SYNC is not specified as CL_TRUE during context creation, **clEnqueueAcquireDX9MediaSurfacesKHR** provides the synchronization guarantee that any media adapter API calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireDX9MediaSurfacesKHR** is called will complete executing before *event* reports completion and before the execution of any subsequent OpenCL work issued in *command_queue* begins. If the context was created with properties specifying CL_CONTEXT_INTEROP_USER_SYNC as CL_TRUE, the user is responsible for guaranteeing that any media adapter API calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireDX9MediaSurfacesKHR** is called have completed before calling **clEnqueueAcquireDX9MediaSurfacesKHR** .

clEnqueueAcquireDX9MediaSurfacesKHR returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is NULL then the function does nothing and returns CL_SUCCESS. Otherwise it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from media surfaces.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from a device that can share the media surface referenced by *mem_objects*.
- CL_DX9_MEDIA_SURFACE_ALREADY_ACQUIRED_KHR if memory objects in *mem_objects* have previously been acquired using **clEnqueueAcquireDX9MediaSurfacesKHR** but have not been released using **clEnqueueReleaseDX9MediaSurfacesKHR**.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list*

are not valid events.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release OpenCL memory objects that have been created from media surfaces, call the function

```
// Provided by cl_khr_dx9_media_sharing
cl_int clEnqueueReleaseDX9MediaSurfacesKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReleaseDX9MediaSurfacesKHR is provided by the `cl_khr_dx9_media_sharing` extension.

- *num_objects* is the number of memory objects to be released in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from media surfaces.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The media surfaces are released by the OpenCL context associated with *command_queue*.

OpenCL memory objects created from media surfaces which have been acquired by OpenCL must be released by OpenCL before they may be accessed by the media adapter API. Accessing a media surface while its corresponding OpenCL memory object is acquired is in error and will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

If `CL_CONTEXT_INTEROP_USER_SYNC` is not specified as `CL_TRUE` during context creation, **clEnqueueReleaseDX9MediaSurfacesKHR** provides the synchronization guarantee that any calls to media adapter APIs involving the interop device(s) used in the OpenCL context made after the call to **clEnqueueReleaseDX9MediaSurfacesKHR** will not start executing until after all events in *event_wait_list* are complete and all work already submitted to *command_queue* completes execution. If the context was created with properties specifying `CL_CONTEXT_INTEROP_USER_SYNC` as

CL_TRUE, the user is responsible for guaranteeing that any media adapter API calls involving the interop device(s) used in the OpenCL context made after **clEnqueueReleaseDX9MediaSurfacesKHR** will not start executing until after event returned by **clEnqueueReleaseDX9MediaSurfacesKHR** reports completion.

clEnqueueReleaseDX9MediaSurfacesKHR returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and <*mem_objects*> is NULL the function does nothing and returns CL_SUCCESS. Otherwise it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from valid media surfaces.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from a media object.
- CL_DX9_MEDIA_SURFACE_NOT_ACQUIRED_KHR if memory objects in *mem_objects* have not previously been acquired using **clEnqueueAcquireDX9MediaSurfacesKHR**, or have been released using **clEnqueueReleaseDX9MediaSurfacesKHR** since the last time that they were acquired.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list*> is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.11. Sharing Memory Objects Created From Direct3D Resources Between Direct3D and OpenCL Contexts

To acquire OpenCL memory objects that have been created from Direct3D 10 resources, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_int clEnqueueAcquireD3D10ObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireD3D10ObjectsKHR is provided by the cl_khr_d3d10_sharing extension.

- *command_queue* is a valid command-queue.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.

- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from Direct3D 10 resources.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The Direct3D 10 objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

OpenCL memory objects created from Direct3D 10 resources must be acquired before they can be used by any OpenCL commands queued to a command-queue. If an OpenCL memory object created from a Direct3D 10 resource is used while it is not currently acquired by OpenCL, the behavior is undefined. Implementations may fail the execution of commands attempting to use that OpenCL memory object and set their associated event's execution status to `CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR`.

If `CL_CONTEXT_INTEROP_USER_SYNC` is not specified as `CL_TRUE` during context creation, **clEnqueueAcquireD3D10ObjectsKHR** provides the synchronization guarantee that any Direct3D 10 calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireD3D10ObjectsKHR** is called will complete executing before *event* reports completion and before the execution of any subsequent OpenCL work issued in *command_queue* begins. If the context was created with properties specifying `CL_CONTEXT_INTEROP_USER_SYNC` as `CL_TRUE`, the user is responsible for guaranteeing that any Direct3D 10 calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireD3D10ObjectsKHR** is called have completed before calling **clEnqueueAcquireD3D10ObjectsKHR**.

clEnqueueAcquireD3D10ObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` then the function does nothing and returns `CL_SUCCESS`. Otherwise it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if *num_objects* > 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 10 resources.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* was not created from an Direct3D 10 context.
- `CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR` if memory objects in *mem_objects* have previously been

acquired using **clEnqueueAcquireD3D10ObjectsKHR** but have not been released using **clEnqueueReleaseD3D10ObjectsKHR**.

- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release OpenCL memory objects that have been created from Direct3D 10 resources, call the function

```
// Provided by cl_khr_d3d10_sharing
cl_int clEnqueueReleaseD3D10ObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReleaseD3D10ObjectsKHR is provided by the `cl_khr_d3d10_sharing` extension.

- *num_objects* is the number of memory objects to be released in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from Direct3D 10 resources.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The Direct3D 10 objects are released by the OpenCL context associated with *command_queue*.

OpenCL memory objects created from Direct3D 10 resources which have been acquired by OpenCL must be released by OpenCL before they may be accessed by Direct3D 10. Accessing a Direct3D 10 resource while its corresponding OpenCL memory object is acquired is in error and will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

If `CL_CONTEXT_INTEROP_USER_SYNC` is not specified as `CL_TRUE` during context creation,

clEnqueueReleaseD3D10ObjectsKHR provides the synchronization guarantee that any calls to Direct3D 10 calls involving the interop device(s) used in the OpenCL context made after the call to **clEnqueueReleaseD3D10ObjectsKHR** will not start executing until after all events in *event_wait_list* are complete and all work already submitted to *command_queue* completes execution. If the context was created with properties specifying `CL_CONTEXT_INTEROP_USER_SYNC` as `CL_TRUE`, the user is responsible for guaranteeing that any Direct3D 10 calls involving the interop device(s) used in the OpenCL context made after **clEnqueueReleaseD3D10ObjectsKHR** will not start executing until after event returned by **clEnqueueReleaseD3D10ObjectsKHR** reports completion.

clEnqueueReleaseD3D10ObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` the function does nothing and returns `CL_SUCCESS`. Otherwise it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if *num_objects* > 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 10 resources.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* was not created from a Direct3D 10 device.
- `CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR` if memory objects in *mem_objects* have not previously been acquired using **clEnqueueAcquireD3D10ObjectsKHR**, or have been released using **clEnqueueReleaseD3D10ObjectsKHR** since the last time that they were acquired.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list*> is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To acquire OpenCL memory objects that have been created from Direct3D 11 resources, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_int clEnqueueAcquireD3D11ObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireD3D11ObjectsKHR is provided by the `cl_khr_d3d11_sharing` extension.

- *command_queue* is a valid command-queue.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from Direct3D 11 resources.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The Direct3D 11 objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

OpenCL memory objects created from Direct3D 11 resources must be acquired before they can be used by any OpenCL commands queued to a command-queue. If an OpenCL memory object created from a Direct3D 11 resource is used while it is not currently acquired by OpenCL, the behavior is undefined. Implementations may fail the execution of commands attempting to use that OpenCL memory object and set their associated event's execution status to `CL_D3D11_RESOURCE_NOT_ACQUIRED_KHR`.

If `CL_CONTEXT_INTEROP_USER_SYNC` is not specified as `CL_TRUE` during context creation, **clEnqueueAcquireD3D11ObjectsKHR** provides the synchronization guarantee that any Direct3D 11 calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireD3D11ObjectsKHR** is called will complete executing before *event* reports completion and before the execution of any subsequent OpenCL work issued in *command_queue* begins. If the context was created with properties specifying `CL_CONTEXT_INTEROP_USER_SYNC` as `CL_TRUE`, the user is responsible for guaranteeing that any Direct3D 11 calls involving the interop device(s) used in the OpenCL context made before **clEnqueueAcquireD3D11ObjectsKHR** is called have completed before calling **clEnqueueAcquireD3D11ObjectsKHR**.

clEnqueueAcquireD3D11ObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` then the function does nothing and returns `CL_SUCCESS`. Otherwise it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if *num_objects* > 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 11 resources.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* was not created from an Direct3D

11 context.

- `CL_D3D11_RESOURCE_ALREADY_ACQUIRED_KHR` if memory objects in *mem_objects* have previously been acquired using **clEnqueueAcquireD3D11ObjectsKHR** but have not been released using **clEnqueueReleaseD3D11ObjectsKHR**.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release OpenCL memory objects that have been created from Direct3D 11 resources, call the function

```
// Provided by cl_khr_d3d11_sharing
cl_int clEnqueueReleaseD3D11ObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReleaseD3D11ObjectsKHR is provided by the `cl_khr_d3d11_sharing` extension.

- *num_objects* is the number of memory objects to be released in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from Direct3D 11 resources.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The Direct3D 11 objects are released by the OpenCL context associated with *command_queue*.

OpenCL memory objects created from Direct3D 11 resources which have been acquired by OpenCL must be released by OpenCL before they may be accessed by Direct3D 11. Accessing a Direct3D 11 resource while its corresponding OpenCL memory object is acquired is in error and will result in undefined behavior, including but not limited to possible OpenCL errors, data corruption, and

program termination.

If CL_CONTEXT_INTEROP_USER_SYNC is not specified as CL_TRUE during context creation, **clEnqueueReleaseD3D11ObjectsKHR** provides the synchronization guarantee that any calls to Direct3D 11 calls involving the interop device(s) used in the OpenCL context made after the call to **clEnqueueReleaseD3D11ObjectsKHR** will not start executing until after all events in *event_wait_list* are complete and all work already submitted to *command_queue* completes execution. If the context was created with properties specifying CL_CONTEXT_INTEROP_USER_SYNC as CL_TRUE, the user is responsible for guaranteeing that any Direct3D 11 calls involving the interop device(s) used in the OpenCL context made after **clEnqueueReleaseD3D11ObjectsKHR** will not start executing until after event returned by **clEnqueueReleaseD3D11ObjectsKHR** reports completion.

clEnqueueReleaseD3D11ObjectsKHR returns CL_SUCCESS if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is NULL the function does nothing and returns CL_SUCCESS. Otherwise it returns one of the following errors:

- CL_INVALID_VALUE if *num_objects* is zero and *mem_objects* is not a NULL value or if *num_objects* > 0 and *mem_objects* is NULL.
- CL_INVALID_MEM_OBJECT if memory objects in *mem_objects* are not valid OpenCL memory objects or if memory objects in *mem_objects* have not been created from Direct3D 11 resources.
- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* was not created from a Direct3D 11 device.
- CL_D3D11_RESOURCE_NOT_ACQUIRED_KHR if memory objects in *mem_objects* have not previously been acquired using **clEnqueueAcquireD3D11ObjectsKHR**, or have been released using **clEnqueueReleaseD3D11ObjectsKHR** since the last time that they were acquired.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list*> is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.12. Sharing Memory Objects Created From EGL Resources Between EGL and OpenCL Contexts

To acquire OpenCL memory objects that have been created from EGL resources, call the function

```
// Provided by cl_khr_egl_image
cl_int clEnqueueAcquireEGLObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireEGLObjectsKHR is provided by the `cl_khr_egl_image` extension.

- *command_queue* is a valid command-queue.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from EGL resources, within the context associate with command_queue.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The EGL objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

OpenCL memory objects created from EGL resources must be acquired before they can be used by any OpenCL commands queued to a command-queue. If an OpenCL memory object created from a EGL resource is used while it is not currently acquired by OpenCL, the behavior is undefined. Implementations may fail the execution of commands attempting to use that OpenCL memory object and set their associated event's execution status to `CL_EGL_RESOURCE_NOT_ACQUIRED_KHR`.

clEnqueueAcquireEGLObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` then the function does nothing and returns `CL_SUCCESS`. Otherwise it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if num_objects > 0 and mem_objects is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects in the context associated with *command_queue*.

- `CL_INVALID_EGL_OBJECT_KHR` if memory objects in *mem_objects* have not been created from EGL resources.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release OpenCL memory objects that have been created from EGL resources, call the function

```
// Provided by cl_khr_egl_image
cl_int clEnqueueReleaseEGLObjectsKHR(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReleaseEGLObjectsKHR is provided by the `cl_khr_egl_image` extension.

- *command_queue* is a valid command-queue.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that were created from EGL resources, within the context associate with command_queue.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The EGL objects are released by the OpenCL context associated with <command_queue>.

OpenCL memory objects created from EGL resources which have been acquired by OpenCL must be released by OpenCL before they may be accessed by EGL or by EGL client APIs. Accessing a EGL resource while its corresponding OpenCL memory object is acquired is in error and will result in

undefined behavior, including but not limited to possible OpenCL errors, data corruption, and program termination.

clEnqueueReleaseEGLObjectsKHR returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` then the function does nothing and returns `CL_SUCCESS`. Otherwise it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if num_objects > 0 and mem_objects is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects in the context associated with *command_queue*.
- `CL_INVALID_EGL_OBJECT_KHR` if memory objects in *mem_objects* have not been created from EGL resources.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.13. Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects

To acquire OpenCL memory objects that have been created from OpenGL objects, call the function

```
// Provided by cl_khr_gl_sharing
cl_int clEnqueueAcquireGLObjects(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueAcquireGLObjects is provided by the `cl_khr_gl_sharing` extension.

- *command_queue* is a valid command-queue. All devices used to create the OpenCL context associated with *command_queue* must support acquiring shared OpenCL/OpenGL objects. This constraint is enforced at context creation time.
- *num_objects* is the number of memory objects to be acquired in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that correspond to OpenGL objects.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this

particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.

- *event* returns an event object that identifies this command and can be used to query wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If an OpenGL context is bound to the current thread, then any OpenGL commands which

1. affect or access the contents of a memory object listed in the *mem_objects* list, and
2. were issued on that OpenGL context prior to the call to **clEnqueueAcquireGLObjects**

will complete before execution of any OpenCL commands following the **clEnqueueAcquireGLObjects** which affect or access any of those memory objects. If a non-`NULL` *event* object is returned, it will report completion only after completion of such OpenGL commands.

These objects need to be acquired before they can be used by any OpenCL commands queued to a command-queue or the behavior is undefined. The OpenGL objects are acquired by the OpenCL context associated with *command_queue* and can therefore be used by all command-queues associated with the OpenCL context.

clEnqueueAcquireGLObjects returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` the function does nothing and returns `CL_SUCCESS`. Otherwise, it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if *num_objects* > 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* was not created from an OpenGL context
- `CL_INVALID_GL_OBJECT` if memory objects in *mem_objects* have not been created from an OpenGL object(s).
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release OpenCL memory objects that have been created from OpenGL objects, call the function

```
// Provided by cl_khr_gl_sharing
cl_int clEnqueueReleaseGLObjects(
    cl_command_queue command_queue,
    cl_uint num_objects,
    const cl_mem* mem_objects,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueReleaseGLObjects is provided by the `cl_khr_gl_sharing` extension.

- *num_objects* is the number of memory objects to be released in *mem_objects*.
- *mem_objects* is a pointer to a list of OpenCL memory objects that correspond to OpenGL objects.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If an OpenGL context is bound to the current thread, then any OpenGL commands which

1. affect or access the contents of the memory objects listed in the *mem_objects* list, and
2. are issued on that context after the call to **clEnqueueReleaseGLObjects**

will not execute until after execution of any OpenCL commands preceding the

clEnqueueReleaseGLObjects which affect or access any of those memory objects. If a non-`NULL` *event* object is returned, it will report completion before execution of such OpenGL commands.

These objects need to be released before they can be used by OpenGL. The OpenGL objects are released by the OpenCL context associated with *command_queue*.

clEnqueueReleaseGLObjects returns `CL_SUCCESS` if the function is executed successfully. If *num_objects* is 0 and *mem_objects* is `NULL` the function does nothing and returns `CL_SUCCESS`. Otherwise, it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_objects* is zero and *mem_objects* is not a `NULL` value or if *num_objects* > 0 and *mem_objects* is `NULL`.
- `CL_INVALID_MEM_OBJECT` if memory objects in *mem_objects* are not valid OpenCL memory objects.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid command-queue.

- `CL_INVALID_CONTEXT` if context associated with *command_queue* was not created from an OpenGL context
- `CL_INVALID_GL_OBJECT` if memory objects in *mem_objects* have not been created from an OpenGL object(s).
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.5.13.1. Synchronizing Access to Memory Objects Shared With EGL or OpenGL

When sharing objects such as EGL images (if the `cl_khr_egl_image` extension is supported) or OpenGL buffers, textures, and renderbuffers (if the `cl_khr_gl_sharing` extension is supported), in order to ensure data integrity, the application is responsible for synchronizing access to shared memory objects through the other API with which such objects are shared.

Failure to provide such synchronization may result in race conditions and other undefined behavior including non-portability between implementations.

Prior to acquiring objects shared with the other API, the application must ensure that any pending operations in that API which accesses the objects specified in *mem_objects* have completed.

Depending on the application and the implementation, there are two extensions which may be used to synchronize with other APIs:

5.5.13.1.1. Synchronization With EGL and EGL Client APIs

When sharing with an EGL context via the `cl_khr_egl_image` extension, if the `cl_khr_egl_event` extension is supported, and the EGL context in question supports fence sync objects, *explicit synchronization* with EGL or EGL client APIs can be achieved as described in the Explicit Synchronization Using EGL Fence Sync Objects section.

If the `cl_khr_egl_event` extension is not supported, completion of EGL client API commands may be determined by issuing and waiting for completion of commands such as `glFinish` or `vgFinish` on all client API contexts with pending references to these objects.

5.5.13.1.2. Synchronization With OpenGL

When sharing with an OpenGL context via the `cl_khr_gl_sharing` extension, the OpenCL implementation will ensure that any such pending OpenGL operations are complete for an OpenGL context bound to the same thread as the OpenCL context. This is referred to as *implicit synchronization*.

If the `cl_khr_gl_event` extension is supported, and the OpenGL context in question supports fence sync objects, *explicit synchronization* with OpenGL can be achieved as described in the Explicit Synchronization Using OpenGL Fence Sync Objects section.

If the `cl_khr_gl_event` extension is not supported, completion of OpenGL commands may be determined by issuing and waiting for completion of a `glFinish` command on all OpenGL contexts with pending references to these objects.

5.5.13.1.3. General Considerations for Synchronization With Other APIs

Some implementations may offer other efficient synchronization methods. If such methods exist they will be described in platform-specific documentation.

Note that no synchronization method other than `glFinish` is portable between all OpenGL implementations and all OpenCL implementations. While this is the only way to ensure completion that is portable to all platforms, `glFinish` is an expensive operation and its use should be avoided if the `cl_khr_egl_event` or `cl_khr_gl_event` extensions are supported on a platform.

5.5.13.1.4. Synchronizing OpenCL Operations With Other APIs

After releasing a shared memory object, the application is responsible for ensuring that any pending OpenCL operations which access the objects specified in *mem_objects* have completed prior to executing subsequent commands in the other API which reference these objects.

This may be accomplished portably by calling **clWaitForEvents** with the event object returned by **clEnqueueReleaseGLObjects**, or by calling **clFinish**. As above, some implementations may offer more efficient methods.

The application is responsible for maintaining the proper order of operations if the OpenCL context and the other API context are in separate threads.

If an OpenGL context is bound to a thread other than the one in which **clEnqueueReleaseGLObjects** is called, changes to any of the objects in *mem_objects* may not be visible to that context without additional steps being taken by the application. For an OpenGL 3.1 (or later) context, the requirements are described in Appendix D (“Shared Objects and Multiple Contexts”) of the OpenGL 3.1 Specification. For prior versions of OpenGL, the requirements are implementation-dependent.

Attempting to access the data store of an OpenGL object after it has been acquired by OpenCL and before it has been released will result in undefined behavior. Similarly, attempting to access a shared OpenCL/OpenGL object from OpenCL before it has been acquired by the OpenCL command-queue, or after it has been released, will result in undefined behavior.

5.6. Shared Virtual Memory

Shared virtual memory is missing before version 2.0.

Shared virtual memory (a.k.a. SVM) allows the host and kernels executing on devices to directly share complex, pointer-containing data structures such as trees and linked lists. It also eliminates the need to marshal data between the host and devices. As a result, SVM substantially simplifies OpenCL programming and may improve performance.

5.6.1. SVM Sharing Granularity: Coarse- and Fine- Grained Sharing

OpenCL maintains memory consistency in a coarse-grained fashion in regions of buffers. We call this coarse-grained sharing. Many platforms such as those with integrated CPU-GPU processors and ones using the SVM-related PCI-SIG IOMMU services can do better, and can support sharing at a granularity smaller than a buffer. We call this fine-grained sharing.

- Coarse-grained sharing: Coarse-grain sharing may be used for memory and virtual pointer sharing between multiple devices as well as between the host and one or more devices. The shared memory region is a memory buffer allocated using **clSVMAlloc**. Memory consistency is guaranteed at synchronization points and the host can use calls to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or create a `cl_mem` buffer object using the SVM pointer and use OpenCL's existing host API functions **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** to update regions of the buffer. What coarse-grain buffer SVM adds to OpenCL's earlier buffer support are the ability to share virtual memory pointers and a guarantee that concurrent access to the same memory allocation from multiple kernels on a single device is valid. The coarse-grain buffer SVM provides a memory consistency model similar to the global memory consistency model described in *sections 3.3.1* and *3.4.3* of the OpenCL 1.2 specification. This memory consistency applies to the regions of buffers being shared in a coarse-grained fashion. It is enforced at the synchronization points between commands enqueued to command-queues in a single context with the additional consideration that multiple kernels concurrently running on the same device may safely share the data.
- Fine-grained sharing: Shared virtual memory where memory consistency is maintained at a granularity smaller than a buffer. How fine-grained SVM is used depends on whether the device supports SVM atomic operations.
  - If SVM atomic operations are supported, they provide memory consistency for loads and stores by the host and kernels executing on devices supporting SVM. This means that the host and devices can concurrently read and update the same memory. The consistency provided by SVM atomics is in addition to the consistency provided at synchronization points. There is no need for explicit calls to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** on a `cl_mem` buffer object created using the SVM pointer.
  - If SVM atomic operations are not supported, the host and devices can concurrently read the same memory locations and can concurrently update non-overlapping memory regions, but attempts to update the same memory locations are undefined. Memory consistency is guaranteed at synchronization points without the need for explicit calls to **clEnqueueSVMMap** and **clEnqueueSVMUnmap** or **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** on a `cl_mem` buffer object created using the SVM pointer.
- There are two kinds of fine-grain sharing support. Devices may support either fine-grain buffer sharing or fine-grain system sharing.
  - Fine-grain buffer sharing provides fine-grain SVM only within buffers and is an extension of coarse-grain sharing. To support fine-grain buffer sharing in an OpenCL context, all devices in the context must support `CL_DEVICE_SVM_FINE_GRAIN_BUFFER`.
  - Fine-grain system sharing enables fine-grain sharing of the host's entire virtual memory, including memory regions allocated by the system **malloc** API. OpenCL buffer objects are unnecessary and programmers can pass pointers allocated using **malloc** to OpenCL kernels.

As an illustration of fine-grain SVM using SVM atomic operations to maintain memory consistency, consider the following example. The host and a set of devices can simultaneously access and update a shared work-queue data structure holding work-items to be done. The host can use atomic operations to insert new work-items into the queue at the same time as the devices using similar atomic operations to remove work-items for processing.

It is the programmer's responsibility to ensure that no host code or executing kernels attempt to access a shared memory region after that memory is freed. We require the SVM implementation to work with either 32- or 64- bit host applications subject to the following requirement: the address space size must be the same for the host and all OpenCL devices in the context.

To allocate a shared virtual memory buffer (referred to as a SVM buffer) that can be shared by the host and all devices in an OpenCL context that support shared virtual memory, call the function

```
// Provided by CL_VERSION_2_0
void* clSVMAlloc(
    cl_context context,
    cl_svm_mem_flags flags,
    size_t size,
    cl_uint alignment);
```

clSVMAlloc is missing before version 2.0.

- *context* is a valid OpenCL context used to create the SVM buffer.
- *flags* is a bit-field that is used to specify allocation and usage information. The SVM Memory Flags table describes the possible values for *flags*.
- *size* is the size in bytes of the SVM buffer to be allocated.
- *alignment* is the minimum alignment in bytes that is required for the newly created buffers memory region. It must be a power of two up to the largest data type supported by the OpenCL device. For the full profile, the largest data type is long16. For the embedded profile, it is long16 if the device supports 64-bit integers; otherwise it is int16. If alignment is 0, a default alignment will be used that is equal to the size of largest data type supported by the OpenCL implementation.

Table 41. List of supported SVM memory flag values

SVM Memory Flags	Description
`CL_MEM_READ_WRITE`	This flag specifies that the SVM buffer will be read and written by a kernel. This is the default.

SVM Memory Flags	Description
CL_MEM_WRITE_ONLY	This flag specifies that the SVM buffer will be written but not read by a kernel. Reading from a SVM buffer created with CL_MEM_WRITE_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE and CL_MEM_WRITE_ONLY are mutually exclusive.
CL_MEM_READ_ONLY	This flag specifies that the SVM buffer object is a read-only memory object when used inside a kernel. Writing to a SVM buffer created with CL_MEM_READ_ONLY inside a kernel is undefined. CL_MEM_READ_WRITE or CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive.
CL_MEM_SVM_FINE_GRAIN_BUFFER missing before version 2.0.	This specifies that the application wants the OpenCL implementation to do a fine-grained allocation.
CL_MEM_SVM_ATOMICS missing before version 2.0.	This flag is valid only if CL_MEM_SVM_FINE_GRAIN_BUFFER is specified in flags. It is used to indicate that SVM atomic operations can control visibility of memory accesses in this SVM buffer.

If CL_MEM_SVM_FINE_GRAIN_BUFFER is not specified, the buffer can be created as a coarse grained SVM allocation. Similarly, if CL_MEM_SVM_ATOMICS is not specified, the buffer can be created without support for SVM atomic operations (refer to an OpenCL kernel language specifications).

Calling **clSVMAlloc** does not itself provide consistency for the shared memory region. When the host cannot use the SVM atomic operations, it must rely on OpenCL's guaranteed memory consistency at synchronization points.

For SVM to be used efficiently, the host and any devices sharing a buffer containing virtual memory pointers should have the same endianness. If the context passed to **clSVMAlloc** has devices with mixed endianness and the OpenCL implementation is unable to implement SVM because of that mixed endianness, **clSVMAlloc** will fail and return NULL.

Although SVM is generally not supported for image objects, **clCreateImage** and **clCreateImageWithProperties** may create an image from a buffer (a 1D image from a buffer or a 2D image from buffer) if the buffer specified in its image description parameter is a SVM buffer. Such images have a linear memory representation so their memory can be shared using SVM. However, fine grained sharing and atomics are not supported for image reads and writes in a kernel.

clSVMAlloc returns a valid non-NULL shared virtual memory address if the SVM buffer is

successfully allocated. Otherwise, like **malloc**, it returns a `NULL` pointer value. **clSVMAlloc** will fail if

- *context* is not a valid context, or no devices in *context* support SVM.
- *flags* does not contain `CL_MEM_SVM_FINE_GRAIN_BUFFER` but does contain `CL_MEM_SVM_ATOMICS`.
- *flags* contains `CL_MEM_IMMUTABLE_EXT`.
- Values specified in *flags* do not follow rules described for supported values in the SVM Memory Flags table.
- `CL_MEM_SVM_FINE_GRAIN_BUFFER` or `CL_MEM_SVM_ATOMICS` is specified in *flags* and these are not supported by at least one device in *context*.
- The values specified in *flags* are not valid, i.e. do not match those defined in the SVM Memory Flags table.
- *size* is 0 or > `CL_DEVICE_MAX_MEM_ALLOC_SIZE` value for any device in *context*.
- *alignment* is not a power of two or the OpenCL implementation cannot support the specified alignment for at least one device in *context*.
- There was a failure to allocate resources.

To free a shared virtual memory buffer allocated using **clSVMAlloc**, call the function

```
// Provided by CL_VERSION_2_0
void clSVMFree(
    cl_context context,
    void* svm_pointer);
```

clSVMFree is missing before version 2.0.

- *context* is a valid OpenCL context used to create the SVM buffer. If no devices in *context* support SVM, no action occurs.
- *svm_pointer* must be the value returned by a call to **clSVMAlloc**. If a `NULL` pointer is passed in *svm_pointer*, no action occurs.

Note that **clSVMFree** does not wait for previously enqueued commands that may be using *svm_pointer* to finish before freeing *svm_pointer*. It is the responsibility of the application to make sure that enqueued commands that use *svm_pointer* have finished before freeing *svm_pointer*. This can be done by enqueuing a blocking operation such as **clFinish**, **clWaitForEvents**, **clEnqueueReadBuffer** or by registering a callback with the events associated with enqueued commands and when the last enqueued command has finished freeing *svm_pointer*.

The behavior of using *svm_pointer* after it has been freed is undefined. In addition, if a buffer object is created using **clCreateBuffer** or **clCreateBufferWithProperties** with *svm_pointer*, the buffer object must first be released before the *svm_pointer* is freed.

The **clEnqueueSVMFree** API can also be used to enqueue a callback to free the shared virtual memory buffer allocated using **clSVMAlloc** or a shared system memory pointer.

To enqueue a command to free the shared virtual memory allocated using **clSVMAlloc** or a shared

system memory pointer, call the function

```
// Provided by CL_VERSION_2_0
cl_int clEnqueueSVMFree(
    cl_command_queue command_queue,
    cl_uint num_svm_pointers,
    void* svm_pointers[],
    void (CL_CALLBACK* pfn_free_func)(cl_command_queue queue, cl_uint
num_svm_pointers, void* svm_pointers[], void* user_data),
    void* user_data,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMFree is missing before version 2.0.

- *command_queue* is a valid host command-queue.
- *svm_pointers* and *num_svm_pointers* specify shared virtual memory pointers to be freed. Each pointer in *svm_pointers* that was allocated using **clSVMAlloc** must have been allocated from the same context from which *command_queue* was created. The memory associated with *svm_pointers* can be reused or freed after the function returns.
- *pfn_free_func* specifies the callback function to be called to free the SVM pointers. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. *pfn_free_func* takes four arguments: *queue* which is the command-queue in which **clEnqueueSVMFree** was enqueued, the count and list of SVM pointers to free and *user_data* which is a pointer to user specified data. If *pfn_free_func* is `NULL`, all pointers specified in *svm_pointers* must be allocated using **clSVMAlloc** and the OpenCL implementation will free these SVM pointers. *pfn_free_func* must be a valid callback function if any SVM pointer to be freed is a shared system memory pointer i.e. not allocated using **clSVMAlloc**. If *pfn_free_func* is a valid callback function, the OpenCL implementation will call *pfn_free_func* to free all the SVM pointers specified in *svm_pointers*.
- *user_data* will be passed as the *user_data* argument when *pfn_free_func* is called. *user_data* can be `NULL`.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before **clEnqueueSVMFree** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueSVMFree** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to

an element of the *event_wait_list* array.

clEnqueueSVMFree returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_OPERATION` if the device associated with *command_queue* does not support SVM.
- `CL_INVALID_VALUE` if *num_svm_pointers* is 0 and *svm_pointers* is non-`NULL`, *or* if *svm_pointers* is `NULL` and *num_svm_pointers* is not 0.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a command to do a memcpy operation, call the function

```
// Provided by CL_VERSION_2_0
cl_int clEnqueueSVMMemcpy(
    cl_command_queue command_queue,
    cl_bool blocking_copy,
    void* dst_ptr,
    const void* src_ptr,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMMemcpy is missing before version 2.0.

- *command_queue* refers to the host command-queue in which the read / write command will be queued. If either *dst_ptr* or *src_ptr* is allocated using **clSVMAlloc** then the OpenCL context allocated against must match that of *command_queue*.
- *blocking_copy* indicates if the copy operation is *blocking* or *non-blocking*.
- If *blocking_copy* is `CL_TRUE` i.e. the copy command is blocking, **clEnqueueSVMMemcpy** does not return until the buffer data has been copied into memory pointed to by *dst_ptr*.
- *size* is the size in bytes of data being copied.
- *dst_ptr* is the pointer to a host or SVM memory allocation where data is copied to.
- *src_ptr* is the pointer to a host or SVM memory allocation where data is copied from.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must

be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this read / write command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If *blocking_copy* is `CL_FALSE` i.e. the copy command is non-blocking, **clEnqueueSVMMemcpy** queues a non-blocking copy command and returns. The contents of the buffer that *dst_ptr* points to cannot be used until the copy command has completed. The *event* argument returns an event object which can be used to query the execution status of the read command. When the copy command has completed, the contents of the buffer that *dst_ptr* points to can be used by the application.

If the memory allocation(s) containing *dst_ptr* and/or *src_ptr* are allocated using **clSVMAlloc** and either is not allocated from the same context from which *command_queue* was created the behavior is undefined.

clEnqueueSVMMemcpy returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support SVM
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_VALUE`
  - if *dst_ptr* is `NULL`
  - if *src_ptr* is `NULL`
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the copy operations is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error
- `CL_MEM_COPY_OVERLAP`
  - if the values specified for *dst_ptr*, *src_ptr* and *size* result in an overlapping copy

- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command to fill a region in memory with a pattern of a given pattern size, call the function

```
// Provided by CL_VERSION_2_0
cl_int clEnqueueSVMMemFill(
    cl_command_queue command_queue,
    void* svm_ptr,
    const void* pattern,
    size_t pattern_size,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMMemFill is missing before version 2.0.

- *command_queue* refers to the host command-queue in which the fill command will be queued. The OpenCL context associated with *command_queue* and SVM pointer referred to by *svm_ptr* must be the same.
- *svm_ptr* is a pointer to a memory region that will be filled with *pattern*. It must be aligned to *pattern_size* bytes. If *svm_ptr* is allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.
- *pattern* is a pointer to the data pattern of size *pattern_size* in bytes. *pattern* will be used to fill a region in *buffer* starting at *svm_ptr* and is *size* bytes in size. The data pattern must be a scalar or vector integer or floating-point data type supported by OpenCL as described in Shared Application Scalar Data Types and Supported Application Vector Data Types. For example, if region pointed to by *svm_ptr* is to be filled with a pattern of float4 values, then *pattern* will be a pointer to a cl_float4 value and *pattern_size* will be `sizeof(cl_float4)`. The maximum value of *pattern_size* is the size of the largest integer or floating-point vector data type supported by the OpenCL device. The memory associated with *pattern* can be reused or freed after the function returns.
- *size* is the size in bytes of region being filled starting with *svm_ptr* and must be a multiple of *pattern_size*.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and

command_queue must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

clEnqueueSVMMemFill returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support SVM
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_VALUE`
  - if *svm_ptr* is `NULL`
  - if *svm_ptr* is not aligned to *pattern_size* bytes
  - if *pattern* is `NULL`
  - if *pattern_size* is zero
  - if *pattern_size* is not a power of two
  - if *pattern_size* is greater than 128
  - if *size* is not a multiple of *pattern_size*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command that will allow the host to update a region of a SVM buffer, call the function

```
// Provided by CL_VERSION_2_0
cl_int clEnqueueSVMMap(
    cl_command_queue command_queue,
    cl_bool blocking_map,
    cl_map_flags flags,
    void* svm_ptr,
    size_t size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMMap is missing before version 2.0.

- *command_queue* must be a valid host command-queue.
- *blocking_map* indicates if the map operation is *blocking* or *non-blocking*.
- *map_flags* is a bit-field and is described in the Memory Map Flags table.
- *svm_ptr* and *size* are a pointer to a memory region and size in bytes that will be updated by the host. If *svm_ptr* is allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If *blocking_map* is `CL_TRUE`, **clEnqueueSVMMap** does not return until the application can access the contents of the SVM region specified by *svm_ptr* and *size* on the host.

If *blocking_map* is `CL_FALSE` i.e. map operation is non-blocking, the region specified by *svm_ptr* and *size* cannot be used until the map command has completed. The *event* argument returns an event object which can be used to query the execution status of the map command. When the map command is completed, the application can access the contents of the region specified by *svm_ptr* and *size*.

Note that since we are enqueuing a command with a SVM buffer, the region is already mapped in the host address space.

clEnqueueSVMMap returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it

returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support SVM
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_VALUE`
  - if *svm_ptr* is `NULL`
  - if *size* is zero
  - if values specified in *map_flags* are not valid
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST`
  - if the map operations is blocking and the execution status of any of the events in *event_wait_list* is a negative integer value indicating an error
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command to indicate that the host has completed updating the region given by *svm_ptr* and which was specified in a previous call to **clEnqueueSVMMap**, call the function

```
// Provided by CL_VERSION_2_0
cl_int clEnqueueSVMUnmap(
    cl_command_queue command_queue,
    void* svm_ptr,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMUnmap is missing before version 2.0.

- *command_queue* must be a valid host command-queue.
- *svm_ptr* is a pointer that was specified in a previous call to **clEnqueueSVMMap**. If *svm_ptr* is

allocated using **clSVMAlloc** then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.

- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before **clEnqueueSVMUnmap** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueSVMUnmap** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

clEnqueueSVMMap and **clEnqueueSVMUnmap** act as synchronization points for the region of the SVM buffer specified in these calls.

clEnqueueSVMUnmap returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support SVM
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_VALUE`
  - if *svm_ptr* is `NULL`
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

If a coarse-grained SVM buffer is currently mapped for writing, the application must ensure that the SVM buffer is unmapped before any enqueued kernels or

commands that read from or write to this SVM buffer or any of its associated `cl_mem` buffer objects begin execution; otherwise the behavior is undefined.

If a coarse-grained SVM buffer is currently mapped for reading, the application must ensure that the SVM buffer is unmapped before any enqueued kernels or commands that write to this memory object or any of its associated `cl_mem` buffer objects begin execution; otherwise the behavior is undefined.

A SVM buffer is considered as mapped if there are one or more active mappings for the SVM buffer irrespective of whether the mapped regions span the entire SVM buffer.

The above note does not apply to fine-grained SVM buffers (fine-grained buffers allocated using **clSVMAlloc** or fine-grained system allocations).

To enqueue a command to indicate which device a set of ranges of SVM allocations should be associated with, call the function

```
// Provided by CL_VERSION_2_1
cl_int clEnqueueSVMMigrateMem(
    cl_command_queue command_queue,
    cl_uint num_svm_pointers,
    const void** svm_pointers,
    const size_t* sizes,
    cl_mem_migration_flags flags,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSVMMigrateMem is missing before version 2.1.

- *command_queue* is a valid host command-queue. The specified set of allocation ranges will be migrated to the OpenCL device associated with *command_queue*.
- *num_svm_pointers* is the number of pointers in the specified *svm_pointers* array, and the number of sizes in the *sizes* array, if *sizes* is not `NULL`.
- *svm_pointers* is a pointer to an array of pointers. Each pointer in this array must be within an allocation produced by a call to **clSVMAlloc**.
- *sizes* is an array of sizes. The pair *svm_pointers*[i] and *sizes*[i] together define the starting address and number of bytes in a range to be migrated. *sizes* may be `NULL` indicating that every allocation containing any *svm_pointer*[i] is to be migrated. Also, if *sizes*[i] is zero, then the entire allocation containing *svm_pointer*[i] is migrated.
- *flags* is a bit-field that is used to specify migration options. The Memory Migration Flags describes the possible values for *flags*.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must

be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

- *event* returns an event object that identifies this command and can be used to query or queue a wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

Once the event returned by **clEnqueueSVMMigrateMem** has become `CL_COMPLETE`, the ranges specified by svm pointers and sizes have been successfully migrated to the device associated with command-queue.

The user is responsible for managing the event dependencies associated with this command in order to avoid overlapping access to SVM allocations. Improperly specified event dependencies passed to **clEnqueueSVMMigrateMem** could result in undefined results.

clEnqueueSVMMigrateMem returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_OPERATION`
  - if the device associated with *command_queue* does not support SVM
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_VALUE`
  - if *num_svm_pointers* is zero
  - if *svm_pointers* is `NULL`
  - if *flags* is not zero and is not a valid combination of the values described in the Memory migration Flags table
  - if *sizes*[i] is non-zero and the memory range described by *svm_pointers*[i] and *sizes*[i] is not contained within an SVM allocation returned by **clSVMAlloc**
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device

- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

5.6.2. Memory Consistency for SVM Allocations

To ensure memory consistency in SVM allocations, the program can rely on the guaranteed memory consistency at synchronization points. This consistency support already exists in OpenCL 1.x and can be used for coarse-grained SVM allocations or for fine-grained buffer SVM allocations; what SVM adds is the ability to share pointers between the host and all SVM devices.

In addition, sub-buffers can also be used to ensure that each device gets a consistent view of a SVM buffers memory when it is shared by multiple devices. For example, assume that two devices share a SVM pointer. The host can create a `cl_mem` buffer object using **clCreateBuffer** or **clCreateBufferWithProperties** with `CL_MEM_USE_HOST_PTR` and *host_ptr* set to the SVM pointer and then create two disjoint sub-buffers with starting virtual addresses *sb1_ptr* and *sb2_ptr*. These pointers (*sb1_ptr* and *sb2_ptr*) can be passed to kernels executing on the two devices. **clEnqueueMapBuffer** and **clEnqueueUnmapMemObject** and the existing access rules for memory objects ensure consistency for buffer regions (*sb1_ptr* and *sb2_ptr*) read and written by these kernels.

When the host and devices are able to use SVM atomic operations (i.e. `CL_DEVICE_SVM_ATOMICS` is set in `CL_DEVICE_SVM_CAPABILITIES`), these atomic operations can be used to provide memory consistency at a fine grain in a shared memory region. The effect of these operations is visible to the host and all devices with which that memory is shared.

5.7. Sampler Objects

A sampler object describes how to sample an image when the image is read in the kernel. The built-in functions to read from an image in a kernel take a sampler as an argument. The sampler arguments to the image read function can be sampler objects created using OpenCL functions and passed as argument values to the kernel or can be samplers declared inside a kernel. In this section we discuss how sampler objects are created using OpenCL functions.

5.7.1. Creating Sampler Objects

To create a sampler object, call the function

```
// Provided by CL_VERSION_2_0
cl_sampler clCreateSamplerWithProperties(
    cl_context context,
    const cl_sampler_properties* sampler_properties,
    cl_int* errcode_ret);
```

clCreateSamplerWithProperties is missing before version 2.0.

- *context* must be a valid OpenCL context.

- *sampler_properties* specifies a list of sampler property names and their corresponding values. Each sampler property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in the Sampler Properties table. If a supported property and its value is not specified in *sampler_properties*, its default value will be used. *sampler_properties* can be `NULL` in which case the default values for supported sampler properties will be used.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Table 42. List of supported sampler creation properties by **clCreateSamplerWithProperties**

Sampler Property	Property Value	Description
`CL_SAMPLER_NORMALIZED_COORDS`	`cl_bool`	A boolean value that specifies whether the image coordinates specified are normalized or not. The default value (i.e. the value used if this property is not specified in sampler_properties) is `CL_TRUE`.
`CL_SAMPLER_ADDRESSING_MODE`	`cl_addressing_mode`	Specifies how out-of-range image coordinates are handled when reading from an image. Valid values are: `CL_ADDRESS_NONE` - Behavior is undefined for out-of-range image coordinates. `CL_ADDRESS_CLAMP_TO_EDGE` - Out-of-range image coordinates are clamped to the edge of the image. `CL_ADDRESS_CLAMP` - Out-of-range image coordinates are assigned a border color value. `CL_ADDRESS_REPEAT` - Out-of-range image coordinates read from the image as if the image data were replicated in all dimensions. `CL_ADDRESS_MIRRORED_REPEAT` - Out-of-range image coordinates read from the image as if the image data were replicated in all dimensions, mirroring the image contents at the edge of each replication. The default is `CL_ADDRESS_CLAMP`.

Sampler Property	Property Value	Description
CL_SAMPLER_FILTER_MODE	cl_filter_mode	Specifies the type of filter that is applied when reading an image. Valid values are: CL_FILTER_NEAREST - Returns the image element nearest to the image coordinate. CL_FILTER_LINEAR - Returns a weighted average of the four image elements nearest to the image coordinate. The default value is CL_FILTER_NEAREST.
CL_SAMPLER_MIP_FILTER_MODE_KHR provided by the cl_khr_mipmap_image extension.	cl_filter_mode	Specifies the mipmap filter used when sampling from a mipmapped image. The available filter are: CL_FILTER_NEAREST - Use the nearest mipmap level to the image coordinate. CL_FILTER_LINEAR - Use a weighted average of the two mipmap levels nearest to the image coordinate. The default is CL_FILTER_NEAREST.
CL_SAMPLER_LOD_MIN_KHR provided by the cl_khr_mipmap_image extension.	cl_float	Specifies the minimum value to which the computed level of detail *lambda* is clamped when sampling from a mipmapped image. The default is 0.0f.
CL_SAMPLER_LOD_MAX_KHR provided by the cl_khr_mipmap_image extension.	cl_float	Specifies the maximum value to which the computed level of detail *lambda* is clamped when sampling from a mipmapped image. The default is MAXFLOAT.

When the cl_khr_mipmap_image extension is supported, the sampler properties CL_SAMPLER_MIP_FILTER_MODE_KHR, CL_SAMPLER_LOD_MIN_KHR and CL_SAMPLER_LOD_MAX_KHR cannot be specified with any samplers initialized in the OpenCL program source. Only the default values for these properties will be used. To create a sampler with specific values for these properties, a sampler object must be created with **clCreateSamplerWithProperties** and passed as an argument to a kernel.

clCreateSamplerWithProperties returns a valid non-zero sampler object and *errcode_ret* is set to CL_SUCCESS if the sampler object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if the property name in *sampler_properties* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once.
- CL_INVALID_OPERATION if images are not supported by any device associated with *context* (i.e. CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create a sampler object, call the function

```
// Provided by CL_VERSION_1_0
cl_sampler clCreateSampler(
    cl_context context,
    cl_bool normalized_coords,
    cl_addressing_mode addressing_mode,
    cl_filter_mode filter_mode,
    cl_int* errcode_ret);
```

clCreateSampler is deprecated by version 2.0.

- *context* must be a valid OpenCL context.
- *normalized_coords* has the same interpretation as CL_SAMPLER_NORMALIZED_COORDS in the sampler creation properties table.
- *addressing_mode* has the same interpretation as CL_SAMPLER_ADDRESSING_MODE in the sampler creation properties table.
- *filter_mode* has the same interpretation as CL_SAMPLER_FILTER_MODE in the sampler creation properties table.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

clCreateSampler returns a valid non-zero sampler object and *errcode_ret* is set to CL_SUCCESS if the sampler object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *addressing_mode*, *filter_mode*, *normalized_coords* or a combination of these arguments are not valid.
- CL_INVALID_OPERATION if images are not supported by any device associated with *context* (i.e. CL_DEVICE_IMAGE_SUPPORT specified in the Device Queries table is CL_FALSE).
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL

implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To retain a sampler object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainSampler(
    cl_sampler sampler);
```

- *sampler* specifies the sampler to be released.

The *sampler* reference count is incremented. **clCreateSamplerWithProperties** and **clCreateSampler** perform an implicit retain.

clRetainSampler returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SAMPLER` if *sampler* is not a valid sampler object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release a sampler object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseSampler(
    cl_sampler sampler);
```

- *sampler* specifies the sampler to be released.

The *sampler* reference count is decremented. The sampler object is deleted after the reference count becomes zero and commands queued for execution on a command-queue(s) that use *sampler* have finished.

clReleaseSampler returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SAMPLER` if *sampler* is not a valid sampler object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

Using this function to release a reference that was not obtained by creating the object or by calling

clRetainSampler causes undefined behavior.

5.7.2. Sampler Object Queries

To return information about a sampler object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetSamplerInfo(
    cl_sampler sampler,
    cl_sampler_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *sampler* specifies the sampler being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetSamplerInfo** is described in the Sampler Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Sampler Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 43. List of supported param_names by **clGetSamplerInfo**

Sampler Info	Return Type	Description
`CL_SAMPLER_REFERENCE_COUNT` [12]	`cl_uint`	Return the *sampler* reference count.
`CL_SAMPLER_CONTEXT`	`cl_context`	Return the context specified when the sampler is created.
`CL_SAMPLER_NORMALIZED_COORDS`	`cl_bool`	Return the normalized coords value associated with *sampler*.
`CL_SAMPLER_ADDRESSING_MODE`	`cl_addressing_mode`	Return the addressing mode value associated with *sampler*.
`CL_SAMPLER_FILTER_MODE`	`cl_filter_mode`	Return the filter mode value associated with *sampler*.

Sampler Info	Return Type	Description
`CL_SAMPLER_PROPERTIES` missing before version 3.0.	`cl_sampler_properties`[]	Return the properties argument specified in **clCreateSamplerWithProperties**. If the *properties* argument specified in **clCreateSamplerWithProperties** used to create *sampler* was not `NULL`, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If *sampler* was created using **clCreateSampler**, or if the *properties* argument specified in **clCreateSamplerWithProperties** was `NULL`, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.

clGetSamplerInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SAMPLER` if *sampler* is a not a valid sampler object.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Sampler Object Queries table and *param_value* is not `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8. Program Objects

An OpenCL program consists of a set of kernels that are identified as functions declared with the `__kernel` qualifier in the program source. OpenCL programs may also contain auxiliary functions and constant data that can be used by kernel functions. The program executable can be generated *online* or *offline* by the OpenCL compiler for the appropriate target device(s).

A program object encapsulates the following information:

- An associated context.
- A program source or binary.
- The latest successfully built program executable, library or compiled binary, the list of devices for which the program executable, library or compiled binary is built, the build options used and a build log.
- The number of kernel objects currently attached.

5.8.1. Creating Program Objects

To create a program object for a context and load source code into that object, call the function

```
// Provided by CL_VERSION_1_0
cl_program clCreateProgramWithSource(
    cl_context context,
    cl_uint count,
    const char** strings,
    const size_t* lengths,
    cl_int* errcode_ret);
```

- *context* must be a valid OpenCL context.
- *strings* is an array of *count* pointers to optionally null-terminated character strings that make up the source code.
- *lengths* argument is an array with the number of chars in each string (the string length). If an element in *lengths* is zero, its accompanying string is null-terminated. If *lengths* is `NULL`, all strings in the *strings* argument are considered null-terminated. Any length value passed in that is greater than zero excludes the null terminator in its count.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The devices associated with the program object are the devices associated with *context*.

The source code specified by *strings* are loaded into the program object. The source languages supported for *strings* are defined by device queries such as `CL_DEVICE_OPENCL_C_ALL_VERSIONS`.

clCreateProgramWithSource returns a valid non-zero program object and *errcode_ret* is set to `CL_SUCCESS` if the program object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if *count* is zero or if *strings* or any entry in *strings* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create a program object for a context and load code in an intermediate language into that object, call the function

```
// Provided by CL_VERSION_2_1
cl_program clCreateProgramWithIL(
    cl_context context,
    const void* il,
    size_t length,
    cl_int* errcode_ret);
```

clCreateProgramWithIL is missing before version 2.1.

or the equivalent

```
// Provided by cl_khr_il_program
cl_program clCreateProgramWithILKHR(
    cl_context context,
    const void* il,
    size_t length,
    cl_int* errcode_ret);
```

clCreateProgramWithILKHR is provided by the `cl_khr_il_program` extension.

- *context* must be a valid OpenCL context.
- *il* is a pointer to a block of memory containing SPIR-V or an implementation-defined intermediate language.
- *length* is the length of the block pointed to by *il*.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The intermediate language pointed to by *il* and with length in bytes *length* will be loaded into the program object. The devices associated with the program object are the devices associated with *context*.

clCreateProgramWithIL returns a valid non-zero program object and *errcode_ret* is set to `CL_SUCCESS` if the program object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_OPERATION` if no devices in *context* support intermediate language programs.
- `CL_INVALID_VALUE` if *il* is `NULL` or if *length* is zero.
- `CL_INVALID_VALUE` if the *length*-byte block of memory pointed to by *il* does not contain well-formed intermediate language input that can be consumed by the OpenCL runtime.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL

implementation on the host.

To create a program object for a context and load binary bits into that object, call the function

```
// Provided by CL_VERSION_1_0
cl_program clCreateProgramWithBinary(
    cl_context context,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const size_t* lengths,
    const unsigned char** binaries,
    cl_int* binary_status,
    cl_int* errcode_ret);
```

- *context* must be a valid OpenCL context.
- *device_list* is a pointer to a list of devices that are in *context*. *device_list* must be a non-`NULL` value. The binaries are loaded for devices specified in this list.
- *num_devices* is the number of devices listed in *device_list*.
- *lengths* is an array of the size in bytes of the program binaries to be loaded for devices specified by *device_list*.
- *binaries* is an array of pointers to program binaries to be loaded for devices specified by *device_list*. For each device given by *device_list*[i], the pointer to the program binary for that device is given by *binaries*[i] and the length of this corresponding binary is given by *lengths*[i]. *lengths*[i] cannot be zero and *binaries*[i] cannot be a `NULL` pointer.
- *binary_status* returns whether the program binary for each device specified in *device_list* was loaded successfully or not. It is an array of *num_devices* entries and returns `CL_SUCCESS` in *binary_status*[i] if binary was successfully loaded for device specified by *device_list*[i]; otherwise returns `CL_INVALID_VALUE` if *lengths*[i] is zero or if *binaries*[i] is a `NULL` value or `CL_INVALID_BINARY` in *binary_status*[i] if program binary is not a valid binary for the specified device. If *binary_status* is `NULL`, it is ignored.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The devices associated with the program object will be the list of devices specified by *device_list*. The list of devices specified by *device_list* must be devices associated with *context*.

The program binaries specified by *binaries* will be loaded into the program object. They contain bits that describe one of the following:

- a program executable to be run on the device(s) associated with *context*,
- a compiled program for device(s) associated with *context*, or
- a library of compiled programs for device(s) associated with *context*.

The program binary can consist of either or both:

- Device-specific code and/or,

- Implementation-specific intermediate representation (IR) which will be converted to the device-specific code.

OpenCL allows applications to create a program object using the program source or binary and build appropriate program executables. This can be very useful as it allows applications to load program source and then compile and link to generate a program executable online on its first instance for appropriate OpenCL devices in the system. These executables can now be queried and cached by the application. The cached executables can be read and loaded by the application, which can help significantly reduce the application initialization time.

If the `cl_khr_spir` extension is supported, **clCreateProgramWithBinary** can be used to load a SPIR binary. Once a program object has been created from a SPIR binary, **clBuildProgram** can be called to build a program executable or **clCompileProgram** can be called to compile the SPIR binary.

clCreateProgramWithBinary returns a valid non-zero program object and *errcode_ret* is set to `CL_SUCCESS` if the program object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_VALUE` if *device_list* is `NULL` or *num_devices* is zero.
- `CL_INVALID_DEVICE` if any device in *device_list* is not in the list of devices associated with *context*.
- `CL_INVALID_VALUE` if *lengths* or *binaries* is `NULL` or if any entry in *lengths*[i] is zero or *binaries*[i] is `NULL`.
- `CL_INVALID_BINARY` if an invalid program binary was encountered for any device. *binary_status* will return specific status for each device.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create a program object for a context and loads the information related to the built-in kernels into that object, call the function

```
// Provided by CL_VERSION_1_2
cl_program clCreateProgramWithBuiltInKernels(
    cl_context context,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const char* kernel_names,
    cl_int* errcode_ret);
```

clCreateProgramWithBuiltInKernels is missing before version 1.2.

- *context* must be a valid OpenCL context.
- *num_devices* is the number of devices listed in *device_list*.

- *device_list* is a pointer to a list of devices that are in *context*. *device_list* must be a non-NULL value. The built-in kernels are loaded for devices specified in this list.
- *kernel_names* is a semi-colon separated list of built-in kernel names.

The devices associated with the program object will be the list of devices specified by *device_list*. The list of devices specified by *device_list* must be devices associated with *context*.

clCreateProgramWithBuiltInKernels returns a valid non-zero program object and *errcode_ret* is set to CL_SUCCESS if the program object is created successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_VALUE if *device_list* is NULL or *num_devices* is zero.
- CL_INVALID_VALUE if *kernel_names* is NULL or *kernel_names* contains a kernel name that is not supported by any of the devices in *device_list*.
- CL_INVALID_DEVICE if any device in *device_list* is not in the list of devices associated with *context*.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.2. Retaining and Releasing Program Objects

To retain a program object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainProgram(
    cl_program program);
```

- *program* is the program object to be retained.

The *program* reference count is incremented. All APIs that create a program do an implicit retain.

clRetainProgram returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM if *program* is not a valid program object.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release a program object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseProgram(
    cl_program program);
```

- *program* is the program object to be released.

The *program* reference count is decremented. The program object is deleted after all kernel objects associated with *program* have been deleted and the *program* reference count becomes zero.

clReleaseProgram returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

Using this function to release a reference that was not obtained by creating the object or by calling **clRetainProgram** causes undefined behavior.

To register a callback function with a program object that is called when the program object is destroyed, call the function

```
// Provided by CL_VERSION_2_2
cl_int clSetProgramReleaseCallback(
    cl_program program,
    void (CL_CALLBACK* pfn_notify)(cl_program program, void* user_data),
    void* user_data);
```

clSetProgramReleaseCallback is missing before version 2.2 and deprecated by version 3.0.

- *program* specifies the memory object to register the callback to.
- *pfn_notify* is the callback function to register. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:
  - *program* is the program being deleted. When the callback function is called by the implementation, this program object is not longer valid. *program* is only provided for reference purposes.
  - *user_data* is a pointer to user supplied data.
- *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be `NULL`.

Each call to **clSetProgramReleaseCallback** registers the specified callback function on a callback

stack associated with *program*. The registered callback functions are called in the reverse order in which they were registered. The registered callback functions are called after destructors (if any) for program scope global variables (if any) are called and before the program object is deleted. This provides a mechanism for an application to be notified when destructors for program scope global variables are complete.

clSetProgramReleaseCallback may unconditionally return an error if no devices in the context associated with *program* support destructors for program scope global variables. Support for constructors and destructors for program scope global variables is required only for OpenCL 2.2 devices.

clSetProgramReleaseCallback returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_INVALID_OPERATION` if no devices in the context associated with *program* support destructors for program scope global variables.
- `CL_INVALID_VALUE` if *pfn_notify* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.3. Setting SPIR-V Specialization Constants

Specialization constants are missing before version 2.2.

To set the value of a specialization constant, call the function

```
// Provided by CL_VERSION_2_2
cl_int clSetProgramSpecializationConstant(
    cl_program program,
    cl_uint spec_id,
    size_t spec_size,
    const void* spec_value);
```

clSetProgramSpecializationConstant is missing before version 2.2.

- *program* must be a valid OpenCL program created from an intermediate language (e.g. SPIR-V).
- *spec_id* identifies the specialization constant whose value will be set.
- *spec_size* specifies the size in bytes of the data pointed to by *spec_value*. This should be 1 for boolean constants. For all other constant types this should match the size of the specialization constant in the module.
- *spec_value* is a pointer to the memory location that contains the value of the specialization constant. The data pointed to by *spec_value* are copied and can be safely reused by the

application after **clSetProgramSpecializationConstant** returns. This specialization value will be used by subsequent calls to **clBuildProgram** until another call to **clSetProgramSpecializationConstant** changes it. If a specialization constant is a boolean constant, *spec_value* should be a pointer to a `cl_uchar` value. A value of zero will set the specialization constant to false; any other value will set it to true.

Calling this function multiple times for the same specialization constant shall cause the last provided value to override any previously specified value. The values are used by a subsequent **clBuildProgram** call for the *program*.

Application is not required to provide values for every specialization constant contained in the module. If the value is not set by this API call, default values will be used during the build.

clSetProgramSpecializationConstant returns `CL_SUCCESS` if the function is executed successfully.

Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object created from an intermediate language (e.g. SPIR-V), or if the intermediate language does not support specialization constants.
- `CL_INVALID_OPERATION` if no devices associated with *program* support intermediate language programs.
- `CL_COMPILER_NOT_AVAILABLE` if *program* is created with **clCreateProgramWithIL** and a compiler is not available, i.e. `CL_DEVICE_COMPILER_AVAILABLE` specified in the Device Queries table is set to `CL_FALSE`.
- `CL_INVALID_SPEC_ID` if *spec_id* is not a valid specialization constant identifier.
- `CL_INVALID_VALUE` if *spec_size* does not match the size of the specialization constant in the module, or if *spec_value* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.4. Building Program Executables

To build (compile & link) a program executable, call the function

```
// Provided by CL_VERSION_1_0
cl_int clBuildProgram(
    cl_program program,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const char* options,
    void (CL_CALLBACK* pfn_notify)(cl_program program, void* user_data),
    void* user_data);
```

- *program* is the program object.

- *device_list* is a pointer to a list of devices associated with *program*. If *device_list* is a `NULL` value, the program executable is built for all devices associated with *program* for which a source or binary has been loaded. If *device_list* is a non-`NULL` value, the program executable is built for devices specified in this list for which a source or binary has been loaded.
- *num_devices* is the number of devices listed in *device_list*.
- *options* is a pointer to a null-terminated string of characters that describes the build options to be used for building the program executable. The list of supported options is described in Compiler Options. If the program was created using **clCreateProgramWithBinary** and *options* is a `NULL` pointer, the program will be built as if *options* were the same as when the program binary was originally built. If the program was created using **clCreateProgramWithBinary** and *options* string contains anything other than the same options in the same order (whitespace ignored) as when the program binary was originally built, then the behavior is implementation-defined. Otherwise, if *options* is a `NULL` pointer then it will have the same result as the empty string.
- *pfn_notify* is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not `NULL`, **clBuildProgram** does not need to wait for the build to complete and can return immediately once the build operation can begin. Any state changes of the program object that result from calling **clBuildProgram** (e.g. build status or log) will be observable from this callback function. The build operation can begin if the context, program whose sources are being compiled and linked, list of devices and build options specified are all valid and appropriate host and device resources needed to perform the build are available. If *pfn_notify* is `NULL`, **clBuildProgram** does not return until the build has completed. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe.
- *user_data* will be passed as an argument when *pfn_notify* is called. *user_data* can be `NULL`.

The program executable is built from the program source or binary for all the devices, or a specific device(s) in the OpenCL context associated with *program*. OpenCL allows program executables to be built using the source or the binary. **clBuildProgram** must be called for *program* created using **clCreateProgramWithSource**, **clCreateProgramWithIL** or **clCreateProgramWithBinary** to build the program executable for one or more devices associated with *program*. If *program* is created with **clCreateProgramWithBinary**, then the program binary must be an executable binary (not a compiled binary or library).

The executable binary can be queried using **clGetProgramInfo**(*program*, `CL_PROGRAM_BINARIES`, ...) and can be specified to **clCreateProgramWithBinary** to create a new program object.

clBuildProgram returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_INVALID_VALUE` if *device_list* is `NULL` and *num_devices* is greater than zero, or if *device_list* is not `NULL` and *num_devices* is zero.
- `CL_INVALID_VALUE` if *pfn_notify* is `NULL` but *user_data* is not `NULL`.

- `CL_INVALID_DEVICE` if any device in *device_list* is not in the list of devices associated with *program*.
- `CL_INVALID_BINARY` if *program* is created with **clCreateProgramWithBinary** and devices listed in *device_list* do not have a valid program binary loaded.
- `CL_INVALID_BUILD_OPTIONS` if the build options specified by *options* are invalid.
- `CL_COMPILER_NOT_AVAILABLE` if *program* is created with **clCreateProgramWithILKHR**, **clCreateProgramWithSource** or **clCreateProgramWithIL** and a compiler is not available, i.e. `CL_DEVICE_COMPILER_AVAILABLE` specified in the Device Queries table is set to `CL_FALSE`.
- `CL_BUILD_PROGRAM_FAILURE` if there is a failure to build the program executable. This error will be returned if **clBuildProgram** does not return until the build has completed.
- `CL_INVALID_OPERATION` if the build of a program executable for any of the devices listed in *device_list* by a previous call to **clBuildProgram** for *program* has not completed.
- `CL_INVALID_OPERATION` if there are kernel objects attached to *program*.
- `CL_INVALID_OPERATION` if *program* was not created with **clCreateProgramWithSource**, **clCreateProgramWithIL** or **clCreateProgramWithBinary**.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.5. Separate Compilation and Linking of Programs

Separate compilation and linking are missing before version 1.2.

OpenCL programs are compiled and linked to support the following:

- Separate compilation and link stages. Program sources can be compiled to generate a compiled binary object and linked in a separate stage with other compiled program objects to the program executable.
- Embedded headers. In OpenCL 1.0 and 1.1, the I build option could be used to specify the list of directories to be searched for headers files that are included by a program source(s). OpenCL 1.2 extends this by allowing the header sources to come from program objects instead of just header files.
- Libraries. The linker can be used to link compiled objects and libraries into a program executable or to create a library of compiled binaries.

To compile a program's source for all the devices or a specific device(s) in the OpenCL context associated with the program, call the function

```
// Provided by CL_VERSION_1_2
cl_int clCompileProgram(
    cl_program program,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const char* options,
    cl_uint num_input_headers,
    const cl_program* input_headers,
    const char** header_include_names,
    void (CL_CALLBACK* pfn_notify)(cl_program program, void* user_data),
    void* user_data);
```

clCompileProgram is missing before version 1.2.

- *program* is the program object that is the compilation target.
- *device_list* is a pointer to a list of devices associated with *program*. If *device_list* is a `NULL` value, the compile is performed for all devices associated with *program*. If *device_list* is a non-`NULL` value, the compile is performed for devices specified in this list.
- *num_devices* is the number of devices listed in *device_list*.
- *options* is a pointer to a null-terminated string of characters that describes the compilation options to be used for building the program executable. If *options* is a `NULL` pointer then it will have the same result as the empty string. Certain options are ignored when *program* is created with IL. The list of supported options is as described in Compiler Options.
- *num_input_headers* specifies the number of programs that describe headers in the array referenced by *input_headers*.
- *input_headers* is an array of program embedded headers created with **clCreateProgramWithSource**.
- *header_include_names* is an array that has a one to one correspondence with *input_headers*. Each entry in *header_include_names* specifies the include name used by source in *program* that comes from an embedded header. The corresponding entry in *input_headers* identifies the program object which contains the header source to be used. The embedded headers are first searched before the headers in the list of directories specified by the `-I` compile option (as described in Preprocessor options). If multiple entries in *header_include_names* refer to the same header name, the first one encountered will be used.
- *pfn_notify* is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully). If *pfn_notify* is not `NULL`, **clCompileProgram** does not need to wait for the compiler to complete and can return immediately once the compilation can begin. Any state changes of the program object that result from calling **clCompileProgram** (e.g. compile status or log) will be observable from this callback function. The compilation can begin if the context, program whose sources are being compiled, list of devices, input headers, programs that describe input headers and compiler options specified are all valid and appropriate host and device resources needed to perform the compile are available. If *pfn_notify* is `NULL`, **clCompileProgram** does not return until the compiler has completed. This

callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe.

- *user_data* will be passed as an argument when *pfn_notify* is called. *user_data* can be `NULL`.

The pre-processor runs before the program sources are compiled. The compiled binary is built for all devices associated with *program* or the list of devices specified. The compiled binary can be queried using **clGetProgramInfo**(*program*, `CL_PROGRAM_BINARIES`, ...) and can be passed to **clCreateProgramWithBinary** to create a new program object.

If *program* was created using **clCreateProgramWithIL**, then *num_input_headers*, *input_headers*, and *header_include_names* are ignored.

For example, consider the following program source:

```
#include <foo.h>
#include <mydir/myinc.h>
__kernel void
image_filter (int n, int m,
              __constant float *filter_weights,
              __read_only image2d_t src_image,
              __write_only image2d_t dst_image)
{
...
}
```

This kernel includes two headers foo.h and mydir/myinc.h. The following describes how these headers can be passed as embedded headers in program objects:

```
cl_program foo_pg = clCreateProgramWithSource(context,
    1, &foo_header_src, NULL, &err);
cl_program myinc_pg = clCreateProgramWithSource(context,
    1, &myinc_header_src, NULL, &err);

// lets assume the program source described above is given
// by program_A and is loaded via clCreateProgramWithSource
cl_program input_headers[2] = { foo_pg, myinc_pg };
char * input_header_names[2] = { foo.h, mydir/myinc.h };
clCompileProgram(program_A,
                 0, NULL, // num_devices & device_list
                 NULL,    // compile_options
                 2,       // num_input_headers
                 input_headers,
                 input_header_names,
                 NULL, NULL); // pfn_notify & user_data
```

clCompileProgram returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_INVALID_CONTEXT` if the context associated with *program* and programs in *input_headers* are not the same.
- `CL_INVALID_VALUE` if *device_list* is `NULL` and *num_devices* is greater than zero, or if *device_list* is not `NULL` and *num_devices* is zero.
- `CL_INVALID_VALUE` if *num_input_headers* is zero and *header_include_names* or *input_headers* are not `NULL` or if *num_input_headers* is not zero and *header_include_names* or *input_headers* are `NULL`.
- `CL_INVALID_VALUE` if *pfn_notify* is `NULL` but *user_data* is not `NULL`.
- `CL_INVALID_DEVICE` if device in *device_list* is not in the list of devices associated with *program*.
- `CL_INVALID_COMPILER_OPTIONS` if the compiler options specified by *options* are invalid.
- `CL_INVALID_OPERATION` if the compilation or build of a program executable for any of the devices listed in *device_list* by a previous call to **clCompileProgram** or **clBuildProgram** for *program* has not completed.
- `CL_COMPILER_NOT_AVAILABLE` if a compiler is not available, i.e. `CL_DEVICE_COMPILER_AVAILABLE` specified in the Device Queries table is set to `CL_FALSE`.
- `CL_COMPILE_PROGRAM_FAILURE` if there is a failure to compile the program source. This error will be returned if **clCompileProgram** does not return until the compile has completed.
- `CL_INVALID_OPERATION` if there are kernel objects attached to *program*.
- `CL_INVALID_OPERATION` if *program* has no source or IL available, i.e. it has not been created with one of
  - **clCreateProgramWithIL** or **clCreateProgramWithILKHR**
  - **clCreateProgramWithBinary** where `-x spir` is present in *options*, if the `cl_khr_spir` extension is supported.
  - **clCreateProgramWithSource**
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To link a set of compiled program objects and libraries for all the devices or a specific device(s) in the OpenCL context and create a library or executable, call the function

