The OpenCL™ C 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

6. The OpenCL C Programming Language . . . 2
6.1. Unified Specification . . . 2
6.2. Optional functionality . . . 3
6.2.1. Features . . . 3
6.2.2. Extensions . . . 5
6.3. Supported Data Types . . . 11
6.3.1. Built-in Scalar Data Types . . . 11
6.3.2. Built-in Vector Data Types . . . 14
6.3.3. Other Built-in Data Types . . . 15
6.3.4. Reserved Data Types . . . 18
6.3.5. Alignment of Types . . . 19
6.3.6. Vector Literals . . . 19
6.3.7. Vector Components . . . 20
6.3.8. Aliasing Rules . . . 24
6.3.9. Keywords . . . 24
6.4. Conversions and Type Casting . . . 25
6.4.1. Implicit Conversions . . . 25
6.4.2. Explicit Casts . . . 25
6.4.3. Explicit Conversions . . . 26
6.4.4. Reinterpreting Data as Another Type . . . 29
6.4.5. Pointer Casting . . . 31
6.4.6. Usual Arithmetic Conversions . . . 31
6.5. Operators . . . 32
6.5.1. Arithmetic Operators . . . 32
6.5.2. Unary Operators . . . 32
6.5.3. Pre- and Post-Operators . . . 33
6.5.4. Relational Operators . . . 33
6.5.5. Equality Operators . . . 33
6.5.6. Bitwise Operators . . . 34
6.5.7. Logical Operators . . . 34
6.5.8. Unary Logical Operator . . . 35
6.5.9. Ternary Selection Operator . . . 35
6.5.10. Shift Operators . . . 35
6.5.11. Sizeof Operator . . . 36
6.5.12. Comma Operator . . . 36
6.5.13. Indirection Operator . . . 36
6.5.14. Address Operator . . . 36
6.5.15. Assignment Operator . . . 37

6.6. Vector Operations . . . 37
6.7. Address Space Qualifiers . . . 38
6.7.1. `__global` (or `global`) . . . 40
6.7.2. `__local` (or `local`) . . . 40
6.7.3. `__constant` (or `constant`) . . . 41
6.7.4. `__private` (or `private`) . . . 41
6.7.5. The Generic Address Space . . . 42
6.7.6. Usage for Declaration Scopes and Variable Types . . . 42
6.7.7. Initialization . . . 43
6.7.8. Inference . . . 44
6.7.9. Address Space Conversions . . . 47
6.8. Access Qualifiers . . . 55
6.9. Function Qualifiers . . . 55
6.9.1. `__kernel` (or `kernel`) . . . 55
6.9.2. Optional Attribute Qualifiers . . . 56
6.10. Storage-Class Specifiers . . . 57
6.11. Restrictions . . . 58
6.12. Preprocessor Directives and Macros . . . 60
6.12.1. Preprocessor Directives for Optional Extensions . . . 63
6.13. Attribute Qualifiers . . . 64
6.13.1. Specifying Attributes of Types . . . 65
6.13.2. Specifying Attributes of Functions . . . 66
6.13.3. Specifying Attributes of Variables . . . 66
6.13.4. Specifying Attributes of Blocks and Control-Flow-Statements . . . 69
6.13.5. Specifying Attribute for Unrolling Loops . . . 69
6.14. Blocks . . . 70
6.14.1. Declaring and Using a Block . . . 71
6.14.2. Declaring a Block Reference . . . 71
6.14.3. Block Literal Expressions . . . 72
6.14.4. Control Flow . . . 73
6.14.5. Restrictions . . . 73
6.15. Built-in Functions . . . 76
6.15.1. Work-Item Functions . . . 76
6.15.2. Math Functions . . . 81
6.15.3. Integer Functions . . . 97
6.15.4. Common Functions . . . 103
6.15.5. Geometric Functions . . . 106
6.15.6. Relational Functions . . . 108
6.15.7. Vector Data Load and Store Functions . . . 112
6.15.8. Synchronization Functions . . . 119
6.15.9. Legacy Explicit Memory Fence Functions . . . 122

6.15.10. Address Space Qualifier Functions . . . 123
6.15.11. Async Copies From Global to Local Memory, Local to Global Memory, and Prefetch . . . 123
6.15.12. Atomic Functions . . . 129
6.15.13. Miscellaneous Vector Functions . . . 156
6.15.14. printf . . . 158
6.15.15. Image Read and Write Functions . . . 164
6.15.16. Work-group Collective Functions . . . 232
6.15.17. Work-group Collective Uniform Arithmetic Functions . . . 233
6.15.18. Pipe Functions . . . 236
6.15.19. Enqueuing Kernels . . . 240
6.15.20. Sub-Group Functions . . . 253
6.15.21. Kernel Clock Functions . . . 274
7. OpenCL Numerical Compliance . . . 280
7.1. Rounding Modes . . . 280
7.2. INF, NaN and Denormalized Numbers . . . 281
7.3. Floating-Point Exceptions . . . 282
7.4. Relative Error as ULPs . . . 282
7.5. Edge Case Behavior . . . 299
7.5.1. Additional Requirements Beyond C99 TC2 . . . 299
7.5.2. Changes to C99 TC2 Behavior . . . 302
7.5.3. Edge Case Behavior in Flush to Zero Mode . . . 302
8. Image Addressing and Filtering . . . 304
8.1. Image Coordinates . . . 304
8.2. Addressing and Filter Modes . . . 304
8.3. Conversion Rules . . . 310
8.3.1. Conversion Rules for Normalized Integer Channel Data Types . . . 310
8.3.2. Conversion Rules for Half-Precision Floating-Point Channel Data Type . . . 318
8.3.3. Conversion Rules for Floating-Point Channel Data Type . . . 319
8.3.4. Conversion Rules for Signed and Unsigned Integer Channel Data Types . . . 319
8.3.5. Conversion Rules for sRGBA and sBGRA Images . . . 320
8.4. Selecting an Image From an Image Array . . . 321
9. Normative References . . . 323
Appendix A: Changes to OpenCL . . . 324
Summary of Changes to OpenCL 3.0 . . . 324
Summary of Changes from OpenCL 3.0 to OpenCL 3.1 . . . 325

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 6. The OpenCL C Programming Language

This document starts at chapter 6 to keep the section numbers historically consistent with previous versions of the OpenCL and OpenCL C Programming Language specifications.

This section describes the OpenCL C programming language. The OpenCL C programming language may be used to write kernels that execute on an OpenCL device.

The OpenCL C programming language (also referred to as OpenCL C) is based on the ISO/IEC 9899:1999 Programming languages - C specification (also referred to as the C99 specification, or just C99), with extensions and restrictions to support parallel kernels. In addition, some features of OpenCL C are based on the ISO/IEC 9899:2011 Information technology - Programming languages - C specification (also referred to as the C11 specification, or just C11).

This document describes the modifications and restrictions to C99 and C11 in OpenCL C. Please refer to the C99 specification for a detailed description of the language grammar.

6.1. Unified Specification

This document specifies all versions of OpenCL C.

There are several ways that an OpenCL C feature may be described in terms of what versions of OpenCL C specify that feature.

- Requires support for OpenCL C *major.minor* or newer: Features that were introduced in version *major.minor*. Compilers for an earlier version of OpenCL C will not provide these features.
  - In some instances the variation of "For OpenCL C *major.minor* or newer" is used, it has the identical meaning.
- Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_<feature_name>` feature: Features that were introduced in OpenCL C 2.0 as mandatory, but are optional in OpenCL C 3.0 or newer. Compilers for versions of OpenCL C 1.2 or below will not provide these features, compilers for OpenCL C 2.0 will provide these features, compilers for OpenCL C 3.0 or newer may provide these features.
- Requires support for OpenCL C 3.0 or newer and the `__opencl_c_<feature_name>` feature: Optional features that were introduced in OpenCL C 3.0. Compilers for an earlier version of OpenCL C will not provide these features, compilers for OpenCL C 3.0 or newer may provide these features.
- Deprecated by OpenCL C *major.minor*: Features that were deprecated in version *major.minor*, see the definition of deprecation in the glossary of the main OpenCL specification.
- Universal: Features that have no mention of what version they are missing before or deprecated by are specified for all versions of OpenCL C.

6.2. Optional functionality

Some language functionality is optional and will not be supported by all devices. Such functionality is represented by optional language features or language extensions. Support of optional functionality in OpenCL C is indicated by the presence of special predefined macros.

6.2.1. Features

Feature test macros require support for OpenCL C 3.0 or newer.

Optional core language features are described in this document. They are optional from OpenCL C 3.0 onwards and therefore are not supported by all implementations. When an optional OpenCL C 3.0 or newer feature is supported, an associated *feature test macro* will be predefined.

The following table describes OpenCL C 3.0 or newer features and their meaning. The naming convention for the feature macros is `__opencl_c_<feature_name>`.

Feature macro identifiers are used as names of features in this document.

Table 1. Optional features in OpenCL C 3.0 or newer and their predefined macros.

Feature Macro/Name	Brief Description
`__opencl_c_3d_image_writes`	The OpenCL C compiler supports built-in functions for writing to 3D image objects. OpenCL C compilers that define the feature macro `__opencl_c_3d_image_writes` must also define the feature macro `__opencl_c_images`.
`__opencl_c_atomic_order_acq_rel`	The OpenCL C compiler supports enumerations and built-in functions for atomic operations with acquire and release memory consistency orders.
`__opencl_c_atomic_order_seq_cst`	The OpenCL C compiler supports enumerations and built-in functions for atomic operations and fences with sequentially consistent memory consistency order.
`__opencl_c_atomic_scope_device`	The OpenCL C compiler supports enumerations and built-in functions for atomic operations and fences with device memory scope.
`__opencl_c_atomic_scope_all_devices`	The OpenCL C compiler supports enumerations and built-in functions for atomic operations and fences with all with memory scope across all devices that can share SVM memory with each other and the host process.

Feature Macro/Name	Brief Description
`__opencl_c_device_enqueue`	The OpenCL C compiler supports built-in functions to enqueue additional work from the device. OpenCL C compilers that define the feature macro `__opencl_c_device_enqueue` must also define `__opencl_c_generic_address_space` and `__opencl_c_program_scope_global_variables` feature macros.
`__opencl_c_generic_address_space`	The OpenCL C compiler supports the unnamed generic address space.
`__opencl_c_fp64`	The OpenCL C compiler supports types and built-in functions with 64-bit floating-point types.
`__opencl_c_images`	The OpenCL C compiler supports types and built-in functions for images.
`__opencl_c_int64`	The OpenCL C compiler supports types and built-in functions with 64-bit integers. OpenCL C compilers for FULL profile devices or devices with 64-bit pointers must always define the `__opencl_c_int64` feature macro.
`__opencl_c_pipes`	The OpenCL C compiler supports the pipe specifier and built-in functions to read and write from a pipe. OpenCL C compilers that define the feature macro `__opencl_c_pipes` must also define the feature macro `__opencl_c_generic_address_space`.
`__opencl_c_program_scope_global_variables`	The OpenCL C compiler supports program scope variables in the global address space.
`__opencl_c_read_write_images`	The OpenCL C compiler supports reading from and writing to the same image object in a kernel. OpenCL C compilers that define the feature macro `__opencl_c_read_write_images` must also define the feature macro `__opencl_c_images`.
`__opencl_c_subgroups`	The OpenCL C compiler supports built-in functions operating on sub-groupings of work-items.
`__opencl_c_work_group_collective_functions`	The OpenCL C compiler supports built-in functions that perform collective operations across a work-group.

Feature Macro/Name	Brief Description
`__opencl_c_integer_dot_product_input_4x8bit_packed`	The OpenCL C compiler supports built-in functions that perform dot products on 4x8 bit packed integer vectors.
`__opencl_c_integer_dot_product_input_4x8bit`	The OpenCL C compiler supports built-in functions that perform dot products on 4x8 bit integer vectors.
`__opencl_c_kernel_clock_scope_device`	The OpenCL C compiler supports built-in functions that sample the value from a clock shared by all work-items executing on the device.
`__opencl_c_kernel_clock_scope_work_group`	The OpenCL C compiler supports built-in functions that sample the value from a clock shared by all work-items executing in the same work-group.
`__opencl_c_kernel_clock_scope_sub_group`	The OpenCL C compiler supports built-in functions that sample the value from a clock shared by all work-items executing in the same sub-group.
`__opencl_c_ext_image_unorm_int_2_101010`	The OpenCL C compiler supports `CLK_UNORM_INT_2_101010_EXT` and returning it from `get_image_channel_data_type`.
`__opencl_c_ext_image_unsigned_10x6_12x4_14x2`	The OpenCL C compiler supports returning `CLK_UNSIGNED_INT10X6_EXT`, `CLK_UNSIGNED_INT12X4_EXT`, `CLK_UNSIGNED_INT14X2_EXT`, `CLK_UNORM_10X6_EXT`, `CLK_UNORM_12X4_EXT`, or `CLK_UNORM_14X2_EXT` from `get_image_channel_data_type`.

In OpenCL C 3.0 or newer, feature macros must expand to the value `1` if the feature macro is defined by the OpenCL C compiler. A feature macro must not be defined if the feature is not supported by the OpenCL C compiler. A feature macro may expand to a different value in the future, but if this occurs the value of the feature macro must compare greater than the prior value of the feature macro.

As specified in section 7.1.3 of the C99 Specification double underscore identifiers are reserved and therefore implementations for earlier OpenCL C versions are allowed to define feature test macros but they are not required to do so. This means that applications which target earlier OpenCL C versions should not rely on the presence of feature test macros because there is no guarantee that feature test macros will be defined and that if defined they will indicate the presence of the corresponding optional functionality.

6.2.2. Extensions

Other optional functionality may be described by language extensions to OpenCL C. Extensions to

OpenCL C are described in the OpenCL C specification or in vendor-provided documentation. When an OpenCL C extension is supported an associated extension macro will be predefined. Please refer to Preprocessor Directives for Optional Extensions for more information about predefined extension macros.

When an optional core language feature began as an extension it may have both an associated feature macro and an associated extension macro. If an optional core language feature was an optional extension to an earlier version of OpenCL C it can still be used as an extension, i.e. the same predefined extension macros are still valid in OpenCL C 3.0 or newer, however the use of feature macros is preferred whenever possible.

6.2.2.1. 3D Image Writes

The `cl_khr_3d_image_writes` extension was promoted to OpenCL C 2.0, and to OpenCL C 3.0 and newer as the `__opencl_c_3d_image_writes` feature. The extension adds Built-in Image Write Functions that allow a kernel to write to 3D image objects in addition to 2D image objects.

6.2.2.2. Async Work-group Copy Fence

The `cl_khr_async_work_group_copy_fence` extension supports establishing a memory synchronization ordering of asynchronous copies. The extension provides the `async_work_group_copy_fence` function, as described in the Built-in Async Copy and Prefetch Functions table

6.2.2.3. Byte-Addressable Storage

The `cl_khr_byte_addressable_store` extension was promoted to OpenCL C 1.1. The extension relaxes Restrictions on pointers to `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort` and `half`, allowing applications to read from and write to pointers to these types.

6.2.2.4. Depth Images

The `cl_khr_depth_images` extension was promoted to OpenCL 2.0. The extension provides new built-in depth image types, as well as read functions, sampler-less read functions, write functions, and image queries operating on those types.

6.2.2.5. Device Enqueue Local Argument Types

The `cl_khr_device_enqueue_local_arg_types` extension allows arguments to blocks that are passed to the Built-in Kernel Enqueue Functions and to the Built-in Kernel Query Functions 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.

6.2.2.6. Extended Async Copy Functions

The `cl_khr_extended_async_copies` extension provides additional Extended Async Copy Functions which interpret the source and destination as 2D or 3D images.

6.2.2.7. Extended Bit Operations

The `cl_khr_extended_bit_ops` extension was promoted to OpenCL C 3.1. The extension provides

additional Extended Bit Operations including bitfield insert, bitfield extract, and bit reverse.

6.2.2.8. Half-Precision Floating-Point

The `cl_khr_fp16` extension provides 16-bit half-precision scalar and vector floating-point data types and extends many functions to accept these types.

6.2.2.9. Double-Precision Floating-Point

The `cl_khr_fp64` extension was promoted to OpenCL C 1.2 as an optional feature. It provides double-precision scalar and vector floating-point data types and extends many functions to accept these types.

6.2.2.10. Multi-Sample Shared OpenCL/OpenGL Images

The `cl_khr_gl_msaa_sharing` extension adds support for multi-sample images shared with OpenGL multi-sample textures. The extension provides new built-in multisample image types, as well as sampler-less read functions and image queries operating on those types.

6.2.2.11. Global 32-Bit Base Atomics

The `cl_khr_global_int32_base_atomics` extension was promoted to OpenCL C 1.1, with the supported functions renamed to use the **atomic**_ prefix rather than the **atom**_ prefix. The extension provides base atomic functions for __global variables, as described in the Atomic Function Extensions table.

6.2.2.12. Global 32-Bit Extended Atomics

The `cl_khr_global_int32_extended_atomics` extension was promoted to OpenCL C 1.1, with the supported functions renamed to use the **atomic**_ prefix rather than the **atom**_ prefix. The extension provides extended atomic functions for __global variables, as described in the Atomic Function Extensions table.

6.2.2.13. Initializing Memory

The `cl_khr_initialize_memory` extension allows creating a context which initializes specified types (local or private) of memory prior to the start of kernel execution.

There is one restriction on the timing of this initialization discussed in this document, although most of the extension is defined by the OpenCL API Specification.

6.2.2.14. 64-Bit Base Atomics

The `cl_khr_int64_base_atomics` extension provides base atomic functions for __global and __local 64-bit signed and unsigned integer variables, as described in the Built-in 64-Bit Base Atomic Functions table.

6.2.2.15. 64-Bit Extended Atomics

The `cl_khr_int64_extended_atomics` extension provides extended atomic functions for __global and __local 64-bit signed and unsigned integer variables, as described in the Built-in 64-Bit Extended Atomic Functions table.

6.2.2.16. Integer Dot Product

The `cl_khr_integer_dot_product` extension was promoted to OpenCL C 3.1. The extension adds support for OpenCL C built-in functions to compute the dot product of vectors of integers, as described in the built-in vector integer argument functions table.

6.2.2.17. Kernel Clock

The `cl_khr_kernel_clock` extension adds OpenCL C built-in functions to sample the value from one of three clocks provided by compute units. The extension provides the following functions:

- Built-in Kernel Clock Functions

6.2.2.18. Local 32-Bit Base Atomics

The `cl_khr_local_int32_base_atomics` extension was promoted to OpenCL C 1.1, with the supported functions renamed to use the **atomic**_ prefix rather than the **atom**_ prefix. The extension provides base atomic functions for __local variables, as described in the Atomic Function Extensions table.

6.2.2.19. Local 32-Bit Extended Atomics

The `cl_khr_local_int32_extended_atomics` extension was promoted to OpenCL C 1.1, with the supported functions renamed to use the **atomic**_ prefix rather than the **atom**_ prefix. The extension provides extended atomic functions for __local variables, as described in the Atomic Function Extensions table.

6.2.2.20. Mipmapped Image Reads and Queries

The `cl_khr_mipmap_image` extension adds support for mipmap images. The extension provides built-in image read and image query functions operating on these images.

6.2.2.21. Mipmapped Image Writes

The `cl_khr_mipmap_image_writes` extension adds support for writing to mipmap images, and requires support for the `cl_khr_mipmap_image` extension macro. The extension provides built-in image write functions operating on these images.

6.2.2.22. Select Floating-Point Rounding Mode

The `cl_khr_select_fprounding_mode` extension allows specifying the floating-point rounding mode for an instruction or group of instructions in the program source by use of a **#pragma**.

The extension was deprecated in OpenCL 1.1 and its use is not recommended.

6.2.2.23. sRGB Image Write Functions

The `cl_khr_srgb_image_writes` extension adds support for writing to sRGB images using the **write_imagef** functions. Color space conversion is performed by the function.

6.2.2.24. Sub-Groups

The `cl_khr_subgroups` extension was promoted to OpenCL C 2.1 as the `__opencl_c_subgroups` feature. The extension provides the following functions:

- Built-in Work-Item Functions for Sub-Groups
- Built-in Synchronization Functions for Sub-Groups
- Built-in Collective Functions for Sub-Groups
- Built-in Pipe Functions for Sub-Groups
- Built-in Kernel Query Functions for Sub-Groups
- The `memory_scope_sub_group` type and associated restrictions

6.2.2.25. Sub-Group Ballots

The `cl_khr_subgroup_ballot` extension adds the ability to collect and operate on ballots from work items in a sub-group. The extension provides the following functions:

- Built-in Ballot Functions for Sub-Groups

6.2.2.26. Sub-Group Clustered Reductions

The `cl_khr_subgroup_clustered_reduce` extension adds support for clustered reductions that operate on a subset of work items in the sub-group. The extension provides the following functions:

- Built-in Arithmetic Functions for Sub-Groups
- Built-in Bitwise Functions for Sub-Groups
- Built-in Logical Functions for Sub-Groups

6.2.2.27. Sub-Group Extended Types

The `cl_khr_subgroup_extended_types` extension was promoted to OpenCL C 3.1. The extension adds additional supported data types to the existing sub-group broadcast, scan, and reduction functions.

6.2.2.28. Sub-Group Non-Uniform Arithmetic

The `cl_khr_subgroup_non_uniform_arithmetic` extension adds the ability to use some sub-group functions within non-uniform flow control, including additional scan and reduction operators.

The extension provides the following functions:

- Built-in Non-Uniform Arithmetic Functions for Sub-Groups
- Built-in Non-Uniform Bitwise Functions for Sub-Groups
- Built-in Non-Uniform Logical Functions for Sub-Groups

6.2.2.29. Sub-Group Non-Uniform Vote and Election Functions

The `cl_khr_subgroup_non_uniform_vote` extension adds 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.

The extension provides the following functions:

- Built-in Non-Uniform Vote Functions for Sub-Groups

6.2.2.30. Sub-Group Rotation

The `cl_khr_subgroup_rotate` extension was promoted to OpenCL C 3.1. The extension adds support for a new sub-group data exchange operation that makes it possible to rotate values through the work items in a sub-group.

The extension provides the following functions:

- Built-in Rotation Functions for Sub-Groups

6.2.2.31. Sub-Group General Purpose Shuffles

The `cl_khr_subgroup_shuffle` extension was promoted to OpenCL C 3.1. The extension adds support for flexibly exchanging data among work items in a sub-group.

The extension provides the following functions:

- Built-in Shuffle Functions for Sub-Groups

6.2.2.32. Sub-Group Relative Shuffles

The `cl_khr_subgroup_shuffle_relative` was promoted to OpenCL C 3.1. The extension adds specialized ways to exchange data among work items in a sub-group that may perform better on some implementations.

The extension provides the following functions:

- Built-in Relative Shuffle Functions for Sub-Groups

6.2.2.33. Work-group Collective Uniform Arithmetic Functions

The `cl_khr_work_group_uniform_arithmetic` extension adds additional work-group collective functions, including 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`).

The extension provides the following functions:

- Built-in Work-group Logical Arithmetic Functions
- Built-in Work-group Bitwise Integer Functions
- Built-in Work-group Multiplicative Functions

6.3. Supported Data Types

The following data types are supported.

6.3.1. Built-in Scalar Data Types

The following table describes the list of built-in scalar data types.

Table 2. Built-in Scalar Data Types

Type	Description
`bool` [1]	A conditional data type which is either *true* or *false*. The value *true* expands to the integer constant 1 and the value *false* expands to the integer constant 0.
`char`	A signed two's complement 8-bit integer.
`unsigned char`, `uchar`	An unsigned 8-bit integer.
`short`	A signed two's complement 16-bit integer.
`unsigned short`, `ushort`	An unsigned 16-bit integer.
`int`	A signed two's complement 32-bit integer.
`unsigned int`, `uint`	An unsigned 32-bit integer.
`long` [2]	A signed two's complement 64-bit integer.
`unsigned long`, `ulong` [2]	An unsigned 64-bit integer.
`float`	A 32-bit floating-point number. The `float` data type must conform to the IEEE 754 single precision storage format.
`double` [3]	A 64-bit floating-point number. The `double` data type must conform to the IEEE 754 double-precision storage format. Requires support for double-precision.
`half`	A 16-bit floating-point number. The `half` data type must conform to the IEEE 754-2008 half-precision storage format.
`size_t` [4]	The unsigned integer type of the result of the `sizeof` operator.
`ptrdiff_t` [4]	A signed integer type that is the result of subtracting two pointers.

Type	Description
`intptr_t` [4]	A signed integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer.
`uintptr_t` [4]	An unsigned integer type with the property that any valid pointer to `void` can be converted to this type, then converted back to pointer to `void`, and the result will compare equal to the original pointer.
`void`	The `void` type comprises an empty set of values; it is an incomplete type that cannot be completed.

Most built-in scalar data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in scalar data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
`bool`	n/a
`char`	`cl_char`
`unsigned char`, `uchar`	`cl_uchar`
`short`	`cl_short`
`unsigned short`, `ushort`	`cl_ushort`
`int`	`cl_int`
`unsigned int`, `uint`	`cl_uint`
`long`	`cl_long`
`unsigned long`, `ulong`	`cl_ulong`
`float`	`cl_float`
`double`	`cl_double` [5]
`half`	`cl_half`
`size_t`	n/a
`ptrdiff_t`	n/a
`intptr_t`	n/a
`uintptr_t`	n/a
`void`	`void`

6.3.1.1. Double-Precision Floating-Point Support

Double-precision floating-point is supported if the `cl_khr_fp64` extension is supported, or if OpenCL 3.0 or newer and the `__opencl_c_fp64` feature is supported.

If double-precision is not supported, implementations may implicitly cast double-precision floating-point literals to single-precision literals. The use of double-precision literals without double-precision support should result in a diagnostic.

6.3.1.2. The `half` Data Type

The `half` data type must be IEEE 754-2008 compliant. `half` numbers have 1 sign bit, 5 exponent bits, and 10 mantissa bits. The interpretation of the sign, exponent and mantissa is analogous to IEEE 754 floating-point numbers. The exponent bias is 15. The `half` data type must represent finite and normal numbers, denormalized numbers, infinities and NaN. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` using **vstore_half** and converting a `half` to a `float` using **vload_half** cannot be flushed to zero. Conversions from `float` to `half` correctly round the mantissa to 11 bits of precision. Conversions from `half` to `float` are lossless; all `half` numbers are exactly representable as `float` values. Conversions from `double` to `half` are correctly rounded. Conversions from `half` to `double` are lossless.

The `half` data type can only be used to declare a pointer to a buffer that contains `half` values. A few valid examples are given below:

```
void
bar (__global half *p)
{
    ...
}

__kernel void
foo (__global half *pg, __local half *pl)
{
    __global half *ptr;
    int offset;

    ptr = pg + offset;
    bar(ptr);
}
```

Below are some examples that are not valid usage of the `half` type:

```
half a;
half b[100];
half *p;
a = *p; //  not allowed. must use *vload_half* function
```

Loads from a pointer to a `half` and stores to a pointer to a `half` can be performed using the vector

data load and store functions **vload_half**, **vload_half*n***, **vloada_halfn** and **vstore_half**, **vstore_half*n***, and **vstorea_halfn**. The load functions read scalar or vector `half` values from memory and convert them to a scalar or vector `float` value. The store functions take a scalar or vector `float` value as input, convert it to a `half` scalar or vector value (with appropriate rounding mode) and write the `half` scalar or vector value to memory.

6.3.2. Built-in Vector Data Types

The `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float` and `double` vector data types are supported. [6] The vector data type is defined with the type name, i.e. `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, or `double` followed by a literal value *n* that defines the number of elements in the vector. Supported values of *n* are 2, 3, 4, 8, and 16 for all vector data types.

Vector types with three elements, i.e. where *n* is 3, require support for OpenCL C 1.1 or newer.

The following table describes the list of built-in vector data types.

Table 3. Built-in Vector Data Types

Type	Description
`charn`	A vector of *n* 8-bit signed two's complement integer values.
`ucharn`	A vector of *n* 8-bit unsigned integer values.
`shortn`	A vector of *n* 16-bit signed two's complement integer values.
`ushortn`	A vector of *n* 16-bit unsigned integer values.
`intn`	A vector of *n* 32-bit signed two's complement integer values.
`uintn`	A vector of *n* 32-bit unsigned integer values.
`longn` [7]	A vector of *n* 64-bit signed two's complement integer values.
`ulongn` [7]	A vector of *n* 64-bit unsigned integer values.
`halfn` [8]	A vector of *n* 16-bit floating-point values.
`floatn`	A vector of *n* 32-bit floating-point values.
`doublen` [9]	A vector of *n* 64-bit floating-point values. Requires support for double-precision.

The built-in vector data types are also declared as appropriate types in the OpenCL API (and header files) that can be used by an application. The following table describes the built-in vector data type in the OpenCL C programming language and the corresponding data type available to the application:

Type in OpenCL Language	API type for application
charn	cl_charn
ucharn	cl_ucharn
shortn	cl_shortn
ushortn	cl_ushortn
intn	cl_intn
uintn	cl_uintn
longn	cl_longn
ulongn	cl_ulongn
halfn	cl_halfn
floatn	cl_floatn
doublen	cl_doublen

6.3.3. Other Built-in Data Types

The following table describes the list of additional data types supported by OpenCL.

Table 4. Other Built-in Data Types

Type	Description
image2d_t [10]	A 2D image.
image3d_t [10]	A 3D image.
image2d_array_t [10]	A 2D image array. Requires support for OpenCL C 1.2 or newer.
image1d_t [10]	A 1D image. Requires support for OpenCL C 1.2 or newer.
image1d_buffer_t [10]	A 1D image created from a buffer object. Requires support for OpenCL C 1.2 or newer.
image1d_array_t [10]	A 1D image array. Requires support for OpenCL C 1.2 or newer.
image2d_depth_t [10]	A 2D depth image. Requires support for OpenCL C 2.0 or newer, or for the cl_khr_depth_images extension macro.
image2d_array_depth_t [10]	A 2D depth image array. Requires support for OpenCL C 2.0 or newer, or for the cl_khr_depth_images extension macro.

Type	Description
`sampler_t` [10]	A sampler type.
`queue_t`	A device command-queue. This queue can only be used to enqueue commands from kernels executing on the device. Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_device_enqueue` feature.
`ndrange_t`	The N-dimensional range over which a kernel executes. Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_device_enqueue` feature.
`clk_event_t`	A device-side event that identifies a command enqueued to a device command-queue. Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_device_enqueue` feature.
`reserve_id_t`	A reservation ID. This opaque type is used to identify the reservation for reading and writing a pipe. Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_pipes` feature.
`event_t`	An event. This can be used to identify async copies from `global` to `local` memory and vice-versa.

`cl_mem_fence_flags`	This is a bitfield and can be 0 or a combination of the following values ORed together:
	
	`CLK_GLOBAL_MEM_FENCE`
	`CLK_LOCAL_MEM_FENCE`
	`CLK_IMAGE_MEM_FENCE`
	
	These flags are described in detail in the synchronization functions section.

Type	Description
image2d_msaa_t	A 2D multi-sample color image. Refer to the Built-in Image Sampler-less Read Functions section for a detailed description of the built-in functions that use this type. Requires support for the cl_khr_gl_msaa_sharing extension macro.
image2d_array_msaa_t	A 2D multi-sample color image array. Refer to the Built-in Image Sampler-less Read Functions section for a detailed description of the built-in functions that use this type. Requires support for the cl_khr_gl_msaa_sharing extension macro.
image2d_msaa_depth_t	A 2D multi-sample depth image. Refer to the Built-in Image Sampler-less Read Functions section for a detailed description of the built-in functions that use this type. Requires support for the cl_khr_gl_msaa_sharing extension macro.
image2d_array_msaa_depth_t	A 2D multi-sample depth image array. Refer to the Built-in Image Sampler-less Read Functions section for a detailed description of the built-in functions that use this type. Requires support for the cl_khr_gl_msaa_sharing extension macro.

The image2d_t, image3d_t, image2d_array_t, image1d_t, image1d_buffer_t, image1d_array_t, image2d_depth_t, image2d_array_depth_t and sampler_t types are only defined if the device supports images, i.e. the value of the CL_DEVICE_IMAGE_SUPPORT device query) is CL_TRUE. If this is the case then an OpenCL C 3.0 or newer compiler must also define the __opencl_c_images feature macro.

The C99 derived types (arrays, structs, unions, functions, and pointers), constructed from the built-in scalar, vector, and other data types are supported, with specified restrictions.

The following tables describe the other built-in data types in OpenCL described in Other Built-in Data Types and the corresponding data type available to the application:

Type in OpenCL C	API type for application
image2d_t	cl_mem
image3d_t	cl_mem

Type in OpenCL C	API type for application
image2d_array_t	cl_mem
image1d_t	cl_mem
image1d_buffer_t	cl_mem
image1d_array_t	cl_mem
image2d_depth_t	cl_mem
image2d_array_depth_t	cl_mem
sampler_t	cl_sampler
queue_t	cl_command_queue
ndrange_t	N/A
clk_event_t	N/A
reserve_id_t	N/A
event_t	N/A
cl_mem_fence_flags	N/A

6.3.4. Reserved Data Types

The data type names described in the following table are reserved and cannot be used by applications as type names. The vector data type names defined in Built-in Vector Data Types, but where *n* is any value other than 2, 3, 4, 8 and 16, are also reserved.

Table 5. Reserved Data Types

Type	Description
booln	A boolean vector.
quad, quadn	A 128-bit floating-point scalar and vector.
complex half, complex halfn	A complex 16-bit floating-point scalar and vector.
imaginary half, imaginary halfn	An imaginary 16-bit floating-point scalar and vector.
complex float, complex floatn	A complex 32-bit floating-point scalar and vector.
imaginary float, imaginary floatn	An imaginary 32-bit floating-point scalar and vector.
complex double, complex doublen	A complex 64-bit floating-point scalar and vector.
imaginary double, imaginary doublen	An imaginary 64-bit floating-point scalar and vector.
complex quad, complex quadn	A complex 128-bit floating-point scalar and vector.

Type	Description
`imaginary quad`, `imaginary quadn`	An imaginary 128-bit floating-point scalar and vector.
`floatnxm`	An $n \times m$ matrix of single precision floating-point values stored in column-major order.
`doublenxm`	An $n \times m$ matrix of double-precision floating-point values stored in column-major order.
`long double`, `long doublen`	A floating-point scalar and vector type with at least as much precision and range as a `double` and no more precision and range than a quad.
`long long, long longn`	A 128-bit signed integer scalar and vector.
`unsigned long long`, `ulong long`, `ulong longn`	A 128-bit unsigned integer scalar and vector.

6.3.5. Alignment of Types

A data item declared to be a data type in memory is always aligned to the size of the data type in bytes. For example, a `float4` variable will be aligned to a 16-byte boundary, a `char2` variable will be aligned to a 2-byte boundary.

For 3-component vector data types, the size of the data type is `4 * sizeof(component)`. This means that a 3-component vector data type will be aligned to a `4 * sizeof(component)` boundary. The **vload3** and **vstore3** built-in functions can be used to read and write, respectively, 3-component vector data types from an array of packed scalar data type.

A built-in data type that is not a power of two bytes in size must be aligned to the next larger power of two. This rule applies to built-in types only, not structs or unions.

The OpenCL compiler is responsible for aligning data items to the appropriate alignment as required by the data type. For arguments to a `__kernel` function declared to be a pointer to a data type, the OpenCL compiler can assume that the pointee is always appropriately aligned as required by the data type. The behavior of an unaligned load or store is undefined, except for the vector data load and store functions **vload*n***, **vload_half*n***, **vstore*n***, and **vstore_half*n***. The vector load functions can read a vector from an address aligned to the element type of the vector. The vector store functions can write a vector to an address aligned to the element type of the vector.

6.3.6. Vector Literals

Vector literals can be used to create vectors from a list of scalars, vectors or a mixture thereof. A vector literal can be used either as a vector initializer or as a primary expression. Whether a vector literal can be used as an l-value is implementation-defined.

A vector literal is written as a parenthesized vector type followed by a parenthesized comma delimited list of parameters. A vector literal operates as an overloaded function. The forms of the function that are available is the set of possible argument lists for which all arguments have the same element type as the result vector, and the total number of elements is equal to the number of elements in the result vector. In addition, a form with a single scalar of the same type as the

element type of the vector is available. For example, the following forms are available for `float4`:

```
(float4)( float, float, float, float )
(float4)( float2, float, float )
(float4)( float, float2, float )
(float4)( float, float, float2 )
(float4)( float2, float2 )
(float4)( float3, float )
(float4)( float, float3 )
(float4)( float )
```

Operands are evaluated by standard rules for function evaluation, except that implicit scalar widening shall not occur. The order in which the operands are evaluated is undefined. The operands are assigned to their respective positions in the result vector as they appear in memory order. That is, the first element of the first operand is assigned to `result.x`, the second element of the first operand (or the first element of the second operand if the first operand was a scalar) is assigned to `result.y`, etc. In the case of the form that has a single scalar operand, the operand is replicated across all lanes of the vector.

Examples:

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uint4 u = (uint4)(1); //  u will be (1, 1, 1, 1).
float4 f = (float4)((float2)(1.0f, 2.0f), (float2)(3.0f, 4.0f));
float4 f = (float4)(1.0f, (float2)(2.0f, 3.0f), 4.0f);
float4 f = (float4)(1.0f, 2.0f); //  error
```

6.3.7. Vector Components

The components of vector data types can be addressed as `<vector_data_type>.xyzw`. Vector data types with two or more components, such as `char2`, can access `.xy` elements. Vector data types with three or more components, such as `uint3`, can access `.xyz` elements. Vector data types with four or more components, such as `ulong4` or `float8`, can access `.xyzw` elements.

In OpenCL C 3.0 or newer, the components of vector data types can also be addressed as `<vector_data_type>.rgba`. Vector data types with two or more components can access `.rg` elements. Vector data types with three or more components can access `.rgb` elements. Vector data types with four or more components can access `.rgba` elements.

Accessing components beyond those declared for the vector type is an error so, for example:

```
float2 coord;
coord.x = 1.0f; // is legal
coord.r = 1.0f; // is legal in OpenCL C 3.0 or newer
coord.z = 1.0f; // is illegal, since coord only has two components

float3 pos;
```

```
pos.z = 1.0f; // is legal
pos.b = 1.0f; // is legal in OpenCL C 3.0 or newer
pos.w = 1.0f; // is illegal, since pos only has three components
```

The component selection syntax allows multiple components to be selected by appending their names after the period (.).

```
float4 c;

c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyz = (float3)(3.0f, 4.0f, 5.0f);
```

The component selection syntax also allows components to be permuted or replicated.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 swiz= pos.wzyx; // swiz = (4.0f, 3.0f, 2.0f, 1.0f)

float4 dup = pos.xxyy; // dup = (1.0f, 1.0f, 2.0f, 2.0f)
```

The component group notation can occur on the left hand side of an expression. To form an l-value, swizzling must be applied to an l-value of vector type, contain no duplicate components, and it results in an l-value of scalar or vector type, depending on number of components specified. Each component must be a supported scalar or vector type.

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

pos.xw = (float2)(5.0f, 6.0f);// pos = (5.0f, 2.0f, 3.0f, 6.0f)
pos.wx = (float2)(7.0f, 8.0f);// pos = (8.0f, 2.0f, 3.0f, 7.0f)
pos.xyz = (float3)(3.0f, 5.0f, 9.0f); // pos = (3.0f, 5.0f, 9.0f, 4.0f)
pos.xx = (float2)(3.0f, 4.0f);// illegal - 'x' used twice

// illegal - mismatch between float2 and float4
pos.xy = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 a, b, c, d;
float16 x;
x = (float16)(a, b, c, d);
x = (float16)(a.xxxx, b.xyz, c.xyz, d.xyz, a.yzw);

// illegal - component a.xxxxxxx is not a valid vector type
x = (float16)(a.xxxxxxx, b.xyz, c.xyz, d.xyz);
```

Elements of vector data types can also be accessed using a numeric index to refer to the

appropriate element in the vector. The numeric indices that can be used are given in the table below:

Table 6. Numeric indices for built-in vector data types

Vector Components	Numeric indices that can be used
2-component	0, 1
3-component	0, 1, 2
4-component	0, 1, 2, 3
8-component	0, 1, 2, 3, 4, 5, 6, 7
16-component	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F

The numeric indices must be preceded by the letter `s` or `S`.

In the following example

```
float8 f;
```

`f.s0` refers to the 1st element of the `float8` variable `f` and `f.s7` refers to the 8th element of the `float8` variable `f`.

In the following example

```
float16 x;
```

`x.sa` (or `x.sA`) refers to the 11th element of the `float16` variable `x` and `x.sf` (or `x.sF`) refers to the 16th element of the `float16` variable `x`.

The numeric indices used to refer to an appropriate element in the vector cannot be intermixed with `.xyzw` notation used to access elements of a 1 .. 4 component vector.

For example

```
float4 f, a;

a = f.x12w;       // illegal use of numeric indices with .xyzw

a.xyzw = f.s0123; // valid
```

Vector data types can use the `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes to get smaller vector types or to combine smaller vector types to a larger vector type. Multiple levels of `.lo` (or `.even`) and `.hi` (or `.odd`) suffixes can be used until they refer to a scalar term.

The `.lo` suffix refers to the lower half of a given vector. The `.hi` suffix refers to the upper half of a given vector.

The `.even` suffix refers to the even elements of a vector. The `.odd` suffix refers to the odd elements of a vector.

Some examples to help illustrate this are given below:

```
float4 vf;

float2 low = vf.lo;    // returns vf.xy
float2 high = vf.hi;   // returns vf.zw

float2 even = vf.even; // returns vf.xz
float2 odd = vf.odd;   // returns vf.yw
```

The suffixes `.lo` (or `.even`) and `.hi` (or `.odd`) for a 3-component vector type operate as if the 3-component vector type is a 4-component vector type with the value in the `w` component undefined.

Some examples are given below:

```
float8 vf;
float4 odd = vf.odd;
float4 even = vf.even;
float2 high = vf.even.hi;
float2 low = vf.odd.lo;

// interleave LR stereo stream
float4 left, right;
float8 interleaved;
interleaved.even = left;
interleaved.odd = right;

// deinterleave
left = interleaved.even;
right = interleaved.odd;

// transpose a 4x4 matrix

void transpose( float4 m[4] )
{
    // read matrix into a float16 vector
    float16 x = (float16)( m[0], m[1], m[2], m[3] );
    float16 t;

    // transpose
    t.even = x.lo;
    t.odd = x.hi;
    x.even = t.lo;
    x.odd = t.hi;

    // write back
```

```
    m[0] = x.lo.lo; // { m[0][0], m[1][0], m[2][0], m[3][0] }
    m[1] = x.lo.hi; // { m[0][1], m[1][1], m[2][1], m[3][1] }
    m[2] = x.hi.lo; // { m[0][2], m[1][2], m[2][2], m[3][2] }
    m[3] = x.hi.hi; // { m[0][3], m[1][3], m[2][3], m[3][3] }
}

float3 vf = (float3)(1.0f, 2.0f, 3.0f);
float2 low = vf.lo; // (1.0f, 2.0f);
float2 high = vf.hi; // (3.0f, _undefined_);
```

It is illegal to take the address of a vector element and will result in a compilation error. For example:

```
float8 vf;

float *f = &vf.x;           // is illegal
float2 *f2 = &vf.s07;       // is illegal

float4 *odd = &vf.odd;      // is illegal
float4 *even = &vf.even;    // is illegal
float2 *high = &vf.even.hi; // is illegal
float2 *low = &vf.odd.lo;   // is illegal
```

6.3.8. Aliasing Rules

OpenCL C programs shall comply with the C99 type-based aliasing rules defined in section 6.5, item 7 of the C99 Specification. The OpenCL C built-in vector data types are considered aggregate types [11] for the purpose of applying these aliasing rules.

6.3.9. Keywords

The following names are reserved for use as keywords in OpenCL C and shall not be used otherwise.

- Names reserved as keywords by C99.
- OpenCL C data types defined in Built-in Vector Data Types, Other Built-in Data Types, and Reserved Data Types.
- Address space qualifiers: `__global`, `global`, `__local`, `local`, `__constant`, `constant`, `__private`, and `private`. `__generic` and `generic` are reserved for future use.
- Function qualifiers: `__kernel` and `kernel`.
- Access qualifiers: `__read_only`, `read_only`, `__write_only`, `write_only`, `__read_write` and `read_write`.
- `uniform`, `pipe`.

6.4. Conversions and Type Casting

6.4.1. Implicit Conversions

Implicit conversions between scalar built-in types defined in Built-in Scalar Data Types (except `void` and `half` [12]) are supported. When an implicit conversion is done, it is not just a re-interpretation of the expression's value but a conversion of that value to an equivalent value in the new type. For example, the integer value 5 will be converted to the floating-point value 5.0.

Implicit conversions from a scalar type to a vector type are allowed. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector. The scalar type is then widened to the vector.

Implicit conversions between built-in vector data types are disallowed.

Implicit conversions for pointer types follow the rules described in the C99 Specification.

6.4.2. Explicit Casts

Standard typecasts for built-in scalar data types defined in Built-in Scalar Data Types will perform appropriate conversion (except `void` and `half` [13]). In the example below:

```
float f = 1.0f;
int i = (int)f;
```

`f` stores `0x3F800000` and `i` stores `0x1` which is the floating-point value `1.0f` in `f` converted to an integer value.

Explicit casts between vector types are not legal. The examples below will generate a compilation error.

```
int4 i;
uint4 u = (uint4) i; //  not allowed

float4 f;
int4 i = (int4) f; //  not allowed

float4 f;
int8 i = (int8) f; //  not allowed
```

Scalar to vector conversions may be performed by casting the scalar to the desired vector data type. Type casting will also perform appropriate arithmetic conversion. The round to zero rounding mode will be used for conversions to built-in integer vector types. The default rounding mode will be used for conversions to floating-point vector types. When casting a `bool` to a vector integer data type, the vector components will be set to -1 (i.e. all bits set) if the bool value is *true* and 0 otherwise.

Below are some correct examples of explicit casts.

```
float f = 1.0f;
float4 va = (float4)f;

// va is a float4 vector with elements (f, f, f, f).

uchar u = 0xFF;
float4 vb = (float4)u;

// vb is a float4 vector with elements
// ((float)u, (float)u, (float)u, (float)u).

float f = 2.0f;
int2 vc = (int2)f;

// vc is an int2 vector with elements ((int)f, (int)f).

uchar4 vtrue = (uchar4)true;

// vtrue is a uchar4 vector with elements (0xff, 0xff,
// 0xff, 0xff).
```

6.4.3. Explicit Conversions

Explicit conversions may be performed using the

```
convert_destType(sourceType)
```

suite of functions. These provide a full set of type conversions between supported scalar and vector data types:

- `char`, `charn`, `uchar`, `ucharn`
- `short`, `shortn`, `ushort`, `ushortn`
- `int`, `intn`, `uint`, `uintn`
- `long` [14], `longn`, `ulong`, `ulongn`
- `float`, `floatn`
- `half` [15], `halfn`
- `double` [16], `doublen`

The number of elements in the source and destination vectors must match.

In the example below:

```
uchar4 u;
int4 c = convert_int4(u);
```

`convert_int4` converts a `uchar4` vector `u` to an `int4` vector `c`.

```
float f;
int i = convert_int(f);
```

`convert_int` converts a `float` scalar `f` to an int scalar `i`.

The behavior of the conversion may be modified by one or two optional modifiers that specify saturation for out-of-range inputs and rounding behavior.

The full form of the scalar convert function is:

```
destType convert_destType<_sat><_roundingMode>(sourceType)
```

where `dstType` is the destination scalar type and `sourceType` is the source scalar type.

The full form of the vector convert function is:

```
destTypen convert_destTypen<_sat><_roundingMode>(sourceTypen)
```

where `destTypen` is the n-element destination vector type and `sourceTypen` is the n-element source vector type.

6.4.3.1. Data Types

Conversions are available for the following scalar types: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, and built-in vector types derived therefrom. The operand and result type must have the same number of elements. The operand and result type may be the same type in which case the conversion has no effect on the type or value of an expression.

Conversions between integer types follow the conversion rules specified in sections 6.3.1.1 and 6.3.1.3 of the C99 Specification except for out-of-range behavior and saturated conversions.

6.4.3.2. Rounding Modes

Conversions to and from floating-point type shall conform to IEEE-754 rounding rules. Conversions may have an optional rounding mode modifier described in the following table.

Table 7. Rounding Modes

Modifier	Rounding Mode Description
`_rte`	Round to nearest even
`_rtz`	Round toward zero
`_rtp`	Round toward positive infinity
`_rtn`	Round toward negative infinity

Modifier	Rounding Mode Description
no modifier specified	Use the default rounding mode for this destination type, `_rtz` for conversion to integers or the default rounding mode for conversion to floating-point types.

By default, conversions to integer type use the `_rtz` (round toward zero) rounding mode and conversions to floating-point type [17] use the default rounding mode. The only default floating-point rounding mode supported is round to nearest even i.e the default rounding mode will be `_rte` for floating-point types.

6.4.3.3. Out-of-Range Behavior and Saturated Conversions

When the conversion operand is either greater than the greatest representable destination value or less than the least representable destination value, it is said to be out-of-range. The result of out-of-range conversion is determined by the conversion rules specified by section 6.3 of the C99 Specification. When converting from a floating-point type to integer type, the behavior is implementation-defined.

Conversions to integer types may opt to convert using the optional saturated mode by appending the `_sat` modifier to the conversion function name. When in saturated mode, values that are outside the representable range are clamped to the nearest representable value in the destination format, and NaN is converted to zero.

Conversions to floating-point types conform to IEEE-754 rounding rules. The `_sat` modifier may not be used for conversions to floating-point formats.

6.4.3.4. Explicit Conversion Examples

Example 1:

```
short4 s;

// negative values clamped to zero
ushort4 u = convert_ushort4_sat( s );

// values > CHAR_MAX converted to CHAR_MAX
// values < CHAR_MIN converted to CHAR_MIN
char4 c = convert_char4_sat( s );
```

Example 2:

```
float4 f;

// values implementation-defined for
// f > INT_MAX, f < INT_MIN or NaN
int4 i = convert_int4( f );
```

```
// values > INT_MAX clamp to INT_MAX, values < INT_MIN clamp
// to INT_MIN, and NaN is converted to zero
// The _rtz_ rounding mode is used to produce the integer values.
int4 i2 = convert_int4_sat( f );

// similar to convert_int4, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i3 = convert_int4_rte( f );

// similar to convert_int4_sat, except that floating-point values
// are rounded to the nearest integer instead of truncated
int4 i4 = convert_int4_sat_rte( f );
```

Example 3:

```
int4 i;

// convert ints to floats using the default rounding mode.
float4 f = convert_float4( i );

// convert ints to floats. integer values that cannot
// be exactly represented as floats should round up to the
// next representable float.
float4 f = convert_float4_rtp( i );
```

6.4.4. Reinterpreting Data as Another Type

It is frequently necessary to reinterpret bits in a data type as another data type in OpenCL. This is typically required when direct access to the bits in a floating-point type is needed, for example to mask off the sign bit or make use of the result of a vector relational operator on floating-point data [18]. Several methods to achieve this (non-) conversion are frequently practiced in C, including pointer aliasing, unions and memcpy. Of these, only memcpy is strictly correct in C99. Since OpenCL does not provide **memcpy**, other methods are needed.

6.4.4.1. Reinterpreting Types Using Unions

The OpenCL language extends the union to allow the program to access a member of a union object using a member of a different type. The relevant bytes of the representation of the object are treated as an object of the type used for the access. If the type used for access is larger than the representation of the object, then the value of the additional bytes is undefined.

Examples:

```
// d only if double-precision is supported
union { float f; uint u; double d; } u;

u.u = 1;    // u.f contains 2**-149.  u.d is undefined --
            // depending on endianness the low or high half
```

```
            // of d is unknown

u.f = 1.0f; // u.u contains 0x3f800000, u.d contains an
            // undefined value -- depending on endianness
            // the low or high half of d is unknown

u.d = 1.0;  // u.u contains 0x3ff00000 (big endian) or 0
            // (little endian). u.f contains either 0x1.ep0f
            // (big endian) or 0.0f (little endian)
```

6.4.4.2. Reinterpreting Types Using as_type() and as_type*n*()

All data types described in Built-in Scalar Data Types and Built-in Vector Data Types (except `bool`, `void`, and `half` [19]) may be also reinterpreted as another data type of the same size using the **as_type**() operator for scalar data types and the **as_type*n***() operator [20] for vector data types. When the operand and result type contain the same number of elements, the bits in the operand shall be returned directly without modification as the new type. The usual type promotion for function arguments shall not be performed.

For example, `as_float(0x3f800000)` returns `1.0f`, which is the value that the bit pattern `0x3f800000` has if viewed as an IEEE-754 single precision value.

When the operand and result type contain a different number of elements, the result shall be implementation-defined except if the operand is a 4-component vector and the result is a 3-component vector. In this case, the bits in the operand shall be returned directly without modification as the new type. That is, a conforming implementation shall explicitly define a behavior, but two conforming implementations need not have the same behavior when the number of elements in the result and operand types does not match. The implementation may define the result to contain all, some or none of the original bits in whatever order it chooses. It is an error to use **as_type**() or **as_type*n***() operator to reinterpret data to a type of a different number of bytes.

Examples:

```
float f = 1.0f;
uint u = as_uint(f); // Legal. Contains:  0x3f800000

float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
// Legal. Contains:
// (int4)(0x3f800000, 0x40000000, 0x40400000, 0x40800000)
int4 i = as_int4(f);

float4 f, g;
int4  is_less = f < g;

// Legal. f[i] = f[i] < g[i] ? f[i] : 0.0f
f = as_float4(as_int4(f) & is_less);

int i;
```

```
// Legal. Result is implementation-defined.
short2 j = as_short2(i);

int4 i;
// Legal. Result is implementation-defined.
short8 j = as_short8(i);

float4 f;
// Error. Result and operand have different sizes
double4 g = as_double4(f); // Only if double-precision is supported.