```
// Provided by CL_VERSION_1_2
cl_program clLinkProgram(
    cl_context context,
    cl_uint num_devices,
    const cl_device_id* device_list,
    const char* options,
    cl_uint num_input_programs,
    const cl_program* input_programs,
    void (CL_CALLBACK* pfn_notify)(cl_program program, void* user_data),
    void* user_data,
    cl_int* errcode_ret);
```

clLinkProgram is missing before version 1.2.

- *context* must be a valid OpenCL context.
- *device_list* is a pointer to a list of devices that are in *context*. If *device_list* is a `NULL` value, the link is performed for all devices associated with *context* for which a compiled object is available. If *device_list* is a non-`NULL` value, the link is performed for devices specified in this list for which a compiled object is available.
- *num_devices* is the number of devices listed in *device_list*.
- *options* is a pointer to a null-terminated string of characters that describes the link options to be used for building the program executable. The list of supported options is as described in Linker Options. If the program was created using **clCreateProgramWithBinary** and *options* is a `NULL` pointer, the program will be linked as if *options* were the same as when the program binary was originally built. If the program was created using **clCreateProgramWithBinary** and *options* string contains anything other than the same options in the same order (whitespace ignored) as when the program binary was originally built, then the behavior is implementation-defined. Otherwise, if *options* is a `NULL` pointer then it will have the same result as the empty string.
- *num_input_programs* specifies the number of programs in array referenced by *input_programs*.
- *input_programs* is an array of program objects that are compiled binaries or libraries that are to be linked to create the program executable. For each device in *device_list* or if *device_list* is `NULL` the list of devices associated with context, the following cases occur:
    - All programs specified by *input_programs* contain a compiled binary or library for the device. In this case, a link is performed to generate a program executable for this device.
    - None of the programs contain a compiled binary or library for that device. In this case, no link is performed and there will be no program executable generated for this device.
    - All other cases will return a `CL_INVALID_OPERATION` error.
- *pfn_notify* is a function pointer to a notification routine. The notification routine is a callback function that an application can register and which will be called when the program executable has been built (successfully or unsuccessfully).
- *user_data* will be passed as an argument when *pfn_notify* is called. *user_data* can be `NULL`.

If *pfn_notify* is not `NULL`, **clLinkProgram** does not need to wait for the linker to complete, and can

return immediately once the linking operation can begin. Once the linker has completed, the *pfn_notify* callback function is called which returns the program object returned by **clLinkProgram**. Any state changes of the program object that result from calling **clLinkProgram** (e.g. link status or log) will be observable from this callback function. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe.

If *pfn_notify* is `NULL`, **clLinkProgram** does not return until the linker has completed.

clLinkProgram creates a new program object which contains the library or executable. The library or executable binary can be queried using **clGetProgramInfo**(*program*, `CL_PROGRAM_BINARIES`, ...) and can be specified to **clCreateProgramWithBinary** to create a new program object.

The devices associated with the returned program object will be the list of devices specified by *device_list* or if *device_list* is `NULL` it will be the list of devices associated with *context*.

The linking operation can begin if the context, list of devices, input programs and linker options specified are all valid and appropriate host and device resources needed to perform the link are available. If the linking operation can begin, **clLinkProgram** returns a valid non-zero program object.

If *pfn_notify* is `NULL`, *errcode_ret* will be set to `CL_SUCCESS` if the link operation was successful and `CL_LINK_PROGRAM_FAILURE` if there is a failure to link the compiled binaries and/or libraries.

If *pfn_notify* is not `NULL`, **clLinkProgram** does not have to wait until the linker to complete and can return `CL_SUCCESS` in *errcode_ret* if the linking operation can begin. The *pfn_notify* callback function will return a `CL_SUCCESS` or `CL_LINK_PROGRAM_FAILURE` if the linking operation was successful or not.

Otherwise **clLinkProgram** returns a `NULL` program object with an appropriate error in *errcode_ret*. The application should query the linker status of this program object to check if the link was successful or not. The list of errors that can be returned are:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_INVALID_CONTEXT` if the context associated with programs in *input_programs* is not the same as *context*.
- `CL_INVALID_VALUE` if *device_list* is `NULL` and *num_devices* is greater than zero, or if *device_list* is not `NULL` and *num_devices* is zero.
- `CL_INVALID_VALUE` if *num_input_programs* is zero and *input_programs* is `NULL` or if *num_input_programs* is zero and *input_programs* is not `NULL` or if *num_input_programs* is not zero and *input_programs* is `NULL`.
- `CL_INVALID_PROGRAM` if programs specified in *input_programs* are not valid program objects.
- `CL_INVALID_VALUE` if *pfn_notify* is `NULL` but *user_data* is not `NULL`.
- `CL_INVALID_DEVICE` if any device in *device_list* is not in the list of devices associated with *context*.
- `CL_INVALID_LINKER_OPTIONS` if the linker options specified by *options* are invalid.
- `CL_INVALID_OPERATION` if the compilation or build of a program executable for any of the devices listed in *device_list* by a previous call to **clCompileProgram** or **clBuildProgram** for *program*

has not completed.

- `CL_INVALID_OPERATION` if the rules for devices containing compiled binaries or libraries as described in *input_programs* argument above are not followed.
- `CL_LINKER_NOT_AVAILABLE` if a linker is not available, i.e. `CL_DEVICE_LINKER_AVAILABLE` specified in the Device Queries table is set to `CL_FALSE`.
- `CL_LINK_PROGRAM_FAILURE` if there is a failure to link the compiled binaries and/or libraries.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.8.6. Compiler Options

The compiler options are categorized as pre-processor options, options for math intrinsics, options that control optimization and miscellaneous options. This specification defines a standard set of options that must be supported by the compiler when building program executables online or offline from OpenCL C/C++ or, where relevant, from an IL. These may be extended by a set of vendor- or platform-specific options.

5.8.6.1. Preprocessor Options

These options control the OpenCL C/C++ preprocessor which is run on each program source before actual compilation. These options are ignored for programs created with IL.

`-D name`

Predefine *name* as a macro, with definition 1.

`-D name=definition`

The contents of *definition* are tokenized and processed as if they appeared during translation phase three in a `#define` directive. In particular, the definition will be truncated by embedded newline characters.

`-D` options are processed in the order they are given in the *options* argument to **clBuildProgram** or **clCompileProgram**. Note that a space is required between the `-D` option and the symbol it defines, otherwise behavior is implementation-defined.

`-I dir`

Add the directory *dir* to the list of directories to be searched for header files. *dir* can optionally be enclosed in double quotes.

This option is not portable due to its dependency on host file system and host operating system. It is supported for backwards compatibility with previous OpenCL versions. Developers are encouraged to create and use explicit header objects by means of **clCompileProgram** followed by **clLinkProgram**.

5.8.6.2. Math Intrinsics Options

These options control compiler behavior regarding floating-point arithmetic. These options trade off between speed and correctness.

`-cl-single-precision-constant`

This option forces implicit conversions of double-precision floating-point literals to single precision. This option is ignored for programs created with IL.

`-cl-denorms-are-zero`

This option controls how single precision and double precision denormalized numbers are handled. If specified as a build option, the single precision denormalized numbers may be flushed to zero; double precision denormalized numbers may also be flushed to zero if the optional extension for double precision is supported. This is intended to be a performance hint and the OpenCL compiler can choose not to flush denorms to zero if the device supports single precision (or double precision) denormalized numbers.

This option is ignored for single precision numbers if the device does not support single precision denormalized numbers i.e. `CL_FP_DENORM` bit is not set in `CL_DEVICE_SINGLE_FP_CONFIG`.

This option is ignored for double precision numbers if the device does not support double precision or if it does support double precision but not double precision denormalized numbers i.e. `CL_FP_DENORM` bit is not set in `CL_DEVICE_DOUBLE_FP_CONFIG`.

This flag only applies for scalar and vector single precision floating-point variables and computations on these floating-point variables inside a program. It does not apply to reading from or writing to image objects.

`-cl-fp32-correctly-rounded-divide-sqrt`

The `-cl-fp32-correctly-rounded-divide-sqrt` build option to **clBuildProgram** or **clCompileProgram** allows an application to specify that single precision floating-point divide (x/y and 1/x) and sqrt used in the program source are correctly rounded. If this build option is not specified, the minimum numerical accuracy of single precision floating-point divide and sqrt are as defined in the OpenCL C or OpenCL SPIR-V Environment specifications.

This build option can only be specified if the `CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT` is set in `CL_DEVICE_SINGLE_FP_CONFIG` (as defined in the Device Queries table) for devices that the program is being build. **clBuildProgram** or **clCompileProgram** will fail to compile the program for a device if the `-cl-fp32-correctly-rounded-divide-sqrt` option is specified and `CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT` is not set for the device.

Note: This option is missing before version 1.2.

5.8.6.3. Optimization Options

These options control various sorts of optimizations. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

`-cl-opt-disable`

This option disables all optimizations. The default is optimizations are enabled.

`-cl-strict-aliasing`

This option allows the compiler to assume the strictest aliasing rules.

Note: This option is deprecated by version 1.1.

`-cl-uniform-work-group-size`

This requires that the global work-size be a multiple of the work-group size specified to **clEnqueueNDRangeKernel**. Allow optimizations that are made possible by this restriction.

Note: This option is missing before version 2.0.

`-cl-no-subgroup-ifp`

This indicates that kernels in this program do not require sub-groups to make independent forward progress. Allows optimizations that are made possible by this restriction. This option has no effect for devices that do not support independent forward progress for sub-groups.

Note: This option is missing before version 2.1.

The following options control compiler behavior regarding floating-point arithmetic. These options trade off between performance and correctness and must be specifically enabled. These options are not turned on by default since it can result in incorrect output for programs which depend on an exact implementation of IEEE 754 rules/specifications for math functions.

`-cl-mad-enable`

Allow `a * b + c` to be replaced by a **mad** instruction. The **mad** instruction may compute `a * b + c` with reduced accuracy in the embedded profile. See the OpenCL C or OpenCL SPIR-V Environment specification for accuracy details. On some hardware the **mad** instruction may provide better performance than the expanded computation.

`-cl-no-signed-zeros`

Allow optimizations for floating-point arithmetic that ignore the signedness of zero. IEEE 754 arithmetic specifies the distinct behavior of `+0.0` and `-0.0` values, which then prohibits simplification of expressions such as `x + 0.0` or `0.0 * x` (even with `-cl-finite-math-only`). This option implies that the sign of a zero result is not significant.

`-cl-unsafe-math-optimizations`

Allow optimizations for floating-point arithmetic that (a) assume that arguments and results are valid, (b) may violate the IEEE 754 standard, (c) assume relaxed OpenCL numerical compliance requirements as defined in the unsafe math optimization section of the OpenCL C or OpenCL SPIR-V Environment specifications, and (d) may violate edge case behavior in the OpenCL C or OpenCL SPIR-V Environment specifications. This option includes the `-cl-no-signed-zeros`, `-cl-mad-enable`, and `-cl-denorms-are-zero` [13] options.

`-cl-finite-math-only`

Allow optimizations for floating-point arithmetic that assume that arguments and results are not NaNs, +Inf, -Inf. This option may violate the OpenCL numerical compliance requirements for

single precision and double precision floating-point, as well as edge case behavior.

`-cl-fast-relaxed-math`

Sets the optimization options `-cl-finite-math-only` and `-cl-unsafe-math-optimizations`. This option causes the preprocessor macro `__FAST_RELAXED_MATH__` to be defined in the OpenCL program.

5.8.6.4. Options to Request or Suppress Warnings

Warnings are diagnostic messages that report constructions which are not inherently erroneous but which are risky or suggest there may have been an error. The following language-independent options do not enable specific warnings but control the kinds of diagnostics produced by the OpenCL compiler. These options are ignored for programs created with IL.

`-w`

Inhibit all warning messages.

`-Werror`

Make all warnings into errors.

5.8.6.5. Options Controlling the OpenCL C Version

The following option controls the version of OpenCL C that the compiler accepts. These options are ignored for programs created with IL.

`-cl-std=`

Determine the OpenCL C language version to use. A value for this option must be provided. Valid values are:

- `CL1.1`: Support the OpenCL C 1.1 language features defined in *section 6* of the OpenCL 1.1 specification or in the unified OpenCL C specification.
- `CL1.2`: Support the OpenCL C 1.2 language features defined in *section 6* of the OpenCL 1.2 specification or in the unified OpenCL C specification.
- `CL2.0`: Support the OpenCL C 2.0 language features defined in the OpenCL C 2.0 specification or in the unified OpenCL C specification.
- `CL3.0`: Support the OpenCL C 3.0 language features defined in the unified OpenCL C specification.
- `CL3.1`: Support the OpenCL C 3.1 language features defined in the unified OpenCL C specification.

Calls to **clBuildProgram** or **clCompileProgram** with the `-cl-std=CL1.1` option **will fail** to compile the program for any devices with `CL_DEVICE_OPENCL_C_VERSION` equal to OpenCL C 1.0 and when `CL_DEVICE_OPENCL_C_ALL_VERSIONS` does not include OpenCL C 1.1.

Calls to **clBuildProgram** or **clCompileProgram** with the `-cl-std=CL1.2` option **will fail** to compile the program for any devices with `CL_DEVICE_OPENCL_C_VERSION` equal to OpenCL C 1.1 or earlier and when `CL_DEVICE_OPENCL_C_ALL_VERSIONS` does not include OpenCL C 1.2.

Calls to **clBuildProgram** or **clCompileProgram** with the `-cl-std=CL2.0` option **will fail** to compile the program for any devices with `CL_DEVICE_OPENCL_C_VERSION` equal to OpenCL C 1.2 or earlier and when `CL_DEVICE_OPENCL_C_ALL_VERSIONS` does not include OpenCL C 2.0.

Calls to **clBuildProgram** or **clCompileProgram** with the `-cl-std=CL3.0` option **will fail** to compile the program for any devices when `CL_DEVICE_OPENCL_C_ALL_VERSIONS` does not include OpenCL C 3.0.

Calls to **clBuildProgram** or **clCompileProgram** with the `-cl-std=CL3.1` option **will fail** to compile the program for any devices when `CL_DEVICE_OPENCL_C_ALL_VERSIONS` does not include OpenCL C 3.1.

If the `-cl-std` build option is not specified, the highest OpenCL C 1.x language version supported by each device is used when compiling the program for each device. Applications are required to specify the `-cl-std` build option to compile or build programs with newer versions of OpenCL C.

5.8.6.6. Options for Querying Kernel Argument Information

Querying for kernel argument information is missing before version 1.2.

`-cl-kernel-arg-info`

This option allows the compiler to store information about the arguments of a kernel(s) in the program executable. The argument information stored includes the argument name, its type, the address space and access qualifiers used. Refer to description of **clGetKernelArgInfo** on how to query this information.

5.8.6.7. Options for Debugging Your Program

Debugging options are missing before version 2.0.

`-g`

This option can currently be used to generate additional errors for the built-in functions that allow you to enqueue commands on a device (refer to OpenCL kernel languages specifications).

5.8.6.8. C++ for OpenCL

Applications may pass `-cl-std=CLC++` to **clCompileProgram** or **clBuildProgram** for programs created using **clCreateProgramWithSource** to request the program be built as C++ for OpenCL.

5.8.7. Linker Options

Linker options are missing before version 1.2.

This specification defines a standard set of linker options that must be supported by the OpenCL C compiler when linking compiled programs online or offline. These linker options are categorized as library linking options and program linking options. These may be extended by a set of vendor- or platform-specific options.

5.8.7.1. Library Linking Options

Library linking options are missing before version 1.2.

The following options can be specified when creating a library of compiled binaries.

`-create-library`

Create a library of compiled binaries specified in *input_programs* argument to **clLinkProgram**.

`-enable-link-options`

Allows the linker to modify the library behavior based on one or more link options (described in Program Linking Options) when this library is linked with a program executable. This option must be specified with the create-library option.

5.8.7.2. Program Linking Options

The following options can be specified when linking a program executable.

`-cl-denorms-are-zero`
`-cl-no-signed-zeros`
`-cl-unsafe-math-optimizations`
`-cl-finite-math-only`
`-cl-fast-relaxed-math`
`-cl-no-subgroup-ifp` (missing before version 2.1)

The options are described in Math Intrinsics Options and Optimization Options. The linker may apply these options to all compiled program objects specified to **clLinkProgram**. The linker may apply these options only to libraries which were created with the option `-enable-link-options`.

5.8.7.3. SPIR Compilation Options

If the `cl_khr_spir` extension is supported, the compile option

```
-x spir
```

must be specified to indicate that the binary is in SPIR format, and the compile option

```
-spir-std
```

must be used to specify the version of the SPIR specification that describes the format and meaning of the binary.

For example, if the binary is as described in SPIR version 1.2, then

```
-spir-std=1.2
```

must be specified. Failing to specify these compile options may result in implementation-defined behavior.

5.8.8. Unloading the OpenCL Compiler

To unload an OpenCL compiler for a platform, call the function

```
// Provided by CL_VERSION_1_2
cl_int clUnloadPlatformCompiler(
    cl_platform_id platform);
```

clUnloadPlatformCompiler is missing before version 1.2.

- *platform* is the platform to unload.

This function allows the implementation to release the resources allocated by the OpenCL compiler for *platform*. This is a hint from the application and does not guarantee that the compiler will not be used in the future or that the compiler will actually be unloaded by the implementation. Calls to **clBuildProgram**, **clCompileProgram** or **clLinkProgram** after **clUnloadPlatformCompiler** will reload the compiler, if necessary, to build the appropriate program executable.

clUnloadPlatformCompiler returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PLATFORM` if *platform* is not a valid platform.

Alternatively, if you are not using OpenCL via the ICD loader, you may unload the OpenCL compiler with the function

```
// Provided by CL_VERSION_1_0
cl_int clUnloadCompiler(void);
```

clUnloadCompiler is deprecated by version 1.2.

This function allows the implementation to release the resources allocated by the OpenCL compiler. This is a hint from the application and does not guarantee that the compiler will not be used in the future or that the compiler will actually be unloaded by the implementation. Calls to **clBuildProgram**, **clCompileProgram** or **clLinkProgram** after **clUnloadCompiler** will reload the compiler, if necessary, to build the appropriate program executable.

clUnloadCompiler will always return `CL_SUCCESS`.

5.8.9. Program Object Queries

To return information about a program object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetProgramInfo(
    cl_program program,
    cl_program_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *program* specifies the program object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetProgramInfo** is described in the Program Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Program Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 44. List of supported param_names by **clGetProgramInfo**

Program Info	Return Type	Description
`CL_PROGRAM_REFERENCE_COUNT` [14]	`cl_uint`	Return the *program* reference count.
`CL_PROGRAM_CONTEXT`	`cl_context`	Return the context specified when the program object is created
`CL_PROGRAM_NUM_DEVICES`	`cl_uint`	Return the number of devices associated with *program*.
`CL_PROGRAM_DEVICES`	`cl_device_id`[]	Return the list of devices associated with the program object. This can be the devices associated with context on which the program object has been created or can be a subset of devices that are specified when a program object is created using **clCreateProgramWithBinary**.

Program Info	Return Type	Description
`CL_PROGRAM_SOURCE`	`char`[]	Return the program source code specified by **clCreateProgramWithSource**. The source string returned is a concatenation of all source strings specified to **clCreateProgramWithSource** with a null terminator. The concatenation strips any nulls in the original source strings. If *program* is created using **clCreateProgramWithBinary**, **clCreateProgramWithIL**, **clCreateProgramWithILKHR**, or **clCreateProgramWithBuiltInKernels**, a null string or the appropriate program source code is returned depending on whether or not the program source code is stored in the binary. The actual number of characters that represents the program source code including the null terminator is returned in *param_value_size_ret*.

`CL_PROGRAM_IL`
 	
missing before version 2.1.
 	
`CL_PROGRAM_IL_KHR`
 	
provided by the `cl_khr_il_program` extension.
	`char`[]	Returns the program IL for programs created with **clCreateProgramWithILKHR** or **clCreateProgramWithIL**.
		
		If *program* is created with **clCreateProgramWithSource**, **clCreateProgramWithBinary** or **clCreateProgramWithBuiltInKernels** the memory pointed to by param_value will be unchanged and *param_value_size_ret* will be set to 0.

`CL_PROGRAM_BINARY_SIZES`	`size_t`[]	Returns an array that contains the size in bytes of the program binary (could be an executable binary, compiled binary or library binary) for each device associated with program. The size of the array is the number of devices associated with program. If a binary is not available for a device(s), a size of zero is returned. If *program* is created using **clCreateProgramWithBuiltInKernels**, the implementation may return zero in any entries of the returned array.

Program Info	Return Type	Description
`CL_PROGRAM_BINARIES`	`unsigned char*[]`	Return the program binaries (could be an executable binary, compiled binary or library binary) for all devices associated with program. For each device in program, the binary returned can be the binary specified for the device when program is created with **clCreateProgramWithBinary** or it can be the executable binary generated by **clBuildProgram** or **clLinkProgram**. If *program* is created with **clCreateProgramWithSource** or **clCreateProgramWithIL**, the binary returned is the binary generated by **clBuildProgram**, **clCompileProgram** or **clLinkProgram**. The bits returned can be an implementation-specific intermediate representation (a.k.a. IR) or device specific executable bits or both. The decision on which information is returned in the binary is up to the OpenCL implementation. param_value points to an array of `n` pointers allocated by the caller, where `n` is the number of devices associated with program. The buffer sizes needed to allocate the memory that these `n` pointers refer to can be queried using the `CL_PROGRAM_BINARY_SIZES` query as described in this table. Each entry in this array is used by the implementation as the location in memory where to copy the program binary for a specific device, if there is a binary available. To find out which device the program binary in the array refers to, use the `CL_PROGRAM_DEVICES` query to get the list of devices. There is a one-to-one correspondence between the array of n pointers returned by `CL_PROGRAM_BINARIES` and array of devices returned by `CL_PROGRAM_DEVICES`.
`CL_PROGRAM_NUM_KERNELS` missing before version 1.2.	`size_t`	Returns the number of kernels declared in *program* that can be created with **clCreateKernel**. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*.

Program Info	Return Type	Description
CL_PROGRAM_KERNEL_NAMES missing before version 1.2.	char[]	Returns a semi-colon separated list of kernel names in *program* that can be created with **clCreateKernel**. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*.
CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT missing before version 2.2 and deprecated by version 3.0.	cl_bool	This indicates that the *program* object contains non-trivial constructor(s) that will be executed by runtime before any kernel from the program is executed. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*. Querying CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT may unconditionally return CL_FALSE if no devices associated with *program* support constructors for program scope global variables. Support for constructors and destructors for program scope global variables is required only for OpenCL 2.2 devices.
CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT missing before version 2.2 and deprecated by version 3.0.	cl_bool	This indicates that the program object contains non-trivial destructor(s) that will be executed by runtime when *program* is destroyed. This information is only available after a successful program executable has been built for at least one device in the list of devices associated with *program*. Querying CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT may unconditionally return CL_FALSE if no devices associated with *program* support destructors for program scope global variables. Support for constructors and destructors for program scope global variables is required only for OpenCL 2.2 devices.

clGetProgramInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_PROGRAM if *program* is a not a valid program object.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Program Object Queries table and *param_value* is not NULL.
- CL_INVALID_PROGRAM_EXECUTABLE if *param_name* is CL_PROGRAM_NUM_KERNELS, CL_PROGRAM_KERNEL_

NAMES, CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT, or CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT and a successful program executable has not been built for at least one device in the list of devices associated with *program*.

- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To return build information for each device in the program object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetProgramBuildInfo(
    cl_program program,
    cl_device_id device,
    cl_program_build_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *program* specifies the program object being queried.
- *device* specifies the device for which build information is being queried. *device* must be a valid device associated with *program*.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetProgramBuildInfo** is described in the Program Build Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Program Build Queries table. If *param_value* is NULL, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is NULL, it is ignored.

Table 45. List of supported param_names by **clGetProgramBuildInfo**

Program Build Info	Return Type	Description
CL_PROGRAM_BUILD_STATUS	cl_build_status	Returns the build, compile or link status, whichever was performed last on the specified *program* object for *device*. This can be one of the following: CL_BUILD_NONE - The build status returned if no **clBuildProgram**, **clCompileProgram** or **clLinkProgram** has been performed on the specified *program* object for *device*). CL_BUILD_ERROR - The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** - whichever was performed last on the specified *program* object for *device* - generated an error. CL_BUILD_SUCCESS - The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** - whichever was performed last on the specified *program* object for *device* - was successful. CL_BUILD_IN_PROGRESS - The build status returned if **clBuildProgram**, **clCompileProgram** or **clLinkProgram** - whichever was performed last on the specified *program* object for *device* - has not finished.
CL_PROGRAM_BUILD_OPTIONS	char[]	Return the build, compile or link options specified by the options argument in **clBuildProgram**, **clCompileProgram** or **clLinkProgram**, whichever was performed last on the specified *program* object for *device*. If build status of the specified *program* for *device* is CL_BUILD_NONE, an empty string is returned.
CL_PROGRAM_BUILD_LOG	char[]	Return the build, compile or link log for **clBuildProgram**, **clCompileProgram** or **clLinkProgram**, whichever was performed last on program for device. If build status of the specified *program* for *device* is CL_BUILD_NONE, an empty string is returned.

Program Build Info	Return Type	Description
`CL_PROGRAM_BINARY_TYPE` missing before version 1.2.	`cl_program_binary_type`	Return the program binary type for device. This can be one of the following values: `CL_PROGRAM_BINARY_TYPE_NONE` - There is no binary associated with the specified *program* object for *device*. `CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT` - A compiled binary is associated with *device*. This is the case when the specified *program* object was created using **clCreateProgramWithSource** and compiled using **clCompileProgram**, or when a compiled binary was loaded using **clCreateProgramWithBinary**. `CL_PROGRAM_BINARY_TYPE_LIBRARY` - A library binary is associated with *device*. This is the case when the specified *program* object was linked by **clLinkProgram** using the `-create-library` link option, or when a compiled library binary was loaded using **clCreateProgramWithBinary**. `CL_PROGRAM_BINARY_TYPE_EXECUTABLE` - An executable binary is associated with *device*. This is the case when the specified *program* object was linked by **clLinkProgram** without the `-create-library` link option, or when an executable binary was built using **clBuildProgram**. `CL_PROGRAM_BINARY_TYPE_INTERMEDIATE` — An intermediate (non-source) representation for the program is loaded as a binary. The program must be further processed with **clCompileProgram** or **clBuildProgram**. If processed with **clCompileProgram**, the result will be a binary of type `CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT` or `CL_PROGRAM_BINARY_TYPE_LIBRARY`. If processed with **clBuildProgram**, the result will be a binary of type `CL_PROGRAM_BINARY_TYPE_EXECUTABLE`.

Program Build Info	Return Type	Description
`CL_PROGRAM_BUILD_GLOBAL_VARIABLE_TOTAL_SIZE` missing before version 2.0.	`size_t`	The total amount of storage, in bytes, used by program variables in the global address space.

clGetProgramBuildInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is a not a valid program object.
- `CL_INVALID_DEVICE` if *device* is not in the list of devices associated with *program*.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Program Build Queries table and *param_value* is not `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

A program binary (compiled binary, library binary or executable binary) built for a parent device can be used by all its sub-devices. If a program binary has not been built for a sub-device, the program binary associated with the parent device will be used.

A program binary for a device specified with **clCreateProgramWithBinary** or queried using **clGetProgramInfo** can be used as the binary for the associated root device, and all sub-devices created from the root-level device or sub-devices thereof.

5.9. Kernel Objects

A kernel is a function declared in a program. A kernel is identified by the `__kernel` qualifier applied to any function in a program. A kernel object encapsulates the specific `__kernel` function declared in a program and the argument values to be used when executing this `__kernel` function.

5.9.1. Creating Kernel Objects

To create a kernel object, use the function

```
// Provided by CL_VERSION_1_0
cl_kernel clCreateKernel(
    cl_program program,
    const char* kernel_name,
    cl_int* errcode_ret);
```

- *program* is a program object with a successfully built executable.
- *kernel_name* is a function name in the program declared with the `__kernel` qualifier.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

clCreateKernel returns a valid non-zero kernel object and *errcode_ret* is set to `CL_SUCCESS` if the kernel object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_INVALID_PROGRAM_EXECUTABLE` if there is no successfully built executable for *program*.
- `CL_INVALID_KERNEL_NAME` if *kernel_name* is not found in *program*.
- `CL_INVALID_KERNEL_DEFINITION` if the function definition for `__kernel` function given by *kernel_name* such as the number of arguments, the argument types are not the same for all devices for which the *program* executable has been built.
- `CL_INVALID_VALUE` if *kernel_name* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To create kernel objects for all kernel functions in a program, call the function

```
// Provided by CL_VERSION_1_0
cl_int clCreateKernelsInProgram(
    cl_program program,
    cl_uint num_kernels,
    cl_kernel* kernels,
    cl_uint* num_kernels_ret);
```

- *program* is a program object with a successfully built executable.
- *num_kernels* is the size of memory pointed to by *kernels* specified as the number of `cl_kernel` entries.
- *kernels* is the buffer where the kernel objects for kernels in *program* will be returned. If *kernels* is `NULL`, it is ignored. If *kernels* is not `NULL`, *num_kernels* must be greater than or equal to the number of kernels in *program*.
- *num_kernels_ret* is the number of kernels in *program*. If *num_kernels_ret* is `NULL`, it is ignored.

Kernel objects are not created for any `__kernel` functions in *program* that do not have the same function definition across all devices for which a program executable has been successfully built.

Kernel objects can only be created once you have a program object with a valid program source or binary loaded into the program object and the program executable has been successfully built for one or more devices associated with program. No changes to the program executable are allowed

while there are kernel objects associated with a program object. This means that calls to **clBuildProgram** and **clCompileProgram** return `CL_INVALID_OPERATION` if there are kernel objects attached to a program object. The OpenCL context associated with *program* will be the context associated with *kernel*. The list of devices associated with *program* are the devices associated with *kernel*. Devices associated with a program object for which a valid program executable has been built can be used to execute kernels declared in the program object.

clCreateKernelsInProgram will return `CL_SUCCESS` if the kernel objects were successfully allocated. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM` if *program* is not a valid program object.
- `CL_INVALID_PROGRAM_EXECUTABLE` if there is no successfully built executable for any device in *program*.
- `CL_INVALID_VALUE` if *kernels* is not `NULL` and *num_kernels* is less than the number of kernels in *program*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To retain a kernel object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainKernel(
    cl_kernel kernel);
```

- *kernel* is the kernel object to be retained.

The *kernel* reference count is incremented.

clRetainKernel returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

clCreateKernel or **clCreateKernelsInProgram** do an implicit retain.

To release a kernel object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseKernel(
    cl_kernel kernel);
```

- *kernel* is the kernel object to be released.

The *kernel* reference count is decremented.

The kernel object is deleted once the number of instances that are retained to *kernel* become zero and the kernel object is no longer needed by any enqueued commands that use *kernel*. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainKernel** causes undefined behavior.

clReleaseKernel returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.9.2. Setting Kernel Arguments

To execute a kernel, the kernel arguments must be set.

To set the argument value for a specific argument of a kernel, call the function

```
// Provided by CL_VERSION_1_0
cl_int clSetKernelArg(
    cl_kernel kernel,
    cl_uint arg_index,
    size_t arg_size,
    const void* arg_value);
```

- *kernel* is a valid kernel object.
- *arg_index* is the kernel argument index. Kernel arguments are referred to by indices that go from zero to *n - 1*, where *n* is the total number of arguments declared by the kernel.
- *arg_size* specifies the size of the kernel argument value.
- *arg_value* is a pointer to the data for the kernel argument. The argument data pointed to by *arg_value* is copied and the *arg_value* pointer can therefore be reused by the application after **clSetKernelArg** returns. The argument data is used by all API calls that enqueue the kernel until the argument is changed by another call to **clSetKernelArg** for the kernel.

If the kernel argument being set is a pointer to the `global` or `constant` address space, then *arg_value* must point to a buffer memory object or `NULL`, or *arg_value* must be `NULL`. If *arg_value* is `NULL` or

points to NULL, then the kernel argument will be set to NULL.

If the kernel argument being set is a pointer to the local address space, then *arg_value* must be NULL, and *arg_size* specifies the amount of local memory in bytes that are allocated for the kernel argument. Prior to OpenCL 3.1, *arg_size* must be greater than zero. For OpenCL 3.1 and newer, *arg_size* may be zero, indicating that no local memory is required for the kernel argument. [15]

If the kernel argument being set is an image object, then *arg_value* must point to an image memory object. Additionally:

- If the kernel argument is a 1D image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE1D.
- If the kernel argument is a 2D image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D.
- If the kernel argument is a 3D image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE3D.
- If the kernel argument is a 1D image buffer, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE1D_BUFFER.
- If the kernel argument is a 1D image array, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE1D_ARRAY.
- If the kernel argument is a 2D image array, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D_ARRAY.
- If the kernel argument is a 2D depth image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D and image channel order CL_DEPTH.
- If the kernel argument is a 2D depth image array, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D_ARRAY and image channel order CL_DEPTH.
- If the kernel argument is a 2D MSAA image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D.
- If the kernel argument is a 2D MSAA image array, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D_ARRAY.
- If the kernel argument is a 2D MSAA depth image, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D and image channel order CL_DEPTH.
- If the kernel argument is a 2D MSAA depth image array, then the image memory object must be of image type CL_MEM_OBJECT_IMAGE2D_ARRAY and image channel order CL_DEPTH.

Behavior is undefined if the same image memory object is passed as both a read_only image and a write_only image, or as a read_write image and either a read_only image or a write_only image.

If the kernel argument being set is a sampler, then *arg_value* must point to a sampler object.

If the kernel argument being set is a device queue, then *arg_value* must point to a device queue object.

If the kernel argument being set is a pipe, then *arg_value* must point to a pipe memory object.

For all other kernel arguments, *arg_value* points to the data that is used as the kernel argument value.

All OpenCL objects set as kernel arguments must be created from the same context as the kernel object.

A kernel object does not update the reference count for objects such as memory or sampler objects specified as argument values by **clSetKernelArg**. Users may not rely on a kernel object to retain objects specified as argument values to the kernel.

Implementations shall not allow `cl_kernel` objects to hold reference counts to `cl_kernel` arguments, because no mechanism is provided for the user to tell the kernel to release that ownership right. If the kernel holds ownership rights on kernel args, that would make it impossible for users to tell with certainty when they may safely release user allocated resources associated with OpenCL objects such as the `cl_mem` backing store used with `CL_MEM_USE_HOST_PTR`.

clSetKernelArg returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL`
  - if *kernel* is not a valid kernel
- `CL_INVALID_ARG_INDEX`
  - if *arg_index* is not a valid argument index
- `CL_INVALID_CONTEXT`
  - if the context associated with *kernel* and an OpenCL object pointed to by *arg_value* are not the same
- `CL_INVALID_MEM_OBJECT`
  - if *arg_value* is `NULL` and it must point to a valid memory object
  - if *arg_value* points to `NULL` and it must point to a valid memory object
  - if *arg_value* is not `NULL`, and does not point to `NULL`, and does not point to a valid memory object
  - if *arg_value* points to a valid memory object, but the memory object is not valid for the kernel argument specified by *arg_index*
- `CL_INVALID_SAMPLER`
  - if the kernel argument is a sampler, but *arg_value* does not point to a valid sampler object
- `CL_INVALID_DEVICE_QUEUE`
  - if the kernel argument is a device queue, but *arg_value* does not point to a valid device queue object. This error code is missing before version 2.0.
- `CL_INVALID_ARG_SIZE`
  - if *arg_value* points to a memory object and *arg_size* is not equal to `sizeof(cl_mem)`
  - if *arg_value* points to a sampler object and *arg_size* is not equal to `sizeof(cl_sampler)`

    - if *arg_value* points to a device queue object and *arg_size* is not equal to `sizeof(cl_command_queue)`
    - if the kernel argument is a pointer to the `local` address space and *arg_size* is zero. This error condition does not apply if the platform associated with *kernel* is OpenCL 3.1 or newer.
    - if *arg_value* points to the data to be used as the kernel argument value and *arg_size* does not match the size of the data type for the argument
- `CL_INVALID_ARG_VALUE`
    - if the argument is an image declared with the `read_only` qualifier and *arg_value* points to an image object created with the memory flag `CL_MEM_WRITE_ONLY`
    - if the argument is an image declared with the `write_only` qualifier and *arg_value* points to an image object created with the memory flag `CL_MEM_READ_ONLY` or `CL_MEM_IMMUTABLE_EXT`
- `CL_MAX_SIZE_RESTRICTION_EXCEEDED`
    - if the size in bytes of the memory object (if the argument is a memory object) or *arg_size* (if the argument is declared with `local` qualifier) exceeds a language-specified maximum size restriction for this argument, such as the **MaxByteOffset** SPIR-V decoration. This error code is missing before version 2.2.
- `CL_OUT_OF_RESOURCES`
    - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
    - if there is a failure to allocate resources required by the OpenCL implementation on the host

When **clSetKernelArg** returns an error code different from `CL_SUCCESS`, the internal state of *kernel* may only be modified when that error code is `CL_OUT_OF_RESOURCES` or `CL_OUT_OF_HOST_MEMORY`. When the internal state of *kernel* is modified, it is implementation-defined whether:

- The argument value that was previously set is kept so that it can be used in further kernel enqueues.
- The argument value is unset such that a subsequent kernel enqueue fails with `CL_INVALID_KERNEL_ARGS`. [16]

To set a SVM pointer as the argument value for a specific argument of a kernel, call the function

```
// Provided by CL_VERSION_2_0
cl_int clSetKernelArgSVMPointer(
    cl_kernel kernel,
    cl_uint arg_index,
    const void* arg_value);
```

clSetKernelArgSVMPointer is missing before version 2.0.

- *kernel* is a valid kernel object.

- *arg_index* is the argument index. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to *n* - 1, where *n* is the total number of arguments declared by a kernel.
- *arg_value* is the SVM pointer that should be used as the argument value for argument specified by *arg_index*. The SVM pointer specified is the value used by all API calls that enqueue *kernel* (**clEnqueueNDRangeKernel** and **clEnqueueTask**) until the argument value is changed by a call to **clSetKernelArgSVMPointer** for *kernel*. The SVM pointer can only be used for arguments that are declared to be a pointer to `global` or `constant` memory. The SVM pointer value must be aligned according to the arguments type. For example, if the argument is declared to be `global float4 *p`, the SVM pointer value passed for `p` must be at a minimum aligned to a `float4`. The SVM pointer value specified as the argument value can be the pointer returned by **clSVMAlloc** or can be a pointer offset into the SVM region.

clSetKernelArgSVMPointer returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.
- `CL_INVALID_OPERATION` if no devices in the context associated with *kernel* support SVM.
- `CL_INVALID_ARG_INDEX` if *arg_index* is not a valid argument index.
- `CL_INVALID_ARG_VALUE` if *arg_value* specified is not a valid value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To set a device pointer as the argument value for a specific argument of a kernel, call the function

```
// Provided by cl_ext_buffer_device_address
cl_int clSetKernelArgDevicePointerEXT(
    cl_kernel kernel,
    cl_uint arg_index,
    cl_mem_device_address_ext arg_value);
```

clSetKernelArgDevicePointerEXT is provided by the `cl_ext_buffer_device_address` extension.

- *kernel* is a valid kernel object.
- *arg_index* is the argument index. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to *n* - 1, where *n* is the total number of arguments declared by a kernel.
- *arg_value* is the device pointer that should be used as the argument value for argument specified by *arg_index*. The device pointer specified is the value used by all API calls that enqueue *kernel* (**clEnqueueNDRangeKernel** and **clEnqueueTask**) until the argument value is changed by a call to **clSetKernelArgDevicePointerEXT** for *kernel*. The device pointer can only

be used for arguments that are declared to be a pointer to `global` memory allocated with **clCreateBufferWithProperties** with the `CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT` property. The pointer value specified as the argument value can be the pointer to the beginning of the buffer or any offset into the buffer region. The device pointer value must be naturally aligned according to the argument's type. It should be noted that it's legal to pass invalid pointers as the value (similarly to C/C++ function calls with pointer arguments) as long as the kernel doesn't dereference the pointer.

clSetKernelArgDevicePointerEXT returns `CL_SUCCESS` if the argument was set successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.
- `CL_INVALID_OPERATION` if no devices in the context associated with *kernel* support the `cl_ext_buffer_device_address` extension.
- `CL_INVALID_ARG_INDEX`
  - if *arg_index* is not a valid argument index.
  - if *arg_index* corresponds to an argument that is not a pointer to the `global` address space.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To set additional execution information for a kernel, call the function

```
// Provided by CL_VERSION_2_0
cl_int clSetKernelExecInfo(
    cl_kernel kernel,
    cl_kernel_exec_info param_name,
    size_t param_value_size,
    const void* param_value);
```

clSetKernelExecInfo is missing before version 2.0.

- *kernel* is a valid kernel object.
- *param_name* specifies the type of information to set. The list of supported *param_name* types and the corresponding values passed in *param_value* is described in the Kernel Execution Properties table.
- *param_value_size* specifies the size in bytes of the memory pointed to by *param_value*.
- *param_value* is a pointer to memory where the appropriate values determined by *param_name* are specified.

Table 46. List of supported param_names by **clSetKernelExecInfo**

Kernel Exec Info	Type	Description
`CL_KERNEL_EXEC_INFO_SVM_PTRS` missing before version 2.0.	`void`*[]	Specifies a set of pointers to SVM allocations that may be accessed by the kernel in addition to those set directly as kernel arguments. Each of the pointers can be the pointer returned by **clSVMAlloc** or can be a pointer to the middle of an SVM allocation. It is sufficient to specify one pointer for each SVM allocation. Behavior is undefined if the kernel accesses a coarse-grain or fine-grain buffer SVM allocation that is not set as a kernel argument and is not in the set specified by `CL_KERNEL_EXEC_INFO_SVM_PTRS`. The complete set of pointers is specified by each call to **clSetKernelExecInfo** and replaces any previously specified set of pointers. To specify that no SVM allocations will be accessed by a kernel other than those set as kernel arguments, specify an empty set by passing *param_value_size* equal to zero and *param_value* equal to `NULL`. Non-argument pointers to SVM allocations must be specified for coarse-grain and fine-grain buffer SVM allocations, but not for fine-grain system SVM allocations.

Kernel Exec Info	Type	Description
CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM missing before version 2.0.	cl_bool	Specifies whether the kernel may use pointers to system allocations that are not set directly as kernel arguments on devices that support fine-grain system SVM allocations. When a device supports fine-grain system SVM allocations and CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM is CL_TRUE, the kernel may access system allocations that are not set directly as kernel arguments. Otherwise, if a device does not support fine-grain system SVM allocations or when CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM is CL_FALSE, behavior is undefined if the kernel accesses a system allocation that is not set as a kernel argument. If **clSetKernelExecInfo** has not been called with a value for CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM, the default value is CL_TRUE.
CL_KERNEL_EXEC_INFO_DEVICE_PTRS_EXT provided by the cl_ext_buffer_device_address extension.	cl_mem_device_address_ext[]	Device pointers must reference locations contained entirely within buffers that are passed to kernel as arguments, or that are passed through the execution information. Non-argument device pointers accessed by the kernel must be specified by passing pointers to those buffers via this **clSetKernelExecInfo** option.

clSetKernelExecInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_INVALID_OPERATION if *param_name is* CL_KERNEL_EXEC_INFO_SVM_PTRS *and no devices in the context associated with _kernel* support SVM.
- CL_INVALID_OPERATION if *param_name* is CL_KERNEL_EXEC_INFO_SVM_FINE_GRAIN_SYSTEM and *param_value* is CL_TRUE and no devices in the context associated with *kernel* support fine-grain system SVM allocations.
- CL_INVALID_OPERATION if *param_name* is CL_KERNEL_EXEC_INFO_DEVICE_PTRS_EXT and no devices in the context associated with *kernel* support the cl_ext_buffer_device_address extension.
- CL_INVALID_VALUE if *param_name* is not valid, if *param_value* is NULL and *param_value_size* is greater than zero, or if the size specified by *param_value_size* is not valid.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.9.3. Copying Kernel Objects

Copying kernel objects is missing before version 2.1.

To clone a kernel object, call the function

```
// Provided by CL_VERSION_2_1
cl_kernel clCloneKernel(
    cl_kernel source_kernel,
    cl_int* errcode_ret);
```

clCloneKernel is missing before version 2.1.

- *source_kernel* is a valid `cl_kernel` object that will be copied. *source_kernel* will not be modified in any way by this function.
- *errcode_ret* will be assigned an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Cloning is used to make a shallow copy of the kernel object, its arguments and any information passed to the kernel object using **clSetKernelExecInfo**. If the kernel object was ready to be enqueued before copying it, the clone of the kernel object is ready to enqueue.

The returned kernel object is an exact copy of *source_kernel*, with one caveat: the reference count on the returned kernel object is set as if it had been returned by **clCreateKernel**. The reference count of *source_kernel will* not be changed.

The resulting kernel will be in the same state as if **clCreateKernel** is called to create the resultant kernel with the same arguments as those used to create *source_kernel*, the latest call to **clSetKernelArg** or **clSetKernelArgSVMPointer** for each argument index applied to kernel and the last call to **clSetKernelExecInfo** for each value of the param name parameter are applied to the new kernel object.

All arguments of the new kernel object must be intact and it may be correctly used in the same situations as kernel except those that assume a pre-existing reference count. Setting arguments on the new kernel object will not affect *source_kernel* except insofar as the argument points to a shared underlying entity and in that situation behavior is as if two kernel objects had been created and the same argument applied to each. Only the data stored in the kernel object is copied; data referenced by the kernels arguments are not copied. For example, if a buffer or pointer argument is set on a kernel object, the pointer is copied but the underlying memory allocation is not.

clCloneKernel returns a valid non-zero kernel object and *errcode_ret* is set to `CL_SUCCESS` if the kernel is successfully copied. Otherwise it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.

- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.9.4. Kernel Object Queries

To return information about a kernel object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetKernelInfo(
    cl_kernel kernel,
    cl_kernel_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *kernel* specifies the kernel object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelInfo** is described in the Kernel Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Kernel Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 47. List of supported param_names by **clGetKernelInfo**

Kernel Info	Return Type	Description
`CL_KERNEL_FUNCTION_NAME`	`char[]`	Return the kernel function name.
`CL_KERNEL_NUM_ARGS`	`cl_uint`	Return the number of arguments to kernel.
`CL_KERNEL_REFERENCE_COUNT` [17]	`cl_uint`	Return the *kernel* reference count.
`CL_KERNEL_CONTEXT`	`cl_context`	Return the context associated with *kernel*.
`CL_KERNEL_PROGRAM`	`cl_program`	Return the program object associated with kernel.

Kernel Info	Return Type	Description
`CL_KERNEL_ATTRIBUTES` missing before version 1.2.	`char[]`	Returns any attributes specified using the `__attribute__` OpenCL C qualifier (or using an OpenCL C++ qualifier syntax [[]]) with the kernel function declaration in the program source. These attributes include attributes described in the earlier OpenCL C kernel language specifications and other attributes supported by an implementation. Attributes are returned as they were declared inside `__attribute__((...))`, with any surrounding whitespace and embedded newlines removed. When multiple attributes are present, they are returned as a single, space delimited string. For kernels not created from OpenCL C source and the **clCreateProgramWithSource** API call the string returned from this query will be empty.

clGetKernelInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is a not a valid kernel object.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Kernel Object Queries table and *param_value* is not `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To return information about the kernel object that may be specific to a device, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetKernelWorkGroupInfo(
    cl_kernel kernel,
    cl_device_id device,
    cl_kernel_work_group_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *kernel* specifies the kernel object being queried.

- *device* identifies a specific device in the list of devices associated with *kernel*. The list of devices is the list of devices in the OpenCL context that is associated with *kernel*. If the list of devices associated with *kernel* is a single device, *device* can be a `NULL` value.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelWorkGroupInfo** is described in the Kernel Object Device Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Kernel Object Device Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 48. List of supported param_names by **clGetKernelWorkGroupInfo**

Kernel Work-group Info	Return Type	Description
`CL_KERNEL_GLOBAL_WORK_SIZE` missing before version 1.2.	`size_t`[3]	Provides a mechanism for the application to query the maximum global size that can be used to execute a built-in kernel on a specific device given by *device*. If *kernel* is not a built-in kernel, querying `CL_KERNEL_GLOBAL_WORK_SIZE` returns the error `CL_INVALID_VALUE`.
`CL_KERNEL_WORK_GROUP_SIZE`	`size_t`	Provides a mechanism for the application to query the maximum work-group size that can be used to execute the kernel on a specific device given by *device*. The OpenCL implementation uses the resource requirements of the kernel (register usage etc.) to determine what this work-group size should be. As a result and unlike `CL_DEVICE_MAX_WORK_GROUP_SIZE` this value may vary from one kernel to another as well as one device to another. `CL_KERNEL_WORK_GROUP_SIZE` will be less than or equal to `CL_DEVICE_MAX_WORK_GROUP_SIZE` for a given kernel object.
`CL_KERNEL_COMPILE_WORK_GROUP_SIZE`	`size_t`[3]	Returns the work-group size specified in the kernel source or IL. If the work-group size is not specified in the kernel source or IL, (0, 0, 0) is returned.

Kernel Work-group Info	Return Type	Description
`CL_KERNEL_LOCAL_MEM_SIZE`	`cl_ulong`	Returns the amount of local memory in bytes being used by a kernel. This includes local memory that may be needed by an implementation to execute the kernel, variables declared inside the kernel with the `__local` address qualifier and local memory to be allocated for arguments to the kernel declared as pointers with the `__local` address qualifier and whose size is specified with **clSetKernelArg**. If the local memory size, for any pointer argument to the kernel declared with the `__local` address qualifier, is not specified, its size is assumed to be 0.
`CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`	`size_t`	Returns the preferred multiple of work-group size for launch. This is a performance hint. Specifying a work-group size that is not a multiple of the value returned by this query as the value of the local work size argument to **clEnqueueNDRangeKernel** will not fail to enqueue the kernel for execution unless the work-group size specified is larger than the device maximum.
`CL_KERNEL_PRIVATE_MEM_SIZE`	`cl_ulong`	Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variable declared inside the kernel with the `__private` qualifier.

clGetKernelWorkGroupInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is a not a valid kernel object.
- `CL_INVALID_DEVICE` if *device* is not in the list of devices associated with *kernel* or if *device* is `NULL` but there is more than one device associated with *kernel*.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Kernel Object Device Queries table and *param_value* is not `NULL`.
- `CL_INVALID_VALUE` if *param_name* is `CL_KERNEL_GLOBAL_WORK_SIZE` and *kernel* is not a built-in kernel.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL

implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To return information about a kernel object, call the function

```
// Provided by CL_VERSION_2_1
cl_int clGetKernelSubGroupInfo(
    cl_kernel kernel,
    cl_device_id device,
    cl_kernel_sub_group_info param_name,
    size_t input_value_size,
    const void* input_value,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetKernelSubGroupInfo is missing before version 2.1.

or the equivalent

```
// Provided by cl_khr_subgroups
cl_int clGetKernelSubGroupInfoKHR(
    cl_kernel in_kernel,
    cl_device_id in_device,
    cl_kernel_sub_group_info param_name,
    size_t input_value_size,
    const void* input_value,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetKernelSubGroupInfoKHR is provided by the `cl_khr_subgroups` extension.

- *kernel* specifies the kernel object being queried.
- *device* identifies a specific device in the list of devices associated with *kernel*. The list of devices is the list of devices in the OpenCL context that is associated with *kernel*. If the list of devices associated with *kernel* is a single device, *device* can be a `NULL` value.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelSubGroupInfo** is described in the Kernel Object Sub-group Queries table.
- *input_value_size* is used to specify the size in bytes of memory pointed to by *input_value*. This size must be == size of input type as described in the table below.
- *input_value* is a pointer to memory where the appropriate parameterization of the query is passed from. If *input_value* is `NULL`, it is ignored.

- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Kernel Object Sub-group Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 49. List of supported param_names by **clGetKernelSubGroupInfo**

Kernel Sub-group Info	Input Type	Return Type	Description
`CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE` missing before version 2.1. The equivalent `CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE_KHR` may be used if the `cl_khr_subgroups` extension is supported.	`size_t*`	`size_t`	Returns the maximum sub-group size for this kernel. All sub-groups must be the same size, while the last sub-group in any work-group (i.e. the sub-group with the maximum index) could be the same or smaller size. The *input_value* must be an array of `size_t` values corresponding to the local work size parameter of the intended dispatch. The number of dimensions in the ND-range will be inferred from the value specified for *input_value_size*.

Kernel Sub-group Info	Input Type	Return Type	Description
`CL_KERNEL_SUB_GROUP_COUNT_FOR_NDRANGE` missing before version 2.1. The equivalent `CL_KERNEL_SUB_GROUP_COUNT_FOR_NDRANGE_KHR` may be used if the `cl_khr_subgroups` extension is supported.	`size_t*`	`size_t`	Returns the number of sub-groups that will be present in each work-group for a given local work size. All work-groups, apart from the last work-group in each dimension in the presence of non-uniform work-group sizes, will have the same number of sub-groups. The *input_value* must be an array of `size_t` values corresponding to the local work size parameter of the intended dispatch. The number of dimensions in the ND-range will be inferred from the value specified for *input_value_size*.

Kernel Sub-group Info	Input Type	Return Type	Description
`CL_KERNEL_LOCAL_SIZE_FOR_SUB_GROUP_COUNT` missing before version 2.1. Also see `cl_khr_subgroups`.	`size_t`	`size_t`[]	Returns the local size that will generate the requested number of sub-groups for the kernel. The output array must be an array of `size_t` values corresponding to the local size parameter. Any returned work-group will have one dimension. Other dimensions inferred from the value specified for param_value_size will be filled with the value 1. The returned value will produce an exact number of sub-groups and result in no partial groups for an executing kernel except in the case where the last work-group in a dimension has a size different from that of the other groups. If no work-group size can accommodate the requested number of sub-groups, 0 will be returned in each element of the return array.

Kernel Sub-group Info	Input Type	Return Type	Description
CL_KERNEL_MAX_NUM_SUB_GROUPS missing before version 2.1. Also see cl_khr_subgroups.	ignored	size_t	This provides a mechanism for the application to query the maximum number of sub-groups that may make up each work-group to execute a kernel on a specific device given by device. The OpenCL implementation uses the resource requirements of the kernel (register usage etc.) to determine what this work-group size should be. The returned value may be used to compute a work-group size to enqueue the kernel with to give a round number of sub-groups for an enqueue.
CL_KERNEL_COMPILE_NUM_SUB_GROUPS missing before version 2.1. Also see cl_khr_subgroups.	ignored	size_t	Returns the number of sub-groups per work-group specified in the kernel source or IL. If the sub-group count is not specified then 0 is returned.

clGetKernelSubGroupInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_KERNEL if *kernel* is a not a valid kernel object.
- CL_INVALID_DEVICE if *device* is not in the list of devices associated with *kernel* or if *device* is NULL but there is more than one device associated with *kernel*.
- CL_INVALID_OPERATION if *device* does not support sub-groups.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Kernel Object Sub-group Queries table and *param_value* is not NULL.
- CL_INVALID_VALUE if *param_name* is CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE, CL_KERNEL_SUB_

GROUP_COUNT_FOR_NDRANGE or CL_KERNEL_LOCAL_SIZE_FOR_SUB_GROUP_COUNT and the size in bytes specified by *input_value_size* is not valid or if *input_value* is NULL.

- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To return information about the arguments of a kernel, call the function

```
// Provided by CL_VERSION_1_2
cl_int clGetKernelArgInfo(
    cl_kernel kernel,
    cl_uint arg_index,
    cl_kernel_arg_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetKernelArgInfo is missing before version 1.2.

- *kernel* specifies the kernel object being queried.
- *arg_index* is the argument index. Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to *n* - 1, where *n* is the total number of arguments declared by a kernel.
- *param_name* specifies the argument information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetKernelArgInfo** is described in the Kernel Argument Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is NULL, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Kernel Argument Queries table. If *param_value* is NULL, it is ignored.
- *param_value_size ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is NULL, it is ignored.

Kernel argument information is only available if the program object associated with *kernel*:

- is created with **clCreateProgramWithBinary** and the program executable is built with the -cl -kernel-arg-info and -x spir options specified in the *options* argument to **clBuildProgram** or **clCompileProgram**, if the cl_khr_spir extension is supported; or,
- is created with **clCreateProgramWithSource** and the program executable is built with the -cl -kernel-arg-info option specified in the *options* argument to **clBuildProgram** or **clCompileProgram**,

Table 50. List of supported param_names by **clGetKernelArgInfo**

Kernel Arg Info	Return Type	Description

`CL_KERNEL_ARG_ADDRESS_QUALIFIER`
 	
missing before version 1.2.
	`cl_kernel_arg_address_qualifier`	Returns the address qualifier specified for the argument given by *arg_index*. This can be one of the following values:
		
		`CL_KERNEL_ARG_ADDRESS_GLOBAL`
		`CL_KERNEL_ARG_ADDRESS_LOCAL`
		`CL_KERNEL_ARG_ADDRESS_CONSTANT`
		`CL_KERNEL_ARG_ADDRESS_PRIVATE`
		
		If no address qualifier is specified, the default address qualifier which is `CL_KERNEL_ARG_ADDRESS_PRIVATE` is returned.

`CL_KERNEL_ARG_ACCESS_QUALIFIER`
 	
missing before version 1.2.
	`cl_kernel_arg_access_qualifier`	Returns the access qualifier specified for the argument given by *arg_index*. This can be one of the following values:
		
		`CL_KERNEL_ARG_ACCESS_READ_ONLY`
		`CL_KERNEL_ARG_ACCESS_WRITE_ONLY`
		`CL_KERNEL_ARG_ACCESS_READ_WRITE`
		`CL_KERNEL_ARG_ACCESS_NONE`
		
		If argument is not an image type and is not declared with the pipe qualifier, `CL_KERNEL_ARG_ACCESS_NONE` is returned. If argument is an image type, the access qualifier specified or the default access qualifier is returned.

`CL_KERNEL_ARG_TYPE_NAME` missing before version 1.2.	`char[]`	Returns the type name specified for the argument given by *arg_index*. The type name returned will be the argument type name as it was declared with any whitespace removed. If argument type name is an unsigned scalar type (i.e. unsigned char, unsigned short, unsigned int, unsigned long), uchar, ushort, uint and ulong will be returned. The argument type name returned does not include any type qualifiers.

Kernel Arg Info	Return Type	Description

`CL_KERNEL_ARG_TYPE_QUALIFIER`
 	
missing before version 1.2.
	`cl_kernel_arg_type_qualifier`	Returns a bitfield describing one or more type qualifiers specified for the argument given by *arg_index*. The returned values can be:
		
		`CL_KERNEL_ARG_TYPE_CONST`
		`CL_KERNEL_ARG_TYPE_RESTRICT`
		`CL_KERNEL_ARG_TYPE_VOLATILE`
		`CL_KERNEL_ARG_TYPE_PIPE`, or
		`CL_KERNEL_ARG_TYPE_NONE`
		
		`CL_KERNEL_ARG_TYPE_CONST` is returned if the kernel argument is a pointer and the referenced type is declared with the `const` qualifier. For example, a kernel argument declared as `global int const*` returns `CL_KERNEL_ARG_TYPE_CONST` but a kernel argument declared as `global int* const` does not. Additionally, `CL_KERNEL_ARG_TYPE_CONST` is returned if the kernel argument is declared with the `constant` address space qualifier.
		
		`CL_KERNEL_ARG_TYPE_RESTRICT` is returned if the pointer type is marked `restrict`. For example, `global int* restrict` returns `CL_KERNEL_ARG_TYPE_RESTRICT`.
		
		`CL_KERNEL_ARG_TYPE_VOLATILE` is returned for `CL_KERNEL_ARG_TYPE_QUALIFIER` if the kernel argument is a pointer and the referenced type is declared with the `volatile` qualifier. For example, a kernel argument declared as `global int volatile*` returns `CL_KERNEL_ARG_TYPE_VOLATILE` but a kernel argument declared as `global int* volatile` does not.
		
		`CL_KERNEL_ARG_TYPE_NONE` is returned for all kernel arguments passed by value.

`CL_KERNEL_ARG_NAME` missing before version 1.2.	`char[]`	Returns the name specified for the argument given by *arg_index*.

clGetKernelArgInfo returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_KERNEL` if *kernel* is a not a valid kernel object.
- `CL_INVALID_ARG_INDEX` if *arg_index* is not a valid argument index.

- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Kernel Argument Queries table and *param_value* is not `NULL`.
- `CL_KERNEL_ARG_INFO_NOT_AVAILABLE` if the argument information is not available for kernel.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To query a suggested local work size for a kernel object, call the function

```
// Provided by CL_VERSION_3_1
cl_int clGetKernelSuggestedLocalWorkSize(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    size_t* suggested_local_work_size);
```

clGetKernelSuggestedLocalWorkSize is missing before version 3.1.

or the equivalent

```
// Provided by cl_khr_suggested_local_work_size
cl_int clGetKernelSuggestedLocalWorkSizeKHR(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    size_t* suggested_local_work_size);
```

clGetKernelSuggestedLocalWorkSizeKHR is provided by the `cl_khr_suggested_local_work_size` extension.

- *command_queue* specifies the command-queue and device for the query.
- *kernel* specifies the kernel object and kernel arguments for the query. The OpenCL context associated with *kernel* and *command_queue* must the same.
- *work_dim* specifies the number of work dimensions in the input global work offset and global work size, and the output suggested local work size.
- *global_work_offset* can be used to specify an array of at least *work_dim* global ID offset values for the query. This is optional and may be `NULL` to indicate there is no global ID offset.
- *global_work_size* is an array of at least *work_dim* values describing the global work size for the

query.

- *suggested_local_work_size* is an output array of at least *work_dim* values that will contain the result of the query.

The returned suggested local work size is expected to match the local work size that would be chosen if the specified kernel object, with the same kernel arguments, were enqueued into the specified command-queue with the specified global work size, specified global work offset, and with a `NULL` local work size.

clGetKernelSuggestedLocalWorkSize returns `CL_SUCCESS` if the query executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_KERNEL`
  - if *kernel* is not a valid kernel
- `CL_INVALID_PROGRAM_EXECUTABLE`
  - if there is no successfully built program executable available for the device associated with *command_queue*
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *kernel* are not the same
- `CL_INVALID_KERNEL_ARGS`
  - if any kernel arguments for *kernel* have not been set
- `CL_INVALID_WORK_DIMENSION`
  - if *work_dim* is not valid for the device associated with *command_queue* (is greater than the value returned for `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`)
- `CL_INVALID_GLOBAL_OFFSET`
  - if *global_work_offset* is not `NULL`. This error condition does not apply when the device associated with *command_queue* supports OpenCL 1.1 or newer.
  - if the value specified in *global_work_size* plus the corresponding value in *global_work_offset* for any dimensions is greater than the maximum value representable by `size_t` on the device associated with *command_queue*
- `CL_INVALID_GLOBAL_WORK_SIZE`
  - if *global_work_size* is `NULL`
  - if any of the values specified in *global_work_size*[0], … *global_work_size*[*work_dim* - 1] are zero
  - if any of the values specified in *global_work_size*[0], … *global_work_size*[*work_dim* - 1] exceed the maximum value representable by `size_t` on the device associated with *command_queue*
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`

- if a kernel argument for *kernel* is a sub-buffer object and the offset specified when the sub-buffer object is created is not aligned to `CL_DEVICE_MEM_BASE_ADDR_ALIGN` for the device associated with *command_queue*. This error code is missing before version 1.1.

- `CL_INVALID_IMAGE_SIZE`
  - if a kernel argument for *kernel* is an image and the dimensions of the image, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if a kernel argument for *kernel* is an image and the format of the image, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- `CL_INVALID_OPERATION`
  - if SVM pointers are set as arguments for *kernel* and the device associated with *command_queue* does not support SVM
  - if system pointers are set as arguments for *kernel* and the device associated with *command_queue* does not support fine-grain system SVM
- `CL_INVALID_VALUE`
  - if *suggested_local_work_size* is `NULL`
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

These error conditions are consistent with error conditions for **clEnqueueNDRangeKernel**.

5.10. Executing Kernels

To enqueue a command to execute a kernel on a device, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueNDRangeKernel(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    const size_t* local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* is a valid host command-queue. The kernel will be queued for execution on the device associated with *command_queue*.
- *kernel* is a valid kernel object. The OpenCL context associated with *kernel* and *command-queue* must be the same.
- *work_dim* is the number of dimensions used to specify the global work-items and work-items in the work-group. *work_dim* must be greater than zero and less than or equal to `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`.
- *global_work_offset* can be used to specify an array of *work_dim* unsigned values that describe the offset used to calculate the global ID of a work-item. If *global_work_offset* is `NULL`, the global IDs start at offset (0, 0, 0). *global_work_offset* must be `NULL` before version 1.1.
- *global_work_size* points to an array of *work_dim* unsigned values that describe the number of global work-items in *work_dim* dimensions that will execute the kernel function. The total number of global work-items is computed as *global_work_size*[0] × … × *global_work_size*[*work_dim* - 1]. If the device associated with *command_queue* is an OpenCL 2.1 or newer device, and *global_work_size* is `NULL` or the value in any passed dimension is zero, then the kernel command will trivially succeed after its event dependencies are satisfied and subsequently update its completion event. The behavior in this situation is similar to that of an enqueued marker, except that unlike a marker, an enqueued kernel with no events passed to *event_wait_list* may run at any time.
- *local_work_size* points to an array of *work_dim* unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by *kernel*. The total number of work-items in a work-group is computed as *local_work_size*[0] × … × *local_work_size*[*work_dim* - 1]. The total number of work-items in the work-group must be less than or equal to the `CL_KERNEL_WORK_GROUP_SIZE` value specified in the Kernel Object Device Queries table, and the number of work-items specified in *local_work_size*[0], …, *local_work_size*[*work_dim* - 1] must be less than or equal to the corresponding values specified by `CL_DEVICE_MAX_WORK_GROUP_SIZES`[0], …, `CL_DEVICE_MAX_WORK_GROUP_SIZES`[*work_dim* - 1]. The explicitly specified *local_work_size* will be used to determine how to break the global work-items specified by *global_work_size* into appropriate work-group instances.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

An ND-range kernel command may require uniform work-groups or may support non-uniform

work-groups. To support non-uniform work-groups:

1. The device associated with *command_queue* must support non-uniform work-groups.
2. The program object associated with *kernel* must support non-uniform work-groups. Specifically, this means:
    a. If the program was created with **clCreateProgramWithSource**, then the program must be compiled for OpenCL C 2.0 or newer (e.g. by using the `-cl-std=CL2.0` or newer build option), and without the `-cl-uniform-work-group-size` build option.
    b. If the program was created with **clCreateProgramWithIL** or **clCreateProgramWithBinary**, then the program must be compiled or built without the `-cl-uniform-work-group-size` build options.
    c. If the program was created using **clLinkProgram**, then all input programs must support non-uniform work-groups.

If non-uniform work-groups are supported, any single dimension for which the global size is not divisible by the local size will be partitioned into two regions. One region will have work-groups that have the same number of work-items as was specified by the local size parameter in that dimension. The other region will have work-groups with less than the number of work items specified by the local size parameter in that dimension. The global IDs and group IDs of the work-items in the first region will be numerically lower than those in the second, and the second region will be at most one work-group wide in that dimension. Work-group sizes could be non-uniform in multiple dimensions, potentially producing work-groups of up to 4 different sizes in a 2D range and 8 different sizes in a 3D range.

If non-uniform work-groups are supported and *local_work_size* is `NULL`, the OpenCL runtime may choose a uniform or non-uniform work-group size.

Otherwise, when non-uniform work-groups are not supported, the size of each work-group must be uniform. If *local_work_size* is specified, the values specified in *global_work_size*[0], …, *global_work_size*[*work_dim* - 1] must be evenly divisible by the corresponding values specified in *local_work_size*[0], …, *local_work_size*[*work_dim* - 1]. If *local_work_size* is `NULL`, the OpenCL runtime must choose a uniform work-group size.

The work-group size to be used for *kernel* can also be specified in the program source or intermediate language. In this case the size of work-group specified by *local_work_size* must match the value specified in the program source.

These work-group instances are executed in parallel across multiple compute units or concurrently on the same compute unit.

Each work-item is uniquely identified by a global identifier. The global ID, which can be read inside the kernel, is computed using the value given by *global_work_size* and *global_work_offset*. In addition, a work-item is also identified within a work-group by a unique local ID. The local ID, which can also be read by the kernel, is computed using the value given by *local_work_size*. The starting local ID is always (0, 0, …, 0).

clEnqueueNDRangeKernel returns `CL_SUCCESS` if the kernel-instance was successfully queued. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid host command-queue
- `CL_INVALID_KERNEL`
  - if *kernel* is not a valid kernel
- `CL_INVALID_PROGRAM_EXECUTABLE`
  - if there is no successfully built program executable available for the device associated with *command_queue*
- `CL_INVALID_CONTEXT`
  - if the context associated with *command_queue* and *kernel* are not the same
  - if the context associated with *command_queue* and events in *event_wait_list* are not the same
- `CL_INVALID_KERNEL_ARGS`
  - if any kernel arguments for *kernel* have not been set
- `CL_INVALID_WORK_DIMENSION`
  - if *work_dim* is not valid for the device associated with *command_queue* (is greater than the value returned for `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS`)
- `CL_INVALID_GLOBAL_OFFSET`
  - if *global_work_offset* is not `NULL`. This error condition does not apply when the device associated with *command_queue* supports OpenCL 1.1 or newer.
  - if the value specified in *global_work_size* plus the corresponding value in *global_work_offset* for any dimensions is greater than the maximum value representable by `size_t` on the device associated with *command_queue*
- `CL_INVALID_GLOBAL_WORK_SIZE`
  - if *global_work_size* is `NULL`. This error condition does not apply when the device associated with *command_queue* supports OpenCL 2.1 or newer.
  - if any of the values specified in *global_work_size*[0], … *global_work_size*[*work_dim* - 1] are zero. This error condition does not apply when the device associated with *command_queue* supports OpenCL 2.1 or newer.
  - if any of the values specified in *global_work_size*[0], … *global_work_size*[*work_dim* - 1] exceed the maximum value representable by `size_t` on the device associated with *command_queue*
- `CL_INVALID_WORK_GROUP_SIZE`
  - if *local_work_size* is not `NULL`, if the work-group size must be uniform, and if the *global_work_size* is not evenly divisible by the *local_work_size*
  - if *local_work_size* is not `NULL` and if the total number of work-items in the work-group is zero
  - if *local_work_size* is not `NULL` and if the total number of work-items in the work-group is greater than the maximum work-group size supported for *kernel* on the device associated with *command_queue* (is greater than the value returned for `CL_KERNEL_WORK_GROUP_SIZE`)
  - if *local_work_size* is not `NULL` and if the *local_work_size* does not match the required work-

group size for *kernel*

  - if *local_work_size* is not `NULL` and if the *local_work_size* is not consistent with the required number of sub-groups for *kernel*

- `CL_INVALID_WORK_ITEM_SIZE`
  - if the number of work-items specified in any dimension of *local_work_size* is not valid for the device associated with *command_queue* (is greater than the corresponding value returned for `CL_DEVICE_MAX_WORK_GROUP_SIZES`)
- `CL_MISALIGNED_SUB_BUFFER_OFFSET`
  - if a kernel argument for *kernel* is a sub-buffer object and the offset specified when the sub-buffer object is created is not aligned to `CL_DEVICE_MEM_BASE_ADDR_ALIGN` for the device associated with *command_queue*. This error code is missing before version 1.1.
- `CL_INVALID_IMAGE_SIZE`
  - if a kernel argument for *kernel* is an image and the dimensions of the image, such as the image width or image height, are not supported by the device associated with *command_queue*
- `CL_IMAGE_FORMAT_NOT_SUPPORTED`
  - if a kernel argument for *kernel* is an image and the format of the image, such as the image channel order or image channel data type, are not supported by the device associated with *command_queue*
- `CL_MEM_OBJECT_ALLOCATION_FAILURE`
  - if there is a failure to allocate memory for the data store associated with any buffer or image object kernel arguments for *kernel*
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is greater than zero
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is zero
  - if event objects in *event_wait_list* are not valid events
- `CL_INVALID_OPERATION`
  - if SVM pointers are set as arguments for *kernel* and the device associated with *command_queue* does not support SVM
  - if system pointers are set as arguments for *kernel* and the device associated with *command_queue* does not support fine-grain system SVM
- `CL_OUT_OF_RESOURCES`
  - if there is a failure to allocate resources required by the OpenCL implementation on the device
- `CL_OUT_OF_HOST_MEMORY`
  - if there is a failure to allocate resources required by the OpenCL implementation on the host

To enqueue a command to execute a kernel on a device, using a single work-item, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueTask(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueTask is deprecated by version 2.0.