float4 f;
// Legal. g.xyz will have same values as f.xyz. g.w is undefined
float3 g = as_float3(f);
```

6.4.5. Pointer Casting

Pointers to old and new types may be cast back and forth to each other. Casting a pointer to a new type represents an unchecked assertion that the address is correctly aligned. The developer will also need to know the endianness of the OpenCL device and the endianness of the data to determine how the scalar and vector data elements are stored in memory.

6.4.6. Usual Arithmetic Conversions

Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. For this purpose, all vector types shall be considered to have higher conversion ranks than scalars. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called the usual arithmetic conversions. If the operands are of more than one vector type, then an error shall occur. Implicit conversions between vector types are not permitted.

Otherwise, if there is only a single vector type, and all other operands are scalar types, the scalar types are converted to the type of the vector element, then widened into a new vector containing the same number of elements as the vector, by duplication of the scalar value across the width of the new vector. An error shall occur if any scalar operand has greater rank than the type of the vector element. For this purpose, the rank order defined as follows:

1. The rank of a floating-point type is greater than the rank of another floating-point type, if the first floating-point type can exactly represent all numeric values in the second floating-point type. (For this purpose, the encoding of the floating-point value is used, rather than the subset of the encoding usable by the device.)
2. The rank of any floating-point type is greater than the rank of any integer type.
3. The rank of an integer type is greater than the rank of an integer type with less precision.
4. The rank of an unsigned integer type is **greater than** the rank of a signed integer type with the

same precision [21].

5. The rank of the bool type is less than the rank of any other type.
6. The rank of an enumerated type shall equal the rank of the compatible integer type.
7. For all types, `T1`, `T2` and `T3`, if `T1` has greater rank than `T2`, and `T2` has greater rank than `T3`, then `T1` has greater rank than `T3`.

Otherwise, if all operands are scalar, the usual arithmetic conversions apply, per section 6.3.1.8 of the C99 Specification.

Both the standard orderings in sections 6.3.1.8 and 6.3.1.1 of the C99 Specification were examined and rejected. Had we used integer conversion rank here, `int4 + 0U` would have been legal and had `int4` return type. Had we used standard C99 usual arithmetic conversion rules for scalars, then the standard integer promotion would have been performed on vector integer element types and `short8 + char` would either have return type of `int8` or be illegal.

6.5. Operators

6.5.1. Arithmetic Operators

The arithmetic operators add (+), subtract (-), multiply (*) and divide (/) operate on built-in integer and floating-point scalar, and vector data types. The arithmetic operator remainder (%) operates on built-in integer scalar and integer vector data types. All arithmetic operators return result of the same built-in type (integer or floating-point) as the type of the operands, after operand type conversion. After conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal. Division on integer types which results in a value that lies outside of the range bounded by the maximum and minimum representable values of the integer type will not cause an exception but will result in an unspecified value. A divide by zero with integer types does not cause an exception but will result in an unspecified value. Division by zero for floating-point types will result in ±∞ or NaN as prescribed by the IEEE-754 standard. Use the built-in functions **dot** and **cross** to get, respectively, the vector dot product and the vector cross product.

6.5.2. Unary Operators

The arithmetic unary operators (+ and -) operate on built-in scalar and vector types.

6.5.3. Pre- and Post-Operators

The arithmetic post- and pre-increment and decrement operators (-- and ++) operate on built-in scalar and vector types except the built-in scalar and vector `float` types [22]. All unary operators work component-wise on their operands. These result with the same type they operated on. For post- and pre-increment and decrement, the expression must be one that could be assigned to (an l-value). Pre-increment and pre-decrement add or subtract 1 to the contents of the expression they operate on, and the value of the pre-increment or pre-decrement expression is the resulting value of that modification. Post-increment and post-decrement expressions add or subtract 1 to the contents of the expression they operate on, but the resulting expression has the expression's value before the post-increment or post-decrement was executed.

6.5.4. Relational Operators

The relational operators greater than (>), less than (<), greater than or equal (>=), and less than or equal (<=) operate on scalar and vector types [23]. All relational operators result in an integer type. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in an `int` scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.
- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `halfn` [24], `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the relational operators shall return 0 if the specified relation is *false* and return 1 if the specified relation is *true*. For vector types, the relational operators shall return 0 if the specified relation is *false* and return -1 (i.e. all bits set) if the specified relation is *true*. The relational operators always return 0 if either argument is not a number (NaN).

6.5.5. Equality Operators

The equality operators equal (==) and not equal (!=) operate on built-in scalar and vector types [25]. All equality operators result in an integer type. After operand type conversion, the following cases are valid:

- The two operands are scalars. In this case, the operation is applied, resulting in a scalar.
- One operand is a scalar, and the other is a vector. In this case, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is

then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

- The two operands are vectors of the same type. In this case, the operation is done component-wise resulting in the same size vector.

All other cases of implicit conversions are illegal.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `halfn` [26], `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the equality operators shall return 0 if the specified relation is *false* and return 1 if the specified relation is *true*. For vector types, the equality operators shall return 0 if the specified relation is *false* and return -1 (i.e. all bits set) if the specified relation is *true*. The equality operator equal (==) returns 0 if one or both arguments are not a number (NaN). The equality operator not equal (**!**=) returns 1 (for scalar source operands) or -1 (for vector source operands) if one or both arguments are not a number (NaN).

6.5.6. Bitwise Operators

The bitwise operators and (**&**), or (|), exclusive or (^), and not (~) operate on all scalar and vector built-in types except the built-in scalar and vector `float` types. For vector built-in types, the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector. Vector source operands of type `halfn` [27] return a `shortn` result.

6.5.7. Logical Operators

The logical operators and (**&&**) and or (||) operate on all scalar and vector built-in types. For scalar built-in types only, and (**&&**) will only evaluate the right hand operand if the left hand operand compares unequal to 0. For scalar built-in types only, or (||) will only evaluate the right hand operand if the left hand operand compares equal to 0. For built-in vector types, both operands are evaluated and the operators are applied component-wise. If one operand is a scalar and the other is a vector, the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

The logical operator exclusive or (^^) is reserved.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `halfn` [28], `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the logical operators shall return 0 if the result of the operation is *false* and return 1 if the result is *true*. For vector types, the logical operators shall return 0 if the result of the operation is *false* and return -1 (i.e. all bits set) if the result is *true*.

6.5.8. Unary Logical Operator

The logical unary operator not (!) operates on all scalar and vector built-in types. For built-in vector types, the operators are applied component-wise.

The result is a scalar signed integer of type `int` if the source operands are scalar and a vector signed integer type of the same size as the source operands if the source operands are vector types. Vector source operands of type `charn` and `ucharn` return a `charn` result; vector source operands of type `halfn` [29], `shortn` and `ushortn` return a `shortn` result; vector source operands of type `intn`, `uintn` and `floatn` return an `intn` result; vector source operands of type `longn`, `ulongn` and `doublen` return a `longn` result.

For scalar types, the logical unary operator shall return 0 if the value of its operand compares unequal to 0, and return 1 if the value of its operand compares equal to 0. For vector types, the unary operator shall return 0 if the value of its operand compares unequal to 0, and return -1 (i.e. all bits set) if the value of its operand compares equal to 0.

6.5.9. Ternary Selection Operator

The ternary selection operator (**?:**) operates on three expressions (*exp1* **?** *exp2* **:** *exp3*). This operator evaluates the first expression *exp1*, which can be a scalar or vector result except `float`. If all three expressions are scalar values, the C99 rules for ternary operator are followed. If the result is a vector value, then this is equivalent to calling **select**(*exp3*, *exp2*, *exp1*). The **select** function is described in Built-in Scalar and Vector Relational Functions. The second and third expressions can be any type, as long their types match, or there is an implicit conversion that can be applied to one of the expressions to make their types match, or one is a vector and the other is a scalar and the scalar may be subject to the usual arithmetic conversion to the element type used by the vector operand and widened to the same type as the vector type. This resulting matching type is the type of the entire expression.

6.5.10. Shift Operators

The operators right-shift (>>), left-shift (<<) operate on all scalar and vector built-in types except the built-in scalar and vector `float` types. For built-in vector types, the operators are applied component-wise. For the right-shift (>>), left-shift (<<) operators, the rightmost operand must be a scalar if the first operand is a scalar, and the rightmost operand can be a vector or scalar if the first operand is a vector.

The result of `E1` << `E2` is `E1` left-shifted by $\log_2$(N) least significant bits in `E2` viewed as an unsigned integer value, where N is the number of bits used to represent the data type of `E1` after integer promotion [30], if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a vector. The vacated bits are filled with zeros.

The result of `E1` >> `E2` is `E1` right-shifted by $\log_2$(N) least significant bits in `E2` viewed as an unsigned integer value, where N is the number of bits used to represent the data type of `E1` after integer promotion, if `E1` is a scalar, or the number of bits used to represent the type of `E1` elements, if `E1` is a

vector. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the vacated bits are filled with zeros. If E1 has a signed type and a negative value, the vacated bits are filled with ones.

6.5.11. Sizeof Operator

The sizeof operator yields the size (in bytes) of its operand, including any padding bytes needed for alignment, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is of type size_t. If the type of the operand is a variable length array [31] type, the operand is evaluated; otherwise, the operand is not evaluated and the result is an integer constant.

When applied to an operand that has type char or uchar, the result is 1. When applied to an operand that has type short, ushort, or half the result is 2. When applied to an operand that has type int, uint or float, the result is 4. When applied to an operand that has type long, ulong or double, the result is 8. When applied to an operand that is a vector type, the result is the number of components times the size of each scalar component [32]. When applied to an operand that has array type, the result is the total number of bytes in the array. When applied to an operand that has structure or union type, the result is the total number of bytes in such an object, including internal and trailing padding. The sizeof operator shall not be applied to an expression that has function type or an incomplete type, to the parenthesized name of such a type, or to an expression that designates a bit-field struct member [33].

The behavior of applying the sizeof operator to the bool, image2d_t, image3d_t, image2d_array_t, image1d_t, image1d_buffer_t, image1d_array_t, image2d_depth_t, image2d_array_depth_t, sampler_t, queue_t, ndrange_t, clk_event_t, reserve_id_t, and event_t types is implementation-defined. Additionally, the behavior of applying the sizeof operator to a pipe object (a type with the pipe type specifier keyword) is implementation-defined.

6.5.12. Comma Operator

The comma (,) operator operates on expressions by returning the type and value of the right-most expression in a comma separated list of expressions. All expressions are evaluated, in order, from left to right.

6.5.13. Indirection Operator

The unary (*) operator denotes indirection. If the operand points to an object, the result is an l-value designating the object. If the operand has type "pointer to *type*", the result has type "*type*". If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined [34].

6.5.14. Address Operator

The unary (**&**) operator returns the address of its operand. If the operand has type "*type*", the result has type "pointer to *type*". If the operand is the result of a unary * operator, neither that operator nor the **&** operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an l-value. Similarly, if the operand is the result of a **[]** operator, neither the **&** operator nor the unary * that is implied by the **[]** is evaluated and the

result is as if the **&** operator were removed and the **[]** operator were changed to a + operator. Otherwise, the result is a pointer to the object designated by its operand [35].

6.5.15. Assignment Operator

Assignments of values to variable names are done with the assignment operator (=), like

lvalue = *expression*

The assignment operator stores the value of *expression* into *lvalue*. The *expression* and *lvalue* must have the same type, or the expression must have a type in Built-in Scalar Data Types, in which case an implicit conversion will be done on the expression before the assignment is done.

If *expression* is a scalar type and *lvalue* is a vector type, the scalar is converted to the element type used by the vector operand. The scalar type is then widened to a vector that has the same number of components as the vector operand. The operation is done component-wise resulting in the same size vector.

Any other desired type-conversions must be specified explicitly. L-values must be writable. Variables that are built-in types, entire structures or arrays, structure fields, l-values with the field selector (**.**) applied to select components or swizzles without repeated fields, l-values within parentheses, and l-values dereferenced with the array subscript operator (**[]**) are all l-values. Other binary or unary expressions, function names, swizzles with repeated fields, and constants cannot be l-values. The ternary operator (**?:**) is also not allowed as an l-value.

The order of evaluation of the operands is unspecified. If an attempt is made to modify the result of an assignment operator or to access it after the next sequence point, the behavior is undefined. Other assignment operators are the assignments add into (+=), subtract from (-=), multiply into (=), divide into (/=), modulus into (%=), left shift by (<<=), right shift by (>>=), and into (&=), inclusive or into (|=), and exclusive or into (^=).

The expression

lvalue op = *expression*

is equivalent to

lvalue = *lvalue op expression*

and the *lvalue* and *expression* must satisfy the requirements for both operator *op* and assignment (=).

Except for the `sizeof` operator, the `half` data type cannot be used with any of the operators described in this section.

6.6. Vector Operations

Vector operations are component-wise. Usually, when an operator operates on a vector, it is operating independently on each component of the vector, in a component-wise fashion.

For example,

```
float4 v, u;
float f;

v = u + f;
```

will be equivalent to

```
v.x = u.x + f;
v.y = u.y + f;
v.z = u.z + f;
v.w = u.w + f;
```

And

```
float4 v, u, w;

w = v + u;
```

will be equivalent to

```
w.x = v.x + u.x;
w.y = v.y + u.y;
w.z = v.z + u.z;
w.w = v.w + u.w;
```

and likewise for most operators and all integer and floating-point vector types.

6.7. Address Space Qualifiers

OpenCL C has a hierarchical memory architecture represented by address spaces, as defined in section 5 of the Embedded C Specification. It extends the C syntax to allow an address space name as a valid type qualifier (section 5.1.2 of the Embedded C Specification). OpenCL implements disjoint named address spaces with the spelling `__global`, `__local`, `__constant` and `__private`. The address space qualifier may be used in variable declarations to specify the region where objects are to be allocated. If the type of an object is qualified by an address space name, the object is allocated in the specified address space. Similarly, for pointers, the type pointed to can be qualified by an address space signaling the address space where the object pointed to is located.

The address space name spelling without the `__` prefix, i.e. `global`, `local`, `constant` and `private`, are valid and may be substituted for the corresponding address space names with the `__` prefix.

Examples:

```
// declares a pointer p in the global address space that
// points to an object in the global address space
__global int *__global p;

void foo (...)
{
    // declares an array of 4 floats in the private address space
    __private float x[4];
    ...
}
```

For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature macro, there is an additional unnamed generic address space.

Most of the restrictions from section 5.1.2 and section 5.3 of the Embedded C Specification apply in OpenCL C, e.g. address spaces can not be used with a return type, a function parameter, or a function type, and multiple address space qualifiers are not allowed. However, in OpenCL C it is allowed to qualify local variables with an address space qualifier.

Examples:

```
// OK.
int f() { ... }

// Error. Address space qualifier cannot be used with a non-pointer return type.
private int f() { ... }

// OK. Address space qualifier can be used with a pointer return type.
local int *f() { ... }

// Error. Multiple address spaces specified for a type.
private local int i;

// OK. The first address space qualifies the object pointed to and the second
// qualifies the pointer.
private int *local ptr;
```

The `__global`, `__constant`, `__local`, `__private`, `global`, `constant`, `local`, and `private` names are reserved for use as address space qualifiers and shall not be used otherwise. The `__generic` and `generic` names are reserved for future use.

The size of pointers to different address spaces may differ. It is not correct to assume that, for example, `sizeof(__global int *)` always equals `sizeof(__local int *)`.

6.7.1. `__global` (or `global`)

The `__global` or `global` address space name is used to refer to memory objects (buffer or image objects) allocated from the `global` memory pool.

A buffer memory object can be declared as a pointer to a scalar, vector or user-defined struct. This allows the kernel to read and/or write any location in the buffer.

The actual size of the memory object is determined when the memory object is allocated via appropriate API calls in the host code.

Examples:

```
global float4 *color; // An array of float4 elements

typedef struct {
    float a[3];
    int b[2];
} foo_t;

global foo_t *my_info; // An array of foo_t elements
```

As image objects are always allocated from the `global` address space, the `__global` or `global` qualifier should not be specified for image types. The elements of an image object cannot be directly accessed. Built-in functions to read from and write to an image object are provided.

Variables at program scope or `static` or `extern` variables inside functions can be declared in global address space if the `__opencl_c_program_scope_global_variables` feature is supported. These variables in the `global` address space have the same lifetime as the program, and their values persist between calls to any of the kernels in the program. They are not shared across devices and have distinct storage.

6.7.2. `__local` (or `local`)

The `__local` or `local` address space name is used to describe variables that are allocated in local memory and shared by all work-items in a work-group.

Examples:

```
kernel void my_func(...)
{
    local float a;     // A single float allocated
                       // in the local address space

    local float b[10]; // An array of 10 floats
                       // allocated in the local address space
}
```

Variables allocated in the `__local` address space inside a kernel function are allocated for each work-group executing the kernel and exist only for the lifetime of the work-group executing the kernel.

6.7.3. `__constant` (or `constant`)

The `__constant` or `constant` address space name is used to describe read-only variables that are accessible globally. They may be declared in program scope or in the outermost kernel scope or inside functions with a `static` or `extern` storage class specifier. Such variables can be accessed by all work-items or by different kernels during the program execution.

Each argument to a kernel that is a pointer to the `__constant` address space is counted separately towards the maximum number of such arguments, defined as the value of the `CL_DEVICE_MAX_CONSTANT_ARGS` device query.

It is illegal to write to a variable in the constant address space and will result in a compilation error.

Example:

```
constant int a = 3; // int allocated in the constant address space
kernel void k1(global int *buf)
{
    buf[a] = ...;   // OK. All work items access element with index 3.
}
kernel void k2(global int *buf)
{
    *buf = a;       // OK. All work items store value 3.
    a = 42;         // Error. a is in constant memory.
}
```

Implementations are not required to aggregate these declarations into the fewest number of constant arguments. This behavior is implementation-defined.

Thus portable code must conservatively assume that each variable declared inside a function or in program scope allocated in the `__constant` address space counts as a separate constant argument.

6.7.4. `__private` (or `private`)

The private address space is a memory segment that can only be accessed by one work item. Variables that are not shareable among work items are allocated in the private address space, and it is the default address space for most variables, in particular variables with automatic storage duration.

Example:

```
kernel void foo(...)
{
    private int i;
```

```
}
```

6.7.5. The Generic Address Space

The generic address space requires support for either OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature. It can be used with pointer types and it represents a placeholder for any of the named address spaces - `global`, `local` or `private`. It signals that a pointer points to an object in one of these concrete named address spaces. The exact address space resolution can occur dynamically during the kernel execution.

```
kernel void foo(int a)
{
    private int b;
    local int c;
    int* p =  a ? &b : &c; // p points to the local or private address space.
}
```

6.7.6. Usage for Declaration Scopes and Variable Types

This section describes use of address space qualifiers with respect to declaration scopes or variable types.

Local variables inside functions can be qualified by the private address space qualifier.

Variables declared in the outermost compound statement inside the body of the kernel function can be qualified by the local or constant address spaces.

Examples:

```
kernel void my_func(...)
{
    private float a;    // OK.
    local float b;      // OK.

    if (...)
    {
        // Example of variable in __local address space but not
        // declared at __kernel function scope.
        local float c;  // Error.
    }
}
```

Program scope variables or variables with a `extern` or `static` storage class specifier:

- Must be qualified by `__constant` prior to OpenCL C 2.0, or for OpenCL C 3.0 or newer without `__opencl_c_program_scope_global_variables` feature.
- Can be qualified by either `__constant` or `__global` for OpenCL C 2.0, or OpenCL C 3.0 or newer

with `__opencl_c_program_scope_global_variables` feature.

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_program_scope_global_variables feature macro.

constant int foo;       // OK.
global int baz;         // OK.
global uchar buf[512];  // OK.

static global int bat;  // OK. Internal linkage.

extern constant int foo;  // OK.

void func(...)
{
    constant static int foo = 1; // OK.
    global extern int foo;       // OK.
}

global int *global ptr;           // OK.
constant int *global ptr = &baz;  // Error, baz is in the global address space.
global int *constant ptr = &baz;  // OK.
```

Kernel function arguments declared to be a pointer or an array of a type must point to one of the named address spaces `__global`, `__local` or `__constant`.

Examples:

```
 // OK.
kernel void my_kernel(global int *ptr)
{
    ...
}
 // Error, ptr must point to the global, local, or constant address space.
kernel void my_kernel(int *ptr)
{
    ...
}
```

6.7.7. Initialization

Program scope and `static` variables in the `__global` address space are zero initialized by default. A constant expression may be given as an initializer.

Variables allocated in the `__local` address space inside a kernel function cannot be initialized.

Variables allocated in the `__constant` address space are required to be initialized and the values

used to initialize these variables must be a compile time constant.

Private address space objects are not initialized by default; any initializer is allowed to be given.

Examples:

```
global int a = 12;      // Initialization is allowed.
global int b;           // Zero initialized.
constant int c = 12;    // Initializer is a compile time constant.
constant int d;         // Error. No initializer provided.
kernel void my_func(...)
{
    local float e = 1;  // Error. Initializer is not allowed.

    local float f;
    f = 1;              // Allowed
    private int g;      // Uninitialized.
    constant int h = g; // Error. Initializer is not a constant expression.
}
```

6.7.8. Inference

Address space qualifiers are not required in many cases. If they are not specified explicitly the default address space will be inferred depending on the declaration scope and the object type.

There is no syntax to provide address space in the source for some situations, therefore only the default address space is applicable.

For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_program_scope_global_variables` feature, the address space for a variable at program scope or a `static` or `extern` variable inside a function are inferred to be `__global`.

If the generic address space is supported, pointers that are declared without pointing to a named address space point to the generic address space.

All string literal storage shall be in the `__constant` address space.

For all other cases that are not listed above the address space is inferred to `__private`. This includes:

- All function arguments as well as return values are in the private address space.
- Pointers that are declared without pointing to a named address space point to the `__private` address space if the generic address space is not supported.
- Variables inside a function not declared with an address space qualifier are inferred to be in the private address space.

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_program_scope_global_variables feature macro.
```

```
int foo;                // Inferred to be in the global address space.

static int foo;         // Inferred to be in the global address space.

int *ptr;               // ptr is inferred to be in the global address space.
                        // ptr points to a location in (1) the generic address
                        // space for OpenCL C 2.0, or OpenCL C 3.0 or newer with
                        // the __opencl_c_generic_address_space feature, or
                        // in (2) the private address space otherwise.

int *global ptr;        // ptr is declared to be in the global address space.
                        // ptr points to a location in (1) the generic address
                        // space for OpenCL C 2.0, or OpenCL C 3.0 or newer with
                        // __opencl_c_generic_address_space feature, or
                        // in (2) the private address space otherwise.

constant int *ptr =
               "Hello"; // string literal is in constant address space.

void func(int param)    // param is allocated in the private address space.
{
    int foo;            // foo is allocated in the private address space.
    static int foo;     // foo is allocated in the global address space.
    int *ptr;           // ptr is allocated in the private address space.
                        // ptr points to a location in (1) the generic address
                        // space for OpenCL C 2.0, or OpenCL C 3.0 or newer with
                        // __opencl_c_generic_address_space feature, or
                        // in (2) the private address space otherwise.
    ...
}
```

Qualifiers must be explicitly specified for:

- Program scope variables or variables inside functions with a `static` or `extern` type specifier prior to OpenCL C 2.0, or for OpenCL C 3.0 or newer without `__opencl_c_program_scope_global_variables` feature.
- Pointers used as arguments to kernel functions (the address space pointed to must be specified explicitly).

Table 8. Address space behavior

Address Space	Supported Usage	Initialization	Inference
`__global`	Program scope variables, for OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_program_scope_global_variables` feature, `static` or `extern` local variables, for OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_program_scope_global_variables` feature, Pointers.	Optional constant initializers, 0-initialized by default.	Program scope variables, for OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_program_scope_global_variables` feature. `static` or `extern` local variables, for OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_program_scope_global_variables` feature.
`__private`	Local scope variables, Function arguments and return types, Pointers.	Optional initializers, otherwise no default initialization.	Local scope variables, Function arguments and return types, Pointers in which the address space they point to is not given explicitly, prior to OpenCL C 2.0, or for OpenCL C 3.0 or newer without the `__opencl_c_generic_address_space` feature.
`__constant`	Program scope variables, Kernel scope variables, String literals, Pointers.	Mandatory initialization with a compile time constant.	String literals.
`__local`	Kernel scope variables, Pointers.	Not supported.	Not supported.

Address Space	Supported Usage	Initialization	Inference
Generic	Pointers, for OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature	Not applicable.	Pointers in which the address space they point to is not given explicitly, prior to OpenCL C 2.0, or for OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature.

6.7.9. Address Space Conversions

OpenCL implements the address space nesting model for pointers from Embedded C, section 5.1.3 as follows:

- In OpenCL the named address spaces `__global`, `__local`, `__constant` and `__private` are disjoint.
- The named address spaces `__global`, `__local`, and `__private` are subsets of the unnamed generic address space.
- The unnamed generic address space does not overlap the named `__constant` address space; the named `__constant` address space is not in the generic address space.

The OpenCL definition of the generic address space is different than the definition in section 5 of the Embedded C Specification. In OpenCL C, no objects can be allocated in this address space. It can only be used with pointer types, where a pointer pointing to a location in the generic address space can be used for objects allocated in any of the concrete named address spaces `private`, `local`, or `global`.

Following section 5.3 of the Embedded C Specification, it is only allowed to convert pointers implicitly, i.e. in assignments, function parameters, operations, if the original pointer points to an object qualified by an address space enclosed into the address space pointed by the destination pointer.

In contrast to the Embedded C Specification, explicitly converting i.e. casting between pointers to non-overlapping address spaces is illegal in OpenCL.

Considering the above, the following applies to conversions of pointers pointing to different address spaces:

- A pointer that points to the `global`, `local` or `private` address space can be implicitly converted to a pointer to the unnamed generic address space but not vice-versa.
- Pointer casts can be used to cast a pointer that points to the `global`, `local` or `private` space to the unnamed generic address space and vice-versa.
- A pointer that points to the `constant` address space cannot be cast or implicitly converted to the generic address space.

Examples:

This is the canonical example. In this example, function `foo` is declared with an argument that is a pointer with the unnamed generic address space address space qualifier.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

void foo(int *a)
{
    *a = *a + 2;
}

kernel void k1(local int *a)
{
    ...
    foo(a);
    ...
}

kernel void k2(global int *a)
{
    ...
    foo(a);
    ...
}
```

In the example below, `var` is a pointer to the unnamed generic address space. A pointer to the `global` or `local` address space may be assigned to `var` depending on the result of a conditional expression.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

kernel void bar(global int *g, local int *l)
{
    int *var;

    if (is_even(get_global_id(0))
        var = g;
    else
        var = l;
    *var = 42;
    ...
}
```

In the example below, the same pointer to the unnamed generic address space is used to point to objects allocated in different named address spaces. A pointer to the unnamed generic address space may point to objects in the `global`, `local`, and `private` address spaces, but it is not legal for a pointer to the unnamed generic address to point to an object in the `constant` address space.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

int *ptr;
global int g;
ptr = &g; // legal

local int l;
ptr = &l; // legal

private int p;
ptr = &p; // legal

constant int c;
ptr = &c; // illegal
```

In the example below, pointers to named address spaces are assigned to a pointer to the unnamed generic address space. It is legal to assign a pointer to the `global`, `local`, and `private` address spaces to a pointer to the unnamed generic address space without an explicit cast. It is not legal to assign a pointer to the `constant` address space to a pointer to the unnamed generic address space. It is also not legal to assign a pointer to the unnamed generic address space to a pointer to a named address space without a cast.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

global int *gp;
local int *lp;
private int *pp;
constant int *cp;

int *p;
p = gp; // OK.
p = lp; // OK.
p = pp; // OK.
p = cp; // Error.

// it is illegal to convert from a generic pointer
// to an explicit address space pointer without a cast:
gp = p; // Error.
lp = p; // Error.
pp = p; // Error.
cp = p; // Error.
```

The example below illustrates the implicit conversion between named address spaces.

```
global int *gp;
local int *lp;
```

```
private int *pp;
constant int *cp;

// it is illegal to convert pointers pointing to different
// named address spaces.

gp = lp; // Error.
gp = pp; // Error.
gp = cp; // Error.

lp = gp; // Error.
lp = pp; // Error.
lp = cp; // Error.

pp = lp; // Error.
pp = gp; // Error.
pp = cp; // Error.

cp = lp; // Error.
cp = pp; // Error.
cp = gp; // Error.
```

The example below demonstrates explicit conversions for pointers pointing to different address spaces.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

global int *gp;
local int *lp;
private int *pp;
constant int *cp;

int *p;
gp = (global int *)lp;  // illegal to cast between named address spaces
p = (int *)gp;          // legal to cast from global to generic
gp = (global int*)p;    // legal to cast from generic to global
```

For nested pointers, implicit conversions between address spaces are disallowed. Explicitly casting between different address spaces in nested pointers is allowed but the use of such pointers can lead to incorrect behavior such as accessing invalid memory locations.

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

kernel void mykernel(...)
{
    // ll is a pointer to a pointer in the local address space,
```

```
    // which points to an integer in the local address space
    local int *local *ll;

    // gl is a pointer to a pointer in the local address space,
    // which points to an integer in the global address space
    global int *local *gl;

    // nl is a pointer to a pointer in the local address space,
    // which points to an integer via the unnamed generic address space
    int *local * nl;

    ll = gl;  // Error, cannot convert address spaces implicitly
              // for nested pointers.
    ll = nl;  // Error, cannot convert address spaces implicitly
              // for nested pointers.
    ll = (local int* local*)gl; // OK to convert explicitly,
                                // but uses of 'll' can result in
                                // in ill-formed program.
    ll = (local int* local*)nl; // OK to convert explicitly,
                                // but uses of 'll' can result in
                                // in ill-formed program.
}
```

Various clarifications and examples illustrating how changes to ISO/IEC 9899:1999 detailed in Embedded C, section 5.3 apply to OpenCL C with the generic address space.

Clause 6.2.5 - Types:

If address space qualifier on type T is omitted refer to Address Space Inference.

Clause 6.3.2.3 - Pointers

Conversions between disjoint address spaces are disallowed in OpenCL, refer to Address Space Conversions.

Clause 6.5.8 - Relational operators:

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

kernel void test1()
{
    global int arr[5] = { 0, 1, 2, 3, 4 };
    int *p = &arr[1];
    global int *q = &arr[3];

    // q implicitly converted to the generic address space
    // since the generic address space encloses the global
```

```
    // address space
    if (q >= p)
        printf("true\n");

    // q implicitly converted to the generic address space
    // since the generic address space encloses the global
    // address space
    if (p <= q)
        printf("true\n");
}
```

Clause 6.5.9 - Equality operators:

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

int *ptr = NULL;
local int lval = SOME_VAL;
local int *lptr = &lval;
global int gval = SOME_OTHER_VAL;
global int *gptr = &gval;

ptr = lptr;

if (ptr == gptr) // legal
{
    ...
}

if (ptr == lptr) // legal
{
    ...
}

if (lptr == gptr) // illegal, compiler error
{
    ...
}
```

Consider the following example:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

bool callee(int *p1, int *p2)
{
    if (p1 == p2)
```

```
        return true;
    return false;
}

void caller()
{
    global int *gptr = 0xdeadbeef;
    private int *pptr = 0xdeadbeef;

    // behavior of callee is undefined
    bool b = callee(gptr, pptr);
}
```

The behavior of callee is undefined as gptr and pptr are in different address spaces. The example above would have the same undefined behavior if the equality operator is replaced with a relational operator.

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

int *ptr = NULL;
local int *lptr = NULL;
global int *gptr = NULL;

if (ptr == NULL) // legal
{
    ...
}

if (ptr == lptr) // legal
{
    ...
}

if (lptr == gptr) // compile-time error
{
    ...
}

ptr = lptr; // legal

intptr l = (intptr_t)lptr;
if (l == 0) // legal
{
    ...
}

if (l == NULL) // legal
```

```
{
    ...
}
```

Clause 6.5.15 - Conditional operator:

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

kernel void test1()
{
    global int arr[5] = { 0, 1, 2, 3, 4 };
    int *p = &arr[1];
    global int *q = &arr[3];
    local int *r = NULL;
    int *val = NULL;

    // legal. 2nd and 3rd operands are in address spaces
    // that overlap
    val = (q >= p) ? q : p;

    // compiler error. 2nd and 3rd operands are in disjoint
    // address spaces
    val = (q >= p) ? q : r;
}
```

Clause 6.5.16.1 - Simple assignment:

Examples:

```
// Note: these examples assume OpenCL C 2.0 or the
// __opencl_c_generic_address_space feature support.

kernel void f()
{
    int *ptr;
    local int *lptr;
    global int *gptr;
    local int val = 55;

    ptr = &val;  // legal: implicit cast to generic, then assign
    lptr = ptr;  // illegal: no implicit cast from
                 // generic to local
    lptr = gptr; // illegal: no implicit cast from
                 // global to local
    ptr = gptr;  // legal: implicit cast from global to generic,
                 // then assign
```

```
}
```

Clause 6.7.3 - Type qualifiers

The type of an object with automatic storage duration are in private address space and therefore can be qualified with `private`/`__private`.

6.8. Access Qualifiers

Image objects specified as arguments to a kernel can be declared to be read-only or write-only.

For OpenCL C 2.0, or with the `__opencl_c_read_write_images` feature, image objects specified as arguments to a kernel can additionally be declared to be read-write.

The `__read_only` (or `read_only`) access qualifier specifies that the image object is only being read by a kernel or function. The `__write_only` (or `write_only`) access qualifier specifies that the image object is only being written to by a kernel or function. The `__read_write` (or `read_write`) access qualifier specifies that the image object may be both read from or written to by a kernel or function.

The default access qualifier is `read_only`, if no access qualifier is declared.

In the following example

```
kernel void
foo (read_only image2d_t imageA,
     write_only image2d_t imageB)
{
    ...
}
```

`imageA` is a read-only 2D image object, and `imageB` is a write-only 2D image object.

The sampler-less read image and write image built-ins can be used with image declared with the `__read_write` (or `read_write`) qualifier. Calls to built-ins that read from an image using a sampler for images declared with the `__read_write` (or `read_write`) qualifier will be a compilation error.

Pipe objects specified as arguments to a kernel also use these access qualifiers. See the detailed description on how these access qualifiers can be used with pipes.

The `__read_only`, `__write_only`, `__read_write`, `read_only`, `write_only` and `read_write` names are reserved for use as access qualifiers and shall not be used otherwise.

6.9. Function Qualifiers

6.9.1. `__kernel` (or `kernel`)

The `__kernel` (or `kernel`) qualifier declares a function to be a kernel that can be executed by an application on an OpenCL device(s). The following rules apply to functions that are declared with

this qualifier:

- It can be executed on the device only
- It can be called by the host
- It is just a regular function call if a `__kernel` function is called by another kernel function.

Kernel functions with variables declared inside the function with the `__local` or `local` qualifier can be called by the host using appropriate APIs such as **clEnqueueNDRangeKernel**.

The `__kernel` and `kernel` names are reserved for use as functions qualifiers and shall not be used otherwise.

6.9.2. Optional Attribute Qualifiers

The `__kernel` qualifier can be used with the keyword *attribute* to declare additional information about the kernel function as described below.

The optional `__attribute__((vec_type_hint(<type>)))` [36] is a hint to the compiler and is intended to be a representation of the computational *width* of the `__kernel`, and should serve as the basis for calculating processor bandwidth utilization when the compiler is looking to autovectorize the code. In the `__attribute__((vec_type_hint(<type>)))` qualifier <type> is one of the built-in vector types listed in Built-in Vector Data Types or the constituent scalar element types. If `vec_type_hint (<type>)` is not specified, the kernel is assumed to have the `__attribute__((vec_type_hint(int)))` qualifier.

For example, where the developer specified a width of `float4`, the compiler should assume that the computation usually uses up to 4 lanes of a `float` vector, and would decide to merge work-items or possibly even separate one work-item into many threads to better match the hardware capabilities. A conforming implementation is not required to autovectorize code, but shall support the hint. A compiler may autovectorize, even if no hint is provided. If an implementation merges N work-items into one thread, it is responsible for correctly handling cases where the number of `global` or `local` work-items in any dimension modulo N is not zero.

Examples:

```
// autovectorize assuming float4 as the
// basic computation width
__kernel __attribute__((vec_type_hint(float4)))
void foo( __global float4 *p ) { ... }

// autovectorize assuming double as the
// basic computation width
__kernel __attribute__((vec_type_hint(double)))
void foo( __global float4 *p ) { ... }

// autovectorize assuming int (default)
// as the basic computation width
```

```
__kernel
void foo( __global float4 *p ) { ... }
```

If for example, a `__kernel` function is declared with

```
__attribute__(( vec_type_hint (float4)))
```

(meaning that most operations in the `__kernel` function are explicitly vectorized using `float4`) and the kernel is running using Intel® Advanced Vector Instructions (Intel® AVX) which implements a 8-float-wide vector unit, the autovectorizer might choose to merge two work-items to one thread, running a second work-item in the high half of the 256-bit AVX register.

As another example, a Power4 machine has two scalar double-precision floating-point units with an 6-cycle deep pipe. An autovectorizer for the Power4 machine might choose to interleave six kernels declared with the `__attribute__(( vec_type_hint (double2)))` qualifier into one hardware thread, to ensure that there is always 12-way parallelism available to saturate the FPUs. It might also choose to merge 4 or 8 work-items (or some other number) if it concludes that these are better choices, due to resource utilization concerns or some preference for divisibility by 2.

The optional `__attribute__((work_group_size_hint(X, Y, Z)))` is a hint to the compiler and is intended to specify the work-group size that may be used i.e. value most likely to be specified by the *local_work_size* argument to **clEnqueueNDRangeKernel**. For example, the `__attribute__((work_group_size_hint(1, 1, 1)))` is a hint to the compiler that the kernel will most likely be executed with a work-group size of 1.

The optional `__attribute__((reqd_work_group_size(X, Y, Z)))` is the work-group size that must be used as the *local_work_size* argument to **clEnqueueNDRangeKernel**. This allows the compiler to optimize the generated code appropriately for this kernel.

If `Z` is one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 2 or 3. If `Y` and `Z` are one, the *work_dim* argument to **clEnqueueNDRangeKernel** can be 1, 2 or 3.

6.10. Storage-Class Specifiers

The `typedef` storage-class specifier is supported. The `extern` and `static` storage-class specifiers are supported but require support for OpenCL C 1.2 or newer. The `auto` and `register` storage-class specifiers are not supported.

The `extern` storage-class specifier can only be used for functions (kernel and non-kernel functions) and `global` variables declared in program scope or variables declared inside a function (kernel and non-kernel functions). The `static` storage-class specifier can only be used for non-kernel functions, `global` variables declared in program scope and variables inside a function declared in the `global` or `constant` address space.

Examples:

```
extern constant float4 noise_table[256];
static constant float4 color_table[256];
```

```
extern kernel void my_foo(image2d_t img);
extern void my_bar(global float *a);

kernel void my_func(image2d_t img, global float *a)
{
    extern constant float4 a;
    static constant float4 b = (float4)(1.0f); // OK.
    static float c;  // Error: No implicit address space
    global int hurl; // Error: Must be static
    ...
    my_foo(img);
    ...
    my_bar(a);
    ...
    while (1)
    {
        static global int inside; // OK.
        ...
    }
    ...
}
```

6.11. Restrictions

a. The use of pointers is somewhat restricted. The following rules apply:

   - Arguments to kernel functions declared in a program that are pointers must be declared with the `__global`, `__constant` or `__local` qualifier.
   - A pointer declared with the `__constant` qualifier can only be assigned to a pointer declared with the `__constant` qualifier respectively.
   - Pointers to functions are not allowed.
   - Arguments to kernel functions in a program cannot be declared as a pointer to a pointer(s). Variables inside a function or arguments to non-kernel functions in a program can be declared as a pointer to a pointer(s). This restriction only applies to OpenCL C 1.2 or below.

b. An image type (`image2d_t`, `image3d_t`, `image2d_array_t`, `image1d_t`, `image1d_buffer_t` or `image1d_array_t`) can only be used as the type of a function argument. An image function argument cannot be modified. Elements of an image can only be accessed using the built-in image read and write functions.

   An image type cannot be used to declare a variable, a structure or union field, an array of images, a pointer to an image, or the return type of a function. An image type cannot be used with the `__global`, `__private`, `__local` and `__constant` address space qualifiers.

   The sampler type (`sampler_t`) can only be used as the type of a function argument or a variable declared in the program scope or the outermost scope of a kernel function. The behavior of a sampler variable declared in a non-outermost scope of a kernel function is implementation-defined. A sampler argument or variable cannot be modified.

The sampler type cannot be used to declare a structure or union field, an array of samplers, a pointer to a sampler, or the return type of a function. The sampler type cannot be used with the `__local` and `__global` address space qualifiers.

c. Bit-field struct members are currently not supported.

d. Variable length arrays and structures with flexible (or unsized) arrays are not supported.

e. Variadic functions are not supported, with the exception of `printf` and `enqueue_kernel`.

f. Variadic macros are not supported. This restriction only applies to OpenCL C 2.0 or below.

g. If a list of parameters in a function declaration is empty, the function takes no arguments. This is due to the above restriction on variadic functions.

h. Unless defined in the OpenCL specification, the library functions, macros, types, and constants defined in the C99 standard headers `assert.h`, `ctype.h`, `complex.h`, `errno.h`, `fenv.h`, `float.h`, `inttypes.h`, `limits.h`, `locale.h`, `setjmp.h`, `signal.h`, `stdarg.h`, `stdio.h`, `stdlib.h`, `string.h`, `tgmath.h`, `time.h`, `wchar.h` and `wctype.h` are not available and cannot be included by a program.

i. The `auto` and `register` storage-class specifiers are not supported.

j. Predefined identifiers are not supported. This restriction only applies to OpenCL C 1.1 or below.

k. Recursion is not supported.

l. The return type of a kernel function must be `void`.

m. Arguments to kernel functions in a program cannot be declared with the built-in scalar types `bool`, `size_t`, `ptrdiff_t`, `intptr_t`, and `uintptr_t` or a struct and/or union that contain fields declared to be one of these built-in scalar types.

n. `half` is not supported as `half` can be used as a storage format [37] only and is not a data type on which floating-point arithmetic can be performed.

o. Whether or not irreducible control flow is illegal is implementation defined.

p. The following restriction only applies to OpenCL C 1.0, and only if the `cl_khr_byte_addressable_store` extension macro is not supported:
Built-in types that are less than 32-bits in size, i.e. `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half`, have the following restriction:

- Writes to a pointer (or arrays) of type `char`, `uchar`, `char2`, `uchar2`, `short`, `ushort`, and `half` or to elements of a struct that are of type `char`, `uchar`, `char2`, `uchar2`, `short` and `ushort` are not supported. Refer to *section 9.9* for additional information.

  The kernel example below shows what memory operations are not supported on built-in types less than 32-bits in size.

```
kernel void
do_proc (__global char *pA, short b,
         __global short *pB)
{
    char x[100];
    __private char *px = x;
    int id = (int)get_global_id(0);
    short f;
```

```
    f = pB[id] + b; // is allowed
    px[1] = pA[1]; // error. px cannot be written.
    pB[id] = b; // error. pB cannot be written
}
```

q. The type qualifiers `const`, `restrict` and `volatile` as defined by the C99 specification are supported. These qualifiers cannot be used with `image2d_t`, `image3d_t`, `image2d_array_t`, `image2d_depth_t`, `image2d_array_depth_t`, `image1d_t`, `image1d_buffer_t` and `image1d_array_t` types. Types other than pointer types shall not use the `restrict` qualifier.

r. The event type (`event_t`) cannot be used as the type of a kernel function argument. The event type cannot be used to declare a program scope variable. The event type cannot be used to declare a structure or union field. The event type cannot be used with the `__local`, `__constant` and `__global` address space qualifiers.

s. The `clk_event_t`, `ndrange_t` and `reserve_id_t` types cannot be used as arguments to kernel functions that get enqueued from the host. The `clk_event_t` and `reserve_id_t` types cannot be declared in program scope.

t. Kernels enqueued by the host must continue to have their arguments that are a pointer to a type declared to point to a named address space.

u. A function in an OpenCL program cannot be called `main`.

v. Implicit function declaration is not supported.

w. Program scope variables can be defined with any valid OpenCL C data type except for those in Other Built-in Data Types. Such program scope variables may be of any user-defined type, or a pointer to a user-defined type.

   In the presence of shared virtual memory, these pointers or pointer members should work as expected as long as they are shared virtual memory pointers and the referenced storage has been mapped appropriately. Program scope variables can be declared with `__constant` address space qualifiers or if `__opencl_c_program_scope_global_variables` feature is supported with `__global` address space qualifier.

1. The following restriction only applies if the `cl_khr_initialize_memory` extension is supported:
   If the context is created with `CL_CONTEXT_MEMORY_INITIALIZE_KHR`, appropriate memory locations as specified by the bit-field are initialized with zeroes, prior to the start of execution of any kernel. The driver chooses when, prior to kernel execution, the initialization of local and/or private memory is performed. The only requirement is there should be no values set from outside the context, which can be read during a kernel execution.

6.12. Preprocessor Directives and Macros

The preprocessing directives defined by the C99 specification are supported.

The #**pragma** directive is described as:

#**pragma** *pp-tokens*$_{opt}$ *new-line*

A **#pragma** directive where the preprocessing token `OPENCL` (used instead of `STDC`) does not immediately follow **#pragma** in the directive (prior to any macro replacement) causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any such **#pragma** that is not recognized by the implementation is ignored. If the preprocessing token `OPENCL` does immediately follow **#pragma** in the directive (prior to any macro replacement), then no macro replacement is performed on the directive, and the directive shall have one of the following forms whose meanings are described elsewhere:

```
// on-off-switch is one of ON, OFF, or DEFAULT
#pragma OPENCL FP_CONTRACT on-off-switch
```

The following predefined macro names are available.

`__FILE__`

The presumed name of the current source file (a character string literal).

`__LINE__`

The presumed line number (within the current source file) of the current source line (an integer constant).

`__OPENCL_VERSION__`

For OpenCL devices with OpenCL version less than or equal to OpenCL 2.0, substitutes an integer value reflecting the OpenCL version supported by the device. This predefined macro is deprecated by OpenCL 2.1. For OpenCL devices with OpenCL version greater than OpenCL 2.0, it must be defined but may substitute any implementation-defined integer value greater than 200, reflecting OpenCL 2.0. [38]

`CL_VERSION_1_0`

Substitutes the integer 100 reflecting the OpenCL 1.0 version. Requires support for OpenCL C 1.1 or newer.

`CL_VERSION_1_1`

Substitutes the integer 110 reflecting the OpenCL 1.1 version. Requires support for OpenCL C 1.1 or newer.

`CL_VERSION_1_2`

Substitutes the integer 120 reflecting the OpenCL 1.2 version. Requires support for OpenCL C 1.2 or newer.

`CL_VERSION_2_0`

Substitutes the integer 200 reflecting the OpenCL 2.0 version. Requires support for OpenCL C 2.0 or newer.

`CL_VERSION_3_0`

Substitutes the integer 300 reflecting the OpenCL 3.0 version. Requires support for OpenCL C 3.0 or newer.

`CL_VERSION_3_1`

Substitutes the integer 301 reflecting the OpenCL 3.1 version. Requires support for OpenCL C 3.1 or newer.

`__OPENCL_C_VERSION__`

Substitutes an integer reflecting the OpenCL C version specified by the `-cl-std` build option (see OpenCL Specification) to **clBuildProgram** or **clCompileProgram**. If the `-cl-std` build option is not specified, the highest OpenCL C 1.x language version supported by each device is used as the version of OpenCL C when compiling the program for each device. Requires support for OpenCL C 1.2 or newer.

`__ROUNDING_MODE__`

Used to determine the current rounding mode and is set to rte. Only affects the rounding mode of conversions to a float type. Deprecated by OpenCL C 1.1, along with the `cl_khr_select_fprounding_mode` extension.

`__ENDIAN_LITTLE__`

Used to determine if the OpenCL device is a little endian architecture or a big endian architecture (an integer constant of 1 if device is little endian and is undefined otherwise). Also refer to the value of the `CL_DEVICE_ENDIAN_LITTLE` device query.

`__kernel_exec(X, typen)` **(and** `kernel_exec(X, typen)`**)**

is defined as:

```
__kernel __attribute__((work_group_size_hint(X, 1, 1))) \
    __attribute__((vec_type_hint(typen)))
```

`__IMAGE_SUPPORT__`

Used to determine if the OpenCL device supports images. This is an integer constant of 1 if images are supported and is undefined otherwise. Also refer to the value of the `CL_DEVICE_IMAGE_SUPPORT` device query and the `__opencl_c_images` feature.

`__FAST_RELAXED_MATH__`

Used to determine if the `-cl-fast-relaxed-math` optimization option is specified in build options given to **clBuildProgram** or **clCompileProgram**. This is an integer constant of 1 if the `-cl-fast-relaxed-math` build option is specified and is undefined otherwise.

The `NULL` macro expands to a null pointer constant. An integer constant expression with the value 0, or such an expression cast to type `void *` is called a *null pointer constant*. Requires support for OpenCL C 2.0 or newer.

The macro names defined by the C99 specification but not currently supported by OpenCL are reserved for future use.

The predefined identifier `__func__` is available. Requires support for OpenCL C 1.2 or newer.

In OpenCL C 3.0 or newer there are a number of optional predefined macros indicating optional language features. Such macros are listed in the optional features in OpenCL C 3.0 or newer table.

6.12.1. Preprocessor Directives for Optional Extensions

The **#pragma OPENCL EXTENSION** directive controls the behavior of the OpenCL compiler with respect to extensions. The **#pragma OPENCL EXTENSION** directive is defined as:

```
#pragma OPENCL EXTENSION <extension_name> : <behavior>
#pragma OPENCL EXTENSION all : <behavior>
```

where *extension_name* is the name of the extension. The token **all** means that the behavior applies to all extensions supported by the compiler. The *behavior* can be set to one of the following values given by the table below.

behavior	**Description**
enable	Behave as specified by the extension *extension_name*. Report an error on the `#pragma OPENCL EXTENSION` if the *extension_name* is not supported, or if **all** is specified.
disable	Behave (including issuing errors and warnings) as if the extension *extension_name* is not part of the language definition. If **all** is specified, then behavior must revert back to that of the non-extended core version of the language being compiled to. Warn on the `#pragma OPENCL EXTENSION` if the extension *extension_name* is not supported.

The `#pragma OPENCL EXTENSION` directive is a simple, low-level mechanism to set the behavior for each extension. It does not define policies such as which combinations are appropriate; those must be defined elsewhere. The order of directives matter in setting the behavior for each extension. Directives that occur later override those seen earlier. The **all** variant sets the behavior for all extensions, overriding all previously issued extension directives, but only if the *behavior* is set to **disable**.

The initial state of the compiler is as if the directive

```
#pragma OPENCL EXTENSION all : disable
```

was issued, telling the compiler that all error and warning reporting must be done according to this specification, ignoring any extensions.

Every extension which affects the OpenCL language semantics, syntax or adds built-in functions to the language must create a preprocessor `#define` that matches the extension name string. This `#define` would be available in the language if and only if the extension is supported on a given implementation.

For example, compilers that support the `cl_khr_3d_image_writes` extension will add a preprocessor

`#define` called `cl_khr_3d_image_writes`. A kernel can now use this preprocessor `#define` to do something like:

```
#ifdef cl_khr_3d_image_writes
    // do something using the extension
#else
    // do something else or #error!
#endif
```

6.13. Attribute Qualifiers

This section describes the syntax with which `__attribute__` may be used, and the constructs to which attribute specifiers bind.

An attribute specifier is of the form

`__attribute__ ((_attribute-list_))`.

An attribute list is defined as:

attribute-list **:**

attribute$_{opt}$
attribute-list , *attribute*$_{opt}$

attribute **:**

attribute-token attribute-argument-clause$_{opt}$

attribute-token **:**

identifier

attribute-argument-clause **:**

(*attribute-argument-list*)

attribute-argument-list **:**

attribute-argument
attribute-argument-list , *attribute-argument*

attribute-argument **:**

assignment-expression

This syntax is taken directly from GCC but unlike GCC, which allows attributes to be applied only to functions, types, and variables, OpenCL attributes can be associated with:

- types;
- functions;
- variables;
- blocks; and

- control-flow statements.

In general, the rules for how an attribute binds, for a given context, are non-trivial and the reader is pointed to GCC's documentation and Maurer and Wong's paper [See 16. and 17. in *section 11* - **References**] for the details.

6.13.1. Specifying Attributes of Types

The keyword `__attribute__` allows you to specify special attributes of enum, struct and union types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Two attributes are currently defined for types: aligned, and packed.

You may specify type attributes in an enum, struct or union type declaration or definition, or for other types in a `typedef` declaration.

For an enum, struct or union type, you may specify attributes either between the enum, struct or union tag and the name of the type, or just past the closing curly brace of the *definition*. The former syntax is preferred.

`aligned (alignment)`

This attribute specifies a minimum alignment (in bytes) for variables of the specified type.

For example, the declarations:

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

force the compiler to ensure (as far as it can) that each variable whose type is `struct S` or `more_aligned_int` will be allocated and aligned *at least* on a 8-byte boundary.

Note that the alignment of any given struct or union type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the struct or union in question and must also be a power of two. This means that you *can* effectively adjust the alignment of a struct or union type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire struct or union type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given struct or union type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. In the example above, the size of each `short`

is 2 bytes, and therefore the size of the entire `struct S` type is 6 bytes. The smallest power of two which is greater than or equal to that is 8, so the compiler sets the alignment for the entire `struct S` type to 8 bytes.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

`packed`

> This attribute, attached to struct or union type definition, specifies that each member of the structure or union is placed to minimize the memory required. When attached to an enum definition, it indicates that the smallest integral type should be used.

Specifying this attribute for struct and union types is equivalent to specifying the packed attribute on each of the structure or union members.

In the following example, the members of `my_packed_struct` are packed closely together, but the internal layout of its `s` member is not packed. To do that, struct `my_unpacked_struct` would need to be packed, too.

```
struct my_unpacked_struct
{
    char c;
    int i;
};

struct __attribute__ ((packed)) my_packed_struct
{
    char c;
    int i;
    struct my_unpacked_struct s;
};
```

You may only specify this attribute on the definition of a enum, struct or union, not on a `typedef` which does not also define the enumerated type, structure or union.

6.13.2. Specifying Attributes of Functions

See Function Qualifiers for the function attribute qualifiers currently supported.

6.13.3. Specifying Attributes of Variables

The keyword `__attribute__` allows you to specify special attributes of variables or structure fields.

This keyword is followed by an attribute specification inside double parentheses. The following attribute qualifiers are currently defined:

`aligned (alignment)`

This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable `x` on a 16-byte boundary. The alignment value specified must be a power of two.

You can also specify the alignment of structure fields. For example, to create a double-word aligned `int` pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member that forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the OpenCL compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target device you are compiling for.

When used on a struct, or struct member, the aligned attribute can only increase the alignment; in order to decrease it, the packed attribute must be specified as well. When used as part of a `typedef`, the aligned attribute can both increase and decrease alignment, and specifying the packed attribute will generate a warning.

Note that the effectiveness of aligned attributes may be limited by inherent limitations of the OpenCL device and compiler. For some devices, the OpenCL compiler may only be able to arrange for variables to be aligned up to a certain maximum alignment. If the OpenCL compiler is only able to align variables up to a maximum of 8 byte alignment, then specifying `aligned(16)` in an `__attribute__` will still only provide you with 8 byte alignment. See your platform-specific documentation for further information.

`packed`

The packed attribute specifies that a variable or structure field should have the smallest possible alignment — one byte for a variable, unless you specify a larger value with the aligned attribute.

Here is a structure in which the field `x` is packed, so that it immediately follows a:

```
struct foo
{
    char a;
    int x[2] __attribute__ ((packed));
};
```

An attribute list placed at the beginning of a user-defined type applies to the variable of that type and not the type, while attributes following the type body apply to the type.

For example:

```
/* a has alignment of 128 */
__attribute__((aligned(128))) struct A {int i;} a;

/* b has alignment of 16 */
__attribute__((aligned(16))) struct B {double d;}
__attribute__((aligned(32))) b ;

struct A a1; /* a1 has alignment of 4 */

struct B b1; /* b1 has alignment of 32 */
```

`endian (endiantype)`

The endian attribute determines the byte ordering of a variable. *endiantype* can be set to `host` indicating the variable uses the endianness of the host processor or can be set to `device` indicating the variable uses the endianness of the device on which the kernel will be executed. The default is `device`.

For example:

```
global float4 *p __attribute__ ((endian(host)));
```

specifies that data stored in memory pointed to by p will be in the host endian format.

The endian attribute can only be applied to pointer types that are in the `global` or `constant` address space. The endian attribute cannot be used for variables that are not a pointer type. The endian attribute value for both pointers must be the same when one pointer is assigned to another.

`nosvm`

The `nosvm` attribute can be used with a pointer variable to inform the compiler that the pointer does not refer to a shared virtual memory region. Requires support for OpenCL C 2.0 or newer.

The `nosvm` attribute is deprecated, and the compiler can ignore it.

6.13.4. Specifying Attributes of Blocks and Control-Flow-Statements

For basic blocks and control-flow-statements the attribute is placed before the structure in question, for example:

```
__attribute__((attr1)) {...}

for __attribute__((attr2)) (...) __attribute__((attr3)) {...}
```

Here `attr1` applies to the block in braces and `attr2` and `attr3` apply to the loop's control construct and body, respectively.

No attribute qualifiers for blocks and control-flow-statements are currently defined.

6.13.5. Specifying Attribute for Unrolling Loops

The functionality described in this section requires support for OpenCL C 2.0 or newer.

The `__attribute__((opencl_unroll_hint))` and `__attribute__((opencl_unroll_hint(n)))` attribute qualifiers can be used to specify that a loop (for, while and do loops) can be unrolled. This attribute qualifier can be used to specify full unrolling or partial unrolling by a specified amount. This is a compiler hint and the compiler may ignore this directive.

n is the loop unrolling factor and must be a positive integral compile time constant expression. An unroll factor of 1 disables unrolling. If n is not specified, the compiler determines the unrolling factor for the loop.

The `__attribute__((opencl_unroll_hint(n)))` attribute qualifier must appear immediately before the loop to be affected.

Examples:

```
__attribute__((opencl_unroll_hint(2)))
while (*s != 0)
    *p++ = *s++;
```

The tells the compiler to unroll the above while loop by a factor of 2.

```
__attribute__((opencl_unroll_hint))
for (int i=0; i<2; i++)
{
    ...
}
```

In the example above, the compiler will determine how much to unroll the loop.

```
__attribute__((opencl_unroll_hint(1)))
for (int i=0; i<32; i++)
{
    ...
}
```

The above is an example where the loop should not be unrolled.

Below are some examples of invalid usage of `__attribute__((opencl_unroll_hint(n)))`.

```
__attribute__((opencl_unroll_hint(-1)))
while (...)
{
    ...
}
```

The above example is an invalid usage of the loop unroll factor as the loop unroll factor is negative.

```
__attribute__((opencl_unroll_hint))
if (...)
{
    ...
}
```

The above example is invalid because the unroll attribute qualifier is used on a non-loop construct

```
kernel void
my_kernel( ... )
{
    int x;
    __attribute__((opencl_unroll_hint(x))
    for (int i=0; i<x; i++)
    {
        ...
    }
}
```

The above example is invalid because the loop unroll factor is not a compile-time constant expression.