- *command_queue* is a valid host command-queue. The kernel will be queued for execution on the device associated with *command_queue*.
- *kernel* is a valid kernel object. The OpenCL context associated with *kernel* and *command-queue* must be the same.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

clEnqueueTask is equivalent to calling **clEnqueueNDRangeKernel** with *work_dim* set to 1, *global_work_offset* set to `NULL`, *global_work_size[0]* set to 1, and *local_work_size[0]* set to 1.

clEnqueueTask returns `CL_SUCCESS` if the kernel-instance was successfully queued. Otherwise, it returns one of the following errors:

- `CL_INVALID_PROGRAM_EXECUTABLE` if there is no successfully built program executable available for device associated with *command_queue*.
- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_KERNEL` if *kernel* is not a valid kernel object.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* and *kernel* are not the same or if the context associated with *command_queue* and events in *event_wait_list* are not the same.
- `CL_INVALID_KERNEL_ARGS` if the kernel argument values have not been specified.
- `CL_INVALID_WORK_GROUP_SIZE` if a work-group size is specified for *kernel* in the program source and it is not (1, 1, 1).
- `CL_INVALID_WORK_GROUP_SIZE` if the required number of sub-groups is specified for *kernel* in the

program source and is not consistent with a work-group size of (1, 1, 1).

- `CL_MISALIGNED_SUB_BUFFER_OFFSET` if a sub-buffer object is specified as the value for an argument that is a buffer object and the *offset* specified when the sub-buffer object is created is not aligned to `CL_DEVICE_MEM_BASE_ADDR_ALIGN` value for device associated with *queue*. This error code is missing before version 1.1.
- `CL_INVALID_IMAGE_SIZE` if an image object is specified as an argument value and the image dimensions (image width, height, specified or compute row and/or slice pitch) are not supported by device associated with *queue*.
- `CL_IMAGE_FORMAT_NOT_SUPPORTED` if an image object is specified as an argument value and the image format (image channel order and data type) is not supported by device associated with *queue*.
- `CL_OUT_OF_RESOURCES` if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel. See how this error code is used with **clEnqueueNDRangeKernel** for examples.
- `CL_MEM_OBJECT_ALLOCATION_FAILURE` if there is a failure to allocate memory for data store associated with image or buffer objects specified as arguments to *kernel*.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_INVALID_OPERATION` if SVM pointers are passed as arguments to a kernel and the device does not support SVM, or if system pointers are passed as arguments to a kernel and the device does not support fine-grain system SVM.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a command to execute a native C/C++ function not compiled using the OpenCL compiler, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueNativeKernel(
    cl_command_queue command_queue,
    void (CL_CALLBACK* user_func)(void*),
    void* args,
    size_t cb_args,
    cl_uint num_mem_objects,
    const cl_mem* mem_list,
    const void** args_mem_loc,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

- *command_queue* is a valid host command-queue. A native user function can only be executed on

a command-queue created on a device that has CL_EXEC_NATIVE_KERNEL capability set in CL_DEVICE_EXECUTION_CAPABILITIES as specified in the Device Queries table.

- *user_func* is a pointer to a host-callable user function. It is the application's responsibility to ensure that the host-callable user function is thread-safe.
- *args* is a pointer to the args list that *user_func* should be called with.
- *cb_args* is the size in bytes of the args list that *args* points to.
- *num_mem_objects* is the number of buffer objects that are passed in *args*.
- *mem_list* is a list of valid buffer objects, if *num_mem_objects* > 0. The buffer object values specified in *mem_list* are memory object handles (cl_mem values) returned by **clCreateBuffer** or **clCreateBufferWithProperties**, or NULL.
- *args_mem_loc* is a pointer to appropriate locations that *args* points to where memory object handles (cl_mem values) are stored. Before the user function is executed, the memory object handles are replaced by pointers to global memory.
- *event_wait_list*, *num_events_in_wait_list* and *event* are as described in **clEnqueueNDRangeKernel**.

The data pointed to by *args* and *cb_args* bytes in size will be copied and a pointer to this copied region will be passed to *user_func*. The copy needs to be done because the memory objects (cl_mem values) that *args* may contain need to be modified and replaced by appropriate pointers to global memory. When **clEnqueueNativeKernel** returns, the memory region pointed to by *args* can be reused by the application.

clEnqueueNativeKernel returns CL_SUCCESS if the user function execution instance was successfully queued. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_VALUE if *user_func* is NULL.
- CL_INVALID_VALUE if *args* is a NULL value and *cb_args* > 0, or if *args* is a NULL value and *num_mem_objects* > 0.
- CL_INVALID_VALUE if *args* is not NULL and *cb_args* is 0.
- CL_INVALID_VALUE if *num_mem_objects* > 0 and *mem_list* or *args_mem_loc* are NULL.
- CL_INVALID_VALUE if *num_mem_objects* = 0 and *mem_list* or *args_mem_loc* are not NULL.
- CL_INVALID_OPERATION if the device associated with *command_queue* cannot execute the native kernel.
- CL_INVALID_MEM_OBJECT if one or more memory objects specified in *mem_list* are not valid or are not buffer objects.
- CL_OUT_OF_RESOURCES if there is a failure to queue the execution instance of *kernel* on the command-queue because of insufficient resources needed to execute the kernel.
- CL_MEM_OBJECT_ALLOCATION_FAILURE if there is a failure to allocate memory for data store associated with buffer objects specified as arguments to *kernel*.

- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_INVALID_OPERATION if SVM pointers are passed as arguments to a kernel and the device does not support SVM or if system pointers are passed as arguments to a kernel and/or stored inside SVM allocations passed as kernel arguments and the device does not support fine grain system SVM allocations.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

The total number of read-only images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_READ_IMAGE_ARGS. Each image array argument to a kernel declared with the read_only qualifier counts as one image. The total number of write-only images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_WRITE_IMAGE_ARGS. Each image array argument to a kernel declared with the write_only qualifier counts as one image.

The total number of read-write images specified as arguments to a kernel cannot exceed CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS. Each image array argument to a kernel declared with the read_write qualifier counts as one image.

5.11. Event Objects

An event object can be used to track the execution status of a command. The API calls that enqueue commands to a command-queue create a new event object that is returned in the *event* argument. In case of an error enqueuing the command in the command-queue the event argument does not return an event object.

The execution status of an enqueued command at any given point in time can be one of the following:

- CL_QUEUED: Indicates that the command has been enqueued in a command-queue. This is the initial state of all events except user events.
- CL_SUBMITTED: The initial state for all user events. For all other events, indicates that the command has been submitted by the host to the device.
- CL_RUNNING: Indicates that the device has started executing this command. In order for the execution status of an enqueued command to change from CL_SUBMITTED to CL_RUNNING, all events that this command is waiting on must have completed successfully i.e. their execution status must be CL_COMPLETE.
- CL_COMPLETE: Indicates that the command has successfully completed.
- An Error Code: A negative integer value indicating that the command was abnormally terminated. Abnormal termination may occur for a number of reasons, such as a bad memory access.

A command is considered to be complete if its execution status is `CL_COMPLETE` or is a negative integer value.

If the execution of a command is terminated, the command-queue associated with this terminated command, and the associated context (and all other command-queues in this context) may no longer be available. The behavior of OpenCL API calls that use this context (and command-queues associated with this context) are now considered to be implementation-defined. The user registered callback function specified when context is created can be used to report appropriate error information.

5.11.1. Creating, Waiting for, and Releasing Event Objects

To create a user event object, call the function

```
// Provided by CL_VERSION_1_1
cl_event clCreateUserEvent(
    cl_context context,
    cl_int* errcode_ret);
```

clCreateUserEvent is missing before version 1.1.

- *context* must be a valid OpenCL context.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device.

clCreateUserEvent returns a valid non-zero event object and *errcode_ret* is set to `CL_SUCCESS` if the user event object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

The initial execution status for the user event object is `CL_SUBMITTED`.

To set the execution status of a user event object, call the function

```
// Provided by CL_VERSION_1_1
cl_int clSetUserEventStatus(
    cl_event event,
    cl_int execution_status);
```

clSetUserEventStatus is missing before version 1.1.

- *event* is a user event object created using **clCreateUserEvent**.
- *execution_status* specifies the new execution status to be set and can be `CL_COMPLETE` or a negative integer value to indicate an error. A negative integer value causes all enqueued commands that wait on this user event to be terminated. **clSetUserEventStatus** can only be called once to change the execution status of *event*.

If there are enqueued commands with user events in the *event_wait_list* argument of **clEnqueue*** commands, the user must ensure that the status of these user events being waited on are set using **clSetUserEventStatus** before any OpenCL APIs that release OpenCL objects except for event objects are called; otherwise the behavior is undefined.

For example, the following code sequence will result in undefined behavior of **clReleaseMemObject**.

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ..., 1, &ev1, NULL);
clEnqueueWriteBuffer(cq, buf2, CL_FALSE, ...);
clReleaseMemObject(buf2);
clSetUserEventStatus(ev1, CL_COMPLETE);
```

The following code sequence, however, works correctly.

```
ev1 = clCreateUserEvent(ctx, NULL);
clEnqueueWriteBuffer(cq, buf1, CL_FALSE, ..., 1, &ev1, NULL);
clEnqueueWriteBuffer(cq, buf2, CL_FALSE, ...);
clSetUserEventStatus(ev1, CL_COMPLETE);
clReleaseMemObject(buf2);
```

clSetUserEventStatus returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_EVENT` if *event* is not a valid user event object.
- `CL_INVALID_VALUE` if the *execution_status* is not `CL_COMPLETE` or a negative integer value.
- `CL_INVALID_OPERATION` if the *execution_status* for *event* has already been changed by a previous call to **clSetUserEventStatus**.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL

implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To wait for events to complete, call the function

```
// Provided by CL_VERSION_1_0
cl_int clWaitForEvents(
    cl_uint num_events,
    const cl_event* event_list);
```

- *num_events* is the number of events in *event_list*.
- *event_list* is a pointer to a list of event object handles.

This function waits on the host thread for commands identified by event objects in *event_list* to complete. A command is considered complete if its execution status is `CL_COMPLETE` or a negative value. The events specified in *event_list* act as synchronization points.

clWaitForEvents returns `CL_SUCCESS` if the execution status of all events in *event_list* is `CL_COMPLETE`. Otherwise, it returns one of the following errors:

- `CL_INVALID_VALUE` if *num_events* is zero or *event_list* is `NULL`.
- `CL_INVALID_CONTEXT` if events specified in *event_list* do not belong to the same context.
- `CL_INVALID_EVENT` if event objects specified in *event_list* are not valid event objects.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST` if the execution status of any of the events in *event_list* is a negative integer value. This error code is missing before version 1.1.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To return information about an event object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetEventInfo(
    cl_event event,
    cl_event_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *event* specifies the event object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetEventInfo** is described in the Event Object Queries table.

- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Event Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 51. List of supported param_names by **clGetEventInfo**

Event Info	Return Type	Description
`CL_EVENT_COMMAND_QUEUE`	`cl_command_queue`	Return the command-queue associated with *event*. For user event objects, a `NULL` value is returned. If the `cl_khr_command_buffer_multi_device` extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, `NULL` is returned.
`CL_EVENT_CONTEXT` missing before version 1.1.	`cl_context`	Return the context associated with *event*.
`CL_EVENT_COMMAND_TYPE`	`cl_command_type`	Return the command type associated with *event* as described in the Event Command Types table.

Event Info	Return Type	Description
CL_EVENT_COMMAND_EXECUTION_STATUS [18]	cl_int	Return the execution status of the command identified by event. Valid values are: CL_QUEUED - Command has been enqueued in the command-queue. CL_SUBMITTED - Enqueued command has been submitted by the host to the device associated with the command-queue. CL_RUNNING - Device is currently executing this command. CL_COMPLETE - The command has completed. Or an error code given by a negative integer value (command was abnormally terminated - this may be caused by a bad memory access etc.). These error codes come from the same set of error codes that are returned from the platform or runtime API calls as return values or **errcode_ret** values. If the cl_khr_command_buffer_multi_device extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues the semantics of execution status is as follows: CL_QUEUED - Command-buffer has been enqueued across the command-queues. CL_SUBMITTED - Commands from the command-buffer have been submitted by the host to any device associated with one of the command-queues. CL_RUNNING - Any command from the command-buffer has started execution on a device. CL_COMPLETE - All commands have completed on all devices.
CL_EVENT_REFERENCE_COUNT [19]	cl_uint	Return the *event* reference count.

Table 52. List of supported event command types

Events Created By	Event Command Type
clEnqueueNDRangeKernel	CL_COMMAND_NDRANGE_KERNEL
clEnqueueTask	CL_COMMAND_TASK
clEnqueueNativeKernel	CL_COMMAND_NATIVE_KERNEL
clEnqueueReadBuffer	CL_COMMAND_READ_BUFFER
clEnqueueWriteBuffer	CL_COMMAND_WRITE_BUFFER
clEnqueueCopyBuffer	CL_COMMAND_COPY_BUFFER
clEnqueueReadImage	CL_COMMAND_READ_IMAGE
clEnqueueWriteImage	CL_COMMAND_WRITE_IMAGE
clEnqueueCopyImage	CL_COMMAND_COPY_IMAGE
clEnqueueCopyBufferToImage	CL_COMMAND_COPY_BUFFER_TO_IMAGE
clEnqueueCopyImageToBuffer	CL_COMMAND_COPY_IMAGE_TO_BUFFER
clEnqueueMapBuffer	CL_COMMAND_MAP_BUFFER
clEnqueueMapImage	CL_COMMAND_MAP_IMAGE
clEnqueueUnmapMemObject	CL_COMMAND_UNMAP_MEM_OBJECT
clEnqueueMarker, **clEnqueueMarkerWithWaitList**	CL_COMMAND_MARKER
clEnqueueReadBufferRect	CL_COMMAND_READ_BUFFER_RECT missing before version 1.1.
clEnqueueWriteBufferRect	CL_COMMAND_WRITE_BUFFER_RECT missing before version 1.1.
clEnqueueCopyBufferRect	CL_COMMAND_COPY_BUFFER_RECT missing before version 1.1.
clCreateUserEvent	CL_COMMAND_USER missing before version 1.1.
clEnqueueBarrier, **clEnqueueBarrierWithWaitList**	CL_COMMAND_BARRIER missing before version 1.2.
clEnqueueMigrateMemObjects	CL_COMMAND_MIGRATE_MEM_OBJECTS missing before version 1.2.
clEnqueueFillBuffer	CL_COMMAND_FILL_BUFFER missing before version 1.2.

Events Created By	Event Command Type
clEnqueueFillImage	CL_COMMAND_FILL_IMAGE missing before version 1.2.
clEnqueueSVMFree	CL_COMMAND_SVM_FREE missing before version 2.0.
clEnqueueSVMMemcpy	CL_COMMAND_SVM_MEMCPY missing before version 2.0.
clEnqueueSVMMemFill	CL_COMMAND_SVM_MEMFILL missing before version 2.0.
clEnqueueSVMMap	CL_COMMAND_SVM_MAP missing before version 2.0.
clEnqueueSVMUnmap	CL_COMMAND_SVM_UNMAP missing before version 2.0.
clEnqueueSVMMigrateMem	CL_COMMAND_SVM_MIGRATE_MEM missing before version 3.0. Prior to OpenCL 3.0, implementations should return CL_COMMAND_MIGRATE_MEM_OBJECTS, but may return an implementation-defined event command type for clEnqueueSVMMigrateMem.
clEnqueueCommandBufferKHR	CL_COMMAND_COMMAND_BUFFER_KHR provided by the cl_khr_command_buffer extension.
clEnqueueAcquireDX9MediaSurfacesKHR	CL_COMMAND_ACQUIRE_DX9_MEDIA_SURFACES_KHR provided by the cl_khr_dx9_media_sharing extension.
clEnqueueReleaseDX9MediaSurfacesKHR	CL_COMMAND_RELEASE_DX9_MEDIA_SURFACES_KHR provided by the cl_khr_dx9_media_sharing extension.
clEnqueueAcquireD3D10ObjectsKHR	CL_COMMAND_ACQUIRE_D3D10_OBJECTS_KHR provided by the cl_khr_d3d10_sharing extension.
clEnqueueReleaseD3D10ObjectsKHR	CL_COMMAND_RELEASE_D3D10_OBJECTS_KHR provided by the cl_khr_d3d10_sharing extension.

Events Created By	Event Command Type
clEnqueueAcquireD3D11ObjectsKHR	CL_COMMAND_ACQUIRE_D3D11_OBJECTS_KHR provided by the cl_khr_d3d11_sharing extension.
clEnqueueReleaseD3D11ObjectsKHR	CL_COMMAND_RELEASE_D3D11_OBJECTS_KHR provided by the cl_khr_d3d11_sharing extension.
clEnqueueAcquireEGLObjectsKHR	CL_COMMAND_ACQUIRE_EGL_OBJECTS_KHR provided by the cl_khr_egl_image extension.
clEnqueueReleaseEGLObjectsKHR	CL_COMMAND_RELEASE_EGL_OBJECTS_KHR provided by the cl_khr_egl_image extension.
clCreateEventFromEGLSyncKHR	CL_COMMAND_EGL_FENCE_SYNC_OBJECT_KHR provided by the cl_khr_egl_image extension.
clEnqueueAcquireExternalMemObjectsKHR	CL_COMMAND_ACQUIRE_EXTERNAL_MEM_OBJECTS_KHR provided by the cl_khr_external_memory extension.
clEnqueueReleaseExternalMemObjectsKHR	CL_COMMAND_RELEASE_EXTERNAL_MEM_OBJECTS_KHR provided by the cl_khr_external_memory extension.
clEnqueueAcquireGLObjects	CL_COMMAND_ACQUIRE_GL_OBJECTS
clEnqueueReleaseGLObjects	CL_COMMAND_RELEASE_GL_OBJECTS
clCreateEventFromGLsyncKHR	CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR provided by the cl_khr_gl_event extension.
clEnqueueSignalSemaphoresKHR	CL_COMMAND_SEMAPHORE_SIGNAL_KHR provided by the cl_khr_semaphore extension.
clEnqueueWaitSemaphoresKHR	CL_COMMAND_SEMAPHORE_WAIT_KHR provided by the cl_khr_semaphore extension.

Calling clGetEventInfo with CL_EVENT_COMMAND_EXECUTION_STATUS is intended to be a lightweight query, therefore it is not a host synchronization point. This means that *event* does not global-synchronize-with the call to clGetEventInfo, even if the returned execution status is CL_COMPLETE. When a host synchronization point is needed, call a function that waits on the event instead, such as clWaitForEvents. Refer to the memory consistency model for details.

clGetEventInfo returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_EVENT` if *event* is a not a valid event object.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Event Object Queries table and *param_value* is not `NULL`.
- `CL_INVALID_VALUE` if the information to query given in *param_name* cannot be queried for *event*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To register a user callback function for a specific command execution status, call the function

```
// Provided by CL_VERSION_1_1
cl_int clSetEventCallback(
    cl_event event,
    cl_int command_exec_callback_type,
    void (CL_CALLBACK* pfn_notify)(cl_event event, cl_int event_command_status, void
*user_data),
    void* user_data);
```

clSetEventCallback is missing before version 1.1.

- *event* is a valid event object.
- *command_exec_callback_type* specifies the command execution status for which the callback is registered. The command execution status types for which a callback can be registered are `CL_SUBMITTED`, `CL_RUNNING`, or `CL_COMPLETE`. The callback function registered for a *command_exec_callback_type* value of `CL_COMPLETE` will be called when the command has completed successfully or is abnormally terminated.
- *pfn_event_notify* is the event callback function that can be registered by the application. This callback function may be called asynchronously by the OpenCL implementation. It is the application's responsibility to ensure that the callback function is thread-safe. The parameters to this callback function are:
  - *event* is the event object for which the callback function is invoked.
  - *event_command_status* is equal to the *command_exec_callback_type* used while registering the callback. Refer to the Event Object Queries table for the command execution status values. If the callback is called as the result of the command associated with event being abnormally terminated, an appropriate error code for the error that caused the termination will be passed to *event_command_status* instead.
  - *user_data* is a pointer to user supplied data.
- *user_data* will be passed as the *user_data* argument when *pfn_notify* is called. *user_data* can be `NULL`.

Each call to **clSetEventCallback** registers the specified user callback function on a callback stack associated with *event*. The order in which the registered user callback functions are called is

undefined.

The registered callback function will be called when the execution status of the command associated with *event* changes to an execution status equal to or past the status specified by *command_exec_status*, or for the execution status `CL_COMPLETE`, if the command is abnormally terminated. There is no guarantee that the callback functions registered for various command execution status values for an event will be called in the exact order that the execution status of a command changes. Furthermore, it should be noted that calling a callback for an event execution status other than `CL_COMPLETE` in no way implies that the memory model or execution model as defined by the OpenCL specification has changed. For example, it is not valid to assume that a corresponding memory transfer has completed unless the event is in the state `CL_COMPLETE`.

All callbacks registered for an event object must be called before the event object is destroyed.

Callbacks should return promptly. Behavior is undefined when calling expensive system routines, OpenCL APIs to create contexts or command-queues, or blocking OpenCL APIs in an event callback. Rather than calling a blocking OpenCL API in an event callback, applications may call a non-blocking OpenCL API, then register a completion callback for the non-blocking OpenCL API with the remainder of the work.

Because commands in a command-queue are not required to begin execution until the command-queue is flushed, callbacks that enqueue commands on a command-queue should either call **clFlush** on the queue before returning, or arrange for the command-queue to be flushed later.

clSetEventCallback returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_EVENT` if *event* is not a valid event object.
- `CL_INVALID_VALUE` if *pfn_event_notify* is `NULL` or if *command_exec_callback_type* is not `CL_SUBMITTED`, `CL_RUNNING`, or `CL_COMPLETE`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To retain an event object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clRetainEvent(
    cl_event event);
```

- *event* is the event object to be retained.

The *event* reference count is incremented. The OpenCL commands that return an event perform an implicit retain.

clRetainEvent returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_EVENT` if *event* is not a valid event object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To release an event object, call the function

```
// Provided by CL_VERSION_1_0
cl_int clReleaseEvent(
    cl_event event);
```

- *event* is the event object to be released.

The *event* reference count is decremented.

The event object is deleted once the reference count becomes zero, the specific command identified by this event has completed (or terminated) and there are no commands in the command-queues of a context that require a wait for this event to complete. Using this function to release a reference that was not obtained by creating the object or by calling **clRetainEvent** causes undefined behavior.

Developers should be careful when releasing their last reference count on events created by **clCreateUserEvent** that have not yet been set to status of `CL_COMPLETE` or an error. If the user event was used in the event_wait_list argument passed to a **clEnqueue*** API or another application host thread is waiting for it in **clWaitForEvents**, those commands and host threads will continue to wait for the event status to reach `CL_COMPLETE` or error, even after the application has released the object. Since in this scenario the application has released its last reference count to the user event, it would be in principle no longer valid for the application to change the status of the event to unblock all the other machinery. As a result the waiting tasks will wait forever, and associated events, `cl_mem` objects, command-queues and contexts are likely to leak. In-order command-queues caught up in this deadlock may cease to do any work.

clReleaseEvent returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_EVENT` if *event* is not a valid event object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.11.1.1. Linking Event Objects to EGL Fence Sync Objects

An event object may be created by linking to an EGL **fence sync object**.

To create an OpenCL event object linked to an EGL fence sync object, call the function

```
// Provided by cl_khr_egl_event
cl_event clCreateEventFromEGLSyncKHR(
    cl_context context,
    CLeglSyncKHR sync,
    CLeglDisplayKHR display,
    cl_int* errcode_ret);
```

clCreateEventFromEGLSyncKHR is provided by the `cl_khr_egl_event` extension.

- *context* is a valid OpenCL context created from an OpenGL context or share group, using the `cl_khr_gl_sharing` extension.
- *sync* is the name of a sync object of type `EGL_SYNC_FENCE_KHR` created with respect to `EGLDisplay` *display*.
- *display* is the `EGLDisplay` handle.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Completion of such an event object is equivalent to waiting for completion of the fence command associated with the linked EGL sync object.

The parameters of an event object linked to an EGL sync object will return the following values when queried with **clGetEventInfo**:

- The `CL_EVENT_COMMAND_QUEUE` of a linked event is `NULL`, because the event is not associated with any OpenCL command-queue.
- The `CL_EVENT_COMMAND_TYPE` of a linked event is `CL_COMMAND_EGL_FENCE_SYNC_OBJECT_KHR`, indicating that the event is associated with a EGL sync object, rather than an OpenCL command.
- The `CL_EVENT_COMMAND_EXECUTION_STATUS` of a linked event is either `CL_SUBMITTED`, indicating that the fence command associated with the sync object has not yet completed, or `CL_COMPLETE`, indicating that the fence command has completed.

clCreateEventFromEGLSyncKHR performs an implicit **clRetainEvent** on the returned event object. Creating a linked event object also places a reference on the linked EGL sync object. When the event object is deleted, the reference will be removed from the EGL sync object.

Events returned from **clCreateEventFromEGLSyncKHR** may only be consumed by commands to acquire and release memory objects. Passing such events to any other CL API that enqueues commands will generate a `CL_INVALID_EVENT` error.

clCreateEventFromEGLSyncKHR returns a valid OpenCL event object and *errcode_ret* is set to `CL_SUCCESS` if the event object is created successfully. Otherwise, it returns a `NULL` value with one of

the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context, or was not created from a GL context.
- `CL_INVALID_EGL_OBJECT_KHR` if *sync* is not a valid EGLSyncKHR object of type `EGL_SYNC_FENCE_KHR` created with respect to `EGLDisplay` *display*.

5.11.1.1.1. Explicit Synchronization Using EGL Fence Sync Objects

If the `cl_khr_egl_event` extension is supported, event objects created with **clCreateEventFromEGLSyncKHR** provide another method of coordinating sharing between EGL / EGL client API objects, and OpenCL.

Completion of EGL and EGL client API commands may be determined by

- placing an EGL fence command after commands using `eglCreateSyncKHR`;
- creating an event from the resulting EGL sync object using **clCreateEventFromEGLSyncKHR**; and
- determining completion of that event object via **clEnqueueAcquireGLObjects**.

This method may be considerably more efficient than calling operations like `glFinish`, and is referred to as *explicit synchronization*. The application is responsible for ensuring the command stream associated with the EGL fence is flushed to ensure the CL queue is submitted to the device. Explicit synchronization is most useful when an EGL client API context bound to another thread is accessing the memory objects.

5.11.1.2. Linking Event Objects to OpenGL Fence Sync Objects

An event object may be created by linking to an OpenGL **fence sync object**.

To create an OpenCL event object linked to an OpenGL fence sync object, call the function

```
// Provided by cl_khr_gl_event
cl_event clCreateEventFromGLsyncKHR(
    cl_context context,
    cl_GLsync sync,
    cl_int* errcode_ret);
```

clCreateEventFromGLsyncKHR is provided by the `cl_khr_gl_event` extension.

- *context* is a valid OpenCL context created from an OpenGL context or share group, using the `cl_khr_gl_sharing` extension.
- *sync* is the name of a sync object in the GL share group associated with *context*.
- *errcode_ret* will return an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

Completion of such an event object is equivalent to waiting for completion of the fence command associated with the linked GL sync object.

clCreateEventFromGLsyncKHR returns a valid OpenCL event object and *errcode_ret* is set to `CL_SUCCESS` if the event object is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_CONTEXT` if *context* is not a valid context, or was not created from a GL context.
- `CL_INVALID_GL_OBJECT` if *sync* is not the name of a sync object in the GL share group associated with *context*.

The parameters of an event object linked to a GL sync object will return the following values when queried with **clGetEventInfo**:

- The `CL_EVENT_COMMAND_QUEUE` of a linked event is `NULL`, because the event is not associated with any OpenCL command-queue.
- The `CL_EVENT_COMMAND_TYPE` of a linked event is `CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR`, indicating that the event is associated with a GL sync object, rather than an OpenCL command.
- The `CL_EVENT_COMMAND_EXECUTION_STATUS` of a linked event is either `CL_SUBMITTED`, indicating that the fence command associated with the sync object has not yet completed, or `CL_COMPLETE`, indicating that the fence command has completed.

clCreateEventFromGLsyncKHR performs an implicit **clRetainEvent** on the returned event object. Creating a linked event object also places a reference on the linked GL sync object. When the event object is deleted, the reference will be removed from the GL sync object.

Events returned from **clCreateEventFromGLsyncKHR** can be used in the *event_wait_list* argument to **clEnqueueAcquireGLObjects** and CL APIs that take a `cl_event` as an argument but do not enqueue commands. Passing such events to any other CL API that enqueues commands will generate a `CL_INVALID_EVENT` error.

5.11.1.2.1. Explicit Synchronization Using OpenGL Fence Sync Objects

If the `cl_khr_gl_event` extension is supported, event objects created with **clCreateEventFromGLsyncKHR** provide another method of coordinating sharing of buffers and images between OpenGL and OpenCL.

Completion of OpenGL commands may be determined by

- placing an OpenGL fence command after commands using `glFenceSync`;
- creating an event from the resulting OpenGL sync object using **clCreateEventFromGLsyncKHR**; and
- determining completion of that event object via **clEnqueueAcquireGLObjects**.

This method may be considerably more efficient than calling `glFinish`, and is referred to as *explicit synchronization*. Explicit synchronization is most useful when an OpenGL context bound to another thread is accessing the memory objects.

Explicit synchronization is most useful when an OpenGL context bound to another thread is accessing the memory objects.

5.12. Markers, Barriers and Waiting for Events

To enqueue a marker command which waits for events or commands to complete, call the function

```
// Provided by CL_VERSION_1_2
cl_int clEnqueueMarkerWithWaitList(
    cl_command_queue command_queue,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueMarkerWithWaitList is missing before version 1.2.

- *command_queue* is a valid host command-queue.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.

If *event_wait_list* is `NULL`, then this particular command waits until all previous enqueued commands to *command_queue* have completed.

The marker command either waits for a list of events to complete, or if the list is empty it waits for all commands previously enqueued in *command_queue* to complete before it completes. This command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all events either in the *event_wait_list* or all previously enqueued commands, queued before this command to *command_queue*, have completed.

clEnqueueMarkerWithWaitList returns `CL_SUCCESS` if the function is successfully executed. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* and events in *event_wait_list* are not the same.
- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.

- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a marker command which waits for previous commands to complete, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueMarker(
    cl_command_queue command_queue,
    cl_event* event);
```

clEnqueueMarker is deprecated by version 1.2.

- *command_queue* is a valid host command-queue.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for this command to complete. If *event_wait_list* and *event* are not `NULL`, *event* must not refer to an element of the *event_wait_list* array.

The marker command waits for all commands previously enqueued in *command_queue* to complete before it completes. This command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all previously enqueued commands, queued before this command to *command_queue*, have completed.

clEnqueueMarker returns `CL_SUCCESS` if the function is successfully executed. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_VALUE` if *event* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a wait for a specific event or a list of events to complete before any future commands queued in a command-queue are executed, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueWaitForEvents(
    cl_command_queue command_queue,
    cl_uint num_events,
    const cl_event* event_list);
```

clEnqueueWaitForEvents is deprecated by version 1.2.

- *command_queue* is a valid host command-queue.
- *event_list* and *num_events* specify events that need to complete before this particular command can be executed.

The events specified in *event_list* act as synchronization points. The context associated with events in *event_list* and *command_queue* must be the same. The memory associated with *event_list* can be reused or freed after the function returns.

clEnqueueWaitForEvents returns `CL_SUCCESS` if the function is successfully executed. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_INVALID_CONTEXT` if context associated with *command_queue* and events in *event_list* are not the same.
- `CL_INVALID_VALUE` if *num_events* is 0 or *event_list* is `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a barrier command which waits for events or commands to complete, call the function

```
// Provided by CL_VERSION_1_2
cl_int clEnqueueBarrierWithWaitList(
    cl_command_queue command_queue,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueBarrierWithWaitList is missing before version 1.2.

- *command_queue* is a valid host command-queue.
- *event_wait_list* and *num_events_in_wait_list* specify events that need to complete before this particular command can be executed.
- If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and *command_queue* must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* returns an event object that identifies this command and can be used to query or wait for this command to complete. If *event* is `NULL` or the enqueue is unsuccessful, no event will be created and therefore it will not be possible to query the status of this command or to wait for

this command to complete. If *event_wait_list* and *event* are not NULL, *event* must not refer to an element of the *event_wait_list* array.

If *event_wait_list* is NULL, then this particular command waits until all previous enqueued commands to *command_queue* have completed.

The barrier command either waits for a list of events to complete, or if the list is empty it waits for all commands previously enqueued in *command_queue* to complete before it completes. This command blocks command execution, that is, any following commands enqueued after it do not execute until it completes. This command returns an *event* which can be waited on, i.e. this event can be waited on to insure that all events either in the *event_wait_list* or all previously enqueued commands, queued before this command to *command_queue*, have completed.

clEnqueueBarrierWithWaitList returns CL_SUCCESS if the function is successfully executed. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_INVALID_CONTEXT if context associated with *command_queue* and events in *event_wait_list* are not the same.
- CL_INVALID_EVENT_WAIT_LIST if *event_wait_list* is NULL and *num_events_in_wait_list* > 0, or *event_wait_list* is not NULL and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a barrier command which waits for commands to complete, call the function

```
// Provided by CL_VERSION_1_0
cl_int clEnqueueBarrier(
    cl_command_queue command_queue);
```

clEnqueueBarrier is deprecated by version 1.2.

- *command_queue* is a valid host command-queue.

The barrier command waits for all commands previously enqueued in *command_queue* to complete before it completes. This command blocks command execution, that is, any following commands enqueued after it do not execute until it completes. The barrier command is a synchronization point.

clEnqueueBarrier returns CL_SUCCESS if the function is successfully executed. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_QUEUE if *command_queue* is not a valid host command-queue.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL

implementation on the device.

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.13. Semaphores

This section describes the semaphore types and functions defined by the `cl_khr_semaphore` extension.

5.13.1. Semaphore Types

- `cl_semaphore_type_khr` represent the different types of semaphores.
  - It is mandatory to support `CL_SEMAPHORE_TYPE_BINARY_KHR`.
- `cl_semaphore_properties_khr` represents properties associated with semaphores.
  - `CL_SEMAPHORE_TYPE_KHR` must be supported.
- `cl_semaphore_info_khr` represents queries for additional information about semaphores.
  - All enums described in the “New API Enums” section of the `cl_khr_semaphore` extension for `cl_semaphore_info_khr` must be supported.
- `cl_semaphore_payload_khr` represents payload values of semaphores.
- `cl_semaphore_khr` represent semaphore objects.

5.13.2. Creating Semaphores

To create a **semaphore object**, call the function

```
// Provided by cl_khr_semaphore
cl_semaphore_khr clCreateSemaphoreWithPropertiesKHR(
    cl_context context,
    const cl_semaphore_properties_khr* sema_props,
    cl_int* errcode_ret);
```

clCreateSemaphoreWithPropertiesKHR is provided by the `cl_khr_semaphore` extension.

- *context* identifies a valid OpenCL context that the created `cl_semaphore_khr` will belong to.
- *sema_props* specifies additional semaphore properties in the form list of <property_name, property_value> pairs terminated with 0. `CL_SEMAPHORE_TYPE_KHR` must be part of the list of properties specified by *sema_props*.

Following new properties are added to the list of possible supported properties by `cl_semaphore_properties_khr` that can be passed to **clCreateSemaphoreWithPropertiesKHR**:

Table 53. List of supported semaphore creation properties by **clCreateSemaphoreWithPropertiesKHR**

Semaphore Property	Property Value	Description
CL_SEMAPHORE_TYPE_KHR	cl_semaphore_type_khr	Specifies the type of semaphore to create. This property is always required.
CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR	cl_device_id[]	Specifies the list of OpenCL devices (terminated with CL_SEMAPHORE_DEVICE_HANDLE_LIST_END_KHR) to associate with the semaphore. Only a single device is permitted in the list.
CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR provided by the cl_khr_external_semaphore extension.	cl_external_semaphore_handle_type_khr[]	Specifies the list of semaphore handle type properties (terminated with CL_SEMAPHORE_EXPORT_HANDLE_TYPES_LIST_END_KHR) that can be used to export the semaphore being created.

If CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR is not specified as part of *sema_props*, the semaphore object created by **clCreateSemaphoreWithPropertiesKHR** is by default associated with all devices in the *context*. For a multi-device context CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR must be specified in *sema_props*.

The properties used to create a semaphore from an external semaphore handle are described for the corresponding extensions.

errcode_ret returns an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

clCreateSemaphoreWithPropertiesKHR returns a valid semaphore object in an un-signaled state and *errcode_ret* is set to CL_SUCCESS if the function is executed successfully. Otherwise, it returns a NULL value with one of the following error values returned in *errcode_ret*:

- CL_INVALID_CONTEXT if *context* is not a valid context.
- CL_INVALID_PROPERTY if a property name in *sema_props* is not a supported property name, if the value specified for a supported property name is not valid, or if the same property name is specified more than once. Additionally, if *context* is a multiple device context and *sema_props* does not specify CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR.
- CL_INVALID_DEVICE if CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR is specified as part of *sema_props*, but it does not identify exactly one valid device; or if a device identified by CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR is not one of the devices within *context*.
- CL_INVALID_VALUE
  - if *sema_props* is NULL, or
  - if *sema_props* do not specify <property, value> pairs for minimum set of properties (i.e. CL_SEMAPHORE_TYPE_KHR) required for successful creation of a cl_semaphore_khr, or
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

- CL_INVALID_DEVICE if one or more devices identified by properties CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR cannot import the requested external semaphore handle type.
- CL_INVALID_VALUE if more than one semaphore handle type is specified in the CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR list.
- CL_INVALID_OPERATION If *props_list* specifies a cl_external_semaphore_handle_type_khr followed by a handle as well as CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR. Exporting a semaphore handle from a semaphore that was created by importing an external semaphore handle is not permitted.
- CL_INVALID_PROPERTY if *sema_props* includes more than one external semaphore handle.

5.13.3. Exporting Semaphore External Handles

Export operations have the same transference as the specified handle type's import operations. Additionally, exporting a semaphore payload to a handle with copy transference has the same side effects on the source semaphore's payload as executing a semaphore wait operation.

Please refer to handle specific documentation for more details on transference requirements per handle type. To export an external handle from a semaphore, call the function

```
// Provided by cl_khr_external_semaphore
cl_int clGetSemaphoreHandleForTypeKHR(
    cl_semaphore_khr sema_object,
    cl_device_id device,
    cl_external_semaphore_handle_type_khr handle_type,
    size_t handle_size,
    void* handle_ptr,
    size_t* handle_size_ret);
```

clGetSemaphoreHandleForTypeKHR is provided by the cl_khr_external_semaphore extension.

- *sema_object* specifies a valid semaphore object with exportable properties.
- *device* specifies a valid device for which a semaphore handle is being requested.
- *handle_type* specifies the type of semaphore handle that should be returned for this exportable *sema_object*, and must be one of the values specified when *sema_object* was created.
- *handle_ptr* is a pointer to memory where the exported external handle is returned. If *handle_ptr* is NULL, it is ignored.
- *handle_size* specifies the size in bytes of memory pointed to by *handle_ptr*. This size must be greater than or equal to the size of the handle type specified by *handle_type*. If *handle_ptr* is NULL, it is ignored.
- *handle_size_ret* returns the actual size in bytes for the external handle. If *handle_size_ret* is NULL, it is ignored.

clGetSemaphoreHandleForTypeKHR returns CL_SUCCESS if the semaphore handle is queried successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_SEMAPHORE_KHR if *sema_object* is not a valid semaphore.
- CL_INVALID_DEVICE
  - if *device* is not a valid device, or
  - if *sema_object* belongs to a context that is not associated with *device*, or
  - if *sema_object* can not be shared with *device*.
- CL_INVALID_VALUE if the requested external semaphore handle type was not specified when *sema_object* was created.
- CL_INVALID_VALUE if the size in bytes specified by *handle_size* is less than size of the requested handle and *handle_ptr* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.13.4. Importing Semaphore External Handles

Applications can import a semaphore payload by creating a semaphore from an external handle. The implementation must perform the import operation by either referencing or copying the payload referred to by the specified external semaphore handle, depending on the handle's type. When using handle types with reference transference, importing a payload to a semaphore adds the semaphore to the set of all semaphores sharing that payload. This set includes the semaphore from which the payload was exported. Semaphore signaling and waiting operations performed on any semaphore in the set must behave as if the set were a single semaphore. Importing a payload using handle types with copy transference creates a duplicate copy of the payload at the time of import, but makes no further reference to it. Semaphore signaling and waiting operations performed on the target of copy imports must not affect any other semaphore or payload.

Please refer to handle specific documentation for more details on transference requirements per handle type.

5.13.5. Descriptions of External Semaphore Handle Types

This section describes the external semaphore handle types that are added by related extensions.

Applications can import the same semaphore payload into multiple OpenCL contexts, into the same context from which it was exported, and multiple times into a given OpenCL context. In all cases, each import operation must create a distinct semaphore object.

5.13.5.1. File Descriptor Handle Types

The cl_khr_external_semaphore_opaque_fd extension extends cl_external_semaphore_handle_type_khr to support the following new types of handles, and adds as a property that may be specified when creating a semaphore from an external handle:

- CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR specifies a POSIX file descriptor handle that has only limited valid usage outside of OpenCL and other compatible APIs. It must be compatible with the POSIX

system calls dup, dup2, close, and the non-standard system call dup3. Additionally, it must be transportable over a socket using an SCM_RIGHTS control message. It owns a reference to the underlying synchronization primitive represented by its semaphore object.

The cl_khr_external_semaphore_sync_fd extension extends cl_external_semaphore_handle_type_khr to support the following new types of handles, and adds as a property that may be specified when creating a semaphore from an external handle:

- CL_SEMAPHORE_HANDLE_SYNC_FD_KHR specifies a POSIX file descriptor handle to a Linux Sync File or Android Fence object. It can be used with any native API accepting a valid sync file or fence as input. It owns a reference to the underlying synchronization primitive associated with the file descriptor. Implementations which support importing this handle type must accept any type of sync or fence FD supported by the native system they are running on.

The special value -1 for fd is treated like a valid sync file descriptor referring to an object that has already signaled. The import operation will succeed and the semaphore will have a temporarily imported payload as if a valid file descriptor had been provided.

Note: This special behavior for importing an invalid sync file descriptor allows easier interoperability with other system APIs which use the convention that an invalid sync file descriptor represents work that has already completed and does not need to be waited for. It is consistent with the option for implementations to return a -1 file descriptor when exporting a CL_SEMAPHORE_HANDLE_SYNC_FD_KHR from a cl_semaphore_khr which is signaled.

Table 54. Transference Properties for File Descriptor Handles

Handle Type	Transference
CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR provided by the cl_khr_external_semaphore_opaque_fd extension.	Reference
CL_SEMAPHORE_HANDLE_SYNC_FD_KHR provided by the cl_khr_external_semaphore_sync_fd extension.	Copy

Importing a semaphore payload from a file descriptor transfers ownership of the file descriptor from the application to the OpenCL implementation. The application must not perform any operations on the file descriptor after a successful import.

To re-import a handle of type CL_SEMAPHORE_HANDLE_SYNC_FD_KHR into an existing semaphore, call the function:

```
// Provided by cl_khr_external_semaphore_sync_fd
cl_int clReImportSemaphoreSyncFdKHR(
    cl_semaphore_khr sema_object,
    cl_semaphore_reimport_properties_khr* reimport_props,
    int fd);
```

- *sema_object* specifies a valid semaphore object with importable properties.
- *reimport_props* is an optional list of properties that affect the re-import behavior. The list is terminated with the special property `0`. If no properties are required, *reimport_props* may be `NULL`. This extension does not define any optional properties.
- *fd* specifies an external file descriptor handle to import

Calling **clReImportSemaphoreSyncFdKHR** is equivalent to destroying *sema_object* and re-creating it with the original *sema_props* from **clCreateSemaphoreWithPropertiesKHR**, except a handle specified by *fd* will be imported. The semaphore *sema_object* must have originally imported an external handle of type `CL_SEMAPHORE_HANDLE_SYNC_FD_KHR`.

clReImportSemaphoreSyncFdKHR returns `CL_SUCCESS` if the semaphore handle is re-imported successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SEMAPHORE_KHR`
  - if *sema_object* is not a valid semaphore
- `CL_INVALID_SEMAPHORE_KHR` if a `CL_SEMAPHORE_HANDLE_SYNC_FD_KHR` handle was not imported when *sema_object* was created.
- `CL_INVALID_VALUE` if *fd* is invalid.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.

5.13.5.2. NT Handle Types

The `cl_khr_external_semaphore_dx_fence` extension extends `cl_external_semaphore_handle_type_khr` to support the following new types of handles, and adds as a property that may be specified when creating a semaphore from an external handle:

- `CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR` specifies an NT handle returned by `ID3D12Device::CreateSharedHandle` referring to a Direct3D 12 fence, or `ID3D11Fence::CreateSharedHandle` referring to a Direct3D 11 fence. It owns a reference to the underlying synchronization primitive associated with the Direct3D fence.

When waiting on semaphores using **clEnqueueWaitSemaphoresKHR** or signaling semaphores using **clEnqueueSignalSemaphoresKHR**, the semaphore payload must be provided for semaphores created from `CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR`.

- If *sema_objects* list has a mix of semaphores obtained from `CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR` and other handle types, then the *sema_payload_list* should point to a list of *num_sema_objects* payload values for each semaphore in *sema_objects*. However, the payload values corresponding to semaphores that do not require a payload will be ignored.

clEnqueueWaitSemaphoresKHR and **clEnqueueSignalSemaphoresKHR** may return `CL_INVALID_VALUE` if *sema_objects* list has one or more semaphores obtained from `CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR` and *sema_payload_list* is NULL.

The cl_khr_external_semaphore_win32 extension extends cl_external_semaphore_handle_type_khr to support the following new types of handles, and adds as a property that may be specified when creating a semaphore from an external handle:

- CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KHR specifies an NT handle that has only limited valid usage outside of OpenCL and other compatible APIs. It must be compatible with the functions DuplicateHandle, CloseHandle, CompareObjectHandles, GetHandleInformation, and SetHandleInformation. It owns a reference to the underlying synchronization primitive represented by its semaphore object.
- CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KMT_KHR specifies a global share handle that has only limited valid usage outside of OpenCL and other compatible APIs. It is not compatible with any native APIs. It does not own a reference to the underlying synchronization primitive represented by its semaphore object, and will therefore become invalid when all semaphore objects associated with it are destroyed.
- CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_NAME_KHR specifies an NT handle name that has only limited valid usage outside of OpenCL and other compatible APIs. NT handle name is a null-terminated UTF-16 string naming the payload to import. It must be compatible with the functions DuplicateHandle, CloseHandle, CompareObjectHandles, GetHandleInformation, and SetHandleInformation. It owns a reference to the underlying synchronization primitive represented by its semaphore object.

Table 55. Transference Properties for NT Handle Types

Handle Type	**Transference**
CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR provided by the cl_khr_external_semaphore_dx_fence extension.	Reference
CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KHR provided by the cl_khr_external_semaphore_win32 extension.	Reference
CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KMT_KHR provided by the cl_khr_external_semaphore_win32 extension.	Reference
CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_NAME_KHR provided by the cl_khr_external_semaphore_win32 extension.	Reference

Importing a semaphore payload from Windows handles does not transfer ownership of the handle to the OpenCL implementation. For handle types defined as NT handles, the application must release ownership using the CloseHandle system call when the handle is no longer needed.

5.13.6. Waiting On and Signaling Semaphores

To enqueue a command to wait on a set of semaphores, call the function

```
// Provided by cl_khr_semaphore
cl_int clEnqueueWaitSemaphoresKHR(
    cl_command_queue command_queue,
    cl_uint num_sema_objects,
    const cl_semaphore_khr* sema_objects,
    const cl_semaphore_payload_khr* sema_payload_list,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueWaitSemaphoresKHR is provided by the `cl_khr_semaphore` extension.

- *command_queue* specifies a valid command-queue.
- *num_sema_objects* specifies the number of semaphore objects to wait on.
- *sema_objects* points to the list of semaphore objects to wait on. The length of the list must be at least *num_sema_objects*.
- *sema_payload_list* points to the list of values of type `cl_semaphore_payload_khr` containing valid semaphore payload values to wait on. This can be set to `NULL` or will be ignored if no semaphores in the list of *sema_objects* require a payload.
- *num_events_in_wait_list* specifies the number of events in *event_wait_list*.
- *event_wait_list* specifies list of events that need to complete before **clEnqueueWaitSemaphoresKHR** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueWaitSemaphoresKHR** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and that associated with *command_queue* must be the same.
- *event* returns an event object that identifies this particular command and can be used to query or queue a wait for this particular command to complete. *event* can be `NULL`, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

The semaphore wait command waits for a list of events to complete and a list of semaphore objects to become signaled. The semaphore wait command returns an *event* which can be waited on to ensure that all events in the *event_wait_list* have completed and all semaphores in *sema_objects* have been signaled. **clEnqueueWaitSemaphoresKHR** will not return until the binary semaphores in *sema_objects* are in a state that makes them safe to re-signal. If necessary, implementations may block in **clEnqueueWaitSemaphoresKHR** to ensure the correct state of semaphores when returning. There are no implications from this behavior for the state of *event* or the events in *event_wait_list* when **clEnqueueWaitSemaphoresKHR** returns. Waiting on the same binary semaphore twice without an interleaving signal may lead to undefined behavior.

When *command_queue* is an out-of-order command-queue there are no implicit dependencies between the semaphore wait command and commands enqueued into the command-queue after the semaphore wait command. If such dependencies are required, applications may enqueue a command-queue barrier after the semaphore wait command, to explicitly add dependencies between the semaphore wait command and subsequent commands.

clEnqueueWaitSemaphoresKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid command-queue, or
  - if the device associated with *command_queue* is not same as one of the devices specified by `CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR` at the time of creating one or more of *sema_objects*, or
  - if one or more of *sema_objects* belong to a context that does not contain a device associated with *command_queue*.
- `CL_INVALID_VALUE` if *num_sema_objects* is 0.
- `CL_INVALID_SEMAPHORE_KHR` if any of the semaphore objects specified by *sema_objects* is not valid.
- `CL_INVALID_CONTEXT` if the context associated with *command_queue* and any of the semaphore objects in *sema_objects* are not the same, or if the context associated with *command_queue* and that associated with events in *event_wait_list* are not the same.
- `CL_INVALID_VALUE` if any of the semaphore objects specified by *sema_objects* requires a semaphore payload and *sema_payload_list* is `NULL`.
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is not 0, or
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or
  - if event objects in *event_wait_list* are not valid events.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST` if the execution status of any of the events in *event_wait_list* is a negative integer value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a command to signal a set of semaphores, call the function

```
// Provided by cl_khr_semaphore
cl_int clEnqueueSignalSemaphoresKHR(
    cl_command_queue command_queue,
    cl_uint num_sema_objects,
    const cl_semaphore_khr* sema_objects,
    const cl_semaphore_payload_khr* sema_payload_list,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueSignalSemaphoresKHR is provided by the `cl_khr_semaphore` extension.

- *command_queue* specifies a valid command-queue.
- *num_sema_objects* specifies the number of semaphore objects to signal.
- *sema_objects* points to the list of semaphore objects to signal. The length of the list must be at least *num_sema_objects*.
- *sema_payload_list* points to the list of values of type `cl_semaphore_payload_khr` containing semaphore payload values to signal. This can be set to `NULL` or will be ignored if no semaphores in the list of *sema_objects* require a payload.
- *num_events_in_wait_list* specifies the number of events in
- *event_wait_list* points to the list of events that need to complete before **clEnqueueSignalSemaphoresKHR** can be executed. If *event_wait_list* is `NULL`, then **clEnqueueSignalSemaphoresKHR** does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If *event_wait_list* is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and that associated with *command_queue* must be the same.

  event returns an event object that identifies this particular command and can be used to query or queue a wait for this particular command to complete. *event* can be `NULL`, in which case it will not be possible for the application to query the status of this command or queue a wait for this command to complete.

The semaphore signal command waits for a list of events to complete and then signals a list of semaphore objects. The semaphore signal command returns an *event* which can be waited on to ensure that all events in the *event_wait_list* have completed and all semaphores in *sema_objects* have been signaled. The successful completion of the event generated by **clEnqueueSignalSemaphoresKHR** called on one or more semaphore objects of type `CL_SEMAPHORE_TYPE_BINARY_KHR` changes the state of the corresponding semaphore objects to signaled. **clEnqueueSignalSemaphoresKHR** will not return until the binary semaphores in *sema_objects* are in a state that makes them safe to wait on again. If necessary, implementations may block in **clEnqueueSignalSemaphoresKHR** to ensure the correct state of semaphores when returning. There are no implications from this behavior for the state of *event* or the events in *event_wait_list* when **clEnqueueSignalSemaphoresKHR** returns. Signaling the same binary semaphore twice without an interleaving wait may lead to undefined behavior.

When *command_queue* is an out-of-order command-queue there are no implicit dependencies between commands enqueued into the command-queue before the semaphore signal command and the semaphore signal command. If such dependencies are required, applications may enqueue a command-queue barrier before the semaphore signal command, to explicitly add dependencies between the preceding commands and the semaphore signal command.

clEnqueueSignalSemaphoresKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE`
  - if *command_queue* is not a valid command-queue, or
  - if the device associated with *command_queue* is not same as one of the devices specified by `CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR` at the time of creating one or more of *sema_objects*, or
  - if one or more of *sema_objects* belong to a context that does not contain a device associated with *command_queue*.
- `CL_INVALID_VALUE` if *num_sema_objects* is 0.
- `CL_INVALID_SEMAPHORE_KHR` if any of the semaphore objects specified by *sema_objects* is not valid.
- `CL_INVALID_CONTEXT` if the context associated with *command_queue* and any of the semaphore objects in *sema_objects* are not the same, or if the context associated with *command_queue* and that associated with events in *event_wait_list* are not the same.
- `CL_INVALID_VALUE` if any of the semaphore objects specified by *sema_objects* requires a semaphore payload and *sema_payload_list* is `NULL`.
- `CL_INVALID_EVENT_WAIT_LIST`
  - if *event_wait_list* is `NULL` and *num_events_in_wait_list* is not 0, or
  - if *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or
  - if event objects in *event_wait_list* are not valid events.
- `CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST` if the execution status of any of the events in *event_wait_list* is a negative integer value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.13.7. Retaining and Releasing Semaphores

To release a semaphore object, call the function

```
// Provided by cl_khr_semaphore
cl_int clReleaseSemaphoreKHR(
    cl_semaphore_khr sema_object);
```

clReleaseSemaphoreKHR is provided by the `cl_khr_semaphore` extension.

- *sema_object* specifies the semaphore object to be released.

The *sema_object* reference count is decremented.

clReleaseSemaphoreKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SEMAPHORE_KHR` if *sema_object* is not a valid semaphore object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

After the reference count becomes zero and commands queued for execution on a command-queue(s) that use *sema_object* have finished, the semaphore object is deleted. Using this function to release a reference that was not obtained by creating the object via **clCreateSemaphoreWithPropertiesKHR** or by calling **clRetainSemaphoreKHR** causes undefined behavior.

To retain a semaphore object, call the function

```
// Provided by cl_khr_semaphore
cl_int clRetainSemaphoreKHR(
    cl_semaphore_khr sema_object);
```

clRetainSemaphoreKHR is provided by the `cl_khr_semaphore` extension.

- *sema_object* specifies the semaphore object to be retained.

clRetainSemaphoreKHR increments the reference count of *sema_object*.

clRetainSemaphoreKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_SEMAPHORE_KHR` if *sema_object* is not a valid semaphore object.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.13.8. Semaphore Queries

To query information about a semaphore object, call the function

```
// Provided by cl_khr_semaphore
cl_int clGetSemaphoreInfoKHR(
    cl_semaphore_khr sema_object,
    cl_semaphore_info_khr param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetSemaphoreInfoKHR is provided by the `cl_khr_semaphore` extension.

- *sema_object* specifies the semaphore object being queried.
- *param_name* is a constant that specifies the semaphore information to query, and must be one of the values shown in the Semaphore Queries table.
- *param_value* is a pointer to memory where the result of the query is returned as described in the Semaphore Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Semaphore Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 56. List of parameter names supported by **clGetSemaphoreInfoKHR**

Semaphore Info	Return Type	Description
`CL_SEMAPHORE_CONTEXT_KHR`	`cl_context`	Returns the context specified when the semaphore is created.
`CL_SEMAPHORE_REFERENCE_COUNT_KHR` [20]	`cl_uint`	Returns the semaphore reference count.
`CL_SEMAPHORE_PROPERTIES_KHR`	`cl_semaphore_properties_khr[]`	Return the properties argument specified in **clCreateSemaphoreWithPropertiesKHR**. The implementation must return the values specified in the properties argument in the same order and without including additional properties.
`CL_SEMAPHORE_TYPE_KHR`	`cl_semaphore_type_khr`	Returns the semaphore type.
`CL_SEMAPHORE_PAYLOAD_KHR`	`cl_semaphore_payload_khr`	Returns the semaphore payload value. For semaphores of type `CL_SEMAPHORE_TYPE_BINARY_KHR` the payload value returned will be `0` if the semaphore is in an un-signaled state, and `1` if it is in a signaled state.
`CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR`	`cl_device_id[]`	Returns the list of OpenCL devices the semaphore is associated with.

Semaphore Info	Return Type	Description
CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR	cl_external_semaphore_handle_type_khr[]	Returns the list of external semaphore handle types that may be used for exporting. The size of this query may be 0 indicating that this semaphore does not support any handle types for exporting.
CL_SEMAPHORE_EXPORTABLE_KHR	cl_bool[]	Returns CL_TRUE if the semaphore is exportable and CL_FALSE otherwise.

clGetSemaphoreInfoKHR returns CL_SUCCESS if the information is queried successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_SEMAPHORE_KHR if *sema_object* is not a valid semaphore.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Semaphore Queries table and *param_value* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.14. Out-of-Order Execution of Kernels and Memory Object Commands

The OpenCL functions that are submitted to a command-queue are enqueued in the order the calls are made but can be configured to execute in-order or out-of-order. The *properties* argument in **clCreateCommandQueueWithProperties** or **clCreateCommandQueue** can be used to specify the execution order.

If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is not set, the commands enqueued to a command-queue execute in-order. For example, if an application calls **clEnqueueNDRangeKernel** to execute kernel A followed by a **clEnqueueNDRangeKernel** to execute kernel B, the application can assume that kernel A finishes first and then kernel B is executed. If the memory objects output by kernel A are inputs to kernel B then kernel B will see the correct data in memory objects produced by execution of kernel A. If the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of a command-queue is set, then there is no guarantee that kernel A will finish before kernel B starts execution.

Applications can configure the commands enqueued to a command-queue to execute out-of-order by setting the CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE property of the command-queue. This can be specified when the command-queue is created. In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued. As there is no guarantee that kernels will be executed in-order, i.e. based on when the **clEnqueueNDRangeKernel** or **clEnqueueTask** calls are made within a command-queue, it is therefore possible that an earlier **clEnqueueNDRangeKernel** call to execute kernel A identified by

event A may execute and/or finish later than a **clEnqueueNDRangeKernel** call to execute kernel B which was called by the application at a later point in time. To guarantee a specific order of execution of kernels, a wait on a particular event (in this case event A) can be used. The wait for event A can be specified in the *event_wait_list* argument to **clEnqueueNDRangeKernel** for kernel B.

In addition, a marker (**clEnqueueMarker** or **clEnqueueMarkerWithWaitList**) or a barrier (**clEnqueueBarrier** or **clEnqueueBarrierWithWaitList**) command can be enqueued to the command-queue. The marker command ensures that previously enqueued commands identified by the list of events to wait for (or all previous commands) have finished. A barrier command is similar to a marker command, but additionally guarantees that no later-enqueued commands will execute until the waited-for commands have executed.

Similarly, commands to read, write, copy or map memory objects that are enqueued after **clEnqueueNDRangeKernel**, **clEnqueueTask** or **clEnqueueNativeKernel** commands are not guaranteed to wait for kernels scheduled for execution to have completed (if the `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` property is set). To ensure correct ordering of commands, the event object returned by **clEnqueueNDRangeKernel**, **clEnqueueTask** or **clEnqueueNativeKernel** can be used to enqueue a wait for event or a barrier command can be enqueued that must complete before reads or writes to the memory object(s) occur.

5.15. Profiling Operations on Memory Objects and Kernels

This section describes the profiling of OpenCL functions that are enqueued as commands to a command-queue. Profiling of OpenCL commands can be enabled by using a command-queue created with the `CL_QUEUE_PROFILING_ENABLE` flag set in the `CL_QUEUE_PROPERTIES` bitfield in the *properties* argument to **clCreateCommandQueueWithProperties**, or in the *properties* argument to **clCreateCommandQueue**. When profiling is enabled, the event objects that are created from enqueuing a command store a timestamp for each of their state transitions.

To return profiling information for a command associated with an event when profiling is enabled, call the function

```
// Provided by CL_VERSION_1_0
cl_int clGetEventProfilingInfo(
    cl_event event,
    cl_profiling_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *event* specifies the event object.
- *param_name* specifies the profiling data to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetEventProfilingInfo** is described in the Event Profiling Queries table.

- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Event Profiling Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 57. List of supported param_names by **clGetEventProfilingInfo**

Event Profiling Info	Return Type	Description
`CL_PROFILING_COMMAND_QUEUED`	`cl_ulong`	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event is enqueued in a command-queue by the host. If the `cl_khr_command_buffer_multi_device` extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, the host time when the command-buffer has been enqueued across the command-queues is used.
`CL_PROFILING_COMMAND_SUBMIT`	`cl_ulong`	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event that has been enqueued is submitted by the host to the device associated with the command-queue. If the `cl_khr_command_buffer_multi_device` extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, the host time is used when command-buffer commands have been submitted to any command-queue.
`CL_PROFILING_COMMAND_START`	`cl_ulong`	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event starts execution on the device. If the `cl_khr_command_buffer_multi_device` extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, the host time is used when any device starts executing a command-buffer command.

Event Profiling Info	Return Type	Description
CL_PROFILING_COMMAND_END	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event has finished execution on the device. If the cl_khr_command_buffer_multi_device extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, the host time is used when the last command-buffer command finishes execution on any device.
CL_PROFILING_COMMAND_COMPLETE missing before version 2.0.	cl_ulong	A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event and any child commands enqueued by this command on the device have finished execution. If the cl_khr_command_buffer_multi_device extension is supported, for events returned by a command-buffer enqueue operation to multiple command-queues, the host time is used when the command-buffer has completed execution across all command-queues.

The unsigned 64-bit values returned can be used to measure the time in nano-seconds consumed by OpenCL commands.

OpenCL devices are required to correctly track time across changes in device frequency and power states. The CL_DEVICE_PROFILING_TIMER_RESOLUTION specifies the resolution of the timer i.e. the number of nanoseconds elapsed before the timer is incremented.

If the cl_khr_command_buffer_multi_device extension is supported, and if no reliable device timer sources are available to inform the host side, or parallel runtime scheduling makes it impossible to identify a first/last command, then an implementation may fallback to reporting CL_PROFILING_COMMAND_SUBMIT and CL_PROFILING_COMMAND_COMPLETE for CL_PROFILING_COMMAND_START and CL_PROFILING_COMMAND_END respectively.

clGetEventProfilingInfo returns CL_SUCCESS if the function is executed successfully and the profiling information has been recorded. Otherwise, it returns one of the following errors:

- CL_INVALID_EVENT if *event* is a not a valid event object.
- CL_PROFILING_INFO_NOT_AVAILABLE if the CL_QUEUE_PROFILING_ENABLE flag is not set for the command-queue, if the execution status of the command identified by *event* is not CL_COMPLETE or if *event* is a user event object. Prior to OpenCL 3.0, implementations may return CL_PROFILING_INFO_NOT_AVAILABLE for an event created by **clEnqueueSVMFree**.

If the `cl_khr_command_buffer_multi_device` extension is supported, and if *event* was created from a call to **clEnqueueCommandBufferKHR**, `CL_PROFILING_INFO_NOT_AVAILABLE` is returned if all the queues passed do not have `CL_QUEUE_PROFILING_ENABLE` set.

- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Event Profiling Queries table and *param_value* is not a `NULL` value.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.16. Flush and Finish

To flush commands to a device, call the function

```
// Provided by CL_VERSION_1_0
cl_int clFlush(
    cl_command_queue command_queue);
```

- *command_queue* is the command-queue to flush.

All previously queued OpenCL commands in *command_queue* are issued to the device associated with *command_queue*. **clFlush** only guarantees that all queued commands to *command_queue* will eventually be submitted to the appropriate device. There is no guarantee that they will be complete after **clFlush** returns.

Any blocking commands queued in a command-queue and **clReleaseCommandQueue** perform an implicit flush of the command-queue. These blocking commands are **clEnqueueReadBuffer**, **clEnqueueReadBufferRect**, **clEnqueueReadImage**, with *blocking_read* set to `CL_TRUE`; **clEnqueueWriteBuffer**, **clEnqueueWriteBufferRect**, **clEnqueueWriteImage** with *blocking_write* set to `CL_TRUE`; **clEnqueueMapBuffer**, **clEnqueueMapImage** with *blocking_map* set to `CL_TRUE`; **clEnqueueSVMMemcpy** with *blocking_copy* set to `CL_TRUE`; **clEnqueueSVMMap** with *blocking_map* set to `CL_TRUE` or **clWaitForEvents**.

To use event objects that refer to commands enqueued in a command-queue as event objects to wait on by commands enqueued in a different command-queue, the application must call a **clFlush** or any blocking commands that perform an implicit flush of the command-queue where the commands that refer to these event objects are enqueued.

clFlush returns `CL_SUCCESS` if the function call was executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL

implementation on the host.

To wait for completion of commands on a device, call the function

```
// Provided by CL_VERSION_1_0
cl_int clFinish(
    cl_command_queue command_queue);
```

- *command_queue* is the command-queue to wait for.

All previously queued OpenCL commands in *command_queue* are issued to the associated device, and the function blocks until all previously queued commands have completed. **clFinish** does not return until all previously queued commands in *command_queue* have been processed and completed. **clFinish** is also a synchronization point.

clFinish returns `CL_SUCCESS` if the function call was executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if *command_queue* is not a valid host command-queue.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.17. Command-Buffers

A *command-buffer* object represents a series of operations to be enqueued on one or more command-queues without any application code interaction. Grouping the operations together allows efficient enqueuing of repetitive operations, as well as enabling driver optimizations.

Upon creation a command-buffer is in the Recording state. In order for the command-buffer to be enqueued it must first be finalized using **clFinalizeCommandBufferKHR**, after which no further commands can be recorded. A command-buffer is enqueued for execution on command-queues with a call to **clEnqueueCommandBufferKHR**. It is always valid to call **clEnqueueCommandBufferKHR** with a command-buffer that has previously been enqueued, provided the call doesn't violate the definition of simultaneous use.

Command-buffers are created using an ordered list of command-queues that commands are recorded to and execute on by default. All these queue objects must share the same context, but may be associated with different devices when the `cl_khr_command_buffer_multi_device` extension is supported.

When constructing a command-buffer by appending commands, the queue parameter passed for the command being created is used to set the device with which the command will be associated with, and also inform the scheduling of the command. If the queue is an in-order queue, then an additional dependency is created on the last command appended to the command-buffer using the same queue parameter. If the queue is an out-of-order queue, then no extra dependencies on

previous commands using the same queue are created. All queue properties other than CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE are ignored for the purposes of command creation, with the exception of any vendor extension defined queue properties that explicitly define semantics for this purpose.

When enqueuing a command-buffer, a list of command-queues to execute the command-buffer on can be passed by the user, otherwise the command-queues set on command-buffer creation are used by default for execution. A user passed list may contain different command-queues, provided for each element the substitute command-queue matches the device and context of the command-queue used on command-buffer creation. Each command-queue in the enqueue list must also have the minimum properties defined by CL_DEVICE_COMMAND_BUFFER_REQUIRED_QUEUE_PROPERTIES_KHR and no properties which are not reported by CL_DEVICE_COMMAND_BUFFER_SUPPORTED_QUEUE_PROPERTIES_KHR. These queue properties have the same execution semantics for **clEnqueueCommandBufferKHR** as other operations enqueued to the queue.

While constructing a command-buffer it is valid for the user to interleave calls to the same queue which create commands, such as **clCommandNDRangeKernelKHR**, with queue submission calls, such as **clEnqueueNDRangeKernel** or **clEnqueueCommandBufferKHR**. That is, there is no effect on queue state from recording commands. The purpose of the queue parameter is to define the device and properties of the command, which are constant queries on the queue object.

A command-buffer object should increment the reference count of attached OpenCL objects such as queues, buffers, images, and kernels referenced in commands recorded to the command-buffer. This enables correct behavior of the command-buffer when its attached objects have been released. On destruction of the command-buffer it should decrement these reference counts, allowing the attached objects to be freed if appropriate.

A command-buffer object does not update the reference count of objects set as arguments on kernels recorded into the command-buffer. This is consistent with the reference counting behavior of **clSetKernelArg**.

Applications should ensure that objects passed as arguments to kernels recorded to a command-buffer are not deleted until the command-buffer has been released. Undefined behavior may result from the failure to follow this usage requirement for all the command-buffers an object is used as a kernel argument in.

If using layered extension cl_khr_command_buffer_mutable_dispatch, see related note on safe usage.

Simultaneous use is defined using the *prerequisite* terminology from the execution model. Simultaneous use is an optional feature for devices to support concurrent executions of a command-buffer which have been updated between submissions. A command-buffer must be created with CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR to avoid undefined behavior if a simultaneous use pattern occurs.

Simultaneous Use

When a command-buffer is submitted for execution without a prerequisite on all the previous submissions of the same command-buffer which are not in the CL_COMPLETE state.

An example of simultaneous use would be two submissions of the same command-buffer to a single out-of-order queue, without any events or barriers used to express a dependency between the two enqueue calls. Using a single in-order queue, events, or barriers to express dependencies between submissions of the same command-buffer would each be ways to avoid simultaneous use.

5.17.1. Command-Buffers and Multiple Devices

If the `cl_khr_command_buffer_multi_device` extension is supported, a command-buffer can contain commands recorded to the queues of different devices if a vendor provides support for inter-device `cl_sync_point_khr` synchronization. This feature is reported either through `CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR`, which informs the user what devices can synchronize with each other natively on the device-side, or through `CL_COMMAND_BUFFER_PLATFORM_UNIVERSAL_SYNC_KHR`, which allows synchronization between all devices in a platform, falling back to host-side synchronization when device-side synchronization is not available. These two mechanisms are referred to as **device-side sync** and **universal sync** respectively.

If these mechanisms do not report that more than one device can be used in a command-buffer, it will still be possible to perform multiple queue recording in a command-buffer if the `CL_COMMAND_BUFFER_CAPABILITY_MULTIPLE_QUEUE_KHR` capability is reported for a device. However, with this capability all the queues commands are recorded to must target the same device.

Commands recorded to different command-queues in the same command-buffer may be executed concurrently to each other unless synchronized explicitly with sync-points, barrier commands, or in-order queue implicit dependencies. Ordering of other commands submitted to the same command-queues as used to enqueue a command-buffer is the responsibility of the programmer. A command-buffer enqueue spanning multiple queues can return an event to use for synchronization, which will complete once all commands in the command-buffer have completed. If ordering restrictions are required, this event (or command-queue barriers) may be used by the user to synchronize the command-buffer enqueue with regular commands, or another command-buffer enqueue.

5.17.2. Command-Buffer Lifecycle

A command-buffer is created in the recording state and transitions to the executable state when finalized, at which point it cannot move back to the recording state.

Recording

Initial state of a command-buffer on creation, where commands can be recorded to the command-buffer.

Finalized

State after command recording has finished with **clFinalizeCommandBufferKHR**, but there is at least one command for which not all arguments or parameters have been set (this is a valid state only for mutable command buffers).

Executable

State after command recording has finished with **clFinalizeCommandBufferKHR** and all the

arguments and parameters of all commands have been set. In this state the command-buffer may be enqueued.

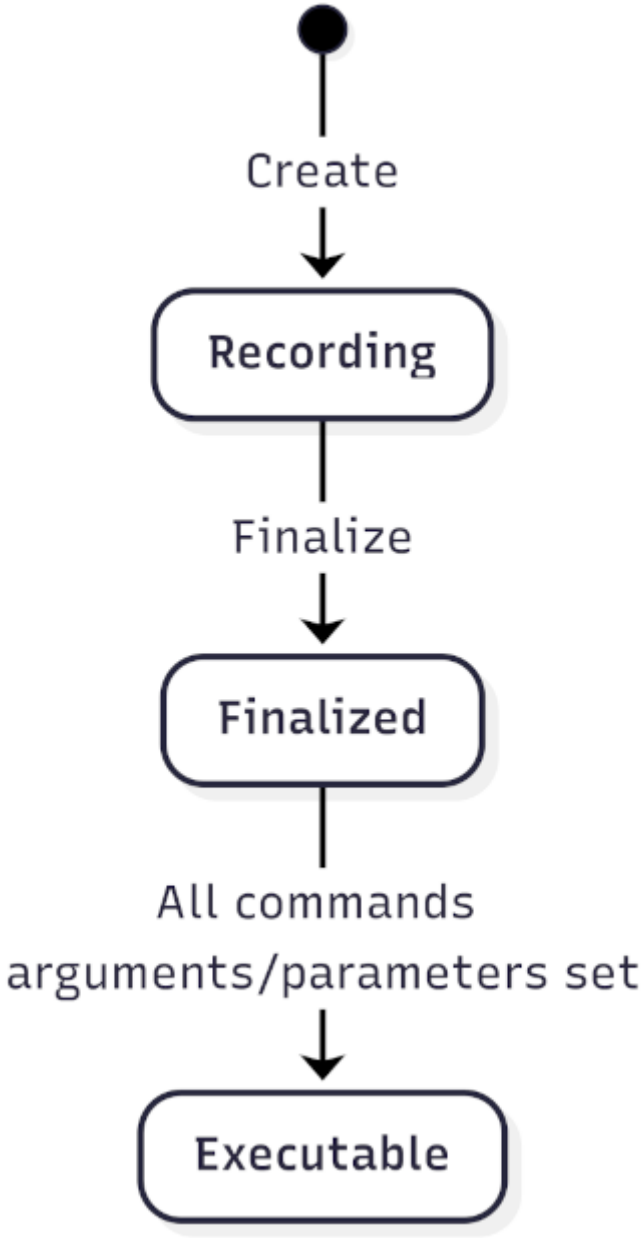

Figure 5. Lifecycle of a command-buffer.

5.17.3. Creating Command-Buffer Objects

To create a command-buffer that can record commands to the specified queues, call the function

```
// Provided by cl_khr_command_buffer
cl_command_buffer_khr clCreateCommandBufferKHR(
    cl_uint num_queues,
    const cl_command_queue* queues,
    const cl_command_buffer_properties_khr* properties,
    cl_int* errcode_ret);
```

clCreateCommandBufferKHR is provided by the `cl_khr_command_buffer`

extension.

- *num_queues* is the number of command-queues listed in *queues*. If the `cl_khr_command_buffer_multi_device` extension is not supported, this **must** be one.
- *queues* is a pointer to a list of command-queues that the command-buffer commands will be recorded to. *queues* must be a non-`NULL` value and the length of the list equal to *num_queues*.
- *properties* specifies a list of properties for the command-buffer and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The list of supported properties is described in the table below. If a supported property and its value is not specified in properties, its default value will be used. *properties* can be `NULL` in which case the default values for supported command-buffer properties will be used.

Table 58. **clCreateCommandBufferKHR** *properties*

Recording Properties	Property Value	Description
CL_COMMAND_BUFFER_FLAGS_KHR provided by the cl_khr_command_buffer extension.	cl_command_buffer_flags_khr	This is a bitfield and can be set to a combination of the following values: CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR - Allow multiple instances of the command-buffer to be scheduled for execution on the device in a usage pattern that exhibits simultaneous use. If set, devices must support CL_COMMAND_BUFFER_CAPABILITY_SIMULTANEOUS_USE_KHR. provided by the cl_khr_command_buffer_mutable_dispatch extension. CL_COMMAND_BUFFER_DEVICE_SIDE_SYNC_KHR - All commands in the command-buffer must use native synchronization, as reported by CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR. This can be used as a safeguard for performant applications that do not want to accidentally fallback to host synchronization when passing multiple queues. provided by the cl_khr_command_buffer_multi_device extension. CL_COMMAND_BUFFER_MUTABLE_KHR - Enables modification of the command-buffer, by default command-buffers are immutable. If set, commands in the command-buffer may be updated via **clUpdateMutableCommandsKHR**. provided by the cl_khr_command_buffer_mutable_dispatch extension. The default value of this property is 0.

Recording Properties	Property Value	Description
CL_COMMAND_BUFFER_MUTABLE_DISPATCH_ASSERTS_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	cl_mutable_dispatch_asserts_khr	This is a bitfield and can be set to a combination of the following values: CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR - An assertion by the user that the number of work-groups of any ND-range kernel recorded in this command buffer will not be updated beyond the number defined when the ND-range kernel was recorded. If the user's update to the values of *local_work_size* and/or *global_work_size* result in an increase in the number of work-groups in the ND-range over the number specified when the ND-range kernel was recorded, the behavior is undefined. provided by the cl_khr_command_buffer_mutable_dispatch extension.

- *errcode_ret* will return an appropriate error code. If *errcode_ret* is NULL, no error code is returned.

Table 59. Summary of multi-device command-buffer creation configurations

All Devices Associated With Queues can Device-side Sync	Platform Supports Universal Sync	Condition	Result
Yes	Yes or No	Any device does not support the multi-queue capability, and has more than one queue targeting it	Error - CL_INCOMPATIBLE_COMMAND_QUEUE_KHR
		User sets CL_COMMAND_BUFFER_DEVICE_SIDE_SYNC_KHR flag	OK
		Otherwise	OK

All Devices Associated With `Queues` can Device-side Sync	Platform Supports Universal Sync	Condition	Result
No	Yes	Any device does not support the multi-queue capability, and has more than one queue targeting it	Error - `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR`
		User sets `CL_COMMAND_BUFFER_DEVICE_SIDE_SYNC_KHR` flag	Error - `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR`
		Otherwise	OK - May be performance implications when synchronizing commands between devices without device-side sync support.
No	No	Always	Error - `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR`

clCreateCommandBufferKHR returns a valid non-zero command-buffer and *errcode_ret* is set to `CL_SUCCESS` if the command-buffer is created successfully. Otherwise, it returns a `NULL` value with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_COMMAND_QUEUE` if any command-queue in *queues* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if all the command-queues in *queues* do not have the same OpenCL context.
- `CL_INVALID_VALUE` if the `cl_khr_command_buffer_multi_device` extension is supported and *num_queues* is zero, or if the `cl_khr_command_buffer_multi_device` extension is not supported and *num_queues* is not one.
- `CL_INVALID_VALUE` if *queues* is `NULL`.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid, or if the same property name is specified more than once.
- `CL_INVALID_PROPERTY` if values specified in *properties* are valid but are not supported by all the devices associated with command-queues in *queues*.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

If the `cl_khr_command_buffer_multi_device` extension is supported:

- `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR` if *queues* includes more than one command-queue associated with a device that does not support capability `CL_COMMAND_BUFFER_CAPABILITY_MULTIPLE_QUEUE_KHR`.
- `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR` if the `CL_COMMAND_BUFFER_DEVICE_SIDE_SYNC_KHR` flag is set, and

any device associated with a command-queue in *queues* cannot natively synchronize with the other devices associated with *queues* as reported by `CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR`.

- `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR` if the platform does not support the `CL_COMMAND_BUFFER_PLATFORM_UNIVERSAL_SYNC_KHR` capability, and any device associated with a command-queue in *queues* cannot natively synchronize with the other devices associated with *queues* as reported by `CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR`.

To increment a command-buffer's reference count, call the function

```
// Provided by cl_khr_command_buffer
cl_int clRetainCommandBufferKHR(
    cl_command_buffer_khr command_buffer);
```

clRetainCommandBufferKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* specifies the command-buffer to retain.

clRetainCommandBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To decrement a command-buffer's reference count, call the function

```
// Provided by cl_khr_command_buffer
cl_int clReleaseCommandBufferKHR(
    cl_command_buffer_khr command_buffer);
```

clReleaseCommandBufferKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* specifies the command-buffer to release.

After the *command_buffer* reference count becomes zero and has finished execution, the command-buffer is deleted.

clReleaseCommandBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.

- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.17.4. Enqueuing a Command-Buffer

To finalize command recording ready for enqueuinga command-buffer on a command-queue, call the function

```
// Provided by cl_khr_command_buffer
cl_int clFinalizeCommandBufferKHR(
    cl_command_buffer_khr command_buffer);
```

clFinalizeCommandBufferKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.

clFinalizeCommandBufferKHR places the command-buffer either in the Finalized or Executable states where commands can no longer be recorded. If the new state is Executable, the command-buffer is ready to be enqueued.

clFinalizeCommandBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* is not in the Recording state.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To enqueue a command-buffer to execute on command-queues, call the function

```
// Provided by cl_khr_command_buffer
cl_int clEnqueueCommandBufferKHR(
    cl_uint num_queues,
    cl_command_queue* queues,
    cl_command_buffer_khr command_buffer,
    cl_uint num_events_in_wait_list,
    const cl_event* event_wait_list,
    cl_event* event);
```

clEnqueueCommandBufferKHR is provided by the `cl_khr_command_buffer`

extension.

- *num_queues* is the number of command-queues listed in *queues*.
- *queues* is a pointer to an ordered list of command-queues to execute the command-buffer on. *queues* can be `NULL`, in which case the default command-queues used on command-buffer creation are used and *num_queues* must be 0.
- *command_buffer* refers to a valid command-buffer object.
- *event_wait_list*, *num_events_in_wait_list* specify events that need to complete before this particular command can be executed. If *event_wait_list* is `NULL`, then this particular command does not wait on any event to complete. If *event_wait_list* is `NULL`, *num_events_in_wait_list* must be 0. If event_wait_list is not `NULL`, the list of events pointed to by *event_wait_list* must be valid and *num_events_in_wait_list* must be greater than 0. The events specified in *event_wait_list* act as synchronization points. The context associated with events in *event_wait_list* and command_queue must be the same. The memory associated with *event_wait_list* can be reused or freed after the function returns.
- *event* will return an event object that identifies this command and can be used to query for profiling information or queue a wait for this particular command to complete. *event* can be `NULL` in which case it will not be possible for the application to wait on this command or query it for profiling information.

clEnqueueCommandBufferKHR returns `CL_SUCCESS` if the command-buffer execution was successfully queued, or one of the errors below:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* is not in the Executable state.
- `CL_INVALID_VALUE` if *queues* is `NULL` and *num_queues* is > 0, or *queues* is not `NULL` and *num_queues* is 0.
- `CL_INVALID_VALUE` if *num_queues* is > 0 and not the same value as *num_queues* set on *command_buffer* creation.
- `CL_INVALID_COMMAND_QUEUE` if any element of *queues* is not a valid command-queue.
- `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR` if the properties of any command-queue in *queues* contains a property not specified by `CL_DEVICE_COMMAND_BUFFER_SUPPORTED_QUEUE_PROPERTIES_KHR`.
- `CL_INCOMPATIBLE_COMMAND_QUEUE_KHR` if the properties of any command-queue in *queues* does not contain the minimum properties specified by `CL_DEVICE_COMMAND_BUFFER_REQUIRED_QUEUE_PROPERTIES_KHR`.
- `CL_INVALID_DEVICE` if any element of *queues* does not have the same device as the command-queue set on *command_buffer* creation at the same list index.
- `CL_INVALID_CONTEXT` if any element of *queues* does not have the same context as the command-queue set on *command_buffer* creation at the same list index.
- `CL_INVALID_CONTEXT` if context associated with *command_buffer* and events in *event_wait_list* are not the same.
- `CL_OUT_OF_RESOURCES` if there is a failure to queue the execution instance of *command_buffer* on the command-queues because of insufficient resources needed to execute *command_buffer*.

- `CL_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL` and *num_events_in_wait_list* > 0, or *event_wait_list* is not `NULL` and *num_events_in_wait_list* is 0, or if event objects in *event_wait_list* are not valid events.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

Calling **clEnqueueCommandBufferKHR** in a usage pattern that exhbits simultaneous use when *command_buffer* was not created with the `CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR` flag results in undefined behavior.

5.17.5. Recording Commands to a Command-Buffer

To record a barrier operation used as a synchronization point, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandBarrierWithWaitListKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandBarrierWithWaitListKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in

sync_point_wait_list are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

If *sync_point_wait_list* is `NULL`, then this particular command waits until all previous recorded commands to *command_queue* have completed.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this barrier command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

clCommandBarrierWithWaitListKHR waits for either a list of synchronization-points to complete, or if the list is empty it waits for all commands previously recorded in *command_buffer* to complete before it completes. This command blocks command execution, that is, any following commands recorded after it do not execute until it completes.

clCommandBarrierWithWaitListKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.
- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.
- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

To record a command to copy from one buffer object to another, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandCopyBufferKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem src_buffer,
    cl_mem dst_buffer,
    size_t src_offset,
    size_t dst_offset,
    size_t size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandCopyBufferKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *src_buffer*, *dst_buffer*, *src_offset*, *dst_offset*, *size* refer to **clEnqueueCopyBuffer**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_buffer* or *dst_buffer* is created is ignored by **clCommandCopyBufferKHR**.

clCommandCopyBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueCopyBuffer** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer*, *src_buffer*, and *dst_buffer* is not the same.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to copy a rectangular region from a buffer object to another buffer object, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandCopyBufferRectKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem src_buffer,
    cl_mem dst_buffer,
    const size_t* src_origin,
    const size_t* dst_origin,
    const size_t* region,
    size_t src_row_pitch,
    size_t src_slice_pitch,
    size_t dst_row_pitch,
    size_t dst_slice_pitch,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandCopyBufferRectKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *src_origin*, *dst_origin*, *region*, *src_row_pitch*, *src_slice_pitch*, *dst_row_pitch*, *dst_slice_pitch* refer to **clEnqueueCopyBufferRect**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command.

Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be NULL in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not NULL, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.

- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be NULL.

clCommandCopyBufferRectKHR records a command to copy a 2D or 3D rectangular region from the buffer object identified by *src_buffer* to a 2D or 3D region in the buffer object identified by *dst_buffer*. Copying begins at the source offset and destination offset which are computed as described in the description for *src_origin* and *dst_origin*.

Each byte of the region's width is copied from the source offset to the destination offset. After copying each width, the source and destination offsets are incremented by their respective source and destination row pitches. After copying each 2D rectangle, the source and destination offsets are incremented by their respective source and destination slice pitches.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the cl_mem_flags argument value specified when *src_buffer* or *dst_buffer* is created is ignored by **clCommandCopyBufferRectKHR**.

clCommandCopyBufferRectKHR returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueCopyBufferRect** except:

CL_INVALID_COMMAND_QUEUE is replaced with:

- CL_INVALID_COMMAND_QUEUE if the cl_khr_command_buffer_multi_device extension is not supported and *command_queue* is not NULL.
- CL_INVALID_COMMAND_QUEUE if the cl_khr_command_buffer_multi_device extension is supported; and either *command_queue* is NULL and *command_buffer* was created with more than one queue, or *command_queue* is not NULL and not a command-queue listed on *command_buffer* creation.

CL_INVALID_CONTEXT is replaced with:

- CL_INVALID_CONTEXT if the context associated with *command_buffer*, *src_buffer*, and *dst_buffer* is not the same.
- CL_INVALID_CONTEXT if *command_queue* is not NULL, and the context associated with *command_queue* and *command_buffer* is not the same.

CL_INVALID_EVENT_WAIT_LIST is replaced with:

- CL_INVALID_SYNC_POINT_WAIT_LIST_KHR if *sync_point_wait_list* is NULL and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not NULL and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to copy a buffer object to an image object, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandCopyBufferToImageKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem src_buffer,
    cl_mem dst_image,
    size_t src_offset,
    const size_t* dst_origin,
    const size_t* region,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandCopyBufferToImageKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *src_buffer*, *dst_image*, *src_offset*, *dst_origin*, *region* refer to **clEnqueueCopyBufferToImage**
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in

sync_point_wait_list are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_buffer* or *dst_image* is created is ignored by **clCommandCopyBufferToImageKHR**.

clCommandCopyBufferToImageKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueCopyBufferToImage** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer*, *src_buffer*, and *dst_image* is not the same.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to copy between two image objects, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandCopyImageKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem src_image,
    cl_mem dst_image,
    const size_t* src_origin,
    const size_t* dst_origin,
    const size_t* region,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandCopyImageKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *src_image*, *dst_image*, *src_origin*, *dst_origin*, *region* refer to **clEnqueueCopyImage**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.

- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

It is currently a requirement that the *src_image* and *dst_image* image memory objects for **clCommandCopyImageKHR** must have the exact same image format, i.e. the `cl_image_format` descriptor specified when *src_image* and *dst_image* are created must match.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_image* or *dst_image* is created is ignored by **clCommandCopyImageKHR**.

clCommandCopyImageKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueCopyImage** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer*, *src_image*, and *dst_image* is not the same.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to copy an image object to a buffer object, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandCopyImageToBufferKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem src_image,
    cl_mem dst_buffer,
    const size_t* src_origin,
    const size_t* region,
    size_t dst_offset,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandCopyImageToBufferKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *src_image*, *dst_buffer*, *src_origin*, *region*, *dst_offset* refer to **clEnqueueCopyImageToBuffer**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *src_image* or *dst_buffer* is created is ignored by **clCommandCopyImageToBufferKHR**.

clCommandCopyImageToBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueCopyImageToBuffer** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer*, *src_image*, and *dst_buffer* is not the same.
- `CL_INVALID_CONTEXT` *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to fill a buffer object with a pattern of a given pattern size, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandFillBufferKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem buffer,
    const void* pattern,
    size_t pattern_size,
    size_t offset,
    size_t size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandFillBufferKHR is provided by the `cl_khr_command_buffer` extension.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when *buffer* is created is ignored by **clCommandFillBufferKHR**.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *buffer*, *pattern*, *pattern_size*, *offset*, *size* refer to **clEnqueueFillBuffer**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait

for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.

- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

clCommandFillBufferKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueFillBuffer** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer* and *buffer* is not the same.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to fill an image object with a specified color, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandFillImageKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_mem image,
    const void* fill_color,
    const size_t* origin,
    const size_t* region,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandFillImageKHR is provided by the `cl_khr_command_buffer` extension.

The usage information which indicates whether the memory object can be read or written by a kernel and/or the host and is given by the `cl_mem_flags` argument value specified when image is created is ignored by **clCommandFillImageKHR**.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *image*, *fill_color*, *origin*, *region* refer to **clEnqueueFillImage**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not

NULL, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.

- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be NULL.

clCommandFillImageKHR returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueFillImage** except:

CL_INVALID_COMMAND_QUEUE is replaced with:

- CL_INVALID_COMMAND_QUEUE if the cl_khr_command_buffer_multi_device extension is not supported and *command_queue* is not NULL.
- CL_INVALID_COMMAND_QUEUE if the cl_khr_command_buffer_multi_device extension is supported; and either *command_queue* is NULL and *command_buffer* was created with more than one queue, or *command_queue* is not NULL and not a command-queue listed on *command_buffer* creation.

CL_INVALID_CONTEXT is replaced with:

- CL_INVALID_CONTEXT if the context associated with *command_buffer* and *image* is not the same.
- CL_INVALID_CONTEXT if *command_queue* is not NULL, and the context associated with *command_queue* and *command_buffer* is not the same.

CL_INVALID_EVENT_WAIT_LIST is replaced with:

- CL_INVALID_SYNC_POINT_WAIT_LIST_KHR if *sync_point_wait_list* is NULL and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not NULL and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- CL_INVALID_COMMAND_BUFFER_KHR if *command_buffer* is not a valid command-buffer.
- CL_INVALID_OPERATION if *command_buffer* has been finalized.
- CL_INVALID_VALUE if values specified in *properties* are not valid.
- CL_INVALID_VALUE if *mutable_handle* is not NULL.

To record a command to execute a kernel on a device, call the function

```
// Provided by cl_khr_command_buffer
cl_int clCommandNDRangeKernelKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t* global_work_offset,
    const size_t* global_work_size,
    const size_t* local_work_size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandNDRangeKernelKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the kernel command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. If a supported property and its value is not specified in *properties*, its default value will be used. *properties* may be `NULL`, in which case the default values for supported properties will be used. The `cl_khr_command_buffer` extension does not define any properties, but supported properties defined by extensions are defined in the List of supported properties by clCommandNDRangeKernelKHR table.
- *kernel* is a valid kernel object, which **must** have its arguments set unless the command has the `CL_MUTABLE_DISPATCH_ARGUMENTS_KHR` property set (see description of argument *mutable_handle*). Any changes to *kernel* after calling **clCommandNDRangeKernelKHR**, such as with **clSetKernelArg** or **clSetKernelExecInfo**, have no effect on the recorded command. If *kernel* is recorded to a following **clCommandNDRangeKernelKHR** command however, then that command will capture the updated state of *kernel*.
- *work_dim*, *global_work_offset*, *global_work_size*, *local_work_size* Refer to **clEnqueueNDRangeKernel**.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and

num_sync_points_in_wait_list must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. If the `cl_khr_command_buffer_mutable_dispatch` extension is supported, and *mutable_handle* is not `NULL`, it can be used in the `cl_mutable_dispatch_config_khr` struct to update the command configuration between recordings. The lifetime of this handle is tied to the parent command-buffer, such that freeing the command-buffer will also free this handle.

Table 60. List of supported properties by **clCommandNDRangeKernelKHR**

Recording Properties	Property Value	Description
`CL_MUTABLE_DISPATCH_ASSERTS_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_mutable_dispatch_asserts_khr`	This is a bitfield and can be set to a combination of the following values: `CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR` An assertion by the user that the number of work-groups of this ND-range kernel will not be updated beyond the number defined when the ND-range kernel was recorded. The number of work-groups is defined as the product for each *i* from *0* to *work_dim - 1* of *ceil(global_work_size[i]/local_work_size[i])*.

Recording Properties	Property Value	Description
CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	cl_mutable_dispatch_fields_khr	This is a bitfield and can be set to a combination of the following values: CL_MUTABLE_DISPATCH_GLOBAL_OFFSET_KHR determines whether the *global_work_offset* of kernel execution can be modified after recording. If set, the *global_work_offset* of the kernel execution can be changed with **clUpdateMutableCommandsKHR** using the cl_mutable_dispatch_config_khr field of the *mutable_config* parameter. Otherwise, the *global_work_offset* cannot be modified. CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR determines whether the *global_work_size* of kernel execution can be modified after recording. If set, the *global_work_size* of the kernel execution can be changed with **clUpdateMutableCommandsKHR** using the cl_mutable_dispatch_config_khr field of the *mutable_config* parameter. Otherwise, the *global_work_size* cannot be modified. CL_MUTABLE_DISPATCH_LOCAL_SIZE_KHR determines whether the *local_work_size* of kernel execution can be modified after recording. If set, the *local_work_size* of the kernel execution can be changed with **clUpdateMutableCommandsKHR** using the cl_mutable_dispatch_config_khr field of the *mutable_config* parameter. Otherwise, the *local_work_size* cannot be modified. CL_MUTABLE_DISPATCH_ARGUMENTS_KHR determines whether the kernel arguments set on *kernel* can be updated between executions. If set, the kernel arguments normally set with **clSetKernelArg** and **clSetKernelArgSVMPointer** can be changed with **clUpdateMutableCommandsKHR** using the cl_mutable_dispatch_config_khr field of the *mutable_config* parameter. Otherwise, the kernel arguments cannot be modified between executions. CL_MUTABLE_DISPATCH_EXEC_INFO_KHR determines whether the information passed to *kernel* can be updated between executions. If set, the execution information of the kernel can be changed with **clUpdateMutableCommandsKHR** using the cl_mutable_dispatch_config_khr field of the *mutable_config* parameter. Otherwise, the kernel execution information cannot be modified. If CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR is not specified then it defaults to the value returned by the CL_DEVICE_MUTABLE_DISPATCH_CAPABILITIES_KHR device query.

The work-group size to be used for *kernel* can also be specified in the program source using the `__attribute__((reqd_work_group_size(X, Y, Z)))` qualifier. In this case the size of work-group specified by *local_work_size* must match the value specified by the `reqd_work_group_size __attribute__` qualifier.

These work-group instances are executed in parallel across multiple compute units or concurrently on the same compute unit.

Each work-item is uniquely identified by a global identifier. The global ID, which can be read inside the kernel, is computed using the value given by *global_work_size* and *global_work_offset*. In addition, a work-item is also identified within a work-group by a unique local ID. The local ID, which can also be read by the kernel, is computed using the value given by *local_work_size*. The starting local ID is always (0, 0, ... 0).

clCommandNDRangeKernelKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueNDRangeKernel** except:

`CL_INVALID_KERNEL_ARGS` is replaced with:

- `CL_INVALID_KERNEL_ARGS` if the kernel argument values have not been specified and the `CL_MUTABLE_DISPATCH_ARGUMENTS_KHR` flag was not set in the *properties* parameter.

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if the context associated with *command_buffer* and *kernel* is not the same.
- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.

- CL_INVALID_VALUE if the cl_khr_command_buffer_mutable_dispatch extension is not supported and *mutable_handle* is not NULL.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support CL_COMMAND_BUFFER_CAPABILITY_KERNEL_PRINTF_KHR and *kernel* contains a printf call.
- CL_INVALID_OPERATION if the device associated with *command_queue* does not support CL_COMMAND_BUFFER_CAPABILITY_DEVICE_SIDE_ENQUEUE_KHR and *kernel* contains a kernel-enqueue call.

If the cl_khr_command_buffer_mutable_dispatch extension is supported:

- CL_INVALID_OPERATION if the requested CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR properties are not reported by CL_DEVICE_MUTABLE_DISPATCH_CAPABILITIES_KHR for the device associated with *command_queue*. If *command_queue* is NULL, the device associated with *command_buffer* must report support for these properties.
- CL_INVALID_VALUE if *command_buffer* was created with the CL_COMMAND_BUFFER_MUTABLE_DISPATCH_ASSERTS_KHR property with CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR and *local_work_size* is NULL, or if *properties* includes the CL_MUTABLE_DISPATCH_ASSERTS_KHR property with CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR and *local_work_size* is NULL.

To record a command to do an SVM memcpy operation, call the function

```
// Provided by CL_VERSION_2_0 with cl_khr_command_buffer
cl_int clCommandSVMMemcpyKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    void* dst_ptr,
    const void* src_ptr,
    size_t size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandSVMMemcpyKHR is provided by the cl_khr_command_buffer extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the cl_khr_command_buffer_multi_device extension is not supported, only a single command-queue is supported, and *command_queue* must be NULL.
  If the cl_khr_command_buffer_multi_device extension is supported and *command_queue* is NULL, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be NULL.
- *properties* specifies a list of properties for the command and thier corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The cl_khr_command_buffer extension does not define any properties, but supported properties may be defined by layered extensions in future.

- *dst_ptr* is the pointer to a host (if the device supports system SVM) or SVM memory allocation where data is copied to.
- *src_ptr* is the pointer to a host (if the device supports system SVM) or SVM memory allocation where data is copied from.
- *size* is the size in bytes of data being copied.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

clCommandSVMMemcpyKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueSVMMemcpy** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.

- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

To record a command to fill a region in SVM with a pattern of a given pattern size, call the function

```
// Provided by CL_VERSION_2_0 with cl_khr_command_buffer
cl_int clCommandSVMMemFillKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_queue command_queue,
    const cl_command_properties_khr* properties,
    void* svm_ptr,
    const void* pattern,
    size_t pattern_size,
    size_t size,
    cl_uint num_sync_points_in_wait_list,
    const cl_sync_point_khr* sync_point_wait_list,
    cl_sync_point_khr* sync_point,
    cl_mutable_command_khr* mutable_handle);
```

clCommandSVMMemFillKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* refers to a valid command-buffer object.
- *command_queue* specifies the command-queue the command will be recorded to.
  If the `cl_khr_command_buffer_multi_device` extension is not supported, only a single command-queue is supported, and *command_queue* must be `NULL`.
  If the `cl_khr_command_buffer_multi_device` extension is supported and *command_queue* is `NULL`, then only one command-queue must have been set on *command_buffer* creation; otherwise, *command_queue* must not be `NULL`.
- *properties* specifies a list of properties for the command and their corresponding values. Each property name is immediately followed by the corresponding desired value. The list is terminated with 0. The `cl_khr_command_buffer` extension does not define any properties, but supported properties may be defined by layered extensions in future.
- *svm_ptr* is a pointer to a (if the device supports system SVM) or SVM memory region that will be filled with *pattern*. It must be aligned to *pattern_size* bytes. If *svm_ptr* is allocated using **clSVMAlloc**, then it must be allocated from the same context from which *command_queue* was created. Otherwise the behavior is undefined.
- *pattern* is a pointer to the data pattern of size *pattern_size* in bytes. *pattern* will be used to fill a region in *buffer* starting at *svm_ptr* and is *size* bytes in size. The data pattern must be a scalar or vector integer or floating-point data type supported by OpenCL. For example, if the region pointed to by *svm_ptr* is to be filled with a pattern of `float4` values, then *pattern* will be a pointer to a `cl_float4` value and *pattern_size* will be `sizeof(cl_float4)`. The maximum value of *pattern_size* is the size of the largest integer or floating-point vector data type supported by the OpenCL device. The memory associated with *pattern* can be reused or freed after the function returns.

- *size* is the size in bytes of region being filled starting with *svm_ptr* and must be a multiple of *pattern_size*.
- *sync_point_wait_list*, *num_sync_points_in_wait_list* specify synchronization-points that need to complete before this particular command can be executed.

  If *sync_point_wait_list* is `NULL`, *num_sync_points_in_wait_list* must be 0. If *sync_point_wait_list* is not `NULL`, the list of synchronization-points pointed to by *sync_point_wait_list* must be valid and *num_sync_points_in_wait_list* must be greater than 0. The synchronization-points specified in *sync_point_wait_list* are **device-side** synchronization-points. The command-buffer associated with synchronization-points in *sync_point_wait_list* must be the same as *command_buffer*. The memory associated with *sync_point_wait_list* can be reused or freed after the function returns.

- *sync_point* returns a synchronization-point ID that identifies this particular command. Synchronization-point objects are unique and can be used to identify this command later on. *sync_point* can be `NULL` in which case it will not be possible for the application to record a wait for this command to complete. If the *sync_point_wait_list* and the *sync_point* arguments are not `NULL`, the *sync_point* argument should not refer to an element of the *sync_point_wait_list* array.
- *mutable_handle* returns a handle to the command. This parameter is unused, and **must** be `NULL`.

clCommandSVMMemFillKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns the errors defined by **clEnqueueSVMMemFill** except:

`CL_INVALID_COMMAND_QUEUE` is replaced with:

- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is not supported and *command_queue* is not `NULL`.
- `CL_INVALID_COMMAND_QUEUE` if the `cl_khr_command_buffer_multi_device` extension is supported; and either *command_queue* is `NULL` and *command_buffer* was created with more than one queue, or *command_queue* is not `NULL` and not a command-queue listed on *command_buffer* creation.

`CL_INVALID_CONTEXT` is replaced with:

- `CL_INVALID_CONTEXT` if *command_queue* is not `NULL`, and the context associated with *command_queue* and *command_buffer* is not the same.

`CL_INVALID_EVENT_WAIT_LIST` is replaced with:

- `CL_INVALID_SYNC_POINT_WAIT_LIST_KHR` if *sync_point_wait_list* is `NULL` and *num_sync_points_in_wait_list* is > 0, or *sync_point_wait_list* is not `NULL` and *num_sync_points_in_wait_list* is 0, or if synchronization-point objects in *sync_point_wait_list* are not valid synchronization-points.

New errors:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has been finalized.
- `CL_INVALID_VALUE` if values specified in *properties* are not valid.
- `CL_INVALID_VALUE` if *mutable_handle* is not `NULL`.

5.17.6. Remapping Command-Buffers

If the cl_khr_command_buffer_multi_device extension is supported, platforms reporting the CL_COMMAND_BUFFER_PLATFORM_REMAP_QUEUES_KHR capability support generating a deep copy of a command-buffer with its commands remapped to different devices than the devices used to create the commands. That is, the remapped command-buffer can execute on queues that differ in terms of properties and/or associated device from the original command-buffer queues.

This functionality is invoked through a new synchronous entry-point **clRemapCommandBufferKHR** which takes a list of queues to which the commands should now target the associated devices of. It then returns a command-buffer containing the same commands as the original, with the same command dependencies, but targeting different devices. A list of command handles may also be passed to the entry-point, which allows handles to the equivalent commands in the remapped command-buffer to be returned by an output parameter.

Device properties restrict remapping possibilities, as existing commands can have a configuration which is not supported by another device, and so remapping may fail with an error relating to this incompatibility. Examples of command configurations which can introduce incompatibilities when trying to map to a new device are:

- Program language features used in a kernel not supported by the new device.
- ND-Range configuration, e.g exceeds new the device max work-group size.
- Misalignment of sub-buffers based on minimum alignment of new device.

In additional to this functionality, platforms reporting CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR allow the user to create a remapped command-buffer where the mapping of queues to commands is determined by the OpenCL runtime in a way it determines as optimal. This is particularly useful in hot plugging environments where devices may appear and disappear during runtime.

To create a deep copy of the input command-buffer with the copied commands remapped to target devices of the passed command-queues, call the function

```
// Provided by cl_khr_command_buffer_multi_device
cl_command_buffer_khr clRemapCommandBufferKHR(
    cl_command_buffer_khr command_buffer,
    cl_bool automatic,
    cl_uint num_queues,
    const cl_command_queue* queues,
    cl_uint num_handles,
    const cl_mutable_command_khr* handles,
    cl_mutable_command_khr* handles_ret,
    cl_int* errcode_ret);
```

clRemapCommandBufferKHR is provided by the cl_khr_command_buffer_multi_device extension.

- *command_buffer* specifies the command-buffer to create a remapped deep copy of.

- *automatic* indicates if the remapping is done explicitly by the user, or automatically by the OpenCL runtime. If *automatic* is `CL_FALSE`, then each element of *queues* will replace the queue used on *command_buffer* creation at the same index. If `CL_TRUE` and `CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR` is supported, then the OpenCL runtime will decide in a way it determines optimal which of the elements in *queues* each command in the returned command-buffer will be associated with.
- *num_queues* is the number of command-queues listed in *queues*, must not be 0.
- *queues* is a pointer to an ordered list of command-queues for the returned command-buffer to target, must be a non-`NULL` value.
- *num_handles* is the number of command handles passed in both *handles* and *handles_ret* lists, may be 0.
- *handles* is an ordered list of handles belonging to *command_buffer* to create remapped copies of, may be `NULL`.
- *handles_ret* returns an ordered list of handles where each handle is equivalent to the handle at the same index in *handles*, but belonging to the returned command-buffer.
- *errcode_ret* returns an appropriate error code. If *errcode_ret* is `NULL`, no error code is returned.

The returned command-buffer has the same state as the input command-buffer.

clRemapCommandBufferKHR returns a valid command-buffer with *errcode_ret* set to `CL_SUCCESS` if the command-buffer is created successfully. Otherwise, it returns a `NULL` value without setting *handles_ret*, and with one of the following error values returned in *errcode_ret*:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_VALUE` if *num_queues* is 0, or if *queues* is `NULL`.
- `CL_INVALID_VALUE` if *automatic* is `CL_FALSE` and *num_queues* is not equal to the number of queues used on creation of *command_buffer*.
- `CL_INVALID_VALUE` if *handles* or *handles_ret* is `NULL` and *num_handles* is > 0, or either *handles* or *handles_ret* is not `NULL` and *num_handles* is 0.
- `CL_INVALID_VALUE` if any handle in *handles* is not a valid command handle belonging to *command_buffer*.
- `CL_INVALID_COMMAND_QUEUE` if any command-queue in *queues* is not a valid command-queue.
- `CL_INVALID_CONTEXT` if *command_buffer* and all the command-queues in *queues* do not have the same OpenCL context.
- `CL_INVALID_OPERATION` if the platform does not support the `CL_COMMAND_BUFFER_PLATFORM_REMAP_QUEUES_KHR` flag.
- `CL_INVALID_OPERATION` if the platform does not support the `CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR` flag and *automatic* is `CL_TRUE`.
- Any error relating to device support that can be returned by a command recording entry-point may also be returned. As a command in *command_buffer* can have a configuration that is not supported by a device that is associated with the queue in *queues* the command is being remapped to.

- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.17.7. Mutable Commands

A generic `cl_mutable_command_khr` handle is called a *mutable-command* object as it can be returned from any command recording entry-point in the `cl_khr_command_buffer` family of extensions. The mutable-command handles returned by **clCommandNDRangeKernelKHR** in particular are referred to as *mutable-dispatch* objects, and can be modified through the fields of `cl_mutable_dispatch_config_khr`.

Mutable-command handles are updated between enqueues using entry-point **clUpdateMutableCommandsKHR**. To enable performant usage, all aspects of mutation can be passed in a single call using an array. This means that the runtime has access to all the information about how the command-buffer will change, allowing the command-buffer to be rebuilt as efficiently as possible. Any modifications to the arguments or execution info of a mutable-dispatch handle using `cl_mutable_dispatch_arg_khr` or `cl_mutable_dispatch_exec_info_khr` have no affect on the original kernel object used when the command was recorded, and only influence the **clCommandNDRangeKernelKHR** command associated with the mutable-dispatch.

The base `cl_khr_command_buffer` extension notes that a command-buffer does not update the reference count of objects set as arguments on kernels recorded into the command-buffer.

The implications for applications using **clUpdateMutableCommandsKHR** is that it is safe to delete objects used as kernel command arguments, if all the kernel commands using that object as an argument have had their arguments replaced with a different object.

Implementations may defer some of the work to allow **clUpdateMutableCommandsKHR** to return immediately, provided the user observable behavior in subsequent command-buffer usage is specification conformant. Deferring work helps keep device occupancy high by avoiding blocking in host code.

When recording an ND-range kernel command, the kernel's arguments do not have to be set, and setting them may be postponed to after the command buffer's finalization (in which case, the command buffer is in Finalized state). All the arguments must be set before the command buffer can be enqueued for execution (Executable state).

To modify the configuration of mutable-command handles returned during *command_buffer* recording, updating the behavior of those commands in future enqueues of *command_buffer*, call the function

```
// Provided by cl_khr_command_buffer_mutable_dispatch
cl_int clUpdateMutableCommandsKHR(
    cl_command_buffer_khr command_buffer,
    cl_uint num_configs,
    const cl_command_buffer_update_type_khr* config_types,
    const void** configs);
```

clUpdateMutableCommandsKHR is provided by the `cl_khr_command_buffer_mutable_dispatch` extension.

- *command_buffer* refers to a valid command-buffer object.
- *num_configs* Number of elements in the *config_types* and *config* arrays.
- *config_types* An array of length *num_configs* with each element identifying the type of each config in *configs* at the same array index.
- *configs* An array of length *num_configs* containing structs which define how a mutable-command handle in *command_buffer* is to be updated, each of which is interpreted using *config_types* at the same index with the mapping defined in the Mutable Command Update Structs section.

clUpdateMutableCommandsKHR returns `CL_SUCCESS` if all the mutable-command objects were updated successfully. Otherwise, none of the updates to mutable-command objects are preserved and one of the errors below is returned:

- `CL_INVALID_COMMAND_BUFFER_KHR` if *command_buffer* is not a valid command-buffer.
- `CL_INVALID_OPERATION` if *command_buffer* has not been finalized.
- `CL_INVALID_OPERATION` if *command_buffer* was not created with the `CL_COMMAND_BUFFER_MUTABLE_KHR` flag.
- `CL_INVALID_VALUE` if *config_types* is `NULL` and *num_configs* > 0, or *config_types* is not `NULL` and *num_configs* is 0.
- `CL_INVALID_VALUE` if *configs* is `NULL` and *num_configs* > 0, or *configs* is not `NULL` and *num_configs* is 0.
- `CL_INVALID_VALUE` if any element of *config_types* is not a valid `cl_command_buffer_update_type_khr` enum.
- `CL_INVALID_VALUE` if any element of *configs* is NULL.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

Performant usage is to call **clUpdateMutableCommandsKHR** only when the desired state of all commands is known, rather than iteratively updating each command individually.

If the command buffer has been created with CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR, or the updated ND-range command has been recorded with this flag, and the ND-range parameters are updated so that the new number of work-groups exceeds the number when the ND-range command was recorded, the behavior is undefined.

If *configs* is non-NULL, then for any cl_mutable_dispatch_config_khr element of the array the errors defined by **clEnqueueNDRangeKernel**, **clSetKernelExecInfo**, **clSetKernelArg**, and **clSetKernelArgSVMPointer** are returned by **clUpdateMutableCommandsKHR** if any of the struct elements are set to an invalid value. Additionally, the following errors are returned if any cl_mutable_dispatch_config_khr element of the array violates the defined conditions:

- CL_INVALID_MUTABLE_COMMAND_KHR if *command* is not a valid mutable command object returned from **clCommandNDRangeKernelKHR**, or created from *command_buffer*.
- CL_INVALID_OPERATION if the values of *local_work_size* and/or *global_work_size* result in a change to work-group uniformity.
- CL_INVALID_OPERATION if *work_dim* is not zero and different from the *work_dim* set on *command* recording.
- CL_INVALID_OPERATION if the CL_MUTABLE_DISPATCH_GLOBAL_OFFSET_KHR property was not set on *command* recording and *global_work_offset* is not NULL.
- CL_INVALID_OPERATION if the CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR property was not set on *command* recording and *global_work_size* is not NULL.
- CL_INVALID_OPERATION if the CL_MUTABLE_DISPATCH_LOCAL_SIZE_KHR property was not set on *command* recording and *local_work_size* is not NULL.
- CL_INVALID_OPERATION if the CL_MUTABLE_DISPATCH_ARGUMENTS_KHR property was not set on *command* recording and *num_args* or *num_svm_args* is non-zero.
- CL_INVALID_OPERATION if the CL_MUTABLE_DISPATCH_EXEC_INFO_KHR property was not set on *command* recording and *num_exec_infos* is non-zero.
- CL_INVALID_VALUE if *arg_list* is NULL and *num_args* > 0, or *arg_list* is not NULL and *num_args* is 0.
- CL_INVALID_VALUE if *arg_svm_list* is NULL and *num_svm_args* > 0, or *arg_svm_list* is not NULL and *num_svm_args* is 0.
- CL_INVALID_VALUE if *exec_info_list* is NULL and *num_exec_infos* > 0, or *exec_info_list* is not NULL and *num_exec_infos* is 0.

5.17.7.1. Mutable Command Update Structs

The following table defines the mapping of cl_command_buffer_update_type_khr values to the structs they define reinterpreting a void pointer as when passed to **clUpdateMutableCommandsKHR**.

Enum Value	Struct Type	Entry Point
CL_STRUCTURE_TYPE_MUTABLE_DISPATCH_CONFIG_KHR	cl_mutable_dispatch_config_khr	**clCommandNDRangeKernelKHR**

5.17.7.2. Kernel Command Update Structs

The `cl_mutable_dispatch_arg_khr` structure is passed to **clUpdateMutableCommandsKHR** to set the kernel configuration of a mutable **clCommandNDRangeKernelKHR** command, and is defined as:

```
// Provided by cl_khr_command_buffer_mutable_dispatch
typedef struct cl_mutable_dispatch_config_khr {
    cl_mutable_command_khr                      command;
    cl_uint                                     num_args;
    cl_uint                                     num_svm_args;
    cl_uint                                     num_exec_infos;
    cl_uint                                     work_dim;
    const cl_mutable_dispatch_arg_khr*          arg_list;
    const cl_mutable_dispatch_arg_khr*          arg_svm_list;
    const cl_mutable_dispatch_exec_info_khr*    exec_info_list;
    const size_t*                               global_work_offset;
    const size_t*                               global_work_size;
    const size_t*                               local_work_size;
} cl_mutable_dispatch_config_khr;
```

- *command* is a mutable-command object returned by **clCommandNDRangeKernelKHR** representing a kernel execution as part of a command-buffer.
- *num_args* is the number of kernel arguments being changed.
- *num_svm_args* is the number of SVM kernel arguments being changed.
- *num_exec_infos* is the number of kernel execution info objects to set for this dispatch.
- *work_dim* is the number of dimensions used to specify the global work-items and work-items in the work-group. May be set to zero to signify the number of dimensions is not changed. See **clEnqueueNDRangeKernel** for valid usage.
- *arg_list* is an array describing the new kernel arguments for this enqueue. It must contain *num_args* array elements, each of which encapsulates parameters passed to **clSetKernelArg**. See **clSetKernelArg** for usage of `cl_mutable_dispatch_arg_khr` members.
- *arg_svm_list* is an array describing the new SVM kernel arguments for this enqueue. It must contain *num_svm_args* array elements, each of which encapsulates parameters passed to **clSetKernelArgSVMPointer**. See **clSetKernelArgSVMPointer** for usage of `cl_mutable_dispatch_arg_khr` members, `arg_size` is ignored.
- *exec_info_list* is an array containing *num_exec_infos* elements specifying the list of execution info objects use for this command-buffer enqueue. See **clSetKernelExecInfo** for usage of `cl_mutable_dispatch_exec_info_khr` members.
- *global_work_offset* can be used to specify an array of *work_dim* unsigned values that describe the offset used to calculate the global ID of a work-item. If *global_work_offset* is `NULL` then the global offset of the dispatch is not changed. See **clEnqueueNDRangeKernel** for valid usage.
- *global_work_size* points to an array of *work_dim* unsigned values that describe the number of global work-items in *work_dim* dimensions that will execute the kernel function. If *global_work_size* is `NULL` then the number of global work-items in the dispatch is not changed. See **clEnqueueNDRangeKernel** for valid usage.

- *local_work_size* points to an array of *work_dim* unsigned values that describe the number of work-items that make up a work-group that will execute the kernel. If *local_work_size* is `NULL` then the number of local work-items in the dispatch is not changed. See **clEnqueueNDRangeKernel** for valid usage.

The `cl_mutable_dispatch_arg_khr` structure sets kernel arguments normally passed using **clSetKernelArg** and **clSetKernelArgSVMPointer**, and is defined as:

```
// Provided by cl_khr_command_buffer_mutable_dispatch
typedef struct cl_mutable_dispatch_arg_khr {
    cl_uint        arg_index;
    size_t         arg_size;
    const void*    arg_value;
} cl_mutable_dispatch_arg_khr;
```

The `cl_mutable_dispatch_exec_info_khr` structure sets kernel execution info normally passed using **clSetKernelExecInfo**, and is defined as:

```
// Provided by cl_khr_command_buffer_mutable_dispatch
typedef struct cl_mutable_dispatch_exec_info_khr {
    cl_uint        param_name;
    size_t         param_value_size;
    const void*    param_value;
} cl_mutable_dispatch_exec_info_khr;
```

param_name is of type `cl_uint` rather than `cl_kernel_exec_info` so that the extension can be implemented on OpenCL 1.2 where the `cl_kernel_exec_info` typedef is unavailable.

5.17.8. Command-Buffer Queries

To query information about a command-buffer, call the function

```
// Provided by cl_khr_command_buffer
cl_int clGetCommandBufferInfoKHR(
    cl_command_buffer_khr command_buffer,
    cl_command_buffer_info_khr param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetCommandBufferInfoKHR is provided by the `cl_khr_command_buffer` extension.

- *command_buffer* specifies the command-buffer being queried.

- *param_name* specifies the information to query.
- *param_value* is a pointer to a memory location where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Command-Buffer Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is `NULL`, it is ignored.

The list of supported *param_name* values and the information returned in *param_value* by **clGetCommandBufferInfoKHR** is described in the table below.

Table 61. **clGetCommandBufferInfoKHR** *values*

Command Buffer Info	Return Type	Description
`CL_COMMAND_BUFFER_NUM_QUEUES_KHR` provided by the `cl_khr_command_buffer` extension.	`cl_uint`	The number of command-queues specified when *command_buffer* was created.
`CL_COMMAND_BUFFER_QUEUES_KHR` provided by the `cl_khr_command_buffer` extension.	`cl_command_queue`[]	Return the list of command-queues specified when the *command_buffer* was created.
`CL_COMMAND_BUFFER_REFERENCE_COUNT_KHR` [21] provided by the `cl_khr_command_buffer` extension.	`cl_uint`	Return the *command_buffer* reference count.

Command Buffer Info	Return Type	Description
`CL_COMMAND_BUFFER_STATE_KHR` provided by the `cl_khr_command_buffer` extension.	`cl_command_buffer_state_khr`	Return the state of *command_buffer*. `CL_COMMAND_BUFFER_STATE_RECORDING_KHR` is returned when *command_buffer* has not been finalized. provided by the `cl_khr_command_buffer` extension. `CL_COMMAND_BUFFER_STATE_FINALIZED_KHR` is returned when an instance of *command_buffer* has been finalized, but there is at least one command for which not all arguments or parameters have been set. provided by the `cl_khr_command_buffer_mutable_dispatch` extension. `CL_COMMAND_BUFFER_STATE_EXECUTABLE_KHR` is returned when *command_buffer* has been finalized, all the arguments and parameters of all commands have been set. provided by the `cl_khr_command_buffer` extension.

Command Buffer Info	Return Type	Description
CL_COMMAND_BUFFER_PROPERTIES_ARRAY_KHR provided by the cl_khr_command_buffer extension.	cl_command_buffer_properties_khr[]	Return the *properties* argument specified in **clCreateCommandBufferKHR**. If the *properties* argument specified in **clCreateCommandBufferKHR** used to create *command_buffer* was not NULL, the implementation must return the values specified in the properties argument. If the *properties* argument specified in **clCreateCommandBufferKHR** used to create *command_buffer* was NULL, the implementation may return either a *param_value_size_ret* of 0 (i.e. there is are no properties to be returned), or the implementation may return a property value of 0 (where 0 is used to terminate the properties list).
CL_COMMAND_BUFFER_CONTEXT_KHR provided by the cl_khr_command_buffer extension.	cl_context	Return the context associated with *command_buffer*.

clGetCommandBufferInfoKHR returns CL_SUCCESS if the function is executed successfully. Otherwise, it returns one of the following errors:

- CL_INVALID_COMMAND_BUFFER_KHR if *command_buffer* is not a valid command-buffer.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Command-Buffer Queries table and *param_value* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device.
- CL_OUT_OF_HOST_MEMORY if there is a failure to allocate resources required by the OpenCL implementation on the host.

To query information about a mutable command object, call the function

```
// Provided by cl_khr_command_buffer_mutable_dispatch
cl_int clGetMutableCommandInfoKHR(
    cl_mutable_command_khr command,
    cl_mutable_command_info_khr param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

clGetMutableCommandInfoKHR is provided by the `cl_khr_command_buffer_mutable_dispatch` extension.

- *command* specifies the mutable-command object being queried.
- *param_name* specifies the information to query. The list of supported *param_name* types and the information returned in *param_value* by **clGetMutableCommandInfoKHR** is described in the Mutable Command Object Queries table.
- *param_value* is a pointer to memory where the appropriate result being queried is returned. If *param_value* is `NULL`, it is ignored.
- *param_value_size* is used to specify the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Mutable Command Object Queries table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_name*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 62. Mutable Command Object Queries

Mutable Command Info	Return Type	Description
`CL_MUTABLE_COMMAND_COMMAND_QUEUE_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_command_queue`	Return the command-queue associated with *command*. If `NULL` was passed as the queue when *command* was recorded, then the queue associated with the command-buffer that *command* belongs to is returned.
`CL_MUTABLE_COMMAND_COMMAND_BUFFER_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_command_buffer_khr`	Return the command-buffer associated with *command*.
`CL_MUTABLE_COMMAND_COMMAND_TYPE_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_command_type`	Return the command-type associated with *command*. The list of supported event command types defined by **clGetEventInfo** is used with the matching command.

Mutable Command Info	Return Type	Description
`CL_MUTABLE_COMMAND_PROPERTIES_ARRAY_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_command_properties_khr[]`	Return the properties argument specified on *command* recording. If the properties argument specified on creation of *command* was not `NULL`, the implementation must return the values specified in the properties argument in the same order and without including additional properties. If the properties argument specified on creation of *command* was `NULL`, the implementation must return *param_value_size_ret* equal to 0, indicating that there are no properties to be returned.
`CL_MUTABLE_DISPATCH_KERNEL_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_kernel`	Return the kernel associated with *command* when recorded with **clCommandNDRangeKernelKHR**. If *command* was not recorded from a **clCommandNDRangeKernelKHR** command, the implementation must return *param_value_size_ret* equal to 0, indicating that the value returned in *param_value* is not valid.
`CL_MUTABLE_DISPATCH_DIMENSIONS_KHR` provided by the `cl_khr_command_buffer_mutable_dispatch` extension.	`cl_uint`	Return the number of work-item dimensions specified when *command* was created. If *command* was not recorded from a **clCommandNDRangeKernelKHR** command, the implementation must return *param_value_size_ret* equal to 0, indicating that the value returned in *param_value* is not valid.

Mutable Command Info	Return Type	Description
CL_MUTABLE_DISPATCH_GLOBAL_WORK_OFFSET_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	size_t[]	Return the global work-item offset set on *command* creation, or from the most recent update via **clUpdateMutableCommandsKHR** where this value was modified. The output array contains *work_dim* values, where *work_dim* is returned by the query CL_MUTABLE_DISPATCH_DIMENSIONS_KHR. If a global work-item offset was not set, zero is returned for each element in the array. If *command* was not recorded from a **clCommandNDRangeKernelKHR** command, the implementation must return *param_value_size_ret* equal to 0, indicating that the value returned in *param_value* is not valid.
CL_MUTABLE_DISPATCH_GLOBAL_WORK_SIZE_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	size_t[]	Return the global work-item size set on *command* creation, or from the most recent update via **clUpdateMutableCommandsKHR** where this value was modified. The output array contains *work_dim* values, where *work_dim* is returned by the query CL_MUTABLE_DISPATCH_DIMENSIONS_KHR. If a global work-item size was not set, zero is returned for each element in the array. If *command* was not recorded from a **clCommandNDRangeKernelKHR** command, the implementation must return *param_value_size_ret* equal to 0, indicating that the value returned in *param_value* is not valid.
CL_MUTABLE_DISPATCH_LOCAL_WORK_SIZE_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	size_t[]	Return the local work-item size set on *command* creation, or from the most recent update via **clUpdateMutableCommandsKHR** where this value was modified. The output array contains *work_dim* values, where *work_dim* is returned by the query CL_MUTABLE_DISPATCH_DIMENSIONS_KHR. If a local work-item size was not set, zero is returned for each element in the array. If *command* was not recorded from a **clCommandNDRangeKernelKHR** command, the implementation must return *param_value_size_ret* equal to 0, indicating that the value returned in *param_value* is not valid.

clGetMutableCommandInfoKHR returns `CL_SUCCESS` if the function is executed successfully. Otherwise, it returns one of the following errors:

- `CL_INVALID_MUTABLE_COMMAND_KHR` if *command* is not a valid mutable command object.
- `CL_INVALID_VALUE` if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Mutable Command Object Queries table and *param_value* is not `NULL`.
- `CL_OUT_OF_RESOURCES` if there is a failure to allocate resources required by the OpenCL implementation on the device.
- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

5.18. Querying Devices That Support Sharing With OpenGL

OpenCL device(s) corresponding to an OpenGL context may be queried. Such a device may not always exist (for example, if an OpenGL context is specified on a GPU not supporting OpenCL command-queues, but which does support shared OpenCL/OpenGL memory objects), and if it does exist, may change over time. When such a device does exist, acquiring and releasing shared OpenCL/OpenGL memory objects may be faster on a command-queue corresponding to this device than on command-queues corresponding to other devices available to an OpenCL context.

To query the OpenCL device corresponding to an OpenGL context, call the function

```
// Provided by cl_khr_gl_sharing
cl_int clGetGLContextInfoKHR(
    const cl_context_properties* properties,
    cl_gl_context_info param_name,
    size_t param_value_size,
    void* param_value,
    size_t* param_value_size_ret);
```

- *properties* points to an property list whose format and valid contents are identical to the *properties* argument of **clCreateContext**. *properties* must identify a single valid GL context or GL share group object.
- *param_name* is a constant that specifies the device types to query, and must be one of the values shown in the Supported Device Types table below.
- *param_value* is a pointer to memory where the result of the query is returned, as described in the Supported Device Types table. If *param_value* is `NULL`, it is ignored.
- *param_value_size* specifies the size in bytes of memory pointed to by *param_value*. This size must be greater than or equal to the size of the return type specified in the Supported Device Types table. If *param_value* is `NULL`, it is ignored.
- *param_value_size_ret* returns the actual size in bytes of data being queried by *param_value*. If *param_value_size_ret* is `NULL`, it is ignored.

Table 63. Supported Device Types for **clGetGLContextInfoKHR**

param_name	Return Type	Information returned in param_value
CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR provided by the cl_khr_gl_sharing extension.	cl_device_id	Return the OpenCL device currently associated with the specified OpenGL context.
CL_DEVICES_FOR_GL_CONTEXT_KHR provided by the cl_khr_gl_sharing extension.	cl_device_id[]	Return all OpenCL devices which may be associated with the specified OpenGL context.

clGetGLContextInfoKHR returns CL_SUCCESS if the function is executed successfully. If no device(s) exist corresponding to *param_name*, the call will not fail, but the value of *param_value_size_ret* will be zero. Otherwise, it returns one of the following errors:

- CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a context was specified for an OpenGL or OpenGL ES implementation using the EGL, GLX, or WGL binding APIs, as described for clCreateContext; and any of the following conditions hold:
  - The specified display and context properties do not identify a valid OpenGL or OpenGL ES context.
  - The specified context does not support buffer and renderbuffer objects.
  - The specified context is not compatible with the OpenCL context being created (for example, it exists in a physically distinct address space, such as another hardware device; or it does not support sharing data with OpenCL due to implementation restrictions).
- CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR if a share group was specified for a CGL-based OpenGL implementation by setting the property CL_CGL_SHAREGROUP_KHR, and the specified share group does not identify a valid CGL share group object.
- CL_INVALID_OPERATION if a context was specified as described above and any of the following conditions hold:
  - A context or share group object was specified for one of CGL, EGL, GLX, or WGL and the OpenGL implementation does not support that window-system binding API.
  - More than one of the properties CL_CGL_SHAREGROUP_KHR, CL_EGL_DISPLAY_KHR, CL_GLX_DISPLAY_KHR, and CL_WGL_HDC_KHR is set to a non-default value.
  - Both of the properties CL_CGL_SHAREGROUP_KHR and CL_GL_CONTEXT_KHR are set to non-default values.
- CL_INVALID_VALUE if a property name specified in *properties* is invalid.
- CL_INVALID_VALUE if *param_name* is not one of the supported values, or if the size in bytes specified by *param_value_size* is less than size of the return type specified in the Supported Device Types table and *param_value* is not NULL.
- CL_OUT_OF_RESOURCES if there is a failure to allocate resources required by the OpenCL implementation on the device

- `CL_OUT_OF_HOST_MEMORY` if there is a failure to allocate resources required by the OpenCL implementation on the host.

[1] Only out-of-order device queues are supported.

[2] The application must create a default device queue if any kernels containing calls to `get_default_queue` are enqueued. There can only be one default device queue for each device within a context. If a default device queue has already been created, calling **clCreateCommandQueueWithProperties** with `CL_QUEUE_PROPERTIES` set to `CL_QUEUE_ON_DEVICE` and `CL_QUEUE_ON_DEVICE_DEFAULT` will return the default device queue that has already been created and increment its reference count by 1.

[3] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[4] Note that reading and writing 2D image arrays from a kernel with `image_array_size` equal to one may perform worse than 2D images.

[5] Therefore, specifying *num_mip_levels* equal to either `0` or `1` creates an image with a single mipmap level.

[6] To create an image from a buffer object that shares the data store between the image and buffer object.

[7] To create an image object from another image object that share the data store between these image objects.

[8] Support for the `CL_DEPTH` image channel order is required only for 2D images and 2D image arrays.

[9] Support for reading from the `CL_sRGBA` image channel order is optional for 1D image buffers. Support for writing to the `CL_sRGBA` image channel order is optional for all image types.

[10] The map count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for debugging.

[11] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[12] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[13] As per the definition of `-cl-denorms-are-zero`, the inclusion of this option with `-cl-unsafe-math-optimizations` means that the implementation may flush denormal numbers to zero but is not required to.

[14] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[15] When the size of a `local` argument is set to zero, the value of the pointer within the kernel is implementation-defined.

[16] Implementations are encouraged to favor this option as it makes it more likely that errors will be managed by applications.

[17] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[18] The error code values are negative, and event state values are positive. The event state values are ordered from the largest value `CL_QUEUED` for the first or initial state to the smallest value (`CL_COMPLETE` or negative integer value) for the last or complete state. The value of `CL_COMPLETE` and `CL_SUCCESS` are the same.

[19] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[20] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

[21] The reference count returned should be considered immediately stale. It is unsuitable for general use in applications. This feature is provided for identifying memory leaks.

Chapter 6. Associated OpenCL Specifications

6.1. The OpenCL C Kernel Language

The OpenCL C kernel language is defined in the OpenCL C unified specification, which specifies all versions of OpenCL C. When an OpenCL device supports one or more versions of the OpenCL C kernel language (see `CL_DEVICE_OPENCL_C_VERSION` and `CL_DEVICE_OPENCL_C_ALL_VERSIONS`), OpenCL program objects may be created by passing OpenCL C source strings to **clCreateProgramWithSource**.

6.2. SPIR-V Intermediate Language

The OpenCL specifications include the SPIR-V specification, which defines a cross-platform intermediate language. When an OpenCL device supports one or more versions of SPIR-V (see `CL_DEVICE_IL_VERSION` or `CL_DEVICE_ILS_WITH_VERSION`), OpenCL program objects may be created by passing SPIR-V to **clCreateProgramWithIL**.

6.3. Extensions to OpenCL

In addition to the specification of core features, OpenCL provides a number of extensions to the API, kernel language, or intermediate representation. These extensions are described in the OpenCL specifications or in vendor-provided documentation.

Chapter 7. OpenCL Embedded Profile

The OpenCL specification describes the feature requirements for desktop platforms. This section describes the OpenCL embedded profile, which defines a subset of the OpenCL specification for handheld and embedded platforms. Optional OpenCL extensions may apply to both profiles.

The OpenCL embedded profile has the following restrictions until version 2.0 (i.e. the optionality described below was deprecated by version 2.0):

1. Support for 3D images is optional.

   If `CL_DEVICE_IMAGE3D_MAX_WIDTH`, `CL_DEVICE_IMAGE3D_MAX_HEIGHT` and `CL_DEVICE_IMAGE3D_MAX_DEPTH` are zero, calls to **clCreateImage** or **clCreateImageWithProperties** will fail to create the 3D image, and the *errcode_ret* argument will return `CL_INVALID_OPERATION`. Declaring arguments of type `image3d_t` in a kernel will result in a compilation error.

   If `CL_DEVICE_IMAGE3D_MAX_WIDTH`, `CL_DEVICE_IMAGE3D_MAX_HEIGHT` and `CL_DEVICE_IMAGE3D_MAX_DEPTH` are greater than zero 0, calls to **clCreateImage** and **clCreateImageWithProperties** will behave as described for full profile implementations, and the `image3d_t` data type can be used in a kernel.

2. Support for 2D image array writes is optional. If the **cles_khr_2d_image_array_writes** extension is supported by the embedded profile, writes to 2D image arrays are supported.
3. Image and image arrays created with an `image_channel_data_type` value of `CL_FLOAT` or `CL_HALF_FLOAT` can only be used with samplers that use a filter mode of `CL_FILTER_NEAREST`. The values returned by **read_imagef** [1] for 2D and 3D images if `image_channel_data_type` value is `CL_FLOAT` or `CL_HALF_FLOAT` and sampler with filter_mode = `CL_FILTER_LINEAR` are undefined.

Furthermore, the OpenCL embedded profile has the following restrictions for all versions:

1. 64 bit integers i.e. long, ulong including the appropriate vector data types and operations on 64-bit integers are optional. The **cles_khr_int64** [2] extension string will be reported if the embedded profile implementation supports 64-bit integers. If double precision is supported i.e. `CL_DEVICE_DOUBLE_FP_CONFIG` is not zero, then **cles_khr_int64** must also be supported.
2. The mandated minimum single precision floating-point capability given by `CL_DEVICE_SINGLE_FP_CONFIG` is `CL_FP_ROUND_TO_ZERO` or `CL_FP_ROUND_TO_NEAREST`. If `CL_FP_ROUND_TO_NEAREST` is supported, the default rounding mode will be round to nearest even; otherwise the default rounding mode will be round to zero.
3. The single precision floating-point operations (addition, subtraction and multiplication) shall be correctly rounded. Zero results may always be positive 0.0. The accuracy of division and sqrt are given in the OpenCL C and OpenCL SPIR-V Environment specifications.

   If `CL_FP_INF_NAN` is not set in `CL_DEVICE_SINGLE_FP_CONFIG`, and one of the operands or the result of addition, subtraction, multiplication or division would signal the overflow or invalid exception (see IEEE 754 specification), the value of the result is implementation-defined. Likewise, single precision comparison operators (<, >, <=, >=, ==, !=) return implementation-defined values when one or more operands is a NaN.

In all cases, conversions (see the OpenCL C and OpenCL SPIR-V Environment specifications) shall be correctly rounded as described for the FULL_PROFILE, including those that consume or produce an INF or NaN. The built-in math functions shall behave as described for the FULL_PROFILE, including edge case behavior, but with slightly different accuracy rules. Edge case behavior and accuracy rules are described in the OpenCL C and OpenCL SPIR-V Environment specifications.

If addition, subtraction and multiplication have default round to zero rounding mode, then **fract**, **fma** and **fdim** shall produce the correctly rounded result for round to zero rounding mode.

This relaxation of the requirement to adhere to IEEE 754 requirements for basic floating-point operations, though extremely undesirable, is to provide flexibility for embedded devices that have lot stricter requirements on hardware area budgets.

4. Denormalized numbers for the half data type which may be generated when converting a float to a half using variants of the **vstore_half** function or when converting from a half to a float using variants of the **vload_half** function can be flushed to zero. The OpenCL SPIR-V Environment Specification for details.
5. The precision of conversions from `CL_UNORM_INT8`, `CL_SNORM_INT8`, `CL_UNORM_INT16`, `CL_SNORM_INT16`, `CL_UNORM_INT_101010`, and `CL_UNORM_INT_101010_2` to float is ≤ 2 ulp for the embedded profile instead of ≤ 1.5 ulp as defined in the full profile. The exception cases described in the full profile and given below apply to the embedded profile.

   For `CL_UNORM_INT8`

   - 0 must convert to 0.0f and
   - 255 must convert to 1.0f

   For `CL_UNORM_INT16`

   - 0 must convert to 0.0f and
   - 65535 must convert to 1.0f

   For `CL_SNORM_INT8`

   - -128 and -127 must convert to -1.0f,
   - 0 must convert to 0.0f and
   - 127 must convert to 1.0f

   For `CL_SNORM_INT16`

   - -32768 and -32767 must convert to -1.0f,
   - 0 must convert to 0.0f and
   - 32767 must convert to 1.0f

For CL_UNORM_INT_101010

- 0 must convert to 0.0f and
- 1023 must convert to 1.0f

For CL_UNORM_INT_101010_2

- 0 must convert to 0.0f and
- 1023 must convert to 1.0f (for RGB)
- 3 must convert to 1.0f (for A)

CL_PLATFORM_PROFILE defined in the OpenCL Platform Queries table will return the string EMBEDDED_PROFILE if the OpenCL implementation supports the embedded profile only.

The minimum maximum values specified in the OpenCL Device Queries table that have been modified for the OpenCL embedded profile are listed in the OpenCL Embedded Device Queries table.

Table 64. List of supported param_names by **clGetDeviceInfo** *for embedded profile*

Device Info	Return Type	Description
CL_DEVICE_MAX_READ_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the read_only qualifier. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only qualifier. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS	cl_uint	Max number of image objects arguments of a kernel declared with the write_only or read_write qualifier. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE2D_MAX_WIDTH	size_t	Max width of 2D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE2D_MAX_HEIGHT	size_t	Max height of 2D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE3D_MAX_WIDTH	size_t	Max width of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE3D_MAX_HEIGHT	size_t	Max height of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.

Device Info	Return Type	Description
CL_DEVICE_IMAGE3D_MAX_DEPTH	size_t	Max depth of 3D image in pixels. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE_MAX_BUFFER_SIZE	size_t	Max number of pixels for a 1D image created from a buffer object. The minimum value is 2048 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_IMAGE_MAX_ARRAY_SIZE	size_t	Max number of images in a 1D or 2D image array. The minimum value is 256 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_SAMPLERS	cl_uint	Maximum number of samplers that can be used in a kernel. The minimum value is 8 if CL_DEVICE_IMAGE_SUPPORT is CL_TRUE, the value is 0 otherwise.
CL_DEVICE_MAX_PARAMETER_SIZE	size_t	Max size in bytes of all arguments that can be passed to a kernel. The minimum value is 256 bytes. A maximum of 255 arguments can be passed to a kernel.

Device Info	Return Type	Description
CL_DEVICE_SINGLE_FP_CONFIG	cl_device_fp_config	Describes single precision floating-point capability of the device. This is a bit-field that describes one or more of the following values: CL_FP_DENORM - denorms are supported CL_FP_INF_NAN - INF and quiet NaNs are supported. CL_FP_ROUND_TO_NEAREST - round to nearest even rounding mode supported CL_FP_ROUND_TO_ZERO - round to zero rounding mode supported CL_FP_ROUND_TO_INF - round to positive and negative infinity rounding modes supported CL_FP_FMA - the fma kernel builtin is implemented using hardware-accelerated fused multiply-add as defined in IEEE754-2008. CL_FP_CORRECTLY_ROUNDED_DIVIDE_SQRT - divide and sqrt are correctly rounded as defined by the IEEE754 specification. CL_FP_SOFT_FLOAT - Basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. The mandated minimum floating-point capability is: CL_FP_ROUND_TO_ZERO or CL_FP_ROUND_TO_NEAREST.
CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE	cl_ulong	Max size in bytes of a constant buffer allocation. The minimum value is 1 KB.
CL_DEVICE_MAX_CONSTANT_ARGS	cl_uint	Max number of arguments declared with the __constant qualifier in a kernel. The minimum value is 4.
CL_DEVICE_LOCAL_MEM_SIZE	cl_ulong	Size of local memory arena in bytes. The minimum value is 1 KB.