6.14. Blocks

The functionality described in this section requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_device_enqueue` feature.

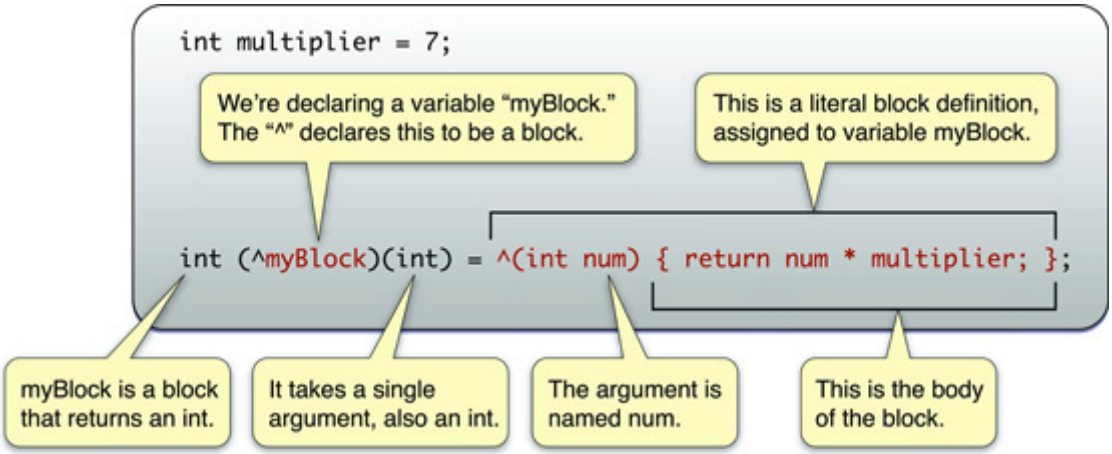
This section describes the clang block syntax [39].

Like function types, the Block type is a pair consisting of a result value type and a list of parameter types very similar to a function type. Blocks are intended to be used much like functions with the key distinction being that in addition to executable code they also contain various variable bindings to automatic (stack) or `global` memory.

6.14.1. Declaring and Using a Block

You use the ^ operator to declare a Block variable and to indicate the beginning of a Block literal. The body of the Block itself is contained within {}, as shown in this example (as usual with C, ; indicates the end of the statement):

The example is explained in the following illustration:



Notice that the Block is able to make use of variables from the same scope in which it was defined.

If you declare a Block as a variable, you can then use it just as you would a function:

```
int multiplier = 7;

int (^myBlock)(int) = ^(int num) {
    return num * multiplier;
};

printf("%d\n", myBlock(3));
// prints 21
```

6.14.2. Declaring a Block Reference

Block variables hold references to Blocks. You declare them using syntax similar to that you use to declare a pointer to a function, except that you use ^ instead of *. The Block type fully interoperates with the rest of the C type system. The following are valid Block variable declarations:

```
void (^blockReturningVoidWithVoidArgument)(void);
```

```
int (^blockReturningIntWithIntAndCharArguments)(int, char);
```

A Block that takes no arguments must specify `void` in the argument list. A Block reference may not be dereferenced via the pointer dereference operation *, and thus a Block's size may not be computed at compile time.

Blocks are designed to be fully type safe by giving the compiler a full set of metadata to use to validate use of Blocks, parameters passed to blocks, and assignment of the return value.

You can also create types for Blocks — doing so is generally considered to be best practice when you use a block with a given signature in multiple places:

```
typedef float (^MyBlockType)(float, float);

MyBlockType myFirstBlock = // ...;
MyBlockType mySecondBlock = // ...;
```

6.14.3. Block Literal Expressions

A Block literal expression produces a reference to a Block. It is introduced by the use of the ^ token as a unary operator.

Block_literal_expression :

^ *block_decl compound_statement_body*

***block_decl* :**

empty
parameter_list
type_expression

where *type_expression* is extended to allow ^ as a Block reference where * is allowed as a function reference.

The following Block literal:

```
^ void (void) { printf("hello world**\n**"); }
```

produces a reference to a Block with no arguments with no return value.

The return type is optional and is inferred from the return statements. If the return statements return a value, they all must return a value of the same type. If there is no value returned the inferred type of the Block is `void`; otherwise it is the type of the return statement value. If the return type is omitted and the argument list is `( void )`, the `( void )` argument list may also be omitted.

So:

```
^ ( void ) { printf("hello world**\n**"); }
```

and:

```
^ { printf("hello world**\n**"); }
```

are exactly equivalent constructs for the same expression.

The compound statement body establishes a new lexical scope within that of its parent. Variables used within the scope of the compound statement are bound to the Block in the normal manner with the exception of those in automatic (stack) storage. Thus one may access functions and global variables as one would expect, as well as `static` local variables.

Local automatic (stack) variables referenced within the compound statement of a Block are imported and captured by the Block as const copies. The capture (binding) is performed at the time of the Block literal expression evaluation.

The compiler is not required to capture a variable if it can prove that no references to the variable will actually be evaluated.

The lifetime of variables declared in a Block is that of a function..

Block literal expressions may occur within Block literal expressions (nested) and all variables captured by any nested blocks are implicitly also captured in the scopes of their enclosing Blocks.

A Block literal expression may be used as the initialization value for Block variables at global or local `static` scope.

You can also declare a Block as a global literal in program scope.

```
int GlobalInt = 0;

int (^getGlobalInt)(void) = ^{ return GlobalInt; };
```

6.14.4. Control Flow

The compound statement of a Block is treated much like a function body with respect to control flow in that continue, break and goto do not escape the Block.

6.14.5. Restrictions

The following Blocks features are currently not supported in OpenCL C.

- The `__block` storage type.
- The **Block_copy**() and **Block_release**() functions that copy and release Blocks.
- Blocks with variadic arguments.

- Arrays of Blocks.
- Blocks as structures and union members.

Block literals are assumed to allocate memory at the point of definition and to be destroyed at the end of the same scope. To support these behaviors, additional restrictions [40] in addition to the above feature restrictions are:

- Block variables must be defined and used in a way that allows them to be statically determinable at build or "link to executable" time. In particular:
  - Block variables assigned in one scope must be used only with the same or any nested scope.
  - The `extern` storage-class specified cannot be used with program scope block variables.
  - Block variable declarations are implicitly qualified with const. Therefore all block variables must be initialized at declaration time and may not be reassigned.
  - A block cannot be a return value or a parameter of a function.
  - Blocks cannot be used as expressions of the ternary selection operator (**?:**).
- The unary operators (*****) and (**&**) cannot be used with a Block.
- Pointers to Blocks are not allowed.
- A Block cannot capture another Block variable declared in the outer scope (Example 4).
- Block capture semantics follows regular C argument passing convention, i.e. arrays are captured by reference (decayed to pointers) and structs are captured by value (Example 5).

Some examples that describe legal and illegal issue of Blocks in OpenCL C are described below.

Example 1:

```
void foo(int *x, int (^bar)(int, int))
{
    *x = bar(*x, *x);
}

kernel
void k(global int *x, global int *z)
{
    if (some expression)
        foo(x, ^int(int x, int y){return x+y+*z;}); // legal
    else
        foo(x, ^int(int x, int y){return (x*y)-*z;}); // legal
}
```

Example 2:

```
kernel
void k(global int *x, global int *z)
{
    int ^(tmp)(int, int);
```

```
    if (some expression)
    {
        tmp = ^int(int x, int y){return x+y+*z;}); // illegal
    }
    *x = foo(x, tmp);
}
```

Example 3:

```
int GlobalInt = 0;
int (^getGlobalInt)(void) = ^{ return GlobalInt; }; // legal
int (^getAnotherGlobalInt)(void);                   // illegal
extern int (^getExternGlobalInt)(void);             // illegal

void foo()

{
    ...
    getGlobalInt = ^{ return 0; }; // illegal - cannot assign to
                                   // a global block variable
    ...
}
```

Example 4:

```
void (^bl0)(void) = ^{
    ...
};

kernel void k()
{
    void(^bl1)(void) = ^{
        ...
    };

    void(^bl2)(void) = ^{
        bl0(); // legal because bl0 is a global
               // variable available in this scope
        bl1(); // illegal because bl1 would have to be captured
    };
}
```

Example 5:

```
struct v {
    int arr[2];
} s = {0, 1};
```

```
void (^bl1)() = ^(){printf("%d\n", s.arr[1]);};
// array content copied into captured struct location

int arr[2] = {0, 1};
void (^bl2)() = ^(){printf("%d\n", arr[1]);};
// array decayed to pointer while captured

s.arr[1] = arr[1] = 8;

bl1(); // prints - 1
bl2(); // prints - 8
```

6.15. Built-in Functions

The OpenCL C programming language provides a rich set of built-in functions for scalar and vector operations. Many of these functions are similar to the function names provided in common C libraries but they support scalar and vector argument types. Applications should use the built-in functions wherever possible instead of writing their own version.

User defined OpenCL C functions behave per C standard rules for functions as defined in section 6.9.1 of the C99 Specification. On entry to the function, the size of each variably modified parameter is evaluated and the value of each argument expression is converted to the type of the corresponding parameter as per the usual arithmetic conversion rules. Built-in functions described in this section behave similarly, except that in order to avoid ambiguity between multiple forms of the same built-in function, implicit scalar widening shall not occur. Note that some built-in functions described in this section do have forms that operate on mixed scalar and vector types, however.

6.15.1. Work-Item Functions

The following table describes the list of built-in work-item functions that can be used to query the number of dimensions, the global and local work size specified to **clEnqueueNDRangeKernel**, and the global and local identifier of each work-item when this kernel is being executed on a device.

Table 9. Built-in Work-Item Functions

Function	Description
uint **get_work_dim**()	Returns the number of dimensions in use. This is the value given to the *work_dim* argument specified in **clEnqueueNDRangeKernel**.

Function	Description
size_t **get_global_size**(uint *dimindx*)	Returns the number of global work-items specified for dimension identified by *dimindx*. This value is given by the *global_work_size* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_global_size**() returns 1.
size_t **get_global_id**(uint *dimindx*)	Returns the unique global work-item ID value for dimension identified by *dimindx*. The global work-item ID specifies the work-item ID based on the number of global work-items specified to execute the kernel. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_global_id**() returns 0.
size_t **get_local_size**(uint *dimindx*)	Returns the number of local work-items specified in dimension identified by *dimindx*. This value is at most the value given by the *local_work_size* argument to **clEnqueueNDRangeKernel** if *local_work_size* is not `NULL`; otherwise the OpenCL implementation chooses an appropriate *local_work_size* value which is returned by this function. If the kernel is executed with a non-uniform work-group size [41], calls to this built-in from some work-groups may return different values than calls to this built-in from other work-groups. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_local_size**() returns 1.

Function	Description
size_t **get_enqueued_local_size**(uint *dimindx*)	Returns the same value as that returned by **get_local_size**(*dimindx*) if the kernel is executed with a uniform work-group size. If the kernel is executed with a non-uniform work-group size, returns the number of local work-items in each of the work-groups that make up the uniform region of the global range in the dimension identified by *dimindx*. If the *local_work_size* argument to **clEnqueueNDRangeKernel** is not `NULL`, this value will match the value specified in *local_work_size*[*dimindx*]. If *local_work_size* is `NULL`, this value will match the local size that the implementation determined would be most efficient at implementing the uniform region of the global range. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_enqueued_local_size**() returns 1. Requires support for OpenCL C 2.0 or newer.
size_t **get_local_id**(uint *dimindx*)	Returns the unique local work-item ID, i.e. a work-item within a specific work-group for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_local_id**() returns 0.
size_t **get_num_groups**(uint *dimindx*)	Returns the number of work-groups that will execute a kernel for dimension identified by *dimindx*. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values of *dimindx*, **get_num_groups**() returns 1.
size_t **get_group_id**(uint *dimindx*)	**get_group_id** returns the work-group ID which is a number from 0 .. **get_num_groups**(*dimindx*) - 1. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values, **get_group_id**() returns 0.

Function	Description
size_t **get_global_offset**(uint *dimindx*)	**get_global_offset** returns the offset values specified in *global_work_offset* argument to **clEnqueueNDRangeKernel**. Valid values of *dimindx* are 0 to **get_work_dim**() - 1. For other values, **get_global_offset**() returns 0. Requires support for OpenCL C 1.1 or newer.
size_t **get_global_linear_id**()	Returns the work-items 1-dimensional global ID. For 1D work-groups, it is computed as **get_global_id**(0) - **get_global_offset**(0). For 2D work-groups, it is computed as (**get_global_id**(1) - **get_global_offset**(1)) * **get_global_size**(0) + (**get_global_id**(0) - **get_global_offset**(0)). For 3D work-groups, it is computed as ((**get_global_id**(2) - **get_global_offset**(2)) * **get_global_size**(1) * **get_global_size**(0)) + ((**get_global_id**(1) - **get_global_offset**(1)) * **get_global_size**(0)) + (**get_global_id**(0) - **get_global_offset**(0)). Requires support for OpenCL C 2.0 or newer.
size_t **get_local_linear_id**()	Returns the work-items 1-dimensional local ID. For 1D work-groups, it is the same value as **get_local_id**(0). For 2D work-groups, it is computed as **get_local_id**(1) * **get_local_size**(0) + **get_local_id**(0). For 3D work-groups, it is computed as (**get_local_id**(2) * **get_local_size**(1) * **get_local_size**(0)) + (**get_local_id**(1) * **get_local_size**(0)) + **get_local_id**(0). Requires support for OpenCL C 2.0 or newer.

The functionality described in the following table requires support for the `cl_khr_subgroups` extension, or OpenCL C 3.0 or newer and the `__opencl_c_subgroups` feature.

The following table describes the list of built-in work-item functions that can be used to query the size of a sub-group, number of sub-groups per work-group, and identifier of the sub-group within a work-group and work-item within a sub-group when this kernel is being executed on a device.

Table 10. Built-in Work-Item Functions for Sub-Groups

Function	Description
uint **get_sub_group_size**()	Returns the number of work-items in the sub-group. This value is no more than the maximum sub-group size and is implementation-defined based on a combination of the compiled kernel and the dispatch dimensions. This will be a constant value for the lifetime of the sub-group.
uint **get_max_sub_group_size**()	Returns the maximum size of a sub-group within the dispatch. This value will be invariant for a given set of dispatch dimensions and a kernel object compiled for a given device.
uint **get_num_sub_groups**()	Returns the number of sub-groups that the current work-group is divided into. This number will be constant for the duration of a work-group's execution. If the kernel is executed with a non-uniform work-group size (i.e. the global_work_size values specified to **clEnqueueNDRangeKernel** are not evenly divisible by the local_work_size values for any dimension, calls to this built-in from some work-groups may return different values than calls to this built-in from other work-groups.
uint **get_enqueued_num_sub_groups**()	Returns the same value as that returned by **get_num_sub_groups** if the kernel is executed with a uniform work-group size. If the kernel is executed with a non-uniform work-group size, returns the number of sub-groups in each of the work-groups that make up the uniform region of the global range.
uint **get_sub_group_id**()	**get_sub_group_id** returns the sub-group ID which is a number from 0 .. **get_num_sub_groups**() - 1. For **clEnqueueTask**, this returns 0.

Function	Description
uint **get_sub_group_local_id**()	Returns the unique work-item ID within the current sub-group. The mapping from **get_local_id**(*dimindx*) to **get_sub_group_local_id** will be invariant for the lifetime of the work-group.

6.15.2. Math Functions

The built-in math functions are categorized into the following:

- A list of built-in functions that have scalar or vector argument versions, and,
- A list of built-in functions that only take scalar `float` arguments.

The vector versions of the math functions operate component-wise. The description is per-component.

The built-in math functions are not affected by the prevailing rounding mode in the calling environment, and always return the same value as they would if called with the round to nearest even rounding mode.

The Built-in Scalar and Vector Argument Math Functions table describes the list of built-in math functions that can take scalar or vector arguments.

The generic type name `gentype` indicates that the function can take any of

- `float`, `float2`, `float3`, `float4`, `float8`, or `float16`
- `double` [42], `double2`, `double3`, `double4`, `double8` or `double16`
- `half` [43], `half2`, `half3`, `half4`, `half8` or `half16`

as the type for the arguments.

The generic type name `gentypef` indicates that the function can take any of

- `float`, `float2`, `float3`, `float4`, `float8`, or `float16`

as the type for the arguments.

The generic type name `gentyped` [44] indicates that the function can take any of

- `double`, `double2`, `double3`, `double4`, `double8` or `double16`

as the type for the arguments.

The generic type name `gentypeh` [45] indicates that the function can take any of

- `half`, `half2`, `half3`, `half4`, `half8` or `half16`

as the type for the arguments.

All functions taking or returning `half` types are supported only when the `cl_khr_fp16` extension macro is supported.

For any specific use of a function with `gentype*` arguments the actual type has to be the same for all arguments and the return type, unless they are explicitly specified as an actual type.

Table 11. Built-in Scalar and Vector Argument Math Functions

Function	Description
gentype **acos**(gentype)	Arc cosine function. Returns an angle in radians.
gentype **acosh**(gentype)	Inverse hyperbolic cosine. Returns an angle in radians.
gentype **acospi**(gentype *x*)	Compute **acos**(*x*) / π.
gentype **asin**(gentype)	Arc sine function. Returns an angle in radians.
gentype **asinh**(gentype)	Inverse hyperbolic sine. Returns an angle in radians.
gentype **asinpi**(gentype *x*)	Compute **asin**(*x*) / π.
gentype **atan**(gentype *y_over_x*)	Arc tangent function. Returns an angle in radians.
gentype **atan2**(gentype *y*, gentype *x*)	Arc tangent of *y* / *x*. Returns an angle in radians.
gentype **atanh**(gentype)	Hyperbolic arc tangent. Returns an angle in radians.
gentype **atanpi**(gentype *x*)	Compute **atan**(*x*) / π.
gentype **atan2pi**(gentype *y*, gentype *x*)	Compute **atan2**(*y*, *x*) / π.
gentype **cbrt**(gentype)	Compute cube-root.
gentype **ceil**(gentype)	Round to integral value using the round to positive infinity rounding mode.
gentype **copysign**(gentype *x*, gentype *y*)	Returns *x* with its sign changed to match the sign of *y*.
gentype **cos**(gentype *x*)	Compute cosine, where *x* is an angle in radians.
gentype **cosh**(gentype *x*)	Compute hyperbolic cosine, where *x* is an angle in radians.
gentype **cospi**(gentype *x*)	Compute **cos**(π *x*).
gentype **erfc**(gentype)	Complementary error function.
gentype **erf**(gentype)	Error function encountered in integrating the *normal distribution*.
gentype **exp**(gentype *x*)	Compute the base-*e* exponential of *x*.
gentype **exp2**(gentype)	Exponential base 2 function.
gentype **exp10**(gentype)	Exponential base 10 function.

Function	Description
gentype **expm1**(gentype *x*)	Compute e^x - 1.0.
gentype **fabs**(gentype)	Compute absolute value of a floating-point number.
gentype **fdim**(gentype *x*, gentype *y*)	*x* - *y* if *x* > *y*, +0 if *x* is less than or equal to *y*.
gentype **floor**(gentype)	Round to integral value using the round to negative infinity rounding mode.
gentype **fma**(gentype *a*, gentype *b*, gentype *c*)	Returns the correctly rounded floating-point representation of the sum of *c* with the infinitely precise product of *a* and *b*. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
gentype **fmax**(gentype *x*, gentype *y*) gentypef **fmax**(gentypef *x*, float *y*) gentyped **fmax**(gentyped *x*, double *y*) gentypeh **fmax**(gentypeh *x*, half *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If one argument is a NaN, **fmax**() returns the other argument. If both arguments are NaNs, **fmax**() returns a NaN.
gentype **fmin**(gentype *x*, gentype *y*) gentypef **fmin**(gentypef *x*, float *y*) gentyped **fmin**(gentyped *x*, double *y*) gentypeh **fmin**(gentypeh *x*, half *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If one argument is a NaN, **fmin**() returns the other argument. If both arguments are NaNs, **fmin**() returns a NaN. [46]
gentype **fmod**(gentype *x*, gentype *y*)	Modulus. Returns *x* - *y* * **trunc**(*x*/*y*).
gentype **fract**(gentype *x*, __global gentype **iptr*) gentype **fract**(gentype *x*, __local gentype **iptr*) gentype **fract**(gentype *x*, __private gentype **iptr*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: gentype **fract**(gentype *x*, gentype **iptr*)	Returns **fmin**(*x* - **floor**(*x*), `C`), where `C` is the constant `0x1.fffffep-1f` for `float` aguments, `0x1.fffffffffffffp-1` for `double` arguments, and `0x1.ffcp-1h` for `half` arguments. **floor**(x) is returned in *iptr*. [47]

Function	Description
half*n* **frexp**(half*n* *x*, __global int*n* *exp) half **frexp**(half *x*, __global int *exp) half*n* **frexp**(half*n* *x*, __local int*n* *exp) half **frexp**(half *x*, __local int *exp) half*n* **frexp**(half*n* *x*, __private int*n* *exp) half **frexp**(half *x*, __private int *exp) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: half*n* **frexp**(half*n* *x*, int*n* *exp) half **frexp**(half *x*, int *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a `half` with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
float*n* **frexp**(float*n* *x*, __global int*n* *exp) float **frexp**(float *x*, __global int *exp) float*n* **frexp**(float*n* *x*, __local int*n* *exp) float **frexp**(float *x*, __local int *exp) float*n* **frexp**(float*n* *x*, __private int*n* *exp) float **frexp**(float *x*, __private int *exp) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float*n* **frexp**(float*n* *x*, int*n* *exp) float **frexp**(float *x*, int *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a `float` with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
double*n* **frexp**(double*n* *x*, __global int*n* *exp) double **frexp**(double *x*, __global int *exp) double*n* **frexp**(double*n* *x*, __local int*n* *exp) double **frexp**(double *x*, __local int *exp) double*n* **frexp**(double*n* *x*, __private int*n* *exp) double **frexp**(double *x*, __private int *exp) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: double*n* **frexp**(double*n* *x*, int*n* *exp) double **frexp**(double *x*, int *exp)	Extract mantissa and exponent from *x*. For each component the mantissa returned is a `double` with magnitude in the interval [1/2, 1) or 0. Each component of *x* equals mantissa returned * 2^{exp}.
gentype **hypot**(gentype *x*, gentype *y*)	Compute the value of the square root of $x^2+ y^2$ without undue overflow or underflow.

Function	Description
int*n* **ilogb**(float*n* *x*) int **ilogb**(float *x*) int*n* **ilogb**(double*n* *x*) int **ilogb**(double *x*) int*n* **ilogb**(half*n* *x*) int **ilogb**(half *x*)	Return the exponent as an integer value.
float*n* **ldexp**(float*n* *x*, int*n* *k*) float*n* **ldexp**(float*n* *x*, int *k*) float **ldexp**(float *x*, int *k*) double*n* **ldexp**(double*n* *x*, int*n* *k*) double*n* **ldexp**(double*n* *x*, int *k*) double **ldexp**(double *x*, int *k*) half*n* **ldexp**(half*n* *x*, int*n* *k*) half*n* **ldexp**(half*n* *x*, int *k*) half **ldexp**(half *x*, int *k*)	Multiply *x* by 2 to the power *k*.

Function	Description
gentype **lgamma**(gentype *x*) float*n* **lgamma_r**(float*n* *x*, __global int*n* **signp*) float **lgamma_r**(float *x*, __global int **signp*) double*n* **lgamma_r**(double*n* *x*, __global int*n* **signp*) double **lgamma_r**(double *x*, __global int **signp*) half*n* **lgamma_r**(half*n* *x*, __global int*n* **signp*) half **lgamma_r**(half *x*, __global int **signp*) float*n* **lgamma_r**(float*n* *x*, __local int*n* **signp*) float **lgamma_r**(float *x*, __local int **signp*) double*n* **lgamma_r**(double*n* *x*, __local int*n* **signp*) double **lgamma_r**(double *x*, __local int **signp*) half*n* **lgamma_r**(half*n* *x*, __local int*n* **signp*) half **lgamma_r**(half *x*, __local int **signp*) float*n* **lgamma_r**(float*n* *x*, __private int*n* **signp*) float **lgamma_r**(float *x*, __private int **signp*) double*n* **lgamma_r**(double*n* *x*, __private int*n* **signp*) double **lgamma_r**(double *x*, __private int **signp*) half*n* **lgamma_r**(half*n* *x*, __private int*n* **signp*) half **lgamma_r**(half *x*, __private int **signp*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float*n* **lgamma_r**(float*n* *x*, int*n* **signp*) float **lgamma_r**(float *x*, int **signp*) double*n* **lgamma_r**(double*n* *x*, int*n* **signp*) double **lgamma_r**(double *x*, int **signp*) half*n* **lgamma_r**(half*n* *x*, int*n* **signp*) half **lgamma_r**(half *x*, int **signp*)	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the *signp* argument of **lgamma_r**.
gentype **log**(gentype)	Compute natural logarithm.
gentype **log2**(gentype)	Compute a base 2 logarithm.
gentype **log10**(gentype)	Compute a base 10 logarithm.

Function	Description
gentype **log1p**(gentype *x*)	Compute $\log_e(1.0 + x)$.
gentype **logb**(gentype *x*)	Compute the exponent of *x*, which is the integral part of $\log_r(\|x\|)$.
gentype **mad**(gentype *a*, gentype *b*, gentype *c*)	**mad** computes *a* * *b* + *c*. The function may compute *a* * *b* + *c* with reduced accuracy in the embedded profile. On some hardware the mad instruction may provide better performance than expanded computation of *a* * *b* + *c*. [48]
gentype **maxmag**(gentype *x*, gentype *y*)	Returns *x* if $\|x\| > \|y\|$, *y* if $\|y\| > \|x\|$, otherwise **fmax**(*x*, *y*). Requires support for OpenCL C 1.1 or newer.
gentype **minmag**(gentype *x*, gentype *y*)	Returns *x* if $\|x\| < \|y\|$, *y* if $\|y\| < \|x\|$, otherwise **fmin**(*x*, *y*). Requires support for OpenCL C 1.1 or newer.
gentype **modf**(gentype *x*, __global gentype **iptr*) gentype **modf**(gentype *x*, __local gentype **iptr*) gentype **modf**(gentype *x*, __private gentype **iptr*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: gentype **modf**(gentype *x*, gentype **iptr*)	Decompose a floating-point number. The **modf** function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by *iptr*.
float*n* **nan**(uint*n* *nancode*) float **nan**(uint *nancode*) double*n* **nan**(ulong*n* *nancode*) double **nan**(ulong *nancode*) half*n* **nan**(ushort*n* *nancode*) half **nan**(ushort *nancode*)	Returns a quiet NaN. The *nancode* may be placed in the significand of the resulting NaN.
gentype **nextafter**(gentype *x*, gentype *y*)	Computes the next representable floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, **nextafter**() returns the largest representable floating-point number less than *x*.
gentype **pow**(gentype *x*, gentype *y*)	Compute *x* to the power *y*.

Function	Description
float*n* **pown**(float*n* *x*, int*n* *y*) float **pown**(float *x*, int *y*) double*n* **pown**(double*n* *x*, int*n* *y*) double **pown**(double *x*, int *y*) half*n* **pown**(half*n* *x*, int*n* *y*) half **pown**(half *x*, int *y*)	Compute *x* to the power *y*, where *y* is an integer.
gentype **powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0.
gentype **remainder**(gentype *x*, gentype *y*)	Compute the value *r* such that *r* = *x* - *n***y*, where *n* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *n* shall be the even one. If *r* is zero, it is given the same sign as *x*.
float*n* **remquo**(float*n* *x*, float*n* *y*, __global int*n* **quo*) float **remquo**(float *x*, float *y*, __global int **quo*) float*n* **remquo**(float*n* *x*, float*n* *y*, __local int*n* **quo*) float **remquo**(float *x*, float *y*, __local int **quo*) float*n* **remquo**(float*n* *x*, float*n* *y*, __private int*n* **quo*) float **remquo**(float *x*, float *y*, __private int **quo*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float*n* **remquo**(float*n* *x*, float*n* *y*, int*n* **quo*) float **remquo**(float *x*, float *y*, int **quo*)	The **remquo** function computes the value r such that *r* = *x* - *k***y*, where *k* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *k* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.

Function	Description
double*n* **remquo**(double*n* *x*, double*n* *y*, __global int*n* **quo*) double **remquo**(double *x*, double *y*, __global int **quo*) double*n* **remquo**(double*n* *x*, double*n* *y*, __local int*n* **quo*) double **remquo**(double *x*, double *y*, __local int **quo*) double*n* **remquo**(double*n* *x*, double*n* *y*, __private int*n* **quo*) double **remquo**(double *x*, double *y*, __private int **quo*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: double*n* **remquo**(double*n* *x*, double*n* *y*, int*n* **quo*) double **remquo**(double *x*, double *y*, int **quo*)	The **remquo** function computes the value r such that *r* = *x* - *k***y*, where *k* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *k* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
half*n* **remquo**(half*n* *x*, half*n* *y*, __global int*n* **quo*) half **remquo**(half *x*, half *y*, __global int **quo*) half*n* **remquo**(half*n* *x*, half*n* *y*, __local int*n* **quo*) half **remquo**(half *x*, half *y*, __local int **quo*) half*n* **remquo**(half*n* *x*, half*n* *y*, __private int*n* **quo*) half **remquo**(half *x*, half *y*, __private int **quo*) For OpenCL C 2.0 or with the `__opencl_c_generic_address_space` feature: half*n* **remquo**(half*n* *x*, half*n* *y*, int*n* **quo*) half **remquo**(half *x*, half *y*, int **quo*)	The **remquo** function computes the value r such that *r* = *x* - *k***y*, where *k* is the integer nearest the exact value of *x*/*y*. If there are two integers closest to *x*/*y*, *k* shall be the even one. If *r* is zero, it is given the same sign as *x*. This is the same value that is returned by the **remainder** function. **remquo** also calculates the lower seven bits of the integral quotient *x*/*y*, and gives that value the same sign as *x*/*y*. It stores this signed value in the object pointed to by *quo*.
gentype **rint**(gentype)	Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 for description of rounding modes.

Function	Description
float*n* **rootn**(float*n* *x*, int*n* *y*) float **rootn**(float *x*, int *y*) double*n* **rootn**(double*n* *x*, int*n* *y*) double **rootn**(double *x*, int *y*) half*n* **rootn**(half*n* *x*, int*n* *y*) half **rootn**(half *x*, int *y*)	Compute *x* to the power 1/*y*.
gentype **round**(gentype *x*)	Return the integral value nearest to *x* rounding halfway cases away from zero, regardless of the current rounding direction.
gentype **rsqrt**(gentype)	Compute inverse square root.
gentype **sin**(gentype *x*)	Compute sine, where *x* is an angle in radians.
gentype **sincos**(gentype *x*, __global gentype **cosval*) gentype **sincos**(gentype *x*, __local gentype **cosval*) gentype **sincos**(gentype *x*, __private gentype **cosval*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: gentype **sincos**(gentype *x*, gentype **cosval*)	Compute sine and cosine of *x*. The computed sine is the return value and computed cosine is returned in *cosval*, where *x* is an angle in radians.
gentype **sinh**(gentype *x*)	Compute hyperbolic sine, where *x* is an angle in radians
gentype **sinpi**(gentype *x*)	Compute **sin**(π *x*).
gentype **sqrt**(gentype)	Compute square root.
gentype **tan**(gentype *x*)	Compute tangent, where *x* is an angle in radians.
gentype **tanh**(gentype *x*)	Compute hyperbolic tangent, where *x* is an angle in radians.
gentype **tanpi**(gentype *x*)	Compute **tan**(π *x*).
gentype **tgamma**(gentype)	Compute the gamma function.
gentype **trunc**(gentype)	Round to integral value using the round to zero rounding mode.

The following table describes the following functions:

- A subset of functions from Built-in Scalar and Vector Argument Math Functions that are defined with the half_ prefix. These functions are implemented with a minimum of 10-bits of accuracy, i.e. the maximum error value <= 8192 ulp.
- A subset of functions from Built-in Scalar and Vector Argument Math Functions that are defined

with the native_ prefix. These functions may map to one or more native device instructions and will typically have better performance compared to the corresponding functions without the `native_` prefix). The accuracy (and in some cases the input range(s)) of these functions is implementation-defined.

- `half_` and `native_` functions for following basic operations: divide and reciprocal.

We use the generic type name `gentype` to indicate that the functions in the following table can take `float`, `float2`, `float3`, `float4`, `float8` or `float16` as the type for the arguments.

The use of `half` in this table does not refer to the argument and return types, which are 32-bit floating-point values, but to the accuracy requirements of the function results.

Table 12. Built-in Scalar and Vector half and native Math Functions

Function	Description
gentype **half_cos**(gentype *x*)	Compute cosine. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **half_divide**(gentype *x*, gentype *y*)	Compute *x* / *y*.
gentype **half_exp**(gentype *x*)	Compute the base-*e* exponential of *x*.
gentype **half_exp2**(gentype *x*)	Compute the base- 2 exponential of *x*.
gentype **half_exp10**(gentype *x*)	Compute the base- 10 exponential of *x*.
gentype **half_log**(gentype *x*)	Compute natural logarithm.
gentype **half_log2**(gentype *x*)	Compute a base 2 logarithm.
gentype **half_log10**(gentype *x*)	Compute a base 10 logarithm.
gentype **half_powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0.
gentype **half_recip**(gentype *x*)	Compute reciprocal.
gentype **half_rsqrt**(gentype *x*)	Compute inverse square root.
gentype **half_sin**(gentype *x*)	Compute sine. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **half_sqrt**(gentype *x*)	Compute square root.
gentype **half_tan**(gentype *x*)	Compute tangent. *x* is an angle in radians, and must be in the range [-2^{16}, $+2^{16}$].
gentype **native_cos**(gentype *x*)	Compute cosine over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.
gentype **native_divide**(gentype *x*, gentype *y*)	Compute *x* / *y* over an implementation-defined range. The maximum error is implementation-defined.

Function	Description
gentype **native_exp**(gentype *x*)	Compute the base-*e* exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp2**(gentype *x*)	Compute the base-2 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_exp10**(gentype *x*)	Compute the base-10 exponential of *x* over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log**(gentype *x*)	Compute natural logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log2**(gentype *x*)	Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_log10**(gentype *x*)	Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_powr**(gentype *x*, gentype *y*)	Compute *x* to the power *y*, where *x* is >= 0. The range of *x* and *y* are implementation-defined. The maximum error is implementation-defined.
gentype **native_recip**(gentype *x*)	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_rsqrt**(gentype *x*)	Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_sin**(gentype *x*)	Compute sine over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.
gentype **native_sqrt**(gentype *x*)	Compute square root over an implementation-defined range. The maximum error is implementation-defined.
gentype **native_tan**(gentype *x*)	Compute tangent over an implementation-defined range, where *x* is an angle in radians. The maximum error is implementation-defined.

Support for denormal values is optional for **half_** functions. The **half_** functions may return any result allowed by Edge Case Behavior, even when `-cl-denorms-are-zero` (see section 5.8.4.2 of the OpenCL Specification) is not in force. Support for denormal values is implementation-defined for **native_** functions.

The following constants are available. Their values are of type `float` and are accurate within the precision of a single precision floating-point number.

Constant Name	Description
`MAXFLOAT`	Value of maximum non-infinite single-precision floating-point number.
`HUGE_VALF`	A positive `float` constant expression. `HUGE_VALF` evaluates to +infinity. Used as an error value returned by the built-in math functions.
`INFINITY`	A constant expression of type `float` representing positive or unsigned infinity.
`NAN`	A constant expression of type `float` representing a quiet NaN.

If double-precision is supported by the device, then the following constants are also available:

Constant Name	Description
`HUGE_VAL`	A positive double constant expression. `HUGE_VAL` evaluates to +infinity. Used as an error value returned by the built-in math functions.

6.15.2.1. Floating-point Macros and Pragmas

The `FP_CONTRACT` pragma can be used to allow (if the state is on) or disallow (if the state is off) the implementation to contract expressions. Each pragma can occur either outside external declarations or preceding all explicit declarations and statements inside a compound statement. When outside external declarations, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered, or until the end of the translation unit. When inside a compound statement, the pragma takes effect from its occurrence until another `FP_CONTRACT` pragma is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. If this pragma is used in any other context, the behavior is undefined.

The pragma definition to set `FP_CONTRACT` is:

```
// on-off-switch is one of ON, OFF, or DEFAULT.
// The DEFAULT value is ON.
#pragma OPENCL FP_CONTRACT on-off-switch
```

The `FP_FAST_FMAF` macro indicates whether the **fma** function is fast compared with direct code for single precision floating-point. If defined, the `FP_FAST_FMAF` macro shall indicate that the **fma** function generally executes about as fast as, or faster than, a multiply and an add of `float` operands.

The macro names given in the following list must use the values specified. These constant

expressions are suitable for use in `#if` preprocessing directives.

```
#define FLT_DIG         6
#define FLT_MANT_DIG    24
#define FLT_MAX_10_EXP  +38
#define FLT_MAX_EXP     +128
#define FLT_MIN_10_EXP  -37
#define FLT_MIN_EXP     -125
#define FLT_RADIX       2
#define FLT_MAX         0x1.fffffep127f
#define FLT_MIN         0x1.0p-126f
#define FLT_EPSILON     0x1.0p-23f
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
`FLT_DIG`	`CL_FLT_DIG`
`FLT_MANT_DIG`	`CL_FLT_MANT_DIG`
`FLT_MAX_10_EXP`	`CL_FLT_MAX_10_EXP`
`FLT_MAX_EXP`	`CL_FLT_MAX_EXP`
`FLT_MIN_10_EXP`	`CL_FLT_MIN_10_EXP`
`FLT_MIN_EXP`	`CL_FLT_MIN_EXP`
`FLT_RADIX`	`CL_FLT_RADIX`
`FLT_MAX`	`CL_FLT_MAX`
`FLT_MIN`	`CL_FLT_MIN`
`FLT_EPSILON`	`CL_FLT_EPSILON`

The following macros shall expand to integer constant expressions whose values are returned by **ilogb**(*x*) if *x* is zero or NaN, respectively. The value of `FP_ILOGB0` shall be either `INT_MIN` or `-INT_MAX`. The value of `FP_ILOGBNAN` shall be either `INT_MAX` or `INT_MIN`.

The following constants are also available. They are of type `float` and are accurate within the precision of the `float` type.

Constant	Description
`M_E_F`	Value of *e*
`M_LOG2E_F`	Value of $\log_2 e$
`M_LOG10E_F`	Value of $\log_{10} e$
`M_LN2_F`	Value of $\log_e 2$
`M_LN10_F`	Value of $\log_e 10$
`M_PI_F`	Value of π
`M_PI_2_F`	Value of $\pi / 2$

Constant	Description
M_PI_4_F	Value of π / 4
M_1_PI_F	Value of 1 / π
M_2_PI_F	Value of 2 / π
M_2_SQRTPI_F	Value of 2 / √π
M_SQRT2_F	Value of √2
M_SQRT1_2_F	Value of 1 / √2

If double-precision is supported by the device, then the following macros and constants are also available:

The FP_FAST_FMA macro indicates whether the **fma**() family of functions are fast compared with direct code for double-precision floating-point. If defined, the FP_FAST_FMA macro shall indicate that the **fma**() function generally executes about as fast as, or faster than, a multiply and an add of double operands

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in #if preprocessing directives.

```
#define DBL_DIG         15
#define DBL_MANT_DIG    53
#define DBL_MAX_10_EXP  +308
#define DBL_MAX_EXP     +1024
#define DBL_MIN_10_EXP  -307
#define DBL_MIN_EXP     -1021
#define DBL_MAX         0x1.fffffffffffffp1023
#define DBL_MIN         0x1.0p-1022
#define DBL_EPSILON     0x1.0p-52
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
DBL_DIG	CL_DBL_DIG
DBL_MANT_DIG	CL_DBL_MANT_DIG
DBL_MAX_10_EXP	CL_DBL_MAX_10_EXP
DBL_MAX_EXP	CL_DBL_MAX_EXP
DBL_MIN_10_EXP	CL_DBL_MIN_10_EXP
DBL_MIN_EXP	CL_DBL_MIN_EXP
DBL_MAX	CL_DBL_MAX
DBL_MIN	CL_DBL_MIN
DBL_EPSILON	CL_DBL_EPSILON

The following constants are also available. They are of type double and are accurate within the

precision of the double type.

Constant	Description
`M_E`	Value of *e*
`M_LOG2E`	Value of $\log_2 e$
`M_LOG10E`	Value of $\log_{10} e$
`M_LN2`	Value of $\log_e 2$
`M_LN10`	Value of $\log_e 10$
`M_PI`	Value of π
`M_PI_2`	Value of $\pi / 2$
`M_PI_4`	Value of $\pi / 4$
`M_1_PI`	Value of $1 / \pi$
`M_2_PI`	Value of $2 / \pi$
`M_2_SQRTPI`	Value of $2 / \sqrt{\pi}$
`M_SQRT2`	Value of $\sqrt{2}$
`M_SQRT1_2`	Value of $1 / \sqrt{2}$

If the `cl_khr_fp16` extension macro is supported, then the following macros and constants are also available:

The `FP_FAST_FMA_HALF` macro indicates whether the **fma**() family of functions are fast compared with direct code for half-precision floating-point. If defined, the `FP_FAST_FMA_HALF` macro shall indicate that the **fma**() function generally executes about as fast as, or faster than, a multiply and an add of `half` operands.

The macro names given in the following list must use the values specified. These constant expressions are suitable for use in #if preprocessing directives.

```
#define HALF_DIG            3
#define HALF_MANT_DIG       11
#define HALF_MAX_10_EXP     +4
#define HALF_MAX_EXP        +16
#define HALF_MIN_10_EXP     -4
#define HALF_MIN_EXP        -13
#define HALF_RADIX          2
#define HALF_MAX            0x1.ffcp15h
#define HALF_MIN            0x1.0p-14h
#define HALF_EPSILON        0x1.0p-10h
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
HALF_DIG	CL_HALF_DIG
HALF_MANT_DIG	CL_HALF_MANT_DIG
HALF_MAX_10_EXP	CL_HALF_MAX_10_EXP
HALF_MAX_EXP	CL_HALF_MAX_EXP
HALF_MIN_10_EXP	CL_HALF_MIN_10_EXP
HALF_MIN_EXP	CL_HALF_MIN_EXP
HALF_RADIX	CL_HALF_RADIX
HALF_MAX	CL_HALF_MAX
HALF_MIN	CL_HALF_MIN
HALF_EPSILON	CL_HALF_EPSILON

The following constants are also available. They are of type half and are accurate within the precision of the half type.

Constant	Description
M_E_H	Value of e
M_LOG2E_H	Value of $\log_2 e$
M_LOG10E_H	Value of $\log_{10} e$
M_LN2_H	Value of $\log_e 2$
M_LN10_H	Value of $\log_e 10$
M_PI_H	Value of π
M_PI_2_H	Value of π / 2
M_PI_4_H	Value of π / 4
M_1_PI_H	Value of 1 / π
M_2_PI_H	Value of 2 / π
M_2_SQRTPI_H	Value of 2 / √π
M_SQRT2_H	Value of √2
M_SQRT1_2_H	Value of 1 / √2

6.15.3. Integer Functions

The following table describes the built-in integer functions that take scalar or vector arguments. The vector versions of the integer functions operate component-wise. The description is per-component.

We use the generic type name gentype to indicate that the function can take char, charn, uchar, ucharn, short, shortn, ushort, ushortn, int, intn, uint, uintn, long [49], longn, ulong, or ulongn as the type for the arguments. We use the generic type name ugentype to refer to unsigned versions of gentype. For example, if gentype is char4, ugentype is uchar4. We also use the generic type name sgentype to indicate that the function can take a scalar data type, i.e. char, uchar, short, ushort, int, uint, long, or

`ulong`, as the type for the arguments. For built-in integer functions that take `gentype` and `sgentype` arguments, the `gentype` argument must be a vector or scalar version of the `sgentype` argument. For example, if `sgentype` is `uchar`, `gentype` must be `uchar` or `ucharn`. For vector versions, `sgentype` is implicitly widened to `gentype` as described for arithmetic operators. *n* is 2, 3, 4, 8, or 16.

For any specific use of a function with `gentype*` arguments the actual type has to be the same for all arguments and the return type, unless they are explicitly specified as an actual type.

Table 13. Built-in Scalar and Vector Integer Argument Functions

Function	Description
ugentype **abs**(gentype *x*)	Returns |x|.
ugentype **abs_diff**(gentype *x*, gentype *y*)	Returns |x - y| without modulo overflow.
gentype **add_sat**(gentype *x*, gentype *y*)	Returns *x* + *y* and saturates the result.
gentype **hadd**(gentype *x*, gentype *y*)	Returns (*x* + *y*) >> 1. The intermediate sum does not modulo overflow.
gentype **rhadd**(gentype *x*, gentype *y*)	Returns (*x* + *y* + 1) >> 1. The intermediate sum does not modulo overflow. [50]
gentype **clamp**(gentype *x*, gentype *minval*, gentype *maxval*) gentype **clamp**(gentype *x*, sgentype *minval*, sgentype *maxval*)	Returns **min**(**max**(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*. Requires support for OpenCL C 1.1 or newer.
gentype **clz**(gentype *x*)	Returns the number of leading 0-bits in *x*, starting at the most significant bit position. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector.
gentype **ctz**(gentype *x*)	Returns the count of trailing 0-bits in *x*. If *x* is 0, returns the size in bits of the type of *x* or component type of *x*, if *x* is a vector. Requires support for OpenCL C 2.0 or newer.
uint **dot**(uchar4 a, uchar4 b) int **dot**(char4 a, char4 b) int **dot**(uchar4 a, char4 b) int **dot**(char4 a, uchar4 b)	`dot` returns the dot product of the two input vectors `a` and `b`. The components of `a` and `b` are sign- or zero-extended to the width of the destination type and the vectors with extended components are multiplied component-wise. All the components of the resulting vectors are added together to form the final result. Requires support for the `__opencl_c_integer_dot_product_input_4x8bit` feature.

Function	Description

uint **dot_acc_sat**(uchar4 a, uchar4 b, uint acc)
int **dot_acc_sat**(char4 a, char4 b, int acc)
int **dot_acc_sat**(uchar4 a, char4 b, int acc)
int **dot_acc_sat**(char4 a, uchar4 b, int acc)
	`dot_acc_sat` returns the saturating addition of the dot product of the two input vectors `a` and `b` and the accumulator `acc`:
	
	`product = dot(a,b);`
	`result = add_sat(product, acc);`
	
	Requires support for the `__opencl_c_integer_dot_product_input_4x8bit` feature.

uint **dot_4x8packed_uu_uint**(uint a, uint b) int **dot_4x8packed_ss_int**(uint a, uint b) int **dot_4x8packed_us_int**(uint a, uint b) int **dot_4x8packed_su_int**(uint a, uint b)	Returns **dot** for 4x8 bit input vectors packed into a 32-bit word. Requires support for the `__opencl_c_integer_dot_product_input_4x8bit_packed` feature.
uint **dot_acc_sat_4x8packed_uu_uint**(uint a, uint b, uint acc) int **dot_acc_sat_4x8packed_ss_int**(uint a, uint b, int acc) int **dot_acc_sat_4x8packed_us_int**(uint a, uint b, int acc) int **dot_acc_sat_4x8packed_su_int**(uint a, uint b, int acc)	Returns **dot_acc_set** for 4x8 bit input vectors packed into a 32-bit word. Requires support for the `__opencl_c_integer_dot_product_input_4x8bit_packed` feature.
gentype **mad_hi**(gentype *a*, gentype *b*, gentype *c*)	Returns **mul_hi**(*a*, *b*) + *c*.
gentype **mad_sat**(gentype *a*, gentype *b*, gentype *c*)	Returns *a* * *b* + *c* and saturates the result.
gentype **max**(gentype *x*, gentype *y*) For OpenCL C 1.1 or newer: gentype **max**(gentype *x*, sgentype *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*.
gentype **min**(gentype *x*, gentype *y*) For OpenCL C 1.1 or newer: gentype **min**(gentype *x*, sgentype *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*.
gentype **mul_hi**(gentype *x*, gentype *y*)	Computes *x* * *y* and returns the high half of the product of *x* and *y*.
gentype **rotate**(gentype *v*, gentype *i*)	For each element in *v*, the bits are shifted left by the number of bits given by the corresponding element in *i* (subject to the usual shift modulo rules). Bits shifted off the left side of the element are shifted back in from the right.

Function	Description
gentype **sub_sat**(gentype *x*, gentype *y*)	Returns *x* - *y* and saturates the result.
short **upsample**(char *hi*, uchar *lo*) ushort **upsample**(uchar *hi*, uchar *lo*) short*n* **upsample**(char*n* *hi*, uchar*n* *lo*) ushort*n* **upsample**(uchar*n* *hi*, uchar*n* *lo*)	*result*[i] = ((short)*hi*[i] << 8) | *lo*[i] *result*[i] = ((ushort)*hi*[i] << 8) | *lo*[i]
int **upsample**(short *hi*, ushort *lo*) uint **upsample**(ushort *hi*, ushort *lo*) int*n* **upsample**(short*n* *hi*, ushort*n* *lo*) uint*n* **upsample**(ushort*n* *hi*, ushort*n* *lo*)	*result*[i] = ((int)*hi*[i] << 16) | *lo*[i] *result*[i] = ((uint)*hi*[i] << 16) | *lo*[i]
long **upsample**(int *hi*, uint *lo*) ulong **upsample**(uint *hi*, uint *lo*) long*n* **upsample**(int*n* *hi*, uint*n* *lo*) ulong*n* **upsample**(uint*n* *hi*, uint*n* *lo*)	*result*[i] = ((long)*hi*[i] << 32) | *lo*[i] *result*[i] = ((ulong)*hi*[i] << 32) | *lo*[i]
gentype **popcount**(gentype *x*)	Returns the number of non-zero bits in *x*. Requires support for OpenCL C 1.2 or newer.

The following table describes fast integer functions that can be used for optimizing performance of kernels. We use the generic type name `gentype` to indicate that the function can take `int`, `int2`, `int3`, `int4`, `int8`, `int16`, `uint`, `uint2`, `uint3`, `uint4`, `uint8` or `uint16` as the type for the arguments.

Table 14. Built-in 24-bit Integer Functions

Function	Description
gentype **mad24**(gentype *x*, gentype *y*, gentype z)	Multipy two 24-bit integer values *x* and *y* and add the 32-bit integer result to the 32-bit integer *z*. Refer to definition of **mul24** to see how the 24-bit integer multiplication is performed.
gentype **mul24**(gentype *x*, gentype *y*)	Multiply two 24-bit integer values *x* and *y*. *x* and *y* are 32-bit integers but only the low 24-bits are used to perform the multiplication. **mul24** should only be used when values in *x* and *y* are in the range [-2^{23}, $2^{23}-1$] if *x* and *y* are signed integers and in the range [0, $2^{24}-1$] if *x* and *y* are unsigned integers. If *x* and *y* are not in this range, the multiplication result is implementation-defined.

6.15.3.1. Extended Bit Operations

If the `cl_khr_extended_bit_ops` extension macro or OpenCL C 3.1 or newer is supported, the functions described in the Built-in Scalar and Vector Extended Bit Operations table can be used with built-in scalar or vector integer types to perform extended bit operations. The functions that operate on vector types operate component-wise. The description is per-component.

In the table below, the generic type name `gentype` refers to the built-in integer types `char`, `charn`, `uchar`, `ucharn`, `short`, `shortn`, `ushort`, `ushortn`, `int`, `intn`, `uint`, `uintn`, `long`, `longn`, `ulong`, and `ulongn`. The generic type name `igentype` refers to the built-in signed integer types `char`, `charn`, `short`, `shortn`, `int`, `intn`, `long`, and `longn`. The generic type name `ugentype` refers to the built-in unsigned integer types `uchar`, `ucharn`, `ushort`, `ushortn`, `uint`, `uintn`, `ulong`, and `ulongn`. *n* is 2, 3, 4, 8, or 16.

The functionality described in the following table requires support for the `cl_khr_extended_bit_ops` extension macro or OpenCL C 3.1 or newer.

Table 15. Built-in Scalar and Vector Extended Bit Operations

Function	Description

`gentype bitfield_insert(`
`  gentype base, gentype insert,`
`  uint offset, uint count)`
	Returns a copy of *base*, with a modified bitfield that comes from *insert*.
	
	Any bits of the result value numbered outside [*offset*, *offset* + *count* - 1] (inclusive) will come from the corresponding bits in *base*.
	
	Any bits of the result value numbered inside [*offset*, *offset* + *count* - 1] (inclusive) will come from the bits numbered [0, *count* - 1] (inclusive) of *insert*.
	
	count is the number of bits to be modified. If *count* equals 0, the return value will be equal to *base*.
	
	If *count* or *offset* or *offset* + *count* is greater than number of bits in `gentype` (for scalar types) or components of `gentype` (for vector types), the result is undefined.

Function	Description

`igentype bitfield_extract_signed(`
`  gentype base,`
`  uint offset, uint count)`
	Returns an extracted bitfield from *base* with sign extension. The type of the return value is always a signed type.
	
	The bits of *base* numbered in [*offset*, *offset* + *count* - 1] (inclusive) are returned as the bits numbered in [0, *count* - 1] (inclusive) of the result. The remaining bits in the result will be sign extended by replicating the bit numbered *offset* + *count* - 1 of *base*.
	
	count is the number of bits to be extracted. If *count* equals 0, the result is 0.
	
	If the *count* or *offset* or *offset* + *count* is greater than number of bits in `gentype` (for scalar types) or components of `gentype` (for vector types), the result is undefined.

`ugentype bitfield_extract_unsigned(`
`  gentype base,`
`  uint offset, uint count)`
	Returns an extracted bitfield from *base* with zero extension. The type of the return value is always an unsigned type.
	
	The bits of *base* numbered in [*offset*, *offset* + *count* - 1] (inclusive) are returned as the bits numbered in [0, *count* - 1] (inclusive) of the result. The remaining bits in the result will be zero.
	
	count is the number of bits to be extracted. If *count* equals 0, the result is 0.
	
	If the *count* or *offset* or *offset* + *count* is greater than number of bits in `gentype` (for scalar types) or components of `gentype` (for vector types), the result is undefined.

`gentype bit_reverse(` `  gentype base)`	Returns the value of *base* with reversed bits. That is, the bit numbered *n* of the result value will be taken from the bit numbered *width* - *n* - 1 of *base* (for scalar types) or a component of *base* (for vector types), where *width* is number of bits of `gentype` (for scalar types) or components of `gentype` (for vector types).

6.15.3.2. Integer Macros

The macro names given in the following list must use the values specified. The values shall all be constant expressions suitable for use in `#if` preprocessing directives.