Device Info	Return Type	Description
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Is CL_TRUE if a compiler is available to compile programs created from source or IL, or CL_FALSE if the implementation does not have a compiler available. This can be CL_FALSE for devices supporting the embedded profile. When an online compiler is not available, OpenCL programs may still be created from binaries or built-in kernels supported by the device.
CL_DEVICE_LINKER_AVAILABLE	cl_bool	Is CL_TRUE if a linker is available to link compiled programs, or CL_FALSE if the implementation does not have a linker available. This can be CL_FALSE for devices supporting the embedded profile, but must be CL_TRUE if CL_DEVICE_COMPILER_AVAILABLE is CL_TRUE.
CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE	cl_uint	The max. size of the device queue in bytes. The minimum value is 64 KB for the embedded profile
CL_DEVICE_PRINTF_BUFFER_SIZE	size_t	Maximum size in bytes of the internal buffer that holds the output of printf calls from a kernel. The minimum value for the EMBEDDED profile is 1 KB.

If CL_DEVICE_IMAGE_SUPPORT specified in the OpenCL Device Queries table is CL_TRUE, the values assigned to CL_DEVICE_MAX_READ_IMAGE_ARGS, CL_DEVICE_MAX_WRITE_IMAGE_ARGS, CL_DEVICE_IMAGE2D_MAX_WIDTH, CL_DEVICE_IMAGE2D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_WIDTH, CL_DEVICE_IMAGE3D_MAX_HEIGHT, CL_DEVICE_IMAGE3D_MAX_DEPTH, and CL_DEVICE_MAX_SAMPLERS by the implementation must be greater than or equal to the minimum values specified in the OpenCL Embedded Device Queries table.

If CL_DEVICE_IMAGE_SUPPORT specified in the OpenCL Device Queries table is CL_TRUE, the minimum list of supported image formats for either reading or writing in a kernel for embedded profile devices is:

Table 65. Minimum list of supported image formats for reading or writing (embedded profile)

num_channels	channel_order	channel_data_type
4	CL_RGBA	CL_UNORM_INT8 CL_UNORM_INT16 CL_SIGNED_INT8 CL_SIGNED_INT16 CL_SIGNED_INT32 CL_UNSIGNED_INT8 CL_UNSIGNED_INT16 CL_UNSIGNED_INT32 CL_HALF_FLOAT CL_FLOAT

For embedded profiles devices that support reading from and writing to the same image object from the same kernel-instance (see CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS) there is no required minimum list of supported image formats.

[1] And **read_imageh**, if the cl_khr_fp16 extension is supported.

[2] Note that the performance of 64-bit integer arithmetic can vary significantly between embedded devices.

Appendix A: Host environment and thread safety

Shared OpenCL Objects

This section describes which objects can be shared across multiple command-queues. The command-queues can be created in one host thread or across multiple host threads within a host process.

OpenCL memory objects, program objects, and kernel objects are created using an OpenCL context and can be shared across multiple command-queues created using the same context. Event objects can be created when a command is queued to a command-queue. These event objects can be shared across multiple command-queues created using the same context.

The application must implement appropriate synchronization to ensure that the changes to the state of a shared object (such as a command-queue object, memory object, program object, or kernel object) happen in the correct order (deemed correct by the application) when multiple host threads or command-queues change the state of a shared object.

The OpenCL memory consistency model describes how to correctly order operations that change the state of a shared object.

Multiple Host Threads

OpenCL calls that modify the state of `cl_kernel` objects are not thread-safe [1]: **clSetKernelArg**, **clSetKernelArgSVMPointer**, **clSetKernelExecInfo** and **clCloneKernel**. These APIs are safe to call from any host thread so long as concurrent calls operate on different `cl_kernel` objects. The state of a `cl_kernel` object is undefined if **clSetKernelArg**, **clSetKernelArgSVMPointer**, **clSetKernelExecInfo** or **clCloneKernel** are called from multiple host threads on the same `cl_kernel` object at the same time [2].

Additionally, OpenCL calls that modify the state of `cl_command_buffer_khr` objects are not thread-safe: **clFinalizeCommandBufferKHR** and **clUpdateMutableCommandsKHR**. These APIs are safe to call from any host thread so long as concurrent calls operate on different `cl_command_buffer_khr` objects. The state of the `cl_command_buffer_khr` object is undefined if **clFinalizeCommandBufferKHR** or **clUpdateMutableCommandsKHR** are called from multiple host threads on the same `cl_command_buffer_khr` object at the same time.

All other OpenCL API calls are thread-safe. Please note that there are additional limitations as to which OpenCL APIs may be called from OpenCL callback functions.

The behavior of OpenCL APIs called from an interrupt or signal handler is implementation-defined.

The OpenCL implementation should be able to create multiple command-queues for a given OpenCL context and multiple OpenCL contexts in an application running on the host processor.

Global Constructors and Destructors

The execution order of global constructors and destructors is left undefined by the C and C++ standards. It is therefore not possible to know the relative execution order of an OpenCL implementation's global constructors and destructors with respect to an OpenCL application's or library's.

The behavior of OpenCL API functions called from global constructors or destructors is therefore implementation-defined.

[1] Please refer to the OpenCL glossary for the OpenCL definition of thread-safe. This definition may be different from usage of the term in other contexts.

[2] There is an inherent race condition in the design of OpenCL that occurs between setting a kernel argument and using the kernel with **clEnqueueNDRangeKernel**. Another host thread might change the kernel arguments between when a host thread sets the kernel arguments and then enqueues the kernel, causing the wrong kernel arguments to be enqueued. Rather than attempt to share `cl_kernel` objects among multiple host threads, applications are strongly encouraged to make additional `cl_kernel` objects for each host thread.

Appendix B: Portability

OpenCL is designed to be portable to other architectures and hardware designs. OpenCL has used at its core a C99 based programming language and follows rules based on that heritage. Floating-point arithmetic is based on the **IEEE-754** and **IEEE-754-2008** standards. The memory objects, pointer qualifiers and weakly ordered memory are designed to provide maximum compatibility with discrete memory architectures implemented by OpenCL devices. Command-queues and barriers allow for synchronization between the host and OpenCL devices. The design, capabilities and limitations of OpenCL are very much a reflection of the capabilities of underlying hardware.

Unfortunately, there are a number of areas where idiosyncrasies of one hardware platform may allow it to do some things that do not work on another. By virtue of the rich operating system resident on the CPU, on some implementations the kernels executing on a CPU may be able to call out to system services whereas the same calls on the GPU will likely fail for now. Since there is some advantage to having these services available for debugging purposes, implementations can use the OpenCL extension mechanism to implement these services.

Likewise, the heterogeneity of computing architectures might mean that a particular loop construct might execute at an acceptable speed on the CPU but very poorly on a GPU, for example. CPUs are designed in general to work well on latency sensitive algorithms on single threaded tasks, whereas common GPUs may encounter extremely long latencies, potentially orders of magnitude worse. Developers interested in writing portable code may need to test their software on a diversity of hardware designs to make sure that key algorithms are structured in a way that works well on a diversity of hardware. We suggest favoring more work-items over fewer. It is anticipated that over the coming months and years experience will produce a set of best practices that will help foster a uniformly favorable experience on a diversity of computing devices.

Of somewhat more concern is the topic of endianness. Since a majority of devices supported by the initial implementation of OpenCL are little-endian, developers need to make sure that their kernels are tested on both big-endian and little-endian devices to ensure source compatibility with OpenCL devices now and in the future. The endian attribute qualifier is supported by the SPIR-V IL to allow developers to specify whether the data uses the endianness of the host or the OpenCL device. This allows the OpenCL compiler to do appropriate endian-conversion on load and store operations from or to this data.

We also describe how endianness can leak into an implementation causing kernels to produce unintended results:

When a big-endian vector machine (e.g. AltiVec, CELL SPE) loads a vector, the order of the data is retained. That is both the order of the bytes within each element and the order of the elements in the vector are the same as in memory. When a little-endian vector machine (e.g. SSE) loads a vector, the order of the data in register (where all the work is done) is reversed. **Both** the order of the bytes within each element and the order of the elements with respect to one another in the vector are reversed.

Memory:

uint4 a =

0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F

In register (big-endian):

uint4 a =

0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F

In register (little-endian):

uint4 a =

0x0F0E0D0C	0x0B0A0908	0x07060504	0x03020100

This allows little-endian machines to use a single vector load to load little-endian data, regardless of how large each piece of data is in the vector. That is the transformation is equally valid whether that vector was a `uchar16` or a `ulong2`. Of course, as is well known, little-endian machines actually [1] store their data in reverse byte order to compensate for the little-endian storage format of the array elements:

Memory (big-endian):

uint4 a =

0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F

Memory (little-endian):

uint4 a =

0x03020100	0x07060504	0x0B0A0908	0x0F0E0D0C

Once that data is loaded into a vector, we end up with this:

In register (big-endian):

uint4 a =

0x00010203	0x04050607	0x08090A0B	0x0C0D0E0F

In register (little-endian):

uint4 a =

0x0C0D0E0F	0x08090A0B	0x04050607	0x00010203

That is, in the process of correcting the endianness of the bytes within each element, the machine ends up reversing the order that the elements appear in the vector with respect to each other within the vector. 0x00010203 appears at the left of the big-endian vector and at the right of the little-endian vector.

When the host and device have different endianness, the developer must ensure that kernel argument values are processed correctly. The implementation may or may not automatically convert endianness of kernel arguments. Developers should consult vendor documentation for guidance on how to handle kernel arguments in these situations.

OpenCL provides a consistent programming model across architectures by numbering elements according to their order in memory. Concepts such as `even`/`odd` and `high`/`low` follow accordingly. Once the data is loaded into registers, we find that element 0 is at the left of the big-endian vector and element 0 is at the right of the little-endian vector:

```
float x[4];
float4 v = vload4( 0, x );
```

Big-endian:

```
v contains { x[0], x[1], x[2], x[3] }
```

Little-endian:

```
v contains { x[3], x[2], x[1], x[0] }
```

The compiler is aware that this swap occurs and references elements accordingly. So long as we refer to them by a numeric index such as `.s0123456789abcdef` or by descriptors such as `.xyzw`, `.hi`, `.lo`, `.even` and `.odd`, everything works transparently. Any ordering reversal is undone when the data is stored back to memory. The developer should be able to work with a big-endian programming model and ignore the element ordering problem in the vector ... for most problems. This mechanism relies on the fact that we can rely on a consistent element numbering. Once we change numbering system, for example by conversion-free casting (using `as_type_`*n*) a vector to another vector of the same size but a different number of elements, then we get different results on different implementations depending on whether the system is big-endian, or little-endian or indeed has no vector unit at all. (Thus, the behavior of bitcasts to vectors of different numbers of elements is implementation-defined, see section 6.4.4 of OpenCL C specification.)

An example follows:

```
float x[4] = { 0.0f, 1.0f, 2.0f, 3.0f };
float4 v = vload4( 0, x );
uint4 y = as_uint4(v);      // legal, portable
ushort8 z = as_ushort8(v);  // legal, not portable
                            // element size changed
```

Big-endian:

```
v contains { 0.0f, 1.0f, 2.0f, 3.0f }
y contains { 0x00000000, 0x3f800000,
```

```
              0x40000000, 0x40400000 }
z contains { 0x0000, 0x0000, 0x3f80, 0x0000,
             0x4000, 0x0000, 0x4040, 0x0000 }
z.z is 0x3f80
```

Little-endian:

```
v contains { 3.0f, 2.0f, 1.0f, 0.0f }
y contains { 0x40400000, 0x40000000,
             0x3f800000, 0x00000000 }
z contains { 0x4040, 0x0000, 0x4000, 0x0000,
             0x3f80, 0x0000, 0x0000, 0x0000 }
z.z is 0
```

Here, the value in `z.z` is not the same between big- and little-endian vector machines

OpenCL could have made it illegal to do a conversion free cast that changes the number of elements in the name of portability. However, while OpenCL provides a common set of operators drawing from the set that are typically found on vector machines, it cannot provide access to everything every ISA may offer in a consistent uniform portable manner. Many vector ISAs provide special purpose instructions that greatly accelerate specific operations such as DCT, SAD, or 3D geometry. It is not intended for OpenCL to be so heavy handed that time-critical performance sensitive algorithms cannot be written by knowledgeable developers to perform at near peak performance. Developers willing to throw away portability should be able to use the platform-specific instructions in their code. For this reason, OpenCL is designed to allow traditional vector C language programming extensions, such as the AltiVec C Programming Interface or the Intel C programming interfaces (such as those found in emmintrin.h) to be used directly in OpenCL with OpenCL data types as an extension to OpenCL. As these interfaces rely on the ability to do conversion-free casts that change the number of elements in the vector to function properly, OpenCL allows them too.

As a general rule, any operation that operates on vector types in segments that are not the same size as the vector element size may break on other hardware with different endianness or different vector architecture.

Examples might include:

- Combining two `uchar8`'s containing high and low bytes of a ushort, to make a `ushort8` using `.even` and `.odd` operators (please use **upsample()** for this)
- Any bitcast that changes the number of elements in the vector. (Operations on the new type are non-portable.)
- Swizzle operations that change the order of data using chunk sizes that are not the same as the element size

Examples of operations that are portable:

- Combining two `uint8`'s to make a `uchar16` using `.even` and `.odd` operators. For example to interleave left and right audio streams.

- Any bitcast that does not change the number of elements (e.g. `(float4) uint4`) — we define the storage format for floating-point types)
- Swizzle operations that swizzle elements of the same size as the elements of the vector.

OpenCL has made some additions to C to make application behavior more dependable than C. Most notably in a few cases OpenCL defines the behavior of some operations that are undefined in C99:

- OpenCL provides `convert_` functions for conversion between all types. C99 does not define what happens when a floating-point type is converted to an integer type and the floating-point value lies outside the representable range of the integer type after rounding. When the `sat` variant of the conversion is used, the float shall be converted to the nearest representable integer value. Similarly, OpenCL also makes recommendations about what should happen with NaN. Hardware manufacturers that provide the saturated conversion in hardware may use the saturated conversion hardware for both the saturated and non-saturated versions of the OpenCL `convert_` functions. OpenCL does not define what happens for the non-saturated conversions when floating-point operands are outside the range representable integers after rounding.
- The format of `half`, `float`, and `double` types is defined to be the binary16, binary32 and binary64 formats in the draft IEEE-754 standard. (The latter two are identical to the existing IEEE-754 standard.) You may depend on the positioning and meaning of the bits in these types.
- OpenCL defines behavior for oversized shift values. Shift operations that shift greater than or equal to the number of bits in the first operand reduce the shift value modulo the number of bits in the element. For example, if we shift an `int4` left by `33` bits, OpenCL treats this as shift left by `33%32 = 1` bit.
- A number of edge cases for math library functions are more rigorously defined than in C99. Please see *section 7.5* of the OpenCL C specification.

[1] Note that we are talking about the programming model here. In reality, little endian systems might choose to simply address their bytes from "the right" or reverse the "order" of the bits in the byte. Either of these choices would mean that no big swap would need to occur in hardware.

Appendix C: Application Data Types

This section documents the provided host application types and constant definitions. The documented material describes the commonly defined data structures, types and constant values available to all platforms and architectures. The addition of these details demonstrates our commitment to maintaining a portable programming environment and potentially deters changes to the supplied headers.

Supported Application Scalar Data Types

The following application scalar types are provided for application convenience.

```
cl_char
cl_uchar
cl_short
cl_ushort
cl_int
cl_uint
cl_long
cl_ulong
cl_half
cl_float
cl_double
```

Supported Application Vector Data Types

Application vector types are unions used to create vectors of the above application scalar types. The following application vector types are provided for application convenience.

```
cl_char<n>
cl_uchar<n>
cl_short<n>
cl_ushort<n>
cl_int<n>
cl_uint<n>
cl_long<n>
cl_ulong<n>
cl_half<n>
cl_float<n>
cl_double<n>
```

n can be 2, 3, 4, 8 or 16.

The application scalar and vector data types are defined in the **cl_platform.h** header file.

Alignment of Application Data Types

The user is responsible for ensuring that pointers passed into and out of OpenCL kernels are natively aligned relative to the data type of the parameter as defined in the kernel language and SPIR-V specifications. This implies that OpenCL buffers created with `CL_MEM_USE_HOST_PTR` need to provide an appropriately aligned host memory pointer that is aligned to the data types used to access these buffers in a kernel(s), that SVM allocations must correctly align and that pointers into SVM allocations must also be correctly aligned. The user is also responsible for ensuring image data passed is aligned to the granularity of the data representing a single pixel (e.g. `image_num_channels * sizeof(image_channel_data_type)`) except for `CL_RGB` and `CL_RGBx` images where the data must be aligned to the granularity of a single channel in a pixel (i.e. `sizeof(image_channel_data_type)`). This implies that OpenCL images created with `CL_MEM_USE_HOST_PTR` must align correctly. The image alignment value can be queried using the `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT` query. In addition, source pointers for **clEnqueueWriteImage** and other operations that copy to the OpenCL runtime, as well as destination pointers for **clEnqueueReadImage** and other operations that copy from the OpenCL runtime must follow the same alignment rules.

OpenCL makes no requirement about the alignment of OpenCL application defined data types outside of buffers and images, except that the underlying vector primitives (e.g. `__cl_float4`) where defined shall be directly accessible as such using appropriate named fields in the cl_type union (see Vector Components. Nevertheless, it is recommended that the **cl_platform.h** header should attempt to naturally align OpenCL defined application data types (e.g. `cl_float4`) according to their type.

Vector Literals

Application vector literals may be used in assignments of individual vector components. Literal usage follows the convention of the underlying application compiler.

```
cl_float2 foo = { .s[1] = 2.0f };
cl_int8 bar = {{ 2, 4, 6, 8, 10, 12, 14, 16 }};
```

Vector Components

The components of application vector types can be addressed using the `<vector_name>.s[<index>]` notation.

For example:

```
foo.s[0] = 1.0f; // Sets the 1st vector component of foo
pos.s[6] = 2;    // Sets the 7th vector component of bar
```

In some cases vector components may also be accessed using the following notations. These notations are not guaranteed to be supported on all implementations, so their use should be accompanied by a check of the corresponding preprocessor symbol.

Named Vector Components Notation

Vector data type components may be accessed using the `.sN`, `.sn` or `.xyzw` field naming convention, similar to how they are used within the OpenCL C language. Use of the `.xyzw` field naming convention only allows accessing of the first 4 component fields. Support of these notations is identified by the `CL_HAS_NAMED_VECTOR_FIELDS` preprocessor symbol. For example:

```
#ifdef CL_HAS_NAMED_VECTOR_FIELDS
    cl_float4 foo;
    cl_int16 bar;
    foo.x = 1.0f;  // Set first component
    foo.s0 = 1.0f; // Same as above
    bar.z = 3;     // Set third component
    bar.se = 11;   // Same as bar.s[0xe]
    bar.sD = 12;   // Same as bar.s[0xd]
#endif
```

Vector data type components may also be accessed using the `.rgba` field naming convention, similar to how they are used within the OpenCL C 3.0 or newer language. Use of the `.rgba` field naming convention only allows accessing of the first 4 component fields. Support of these notations is identified by the `CL_HAS_NAMED_RGBA_VECTOR_FIELDS` preprocessor symbol. For example:

```
#ifdef CL_HAS_NAMED_RGBA_VECTOR_FIELDS
    cl_float4 foo;
    cl_int16 bar;
    foo.r = 1.0f;  // Set first component
    bar.b = 3;     // Set third component
#endif
```

Unlike the OpenCL C language type usage of named vector fields, only one component field may be accessed at a time. This restriction prevents the ability to swizzle or replicate components as is possible with the OpenCL C language types. Attempting to access beyond the number of components for a type also results in a failure.

```
foo.xy    // illegal - illegal field name combination
bar.s1234 // illegal - illegal field name combination
foo.s7    // illegal - no component s7
```

High/Low Vector Component Notation

Vector data type components may be accessed using the `.hi` and `.lo` notation similar to that supported within the language types. Support of this notation is identified by the `CL_HAS_HI_LO_VECTOR_FIELDS` preprocessor symbol. For example:

```
#ifdef CL_HAS_HI_LO_VECTOR_FIELDS
    cl_float4 foo;
```

```
    cl_float2 new_hi = 2.0f, new_lo = 4.0f;
    foo.hi = new_hi;
    foo.lo = new_lo;
#endif
```

Native Vector Type Notation

Certain native vector types are defined for providing a mapping of vector types to architecturally built-in vector types. Unlike the above described application vector types, these native types are supported on a limited basis depending on the supporting architecture and compiler.

These types are not unions, but rather convenience mappings to the underlying architectures' built-in vector types. The native types share the name of their application counterparts but are preceded by a double underscore "__".

For example, `__cl_float4` is the native built-in vector type equivalent of the `cl_float4` application vector type. The `__cl_float4` type may provide direct access to the architectural built-in `__m128` or vector float type, whereas the `cl_float4` is treated as a union.

In addition, the above described application data types may have native vector data type members for access convenience. The native components are accessed using the `.vN` sub-vector notation, where `N` is the number of elements in the sub-vector. In cases where the native type is a subset of a larger type (more components), the notation becomes an index based array of the sub-vector type.

Support of the native vector types is identified by a `__CL_TYPEN__` preprocessor symbol matching the native type name. For example:

```
#ifdef __CL_FLOAT4__ // Check for native cl_float4 type
    cl_float8 foo;
    __cl_float4 bar; // Use of native type
    bar = foo.v4[1]; // Access the second native float4 vector
#endif
```

Implicit Conversions

Implicit conversions between application vector types are not supported.

Explicit Casts

Explicit casting of application vector types (`cl_typen`) is not supported. Explicit casting of native vector types (`__cl_typen`) is defined by the external compiler.

Other Operators and Functions

The behavior of standard operators and function on both application vector types (`cl_typen`) and native vector types (`__cl_typen`) is defined by the external compiler.

Application Constant Definitions

In addition to the above application type definitions, the following literal definitions are also available.

CL_CHAR_BIT	**Bit width of a character**
CL_SCHAR_MAX	Maximum value of a type cl_char
CL_SCHAR_MIN	Minimum value of a type cl_char
CL_CHAR_MAX	Maximum value of a type cl_char
CL_CHAR_MIN	Minimum value of a type cl_char
CL_UCHAR_MAX	Maximum value of a type cl_uchar
CL_SHRT_MAX	Maximum value of a type cl_short
CL_SHRT_MIN	Minimum value of a type cl_short
CL_USHRT_MAX	Maximum value of a type cl_ushort
CL_INT_MAX	Maximum value of a type cl_int
CL_INT_MIN	Minimum value of a type cl_int
CL_UINT_MAX	Maximum value of a type cl_uint
CL_LONG_MAX	Maximum value of a type cl_long
CL_LONG_MIN	Minimum value of a type cl_long
CL_ULONG_MAX	Maximum value of a type cl_ulong
CL_FLT_DIG	Number of decimal digits of precision for the type cl_float
CL_FLT_MANT_DIG	Number of digits in the mantissa of type cl_float
CL_FLT_MAX_10_EXP	Maximum positive integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_float
CL_FLT_MAX_EXP	Maximum exponent value of type cl_float
CL_FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_float
CL_FLT_MIN_EXP	Minimum exponent value of type cl_float
CL_FLT_RADIX	Base value of type cl_float
CL_FLT_MAX	Maximum value of type cl_float
CL_FLT_MIN	Minimum value of type cl_float

CL_CHAR_BIT	**Bit width of a character**
CL_FLT_EPSILON	Minimum positive floating-point number of type cl_float such that 1.0 + CL_FLT_EPSILON != 1 is true.
CL_DBL_DIG missing before version 1.2. Also see cl_khr_fp64.	Number of decimal digits of precision for the type cl_double
CL_DBL_MANT_DIG missing before version 1.2. Also see cl_khr_fp64.	Number of digits in the mantissa of type cl_double
CL_DBL_MAX_10_EXP missing before version 1.2. Also see cl_khr_fp64.	Maximum positive integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_double
CL_DBL_MAX_EXP missing before version 1.2. Also see cl_khr_fp64.	Maximum exponent value of type cl_double
CL_DBL_MIN_10_EXP missing before version 1.2. Also see cl_khr_fp64.	Minimum negative integer such that 10 raised to this power minus one can be represented as a normalized floating-point number of type cl_double
CL_DBL_MIN_EXP missing before version 1.2. Also see cl_khr_fp64.	Minimum exponent value of type cl_double
CL_DBL_RADIX missing before version 1.2. Also see cl_khr_fp64.	Base value of type cl_double
CL_DBL_MAX missing before version 1.2. Also see cl_khr_fp64.	Maximum value of type cl_double
CL_DBL_MIN missing before version 1.2. Also see cl_khr_fp64.	Minimum value of type cl_double
CL_DBL_EPSILON missing before version 1.2. Also see cl_khr_fp64.	Minimum positive floating-point number of type cl_double such that 1.0 + CL_DBL_EPSILON != 1 is true.
CL_NAN missing before version 1.1.	Macro expanding to a value representing NaN
CL_HUGE_VALF missing before version 1.1.	Largest representative value of type cl_float

`CL_CHAR_BIT`	**Bit width of a character**
`CL_HUGE_VAL` missing before version 1.1.	Largest representative value of type `cl_double`
`CL_MAXFLOAT` missing before version 1.1.	Maximum value of type `cl_float`
`CL_INFINITY` missing before version 1.1.	Macro expanding to a value representing infinity

These literal definitions are defined in the **cl_platform.h** header.

Appendix D: Checking for Memory Copy Overlap

The following code describes how to determine if there is overlap between the source and destination rectangles specified to **clEnqueueCopyBufferRect** provided the source and destination buffers refer to the same buffer object.

```
unsigned int
check_copy_overlap(const size_t src_origin[],
                   const size_t dst_origin[],
                   const size_t region[],
                   const size_t row_pitch,
                   const size_t slice_pitch )
{
  const size_t slice_size = (region[1] - 1) * row_pitch + region[0];
  const size_t block_size = (region[2] - 1) * slice_pitch + slice_size;
  const size_t src_start = src_origin[2] * slice_pitch
                           + src_origin[1] * row_pitch
                           + src_origin[0];
  const size_t src_end = src_start + block_size;
  const size_t dst_start = dst_origin[2] * slice_pitch
                           + dst_origin[1] * row_pitch
                           + dst_origin[0];
  const size_t dst_end = dst_start + block_size;

  /* No overlap if dst ends before src starts or if src ends
   * before dst starts.
   */
  if( (dst_end <= src_start) || (src_end <= dst_start) ){
    return 0;
  }

  /* No overlap if region[0] for dst or src fits in the gap
   * between region[0] and row_pitch.
   */
  {
    const size_t src_dx = src_origin[0] % row_pitch;
    const size_t dst_dx = dst_origin[0] % row_pitch;

    if( ((dst_dx >= src_dx + region[0]) &&
        (dst_dx + region[0] <= src_dx + row_pitch)) ||
        ((src_dx >= dst_dx + region[0]) &&
        (src_dx + region[0] <= dst_dx + row_pitch)) )
      {
        return 0;
      }
  }
```

```
  /* No overlap if region[1] for dst or src fits in the gap
   * between region[1] and slice_pitch.
   */
   {
      const size_t src_dy =
        (src_origin[1] * row_pitch + src_origin[0]) % slice_pitch;
      const size_t dst_dy =
        (dst_origin[1] * row_pitch + dst_origin[0]) % slice_pitch;

      if( ((dst_dy >= src_dy + slice_size) &&
          (dst_dy + slice_size <= src_dy + slice_pitch)) ||
          ((src_dy >= dst_dy + slice_size) &&
          (src_dy + slice_size <= dst_dy + slice_pitch)) ) {
        return 0;
      }
   }

 /* Otherwise src and dst overlap. */
 return 1;
}
```

Appendix E: Changes to OpenCL

Changes to the OpenCL API and OpenCL C specifications between successive versions are summarized below.

Summary of Changes from OpenCL 1.0 to OpenCL 1.1

The following features are added to the OpenCL 1.1 platform layer and runtime (*sections 4 and 5*):

- Following queries to *table 4.3*
  - `CL_DEVICE_NATIVE_VECTOR_WIDTH_CHAR`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_SHORT`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_INT`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_LONG`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_FLOAT`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_DOUBLE`, `CL_DEVICE_NATIVE_VECTOR_WIDTH_HALF`
  - `CL_DEVICE_HOST_UNIFIED_MEMORY`
  - `CL_DEVICE_OPENCL_C_VERSION`
- `CL_CONTEXT_NUM_DEVICES` to the list of queries specified to **clGetContextInfo**.
- Optional image formats: `CL_Rx`, `CL_RGx`, and `CL_RGBx`.
- Support for sub-buffer objects ability to create a buffer object that refers to a specific region in another buffer object using **clCreateSubBuffer**.
- **clEnqueueReadBufferRect**, **clEnqueueWriteBufferRect** and **clEnqueueCopyBufferRect** APIs to read from, write to and copy a rectangular region of a buffer object respectively.
- **clSetMemObjectDestructorCallback** API to allow a user to register a callback function that will be called when the memory object is deleted and its resources freed.
- Options that control the OpenCL C version used when building a program executable.
- `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE` to the list of queries specified to **clGetKernelWorkGroupInfo**.
- Support for user events. User events allow applications to enqueue commands that wait on a user event to finish before the command is executed by the device. Following new APIs are added - **clCreateUserEvent** and **clSetUserEventStatus**.
- **clSetEventCallback** API to register a callback function for a specific command execution status.

The following modifications are made to the OpenCL 1.1 platform layer and runtime (*sections 4 and 5*):

- Following queries in *table 4.3*
  - The minimum FULL_PROFILE value for `CL_DEVICE_MAX_PARAMETER_SIZE` increased from 256 to 1024 bytes
  - The minimum FULL_PROFILE value for `CL_DEVICE_LOCAL_MEM_SIZE` increased from 16 KB to 32 KB.
- The *global_work_offset* argument in **clEnqueueNDRangeKernel** can be a non-`NULL` value.
- All API calls except **clSetKernelArg** are thread-safe.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 1.1:

- 3-component vector data types.
- New built-in functions
  - **get_global_offset** work-item function defined in section *6.15.1*.
  - **minmag**, **maxmag** math functions defined in section *6.15.2*.
  - **clamp** integer function defined in *section 6.15.3*.
  - (vector, scalar) variant of integer functions **min** and **max** in *section 6.12.3*.
  - **async_work_group_strided_copy** defined in section *6.15.11*.
  - **vec_step**, **shuffle** and **shuffle2** defined in section *6.15.13*.
- `cl_khr_byte_addressable_store` extension is a core feature.
- `cl_khr_global_int32_base_atomics`, `cl_khr_global_int32_extended_atomics`, `cl_khr_local_int32_base_atomics` and `cl_khr_local_int32_extended_atomics` extensions are core features. The built-in atomic function names are changed to use the **atomic_** prefix instead of **atom_**.
- Macros `CL_VERSION_1_0` and `CL_VERSION_1_1`.

The following features in OpenCL 1.0 are deprecated (see glossary) in OpenCL 1.1:

- The **clSetCommandQueueProperty** API is deprecated, which simplifies implementations and possibly improves performance by enforcing that command-queue properties are invariant. Applications are encouraged to create multiple command-queues with different properties versus modifying the properties of a single command-queue.
- The `-cl-strict-aliasing` build option has been deprecated. It is no longer required after defining type-based aliasing rules.
- The `cl_khr_select_fprounding_mode` extension is deprecated and its use is no longer recommended.

The following new extensions are added to *section 9* in OpenCL 1.1:

- `cl_khr_gl_event` for creating a CL event object from a GL sync object.
- `cl_khr_d3d10_sharing` for sharing memory objects with Direct3D 10.

The following modifications are made to the OpenCL ES Profile described in *section 10* in OpenCL 1.1:

- 64-bit integer support is optional.

Summary of Changes from OpenCL 1.1 to OpenCL 1.2

The following features are added to the OpenCL 1.2 platform layer and runtime (*sections 4 and 5*):

- Custom devices and built-in kernels are supported. **clCreateProgramWithBuiltInKernels** has been added to allow creation of a `cl_program` using built-in kernels.
- Device partitioning that allows a device to be partitioned based on a number of partitioning

schemes supported by the device. This is done by using **clCreateSubDevices** to create a new `cl_device_id` based on a partitioning.

- **clCompileProgram** and **clLinkProgram** to allow handling these aspects **clBuildProgram** separately.
- Extend `cl_mem_flags` to describe how the host accesses the data in a `cl_mem` object.
- **clEnqueueFillBuffer** and **clEnqueueFillImage** to support filling a buffer with a pattern or an image with a color.
- Add `CL_MAP_WRITE_INVALIDATE_REGION` to `cl_map_flags`. Appropriate clarification to the behavior of `CL_MAP_WRITE` has been added to the spec.
- New image types: 1D image, 1D image from a buffer object, 1D image array and 2D image arrays.
- **clCreateImage** to create an image object.
- **clEnqueueMigrateMemObjects** API that allows a developer to have explicit control over the location of memory objects or to migrate a memory object from one device to another.
- Support separate compilation and linking of programs.
- Additional queries to get the number of kernels and kernel names in a program have been added to **clGetProgramInfo**.
- Additional queries to get the compile and link status and options have been added to **clGetProgramBuildInfo**.
- **clGetKernelArgInfo** API that returns information about the arguments of a kernel.
- **clEnqueueMarkerWithWaitList** and **clEnqueueBarrierWithWaitList** APIs.
- **clUnloadPlatformCompiler** to request that a single platform's compiler is unloaded. This is compatible with the `cl_khr_icd` extension if that is supported, unlike **clUnloadCompiler**.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 1.2:

- Double-precision is now an optional core feature instead of an extension.
- New built in image types: **image1d_t**, **image1d_buffer_t**, **image1d_array_t**, and **image2d_array_t**.
- New built-in functions
  - Functions to read from and write to a 1D image, 1D and 2D image arrays described in *sections 6.15.15.2*, *6.15.15.3* and *6.15.15.4*.
  - Sampler-less image read functions described in *section 6.15.15.3*.
  - **popcount** integer function described in *section 6.15.3*.
  - **printf** function described in *section 6.15.14*.
- Storage class specifiers extern and static as described in *section 6.10*.
- Macros `CL_VERSION_1_2` and `__OPENCL_C_VERSION__`.

The following APIs in OpenCL 1.1 are deprecated (see glossary) in OpenCL 1.2:

- The **clEnqueueMarker**, **clEnqueueBarrier** and **clEnqueueWaitForEvents** APIs are deprecated

to simplify the API. The **clEnqueueMarkerWithWaitList** and **clEnqueueBarrierWithWaitList** APIs provide equivalent functionality and support explicit event wait lists.

- The **clCreateImage2D**, **clCreateImage3D**, **clCreateFromGLTexture2D** and **clCreateFromGLTexture3D** APIs are deprecated to simplify the API. The **clCreateImage** and **clCreateFromGLTexture** APIs provide equivalent functionality and support additional image types and properties.
- **clUnloadCompiler** and **clGetExtensionFunctionAddress** APIs are deprecated. The **clUnloadPlatformCompiler** and **clGetExtensionFunctionAddressForPlatform** APIs provide equivalent functionality are compatible with the `cl_khr_icd` extension.

The following queries are deprecated (see glossary) in OpenCL 1.2:

- The `CL_DEVICE_MIN_DATA_TYPE_ALIGN_SIZE` query is deprecated. The minimum data type alignment can be derived from `CL_DEVICE_MEM_BASE_ADDR_ALIGN`.

Summary of Changes from OpenCL 1.2 to OpenCL 2.0

The following features are added to the OpenCL 2.0 platform layer and runtime (*sections 4 and 5*):

- Shared virtual memory. The associated API additions are:
  - **clSetKernelArgSVMPointer** to control which shared virtual memory (SVM) pointer to associate with a kernel-instance.
  - **clSVMAlloc**, **clSVMFree** and **clEnqueueSVMFree** to allocate and free memory for use with SVM.
  - **clEnqueueSVMMap** and **clEnqueueSVMUnmap** to map and unmap to update regions of an SVM buffer from host.
  - **clEnqueueSVMMemcpy** and **clEnqueueSVMMemFill** to copy or fill SVM memory regions.
- Device queues used to enqueue kernels on the device.
  - **clCreateCommandQueueWithProperties** is added to allow creation of a command-queue with properties that affect both host command-queues and device queues.
- Pipes.
  - **clCreatePipe** and **clGetPipeInfo** have been added to the API for host side creation and querying of pipes.
- Images support for 2D image from buffer, depth images and sRGB images.
- **clCreateSamplerWithProperties**.

The following modifications are made to the OpenCL 2.0 platform layer and runtime (sections 4 and 5):

- All API calls except **clSetKernelArg**, **clSetKernelArgSVMPointer** and **clSetKernelExecInfo** are thread-safe. Note that this statement does not imply that other API calls were not thread-safe in earlier versions of the specification.

The following features are added to the OpenCL C programming language (*section 6*) in OpenCL 2.0:

- Clang Blocks.
- Kernels enqueuing kernels to a device queue.
- Program scope variables in global address space.
- Generic address space.
- C1x atomics.
- New built-in functions (sections 6.15.10, 6.15.12, and 6.15.16).
- Support images with the read_write qualifier.
- 3D image writes are a core feature.
- The `CL_VERSION_2_0` and `NULL` macros.
- The `opencl_unroll_hint` attribute.

The following APIs are deprecated (see glossary) in OpenCL 2.0:

- The **clCreateCommandQueue** API has been deprecated to simplify the API. The **clCreateCommandQueueWithProperties** API provides equivalent functionality and supports specifying additional command-queue properties.
- The **clCreateSampler** API has been deprecated to simplify the API. The **clCreateSamplerWithProperties** API provides equivalent functionality and supports specifying additional sampler properties.
- The **clEnqueueTask** API has been deprecated to simplify the API. The **clEnqueueNDRangeKernel** API provides equivalent functionality.

The following queries are deprecated (see glossary) in OpenCL 2.0:

- The `CL_DEVICE_HOST_UNIFIED_MEMORY` query is deprecated. This query was purely informational and had different meanings for different implementations. Its use is no longer recommended.
- The `CL_IMAGE_BUFFER` query has been deprecated to simplify the API. The `CL_MEM_ASSOCIATED_MEMOBJECT` query provides equivalent functionality.
- The `CL_DEVICE_QUEUE_PROPERTIES` query has been deprecated and replaced by `CL_DEVICE_QUEUE_ON_HOST_PROPERTIES`.
- Atomics and Fences
  - The Explicit Memory Fence Functions defined in section 6.12.9 of the OpenCL 1.2 specification have been deprecated to simplify the programming language. The **atomic_work_item_fence** function provides equivalent functionality. The deprecated functions are still described in section 6.15.9 of this specification.
  - The Atomic Functions defined in section 6.12.11 of the OpenCL 1.2 specification have been deprecated to simplify the programming language. The **atomic_fetch** and modify functions provide equivalent functionality. The deprecated functions are still described in section 6.15.12.8 of this specification.

Summary of Changes from OpenCL 2.0 to OpenCL 2.1

The following features are added to the OpenCL 2.1 platform layer and runtime (*sections 4 and 5*):

- **clGetKernelSubGroupInfo** API call.
- `CL_KERNEL_MAX_NUM_SUB_GROUPS`, `CL_KERNEL_COMPILE_NUM_SUB_GROUPS` additions to table 5.21 of the API specification.
- **clCreateProgramWithIL** API call.
- **clGetHostTimer** and **clGetDeviceAndHostTimer** API calls.
- **clEnqueueSVMMigrateMem** API call.
- **clCloneKernel** API call.
- **clSetDefaultDeviceCommandQueue** API call.
- `CL_PLATFORM_HOST_TIMER_RESOLUTION` added to table 4.1 of the API specification.
- `CL_DEVICE_IL_VERSION`, `CL_DEVICE_MAX_NUM_SUB_GROUPS`, `CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS` added to table 4.3 of the API specification.
- `CL_PROGRAM_IL` to table 5.17 of the API specification.
- `CL_QUEUE_DEVICE_DEFAULT` added to table 5.2 of the API specification.
- Added table 5.22 to the API specification with the enums: `CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE`, `CL_KERNEL_SUB_GROUP_COUNT_FOR_NDRANGE` and `CL_KERNEL_LOCAL_SIZE_FOR_SUB_GROUP_COUNT`

The following modifications are made to the OpenCL 2.1 platform layer and runtime (sections 4 and 5):

- All API calls except **clSetKernelArg**, **clSetKernelArgSVMPointer**, **clSetKernelExecInfo** and **clCloneKernel** are thread-safe. Note that this statement does not imply that other API calls were not thread-safe in earlier versions of the specification.

Note that the OpenCL C kernel language is not updated for OpenCL 2.1. The OpenCL 2.0 kernel language will still be consumed by OpenCL 2.1 runtimes.

The SPIR-V and OpenCL SPIR-V Environment specifications have been added. OpenCL 2.1 requires support for the SPIR-V 1.0 intermediate language.

Summary of Changes from OpenCL 2.1 to OpenCL 2.2

The following changes have been made to the OpenCL 2.2 execution model (section 3)

- Added the third prerequisite (executing non-trivial constructors for program scope global variables).

The following features are added to the OpenCL 2.2 platform layer and runtime (*sections 4 and 5*):

- **clSetProgramSpecializationConstant** API call
- **clSetProgramReleaseCallback** API call

- Queries for CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT and CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT

The following modifications are made to the OpenCL 2.2 platform layer and runtime (section 4 and 5):

- Modified description of CL_DEVICE_MAX_CLOCK_FREQUENCY query.
- Added a new error code CL_MAX_SIZE_RESTRICTION_EXCEEDED to **clSetKernelArg** API call

Added definition of Deprecation and Specialization constants to the glossary.

OpenCL 2.2 requires support for the SPIR-V 1.0, SPIR-V 1.1, and SPIR-V 1.2 intermediate languages.

Summary of Changes from OpenCL 2.2 to OpenCL 3.0

OpenCL 3.0 is a major revision that breaks backwards compatibility with previous versions of OpenCL, see OpenCL 3.0 Backwards Compatibility for details.

OpenCL 3.0 adds new queries to determine optional capabilities for a device:

- CL_DEVICE_ATOMIC_MEMORY_CAPABILITIES and CL_DEVICE_ATOMIC_FENCE_CAPABILITIES to determine the atomic memory and atomic fence capabilities of a device.
- CL_DEVICE_NON_UNIFORM_WORK_GROUP_SUPPORT to determine if a device supports non-uniform work-group sizes.
- CL_DEVICE_WORK_GROUP_COLLECTIVE_FUNCTIONS_SUPPORT to determine whether a device supports optional work-group collective functions, such as broadcasts, scans, and reductions.
- CL_DEVICE_GENERIC_ADDRESS_SPACE_SUPPORT to determine whether a device supports the generic address space.
- CL_DEVICE_DEVICE_ENQUEUE_CAPABILITIES to determine the device-side enqueue capabilities of a device.
- CL_DEVICE_PIPE_SUPPORT to determine whether a device supports pipe memory objects.
- CL_DEVICE_PREFERRED_WORK_GROUP_SIZE_MULTIPLE to determine the preferred work-group size multiple for a device.

OpenCL 3.0 adds new queries to conveniently and precisely describe supported features and versions:

- CL_PLATFORM_NUMERIC_VERSION to describe the platform version as a numeric value.
- CL_PLATFORM_EXTENSIONS_WITH_VERSION to describe supported platform extensions and their supported version.
- CL_DEVICE_NUMERIC_VERSION to describe the device version as a numeric value.
- CL_DEVICE_EXTENSIONS_WITH_VERSION to describe supported device extensions and their supported version.
- CL_DEVICE_ILS_WITH_VERSION to describe supported intermediate languages (ILs) and their supported version.
- CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION to describe supported built-in kernels and their

supported version.

OpenCL 3.0 adds a new API to register a function that will be called when a context is destroyed, enabling an application to safely free user data associated with a context callback function.

- **clSetContextDestructorCallback**

OpenCL 3.0 adds two new APIs to support creating buffer and image memory objects with additional properties. Although no new properties are added in OpenCL 3.0, these APIs enable new buffer and image extensions to be added easily and consistently:

- **clCreateBufferWithProperties**
- **clCreateImageWithProperties**

OpenCL 3.0 adds new queries for the properties arrays specified when creating buffers, images, pipes, samplers, and command-queues:

- `CL_MEM_PROPERTIES`
- `CL_PIPE_PROPERTIES`
- `CL_SAMPLER_PROPERTIES`
- `CL_QUEUE_PROPERTIES_ARRAY`

Program initialization and clean-up kernels are not supported in OpenCL 3.0 due to implementation complexity and lack of demand. The following APIs and queries for program initialization and clean-up kernels are deprecated in OpenCL 3.0:

- `CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT`
- `CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT`
- **clSetProgramReleaseCallback**

OpenCL 3.0 adds the OpenCL 3.0 C kernel language, which includes feature macros to describe OpenCL C language support. Please refer to the OpenCL C specification for details.

Scalar input arguments to the **any** and **all** built-in functions have been deprecated in the OpenCL 3.0 C kernel language. These functions behaved inconsistently with the C language's use of scalar integers as logical values.

OpenCL 3.0 adds new queries to determine supported OpenCL C language versions and supported OpenCL C features:

- `CL_DEVICE_OPENCL_C_ALL_VERSIONS` to determine the set of OpenCL C language versions supported by a device.
- `CL_DEVICE_OPENCL_C_FEATURES` to determine optional OpenCL C language features supported by a device.

OpenCL 3.0 adds an event command type to identify events associated with the OpenCL 2.1 command **clEnqueueSVMMigrateMem**:

- `CL_COMMAND_SVM_MIGRATE_MEM`

OpenCL 3.0 adds a new query to determine the latest version of the conformance test suite that the device has fully passed in accordance with the official conformance process:

- `CL_DEVICE_LATEST_CONFORMANCE_VERSION_PASSED`

The first non-experimental version of the OpenCL 3.0 specifications was **v3.0.5**.

Changes from **v3.0.5** to **v3.0.6**:

- Fixed the calculation in "mapping work-items onto an ND-range".
- Added new extensions:
  - `cl_khr_extended_versioning`
  - `cl_khr_subgroup_extended_types`
  - `cl_khr_subgroup_non_uniform_vote`
  - `cl_khr_subgroup_ballot`
  - `cl_khr_subgroup_non_uniform_arithmetic`
  - `cl_khr_subgroup_shuffle`
  - `cl_khr_subgroup_shuffle_relative`
  - `cl_khr_subgroup_clustered_reduce`

Changes from **v3.0.6** to **v3.0.7**:

- Removed erroneous condition for `CL_INVALID_KERNEL_ARGS`.
- Fixed the spelling of `-cl-no-signed-zeros`.
- Clarified the table structure in the backwards compatibility appendix.
- Clarified that `-cl-unsafe-math-optimizations` also implies `-cl-denorms-are-zero`.
- Added new extensions:
  - `cl_khr_extended_bit_ops`
  - `cl_khr_pci_bus_info`
  - `cl_khr_spirv_extended_debug_info`
  - `cl_khr_spirv_linkonce_odr`
  - `cl_khr_suggested_local_work_size`

Changes from **v3.0.7** to **v3.0.8**:

- Clarified optionality support for double-precision literals.
- Removed unnecessary phrase from sub-group mask function descriptions.
- Added *input_slice_pitch* error condition for read and write image APIs.
- Added new extension:

    - `cl_khr_integer_dot_product`

Changes from **v3.0.8** to **v3.0.9**:

- Added a missing error condition for **clGetKernelSuggestedLocalWorkSizeKHR**.
- Clarified requirements for `CL_DEVICE_DOUBLE_FP_CONFIG` prior to OpenCL 2.0.
- Clarified the behavior of ballot operations for remainder sub-groups.
- Added new extensions:
    - `cl_khr_integer_dot_product` (version 2)
    - `cl_khr_semaphore` (experimental)
    - `cl_khr_external_semaphore` (experimental)
    - `cl_khr_external_semaphore_dx_fence` (experimental)
    - `cl_khr_external_semaphore_opaque_fd` (experimental)
    - `cl_khr_external_semaphore_sync_fd` (experimental)
    - `cl_khr_external_semaphore_win32` (experimental)
    - `cl_khr_external_memory` (experimental)
    - `cl_khr_external_memory_dma_buf` (experimental)
    - `cl_khr_external_memory_dx` (experimental)
    - `cl_khr_external_memory_opaque_fd` (experimental)
    - `cl_khr_external_memory_win32` (experimental)

Changes from **v3.0.9** to **v3.0.10**:

- Relaxed memory object acquire error checking requirements for OpenGL, EGL, and DirectX interop extensions.
- Added a missing error condition for **clGetSemaphoreHandleForTypeKHR**.
- Clarified that **clCompileProgram** is valid for programs created from SPIR.
- Documented the possible state of a kernel object after a failed call to **clSetKernelArg**.
- Added new extensions:
    - `cl_khr_async_work_group_copy_fence` (final)
    - `cl_khr_extended_async_copies` (final)
    - `cl_khr_expect_assume`
    - `cl_khr_command_buffer` (experimental)

Changes from **v3.0.10** to **v3.0.11**:

- Added a requirement for implementations supporting device-side enqueue to also support program scope global variables.
- Added missing device scope atomic feature guards to several atomic function overloads.

- Added a possible error condition for **clGetEventProfilingInfo** for pre-OpenCL 3.0 devices.
- Added several missing error conditions for **clGetKernelSubGroupInfo**.
- Clarified the expected return value for the of `CL_IMAGE_ROW_PITCH` and `CL_IMAGE_SLICE_PITCH` queries.
- Updated descriptions of the extended async copies functions to remove references to nonexistent function arguments.
- Clarified that the extended versioning extension is a core OpenCL 3.0 feature.
- Clarified sub-group clustered reduction behavior when the cluster size is not an integer constant or a power of two.
- Added new extensions:
  - `cl_khr_subgroup_rotate`
  - `cl_khr_work_group_uniform_arithmetic`

Changes from **v3.0.11** to **v3.0.12**:

- Added a definition for a valid object and requirements for testing for valid objects.
- Added a maximum limit for the number of arguments supported by a kernel.
- Clarified requirements for comparability and uniqueness of object handles.
- Clarified behavior for invalid device-side enqueue `clk_event_t` handles.
- Clarified `cl_khr_command_buffer` interactions with other extensions.
- Specified error behavior when a command buffer is finalized multiple times.
- Added new extension:
  - `cl_khr_command_buffer_mutable_dispatch` (experimental)

Changes from **v3.0.12** to **v3.0.13**:

- Fixed the accuracy requirements description for half-precision math functions (those prefixed by `half_`).
- Clarified that the semaphore type must always be provided when creating a semaphore.
- Removed an unnecessary and contradictory error condition when creating a semaphore.
- Added an issue regarding non-linear image import to the `cl_khr_external_memory` extension.
- Added missing calls to **clBuildProgram** to the `cl_khr_command_buffer` and `cl_khr_command_buffer_mutable_dispatch` sample code.
- Fixed a copy-paste error in the extensions quick reference appendix.
- Fixed typos and improved formatting consistency in the extensions spec.

Changes from **v3.0.13** to **v3.0.14**:

- Corrected the precision for `cross` and `dot` to be based on `HALF_EPSILON` in `cl_khr_fp16`, see #893.
- Added a context query for command-buffers to `cl_khr_command_buffer`, see #899.

- Updated the semaphore wait and signal rules for binary semaphores in `cl_khr_semaphore`, see #882.
- Removed redundant error conditions from `cl_khr_external_semaphore` and `cl_khr_external_memory`, see #903 and #904.
- Added new extension:
  - `cl_khr_command_buffer_multi_device` (experimental)

Changes from **v3.0.14** to **v3.0.15**:

- Clarified which error code should be returned when calling **clCreateBuffer** with a pointer to an SVM allocation that is too small, see #879.
- Improved capitalization and hyphenation consistency throughout the specs, see #902.
- Clarified that SVM is optional for all OpenCL 3.0 devices, see #913.
- Clarified that **clSetCommandQueueProperty** is only required for OpenCL 1.0 devices and may return an error otherwise, see #980.
- Clarified that the application must ensure the free function passed to **clEnqueueSVMFree** is thread safe, see #1016.
- Clarified that the application must ensure the user function passed to **clEnqueueNativeKernel** is thread safe, see #1026.
- `cl_khr_command_buffer` (experimental):
  - Removed the "invalid" command buffer state, see #885.
  - Added support for recording SVM memory copies and memory fills in a command buffer, see #915.
- `cl_khr_command_buffer_multi_device` (experimental):
  - Clarified that the sync devices query should only return root devices, see #925.
- `cl_khr_external_memory` (experimental):
  - Disallowed specifying a device handle list without also specifying an external memory handle, see #922.
  - Added a query to determine the handle types an implementation will assume have a linear memory layout, see #940.
  - Added an external memory-specific device handle list enum, see #956.
  - Clarified that implementations may acquire information about an image from an external memory handle when the image is created, see #970.
- `cl_khr_external_semaphore` (experimental):
  - Added the ability to re-import "sync fd" handles into an existing semaphore, see #939.
  - Clarified that a semaphore may only export one handle type, and that a semaphore created from an external handle cannot also export a handle, see #975.
  - Clarified that `cl_khr_external_semaphore` requires support for `cl_khr_semaphore`, see #976.
  - Added a query to determine if a semaphore may export an external handle, see #997.

- `cl_khr_semaphore` (experimental):
  - Added an semaphore-specific device handle list enum, see #956.
  - Restricted semaphores to a single associated device, see #996.
- `cl_khr_subgroup_rotate`:
  - Clarified that only rotating within a sub-group is supported, see #967.

Changes from **v3.0.15** to **v3.0.16**:

- Moved all KHR extension text out of the OpenCL Extension specification and into the main specifications. The OpenCL Extension specification will be removed in a subsequent revision.
- Clarified several error conditions that could return `CL_INVALID_PLATFORM`, see #1063.
- Strengthened requirements for the `CL_DEVICE_TYPE` query, see #1069.
- Clarified **clSetEventCallback** behavior for command errors, see #1071.
- Moved footnote text for `CL_KERNEL_ARG_TYPE_QUALIFIER` into the main spec, see #1097.
- `cl_khr_command_buffer_mutable_dispatch` (experimental):
  - Added `CL_MUTABLE_DISPATCH_ASSERTS_KHR`, see #992.
- `cl_khr_semaphore`:
  - Removed a redundant error condition, see #1052
- The following extensions have been finalized and are no longer experimental:
  - `cl_khr_semaphore`
  - `cl_khr_external_semaphore`
  - `cl_khr_external_semaphore_opaque_fd`
  - `cl_khr_external_semaphore_sync_fd`
  - `cl_khr_external_memory`
  - `cl_khr_external_memory_dma_buf`
  - `cl_khr_external_memory_opaque_fd`
  - `cl_khr_external_memory_win32`
- Added new extension:
  - `cl_khr_kernel_clock` (experimental)

Changes from **v3.0.16** to **v3.0.17**:

- Clarified the definition of command prerequisites, see #923.
- Clarified the behavior of `CL_DEVICE_TYPE_DEFAULT` and `CL_DEVICE_TYPE_ALL` for custom devices, see #1117.
- Clarified how `CL_DEVICE_ATOMIC_SCOPE_ALL_DEVICES` behaves for devices that do not support `CL_DEVICE_SVM_ATOMICS`, see #1171.
- Fixed links to extension API functions, see #1179.

- Further clarified an error condition for **clCreateBuffer** with `CL_MEM_COPY_HOST_PTR` and an SVM pointer that is too small, see #1189.
- Fixed a minor typo in the **clCreateProgramWithSource** introduction, see #1204.
- Clarified how to properly use and modify OpenCL objects across multiple command-queues, see #1243.
- Clarified and corrected many parts of **clSetKernelExecInfo**, see #1245.
- Improved wording consistency for *param_value_size* parameters, see #1254.
- Clarified the meaning of *num_mip_levels* in `cl_image_desc`, see #1255 and #1272.
- Clarified that functionality will never be removed in minor OpenCL specification revisions, see #1265.
- Clarified that the minimum value for `CL_DEVICE_HALF_FP_CONFIG` applies to all OpenCL versions, see #1273.
- `cl_khr_command_buffer` (experimental):
  - Added multi-device wording to **clCommandBarrierWithWaitListKHR**, see #1146.
  - Fixed `CL_INVALID_CONTEXT` command-buffer error definitions, see #1149.
  - Added a *properties* parameter to all command-buffer commands to improve extensibility, see #1215.
- `cl_khr_command_buffer_mutable_dispatch` (experimental):
  - Modified the extension to pass update configs as arrays, rather than linked lists, see #1045.
- `cl_khr_external_memory`:
  - Clarified acquire and release behavior, see #1176.
  - Added a mechanism to import NT handles by name, see #1177.
  - Documented which error condition should be returned when attempting to create a memory object with more than one external handle, see #1249.
- `cl_khr_external_semaphore`:
  - Added a mechanism to import NT handles by name, see #1177.
  - Fixed a typo in the description of **clGetSemaphoreHandleForTypeKHR**, see #1220.
  - Clarified that there are no implicit dependencies when waiting on or signaling semaphores using out-of-order queues, see #1231.
  - Documented which error condition should be returned when attempting to create a semaphore with more than one external handle, see #1249.
  - Unified the `CL_INVALID_COMMAND_QUEUE` behavior for semaphore signals and waits, see #1256.
  - Clarified that **clGetSemaphoreHandleForTypeKHR** is part of `cl_khr_external_semaphore` and not `cl_khr_external_semaphore_sync_fd`, see #1257.
- `cl_khr_external_semaphore_sync_fd`:
  - Fixed typos in the description of **clReImportSemaphoreSyncFdKHR**, see #1208.
  - Clarified which re-import properties are accepted by **clReImportSemaphoreSyncFdKHR**,

see #1219.

- `cl_khr_semaphore`:
  - Clarified external semaphore behavior, removing references to permanence, see #938.
- Removed experimental extensions due to lack of implementations and tests, see #1160.
  - `cl_khr_external_semaphore_dx_fence` (experimental)
  - `cl_khr_external_memory_dx` (experimental)

Changes from **v3.0.17** to **v3.0.18**:

- Added version 2.0 of the extension for loader-managed dispatch, see #1005.
- Added the `cl_ext_buffer_device_address` extension, see #1159.
- Clarified that `cl_mem_flags` do not affect copies, see #1230.
- Added the `cl_ext_immutable_memory_objects` extension, see #1280.
- Integrated the `cl_ext_image_requirements_info` extension into the unified specification, see #1295.
- Integrated the `cl_ext_image_from_buffer` extension into the unified specification, see #1299.
- Improved the description of error codes for **clSetKernelArg**, see #1319.
- Clarified that in-flight commands may be abnormally terminated when a context is deleted, see #1341.
- Clarified an error condition for Direct3D 10 and Direct3D 11 3D textures, see #1342.
- Clarified `CL_INVALID_IMAGE_SIZE` error conditions, see #1343.
- Added a section describing document conventions, see #1347.
- `cl_khr_command_buffer` (experimental):
  - Refactored the command-buffer query for supported queue properties, see #850.
  - Documented behavior when the command-buffer capacity is reached, see #1286.
  - Refactored command-buffer queue compatibility, see #1292.
  - Documented that **clFinalizeCommandBufferKHR** is not thread-safe, see #1345.
- `cl_khr_command_buffer_mutable_dispatch` (experimental):
  - Clarified how implementations may defer updates to the command-buffer, see #1315.
  - Documented that **clUpdateMutableCommandsKHR** is not thread-safe, see #1345.
- `cl_khr_external_memory`:
  - Improved documentation for **clEnqueueReleaseExternalMemObjectsKHR**, see #1332.
- `cl_khr_external_memory_dma_buf`:
  - Clarified usage, see #1335.

Changes from **v3.0.18** to **v3.0.19**:

- Moved description of restrictions for image kernel arguments to **clSetKernelArg**, see #1346.

- Fixed document linking for several extension APIs and enums, see #1360.
- Fixed a wording inconsistency for device and host timer synchronization, see #1364.
- Clarified the conditions when the global work size may be null or zero, see #1381.
- Switched to the term "experimental" to describe extensions that are subject to change, see #1383.
- Removed specific restrictions for custom devices, see #1388.
- Clarified `CL_FP_FMA` description to avoid ambiguity, see #1391.
- `cl_khr_command_buffer_mutable_dispatch` (experimental):
  - Clarified that the mutable dispatch work dimension may be set to zero to indicate that it is unchanged, see #1395.
- `cl_khr_external_semaphore`:
  - Clarified semaphore behavior when payloads are required, see #1362.
- Re-added a previously removed experimental extension, see #1359.
  - `cl_khr_external_semaphore_dx_fence` (experimental)
- The following extension has been finalized and is no longer experimental:
  - `cl_khr_kernel_clock`
- Added new extensions:
  - `cl_khr_external_memory_android_hardware_buffer` (experimental)
  - `cl_khr_spirv_queries`

Summary of Changes from OpenCL 3.0 to OpenCL 3.1

OpenCL 3.1 adds the OpenCL 3.1 C kernel language. Please refer to the OpenCL C specification for details.

OpenCL 3.1 requires support for the SPIR-V 1.0, SPIR-V 1.1, SPIR-V 1.2, SPIR-V 1.3, and SPIR-V 1.4 intermediate languages. Please refer to the OpenCL SPIR-V Environment specification for details.

OpenCL 3.1 removes the OpenCL Extension Specification. The OpenCL Extension Specification is no longer needed now that EXT and KHR extensions are documented in the main OpenCL specifications. See #1516.

Other changes in OpenCL 3.1:

- Added required support for sub-groups, see `CL_DEVICE_MAX_NUM_SUB_GROUPS`.
- Relaxed the definition of inclusive scopes, see internal issue 367.
- Clarified and un-deprecated the `CL_DEVICE_HOST_UNIFIED_MEMORY` query, see internal issue 370.
- Updated the memory model so observing that an event is `CL_COMPLETE` is a synchronization point, see internal issue 373.
- Allowed **clSetKernelArg** to set a local memory kernel argument to zero, see internal issue 374.

- Deprecated the confusingly named `CL_DEVICE_MAX_WORK_ITEM_SIZES` query, use `CL_DEVICE_MAX_WORK_GROUP_SIZES` instead, see internal issue 375.
- Cleaned up inconsistencies in the error condition descriptions for **clSetKernelExecInfo**, see #1419.
- Improved error code documentation consistency for many OpenCL APIs, see #1439 and others.
- Added missing error conditions for **clCompileProgram** and **clLinkProgram**, see #1453.
- Clarified the description of sub-group functions, see #1483.
- Refactored and clarified the description for **clSetKernelArg**, see #1493.
- Clarified the description of atomic operations, see #1500.
- Cleaned up a few more descriptions of custom devices, see #1540.
- Added missing error condition for **clEnqueueNDRangeKernel** when the local work-group size is zero, see #1542.
- `cl_khr_command_buffer_mutable_dispatch` (experimental):
  - Relaxed the requirement to set all kernel arguments before recording a command buffer, see #1382.
  - Redefined and clarified command-buffer simultaneous use, see #1411.
- `cl_khr_external_memory_android_hardware_buffer` (experimental)
  - Clarified that images cannot be created if the format is `AHARDWAREBUFFER_FORMAT_BLOB`, see #1477.
-
  - Added a missing error condition for **clSetKernelArgDevicePointerEXT**, see #1492.
- Added new extensions:
  - `cl_khr_unified_svm` (experimental)
- Promoted the following extensions to the core API:
  - `cl_khr_device_uuid`
  - `cl_khr_extended_bit_ops`
  - `cl_khr_integer_dot_product`
  - `cl_khr_spirv_queries`
  - `cl_khr_subgroup_extended_types`
  - `cl_khr_subgroup_rotate`
  - `cl_khr_subgroup_shuffle`
  - `cl_khr_subgroup_shuffle_relative`
  - `cl_khr_suggested_local_work_size`

Changes from **v3.1.0** to **v3.1.1**:

- Reverted the change where observing that an event is `CL_COMPLETE` is a synchronization point due to possible performance regressions, see internal issue 386 and #1558.

Appendix F: Error Codes

This section lists OpenCL error codes and their meanings.

Error Code	Brief Description
CL_SUCCESS	This is a special error code to indicate that the API executed successfully, without errors.
CL_BUILD_PROGRAM_FAILURE	Returned when **clBuildProgram** failed to build the specified program.
CL_COMPILE_PROGRAM_FAILURE missing before version 1.2.	Returned when **clCompileProgram** failed to compile the specified program.
CL_COMPILER_NOT_AVAILABLE	Returned when compiling or building a program from source or IL when CL_DEVICE_COMPILER_AVAILABLE is CL_FALSE.
CL_DEVICE_NOT_FOUND	Returned when no devices were found that match the specified device type.
CL_DEVICE_NOT_AVAILABLE	Returned when attempting to use a device when CL_DEVICE_AVAILABLE is CL_FALSE.
CL_DEVICE_PARTITION_FAILED missing before version 1.2.	Returned when device partitioning is supported but the device could not be further partitioned.
CL_EXEC_STATUS_ERROR_FOR_EVENTS_IN_WAIT_LIST missing before version 1.1.	Returned by blocking APIs when an event in the event wait list has a negative value, indicating it is in an error state.
CL_IMAGE_FORMAT_MISMATCH	Returned when attempting to copy images that do not use the same image format.
CL_IMAGE_FORMAT_NOT_SUPPORTED	Returned when attempting to create or use an image format that is not supported.
CL_INVALID_ARG_INDEX	Returned when attempting to get or set a kernel argument using an invalid index for the specified kernel.
CL_INVALID_ARG_SIZE	Returned when the specified size of a kernel argument does not match the size of the kernel argument.
CL_INVALID_ARG_VALUE	Returned when attempting to set a kernel argument that is not valid.
CL_INVALID_BINARY	Returned when a program binary is not valid for a device.
CL_INVALID_BUFFER_SIZE	Returned when attempting to create a buffer or a sub-buffer with an invalid size.
CL_INVALID_BUILD_OPTIONS	Returned when build options passed to **clBuildProgram** are not valid.

Error Code	Brief Description
CL_INVALID_COMMAND_QUEUE	Returned when the specified command-queue is not a valid command-queue.
CL_INVALID_COMPILER_OPTIONS missing before version 1.2.	Returned when compiler options passed to **clCompileProgram** are not valid.
CL_INVALID_CONTEXT	Returned when a specified context is not a valid context, or when mixing objects from multiple contexts.
CL_INVALID_DEVICE	Returned when a specified device is not a valid device.
CL_INVALID_DEVICE_PARTITION_COUNT missing before version 1.2.	Returned when the requested device partitioning using CL_DEVICE_PARTITION_BY_COUNTS is not valid.
CL_INVALID_DEVICE_QUEUE missing before version 2.0.	Returned when setting a device queue kernel argument to a value that is not a valid device command-queue.
CL_INVALID_DEVICE_TYPE	Returned when the requested device type is not a valid value.
CL_INVALID_EVENT	Returned when a specified event object is not a valid event object.
CL_INVALID_EVENT_WAIT_LIST	Returned when the specified event wait list or number of events in the wait list is not valid.
CL_INVALID_GLOBAL_OFFSET	Returned when the specified global offset and global work size exceeds the limits of the device.
CL_INVALID_GLOBAL_WORK_SIZE	Returned when the specified global work size exceeds the limits of the device.
CL_INVALID_HOST_PTR	Returned when the specified host pointer is not valid for the specified flags.
CL_INVALID_IMAGE_DESCRIPTOR missing before version 1.2.	Returned when the specified image descriptor is NULL or specifies invalid values.
CL_INVALID_IMAGE_FORMAT_DESCRIPTOR	Returned when the specified image format descriptor is NULL or specifies invalid value.
CL_INVALID_IMAGE_SIZE	Returned when the specified image dimensions exceed the maximum dimensions for a device or all devices in a context.
CL_INVALID_KERNEL	Returned when the specified kernel is not a valid kernel object.
CL_INVALID_KERNEL_ARGS	Returned when enqueing a kernel when some kernel arguments have not been set or are invalid.

Error Code	Brief Description
CL_INVALID_KERNEL_DEFINITION	Returned when creating a kernel for multiple devices where the number of kernel arguments or kernel argument types are not the same for all devices.
CL_INVALID_KERNEL_NAME	Returned when creating a kernel when no kernel with the specified name exists in the program object.
CL_INVALID_LINKER_OPTIONS missing before version 1.2.	Returned when build options passed to **clLinkProgram** are not valid.
CL_INVALID_MEM_OBJECT	Returned when a specified memory object is not a valid memory object.
CL_INVALID_OPERATION	This is a generic error code that is returned when the requested operation is not a valid operation.
CL_INVALID_PIPE_SIZE missing before version 2.0.	Returned when attempting to create a pipe with an invalid packet size or number of packets.
CL_INVALID_PLATFORM	Returned when the specified platform is not a valid platform.
CL_INVALID_PROGRAM	Returned when a specified program is not a valid program object.
CL_INVALID_PROGRAM_EXECUTABLE	Returned when the specified program is valid but has not been successfully built.
CL_INVALID_PROPERTY missing before version 1.1.	Returned when a specified property name is invalid, when the value for a property name is invalid, or when the same property name is specified more than once.
CL_INVALID_QUEUE_PROPERTIES	Returned when specified queue properties are valid but are not supported by the device.
CL_INVALID_SAMPLER	Returned when a specified sampler is not a valid sampler object.
CL_INVALID_SPEC_ID missing before version 2.2.	Returned when the specified specialization constant ID is not valid for the specified program.
CL_INVALID_VALUE	This is a generic error that is returned when a specified value is not a valid value.
CL_INVALID_WORK_DIMENSION	Returned by **clEnqueueNDRangeKernel** when the specified work dimension is not valid.
CL_INVALID_WORK_GROUP_SIZE	Returned by **clEnqueueNDRangeKernel** when the specified total work-group size is not valid for the specified kernel or device.

Error Code	Brief Description
CL_INVALID_WORK_ITEM_SIZE	Returned by **clEnqueueNDRangeKernel** when the specified work-group size in one dimension is not valid for the device.
CL_KERNEL_ARG_INFO_NOT_AVAILABLE missing before version 1.2.	Returned by **clGetKernelArgInfo** when kernel argument information is not available for the specified kernel.
CL_LINK_PROGRAM_FAILURE missing before version 1.2.	Returned by **clLinkProgram** when there is a failure to link the specified binaries or libraries.
CL_LINKER_NOT_AVAILABLE missing before version 1.2.	Returned by **clLinkProgram** when CL_DEVICE_LINKER_AVAILABLE is CL_FALSE.
CL_MAP_FAILURE	Returned when there is a failure to map the specified region into the host address space.
CL_MEM_COPY_OVERLAP	Returned when copying from one region of a memory object to another where the source and destination regions overlap.
CL_MEM_OBJECT_ALLOCATION_FAILURE	Returned when there is a failure to allocate memory for a memory object.
CL_MISALIGNED_SUB_BUFFER_OFFSET missing before version 1.1.	Returned when a sub-buffer object is created or used that is not aligned to CL_DEVICE_MEM_BASE_ADDR_ALIGN for the device.
CL_OUT_OF_HOST_MEMORY	This is a generic error that is returned when memory could not be allocated on the host.
CL_OUT_OF_RESOURCES	This is a generic error that is returned when resources could not be allocated on the device.
CL_MAX_SIZE_RESTRICTION_EXCEEDED missing before version 2.2.	Returned when the size of the specified kernel argument value exceeds the maximum size defined for the kernel argument.
CL_PROFILING_INFO_NOT_AVAILABLE	Returned by **clGetEventProfilingInfo** when the command associated with the specified event was not enqueued into a command-queue with CL_QUEUE_PROFILING_ENABLE.
CL_INVALID_COMMAND_BUFFER_KHR provided by the cl_khr_command_buffer extension.	Returned when the specified command-buffer is not a valid command-buffer.
CL_INVALID_SYNC_POINT_WAIT_LIST_KHR provided by the cl_khr_command_buffer extension.	Returned when the specified sync point wait list or number of sync points in the wait list is not valid.

Error Code	Brief Description
CL_INCOMPATIBLE_COMMAND_QUEUE_KHR provided by the cl_khr_command_buffer extension.	Returned when one or more command-queues is incompatible with a command-buffer.
CL_INVALID_MUTABLE_COMMAND_KHR provided by the cl_khr_command_buffer_mutable_dispatch extension.	Returned when a specified command is not a valid mutable-command object.
CL_INVALID_D3D10_DEVICE_KHR provided by the cl_khr_d3d10_sharing extension.	Returned when a Direct3D 10 device cannot interoperate with OpenCL device IDs.
CL_INVALID_D3D10_RESOURCE_KHR provided by the cl_khr_d3d10_sharing extension.	Returned when an OpenCL object cannot be created from a Direct3D 10 resource.
CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR provided by the cl_khr_d3d10_sharing extension.	Returned when attempting to acquire an OpenCL object created from a Direct3D 10 resource that was already acquired.
CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR provided by the cl_khr_d3d10_sharing extension.	Returned when attempting to release an OpenCL object created from a Direct3D 10 resource that has not been acquired.
CL_INVALID_D3D11_DEVICE_KHR provided by the cl_khr_d3d11_sharing extension.	Returned when a Direct3D 11 device cannot interoperate with OpenCL device IDs.
CL_INVALID_D3D11_RESOURCE_KHR provided by the cl_khr_d3d11_sharing extension.	Returned when an OpenCL object cannot be created from a Direct3D 11 resource.
CL_D3D11_RESOURCE_ALREADY_ACQUIRED_KHR provided by the cl_khr_d3d11_sharing extension.	Returned when attempting to acquire an OpenCL object created from a Direct3D 11 resource that was already acquired.
CL_D3D11_RESOURCE_NOT_ACQUIRED_KHR provided by the cl_khr_d3d11_sharing extension.	Returned when attempting to release an OpenCL object created from a Direct3D 11 resource that has not been acquired.

Error Code	Brief Description
`CL_INVALID_DX9_MEDIA_ADAPTER_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	Returned when a DirectX 9 media adapter cannot interoperate with OpenCL device IDs.
`CL_INVALID_DX9_MEDIA_SURFACE_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	Returned when an OpenCL object cannot be created from a DirectX 9 media surface.
`CL_DX9_MEDIA_SURFACE_ALREADY_ACQUIRED_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	Returned when attempting to acquire an OpenCL object created from a DirectX 9 media surface that was already acquired.
`CL_DX9_MEDIA_SURFACE_NOT_ACQUIRED_KHR` provided by the `cl_khr_dx9_media_sharing` extension.	Returned when attempting to release an OpenCL object created from a DirectX 9 media surface that has not been acquired.
`CL_EGL_RESOURCE_NOT_ACQUIRED_KHR` provided by the `cl_khr_egl_image` extension.	Possible event status if an EGL resource is used without being acquired.
`CL_INVALID_EGL_OBJECT_KHR` provided by the `cl_khr_egl_image` extension.	Returned when the specified EGL object is not valid.
`CL_INVALID_GL_OBJECT`	Returned when the specified OpenGL object is not valid, or when there is no associated OpenGL object for an OpenCL object.
`CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR` provided by the `cl_khr_gl_sharing` extension.	Returned when the specified OpenGL sharing context creation property is not valid.
`CL_PLATFORM_NOT_FOUND_KHR` provided by the `cl_khr_icd` extension.	Returned by **clGetPlatformIDs** when no platforms are available.
`CL_INVALID_SEMAPHORE_KHR` provided by the `cl_khr_semaphore` extension.	Returned when the specified semaphore is not a valid semaphore.

Error Code	Brief Description
`CL_CONTEXT_TERMINATED_KHR` provided by the `cl_khr_terminate_context` extension.	Returned when the specified context has already been terminated, or as an event status for terminated commands.

Appendix G: Other Miscellaneous Enums

This section lists other miscellaneous OpenCL enumerants and their meanings.

Enumerant	Brief Description
`CL_TRUE`	Indicates a boolean "true" value.
`CL_FALSE`	Indicates a boolean "false" value.
`CL_NONE`	Indicates that none of the other enumerations or conditions are applicable.
`CL_BLOCKING` missing before version 1.2.	Alias of `CL_TRUE` that can be used to improve the readability of calls to enqueue functions that can block.
`CL_NON_BLOCKING` missing before version 1.2.	Alias of `CL_FALSE` that can be used to improve the readability of calls to enqueue function that can block.

Appendix H: OpenCL 3.0 Backwards Compatibility

OpenCL 3.0 breaks backwards compatibility with earlier versions of OpenCL by making some features that were previously required for FULL_PROFILE or EMBEDDED_PROFILE devices optional. This appendix describes the features that were previously required that are now optional, how to detect whether an optional feature is supported, and expected behavior when an optional feature is not supported.

Informally, in the tables below the first row usually describes a feature detection mechanism ("May return this value indicating that the feature is not supported") and subsequent rows usually describe behavior when a feature is not supported ("Returns this value if the feature is not supported").

Shared Virtual Memory

Shared Virtual Memory (SVM) is optional for devices supporting OpenCL 3.0. When Shared Virtual Memory is not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_SVM_CAPABILITIES`	May return `0`, indicating that *device* does not support Shared Virtual Memory.
clGetMemObjectInfo, passing `CL_MEM_USES_SVM_POINTER`	Returns `CL_FALSE` if no devices in the context associated with *memobj* support Shared Virtual Memory.
clSVMAlloc	Returns `NULL` if no devices in *context* support Shared Virtual Memory.
clSVMFree	Is a NOP if no devices in *context* support Shared Virtual Memory.
clEnqueueSVMFree, **clEnqueueSVMMemcpy**, **clEnqueueSVMMemFill**, **clEnqueueSVMMap**, **clEnqueueSVMUnmap**, **clEnqueueSVMMigrateMem**	Returns `CL_INVALID_OPERATION` if the device associated with *command_queue* does not support Shared Virtual Memory.
clSetKernelArgSVMPointer, **clSetKernelExecInfo**	Returns `CL_INVALID_OPERATION` if no devices in the context associated with *kernel* support Shared Virtual Memory.

Memory Consistency Model

Some aspects of the OpenCL memory consistency model are optional for devices supporting OpenCL 3.0. New device queries were added to **clGetDeviceInfo** to allow capabilities to be precisely reported. When the full memory consistency model is not supported:

API	Behavior

clGetDeviceInfo, passing `CL_DEVICE_ATOMIC_MEMORY_CAPABILITIES`	May return:
	
	`CL_DEVICE_ATOMIC_ORDER_RELAXED` |
	`CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP`
	
	indicating that *device* does not support the full memory consistency model for atomic memory operations.
	
	Note that a device that provides the same level of capabilities as an OpenCL 2.x device would be expected to return:
	
	`CL_DEVICE_ATOMIC_ORDER_RELAXED` |
	`CL_DEVICE_ATOMIC_ORDER_ACQ_REL` |
	`CL_DEVICE_ATOMIC_ORDER_SEQ_CST` |
	`CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP` |
	`CL_DEVICE_ATOMIC_SCOPE_DEVICE` |
	`CL_DEVICE_ATOMIC_SCOPE_ALL_DEVICES`

clGetDeviceInfo, passing `CL_DEVICE_ATOMIC_FENCE_CAPABILITIES`	May return:
	
	`CL_DEVICE_ATOMIC_ORDER_RELAXED` |
	`CL_DEVICE_ATOMIC_ORDER_ACQ_REL` |
	`CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP`
	
	indicating that *device* does not support the full memory consistency model for atomic fence operations.
	
	Note that a device that provides the same level of capabilities as an OpenCL 2.x device would be expected to return:
	
	`CL_DEVICE_ATOMIC_ORDER_RELAXED` |
	`CL_DEVICE_ATOMIC_ORDER_ACQ_REL` |
	`CL_DEVICE_ATOMIC_ORDER_SEQ_CST` |
	`CL_DEVICE_ATOMIC_SCOPE_WORK_ITEM` |
	`CL_DEVICE_ATOMIC_SCOPE_WORK_GROUP` |
	`CL_DEVICE_ATOMIC_SCOPE_DEVICE` |
	`CL_DEVICE_ATOMIC_SCOPE_ALL_DEVICES`

OpenCL C compilers supporting atomics orders or scopes beyond the mandated minimum will define some or all of following feature macros as appropriate:

`__opencl_c_atomic_order_acq_rel` — Indicating atomic operations support acquire-release orderings.

`__opencl_c_atomic_order_seq_cst` — Indicating atomic operations and fences support acquire sequentially consistent orderings.

`__opencl_c_atomic_scope_device` — Indicating atomic operations and fences support device-wide memory ordering constraints.

`__opencl_c_atomic_scope_all_devices` — Indicating atomic operations and fences support all-device memory ordering constraints, across any host threads and all devices that can share SVM memory with each other and the host process.

Device-Side Enqueue

Device-side enqueue and on-device queues are optional for devices supporting OpenCL 3.0. When device-side enqueue is not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_DEVICE_ENQUEUE_CAPABILITIES`	May return `0`, indicating that *device* does not support device-side enqueue and on-device queues.
clGetDeviceInfo, passing `CL_DEVICE_QUEUE_ON_DEVICE_PROPERTIES`	Returns `0` if *device* does not support device-side enqueue and on-device queues.
clGetDeviceInfo, passing `CL_DEVICE_QUEUE_ON_DEVICE_PREFERRED_SIZE`, `CL_DEVICE_QUEUE_ON_DEVICE_MAX_SIZE`, `CL_DEVICE_MAX_ON_DEVICE_QUEUES`, or `CL_DEVICE_MAX_ON_DEVICE_EVENTS`	Returns `0` if *device* does not support device-side enqueue and on-device queues.
clGetCommandQueueInfo, passing `CL_QUEUE_SIZE`	Returns `CL_INVALID_COMMAND_QUEUE` since *command_queue* cannot be a valid device command-queue.
clGetCommandQueueInfo, passing `CL_QUEUE_DEVICE_DEFAULT`	Returns `NULL` if the device associated with *command_queue* does not support on-device queues.
clGetEventProfilingInfo, passing `CL_PROFILING_COMMAND_COMPLETE`	Returns a value equivalent to passing `CL_PROFILING_COMMAND_END` if the device associated with *event* does not support device-side enqueue.
clSetDefaultDeviceCommandQueue	Returns `CL_INVALID_OPERATION` if *device* does not support on-device queues.

When device-side enqueue is supported but a replaceable default on-device queue is not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_DEVICE_ENQUEUE_CAPABILITIES`	May omit `CL_DEVICE_QUEUE_REPLACEABLE_DEFAULT`, indicating that *device* does not support a replaceable default on-device queue.
clSetDefaultDeviceCommandQueue	Returns `CL_INVALID_OPERATION` if *device* does not support a replaceable default on-device queue.

OpenCL C compilers supporting device-side enqueue and on-device queues will define the feature macro `__opencl_c_device_enqueue`. OpenCL C compilers that define the feature macro `__opencl_c_device_enqueue` must also define the feature macro `__opencl_c_generic_address_space` because some

OpenCL C functions for device-side enqueue accept pointers to the generic address space. OpenCL C compilers that define the feature macro `__opencl_c_device_enqueue` must also define the feature macro `__opencl_c_program_scope_global_variables` because an implementation of blocks may interact with program scope variables in global address space as part of ABI.

Pipes

Pipe memory objects are optional for devices supporting OpenCL 3.0. When pipes are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_PIPE_SUPPORT`	May return `CL_FALSE`, indicating that *device* does not support pipes.
clGetDeviceInfo, passing `CL_DEVICE_MAX_PIPE_ARGS`, `CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS`, or `CL_DEVICE_PIPE_MAX_PACKET_SIZE`	Returns `0` if *device* does not support pipes.
clCreatePipe	Returns `CL_INVALID_OPERATION` if no devices in *context* support pipes.
clGetPipeInfo	Returns `CL_INVALID_MEM_OBJECT` since *pipe* cannot be a valid pipe object.

OpenCL C compilers supporting pipes will define the feature macro `__opencl_c_pipes`. OpenCL C compilers that define the feature macro `__opencl_c_pipes` must also define the feature macro `__opencl_c_generic_address_space` because some OpenCL C functions for pipes accept pointers to the generic address space.

Program Scope Global Variables

Program scope global variables are optional for devices supporting OpenCL 3.0. When program scope global variables are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_MAX_GLOBAL_VARIABLE_SIZE`	May return `0`, indicating that *device* does not support program scope global variables.
clGetDeviceInfo, passing `CL_DEVICE_GLOBAL_VARIABLE_PREFERRED_TOTAL_SIZE`	Returns `0` if *device* does not support program scope global variables.
clGetProgramBuildInfo, passing `CL_PROGRAM_BUILD_GLOBAL_VARIABLE_TOTAL_SIZE`	Returns `0` if *device* does not support program scope global variables.

OpenCL C compilers supporting program scope global variables will define the feature macro `__opencl_c_program_scope_global_variables`.

Non-Uniform Work-groups

Support for non-uniform work-groups is optional for devices supporting OpenCL 3.0. When non-uniform work-groups are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_NON_UNIFORM_WORK_GROUP_SUPPORT`	May return `CL_FALSE`, indicating that *device* does not support non-uniform work-groups.
clEnqueueNDRangeKernel	Behaves as though non-uniform work-groups were not enabled for *kernel*, if the device associated with *command_queue* does not support non-uniform work-groups.

Read-Write Images

Read-write images, that may be read from and written to in the same kernel, are optional for devices supporting OpenCL 3.0. When read-write images are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_MAX_READ_WRITE_IMAGE_ARGS`	May return `0`, indicating that *device* does not support read-write images.
clGetSupportedImageFormats, passing `CL_MEM_KERNEL_READ_AND_WRITE`	Returns an empty set (such as *num_image_formats* equal to `0`), indicating that no image formats are supported for reading and writing in the same kernel, if no devices in *context* support read-write images.

OpenCL C compilers supporting read-write images will define the feature macro `__opencl_c_read_write_images`.

Creating 2D Images From Buffers

Creating a 2D image from a buffer is optional for devices supporting OpenCL 3.0. When creating a 2D image from a buffer is not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_IMAGE_PITCH_ALIGNMENT` or `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT`	May return `0`, indicating that *device* does not support creating a 2D image from a Buffer.
clGetDeviceInfo, passing `CL_DEVICE_EXTENSIONS`	Will not describe support for the `cl_khr_image2d_from_buffer` extension if *device* does not support creating a 2D image from a buffer.

API	Behavior
clCreateImage or clCreateImageWithProperties, passing *image_type* equal to CL_MEM_OBJECT_IMAGE2D and *mem_object* not equal to NULL	Returns CL_INVALID_OPERATION if no devices in *context* support creating a 2D image from a buffer.

sRGB Images

All of the sRGB image channel orders (such as CL_sRGBA) are optional for devices supporting OpenCL 3.0. When sRGB images are not supported:

API	Behavior
clGetSupportedImageFormats	Will not return any image formats with image_channel_order equal to an sRGB image channel order if no devices in *context* support sRGB images.

Depth Images

The CL_DEPTH image channel order is optional for devices supporting OpenCL 3.0. When depth images are not supported:

API	Behavior
clGetSupportedImageFormats	Will not return any image formats with image_channel_order equal to CL_DEPTH if no devices in *context* support depth images.

Device and Host Timer Synchronization

Synchronizing the device and host timers is optional for platforms supporting OpenCL 3.0. When device and host timer synchronization is not supported:

API	Behavior
clGetPlatformInfo, passing CL_PLATFORM_HOST_TIMER_RESOLUTION	May return 0, indicating that *platform* does not support device and host timer synchronization.
clGetDeviceAndHostTimer, clGetHostTimer	Returns CL_INVALID_OPERATION if the platform associated with *device* does not support device and host timer synchronization.

Intermediate Language Programs

Creating programs from an intermediate language (such as SPIR-V) is optional for devices supporting OpenCL 3.0. When intermediate language programs are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_IL_VERSION` or `CL_DEVICE_ILS_WITH_VERSION`	May return an empty string and empty array, indicating that *device* does not support intermediate language programs.
clGetProgramInfo, passing `CL_PROGRAM_IL`	Returns an empty buffer (such as *param_value_size_ret* equal to `0`) if no devices in the context associated with *program* support intermediate language programs.
clCreateProgramWithIL	Returns `CL_INVALID_OPERATION` if no devices in *context* support intermediate language programs.
clSetProgramSpecializationConstant	Returns `CL_INVALID_OPERATION` if no devices associated with *program* support intermediate language programs.
clGetKernelSubGroupInfo, passing `CL_KERNEL_COMPILE_NUM_SUB_GROUPS`	Returns `0` if *device* does not support intermediate language programs, since there is currently no way to require a number of sub-groups per work-group for programs created from source.

Sub-groups

Sub-groups are optional for devices supporting OpenCL 3.0. When sub-groups are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_MAX_NUM_SUB_GROUPS`	May return `0`, indicating that *device* does not support sub-groups.
clGetDeviceInfo, passing `CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS`	Returns `CL_FALSE` if *device* does not support sub-groups.
clGetDeviceInfo, passing `CL_DEVICE_EXTENSIONS`	Will not describe support for the `cl_khr_subgroups` extension if *device* does not support sub-groups.
clGetKernelSubGroupInfo	Returns `CL_INVALID_OPERATION` if *device* does not support sub-groups.

OpenCL C compilers supporting sub-groups will define the feature macro `__opencl_c_subgroups`.

Program Initialization and Clean-Up Kernels

Program initialization and clean-up kernels are not supported in OpenCL 3.0, and the APIs and queries for program initialization and clean-up kernels are deprecated in OpenCL 3.0. When program initialization and clean-up kernels are not supported:

API	Behavior
clGetProgramInfo, passing `CL_PROGRAM_SCOPE_GLOBAL_CTORS_PRESENT` or `CL_PROGRAM_SCOPE_GLOBAL_DTORS_PRESENT`	Returns `CL_FALSE` if no devices in the context associated with *program* support program initialization and clean-up kernels.
clSetProgramReleaseCallback	Returns `CL_INVALID_OPERATION` if no devices in the context associated with *program* support program initialization and clean-up kernels.

3D Image Writes

Kernel built-in functions for writing to 3D image objects are optional for devices supporting OpenCL 3.0. When writing to 3D image objects is not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_EXTENSIONS`	Will not describe support for the `cl_khr_3d_image_writes` extension if *device* does not support writing to 3D image objects.
clGetSupportedImageFormats, passing `CL_MEM_OBJECT_IMAGE3D` and one of `CL_MEM_WRITE_ONLY`, `CL_MEM_READ_WRITE`, or `CL_MEM_KERNEL_READ_AND_WRITE`	Returns an empty set (such as *num_image_formats* equal to `0`), indicating that no image formats are supported for writing to 3D image objects, if no devices in *context* support writing to 3D image objects.

OpenCL C compilers supporting writing to 3D image objects will define the feature macro `__opencl_c_3d_image_writes`.

Work-group Collective Functions

Work-group collective functions for broadcasts, scans, and reductions are optional for devices supporting OpenCL 3.0. When work-group collective functions are not supported:

API	Behavior
clGetDeviceInfo, passing `CL_DEVICE_WORK_GROUP_COLLECTIVE_FUNCTIONS_SUPPORT`	May return `CL_FALSE`, indicating that *device* does not support work-group collective functions.

OpenCL C compilers supporting work-group collective functions will define the feature macro `__opencl_c_work_group_collective_functions`.

Generic Address Space

Support for the generic address space is optional for devices supporting OpenCL 3.0. When the generic address space is not supported:

API	Behavior
clGetDeviceInfo, passing CL_DEVICE_GENERIC_ADDRESS_SPACE_SUPPORT	May return CL_FALSE, indicating that *device* does not support the generic address space.

OpenCL C compilers supporting the generic address space will define the feature macro __opencl_c_generic_address_space.

Language Features That Were Already Optional

Some OpenCL C language features were already optional before OpenCL 3.0, the API mechanisms for querying these have not changed.

New feature macros for these optional features have been added to OpenCL C to provide a consistent mechanism for using optional features in OpenCL C 3.0. OpenCL C compilers supporting images will define the feature macro __opencl_c_images. OpenCL C compilers supporting the double type will define the feature macro __opencl_c_fp64. OpenCL C compilers supporting the long, unsigned long and ulong types will define the feature macro __opencl_c_int64, note that compilers for FULL_PROFILE devices must support these types and define the macro unconditionally.

Appendix I: OpenCL Extensions (Informative)

Extensions to the OpenCL API can be defined by authors, groups of authors, and the Khronos OpenCL Working Group. The online Registry of extensions is available at URL

https://registry.khronos.org/OpenCL

It is possible to generate versions of the API Specification incorporating different extensions. At present only a subset of defined extensions can be incorporated in this fashion.

The remainder of this appendix documents a set of extensions chosen when this document was built.

Extensions are grouped as Khronos `khr`, multivendor `ext`, and then alphabetically by author ID. Within each group, extensions are listed in alphabetical order by their names.

Naming Convention for Optional Extensions

OpenCL extensions approved by the OpenCL working group use the following naming convention:

- A unique *name string* of the form **`cl_khr_<name>`** or **`cl_ext_<name>`** is associated with each extension. If the extension is supported by an implementation, this string will be present in the implementation's `CL_PLATFORM_EXTENSIONS` string or `CL_DEVICE_EXTENSIONS` string.
- All enumerants defined by the extension will have names of the form **CL_<*enum_name*>_KHR** or **CL_<*enum_name*>_EXT**.
- All types defined by the extension will have names of the form **cl_<*type_name*>_khr** or **cl_<*type_name*>_ext**.
- All API functions defined by the extension will have names of the form **cl<*function_name*>KHR** or **cl<*function_name*>EXT**.

Functions and enumerants defined by extensions that are promoted to core features will have their **KHR** or **EXT** affix removed. OpenCL implementations of such later revisions must also export the name strings of promoted extensions in the `CL_PLATFORM_EXTENSIONS` or `CL_DEVICE_EXTENSIONS` string, and support the **KHR**- or **EXT**-affixed versions of functions and enumerants as a transition aid.

Vendor extensions are strongly encouraged to follow a similar naming convention:

- A unique *name string* of the form **`cl_<vendor_tag>_<name>`** is associated with each extension. If the extension is supported by an implementation, this string will be present in the implementation's `CL_PLATFORM_EXTENSIONS` string or `CL_DEVICE_EXTENSIONS` string.
- All enumerants defined by the vendor extension should have names of the form **CL_<*enum_name*>_<*VENDOR_TAG*>.**
- All types defined by the vendor extension should have names of the form **cl_<*type_name*>_<*vendor_tag*>.**

- All API functions defined by the vendor extension should have names of the form **cl**<***function_name***><***VENDOR_TAG***>.
- All OpenCL C functions, types, and attribute qualifiers defined by the vendor extension should have names of the form <***vendor_tag***>_<***name***>.

Vendor extensions are not currently included in the OpenCL specifications, but vendor extension specifications are frequently included in the online Registry of extensions.

Header Conventions for Optional Extensions

Function pointer typedefs should be declared for all extensions that add API entrypoints. These typedefs are a required part of the extension interface, and should be provided in an appropriate header, such as `cl_ext.h`.

The following convention should be followed for all extensions affecting the host API:

```
#define cl_extension_name 1
#define CL_EXTENSION_NAME_NAME "cl_extension_name"

#define CL_EXTENSION_NAME_VERSION CL_MAKE_VERSION(major, minor, patch)

// all data typedefs and token #defines for this extension
#define CL_EXTENSION_ENUM_NAME_TAG    0xXXXX

// function types for extension functions
typedef return_type
clExtensionFunctionNameTAG_t(...);

// function pointer typedefs for extension functions
typedef clExtensionFunctionNameTAG_t *
clExtensionFunctionNameTAG_fn;

// extension function prototypes (optional)
extern return_type
clExtensionFunctionTAG(...);
```

where `TAG` can be `KHR`, `EXT` or `vendor-specific`.

Consider, for example, the `cl_khr_create_command_queue` extension. This extension adds the following to `cl_ext.h`:

```
#define cl_khr_create_command_queue 1
#define CL_KHR_CREATE_COMMAND_QUEUE_EXTENSION_NAME \
    "cl_khr_create_command_queue"

#define CL_KHR_CREATE_COMMAND_QUEUE_EXTENSION_VERSION CL_MAKE_VERSION(1, 0, 0)

typedef cl_properties       cl_queue_properties_khr;
```

```
typedef cl_command_queue CL_API_CALL
clCreateCommandQueueWithPropertiesKHR_t(
    cl_context context,
    cl_device_id device,
    const cl_queue_properties_khr* properties,
    cl_int* errcode_ret);

typedef clCreateCommandQueueWithPropertiesKHR_t *
clCreateCommandQueueWithPropertiesKHR_fn CL_API_SUFFIX__VERSION_1_2;

#if !defined(CL_NO_NON_ICD_DISPATCH_EXTENSION_PROTOTYPES)

extern CL_API_ENTRY cl_command_queue CL_API_CALL
clCreateCommandQueueWithPropertiesKHR(
    cl_context context,
    cl_device_id device,
    const cl_queue_properties_khr* properties,
    cl_int* errcode_ret) CL_API_SUFFIX__VERSION_1_2;

#endif /* !defined(CL_NO_NON_ICD_DISPATCH_EXTENSION_PROTOTYPES) */
```

Experimental Extensions

Experimental OpenCL extensions described in this appendix are being made publicly available to enable review and feedback from the community. While an extension is experimental, features may be added, removed, or changed in non-backward compatible ways.

Previous revisions of this specification used the term *provisional* to denote ratified extensions with an experimental status.

If you have feedback on an experimental extension, please create an issue on the OpenCL-Docs repository.

Extension Dependencies

Extensions which have dependencies on specific core versions or on other extensions will list such dependencies.

All extensions implicitly require support for OpenCL 1.0.

List of Current Extensions

- `cl_khr_3d_image_writes`
- `cl_khr_async_work_group_copy_fence`
- `cl_khr_byte_addressable_store`

- cl_khr_create_command_queue
- cl_khr_d3d10_sharing
- cl_khr_d3d11_sharing
- cl_khr_depth_images
- cl_khr_device_enqueue_local_arg_types
- cl_khr_device_uuid
- cl_khr_dx9_media_sharing
- cl_khr_egl_event
- cl_khr_egl_image
- cl_khr_expect_assume
- cl_khr_extended_async_copies
- cl_khr_extended_bit_ops
- cl_khr_extended_versioning
- cl_khr_external_memory
- cl_khr_external_memory_dma_buf
- cl_khr_external_memory_opaque_fd
- cl_khr_external_memory_win32
- cl_khr_external_semaphore
- cl_khr_external_semaphore_opaque_fd
- cl_khr_external_semaphore_sync_fd
- cl_khr_fp16
- cl_khr_fp64
- cl_khr_gl_depth_images
- cl_khr_gl_event
- cl_khr_gl_msaa_sharing
- cl_khr_gl_sharing
- cl_khr_global_int32_base_atomics
- cl_khr_global_int32_extended_atomics
- cl_khr_icd
- cl_khr_icd_unloadable
- cl_khr_il_program
- cl_khr_image2d_from_buffer
- cl_khr_initialize_memory
- cl_khr_int64_base_atomics
- cl_khr_int64_extended_atomics

- cl_khr_integer_dot_product
- cl_khr_kernel_clock
- cl_khr_local_int32_base_atomics
- cl_khr_local_int32_extended_atomics
- cl_khr_mipmap_image
- cl_khr_mipmap_image_writes
- cl_khr_pci_bus_info
- cl_khr_priority_hints
- cl_khr_select_fprounding_mode
- cl_khr_semaphore
- cl_khr_spirv_extended_debug_info
- cl_khr_spirv_linkonce_odr
- cl_khr_spirv_no_integer_wrap_decoration
- cl_khr_spirv_queries
- cl_khr_srgb_image_writes
- cl_khr_subgroup_ballot
- cl_khr_subgroup_clustered_reduce
- cl_khr_subgroup_extended_types
- cl_khr_subgroup_named_barrier
- cl_khr_subgroup_non_uniform_arithmetic
- cl_khr_subgroup_non_uniform_vote
- cl_khr_subgroup_rotate
- cl_khr_subgroup_shuffle
- cl_khr_subgroup_shuffle_relative
- cl_khr_subgroups
- cl_khr_suggested_local_work_size
- cl_khr_terminate_context
- cl_khr_throttle_hints
- cl_khr_work_group_uniform_arithmetic
- cl_ext_buffer_device_address
- cl_ext_cxx_for_opencl
- cl_ext_device_fission
- cl_ext_float_atomics
- cl_ext_image_from_buffer
- cl_ext_image_raw10_raw12

- cl_ext_image_requirements_info
- cl_ext_image_unorm_int_2_101010
- cl_ext_image_unsigned_10x6_12x4_14x2
- cl_ext_immutable_memory_objects
- cl_ext_migrate_memobject

cl_khr_3d_image_writes

Name String

`cl_khr_3d_image_writes`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.0

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_3d_image_writes` adds built-in OpenCL C functions that allow a kernel to write to 3D image objects in addition to 2D image objects.

See the 3D Image Writes section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_async_work_group_copy_fence

Name String

`cl_khr_async_work_group_copy_fence`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2021-11-10

IP Status

No known IP claims.

Description

`cl_khr_async_work_group_copy_fence` adds a new built-in OpenCL C function to establish a memory synchronization ordering of asynchronous copies.

See the Async Work-group Copy Fence section of the OpenCL C specification for more information.

Version History

- Revision 0.9.0, 2020-04-21
  - First assigned version (experimental).
- Revision 1.0.0, 2021-11-10
  - First non-experimental version.

cl_khr_byte_addressable_store

Name String

`cl_khr_byte_addressable_store`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

Interactions and External Dependencies

- Promoted to OpenCL 1.1 core

IP Status

No known IP claims.

Description

`cl_khr_byte_addressable_store` relaxes restrictions on pointers to `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort` and `half` that were present in *Section 6.8m: Restrictions* of the OpenCL 1.0 specification. With this extension, applications are able to read from and write to pointers to these types.

This extension became a core feature in OpenCL 1.1.

See the Byte-Addressable Storage section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_create_command_queue

Name String

`cl_khr_create_command_queue`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.0

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_create_command_queue` allows OpenCL 1.x devices to support an equivalent of the **clCreateCommandQueueWithProperties** API that was added in OpenCL 2.0. This allows OpenCL 1.x devices to support other optional extensions or features that use the **clCreateCommandQueueWithProperties** API to specify additional command-queue properties that cannot be specified using the OpenCL 1.x **clCreateCommandQueue** API.

No new command-queue properties are required by this extension. Applications may use the existing `CL_DEVICE_QUEUE_PROPERTIES` query to determine command-queue properties that are supported by the device.

Newer OpenCL devices may support this extension for compatibility. In this scenario, the function added by this extension will have the same capabilities as the core **clCreateCommandQueueWithProperties** API. Applications that only target newer OpenCL devices should use the core **clCreateCommandQueueWithProperties** API instead of this extension API.

New Commands

- **clCreateCommandQueueWithPropertiesKHR**

New Types

- `cl_queue_properties_khr`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_d3d10_sharing

Name String

`cl_khr_d3d10_sharing`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_d3d10_sharing` provides interoperability between OpenCL and Direct3D 10.

New Commands

- **clGetDeviceIDsFromD3D10KHR**
- **clCreateFromD3D10BufferKHR**
- **clCreateFromD3D10Texture2DKHR**
- **clCreateFromD3D10Texture3DKHR**
- **clEnqueueAcquireD3D10ObjectsKHR**
- **clEnqueueReleaseD3D10ObjectsKHR**

New Types

- `cl_d3d10_device_source_khr`
- `cl_d3d10_device_set_khr`

New Enums

- `cl_d3d10_device_source_khr`
  - `CL_D3D10_DEVICE_KHR`
  - `CL_D3D10_DXGI_ADAPTER_KHR`
- `cl_d3d10_device_set_khr`
  - `CL_PREFERRED_DEVICES_FOR_D3D10_KHR`
  - `CL_ALL_DEVICES_FOR_D3D10_KHR`
- `cl_context_properties`
  - `CL_CONTEXT_D3D10_DEVICE_KHR`
- `cl_context_info`
  - `CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR`
- `cl_mem_info`
  - `CL_MEM_D3D10_RESOURCE_KHR`
- `cl_image_info`
  - `CL_IMAGE_D3D10_SUBRESOURCE_KHR`
- `cl_command_type`
  - `CL_COMMAND_ACQUIRE_D3D10_OBJECTS_KHR`
  - `CL_COMMAND_RELEASE_D3D10_OBJECTS_KHR`
- New Error Codes
  - `CL_INVALID_D3D10_DEVICE_KHR`
  - `CL_INVALID_D3D10_RESOURCE_KHR`
  - `CL_D3D10_RESOURCE_ALREADY_ACQUIRED_KHR`
  - `CL_D3D10_RESOURCE_NOT_ACQUIRED_KHR`

Issues

1. Should this extension be KHR or EXT?

   PROPOSED: KHR. If this extension is to be approved by Khronos then it should be KHR, otherwise EXT. Not all platforms can support this extension, but that is also true of OpenGL interop.

   RESOLVED: KHR.

2. Requiring SharedHandle on ID3D10Resource

Requiring this can largely simplify things at the DDI level and make some implementations faster. However, the DirectX spec only defines the shared handle for a subset of the resources we would like to support:

- `D3D10_RESOURCE_MISC_SHARED` - Enables the sharing of resource data between two or more Direct3D devices. The only resources that can be shared are 2D non-mipmapped textures.

PROPOSED: A: Add wording to the spec about some implementations needing the resource setup as shared:

Some implementations may require the resource to be shared on the D3D10 side of the API.

If we do that, do we need another enum to describe this failure case?

PROPOSED: B: Require that all implementations support both shared and non-shared resources. The restrictions prohibiting multisample textures and the flag D3D10_USAGE_IMMUTABLE guarantee software access to all shareable resources.

RESOLVED: Require that implementations support both D3D10_RESOURCE_MISC_SHARED being set and not set. Add the query for `CL_CONTEXT_D3D10_PREFER_SHARED_RESOURCES_KHR` to determine on a per-context basis which method will be faster.

3. Texture1D support

   There is not a matching CL type, so do we want to support this and map to buffer or Texture2D?

   RESOLVED: We will not add support for ID3D10Texture1D objects unless a corresponding OpenCL 1D Image type is created.

4. CL/D3D10 queries

   The GL interop has **clGetGLObjectInfo** and **clGetGLTextureInfo**. It is unclear if these are needed on the D3D10 interop side since the D3D10 spec makes these queries trivial on the D3D10 object itself. Also, not all of the semantics of the GL call map across.

   PROPOSED: Add the **clGetMemObjectInfo** and **clGetImageInfo** parameter names `CL_MEM_D3D10_RESOURCE_KHR` and `CL_IMAGE_D3D10_SUBRESOURCE_KHR` to query the D3D10 resource from which a `cl_mem` was created. From this data, any D3D10 side information may be queried using the D3D10 API.

   RESOLVED: We will use **clGetMemObjectInfo** and **clGetImageInfo** to access this information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_d3d11_sharing

Name String

`cl_khr_d3d11_sharing`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_d3d11_sharing` provides interoperability between OpenCL and Direct3D 11.

New Commands

- **clGetDeviceIDsFromD3D11KHR**
- **clCreateFromD3D11BufferKHR**
- **clCreateFromD3D11Texture2DKHR**
- **clCreateFromD3D11Texture3DKHR**
- **clEnqueueAcquireD3D11ObjectsKHR**
- **clEnqueueReleaseD3D11ObjectsKHR**

New Types

- `cl_d3d11_device_source_khr`
- `cl_d3d11_device_set_khr`

New Enums

- `cl_d3d11_device_source_khr`
  - `CL_D3D11_DEVICE_KHR`
  - `CL_D3D11_DXGI_ADAPTER_KHR`
- `cl_d3d11_device_set_khr`
  - `CL_PREFERRED_DEVICES_FOR_D3D11_KHR`
  - `CL_ALL_DEVICES_FOR_D3D11_KHR`
- `cl_context_properties`
  - `CL_CONTEXT_D3D11_DEVICE_KHR`
- `cl_context_info`

    - `CL_CONTEXT_D3D11_PREFER_SHARED_RESOURCES_KHR`
- `cl_mem_info`
    - `CL_MEM_D3D11_RESOURCE_KHR`
- `cl_image_info`
    - `CL_IMAGE_D3D11_SUBRESOURCE_KHR`
- `cl_command_type`
    - `CL_COMMAND_ACQUIRE_D3D11_OBJECTS_KHR`
    - `CL_COMMAND_RELEASE_D3D11_OBJECTS_KHR`
- New Error Codes
    - `CL_INVALID_D3D11_DEVICE_KHR`
    - `CL_INVALID_D3D11_RESOURCE_KHR`
    - `CL_D3D11_RESOURCE_ALREADY_ACQUIRED_KHR`
    - `CL_D3D11_RESOURCE_NOT_ACQUIRED_KHR`

Version History

- Revision 1.0.0, 2020-04-21
    - First assigned version.

cl_khr_depth_images

Name String

`cl_khr_depth_images`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.0

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_depth_images` adds OpenCL C support for depth images.

See the Depth Images section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_device_enqueue_local_arg_types

Name String

`cl_khr_device_enqueue_local_arg_types`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_device_enqueue_local_arg_types` allows arguments to blocks that are passed to the **enqueue_kernel** built-in OpenCL C function to be pointers to any type (built-in or user-defined) in local memory, instead of requiring arguments to blocks to be pointers to void in local memory.

See the Device Enqueue Local Argument Types section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_device_uuid

Name String

`cl_khr_device_uuid`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2020-08-27

IP Status

No known IP claims.

Description

`cl_khr_device_uuid` adds the ability to query a universally unique identifier (UUID) for an OpenCL driver and OpenCL device. The UUIDs returned by the query may be used to identify drivers and devices across processes or APIs.

New Enums

Accepted value for the *param_name* parameter to **clGetDeviceInfo**:

- `cl_device_info`
  - `CL_DEVICE_UUID_KHR`
  - `CL_DRIVER_UUID_KHR`
  - `CL_DEVICE_LUID_VALID_KHR`
  - `CL_DEVICE_LUID_KHR`
  - `CL_DEVICE_NODE_MASK_KHR`
- Constants describing the size of the driver and device UUIDs, and the device LUID:
  - `CL_UUID_SIZE_KHR`
  - `CL_LUID_SIZE_KHR`

Version History

- Revision 1.0.0, 2020-08-27
  - First assigned version.

cl_khr_dx9_media_sharing

Name String

`cl_khr_dx9_media_sharing`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_dx9_media_sharing` allows applications to use media surfaces as OpenCL memory objects. This allows efficient sharing of data between OpenCL and selected adapter APIs (only DX9 for now). If this extension is supported, an OpenCL image object can be created from a media surface and the OpenCL API can be used to execute kernels that read and/or write memory objects that are media surfaces. Note that OpenCL memory objects may be created from the adapter media surface if and only if the OpenCL context has been created from that adapter.

New Commands

- **clGetDeviceIDsFromDX9MediaAdapterKHR**
- **clCreateFromDX9MediaSurfaceKHR**
- **clEnqueueAcquireDX9MediaSurfacesKHR**
- **clEnqueueReleaseDX9MediaSurfacesKHR**

New Types

- `cl_dx9_media_adapter_type_khr`
- `cl_dx9_media_adapter_set_khr`

New Enums

- `cl_dx9_media_adapter_type_khr`
  - `CL_ADAPTER_D3D9_KHR`
  - `CL_ADAPTER_D3D9EX_KHR`
  - `CL_ADAPTER_DXVA_KHR`
- `cl_dx9_media_adapter_set_khr`
  - `CL_PREFERRED_DEVICES_FOR_DX9_MEDIA_ADAPTER_KHR`

    - CL_ALL_DEVICES_FOR_DX9_MEDIA_ADAPTER_KHR
- cl_context_info
    - CL_CONTEXT_ADAPTER_D3D9_KHR
    - CL_CONTEXT_ADAPTER_D3D9EX_KHR
    - CL_CONTEXT_ADAPTER_DXVA_KHR
- cl_mem_info
    - CL_MEM_DX9_MEDIA_ADAPTER_TYPE_KHR
    - CL_MEM_DX9_MEDIA_SURFACE_INFO_KHR
- cl_image_info
    - CL_IMAGE_DX9_MEDIA_PLANE_KHR
- cl_command_type
    - CL_COMMAND_ACQUIRE_DX9_MEDIA_SURFACES_KHR
    - CL_COMMAND_RELEASE_DX9_MEDIA_SURFACES_KHR
- New Error Codes
    - CL_INVALID_DX9_MEDIA_ADAPTER_KHR
    - CL_INVALID_DX9_MEDIA_SURFACE_KHR
    - CL_DX9_MEDIA_SURFACE_ALREADY_ACQUIRED_KHR
    - CL_DX9_MEDIA_SURFACE_NOT_ACQUIRED_KHR

Version History

- Revision 1.0.0, 2020-04-21
    - First assigned version.

cl_khr_egl_event

Name String

cl_khr_egl_event

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_egl_event` allows creating OpenCL event objects linked to EGL fence sync objects, potentially improving efficiency of sharing images and buffers between the two APIs. The companion `EGL_KHR_cl_event` extension provides the complementary functionality of creating an EGL sync object from an OpenCL event object.

New Commands

- **clCreateEventFromEGLSyncKHR**

New Enums

- `cl_command_type`
  - `CL_COMMAND_EGL_FENCE_SYNC_OBJECT_KHR`

Issues

Most issues are shared with `cl_khr_gl_event` and are resolved as described in that extension.

1. Should we support implicit synchronization?

   RESOLVED: No, as this may be very difficult since the synchronization would not be with EGL, it would be with currently bound EGL client APIs. It would be necessary to know which client APIs might be bound, to validate that they're associated with the `EGLDisplay` associated with the OpenCL context, and to reach into each such context.

2. Do we need to have typedefs to use EGL handles in OpenCL?

   RESOLVED Using typedefs for EGL handles.

3. Should we restrict which CL APIs can be used with this cl_event?

   RESOLVED Use is limited to calls to acquire and release memory objects only.

4. What is the desired behavior for this extension when EGLSyncKHR is of a type other than `EGL_SYNC_FENCE_KHR`?

   RESOLVED This extension only requires support for `EGL_SYNC_FENCE_KHR`. Support of other types is an implementation choice, and will result in `CL_INVALID_EGL_OBJECT_KHR` if unsupported.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_egl_image

Name String

`cl_khr_egl_image`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_egl_image` provides a mechanism to creating OpenCL memory objects from EGLImages.

New Commands

- **clCreateFromEGLImageKHR**
- **clEnqueueAcquireEGLObjectsKHR**
- **clEnqueueReleaseEGLObjectsKHR**

New Enums

- `cl_command_type`
  - `CL_COMMAND_ACQUIRE_EGL_OBJECTS_KHR`
  - `CL_COMMAND_RELEASE_EGL_OBJECTS_KHR`
- New Error Codes
  - `CL_EGL_RESOURCE_NOT_ACQUIRED_KHR`
  - `CL_INVALID_EGL_OBJECT_KHR`

Issues

1. This extension does not support reference counting of the images, so the onus is on the application to behave sensibly and not release the underlying `cl_mem` object while the `EGLImage` is still being used.
2. In order to ensure data integrity, the application is responsible for synchronizing access to shared CL/EGL image objects by their respective APIs. Failure to provide such synchronization may result in race conditions and other undefined behavior. This may be accomplished by

calling **clWaitForEvents** with the event objects returned by any OpenCL commands which use the shared image object or by calling **clFinish**.

3. Currently `CL_MEM_READ_ONLY` is the only supported flag for *flags*.

   RESOLVED: Implementation will now return an error if writing to a shared object that is not supported rather than disallowing it entirely.

4. Currently restricted to 2D image objects.
5. What should happen for YUV color-space conversion, multi plane images, and chroma-siting, and channel mapping?

   RESOLVED: YUV is no longer explicitly described in this extension. Before this removal the behavior was dependent on the platform. This extension explicitly leaves the YUV layout to the platform and `EGLImage` source extension (i.e. is implementation specific). Colorspace conversion must be applied by the application using a color conversion matrix.

   The expected extension path if YUV color-space conversion is to be supported is to introduce a YUV image type and provide overloaded versions of the read_image built-in functions.

   Getting image information for a YUV image should return the original image size (non quantized size) when all of Y U and V are present in the image. If the planes have been separated then the actual dimensionality of the separated plane should be reported. For example with YUV 4:2:0 (NV12) with a YUV image of 256x256, the Y only image would return 256x256 whereas the UV only image would return 128x128.

6. Should an attribute list be used instead?

   RESOLVED: function has been changed to use an attribute list.

7. What should happen for `EGLImage` extensions which introduce formats without a mapping to an OpenCL image channel data type or channel order?

   RESOLVED: This extension does not define those formats. It is expected that as additional EGL extensions are added to create EGL images from other sources, an extension to CL will be introduced where needed to represent those image types.

8. What are the guarantees to synchronization behavior provided by the implementation?

   The basic portable form of synchronization is to use a **clFinish**, as is the case for GL interop. In addition implementations which support the synchronization extensions `cl_khr_egl_event` and `EGL_KHR_cl_event` can interoperate more efficiently as described in those extensions.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_expect_assume

Name String

`cl_khr_expect_assume`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2021-11-10

Interactions and External Dependencies

The initial version of this extension extends the OpenCL SPIR-V environment to support new instructions. Please refer to the OpenCL SPIR-V Environment Specification that describes how this extension modifies the OpenCL SPIR-V environment.

IP Status

No known IP claims.

Description

`cl_khr_expect_assume` adds mechanisms to provide information to the compiler that may improve the performance of some kernels. Specifically, this extension adds the ability to:

- Tell the compiler the *expected* value of a variable.
- Allow the compiler to *assume* a condition is true.

These functions are not required for functional correctness.

The initial version of this extension extends the OpenCL SPIR-V environment to support new instructions for offline compilation tool chains. Similar functionality may be provided by some OpenCL C online compilation tool chains, but formal support in OpenCL C is not required by the initial version of the extension.

Sample Code

Although this extension does not formally extend OpenCL C, the ability to provide *expect* and *assume* information is supported by many OpenCL C compiler tool chains. The sample code below describes how to test for and provide *expect* and *assume* information to compilers based on Clang:

```
// __has_builtin is an optional compiler feature that is supported by Clang.
// If this feature is not supported, we will assume the builtin is not present.
#ifndef __has_builtin
#define __has_builtin(x)    0
```

```
#endif

kernel void test(global int* dst, global int* src)
{
    int value = src[get_global_id(0)];

    // Tell the compiler that the most likely source value is zero.
#if __has_builtin(__builtin_expect)
    value = __builtin_expect(value, 0);
#endif

    // Tell the compiler that the source value is non-negative.
    // Behavior is undefined if the source value is actually negative.
#if __has_builtin(__builtin_assume)
    __builtin_assume(value >= 0);
#endif

    dst[get_global_id(0)] = value % 4;
}
```

Version History

- Revision 1.0.0, 2021-11-10
  - First assigned version.

cl_khr_extended_async_copies

Name String

`cl_khr_extended_async_copies`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2021-11-10

IP Status

No known IP claims.

Description

`cl_khr_extended_async_copies` augments built-in OpenCL C asynchronous copy functions to support more patterns:

1. For async copy between 2D source and 2D destination.
2. For async copy between 3D source and 3D destination.

See the Extended Async Copy Functions section of the OpenCL C specification for more information.

Version History

- Revision 0.9.0, 2020-04-21
  - First assigned version (experimental).
- Revision 0.9.1, 2021-09-06
  - Elements-based proposal update.
- Revision 1.0.0, 2021-11-10
  - First non-experimental version.

cl_khr_extended_bit_ops

Name String

`cl_khr_extended_bit_ops`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2021-04-22

IP Status

No known IP claims.

Description

`cl_khr_extended_bit_ops` adds built-in OpenCL C functions for performing extended bit operations. Specifically, the following functions are added:

- bitfield insert: insert bits from one source operand into another source operand.
- bitfield extract: extract bits from a source operand, with sign- or zero-extension.
- bit reverse: reverse the bits of a source operand.

See the Extended Bit Operations section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2021-04-22
  - Initial version.

cl_khr_extended_versioning

Name String

`cl_khr_extended_versioning`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.0

Other Extension Metadata

Last Modified Date

2020-02-12

IP Status

No known IP claims.

Contributors

- Kévin Petit, Arm Ltd.
- Ben Ashbaugh, Intel
- Alastair Murray, Codeplay Software Ltd.
- Einar Hov, Arm Ltd.

Description

The `cl_khr_extended_versioning` extension introduces new platform and device queries that return detailed version information to applications. It makes it possible to return the exact revision of the specification or intermediate languages supported by an implementation. It also enables implementations to communicate a version number for each of the extensions they support and remove the requirement for applications to process strings to test for the presence of an extension or intermediate language or built-in kernel.

Extended versioning was promoted to a core feature in OpenCL 3.0. However, the query for `CL_DEVICE_OPENCL_C_NUMERIC_VERSION_KHR` was replaced by the query for `CL_DEVICE_OPENCL_C_ALL_VERSIONS`. With the exception of this query, all types, structures, enums, and macro names defined by this extension are equivalent to the corresponding core name (with the `_KHR` or `_khr` suffix removed).

The version number encoding scheme is described in the Versioning section.

New Types

- cl_version_khr

New Structures

- cl_name_version_khr
- CL_NAME_VERSION_MAX_NAME_SIZE_KHR

New Macro Names

- CL_VERSION_MAJOR_BITS_KHR
- CL_VERSION_MINOR_BITS_KHR
- CL_VERSION_PATCH_BITS_KHR
- CL_VERSION_MAJOR_MASK_KHR
- CL_VERSION_MINOR_MASK_KHR
- CL_VERSION_PATCH_MASK_KHR
- CL_VERSION_MAJOR_KHR
- CL_VERSION_MINOR_KHR
- CL_VERSION_PATCH_KHR
- CL_MAKE_VERSION_KHR

New Enums

- cl_device_info
  - CL_DEVICE_NUMERIC_VERSION_KHR
  - CL_DEVICE_OPENCL_C_NUMERIC_VERSION_KHR
  - CL_DEVICE_EXTENSIONS_WITH_VERSION_KHR
  - CL_DEVICE_ILS_WITH_VERSION_KHR
  - CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION_KHR
- cl_platform_info
  - CL_PLATFORM_NUMERIC_VERSION_KHR
  - CL_PLATFORM_EXTENSIONS_WITH_VERSION_KHR

Conformance Tests

1. Each of the new queries described in this extension must be attempted and succeed.
2. It must be verified that the information returned by all queries that extend existing queries is consistent with the information returned by existing queries.
3. Some of the queries introduced by this extension impose uniqueness constraints on the list of

returned values. It must be verified that these constraints are satisfied.

Issues

1. What compatibility policy should we define? e.g. a *revision* has to be backwards-compatible with previous ones

   RESOLVED: No general rules as that wouldn't be testable. Here's a recommended policy:

   - Patch version bump: only clarifications and small/obvious bugfixes.
   - Minor version bump: backwards-compatible changes only.
   - Major version bump: backwards compatibility may break.

2. Do we want versioning for built-in kernels as returned by `CL_DEVICE_BUILT_IN_KERNELS`?

   RESOLVED: No immediate use-case for versioning but being able to get a list of individual kernels without parsing a string is desirable. Adding `CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION_KHR`.

3. What is the behavior of the queries that return an array of structures when there are no elements to return?

   RESOLVED: The query succeeds and the size returned is zero.

4. What value should be returned when version information is not available?

   RESOLVED: If a patch version is not available, it should be reported as 0. If no version information is available, 0.0.0 should be reported. These values have been chosen as they are guaranteed to be lower than or equal to any other version.

5. Should we add a query to report SPIR-V extended instruction sets?

   RESOLVED: It is unlikely that we will introduce many SPIR-V extended instruction sets without an accompanying API extension. Decided not to do this.

6. Should the queries for which the old-style query doesn't exist in a given OpenCL version be present (e.g. `CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION_KHR` prior to OpenCL 2.1 or without support for `cl_khr_il_program` or `CL_DEVICE_OPENCL_C_NUMERIC_VERSION_KHR` on OpenCL 1.0)?

   RESOLVED: All the queries are always present. `CL_DEVICE_BUILT_IN_KERNELS_WITH_VERSION_KHR` returns an empty set when Intermediate Languages are not supported. `CL_DEVICE_OPENCL_C_NUMERIC_VERSION_KHR` always returns 1.0 on an OpenCL 1.0 platform.

7. Is reporting multiple Intermediate Languages with the same name and major/minor versions but differing patch versions allowed?

   RESOLVED: No. This isn't aligned with the intended use for patch versions and makes it harder for implementations to guarantee consistency with the existing IL queries.

Version History

- Revision 1.0.0, 2020-02-12
  - Initial version.

cl_khr_external_memory

Name String

`cl_khr_external_memory`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 3.0`

Other Extension Metadata

Last Modified Date

2024-09-03

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_memory` defines a generic mechanism to share buffer and image objects between OpenCL and many other APIs, including:

- Optional properties to import external memory exported by other APIs into OpenCL for a set of devices.
- Routines to explicitly hand off memory ownership between OpenCL and other APIs.

Other related extensions define specific external memory types that may be imported into OpenCL.

New Commands

- **clEnqueueAcquireExternalMemObjectsKHR**
- **clEnqueueReleaseExternalMemObjectsKHR**

New Types

- `cl_external_memory_handle_type_khr`

New Enums

- `cl_platform_info`
  - `CL_PLATFORM_EXTERNAL_MEMORY_IMPORT_HANDLE_TYPES_KHR`
- `cl_device_info`
  - `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_HANDLE_TYPES_KHR`
  - `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR`
- `cl_mem_properties`
  - `CL_MEM_DEVICE_HANDLE_LIST_KHR`
  - `CL_MEM_DEVICE_HANDLE_LIST_END_KHR`
- Return values from **clGetEventInfo** when *param_name* is `cl_command_type`:
  - `CL_COMMAND_ACQUIRE_EXTERNAL_MEM_OBJECTS_KHR`
  - `CL_COMMAND_RELEASE_EXTERNAL_MEM_OBJECTS_KHR`

Sample Code

Example for Creating a CL Buffer From an Exported External Buffer in a Single Device Context

This example also requires use of the `cl_khr_external_memory_opaque_fd` extension.

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with just first device
clCreateContext(..., 1, devices, ...);
```

```
// Obtain fd/win32 or similar handle for external memory to be imported
// from other API.
int fd = getFdForExternalMemory();

// Create extMemBuffer of type cl_mem from fd.
cl_mem_properties_khr extMemProperties[] =
{
    (cl_mem_properties_khr)CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_FD_KHR,
    (cl_mem_properties_khr)fd,
    0
};

cl_mem extMemBuffer = clCreateBufferWithProperties(/*context*/          clContext,
                                                   /*properties*/
extMemProperties,
                                                   /*flags*/            0,
                                                   /*size*/             size,
                                                   /*host_ptr*/         NULL,
                                                   /*errcode_ret*/      &errcode_ret);
```

Example for Creating a CL Image From an Exported External Image for Single Device Usage in a Multi-Device Context

This example also requires use of the `cl_khr_external_memory_opaque_fd` extension.

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with first two devices
clCreateContext(..., 2, devices, ...);

// Create img of type cl_mem usable only on devices[0]

// Create img of type cl_mem.
// Obtain fd/win32 or similar handle for external memory to be imported
// from other API.
int fd = getFdForExternalMemory();

// Set cl_image_format based on external image info
cl_image_format clImgFormat = { };
clImageFormat.image_channel_order = CL_RGBA;
clImageFormat.image_channel_data_type = CL_UNORM_INT8;

// Set cl_image_desc based on external image info
size_t clImageFormatSize;
cl_image_desc image_desc = { };
image_desc.image_type = CL_MEM_OBJECT_IMAGE2D_ARRAY;
image_desc.image_width = width;
image_desc.image_height = height;
```

```
image_desc.image_depth = depth;
image_desc.image_array_size = num_slices;
image_desc.image_row_pitch = width * 8 * 4; // May need alignment
image_desc.image_slice_pitch = image_desc.image_row_pitch * height;
image_desc.num_mip_levels = 1;
image_desc.num_samples = 0;
image_desc.buffer = NULL;

cl_mem_properties_khr extMemProperties[] = {
    (cl_mem_properties_khr)CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_FD_KHR,
    (cl_mem_properties_khr)fd,
    (cl_mem_properties_khr)CL_MEM_DEVICE_HANDLE_LIST_KHR,
    (cl_mem_properties_khr)devices[0],
    CL_MEM_DEVICE_HANDLE_LIST_END_KHR,
    0
};

cl_mem img = clCreateImageWithProperties(/*context*/        clContext,
                                         /*properties*/     extMemProperties,
                                         /*flags*/          0,
                                         /*image_format*/   &clImgFormat,
                                         /*image_desc*/     &image_desc,
                                         /*errcode_ret*/    &errcode_ret);

// Use clGetImageInfo to get cl_image_format details.
size_t clImageFormatSize;
clGetImageInfo(img,
               CL_IMAGE_FORMAT,
               sizeof(cl_image_format),
               &clImageFormat,
               &clImageFormatSize);
```

Example for Synchronization Using Wait and Signal

```
// Start the main rendering loop

// Create extSem of type cl_semaphore_khr using clCreateSemaphoreWithPropertiesKHR

// Create extMem of type cl_mem using clCreateBufferWithProperties or
clCreateImageWithProperties

while (true) {
    // (not shown) Signal the semaphore from the other API

    // Wait for the semaphore in OpenCL, by calling clEnqueueWaitSemaphoresKHR on
'extSem'
    clEnqueueWaitSemaphoresKHR(/*command_queue*/           command_queue,
                               /*num_sema_objects*/       1,
                               /*sema_objects*/           &extSem,
                               /*sema_payload_list*/      NULL,
```

```
                                /*num_events_in_wait_list*/  0,
                                /*event_wait_list*/          NULL,
                                /*event*/                    NULL);

    // Launch kernel that accesses extMem
    clEnqueueNDRangeKernel(command_queue, ...);

    // Signal the semaphore in OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/           command_queue,
                                 /*num_sema_objects*/        1,
                                 /*sema_objects*/            &extSem,
                                 /*sema_payload_list*/       NULL,
                                 /*num_events_in_wait_list*/ 0,
                                 /*event_wait_list*/         NULL,
                                 /*event*/                   NULL);

    // (not shown) Launch work in other API that waits on 'extSem'
}
```

Example With Memory Sharing Using Acquire/Release

```
// Create extSem of type cl_semaphore_khr using
// clCreateSemaphoreWithPropertiesKHR with CL_SEMAPHORE_HANDLE_*_KHR.

// Create extMem1 and extMem2 of type cl_mem using clCreateBufferWithProperties
// or clCreateImageWithProperties

while (true) {
    // (not shown) Signal the semaphore from the other API. Wait for the
    // semaphore in OpenCL, by calling clEnqueueWaitForSemaphore on extSem
    clEnqueueWaitSemaphoresKHR(/*command_queue*/           cq1,
                               /*num_sema_objects*/        1,
                               /*sema_objects*/            &extSem,
                               /*sema_payload_list*/       NULL,
                               /*num_events_in_wait_list*/ 0,
                               /*event_wait_list*/         NULL,
                               /*event*/                   NULL);

    // Get explicit ownership of extMem1
    clEnqueueAcquireExternalMemObjectsKHR(/*command_queue*/            cq1,
                                          /*num_mem_objects*/          1,
                                          /*mem_objects*/              &extMem1,
                                          /*num_events_in_wait_list*/  0,
                                          /*event_wait_list*/          NULL,
                                          /*event*/                    NULL);

    // Launch kernel that accesses extMem1 on cq1 on cl_device1
    clEnqueueNDRangeKernel(cq1,  ..., &event1);

    // Launch kernel that accesses both extMem1 and extMem2 on cq2 on cl_device2
```

```
    // Migration of extMem1 and extMem2 handles through regular CL memory
    // migration.
    clEnqueueNDRangeKernel(cq2, ..., &event1, &event2);

    // Give up ownership of extMem1 before you signal the semaphore. Handle
    // memory migration here.
    clEnqueueReleaseExternalMemObjectsKHR(/*command_queue*/           cq2
                                          /*num_mem_objects*/         1,
                                          /*mem_objects*/             &extMem1,
                                          /*num_events_in_wait_list*/ 0,
                                          /*event_wait_list*/         NULL,
                                          /*event*/                   NULL);

    // Signal the semaphore from OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/           cq2,
                                 /*num_sema_objects*/        1,
                                 /*sema_objects*/            &extSem,
                                 /*sema_payload_list*/       NULL,
                                 /*num_events_in_wait_list*/ 0,
                                 /*event_wait_list*/         NULL,
                                 /*event*/                   NULL);

    // (not shown) Launch work in other API that waits on 'extSem'
    // Other API accesses ext1, but not ext2 on device-1
}
```

Issues

1. How should the import of images that are created in external APIs with non-linear tiling be robustly handled?

   UNRESOLVED

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-05-04
  - Clarified device handle list enum cannot be specified without an external memory handle (experimental).
- Revision 0.9.2, 2023-08-01
  - Changed device handle list enum to the memory-specific `CL_MEM_DEVICE_HANDLE_LIST_KHR` (experimental).
- Revision 0.9.3, 2023-08-29
  - Added query for `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR` (experimental).

- Revision 1.0.0, 2024-03-15
  - First non-experimental version.
- Revision 1.0.1, 2024-09-03
  - Return `CL_INVALID_PROPERTY` when multiple external handles are provided when creating a memory object.

cl_khr_external_memory_dma_buf

Name String

`cl_khr_external_memory_dma_buf`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 3.0`
and
`cl_khr_external_memory`

Other Extension Metadata

Last Modified Date

2024-03-15

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM

- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_memory_dma_buf` extends `cl_external_memory_handle_type_khr` to support Linux `dma_buf` as an external memory handle type that may be specified when creating a buffer or image memory object.

New Enums

- `cl_external_memory_handle_type_khr`
  - `CL_EXTERNAL_MEMORY_HANDLE_DMA_BUF_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-05-04
  - Clarified device handle list enum cannot be specified without an external memory handle (experimental).
- Revision 0.9.2, 2023-08-01
  - Changed device handle list enum to the memory-specific `CL_MEM_DEVICE_HANDLE_LIST_KHR` (experimental).
- Revision 0.9.3, 2023-08-29
  - Added query for `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR` (experimental).
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.

cl_khr_external_memory_opaque_fd

Name String

`cl_khr_external_memory_opaque_fd`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 3.0`
and
`cl_khr_external_memory`

Other Extension Metadata

Last Modified Date

2024-03-15

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_memory_opaque_fd` extends `cl_external_memory_handle_type_khr` to support a POSIX file descriptor handle as an external memory handle type that may be specified when creating a buffer or image memory object.

New Enums

- `cl_external_memory_handle_type_khr`
  - `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_FD_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).

- Revision 0.9.1, 2023-05-04
  - Clarified device handle list enum cannot be specified without an external memory handle (experimental).
- Revision 0.9.2, 2023-08-01
  - Changed device handle list enum to the memory-specific `CL_MEM_DEVICE_HANDLE_LIST_KHR` (experimental).
- Revision 0.9.3, 2023-08-29
  - Added query for `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR` (experimental).
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.

cl_khr_external_memory_win32

Name String

`cl_khr_external_memory_win32`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 3.0`
and
`cl_khr_external_memory`

Other Extension Metadata

Last Modified Date

2024-06-11

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL

- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_memory_win32` extends `cl_external_memory_handle_type_khr` to support Windows handles as external memory handle types that may be specified when creating a buffer or image memory object.

New Enums

- `cl_external_memory_handle_type_khr`
  - `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_KHR`
  - `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_KMT_KHR`
  - `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_NAME_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-05-04
  - Clarified device handle list enum cannot be specified without an external memory handle (experimental).
- Revision 0.9.2, 2023-08-01
  - Changed device handle list enum to the memory-specific `CL_MEM_DEVICE_HANDLE_LIST_KHR` (experimental).
- Revision 0.9.3, 2023-08-29
  - Added query for `CL_DEVICE_EXTERNAL_MEMORY_IMPORT_ASSUME_LINEAR_IMAGES_HANDLE_TYPES_KHR` (experimental).
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.
- Revision 1.1.0, 2024-06-11
  - Added `CL_EXTERNAL_MEMORY_HANDLE_OPAQUE_WIN32_NAME_KHR`.

cl_khr_external_semaphore

Name String

`cl_khr_external_semaphore`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`
and
`cl_khr_semaphore`

Other Extension Metadata

Last Modified Date

2024-09-03

Interactions and External Dependencies

- This extension requires OpenCL 1.2.
- The `cl_khr_semaphore` extension is required as it defines semaphore objects as well as for wait and signal operations on semaphores.
- For OpenCL to be able to import external semaphores from other APIs using this extension, the other API is required to provide below mechanisms:
  - Ability to export semaphore handles
  - Ability to query semaphore handle in the form of one of the handle type supported by OpenCL.
- The other APIs that want to use semaphore exported by OpenCL using this extension are required to provide below mechanism:
  - Ability to import semaphore handles using handle types exported by OpenCL.

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL

- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_semaphore` introduced semaphores as a new type along with a set of APIs for create, release, retain, wait and signal operations on it. This extension defines APIs and mechanisms to share semaphores created in an external API by importing into and exporting from OpenCL.

This extension defines:

- New attributes that can be passed as part of `cl_semaphore_properties_khr` for specifying properties of external semaphores to be imported or exported.
- New attributes that can be passed as part of `cl_semaphore_info_khr` for specifying properties of external semaphores to be exported.
- An extension to **clCreateSemaphoreWithPropertiesKHR** to accept external semaphore properties allowing to import or export an external semaphore into or from OpenCL.
- Semaphore handle types required for importing and exporting semaphores.
- Modifications to Wait and Signal API behavior when dealing with external semaphores created from different handle types.
- API query exportable semaphores handles using specified handle type.

The layered extensions `cl_khr_external_semaphore_dx_fence`, `cl_khr_external_semaphore_opaque_fd`, `cl_khr_external_semaphore_sync_fd`, and `cl_khr_external_semaphore_win32` define specific external semaphores that may be imported into or exported from OpenCL.

New Commands

- **clGetSemaphoreHandleForTypeKHR**

New Types

- `cl_external_semaphore_handle_type_khr`

New Enums

- `cl_platform_info`

    - CL_PLATFORM_SEMAPHORE_IMPORT_HANDLE_TYPES_KHR
    - CL_PLATFORM_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR
- cl_device_info
    - CL_DEVICE_SEMAPHORE_IMPORT_HANDLE_TYPES_KHR
    - CL_DEVICE_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR
- cl_semaphore_properties_khr and cl_semaphore_info_khr:
    - CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR
    - CL_SEMAPHORE_EXPORT_HANDLE_TYPES_LIST_END_KHR
- cl_semaphore_info_khr
    - CL_SEMAPHORE_EXPORTABLE_KHR

Sample Code

The following examples use the cl_khr_external_semaphore_opaque_fd extension to obtain an external semaphore. Similar code can be written using the other layered extensions.