```
#define CHAR_BIT        8
#define CHAR_MAX        SCHAR_MAX
#define CHAR_MIN        SCHAR_MIN
#define INT_MAX         2147483647
#define INT_MIN         (-2147483647 - 1)
#define LONG_MAX        0x7fffffffffffffffL
#define LONG_MIN        (-0x7fffffffffffffffL - 1)
#define SCHAR_MAX       127
#define SCHAR_MIN       (-127 - 1)
#define SHRT_MAX        32767
#define SHRT_MIN        (-32767 - 1)
#define UCHAR_MAX       255
#define USHRT_MAX       65535
#define UINT_MAX        0xffffffff
#define ULONG_MAX       0xffffffffffffffffUL
```

The following table describes the built-in macro names given above in the OpenCL C programming language and the corresponding macro names available to the application.

Macro in OpenCL Language	Macro for application
CHAR_BIT	CL_CHAR_BIT
CHAR_MAX	CL_CHAR_MAX
CHAR_MIN	CL_CHAR_MIN
INT_MAX	CL_INT_MAX
INT_MIN	CL_INT_MIN
LONG_MAX	CL_LONG_MAX
LONG_MIN	CL_LONG_MIN
SCHAR_MAX	CL_SCHAR_MAX
SCHAR_MIN	CL_SCHAR_MIN
SHRT_MAX	CL_SHRT_MAX
SHRT_MIN	CL_SHRT_MIN
UCHAR_MAX	CL_UCHAR_MAX
USHRT_MAX	CL_USHRT_MAX
UINT_MAX	CL_UINT_MAX
ULONG_MAX	CL_ULONG_MAX

6.15.4. Common Functions

The following table describes the list of built-in common functions. These all operate component-wise. The description is per-component.

The generic type name `gentype` indicates that the function can take any of

- `float`, `float2`, `float3`, `float4`, `float8`, or `float16`
- `double` [42], `double2`, `double3`, `double4`, `double8` or `double16`
- `half` [51], `half2`, `half3`, `half4`, `half8` or `half16`

as the type for the arguments.

The generic type name `gentypef` indicates that the function can take any of

- `float`, `float2`, `float3`, `float4`, `float8`, or `float16`

as the type for the arguments.

The generic type name `gentyped` [52] indicates that the function can take any of

- `double`, `double2`, `double3`, `double4`, `double8` or `double16`

as the type for the arguments.

The generic type name `gentypeh` [53] indicates that the function can take any of

- `half`, `half2`, `half3`, `half4`, `half8` or `half16`

as the type for the arguments.

All functions taking or returning `half` types are supported only when the `cl_khr_fp16` extension macro is supported.

Table 16. Built-in Scalar and Vector Argument Common Functions

Function	Description
gentype **clamp**(gentype *x*, gentype *minval*, gentype *maxval*) gentypef **clamp**(gentypef *x*, float *minval*, float *maxval*) gentyped **clamp**(gentyped *x*, double *minval*, double *maxval*) gentypeh **clamp**(gentypeh *x*, half *minval*, half *maxval*)	Returns **fmin**(**fmax**(*x*, *minval*), *maxval*). Results are undefined if *minval* > *maxval*.
gentype **degrees**(gentype *radians*)	Converts *radians* to degrees, i.e. (180 / π) * *radians*.
gentype **max**(gentype *x*, gentype *y*) gentypef **max**(gentypef *x*, float *y*) gentyped **max**(gentyped *x*, double *y*) gentypeh **max**(gentypeh *x*, half *y*)	Returns *y* if *x* < *y*, otherwise it returns *x*. If *x* or *y* are infinite or NaN, the return values are undefined.

Function	Description
gentype **min**(gentype *x*, gentype *y*) gentypef **min**(gentypef *x*, float *y*) gentyped **min**(gentyped *x*, double *y*) gentypeh **min**(gentypeh *x*, half *y*)	Returns *y* if *y* < *x*, otherwise it returns *x*. If *x* or *y* are infinite or NaN, the return values are undefined.
gentype **mix**(gentype *x*, gentype *y*, gentype *a*) gentypef **mix**(gentypef *x*, gentypef *y*, float *a*) gentyped **mix**(gentyped *x*, gentyped *y*, double *a*) gentypeh **mix**(gentypeh *x*, gentypeh *y*, half *a*)	Returns the linear blend of *x* and *y* implemented as: *x* + (*y* - *x*) * *a* *a* must be a value in the range [0.0, 1.0]. If *a* is not in the range [0.0, 1.0], the return values are undefined. The half-precision **mix** function can be implemented using contractions such as **mad** or **fma**.
gentype **radians**(gentype *degrees*)	Converts *degrees* to radians, i.e. (π / 180) * *degrees*.
gentype **step**(gentype *edge*, gentype *x*) gentypef **step**(float *edge*, gentypef *x*) gentyped **step**(double *edge*, gentyped *x*) gentypeh **step**(half *edge*, gentypeh *x*)	Returns 0.0 if *x* < *edge*, otherwise it returns 1.0.

Function	Description

gentype **smoothstep**(gentype *edge0*, gentype *edge1*, gentype *x*)
gentypef **smoothstep**(float *edge0*, float *edge1*, gentypef *x*)
gentyped **smoothstep**(double *edge0*, double *edge1*, gentyped *x*)
 	
gentypeh **smoothstep**(half *edge0*, half *edge1*, gentypeh *x*)
	Returns 0.0 if *x* <= *edge0* and 1.0 if *x* >= *edge1* and performs smooth Hermite interpolation between 0 and 1 when *edge0* < *x* < *edge1*. This is useful in cases where you would want a threshold function with a smooth transition.
	
	This is equivalent to:
	
	`gentype t;`
	`t = clamp ((x - edge0) / (edge1 - edge0), 0, 1);`
	`return t * t * (3 - 2 * t);`
	
	Results are undefined if *edge0* >= *edge1* or if *x*, *edge0* or *edge1* is a NaN.
	
	The half-precision **mix** function can be implemented using contractions such as **mad** or **fma**.

gentype **sign**(gentype *x*)	Returns 1.0 if *x* > 0, -0.0 if *x* = -0.0, +0.0 if *x* = +0.0, or -1.0 if *x* < 0. Returns 0.0 if *x* is a NaN.

6.15.5. Geometric Functions

The following table describes the list of built-in geometric functions.

The generic type name `gentypef` indicates that the function can take any of

- `float`, `float2`, `float3`, or `float4`

as the type for the arguments.

The generic type name `gentyped` [54] indicates that the function can take any of

- `double`, `double2`, `double3`, or `double4`

as the type for the arguments.

The generic type name `gentypeh` [55] indicates that the function can take any of

- `half`, `half2`, `half3`, or `half4`

as the type for the arguments.

All functions taking or returning `half` types are supported only when the `cl_khr_fp16` extension macro is supported.

For any specific use of a function with `gentype*` arguments the actual type has to be the same for all arguments and the return type, unless they are explicitly specified as an actual type.

Table 17. Built-in Scalar and Vector Argument Geometric Functions

Function	Description
float4 **cross**(float4 *p0*, float4 *p1*) float3 **cross**(float3 *p0*, float3 *p1*) double4 **cross**(double4 *p0*, double4 *p1*) double3 **cross**(double3 *p0*, double3 *p1*) half4 **cross**(half4 *p0*, half4 *p1*) half3 **cross**(half3 *p0*, half3 *p1*)	Returns the cross product of *p0.xyz* and *p1.xyz*. The *w* component of `float4` result returned will be 0.0.
float **dot**(gentypef *p0*, gentypef *p1*) double **dot**(gentyped *p0*, gentyped *p1*) half **dot**(gentypeh *p0*, gentypeh *p1*)	Compute the dot product of *p0* and *p1*.
float **distance**(gentypef *p0*, gentypef *p1*) double **distance**(gentyped *p0*, gentyped *p1*) half **distance**(gentypeh *p0*, gentypeh *p1*)	Returns the distance between *p0* and *p1*. This is calculated as **length**(*p0* - *p1*).
float **length**(gentypef *p*) double **length**(gentyped *p*) half **length**(gentypeh *p*)	Return the length of vector *p*, i.e., $\sqrt{p.x^2 + p.y^2 + \ldots}$
gentypef **normalize**(gentypef *p*) gentyped **normalize**(gentyped *p*) gentypeh **normalize**(gentypeh *p*)	Returns a vector in the same direction as *p* but with a length of 1.
float **fast_distance**(float*n* *p0*, float*n* *p1*)	Returns **fast_length**(*p0* - *p1*).
float **fast_length**(float*n* *p*)	Returns the length of vector *p* computed as: **half_sqrt**($p.x^2 + p.y^2 + \ldots$)

Function	Description

float*n* **fast_normalize**(float*n* *p*)	Returns a vector in the same direction as *p* but with a length of 1. **fast_normalize** is computed as:
	
	p * **half_rsqrt**($p.x^2 + p.y^2 + ...$)
	
	The result shall be within 8192 ulps error from the infinitely precise result of
	
	``
	if (all(p == 0.0f))
	  result = p;
	else
	  result = p /
	    sqrt(p.x*p.x + p.y*p.y + ...);
	``
	
	with the following exceptions:
	
	1. If the sum of squares is greater than `FLT_MAX` then the value of the floating-point values in the result vector are undefined.
	2. If the sum of squares is less than `FLT_MIN` then the implementation may return back *p*.
	3. If the device is in "denorms are flushed to zero" mode, individual operand elements with magnitude less than **sqrt**(`FLT_MIN`) may be flushed to zero before proceeding with the calculation.

6.15.6. Relational Functions

The relational and equality operators (<, <=, >, >=, !=, ==) can be used with scalar and vector built-in types and produce a scalar or vector signed integer result respectively.

The functions described in the Built-in Scalar and Vector Relational Functions table can be used with built-in scalar or vector types as arguments and return a scalar or vector integer result [56]. The argument type `gentype` refers to the following built-in types: `char`, `charn`, `uchar`, `ucharn`, `short`, `shortn`, `ushort`, `ushortn`, `int`, `intn`, `uint`, `uintn`, `long` [57], `longn`, `ulong`, `ulongn`, `float`, `floatn`, `double` [58], and `doublen`. The argument type `igentype` refers to the built-in signed integer types i.e. `char`, `charn`, `short`, `shortn`, `int`, `intn`, `long` and `longn`. The argument type `ugentype` refers to the built-in unsigned integer types i.e. `uchar`, `ucharn`, `ushort`, `ushortn`, `uint`, `uintn`, `ulong` and `ulongn`. *n* is 2, 3, 4, 8, or 16.

The functions **isequal**, **isnotequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, **islessgreater**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isordered**, **isunordered** and **signbit** described in the following table shall return a 0 if the specified relation is *false* and a 1 if the specified relation is *true* for scalar argument types. These functions shall return a 0 if the specified relation is *false* and a -1 (i.e. all bits set) if the specified relation is *true* for vector argument types.

The relational functions **isequal**, **isgreater**, **isgreaterequal**, **isless**, **islessequal**, and **islessgreater** always return 0 if either argument is not a number (NaN). **isnotequal** returns 1 if one or both arguments are not a number (NaN) and the argument type is a scalar and returns -1 if one or both arguments are not a number (NaN) and the argument type is a vector.

Table 18. Built-in Scalar and Vector Relational Functions

Function	Description
int **isequal**(float *x*, float *y*) int*n* **isequal**(float*n* *x*, float*n* *y*) int **isequal**(double *x*, double *y*) long*n* **isequal**(double*n* *x*, double*n* *y*) int **isequal**(half *x*, half *y*) short*n* **isequal**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* == *y*.
int **isnotequal**(float *x*, float *y*) int*n* **isnotequal**(float*n* *x*, float*n* *y*) int **isnotequal**(double *x*, double *y*) long*n* **isnotequal**(double*n* *x*, double*n* *y*) int **isnotequal**(half *x*, half *y*) short*n* **isnotequal**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* != *y*.
int **isgreater**(float *x*, float *y*) int*n* **isgreater**(float*n* *x*, float*n* *y*) int **isgreater**(double *x*, double *y*) long*n* **isgreater**(double*n* *x*, double*n* *y*) int **isgreater**(half *x*, half *y*) short*n* **isgreater**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* > *y*.
int **isgreaterequal**(float *x*, float *y*) int*n* **isgreaterequal**(float*n* *x*, float*n* *y*) int **isgreaterequal**(double *x*, double *y*) long*n* **isgreaterequal**(double*n* *x*, double*n* *y*) int **isgreaterequal**(half *x*, half *y*) short*n* **isgreaterequal**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* >= *y*.
int **isless**(float *x*, float *y*) int*n* **isless**(float*n* *x*, float*n* *y*) int **isless**(double *x*, double *y*) long*n* **isless**(double*n* *x*, double*n* *y*) int **isless**(half *x*, half *y*) short*n* **isless**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* < *y*.

Function	Description
int **islessequal**(float *x*, float *y*) int*n* **islessequal**(float*n* *x*, float*n* *y*) int **islessequal**(double *x*, double *y*) long*n* **islessequal**(double*n* *x*, double*n* *y*) int **islessequal**(half *x*, half *y*) short*n* **islessequal**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of *x* <= *y*.
int **islessgreater**(float *x*, float *y*) int*n* **islessgreater**(float*n* *x*, float*n* *y*) int **islessgreater**(double *x*, double *y*) long*n* **islessgreater**(double*n* *x*, double*n* *y*) int **islessgreater**(half *x*, half *y*) short*n* **islessgreater**(half*n* *x*, half*n* *y*)	Returns the component-wise compare of (*x* < *y*) || (*x* > *y*).
int **isfinite**(float) int*n* **isfinite**(float*n*) int **isfinite**(double) long*n* **isfinite**(double*n*) int **isfinite**(half) short*n* **isfinite**(half*n*)	Test for finite value.
int **isinf**(float) int*n* **isinf**(float*n*) int **isinf**(double) long*n* **isinf**(double*n*) int **isinf**(half) short*n* **isinf**(half*n*)	Test for infinity value (positive or negative).
int **isnan**(float) int*n* **isnan**(float*n*) int **isnan**(double) long*n* **isnan**(double*n*) int **isnan**(half) short*n* **isnan**(half*n*)	Test for a NaN.
int **isnormal**(float) int*n* **isnormal**(float*n*) int **isnormal**(double) long*n* **isnormal**(double*n*) int **isnormal**(half) short*n* **isnormal**(half*n*)	Test for a normal value.

Function	Description
int **isordered**(float *x*, float *y*) int*n* **isordered**(float*n* *x*, float*n* *y*) int **isordered**(double *x*, double *y*) long*n* **isordered**(double*n* *x*, double*n* *y*) int **isordered**(half *x*, half *y*) short*n* **isordered**(half*n* *x*, half*n* *y*)	Test if arguments are ordered. **isordered**() takes arguments *x* and *y*, and returns the result **isequal**(*x*, *x*) && **isequal**(*y*, *y*).
int **isunordered**(float *x*, float *y*) int*n* **isunordered**(float*n* *x*, float*n* *y*) int **isunordered**(double *x*, double *y*) long*n* **isunordered**(double*n* *x*, double*n* *y*) int **isunordered**(half *x*, half *y*) short*n* **isunordered**(half*n* *x*, half*n* *y*)	Test if arguments are unordered. **isunordered**() takes arguments *x* and *y*, returning non-zero if *x* or *y* is NaN, and zero otherwise.
int **signbit**(float *x*) int*n* **signbit**(float*n* *x*) int **signbit**(double *x*) long*n* **signbit**(double*n* *x*) int **signbit**(half *x*) short*n* **signbit**(half*n* *x*)	Test for sign bit. The scalar version of the function returns a 1 if the sign bit in *x* is set else returns 0. The vector version of the function returns the following for each component in *x*: -1 (i.e all bits set) if the sign bit in the float is set else returns 0.
int **any**(igentype *x*) Scalar inputs to **any** are deprecated by OpenCL C version 3.0.	Returns 1 if the most significant bit of *x* is set for scalar inputs, or if the most significant bit is set for any component of *x* for vector inputs; otherwise returns 0.
int **all**(igentype *x*) Scalar inputs to **all** are deprecated by OpenCL C version 3.0.	Returns 1 if the most significant bit of *x* is set for scalar inputs, or if the most significant bit is set for all components of *x* for vector inputs; otherwise returns 0.
gentype **bitselect**(gentype *a*, gentype *b*, gentype *c*)	Each bit of the result is the corresponding bit of *a* if the corresponding bit of *c* is 0. Otherwise it is the corresponding bit of *b*.
gentype **select**(gentype *a*, gentype *b*, igentype *c*) gentype **select**(gentype *a*, gentype *b*, ugentype *c*)	For each component of a vector type, *result[i]* = if MSB of *c[i]* is set ? *b[i]* : *a[i]*. For a scalar type, *result* = *c* ? *b* : *a*. `igentype` and `ugentype` must have the same number of elements and bits as `gentype` [59].

6.15.7. Vector Data Load and Store Functions

The Built-in Vector Data Load and Store Functions table describes the list of supported functions that allow you to read and write vector types from a pointer to memory.

The generic type name `gentype` indicates that the function can take any of

- `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long` [60] or `ulong`
- `float` or `double` [42]
- `half` [61]

All functions taking or returning `half` types are supported only when the `cl_khr_fp16` extension macro is supported.

as the type for the arguments.

The generic type name `gentypen` indicates an *n*-element vector of `gentype` elements.

The generic type name `halfn` indicates an *n*-element vector of `half` elements.

The suffix *n* is also used in the function names (i.e. **vload*n***, **vstore*n*** etc.), where *n* = 2, 3 [62], 4, 8 or 16.

Table 19. Built-in Vector Data Load and Store Functions

Function	Description
gentype*n* **vload*n***(size_t *offset*, const __global gentype *p*) gentype*n* **vload*n***(size_t *offset*, const __local gentype *p*) gentype*n* **vload*n***(size_t *offset*, const __constant gentype *p*) gentype*n* **vload*n***(size_t *offset*, const __private gentype *p*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: gentype*n* **vload*n***(size_t *offset*, const gentype *p*)	Return `sizeof(gentypen)` bytes of data, where the first `(n * sizeof(gentype))` bytes are read from the address computed as `(p + (offset * n))`. The computed address must be 8-bit aligned if `gentype` is `char` or `uchar`; 16-bit aligned if `gentype` is `half`, `short` or `ushort`; 32-bit aligned if `gentype` is `int`, `uint`, or `float`; and 64-bit aligned if `gentype` is `long` or `ulong`.

Function	Description
void **vstore*n***(gentype*n* *data*, size_t *offset*, __global gentype *p*) void **vstore*n***(gentype*n* *data*, size_t *offset*, __local gentype *p*) void **vstore*n***(gentype*n* *data*, size_t *offset*, __private gentype *p*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstore*n***(gentype*n* *data*, size_t *offset*, gentype *p*)	Write `n * sizeof(gentype)` bytes given by *data* to the address computed as `(p + (offset * n))`. The computed address must be 8-bit aligned if `gentype` is `char` or `uchar`; 16-bit aligned if `gentype` is `half`, `short` or `ushort`; 32-bit aligned if `gentype` is `int`, `uint`, or `float`; and 64-bit aligned if `gentype` is `long` or `ulong`.
float **vload_half**(size_t *offset*, const __global half *p*) float **vload_half**(size_t *offset*, const __local half *p*) float **vload_half**(size_t *offset*, const __constant half *p*) float **vload_half**(size_t *offset*, const __private half *p*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float **vload_half**(size_t *offset*, const half *p*)	Read `sizeof(half)` bytes of data from the address computed as `(p + offset)`. The data read is interpreted as a `half` value. The `half` value is converted to a `float` value and the `float` value is returned. The computed read address must be 16-bit aligned.
float*n* **vload_half*n***(size_t *offset*, const __global half *p*) float*n* **vload_half*n***(size_t *offset*, const __local half *p*) float*n* **vload_half*n***(size_t *offset*, const __constant half *p*) float*n* **vload_half*n***(size_t *offset*, const __private half *p*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float*n* **vload_half*n***(size_t *offset*, const half *p*)	Read `(n * sizeof(half))` bytes of data from the address computed as `(p + (offset * n))`. The data read is interpreted as a `halfn` value. The `halfn` value read is converted to a `floatn` value and the `floatn` value is returned. The computed read address must be 16-bit aligned.

Function	Description
void **vstore_half**(float *data*, size_t *offset*, __global half *p) void **vstore_half_rte**(float *data*, size_t *offset*, __global half *p) void **vstore_half_rtz**(float *data*, size_t *offset*, __global half *p) void **vstore_half_rtp**(float *data*, size_t *offset*, __global half *p) void **vstore_half_rtn**(float *data*, size_t *offset*, __global half *p) void **vstore_half**(float *data*, size_t *offset*, __local half *p) void **vstore_half_rte**(float *data*, size_t *offset*, __local half *p) void **vstore_half_rtz**(float *data*, size_t *offset*, __local half *p) void **vstore_half_rtp**(float *data*, size_t *offset*, __local half *p) void **vstore_half_rtn**(float *data*, size_t *offset*, __local half *p) void **vstore_half**(float *data*, size_t *offset*, __private half *p) void **vstore_half_rte**(float *data*, size_t *offset*, __private half *p) void **vstore_half_rtz**(float *data*, size_t *offset*, __private half *p) void **vstore_half_rtp**(float *data*, size_t *offset*, __private half *p) void **vstore_half_rtn**(float *data*, size_t *offset*, __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstore_half**(float *data*, size_t *offset*, half *p) void **vstore_half_rte**(float *data*, size_t *offset*, half *p) void **vstore_half_rtz**(float *data*, size_t *offset*, half *p) void **vstore_half_rtp**(float *data*, size_t *offset*, half *p) void **vstore_half_rtn**(float *data*, size_t *offset*, half *p)	The `float` value given by *data* is first converted to a `half` value using the appropriate rounding mode. The `half` value is then written to the address computed as `(p + offset)`. The computed address must be 16-bit aligned. **vstore_half** uses the default rounding mode. The default rounding mode is round to nearest even.

Function	Description
void **vstore_half*n***(float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rte**(float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtz**(float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtp**(float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtn**(float*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n***(float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rte**(float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtz**(float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtp**(float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtn**(float*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n***(float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rte**(float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtz**(float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtp**(float*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtn**(float*n* *data*, size_t *offset*, __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstore_half*n***(float*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rte**(float*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtz**(float*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtp**(float*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtn**(float*n* *data*, size_t *offset*, half *p)	The `floatn` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. `n * sizeof(half)` bytes from the `halfn` value are then written to the address computed as `(p + (offset * n))`. The computed address must be 16-bit aligned. **vstore_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.

Function	Description
void **vstore_half**(double *data*, size_t *offset*, __global half *p) void **vstore_half_rte**(double *data*, size_t *offset*, __global half *p) void **vstore_half_rtz**(double *data*, size_t *offset*, __global half *p) void **vstore_half_rtp**(double *data*, size_t *offset*, __global half *p) void **vstore_half_rtn**(double *data*, size_t *offset*, __global half *p) void **vstore_half**(double *data*, size_t *offset*, __local half *p) void **vstore_half_rte**(double *data*, size_t *offset*, __local half *p) void **vstore_half_rtz**(double *data*, size_t *offset*, __local half *p) void **vstore_half_rtp**(double *data*, size_t *offset*, __local half *p) void **vstore_half_rtn**(double *data*, size_t *offset*, __local half *p) void **vstore_half**(double *data*, size_t *offset*, __private half *p) void **vstore_half_rte**(double *data*, size_t *offset*, __private half *p) void **vstore_half_rtz**(double *data*, size_t *offset*, __private half *p) void **vstore_half_rtp**(double *data*, size_t *offset*, __private half *p) void **vstore_half_rtn**(double *data*, size_t *offset*, __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstore_half**(double *data*, size_t *offset*, half *p) void **vstore_half_rte**(double *data*, size_t *offset*, half *p) void **vstore_half_rtz**(double *data*, size_t *offset*, half *p) void **vstore_half_rtp**(double *data*, size_t *offset*, half *p) void **vstore_half_rtn**(double *data*, size_t *offset*, half *p)	The `double` value given by *data* is first converted to a `half` value using the appropriate rounding mode. The `half` value is then written to the address computed as `(p + offset)`. The computed address must be 16-bit aligned. **vstore_half** uses the default rounding mode. The default rounding mode is round to nearest even.

Function	Description
void **vstore_half*n***(double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rte**(double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtz**(double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtp**(double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n*_rtn**(double*n* *data*, size_t *offset*, __global half *p) void **vstore_half*n***(double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rte**(double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtz**(double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtp**(double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n*_rtn**(double*n* *data*, size_t *offset*, __local half *p) void **vstore_half*n***(double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rte**(double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtz**(double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtp**(double*n* *data*, size_t *offset*, __private half *p) void **vstore_half*n*_rtn**(double*n* *data*, size_t *offset*, __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstore_half*n***(double*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rte**(double*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtz**(double*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtp**(double*n* *data*, size_t *offset*, half *p) void **vstore_half*n*_rtn**(double*n* *data*, size_t *offset*, half *p)	The `doublen` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. `n * sizeof(half)` bytes from the `halfn` value are then written to the address computed as `(p + (offset * n))`. The computed address must be 16-bit aligned. **vstore_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.
float*n* **vloada_half*n***(size_t *offset*, const __global half *p) float*n* **vloada_half*n***(size_t *offset*, const __local half *p) float*n* **vloada_half*n***(size_t *offset*, const __constant half *p) float*n* **vloada_half*n***(size_t *offset*, const __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: float*n* **vloada_half*n***(size_t *offset*, const half *p)	For n = 2, 4, 8 and 16, read `sizeof(halfn)` bytes of data from the address computed as (*p* + (*offset* * *n*)). The data read is interpreted as a `halfn` value. The `halfn` value read is converted to a `floatn` value and the `floatn` value is returned. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, **vloada_half3** reads a `half3` from the address computed as `(p + (offset * 4))` and returns a `float3`. The computed address must be aligned to `sizeof(half)` * 4 bytes.

Function	Description
void **vstorea_half*n***(float*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rte**(float*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtz**(float*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtp**(float*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n*_rtn**(float*n* *data*, size_t *offset*, __global half *p) void **vstorea_half*n***(float*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rte**(float*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtz**(float*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtp**(float*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n*_rtn**(float*n* *data*, size_t *offset*, __local half *p) void **vstorea_half*n***(float*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rte**(float*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtz**(float*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtp**(float*n* *data*, size_t *offset*, __private half *p) void **vstorea_half*n*_rtn**(float*n* *data*, size_t *offset*, __private half *p) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstorea_half*n***(float*n* *data*, size_t *offset*, half *p) void **vstorea_half*n*_rte**(float*n* *data*, size_t *offset*, half *p) void **vstorea_half*n*_rtz**(float*n* *data*, size_t *offset*, half *p) void **vstorea_half*n*_rtp**(float*n* *data*, size_t *offset*, half *p) void **vstorea_half*n*_rtn**(float*n* *data*, size_t *offset*, half *p)	The `floatn` value given by *data* is converted to a `halfn` value using the appropriate rounding mode. For n = 2, 4, 8 and 16, the `halfn` value is written to the address computed as `(p + (offset * n))`. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, the `half3` value is written to the address computed as `(p + (offset * 4))`. The computed address must be aligned to `sizeof(half) * 4` bytes. **vstorea_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.

Function	Description
void **vstorea_half*n***(double*n* *data*, size_t *offset*, __global half *p*) void **vstorea_half*n*_rte**(double*n* *data*, size_t *offset*, __global half *p*) void **vstorea_half*n*_rtz**(double*n* *data*, size_t *offset*, __global half *p*) void **vstorea_half*n*_rtp**(double*n* *data*, size_t *offset*, __global half *p*) void **vstorea_half*n*_rtn**(double*n* *data*, size_t *offset*, __global half *p*) void **vstorea_half*n***(double*n* *data*, size_t *offset*, __local half *p*) void **vstorea_half*n*_rte**(double*n* *data*, size_t *offset*, __local half *p*) void **vstorea_half*n*_rtz**(double*n* *data*, size_t *offset*, __local half *p*) void **vstorea_half*n*_rtp**(double*n* *data*, size_t *offset*, __local half *p*) void **vstorea_half*n*_rtn**(double*n* *data*, size_t *offset*, __local half *p*) void **vstorea_half*n***(double*n* *data*, size_t *offset*, __private half *p*) void **vstorea_half*n*_rte**(double*n* *data*, size_t *offset*, __private half *p*) void **vstorea_half*n*_rtz**(double*n* *data*, size_t *offset*, __private half *p*) void **vstorea_half*n*_rtp**(double*n* *data*, size_t *offset*, __private half *p*) void **vstorea_half*n*_rtn**(double*n* *data*, size_t *offset*, __private half *p*) For OpenCL C 2.0, or OpenCL C 3.0 or newer with the `__opencl_c_generic_address_space` feature: void **vstorea_half*n***(double*n* *data*, size_t *offset*, half *p*) void **vstorea_half*n*_rte**(double*n* *data*, size_t *offset*, half *p*) void **vstorea_half*n*_rtz**(double*n* *data*, size_t *offset*, half *p*) void **vstorea_half*n*_rtp**(double*n* *data*, size_t *offset*, half *p*) void **vstorea_half*n*_rtn**(double*n* *data*, size_t *offset*, half *p*)	The `doublen` value is converted to a `halfn` value using the appropriate rounding mode. For n = 2, 4, 8 or 16, the `halfn` value is written to the address computed as `(p + (offset * n))`. The computed address must be aligned to `sizeof(halfn)` bytes. For n = 3, the `half3` value is written to the address computed as `(p + (offset * 4))`. The computed address must be aligned to `sizeof(half) * 4` bytes. **vstorea_half*n*** uses the default rounding mode. The default rounding mode is round to nearest even.

The results of vector data load and store functions are undefined if the address being read from or written to is not correctly aligned as described in Built-in Vector Data Load and Store Functions. The pointer argument p can be a pointer to `global`, `local`, or `private` memory for store functions described in Built-in Vector Data Load and Store Functions. The pointer argument p can be a pointer to `global`, `local`, `constant`, or `private` memory for load functions described in Built-in Vector Data Load and Store Functions.

The vector data load and store functions variants that take pointer arguments which point to the generic address space are also supported.

6.15.8. Synchronization Functions

The following table describes built-in functions to synchronize the work-items in a work-group.

Table 20. Built-in Work-group Synchronization Functions

Function	Description
void **barrier**(cl_mem_fence_flags *flags*) For OpenCL C 2.0 or newer, as an alias for **barrier**: void **work_group_barrier**(cl_mem_fence_flags *flags*) void **work_group_barrier**(cl_mem_fence_flags *flags*, memory_scope *scope*)	For these functions, if any work-item in a work-group encounters a barrier, the barrier must be encountered by all work-items in the work-group before any are allowed to continue execution beyond the barrier. If the barrier is inside a conditional statement, then all work-items in the work-group must enter the conditional if any work-item in the work-group enters the conditional statement and executes the barrier. If the barrier is inside a loop, then all work-items in the work-group must execute the barrier on each iteration of the loop if any work-item executes the barrier on that iteration. The **barrier** and **work_group_barrier** functions can specify which memory operations become visible to the appropriate memory scope identified by *scope* [63]. The *flags* argument specifies the memory address spaces. This is a bitfield and can be set to 0 or a combination of the following values OR'ed together. When these flags are OR'ed together the barrier acts as a combined barrier for all address spaces specified by the flags ordering memory accesses both within and across the specified address spaces. For **barrier** and the **work_group_barrier** variant that does not take a memory scope, the *scope* is `memory_scope_work_group`. `CLK_LOCAL_MEM_FENCE` - ensure that all `local` memory accesses become visible to all work-items in the work-group. Note that the value of *scope* is ignored as the memory scope is always `memory_scope_work_group`. `CLK_GLOBAL_MEM_FENCE` - ensure that all `global` memory accesses become visible to the appropriate memory scope as given by *scope*. `CLK_IMAGE_MEM_FENCE` - ensure that all image memory accesses become visible to the appropriate scope given by *scope*. The value of *scope* must be `memory_scope_work_group`. The values of *flags* and *scope* must be the same for all work-items in the work-group.

The functionality described in the following table requires support for the `cl_khr_subgroups` extension, or OpenCL 3.0 or newer and the `__opencl_c_subgroups` feature.

The following table describes built-in functions to synchronize the work-items in a sub-group.

Table 21. Built-in Sub-Group Synchronization Functions

Function	Description
void **sub_group_barrier**(cl_mem_fence_flags *flags*) void **sub_group_barrier**(cl_mem_fence_flags *flags*, memory_scope *scope*)	For these functions, if any work-item in a sub-group encounters a **sub_group_barrier**, the barrier must be encountered by all work-items in the sub-group before any are allowed to continue execution beyond the barrier. If **sub_group_barrier** is inside a conditional statement, then all work-items within the sub-group must enter the conditional if any work-item in the sub-group enters the conditional statement and executes the **sub_group_barrier**. If the **sub_group_barrier** is inside a loop, then all work-items in the sub-group must execute the barrier on each iteration of the loop if any work-item executes the barrier on that iteration. The **sub_group_barrier** function can specify which memory operations become visible to the appropriate memory scope identified by *scope*. The *flags* argument specifies the memory address spaces. This is a bitfield and can be set to 0 or a combination of the following values OR'ed together. When these flags are OR'ed together the barrier acts as a combined barrier for all address spaces specified by the flags ordering memory accesses both within and across the specified address spaces. For the **sub_group_barrier** variant that does not take a memory scope, the *scope* is `memory_scope_sub_group`. `CLK_LOCAL_MEM_FENCE` - The **sub_group_barrier** function will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory. `CLK_GLOBAL_MEM_FENCE` - The **sub_group_barrier** function will queue a memory fence to ensure correct ordering of memory operations to global memory. This can be useful when work-items, for example, write to buffer objects and then want to read the updated data from these buffer objects. `CLK_IMAGE_MEM_FENCE` - The **sub_group_barrier** function will queue a memory fence to ensure correct ordering of memory operations to image objects. This can be useful when work-items, for example, write to image objects and then want to read the updated data from these image objects. The value of *scope* must match requirements of the atomic restrictions section.

6.15.9. Legacy Explicit Memory Fence Functions

The memory fence functions described in this sub-section are deprecated by OpenCL C 2.0.

The OpenCL C programming language implements the following explicit memory fence functions to provide ordering between memory operations of a work-item.

Table 22. Built-in Explicit Memory Fence Functions

Function	Description

void **mem_fence**(cl_mem_fence_flags *flags*)	Orders loads and stores of a work-item executing a kernel. This means that loads and stores preceding the **mem_fence** will be committed to memory before any loads and stores following the **mem_fence**.
	
	The *flags* argument specifies the memory address space and can be set to a combination of the following literal values:
	
	`CLK_LOCAL_MEM_FENCE`
	`CLK_GLOBAL_MEM_FENCE`
	
	The value of *flags* must be the same for all work-items in the work-group.

void **read_mem_fence**(cl_mem_fence_flags *flags*)	Read memory barrier that orders only loads.
	
	The *flags* argument specifies the memory address space and can be set to a combination of the following literal values:
	
	`CLK_LOCAL_MEM_FENCE`
	`CLK_GLOBAL_MEM_FENCE`
	
	The value of *flags* must be the same for all work-items in the work-group.

void **write_mem_fence**(cl_mem_fence_flags *flags*)	Write memory barrier that orders only stores.
	
	The *flags* argument specifies the memory address space and can be set to a combination of the following literal values:
	
	`CLK_LOCAL_MEM_FENCE`
	`CLK_GLOBAL_MEM_FENCE`
	
	The value of *flags* must be the same for all work-items in the work-group.

6.15.10. Address Space Qualifier Functions

The functionality described in this section requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

This section describes built-in functions to safely convert from pointers to the generic address space to pointers to named address spaces, and to query the appropriate fence flags for a pointer to the generic address space. We use the generic type name `gentype` to indicate any of the built-in data types supported by OpenCL C or a user defined type.

Table 23. Built-in Address Space Qualifier Functions

Function	Description
global gentype * **to_global**(gentype **ptr*) const global gentype * **to_global**(const gentype **ptr*)	Returns a pointer that points to a region in the `global` address space if **to_global** can cast *ptr* to the `global` address space. Otherwise it returns `NULL`.
local gentype * **to_local**(gentype **ptr*) const local gentype * **to_local**(const gentype **ptr*)	Returns a pointer that points to a region in the `local` address space if **to_local** can cast *ptr* to the local address space. Otherwise it returns `NULL`.
private gentype * **to_private**(gentype **ptr*) const private gentype * **to_private**(const gentype **ptr*)	Returns a pointer that points to a region in the `private` address space if **to_private** can cast *ptr* to the `private` address space. Otherwise it returns `NULL`.
cl_mem_fence_flags **get_fence**(gentype **ptr*) cl_mem_fence_flags **get_fence**(const gentype **ptr*)	Returns a valid memory fence value for *ptr*.

6.15.11. Async Copies From Global to Local Memory, Local to Global Memory, and Prefetch

The OpenCL C programming language implements the following functions that provide asynchronous copies between `global` and local memory and a prefetch from `global` memory.

The async copy and wait group events functions are performed by all work-items in a work-group and therefore must be encountered by all work-items in a work-group executing the kernel with the same argument values, otherwise the results are undefined. This rule applies to ND-ranges implemented with uniform and non-uniform work-groups.

If an async copy or wait group events function is inside a conditional statement then all work-items in the work-group must enter the conditional if any work-item in the work-group enters the conditional statement and executes the async copy or wait group events function.

If an async copy or wait group events function is inside a loop then all work-items in the work-group must execute the async copy or wait group events function on each iteration of the loop if any work-item executes the async copy or wait group events function on that iteration.

The generic type name `gentype` indicates that the function can take any of

- char, charn, uchar, or ucharn
- short, shortn, ushort, or ushortn
- int, intn, uint, or uintn
- long [64], longn, ulong, or ulongn
- float, floatn
- double [65] or doublen
- half [66] or halfn

All functions taking or returning half types are supported only when the cl_khr_fp16 extension macro is supported.

as the type for the arguments unless otherwise stated. *n* is 2, 3 [67], 4, 8, or 16.

Table 24. Built-in Async Copy and Prefetch Functions

Function	Description
event_t **async_work_group_copy**(__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, event_t *event*) event_t **async_work_group_copy**(__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, event_t *event*)	Perform an async copy of *num_gentypes* gentype elements from *src* to *dst*. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. 0 can be implicitly and explicitly cast to event_t type. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.

Function	Description
event_t **async_work_group_strided_copy**(__local gentype **dst*, const __global gentype **src*, size_t *num_gentypes*, size_t *src_stride*, event_t *event*) event_t **async_work_group_strided_copy**(__global gentype **dst*, const __local gentype **src*, size_t *num_gentypes*, size_t *dst_stride*, event_t *event*)	Perform an async gather of *num_gentypes* `gentype` elements from *src* to *dst*. The *src_stride* is the stride in elements for each `gentype` element read from *src*. The *dst_stride* is the stride in elements for each `gentype` element written to *dst*. Returns an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the **async_work_group_strided_copy** with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. 0 can be implicitly and explicitly cast to event_t type. If *event* argument is non-zero, the event object supplied in *event* argument will be returned. This function does not perform any implicit synchronization of source data such as using a **barrier** before performing the copy. The behavior of **async_work_group_strided_copy** is undefined if *src_stride* or *dst_stride* is 0, or if the *src_stride* or *dst_stride* values cause the *src* or *dst* pointers to exceed the upper bounds of the address space during the copy. Requires support for OpenCL C 1.1 or newer.
void **wait_group_events**(int *num_events*, event_t **event_list*)	Wait for events that identify the **async_work_group_copy** operations to complete. The event objects specified in *event_list* will be released after the wait is performed.
void **prefetch**(const __global gentype **p*, size_t *num_gentypes*)	Prefetch `num_gentypes * sizeof(gentype)` bytes into the global cache. The prefetch instruction is applied to a work-item in a work-group and does not affect the functional behavior of the kernel.

Function	Description

`void async_work_group_copy_fence(`
`  cl_mem_fence_flags flags)`
	Orders async copies produced by the work-items of a work-group executing a kernel. Async copies preceding the **async_work_group_copy_fence** must complete their access to the designated memory or memories, including both reads-from and writes-to it, before async copies following the fence are allowed to start accessing these memories. In other words, every async copy preceding the **async_work_group_copy_fence** must happen-before every async copy following the fence, with respect to the designated memory or memories.
	
	The *flags* argument specifies the memory address space and can be set to a combination of the following literal values:
	
	`CLK_LOCAL_MEM_FENCE`
	`CLK_GLOBAL_MEM_FENCE`
	
	The async fence is performed by all work-items in a work-group and this built-in function must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined. This rule applies to ND-ranges implemented with uniform and non-uniform work-groups.
	
	Requires support for the `cl_khr_async_work_group_copy_fence` extension macro.

The kernel must wait for the completion of all async copies using the **wait_group_events** built-in function before exiting; otherwise the behavior is undefined.

6.15.11.1. Extended Async Copy Functions

If the `cl_khr_extended_async_copies` extension macro is supported, additional Built-in Extended Async Copy Functions are provided which interpret the source and destination as 2D or 3D data.

async_work_group_strided_copy is a special case of **async_work_group_copy_2D2D**, namely one which copies a single column to a single line or vice versa. For example:
`async_work_group_strided_copy(dst, src, num_gentypes, src_stride, event)` is

> equal to `async_work_group_copy_2D2D(dst, 0, src, 0, sizeof(gentype), 1, num_gentypes, src_stride, 1, event)`

The functions described in this section support arbitrary `gentype`-based buffers by casting pointers to `void*`.

These functions do not perform any implicit synchronization of source data such as using a **barrier** before performing the copy.

These functions are performed by all work-items in a work-group and must therefore be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the results are undefined.

The *src_offset*, *dst_offset*, *src_total_line_length*, *dst_total_line_length*, *src_total_plane_area* and *dst_total_plane_area* function arguments are expressed in elements.

Both *src_total_line_length* and *dst_total_line_length* describe the number of elements between the beginning of the current line and the beginning of the next line.

Both *src_total_plane_area* and *dst_total_plane_area* describe the number of elements between the beginning of the current plane and the beginning of the next plane.

These functions return an event object that can be used by **wait_group_events** to wait for the async copy to finish. The *event* argument can also be used to associate the async copy with a previous async copy allowing an event to be shared by multiple async copies; otherwise *event* should be zero. If the *event* argument is non-zero, the event object supplied as the *event* argument will be returned.

Table 25. Built-in Extended Async Copy Functions

Function	Description

`event_t async_work_group_copy_2D2D(`
`  __local void *dst,`
`  size_t dst_offset,`
`  const __global void *src,`
`  size_t src_offset,`
`  size_t num_bytes_per_element,`
`  size_t num_elements_per_line,`
`  size_t num_lines,`
`  size_t src_total_line_length,`
`  size_t dst_total_line_length,`
`  event_t event)`
 	
`event_t async_work_group_copy_2D2D(`
`  __global void *dst,`
`  size_t dst_offset,`
`  const __local void *src,`
`  size_t src_offset,`
`  size_t num_bytes_per_element,`
`  size_t num_elements_per_line,`
`  size_t num_lines,`
`  size_t src_total_line_length,`
`  size_t dst_total_line_length,`
`  event_t event)`
	Perform an async copy of (*num_elements_per_line* * *num_lines*) elements of size *num_bytes_per_element* from (*src* + (*src_offset* * *num_bytes_per_element*)) to (*dst* + (*dst_offset* * *num_bytes_per_element*)). All pointer arithmetic is performed with implicit casting to `char*` by the implementation. Each line contains *num_elements_per_line* elements of size *num_bytes_per_element*. After each line of transfer, the *src* address is incremented by *src_total_line_length* elements (i.e. *src_total_line_length* * *num_bytes_per_element* bytes), and the *dst* address is incremented by *dst_total_line_length* elements (i.e. *dst_total_line_length* * *num_bytes_per_element* bytes), for the next line of transfer.
	
	The behavior of **async_work_group_copy_2D2D** is undefined if the source or destination addresses exceed the upper bounds of the address space during the copy.
	
	The behavior of **async_work_group_copy_2D2D** is also undefined if the *src_total_line_length* or *dst_total_line_length* values are smaller than *num_elements_per_line*, i.e. overlapping of lines is undefined.

Function	Description

`event_t async_work_group_copy_3D3D(`
`  __local void *dst,`
`  size_t dst_offset,`
`  const __global void *src,`
`  size_t src_offset,`
`  size_t num_bytes_per_element,`
`  size_t num_elements_per_line,`
`  size_t num_lines,`
`  size_t num_planes,`
`  size_t src_total_line_length,`
`  size_t src_total_plane_area,`
`  size_t dst_total_line_length,`
`  size_t dst_total_plane_area,`
`  event_t event)`
 	
`event_t async_work_group_copy_3D3D(`
`  __global void *dst,`
`  size_t dst_offset,`
`  const __local void *src,`
`  size_t src_offset,`
`  size_t num_bytes_per_element,`
`  size_t num_elements_per_line,`
`  size_t num_lines,`
`  size_t num_planes,`
`  size_t src_total_line_length,`
`  size_t src_total_plane_area,`
`  size_t dst_total_line_length,`
`  size_t dst_total_plane_area,`
`  event_t event)`
	Perform an async copy of *num_elements_per_line* * *num_lines*) * *num_planes*) elements of size *num_bytes_per_element* from (*src* + (*src_offset* * *num_bytes_per_element* to (*dst* + (*dst_offset* * *num_bytes_per_element*)), arranged in *num_planes* planes. All pointer arithmetic is performed with implicit casting to `char*` by the implementation. Each plane contains *num_lines* lines. Each line contains *num_elements_per_line* elements. After each line of transfer, the *src* address is incremented by *src_total_line_length* elements (i.e. *src_total_line_length* * *num_bytes_per_element* bytes), and the *dst* address is incremented by *dst_total_line_length* elements (i.e. *dst_total_line_length* * *num_bytes_per_element* bytes), for the next line of transfer.
	
	The behavior of **async_work_group_copy_3D3D** is undefined if the source or destination addresses exceed the upper bounds of the address space during the copy.
	
	The behavior of **async_work_group_copy_3D3D** is also undefined if the *src_total_line_length* or *dst_total_line_length* values are smaller than *num_elements_per_line*, i.e. overlapping of lines is undefined.
	
	The behavior of **async_work_group_copy_3D3D** is also undefined if *src_total_plane_area* is smaller than (*num_lines* * *src_total_line_length*), or *dst_total_plane_area* is smaller than (*num_lines* * *dst_total_line_length*), i.e. overlapping of planes is undefined.

6.15.12. Atomic Functions

The C11 style atomic functions in this sub-section require support for OpenCL C 2.0 or newer. However, this statement does not apply to the "OpenCL C 1.x Legacy Atomics" descriptions at the end of this sub-section.

The OpenCL C programming language implements a subset of the C11 atomics (refer to section 7.17 of the C11 Specification) and synchronization operations. These operations play a special role in making assignments in one work-item visible to another. A synchronization operation on one or more memory locations is either an acquire operation, a release operation, or both an acquire and

release operation [68]. A synchronization 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 are not synchronization operations, and atomic read-modify-write operations which have special characteristics.

The types include

`memory_order`

which is an enumerated type whose enumerators identify memory ordering constraints;

`memory_scope`

which is an enumerated type whose enumerators identify scope of memory ordering constraints;

`atomic_flag`

which is a 32-bit integer type representing a primitive atomic flag; and several atomic analogs of integer types.

In the following operation definitions:

- An A refers to one of the atomic types.
- A C refers to its corresponding non-atomic type.
- An M refers to the type of the other argument for arithmetic operations. For atomic integer types, M is C.
- The functions not ending in explicit have the same semantics as the corresponding explicit function with `memory_order_seq_cst` for the `memory_order` argument.
- The functions that do not have `memory_scope` argument have the same semantics as the corresponding functions with the `memory_scope` argument set to `memory_scope_device`.

With fine-grained system SVM, sharing happens at the granularity of individual loads and stores anywhere in host memory. Memory consistency is always guaranteed at synchronization points, but to obtain finer control over consistency, the OpenCL atomics functions may be used to ensure that the updates to individual data values made by one unit of execution are visible to other execution units. In particular, when a host thread needs fine control over the consistency of memory that is shared with one or more OpenCL devices, it must use atomic and fence operations that are compatible with the C11 atomic operations.

We can't require C11 atomics since host programs can be implemented in other programming languages and versions of C or C++, but we do require that the host programs use atomics and that those atomics be compatible with those in C11.

6.15.12.1. The `ATOMIC_VAR_INIT` Macro

The `ATOMIC_VAR_INIT` macro expands to a token sequence suitable for initializing an atomic object of a type that is initialization-compatible with value. An atomic object with automatic storage duration that is not explicitly initialized using `ATOMIC_VAR_INIT` is initially in an indeterminate state; however,

the default (zero) initialization for objects with `static` storage duration is guaranteed to produce a valid state.

```
#define ATOMIC_VAR_INIT(C value)
```

This macro can only be used to initialize atomic objects that are declared in program scope in the `global` address space.

Examples:

```
global atomic_int guide = ATOMIC_VAR_INIT(42);
```

Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data-race.

6.15.12.2. The atomic_init Function

The `atomic_init` function non-atomically initializes the atomic object pointed to by *obj* to the value *value*.

```
// Requires OpenCL C 3.0 or newer.
void atomic_init(volatile __global A *obj, C value)
void atomic_init(volatile __local A *obj, C value)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
void atomic_init(volatile A *obj, C value)
```

Examples:

```
local atomic_int guide;
if (get_local_id(0) == 0)
    atomic_init(&guide, 42);
work_group_barrier(CLK_LOCAL_MEM_FENCE);
```

The function variant that uses the generic address space, i.e. no explicit address space is listed, requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.3. Order and Consistency

The enumerated type `memory_order` specifies the detailed regular (non-atomic) memory synchronization operations as defined in section 5.1.2.4 of the C11 Specification, and may provide for operation ordering. The following table lists the enumeration constants:

Memory Order	Additional Notes
memory_order_relaxed	Requires support for OpenCL C 2.0 or newer.
memory_order_acquire	Requires support for OpenCL C 2.0, but in OpenCL C 3.0 or newer some uses require the __opencl_c_atomic_order_acq_rel feature.
memory_order_release	Requires support for OpenCL C 2.0, but in OpenCL C 3.0 or newer some uses require the __opencl_c_atomic_order_acq_rel feature.
memory_order_acq_rel	Requires support for OpenCL C 2.0, but in OpenCL C 3.0 or newer some uses require the __opencl_c_atomic_order_acq_rel feature.
memory_order_seq_cst	Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the __opencl_c_atomic_order_seq_cst feature.

The memory_order can be used when performing atomic operations to global or local memory.

6.15.12.4. Memory Scope

The enumerated type memory_scope specifies whether the memory ordering constraints given by memory_order apply to work-items in a sub-group, work-items in a work-group, or work-items from one or more kernels executing on the device or across devices (in the case of shared virtual memory). The following table lists the enumeration constants:

Memory Scope	Additional Notes
memory_scope_work_item	memory_scope_work_item can only be used with atomic_work_item_fence with flags set to CLK_IMAGE_MEM_FENCE. Requires support for OpenCL C 2.0 or newer.
memory_scope_sub_group	Requires support for the cl_khr_subgroups extension, or OpenCL C 3.0 or newer and the __opencl_c_subgroups feature.
memory_scope_work_group	Requires support for OpenCL C 2.0 or newer.
memory_scope_device	Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device feature.
memory_scope_all_svm_devices	Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_all_devices feature.
memory_scope_all_devices	An alias for memory_scope_all_svm_devices. Requires support for OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_all_devices feature.

6.15.12.5. Fences

The following fence operations are supported.

```
void atomic_work_item_fence(cl_mem_fence_flags flags,
                            memory_order order,
                            memory_scope scope)

// Older syntax memory fences are equivalent to atomic_work_item_fence with the
// same flags parameter, memory_scope_work_group scope, and ordering as follows:
void mem_fence(cl_mem_fence_flags flags)        // memory_order_acq_rel
void read_mem_fence(cl_mem_fence_flags flags)   // memory_order_acquire
void write_mem_fence(cl_mem_fence_flags flags)  // memory_order_release
```

`flags` must be set to `CLK_GLOBAL_MEM_FENCE`, `CLK_LOCAL_MEM_FENCE`, `CLK_IMAGE_MEM_FENCE` or a combination of these values ORed together; otherwise the behavior is undefined. The behavior of calling `atomic_work_item_fence` with `CLK_IMAGE_MEM_FENCE` ORed together with either `CLK_GLOBAL_MEM_FENCE` or `CLK_LOCAL_MEM_FENCE` is equivalent to calling `atomic_work_item_fence` individually for `CLK_IMAGE_MEM_FENCE` and the other flags. Passing both `CLK_GLOBAL_MEM_FENCE` and `CLK_LOCAL_MEM_FENCE` to `atomic_work_item_fence` will synchronize memory operations to both `local` and `global` memory through some shared atomic action, as described in section 3.3.6.2 of the OpenCL Specification.

Depending on the value of order, this operation:

- has no effects, if *order* == `memory_order_relaxed`.
- is an acquire fence, if *order* == `memory_order_acquire`.
- is a release fence, if *order* == `memory_order_release`.
- is both an acquire fence and a release fence, if *order* == `memory_order_acq_rel`.
- is a sequentially consistent acquire and release fence, if *order* == `memory_order_seq_cst`.

For images declared with the `read_write` qualifier, the `atomic_work_item_fence` must be called to make sure that writes to the image by a work-item become visible to that work-item on subsequent reads to that image by that work-item.

The use of memory order and scope enumerations must respect the restrictions section below.

6.15.12.6. Atomic Integer and Floating-point Types

The list of supported atomic type names are:

`atomic_int`

`atomic_uint`

`atomic_long` [69]

`atomic_ulong` [69]

`atomic_float`

`atomic_double` [70]

`atomic_intptr_t` [71]

`atomic_uintptr_t` [71]

`atomic_size_t` [71]

`atomic_ptrdiff_t` [71]

Arguments to a kernel can be declared to be a pointer to the above atomic types or the atomic_flag type.

The representation of atomic integer, floating-point and pointer types have the same size as their corresponding regular types. The atomic_flag type must be implemented as a 32-bit integer.

6.15.12.7. Operations on Atomic Types

There are only a few kinds of operations on atomic types, though there are many instances of those kinds. This section specifies each general kind.

6.15.12.7.1. The atomic_store Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
void atomic_store(volatile __global A *object, C desired)
void atomic_store(volatile __local A *object, C desired)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
void atomic_store(volatile A *object, C desired)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
void atomic_store_explicit(volatile __global A *object,
                           C desired,
                           memory_order order)
void atomic_store_explicit(volatile __local A *object,
                           C desired,
                           memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
// features.
void atomic_store_explicit(volatile A *object,
                           C desired,
                           memory_order order)

// Requires OpenCL C 3.0 or newer.
void atomic_store_explicit(volatile __global A *object,
```

```
                            C desired,
                            memory_order order,
                            memory_scope scope)
void atomic_store_explicit(volatile __local A *object,
                            C desired,
                            memory_order order,
                            memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
void atomic_store_explicit(volatile A *object,
                            C desired,
                            memory_order order,
                            memory_scope scope)
```

The *order* argument shall not be `memory_order_acquire`, nor `memory_order_acq_rel`. Atomically replace the value pointed to by *object* with the value of *desired*. Memory is affected according to the value of *order*.

The non-explicit `atomic_store` function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.2. The atomic_load Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
C atomic_load(volatile __global A *object)
C atomic_load(volatile __local A *object)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
C atomic_load(volatile A *object)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
C atomic_load_explicit(volatile __global A *object,
                       memory_order order)
C atomic_load_explicit(volatile __local A *object,
                       memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
```

```
// features.
C atomic_load_explicit(volatile A *object,
                       memory_order order)

// Requires OpenCL C 3.0 or newer.
C atomic_load_explicit(volatile __global A *object,
                       memory_order order,
                       memory_scope scope)
C atomic_load_explicit(volatile __local A *object,
                       memory_order order,
                       memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
C atomic_load_explicit(volatile A *object,
                       memory_order order,
                       memory_scope scope)
```

The *order* argument shall not be `memory_order_release` nor `memory_order_acq_rel`. Memory is affected according to the value of *order*. Atomically returns the value pointed to by *object*.

The non-explicit `atomic_load` function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.3. The atomic_exchange Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
C atomic_exchange(volatile __global A *object, C desired)
C atomic_exchange(volatile __local A *object, C desired)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
C atomic_exchange(volatile A *object, C desired)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
C atomic_exchange_explicit(volatile __global A *object,
                           C desired,
                           memory_order order)
C atomic_exchange_explicit(volatile __local A *object,
                           C desired,
```

```
                            memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
// feature.
C atomic_exchange_explicit(volatile A *object,
                           C desired,
                           memory_order order)

// Requires OpenCL C 3.0 or newer.
C atomic_exchange_explicit(volatile __global A *object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
C atomic_exchange_explicit(volatile __local A *object,
                           C desired,
                           memory_order order,
                           memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
C atomic_exchange_explicit(volatile A *object,
                           C desired,
                           memory_order order,
                           memory_scope scope)
```

Atomically replace the value pointed to by `object` with `desired`. Memory is affected according to the value of `order`. These operations are read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Atomically returns the value pointed to by `object` immediately before the effects.