Example for Importing a Semaphore Created by Another API in OpenCL in a Single-Device Context

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with just first device
clCreateContext(..., 1, devices, ...);

// Obtain fd/win32 or similar handle for external semaphore to be imported
// from the other API.
int fd = getFdForExternalSemaphore();

// Create clSema of type cl_semaphore_khr usable on the only available device
// assuming the semaphore was imported from the same device.

cl_semaphore_properties_khr sema_props[] =
        {(cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_KHR,
         (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_BINARY_KHR,
         (cl_semaphore_properties_khr)CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR,
         (cl_semaphore_properties_khr)fd,
          0};

int errcode_ret = 0;
cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);
```

Example for Importing a Semaphore Created by Another API in OpenCL in a Multi-device Context for Single Device Usage

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with first two devices
clCreateContext(..., 2, devices, ...);

// Obtain fd/win32 or similar handle for external semaphore to be imported
// from the other API.
int fd = getFdForExternalSemaphore();

// Create clSema of type cl_semaphore_khr usable only on device 1
// assuming the semaphore was imported from the same device.
cl_semaphore_properties_khr sema_props[] = {
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_BINARY_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR,
    (cl_semaphore_properties_khr)fd,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR,
    (cl_semaphore_properties_khr)devices[1],
    CL_SEMAPHORE_DEVICE_HANDLE_LIST_END_KHR,
    0
};

int errcode_ret = 0;
cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);
```

Example for Synchronization Using a Semaphore Created by Another API and Imported in OpenCL

```
// Create clSema using one of the above examples of external semaphore creation.

int errcode_ret = 0;
cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);

// Start the main loop

while (true) {
    // (not shown) Signal the semaphore from the other API

    // Wait for the semaphore in OpenCL
    clEnqueueWaitSemaphoresKHR(/*command_queue*/           command_queue,
                               /*num_sema_objects*/        1,
                               /*sema_objects*/            &clSema,
                               /*num_events_in_wait_list*/ 0,
```

```
                                /*event_wait_list*/         NULL,
                                /*event*/                   NULL);

    // Launch kernel
    clEnqueueNDRangeKernel(command_queue, ...);

    // Signal the semaphore in OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/           command_queue,
                                 /*num_sema_objects*/        1,
                                 /*sema_objects*/            &clSema,
                                 /*num_events_in_wait_list*/ 0,
                                 /*event_wait_list*/         NULL,
                                 /*event*/                   NULL);

    // (not shown) Launch work in the other API that waits on 'clSema'

}
```

Example for Synchronization Using a Semaphore Exported by OpenCL

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with first two devices
clCreateContext(..., 2, devices, ...);

// Create clSema of type cl_semaphore_khr usable only on device 1
cl_semaphore_properties_khr sema_props[] = {
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_BINARY_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR,
    CL_SEMAPHORE_EXPORT_HANDLE_TYPES_LIST_END_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR,
    (cl_semaphore_properties_khr)devices[1],
    CL_SEMAPHORE_DEVICE_HANDLE_LIST_END_KHR,
    0
};

int errcode_ret = 0;
cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);

// Application queries handle-type and the exportable handle associated with the
semaphore.
clGetSemaphoreInfoKHR(clSema,
                      CL_SEMAPHORE_EXPORT_HANDLE_TYPES_KHR,
                      sizeof(cl_external_semaphore_handle_type_khr),
                      &handle_type,
```

```
                      &handle_type_size);

// The other API or process can use the exported semaphore handle
// to import
int fd = -1;
if (handle_type == CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR) {
    clGetSemaphoreHandleForTypeKHR(clSema,
                                   device,
                                   CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR,
                                   sizeof(int),
                                   &fd,
                                   NULL);
}

// Start the main rendering loop

while (true) {
    // (not shown) Signal the semaphore from the other API

    // Wait for the semaphore in OpenCL
    clEnqueueWaitSemaphoresKHR(/*command_queue*/           command_queue,
                               /*num_sema_objects*/        1,
                               /*sema_objects*/            &clSema,
                               /*num_events_in_wait_list*/ 0,
                               /*event_wait_list*/         NULL,
                               /*event*/                   NULL);

    // Launch kernel
    clEnqueueNDRangeKernel(command_queue, ...);

    // Signal the semaphore in OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/           command_queue,
                                 /*num_sema_objects*/        1,
                                 /*sema_objects*/            &clSema,
                                 /*num_events_in_wait_list*/ 0,
                                 /*event_wait_list*/         NULL,
                                 /*event*/                   NULL);

    // (not shown) Launch work in the other API that waits on 'clSema'
}
```

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-11-16
  - Added `CL_SEMAPHORE_EXPORTABLE_KHR`.
- Revision 0.9.2, 2023-11-21

    - Added re-import function call to `cl_khr_external_semaphore_sync_fd`
- Revision 1.0.0, 2024-03-15
    - First non-experimental version.
- Revision 1.0.1, 2024-09-03
    - Return `CL_INVALID_PROPERTY` when multiple external handles are provided when creating a semaphore.

cl_khr_external_semaphore_opaque_fd

Name String

`cl_khr_external_semaphore_opaque_fd`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`
and
`cl_khr_semaphore`
and
`cl_khr_external_semaphore`

Other Extension Metadata

Last Modified Date

2024-03-15

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM

- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_semaphore_opaque_fd` supports importing and exporting a restricted POSIX file descriptor as an external semaphore using the APIs introduced by `cl_khr_external_semaphore`.

New Enums

- `cl_external_semaphore_handle_type_khr`
  - `CL_SEMAPHORE_HANDLE_OPAQUE_FD_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.

cl_khr_external_semaphore_sync_fd

Name String

`cl_khr_external_semaphore_sync_fd`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`
and
`cl_khr_semaphore`
and
`cl_khr_external_semaphore`

Other Extension Metadata

Last Modified Date

2024-03-15

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_semaphore_sync_fd` supports importing and exporting a POSIX file descriptor handle to a Linux Sync File or Android Fence object as an external semaphore using the APIs introduced by `cl_khr_external_semaphore`.

New Commands

- **clReImportSemaphoreSyncFdKHR**

New Types

- `cl_semaphore_reimport_properties_khr`

New Enums

- `cl_external_semaphore_handle_type_khr`
  - `CL_SEMAPHORE_HANDLE_SYNC_FD_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-11-16
  - Added `CL_SEMAPHORE_EXPORTABLE_KHR`.
- Revision 0.9.2, 2023-11-21
  - Added re-import function call to `cl_khr_external_semaphore_sync_fd`
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.
- Revision 1.0.1, 2024-08-06
  - Clarify what re-import properties are accepted by **clReImportSemaphoreSyncFdKHR**.

cl_khr_fp16

Name String

`cl_khr_fp16`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_fp16` adds support to OpenCL C for half scalar and vector types as built-in types that can be used for arithmetic operations, conversions, etc.

See the Half-Precision Floating-Point section of the OpenCL C specification for more information.

New Enums

- `cl_device_info`
  - `CL_DEVICE_HALF_FP_CONFIG`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_fp64

Name String

`cl_khr_fp64`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.2

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_fp64` adds support to OpenCL C for double-precision scalar and vector types as built-in types that can be used for arithmetic operations, conversions, etc.

See the Double-Precision Floating-Point section of the OpenCL C specification for more information.

New Enums

- `cl_device_info`
  - `CL_DEVICE_DOUBLE_FP_CONFIG`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_gl_depth_images

Name String

`cl_khr_gl_depth_images`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_gl_sharing`

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_gl_depth_images` extends OpenCL / OpenGL sharing defined by the `cl_khr_gl_sharing` extension to allow an OpenCL image to be created from an OpenGL depth or depth-stencil texture.

Depth images with an image channel order of `CL_DEPTH_STENCIL` can only be created using the **clCreateFromGLTexture** API.

New Enums

- `cl_channel_order`
  - `CL_DEPTH_STENCIL`
- `cl_channel_type`
  - `CL_UNORM_INT24`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_gl_event

Name String

`cl_khr_gl_event`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_gl_sharing`

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_gl_event` allows creating OpenCL event objects linked to OpenGL fence sync objects, potentially improving efficiency of sharing images and buffers between the two APIs. The companion `GL_ARB_cl_event` extension provides the complementary functionality of creating an OpenGL sync object from an OpenCL event object.

In addition, this extension modifies the behavior of **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** to implicitly guarantee synchronization with an OpenGL context bound in the same thread as the OpenCL context.

New Commands

- **clCreateEventFromGLsyncKHR**

New Enums

- `cl_command_type`
  - `CL_COMMAND_GL_FENCE_SYNC_OBJECT_KHR`

Issues

1. How are references between CL events and GL syncs handled?

   PROPOSED: The linked CL event places a single reference on the GL sync object. That reference is removed when the CL event is deleted. A more expensive alternative would be to reflect changes in the CL event reference count through to the GL sync.

2. How are linkages to synchronization primitives in other APIs handled?

   UNRESOLVED. We will at least want to have a way to link events to EGL sync objects. There is probably no analogous DX concept. There would be an entry point for each type of synchronization primitive to be linked to, such as **clCreateEventFromEGLSyncKHR**.

   An alternative is a generic clCreateEventFromExternalEvent taking an attribute list. The attribute list would include information defining the type of the external primitive and additional information (GL sync object handle, EGL display and sync object handle, etc.) specific to that type. This allows a single entry point to be reused.

   These will probably be separate extensions following the API proposed here.

3. Should the `CL_EVENT_COMMAND_TYPE` correspond to the type of command (fence) or the type of the linked sync object?

PROPOSED: To the type of the linked sync object.

4. Should we support both explicit and implicit synchronization?

   PROPOSED: Yes. Implicit synchronization is suitable when GL and CL are executing in the same application thread. Explicit synchronization is suitable when they are executing in different threads but the expense of glFinish is too high.

5. Should this be a platform or device extension?

   PROPOSED: Platform extension. This may result in considerable under-the-hood work to implement the sync→event semantics using only the public GL API, however, when multiple drivers and devices with different GL support levels coexist in the same runtime.

6. Where can events generated from GL syncs be usable?

   PROPOSED: Only with **clEnqueueAcquireGLObjects**, and attempting to use such an event elsewhere will generate an error. There is no apparent use case for using such events elsewhere, and possibly some cost to supporting it, balanced by the cost of checking the source of events in all other commands accepting them as parameters.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_gl_msaa_sharing

Name String

`cl_khr_gl_msaa_sharing`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_gl_depth_images`
and
`cl_khr_gl_sharing`

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_gl_msaa_sharing` extends the `cl_khr_gl_sharing` extension to allow a shared OpenCL/OpenGL image object to be created from an OpenGL multi-sampled ("MSAA") color or depth texture.

This extension adds multi-sample support to **clCreateFromGLTexture** and **clGetGLTextureInfo**, and allows passing multi-sample images to compute kernels.

This extension requires `cl_khr_gl_depth_images`.

See the cl_khr_gl_msaa_sharing section of the OpenCL C specification for more information.

New Enums

- `cl_gl_texture_info`
  - `CL_GL_NUM_SAMPLES`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_gl_sharing

Name String

`cl_khr_gl_sharing`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_gl_sharing` extension allows use of OpenGL buffer, texture, and renderbuffer objects as OpenCL memory objects, referred to as "Shared OpenCL/OpenGL Memory Objects".

An OpenCL context may be associated with an OpenGL context or share group object, using additional attributes described for **clCreateContext**.

An OpenCL image object may be created from an OpenGL texture or renderbuffer object as

described for **clCreateFromGLTexture** and **clCreateFromGLRenderbuffer**, respectively.

An OpenCL buffer object may be created from an OpenGL buffer object using **clCreateFromGLBuffer**.

Any supported OpenGL object defined within the associated OpenGL context or share group object may be shared, with the exception of the default OpenGL objects (i.e. objects named zero), which may not be shared.

Additional information on the use of shared OpenCL/OpenGL memory objects is found in the Lifetime of Shared OpenCL/OpenGL Memory Objects, Acquiring, Releasing, and Synchronizing Access to Shared OpenCL/OpenGL Memory Objects and Querying Devices that Support Sharing with OpenGL sections.

An OpenGL implementation supporting buffer objects and sharing of texture and buffer object images with OpenCL is required by this extension.

New Commands

- **clGetGLContextInfoKHR**
- **clCreateFromGLBuffer**
- **clCreateFromGLTexture**
- **clCreateFromGLRenderbuffer**
- **clGetGLObjectInfo**
- **clGetGLTextureInfo**
- **clEnqueueAcquireGLObjects**
- **clEnqueueReleaseGLObjects**

New Types

- `cl_gl_context_info`
- `cl_gl_object_type`
- `cl_gl_texture_info`
- `cl_gl_platform_info`

New Enums

- `cl_gl_context_info`
  - `CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR`
  - `CL_DEVICES_FOR_GL_CONTEXT_KHR`
- `cl_context_properties`
  - `CL_GL_CONTEXT_KHR`
  - `CL_EGL_DISPLAY_KHR`

    - CL_GLX_DISPLAY_KHR
    - CL_WGL_HDC_KHR
    - CL_CGL_SHAREGROUP_KHR
- cl_gl_object_type
    - CL_GL_OBJECT_BUFFER
    - CL_GL_OBJECT_TEXTURE2D
    - CL_GL_OBJECT_TEXTURE3D
    - CL_GL_OBJECT_RENDERBUFFER
    - CL_GL_OBJECT_TEXTURE2D_ARRAY
    - CL_GL_OBJECT_TEXTURE1D
    - CL_GL_OBJECT_TEXTURE1D_ARRAY
    - CL_GL_OBJECT_TEXTURE_BUFFER
- cl_gl_texture_info
    - CL_GL_TEXTURE_TARGET
    - CL_GL_MIPMAP_LEVEL
- New Error Codes
    - CL_INVALID_GL_SHAREGROUP_REFERENCE_KHR

Issues

1. How should the OpenGL context be identified when creating an associated OpenCL context?

   RESOLVED: by using a (display,context handle) attribute pair to identify an arbitrary OpenGL or OpenGL ES context with respect to one of the window-system binding layers EGL, GLX, or WGL, or a share group handle to identify a CGL share group. If a context is specified, it need not be current to the thread calling **clCreateContext**.

   A previously suggested approach would use a single boolean attribute CL_USE_GL_CONTEXT_KHR to allow creating a context associated with the currently bound OpenGL context. This may still be implemented as a separate extension, and might allow more efficient acquire/release behavior in the special case where they are being executed in the same thread as the bound GL context used to create the CL context.

2. What should the format of an attribute list be?

   After considerable discussion, we think we can live with a list of <attribute name,value> pairs terminated by zero. The list is passed as 'cl_context_properties *_properties'_, where cl_context_properties is typedefed to be 'intptr_t' in cl.h.

   This effectively allows encoding all scalar integer, pointer, and handle values in the host API into the argument list and is analogous to the structure and type of EGL attribute lists. NULL attribute lists are also allowed. Again as for EGL, any attributes not explicitly passed in the list will take on a defined default value that does something reasonable.

Experience with EGL, GLX, and WGL has shown attribute lists to be a sufficiently flexible and general mechanism to serve the needs of management calls such as context creation. It is not completely general (encoding floating-point and non-scalar attribute values is not straightforward), and other approaches were suggested such as opaque attribute lists with getter/setter methods, or arrays of varadic structures.

3. What's the behavior of an associated OpenGL or OpenCL context when using resources defined by the other associated context, and that context is destroyed?

   RESOLVED: OpenCL objects place a reference on the data store underlying the corresponding GL object when they're created. The GL name corresponding to that data store may be deleted, but the data store itself remains so long as any CL object has a reference to it. However, destroying all GL contexts in the share group corresponding to a CL context results in implementation-dependent behavior when using a corresponding CL object, up to and including program termination.

4. How about sharing with D3D?

   Sharing between D3D and OpenCL should use the same attribute list mechanism, though obviously with different parameters, and be exposed as a similar parallel OpenCL extension. There may be an interaction between that extension and this one since it's not yet clear if it will be possible to create a CL context simultaneously sharing GL and D3D objects.

5. Under what conditions will context creation fail due to sharing?

   RESOLVED: Several cross-platform failure conditions are described (GL context or CGL share group doesn't exist, GL context doesn't support types of GL objects, GL context implementation doesn't allow sharing), but additional failures may result due to implementation-dependent reasons and should be added to this extension as such failures are discovered. Sharing between OpenCL and OpenGL requires integration at the driver internals level.

6. What command-queues can **clEnqueueAcquireGLObjects** and **clEnqueueReleaseGLObjects** be placed on?

   RESOLVED: All command-queues. This restriction is enforced at context creation time. If any device passed to context creation cannot support shared OpenCL/OpenGL memory objects, context creation will fail with a `CL_INVALID_OPERATION` error.

7. How can applications determine which command-queue to place an Acquire/Release on?

   RESOLVED: The **clGetGLContextInfoKHR** returns either the CL device currently corresponding to a specified GL context (typically the display it's running on), or a list of all the CL devices the specified context might run on (potentially useful in multiheaded / "virtual screen" environments). This command is not placed together with commands to create shared OpenCL/OpenGL memory objects because it relies on the same property-list method of specifying a GL context introduced by this extension.

   If no devices are returned, it means that the GL context exists on an older GPU not capable of running OpenCL, but still capable of sharing objects between GL running on that GPU and CL running elsewhere.

8. What is the meaning of the `CL_DEVICES_FOR_GL_CONTEXT_KHR` query?

   RESOLVED: The list of all CL devices that may ever be associated with a specific GL context. On platforms such as MacOS X, the “virtual screen” concept allows multiple GPUs to back a single virtual display. Similar functionality might be implemented on other windowing systems, such as a transparent heterogenous multiheaded X server. Therefore the exact meaning of this query is interpreted relative to the binding layer API in use.

9. What happened to the "`extension`"s `cl_khr_gl_sharing__context` and `cl_khr_gl_sharing__memobjs` that were previously published?

   RESOLVED: These were not actual extensions, but the result of splitting the `cl_khr_gl_sharing` extension language into two separate sections for publication. All extension language has now been integrated into the unified Specification and this distinction is not useful.

10. Where are the **clCreateFromGLTexture2D** and **clCreateFromGLTexture3D** fuctions described?

    PROPOSED: These functions are present in cl.xml, listed as OpenCL 1.0 APIs that were deprecated in OpenCL 1.2, but the current extension language does not described them. Since OpenCL 1.2 itself is so old, it is not worth the effort to look back and determine the exact details of these APIs.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_global_int32_base_atomics

Name String

`cl_khr_global_int32_base_atomics`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_global_int32_base_atomics` allows OpenCL C atomic operations to be performed on 32-bit signed and unsigned integers in global memory.

This extension became a core feature in OpenCL 1.1, with the built-in atomic function names changed to use the **atomic**_ prefix instead of **atom**_.

See the Global 32-Bit Base Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_global_int32_extended_atomics

Name String

`cl_khr_global_int32_extended_atomics`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_global_int32_extended_atomics` allows OpenCL C extended atomic operations to be performed on 32-bit signed and unsigned integers in global memory.

This extension became a core feature in OpenCL 1.1, with the built-in atomic function names changed to use the **atomic**_ prefix instead of **atom**_.

See the Global 32-Bit Extended Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21

- First assigned version.

cl_khr_icd

Name String

`cl_khr_icd`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2025-12-15

IP Status

No known IP claims.

Description

`cl_khr_icd` describes a platform extension which defines a simple mechanism through which the Khronos OpenCL installable client driver loader (ICD Loader) may expose multiple separate vendor installable client drivers (Vendor ICDs) for OpenCL. An application written against the ICD Loader will be able to access all `cl_platform_id`s exposed by all vendor implementations with the ICD Loader acting as a demultiplexor.

This is a platform extension, so if this extension is supported by an implementation, the string `"cl_khr_icd"` will be present in the `CL_PLATFORM_EXTENSIONS` string.

Source Code

The official source for the ICD Loader is available on github, at:

https://github.com/KhronosGroup/OpenCL-ICD-Loader

The complete `_cl_icd_dispatch` structure is defined in the header `cl_icd.h`, which is available as a part of the OpenCL headers.

Inferring Vendors From Function Call Arguments

At every OpenCL function call, the ICD Loader infers the vendor ICD function to call from the arguments to the function. An object is said to be ICD compatible if it is of the following structure:

```
struct _cl_<object>
{
    struct _cl_icd_dispatch *dispatch;
```

```
    // Since version 2.0.0
    void *dispatch_data;
    // ... remainder of internal data
};
```

<object> is one of platform_id, device_id, context, command_queue, mem, program, kernel, event, or sampler.

The structure `_cl_icd_dispatch` is a function pointer dispatch table which is used to direct calls to a particular vendor implementation. All objects created from ICD compatible objects must be ICD compatible.

The definition for `_cl_icd_dispatch` is provided along with the OpenCL headers. Existing members can never be removed from that structure but new members can be appended.

Functions which do not have an argument from which the vendor implementation may be inferred have been deprecated and may be ignored.

New in version 2.0.0, objects that are ICD 2 compatible should also contain a `dispatch_data` field of type `void *` that the loader is free to modify for a platform through the **clIcdSetPlatformDispatchDataKHR** new API entry point. In order to distinguish ICD 1 and ICD 2 objects, the pointers to **clGetPlatformIDs** and **clUnloadCompiler** in the dispatch structure must be set to `CL_ICD2_TAG_KHR`. The vendor driver is responsible for propagating the `dispatch_data` value when returning objects to users.

ICD Data

A Vendor ICD is defined by two pieces of data:

- The Vendor ICD library specifies a library which contains the OpenCL entry points for the vendor's OpenCL implementation. The vendor ICD's library file name should include the vendor name, or a vendor-specific implementation identifier.
- The Vendor ICD extension suffix is a short string which specifies the default suffix for extensions implemented only by that vendor. The vendor suffix string is optional.

ICD Loader Vendor Enumeration on Windows

To enumerate Vendor ICDs on Windows, the ICD Loader will first scan for REG_SZ string values in the "Display Adapter" and "Software Components" HKR registry keys. The exact registry keys to scan should be obtained via PnP Configuration Manager APIs, but will look like:

For 64-bit ICDs:

```
HKLM\SYSTEM\CurrentControlSet\Control\Class\
{Display Adapter GUID}\{Instance ID}\OpenCLDriverName, or

HKLM\SYSTEM\CurrentControlSet\Control\Class\
{Software Component GUID}\{Instance ID}\OpenCLDriverName
```

For 32-bit ICDs:

```
HKLM\SYSTEM\CurrentControlSet\Control\Class\
{Display Adapter GUID}\{Instance ID}\OpenCLDriverNameWoW, or

HKLM\SYSTEM\CurrentControlSet\Control\Class\
{Software Component GUID}\{Instance ID}\OpenCLDriverNameWoW
```

These registry values contain the path to the Vendor ICD library. For example, if the registry contains the value:

```
[HKLM\SYSTEM\CurrentControlSet\Control\Class\{GUID}\{Instance}]
"OpenCLDriverName"="c:\\vendor a\\vndra_ocl.dll"
```

Then the ICD Loader will open the Vendor ICD library:

```
c:\vendor a\vndra_ocl.dll
```

The ICD Loader will also scan for REG_DWORD values in the registry key:

```
HKLM\SOFTWARE\Khronos\OpenCL\Vendors
```

For each registry value in this key which has data set to 0, the ICD Loader will open the Vendor ICD library specified by the name of the registry value.

For example, if the registry contains the value:

```
[HKLM\SOFTWARE\Khronos\OpenCL\Vendors]
"c:\\vendor a\\vndra_ocl.dll"=dword:00000000
```

Then the ICD Loader will open the Vendor ICD library:

```
c:\vendor a\vndra_ocl.dll
```

ICD Loader Vendor Enumeration on Linux

To enumerate vendor ICDs on Linux, the ICD Loader scans the files in the path `/etc/OpenCL/vendors`. For each file in this path, the ICD Loader opens the file as a text file. The expected format for the file is a single line of text which specifies the Vendor ICD's library. The ICD Loader will attempt to open that file as a shared object using dlopen(). Note that the library specified may be an absolute path or just a file name.

For example, if the following file exists

```
/etc/OpenCL/vendors/VendorA.icd
```

and contains the text

```
libVendorAOpenCL.so
```

then the ICD Loader will load the library `libVendorAOpenCL.so`.

ICD Loader Vendor Enumeration on Android

To enumerate vendor ICDs on Android, the ICD Loader scans the files in the path `/system/vendor/Khronos/OpenCL/vendors`. For each file in this path, the ICD Loader opens the file as a text file. The expected format for the file is a single line of text which specifies the Vendor ICD's library. The ICD Loader will attempt to open that file as a shared object using dlopen(). Note that the library specified may be an absolute path or just a file name.

For example, if the following file exists

```
/system/vendor/Khronos/OpenCL/vendors/VendorA.icd
```

and contains the text

```
libVendorAOpenCL.so
```

then the ICD Loader will load the library `libVendorAOpenCL.so`.

Adding a Vendor Library

Upon successfully loading a Vendor ICD's library, the ICD Loader queries the following functions from the library: **clIcdGetPlatformIDsKHR**, **clGetPlatformInfo**, and **clGetExtensionFunctionAddress** (note: **clGetExtensionFunctionAddress** has been deprecated, but is still required for the ICD Loader). If any of these functions are not present then the ICD Loader will close and ignore the library.

Next the ICD Loader queries available ICD-enabled platforms in the library using **clIcdGetPlatformIDsKHR**. For each of these platforms, the ICD Loader queries the platform's extension string to verify that `cl_khr_icd` is supported, then queries the platform's Vendor ICD extension suffix using **clGetPlatformInfo** with the value `CL_PLATFORM_ICD_SUFFIX_KHR`.

If any of these steps fail, the ICD Loader will ignore the Vendor ICD and continue on to the next.

New in version 2.0.0, the ICD Loader will also query the following functions from the library: **clIcdGetFunctionAddressForPlatformKHR** and **clIcdSetPlatformDispatchDataKHR**. If these two functions are present and the pointers to **clGetPlatformIDs** and **clUnloadCompiler** in the dispatch structure of a platform is set to `CL_ICD2_TAG_KHR` the platform will be deemed ICD 2 compatible and dispatch will be managed by the ICD Loader. If the `CL_ICD2_TAG_KHR` tag is present but one of the two

functions above is missing or if the tag is present in only one of the pointers, the ICD Loader will ignore the Vendor ICD and continue on to the next.

During initialization, after calling **clIcdGetPlatformIDsKHR** to query the available platforms, the ICD Loader will, for each ICD 2 compatible platform, query dispatchable entry points using **clIcdGetFunctionAddressForPlatformKHR** and then set the platform dispatch data using **clIcdSetPlatformDispatchDataKHR**.

New Commands

- **clIcdGetPlatformIDsKHR**

New in version 2.0.0:

- **clIcdGetFunctionAddressForPlatformKHR**
- **clIcdSetPlatformDispatchDataKHR**

New Enums

Accepted as *param_name* to the function **clGetPlatformInfo**:

- `CL_PLATFORM_ICD_SUFFIX_KHR`

Returned by **clGetPlatformIDs** when no platforms are found:

- `CL_PLATFORM_NOT_FOUND_KHR`

New in version 2.0.0, used as a value in the pointers to **clGetPlatformIDs** and **clUnloadCompiler** in the dispatch structure:

- `CL_ICD2_TAG_KHR`

Issues

1. Some OpenCL functions do not take an object argument from which their vendor library may be identified (e.g, **clUnloadCompiler**), how will they be handled?

   RESOLVED: Such functions will be a noop for all calls through the ICD Loader.

2. How are OpenCL extension to be handled?

   RESOLVED: Extension APIs must be queried using **clGetExtensionFunctionAddressForPlatform**.

3. How will the ICD Loader handle a `NULL cl_platform_id`?

   RESOLVED: The ICD will by default choose the first enumerated platform as the `NULL` platform.

4. There exists no mechanism to unload the ICD Loader, should there be one?

   RESOLVED: As there is no standard mechanism for unloading a vendor implementation, do not add one for the ICD Loader.

New in version 2.0.1, the loader now supports deinitialization and platform that support unloading their driver can advertise it through the `cl_khr_icd_unloadable` extension.

5. How will the ICD Loader that support deinitialization enable compatibility with old layers.

   The ICD Loader that support deinitialization will check for the `OCL_ICD_FORCE_LEGACY_TERMINATION` environment variable, and revert to the legacy behavior if it evaluates to true.

6. How will the ICD Loader handle `NULL` objects passed to the OpenCL functions?

   RESOLVED: The ICD Loader will check for `NULL` objects passed to the OpenCL functions without trying to dereference the `NULL` objects for obtaining the ICD dispatch table. On detecting a `NULL` object it will return one of the an invalid object error values (e.g. `CL_INVALID_DEVICE` corresponding to the object in question.

7. How will the ICD loader avoid calling entry points outside of the dispatch table of old implementations, potentially causing segfaults?

   RESOLVED: From versions 2.0.0 and above, the loader will be responsible for managing dispatch data information per platform. The content of the dispatch table is obtained through a dedicated entry point, and missing entry points are replaced by stubs that return `CL_INVALID_OPERATION`.

8. How will the ICD loader set the loader managed dispatch information in new objects? Some of these objects may be created by extension functions that the loader is unaware of.

   RESOLVED: During initialization, the loader will set the platform dispatch information through a new entry point. The implementation will be responsible for propagating this information to new objects.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.
- Revision 2.0.0, 2025-04-01
  - Loader managed dispatch.
- Revision 2.0.1, 2025-12-15
  - Loader deinitialization.

cl_khr_icd_unloadable

Name String

`cl_khr_icd_unloadable`

Ratification Status

Not ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2025-11-25

IP Status

No known IP claims.

Description

`cl_khr_icd_unloadable` describes a platform extension which defines a simple mechanism through which a Khronos OpenCL installable client driver (ICD), as defined by the `cl_khr_icd`, can advertise being unloaded by the ICD Loader.

This is a platform extension, so if this extension is supported by an implementation, the string `"cl_khr_icd_unloadable"` will be present in the `CL_PLATFORM_EXTENSIONS` string.

New Enums

Used as a parameter name to query if a platform can be unloaded by the ICD Loader:

- `CL_PLATFORM_UNLOADABLE_KHR`

Version History

- Revision 1.0.0, 2025-12-15
  - First assigned version.

cl_khr_il_program

Name String

`cl_khr_il_program`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_il_program` adds the ability to create programs with intermediate language (IL), usually SPIR-V. Further information about the format and contents of SPIR-V may be found in the SPIR-V Specification. Information about how SPIR-V modules behave in the OpenCL environment may be found in the OpenCL SPIR-V Environment Specification.

This functionality described by this extension is a core feature in OpenCL 2.1.

New Commands

- **clCreateProgramWithILKHR**

New Enums

- `cl_device_info`
  - `CL_DEVICE_IL_VERSION_KHR`
- `cl_platform_info`
  - `CL_PROGRAM_IL_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_image2d_from_buffer

Name String

`cl_khr_image2d_from_buffer`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.0

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_image2d_from_buffer` allows a 2D image to be created from an existing OpenCL buffer memory object.

This extension became a core feature in OpenCL 2.0.

Refer to the discussion of 2D images created from buffers in the Image Descriptor section for additional details.

New Enums

- `CL_DEVICE_IMAGE_PITCH_ALIGNMENT_KHR`
- `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_initialize_memory

Name String

`cl_khr_initialize_memory`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_initialize_memory` adds OpenCL C support for initializing local and private memory before a kernel begins execution. This is accomplished by specifying a flag at context creation time affecting all such memory.

Memory is allocated in various forms in OpenCL both explicitly (global memory) or implicitly (local,

private memory). This allocation so far does not provide a straightforward mechanism to initialize the memory on allocation. In other words what is lacking is the equivalent of `calloc` for the currently supported `malloc` like capability. This functionality is useful for a variety of reasons including ease of debugging, application controlled limiting of visibility to previous contents of memory and in some cases, optimization.

See the Initializing Memory section of the OpenCL C specification for more information.

New Enums

- `cl_context_properties`
  - `CL_CONTEXT_MEMORY_INITIALIZE_KHR`
- `cl_context_memory_initialize_khr`
  - `CL_CONTEXT_MEMORY_INITIALIZE_LOCAL_KHR`
  - `CL_CONTEXT_MEMORY_INITIALIZE_PRIVATE_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_int64_base_atomics

Name String

`cl_khr_int64_base_atomics`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_int64_base_atomics` adds built-in OpenCL functions supporting atomic operations to be performed on 64-bit signed and unsigned integers in global and local memory.

See the 64-Bit Base Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_int64_extended_atomics

Name String

`cl_khr_int64_extended_atomics`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_int64_extended_atomics` adds built-in OpenCL functions supporting extended atomic operations to be performed on 64-bit signed and unsigned integers in global and local memory.

See the 64-Bit Extended Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_integer_dot_product

Name String

`cl_khr_integer_dot_product`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2021-06-23

IP Status

No known IP claims.

Contributors

- Kévin Petit, Arm Ltd.
- Jeremy Kemp, Imagination Technologies
- Ben Ashbaugh, Intel
- Ruihao Zhang, Qualcomm
- Stuart Brady, Arm Ltd
- Balaji Calidas, Qualcomm
- Ayal Zaks, Intel

Description

`cl_khr_integer_dot_product` adds support for SPIR-V instructions and OpenCL C built-in functions to compute the dot product of vectors of integers.

OpenCL C compilers supporting this extension will define the extension macro `cl_khr_integer_dot_product`, and may define corresponding feature macros `__opencl_c_integer_dot_product_input_4x8bit` and `__opencl_c_integer_dot_product_input_4x8bit_packed` depending on the reported capabilities.

See the Integer Dot Product section of the OpenCL C specification for more information.

New Structures

- `cl_device_integer_dot_product_acceleration_properties_khr`

New Types

- `cl_device_integer_dot_product_capabilities_khr`

New Enums

- `cl_device_integer_dot_product_capabilities_khr`
  - `CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_KHR`
  - `CL_DEVICE_INTEGER_DOT_PRODUCT_INPUT_4x8BIT_PACKED_KHR`
- `cl_device_info`

- `CL_DEVICE_INTEGER_DOT_PRODUCT_CAPABILITIES_KHR`
- `CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_8BIT_KHR`
- `CL_DEVICE_INTEGER_DOT_PRODUCT_ACCELERATION_PROPERTIES_4x8BIT_PACKED_KHR`

Version History

- Revision 1.0.0, 2021-06-17
  - Initial version
- Revision 2.0.0, 2021-06-23
  - 8-bit support is mandatory, added 8-bit acceleration properties.

cl_khr_kernel_clock

Name String

`cl_khr_kernel_clock`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2024-05-02

IP Status

No known IP claims.

Contributors

- Kevin Petit, Arm Ltd.
- Paul Fradgley, Imagination Technologies
- Jeremy Kemp, Imagination Technologies
- Ben Ashbaugh, Intel
- Balaji Calidas, Qualcomm Technologies, Inc.
- Ruihao Zhang, Qualcomm Technologies, Inc.

Description

`cl_khr_kernel_clock` adds the ability for a kernel to sample the value from one of three clocks provided by compute units.

OpenCL C compilers supporting this extension will define the extension macro `cl_khr_kernel_clock`, and may define corresponding feature macros `__opencl_c_kernel_clock_scope_device`, `__opencl_c_`

kernel_clock_scope_work_group, and __opencl_c_kernel_clock_scope_sub_group depending on the reported capabilities.

See the Kernel Clock section of the OpenCL C specification for more information.

Interactions With Other Extensions

On devices that implement the EMBEDDED profile, the cles_khr_int64 extension is required for the clock_read_device, clock_read_work_group and clock_read_sub_group functions to be present.

Support for sub-groups is required for the clock_read_sub_group and clock_read_hilo_sub_group functions to be present.

New Types

- cl_device_kernel_clock_capabilities_khr

New Enums

- cl_device_info
  - CL_DEVICE_KERNEL_CLOCK_CAPABILITIES_KHR
- cl_device_kernel_clock_capabilities_khr
  - CL_DEVICE_KERNEL_CLOCK_SCOPE_DEVICE_KHR
  - CL_DEVICE_KERNEL_CLOCK_SCOPE_WORK_GROUP_KHR
  - CL_DEVICE_KERNEL_CLOCK_SCOPE_SUB_GROUP_KHR

Version History

- Revision 0.9.0, 2024-03-25
  - First assigned version (experimental).
- Revision 1.0.0, 2025-05-02
  - First non-experimental version.

cl_khr_local_int32_base_atomics

Name String

cl_khr_local_int32_base_atomics

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_local_int32_base_atomics` allows OpenCL C atomic operations to be performed on 32-bit signed and unsigned integers in local memory.

This extension became a core feature in OpenCL 1.1, with the built-in atomic function names changed to use the **atomic**_ prefix instead of **atom**_.

See the Local 32-Bit Base Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_local_int32_extended_atomics

Name String

`cl_khr_local_int32_extended_atomics`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_local_int32_extended_atomics` allows OpenCL C extended atomic operations to be performed on 32-bit signed and unsigned integers in local memory.

This extension became a core feature in OpenCL 1.1, with the built-in atomic function names changed to use the **atomic**_ prefix instead of **atom**_.

See the Local 32-Bit Extended Atomics section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_mipmap_image

Name String

`cl_khr_mipmap_image`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_mipmap_image` extension adds the ability to create and access mipmapped images:

- **clCreateImage** is extended to create mipmapped images.
- **clCreateFromGLTexture** is extended to create a mipmapped image from a mipmapped GL texture.
- **clEnqueueReadImage**, **clEnqueueWriteImage**, **clEnqueueCopyImage**, **clEnqueueFillImage**, **clEnqueueCopyImageToBuffer**, **clEnqueueCopyBufferToImage**, and **clEnqueueMapImage** are extended to operate on regions of mipmapped images.
  - The Specifying Mipmap Levels to Image Operations section describes how mipmap levels are encoded in existing parameters to these commands.
- OpenCL C built-in functions are added to read from and query a mipmapped image.

See the Mipmapped Image Reads and Queries section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_mipmap_image_writes

Name String

`cl_khr_mipmap_image_writes`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_mipmap_image`

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_mipmap_image_writes` extension adds OpenCL C built-in functions to write to a mipmapped image.

If `cl_khr_mipmap_image_writes` is supported by the OpenCL device, the `cl_khr_mipmap_image` extension must also be supported.

See the Mipmapped Image Writes section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_pci_bus_info

Name String

`cl_khr_pci_bus_info`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2021-04-19

IP Status

No known IP claims.

Description

The `cl_khr_pci_bus_info` extension adds a new query to obtain PCI bus information about an OpenCL device.

Not all OpenCL devices have PCI bus information, either due to the device not being connected to the system through a PCI interface or due to platform specific restrictions and policies. Thus this extension is only expected to be supported by OpenCL devices which can provide the information.

As a consequence, applications should always check for the presence of the extension string for each individual OpenCL device for which they intend to issue the new query for and should not have any assumptions about the availability of the extension on any given platform.

New Types

- `cl_device_pci_bus_info_khr`

New Enums

- `cl_device_info`
  - `CL_DEVICE_PCI_BUS_INFO_KHR`

Version History

- Revision 1.0.0, 2021-04-19
  - Initial version.

cl_khr_priority_hints

Name String

`cl_khr_priority_hints`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_priority_hints` extension adds priority hints for OpenCL, but does not specify the scheduling behavior or minimum guarantees. It is expected that the user guides associated with each implementation which supports this extension will describe the scheduling behavior guarantees.

Note that the priority hint is orthogonal to functionality defined in the `cl_khr_throttle_hints` extension. For example, a task may have high priority (`CL_QUEUE_PRIORITY_HIGH_KHR`) but should at the same time be executed at an optimized throttle setting (`CL_QUEUE_THROTTLE_LOW_KHR`).

New Types

- `cl_queue_priority_khr`

New Enums

- `cl_queue_properties`
  - `CL_QUEUE_PRIORITY_KHR`
- `cl_queue_priority_khr`
  - `CL_QUEUE_PRIORITY_HIGH_KHR`
  - `CL_QUEUE_PRIORITY_MED_KHR`
  - `CL_QUEUE_PRIORITY_LOW_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_select_fprounding_mode

Name String

`cl_khr_select_fprounding_mode`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Obsoleted* by OpenCL 1.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_select_fprounding_mode` allows an application to specify the rounding mode for an instruction or group of instructions in the OpenCL C program source.

This extension was deprecated in OpenCL 1.1, and its use is not recommended.

See the Select Floating-Point Rounding Mode section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_semaphore

Name String

`cl_khr_semaphore`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`

Other Extension Metadata

Last Modified Date

2024-03-15

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA

- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- Gorazd Sumkovski, ARM
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

OpenCL provides `cl_event` as a primary mechanism of synchronization between host and device as well as across devices. While events can be waited on or can be passed as dependencies across work-submissions, they suffer from following limitations:

- They are immutable.
- They are not reusable.

`cl_khr_semaphore` introduces a new type of synchronization object to represent *semaphores* that can be reused, waited on, and signaled multiple times by OpenCL work-submissions.

In particular, this extension defines:

- a new type called `cl_semaphore_khr` to represent the semaphore objects.
- A new type called `cl_semaphore_properties_khr` to specify metadata associated with semaphores.
- Functions to create, retain, and release semaphores.
- Functions to wait on and signal semaphore objects.
- Functions to query the properties of semaphore objects.

New Commands

- **clCreateSemaphoreWithPropertiesKHR**
- **clEnqueueWaitSemaphoresKHR**

- **clEnqueueSignalSemaphoresKHR**
- **clGetSemaphoreInfoKHR**
- **clReleaseSemaphoreKHR**
- **clRetainSemaphoreKHR**

New Types

- `cl_semaphore_khr`
- `cl_semaphore_properties_khr`
- `cl_semaphore_info_khr`
- `cl_semaphore_type_khr`
- `cl_semaphore_payload_khr`

New Enums

- `cl_platform_info`
  - `CL_PLATFORM_SEMAPHORE_TYPES_KHR`
- `cl_device_info`
  - `CL_DEVICE_SEMAPHORE_TYPES_KHR`
- `cl_semaphore_type_khr`
  - `CL_SEMAPHORE_TYPE_BINARY_KHR`
- `cl_semaphore_info_khr`
  - `CL_SEMAPHORE_CONTEXT_KHR`
  - `CL_SEMAPHORE_REFERENCE_COUNT_KHR`
  - `CL_SEMAPHORE_PROPERTIES_KHR`
  - `CL_SEMAPHORE_PAYLOAD_KHR`
- `cl_semaphore_info_khr` or `cl_semaphore_properties_khr`
  - `CL_SEMAPHORE_TYPE_KHR`
  - `CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR`
  - `CL_SEMAPHORE_DEVICE_HANDLE_LIST_END_KHR`
- `cl_command_type`
  - `CL_COMMAND_SEMAPHORE_WAIT_KHR`
  - `CL_COMMAND_SEMAPHORE_SIGNAL_KHR`
- New Error Codes
  - `CL_INVALID_SEMAPHORE_KHR`

Sample Code

Example for Semaphore Creation in a Single Device Context

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with just first device
context = clCreateContext(..., 1, devices, ...);

// Create clSema of type cl_semaphore_khr usable on single device in the context

cl_semaphore_properties_khr sema_props[] =
        {(cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_KHR,
         (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_BINARY_KHR,
          0};

int errcode_ret = 0;

cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);
```

Example for Semaphore Creation for a Single Device in a Multi-Device Context

```
// Get cl_devices of the platform.
clGetDeviceIDs(..., &devices, &deviceCount);

// Create cl_context with first two devices
clCreateContext(..., 2, devices, ...);

// Create clSema of type cl_semaphore_khr usable only on device 0
cl_semaphore_properties_khr sema_props[] = {
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_TYPE_BINARY_KHR,
    (cl_semaphore_properties_khr)CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR,
    (cl_semaphore_properties_khr)devices[0],
    CL_SEMAPHORE_DEVICE_HANDLE_LIST_END_KHR,
    0
};

int errcode_ret = 0;

cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);
```

Example for Synchronization Using Wait and Signal

```
// clSema is created using clCreateSemaphoreWithPropertiesKHR
```

```
// using one of the examples for semaphore creation.

cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);

// Start the main loop

while (true) {
    // (not shown) Signal the semaphore from other work

    // Wait for the semaphore in OpenCL
    // by calling clEnqueueWaitSemaphoresKHR on 'clSema'
    clEnqueueWaitSemaphoresKHR(/*command_queue*/              command_queue,
                               /*num_sema_objects*/           1,
                               /*sema_objects*/               &clSema,
                               /*sema_payload_list*/          NULL,
                               /*num_events_in_wait_list*/    0,
                               /*event_wait_list*/            NULL,
                               /*event*/                      NULL);

    // Launch kernel that accesses extMem
    clEnqueueNDRangeKernel(command_queue, ...);

    // Signal the semaphore in OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/              command_queue,
                                 /*num_sema_objects*/           1,
                                 /*sema_objects*/               &clSema,
                                 /*sema_payload_list*/          NULL,
                                 /*num_events_in_wait_list*/    0,
                                 /*event_wait_list*/            NULL,
                                 /*event*/                      NULL);

    // (not shown) Launch other work that waits on 'clSema'
}
```

Example for clGetSemaphoreInfoKHR

```
// clSema is created using clCreateSemaphoreWithPropertiesKHR
// using one of the examples for semaphore creation.

cl_semaphore_khr clSema = clCreateSemaphoreWithPropertiesKHR(context,
                                                             sema_props,
                                                             &errcode_ret);

// Start the main rendering loop

while (true) {
    // (not shown) Signal the semaphore from other work
```

```
    // Wait for the semaphore in OpenCL, by calling clEnqueueWaitSemaphoresKHR on
'clSema'
    clEnqueueWaitSemaphoresKHR(/*command_queue*/              command_queue,
                               /*num_sema_objects*/           1,
                               /*sema_objects*/               &clSema,
                               /*sema_payload_list*/          NULL,
                               /*num_events_in_wait_list*/    0,
                               /*event_wait_list*/            NULL,
                               /*event*/                      NULL);

    // Launch kernel in OpenCL
    clEnqueueNDRangeKernel(command_queue, ...);

    // Signal the semaphore in OpenCL
    clEnqueueSignalSemaphoresKHR(/*command_queue*/              command_queue,
                                 /*num_sema_objects*/           1,
                                 /*sema_objects*/               &clSema,
                                 /*sema_payload_list*/          NULL,
                                 /*num_events_in_wait_list*/    0,
                                 /*event_wait_list*/            NULL,
                                 /*event*/                      NULL);

    // Query type of clSema
    clGetSemaphoreInfoKHR(/*sema_object*/           clSema,
                          /*param_name*/            CL_SEMAPHORE_TYPE_KHR,
                          /*param_value_size*/      sizeof(cl_semaphore_type_khr),
                          /*param_value*/           &clSemaType,
                          /*param_value_ret_size*/  &clSemaTypeSize);

    if (clSemaType == CL_SEMAPHORE_TYPE_BINARY_KHR) {
        // Do something
    }
    else {
        // Do something else
    }
    // (not shown) Launch other work that waits on 'clSema'
}
```

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2023-08-01
  - Changed device handle list enum to the semaphore-specific `CL_SEMAPHORE_DEVICE_HANDLE_LIST_KHR` (experimental).
- Revision 1.0.0, 2024-03-15
  - First non-experimental version.

- Revision 1.0.1, 2024-09-08
  - Unified `CL_INVALID_COMMAND_QUEUE` error behavior for **clEnqueueSignalSemaphoresKHR** and **clEnqueueWaitSemaphoresKHR**.

cl_khr_spirv_extended_debug_info

Name String

`cl_khr_spirv_extended_debug_info`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_spirv_extended_debug_info` allows use of the SPIR-V `OpenCL.DebugInfo.100` extended instruction set.

See the cl_khr_spirv_extended_debug_info section of the OpenCL SPIR-V Environment specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_spirv_linkonce_odr

Name String

`cl_khr_spirv_linkonce_odr`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_spirv_linkonce_odr` extension allows use of the SPIR-V extension `SPV_KHR_linkonce_odr`.

See the cl_khr_spirv_linkonce_odr section of the OpenCL SPIR-V Environment specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_spirv_no_integer_wrap_decoration

Name String

`cl_khr_spirv_no_integer_wrap_decoration`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_spirv_no_integer_wrap_decoration` allows use of the SPIR-V extension `SPV_KHR_no_integer_wrap_decoration`, which adds new decorations to indicate that a given instruction does not cause integer wrapping to occur.

See the cl_khr_spirv_no_integer_wrap_decoration section of the OpenCL SPIR-V Environment specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_spirv_queries

Name String

`cl_khr_spirv_queries`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2025-06-04

IP Status

No known IP claims.

Description

`cl_khr_spirv_queries` adds queries to determine the SPIR-V extended instruction sets, SPIR-V extensions, and SPIR-V capabilities that are supported by an OpenCL device.

New Enums

- `cl_device_info`
  - `CL_DEVICE_SPIRV_EXTENDED_INSTRUCTION_SETS_KHR`
  - `CL_DEVICE_SPIRV_EXTENSIONS_KHR`
  - `CL_DEVICE_SPIRV_CAPABILITIES_KHR`

Version History

- Revision 1.0.0, 2025-06-04
  - Initial version.

cl_khr_srgb_image_writes

Name String

`cl_khr_srgb_image_writes`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_srgb_image_writes` enables OpenCL C kernels to write to sRGB images using the **write_imagef** built-in function. The sRGB image formats that may be written to will be returned by **clGetSupportedImageFormats**.

When the image is an sRGB image, the **write_imagef** built-in function will perform the linear to sRGB conversion. Only the R, G, and B components are converted from linear to sRGB; the A component is written as-is.

See the sRGB Image Write Functions section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_subgroup_ballot

Name String

`cl_khr_subgroup_ballot`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_ballot` adds built-in OpenCL C functions with the ability to collect and operate on ballots from work items in a sub-group.

See the Sub-Group Ballots section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_ballot:

gentype sub_group_non_uniform_broadcast( gentype value, uint index )
gentype sub_group_broadcast_first( gentype value )

uint4 sub_group_ballot( int predicate )
int   sub_group_inverse_ballot( uint4 value )
int   sub_group_ballot_bit_extract( uint4 value, uint index )
uint  sub_group_ballot_bit_count( uint4 value )
uint  sub_group_ballot_inclusive_scan( uint4 value )
uint  sub_group_ballot_exclusive_scan( uint4 value )
uint  sub_group_ballot_find_lsb( uint4 value )
uint  sub_group_ballot_find_msb( uint4 value )

uint4 get_sub_group_eq_mask()
uint4 get_sub_group_ge_mask()
uint4 get_sub_group_gt_mask()
uint4 get_sub_group_le_mask()
uint4 get_sub_group_lt_mask()
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroup_clustered_reduce

Name String

`cl_khr_subgroup_clustered_reduce`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_clustered_reduce` adds built-in OpenCL functions for clustered reductions that operate on a subset of work items in the sub-group.

See the Clustered Reductions section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_clustered_reduce:

gentype sub_group_clustered_reduce_add( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_mul( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_min( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_max( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_and( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_or( gentype value, uint clustersize )
gentype sub_group_clustered_reduce_xor( gentype value, uint clustersize )
int     sub_group_clustered_reduce_logical_and( int predicate, uint clustersize )
int     sub_group_clustered_reduce_logical_or( int predicate, uint clustersize )
int     sub_group_clustered_reduce_logical_xor( int predicate, uint clustersize )
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroup_extended_types

Name String

`cl_khr_subgroup_extended_types`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_extended_types` adds additional supported OpenCL C data types to the existing sub-group broadcast, scan, and reduction functions.

See the Sub-Group Extended Types section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_extended_types:

// Note: Existing functions supporting additional data types.

gentype sub_group_broadcast( gentype value, uint index )

gentype sub_group_reduce_add( gentype value )
gentype sub_group_reduce_min( gentype value )
gentype sub_group_reduce_max( gentype value )

gentype sub_group_scan_inclusive_add( gentype value )
gentype sub_group_scan_inclusive_min( gentype value )
gentype sub_group_scan_inclusive_max( gentype value )

gentype sub_group_scan_exclusive_add( gentype value )
gentype sub_group_scan_exclusive_min( gentype value )
gentype sub_group_scan_exclusive_max( gentype value )
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroup_named_barrier

Name String

`cl_khr_subgroup_named_barrier`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_subgroup_named_barrier` adds barrier operations that cover subsets of an OpenCL work-group. Only the OpenCL API changes are described in this section. Please refer to the SPIR-V specification for information about using sub-group named barriers in the SPIR-V intermediate representation, and to the OpenCL C++ specification for descriptions of the sub-group named barrier built-in functions in the OpenCL C++ kernel language.

New Enums

- `cl_device_info`
  - `CL_DEVICE_MAX_NAMED_BARRIER_COUNT_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_subgroup_non_uniform_arithmetic

Name String

`cl_khr_subgroup_non_uniform_arithmetic`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_non_uniform_arithmetic` adds built-in OpenCL C functions providing the ability to use some sub-group functions within non-uniform flow control, including additional scan and reduction operators.

See the Built-in Non-Uniform Arithmetic Functions for Sub-Groups section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_non_uniform_arithmetic:

gentype sub_group_non_uniform_reduce_add( gentype value )
gentype sub_group_non_uniform_reduce_mul( gentype value )
gentype sub_group_non_uniform_reduce_min( gentype value )
gentype sub_group_non_uniform_reduce_max( gentype value )
gentype sub_group_non_uniform_reduce_and( gentype value )
gentype sub_group_non_uniform_reduce_or( gentype value )
gentype sub_group_non_uniform_reduce_xor( gentype value )
int     sub_group_non_uniform_reduce_logical_and( int predicate )
int     sub_group_non_uniform_reduce_logical_or( int predicate )
int     sub_group_non_uniform_reduce_logical_xor( int predicate )

gentype sub_group_non_uniform_scan_inclusive_add( gentype value )
gentype sub_group_non_uniform_scan_inclusive_mul( gentype value )
gentype sub_group_non_uniform_scan_inclusive_min( gentype value )
gentype sub_group_non_uniform_scan_inclusive_max( gentype value )
gentype sub_group_non_uniform_scan_inclusive_and( gentype value )
gentype sub_group_non_uniform_scan_inclusive_or( gentype value )
gentype sub_group_non_uniform_scan_inclusive_xor( gentype value )
int     sub_group_non_uniform_scan_inclusive_logical_and( int predicate )
int     sub_group_non_uniform_scan_inclusive_logical_or( int predicate )
int     sub_group_non_uniform_scan_inclusive_logical_xor( int predicate )

gentype sub_group_non_uniform_scan_exclusive_add( gentype value )
gentype sub_group_non_uniform_scan_exclusive_mul( gentype value )
gentype sub_group_non_uniform_scan_exclusive_min( gentype value )
gentype sub_group_non_uniform_scan_exclusive_max( gentype value )
gentype sub_group_non_uniform_scan_exclusive_and( gentype value )
gentype sub_group_non_uniform_scan_exclusive_or( gentype value )
gentype sub_group_non_uniform_scan_exclusive_xor( gentype value )
int     sub_group_non_uniform_scan_exclusive_logical_and( int predicate )
int     sub_group_non_uniform_scan_exclusive_logical_or( int predicate )
int     sub_group_non_uniform_scan_exclusive_logical_xor( int predicate )
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroup_non_uniform_vote

Name String

`cl_khr_subgroup_non_uniform_vote`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

Description

`cl_khr_subgroup_non_uniform_vote` adds built-in OpenCL C functions with the ability to elect a single work item from a sub-group to perform a task and to hold votes among work items in a sub-group.

See the Built-in Non-Uniform Vote and Election Functions for Sub-Groups section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_non_uniform_vote:

int sub_group_elect()

int sub_group_non_uniform_all( int predicate )
int sub_group_non_uniform_any( int predicate )
int sub_group_non_uniform_all_equal( gentype value )
```

Version History

- Revision 1.0.0, 2020-12-15

    - First assigned version.

cl_khr_subgroup_rotate

Name String

`cl_khr_subgroup_rotate`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2022-04-22

IP Status

No known IP claims.

Contributors

- Kévin Petit, Arm Ltd.
- Ben Ashbaugh, Intel
- Ruihao Zhang, Qualcomm
- Sven van Haastregt, Arm Ltd.
- Anastasia Stulova, Arm Ltd.
- Stuart Brady, Arm Ltd.

Description

`cl_khr_subgroup_rotate` adds built-in OpenCL C functions with support for a new sub-group data exchange operation that makes it possible to rotate values through the work items in a sub-group.

See the Sub-Group Rotation section of the OpenCL C specification for more information.

Version History

- Revision 1.0.0, 2022-04-22
    - Initial version.

cl_khr_subgroup_shuffle

Name String

`cl_khr_subgroup_shuffle`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_shuffle` adds built-in OpenCL C functions providing additional ways to exchange data among work items in a sub-group.

See the General Purpose Shuffles section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_shuffle:

gentype sub_group_shuffle( gentype value, uint index )
gentype sub_group_shuffle_xor( gentype value, uint mask )
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroup_shuffle_relative

Name String

`cl_khr_subgroup_shuffle_relative`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2020-12-15

IP Status

No known IP claims.

Description

`cl_khr_subgroup_shuffle_relative` adds built-in OpenCL C functions providing specialized ways to exchange data among work items in a sub-group that may perform better on some implementations.

See the Relative Shuffles section of the OpenCL C specification for more information.

Summary of New OpenCL C Functions

```
// These functions are available to devices supporting
// cl_khr_subgroup_shuffle_relative:

gentype sub_group_shuffle_up( gentype value, uint delta )
gentype sub_group_shuffle_down( gentype value, uint delta )
```

Version History

- Revision 1.0.0, 2020-12-15
  - First assigned version.

cl_khr_subgroups

Name String

`cl_khr_subgroups`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 2.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_subgroups` adds support for implementation-controlled groups of work items, known as sub-groups. Sub-groups behave similarly to work-groups and have their own sets of built-ins and synchronization primitives. Sub-groups within a work-group are independent, may make forward progress with respect to each other, and may map to optimized hardware structures where that makes sense.

Sub-groups were promoted to a core feature in OpenCL 2.1. However, note that:

- The sub-group OpenCL C built-in functions described by this extension must still be accessed as an OpenCL C extension in OpenCL 2.1.
- Sub-group independent forward progress is an optional device property in OpenCL 2.1, see `CL_DEVICE_SUB_GROUP_INDEPENDENT_FORWARD_PROGRESS`.

See the Sub-Groups section of the OpenCL C specification for more information.

New Commands

- **clGetKernelSubGroupInfoKHR**

New Types

- `cl_kernel_sub_group_info`

New Enums

- `cl_kernel_sub_group_info`
  - `CL_KERNEL_MAX_SUB_GROUP_SIZE_FOR_NDRANGE_KHR`
  - `CL_KERNEL_SUB_GROUP_COUNT_FOR_NDRANGE_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_suggested_local_work_size

Name String

`cl_khr_suggested_local_work_size`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Promoted* to OpenCL 3.1

Other Extension Metadata

Last Modified Date

2021-04-22

IP Status

No known IP claims.

Description

`cl_khr_suggested_local_work_size` adds the ability to query a suggested local work-group size for a kernel running on a device for a specified global work size and global work offset. The suggested local work-group size will match the work-group size that would be chosen if the kernel were enqueued with the specified global work size and global work offset and a `NULL` local work size.

By using the suggested local work-group size query an application has greater insight into the local work-group size chosen by the OpenCL implementation, and the OpenCL implementation need not re-compute the local work-group size if the same kernel is enqueued multiple times with the same parameters.

New Commands

- **clGetKernelSuggestedLocalWorkSizeKHR**

Version History

- Revision 1.0.0, 2021-04-22
  - Initial version.

cl_khr_terminate_context

Name String

`cl_khr_terminate_context`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_terminate_context` extension provides a new query to check whether a device can terminate an OpenCL context, and adds an API to terminate a context.

Today, OpenCL provides an API to release a context. This operation is done only after all queues, memory object, programs and kernels are released, which in turn might wait for all ongoing operations to complete. However, there are cases in which a fast release is required, or release operation cannot be done, as commands are stuck in mid execution. An example of the first case can be program termination due to exception, or quick shutdown due to low power. Examples of the second case are when a kernel is running too long, or gets stuck, or it may result from user action which makes the results of the computation unnecessary.

In many cases, the driver or the device is capable of speeding up the closure of ongoing operations when the results are no longer required in a much more expedient manner than waiting for all previously enqueued operations to finish.

New Commands

- **clTerminateContextKHR**

New Types

- `cl_device_terminate_capability_khr`

New Enums

- `cl_device_info`
  - `CL_DEVICE_TERMINATE_CAPABILITY_KHR`
- `cl_context_properties`
  - `CL_CONTEXT_TERMINATE_KHR`
- `cl_device_terminate_capability_khr`
  - `CL_DEVICE_TERMINATE_CAPABILITY_CONTEXT_KHR`

- New Error codes
  - `CL_CONTEXT_TERMINATED_KHR`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

cl_khr_throttle_hints

Name String

`cl_khr_throttle_hints`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

The `cl_khr_throttle_hints` extension adds throttle hints for OpenCL, but does not specify the throttling behavior or minimum guarantees. It is expected that the user guide associated with each implementation which supports this extension will describe the throttling behavior guarantees.

Note that the throttle hint is orthogonal to functionality defined in `cl_khr_priority_hints` extension. For example, a task may have high priority (`CL_QUEUE_PRIORITY_HIGH_KHR`) but should at the same time be executed at an optimized throttle setting (`CL_QUEUE_THROTTLE_LOW_KHR`).

New Types

- `cl_queue_throttle_khr`

New Enums

- `cl_queue_properties`
  - `CL_QUEUE_THROTTLE_KHR`
- `cl_queue_throttle_khr`
  - `CL_QUEUE_THROTTLE_HIGH_KHR`

    - `CL_QUEUE_THROTTLE_MED_KHR`
    - `CL_QUEUE_THROTTLE_LOW_KHR`

Version History

- Revision 1.0.0, 2020-04-21
    - First assigned version.

cl_khr_work_group_uniform_arithmetic

Name String

`cl_khr_work_group_uniform_arithmetic`

Ratification Status

Ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2022-04-29

IP Status

No known IP claims.

Contributors

- Kevin Petit, Arm Ltd.
- Ben Ashbaugh, Intel

Description

`cl_khr_work_group_uniform_arithmetic` adds additional built-in work-group collective functions to OpenCL C. Specifically, this extension adds support for work-group scans and reductions for the following operators:

- Logical operations (`and`, `or`, and `xor`).
- Bitwise operations (`and`, `or`, and `xor`).
- Integer multiplication (`mul`).
- Floating-point multiplication (`mul`).

See the Work-group Collective Uniform Arithmetic Functions section of the OpenCL C specification for more information.

Issues

1. For these built-in functions, do we only want to support the types supported by the existing work-group collective functions, or do we want to support the types supported by the sub-group collective functions?

   `RESOLVED`: The extension will require the same types as the existing work-group collective functions.

   The difference are the 8-bit and 16-bit types: `char`, `uchar`, `short`, and `ushort`. Note that `half` is already supported, if half-precision is supported.

Version History

- Revision 1.0.0, 2022-04-29
  - Initial version.

cl_ext_buffer_device_address

Name String

`cl_ext_buffer_device_address`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 3.0`

Other Extension Metadata

Last Modified Date

2025-02-04

IP Status

No known IP claims.

Contributors

- Pekka Jääskeläinen, Intel
- Karol Herbst, Red Hat
- Ben Ashbaugh, Intel
- Kevin Petit, Arm
- Henry Linjamäki, Intel

Description

This extension provides access to raw device pointers for `cl_mem` buffers without requiring a shared virtual address space between the host and the device.

Background

Shared Virtual Memory (SVM) introduced in OpenCL 2.0 is the first feature that enables raw pointers in the OpenCL standard. Its coarse-grain variant is relatively simple to implement on various platforms in terms of coherency requirements, but it requires mapping the buffer's address range to the host virtual address space. However, various higher-level heterogeneous APIs present a memory allocation routine which can allocate device-only memory and provide raw addresses to it without guarentees of system-wide uniqueness. For example, minimal implementations of OpenMP's omp_target_alloc() and CUDA/HIP's cudaMalloc()/hipMalloc() do not require a shared address space between the host and the device.

Host-device unified addressing might not be a major implementation issue in systems which can provide virtual memory across the platform, but might bring challenges in cases where the device presents a global memory with a disjoint address space (that can also be a physical memory address space) or, for example, when a barebone embedded system lacks virtual memory support altogether. This extension is targeted to complement the OpenCL SVM extension by providing an additional lower-end step in the spectrum of type of pointers/buffers OpenCL can allocate.

New Commands

- **clSetKernelArgDevicePointerEXT**

New Types

- `cl_mem_device_address_ext`

New enums

- `cl_mem_properties`
  - `CL_MEM_DEVICE_PRIVATE_ADDRESS_EXT`
- `cl_mem_info`
  - `CL_MEM_DEVICE_ADDRESS_EXT`
- `cl_kernel_exec_info`
  - `CL_KERNEL_EXEC_INFO_DEVICE_PTRS_EXT`

Version History

- Revision 1.0.0, 2025-01-15
  - Initial version for detailed review.
- Revision 1.0.1, 2025-01-28
  - Made it explicit that passing illegal pointers is legal as long as they are not referenced. Removed `CL_INVALID_ARG_VALUE` as a possible error in **clSetKernelArgDevicePointerEXT** as there are no illegal pointer cases when calling this function. Return `CL_INVALID_OPERATION` for **clGetMemObjectInfo** if the pointer is not a buffer device pointer. **clSetKernelExecInfo** and **clSetKernelArgDevicePointerEXT** now only error out if no devices in the context associated with kernel support device pointers.

- Revision 1.0.2, 2025-02-04
    - Converted the **clSetKernelArgDevicePointerEXT** address parameter to a value instead of a pointer to the value.

cl_ext_cxx_for_opencl

Name String

`cl_ext_cxx_for_opencl`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 3.0`
or
`OpenCL 2.0`
and
`cl_khr_extended_versioning`

Other Extension Metadata

Last Modified Date

2020-08-25

IP Status

No known IP claims.

Contributors

- Kevin Petit, Arm Ltd.
- Sven Van Haastregt, Arm Ltd.
- Anastasia Stulova, Arm Ltd.
- Marco Antognini, Arm Ltd.
- Neil Hickey, Arm Ltd.
- Alastair Murray, Codeplay

Description

This extension adds support for building programs written using the C++ for OpenCL kernel language documented in the **OpenCL-Docs** repository (https://github.com/KhronosGroup/OpenCL-Docs) with stable versions published in releases of the repository.

This extension also enables applications to query the version of the language supported by the device compiler.

New enums

- `cl_device_info`
  - `CL_DEVICE_CXX_FOR_OPENCL_NUMERIC_VERSION_EXT`

New build option

This extension adds support for a new `CLC++` value to be passed to the `-cl-std` build option accepted by **clBuildProgram** and **clCompileProgram**.

Preprocessor Macros

This extension defines a new language, instead of extending an existing language. As such, there will be no preprocessor `#define` matching the extension name string. Instead, dedicated preprocessor macros conveying language version information are available as described in the C++ for OpenCL Programming Language Documentation, section 2.2.2.2 "Predefined macros".

Conformance tests

1. Test that a program can successfully be compiled with `-cl-std=CLC++`.
2. Test with a program compiled with `-cl-std=CLC++` that the value of the `__OPENCL_CPP_VERSION__` macro agrees with the version returned by `CL_DEVICE_CXX_FOR_OPENCL_NUMERIC_VERSION_EXT`.

Version History

- Revision 1.0.0, 2020-08-24
  - Initial version.

cl_ext_device_fission

Name String

`cl_ext_device_fission`

Ratification Status

Not ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2024-07-17

IP Status

No known IP claims.

Description

Precursor to the functionality described in Partitioning a Device.

Version History

- Revision 1.0.0, 2024-07-17
  - First version.

cl_ext_float_atomics

Name String

`cl_ext_float_atomics`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 2.0`
or
`OpenCL 3.0`

Other Extension Metadata

Last Modified Date

2020-08-12

IP Status

No known IP claims.

Description

The latest published specification for this extension is available on the OpenCL registry.

Version History

- Revision 1.0.0, 2020-08-12
  - First version.

cl_ext_image_from_buffer

Name String

`cl_ext_image_from_buffer`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 3.0`
and
`cl_ext_image_requirements_info`

Other Extension Metadata

Last Modified Date

2025-01-15

IP Status

No known IP claims.

Contributors

- Kevin Petit, Arm Ltd.
- Jeremy Kemp, Imagination Technologies
- Alastair Murray, Codeplay Software Ltd.
- Balaji Calidas, Qualcomm

Description

This extension enables all types of images to be created from an existing buffer object.

New enums

- `cl_image_requirements_info_ext`
  - `CL_IMAGE_REQUIREMENTS_SLICE_PITCH_ALIGNMENT_EXT`

Conformance tests

1. Test access from kernel
   - For all image types
   - For a few/all image formats
   - For several values of row/slice pitch
   - With or without a host_ptr
   - Create buffer and fill with data
   - Optionally create a sub-buffer with a randomly selected offset?
   - Create an image from the buffer
   - Read the image from a kernel and compare with values read using the buffer and direct addressing. They must match.

1. Test **clGetImageInfo**
   - For all image types (one format per element size)
   - For a few different row/pitch sizes (image dimensions being equal or not)

    - Create an image from a buffer
    - Check that the returned values for `CL_IMAGE_ROW_PITCH` and `CL_IMAGE_SLICE_PITCH` are correct.
2. Test **clGetMemObjectInfo**
    - For all image types (1 format only)
    - Create an image from a buffer
    - Check that `CL_MEM_ASSOCIATED_MEMOBJECT` correctly returns the buffer that was used.
3. Negative testing for **clCreateImage** (alignment)
    - For a few/all image formats
    - For all image types
    - Query row pitch, slice pitch and base image address alignment for the format
    - Create an image from a buffer with invalid row pitch (not a multiple of required alignment) and check that `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` is returned.
    - Create an image from a buffer with invalid slice pitch (not a multiple of required alignment) and check that `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` is returned.
    - Create an image from a buffer with invalid base address alignment (not a multiple of required alignment) and check that `CL_INVALID_IMAGE_FORMAT_DESCRIPTOR` is returned.
4. Negative testing for **clCreateImage** (buffer size)
    - For a few image formats (at least smallest and biggest element types)
    - For all image types
    - Create a buffer too small
    - Check that image creation from that buffer is rejected with `CL_INVALID_IMAGE_SIZE`

Issues

None.

Version History

- Revision 1.0.0, 2022-01-25
    - First version.

cl_ext_image_raw10_raw12

Name String

`cl_ext_image_raw10_raw12`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 1.2`

Other Extension Metadata

Last Modified Date

2023-05-03

IP Status

No known IP claims.

Description

The latest published specification for this extension is available on the OpenCL registry.

New enums

- `cl_channel_type`
  - `CL_UNSIGNED_INT_RAW10_EXT`
  - `CL_UNSIGNED_INT_RAW12_EXT`

Version History

- Revision 1.0.0, 2023-05-03
  - First version.

cl_ext_image_requirements_info

Name String

`cl_ext_image_requirements_info`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 3.0`

Other Extension Metadata

Last Modified Date

2025-01-06

IP Status

No known IP claims.

Interactions and External Dependencies

- This extension interacts with `cl_khr_image2d_from_buffer`

Contributors

- Kevin Petit, Arm Ltd.
- Jeremy Kemp, Imagination Technologies

- Alastair Murray, Codeplay Software Ltd.
- Balaji Calidas, Qualcomm

Description

This extension enables applications to query requirements for an image without having to create the image.

New Commands

- **clGetImageRequirementsInfoEXT**

New Types

- `cl_image_requirements_info_ext`

New enums

- `cl_image_requirements_info_ext`
  - `CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT`
  - `CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT`
  - `CL_IMAGE_REQUIREMENTS_SIZE_EXT`
  - `CL_IMAGE_REQUIREMENTS_MAX_WIDTH_EXT`
  - `CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT`
  - `CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT`
  - `CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT`

Conformance tests

1. Basic checks for `CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT` and `CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT`
   - For all image formats and types
     - Check that the `CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT` and `CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT` queries can be performed successfully and that the values returned are a power of two.
2. Check consistency with `cl_khr_image2d_from_buffer`
   - When `cl_khr_image2d_from_buffer` is supported, check that the value returned by `CL_DEVICE_IMAGE_PITCH_ALIGNMENT` after converting in bytes for the supported format with the biggest element size (channel data type size * number of channels) is greater than or equal to the value returned by `CL_IMAGE_REQUIREMENTS_ROW_PITCH_ALIGNMENT_EXT` for all supported formats.
   - When `cl_khr_image2d_from_buffer` is supported, check that the value returned by `CL_DEVICE_IMAGE_BASE_ADDRESS_ALIGNMENT` after converting in bytes for the supported format with the biggest element size (channel data type size * number of channels) is greater than or equal to the value returned by `CL_IMAGE_REQUIREMENTS_BASE_ADDRESS_ALIGNMENT_EXT` for all supported

formats.

3. Negative tests for `CL_IMAGE_REQUIREMENTS_SIZE_EXT`
   - Check that attempting to perform the `CL_IMAGE_REQUIREMENTS_SIZE_EXT` query without specifying the *image_format* results in `CL_INVALID_VALUE` being returned.
   - Check that attempting to perform the `CL_IMAGE_REQUIREMENTS_SIZE_EXT` query without specifying the *image_desc* results in `CL_INVALID_VALUE` being returned.
4. Consistency checks for `CL_IMAGE_REQUIREMENTS_SIZE_EXT`
   - When creating 2D images from a buffer is supported, for all formats and a selection of image dimensions
     - Check that the `CL_IMAGE_REQUIREMENTS_SIZE_EXT` query can be performed successfully.
     - Create a buffer with the size returned and check that an image can successfully be created from the buffer.
     - Check that the value returned for `CL_MEM_SIZE` for the image is the same as the value returned for `CL_IMAGE_REQUIREMENTS_SIZE_EXT`.
5. Consistency checks for `CL_IMAGE_REQUIREMENTS_MAX_WIDTH_EXT`
   - For all image formats, image types and a selection of values for other members in *image_desc* (that MUST include `0`)
     - Check that the `CL_IMAGE_REQUIREMENTS_MAX_WIDTH_EXT` query can be performed successfully
     - Check that the value is smaller than or equal to the value returned for `CL_DEVICE_IMAGE_MAX_BUFFER_SIZE` for images of `CL_MEM_OBJECT_IMAGE1D_BUFFER` type or smaller than or equal to the value returned for `CL_DEVICE_IMAGE3D_MAX_WIDTH` for images of `CL_MEM_OBJECT_IMAGE3D` type or smaller than or equal to the value returned for `CL_DEVICE_IMAGE2D_MAX_WIDTH` for all other image types.
6. Negative tests for `CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT`
   - Attempt to perform the `CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT` query on all image types for which it is not valid
   - Check that `CL_INVALID_IMAGE_DESCRIPTOR` is returned in all cases.
7. Consistency checks for `CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT`
   - For all image formats, valid image types and a selection of values for other members in *image_desc* (that MUST include `0`)
     - Check that the `CL_IMAGE_REQUIREMENTS_MAX_HEIGHT_EXT` query can be performed successfully
     - Check that the value is smaller than or equal to the value returned for `CL_DEVICE_IMAGE2D_MAX_HEIGHT` for 2D or 2D array images or `CL_DEVICE_IMAGE3D_MAX_HEIGHT` for 3D images.
8. Negative testing for `CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT`
   - Attempt to perform the `CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT` query on all image types for which it is not valid
   - Check that `CL_INVALID_IMAGE_DESCRIPTOR` is returned in all cases.

9. Consistency checks for `CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT`
    - For all image formats, valid image types and a selection of values for other members in *image_desc* (that MUST include `0`)
        - Check that the `CL_IMAGE_REQUIREMENTS_MAX_DEPTH_EXT` query can be performed successfully
        - Check that the value is smaller than or equal to the value returned for `CL_DEVICE_IMAGE3D_MAX_DEPTH`.
10. Negative testing for `CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT`
    - Attempt to perform the `CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT` query on all image types for which it is not valid
    - Check that `CL_INVALID_IMAGE_DESCRIPTOR` is returned in all cases.
11. Consistency checks for `CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT`
    - For all image formats, valid image types and a selection of values for other members in *image_desc* (that MUST include `0`)
        - Check that the `CL_IMAGE_REQUIREMENTS_MAX_ARRAY_SIZE_EXT` query can be performed successfully
        - Check that the value is smaller than or equal to the value returned for `CL_DEVICE_IMAGE_MAX_ARRAY_SIZE`.
12. General negative testing for **clGetImageRequirementsInfoEXT**
    - Write tests for all possible testable generic error codes.

Issues

None.

Version History

- Revision 0.5.0, 2022-01-18
    - First version.

cl_ext_image_unorm_int_2_101010

Name String

`cl_ext_image_unorm_int_2_101010`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 1.2`

Other Extension Metadata

Last Modified Date

2024-05-10

IP Status

No known IP claims.

Contributors

- Kévin Petit, Arm Ltd.

Description

`cl_ext_image_unorm_int_2_101010` adds support for the `CL_UNORM_INT_2_101010_EXT` image format.

OpenCL C compilers supporting this extension will define the `__opencl_c_ext_image_unorm_int_2_101010` feature macro.

New enums

- `cl_channel_type`
  - `CL_UNORM_INT_2_101010_EXT`

New feature macro

- `__opencl_c_ext_image_unorm_int_2_101010`

New OpenCL C channel data type

- `CLK_UNORM_INT_2_101010_EXT`

Version History

- Revision 1.0.0, 2024-05-10
  - Initial version

cl_ext_image_unsigned_10x6_12x4_14x2

Name String

`cl_ext_image_unsigned_10x6_12x4_14x2`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 1.2`

Other Extension Metadata

Last Modified Date

2025-03-31

IP Status

No known IP claims.

Contributors

- Kévin Petit, Arm Ltd.

Description

`cl_ext_image_unsigned_10x6_12x4_14x2` adds support for the following image formats:

- `CL_UNSIGNED_INT10X6_EXT`
- `CL_UNSIGNED_INT12X4_EXT`
- `CL_UNSIGNED_INT14X2_EXT`
- `CL_UNORM_INT10X6_EXT`
- `CL_UNORM_INT12X4_EXT`
- `CL_UNORM_INT14X2_EXT`

OpenCL C compilers supporting this extension will define the `__opencl_c_ext_image_unsigned_10x6_12x4_14x2` feature macro.

New enums

- `cl_channel_type`
  - `CL_UNSIGNED_INT10X6_EXT`
  - `CL_UNSIGNED_INT12X4_EXT`
  - `CL_UNSIGNED_INT14X2_EXT`
  - `CL_UNORM_INT10X6_EXT`
  - `CL_UNORM_INT12X4_EXT`
  - `CL_UNORM_INT14X2_EXT`

New feature macro

- `__opencl_c_ext_image_unsigned_10x6_12x4_14x2`

New OpenCL C channel data type

- `CLK_UNSIGNED_INT10X6_EXT`
- `CLK_UNSIGNED_INT12X4_EXT`
- `CLK_UNSIGNED_INT14X2_EXT`
- `CLK_UNORM_10X6_EXT`
- `CLK_UNORM_12X4_EXT`
- `CLK_UNORM_14X2_EXT`

Version History

- Revision 1.0.0, 2025-03-31
  - Initial version

cl_ext_immutable_memory_objects

Name String

`cl_ext_immutable_memory_objects`

Ratification Status

Not ratified

Extension and Version Dependencies

`OpenCL 1.2`

Other Extension Metadata

Last Modified Date

2024-11-06

IP Status

No known IP claims.

Contributors

- Kevin Petit, Arm Ltd.
- Ahmed Hesham, Arm Ltd.
- Michael Rizkalla, Arm Ltd.

Interactions

- This extension interacts with `cl_khr_egl_image`.

Description

`cl_ext_immutable_memory_objects` provides a mechanism to create memory objects that cannot be modified by the implementation after creation. This, for example, enables implementation to support image formats for which no write operations can be supported.

New enums

- `cl_mem_flags`
  - `CL_MEM_IMMUTABLE_EXT`

Issues

1) Can `CL_MEM_READ_ONLY` be used instead of `CL_MEM_IMMUTABLE_EXT`?

RESOLVED: No. Memory objects created with `CL_MEM_READ_ONLY` can be modified by copy or fill

commands and this behavior cannot be changed without breaking backwards compatibility.

Version History

- Revision 1.0.0, 2024-11-06
  - Initial version

cl_ext_migrate_memobject

Name String

`cl_ext_migrate_memobject`

Ratification Status

Not ratified

Extension and Version Dependencies

None

Other Extension Metadata

Last Modified Date

2024-07-17

IP Status

No known IP claims.

Description

Precursor to **clEnqueueMigrateMemObjects**.

Version History

- Revision 1.0.0, 2024-07-17
  - Initial version.

List of Experimental Extensions

- `cl_khr_command_buffer`
- `cl_khr_command_buffer_multi_device`
- `cl_khr_command_buffer_mutable_dispatch`
- `cl_khr_external_memory_android_hardware_buffer`
- `cl_khr_external_semaphore_dx_fence`
- `cl_khr_external_semaphore_win32`
- `cl_khr_unified_svm`

cl_khr_command_buffer

Name String

`cl_khr_command_buffer`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

API Interactions

- Interacts with CL_VERSION_2_0

Other Extension Metadata

Last Modified Date

2025-07-10

IP Status

No known IP claims.

Contributors

- Ewan Crawford, Codeplay Software Ltd.
- Gordon Brown, Codeplay Software Ltd.
- Kenneth Benzie, Codeplay Software Ltd.
- Alastair Murray, Codeplay Software Ltd.
- Jack Frankland, Codeplay Software Ltd.
- Balaji Calidas, Qualcomm Technologies Inc.
- Joshua Kelly, Qualcomm Technologies, Inc.
- Kevin Petit, Arm Ltd.
- Aharon Abramson, Intel.
- Ben Ashbaugh, Intel.
- Boaz Ouriel, Intel.
- Chris Gearing, Intel.
- Pekka Jääskeläinen, Tampere University and Intel
- Jan Solanti, Tampere University
- Nikhil Joshi, NVIDIA
- James Price, Google
- Brice Videau, Argonne National Laboratory

Description

`cl_khr_command_buffer` adds the ability to record and replay buffers of OpenCL commands.

Command-buffers enable a reduction in overhead when enqueuing the same workload multiple times. By separating the command-queue setup from dispatch, the ability to replay a set of previously created commands is introduced.

Background

On embedded devices where building a command stream accounts for a significant expenditure of resources and where workloads are often required to be pipelined, a solution that minimizes driver overhead can significantly improve the utilization of accelerators by removing a bottleneck in repeated command stream generation.

An additional motivator is lowering task execution latency, as devices can be kept occupied with work by repeated submissions, without having to wait on the host to construct commands again for a similar workload.

Rationale

The command-buffer abstraction over the generation of command streams is a proven approach which facilitates a significant reduction in driver overhead in existing real-world applications with repetitive pipelined workloads which are built on top of Vulkan, DirectX 12, and Metal.

A primary goal is for a command-buffer to avoid any interaction with application code after being enqueued until all recorded commands have completed. As such, any command which maps or migrates memory objects; reads or writes memory objects; or enqueues a native kernel, is not available for command-buffer recording. Finally commands recorded into a command buffer do not wait for or return event objects, these are instead replaced with device-side synchronization-point identifiers which enable out-of-order execution of the command-buffer commands.

Adding new entry-points for individual commands, rather than recording existing command-queue APIs with begin/end markers was a design decision made for the following reasons:

- Individually specified entry points makes it clearer to the user what's supported, as opposed to adding a large number of error conditions throughout the specification with all the restrictions.
- Prevents code forking in existing entry points for the implementer, as otherwise separate paths in each entry point need to be maintained for both the recording and normal cases.
- Allows the definition of a new device-side synchronization primitive rather than overloading `cl_event`. As use of `cl_event` in individual commands allows host interaction from callback and user-events, as well as introducing complexities when a command-buffer is enqueued multiple times regarding profiling and execution status.
- New entry points facilitate returning handles to individual commands, allowing those commands to be modified between enqueues of the command buffer. Not all command handles are used in this extension, but providing them facilitates other extensions layered on top to take advantage of them to provide additional mutable functionality.

Command Synchronization

The command-buffer object has no in-order/out-of-order property set on creation, it is out-of-order, and command ordering is defined by the dependencies set when commands are created. Command dependencies can be defined in 3 ways:

1. Device-side `cl_sync_point_khr` synchronization-points, providing an explicit list of the commands to depend on.
2. Appending a **clCommandBarrierWithWaitListKHR** barrier command.
3. Passing an in-order queue when creating the command, creating an implicit dependency on the previous command created in the command-buffer using the same queue.

clEnqueueCommandBufferKHR submissions to an out-of-order queue have the same execution semantics as other operations enqueued to an out-of-order queue, such as **clEnqueueFillBuffer**, where execution between enqueued operations may happen concurrently unless dependencies between the operations are expressed with events.

The `cl_sync_point_khr` type is defined as a `cl_uint`, giving a hard upper limit on the number of commands a command-buffer can hold as `CL_UINT_MAX`, at which point `CL_OUT_OF_RESOURCES` will be returned. However, it is likely an implementation will reach capacity before this threshold is hit.

There are no gurantees made around the values of sync-points returned from adding commands to a command-buffer. Any semantics that a could be inferred from the sync-point values returned is implementation defined.

Interactions With Other Extensions

The introduction of the command-buffer abstraction enables functionality beyond what the `cl_khr_command_buffer` extension currently provides, i.e. the recording of immutable commands to a single queue which can then be executed without commands synchronizing outside the command-buffer. Extra functionality expanding on this is provided as layered extensions on top of `cl_khr_command_buffer`. The layered extensions that currently exist are:

- `cl_khr_command_buffer_multi_device`
- `cl_khr_command_buffer_mutable_dispatch`

Having `cl_khr_command_buffer` as a minimal base specification means that the API defines mechanisms for functionality that is not enabled by this extension, these are described in the following sub-sections. `cl_khr_command_buffer` will retain its experimental extension status until other layered extensions are released, as these may reveal modifications needed to the base specification to support their intended use cases.

Command Properties

The command recording entry-points allow a `properties` parameter of new type `cl_command_properties_khr` to be passed. No properties are defined in `cl_khr_command_buffer`, but the parameter enables layered extensions to define characteristics of the individual commands.

For example, `cl_khr_command_buffer_mutable_dispatch` defines properties that can be set when

appending a kernel command with **clCommandNDRangeKernelKHR**.

Command Handles

All command recording entry-points define a `cl_mutable_command_khr` output parameter which provides a handle to the specific command being recorded. Use of these output handles is not enabled by the `cl_khr_command_buffer` extension, but the handles allow individual commands in a command-buffer to be referenced by the user.

Use of these handles is enabled in `cl_khr_command_buffer_mutable_dispatch` to give the capability for an application to use the handles to modify commands between enqueues of a command-buffer.

List of Queues

Only a single command-queue can be associated with a command-buffer in the `cl_khr_command_buffer` extension, but the API is designed so that the layered `cl_khr_command_buffer_multi_device` extension can relax this constraint to allow commands to be recorded across multiple queues in the same command-buffer, providing replay of heterogeneous task graphs.

Using multiple queue functionality will result in an error without `cl_khr_command_buffer_multi_device` to relax usage of the following API features:

- When a command-buffer is created the API enables passing a list of queues that the command-buffer will record commands to. Only a single queue is permitted in `cl_khr_command_buffer`.
- Individual command recording entry-points define a `cl_command_queue` parameter for which of the queues set on command-buffer creation that command should be record to. This must be passed as NULL in `cl_khr_command_buffer`.
- **clEnqueueCommandBufferKHR** takes a list of queues for command-buffer execution, correspond to those set on creation. Only a single queue is permitted in `cl_khr_command_buffer`.

New Commands

- **clCreateCommandBufferKHR**
- **clRetainCommandBufferKHR**
- **clReleaseCommandBufferKHR**
- **clFinalizeCommandBufferKHR**
- **clEnqueueCommandBufferKHR**
- **clCommandBarrierWithWaitListKHR**
- **clCommandCopyBufferKHR**
- **clCommandCopyBufferRectKHR**
- **clCommandCopyBufferToImageKHR**
- **clCommandCopyImageKHR**
- **clCommandCopyImageToBufferKHR**
- **clCommandFillBufferKHR**

- **clCommandFillImageKHR**
- **clCommandNDRangeKernelKHR**
- **clGetCommandBufferInfoKHR**
- The following SVM entry points are supported only with at least OpenCL 2.0, and starting from version 0.9.4 of this extension
  - **clCommandSVMMemcpyKHR**
  - **clCommandSVMMemFillKHR**

New Types

- `cl_device_command_buffer_capabilities_khr`
- `cl_command_buffer_khr`
- `cl_sync_point_khr`
- `cl_command_buffer_info_khr`
- `cl_command_buffer_state_khr`
- `cl_command_buffer_properties_khr`
- `cl_command_buffer_flags_khr`
- `cl_command_properties_khr`
- `cl_mutable_command_khr`

New Enums

- `cl_device_info`
  - `CL_DEVICE_COMMAND_BUFFER_CAPABILITIES_KHR`
  - `CL_DEVICE_COMMAND_BUFFER_REQUIRED_QUEUE_PROPERTIES_KHR`
  - `CL_DEVICE_COMMAND_BUFFER_SUPPORTED_QUEUE_PROPERTIES_KHR`
- `cl_device_command_buffer_capabilities_khr`
  - `CL_COMMAND_BUFFER_CAPABILITY_KERNEL_PRINTF_KHR`
  - `CL_COMMAND_BUFFER_CAPABILITY_DEVICE_SIDE_ENQUEUE_KHR`
- `cl_command_buffer_properties_khr`
  - `CL_COMMAND_BUFFER_FLAGS_KHR`
- `cl_command_buffer_info_khr`
  - `CL_COMMAND_BUFFER_QUEUES_KHR`
  - `CL_COMMAND_BUFFER_NUM_QUEUES_KHR`
  - `CL_COMMAND_BUFFER_REFERENCE_COUNT_KHR`
  - `CL_COMMAND_BUFFER_STATE_KHR`
  - `CL_COMMAND_BUFFER_PROPERTIES_ARRAY_KHR`
  - `CL_COMMAND_BUFFER_CONTEXT_KHR`

- cl_command_buffer_state_khr
  - CL_COMMAND_BUFFER_STATE_RECORDING_KHR
  - CL_COMMAND_BUFFER_STATE_EXECUTABLE_KHR
- cl_command_type
  - CL_COMMAND_COMMAND_BUFFER_KHR
- New Error Codes
  - CL_INVALID_COMMAND_BUFFER_KHR
  - CL_INVALID_SYNC_POINT_WAIT_LIST_KHR
  - CL_INCOMPATIBLE_COMMAND_QUEUE_KHR

Sample Code

```
#define CL_CHECK(ERROR)                                \
  if (ERROR) {                                         \
    std::cerr << "OpenCL error: " << ERROR << "\n"; \
    return ERROR;                                      \
  }

int main() {
  cl_platform_id platform;
  CL_CHECK(clGetPlatformIDs(1, &platform, nullptr));
  cl_device_id device;
  CL_CHECK(clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, nullptr));

  cl_int error;
  cl_context context =
      clCreateContext(nullptr, 1, &device, nullptr, nullptr, &error);
  CL_CHECK(error);

  const char* code = R"OpenCLC(
kernel void vector_addition(global int* tile1, global int* tile2,
                            global int* res) {
  size_t index = get_global_id(0);
  res[index] = tile1[index] + tile2[index];
}
)OpenCLC";
  const size_t length = std::strlen(code);

  cl_program program =
      clCreateProgramWithSource(context, 1, &code, &length, &error);
  CL_CHECK(error);

  CL_CHECK(clBuildProgram(program, 1, &device, nullptr, nullptr, nullptr));

  cl_kernel kernel = clCreateKernel(program, "vector_addition", &error);
  CL_CHECK(error);
```

```
    constexpr size_t frame_count = 60;
    constexpr size_t frame_elements = 1024;
    constexpr size_t frame_size = frame_elements * sizeof(cl_int);

    constexpr size_t tile_count = 16;
    constexpr size_t tile_elements = frame_elements / tile_count;
    constexpr size_t tile_size = tile_elements * sizeof(cl_int);

    cl_mem buffer_tile1 =
        clCreateBuffer(context, CL_MEM_READ_ONLY, tile_size, nullptr, &error);
    CL_CHECK(error);
    cl_mem buffer_tile2 =
        clCreateBuffer(context, CL_MEM_READ_ONLY, tile_size, nullptr, &error);
    CL_CHECK(error);
    cl_mem buffer_res =
        clCreateBuffer(context, CL_MEM_WRITE_ONLY, tile_size, nullptr, &error);
    CL_CHECK(error);

    CL_CHECK(clSetKernelArg(kernel, 0, sizeof(buffer_tile1), &buffer_tile1));
    CL_CHECK(clSetKernelArg(kernel, 1, sizeof(buffer_tile2), &buffer_tile2));
    CL_CHECK(clSetKernelArg(kernel, 2, sizeof(buffer_res), &buffer_res));

    cl_command_queue command_queue =
      clCreateCommandQueue(context, device,
                           CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &error);
    CL_CHECK(error);

    cl_command_buffer_khr command_buffer =
        clCreateCommandBufferKHR(1, &command_queue, nullptr, &error);
    CL_CHECK(error);

    cl_mem buffer_src1 =
        clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);
    cl_mem buffer_src2 =
        clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);
    cl_mem buffer_dst =
        clCreateBuffer(context, CL_MEM_WRITE_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);

    cl_sync_point_khr tile_sync_point = 0;
    for (size_t tile_index = 0; tile_index < tile_count; tile_index++) {
      std::array<cl_sync_point_khr, 2> copy_sync_points;
      CL_CHECK(clCommandCopyBufferKHR(command_buffer,
          command_queue, buffer_src1, buffer_tile1, tile_index * tile_size, 0,
          tile_size, tile_sync_point ? 1 : 0,
          tile_sync_point ? &tile_sync_point : nullptr, &copy_sync_points[0]),
          nullptr);
      CL_CHECK(clCommandCopyBufferKHR(command_buffer,
          command_queue, buffer_src2, buffer_tile2, tile_index * tile_size, 0,
```

```
        tile_size, tile_sync_point ? 1 : 0,
        tile_sync_point ? &tile_sync_point : nullptr, &copy_sync_points[1]),
        nullptr);

    cl_sync_point_khr nd_sync_point;
    CL_CHECK(clCommandNDRangeKernelKHR(command_buffer,
        command_queue, nullptr, kernel, 1, nullptr, &tile_elements, nullptr,
        copy_sync_points.size(), copy_sync_points.data(), &nd_sync_point,
        nullptr));

    CL_CHECK(clCommandCopyBufferKHR(command_buffer,
        command_queue, buffer_res, buffer_dst, 0, tile_index * tile_size,
        tile_size, 1, &nd_sync_point, &tile_sync_point, nullptr));
  }

  CL_CHECK(clFinalizeCommandBufferKHR(command_buffer));

  std::random_device random_device;
  std::mt19937 random_engine{random_device()};
  std::uniform_int_distribution<cl_int> random_distribution{
      0, std::numeric_limits<cl_int>::max() / 2};
  auto random_generator = [&]() { return random_distribution(random_engine); };

  for (size_t frame_index = 0; frame_index < frame_count; frame_index++) {
    std::array<cl_event, 2> write_src_events;
    std::vector<cl_int> src1(frame_elements);
    std::generate(src1.begin(), src1.end(), random_generator);
    CL_CHECK(clEnqueueWriteBuffer(command_queue, buffer_src1, CL_FALSE, 0,
                                  frame_size, src1.data(), 0, nullptr,
                                  &write_src_events[0]));
    std::vector<cl_int> src2(frame_elements);
    std::generate(src2.begin(), src2.end(), random_generator);
    CL_CHECK(clEnqueueWriteBuffer(command_queue, buffer_src2, CL_FALSE, 0,
                                  frame_size, src2.data(), 0, nullptr,
                                  &write_src_events[1]));

    CL_CHECK(clEnqueueCommandBufferKHR(0, NULL, command_buffer, 2,
                                       write_src_events.data(), nullptr));

    CL_CHECK(clFinish(command_queue));

    CL_CHECK(clReleaseEvent(write_src_event[0]));
    CL_CHECK(clReleaseEvent(write_src_event[1]));
  }

  CL_CHECK(clReleaseCommandBufferKHR(command_buffer));
  CL_CHECK(clReleaseCommandQueue(command_queue));

  CL_CHECK(clReleaseMemObject(buffer_src1));
  CL_CHECK(clReleaseMemObject(buffer_src2));
  CL_CHECK(clReleaseMemObject(buffer_dst));
```

```
    CL_CHECK(clReleaseMemObject(buffer_tile1));
    CL_CHECK(clReleaseMemObject(buffer_tile2));
    CL_CHECK(clReleaseMemObject(buffer_res));

    CL_CHECK(clReleaseKernel(kernel));
    CL_CHECK(clReleaseProgram(program));
    CL_CHECK(clReleaseContext(context));

    return 0;
  }
```

Issues

1. Introduce a `clCloneCommandBufferKHR` entry-point for cloning a command-buffer.

   UNRESOLVED

2. Enable detached command-buffer execution, where command-buffers are executed on their own internal queue to prevent locking user created queues for the duration of their execution.

   UNRESOLVED

3. Give users more control over command-buffer command capacity via some or all of the following mechanisms.

   - Provide a way for a user to query a command-buffer for the maximum number of commands it can hold.
   - Guarantee a minimum command capacity that an implementation must support.
   - Provide a mechanism for users to reserve command-buffer capacity on command-buffer creation.

   RESOLVED - Mechanisms to achieve this could be provided as a layered extension.

Version History

- Revision 0.9.0, 2021-11-10
  - First assigned version (experimental).
- 0.9.1, 2022-08-24
  - Specify an error if a command-buffer is finalized multiple times (experimental).
- 0.9.2, 2023-03-31
  - Introduce context query `CL_COMMAND_BUFFER_CONTEXT_KHR` (experimental).
- 0.9.3, 2023-04-04
  - Remove Invalid command-buffer state (experimental).
- 0.9.4, 2023-05-11
  - Add clCommandSVMMemcpyKHR and clCommandSVMMemFillKHR command entries

(experimental).

- 0.9.5, 2024-07-24
  - Add a properties parameter to all command recording entry-points (experimental).
- 0.9.6, 2024-10-02
  - Add device query for supported queue properties (experimental).
- 0.9.7, 2024-12-13
  - Refactor queue compatability between command-buffer creation and enqueue (experimental).
- 0.9.8, 2025-07-10
  - Rework simultaneous use definition and remove pending state (experimental).

cl_khr_command_buffer_multi_device

Name String

`cl_khr_command_buffer_multi_device`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_command_buffer`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2024-12-13

IP Status

No known IP claims.

Contributors

- Ewan Crawford, Codeplay Software Ltd.
- Gordon Brown, Codeplay Software Ltd.
- Kenneth Benzie, Codeplay Software Ltd.
- Alastair Murray, Codeplay Software Ltd.
- Jack Frankland, Codeplay Software Ltd.
- Balaji Calidas, Qualcomm Technologies Inc.
- Joshua Kelly, Qualcomm Technologies, Inc.
- Kevin Petit, Arm Ltd.

- Aharon Abramson, Intel.
- Ben Ashbaugh, Intel.
- Boaz Ouriel, Intel.
- Pekka Jääskeläinen, Tampere University and Intel.
- Jan Solanti, Tampere University
- Nikhil Joshi, NVIDIA
- James Price, Google

Description

The `cl_khr_command_buffer` extension separates command construction from enqueue by providing a mechanism to record a set of commands which can then be repeatedly enqueued. However, the commands in a command-buffer can only be recorded to a single command-queue specified on command-buffer creation.

`cl_khr_command_buffer_multi_device` extends the scope of a command-buffer to allow commands to be recorded across multiple queues in the same command-buffer, providing execution of heterogeneous task graphs from command-queues associated with different devices.

The ability for a user to deep copy an existing command-buffer so that the commands target a different device is also made possible by `cl_khr_command_buffer_multi_device`. Depending on platform support the mapping of commands to the new target device can be done either explicitly by the user, or automatically by the OpenCL runtime.

New Commands

- **clRemapCommandBufferKHR**

New Types

Bitfield for querying command-buffer capabilities of an OpenCL Platform with **clGetPlatformInfo**, see the platform queries table:

- `cl_platform_command_buffer_capabilities_khr`

New Enums

Enums for querying device command-buffer capabilities with **clGetDeviceInfo**, see the device queries table:

- `cl_device_info`
  - `CL_DEVICE_COMMAND_BUFFER_NUM_SYNC_DEVICES_KHR`
  - `CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR`
- `cl_device_command_buffer_capabilities_khr`
  - `CL_COMMAND_BUFFER_CAPABILITY_MULTIPLE_QUEUE_KHR`
- `cl_command_buffer_flags_khr`

    - CL_COMMAND_BUFFER_DEVICE_SIDE_SYNC_KHR
- cl_platform_info
    - CL_PLATFORM_COMMAND_BUFFER_CAPABILITIES_KHR
- cl_platform_command_buffer_capabilities_khr
    - CL_COMMAND_BUFFER_PLATFORM_UNIVERSAL_SYNC_KHR
    - CL_COMMAND_BUFFER_PLATFORM_REMAP_QUEUES_KHR
    - CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR

Sample Code

```
#define CL_CHECK(ERROR)                                                \
  if (ERROR) {                                                         \
    std::cerr << "OpenCL error: " << ERROR << "\n";                    \
    return ERROR;                                                      \
  }

int main() {
  cl_platform_id platform;
  CL_CHECK(clGetPlatformIDs(1, &platform, nullptr));
  cl_platform_command_buffer_capabilities_khr platform_caps;
  CL_CHECK(clGetPlatformInfo(platform,
                             CL_PLATFORM_COMMAND_BUFFER_CAPABILITIES_KHR,
                             sizeof(platform_caps), &platform_caps, NULL));
  if (!(platform_caps & CL_COMMAND_BUFFER_PLATFORM_AUTOMATIC_REMAP_KHR)) {
    std::cerr << "Command-buffer remapping not supported but used in example, "
                 "skipping\n";
    return 0;
  }

  cl_uint num_devices = 0;
  CL_CHECK(clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices));
  std::vector<cl_device_id> devices(num_devices);
  CL_CHECK(
      clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, devices.data(), nullptr));

  // Checks omitted for brevity that either a) the platform supports
  // CL_COMMAND_BUFFER_PLATFORM_UNIVERSAL_SYNC_KHR or b) each device is listed
  // in the others CL_DEVICE_COMMAND_BUFFER_SYNC_DEVICES_KHR

  cl_int error;
  cl_context context =
      clCreateContext(NULL, num_devices, devices.data(), NULL, NULL, &error);
  CL_CHECK(error);

  std::vector<cl_command_queue> queues(num_devices);
  for (cl_uint i = 0; i < num_devices; i++) {
    queues[i] = clCreateCommandQueue(context, devices[i], 0, &error);
```

```
  CL_CHECK(error);
}

const char *code = R"OpenCLC(
kernel void vector_addition(global int* tile1, global int* tile2,
                            global int* res) {
  size_t index = get_global_id(0);
  res[index] = tile1[index] + tile2[index];
}
)OpenCLC";
const size_t length = std::strlen(code);

cl_program program =
    clCreateProgramWithSource(context, 1, &code, &length, &error);
CL_CHECK(error);

CL_CHECK(
    clBuildProgram(program, num_devices, devices.data(), NULL, NULL, NULL));

cl_kernel kernel = clCreateKernel(program, "vector_addition", &error);
CL_CHECK(error);

constexpr size_t frame_count = 60;
constexpr size_t frame_elements = 1024;
constexpr size_t frame_size = frame_elements * sizeof(cl_int);

constexpr size_t tile_count = 16;
constexpr size_t tile_elements = frame_elements / tile_count;
constexpr size_t tile_size = tile_elements * sizeof(cl_int);

cl_mem buffer_tile1 =
    clCreateBuffer(context, CL_MEM_READ_ONLY, tile_size, NULL, &error);
CL_CHECK(error);

cl_mem buffer_tile2 =
    clCreateBuffer(context, CL_MEM_READ_ONLY, tile_size, NULL, &error);
CL_CHECK(error);

cl_mem buffer_res =
    clCreateBuffer(context, CL_MEM_WRITE_ONLY, tile_size, NULL, &error);
CL_CHECK(error);

CL_CHECK(clSetKernelArg(kernel, 0, sizeof(buffer_tile1), &buffer_tile1));
CL_CHECK(clSetKernelArg(kernel, 1, sizeof(buffer_tile2), &buffer_tile2));
CL_CHECK(clSetKernelArg(kernel, 2, sizeof(buffer_res), &buffer_res));

cl_command_buffer_khr original_cmdbuf =
    clCreateCommandBufferKHR(num_devices, queues.data(), nullptr, &error);
CL_CHECK(error);

cl_mem buffer_src1 =
```

```
    clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, NULL, &error);
CL_CHECK(error);

cl_mem buffer_src2 =
    clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, NULL, &error);
CL_CHECK(error);

cl_mem buffer_dst =
    clCreateBuffer(context, CL_MEM_READ_WRITE, frame_size, NULL, &error);
CL_CHECK(error);

cl_sync_point_khr tile_sync_point = 0;
for (size_t tile_index = 0; tile_index < tile_count; tile_index++) {
  cl_sync_point_khr copy_sync_points[2];
  CL_CHECK(clCommandCopyBufferKHR(
      original_cmdbuf, queues[tile_index % num_devices], buffer_src1,
      buffer_tile1, tile_index * tile_size, 0, tile_size,
      tile_sync_point ? 1 : 0, tile_sync_point ? &tile_sync_point : NULL,
      &copy_sync_points[0], NULL));

  CL_CHECK(clCommandCopyBufferKHR(
              original_cmdbuf, queues[tile_index % num_devices], buffer_src2,
              buffer_tile2, tile_index * tile_size, 0, tile_size,
              tile_sync_point ? 1 : 0,
              tile_sync_point ? &tile_sync_point : nullptr,
              &copy_sync_points[1], NULL));

  cl_sync_point_khr nd_sync_point;
  CL_CHECK(clCommandNDRangeKernelKHR(
      original_cmdbuf, queues[tile_index % num_devices], NULL, kernel, 1,
      NULL, &tile_elements, NULL, 2, copy_sync_points, &nd_sync_point, NULL));

  CL_CHECK(clCommandCopyBufferKHR(
      original_cmdbuf, queues[tile_index % num_devices], buffer_res,
      buffer_dst, 0, tile_index * tile_size, tile_size, 1, &nd_sync_point,
      &tile_sync_point, NULL));
}

CL_CHECK(clFinalizeCommandBufferKHR(original_cmdbuf));

std::random_device random_device;
std::mt19937 random_engine{random_device()};
std::uniform_int_distribution<cl_int> random_distribution{
    0, std::numeric_limits<cl_int>::max() / 2};
auto random_generator = [&]() { return random_distribution(random_engine); };

auto enqueue_frame = [&](cl_command_buffer_khr command_buffer) {
  for (size_t frame_index = 0; frame_index < frame_count; frame_index++) {
    std::array<cl_event, 3> enqueue_events;
    std::vector<cl_int> src1(frame_elements);
    std::generate(src1.begin(), src1.end(), random_generator);
```

```
    CL_CHECK(clEnqueueWriteBuffer(queues[0], buffer_src1, CL_FALSE, 0,
                                  frame_size, src1.data(), 0, nullptr,
                                  &enqueue_events[0]));
    std::vector<cl_int> src2(frame_elements);
    std::generate(src2.begin(), src2.end(), random_generator);
    CL_CHECK(clEnqueueWriteBuffer(queues[0], buffer_src2, CL_FALSE, 0,
                                  frame_size, src2.data(), 0, nullptr,
                                  &enqueue_events[1]));

    CL_CHECK(clEnqueueCommandBufferKHR(0, NULL, command_buffer, 2,
                                       enqueue_events.data(),
                                       &enqueue_events[2]));

    CL_CHECK(clWaitForEvents(1, enqueue_events[2]));

    for (auto e : enqueue_events) {
      CL_CHECK(clReleaseEvent(e));
    }
  }
  return 0;
};

error = enqueue_frame(original_cmdbuf);
CL_CHECK(error);

// Remap from N queues to 1 queue and run again
cl_command_buffer_khr remapped_cmdbuf = clRemapCommandBufferKHR(
    original_cmdbuf, CL_TRUE, 1, queues.data(), 0, NULL, NULL, &error);
CL_CHECK(error);

error = enqueue_frame(remapped_cmdbuf);
CL_CHECK(error);

for (unsigned i = 0; i < num_devices; ++i) {
  CL_CHECK(clReleaseCommandQueue(queues[i]));
}
CL_CHECK(clReleaseMemObject(buffer_src1));
CL_CHECK(clReleaseMemObject(buffer_src2));
CL_CHECK(clReleaseMemObject(buffer_dst));

CL_CHECK(clReleaseMemObject(buffer_tile1));
CL_CHECK(clReleaseMemObject(buffer_tile2));
CL_CHECK(clReleaseMemObject(buffer_res));

CL_CHECK(clReleaseCommandBufferKHR(original_cmdbuf));
CL_CHECK(clReleaseCommandBufferKHR(remapped_cmdbuf));

CL_CHECK(clReleaseKernel(kernel));
CL_CHECK(clReleaseProgram(program));
CL_CHECK(clReleaseContext(context));
```

```
  return 0;
}
```

Issues

1. In cl_event profiling info for a command-buffer running across the queues for several devices, how do we know what the first & last commands executed are if there is concurrent execution across devices.

   RESOLVED: Allowed an implementation to fallback to `CL_PROFILING_COMMAND_SUBMIT` and `CL_PROFILING_COMMAND_COMPLETE` when reporting `CL_PROFILING_COMMAND_START` & `CL_PROFILING_COMMAND_END`.

2. Is an atomic constraint required? This would forbid regular clEnqueue* commands, from interleaving execution on a queue which a command-buffer is being executed on.

   RESOLVED: This behavior can block parallelism, and constraint is expressible by the user through existing synchronization mechanisms if they require it.

3. It is currently an error if a set of command-queues passed to **clEnqueueCommandBufferKHR** aren't compatible with those set on recording. Should we relax this as an optional capability that allows an implementation to do a more expensive command-buffer enqueue for this case?

   RESOLVED: Added as an optional feature.

Version History

- Revision 0.9.0, 2023-04-14
  - First assigned version (experimental).
- Revision 0.9.1, 2023-04-30
  - Added clCommandSVMMemcpyKHR and clCommandSVMMemFillKHR as affected functions (experimental).
- Revision 0.9.2, 2024-12-13
  - Update clRemapCommandBufferKHR behavior to match cl_khr_command_buffer version 0.9.7 (experimental).

cl_khr_command_buffer_mutable_dispatch

Name String

`cl_khr_command_buffer_mutable_dispatch`

Ratification Status

Ratified

Extension and Version Dependencies

`cl_khr_command_buffer`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2025-08-08

IP Status

No known IP claims.

Contributors

- Ewan Crawford, Codeplay Software Ltd.
- Gordon Brown, Codeplay Software Ltd.
- Kenneth Benzie, Codeplay Software Ltd.
- Alastair Murray, Codeplay Software Ltd.
- Jack Frankland, Codeplay Software Ltd.
- Balaji Calidas, Qualcomm Technologies Inc.
- Joshua Kelly, Qualcomm Technologies, Inc.
- Kevin Petit, Arm Ltd.
- Aharon Abramson, Intel.
- Ben Ashbaugh, Intel.
- Boaz Ouriel, Intel.
- Pekka Jääskeläinen, Tampere University
- Jan Solanti, Tampere University
- Nikhil Joshi, NVIDIA
- James Price, Google

Description

The `cl_khr_command_buffer` extension separates command construction from enqueue by providing a mechanism to record a set of commands which can then be repeatedly enqueued. However, the commands recorded to the command-buffer are immutable between enqueues.

`cl_khr_command_buffer_mutable_dispatch` removes this restriction. In particular, this extension allows the configuration of a kernel execution command in a command-buffer, called a *mutable-dispatch*, to be modified. This allows inputs and outputs to the kernel, as well as work-item sizes and offsets, to change without having to re-record the entire command sequence in a new command-buffer.

Simultaneous Use

The optional simultaneous use capability was added to the extension so that vendors could support concurrent execution of the same command-buffer object which has been updated between

submissions. However, simultaneous use may result in command-buffers having a larger overhead to implement, so the capability is optional to enable optimizations when this usage isn't required by a user.

Interactions With Other Extensions

The **clUpdateMutableCommandsKHR** entry-point has been designed for the purpose of allowing expansion of mutable functionality in future extensions layered on top of `cl_khr_command_buffer_mutable_dispatch`.

A new extension can define its own structure type to specify the update configuration it requires, with a matching `cl_command_buffer_update_type_khr` value. This new structure type can then be passed to **clUpdateMutableCommandsKHR** where it is reinterpreted from a void pointer using `cl_command_buffer_update_type_khr`.

New Commands

- **clUpdateMutableCommandsKHR**
- **clGetMutableCommandInfoKHR**

New Types

- `cl_mutable_dispatch_fields_khr`
- `cl_mutable_command_info_khr`
- `cl_command_buffer_update_type_khr`
- `cl_mutable_dispatch_asserts_khr`
- `cl_mutable_dispatch_config_khr`
- `cl_mutable_dispatch_exec_info_khr`
- `cl_mutable_dispatch_arg_khr`

New Enums

- `cl_device_info`
  - `CL_DEVICE_MUTABLE_DISPATCH_CAPABILITIES_KHR`
- `cl_device_command_buffer_capabilities_khr`
  - `CL_COMMAND_BUFFER_CAPABILITY_SIMULTANEOUS_USE_KHR`
- `cl_command_properties_khr`
  - `CL_MUTABLE_DISPATCH_ASSERTS_KHR`
  - `CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR`
- `cl_mutable_dispatch_asserts_khr`
  - `CL_MUTABLE_DISPATCH_ASSERT_NO_ADDITIONAL_WORK_GROUPS_KHR`
- `cl_mutable_dispatch_fields_khr`
  - `CL_MUTABLE_DISPATCH_GLOBAL_OFFSET_KHR`

    - CL_MUTABLE_DISPATCH_GLOBAL_SIZE_KHR
    - CL_MUTABLE_DISPATCH_LOCAL_SIZE_KHR
    - CL_MUTABLE_DISPATCH_ARGUMENTS_KHR
    - CL_MUTABLE_DISPATCH_EXEC_INFO_KHR
- cl_mutable_command_info_khr
    - CL_MUTABLE_COMMAND_COMMAND_QUEUE_KHR
    - CL_MUTABLE_COMMAND_COMMAND_BUFFER_KHR
    - CL_MUTABLE_COMMAND_PROPERTIES_ARRAY_KHR
    - CL_MUTABLE_DISPATCH_KERNEL_KHR
    - CL_MUTABLE_DISPATCH_DIMENSIONS_KHR
    - CL_MUTABLE_DISPATCH_GLOBAL_WORK_OFFSET_KHR
    - CL_MUTABLE_DISPATCH_GLOBAL_WORK_SIZE_KHR
    - CL_MUTABLE_DISPATCH_LOCAL_WORK_SIZE_KHR
    - CL_MUTABLE_COMMAND_COMMAND_TYPE_KHR
- cl_command_buffer_flags_khr
    - CL_COMMAND_BUFFER_MUTABLE_KHR
    - CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR
- cl_command_buffer_properties_khr
    - CL_COMMAND_BUFFER_MUTABLE_DISPATCH_ASSERTS_KHR
- cl_command_buffer_update_type_khr
    - CL_STRUCTURE_TYPE_MUTABLE_DISPATCH_CONFIG_KHR
- cl_command_buffer_state_khr
    - CL_COMMAND_BUFFER_STATE_FINALIZED_KHR
- New Error Codes
    - CL_INVALID_MUTABLE_COMMAND_KHR

Sample Code

Sample Application Updating the Arguments to a Mutable-dispatch Between Command-buffer Submissions

```
#define CL_CHECK(ERROR)                                \
  if (ERROR) {                                         \
    std::cerr << "OpenCL error: " << ERROR << "\n"; \
    return ERROR;                                      \
  }

int main() {
  cl_platform_id platform;
  CL_CHECK(clGetPlatformIDs(1, &platform, nullptr));
```

```
  cl_device_id device;
  CL_CHECK(clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device, nullptr));

  cl_mutable_dispatch_fields_khr mutable_capabilities;
  CL_CHECK(clGetDeviceInfo(device, CL_DEVICE_MUTABLE_DISPATCH_CAPABILITIES_KHR,
                           sizeof(mutable_capabilities), &mutable_capabilities,
                           nullptr));
  if (!(mutable_capabilities & CL_MUTABLE_DISPATCH_ARGUMENTS_KHR)) {
    std::cerr
        << "Device does not support update arguments to a mutable-dispatch, "
           "skipping example.\n";
    return 0;
  }

  cl_int error;
  cl_context context =
      clCreateContext(nullptr, 1, &device, nullptr, nullptr, &error);
  CL_CHECK(error);

  const char* code = R"OpenCLC(
kernel void vector_addition(global int* tile1, global int* tile2,
                            global int* res) {
  size_t index = get_global_id(0);
  res[index] = tile1[index] + tile2[index];
}
)OpenCLC";
  const size_t length = std::strlen(code);

  cl_program program =
      clCreateProgramWithSource(context, 1, &code, &length, &error);
  CL_CHECK(error);

  CL_CHECK(clBuildProgram(program, 1, &device, nullptr, nullptr, nullptr));

  cl_kernel kernel = clCreateKernel(program, "vector_addition", &error);
  CL_CHECK(error);

  // Set the parameters of the frames
  constexpr size_t iterations = 60;
  constexpr size_t elem_size = sizeof(cl_int);
  constexpr size_t frame_width = 32;
  constexpr size_t frame_count = frame_width * frame_width;
  constexpr size_t frame_size = frame_count * elem_size;

  cl_mem input_A_buffers[2] = {nullptr, nullptr};
  cl_mem input_B_buffers[2] = {nullptr, nullptr};
  cl_mem output_buffers[2] = {nullptr, nullptr};

  // Create the buffer to swap between even and odd kernel iterations
  for (size_t i = 0; i < 2; i++) {
    input_A_buffers[i] =
```

```
        clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);

    input_B_buffers[i] =
        clCreateBuffer(context, CL_MEM_READ_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);

    output_buffers[i] =
        clCreateBuffer(context, CL_MEM_WRITE_ONLY, frame_size, nullptr, &error);
    CL_CHECK(error);
  }

  cl_command_queue command_queue =
      clCreateCommandQueue(context, device, 0, &error);
  CL_CHECK(error);

  // Create command-buffer with mutable flag so we can update it
  cl_command_buffer_properties_khr properties[3] = {
      CL_COMMAND_BUFFER_FLAGS_KHR, CL_COMMAND_BUFFER_MUTABLE_KHR, 0};
  cl_command_buffer_khr command_buffer =
      clCreateCommandBufferKHR(1, &command_queue, properties, &error);
  CL_CHECK(error);

  CL_CHECK(clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_A_buffers[0]));
  CL_CHECK(clSetKernelArg(kernel, 1, sizeof(cl_mem), &input_B_buffers[0]));
  CL_CHECK(clSetKernelArg(kernel, 2, sizeof(cl_mem), &output_buffers[0]));

  // Instruct the nd-range command to allow for mutable kernel arguments
  cl_command_properties_khr mutable_properties[] = {
      CL_MUTABLE_DISPATCH_UPDATABLE_FIELDS_KHR,
      CL_MUTABLE_DISPATCH_ARGUMENTS_KHR, 0};

  // Create command handle for mutating nd-range command
  cl_mutable_command_khr command_handle = nullptr;

  // Add the nd-range kernel command
  error = clCommandNDRangeKernelKHR(
      command_buffer, command_queue, mutable_properties, kernel, 1, nullptr,
      &frame_count, nullptr, 0, nullptr, nullptr, &command_handle);
  CL_CHECK(error);

  CL_CHECK(clFinalizeCommandBufferKHR(command_buffer));

  // Prepare for random input generation
  std::random_device random_device;
  std::mt19937 random_engine{random_device()};
  std::uniform_int_distribution<cl_int> random_distribution{
      std::numeric_limits<cl_int>::min() / 2,
      std::numeric_limits<cl_int>::max() / 2};

  // Iterate over each frame
```

```
  for (size_t i = 0; i < iterations; i++) {
    // Set the buffers for the current frame
    cl_mem input_A_buffer = input_A_buffers[i % 2];
    cl_mem input_B_buffer = input_B_buffers[i % 2];
    cl_mem output_buffer = output_buffers[i % 2];

    // Generate input A data
    std::vector<cl_int> input_a(frame_count);
    std::generate(std::begin(input_a), std::end(input_a),
                  [&]() { return random_distribution(random_engine); });

    // Write the generated data to the input A buffer
    error =
        clEnqueueWriteBuffer(command_queue, input_A_buffer, CL_FALSE, 0,
                             frame_size, input_a.data(), 0, nullptr, nullptr);
    CL_CHECK(error);

    // Generate input B data
    std::vector<cl_int> input_b(frame_count);
    std::generate(std::begin(input_b), std::end(input_b),
                  [&]() { return random_distribution(random_engine); });

    // Write the generated data to the input B buffer
    error =
        clEnqueueWriteBuffer(command_queue, input_B_buffer, CL_FALSE, 0,
                             frame_size, input_b.data(), 0, nullptr, nullptr);
    CL_CHECK(error);

    // If not executing the first frame
    if (i != 0) {
      // Configure the mutable configuration to update the kernel arguments
      cl_mutable_dispatch_arg_khr arg_0{0, sizeof(cl_mem), &input_A_buffer};
      cl_mutable_dispatch_arg_khr arg_1{1, sizeof(cl_mem), &input_B_buffer};
      cl_mutable_dispatch_arg_khr arg_2{2, sizeof(cl_mem), &output_buffer};
      cl_mutable_dispatch_arg_khr args[] = {arg_0, arg_1, arg_2};
      cl_mutable_dispatch_config_khr dispatch_config{
          command_handle,
          3 /* num_args */,
          0 /* num_svm_arg */,
          0 /* num_exec_infos */,
          0 /* work_dim - 0 means no change to dimensions */,
          args /* arg_list */,
          nullptr /* arg_svm_list - nullptr means no change*/,
          nullptr /* exec_info_list */,
          nullptr /* global_work_offset */,
          nullptr /* global_work_size */,
          nullptr /* local_work_size */};

      // Update the command buffer with the mutable configuration
      cl_uint num_configs = 1;
      cl_command_buffer_update_type_khr config_types[1] = {
```

```
        CL_STRUCTURE_TYPE_MUTABLE_DISPATCH_CONFIG_KHR
    };
    const void* configs[1] = {&dispatch_config};
    error = clUpdateMutableCommandsKHR(command_buffer, num_configs,
                                       config_types, configs);

    CL_CHECK(error);
  }

  // Enqueue the command buffer
  error = clEnqueueCommandBufferKHR(0, nullptr, command_buffer, 0, nullptr,
                                    nullptr);
  CL_CHECK(error);

  // Allocate memory for the output data
  std::vector<cl_int> output(frame_count);

  // Read the output data from the output buffer
  error = clEnqueueReadBuffer(command_queue, output_buffer, CL_TRUE, 0,
                              frame_size, output.data(), 0, nullptr, nullptr);
  CL_CHECK(error);

  // Flush and execute the read buffer
  error = clFinish(command_queue);
  CL_CHECK(error);

  // Verify the results of the frame
  for (size_t i = 0; i < frame_count; ++i) {
    const cl_int result = input_a[i] + input_b[i];
    if (output[i] != result) {
      std::cerr << "Error: Incorrect result at index " << i << " - Expected "
                << output[i] << " was " << result << std::endl;
      std::exit(1);
    }
  }
}

std::cout << "Result verified\n";

CL_CHECK(clReleaseCommandBufferKHR(command_buffer));
for (size_t i = 0; i < 2; i++) {
  CL_CHECK(clReleaseMemObject(input_A_buffers[i]));
  CL_CHECK(clReleaseMemObject(input_B_buffers[i]));
  CL_CHECK(clReleaseMemObject(output_buffers[i]));
}
CL_CHECK(clReleaseCommandQueue(command_queue));
CL_CHECK(clReleaseKernel(kernel));
CL_CHECK(clReleaseProgram(program));
CL_CHECK(clReleaseContext(context));
CL_CHECK(clReleaseDevice(device));
return 0;
```

```
}
```

Issues

1. Include simpler, more user friendly, entry-points for updating kernel arguments?

   RESOLVED: Can be implemented in the ecosystem as a layer on top, if that layer proves popular then can be introduced, possibly as another extension on top.

2. Add a command-buffer clone entry-point for deep copying a command-buffer? Arguments could then be updated and both command-buffers used. Useful for techniques like double buffering.

   RESOLVED: In the use-case we're targeting a user would only have a handle to the original command-buffer, but not the clone, which may limit the usefulness of this capability. Additionally, an implementation could be complicated by non-trivial deep copying of the underlying objects contained in the command-buffer. As a result of this new entry-point being an additive change to the specification it is omitted, and if its functionality has demand later, it may be a introduced as a stand alone extension.

Version History

- Revision 0.9.0, 2022-08-31
  - First assigned version (experimental).
- Revision 0.9.1, 2023-11-07
  - Add type `cl_mutable_dispatch_asserts_khr` and its possible values (experimental).
- Revision 0.9.2, 2024-06-19
  - Change **clUpdateMutableCommandsKHR** API to pass configs as an array rather than linked list (experimental).
- Revision 0.9.3, 2024-09-05
  - Rename `CL_MUTABLE_DISPATCH_PROPERTIES_ARRAY_KHR` to `CL_MUTABLE_COMMAND_PROPERTIES_ARRAY_KHR` (experimental).
- Revision 0.9.4, 2025-08-08
  - Move `CL_COMMAND_BUFFER_SIMULTANEOUS_USE_KHR` and `CL_COMMAND_BUFFER_CAPABILITY_SIMULTANEOUS_USE_KHR` in this extension from the base extension (experimental).
- Revision 0.9.5 2025-08-28
  - Permitting recording ND-range kernel commands without having set all of their arguments (experimental).

cl_khr_external_memory_android_hardware_buffer

Name String

`cl_khr_external_memory_android_hardware_buffer`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 3.0`
and
`cl_khr_external_memory`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2025-05-02

IP Status

No known IP claims.

Contributors

- Joshua Kelly, QUALCOMM
- Balaji Calidas, QUALCOMM
- Kevin Petit, ARM
- Jeremy Kemp, GOOGLE
- Paul Fradgley, IMAGINATION

Description

`cl_khr_external_memory_android_hardware_buffer` extends the functionality provided by `cl_khr_external_memory`. It enables an application to create an OpenCL image or buffer object from an AHardwareBuffer. When creating an image, the implementation will infer the image format and dimensions from the AHardwareBuffer. When creating a buffer from an AHardwareBuffer, the format must be AHARDWAREBUFFER_FORMAT_BLOB.

AHardwareBuffers are reference counted objects. The OpenCL implementation will increment the reference count on the AHardwareBuffer when the image or buffer is created, and decrement the reference count when the corresponding `cl_mem` object is deleted.

Applications should use **clEnqueueMapBuffer**/**clEnqueueMapImage** for host access to the imported `cl_mem` object. If a **clEnqueueMapBuffer**/**clEnqueueMapImage** command on the imported cl_mem object has completed, the corresponding **clEnqueueUnmapMemObject** command must also complete before AHardwareBuffer_lock may be called. Conversely, if an application has called AHardwareBuffer_lock on an AHardwareBuffer, it must also call AHardwareBuffer_unlock before any **clEnqueueMapBuffer**/**clEnqueueMapImage** command on the imported `cl_mem` object reaches the running, ended, or completed state.

New API Enums

New Enums

- cl_external_memory_handle_type_khr
  - CL_EXTERNAL_MEMORY_HANDLE_ANDROID_HARDWARE_BUFFER_KHR

List of minimum supported AHardwareBuffer formats and the corresponding OpenCL formats:

Table 66. Minimum list of AHardwareBuffer formats and corresponding OpenCL formats

AHardwareBuffer Format	OpenCL Image Channel Order	OpenCL Image Channel Data Type	OpenCL Memory Object Type
AHARDWAREBUFFER_FORMAT_R8G8B8A8_UNORM	CL_RGBA	CL_UNORM_INT8	Image, e.g. CL_MEM_OBJECT_IMAGE2D
AHARDWAREBUFFER_FORMAT_R16G16B16A16_FLOAT	CL_RGBA	CL_HALF_FLOAT	Image, e.g. CL_MEM_OBJECT_IMAGE2D
AHARDWAREBUFFER_FORMAT_R8_UNORM	CL_R	CL_UNORM_INT8	Image, e.g. CL_MEM_OBJECT_IMAGE2D
AHARDWAREBUFFER_FORMAT_BLOB	N/A	N/A	CL_MEM_OBJECT_BUFFER

Sample Code

1. Using the extension for CL buffer objects

```
// Create AHB
const AHardwareBuffer_Desc desc = {, AHARDWAREBUFFER_FORMAT_BLOB, .....};
AHardwareBuffer *ahb_handle = NULL;
AHardwareBuffer_allocate(&desc,  &ahb_handle);
cl_mem_properties props[] = {
    (cl_mem_properties) CL_EXTERNAL_MEMORY_HANDLE_ANDROID_HARDWARE_BUFFER_KHR,
    (cl_mem_properties) ahb_handle,
    0};
cl_mem imported_buffer = clCreateBufferWithProperties(context, props,
CL_MEM_READ_WRITE,
                            0 /* size */, NULL, &error);
If (error == CL_SUCCESS)
{
    clEnqueueAcquireExternalMemObjectsKHR(queue, 1, &imported_buffer, ...);
    clEnqueueCopyBuffer(queue, imported_buffer, ...);
    clEnqueueReleaseExternalMemObjectsKHR(queue, 1, &imported_buffer, ...);
}
clReleaseMemObject(imported_buffer);
```

2. Using the extension for CL image objects

```
const AHardwareBuffer_Desc desc = {, AHARDWAREBUFFER_FORMAT_R8G8B8A8_UNORM, .....};
AHardwareBuffer *ahb_handle = NULL;
AHardwareBuffer_allocate(&desc,  &ahb_handle);
cl_mem_properties props[] = {
    (cl_mem_properties) CL_EXTERNAL_MEMORY_HANDLE_ANDROID_HARDWARE_BUFFER_KHR,
    (cl_mem_properties) ahb_handle,
    0};
cl_mem imported_image = clCreateImageWithProperties(context, props,
CL_MEM_READ_WRITE,
                              NULL, NULL, NULL, &error);
if (error == CL_SUCCESS)
{
     clGetImageInfo(imported_image, ...);
}
```

Version History

- Revision 0.9.0, 2025-03-13
  - First assigned version (provisional).
- Revision 0.9.2 2025-05-02
  - Reduced required formats, first published version (provisional).

cl_khr_external_semaphore_dx_fence

Name String

`cl_khr_external_semaphore_dx_fence`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`
and
`cl_khr_semaphore`
and
`cl_khr_external_semaphore`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2021-09-10

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_semaphore_dx_fence` supports importing and exporting a D3D12 fence as an external semaphore using the APIs introduced by `cl_khr_external_semaphore`.

New Enums

- `cl_external_semaphore_handle_type_khr`
  - `CL_SEMAPHORE_HANDLE_D3D12_FENCE_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).

cl_khr_external_semaphore_win32

Name String

`cl_khr_external_semaphore_win32`

Ratification Status

Ratified

Extension and Version Dependencies

`OpenCL 1.2`
and
`cl_khr_semaphore`
and
`cl_khr_external_semaphore`

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2024-06-11

IP Status

No known IP claims.

Contributors

- Ajit Hakke-Patil, NVIDIA
- Amit Rao, NVIDIA
- Balaji Calidas, QUALCOMM
- Ben Ashbaugh, INTEL
- Carsten Rohde, NVIDIA
- Christoph Kubisch, NVIDIA
- Debalina Bhattacharjee, NVIDIA
- Faith Ekstrand, INTEL
- James Jones, NVIDIA
- Jeremy Kemp, IMAGINATION
- Joshua Kelly, QUALCOMM
- Karthik Raghavan Ravi, NVIDIA
- Kedar Patil, NVIDIA
- Kevin Petit, ARM
- Nikhil Joshi, NVIDIA
- Sharan Ashwathnarayan, NVIDIA
- Vivek Kini, NVIDIA

Description

`cl_khr_external_semaphore_win32` supports importing and exporting an NT handle or global share handle as an external semaphore using the APIs introduced by `cl_khr_external_semaphore`.

New Enums

- `cl_external_semaphore_handle_type_khr`
  - `CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KHR`
  - `CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_KMT_KHR`
  - `CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_NAME_KHR`

Version History

- Revision 0.9.0, 2021-09-10
  - Initial version (experimental).
- Revision 0.9.1, 2024-06-11
  - Added `CL_SEMAPHORE_HANDLE_OPAQUE_WIN32_NAME_KHR`.

cl_khr_unified_svm

Name String

`cl_khr_unified_svm`

Ratification Status

Not ratified

Extension and Version Dependencies

None

- **This is a *experimental* extension and must be used with caution. See the description of experimental header files for enablement and stability details.**

Other Extension Metadata

Last Modified Date

2025-08-30

IP Status

No known IP claims.

Contributors

- Brice Videau, Argonne National Laboratory
- Kévin Petit, Arm Ltd.
- Ewan Crawford, Codeplay Software Ltd.
- Paul Fradgley, Imagination Technologies
- Ben Ashbaugh, Intel
- Pekka Jääskeläinen, Intel
- Nikhil Joshi, NVIDIA

- Balaji Calidas, Qualcomm Technologies Inc.

Description

This extension adds additional types of Shared Virtual Memory (SVM) to OpenCL.

The extension is still under development. To learn more about the APIs proposed for this extension, or to provide feedback, please visit:

https://github.com/KhronosGroup/OpenCL-Docs/pull/1282

New Commands

- **clSVMAllocWithPropertiesKHR**
- **clSVMFreeWithPropertiesKHR**
- **clGetSVMPointerInfoKHR**
- **clGetSVMSuggestedTypeIndexKHR**

New Types

- `cl_svm_capabilities_khr`
- `cl_svm_alloc_properties_khr`
- `cl_svm_alloc_access_flags_khr`
- `cl_svm_free_properties_khr`
- `cl_svm_free_flags_khr`
- `cl_svm_pointer_info_khr`

New Enums

- `cl_platform_info`
  - `CL_PLATFORM_SVM_TYPE_CAPABILITIES_KHR`
- `cl_device_info`
  - `CL_DEVICE_SVM_TYPE_CAPABILITIES_KHR`
- `cl_svm_capabilities_khr`
  - `CL_SVM_CAPABILITY_SINGLE_ADDRESS_SPACE_KHR`
  - `CL_SVM_CAPABILITY_SYSTEM_ALLOCATED_KHR`
  - `CL_SVM_CAPABILITY_DEVICE_OWNED_KHR`
  - `CL_SVM_CAPABILITY_DEVICE_UNASSOCIATED_KHR`
  - `CL_SVM_CAPABILITY_CONTEXT_ACCESS_KHR`
  - `CL_SVM_CAPABILITY_HOST_OWNED_KHR`
  - `CL_SVM_CAPABILITY_HOST_READ_KHR`
  - `CL_SVM_CAPABILITY_HOST_WRITE_KHR`

    - CL_SVM_CAPABILITY_HOST_MAP_KHR
    - CL_SVM_CAPABILITY_DEVICE_READ_KHR
    - CL_SVM_CAPABILITY_DEVICE_WRITE_KHR
    - CL_SVM_CAPABILITY_DEVICE_ATOMIC_ACCESS_KHR
    - CL_SVM_CAPABILITY_CONCURRENT_ACCESS_KHR
    - CL_SVM_CAPABILITY_CONCURRENT_ATOMIC_ACCESS_KHR
    - CL_SVM_CAPABILITY_INDIRECT_ACCESS_KHR
- cl_svm_alloc_properties_khr
    - CL_SVM_ALLOC_ASSOCIATED_DEVICE_HANDLE_KHR
    - CL_SVM_ALLOC_ACCESS_FLAGS_KHR
    - CL_SVM_ALLOC_ALIGNMENT_KHR
- cl_svm_alloc_access_flags_khr
    - CL_SVM_ALLOC_ACCESS_HOST_NOREAD_KHR
    - CL_SVM_ALLOC_ACCESS_HOST_NOWRITE_KHR
    - CL_SVM_ALLOC_ACCESS_DEVICE_NOREAD_KHR
    - CL_SVM_ALLOC_ACCESS_DEVICE_NOWRITE_KHR
- cl_svm_pointer_info_khr
    - CL_SVM_INFO_TYPE_INDEX_KHR
    - CL_SVM_INFO_CAPABILITIES_KHR
    - CL_SVM_INFO_PROPERTIES_KHR
    - CL_SVM_INFO_ACCESS_FLAGS_KHR
    - CL_SVM_INFO_BASE_PTR_KHR
    - CL_SVM_INFO_SIZE_KHR
    - CL_SVM_INFO_ASSOCIATED_DEVICE_HANDLE_KHR
- cl_kernel_exec_info
    - CL_KERNEL_EXEC_INFO_SVM_INDIRECT_ACCESS_KHR

SVM Type Convenience Macros

- CL_SVM_TYPE_MACRO_COARSE_GRAIN_BUFFER_KHR
- CL_SVM_TYPE_MACRO_FINE_GRAIN_BUFFER_KHR
- CL_SVM_TYPE_MACRO_DEVICE_KHR
- CL_SVM_TYPE_MACRO_HOST_KHR
- CL_SVM_TYPE_MACRO_SINGLE_DEVICE_SHARED_KHR
- CL_SVM_TYPE_MACRO_SYSTEM_KHR

Version History

- Revision 0.9.0, 2025-08-30
  - Initial revision incorporated into the main specification (experimental).

List of Deprecated Extensions

- `cl_khr_spir`

cl_khr_spir

Name String

`cl_khr_spir`

Ratification Status

Ratified

Extension and Version Dependencies

None

Deprecation State

- *Obsoleted* by `cl_khr_il_program` extension
  - Which in turn was *promoted* to OpenCL 2.1

Other Extension Metadata

Last Modified Date

2020-04-21

IP Status

No known IP claims.

Description

`cl_khr_spir` adds the ability to create an OpenCL program object from a Standard Portable Intermediate Representation (SPIR) instance. A SPIR instance is a vendor-neutral non-source representation for OpenCL C programs.

See the SPIR Compilation Options for information on compiling SPIR binaries.

`cl_khr_spir` has been superseded by the SPIR-V intermediate representation, which is supported by the `cl_khr_il_program` extension, and is a core feature in OpenCL 2.1.

New Enums

- `cl_device_info`
  - `CL_DEVICE_SPIR_VERSIONS`
- `cl_program_binary_type`
  - `CL_PROGRAM_BINARY_TYPE_INTERMEDIATE`

Version History

- Revision 1.0.0, 2020-04-21
  - First assigned version.

Acknowledgements

The OpenCL specification is the result of the contributions of many people, representing a cross section of the desktop, hand-held, and embedded computer industry. Following is a partial list of the contributors, including the company that they represented at the time of their contribution:

Chuck Rose, Adobe
Eric Berdahl, Adobe
Shivani Gupta, Adobe
Bill Licea Kane, AMD
Ed Buckingham, AMD
Jan Civlin, AMD
Laurent Morichetti, AMD
Mark Fowler, AMD
Marty Johnson, AMD
Michael Mantor, AMD
Norm Rubin, AMD
Ofer Rosenberg, AMD
Brian Sumner, AMD
Victor Odintsov, AMD
Aaftab Munshi, Apple
Abe Stephens, Apple
Alexandre Namaan, Apple
Anna Tikhonova, Apple
Chendi Zhang, Apple
Eric Bainville, Apple
David Hayward, Apple
Giridhar Murthy, Apple
Ian Ollmann, Apple
Inam Rahman, Apple
James Shearer, Apple
MonPing Wang, Apple
Tanya Lattner, Apple
Mikael Bourges-Sevenier, Aptina
Brice Videau, Argonne National Laboratory
Thomas Applencourt, Argonne National Laboratory
Nevin Liber, Argonne National Laboratory
Anton Lokhmotov, ARM
Dave Shreiner, ARM
Einar Hov, ARM
Hedley Francis, ARM
Kevin Petit, ARM
Neil Hickey, ARM
Robert Elliott, ARM
Scott Moyers, ARM
Stuart Brady, ARM
Sven Van Haastregt, Arm

Tom Olson, ARM
Anastasia Stulova, ARM
Christopher Thompson-Walsh, Broadcom
Holger Waechtler, Broadcom
Norman Rink, Broadcom
Andrew Richards, Codeplay
Maria Rovatsou, Codeplay
Alistair Donaldson, Codeplay
Alastair Murray, Codeplay
Ewan Crawford, Codeplay
Ewa Gałamon, Cognizant
Marcin Hajder, Cognizant
Faith Ekstrand, Collabora
Stephen Frye, Electronic Arts
Eric Schenk, Electronic Arts
Daniel Laroche, Freescale
David Neto, Google
James Price, Google
Romaric Jodin, Google
Robin Grosman, Huawei
Craig Davies, Huawei
Brian Horton, IBM
Brian Watt, IBM
Gordon Fossum, IBM
Greg Bellows, IBM
Joaquin Madruga, IBM
Mark Nutter, IBM
Mike Perks, IBM
Sean Wagner, IBM
Ahmed Amrani Akdi, Imagination Technologies
Jeremy Kemp, Imagination Technologies
Jon Parr, Imagination Technologies
Paul Fradgley, Imagination Technologies
Robert Quill, Imagination Technologies
James McCarthy, Imagination Technologies
Jon Leech, Independent
Aaron Kunze, Intel
Aaron Lefohn, Intel
Adam Lake, Intel
Alexey Bader, Intel
Allen Hux, Intel
Andrew Brownsword, Intel
Andrew Lauritzen, Intel
Anton Zabaznov, Intel
Bartosz Sochacki, Intel
Ben Ashbaugh, Intel
Boaz Ouriel, Intel
Brian Lewis, Intel

Filip Hazubski, Intel
Geoff Berry, Intel
Grzegorz Wawiorko, Intel
Hong Jiang, Intel
Jayanth Rao, Intel
Josh Fryman, Intel
Kevin Stevens, Intel
Konrad Trifunovic, Intel
Larry Seiler, Intel
Michael Kinsner, Intel
Michal Mrozek, Intel
Mike MacPherson, Intel
Murali Sundaresan, Intel
Paul Lalonde, Intel
Stephen Junkins, Intel
Tim Foley, Intel
Timothy Mattson, Intel
Yariv Aridor, Intel
Sébastien Le Duc, Kalray
Benjamin Bergen, Los Alamos National Laboratory
Roy Ju, Mediatek
Bor-Sung Liang, Mediatek
Rahul Agarwal, Mediatek
Michal Witaszek, Mobica
Aharon Abramson, Mobileye
JenqKuen Lee, NTHU
Amit Rao, NVIDIA
Ashish Srivastava, NVIDIA
Bastiaan Aarts, NVIDIA
Chris Cameron, NVIDIA
Christopher Lamb, NVIDIA
Dibyapran Sanyal, NVIDIA
Guatam Chakrabarti, NVIDIA
Ian Buck, NVIDIA
Jaydeep Marathe, NVIDIA
Jian-Zhong Wang, NVIDIA
Karthik Raghavan Ravi, NVIDIA
Kedar Patil, NVIDIA
Manjunath Kudlur, NVIDIA
Mark Harris, NVIDIA
Michael Gold, NVIDIA
Neil Trevett, NVIDIA
Nikhil Joshi, NVIDIA
Richard Johnson, NVIDIA
Sean Lee, NVIDIA
Tushar Kashalikar, NVIDIA
Vinod Grover, NVIDIA
Xiangyun Kong, NVIDIA

Yogesh Kini, NVIDIA
Yuan Lin, NVIDIA
Mayuresh Pise, NVIDIA
Allan Tzeng, QUALCOMM
Alex Bourd, QUALCOMM
Alexander Galazin, QUALCOMM
Andrew Gruber, QUALCOMM
Andrzej Mamona, QUALCOMM
Anirudh Acharya, QUALCOMM
Arvind Sudarsanam, QUALCOMM
Balaji Calidas, QUALCOMM
Benedict Gaster, QUALCOMM
Bill Torzewski, QUALCOMM
Bob Rychlik, QUALCOMM
Chihong Zhang, QUALCOMM
Chris Mei, QUALCOMM
Colin Sharp, QUALCOMM
David Garcia, QUALCOMM
David Ligon, QUALCOMM
Hongqiang Wang, QUALCOMM
Jay Yun, QUALCOMM
Jian Liu, QUALCOMM
Jose Lopez, QUALCOMM
Joshua Kelly, QUALCOMM
Lee Howes, QUALCOMM
Lihan Bin, QUALCOMM
Richard Ruigrok, QUALCOMM
Robert J. Simpson, QUALCOMM
Ruihao Zhang, QUALCOMM
Samuel Pauls, QUALCOMM
Sreelakshmi Haridas, QUALCOMM
Sumesh Udayakumaran, QUALCOMM
Vineet Goel, QUALCOMM
Vlad Shimanskiy, QUALCOMM
Yu-Chi Huang, QUALCOMM
Yuehai Du, QUALCOMM
Karol Herbst, Red Hat
Austin Annestrand, Samsung
Raun Krisch, Samsung
Gowtham Tammana, Samsung
Pavan Lanka, Samsung
Tasneem Brutch, Samsung
Yoonseo Choi, Samsung
Dennis Adams, Sony
Pr-Anders Aronsson, Sony
Jim Rasmusson, Sony
Thierry Lepley, STMicroelectronics
Anton Gorenko, StreamHPC

Jakub Szuppe, StreamHPC
Máté Ferenc Nagy-Egri, StreamHPC
Vincent Hindriksen, StreamHPC
Pekka Jääskeläinen, Tampere University
Ajay Jayaraj, Texas Instruments
Alan Ward, Texas Instruments
Yuan Zhao, Texas Instruments
Pete Curry, Texas Instruments
Simon McIntosh-Smith, University of Bristol
Paul Preney, University of Windsor
Shane Peelar, University of Windsor
Wei-Lun Kao, VeriSilicon
Yanjun Zhang, VeriSilicon
Brian Hutsell, Vivante
Mike Cai, Vivante
Sumeet Kumar, Vivante
Xing Wang, Vivante
Jeff Fifield, Xilinx
Hem C. Neema, Xilinx
Henry Styles, Xilinx
Ralph Wittig, Xilinx
Ronan Keryell, Xilinx
AJ Guillon, YetiWare Inc