The non-explicit `atomic_exchange` function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.4. The atomic_compare_exchange Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_strong(
    volatile __global A *object,
    __global C *expected, C desired)
bool atomic_compare_exchange_strong(
```

```
    volatile __global A *object,
    __local C *expected, C desired)
bool atomic_compare_exchange_strong(
    volatile __global A *object,
    __private C *expected, C desired)
bool atomic_compare_exchange_strong(
    volatile __local A *object,
    __global C *expected, C desired)
bool atomic_compare_exchange_strong(
    volatile __local A *object,
    __local C *expected, C desired)
bool atomic_compare_exchange_strong(
    volatile __local A *object,
    __private C *expected, C desired)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_strong(
    volatile A *object,
    C *expected, C desired)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
    __local C *expected,
```

```
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and
// __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_strong_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure)

// Requires OpenCL C 3.0 or newer.
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_strong_explicit(
    volatile __global A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
```

```
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_strong_explicit(
    volatile __local A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
bool atomic_compare_exchange_strong_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)

// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_weak(
    volatile __global A *object,
    __global C *expected, C desired)
bool atomic_compare_exchange_weak(
    volatile __global A *object,
    __local C *expected, C desired)
bool atomic_compare_exchange_weak(
    volatile __global A *object,
    __private C *expected, C desired)
bool atomic_compare_exchange_weak(
    volatile __local A *object,
    __global C *expected, C desired)
bool atomic_compare_exchange_weak(
    volatile __local A *object,
    __local C *expected, C desired)
bool atomic_compare_exchange_weak(
    volatile __local A *object,
    __private C *expected, C desired)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_weak(
    volatile A *object,
    C *expected, C desired)
```

```
// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
bool atomic_compare_exchange_weak_explicit(
    volatile __global A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_weak_explicit(
    volatile __global A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_weak_explicit(
    volatile __global A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and
// __opencl_c_atomic_scope_device features.
bool atomic_compare_exchange_weak_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure)

// Requires OpenCL C 3.0 or newer.
bool atomic_compare_exchange_weak_explicit(
```

```
    volatile __global A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_weak_explicit(
    volatile __global A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_weak_explicit(
    volatile __global A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __global C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __local C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)
bool atomic_compare_exchange_weak_explicit(
    volatile __local A *object,
    __private C *expected,
    C desired,
    memory_order success,
    memory_order failure,
    memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
bool atomic_compare_exchange_weak_explicit(
    volatile A *object,
    C *expected,
    C desired,
    memory_order success,
    memory_order failure,
```

```
    memory_scope scope)
```

The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the `success` argument. Atomically, compares the value pointed to by `object` for equality with that in `expected`, and if *true*, replaces the value pointed to by `object` with `desired`, and if *false*, updates the value in `expected` with the value pointed to by `object`. Further, if the comparison is *true*, memory is affected according to the value of `success`, and if the comparison is *false*, memory is affected according to the value of `failure`. If the comparison is *true*, these operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Otherwise, these operations are atomic load operations.

The effect of the compare-and-exchange operations is

```
if (memcmp(object, expected, sizeof(*object)) == 0) {
    memcpy(object, &desired, sizeof(*object));
} else {
    memcpy(expected, object, sizeof(*object));
}
```

The weak compare-and-exchange operations may fail spuriously [72]. That is, even when the contents of memory referred to by `expected` and `object` are equal, it may return zero and store back to `expected` the same memory contents that were originally there.

These generic functions return the result of the comparison.

The non-explicit `atomic_compare_exchange_strong` and `atomic_compare_exchange_weak` functions requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.5. The atomic_fetch and modify Functions

The following operations perform arithmetic and bitwise computations. All of these operations are applicable to an object of any atomic integer type. The key, operator, and computation correspondence is given in table below:

key	op	computation
`add`	+	addition
`sub`	-	subtraction
`or`	|	bitwise inclusive or

key	op	computation
xor	^	bitwise exclusive or
and	&	bitwise and
min	min	compute min
max	max	compute max

For **atomic_fetch** and modify functions with **key** = add or sub on atomic types atomic_intptr_t and atomic_uintptr_t, M is ptrdiff_t. For **atomic_fetch** and modify functions with **key** = or, xor, and, min and max on atomic type atomic_intptr_t, M is intptr_t, and on atomic type atomic_uintptr_t, M is uintptr_t.

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
C atomic_fetch_key(volatile __global A *object, M operand)
C atomic_fetch_key(volatile __local A *object, M operand)

// Requires OpenCL C 2.0, or all of the __opencl_c_generic_address_space,
// __opencl_c_atomic_order_seq_cst and __opencl_c_atomic_scope_device features.
C atomic_fetch_key(volatile A *object, M operand)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device feature.
C atomic_fetch_key_explicit(volatile __global A *object,
                            M operand,
                            memory_order order)
C atomic_fetch_key_explicit(volatile __local A *object,
                            M operand,
                            memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
// features.
C atomic_fetch_key_explicit(volatile A *object,
                            M operand,
                            memory_order order)

// Requires OpenCL C 3.0 or newer.
C atomic_fetch_key_explicit(volatile __global A *object,
                            M operand,
                            memory_order order,
                            memory_scope scope)
C atomic_fetch_key_explicit(volatile __local A *object,
                            M operand,
                            memory_order order,
                            memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
```

```
C atomic_fetch_key_explicit(volatile A *object,
                            M operand,
                            memory_order order,
                            memory_scope scope)
```

Atomically replaces the value pointed to by `object` with the result of the computation applied to the value pointed to by `object` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow; there are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior. Returns atomically the value pointed to by `object` immediately before the effects.

The non-explicit `atomic_fetch_key` functions require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.6. Atomic Flag Type and Operations

The `atomic_flag` type provides the classic test-and-set functionality. It has two states, *set* (value is non-zero) and *clear* (value is 0).

In OpenCL C 2.0 Operations on an object of type `atomic_flag` shall be lock-free, in OpenCL C 3.0 or newer they may be lock-free.

The macro `ATOMIC_FLAG_INIT` may be used to initialize an `atomic_flag` to the *clear* state. An `atomic_flag` that is not explicitly initialized with `ATOMIC_FLAG_INIT` is initially in an indeterminate state.

This macro can only be used for atomic objects that are declared in program scope in the `global` address space with the `atomic_flag` type.

Example:

```
global atomic_flag guard = ATOMIC_FLAG_INIT;
```

6.15.12.7.7. The atomic_flag_test_and_set Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
bool atomic_flag_test_and_set(
    volatile __global atomic_flag *object)
```

```
bool atomic_flag_test_and_set(
    volatile __local atomic_flag *object)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
bool atomic_flag_test_and_set(
    volatile atomic_flag *object)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
bool atomic_flag_test_and_set_explicit(
    volatile __global atomic_flag *object,
    memory_order order)
bool atomic_flag_test_and_set_explicit(
    volatile __local atomic_flag *object,
    memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
// features.
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag *object,
    memory_order order)

// Requires OpenCL C 3.0 or newer.
bool atomic_flag_test_and_set_explicit(
    volatile __global atomic_flag *object,
    memory_order order,
    memory_scope scope)
bool atomic_flag_test_and_set_explicit(
    volatile __local atomic_flag *object,
    memory_order order,
    memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag *object,
    memory_order order,
    memory_scope scope)
```

Atomically sets the value pointed to by `object` to *true*. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (as defined by section 5.1.2.4 of the C11 Specification). Returns atomically the value of the `object` immediately before the effects.

The non-explicit `atomic_flag_test_and_set` function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.7.8. The atomic_flag_clear Functions

```
// Requires OpenCL C 3.0 or newer and both the __opencl_c_atomic_order_seq_cst
// and __opencl_c_atomic_scope_device features.
void atomic_flag_clear(volatile __global atomic_flag *object)
void atomic_flag_clear(volatile __local atomic_flag *object)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and all of the
// __opencl_c_generic_address_space, __opencl_c_atomic_order_seq_cst and
// __opencl_c_atomic_scope_device features.
void atomic_flag_clear(volatile atomic_flag *object)

// Requires OpenCL C 3.0 or newer and the __opencl_c_atomic_scope_device
// feature.
void atomic_flag_clear_explicit(
    volatile __global atomic_flag *object,
    memory_order order)
void atomic_flag_clear_explicit(
    volatile __local atomic_flag *object,
    memory_order order)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and both the
// __opencl_c_generic_address_space and __opencl_c_atomic_scope_device
// features.
void atomic_flag_clear_explicit(
    volatile atomic_flag *object,
    memory_order order)

// Requires OpenCL C 3.0 or newer.
void atomic_flag_clear_explicit(
    volatile __global atomic_flag *object,
    memory_order order,
    memory_scope scope)
void atomic_flag_clear_explicit(
    volatile __local atomic_flag *object,
    memory_order order,
    memory_scope scope)

// Requires OpenCL C 2.0, or OpenCL C 3.0 or newer and the
// __opencl_c_generic_address_space feature.
void atomic_flag_clear_explicit(
    volatile atomic_flag *object,
    memory_order order,
    memory_scope scope)
```

The `order` argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`. Atomically sets the value pointed to by `object` to *false*. Memory is affected according to the value of `order`.

The non-explicit `atomic_flag_clear` function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and both the `__opencl_c_atomic_order_seq_cst` and `__opencl_c_atomic_scope_device` features. For the explicit variants, memory order and scope enumerations must respect the restrictions section below.

The function variants that use the generic address space, i.e. no explicit address space is listed, require support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_generic_address_space` feature.

6.15.12.8. OpenCL C 1.x Legacy Atomics

The atomic functions described in this sub-section require support for OpenCL C 1.1 or newer, and are deprecated by OpenCL C 2.0.

OpenCL C 1.x had support for relaxed atomic operations via built-in functions that could operate on any memory address in `__global` or `__local` spaces. Unlike C11 style atomics these did not require using dedicated atomic types, and instead operated on 32-bit signed integers, 32-bit unsigned integers, and only in the case of **atomic_xchg** additionally single precision floating-point. These were equivalent to atomic operations with `memory_order_relaxed` consistency, and `memory_scope_work_group` scope.

Some implementations may implement legacy atomics with a stricter memory consistency order than `memory_order_relaxed` or a broader scope than `memory_scope_work_group`. This is because all the stricter orders and broader scopes fully satisfy the semantics of the minimum requirements.

Table 26. Legacy Atomic Functions

Function	Description
int **atomic_add**(volatile __global int *p, int *val*) int **atom_add**(volatile __global int *p, int *val*) uint **atomic_add**(volatile __global uint *p, uint *val*) uint **atom_add**(volatile __global uint *p, uint *val*) int **atomic_add**(volatile __local int *p, int *val*) int **atom_add**(volatile __local int *p, int *val*) uint **atomic_add**(volatile __local uint *p, uint *val*) uint **atom_add**(volatile __local uint *p, uint *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *val*) and store result at location pointed by *p*. The function returns *old*.

Function	Description
int **atomic_sub**(volatile __global int *_p_, int _val_) int **atom_sub**(volatile __global int *_p_, int _val_) uint **atomic_sub**(volatile __global uint *_p_, uint _val_) uint **atom_sub**(volatile __global uint *_p_, uint _val_) int **atomic_sub**(volatile __local int *_p_, int _val_) int **atom_sub**(volatile __local int *_p_, int _val_) uint **atomic_sub**(volatile __local uint *_p_, uint _val_) uint **atom_sub**(volatile __local uint *_p_, uint _val_)	Read the 32-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ - _val_) and store result at location pointed by _p_. The function returns _old_.
int **atomic_xchg**(volatile __global int *_p_, int _val_) int **atom_xchg**(volatile __global int *_p_, int _val_) uint **atomic_xchg**(volatile __global uint *_p_, uint _val_) uint **atom_xchg**(volatile __global uint *_p_, uint _val_) float **atomic_xchg**(volatile __global float *_p_, float _val_) int **atomic_xchg**(volatile __local int *_p_, int _val_) int **atom_xchg**(volatile __local int *_p_, int _val_) uint **atomic_xchg**(volatile __local uint *_p_, uint _val_) uint **atom_xchg**(volatile __local uint *_p_, uint _val_) float **atomic_xchg**(volatile __local float *_p_, float _val_)	Swaps the _old_ value stored at location _p_ with new value given by _val_. Returns _old_ value.
int **atomic_inc**(volatile __global int *_p_) int **atom_inc**(volatile __global int *_p_) uint **atomic_inc**(volatile __global uint *_p_) uint **atom_inc**(volatile __global uint *_p_) int **atomic_inc**(volatile __local int *_p_) int **atom_inc**(volatile __local int *_p_) uint **atomic_inc**(volatile __local uint *_p_) uint **atom_inc**(volatile __local uint *_p_)	Read the 32-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ + 1) and store result at location pointed by _p_. The function returns _old_.

Function	Description
int **atomic_dec**(volatile __global int *_p_) int **atom_dec**(volatile __global int *_p_) uint **atomic_dec**(volatile __global uint *_p_) uint **atom_dec**(__global uint *_p_) int **atomic_dec**(volatile __local int *_p_) int **atom_dec**(volatile __local int *_p_) uint **atomic_dec**(volatile __local uint *_p_) uint **atom_dec**(volatile __local uint *_p_)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - 1) and store result at location pointed by *p*. The function returns *old*.
int **atomic_cmpxchg**(volatile __global int *_p_, int *cmp*, int *val*) int **atom_cmpxchg**(volatile __global int *_p_, int *cmp*, int *val*) uint **atomic_cmpxchg**(volatile __global uint *_p_, uint *cmp*, uint *val*) uint **atom_cmpxchg**(volatile __global uint *_p_, uint *cmp*, uint *val*) int **atomic_cmpxchg**(volatile __local int *_p_, int *cmp*, int *val*) int **atom_cmpxchg**(volatile __local int *_p_, int *cmp*, int *val*) uint **atomic_cmpxchg**(volatile __local uint *_p_, uint *cmp*, uint *val*) uint **atom_cmpxchg**(volatile __local uint *_p_, uint *cmp*, uint *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* == *cmp*) ? *val* : *old* and store result at location pointed by *p*. The function returns *old*.
int **atomic_min**(volatile __global int *_p_, int *val*) int **atom_min**(volatile __global int *_p_, int *val*) uint **atomic_min**(volatile __global uint *_p_, uint *val*) uint **atom_min**(volatile __global uint *_p_, uint *val*) int **atomic_min**(volatile __local int *_p_, int *val*) int **atom_min**(volatile __local int *_p_, int *val*) uint **atomic_min**(volatile __local uint *_p_, uint *val*) uint **atom_min**(volatile __local uint *_p_, uint *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute **min**(*old*, *val*) and store minimum value at location pointed by *p*. The function returns *old*.

Function	Description
int **atomic_max**(volatile __global int *_p_, int _val_) int **atom_max**(volatile __global int *_p_, int _val_) uint **atomic_max**(volatile __global uint *_p_, uint _val_) uint **atom_max**(volatile __global uint *_p_, uint _val_) int **atomic_max**(volatile __local int *_p_, int _val_) int **atom_max**(volatile __local int *_p_, int _val_) uint **atomic_max**(volatile __local uint *_p_, uint _val_) uint **atom_max**(volatile __local uint *_p_, uint _val_)	Read the 32-bit value (referred to as _old_) stored at location pointed by _p_. Compute **max**(_old_, _val_) and store maximum value at location pointed by _p_. The function returns _old_.
int **atomic_and**(volatile __global int *_p_, int _val_) int **atom_and**(volatile __global int *_p_, int _val_) uint **atomic_and**(volatile __global uint *_p_, uint _val_) uint **atom_and**(volatile __global uint *_p_, uint _val_) int **atomic_and**(volatile __local int *_p_, int _val_) int **atom_and**(volatile __local int *_p_, int _val_) uint **atomic_and**(volatile __local uint *_p_, uint _val_) uint **atom_and**(volatile __local uint *_p_, uint _val_)	Read the 32-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ & _val_) and store result at location pointed by _p_. The function returns _old_.
int **atomic_or**(volatile __global int *_p_, int _val_) int **atom_or**(volatile __global int *_p_, int _val_) uint **atomic_or**(volatile __global uint *_p_, uint _val_) uint **atom_or**(volatile __global uint *_p_, uint _val_) int **atomic_or**(volatile __local int *_p_, int _val_) int **atom_or**(volatile __local int *_p_, int _val_) uint **atomic_or**(volatile __local uint *_p_, uint _val_) uint **atom_or**(volatile __local uint *_p_, uint _val_)	Read the 32-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ | _val_) and store result at location pointed by _p_. The function returns _old_.

Function	Description
int **atomic_xor**(volatile __global int *p, int *val*) int **atom_xor**(volatile __global int *p, int *val*) uint **atomic_xor**(volatile __global uint *p, uint *val*) uint **atom_xor**(volatile __global uint *p, uint *val*) int **atomic_xor**(volatile __local int *p, int *val*) int **atom_xor**(volatile __local int *p, int *val*) uint **atomic_xor**(volatile __local uint *p, uint *val*) uint **atom_xor**(volatile __local uint *p, uint *val*)	Read the 32-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* ^ *val*) and store result at location pointed by *p*. The function returns *old*.

A subset of the atomic functions described above are also supported in OpenCL 1.0 when appropriate OpenCL extension macros are supported, as described in the Atomic Function Extensions table below.

Table 27. Atomic Function Extensions

Extension Macro	Supported Functions
`cl_khr_global_int32_base_atomics`	**atom_add** **atom_sub** **atom_xchg** **atom_inc** **atom_dec** **atom_cmpxchg** (with __global parameters)
`cl_khr_global_int32_extended_atomics`	**atom_min** **atom_max** **atom_and** **atom_or** **atom_xor** (with __global parameters)
`cl_khr_local_int32_base_atomics`	**atom_add** **atom_sub** **atom_xchg** **atom_inc** **atom_dec** **atom_cmpxchg** (with __local parameters)

Extension Macro	Supported Functions
cl_khr_local_int32_extended_atomics	**atom_min** **atom_max** **atom_and** **atom_or** **atom_xor** (with __local parameters)

6.15.12.9. Legacy 64-Bit Atomic Extensions

Similar to the OpenCL C 1.x Legacy Atomics, atomic functions operating on 64-bit integers are provided by extensions.

If the cl_khr_int64_base_atomics extension macro is supported, it provides the functions described in the Built-in 64-Bit Base Atomic Functions table below.

Table 28. Built-in 64-Bit Base Atomic Functions

Function	Description
long **atom_add** (volatile __global long *p, long *val*) long **atom_add** (volatile __local long *p, long *val*) ulong **atom_add** (volatile __global ulong *p, ulong *val*) ulong **atom_add** (volatile __local ulong *p, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *val*) and store result at location pointed by *p*. The function returns *old*.
long **atom_sub** (volatile __global long *p, long *val*) long **atom_sub** (volatile __local long *p, long *val*) ulong **atom_sub** (volatile __global ulong *p, ulong *val*) ulong **atom_sub** (volatile __local ulong *p, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - *val*) and store result at location pointed by *p*. The function returns *old*.
long **atom_xchg** (volatile __global long *p, long *val*) long **atom_xchg** (volatile __local long *p, long *val*) ulong **atom_xchg** (volatile __global ulong *p, ulong *val*) ulong **atom_xchg** (volatile __local ulong *p, ulong *val*)	Swaps the *old* value stored at location *p* with new value given by *val*. Returns *old* value.
long **atom_inc** (volatile __global long *p) long **atom_inc** (volatile __local long *p) ulong **atom_inc** (volatile __global ulong *p) ulong **atom_inc** (volatile __local ulong *p)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* + *1*) and store result at location pointed by *p*. The function returns *old*.
long **atom_dec** (volatile __global long *p) long **atom_dec** (volatile __local long *p) ulong **atom_dec** (volatile __global ulong *p) ulong **atom_dec** (volatile __local ulong *p)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* - *1*) and store result at location pointed by *p*. The function returns *old*.

Function	Description
long **atom_cmpxchg** (volatile __global long *_p_, long _cmp_, long _val_) long **atom_cmpxchg** (volatile __local long *_p_, long _cmp_, long _val_) ulong **atom_cmpxchg** (volatile __global ulong *_p_, ulong _cmp_, ulong _val_) ulong **atom_cmpxchg** (volatile __local ulong *_p_, ulong _cmp_, ulong _val_)	Read the 64-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ == _cmp_) ? _val_ : _old_ and store result at location pointed by _p_. The function returns _old_.

If the `cl_khr_int64_extended_atomics` extension macro is supported, it provides the functions described in the Built-in 64-Bit Extended Atomic Functions table below.

Table 29. Built-in 64-Bit Extended Atomic Functions

Function	Description
long **atom_min** (volatile __global long *_p_, long _val_) long **atom_min** (volatile __local long *_p_, long _val_) ulong **atom_min** (volatile __global ulong *_p_, ulong _val_) ulong **atom_min** (volatile __local ulong *_p_, ulong _val_)	Read the 64-bit value (referred to as _old_) stored at location pointed by _p_. Compute **min**(_old_, _val_) and store minimum value at location pointed by _p_. The function returns _old_.
long **atom_max** (volatile __global long *_p_, long _val_) long **atom_max** (volatile __local long *_p_, long _val_) ulong **atom_max** (volatile __global ulong *_p_, ulong _val_) ulong **atom_max** (volatile __local ulong *_p_, ulong _val_)	Read the 64-bit value (referred to as _old_) stored at location pointed by _p_. Compute **max**(_old_, _val_) and store maximum value at location pointed by _p_. The function returns _old_.
long **atom_and** (volatile __global long *_p_, long _val_) long **atom_and** (volatile __local long *_p_, long _val_) ulong **atom_and** (volatile __global ulong *_p_, ulong _val_) ulong **atom_and** (volatile __local ulong *_p_, ulong _val_)	Read the 64-bit value (referred to as _old_) stored at location pointed by _p_. Compute (_old_ & val) and store result at location pointed by _p_. The function returns _old_.

Function	Description
long **atom_or** (volatile __global long *p, long *val*) long **atom_or** (volatile __local long *p, long *val*) ulong **atom_or** (volatile __global ulong *p, ulong *val*) ulong **atom_or** (volatile __local ulong *p, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* | val) and store result at location pointed by *p*. The function returns *old*.
long **atom_xor** (volatile __global long *p, long *val*) long **atom_xor** (volatile __local long *p, long *val*) ulong **atom_xor** (volatile __global ulong *p, ulong *val*) ulong **atom_xor** (volatile __local ulong *p, ulong *val*)	Read the 64-bit value (referred to as *old*) stored at location pointed by *p*. Compute (*old* ^ val) and store result at location pointed by *p*. The function returns *old*.

Atomic operations on 64-bit integers and 32-bit integers (and floats) are also atomic with respect to each other.

6.15.12.10. Restrictions

- All operations on atomic types must be performed using the built-in atomic functions. C11 and C++11 support operators on atomic types. OpenCL C does not support operators with atomic types. Using atomic types with operators should result in a compilation error.
- The `atomic_bool`, `atomic_char`, `atomic_uchar`, `atomic_short`, `atomic_ushort`, `atomic_intmax_t` and `atomic_uintmax_t` types are not supported by OpenCL C.
- OpenCL C 2.0 requires that the built-in atomic functions on atomic types are lock-free. In OpenCL C 3.0 or newer, built-in atomic functions on atomic types may be lock-free.
- The `_Atomic` type specifier and `_Atomic` type qualifier are not supported by OpenCL C.
- The behavior of atomic operations where pointer arguments to the atomic functions refers to an atomic type in the `private` address space is undefined.
- Using `memory_order_acquire` with any built-in atomic function except `atomic_work_item_fence` requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_order_acq_rel` feature.
- Using `memory_order_release` with any built-in atomic function except `atomic_work_item_fence` requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_order_acq_rel` feature.
- Using `memory_order_acq_rel` with any built-in atomic function except `atomic_work_item_fence` requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_order_acq_rel` feature.
- Using `memory_order_seq_cst` with any built-in atomic function requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_order_seq_cst` feature.

- Using `memory_scope_sub_group` with any built-in atomic function requires support for the `cl_khr_subgroups` extension, or OpenCL C 3.0 or newer and the `__opencl_c_subgroups` feature.
- Using `memory_scope_device` requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_scope_device` feature.
- Using `memory_scope_all_svm_devices` requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_atomic_scope_all_devices` feature.
- Using `memory_scope_all_devices` requires support for OpenCL C 3.0 or newer and the `__opencl_c_atomic_scope_all_devices` feature.

6.15.13. Miscellaneous Vector Functions

The OpenCL C programming language implements the following additional built-in vector functions. We use the generic type name `gentypen` (or `gentypem`) to indicate the built-in data types `charn`, `ucharn`, `shortn`, `ushortn`, `intn`, `uintn`, `longn` [73], `ulongn`, `halfn` [74], `floatn`, or `doublen` [75] as the type for the arguments unless otherwise stated. We use the generic name `ugentypen` to indicate the built-in unsigned integer data types. *n* is 2, 4, 8, or 16.

Table 30. Built-in Miscellaneous Vector Functions

Function	Description
int **vec_step**(gentype*n* *a*) int **vec_step**(char3 *a*) int **vec_step**(uchar3 *a*) int **vec_step**(short3 *a*) int **vec_step**(ushort3 *a*) int **vec_step**(half3 *a*) int **vec_step**(int3 *a*) int **vec_step**(uint3 *a*) int **vec_step**(long3 *a*) int **vec_step**(ulong3 *a*) int **vec_step**(float3 *a*) int **vec_step**(double3 *a*) int **vec_step**(*type*)	The **vec_step** built-in function takes a built-in scalar or vector data type argument and returns an integer value representing the number of elements in the scalar or vector. The argument is not evaluated. For all scalar types, **vec_step** returns 1. The **vec_step** built-in functions that take a 3-component vector return 4. **vec_step** may also take a type name as an argument, e.g. **vec_step**(float2) Requires support for OpenCL C 1.1 or newer.

Function	Description

gentype*n* **shuffle**(gentype*m* *x*, ugentype*n* *mask*)
gentype*n* **shuffle2**(gentype*m* *x*, gentype*m* *y*, ugentype*n* *mask*)
	The **shuffle** and **shuffle2** built-in functions construct a permutation of elements from one or two input vectors respectively that are of the same type, returning a vector with the same element type as the input and length that is the same as the shuffle mask. The size of each element in the *mask* must match the size of each element in the result. For **shuffle**, only the **ilogb**(2*m*-1) least significant bits of each *mask* element are considered. For **shuffle2**, only the **ilogb**(2*m*-1)+1 least significant bits of each *mask* element are considered. Other bits in the mask shall be ignored.
	
	The elements of the input vectors are numbered from left to right across one or both of the vectors. For this purpose, the number of elements in a vector is given by **vec_step**(gentype*m*). The shuffle *mask* operand specifies, for each element of the result vector, which element of the one or two input vectors the result element gets.
	
	Requires support for OpenCL C 1.1 or newer.
	
	Examples:
	
	`uint4 mask = (uint4)(3, 2, 1, 0);`
	`float4 a;`
	`float4 r = shuffle(a, mask);`
	
	`uint8 mask = (uint8)(0, 1, 2, 3, 4, 5, 6, 7);`
	`float4 a, b;`
	`float8 r = shuffle2(a, b, mask);`
	
	`uint4 mask;`
	`float8 a;`
	`float4 b;`
	
	`b = shuffle(a, mask);`
	
	Examples that are not valid are:
	
	`uint8 mask;`
	`short16 a;`
	`short8 b;`
	
	`b = shuffle(a, mask); //  not valid`

6.15.14. printf

printf requires support for OpenCL C 1.2.

The OpenCL C programming language implements the **printf** function.

Table 31. Built-in printf Function

Function	Description
int **printf**(constant char *restrict *format*, ...)	The **printf** built-in function writes output to an implementation-defined stream such as stdout under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The **printf** function returns when the end of the format string is encountered. **printf** returns 0 if it was executed successfully and -1 otherwise.

6.15.14.1. printf Output Synchronization

When the event that is associated with a particular kernel invocation is completed, the output of all printf() calls executed by this kernel invocation is flushed to the implementation-defined output stream. Calling **clFinish** on a command-queue flushes all pending output by printf in previously enqueued and completed commands to the implementation-defined output stream. In the case that printf is executed from multiple work-items concurrently, there is no guarantee of ordering with respect to written data. For example, it is valid for the output of a work-item with a global id (0,0,1) to appear intermixed with the output of a work-item with a global id (0,0,4) and so on.

6.15.14.2. printf Format String

The format shall be a character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream. The format is in the constant address space and must be resolvable at compile time, i.e. cannot be dynamically created by the executing program itself.

Each conversion specification is introduced by the character **%**. After the **%**, the following appear in sequence:

- Zero or more *flags* (in any order) that modify the meaning of the conversion specification.
- An optional minimum *field width*. If the converted value has fewer characters than the field

width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of a nonnegative decimal integer [76].

- An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of bytes to be written for **s** conversions. The precision takes the form of a period (**.**) followed by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined.
- An optional *vector specifier*.
- A *length modifier* that specifies the size of the argument. The *length modifier* is required with a vector specifier and together specifies the vector type. Implicit conversions between vector types are disallowed. If the *vector specifier* is not specified, the *length modifier* is optional.
- A *conversion specifier* character that specifies the type of conversion to be applied.

The flag characters and their meanings are:

- The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

+ The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.) [77]

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the *space* and **+** flags both appear, the *space* flag is ignored.

The result is converted to an “alternative form”. For **o** conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For **x** (or **X**) conversion, a nonzero result has **0x** (or **0X**) prefixed to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result of converting a floating-point number always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For **g** and **G** conversions, trailing zeros are **not** removed from the result. For other conversions, the behavior is undefined.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the **0** and **-** flags both appear, the **0** flag is ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag is ignored. For other conversions, the behavior is undefined.

The vector specifier and its meaning is:

v*n* Specifies that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**, **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a vector argument, where *n* is the size of the vector and must be 2, 3, 4, 8 or 16.

The vector value is displayed in the following general form:

value1 C value2 C ... C value*n*

where C is a separator character. The value for this separator character is a comma.

If the vector specifier is not used, the length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `char` or `uchar` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `char` or `uchar` before printing).

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `short` or `ushort` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short` or `unsigned short` before printing).

l (ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `long` or `ulong` argument. The **l** modifier is supported by the full profile. For the embedded profile, the **l** modifier is supported only if 64-bit integers are supported by the device.

z Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `size_t` argument. The **z** length modifier requires support for OpenCL C 3.1 or newer.

t Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `ptrdiff_t` argument. The **t** length modifier requires support for OpenCL C 3.1 or newer.

If the vector specifier is used, the length modifiers and their meanings are:

hh Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `charn` or `ucharn` argument (the argument will not be promoted).

h Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `shortn` or `ushortn` argument (the argument will not be promoted); that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `halfn` [78] argument.

hl This modifier can only be used with the vector specifier. Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `intn` or `uintn` argument; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `floatn` argument.

l(ell) Specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a `longn` or `ulongn` argument; that a following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a `doublen` argument. The **l** modifier is supported by the full profile. For the embedded profile, the **l** modifier is supported only if 64-bit integers or double-precision floating-point are supported by the device.

If a vector specifier appears without a length modifier, the behavior is undefined. The vector data type described by the vector specifier and length modifier must match the data type of the argument; otherwise the behavior is undefined.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

d,i The `int`, `charn`, `shortn`, `intn` or `longn` argument is converted to signed decimal in the style *[-]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

o,u,

x,X The `uint`, `ucharn`, `ushortn`, `uintn` or `ulongn` argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** or **X**) in the style *dddd*; the letters **abcdef** are used for **x** conversion and the letters **ABCDEF** for **X** conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f,F A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted to decimal notation in the style *[-]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity is converted in one of the styles *[-]***inf** or *[-]***infinity** — which style is implementation-defined. A `double`, `halfn`, `floatn` or `doublen` argument representing a NaN is converted in one of the styles *[-]***nan** or *[-]***nan(***n-char-sequence***)** — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The **F** conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of **inf**, **infinity**, or **nan**, respectively [79].

e,E A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in the style *[-]d.ddd* **e**±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The **E** conversion specifier produces a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

g,G A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in style **f** or **e** (or in style **F** or **E** in the case of a **G** conversion specifier), depending on the value converted and the precision. Let *P* equal the precision if nonzero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style **E** would have an exponent of *X*: — if $P > X \geq -4$, the conversion is with style **f** (or **F**) and precision $P - (X + 1)$. — otherwise, the conversion is with style **e *(or *E)** and precision $P - 1$. Finally, unless the # flag is used, any trailing zeros are removed from the fractional portion of the result and the decimal-point character is removed if there is no fractional portion remaining. A `double`, `halfn`, `floatn` or `doublen` **e** argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

a,A A `double`, `halfn`, `floatn` or `doublen` argument representing a floating-point number is converted in the style *[-]***0x***h.hhhh* **p**±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-

point character [80] and the number of hexadecimal digits after it is equal to the precision; if the precision is missing, then the precision is sufficient for an exact representation of the value; if the precision is zero and the # flag is not specified, no decimal point character appears. The letters **abcdef** are used for **a** conversion and the letters **ABCDEF** for **A** conversion. The **A** conversion specifier produces a number with **X** and **P** instead of **x** and **p**. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero. A `double`, `halfn`, `floatn` or `doublen` argument representing an infinity or NaN is converted in the style of an **f** or **F** conversion specifier.

The conversion specifiers **e,E,g,G,a,A** convert a `float` or `half` argument that is a scalar type to a `double` only if double precision is supported. Otherwise, the argument will be a `float` instead of a `double` and the `half` type will be converted to a `float`.

c The `int` argument is converted to an `unsigned char`, and the resulting character is written.

s The argument shall be a literal string [81]. Characters from the literal string array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many bytes are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

p The argument shall be a pointer to **void**. The pointer can refer to a memory region in the `global`, `constant`, `local`, `private`, or generic address space. The value of the pointer is converted to a sequence of printing characters in an implementation-defined manner.

% A **%** character is written. No argument is converted. The complete conversion specification shall be **%%**.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For **a** and **A** conversions, the value is correctly rounded to a hexadecimal floating number with the given precision.

A few examples of printf are given below:

```
float4  f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
uchar4 uc = (uchar4)(0xFA, 0xFB, 0xFC, 0xFD);

printf("f4 = %2.2v4hlf\n", f);
printf("uc = %#v4hhx\n", uc);
```

The above two printf calls print the following:

```
f4 = 1.00,2.00,3.00,4.00
```

```
uc = 0xfa,0xfb,0xfc,0xfd
```

A few examples of valid use cases of printf for the conversion specifier **s** are given below. The argument value must be a pointer to a literal string.

```
kernel void my_kernel( ... )
{
    printf("%s\n", "this is a test string\n");
}
```

A few examples of invalid use cases of printf for the conversion specifier **s** are given below:

```
kernel void my_kernel(global char *s, ... )
{
    printf("%s\n", s);
    constant char *p = "`this is a test string\n`";
    printf("%s\n", p);
    printf("%s\n", &p[3]);
}
```

A few examples of invalid use cases of printf where data types given by the vector specifier and length modifier do not match the argument type are given below:

```
kernel void my_kernel(global char *s, ... )
{
    uint2 ui = (uint2)(0x12345678, 0x87654321);

    printf("unsigned short value = (%#v2hx)\n", ui)
    printf("unsigned char value = (%#v2hhx)\n", ui)
}
```

6.15.14.3. Differences Between OpenCL C and C99 printf

- The asterisk * field width specifier is not supported by OpenCL C.
- The **l** modifier followed by a **c** conversion specifier or **s** conversion specifier is not supported by OpenCL C.
- The **ll**, **j**, and **L** length modifiers are not supported by OpenCL C but are reserved. Prior to OpenCL C 3.1, the **z** and **t** length modifiers additionally are not supported by OpenCL C but are reserved.
- The **n** conversion specifier is not supported by OpenCL C but is reserved.
- OpenCL C adds the optional **v***n* vector specifier to support printing of vector types.
- The conversion specifiers **f**, **F**, **e**, **E**, **g**, **G**, **a**, **A** convert a `float` argument to a `double` only if the `double` data type is supported. Refer to the value of the `CL_DEVICE_DOUBLE_FP_CONFIG` device query. If the `double` data type is not supported, the argument will be a `float` instead of a `double`.

- For the embedded profile, the **l** length modifier is supported only if 64-bit integers are supported.
- In OpenCL C, **printf** returns 0 if it was executed successfully and -1 otherwise vs. C99 where **printf** returns the number of characters printed or a negative value if an output or encoding error occurred.
- In OpenCL C, the conversion specifier **s** can only be used for arguments that are literal strings.

6.15.15. Image Read and Write Functions

The built-in functions defined in this section can only be used with image memory objects. An image memory object can be accessed by specific function calls that read from and/or write to specific locations in the image.

Support for the image built-in functions is optional. If a device supports images then the value of the `CL_DEVICE_IMAGE_SUPPORT` device query) is `CL_TRUE` and the OpenCL C compiler for that device must define the `__IMAGE_SUPPORT__` macro. A compiler for OpenCL C 3.0 or newer for that device must also support the `__opencl_c_images` feature.

Image memory objects that are being read by a kernel should be declared with the `read_only` qualifier. **write_image** calls to image memory objects declared with the read_only qualifier will generate a compilation error. Image memory objects that are being written to by a kernel should be declared with the write_only qualifier. **read_image** calls to image memory objects declared with the `write_only` qualifier will generate a compilation error. **read_image** and **write_image** calls to the same image memory object in a kernel are supported. Image memory objects that are being read and written by a kernel should be declared with the `read_write` qualifier.

The **read_image** calls returns a four component floating-point, integer or unsigned integer color value. The color values returned by **read_image** are identified as *x*, *y*, *z*, *w* where *x* refers to the red component, *y* refers to the green component, *z* refers to the blue component and *w* refers to the alpha component.

6.15.15.1. Samplers

The image read functions take a sampler argument. The sampler can be passed as an argument to the kernel using **clSetKernelArg**, or can be declared in the outermost scope of kernel functions, or it can be a constant variable of type `sampler_t` declared in the program source.

Sampler variables in a program are declared to be of type `sampler_t`. A variable of `sampler_t` type declared in the program source must be initialized with a 32-bit unsigned integer constant, which is interpreted as a bit-field specifying the following properties:

- Addressing Mode
- Filter Mode
- Normalized Coordinates

These properties control how elements of an image object are read by **read_image{f|i|ui}**.

Samplers can also be declared as global constants in the program source using the following syntax.

```
const sampler_t <sampler name> = <value>
```

or

```
constant sampler_t <sampler name> = <value>
```

or

```
__constant sampler_t <sampler_name> = <value>
```

Note that samplers declared using the `constant` qualifier are not counted towards the maximum number of arguments pointing to the constant address space or the maximum size of the `constant` address space allowed per device (i.e. the value of the `CL_DEVICE_MAX_CONSTANT_ARGS` and `CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE` device queries).

The sampler fields are described in the following table.

Table 32. Sampler Descriptor

Sampler State	Description
`<normalized coords>`	Specifies whether the *x*, *y* and *z* coordinates are passed in as normalized or unnormalized values. This must be a literal value and can be one of the following predefined enums: `CLK_NORMALIZED_COORDS_TRUE` or `CLK_NORMALIZED_COORDS_FALSE`. The samplers used with an image in multiple calls to **read_image{f|i|ui}** declared in a kernel must use the same value for <normalized coords>.

Sampler State	Description
`<addressing mode>`	Specifies the image addressing mode, i.e. how out-of-range image coordinates are handled. This must be a literal value and can be one of the following predefined enums: `CLK_ADDRESS_MIRRORED_REPEAT` - Flip the image coordinate at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. `CLK_ADDRESS_REPEAT` - out-of-range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined. `CLK_ADDRESS_CLAMP_TO_EDGE` - out-of-range image coordinates are clamped to the extent. `CLK_ADDRESS_CLAMP` - out-of-range image coordinates will return a border color [82]. `CLK_ADDRESS_NONE` - for this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined. For 1D and 2D image arrays, the addressing mode applies only to the *x* and (*x*, *y*) coordinates. The addressing mode for the coordinate which specifies the array index is always `CLK_ADDRESS_CLAMP_TO_EDGE`.
`<filter mode>`	Specifies the filter mode to use. This must be a literal value and can be one of the following predefined enums: `CLK_FILTER_NEAREST` or `CLK_FILTER_LINEAR`. Refer to the detailed description of these filter modes.

Examples:

```
const sampler_t samplerA = CLK_NORMALIZED_COORDS_TRUE |
                           CLK_ADDRESS_REPEAT |
                           CLK_FILTER_NEAREST;
```

`samplerA` specifies a sampler that uses normalized coordinates, the repeat addressing mode and a nearest filter.

The maximum number of samplers that can be declared in a kernel can be queried using the `CL_DEVICE_MAX_SAMPLERS` token in **clGetDeviceInfo**.

6.15.15.1.1. Determining the Border Color or Value

If `<addressing mode>` in sampler is `CLK_ADDRESS_CLAMP`, then out-of-range image coordinates return the border color. The border color selected depends on the image channel order and can be one of the following values:

- If the image channel order is `CL_A`, `CL_INTENSITY`, `CL_Rx`, `CL_RA`, `CL_RGx`, `CL_RGBx`, `CL_sRGBx`, `CL_ARGB`, `CL_BGRA`, `CL_ABGR`, `CL_RGBA`, `CL_sRGBA` or `CL_sBGRA`, the border color is `(0.0f, 0.0f, 0.0f, 0.0f)`.
- If the image channel order is `CL_R`, `CL_RG`, `CL_RGB`, or `CL_LUMINANCE`, the border color is `(0.0f, 0.0f, 0.0f, 1.0f)`.
- If the image channel order is `CL_DEPTH`, the border value is `0.0f`.

6.15.15.1.2. sRGB Images

The built-in image read functions will perform sRGB to linear RGB conversions if the image is an sRGB image. Likewise, the built-in image write functions perform the linear to sRGB conversion if the image is an sRGB image.

Only the R, G and B components are converted from linear to sRGB and vice-versa. The alpha component is returned as is.

6.15.15.2. Built-in Image Read Functions

The following built-in function calls to read images with a sampler are supported [83].

If the `cl_khr_mipmap_image` extension macro is supported, read functions which do not either

- explicitly specify a level of detail *lod*, or
- compute a level of detail from *gradient* parameters

read from mip level 0 if *image* is a mipmapped image.

Table 33. Built-in Image Read Functions

Function	Description
float4 **read_imagef**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) half4 **read_imageh**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate *(coord.x, coord.y)* to do an element lookup in the 2D image object specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*) uint4 **read_imageui**(read_only image2d_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui**(read_only image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

Function	Description
float4 **read_imagef**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description are undefined.

Function	Description
half4 **read_imageh**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) half4 **read_imageh**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate *(coord.x*, *coord.y*, *coord.z)* to do an elementlookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*) uint4 **read_imageui**(read_only image3d_t *image*, sampler_t *sampler*, int4 *coord*) uint4 **read_imageui**(read_only image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

Function	Description
float4 **read_imagef**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) half4 **read_imageh**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*) uint4 **read_imageui**(read_only image2d_array_t *image*, sampler_t *sampler*, int4 *coord*) uint4 **read_imageui**(read_only image2d_array_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined.

Function	Description
float4 **read_imagef**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) float4 **read_imagef**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*)	Use *coord* to do an element lookup in the 1D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 1.2 or newer.

Function	Description
half4 **read_imageh**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) half4 **read_imageh**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*)	Use *coord* to do an element lookup in the 1D image object specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) int4 **read_imagei**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*) uint4 **read_imageui**(read_only image1d_t *image*, sampler_t *sampler*, int *coord*) uint4 **read_imageui**(read_only image1d_t *image*, sampler_t *sampler*, float *coord*)	Use *coord* to do an element lookup in the 1D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Requires support for OpenCL C 1.2 or newer.

Function	Description
float4 **read_imagef**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 1.2 or newer.

Function	Description
half4 **read_imageh**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) half4 **read_imageh**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. The **read_imageh** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*) uint4 **read_imageui**(read_only image1d_array_t *image*, sampler_t *sampler*, int2 *coord*) uint4 **read_imageui**(read_only image1d_array_t *image*, sampler_t *sampler*, float2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined. The **read_image{i|ui}** calls support a nearest filter only. The filter_mode specified in *sampler* must be set to `CLK_FILTER_NEAREST`; otherwise the values returned are undefined. Furthermore, the **read_image{i|ui}** calls that take integer coordinates must use a sampler with normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Requires support for OpenCL C 1.2 or newer.

Function	Description
float **read_imagef**(read_only image2d_depth_t *image*, sampler_t *sampler*, int2 *coord*) float **read_imagef**(read_only image2d_depth_t *image*, sampler_t *sampler*, float2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D depth image object specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for depth image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

Function	Description
float **read_imagef**(read_only image2d_array_depth_t *image*, sampler_t *sampler*, int4 *coord*) float **read_imagef**(read_only image2d_array_depth_t *image*, sampler_t *sampler*, float4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. The **read_imagef** calls that take integer coordinates must use a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode set to `CLK_ADDRESS_CLAMP_TO_EDGE`, `CLK_ADDRESS_CLAMP` or `CLK_ADDRESS_NONE`; otherwise the values returned are undefined. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

Function	Description

`float4 read_imagef(`
`  read_only image2d_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
 	
`int4 read_imagei(`
`  read_only image2d_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
 	
`uint4 read_imageui(`
`  read_only image2d_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
 	
`float read_imagef(`
`  read_only image2d_depth_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
	Use the coordinate *coord.xy* to do an element lookup in the mip level specified by *lod* in the 2D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

Function	Description

``
float4 read_imagef(
  read_only image2d_t image,
  sampler_t sampler,
  float2 coord,
  float2 gradient_x,
  float2 gradient_y)
 	
int4 read_imagei(
  read_only image2d_t image,
  sampler_t sampler,
  float2 coord,
  float2 gradient_x,
  float2 gradient_y)
 	
uint4 read_imageui(
  read_only image2d_t image,
  sampler_t sampler,
  float2 coord,
  float2 gradient_x,
  float2 gradient_y)
 	
float read_imagef(
  read_only image2d_depth_t image,
  sampler_t sampler,
  float2 coord,
  float2 gradient_x,
  float2 gradient_y)
``
	Use the gradients to compute the lod and coordinate *coord.xy* to do an element lookup in the mip level specified by the computed lod in the 2D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

``
float4 read_imagef(
  read_only image1d_t image,
  sampler_t sampler,
  float coord,
  float lod)
 	
int4 read_imagei(
  read_only image1d_t image,
  sampler_t sampler,
  float coord,
  float lod)
 	
uint4 read_imageui(
  read_only image1d_t image,
  sampler_t sampler,
  float coord,
  float lod)
``
	Use the coordinate *coord* to do an element lookup in the mip level specified by *lod* in the 1D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

Function	Description

`float4 read_imagef(`
`  read_only image1d_t image,`
`  sampler_t sampler,`
`  float coord,`
`  float gradient_x,`
`  float gradient_y)`
 	
`int4 read_imagei(`
`  read_only image1d_t image,`
`  sampler_t sampler,`
`  float coord,`
`  float gradient_x,`
`  float gradient_y)`
 	
`uint4 read_imageui(`
`  read_only image1d_t image,`
`  sampler_t sampler,`
`  float coord,`
`  float gradient_x,`
`  float gradient_y)`
	Use the gradients to compute the lod and coordinate *coord* to do an element lookup in the mip level specified by the computed lod in the 1D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

`float4 read_imagef(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
 	
`int4 read_imagei(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
 	
`uint4 read_imageui(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
	Use the coordinate *coord.xyz* to do an element lookup in the mip level specified by *lod* in the 3D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

Function	Description

`float4 read_imagef(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float4 gradient_x,`
`  float4 gradient_y)`
 	
`int4 read_imagei(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float4 gradient_x,`
`  float4 gradient_y)`
 	
`uint4 read_imageui(`
`  read_only image3d_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float4 gradient_x,`
`  float4 gradient_y)`
	Use the gradients to compute the lod and coordinate *coord.xyz* to do an element lookup in the mip level specified by the computed lod in the 3D image object specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

`float4 read_imagef(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
 	
`int4 read_imagei(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
 	
`uint4 read_imageui(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float lod)`
	Use the coordinate *coord.x* to do an element lookup in the 1D image identified by *coord.x* and mip level specified by *lod* in the 1D image array specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

Function	Description

`float4 read_imagef(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float gradient_x,`
`  float gradient_y)`
 	
`int4 read_imagei(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float gradient_x,`
`  float gradient_y)`
 	
`uint4 read_imageui(`
`  read_only image1d_array_t image,`
`  sampler_t sampler,`
`  float2 coord,`
`  float gradient_x,`
`  float gradient_y)`
	Use the gradients to compute the lod and coordinate *coord.x* to do an element lookup in the mip level specified by the computed lod in the 1D image array specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

`float4 read_imagef(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
 	
`int4 read_imagei(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
 	
`uint4 read_imageui(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
 	
`float read_imagef(`
`  read_only image2d_array_depth_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float lod)`
	Use the coordinate *coord.xy* to do an element lookup in the 2D image identified by *coord.z* and mip level specified by *lod* in the 2D image array specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

Function	Description

`float4 read_imagef(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float2 gradient_x,`
`  float2 gradient_y)`
 	
`int4 read_imagei(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float2 gradient_x,`
`  float2 gradient_y)`
 	
`uint4 read_imageui(`
`  read_only image2d_array_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float2 gradient_x,`
`  float2 gradient_y)`
 	
`float read_imagef(`
`  read_only image2d_array_depth_t image,`
`  sampler_t sampler,`
`  float4 coord,`
`  float2 gradient_x,`
`  float2 gradient_y)`
	Use the gradients to compute the lod coordinate and *coord.xy* to do an element lookup in the 2D image identified by *coord.z* and mip level specified by the computed lod in the 2D image array specified by *image*.
	
	Requires support for the `cl_khr_mipmap_image` extension macro.

If the `cl_khr_mipmap_image` extension macro is supported, `CL_SAMPLER_NORMALIZED_COORDS` must be `CL_TRUE` for built-in functions described in the table above that read from a mipmapped image; otherwise behavior is undefined. The value specified in the *lod* argument is clamped to the minimum of (actual number of mip levels - 1) in the image or the value specified for `CL_SAMPLER_LOD_MAX`.

6.15.15.3. Built-in Image Sampler-less Read Functions

Sampler-less image read functions require support for OpenCL C 1.2 or newer, with some functions requiring support for newer versions of OpenCL C as noted in the table below.

The sampler-less image read functions behave exactly as the corresponding built-in image read functions that take integer coordinates and a sampler with filter mode set to `CLK_FILTER_NEAREST`, normalized coordinates set to `CLK_NORMALIZED_COORDS_FALSE` and addressing mode to `CLK_ADDRESS_NONE`. There is one exception when the *image_channel_data_type* is a floating-point type (such as `CL_FLOAT`). In this exceptional case, when channel data values are denormalized, the sampler-less image read function may return the denormalized data, while the image read function

with a sampler argument may flush the denormalized channel data values to zero.

aQual in the following table refers to one of the access qualifiers. For sampler-less read functions this may be `read_only` or `read_write`.

Table 34. Built-in Image Sampler-less Read Functions

Function	Description
float4 **read_imagef**(*aQual* image2d_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(*aQual* image2d_t *image*, int2 *coord*)	Use the coordinate *(coord.x, coord.y)* to do an element lookup in the 2D image object specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(*aQual* image2d_t *image*, int2 *coord*) uint4 **read_imageui**(*aQual* image2d_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D image object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

Function	Description
float4 **read_imagef**(*aQual* image3d_t *image*, int4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description are undefined.

Function	Description
half4 **read_imageh**(*aQual* image3d_t *image*, int4 *coord*)	Use the coordinate *(coord.x*, *coord.y*, *coord.z)* to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(*aQual* image3d_t *image*, int4 *coord*) uint4 **read_imageui**(*aQual* image3d_t *image*, int4 *coord*)	Use the coordinate (*coord.x*, *coord.y*, *coord.z*) to do an element lookup in the 3D image object specified by *image*. *coord.w* is ignored. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

Function	Description
float4 **read_imagef**(*aQual* image2d_array_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(*aQual* image2d_array_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(*aQual* image2d_array_t *image*, int4 *coord*) uint4 **read_imageui**(*aQual* image2d_array_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

Function	Description
float4 **read_imagef**(*aQual* image1d_t *image*, int *coord*) float4 **read_imagef**(*aQual* image1d_buffer_t *image*, int *coord*)	Use *coord* to do an element lookup in the 1D image or 1D image buffer object specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(*aQual* image1d_t *image*, int *coord*) half4 **read_imageh**(*aQual* image1d_buffer_t *image*, int *coord*)	Use *coord* to do an element lookup in the 1D image or 1D image buffer object specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(*aQual* image1d_t *image*, int *coord*) uint4 **read_imageui**(*aQual* image1d_t *image*, int *coord*) int4 **read_imagei**(*aQual* image1d_buffer_t *image*, int *coord*) uint4 **read_imageui**(*aQual* image1d_buffer_t *image*, int *coord*)	Use *coord* to do an element lookup in the 1D image or 1D image buffer object specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

Function	Description
float4 **read_imagef**(*aQual* image1d_array_t *image*, int2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagef** returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imagef** returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imagef** returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.

Function	Description
half4 **read_imageh**(*aQual* image1d_array_t *image*, int2 *coord*)	Use *coord.x* to do an element lookup in the 2D image identified by *coord.y* in the 2D image array specified by *image*. **read_imageh** returns half-precision floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`. **read_imageh** returns half-precision floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`. **read_imageh** returns half-precision floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT`. Values returned by **read_imageh** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for the `cl_khr_fp16` extension macro.

Function	Description
int4 **read_imagei**(*aQual* image1d_array_t *image*, int2 *coord*) uint4 **read_imageui**(*aQual* image1d_array_t *image*, int2 *coord*)	Use *coord.x* to do an element lookup in the 1D image identified by *coord.y* in the 1D image array specified by *image*. **read_imagei** and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer. **read_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined. **read_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.

Function	Description
float **read_imagef**(*aQual* image2d_depth_t *image*, int2 *coord*)	Use the coordinate (*coord.x*, *coord.y*) to do an element lookup in the 2D depth image object specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.
float **read_imagef**(*aQual* image2d_array_depth_t *image*, int4 *coord*)	Use *coord.xy* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. **read_imagef** returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`. **read_imagef** returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`. Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

Function	Description

`float4 read_imagef(`
`  image2d_msaa_t image,`
`  int2 coord,`
`  int sample)`
	Use the coordinate *(coord.x, coord.y)* and *sample* to do an element lookup in the 2D image object specified by *image*.
	
	read_imagef returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`.
	
	read_imagef returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`.
	
	read_imagef returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`.
	
	Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

Function	Description

``
int4 read_imagei(
  image2d_msaa_t image,
  int2 coord,
  int sample)
 	
uint4 read_imageui(
  image2d_msaa_t image,
  int2 coord,
  int sample)
``
	Use the coordinate *(coord.x, coord.y)* and *sample* to do an element lookup in the 2D image object specified by *image*.
	
	read_imagei and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer.
	
	read_imagei can only be used with image objects created with *image_channel_data_type* set to one of the following values:
	
	`CL_SIGNED_INT8`,
	`CL_SIGNED_INT16` and
	`CL_SIGNED_INT32`.
	
	If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined.
	
	read_imageui can only be used with image objects created with *image_channel_data_type* set to one of the following values:
	
	`CL_UNSIGNED_INT8`,
	`CL_UNSIGNED_INT16` and
	`CL_UNSIGNED_INT32`.
	
	If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

Function	Description

`float4 read_imagef(`
`  image2d_array_msaa_t image,`
`  int4 coord,`
`  int sample)`
	Use *coord.xy* and *sample* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*.
	
	read_imagef returns floating-point values in the range [0.0, 1.0] for image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or `CL_UNORM_INT8`, or `CL_UNORM_INT16`.
	
	read_imagef returns floating-point values in the range [-1.0, 1.0] for image objects created with *image_channel_data_type* set to `CL_SNORM_INT8`, or `CL_SNORM_INT16`.
	
	read_imagef returns floating-point values for image objects created with *image_channel_data_type* set to `CL_HALF_FLOAT` or `CL_FLOAT`.
	
	Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

Function	Description

`int4 read_imagei(`
`  image2d_array_msaa_t image,`
`  int4 coord,`
`  int sample)`
 	
`uint4 read_imageui(`
`  image2d_array_msaa_t image,`
`  int4 coord,`
`  int sample)`
	Use *coord.xy* and *sample* to do an element lookup in the 2D image identified by *coord.z* in the 2D image array specified by *image*.
	
	read_imagei and **read_imageui** return unnormalized signed integer and unsigned integer values respectively. Each channel will be stored in a 32-bit integer.
	
	read_imagei can only be used with image objects created with *image_channel_data_type* set to one of the following values:
	
	`CL_SIGNED_INT8`,
	`CL_SIGNED_INT16` and
	`CL_SIGNED_INT32`.
	
	If the *image_channel_data_type* is not one of the above values, the values returned by **read_imagei** are undefined.
	
	read_imageui can only be used with image objects created with *image_channel_data_type* set to one of the following values:
	
	`CL_UNSIGNED_INT8`,
	`CL_UNSIGNED_INT16` and
	`CL_UNSIGNED_INT32`.
	
	If the *image_channel_data_type* is not one of the above values, the values returned by **read_imageui** are undefined.
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

Function	Description

`float read_imagef(`
`  image2d_msaa_depth_t image,`
`  int2 coord,`
`  int sample)`
	Use the coordinate *(coord.x, coord.y)* and *sample* to do an element lookup in the 2D depth image object specified by *image*.
	
	read_imagef returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`.
	
	read_imagef returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`.
	
	Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

Function	Description

`float read_imagef(`
`  image2d_array_msaa_depth_t image,`
`  int4 coord,`
`  int sample)`
	Use *coord.xy* and *sample* to do an element lookup in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*.
	
	read_imagef returns a floating-point value in the range [0.0, 1.0] for depth image objects created with *image_channel_data_type* set to `CL_UNORM_INT16` or `CL_UNORM_INT24`.
	
	read_imagef returns a floating-point value for depth image objects created with *image_channel_data_type* set to `CL_FLOAT`.
	
	Values returned by **read_imagef** for image objects with *image_channel_data_type* values not specified in the description above are undefined.
	
	Note: When a multisample image is accessed in a kernel, the access takes one vector of integers describing which pixel to fetch and an integer corresponding to the sample numbers describing which sample within the pixel to fetch. *sample* identifies the sample position in the multi-sample image.
	
	For best performance, we recommend that *sample* be a literal value so it is known at compile time and the OpenCL compiler can perform appropriate optimizations for multi-sample reads on the device.
	
	No standard sampling instructions are allowed on the multisample image. Accessing a coordinate outside the image and/or a sample that is outside the number of samples associated with each pixel in the image is undefined
	
	Requires support for the `cl_khr_gl_msaa_sharing` extension macro.

6.15.15.4. Built-in Image Write Functions

The following built-in function calls to write images are supported.

aQual in the following table refers to one of the access qualifiers. For write functions this may be `write_only` or `read_write`.

If the `cl_khr_mipmap_image_writes` extension macro is supported, write functions which do not explicitly specify a level of detail *lod* write to mip level 0 if *image* is a mipmapped image. *mipwidth*, *mipheight*, and *mipdepth* in the table refer to the width, height, and depth of the *image* mip level specified by *lod* respectively; *miplayers* refers to the number of layers in *image*; and *miplevels* refers to the number of mip levels in *image*.

If the `cl_khr_srgb_image_writes` extension macro is supported, the **write_imagef** functions described below may write to sRGB images. Linear to sRGB conversion is performed by the function. Only the R, G, and B components are converted from linear to sRGB; the A component is written as-is.

Table 35. Built-in Image Write Functions

Function	Description
void **write_imagef**(*aQual* image2d_t *image*, int2 *coord*, float4 *color*) void **write_imageh**(*aQual* image2d_t *image*, int2 *coord*, half4 *color*) void **write_imagei**(*aQual* image2d_t *image*, int2 *coord*, int4 *color*) void **write_imageui**(*aQual* image2d_t *image*, int2 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.xy* in the 2D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x* and *coord.y* are considered to be unnormalized coordinates, and must be in the range [0, image width-1] and [0, image height-1] respectively. **write_imagef** and **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imageh**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with *x* and *y* coordinate values that are not in the range [0, image width-1] and [0, image height-1], respectively, is undefined. **write_imageh** requires support for the `cl_khr_fp16` extension macro.

Function	Description
void **write_imagef**(*aQual* image2d_array_t *image*, int4 *coord*, float4 *color*) void **write_imageh**(*aQual* image2d_array_t *image*, int4 *coord*, half4 *color*) void **write_imagei**(*aQual* image2d_array_t *image*, int4 *coord*, int4 *color*) void **write_imageui**(*aQual* image2d_array_t *image*, int4 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.xy* in the 2D image identified by *coord.z* in the 2D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1] and [0, image height-1], and [0, image number of layers-1], respectively. **write_imagef** and **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imageh**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], and [0, image number of layers-1], respectively, is undefined. **write_imageh** requires support for the `cl_khr_fp16` extension macro.

Function	Description
void **write_imagef**(*aQual* image1d_t *image*, int *coord*, float4 *color*) void **write_imageh**(*aQual* image1d_t *image*, int *coord*, half4 *color*) void **write_imagei**(*aQual* image1d_t *image*, int *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_t *image*, int *coord*, uint4 *color*) void **write_imagef**(*aQual* image1d_buffer_t *image*, int *coord*, float4 *color*) void **write_imageh**(*aQual* image1d_buffer_t *image*, int *coord*, half4 *color*) void **write_imagei**(*aQual* image1d_buffer_t *image*, int *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_buffer_t *image*, int *coord*, uint4 *color*)	Write *color* value to location specified by *coord* in the 1D image or 1D image buffer object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord* is considered to be an unnormalized coordinate, and must be in the range [0, image width-1]. **write_imagef** and **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imageh**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above, or with a coordinate value that is not in the range [0, image width-1], is undefined. Requires support for OpenCL C 1.2 or newer. **write_imageh** requires support for the `cl_khr_fp16` extension macro.

Function	Description
void **write_imagef**(*aQual* image1d_array_t *image*, int2 *coord*, float4 *color*) void **write_imageh**(*aQual* image1d_array_t *image*, int2 *coord*, half4 *color*) void **write_imagei**(*aQual* image1d_array_t *image*, int2 *coord*, int4 *color*) void **write_imageui**(*aQual* image1d_array_t *image*, int2 *coord*, uint4 *color*)	Write *color* value to location specified by *coord.x* in the 1D image identified by *coord.y* in the 1D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x* and *coord.y* are considered to be unnormalized coordinates and must be in the range [0, image width-1] and [0, image number of layers-1], respectively. **write_imagef** and **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imageh**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range [0, image width-1] and [0, image number of layers-1], respectively, is undefined. Requires support for OpenCL C 1.2 or newer.

Function	Description
void **write_imagef**(*aQual* image2d_depth_t *image*, int2 *coord*, float *depth*)	Write *depth* value to location specified by *coord.xy* in the 2D depth image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the depth value. *coord.x* and *coord.y* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], and [0, image height-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to `CL_UNORM_INT16`, `CL_UNORM_INT24` or `CL_FLOAT`. Appropriate data format conversion will be done to convert depth valye from a floating-point value to actual data format associated with the image. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*) coordinate values that are not in the range [0, image width-1] and [0, image height-1], respectively, is undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

Function	Description
void **write_imagef**(*aQual* image2d_array_depth_t *image*, int4 *coord*, float *depth*)	Write *depth* value to location specified by *coord.xy* in the 2D image identified by *coord.z* in the 2D depth image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the depth value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], [0, image height-1], and [0, image number of layers-1], respectively. **write_imagef** can only be used with image objects created with *image_channel_data_type* set to `CL_UNORM_INT16`, `CL_UNORM_INT24` or `CL_FLOAT`. Appropriate data format conversion will be done to convert depth valye from a floating-point value to actual data format associated with the image. The behavior of **write_imagef**, **write_imagei** and **write_imageui** for image objects created with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], [0, image number of layers-1], respectively, is undefined. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

Function	Description
void **write_imagef**(*aQual* image3d_t *image*, int4 *coord*, float4 *color*) void **write_imageh**(*aQual* image3d_t *image*, int4 *coord*, half4 *color*) void **write_imagei**(*aQual* image3d_t *image*, int4 *coord*, int4 *color*) void **write_imageui**(*aQual* image3d_t *image*, int4 *coord*, uint4 *color*)	Write *color* value to the location specified by *coord.xyz* in the 3D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates, and must be in the range [0, image width-1], [0, image height-1], and [0, image depth-1], respectively. **write_imagef** and **write_imageh** can only be used with image objects created with *image_channel_data_type* set to one of the pre-defined packed formats or set to `CL_SNORM_INT8`, `CL_UNORM_INT8`, `CL_SNORM_INT16`, `CL_UNORM_INT16`, `CL_HALF_FLOAT` or `CL_FLOAT`. Appropriate data format conversion will be done to convert channel data from a floating-point value to actual data format in which the channels are stored. **write_imagei** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_SIGNED_INT8`, `CL_SIGNED_INT16` and `CL_SIGNED_INT32`. **write_imageui** can only be used with image objects created with *image_channel_data_type* set to one of the following values: `CL_UNSIGNED_INT8`, `CL_UNSIGNED_INT16` and `CL_UNSIGNED_INT32`. The behavior of **write_imagef**, **write_imageh**, **write_imagei** and **write_imageui** for image objects with *image_channel_data_type* values not specified in the description above or with (*x*, *y*, *z*) coordinate values that are not in the range [0, image width-1], [0, image height-1], and [0, image depth-1], respectively, is undefined. Requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_3d_image_writes` feature, or the `cl_khr_3d_image_writes` extension. **write_imageh** requires support for the `cl_khr_fp16` extension macro.

Function	Description

`void write_imagef(`
`  write_only image2d_t image,`
`  int2 coord,`
`  int lod,`
`  float4 color)`
 	
`void write_imagei(`
`  write_only image2d_t image,`
`  int2 coord,`
`  int lod,`
`  int4 color)`
 	
`void write_imageui(`
`  write_only image2d_t image,`
`  int2 coord,`
`  int lod,`
`  uint4 color)`
 	
`void write_imagef(`
`  write_only image2d_depth_t image,`
`  int2 coord,`
`  int lod,`
`  float depth)`
	Write *color* value to location specified by *coord.xy* in the mip level specified by *lod* in the 2D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value.
	
	lod must be in the range [0, *miplevels*-1]. *coord.x* and *coord.y* are considered to be unnormalized coordinates and must be in the range [0, *mipwidth*-1] and [0, *mipheight*-1] respectively. Behavior is undefined if *lod*, *coord.x*, or *coord.y* is not in range.
	
	Requires support for the `cl_khr_mipmap_image_writes` extension macro.

`void write_imagef(`
`  write_only image1d_t image,`
`  int coord,`
`  int lod,`
`  float4 color)`
 	
`void write_imagei(`
`  write_only image1d_t image,`
`  int coord,`
`  int lod,`
`  int4 color)`
 	
`void write_imageui(`
`  write_only image1d_t image,`
`  int coord,`
`  int lod,`
`  uint4 color)`
	Write *color* value to location specified by *coord* in the mip level specified by *lod* in the 1D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value.
	
	lod must be in the range [0, *miplevels*-1]. *coord* is considered to be an unnormalized coordinate and must be in the range [0, *mipwidth*-1]. Behavior is undefined if *lod* or *coord* is not in range.
	
	Requires support for the `cl_khr_mipmap_image_writes` extension macro.

Function	Description

`void write_imagef(`
`  write_only image1d_array_t image,`
`  int2 coord,`
`  int lod,`
`  float4 color)`
 	
`void write_imagei(`
`  write_only image1d_array_t image,`
`  int2 coord,`
`  int lod,`
`  int4 color)`
 	
`void write_imageui(`
`  write_only image1d_array_t image,`
`  int2 coord,`
`  int lod,`
`  uint4 color)`
	Write *color* value to location specified by *coord.x* in the 1D image identified by *coord.y* and mip level *lod* in the 1D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value.
	
	lod must be in the range [0, *miplevels*-1]. *coord.x* and *coord.y* are considered to be unnormalized coordinates and must be in the range [0, *mipwidth*-1] and [0, *miplayers* -1] respectively. Behavior is undefined if *lod*, *coord.x*, or *coord.y* is not in range.
	
	Requires support for the `cl_khr_mipmap_image_writes` extension macro.

Function	Description

`void write_imagef(`
`  write_only image2d_array_t image,`
`  int4 coord,`
`  int lod,`
`  float4 color)`
 	
`void write_imagei(`
`  write_only image2d_array_t image,`
`  int4 coord,`
`  int lod,`
`  int4 color)`
 	
`void write_imageui(`
`  write_only image2d_array_t image,`
`  int4 coord,`
`  int lod,`
`  uint4 color)`
 	
`void write_imagef(`
`  write_only image2d_array_depth_t image,`
`  int4 coord,`
`  int lod,`
`  float depth)`
	Write *color* value to location specified by *coord.xy* in the 2D image identified by *coord.z* and mip level *lod* in the 2D image array specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value.
	
	lod must be in the range [0, *miplevels*-1]. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates and must be in the range [0, *mipwidth*-1], [0, *mipheight*-1], and [0, *miplayers*-1] respectively. Behavior is undefined if *lod*, *coord.x*, *coord.y*, or *coord.z* is not in range.
	
	Requires support for the `cl_khr_mipmap_image_writes` extension macro.

`void write_imagef(`
`  write_only image3d_t image,`
`  int4 coord,`
`  int lod,`
`  float4 color)`
 	
`void write_imagei(`
`  write_only image3d_t image,`
`  int4 coord,`
`  int lod,`
`  int4 color)`
 	
`void write_imageui(`
`  write_only image3d_t image,`
`  int4 coord,`
`  int lod,`
`  uint4 color)`
	Write *color* value to location specified by *coord.xyz* and mip level *lod* in the 3D image object specified by *image*. Appropriate data format conversion to the specified image format is done before writing the color value.
	
	lod must be in the range [0, *miplevels*-1]. *coord.x*, *coord.y* and *coord.z* are considered to be unnormalized coordinates and must be in the range [0, *mipwidth*-1], [0, *mipheight*-1] and [0, *mipdepth*-1] respectively. Behavior is undefined if *lod*, *coord.x*, *coord.y*, or *coord.z* is not in range.
	
	Requires support for the `cl_khr_mipmap_image_writes` extension macro.

6.15.15.5. Built-in Image Query Functions

The following built-in function calls to query image information are supported.

aQual in the following table refers to one of the access qualifiers. For query functions this may be `read_only`, `write_only` or `read_write`.

Table 36. Built-in Image Query Functions

Function	Description
int **get_image_width**(*aQual* image2d_t *image*) int **get_image_width**(*aQual* image3d_t *image*) For OpenCL C 1.2 or newer: int **get_image_width**(*aQual* image1d_t *image*) int **get_image_width**(*aQual* image1d_buffer_t *image*) int **get_image_width**(*aQual* image1d_array_t *image*) int **get_image_width**(*aQual* image2d_array_t *image*) For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported: int **get_image_width**(*aQual* image2d_depth_t *image*) int **get_image_width**(*aQual* image2d_array_depth_t *image*) If the `cl_khr_gl_msaa_sharing` extension macro is supported: int **get_image_width**(*aQual* image2d_msaa_t image) int **get_image_width**(*aQual* image2d_array_msaa_t image) int **get_image_width**(*aQual* image2d_msaa_depth_t image) int **get_image_width**(*aQual* image2d_array_msaa_depth_t image)	Return the image width in pixels.

Function	Description
int **get_image_height**(*aQual* image2d_t *image*) int **get_image_height**(*aQual* image3d_t *image*) For OpenCL C 1.2 or newer: int **get_image_height**(*aQual* image2d_array_t *image*) For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported: int **get_image_height**(*aQual* image2d_depth_t *image*) int **get_image_height**(*aQual* image2d_array_depth_t *image*) If the `cl_khr_gl_msaa_sharing` extension macro is supported: int **get_image_height**(*aQual* image2d_msaa_t image) int **get_image_height**(*aQual* image2d_array_msaa_t image) int **get_image_height**(*aQual* image2d_msaa_depth_t image) int **get_image_height**(*aQual* image2d_array_msaa_depth_t image)	Return the image height in pixels.
int **get_image_depth**(image3d_t *image*)	Return the image depth in pixels.

Function	Description

int **get_image_channel_data_type**(*aQual* image2d_t *image*)
int **get_image_channel_data_type**(*aQual* image3d_t *image*)
 	
For OpenCL C 1.2 or newer:
 	
int **get_image_channel_data_type**(*aQual* image1d_t *image*)
int **get_image_channel_data_type**(*aQual* image1d_buffer_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_t *image*)
int **get_image_channel_data_type**(*aQual* image3d_t *image*)
int **get_image_channel_data_type**(*aQual* image1d_array_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_array_t *image*)
 	
For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported:
 	
int **get_image_channel_data_type**(*aQual* image2d_depth_t *image*)
int **get_image_channel_data_type**(*aQual* image2d_array_depth_t *image*)
 	
If the `cl_khr_gl_msaa_sharing` extension macro is supported:
 	
int **get_image_channel_data_type**(*aQual* image2d_msaa_t image)
int **get_image_channel_data_type**(*aQual* image2d_array_msaa_t image)
int **get_image_channel_data_type**(*aQual* image2d_msaa_depth_t image)
int **get_image_channel_data_type**(*aQual* image2d_array_msaa_depth_t image)
	Return the channel data type. Valid values are:
	
	`CLK_SNORM_INT8`
	`CLK_SNORM_INT16`
	`CLK_UNORM_INT8`
	`CLK_UNORM_INT16`
	`CLK_UNORM_SHORT_565`
	`CLK_UNORM_SHORT_555`
	`CLK_UNORM_INT_101010`
	`CLK_SIGNED_INT8`
	`CLK_SIGNED_INT16`
	`CLK_SIGNED_INT32`
	`CLK_UNSIGNED_INT8`
	`CLK_UNSIGNED_INT16`
	`CLK_UNSIGNED_INT32`
	`CLK_HALF_FLOAT`
	`CLK_FLOAT`
	
	Additionally, for OpenCL C 3.0 or newer:
	
	`CLK_UNORM_INT_101010_2` [84]
	
	Additionally, if the `__opencl_c_ext_image_unorm_int_2_101010` feature is supported:
	
	`CLK_UNORM_INT_2_101010_EXT`
	
	Additionally, if the `__opencl_c_ext_image_unsigned_10x6_12x4_14x2` feature macro is supported:
	
	`CLK_UNSIGNED_INT10X6_EXT`
	`CLK_UNSIGNED_INT12X4_EXT`
	`CLK_UNSIGNED_INT14X2_EXT`
	`CLK_UNORM_INT10X6_EXT`
	`CLK_UNORM_INT12X4_EXT`
	`CLK_UNORM_INT14X2_EXT`

Function	Description

int **get_image_channel_order**(*aQual* image2d_t *image*)
int **get_image_channel_order**(*aQual* image3d_t *image*)
 	
For OpenCL C 1.2 or newer:
 	
int **get_image_channel_order**(*aQual* image1d_t *image*)
int **get_image_channel_order**(*aQual* image1d_buffer_t *image*)
int **get_image_channel_order**(*aQual* image1d_array_t *image*)
int **get_image_channel_order**(*aQual* image2d_array_t *image*)
 	
For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported:
 	
int **get_image_channel_order**(*aQual* image2d_depth_t *image*)
int **get_image_channel_order**(*aQual* image2d_array_depth_t *image*)
 	
If the `cl_khr_gl_msaa_sharing` extension macro is supported:
 	
int **get_image_channel_order**(*aQual* image2d_msaa_t image)
int **get_image_channel_order**(*aQual* image2d_array_msaa_t image)
int **get_image_channel_order**(*aQual* image2d_msaa_depth_t image)
int **get_image_channel_order**(*aQual* image2d_array_msaa_depth_t image)
	Return the image channel order. Valid values are:
	
	`CLK_A`
	`CLK_R`
	`CLK_RG`
	`CLK_RA`
	`CLK_RGB`
	`CLK_RGBA`
	`CLK_ARGB`
	`CLK_BGRA`
	`CLK_INTENSITY`
	`CLK_LUMINANCE`
	
	Additionally, for OpenCL C 1.1 or newer:
	
	`CLK_Rx`
	`CLK_RGx`
	`CLK_RGBx`
	
	Additionally, for OpenCL C 2.0 or newer:
	
	`CLK_ABGR`
	`CLK_DEPTH`
	`CLK_sRGB`
	`CLK_sRGBx`
	`CLK_sRGBA`
	`CLK_sBGRA`

Function	Description
int2 **get_image_dim**(*aQual* image2d_t *image*) For OpenCL C 1.2 or newer: int2 **get_image_dim**(*aQual* image2d_array_t *image*) For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported: int2 **get_image_dim**(*aQual* image2d_depth_t *image*) int2 **get_image_dim**(*aQual* image2d_array_depth_t *image*) If the `cl_khr_gl_msaa_sharing` extension macro is supported: int2 **get_image_dim**(*aQual* image2d_msaa_t image) int2 **get_image_dim**(*aQual* image2d_array_msaa_t image) int2 **get_image_dim**(*aQual* image2d_msaa_depth_t image) int2 **get_image_dim**(*aQual* image2d_array_msaa_depth_t image)	Return the 2D image width and height as an `int2` type. The width is returned in the *x* component, and the height in the *y* component.
int4 **get_image_dim**(*aQual* image3d_t *image*)	Return the 3D image width, height, and depth as an `int4` type. The width is returned in the *x* component, height in the *y* component, depth in the *z* component and the *w* component is 0.

Function	Description
For OpenCL C 1.2 or newer: size_t **get_image_array_size**(*aQual* image2d_array_t *image*) For OpenCL C 2.0 or newer, or if the `cl_khr_depth_images` extension macro is supported: size_t **get_image_array_size**(*aQual* image2d_array_depth_t *image*) If the `cl_khr_gl_msaa_sharing` extension macro is supported: size_t **get_image_array_size**(*aQual* image2d_array_msaa_depth_t *image*)	Return the number of images in the 2D image array.
For OpenCL C 1.2 or newer: size_t **get_image_array_size**(*aQual* image1d_array_t *image*)	Return the number of images in the 1D image array.
If the `cl_khr_gl_msaa_sharing` extension macro is supported: int **get_image_num_samples**(*aQual* image2d_msaa_t *image*) int **get_image_num_samples**(*aQual* image2d_array_msaa_t *image*) int **get_image_num_samples**(*aQual* image2d_msaa_depth_t *image*) int **get_image_num_samples**(*aQual* image2d_array_msaa_depth_t *image*)	Return the number of samples in the 2D MSAA image

Function	Description
If the `cl_khr_mipmap_image` extension macro is supported: int **get_image_num_mip_levels**(*aQual* image1d_t *image*) int **get_image_num_mip_levels**(*aQual* image2d_t *image*) int **get_image_num_mip_levels**(*aQual* image3d_t *image*) int **get_image_num_mip_levels**(*aQual* image1d_array_t *image*) int **get_image_num_mip_levels**(*aQual* image2d_array_t *image*) int **get_image_num_mip_levels**(*aQual* image2d_depth_t *image*) int **get_image_num_mip_levels**(*aQual* image2d_array_depth_t *image*)	Return the number of mip levels in *image*.

The values returned by **get_image_channel_data_type** and **get_image_channel_order** as specified in Built-in Image Query Functions with the `CLK_` prefixes correspond to the `CL_` prefixes used to describe the image channel order and data type in the OpenCL Specification. For example, both `CL_UNORM_INT8` and `CLK_UNORM_INT8` refer to an image channel data type that is an unnormalized unsigned 8-bit integer.

6.15.15.6. Reading and Writing to the Same Image in a Kernel

The **atomic_work_item_fence**(`CLK_IMAGE_MEM_FENCE`) built-in function can be used to make sure that sampler-less writes are visible to later reads by the same work-item. Only a scope of `memory_scope_work_item` and an order of `memory_order_acq_rel` is valid for `atomic_work_item_fence` when passed the `CLK_IMAGE_MEM_FENCE` flag. If multiple work-items are writing to and reading from multiple locations in an image, the **work_group_barrier**(`CLK_IMAGE_MEM_FENCE`) should be used.

Consider the following example:

```
kernel void
foo(read_write image2d_t img, ... )
{
    int2 coord;
    coord.x = (int)get_global_id(0);
    coord.y = (int)get_global_id(1);

    float4 clr = read_imagef(img, coord);
    ...
    write_imagef(img, coord, clr);

    // required to ensure that following read from image at
    // location coord returns the latest color value.
```

```
    atomic_work_item_fence(
        CLK_IMAGE_MEM_FENCE,
        memory_order_acq_rel,
        memory_scope_work_item);

    float4 clr_new = read_imagef(img, coord);
    ...

}
```

6.15.15.7. Mapping Image Channels to Color Values Returned by read_image and Color Values Passed to write_image to Image Channels

The following table describes the mapping of the number of channels of an image element to the appropriate components in the `float4`, `int4` or `uint4` vector data type for the color values returned by **read_image{f|i|ui}** or supplied to **write_image{f|i|ui}**. The unmapped components will be set to 0.0 for red, green and blue channels and will be set to 1.0 for the alpha channel.

Channel Order	**`float4`, `int4` or `uint4` components of channel data**
`CL_R`, `CL_Rx`	(r, 0.0, 0.0, 1.0)
`CL_A`	(0.0, 0.0, 0.0, a)
`CL_RG`, `CL_RGx`	(r, g, 0.0, 1.0)
`CL_RA`	(r, 0.0, 0.0, a)
`CL_RGB`, `CL_RGBx`, `CL_sRGB`, `CL_sRGBx`	(r, g, b, 1.0)
`CL_RGBA`, `CL_BGRA`, `CL_ARGB`, `CL_ABGR`, `CL_sRGBA`, `CL_sBGRA`	(r, g, b, a)
`CL_INTENSITY`	(I, I, I, I)
`CL_LUMINANCE`	(L, L, L, 1.0)

For `CL_DEPTH` images, a scalar value is returned by **read_imagef** or supplied to **write_imagef**. Requires support for OpenCL C 2.0 or newer, or for the `cl_khr_depth_images` extension macro.

A kernel that uses a sampler with the `CL_ADDRESS_CLAMP` addressing mode with multiple images may result in additional samplers being used internally by an implementation. If the same sampler is used with multiple images called via **read_image{f|i|ui}**, then it is possible that an implementation may need to allocate an additional sampler to handle the different border color values that may be needed depending on the image formats being used. These implementation allocated samplers will count against the maximum sampler values supported by the device and given by `CL_DEVICE_MAX_SAMPLERS`. Enqueuing a kernel that requires more samplers than the implementation can support will result in a `CL_OUT_OF_RESOURCES` error being returned.

6.15.16. Work-group Collective Functions

The functionality described in this section requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_work_group_collective_functions` feature.

This section describes built-in functions that perform collective options across a work-group. These built-in functions must be encountered by all work-items in a work-group executing the kernel. We use the generic type name `gentype` to indicate the built-in data types `half` [85], `int`, `uint`, `long` [86], `ulong`, `float` or `double` [87] as the type for the arguments.

Table 37. Built-in Work-group Collective Functions

Function	Description
int **work_group_all**(int *predicate*)	Evaluates *predicate* for all work-items in the work-group and returns a non-zero value if *predicate* evaluates to non-zero for all work-items in the work-group.
int **work_group_any**(int *predicate*)	Evaluates *predicate* for all work-items in the work-group and returns a non-zero value if *predicate* evaluates to non-zero for any work-items in the work-group.
gentype **work_group_broadcast**(gentype *a*, size_t *local_id*) gentype **work_group_broadcast**(gentype *a*, size_t *local_id_x*, size_t *local_id_y*) gentype **work_group_broadcast**(gentype *a*, size_t *local_id_x*, size_t *local_id_y*, size_t *local_id_z*)	Broadcast the value of *a* for work-item identified by *local_id* to all work-items in the work-group. Behavior is undefined when the value of *local_id* is not equivalent for all work-items in the work-group. Behavior is undefined when *local_id* is greater or equal to the work-group size in the corresponding dimension.
gentype **work_group_reduce_<op>**(gentype *x*)	Return result of reduction operation specified by **<op>** for all values of *x* specified by work-items in a work-group.
gentype **work_group_scan_exclusive_<op>**(gentype *x*)	Do an exclusive scan operation specified by **<op>** of all values specified by work-items in the work-group. The scan results are returned for each work-item. The scan order is defined by increasing 1D linear global ID within the work-group.

Function	Description
gentype **work_group_scan_inclusive_<op>**(gentype *x*)	Do an inclusive scan operation specified by **<op>** of all values specified by work-items in the work-group. The scan results are returned for each work-item. The scan order is defined by increasing 1D linear global ID within the work-group.

The **<op>** in **work_group_reduce_<op>**, **work_group_scan_exclusive_<op>** and **work_group_scan_inclusive_<op>** defines the operator and can be **add**, **min** or **max**.

The inclusive scan operation takes a binary operator **op** with *n* (where *n* is the size of the work-group) elements [a_0, a_1, ... a_{n-1}] and returns [a_0, (a_0 **op** a_1), ... (a_0 **op** a_1 **op** ... **op** a_{n-1})].

Consider the following example:

```
void foo(int *p)
{
    ...
    int prefix_sum_val = work_group_scan_inclusive_add(
                             p[get_local_id(0)]);
}
```

For the example above, let's assume that the work-group size is 8 and *p* points to the following elements [3 1 7 0 4 1 6 3]. Work-item 0 calls **work_group_scan_inclusive_add** with 3 and returns 3. Work-item 1 calls **work_group_scan_inclusive_add** with 1 and returns 4. The full set of values returned by **work_group_scan_inclusive_add** for work-items 0 ... 7 are [3 4 11 11 15 16 22 25].

The exclusive scan operation takes a binary associative operator **op** with an identity I and n (where n is the size of the work-group) elements [a_0, a_1, ... a_{n-1}] and returns [I, a_0, (a_0 **op** a_1), ... (a_0 **op** a_1 **op** ... **op** a_{n-2})]. If **op** = add, the identity I is 0. If **op** = min, the identity I is `INT_MAX`, `UINT_MAX`, `LONG_MAX`, `ULONG_MAX`, for `int`, `uint`, `long`, `ulong` types and is `+INF` for floating-point types. Similarly if **op** = max, the identity I is `INT_MIN`, 0, `LONG_MIN`, 0 and `-INF`. For the example above, the exclusive scan add operation on the ordered set [3 1 7 0 4 1 6 3] would return [0 3 4 11 11 15 16 22].

The order of floating-point operations is not guaranteed for the **work_group_reduce_<op>**, **work_group_scan_inclusive_<op>** and **work_group_scan_exclusive_<op>** built-in functions that operate on `half`, `float` and `double` data types. The order of these floating-point operations is also non-deterministic for a given work-group.

6.15.17. Work-group Collective Uniform Arithmetic Functions

The functionality described in this section requires support for OpenCL C 2.0 and the `cl_khr_work_group_uniform_arithmetic` extension macro.

The Built-in Work-group Logical Arithmetic Functions table describes the OpenCL C programming language built-in functions that perform logical arithmetic operations across work items in a work-group. These functions must be encountered by all work items in a work-group executing the kernel, otherwise the behavior is undefined. For these functions, a non-zero *predicate* argument or return value is logically `true` and a zero *predicate* argument or return value is logically `false`.

Table 38. Built-in Work-group Logical Arithmetic Functions

Function	Description

`int work_group_reduce_logical_and(`
`  int predicate);`
`int work_group_reduce_logical_or(`
`  int predicate);`
`int work_group_reduce_logical_xor(`
`  int predicate);`
	Returns the logical **and**, **or**, or **xor** of *predicate* for all work items in the work-group.

`int work_group_scan_inclusive_logical_and(`
`  int predicate);`
`int work_group_scan_inclusive_logical_or(`
`  int predicate);`
`int work_group_scan_inclusive_logical_xor(`
`  int predicate);`
	Returns the result of an inclusive scan operation, which is the logical **and**, **or**, or **xor** of *predicate* for all work items in the work-group with a work-group linear local ID less than or equal to this work item's work-group linear local ID.

`int work_group_scan_exclusive_logical_and(`
`  int predicate);`
`int work_group_scan_exclusive_logical_or(`
`  int predicate);`
`int work_group_scan_exclusive_logical_xor(`
`  int predicate);`
	Returns the result of an exclusive scan operation, which is the logical **and**, **or**, or **xor** of *predicate* for all work items in the work-group with a work-group linear local ID less than this work item's work-group linear local ID.
	
	If there is no work item in the work-group with a work-group linear local ID less than this work item's work-group linear local ID then an identity value `I` is returned. For **and**, the identity value is `true` (non-zero). For **or** and **xor**, the identity value is `false` (zero).

The Built-in Work-group Bitwise Integer Functions table describes the OpenCL C programming language built-in functions that perform bitwise integer operations across work items in a work-group. These functions must be encountered by all work items in a work-group executing the kernel, otherwise the behavior is undefined. For the functions below, the generic type name `gentype` may be one of the supported built-in scalar data types `int`, `uint`, `long`, and `ulong`.

Table 39. Built-in Work-group Bitwise Integer Functions

Function	Description

`gentype work_group_reduce_and(`
`  gentype value);`
`gentype work_group_reduce_or(`
`  gentype value);`
`gentype work_group_reduce_xor(`
`  gentype value);`
	Returns the bitwise **and**, **or**, or **xor** of *value* for all work items in the work-group.

`gentype work_group_scan_inclusive_and(`
`  gentype value);`
`gentype work_group_scan_inclusive_or(`
`  gentype value);`
`gentype work_group_scan_inclusive_xor(`
`  gentype value);`
	Returns the result of an inclusive scan operation, which is the bitwise **and**, **or**, or **xor** of *value* for all work items in the work-group with a work-group linear local ID less than or equal to this work item's work-group linear local ID.

`gentype work_group_scan_exclusive_and(`
`  gentype value);`
`gentype work_group_scan_exclusive_or(`
`  gentype value);`
`gentype work_group_scan_exclusive_xor(`
`  gentype value);`
	Returns the result of an exclusive scan operation, which is the bitwise **and**, **or**, or **xor** of *value* for all work items in the work-group with a work-group linear local ID less than this work item's work-group linear local ID.
	
	If there is no work item in the work-group with a work-group linear local ID less than this work item's work-group linear local ID then an identity value `I` is returned. For **and**, the identity value is `~0` (all bits set). For **or** and **xor**, the identity value is `0`.

The Built-in Work-group Multiplicative Functions table describes the OpenCL C programming language built-in functions that perform multiplicative operations across work items in a work-group. These functions must be encountered by all work items in a work-group executing the kernel, otherwise the behavior is undefined. For the functions below, the generic type name `gentype` may be one of the supported built-in scalar data types `int`, `uint`, `long`, `ulong`, `float`, `double` (if double precision is supported), or `half` (if half precision is supported).

Table 40. Built-in Work-group Multiplicative Functions

Function	Description
`gentype work_group_reduce_mul(` `  gentype value);`	Returns the multiplication of *value* for all work items in the work-group.
`gentype work_group_scan_inclusive_mul(` `  gentype value);`	Returns the result of an inclusive scan operation which is the multiplication of *value* for all work items in the work-group with a work-group linear local ID less than or equal to this work item's work-group linear local ID.

Function	Description
`gentype work_group_scan_exclusive_mul(` `  gentype value);`	Returns the result of an exclusive scan operation which is the multiplication of *value* for all work items in the work-group with a work-group linear local ID less than this work item's work-group linear local ID. If there is no work item in the work-group with a work-group linear local ID less than this work item's work-group linear local ID then the identity value 1 is returned.

6.15.18. Pipe Functions

The functionality described in this section requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_pipes` feature.

A pipe is identified by specifying the `pipe` keyword with a type. The data type specifies the type of each element in the pipe. The `pipe` keyword is a type specifier. When it is applied to another type **T**, the result is a pipe type whose elements (or packets) are of type **T**. The packet type **T** may be any supported OpenCL C scalar and vector integer or floating-point data types, or a user-defined type built from these scalar and vector data types.

Examples:

```
pipe int4 pipeA; // a pipe with int4 packets

pipe user_type_t pipeB; // a pipe with user_type_t packets
```

The `read_only` (or `__read_only`) and `write_only` (or `__write_only`) qualifiers must be used with the `pipe` specifier when a pipe is a parameter of a kernel or of a user-defined function to identify if a pipe can be read from or written to by a kernel and its callees and enqueued child kernels. If no qualifier is specified, `read_only` is assumed.

A kernel cannot read from and write to the same pipe object. Using the `read_write` (or `__read_write`) qualifier with the `pipe` specifier is a compilation error.

In the following example

```
kernel void
foo (read_only pipe fooA_t pipeA,
     write_only pipe fooB_t pipeB)
{
    ...
}
```

`pipeA` is a read-only pipe object, and `pipeB` is a write-only pipe object.

The macro `CLK_NULL_RESERVE_ID` refers to an invalid reservation ID.

6.15.18.1. Restrictions

- Pipes can only be passed as arguments to a function (including kernel functions). The C operators cannot be used with variables declared with the pipe specifier.
- The `pipe` specifier cannot be used with variables declared inside a kernel, a structure or union field, a pointer type, an array, global variables declared in program scope or the return type of a function.

6.15.18.2. Built-in Pipe Read and Write Functions

The OpenCL C programming language implements the following built-in functions that read from or write to a pipe. We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types [88] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

Table 41. Built-in Pipe Functions

Function	Description
int **read_pipe**(read_only pipe gentype *p*, gentype **ptr*)	Read packet from pipe *p* into *ptr*. Returns 0 if **read_pipe** is successful and a negative value if the pipe is empty.
int **write_pipe**(write_only pipe gentype *p*, const gentype **ptr*)	Write packet specified by *ptr* to pipe *p*. Returns 0 if **write_pipe** is successful and a negative value if the pipe is full.
int **read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*, uint *index*, gentype **ptr*)	Read packet from the reserved area of the pipe referred to by *reserve_id* and *index* into *ptr*. The reserved pipe entries are referred to by indices that go from 0 ... *num_packets* - 1. Returns 0 if **read_pipe** is successful and a negative value otherwise.
int **write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*, uint *index*, const gentype **ptr*)	Write packet specified by *ptr* to the reserved area of the pipe referred to by *reserve_id* and *index*. The reserved pipe entries are referred to by indices that go from 0 ... *num_packets* - 1. Returns 0 if **write_pipe** is successful and a negative value otherwise.

Function	Description
reserve_id_t **reserve_read_pipe**(read_only pipe gentype *p*, uint *num_packets*) reserve_id_t **reserve_write_pipe**(write_only pipe gentype *p*, uint *num_packets*)	Reserve *num_packets* entries for reading from or writing to pipe *p*. Returns a valid reservation ID if the reservation is successful.
void **commit_read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*) void **commit_write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*)	Indicates that all reads and writes to *num_packets* associated with reservation *reserve_id* are completed.
bool **is_valid_reserve_id**(reserve_id_t *reserve_id*)	Return *true* if *reserve_id* is a valid reservation ID and *false* otherwise.

6.15.18.3. Built-in Work-group Pipe Read and Write Functions

The OpenCL C programming language implements the following built-in pipe functions that operate at a work-group level. These built-in functions must be encountered by all work-items in a work-group executing the kernel with the same argument values; otherwise the behavior is undefined. We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types [89] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

Table 42. Built-in Pipe Work-group Functions

Function	Description
reserve_id_t **work_group_reserve_read_pipe**(read_only pipe gentype *p*, uint *num_packets*) reserve_id_t **work_group_reserve_write_pipe**(write_only pipe gentype *p*, uint *num_packets*)	Reserve *num_packets* entries for reading from or writing to pipe *p*. Returns a valid reservation ID if the reservation is successful. The reserved pipe entries are referred to by indices that go from 0 ... *num_packets* - 1.
void **work_group_commit_read_pipe**(read_only pipe gentype *p*, reserve_id_t *reserve_id*) void **work_group_commit_write_pipe**(write_only pipe gentype *p*, reserve_id_t *reserve_id*)	Indicates that all reads and writes to *num_packets* associated with reservation *reserve_id* are completed.

The **read_pipe** and **write_pipe** functions that take a reservation ID as an argument can be used to read from or write to a packet index. These built-ins can be used to read from or write to a packet index one or multiple times. If a packet index that is reserved for writing is not written to using the **write_pipe** function, the contents of that packet in the pipe are undefined. **commit_read_pipe** and **work_group_commit_read_pipe** remove the entries reserved for reading from the pipe. **commit_write_pipe** and **work_group_commit_write_pipe** ensures that the entries reserved for writing are all added in-order as one contiguous set of packets to the pipe.

There can only be the value of the CL_DEVICE_PIPE_MAX_ACTIVE_RESERVATIONS device query reservations active (i.e. reservation IDs that have been reserved but not committed) per work-item or work-group for a pipe in a kernel executing on a device.

Work-item based reservations made by a work-item are ordered in the pipe as they are ordered in the program. Reservations made by different work-items that belong to the same work-group can be ordered using the work-group barrier function. The order of work-item based reservations that belong to different work-groups is implementation-defined.

Work-group based reservations made by a work-group are ordered in the pipe as they are ordered in the program. The order of work-group based reservations by different work-groups is implementation-defined.

6.15.18.4. Built-in Pipe Query Functions

The OpenCL C programming language implements the following built-in query functions for a pipe. We use the generic type name gentype to indicate the built-in OpenCL C scalar or vector integer or floating-point data types [90] or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in the following table.

aQual in the following table refers to one of the access qualifiers. For pipe query functions this may be read_only or write_only.

Table 43. Built-in Pipe Query Functions

Function	Description
uint **get_pipe_num_packets**(*aQual* pipe gentype *p*)	Returns the number of available entries in the pipe. The number of available entries in a pipe is a dynamic value. The value returned should be considered immediately stale.
uint **get_pipe_max_packets**(*aQual* pipe gentype *p*)	Returns the maximum number of packets specified when *pipe* was created.

6.15.18.5. Restrictions

The following behavior is undefined

- A kernel fails to call **reserve_pipe** before calling **read_pipe** or **write_pipe** that take a reservation ID.
- A kernel calls **read_pipe**, **write_pipe**, **commit_read_pipe** or **commit_write_pipe** with an invalid reservation ID.
- A kernel calls **read_pipe** or **write_pipe** with an valid reservation ID but with an *index* that is not a value in the range [0, *num_packets*-1] specified to the corresponding call to *reserve_pipe*.
- A kernel calls **read_pipe** or **write_pipe** with a reservation ID that has already been committed (i.e. a **commit_read_pipe** or **commit_write_pipe** with this reservation ID has already been called).
- A kernel fails to call **commit_read_pipe** for any reservation ID obtained by a prior call to **reserve_read_pipe**.

- A kernel fails to call **commit_write_pipe** for any reservation ID obtained by a prior call to **reserve_write_pipe**.
- The contents of the reserved data packets in the pipe are undefined if the kernel does not call **write_pipe** for all entries that were reserved by the corresponding call to **reserve_pipe**.
- Calls to **read_pipe** that takes a reservation ID and **commit_read_pipe** or **write_pipe** that takes a reservation ID and **commit_write_pipe** for a given reservation ID must be called by the same kernel that made the reservation using **reserve_read_pipe** or **reserve_write_pipe**. The reservation ID cannot be passed to another kernel including child kernels.

6.15.19. Enqueuing Kernels

The functionality described in this section requires support for OpenCL C 2.0, or OpenCL C 3.0 or newer and the `__opencl_c_device_enqueue` feature.

This section describes built-in functions that allow a kernel to enqueue additional work to the same device, without host interaction. A kernel may enqueue code represented by Block syntax, and control execution order with event dependencies including user events and markers. There are several advantages to using the Block syntax: it is more compact; it does not require a `cl_kernel` object; and enqueuing can be done as a single semantic step.

The macro `CLK_NULL_EVENT` refers to an invalid device event. The macro `CLK_NULL_QUEUE` refers to an invalid device queue.

6.15.19.1. Built-in Functions - Enqueuing a Kernel

The following table describes the list of built-in functions that can be used to enqueue a kernel.

When the `cl_khr_device_enqueue_local_arg_types` extension macro is supported, the Built-in Kernel Enqueue Functions and Built-in Kernel Query Functions described in this section can use any of the built-in OpenCL C scalar or vector integer or floating-point data types, or any user defined type built from these scalar and vector data types, as the pointee type of their arguments. This is indicated by the generic type name `gentype` in those function signatures.

When the `cl_khr_device_enqueue_local_arg_types` extension macro is not supported, the pointee type of these functions must be `void`.

Table 44. Built-in Kernel Enqueue Functions

Built-in Function	Description
int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, void (^*block*)(void)) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*, void (^*block*)(void)) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, void (^*block*)(local void *, ...), uint size0, ...) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*, void (^*block*)(local void *, ...), uint size0, ...) If the `cl_khr_device_enqueue_local_arg_types` extension macro is supported: int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, void (^*block*)(local gentype *, ...), uint size0, ...) int **enqueue_kernel**(queue_t *queue*, kernel_enqueue_flags_t *flags*, const ndrange_t *ndrange*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*, void (^*block*)(local gentype *, ...), uint size0, ...)	Enqueue the block for execution to *queue*. If an event is returned, **enqueue_kernel** performs an implicit retain on the returned event.

The **enqueue_kernel** built-in function allows a work-item to enqueue a block. Work-items can enqueue multiple blocks to device queues.

The **enqueue_kernel** built-in function returns `CLK_SUCCESS` if the block is enqueued successfully and returns `CLK_ENQUEUE_FAILURE` otherwise. If the -g compile option is specified in compiler options passed to **clCompileProgram** or **clBuildProgram** when compiling or building the parent program, the following errors may be returned instead of `CLK_ENQUEUE_FAILURE` to indicate why **enqueue_kernel** failed to enqueue the block:

- `CLK_INVALID_QUEUE` if *queue* is not a valid device queue.
- `CLK_INVALID_NDRANGE` if *ndrange* is not a valid ND-range descriptor or if the program was compiled with `-cl-uniform-work-group-size` and the *local_work_size* is specified in *ndrange* but the *global_work_size* specified in *ndrange* is not a multiple of the *local_work_size*.

- `CLK_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.
- `CLK_DEVICE_QUEUE_FULL` if *queue* is full.
- `CLK_INVALID_ARG_SIZE` if size of local memory arguments is 0.
- `CLK_EVENT_ALLOCATION_FAILURE` if *event_ret* is not `NULL` and an event could not be allocated.
- `CLK_OUT_OF_RESOURCES` if there is a failure to queue the block in *queue* because of insufficient resources needed to execute the kernel.

Below are some examples of how to enqueue a block.

```
kernel void
my_func_A(global int *a, global int *b, global int *c)
{
    ...
}

kernel void
my_func_B(global int *a, global int *b, global int *c)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // example - enqueue a kernel as a block
    enqueue_kernel(get_default_queue(), ndrange,
                   ^{my_func_A(a, b, c);});

    ...
}

kernel void
my_func_C(global int *a, global int *b, global int *c)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // note that a, b and c are variables in scope of
    // the block
    void (^my_block_A)(void) = ^{my_func_A(a, b, c);};

    // enqueue the block variable
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_block_A);
    ...
```

```
}
```

The example below shows how to declare a block literal and enqueue it.

```
kernel void
my_func(global int *a, global int *b)
{
    ndrange_t ndrange;
    // build ndrange information
    ...

    // note that a, b and c are variables in scope of
    // the block
    void (^my_block_A)(void) =
    ^{
        size_t id = get_global_id(0);
        b[id] += a[id];
    };

    // enqueue the block variable
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_block_A);

    // or we could have done the following
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^{
                       size_t id = get_global_id(0);
                       b[id] += a[id];
                   };
}
```

Blocks passed to enqueue_kernel cannot use global variables or stack variables local to the enclosing lexical scope that are a pointer type in the `local` or `private` address space.

Example:

```
kernel void
foo(global int *a, local int *lptr, ...)
{
    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^{
```

```
                    size_t id = get_global_id(0);
                    local int *p = lptr; // undefined behavior
                } );
}
```

6.15.19.2. Arguments That are a Pointer Type to Local Address Space

A block passed to enqueue_kernel can have arguments declared to be a pointer to `local` memory. The enqueue_kernel built-in function variants allow blocks to be enqueued with a variable number of arguments. Each argument must be declared to be a `void` pointer to local memory. These enqueue_kernel built-in function variants also have a corresponding number of arguments each of type `uint` that follow the block argument. These arguments specify the size of each local memory pointer argument of the enqueued block.

Some examples follow:

```
kernel void
my_func_A_local_arg1(global int *a, local int *lptr, ...)
{
    ...
}

kernel void
my_func_A_local_arg2(global int *a,
                     local int *lptr1, local float4 *lptr2, ...)
{
    ...
}

kernel void
my_func_B(global int *a, ...)
{
    ...

    ndrange_t ndrange = ndrange_1D(...);

    uint local_mem_size = compute_local_mem_size();

    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   ^(local void *p){
                       my_func_A_local_arg1(a, (local int *)p, ...);},
                   local_mem_size);
}

kernel void
my_func_C(global int *a, ...)
{
    ...
```

```
    ndrange_t ndrange = ndrange_1D(...);

    void (^my_blk_A)(local void *, local void *) =
        ^(local void *lptr1, local void *lptr2){
        my_func_A_local_arg2(
            a,
            (local int *)lptr1,
            (local float4 *)lptr2, ...);};

    // calculate local memory size for lptr
    // argument in local address space for my_blk_A
    uint local_mem_size = compute_local_mem_size();

    enqueue_kernel(get_default_queue(),
                   CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                   ndrange,
                   my_blk_A,
                   local_mem_size, local_mem_size*4);
}
```

6.15.19.3. A Complete Example

The example below shows how to implement an iterative algorithm where the host enqueues the first instance of the nd-range kernel (dp_func_A). The kernel dp_func_A will launch a kernel (evaluate_dp_work_A) that will determine if new nd-range work needs to be performed. If new nd-range work does need to be performed, then evaluate_dp_work_A will enqueue a new instance of dp_func_A. This process is repeated until all the work is completed.

```
kernel void
dp_func_A(queue_t q, ...)
{
    ...

    // queue a single instance of evaluate_dp_work_A to
    // device queue q. queued kernel begins execution after
    // kernel dp_func_A finishes

    if (get_global_id(0) == 0)
    {
        enqueue_kernel(q,
                       CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                       ndrange_1D(1),
                       ^{evaluate_dp_work_A(q, ...);});
    }
}

kernel void
evaluate_dp_work_A(queue_t q,...)
{
```

```
    // check if more work needs to be performed
    bool more_work = check_new_work(...);
    if (more_work)
    {
        size_t global_work_size = compute_global_size(...);

        void (^dp_func_A_blk)(void) =
            ^{dp_func_A(q, ...});

        // get local WG-size for kernel dp_func_A
        size_t local_work_size =
            get_kernel_work_group_size(dp_func_A_blk);

        // build nd-range descriptor
        ndrange_t ndrange = ndrange_1D(global_work_size,
                                       local_work_size);

        // enqueue dp_func_A
        enqueue_kernel(q,
                       CLK_ENQUEUE_FLAGS_WAIT_KERNEL,
                       ndrange,
                       dp_func_A_blk);
    }
    ...
}
```

6.15.19.4. Determining when a Child Kernel Begins Execution

The `kernel_enqueue_flags_t` [91] argument to the `enqueue_kernel` built-in functions can be used to specify when the child kernel begins execution. Supported values are described in the following table:

Table 45. Kernel Enqueue Flags

`kernel_enqueue_flags_t` **enum**	**Description**
`CLK_ENQUEUE_FLAGS_NO_WAIT`	Indicates that the enqueued kernels do not need to wait for the parent kernel to finish execution before they begin execution.
`CLK_ENQUEUE_FLAGS_WAIT_KERNEL`	Indicates that all work-items of the parent kernel must finish executing and all immediate [92] side effects committed before the enqueued child kernel may begin execution.
`CLK_ENQUEUE_FLAGS_WAIT_WORK_GROUP`	Indicates that the enqueued kernels wait only for the work-group that enqueued the kernels to finish before they begin execution. [93]

The `kernel_enqueue_flags_t` flags are useful when a kernel enqueued from the host and executing on a device enqueues kernels on the device. The kernel enqueued

from the host may not have an event associated with it. The `kernel_enqueue_flags_t` flags allow the developer to indicate when the child kernels can begin execution.

6.15.19.5. Determining When a Parent Kernel has Finished Execution

A parent kernel's execution status is considered to be complete when it and all its child kernels have finished execution. The execution status of a parent kernel will be `CL_COMPLETE` if this kernel and all its child kernels finish execution successfully. The execution status of the kernel will be an error code (given by a negative integer value) if it or any of its child kernels encounter an error, or are abnormally terminated.

For example, assume that the host enqueues a kernel `k` for execution on a device. Kernel `k` when executing on the device enqueues kernels `A` and `B` to a device queue(s). The enqueue_kernel call to enqueue kernel `B` specifies the event associated with kernel `A` in the `event_wait_list` argument, i.e. wait for kernel `A` to finish execution before kernel `B` can begin execution. Let's assume kernel `A` enqueues kernels `X`, `Y` and `Z`. Kernel `A` is considered to have finished execution, i.e. its execution status is `CL_COMPLETE`, only after `A` and the kernels `A` enqueued (and any kernels these enqueued kernels enqueue and so on) have finished execution.

6.15.19.6. Built-in Functions - Kernel Query Functions

Table 46. Built-in Kernel Query Functions

Built-in Function	Description
uint **get_kernel_work_group_size**(void (^block)(void)) uint **get_kernel_work_group_size**(void (^block)(local void , ...)) **If the `cl_khr_device_enqueue_local_arg_types` extension macro is supported:** **uint *get_kernel_work_group_size**(void (^block)(local gentype *, ...))	This provides a mechanism to query the maximum work-group size that can be used to execute a block on a specific device given by *device*. *block* specifies the block to be enqueued.
uint **get_kernel_preferred_work_group_size_multiple**(void (^block)(void)) uint **get_kernel_preferred_work_group_size_multiple**(void (^block)(local void , ...)) **If the `cl_khr_device_enqueue_local_arg_types` extension macro is supported:** **uint *get_kernel_preferred_work_group_size_multiple**(void (^block)(local gentype *, ...))	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 enqueue_kernel will not fail to enqueue the block for execution unless the work-group size specified is larger than the device maximum.

6.15.19.7. Built-in Functions - Queuing Other Commands

The following table describes the list of built-in functions that can be used to enqueue commands such as a marker.

Table 47. Built-in Other Enqueue Functions

Built-in Function	Description
int **enqueue_marker**(queue_t *queue*, uint *num_events_in_wait_list*, const clk_event_t **event_wait_list*, clk_event_t **event_ret*)	Enqueue a marker command to *queue*. The marker command waits for a list of events specified by *event_wait_list* to complete before the marker completes. *event_ret* must not be `NULL` as otherwise this is a no-op. If an event is returned, **enqueue_marker** performs an implicit retain on the returned event.

The **enqueue_marker** built-in function returns `CLK_SUCCESS` if the marked command is enqueued successfully and returns `CLK_ENQUEUE_FAILURE` otherwise. If the -g compile option is specified in compiler options passed to **clCompileProgram** or **clBuildProgram**, the following errors may be returned instead of `CLK_ENQUEUE_FAILURE` to indicate why **enqueue_marker** failed to enqueue the marker command:

- `CLK_INVALID_QUEUE` if *queue* is not a valid device queue.
- `CLK_INVALID_EVENT_WAIT_LIST` if *event_wait_list* is `NULL`, 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.
- `CLK_DEVICE_QUEUE_FULL` if *queue* is full.
- `CLK_EVENT_ALLOCATION_FAILURE` if *event_ret* is not `NULL` and an event could not be allocated.
- `CLK_OUT_OF_RESOURCES` if there is a failure to queue the block in *queue* because of insufficient resources needed to execute the kernel.

6.15.19.8. Built-in Functions - Event Functions

The following table describes the list of built-in functions that work on events.

Table 48. Built-in Event Functions

Built-in Function	Description
void **retain_event**(clk_event_t *event*)	Increments the event reference count. Behavior is undefined if *event* is not a valid event.

Built-in Function	Description
void **release_event**(clk_event_t *event*)	Decrements the event reference count. The event object is deleted once the event reference count is zero, the specific command identified by this event has completed (or terminated), and there are no commands in any device command-queue that require a wait for this event to complete. Behavior is undefined if *event* is not a valid event.
clk_event_t **create_user_event**()	Create a user event. Returns the user event. The execution status of the user event created is set to `CL_SUBMITTED`.
bool **is_valid_event**(clk_event_t *event*)	Returns *true* if *event* is a valid event. Otherwise returns *false*.
void **set_user_event_status**(clk_event_t *event*, int *status*)	Sets the execution status of a user event. Behavior is undefined if *event* is not a valid event returned by **create_user_event**. *status* can be either `CL_COMPLETE` or a negative integer value indicating an error.

Built-in Function	Description
void **capture_event_profiling_info**(clk_event_t *event*, clk_profiling_info *name*, global void * *value*)	Captures the profiling information for functions that are enqueued as commands. These enqueued commands are identified by unique event objects. The profiling information will be available in *value* once the command identified by *event* has completed. Behavior is undefined if *event* is not a valid event returned by **enqueue_kernel**. *name* identifies which profiling information is to be queried and can be: `CLK_PROFILING_COMMAND_EXEC_TIME` *value* is a pointer to two 64-bit values. The first 64-bit value describes the elapsed time `CL_PROFILING_COMMAND_END` - `CL_PROFILING_COMMAND_START` for the command identified by *event* in nanoseconds. The second 64-bit value describes the elapsed time `CL_PROFILING_COMMAND_COMPLETE` - `CL_PROFILING_COMMAND_START` for the command identified by *event* in nanoseconds. The behavior of capture_event_profiling_info when called multiple times for the same *event* is undefined.

Events can be used to identify commands enqueued to a command-queue from the host. These events created by the OpenCL runtime can only be used on the host, i.e. as events passed in the *event_wait_list* argument to various enqueue APIs or runtime APIs that take events as arguments, such as **clRetainEvent**, **clReleaseEvent**, and **clGetEventProfilingInfo**.

Similarly, events can be used to identify commands enqueued to a device queue (from a kernel). These event objects cannot be passed to the host or used by OpenCL runtime APIs such as the enqueue APIs or runtime APIs that take event arguments.

clRetainEvent and **clReleaseEvent** will return `CL_INVALID_OPERATION` if *event* specified is an event that refers to any kernel enqueued to a device queue using **enqueue_kernel** or **enqueue_marker**, or is a user event created by **create_user_event**.

Similarly, **clSetUserEventStatus** can only be used to set the execution status of events created using **clCreateUserEvent**. User events created on the device can be set using set_user_event_status built-in function.

The example below shows how events can be used with kernels enqueued to multiple device queues.

```
extern void barA_kernel(...);
extern void barB_kernel(...);

kernel void
foo(queue_t q0, queue q1, ...)
{
    ...
    clk_event_t evt0;

    // enqueue kernel to queue q0
    enqueue_kernel(q0,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange_A,
                   0, NULL, &evt0,
                   ^{barA_kernel(...);} );

    // enqueue kernel to queue q1
    enqueue_kernel(q1,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange_B,
                   1, &evt0, NULL,
                   ^{barB_kernel(...);} );

    // release event evt0. This will get released
    // after barA_kernel enqueued in queue q0 has finished
    // execution and barB_kernel enqueued in queue q1 and
    // waits for evt0 is submitted for execution, i.e. wait
    // for evt0 is satisfied.
    release_event(evt0);

}
```

The example below shows how the marker command can be used with kernels enqueued to a device queue.

```
kernel void
foo(queue_t q, ...)
{
    ...
    clk_event_t marker_event;
    clk_event_t events[2];

    enqueue_kernel(q,
                   CLK_ENQUEUE_FLAGS_NO_WAIT,
                   ndrange,
                   0, NULL, &events[0],
```

```
                    ^{barA_kernel(...);} );

    enqueue_kernel(q,
                    CLK_ENQUEUE_FLAGS_NO_WAIT,
                    ndrange,
                    0, NULL, &events[1],
                    ^{barB_kernel(...);} );

    // barA_kernel and barB_kernel can be executed
    // out-of-order. We need to wait for both these
    // kernels to finish execution before barC_kernel
    // starts execution so we enqueue a marker command and
    // then enqueue barC_kernel that waits on the event
    // associated with the marker.
    enqueue_marker(q, 2, events, &marker_event);

    enqueue_kernel(q,
                    CLK_ENQUEUE_FLAGS_NO_WAIT,
                    1, &marker_event, NULL,
                    ^{barC_kernel(...);} );

    release_event(events[0];
    release_event(events[1]);
    release_event(marker_event);
}
```

6.15.19.9. Built-in Functions - Helper Functions

Table 49. Built-in Helper Functions

Built-in Function	Description
queue_t **get_default_queue**(void)	Returns the default device queue. If a default device queue has not been created, `CLK_NULL_QUEUE` is returned.

Built-in Function	Description
ndrange_t **ndrange_1D**(size_t *global_work_size*) ndrange_t **ndrange_1D**(size_t *global_work_size*, size_t *local_work_size*) ndrange_t **ndrange_1D**(size_t *global_work_offset*, size_t *global_work_size*, size_t *local_work_size*) ndrange_t **ndrange_2D**(const size_t *global_work_size*[2]) ndrange_t **ndrange_2D**(const size_t *global_work_size*[2], const size_t *local_work_size*[2]) ndrange_t **ndrange_2D**(const size_t *global_work_offset*[2], const size_t *global_work_size*[2], const size_t *local_work_size*[2]) ndrange_t **ndrange_3D**(const size_t *global_work_size*[3]) ndrange_t **ndrange_3D**(const size_t *global_work_size*[3], const size_t *local_work_size*[3]) ndrange_t **ndrange_3D**(const size_t *global_work_offset*[3], const size_t *global_work_size*[3], const size_t *local_work_size*[3])	Builds a 1D, 2D or 3D ND-range descriptor.

6.15.20. Sub-Group Functions

The functionality described in this section requires support for the `cl_khr_subgroups` extension, or OpenCL C 3.0 or newer and the `__opencl_c_subgroups` feature.

The following table describes OpenCL C programming language built-in functions that operate on a sub-group level. These built-in functions must be encountered by all work-items in the sub-group executing the kernel. For the functions below, the generic type name `gentype` may be the one of the supported built-in scalar data types `int`, `uint`, `long` [94], `ulong`, `half` [95], `float`, and `double` [96].

If the `cl_khr_subgroup_extended_types` extension macro or OpenCL C 3.1 or newer is supported, the generic type name `gentype` may additionally be `char`, `uchar`, `short`, and `ushort`. For the `sub_group_broadcast` function, `gentype` may additionally be one of the supported built-in vector data types `charn`, `ucharn`, `shortn`, `ushortn`, `intn`, `uintn`, `longn`, `ulongn`, `floatn`, `halfn` [97], or `doublen` [98].

Table 50. Built-in Sub-Group Collective Functions

Function	Description
int **sub_group_all** (int *predicate*)	Evaluates *predicate* for all work-items in the sub-group and returns a non-zero value if *predicate* evaluates to non-zero for all work-items in the sub-group.
int **sub_group_any** (int *predicate*)	Evaluates *predicate* for all work-items in the sub-group and returns a non-zero value if *predicate* evaluates to non-zero for any work-items in the sub-group.
gentype **sub_group_broadcast** (gentype *x*, uint *sub_group_local_id*)	Broadcast the value of *x* for work-item identified by *sub_group_local_id* (value returned by **get_sub_group_local_id**) to all work-items in the sub-group. Behavior is undefined when the value of *sub_group_local_id* is not equivalent for all work-items in the sub-group. Behavior is undefined when *sub_group_local_id* is greater or equal to the sub-group size.
gentype **sub_group_reduce_<op>** (gentype *x*)	Return result of reduction operation specified by **<op>** for all values of *x* specified by work-items in a sub-group.
gentype **sub_group_scan_exclusive_<op>** (gentype *x*)	Do an exclusive scan operation specified by **<op>** of all values specified by work-items in a sub-group. The scan results are returned for each work-item. The scan order is defined by increasing sub-group local ID within the sub-group.
gentype **sub_group_scan_inclusive_<op>** (gentype *x*)	Do an inclusive scan operation specified by **<op>** of all values specified by work-items in a sub-group. The scan results are returned for each work-item. The scan order is defined by increasing sub-group local ID within the sub-group.

The **<op>** in **sub_group_reduce_<op>**, **sub_group_scan_inclusive_<op>** and **sub_group_scan_exclusive_<op>** defines the operator and can be **add**, **min** or **max**.

The exclusive scan operation takes a binary operator **op** with an identity I and *n* (where *n* is the size of the sub-group) elements [a_0, a_1, ... a_{n-1}] and returns [I, a_0, (a_0 **op** a_1), ... (a_0 **op** a_1 **op** ... **op** a_{n-2})].

The inclusive scan operation takes a binary operator **op** with an identity I and *n* (where *n* is the size of the sub-group) elements [a_0, a_1, ... a_{n-1}] and returns [a_0, (a_0 **op** a_1), ... (a_0 **op** a_1 **op** ... **op** a_{n-1})].

If **op** = **add**, the identity I is 0. If **op** = **min**, the identity I is `INT_MAX`, `UINT_MAX`, `LONG_MAX`, `ULONG_MAX`, for `int`, `uint`, `long`, `ulong` types and is `+INF` for floating-point types. Similarly if **op** = max, the identity I is `INT_MIN`, 0, `LONG_MIN`, 0 and `-INF`.

The order of floating-point operations is not guaranteed for the **sub_group_reduce_<op>**, **sub_group_scan_inclusive_<op>** and **sub_group_scan_exclusive_<op>** built-in functions that operate on `half`, `float` and `double` data types. The order of these floating-point operations is also non-deterministic for a given sub-group.

The functionality described in the following table requires support for the `cl_khr_subgroups` extension, or OpenCL C 3.0 or newer and the `__opencl_c_subgroups` and `__opencl_c_pipes` features.

The following table describes built-in pipe functions that operate at a sub-group level. These built-in functions must be encountered by all work-items in a sub-group executing the kernel with the same argument values, otherwise the behavior is undefined. We use the generic type name `gentype` to indicate the built-in OpenCL C scalar or vector integer or floating-point data types or any user defined type built from these scalar and vector data types can be used as the type for the arguments to the pipe functions listed in *table 6.29*.

Table 51. Built-in Sub-Group Pipe Functions

Function	Description
reserve_id_t **sub_group_reserve_read_pipe** (read_only pipe gentype *pipe*, uint *num_packets*) reserve_id_t **sub_group_reserve_write_pipe** (write_only pipe gentype *pipe*, uint *num_packets*)	Reserve *num_packets* entries for reading from or writing to *pipe*. Returns a valid non-zero reservation ID if the reservation is successful and 0 otherwise. The reserved pipe entries are referred to by indices that go from 0 ... *num_packets* - 1.
void **sub_group_commit_read_pipe** (read_only pipe gentype *pipe*, reserve_id_t *reserve_id*) void **sub_group_commit_write_pipe** (write_only pipe gentype *pipe*, reserve_id_t *reserve_id*)	Indicates that all reads and writes to *num_packets* associated with reservation *reserve_id* are completed.

Note: Reservations made by a sub-group are ordered in the pipe as they are ordered in the program. Reservations made by different sub-groups that belong to the same work-group can be ordered using sub-group synchronization. The order of sub-group based reservations that belong to different work groups is implementation-defined.

The functionality described in the following table requires support the `cl_khr_subgroups` extension, or OpenCL C 3.0 or newer and the `__opencl_c_subgroups` and `__opencl_c_device_enqueue` features.

The following table describes built-in functions to query sub-group information for a block to be enqueued.

Table 52. Built-in Sub-Group Kernel Query Functions

Built-in Function	Description
uint **get_kernel_sub_group_count_for_ndrange** (const ndrange_t *ndrange*, void (^block)(void)); uint **get_kernel_sub_group_count_for_ndrange** (const ndrange_t *ndrange*, void (^block)(local void *, ...));	Returns the number of sub-groups in each work-group of the dispatch (except for the last in cases where the global size does not divide cleanly into work-groups) given the combination of the passed ndrange and block. *block* specifies the block to be enqueued.
uint **get_kernel_max_sub_group_size_for_ndrange** (const ndrange_t *ndrange*, void (^block)(void)); uint **get_kernel_max_sub_group_size_for_ndrange** (const ndrange_t *ndrange*, void (^block)(local void *, ...));	Returns the maximum sub-group size for a block.

6.15.20.1. Built-in Sub-Group Ballot Functions

The functionality described in this section requires support for the `cl_khr_subgroup_ballot` extension.

The following table describes OpenCL C programming language built-in functions to allow work items in a sub-group to collect and operate on ballots from work items in the sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel.

For the `sub_group_non_uniform_broadcast` and `sub_group_broadcast_first` functions, the generic type name `gentype` may be one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `half` [99], and `double` [100].

For the `sub_group_non_uniform_broadcast` function, the generic type name `gentype` may additionally be one of the supported built-in vector data types `charn`, `ucharn`, `shortn`, `ushortn`, `intn`, `uintn`, `longn`, `ulongn`, `floatn`, `halfn` [101], or `doublen` [102].

Table 53. Built-in Sub-Group Ballot Functions

Function	Description
`gentype sub_group_non_uniform_broadcast( gentype value, uint index )`	Returns *value* for the work item with sub-group local ID equal to *index*. Behavior is undefined when the value of *index* is not equivalent for all active work items in the sub-group. The return value is undefined if the work item with sub-group local ID equal to *index* is inactive or if *index* is greater than or equal to the size of the sub-group.
`gentype sub_group_broadcast_first( gentype value )`	Returns *value* for the work item with the smallest sub-group local ID among active work items in the sub-group.
`uint4 sub_group_ballot( int predicate )`	Returns a bitfield combining the *predicate* values from all work items in the sub-group. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs. The representative bit in the bitfield is set if the work item is active and the *predicate* is non-zero, and is unset otherwise.
`int sub_group_inverse_ballot( uint4 value )`	Returns the predicate value for this work item in the sub-group from the bitfield *value* representing predicate values from all work items in the sub-group. The predicate return value will be non-zero if the bit in the bitfield *value* for this work item is set, and zero otherwise. Behavior is undefined when *value* is not equivalent for all active work items in the sub-group. This is a specialized function that may perform better than the equivalent `sub_group_ballot_bit_extract` on some implementations.

Function	Description

`int sub_group_ballot_bit_extract(`
`  uint4 value,`
`  uint index )`
	Returns the predicate value for the work item with sub-group local ID equal to *index* from the bitfield *value* representing predicate values from all work items in the sub-group. The predicate return value will be non-zero if the bit in the bitfield *value* for the work item with sub-group local ID equal to *index* is set, and zero otherwise.
	
	The predicate return value is undefined if the work item with sub-group local ID equal to *index* is greater than or equal to the size of the sub-group.

`uint sub_group_ballot_bit_count(` `  uint4 value )`	Returns the number of bits that are set in the bitfield *value*, only considering the bits in *value* that represent predicate values corresponding to sub-group local IDs less than the maximum sub-group size within the dispatch (as returned by `get_max_sub_group_size`).
`uint sub_group_ballot_inclusive_scan(` `  uint4 value )`	Returns the number of bits that are set in the bitfield *value*, only considering the bits in *value* representing work items with a sub-group local ID less than or equal to this work item's sub-group local ID.
`uint sub_group_ballot_exclusive_scan(` `  uint4 value )`	Returns the number of bits that are set in the bitfield *value*, only considering the bits in *value* representing work items with a sub-group local ID less than this work item's sub-group local ID.
`uint sub_group_ballot_find_lsb(` `  uint4 value )`	Returns the smallest sub-group local ID with a bit set in the bitfield *value*, only considering the bits in *value* that represent predicate values corresponding to sub-group local IDs less than the maximum sub-group size within the dispatch (as returned by `get_max_sub_group_size`). If no bits representing predicate values from all work items in the sub-group are set in the bitfield *value* then the return value is undefined.

Function	Description
`uint sub_group_ballot_find_msb( uint4 value )`	Returns the largest sub-group local ID with a bit set in the bitfield *value*, only considering the bits in *value* that represent predicate values corresponding to sub-group local IDs less than the maximum sub-group size within the dispatch (as returned by `get_max_sub_group_size`). If no bits representing predicate values from all work items in the sub-group are set in the bitfield *value* then the return value is undefined.
`uint4 get_sub_group_eq_mask()`	Generates a bitmask where the bit is set in the bitmask if the bit index equals the sub-group local ID and unset otherwise. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs.
`uint4 get_sub_group_ge_mask()`	Generates a bitmask where the bit is set in the bitmask if the bit index is greater than or equal to the sub-group local ID and less than the maximum sub-group size, and unset otherwise. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs.
`uint4 get_sub_group_gt_mask()`	Generates a bitmask where the bit is set in the bitmask if the bit index is greater than the sub-group local ID and less than the maximum sub-group size, and unset otherwise. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs.
`uint4 get_sub_group_le_mask()`	Generates a bitmask where the bit is set in the bitmask if the bit index is less than or equal to the sub-group local ID and unset otherwise. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs.

Function	Description
`uint4 get_sub_group_lt_mask()`	Generates a bitmask where the bit is set in the bitmask if the bit index is less than the sub-group local ID and unset otherwise. Bit zero of the first vector component represents the sub-group local ID zero, with higher-order bits and subsequent vector components representing, in order, increasing sub-group local IDs.

6.15.20.2. Built-in Sub-Group Clustered Reduction Functions

The functionality described in this section requires support for the `cl_khr_subgroup_clustered_reduce` extension.

This section describes arithmetic operations that are performed on a subset of work items in a sub-group, referred to as a cluster. A cluster is described by a specified cluster size. Work items in a sub-group are assigned to clusters such that for cluster size *n*, the *n* work items in the sub-group with the smallest sub-group local IDs are assigned to the first cluster, then the *n* remaining work items with the smallest sub-group local IDs are assigned to the next cluster, and so on.

6.15.20.2.1. Arithmetic Operations

The table below describes the OpenCL C programming language built-in functions that perform simple arithmetic operations on a cluster of work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name `gentype` may be one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `half` [103], and `double` [104].

Table 54. Built-in Sub-Group Clustered Reduction Arithmetic Functions

Function	Description

`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 )`
	Returns the summation, multiplication, minimum, or maximum of *value* for all active work items in the sub-group within a cluster of the specified *clustersize*.
	
	Behavior is undefined if *clustersize* is not an integer constant expression, is not a power-of-two, or is greater than the maximum size of a sub-group within the dispatch.

Note: The order of floating-point operations is not guaranteed for the sub-group clustered reduction built-in functions that operate on floating-point types, and the order of operations may additionally be non-deterministic for a given sub-group.

6.15.20.2.2. Bitwise Operations

The table below describes the OpenCL C programming language built-in functions to perform

simple bitwise integer operations across a cluster of work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name `gentype` may be the one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Table 55. Built-in Sub-Group Clustered Reduction Bitwise Functions

Function	Description

``
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)
``
	Returns the bitwise **and**, **or**, or **xor** of *value* for all active work items in the sub-group within a cluster of the specified *clustersize*.
	
	Behavior is undefined if *clustersize* is not an integer constant expression, is not a power-of-two, or is greater than the maximum size of a sub-group within the dispatch.

6.15.20.2.3. Logical Operations

The table below describes the OpenCL C programming language built-in functions to perform simple logical operations across a cluster of work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For these functions, a non-zero *predicate* argument or return value is logically `true` and a zero *predicate* argument or return value is logically `false`.

Table 56. Built-in Sub-Group Clustered Reduction Logical Functions

Function	Description

``
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)
``
	Returns the logical **and**, **or**, or **xor** of *predicate* for all active work items in the sub-group within a cluster of the specified *clustersize*.
	
	Behavior is undefined if *clustersize* is not an integer constant expression, is not a power-of-two, or is greater than the maximum size of a sub-group within the dispatch.

6.15.20.3. Built-in Sub-Group Non-Uniform Scan and Reduction Functions

The functionality described in this section requires support for the `cl_khr_subgroup_non_uniform_arithmetic` extension.

6.15.20.3.1. Arithmetic Operations

The following table describes the OpenCL C programming language built-in functions that perform simple arithmetic operations across work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the

generic type name `gentype` may be one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `half` [105], and `double` [106].

Table 57. Built-in Sub-Group Non-Uniform Arithmetic Functions

Function	Description

``
gentype sub_group_non_uniform_reduce_add(
  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_mul(
  gentype value)
``
	Returns the summation, multiplication, minimum, or maximum of *value* for all active work items in the sub-group.
	
	Note: This behavior is the same as the **add**, **min**, and **max** reduction built-in functions from `cl_khr_subgroups` and OpenCL 2.1, except these functions support additional types and need not be encountered by all work items in the sub-group executing the kernel.

``
gentype sub_group_non_uniform_scan_inclusive_add(
  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_mul(
  gentype value)
``
	Returns the result of an inclusive scan operation, which is the summation, multiplication, minimum, or maximum of *value* for all active work items in the sub-group with a sub-group local ID less than or equal to this work item's sub-group local ID.
	
	Note: This behavior is the same as the **add**, **min**, and **max** inclusive scan built-in functions from `cl_khr_subgroups` and OpenCL 2.1, except these functions support additional types and need not be encountered by all work items in the sub-group executing the kernel.

Function	Description

`gentype sub_group_non_uniform_scan_exclusive_add( 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_mul( gentype value )`
	Returns the result of an exclusive scan operation, which is the summation, multiplication, minimum, or maximum of *value* for all active work items in the sub-group with a sub-group local ID less than this work item's sub-group local ID.
	
	If there is no active work item in the sub-group with a sub-group local ID less than this work item's sub-group local ID then an identity value `I` is returned. For **add**, the identity value is `0`. For **min**, the identity value is the largest representable value for integer types, or `+INF` for floating-point types. For **max**, the identity value is the minimum representable value for integer types, or `-INF` for floating-point types. For **mul**, the identity value is `1`.
	
	Note: This behavior is the same as the **add**, **min**, and **max** exclusive scan built-in functions from `cl_khr_subgroups` and OpenCL 2.1, except these functions support additional types and need not be encountered by all work items in the sub-group executing the kernel.

Note: The order of floating-point operations is not guaranteed for the sub-group scan and reduction built-in functions that operate on floating-point types, and the order of operations may additionally be non-deterministic for a given sub-group.

6.15.20.3.2. Bitwise Operations

The table below describes the OpenCL C programming language built-in functions that perform simple bitwise integer operations across work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name `gentype` may be one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`.

Table 58. Built-in Sub-Group Non-Uniform Bitwise Functions

Function	Description

`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 )`
	Returns the bitwise **and**, **or**, or **xor** of *value* for all active work items in the sub-group.

`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 )`
	Returns the result of an inclusive scan operation, which is the bitwise **and**, **or**, or **xor** of *value* for all active work items in the sub-group with a sub-group local ID less than or equal to this work item's sub-group local ID.

`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 )`
	Returns the result of an exclusive scan operation, which is the bitwise **and**, **or**, or **xor** of *value* for all active work items in the sub-group with a sub-group local ID less than this work item's sub-group local ID.
	
	If there is no active work item in the sub-group with a sub-group local ID less than this work item's sub-group local ID then an identity value `I` is returned. For **and**, the identity value is `~0` (all bits set). For **or** and **xor**, the identity value is `0`.

6.15.20.3.3. Logical Operations

The table below describes the OpenCL C programming language built-in functions that perform simple logical operations across work items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For these functions, a non-zero *predicate* argument or return value is logically `true` and a zero *predicate* argument or return value is logically `false`.

Table 59. Built-in Sub-Group Non-Uniform Logical Functions

Function	Description

`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 )`
	Returns the logical **and**, **or**, or **xor** of *predicate* for all active work items in the sub-group.

`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 )`
	Returns the result of an inclusive scan operation, which is the logical **and**, **or**, or **xor** of *predicate* for all active work items in the sub-group with a sub-group local ID less than or equal to this work item's sub-group local ID.

`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 )`
	Returns the result of an exclusive scan operation, which is the logical **and**, **or**, or **xor** of *predicate* for all active work items in the sub-group with a sub-group local ID less than this work item's sub-group local ID.
	
	If there is no active work item in the sub-group with a sub-group local ID less than this work item's sub-group local ID then an identity value `I` is returned. For **and**, the identity value is `true` (non-zero). For **or** and **xor**, the identity value is `false` (zero).

6.15.20.4. Built-in Sub-Group Non-Uniform Vote Functions

The functionality described in this section requires support for the `cl_khr_subgroup_non_uniform_vote` extension.

The following table describes the OpenCL C programming language built-in functions to elect a single work item in a sub-group to perform a task and to collectively vote to determine a boolean condition for the sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name `gentype` may be the one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `half` [107], and `double` [108].

Table 60. Built-in Sub-Group Non-Uniform Vote Functions

Function	Description
`int sub_group_elect()`	Elects a single work item in the sub-group to perform a task. This function will return true (nonzero) for the active work item in the sub-group with the smallest sub-group local ID, and false (zero) for all other active work items in the sub-group.
`int sub_group_non_uniform_all( int predicate )`	Examines *predicate* for all active work items in the sub-group and returns a non-zero value if *predicate* is non-zero for all active work items in the sub-group and zero otherwise. Note: This behavior is the same as `sub_group_all` from `cl_khr_subgroups` and OpenCL 2.1, except this function need not be encountered by all work items in the sub-group executing the kernel.
`int sub_group_non_uniform_any( int predicate )`	Examines *predicate* for all active work items in the sub-group and returns a non-zero value if *predicate* is non-zero for any active work item in the sub-group and zero otherwise. Note: This behavior is the same as `sub_group_any` from `cl_khr_subgroups` and OpenCL 2.1, except this function need not be encountered by all work items in the sub-group executing the kernel.
`int sub_group_non_uniform_all_equal( gentype value )`	Examines *value* for all active work items in the sub-group and returns a non-zero value if *value* is equivalent for all active invocations in the sub-group and zero otherwise. Integer types use a bitwise test for equality. Floating-point types use an ordered floating-point test for equality.

6.15.20.5. Built-in Sub-Group Rotation Functions

If the `cl_khr_subgroup_rotate` extension macro or OpenCL C 3.1 or newer is supported, the functions described in the Built-in Sub-Group Rotation Functions table can be used to exchange data by rotating values through the work-items in a sub-group. This function need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name `gentype` may be one of the supported built-in scalar data types `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `half` [109], and `double` [110].

The functionality described in the following table requires support for the cl_khr_subgroup_rotate extension macro or OpenCL C 3.1 or newer.

Table 61. Built-in Sub-Group Rotation Functions

Function	Description
`gentype sub_group_rotate(` `  gentype value, int delta)`	Returns *value* for the work item with sub-group local ID equal to the remainder of the division of the sum of this work item's sub-group local ID and *delta* by the maximum sub-group size. The value of *delta* is required to be dynamically-uniform for all work items in the sub-group, otherwise the behavior is undefined. The return value is undefined if the work item with sub-group local ID equal to the calculated index is inactive.

`gentype sub_group_clustered_rotate(`
`  gentype value, int delta,`
`  uint clustersize)`
	Returns *value* for the work item with sub-group local ID equal to the sum of, the remainder of the division of the sum of this work item's ID within the cluster and *delta* by *clustersize*, and the sub-group local ID of the first work-item of the cluster to which the work-item executing the function belongs.
	The value of *delta* is required to be dynamically-uniform for all work items in the sub-group, otherwise the behavior is undefined.
	
	Behavior is undefined if *clustersize* is not an integer constant expression, is not a power-of-two, or is greater than the maximum size of a sub-group within the dispatch.
	
	The return value is undefined if the work item with sub-group local ID equal to the calculated index is inactive.

6.15.20.6. Built-in Sub-Group General Purpose Shuffle Functions

If the cl_khr_subgroup_shuffle extension macro or OpenCL C 3.1 or newer is supported, the functions described in the Built-in Sub-Group General Purpose Shuffle Functions table can be used to flexibly exchange data among work-items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name gentype may be one of the supported built-in scalar data types char, uchar, short, ushort, int, uint, long, ulong, float, half [111], and double [112].

The functionality described in this section requires support for the cl_khr_

subgroup_shuffle extension macro or OpenCL C 3.1 or newer.

Table 62. Built-in Sub-Group General Purpose Shuffle Functions

Function	Description
`gentype sub_group_shuffle( gentype value, uint index )`	Returns *value* for the work item with sub-group local ID equal to *index*. The shuffle *index* need not be the same for all work items in the sub-group. The return value is undefined if the work item with sub-group local ID equal to *index* is inactive or if *index* is greater than or equal to the size of the sub-group.
`gentype sub_group_shuffle_xor( gentype value, uint mask )`	Returns *value* for the work item with sub-group local ID equal to this work item's sub-group local ID xor'd with *mask*. The shuffle *mask* need not be the same for all work items in the sub-group. The return value is undefined if the work item with sub-group local ID equal to the calculated index is inactive or if the calculated index is greater than or equal to the size of the sub-group. This is a specialized function that may perform better than the equivalent sub_group_shuffle on some implementations.

6.15.20.7. Built-in Sub-Group Relative Shuffle Functions

If the cl_khr_subgroup_shuffle_relative extension macro or OpenCL C 3.1 or newer is supported, the functions described in the Built-in Sub-Group Relative Shuffle Functions table can be used to exchange data among relative work-items in a sub-group. These functions need not be encountered by all work items in a sub-group executing the kernel. For the functions below, the generic type name gentype may be one of the supported built-in scalar data types char, uchar, short, ushort, int, uint, long, ulong, float, half [113], and double [114].

The functionality described in this section requires support for the cl_khr_subgroup_shuffle_relative extension macro or OpenCL C 3.1 or newer.

Table 63. Built-in Sub-Group Relative Shuffle Functions

Function	Description
`gentype sub_group_shuffle_up( gentype value, uint delta )`	Returns *value* for the work item with sub-group local ID equal to this work item's sub-group local ID minus *delta*. The shuffle *delta* need not be the same for all work items in the sub-group. The return value is undefined if the work item with sub-group local ID equal to the calculated index is inactive, or *delta* is greater than this work item's sub-group local ID. This is a specialized function that may perform better than the equivalent `sub_group_shuffle` on some implementations.
`gentype sub_group_shuffle_down( gentype value, uint delta )`	Returns *value* for the work item with sub-group local ID equal to this work item's sub-group local ID plus *delta*. The shuffle *delta* need not be the same for all work items in the sub-group. The return value is undefined if the work item with sub-group local ID equal to the calculated index is inactive, or this work item's sub-group local ID plus *delta* is greater than or equal to the size of the sub-group. This is a specialized function that may perform better than the equivalent `sub_group_shuffle` on some implementations.

6.15.20.8. Sub-Groups Function Mapping and Capabilities

This section describes a possible mapping between OpenCL built-in sub-group functions and SPIR-V instructions and required SPIR-V capabilities.

This section is informational and non-normative.

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
For OpenCL C 2.1, OpenCL C 3.0 and the `__opencl_c_subgroups` feature, OpenCL C 3.1 or newer, or `cl_khr_subgroups`:		
`get_sub_group_size`	**SubgroupSize**	**Kernel**
`get_max_sub_group_size`	**SubgroupMaxSize**	**Kernel**
`get_num_sub_groups`	**NumSubgroups**	**Kernel**
`get_enqueued_num_sub_groups`	**NumEnqueuedSubgroups**	**Kernel**
`get_sub_group_id`	**SubgroupId**	**Kernel**

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
`get_sub_group_local_id`	**SubgroupLocalInvocationId**	**Kernel**
`sub_group_barrier`	**OpControlBarrier**	None Needed
`sub_group_all`	**OpGroupAll**	**Groups**
`sub_group_any`	**OpGroupAny**	**Groups**
`sub_group_broadcast`	**OpGroupBroadcast**	**Groups**
`sub_group_reduce_add`	**OpGroupIAdd**, **OpGroupFAdd**	**Groups**
`sub_group_reduce_min`	**OpGroupSMin**, **OpGroupUMin**, **OpGroupFMin**	**Groups**
`sub_group_reduce_max`	**OpGroupSMax**, **OpGroupUMax**, **OpGroupFMax**	**Groups**
`sub_group_scan_exclusive_add`	**OpGroupIAdd**, **OpGroupFAdd**	**Groups**
`sub_group_scan_exclusive_min`	**OpGroupSMin**, **OpGroupUMin**, **OpGroupFMin**	**Groups**
`sub_group_scan_exclusive_max`	**OpGroupSMax**, **OpGroupUMax**, **OpGroupFMax**	**Groups**
`sub_group_scan_inclusive_add`	**OpGroupIAdd**, **OpGroupFAdd**	**Groups**
`sub_group_scan_inclusive_min`	**OpGroupSMin**, **OpGroupUMin**, **OpGroupFMin**	**Groups**
`sub_group_scan_inclusive_max`	**OpGroupSMax**, **OpGroupUMax**, **OpGroupFMax**	**Groups**
`sub_group_reserve_read_pipe`	**OpGroupReserveReadPipePackets**	**Pipes**
`sub_group_reserve_write_pipe`	**OpGroupReserveReadWritePackets**	**Pipes**
`sub_group_commit_read_pipe`	**OpGroupCommitReadPipe**	**Pipes**
`sub_group_commit_write_pipe`	**OpGroupCommitWritePipe**	**Pipes**
`get_kernel_sub_group_count_for_ndrange`	**OpGetKernelNDrangeSubGroupCount**	**DeviceEnqueue**
`get_kernel_max_sub_group_size_for_ndrange`	**OpGetKernelNDrangeMaxSubGroupSize**	**DeviceEnqueue**
For `cl_khr_subgroup_ballot`:		
`sub_group_non_uniform_broadcast`	**OpGroupNonUniformBroadcast**	**GroupNonUniformBallot**

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
`sub_group_broadcast_first`	**OpGroupNonUniformBroadcastFirst**	**GroupNonUniformBallot**
`sub_group_ballot`	**OpGroupNonUniformBallot**	**GroupNonUniformBallot**
`sub_group_inverse_ballot`	**OpGroupNonUniformInverseBallot**	**GroupNonUniformBallot**
`sub_group_ballot_bit_extract`	**OpGroupNonUniformBallotBitExtract**	**GroupNonUniformBallot**
`sub_group_ballot_bit_count`	**OpGroupNonUniformBallotBitCount**	**GroupNonUniformBallot**
`sub_group_ballot_inclusive_scan`	**OpGroupNonUniformBallotBitCount**	**GroupNonUniformBallot**
`sub_group_ballot_exclusive_scan`	**OpGroupNonUniformBallotBitCount**	**GroupNonUniformBallot**
`sub_group_ballot_find_lsb`	**OpGroupNonUniformBallotFindLSB**	**GroupNonUniformBallot**
`sub_group_ballot_find_msb`	**OpGroupNonUniformBallotFindMSB**	**GroupNonUniformBallot**
`get_sub_group_eq_mask`	**SubgroupEqMask**	**GroupNonUniformBallot**
`get_sub_group_ge_mask`	**SubgroupGeMask**	**GroupNonUniformBallot**
`get_sub_group_gt_mask`	**SubgroupGtMask**	**GroupNonUniformBallot**
`get_sub_group_le_mask`	**SubgroupLeMask**	**GroupNonUniformBallot**
`get_sub_group_lt_mask`	**SubgroupLtMask**	**GroupNonUniformBallot**
For `cl_khr_subgroup_clustered_reduce`:		
`sub_group_clustered_reduce_add`	**OpGroupNonUniformIAdd**, **OpGroupNonUniformFAdd**	**GroupNonUniformClustered**
`sub_group_clustered_reduce_mul`	**OpGroupNonUniformIMul**, **OpGroupNonUniformFMul**	**GroupNonUniformClustered**
`sub_group_clustered_reduce_min`	**OpGroupNonUniformSMin**, **OpGroupNonUniformUMin**, **OpGroupNonUniformFMin**	**GroupNonUniformClustered**
`sub_group_clustered_reduce_max`	**OpGroupNonUniformSMax**, **OpGroupNonUniformUMax**, **OpGroupNonUniformFMax**	**GroupNonUniformClustered**
`sub_group_clustered_reduce_and`	**OpGroupNonUniformBitwiseAnd**	**GroupNonUniformClustered**
`sub_group_clustered_reduce_or`	**OpGroupNonUniformBitwiseOr**	**GroupNonUniformClustered**

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
sub_group_clustered_reduce_xor	**OpGroupNonUniformBitwiseXor**	**GroupNonUniformClustered**
sub_group_clustered_reduce_logical_and	**OpGroupNonUniformLogicalAnd**	**GroupNonUniformClustered**
sub_group_clustered_reduce_logical_or	**OpGroupNonUniformLogicalOr**	**GroupNonUniformClustered**
sub_group_clustered_reduce_logical_xor	**OpGroupNonUniformLogicalXor**	**GroupNonUniformClustered**
For cl_khr_subgroup_non_uniform_arithmetic:		
sub_group_non_uniform_reduce_add	**OpGroupNonUniformIAdd**, **OpGroupNonUniformFAdd**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_mul	**OpGroupNonUniformIMul**, **OpGroupNonUniformFMul**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_min	**OpGroupNonUniformSMin**, **OpGroupNonUniformUMin**, **OpGroupNonUniformFMin**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_max	**OpGroupNonUniformSMax**, **OpGroupNonUniformUMax**, **OpGroupNonUniformFMax**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_and	**OpGroupNonUniformBitwiseAnd**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_or	**OpGroupNonUniformBitwiseOr**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_xor	**OpGroupNonUniformBitwiseXor**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_logical_and	**OpGroupNonUniformLogicalAnd**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_logical_or	**OpGroupNonUniformLogicalOr**	**GroupNonUniformArithmetic**
sub_group_non_uniform_reduce_logical_xor	**OpGroupNonUniformLogicalXor**	**GroupNonUniformArithmetic**
sub_group_non_uniform_scan_inclusive_add	**OpGroupNonUniformIAdd**, **OpGroupNonUniformFAdd**	**GroupNonUniformArithmetic**
sub_group_non_uniform_scan_inclusive_mul	**OpGroupNonUniformIMul**, **OpGroupNonUniformFMul**	**GroupNonUniformArithmetic**
sub_group_non_uniform_scan_inclusive_min	**OpGroupNonUniformSMin**, **OpGroupNonUniformUMin**, **OpGroupNonUniformFMin**	**GroupNonUniformArithmetic**

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
`sub_group_non_uniform_scan_inclusive_max`	**OpGroupNonUniformSMax**, **OpGroupNonUniformUMax**, **OpGroupNonUniformFMax**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_and`	**OpGroupNonUniformBitwiseAnd**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_or`	**OpGroupNonUniformBitwiseOr**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_xor`	**OpGroupNonUniformBitwiseXor**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_logical_and`	**OpGroupNonUniformLogicalAnd**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_logical_or`	**OpGroupNonUniformLogicalOr**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_inclusive_logical_xor`	**OpGroupNonUniformLogicalXor**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_add`	**OpGroupNonUniformIAdd**, **OpGroupNonUniformFAdd**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_mul`	**OpGroupNonUniformIMul**, **OpGroupNonUniformFMul**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_min`	**OpGroupNonUniformSMin**, **OpGroupNonUniformUMin**, **OpGroupNonUniformFMin**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_max`	**OpGroupNonUniformSMax**, **OpGroupNonUniformUMax**, **OpGroupNonUniformFMax**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_and`	**OpGroupNonUniformBitwiseAnd**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_or`	**OpGroupNonUniformBitwiseOr**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_xor`	**OpGroupNonUniformBitwiseXor**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_logical_and`	**OpGroupNonUniformLogicalAnd**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_logical_or`	**OpGroupNonUniformLogicalOr**	**GroupNonUniformArithmetic**
`sub_group_non_uniform_scan_exclusive_logical_xor`	**OpGroupNonUniformLogicalXor**	**GroupNonUniformArithmetic**
For `cl_khr_subgroup_non_uniform_vote`:		
`sub_group_elect`	**OpGroupNonUniformElect**	**GroupNonUniform**

OpenCL C Function	SPIR-V BuiltIn or Instruction	Enabling SPIR-V Capability
sub_group_non_uniform_all	**OpGroupNonUniformAll**	**GroupNonUniformVote**
sub_group_non_uniform_any	**OpGroupNonUniformAny**	**GroupNonUniformVote**
sub_group_non_uniform_all_equal	**OpGroupNonUniformAllEqual**	**GroupNonUniformVote**
For OpenCL C 3.1 or newer or cl_khr_subgroup_rotate:		
sub_group_rotate	**OpGroupNonUniformRotateKHR**	**GroupNonUniformRotateKHR**
sub_group_clustered_rotate	**OpGroupNonUniformRotateKHR**	**GroupNonUniformRotateKHR**
For OpenCL C 3.1 or newer or cl_khr_subgroup_shuffle:		
sub_group_shuffle	**OpGroupNonUniformShuffle**	**GroupNonUniformShuffle**
sub_group_shuffle_xor	**OpGroupNonUniformShuffleXor**	**GroupNonUniformShuffle**
For OpenCL C 3.1 or newer or cl_khr_subgroup_shuffle_relative:		
sub_group_shuffle_up	**OpGroupNonUniformShuffleUp**	**GroupNonUniformShuffleRelative**
sub_group_shuffle_down	**OpGroupNonUniformShuffleDown**	**GroupNonUniformShuffleRelative**

6.15.21. Kernel Clock Functions

The functionality described in this section requires support for the cl_khr_kernel_clock extension.
The clock_read_device and clock_read_hilo_device functions require support for the __opencl_c_kernel_clock_scope_device feature. The clock_read_work_group and clock_read_hilo_work_group functions require support for the __opencl_c_kernel_clock_scope_work_group feature. The clock_read_sub_group and clock_read_hilo_sub_group functions require support for the __opencl_c_kernel_clock_scope_sub_group feature.

This section describes OpenCL C built-in functions that sample the value from one of three clocks provided by compute units.

Table 64. Built-in Kernel Clock Functions

Function	Description

`ulong clock_read_device();`
`ulong clock_read_work_group();`
`ulong clock_read_sub_group();`
	Returns a sampled value of a clock as seen by the compute unit.
	
	An idealized clock is an unbounded unsigned scalar integer tick count increasing monotonically over time. A clock's rate of progress may vary within the lifetime of a work-item, may vary across different executions of the program, and may be affected by conditions beyond the control of the programmer. The sampled value read by this function consists of the least significant bits of the idealized clock's tick count at the time the instruction was executed. In particular, an observer may see sampled values wrap around zero.

`uint2 clock_read_hilo_device();`
`uint2 clock_read_hilo_work_group();`
`uint2 clock_read_hilo_sub_group();`
	Performs the same operation as `clock_read`, but returns the value as a `uint2` whose `.lo` component contains the 32 least significant bits of the result and `.hi` component contains the 32 most significant bits of the result.

[1] When any scalar value is converted to `bool`, the result is 0 if the value compares equal to 0; otherwise, the result is 1.

[2] The `long`, `unsigned long` and `ulong` scalar types are optional types for EMBEDDED profile devices that are supported if the value of the `CL_DEVICE_EXTENSIONS` device query contains **cles_khr_int64**. An OpenCL C 3.0 compiler must also define the `__opencl_c_int64` feature macro unconditionally for FULL profile devices, or for EMBEDDED profile devices that support these types.

[3] The `double` scalar type is an optional type that is supported if the value of the `CL_DEVICE_DOUBLE_FP_CONFIG` device query is not zero. If this is the case then an OpenCL C 3.0 compiler must also define the `__opencl_c_fp64` feature macro.

[4] This is a 32-bit type if the value of the `CL_DEVICE_ADDRESS_BITS` device query is 32-bits, and a 64-bit type if the value of the query is 64-bits.

[5] Requires support for OpenCL C 1.2 or above. Also see extension `cl_khr_fp64`.

[6] Built-in vector data types are supported by the OpenCL implementation even if the underlying compute device does not natively support any or all of the vector data types. They are to be converted by the device compiler to appropriate instructions that use underlying built-in types supported natively by the compute device. Refer to Appendix B in the OpenCL API specification for a description of the order of the components of a vector type in memory.

[7] The `longn` and `ulongn` vector types are optional types for EMBEDDED profile devices that are supported if the value of the `CL_DEVICE_EXTENSIONS` device query contains **cles_khr_int64**. An OpenCL C 3.0 compiler must also define the `__opencl_c_int64` feature macro unconditionally for FULL profile devices, or for EMBEDDED profile devices that support these types.

[8] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[9] The `doublen` vector type is an optional type that is supported if the value of the `CL_DEVICE_DOUBLE_FP_CONFIG` device query is not zero. If this is the case then an OpenCL C 3.0 compiler must also define the `__opencl_c_fp64` feature macro.

[10] Refer to the detailed description of the built-in Image Read and Write Functions that use this type.

[11] That is, for the purpose of applying type-based aliasing rules, a built-in vector data type will be considered equivalent to the corresponding array type.

[12] Unless the `cl_khr_fp16` extension is supported and has been enabled.

[13] Unless the `cl_khr_fp16` extension is supported and has been enabled.

[14] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_int64` feature macro.

[15] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[16] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[17] For conversions to floating-point format, when a finite source value exceeds the maximum representable finite floating-point destination value, the rounding mode will affect whether the result is the maximum finite floating-point value or infinity of same sign as the source value, per IEEE-754 rules for rounding.

[18] In addition, some other extensions to the C language designed to support a particular vector ISA (e.g. AltiVec™, CELL Broadband Engine™ Architecture) use such conversions in conjunction with swizzle operators to achieve type un-conversion. So as to support legacy code of this type, **as_typen**() allows conversions between vectors of the same size but different numbers of elements, even though the behavior of this sort of conversion is not likely to be portable except to other OpenCL implementations for the same hardware architecture.
AltiVec is a trademark of Motorola Inc.
Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

[19] Unless the `cl_khr_fp16` extension is supported and has been enabled.

[20] While the union is intended to reflect the organization of data in memory, the **as_type**() and **as_typen*n***() constructs are intended to reflect the organization of data in register. The **as_type**() and **as_typen*n***() constructs are intended to compile to no instructions on devices that use a shared register file designed to operate on both the operand and result types. Note that while differences in memory organization are expected to largely be limited to those arising from endianness, the register based representation may also differ due to size of the element in register. For example, an architecture may load a `char` into a 32-bit register, or a `char` vector into a SIMD vector register with fixed 32-bit element size. If the element count does not match, then the implementation should pick a data representation that most closely matches what would happen if an appropriate result type operator was applied to a register containing data of the source type. If the number of elements matches, then the **as_typen*n***() should faithfully reproduce the behavior expected from a similar data type reinterpretation using memory/unions. So, for example if an implementation stores all single precision data as `double` in register, it should implement **as_int**(`float`) by first down-converting the `double` to single precision and then (if necessary) moving the single precision bits to a register suitable for operating on integer data. If data stored in different address spaces do not have the same endianness, then the “dominant endianness” of the device should prevail.

[21] This is different from the standard integer conversion rank described in section 6.3.1.1 of the C99 Specification.

[22] The pre- and post- increment operators may have unexpected behavior on floating-point values and are therefore not supported for floating-point scalar and vector built-in types. For example, if variable *a* has type `float` and holds the value `0x1.0p25f`, then `a++` returns `0x1.0p25f`.
Also, `(a++)--` is not guaranteed to return *a*, if *a* has fractional value.
In non-default rounding modes, `(a++)--` may produce the same result as `a++` or `a--` for large *a*.

[23] To test whether any or all elements in the result of a vector relational operator test *true*, for example to use in the context in an **if ( )** statement, please see the **any** and **all** built-ins.

[24] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[25] To test whether any or all elements in the result of a vector relational operator test *true*, for example to use in the context in an **if ( )** statement, please see the **any** and **all** built-ins.

[26] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[27] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[28] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[29] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[30] Integer promotion is described in section 6.3.1.1 of the C99 Specification.

[31] Variable length arrays are not supported in OpenCL C.

[32] Except for 3-component vectors whose size is defined as 4 times the size of each scalar component.

[33] Bit-field struct members are not supported in OpenCL C.

[34] Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime. If ***P** is an l-value and **T** is the name of an object pointer type, ***(T)P** is an l-value that has a type compatible with that to which **T** points.

[35] Thus, **&*E** is equivalent to **E** (even if **E** is a null pointer), and **&(E1[E2])** is equivalent to **E1)** + **(E2**. It is always true that if **E** is an l-value that is a valid operand of the unary **&** operator, ***&E** is an l-value equal to **E**.

[36] Implicit in autovectorization is the assumption that any libraries called from the `__kernel` must be recompilable at run time to handle cases where the compiler decides to merge or separate workitems. This probably means that such libraries can never be hard coded binaries or that hard coded binaries must be accompanied either by source or some retargetable intermediate representation. This may be a code security question for some.

[37] Unless the `cl_khr_fp16` extension is supported and has been enabled.

[38] When OpenCL C is compiled offline, `__OPENCL_VERSION__` may be defined and may substitute any implementation-defined integer value.

[39] This syntax is already part of the clang source tree on which most vendors have based their OpenCL implementations. Additionally, blocks based closures are supported by the clang open source C compiler as well as Mac OS X’s C and Objective C

compilers. Specifically, Mac OS X's Grand Central Dispatch allows applications to queue tasks as a block.

[40] OpenCL C does not allow function pointers primarily because it is difficult or expensive to implement generic indirections to executable code in many hardware architectures that OpenCL targets. OpenCL C's design of Blocks is intended to respect that same condition, yielding the restrictions listed here. As such, Blocks allow a form of dynamically enqueued function scheduling without providing a form of runtime synchronous dynamic dispatch analogous to function pointers.

[41] I.e. the *global_work_size* values specified to **clEnqueueNDRangeKernel** are not evenly divisible by the *local_work_size* values for each dimension.

[42] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[43] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[44] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[45] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[46] **fmin** and **fmax** behave as defined by C99 and may not match the IEEE 754-2008 definition for **minNum** and **maxNum** with regard to signaling NaNs. Specifically, signaling NaNs may behave as quiet NaNs.

[47] The **min**() operator is there to prevent **fract**(-small) from returning 1.0. It returns the largest positive floating-point number less than 1.0.

[48] The user is cautioned that for some usages, e.g. **mad**(a, b, -a*b), the definition of **mad**() is loose enough in the embedded profile or with half-precision arguments that almost any result is allowed from **mad**() for some values of a and b.

[49] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_int64` feature macro.

[50] Frequently vector operations need n + 1 bits temporarily to calculate a result. The **rhadd** instruction gives you an extra bit without needing to upsample and downsample. This can be a profound performance win.

[51] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[52] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[53] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[54] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[55] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[56] If an implementation extends this specification to support IEEE-754 flags or exceptions, then all built-in functions defined in the following table shall proceed without raising the *invalid* floating-point exception when one or more of the operands are NaNs.

[57] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_int64` feature macro.

[58] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[59] This definition means that the behavior of select and the ternary operator for vector and scalar types is dependent on different interpretations of the bit pattern of *c*.

[60] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_int64` feature macro.

[61] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[62] **vload3** and **vload_half3** read (*x*, *y*, *z*) components from address `(p + (offset * 3))` into a 3-component vector. **vstore3** and **vstore_half3** write (*x*, *y*, *z*) components from a 3-component vector to address `(p + (offset * 3))`. In addition, **vloada_half3** reads (*x*, *y*, *z*) components from address `(p + (offset * 4))` into a 3-component vector and **vstorea_half3** writes (*x*, *y*, *z*) components from a 3-component vector to address `(p + (offset * 4))`. Whether **vloada_half3** and **vstorea_half3** read/write padding data between the third vector element and the next alignment boundary is implementation-defined. The **vloada_** and **vstorea_** variants are provided to access data that is aligned to the size of the vector, and are intended to enable performance on hardware that can take advantage of the increased alignment.

[63] Refer to the description and restrictions for `memory_scope`.

[64] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_int64` feature macro.

[65] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[66] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[67] **async_work_group_copy** and **async_work_group_strided_copy** for 3-component vector types behave as **async_work_group_copy** and **async_work_group_strided_copy** respectively for 4-component vector types.

[68] The C11 consume operation is not supported.

[69] The atomic_long and atomic_ulong types are supported if the cl_khr_int64_base_atomics and cl_khr_int64_extended_atomics extensions are supported and have been enabled. If this is the case then an OpenCL C 3.0 compiler must also define the __opencl_c_int64 feature.

[70] The atomic_double type is only supported if double precision is supported and the cl_khr_int64_base_atomics and cl_khr_int64_extended_atomics extensions are supported and have been enabled. If this is the case then an OpenCL C 3.0 compiler must also define the __opencl_c_fp64 feature.

[71] If the device address space is 64-bits, the data types atomic_intptr_t, atomic_uintptr_t, atomic_size_t and atomic_ptrdiff_t are supported if the cl_khr_int64_base_atomics and cl_khr_int64_extended_atomics extensions are supported and have been enabled.

[72] This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines.

[73] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_int64 feature macro.

[74] Only if the cl_khr_fp16 extension is supported and has been enabled.

[75] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_fp64 feature macro.

[76] Note that **0** is taken as a flag, not as the beginning of a field width.

[77] The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.

[78] Only if the cl_khr_fp16 extension is supported and has been enabled.

[79] When applied to infinite and NaN values, the -, +, and *space* flag characters have their usual meaning; the # and **0** flag characters have no effect.

[80] Binary implementations can choose the hexadecimal digit to the left of the decimal-point character so that subsequent digits align to nibble (4-bit) boundaries.

[81] No special provisions are made for multibyte characters. The behavior of **printf** with the **s** conversion specifier is undefined if the argument value is not a pointer to a literal string.

[82] This is similar to the GL_ADDRESS_CLAMP_TO_BORDER addressing mode.

[83] Note that the built-in function calls to read images with a sampler are not supported for image1d_buffer_t image types.

[84] Although CL_UNORM_INT_101010_2 was added in OpenCL 2.1, because there was no OpenCL C 2.1 this image channel order requires OpenCL 3.0.

[85] Only if the cl_khr_fp16 extension is supported and has been enabled.

[86] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_int64 feature macro.

[87] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_fp64 feature macro.

[88] The half scalar and vector types can only be used if the cl_khr_fp16 extension is supported and has been enabled. The double scalar and vector types can only be used if double precision is supported, e.g. for OpenCL C 3.0 the __opencl_c_fp64 feature macro is present.

[89] The half scalar and vector types can only be used if the cl_khr_fp16 extension is supported and has been enabled. The double scalar and vector types can only be used if double precision is supported, e.g. for OpenCL C 3.0 the __opencl_c_fp64 feature macro is present.

[90] The half scalar and vector types can only be used if the cl_khr_fp16 extension is supported and has been enabled. The double scalar and vector types can only be used if double precision is supported, e.g. for OpenCL C 3.0 the __opencl_c_fp64 feature macro is present.

[91] Implementations are not required to honor this flag. Implementations may not schedule kernel launch earlier than the point specified by this flag, however.

[92] Immediate meaning not side effects resulting from child kernels. The side effects would include stores to global memory and pipe reads and writes.

[93] This acts as a memory synchronization point between work-items in a work-group and child kernels enqueued by work-items in the work-group.

[94] Only if 64-bit integers are supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_int64 feature macro.

[95] Only if the cl_khr_fp16 extension is supported and has been enabled.

[96] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the __opencl_c_fp64 feature macro.

[97] Only if the cl_khr_fp16 extension is supported and has been enabled.

[98] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[99] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[100] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[101] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[102] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[103] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[104] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[105] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[106] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[107] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[108] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[109] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[110] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[111] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[112] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

[113] Only if the `cl_khr_fp16` extension is supported and has been enabled.

[114] Only if double precision is supported. In OpenCL C 3.0 this will be indicated by the presence of the `__opencl_c_fp64` feature macro.

Chapter 7. OpenCL Numerical Compliance

This section describes features of the C99 and IEEE 754 standards that must be supported by all OpenCL compliant devices.

This section describes the functionality that must be supported by all OpenCL devices for single precision floating-point numbers. Currently, only single precision floating-point is a requirement. Double-precision floating-point is an optional feature.

7.1. Rounding Modes

Floating-point calculations may be carried out internally with extra precision and then rounded to fit into the destination type. IEEE 754 defines four possible rounding modes:

- Round to nearest even
- Round toward +∞
- Round toward -∞
- Round toward zero

Round to nearest even is currently the only rounding mode required by the OpenCL specification for single precision and double-precision operations and is therefore the default rounding mode [1]. In addition, only static selection of rounding mode is supported. Dynamically reconfiguring the rounding modes as specified by the IEEE 754 spec is unsupported.

If the `cl_khr_fp16` extension macro is supported, then if `CL_FP_ROUND_TO_NEAREST` is supported, the default rounding mode for half-precision floating-point operations will be round to nearest even; otherwise the default rounding mode will be round to zero.

Conversions to half floating-point format must be correctly rounded using the indicated `convert` operator rounding mode or the default rounding mode for half-precision floating-point operations if no rounding mode is specified by the operator, or a C-style cast is used.

Conversions from half to integer format shall correctly round using the indicated `convert` operator rounding mode, or towards zero if no rounding mode is specified by the operator or a C-style cast is used. All conversions from half to floating-point formats are exact.

If the `cl_khr_select_fprounding_mode` extension macro is supported, the floating-point rounding mode may be specified using the following **#pragma** in the OpenCL program source:

```
#pragma OPENCL SELECT_ROUNDING_MODE <rounding-mode>
```

The <*rounding-mode*> may be one of the following values:

- **rte** - round to nearest even
- **rtz** - round to zero
- **rtp** - round to positive infinity

- **rtn** - round to negative infinity

If this extensions is supported then the OpenCL implementation must support all four rounding modes for single precision floating-point.

The **#pragma** sets the rounding mode for all instructions that operate on floating-point types (scalar or vector types) or produce floating-point values that follow this pragma in the program source until the next **#pragma**. Note that the rounding mode specified for a block of code is known at compile time. When inside a compound statement, the pragma takes effect from its occurrence until another **#pragma** is encountered (including within a nested compound statement), or until the end of the compound statement; at the end of a compound statement the state for the pragma is restored to its condition just before the compound statement. Except where otherwise documented, the callee functions do not inherit the rounding mode of the caller function.

If the `cl_khr_select_fprounding_mode` extension is enabled, the `__ROUNDING_MODE__` preprocessor symbol shall be defined to be one of the following according to the current rounding mode:

```
#define __ROUNDING_MODE__ rte
#define __ROUNDING_MODE__ rtz
#define __ROUNDING_MODE__ rtp
#define __ROUNDING_MODE__ rtz
```

This is intended to enable remapping `foo()` to `foo_rte()` by the preprocessor by using:

```
#define foo foo ## __ROUNDING_MODE__
```

The default rounding mode is round to nearest even. The Math Functions, Common Functions, and Geometric Functions are implemented with the round to nearest even rounding mode. Various built-in conversions and the **vstore_half** and **vstorea_half** built-in functions that do not specify a rounding mode inherit the current rounding mode. Conversions from floating-point to integer type always use `rtz` mode, except where the user specifically asks for another rounding mode.

The `cl_khr_select_fprounding_mode` extension was deprecated in OpenCL 1.1, and its use is not recommended.

7.2. INF, NaN and Denormalized Numbers

`INF` and NaNs must be supported. Support for signaling NaNs is not required.

Support for denormalized numbers with single precision floating-point is optional. Denormalized single precision floating-point numbers passed as input or produced as the output of single precision floating-point operations such as add, sub, mul, divide, and the functions defined in math functions, common functions, and geometric functions may be flushed to zero.

7.3. Floating-Point Exceptions

Floating-point exceptions are disabled in OpenCL. The result of a floating-point exception must match the IEEE 754 spec for the exceptions not enabled case. Whether and when the implementation sets floating-point flags or raises floating-point exceptions is implementation-defined. This standard provides no method for querying, clearing or setting floating-point flags or trapping raised exceptions. Due to non-performance, non-portability of trap mechanisms and the impracticality of servicing precise exceptions in a vector context (especially on heterogeneous hardware), such features are discouraged.

Implementations that nevertheless support such operations through an extension to the standard shall initialize with all exception flags cleared and the exception masks set so that exceptions raised by arithmetic operations do not trigger a trap to be taken. If the underlying work is reused by the implementation, the implementation is however not responsible for reclearing the flags or resetting exception masks to default values before entering the kernel. That is to say that kernels that do not inspect flags or enable traps are licensed to expect that their arithmetic will not trigger a trap. Those kernels that do examine flags or enable traps are responsible for clearing flag state and disabling all traps before returning control to the implementation. Whether or when the underlying work-item (and accompanying global floating-point state if any) is reused is implementation-defined.

The expressions **math_errorhandling** and `MATH_ERREXCEPT` are reserved for use by this standard, but not defined. Implementations that extend this specification with support for floating-point exceptions shall define **math_errorhandling** and `MATH_ERREXCEPT` per TC2 to the C99 Specification.

7.4. Relative Error as ULPs

In this section we discuss the maximum relative error defined as ulp (units in the last place). Addition, subtraction, multiplication, fused multiply-add and conversion between integer and a single precision floating-point format are IEEE 754 compliant and are therefore correctly rounded. Conversion between floating-point formats and explicit conversions must be correctly rounded.

If the `cl_khr_fp16` extension macro is supported, addition, subtraction, multiplication, fused multiply-add operations on half types are required to be correctly rounded using the default rounding mode for half-precision floating-point operations.

The ULP is defined as follows:

> If x is a real number that lies between two finite consecutive floating-point numbers a and b, without being equal to one of them, then ulp(x) = $|b - a|$, otherwise ulp(x) is the distance between the two non-equal finite floating-point numbers nearest x. Moreover, ulp(NaN) is NaN.

Attribution: This definition was taken with consent from Jean-Michel Muller with slight clarification for behavior at zero.

Jean-Michel Muller. On the definition of ulp(x). RR-5504, INRIA. 2005, pp.16. <inria-00070503> Currently hosted at https://hal.inria.fr/inria-00070503/document.

The following table describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result. 0 ulp is used for math functions that do not require rounding.

Result overflow within the specified ULP error is permitted. Math functions are allowed to return infinity for a finite reference value when the next floating-point number that would be representable after the finite maximum, if there was sufficient range, meets ULP error tolerance.

Table 65. ULP Values for Single-Precision Built-in Math Functions

Function	**Min Accuracy - ULP values**
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	≤ 2.5 ulp
x / *y*	≤ 2.5 ulp
acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp
asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 2 ulp
ceil	Correctly rounded
clamp	0 ulp
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp

Function	Min Accuracy - ULP values
cross	absolute error tolerance of 'max * max * (3 * FLT_EPSILON)' per vector component, where *max* is the maximum input operand magnitude
degrees	≤ 2 ulp
distance	≤ 2.5 + 2n ulp, for gentype with vector width *n*
dot	absolute error tolerance of 'max * max * (2n - 1) * FLT_EPSILON', for vector width *n* and maximum input operand magnitude *max* across all vector components
erfc	≤ 16 ulp
erf	≤ 16 ulp
exp	≤ 3 ulp
exp2	≤ 3 ulp
exp10	≤ 3 ulp
expm1	≤ 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	≤ 4 ulp
ilogb	0 ulp
length	≤ 2.75 + 0.5n ulp, for gentype with vector width *n*
ldexp	Correctly rounded
lgamma	Undefined
lgamma_r	Undefined
log	≤ 3 ulp
log2	≤ 3 ulp
log10	≤ 3 ulp
log1p	≤ 2 ulp
logb	0 ulp

Function	Min Accuracy - ULP values
mad	Implemented either as a correctly rounded fma or as a multiply followed by an add both of which are correctly rounded
max	0 ulp
maxmag	0 ulp
min	0 ulp
minmag	0 ulp
mix	absolute error tolerance of 1e-3
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
normalize	≤ 2 + n ulp, for gentype with vector width *n*
pow(*x*, *y*)	≤ 16 ulp
pown(*x*, *y*)	≤ 16 ulp
powr(*x*, *y*)	≤ 16 ulp
radians	≤ 2 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 2 ulp
sign	0 ulp
sin	≤ 4 ulp
sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp
sinpi	≤ 4 ulp
smoothstep	absolute error tolerance of 1e-5
sqrt	≤ 3 ulp
step	0 ulp
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp

Function	Min Accuracy - ULP values
trunc	Correctly rounded
half_cos	≤ 8192 ulp
half_divide	≤ 8192 ulp
half_exp	≤ 8192 ulp
half_exp2	≤ 8192 ulp
half_exp10	≤ 8192 ulp
half_log	≤ 8192 ulp
half_log2	≤ 8192 ulp
half_log10	≤ 8192 ulp
half_powr	≤ 8192 ulp
half_recip	≤ 8192 ulp
half_rsqrt	≤ 8192 ulp
half_sin	≤ 8192 ulp
half_sqrt	≤ 8192 ulp
half_tan	≤ 8192 ulp
fast_distance	≤ 8191.5 + 2n ulp, for gentype with vector width *n*
fast_length	≤ 8191.5 + n ulp, for gentype with vector width *n*
fast_normalize	≤ 8192 + n ulp, for gentype with vector width *n*
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined
native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

The following table describes the minimum accuracy of single precision floating-point arithmetic operations given as ULP values for the embedded profile. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result. 0 ulp is used for math functions that do not require rounding.

Table 66. ULP Values for the Embedded Profile

Function	**Min Accuracy - ULP values**
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	≤ 3 ulp
x / *y*	≤ 3 ulp
acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp
asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 4 ulp
ceil	Correctly rounded
clamp	0 ulp
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp
cross	Implementation-defined
degrees	≤ 2 ulp
distance	Implementation-defined
dot	Implementation-defined
erfc	≤ 16 ulp
erf	≤ 16 ulp

Function	Min Accuracy - ULP values
exp	≤ 4 ulp
exp2	≤ 4 ulp
exp10	≤ 4 ulp
expm1	≤ 4 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp
hypot	≤ 4 ulp
ilogb	0 ulp
ldexp	Correctly rounded
length	Implementation-defined
log	≤ 4 ulp
log2	≤ 4 ulp
log10	≤ 4 ulp
log1p	≤ 4 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
max	0 ulp
maxmag	0 ulp
min	0 ulp
minmag	0 ulp
mix	Implementation-defined
modf	0 ulp
nan	0 ulp
normalize	Implementation-defined
nextafter	0 ulp
pow(x, y)	≤ 16 ulp

Function	Min Accuracy - ULP values
pown(*x*, *y*)	≤ 16 ulp
powr(*x*, *y*)	≤ 16 ulp
radians	≤ 2 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 4 ulp
sign	0 ulp
sin	≤ 4 ulp
sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp
sinpi	≤ 4 ulp
smoothstep	Implementation-defined
sqrt	≤ 4 ulp
step	0 ulp
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp
trunc	Correctly rounded
half_cos	≤ 8192 ulp
half_divide	≤ 8192 ulp
half_exp	≤ 8192 ulp
half_exp2	≤ 8192 ulp
half_exp10	≤ 8192 ulp
half_log	≤ 8192 ulp
half_log2	≤ 8192 ulp
half_log10	≤ 8192 ulp
half_powr	≤ 8192 ulp
half_recip	≤ 8192 ulp
half_rsqrt	≤ 8192 ulp

Function	Min Accuracy - ULP values
half_sin	≤ 8192 ulp
half_sqrt	≤ 8192 ulp
half_tan	≤ 8192 ulp
fast_distance	Implementation-defined
fast_length	Implementation-defined
fast_normalize	Implementation-defined
native_cos	Implementation-defined
native_divide	Implementation-defined
native_exp	Implementation-defined
native_exp2	Implementation-defined
native_exp10	Implementation-defined
native_log	Implementation-defined
native_log2	Implementation-defined
native_log10	Implementation-defined
native_powr	Implementation-defined
native_recip	Implementation-defined
native_rsqrt	Implementation-defined
native_sin	Implementation-defined
native_sqrt	Implementation-defined
native_tan	Implementation-defined

The following table describes the minimum accuracy of commonly used single precision floating-point arithmetic operations given as ULP values if the `-cl-unsafe-math-optimizations` compiler option is specified when compiling or building an OpenCL program. For derived implementations, the operations used in the derivation may themselves be relaxed according to the following table. The minimum accuracy of math functions not defined in the following table when the `-cl-unsafe-math-optimizations` compiler option is specified is as defined in ULP values for single precision built-in math functions when operating in the full profile, and as defined in ULP values for the embedded profile when operating in the embedded profile. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result. 0 ulp is used for math functions that do not require rounding.

Defined minimum accuracy of single precision floating-point arithmetic operations and builtins with `-cl-unsafe-math-optimizations` requires support for OpenCL C 2.0 or newer.

Table 67. ULP Values for Single-Precision Built-in Math Functions With Unsafe Math Optimizations in the Full and Embedded Profiles

Function	Minimum Accuracy
1.0 / *x*	≤ 2.5 ulp for *x* in the domain of 2^{-126} to 2^{126} for the full profile, and ≤ 3 ulp for *x* in the domain of 2^{-126} to 2^{126} for the embedded profile.
x / *y*	≤ 2.5 ulp for *x* in the domain of 2^{-62} to 2^{62} and *y* in the domain of 2^{-62} to 2^{62} for the full profile, and ≤ 3 ulp for *x* in the domain of 2^{-62} to 2^{62} and *y* in the domain of 2^{-62} to 2^{62} for the embedded profile.
acos(*x*)	≤ 4096 ulp
acosh(*x*)	Derived implementations may implement as **log**(*x* + **sqrt**(*x* * *x* - 1)). For non-derived implementations, the error is ≤ 8192 ulp.
acospi(*x*)	Derived implementations may implement as **acos**(*x*) * `M_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
asin(*x*)	≤ 4096 ulp
asinh(*x*)	Derived implementations may implement as **log**(*x* + **sqrt**(*x* * *x* + 1)). For non-derived implementations, the error is ≤ 8192 ulp.
asinpi(*x*)	Derived implementations may implement as **asin**(*x*) * `M_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
atan(*x*)	≤ 4096 ulp
atanh(*x*)	Defined for *x* in the domain (-1, 1). For *x* in [-2^{-10}, 2^{-10}], derived implementations may implement as *x*. For *x* outside of [-2^{-10}, 2^{-10}], derived implementations may implement as 0.5f * **log**((1.0f + *x*) / (1.0f - *x*)). For non-derived implementations, the error is ≤ 8192 ulp.
atanpi(*x*)	Derived implementations may implement as **atan**(*x*) * `M_1_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
atan2(*y*, *x*)	Derived implementations may implement as **atan**(*y* / *x*) for *x* > 0, **atan**(*y* / *x*) + `M_PI_F` for *x* < 0 and *y* > 0, and **atan**(*y* / *x*) - `M_PI_F` for *x* < 0 and *y* < 0. For non-derived implementations, the error is ≤ 8192 ulp.
atan2pi(*y*, *x*)	Derived implementations may implement as **atan2**(*y*, *x*) * `M_1_PI_F`. For non-derived implementations, the error is ≤ 8192 ulp.
cbrt(*x*)	Derived implementations may implement as **rootn**(*x*, 3). For non-derived implementations, the error is ≤ 8192 ulp.
cos(*x*)	For *x* in the domain [-π, π], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
cosh(*x*)	Defined for *x* in the domain [-88, 88]. Derived implementations may implement as 0.5f * (**exp**(*x*) + **exp**(-*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
cospi(*x*)	For *x* in the domain [-1, 1], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
exp(*x*)	≤ 3 + **floor**(**fabs**(2 * *x*)) ulp for the full profile, and ≤ 4 + **floor**(**fabs**(2 * *x*)) ulp for the embedded profile.

Function	Minimum Accuracy
exp2(*x*)	≤ 3 + **floor**(**fabs**(2 * *x*)) ulp for the full profile, and ≤ 4 + **floor**(**fabs**(2 * *x*)) ulp for the embedded profile.
exp10(*x*)	Derived implementations may implement as **exp2**(*x* * **log2**(10)). For non-derived implementations, the error is ≤ 8192 ulp.
expm1(*x*)	Derived implementations may implement as **exp**(*x*) - 1. For non-derived implementations, the error is ≤ 8192 ulp.
log(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤ 3 ulp for the full profile and ≤ 4 ulp for the embedded profile.
log2(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤ 3 ulp for the full profile and ≤ 4 ulp for the embedded profile.
log10(*x*)	For *x* in the domain [0.5, 2] the maximum absolute error is ≤ 2^{-21}; otherwise the maximum error is ≤ 3 ulp for the full profile and ≤ 4 ulp for the embedded profile.
log1p(*x*)	Derived implementations may implement as **log**(*x* + 1). For non-derived implementations, the error is ≤ 8192 ulp.
pow(*x*, *y*)	Undefined for *x* = 0 and *y* = 0. Undefined for *x* < 0 and non-integer *y*. Undefined for *x* < 0 and *y* outside the domain [-2^{24}, 2^{24}]. For *x* > 0 or *x* < 0 and even *y*, derived implementations may implement as **exp2**(*y* * **log2**(**fabs**(*x*))). For *x* < 0 and odd *y*, derived implementations may implement as -**exp2**(*y* * **log2**(**fabs**(*x*)). For *x* == 0 and non-zero *y*, for derived implementations may return zero. For non-derived implementations, the error is ≤ 8192 ulp. [2]
pown(*x*, *y*)	Defined only for integer values of *y*. Undefined for *x* = 0 and *y* = 0. For *x* >= 0 or *x* < 0 and even *y*, derived implementations may implement as **exp2**(*y* * **log2**(**fabs**(*x*))). For *x* < 0 and odd *y*, derived implementations may implement as -**exp2**(*y* * **log2**(**fabs**(*x*))). For non-derived implementations, the error is ≤ 8192 ulp.
powr(*x*, *y*)	Defined only for *x* >= 0. Undefined for *x* = 0 and *y* = 0. Derived implementations may implement as **exp2**(*y* * **log2**(*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
rootn(*x*, *y*)	Defined for *x* > 0 when *y* is non-zero, derived implementations may implement this case as **exp2**(**log2**(*x*) / *y*). Defined for *x* < 0 when *y* is odd, derived implementations may implement this case as -**exp2**(**log2**(-*x*) / *y*). Defined for *x* = +/-0 when *y* > 0, derived implementations may return +0 in this case. For non-derived implementations, the error is ≤ 8192 ulp.
sin(*x*)	For *x* in the domain [-π, π], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
sincos(*x*)	ulp values as defined for **sin**(*x*) and **cos**(*x*).

Function	Minimum Accuracy
sinh(*x*)	Defined for *x* in the domain [-88,88]. For *x* in [-2^{-10}, 2^{-10}], derived implementations may implement as *x*. For *x* outside of [-2^{-10}, 2^{-10}], derived implementations may implement as 0.5f * (**exp**(*x*) - **exp**(-*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
sinpi(*x*)	For *x* in the domain [-1, 1], the maximum absolute error is ≤ 2^{-11} and larger otherwise.
tan(*x*)	Derived implementations may implement as **sin**(*x*) * (1.0f / **cos**(*x*)). For non-derived implementations, the error is ≤ 8192 ulp.
tanh(*x*)	Defined for *x* in the domain [-∞, ∞]. For *x* in [-2^{-10}, 2^{-10}], derived implementations may implement as *x*. For *x* outside of [-2^{-10}, 2^{-10}], derived implementations may implement as (**exp**(*x*) - **exp**(-*x*)) / (**exp**(*x*) + **exp**(-*x*)). For non-derived implementations, the error is ≤ 8192 ULP.
tanpi(*x*)	Derived implementations may implement as **tan**(*x* * `M_PI_F`). For non-derived implementations, the error is ≤ 8192 ulp for *x* in the domain [-1, 1].
x * *y* + *z*	Implemented either as a correctly rounded **fma** or as a multiply and an add both of which are correctly rounded.

The following table describes the minimum accuracy of double-precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result. 0 ulp is used for math functions that do not require rounding.

Table 68. ULP Values for Double-Precision Built-in Math Functions

Function	Min Accuracy - ULP values
x + *y*	Correctly rounded
x - *y*	Correctly rounded
x * *y*	Correctly rounded
1.0 / *x*	Correctly rounded
x / *y*	Correctly rounded
acos	≤ 4 ulp
acospi	≤ 5 ulp
asin	≤ 4 ulp
asinpi	≤ 5 ulp
atan	≤ 5 ulp
atan2	≤ 6 ulp
atanpi	≤ 5 ulp

Function	Min Accuracy - ULP values
atan2pi	≤ 6 ulp
acosh	≤ 4 ulp
asinh	≤ 4 ulp
atanh	≤ 5 ulp
cbrt	≤ 2 ulp
ceil	Correctly rounded
clamp	0 ulp
copysign	0 ulp
cos	≤ 4 ulp
cosh	≤ 4 ulp
cospi	≤ 4 ulp
cross	absolute error tolerance of 'max * max * (3 * FLT_EPSILON)' per vector component, where *max* is the maximum input operand magnitude
degrees	≤ 2 ulp
distance	≤ 5.5 + 2n ulp, for gentype with vector width *n*
dot	absolute error tolerance of 'max * max * (2n - 1) * FLT_EPSILON', for vector width *n* and maximum input operand magnitude *max* across all vector components
erfc	≤ 16 ulp
erf	≤ 16 ulp
exp	≤ 3 ulp
exp2	≤ 3 ulp
exp10	≤ 3 ulp
expm1	≤ 3 ulp
fabs	0 ulp
fdim	Correctly rounded
floor	Correctly rounded
fma	Correctly rounded
fmax	0 ulp
fmin	0 ulp
fmod	0 ulp
fract	Correctly rounded
frexp	0 ulp

Function	Min Accuracy - ULP values
hypot	≤ 4 ulp
ilogb	0 ulp
length	≤ 5.5 + n ulp, for gentype with vector width *n*
ldexp	Correctly rounded
log	≤ 3 ulp
log2	≤ 3 ulp
log10	≤ 3 ulp
log1p	≤ 2 ulp
logb	0 ulp
mad	Any value allowed (infinite ulp)
max	0 ulp
maxmag	0 ulp
min	0 ulp
minmag	0 ulp
mix	Implementation-defined
modf	0 ulp
nan	0 ulp
nextafter	0 ulp
normalize	≤ 4.5 + n ulp, for gentype with vector width *n*
pow(*x*, *y*)	≤ 16 ulp
pown(*x*, *y*)	≤ 16 ulp
powr(*x*, *y*)	≤ 16 ulp
radians	≤ 2 ulp
remainder	0 ulp
remquo	0 ulp
rint	Correctly rounded
rootn	≤ 16 ulp
round	Correctly rounded
rsqrt	≤ 2 ulp
sign	0 ulp
sin	≤ 4 ulp
sincos	≤ 4 ulp for sine and cosine values
sinh	≤ 4 ulp

Function	Min Accuracy - ULP values
sinpi	≤ 4 ulp
smoothstep	Implementation-defined
step	0 ulp
sqrt	Correctly rounded
tan	≤ 5 ulp
tanh	≤ 5 ulp
tanpi	≤ 6 ulp
tgamma	≤ 16 ulp
trunc	Correctly rounded

If the `cl_khr_fp16` extension macro is supported, the following table describes the minimum accuracy of half-precision floating-point arithmetic operations given as ULP values. The reference value used to compute the ULP value of an arithmetic operation is the infinitely precise result. 0 ulp is used for math functions that do not require rounding.

Table 69. ULP Values for Half-Precision Floating-Point Arithmetic Operations

Function	Min Accuracy - Full Profile	Min Accuracy - Embedded Profile
x* + *y	Correctly rounded	Correctly rounded
x* - *y	Correctly rounded	Correctly rounded
x* * *y	Correctly rounded	Correctly rounded
1.0 / *x*	<= 1 ulp	<= 1 ulp
x* / *y	<= 1 ulp	<= 1 ulp
acos	<= 2 ulp	<= 3 ulp
acosh	<= 2 ulp	<= 3 ulp
acospi	<= 2 ulp	<= 3 ulp
asin	<= 2 ulp	<= 3 ulp
asinh	<= 2 ulp	<= 3 ulp
asinpi	<= 2 ulp	<= 3 ulp
atan	<= 2 ulp	<= 3 ulp
atanh	<= 2 ulp	<= 3 ulp
atanpi	<= 2 ulp	<= 3 ulp
atan2	<= 2 ulp	<= 3 ulp
atan2pi	<= 2 ulp	<= 3 ulp
cbrt	<= 2 ulp	<= 2 ulp

Function	Min Accuracy - Full Profile	Min Accuracy - Embedded Profile
ceil	Correctly rounded	Correctly rounded
clamp	0 ulp	0 ulp
copysign	0 ulp	0 ulp
cos	<= 2 ulp	<= 2 ulp
cosh	<= 2 ulp	<= 3 ulp
cospi	<= 2 ulp	<= 2 ulp
cross	absolute error tolerance of 'max * max * (3 * HALF_EPSILON)' per vector component, where *max* is the maximum input operand magnitude	Implementation-defined
degrees	<= 2 ulp	<= 2 ulp
distance	<= 2n ulp, for gentype with vector width *n*	Implementation-defined
dot	absolute error tolerance of 'max * max * (2n - 1) * HALF_EPSILON', for vector width *n* and maximum input operand magnitude *max* across all vector components	Implementation-defined
erfc	<= 4 ulp	<= 4 ulp
erf	<= 4 ulp	<= 4 ulp
exp	<= 2 ulp	<= 3 ulp
exp2	<= 2 ulp	<= 3 ulp
exp10	<= 2 ulp	<= 3 ulp
expm1	<= 2 ulp	<= 3 ulp
fabs	0 ulp	0 ulp
fdim	Correctly rounded	Correctly rounded
floor	Correctly rounded	Correctly rounded
fma	Correctly rounded	Correctly rounded
fmax	0 ulp	0 ulp
fmin	0 ulp	0 ulp
fmod	0 ulp	0 ulp
fract	Correctly rounded	Correctly rounded
frexp	0 ulp	0 ulp

Function	Min Accuracy - Full Profile	Min Accuracy - Embedded Profile
hypot	<= 2 ulp	<= 3 ulp
ilogb	0 ulp	0 ulp
ldexp	Correctly rounded	Correctly rounded
length	<= 0.25 + 0.5n ulp, for gentype with vector width *n*	Implementation-defined
log	<= 2 ulp	<= 3 ulp
log2	<= 2 ulp	<= 3 ulp
log10	<= 2 ulp	<= 3 ulp
log1p	<= 2 ulp	<= 3 ulp
logb	0 ulp	0 ulp
mad	Implementation-defined	Implementation-defined
max	0 ulp	0 ulp
maxmag	0 ulp	0 ulp
min	0 ulp	0 ulp
minmag	0 ulp	0 ulp
mix	Implementation-defined	Implementation-defined
modf	0 ulp	0 ulp
nan	0 ulp	0 ulp
nextafter	0 ulp	0 ulp
normalize	<= 1 + n ulp, for gentype with vector width *n*	Implementation-defined
pow(x, y)	<= 4 ulp	<= 5 ulp
pown(x, y)	<= 4 ulp	<= 5 ulp
powr(x, y)	<= 4 ulp	<= 5 ulp
radians	<= 2 ulp	<= 2 ulp
remainder	0 ulp	0 ulp
remquo	0 ulp for the remainder, at least the lower 7 bits of the integral quotient	0 ulp for the remainder, at least the lower 7 bits of the integral quotient
rint	Correctly rounded	Correctly rounded
rootn	<= 4 ulp	<= 5 ulp
round	Correctly rounded	Correctly rounded
rsqrt	<=1 ulp	<=1 ulp
sign	0 ulp	0 ulp

Function	Min Accuracy - Full Profile	Min Accuracy - Embedded Profile
sin	<= 2 ulp	<= 2 ulp
sincos	<= 2 ulp for sine and cosine values	<= 2 ulp for sine and cosine values
sinh	<= 2 ulp	<= 3 ulp
sinpi	<= 2 ulp	<= 2 ulp
smoothstep	Implementation-defined	Implementation-defined
sqrt	<= 1.5 ulp	<= 1.5 ulp
step	0 ulp	0 ulp
tan	<= 2.5 ulp	<= 3 ulp
tanh	<= 2 ulp	<= 3 ulp
tanpi	<= 2 ulp	<= 3 ulp
tgamma	<= 4 ulp	<= 4 ulp
trunc	Correctly rounded	Correctly rounded

Implementations may perform floating-point operations on `half` *scalar or vector data types by converting the* `half` *values to single precision floating-point values and performing the operation in single precision floating-point. In this case, the implementation will use the* `half` *scalar or vector data type as a storage only format.*

7.5. Edge Case Behavior

The edge case behavior of the math functions shall conform to sections F.9 and G.6 of the C99 Specification, except where noted below.

7.5.1. Additional Requirements Beyond C99 TC2

All functions that return a NaN should return a quiet NaN.

half_<funcname> functions behave identically to the function of the same name without the **half_** prefix. They must conform to the same edge case requirements (see sections F.9 and G.6 of the C99 Specification). For other cases, except where otherwise noted, these single precision functions are permitted to have up to 8192 ulps of error (as measured in the single precision result), although better accuracy is encouraged.

The usual allowances for rounding error or flushing behavior shall not apply for those values for which section F.9 of the C99 Specification, or the additional requirements and edge case behavior below (and similar sections for other floating-point precisions) prescribe a result (e.g. **ceil**(-1 < *x* < 0) returns -0). Those values shall produce exactly the prescribed answers, and no other. Where the ± symbol is used, the sign shall be preserved. For example, **sin**(±0) = ±0 shall be interpreted to mean **sin**(+0) is +0 and **sin**(-0) is -0.

acospi(1) = +0.

acospi(*x*) returns a NaN for $|x| > 1$.

asinpi(±0) = ±0.

asinpi(*x*) returns a NaN for $|x| > 1$.

atanpi(±0) = ±0.

atanpi(±∞) = ±0.5.

atan2pi(±0, -0) = ±1.

atan2pi(±0, +0) = ±0.

atan2pi(±0, *x*) returns ±1 for $x < 0$.

atan2pi(±0, *x*) returns ±0 for $x > 0$.

atan2pi(*y*, ±0) returns -0.5 for $y < 0$.

atan2pi(*y*, ±0) returns 0.5 for $y > 0$.

atan2pi(±_y_, -∞) returns ±1 for finite $y > 0$.

atan2pi(±_y_, +∞) returns ±0 for finite $y > 0$.

atan2pi(±∞, *x*) returns ±0.5 for finite *x*.

atan2pi(±∞, -∞) returns ±0.75.

atan2pi(±∞, +∞) returns ±0.25.

ceil(-1 < *x* < 0) returns -0.

cospi(±0) returns 1

cospi(*n* + 0.5) is +0 for any integer *n* where *n* + 0.5 is representable.

cospi(±∞) returns a NaN.

exp10(-∞) returns +0.

exp10(+∞) returns +∞.

distance(*x*, *y*) calculates the distance from *x* to *y* without overflow or extraordinary precision loss due to underflow.

fdim(any, NaN) returns NaN.

fdim(NaN, any) returns NaN.

fmod(±0, NaN) returns NaN.

frexp(±∞, *exp*) returns ±∞ and stores 0 in *exp*.

frexp(NaN, *exp*) returns the NaN and stores 0 in *exp*.

fract(*x*, *iptr*) shall not return a value greater than or equal to 1.0, and shall not return a value less than 0.

fract(+0, *iptr*) returns +0 and +0 in iptr.

fract(-0, *iptr*) returns -0 and -0 in iptr.

fract(+∞, *iptr*) returns +0 and +∞ in *iptr*.

fract(-∞, *iptr*) returns -0 and -∞ in *iptr*.

fract(NaN, *iptr*) returns the NaN and NaN in *iptr*.

length calculates the length of a vector without overflow or extraordinary precision loss due to underflow.

lgamma_r(*x*, *signp*) returns 0 in *signp* if *x* is zero or a negative integer.

nextafter(-0, *y* > 0) returns smallest positive denormal value.

nextafter(+0, *y* < 0) returns smallest negative denormal value.

normalize shall reduce the vector to unit length, pointing in the same direction without overflow or extraordinary precision loss due to underflow.

normalize(*v*) returns *v* if all elements of *v* are zero.

normalize(*v*) returns a vector full of NaNs if any element is a NaN.

normalize(*v*) for which any element in *v* is infinite shall proceed as if the elements in *v* were replaced as follows:

```
for (i = 0; i < sizeof(v) / sizeof(v[0]); i++)
   v[i] = isinf(v[i]) ? copysign(1.0, v[i]) : 0.0 * v[i];
```

pow(±0, -∞) returns +∞

pown(*x*, 0) is 1 for any *x*, even zero, NaN or infinity.

pown(±0, *n*) is ±∞ for odd *n* < 0.

pown(±0, *n*) is +∞ for even *n* < 0.

pown(±0, *n*) is +0 for even *n* > 0.

pown(±0, *n*) is ±0 for odd *n* > 0.

powr(*x*, ±0) is 1 for finite *x* > 0.

powr(±0, *y*) is +∞ for finite *y* < 0.

powr(±0, -∞) is +∞.

powr(±0, *y*) is +0 for *y* > 0.

powr(+1, *y*) is 1 for finite *y*.

powr(*x*, *y*) returns NaN for *x* < 0.

powr(±0, ±0) returns NaN.

powr(+∞, ±0) returns NaN.

powr(+1, ±∞) returns NaN.

powr(*x*, NaN) returns the NaN for *x* >= 0.

powr(NaN, *y*) returns the NaN.

rint(-0.5 <= *x* < 0) returns -0.

remquo(*x*, *y*, &_quo_) returns a NaN and 0 in *quo* if *x* is ±∞, or if *y* is 0 and the other argument is non-NaN or if either argument is a NaN.

rootn(±0, *n*) is ±∞ for odd *n* < 0.

rootn(±0, *n*) is +∞ for even *n* < 0.

rootn(±0, *n*) is +0 for even *n* > 0.

rootn(±0, *n*) is ±0 for odd *n* > 0.

rootn(*x*, *n*) returns a NaN for *x* < 0 and *n* is even.

rootn(*x*, 0) returns a NaN.

round(-0.5 < *x* < 0) returns -0.

sinpi(±0) returns ±0.

sinpi(+*n*) returns +0 for positive integers *n*.

sinpi(-*n*) returns -0 for negative integers *n*.

sinpi(±∞) returns a NaN.

tanpi(±0) returns ±0.

tanpi(±∞) returns a NaN.

tanpi(*n*) is **copysign**(0.0, *n*) for even integers *n*.

tanpi(*n*) is **copysign**(0.0, - *n*) for odd integers *n*.

tanpi(*n* + 0.5) for even integer *n* is +∞ where *n* + 0.5 is representable.

tanpi(*n* + 0.5) for odd integer *n* is -∞ where *n* + 0.5 is representable.

trunc(-1 < *x* < 0) returns -0. Binary file (standard input) matches

7.5.2. Changes to C99 TC2 Behavior

modf behaves as though implemented by:

```
gentype modf(gentype value, gentype *iptr)
{
    *iptr = trunc( value );
    return copysign(isinf( value ) ? 0.0 : value - *iptr, value);
}
```

rint always rounds according to round to nearest even rounding mode even if the caller is in some other rounding mode.

7.5.3. Edge Case Behavior in Flush to Zero Mode

If denormals are flushed to zero, then a function may return one of four results:

1. Any conforming result for non-flush-to-zero mode
2. If the result given by 1. is a sub-normal before rounding, it may be flushed to zero
3. Any non-flushed conforming result for the function if one or more of its sub-normal operands are flushed to zero.

4. If the result of 3. is a sub-normal before rounding, the result may be flushed to zero.

In each of the above cases, if an operand or result is flushed to zero, the sign of the zero is undefined.

If subnormals are flushed to zero, a device may choose to conform to the following edge cases for **nextafter** instead of those listed in the additional requirements section.

nextafter(+smallest normal, *y* < +smallest normal) = +0.

nextafter(-smallest normal, *y* > -smallest normal) = -0.

nextafter(-0, *y* > 0) returns smallest positive normal value.

nextafter(+0, *y* < 0) returns smallest negative normal value.

For clarity, subnormals or denormals are defined to be the set of representable numbers in the range 0 < *x* < `TYPE_MIN` and `-TYPE_MIN` < *x* < -0. They do not include ±0. A non-zero number is said to be sub-normal before rounding if after normalization, its radix-2 exponent is less than (`TYPE_MIN_EXP` - 1) [3].

[1] Except for the embedded profile where either round to zero or round to nearest rounding mode may be supported for single precision floating-point.

[2] On some implementations, **powr**() or **pown**() may perform faster than **pow**(). If *x* is known to be >= 0, consider using **powr**() in place of **pow**(), or if *y* is known to be an integer, consider using **pown**() in place of **pow**().

[3] Here `TYPE_MIN` and `TYPE_MIN_EXP` should be substituted by constants appropriate to the floating-point type under consideration, such as `FLT_MIN` and `FLT_MIN_EXP` for `float`.

Chapter 8. Image Addressing and Filtering

Let w_t, h_t and d_t be the width, height (or image array size for a 1D image array) and depth (or image array size for a 2D image array) of the image in pixels. Let *coord.xy* (also referred to as (*s*, *t*)) or *coord.xyz* (also referred to as (*s*, *t*, *r*)) be the coordinates specified to **read_image{f|i|ui}**. The sampler specified in **read_image{f|i|ui}** is used to determine how to sample the image and return an appropriate color.

8.1. Image Coordinates

This affects the interpretation of image coordinates. If image coordinates specified to **read_image{f|i|ui}** are normalized (as specified in the sampler), the *s*, *t*, and *r* coordinate values are multiplied by w_t, h_t, and d_t respectively to generate the unnormalized coordinate values. For image arrays, the image array coordinate (i.e. *t* if it is a 1D image array or *r* if it is a 2D image array) specified to **read_image{f|i|ui}** must always be the un-normalized image coordinate value.

Let (*u*, *v*, *w*) represent the unnormalized image coordinate values.

8.2. Addressing and Filter Modes

We first describe how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is not `CLK_ADDRESS_REPEAT` nor `CLK_ADDRESS_MIRRORED_REPEAT`.

After generating the image coordinate (*u*, *v*, *w*) we apply the appropriate addressing and filter mode to generate the appropriate sample locations to read from the image.

If values in (*u*, *v*, *w*) are `INF` or NaN, the behavior of **read_image{f|i|ui}** is undefined.

Filter Mode `CLK_FILTER_NEAREST`

When filter mode is `CLK_FILTER_NEAREST`, the image element in the image that is nearest (in Manhattan distance) to that specified by (*u*, *v*, *w*) is obtained. This means the image element at location (*i*, *j*, *k*) becomes the image element value, where

```
i = address_mode((int)floor(u))
j = address_mode((int)floor(v))
k = address_mode((int)floor(w))
```

For a 3D image, the image element at location (*i*, *j*, *k*) becomes the color value. For a 2D image, the image element at location (*i*, *j*) becomes the color value.

The following table describes the address_mode function.

Table 70. Addressing modes to generate texel location

Addressing Mode	Result of address_mode(coord)
`CLK_ADDRESS_CLAMP_TO_EDGE`	clamp (coord, 0, size - 1)
`CLK_ADDRESS_CLAMP`	clamp (coord, -1, size)
`CLK_ADDRESS_NONE`	coord

The `size` term in this table is w_t for u, h_t for v and d_t for w.

The `clamp` function used in this table is defined as:

```
clamp(a, b, c) = return (a < b) ? b : ((a > c) ? c : a)
```

If the selected texel location (i, j, k) refers to a location outside the image, the border color is used as the color value for this texel.

Filter Mode `CLK_FILTER_LINEAR`

When filter mode is `CLK_FILTER_LINEAR`, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
i0 = address_mode((int)floor(u - 0.5))
j0 = address_mode((int)floor(v - 0.5))
k0 = address_mode((int)floor(w - 0.5))
i1 = address_mode((int)floor(u - 0.5) + 1)
j1 = address_mode((int)floor(v - 0.5) + 1)
k1 = address_mode((int)floor(w - 0.5) + 1)
a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (i, j, k) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where `T_ij` is the image element at location (*i*, *j*) in the 2D image.

If any of the selected `T_ijk` or `T_ij` in the above equations refers to a location outside the image, the border color is used as the color value for `T_ijk` or `T_ij`.

If the image channel type is `CL_FLOAT` or `CL_HALF_FLOAT` and any of the image elements `T_ijk` or `T_ij` is `INF` or NaN, the behavior of the built-in image read function is undefined.

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is `CLK_ADDRESS_REPEAT`.

If values in (*s*, *t*, *r*) are `INF` or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode `CLK_FILTER_NEAREST`

When filter mode is `CLK_FILTER_NEAREST`, the image element at location (*i*, *j*, *k*) becomes the image element value, with *i*, *j*, and *k* computed as

```
u = (s - floor(s)) * w_t
i = (int)floor(u)
if (i > w_t - 1)
    i = i - w_t

v = (t - floor(t)) * h_t
j = (int)floor(v)
if (j > h_t - 1)
    j = j - h_t

w = (r - floor(r)) * d_t
k = (int)floor(w)
if (k > d_t - 1)
    k = k - d_t
```

For a 3D image, the image element at location (*i*, *j*, *k*) becomes the color value. For a 2D image, the image element at location (*i*, *j*) becomes the color value.

Filter Mode `CLK_FILTER_LINEAR`

When filter mode is `CLK_FILTER_LINEAR`, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
u = (s - floor(s)) * w_t
i0 = (int)floor(u - 0.5)
i1 = i0 + 1
if (i0 < 0)
    i0 = w_t + i0
if (i1 > w_t - 1)
    i1 = i1 - w_t

v = (t - floor(t)) * h_t
j0 = (int)floor(v - 0.5)
j1 = j0 + 1
if (j0 < 0)
    j0 = h_t + j0
if (j1 > h_t - 1)
    j1 = j1 - h_t

w = (r - floor(r)) * d_t
k0 = (int)floor(w - 0.5)
k1 = k0 + 1
if (k0 < 0)
    k0 = d_t + k0
if (k1 > d_t - 1)
    k1 = k1 - d_t

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (*i*, *j*, *k*) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
```

```
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where T_ij is the image element at location (*i*, *j*) in the 2D image.

If the image channel type is CL_FLOAT or CL_HALF_FLOAT and any of the image elements T_ijk or T_ij is INF or NaN, the behavior of the built-in image read function is undefined.

We now discuss how the addressing and filter modes are applied to generate the appropriate sample locations to read from the image if the addressing mode is CLK_ADDRESS_MIRRORED_REPEAT. The CLK_ADDRESS_MIRRORED_REPEAT addressing mode causes the image to be read as if it is tiled at every integer seam with the interpretation of the image data flipped at each integer crossing. For example, the (*s*, *t*, *r*) coordinates between 2 and 3 are addressed into the image as coordinates from 1 down to 0. If values in (*s*, *t*, *r*) are INF or NaN, the behavior of the built-in image read functions is undefined.

Filter Mode CLK_FILTER_NEAREST

When filter mode is CLK_FILTER_NEAREST, the image element at location (*i*, *j*, *k*) becomes the image element value, with *i*, *j* and k computed as

```
s_prime = 2.0f * rint(0.5f * s)
s_prime = fabs(s - s_prime)
u = s_prime * w_t
i = (int)floor(u)
i = min(i, w_t - 1)

t_prime = 2.0f * rint(0.5f * t)
t_prime = fabs(t - t_prime)
v = t_prime * h_t
j = (int)floor(v)
j = min(j, h_t - 1)

r_prime = 2.0f * rint(0.5f * r)
r_prime = fabs(r - r_prime)
w = r_prime * d_t
k = (int)floor(w)
k = min(k, d_t - 1)
```

For a 3D image, the image element at location (*i*, *j*, *k*) becomes the color value. For a 2D image, the image element at location (*i*, *j*) becomes the color value.

Filter Mode CLK_FILTER_LINEAR

When filter mode is CLK_FILTER_LINEAR, a 2×2 square of image elements for a 2D image or a 2×2×2 cube of image elements for a 3D image is selected. This 2×2 square or 2×2×2 cube is obtained as follows.

Let

```
s_prime = 2.0f * rint(0.5f * s)
s_prime = fabs(s - s_prime)
u = s_prime * w_t
i0 = (int)floor(u - 0.5f)
i1 = i0 + 1
i0 = max(i0, 0)
i1 = min(i1, w_t - 1)

t_prime = 2.0f * rint(0.5f * t)
t_prime = fabs(t - t_prime)
v = t_prime * h_t
j0 = (int)floor(v - 0.5f)
j1 = j0 + 1
j0 = max(j0, 0)
j1 = min(j1, h_t - 1)

r_prime = 2.0f * rint(0.5f * r)
r_prime = fabs(r - r_prime)
w = r_prime * d_t
k0 = (int)floor(w - 0.5f)
k1 = k0 + 1
k0 = max(k0, 0)
k1 = min(k1, d_t - 1)

a = frac(u - 0.5)
b = frac(v - 0.5)
c = frac(w - 0.5)
```

where `frac(x)` denotes the fractional part of x and is computed as `x - floor(x)`.

For a 3D image, the image element value is found as

```
T = (1 - a) * (1 - b) * (1 - c) * T_i0j0k0
    + a * (1 - b) * (1 - c) * T_i1j0k0
    + (1 - a) * b * (1 - c) * T_i0j1k0
    + a * b * (1 - c) * T_i1j1k0
    + (1 - a) * (1 - b) * c * T_i0j0k1
    + a * (1 - b) * c * T_i1j0k1
    + (1 - a) * b * c * T_i0j1k1
    + a * b * c * T_i1j1k1
```

where `T_ijk` is the image element at location (*i*, *j*, *k*) in the 3D image.

For a 2D image, the image element value is found as

```
T = (1 - a) * (1 - b) * T_i0j0
```

```
    + a * (1 - b) * T_i1j0
    + (1 - a) * b * T_i0j1
    + a * b * T_i1j1
```

where T_ij is the image element at location (*i*, *j*) in the 2D image.

For a 1D image, the image element value is found as

```
T = (1 - a) * T_i0
    + a * T_i1
```

where T_i is the image element at location (*i*) in the 1D image.

If the image channel type is CL_FLOAT or CL_HALF_FLOAT and any of the image elements T_ijk or T_ij is INF or NaN, the behavior of the built-in image read function is undefined.

If the sampler is specified as using unnormalized coordinates (floating-point or integer coordinates), filter mode set to CLK_FILTER_NEAREST and addressing mode set to one of the following modes - CLK_ADDRESS_NONE, CLK_ADDRESS_CLAMP_TO_EDGE or CLK_ADDRESS_CLAMP, the location of the image element in the image given by (*i*, *j*, *k*) will be computed without any loss of precision.

For all other sampler combinations of normalized or unnormalized coordinates, filter and addressing modes, the relative error or precision of the addressing mode calculations and the image filter operation are not defined by this revision of the OpenCL specification. To ensure a minimum precision of image addressing and filter calculations across any OpenCL device, for these sampler combinations, developers should unnormalize the image coordinate in the kernel and implement the linear filter in the kernel with appropriate calls to **read_image{f|i|ui}** with a sampler that uses unnormalized coordinates, filter mode set to CLK_FILTER_NEAREST, addressing mode set to CLK_ADDRESS_NONE, CLK_ADDRESS_CLAMP_TO_EDGE *or* CLK_ADDRESS_CLAMP, and finally performing the interpolation of color values read from the image to generate the filtered color value.

8.3. Conversion Rules

In this section we discuss conversion rules that are applied when reading and writing images in a kernel.

8.3.1. Conversion Rules for Normalized Integer Channel Data Types

In this section we discuss converting normalized integer channel data types to floating-point values and vice-versa.

8.3.1.1. Converting Normalized Integer Channel Data Types to Floating-point Values

For images created with any of the following image channel data types, **read_imagef** will convert

the channel values from an unsigned integer to normalized floating-point values in the range [0.0f, 1.0f].

- CL_UNORM_INT8
- CL_UNORM_INT16
- CL_UNORM_INT10X6_EXT
- CL_UNORM_INT12X4_EXT
- CL_UNORM_INT14X2_EXT

For images created with any of the following image channel data types, **read_imagef** will convert the channel values from a signed integer to normalized floating-point values in the range [-1.0f, 1.0f].

- CL_SNORM_INT8
- CL_SNORM_INT16

These conversions are performed as follows:

CL_UNORM_INT8 (8-bit unsigned integer) → float

normalized float value = (float)c / 255.0f

CL_UNORM_INT_101010 (10-bit unsigned integer) → float

normalized float value = (float)c / 1023.0f

CL_UNORM_INT16 (16-bit unsigned integer) → float

normalized float value = (float)c / 65535.0f

CL_UNORM_INT10X6_EXT (10-bit unsigned integer) → float

normalized float value = (float)c / 1023.0f

CL_UNORM_INT12X4_EXT (12-bit unsigned integer) → float

normalized float value = (float)c / 4095.0f

CL_UNORM_INT14X2_EXT (14-bit unsigned integer) → float

normalized float value = (float)c / 16383.0f

CL_SNORM_INT8 (8-bit signed integer) → float

normalized float value = **max**(-1.0f, (float)c / 127.0f)

CL_SNORM_INT16 (16-bit signed integer) → float

normalized float value = **max**(-1.0f, (float)c / 32767.0f)

The precision of the above conversions is <= 1.5 ulp except for the following cases:

For `CL_UNORM_INT8`

0 must convert to `0.0f` and

255 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_INT16`

0 must convert to `0.0f` and

65535 must convert to `1.0f`

For `CL_UNORM_INT10X6_EXT`:

0 must convert to `0.0f`, and

1023 must convert to `1.0f`

For `CL_UNORM_INT12X4_EXT`:

0 must convert to `0.0f`, and

4095 must convert to `1.0f`

For `CL_UNORM_INT14X2_EXT`:

0 must convert to `0.0f`, and

16383 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`

8.3.1.2. Converting Normalized Integer Channel Data Types to Half-Precision Floating-Point Values

The conversions described in this section require the `cl_khr_fp16` extension.

For images created with any of the following image channel data types, **read_imageh** will convert

the channel values from an 8-bit or 16-bit unsigned integer to normalized half-precision floating-point values in the range [`0.0h`, `1.0h`].

- `CL_UNORM_INT8`
- `CL_UNORM_INT16`
- `CL_UNORM_INT10X6_EXT`
- `CL_UNORM_INT12X4_EXT`
- `CL_UNORM_INT14X2_EXT`

For images created with any of the following image channel data types, **read_imageh** will convert the channel values from a signed integer to normalized half-precision floating-point values in the range [`-1.0h`, `1.0h`].

- `CL_SNORM_INT8`
- `CL_SNORM_INT16`

These conversions are performed as follows:

`CL_UNORM_INT8` (8-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 255)`

`CL_UNORM_INT_101010` (10-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 1023)`

`CL_UNORM_INT16` (16-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 65535)`

`CL_UNORM_INT10X6_EXT` (10-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 1023)`

`CL_UNORM_INT12X4_EXT` (12-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 4095)`

`CL_UNORM_INT14X2_EXT` (14-bit unsigned integer) → `half`

normalized `half` value = `round_to_half(c / 16383)`

`CL_SNORM_INT8` (8-bit signed integer) → `half`

normalized `half` value = **max**(`-1.0h`, `round_to_half(c / 127)`)

`CL_SNORM_INT16` (16-bit signed integer) → `half`

normalized `half` value = **max**(`-1.0h`, `round_to_half(c / 32767)`)

The precision of the above conversions is <= 1.5 ulp except for the following cases:

For `CL_UNORM_INT8`

0 must convert to `0.0h` and

255 must convert to `1.0h`

For `CL_UNORM_INT_101010`

0 must convert to `0.0h` and

1023 must convert to `1.0h`

For `CL_UNORM_INT16`

0 must convert to `0.0h` and

65535 must convert to `1.0h`

For `CL_UNORM_INT10X6_EXT`:

0 must convert to `0.0h`, and

1023 must convert to `1.0h`

For `CL_UNORM_INT12X4_EXT`:

0 must convert to `0.0h`, and

4095 must convert to `1.0h`

For `CL_UNORM_INT14X2_EXT`:

0 must convert to `0.0h`, and

16383 must convert to `1.0h`

For `CL_SNORM_INT8`

-128 and -127 must convert to `-1.0h`,

0 must convert to `0.0h` and

127 must convert to `1.0h`

For `CL_SNORM_INT16`

-32768 and -32767 must convert to `-1.0h`,

0 must convert to `0.0h` and

32767 must convert to `1.0h`

8.3.1.3. Converting Floating-Point Values to Normalized Integer Channel Data Types

For images created with any of the following image channel data types, **write_imagef** will convert the floating-point color value to an unsigned integer.

- `CL_UNORM_INT8`

- CL_UNORM_INT16
- CL_UNORM_INT10X6_EXT
- CL_UNORM_INT12X4_EXT
- CL_UNORM_INT14X2_EXT

For images created with any of the following image channel data types, **write_imagef** will convert the floating-point color value to a signed integer.

- CL_SNORM_INT8
- CL_SNORM_INT16

The preferred method for how conversions from floating-point values to normalized integer values are performed is as follows:

float → CL_UNORM_INT8 (8-bit unsigned integer)

convert_uchar_sat_rte(f * 255.0f)

float → CL_UNORM_INT_101010 (10-bit unsigned integer)

min(**convert_ushort_sat_rte**(f * 1023.0f), 0x3ff)

float → CL_UNORM_INT16 (16-bit unsigned integer)

convert_ushort_sat_rte(f * 65535.0f)

float → CL_UNORM_INT10X6_EXT (10-bit unsigned integer)

min(**convert_ushort_sat_rte**(f * 1023.0f), 1023)

float → CL_UNORM_INT12X4_EXT (12-bit unsigned integer)

min(**convert_ushort_sat_rte**(f * 4095.0f), 4095)

float → CL_UNORM_INT14X2_EXT (14-bit unsigned integer)

min(**convert_ushort_sat_rte**(f * 16383.0f), 16383)

float → CL_SNORM_INT8 (8-bit signed integer)

convert_char_sat_rte(f * 127.0f)

float → CL_SNORM_INT16 (16-bit signed integer)

convert_short_sat_rte(f * 32767.0f)

Please refer to the out-of-range behavior and saturated conversion rules.

OpenCL implementations may choose to approximate the rounding mode used in the conversions described above. If a rounding mode other than round to nearest even (_rte) is used, the absolute error of the implementation dependant rounding mode vs. the result produced by the round to

nearest even rounding mode must be ≤ 0.6.

`float` → `CL_UNORM_INT8` (8-bit unsigned integer)

Let $f_{preferred}$ = **convert_uchar_sat_rte**(f * `255.0f`)

Let f_{approx} = **convert_uchar_sat_<impl-rounding-mode>**(f * `255.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `1023.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `1023.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT16` (16-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `65535.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `65535.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT10X6_EXT` (10-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `1023.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `1023.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT12X4_EXT` (12-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `4095.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `4095.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_UNORM_INT14X2_EXT` (14-bit unsigned integer)

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * `16383.0f`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * `16383.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_SNORM_INT8` (8-bit signed integer)

Let $f_{preferred}$ = **convert_char_sat_rte**(f * `127.0f`)

Let f_{approx} = **convert_char_sat_<impl_rounding_mode>**(f * `127.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

`float` → `CL_SNORM_INT16` (16-bit signed integer)

Let $f_{preferred}$ = **convert_short_sat_rte**(f * `32767.0f`)

Let f_{approx} = **convert_short_sat_<impl-rounding-mode>**(f * `32767.0f`)

fabs($f_{preferred}$ - f_{approx}) must be <= 0.6

8.3.1.4. Converting Half-Precision Floating-point Values to Normalized Integer Channel Data Types

The conversions described in this section require the `cl_khr_fp16` extension.

For images created with any of the following image channel data types, **write_imageh** will convert the floating-point color value to an unsigned integer.

- `CL_UNORM_INT8`
- `CL_UNORM_INT16`
- `CL_UNORM_INT10X6_EXT`
- `CL_UNORM_INT12X4_EXT`
- `CL_UNORM_INT14X2_EXT`

For images created with any of the following image channel data types, **write_imageh** will convert the floating-point color value to a signed integer.

- `CL_SNORM_INT8`
- `CL_SNORM_INT16`

The preferred conversion uses the round to nearest even (`_rte`) rounding mode, but OpenCL implementations may choose to approximate the rounding mode used in the conversions described below. When approximate rounding is used instead of the preferred rounding, the result of the conversion must satisfy the bound given below.

`half` → `CL_UNORM_INT8` (8-bit unsigned integer)

Let f_{exact} = **max**(`0`, **min**(`f * 255`, `255`))

Let $f_{preferred}$ = **convert_uchar_sat_rte**(`f * 255.0f`)

Let f_{approx} = **convert_uchar_sat_<impl-rounding-mode>**(`f * 255.0f`)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_UNORM_INT_101010` (10-bit unsigned integer)

Let f_{exact} = **max**(`0`, **min**(`f * 1023`, `1023`))

Let $f_{preferred}$ = **min**(**convert_ushort_sat_rte**(`f * 1023.0f`), `1023`)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(`f * 1023.0f`)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_UNORM_INT16` (16-bit unsigned integer)

Let f_{exact} = **max**(0, **min**(f * 65535, 65535))

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * 65535.0f)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * 65535.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_UNORM_INT10X6_EXT` (10-bit unsigned integer)

Let f_{exact} = **max**(0, **min**(f * 1023, 1023))

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * 1023.0f)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * 1023.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_UNORM_INT12X4_EXT` (12-bit unsigned integer)

Let f_{exact} = **max**(0, **min**(f * 4095, 4095))

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * 4095.0f)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * 4095.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_UNORM_INT14X2_EXT` (14-bit unsigned integer)

Let f_{exact} = **max**(0, **min**(f * 16383, 16383))

Let $f_{preferred}$ = **convert_ushort_sat_rte**(f * 16383.0f)

Let f_{approx} = **convert_ushort_sat_<impl-rounding-mode>**(f * 16383.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_SNORM_INT8` (8-bit signed integer)

Let f_{exact} = **max**(-128, **min**(f * 127, 127))

Let $f_{preferred}$ = **convert_char_sat_rte**(f * 127.0f)

Let f_{approx} = **convert_char_sat_<impl_rounding_mode>**(f * 127.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

`half` → `CL_SNORM_INT16` (16-bit signed integer)

Let f_{exact} = **max**(-32768, **min**(f * 32767, 32767))

Let $f_{preferred}$ = **convert_short_sat_rte**(f * 32767.0f)

Let f_{approx} = **convert_short_sat_<impl-rounding-mode>**(f * 32767.0f)

fabs(f_{exact} - f_{approx}) must be <= 0.6

8.3.2. Conversion Rules for Half-Precision Floating-Point Channel Data Type

For images created with a channel data type of `CL_HALF_FLOAT`, the conversions from `half` to `float`

are lossless (as described in "The half data type"). Conversions from `float` to `half` round the mantissa using the round to nearest even or round to zero rounding mode. Denormalized numbers for the `half` data type which may be generated when converting a `float` to a `half` may be flushed to zero. A `float` NaN must be converted to an appropriate NaN in the `half` type. A `float INF` must be converted to an appropriate `INF` in the `half` type.

8.3.3. Conversion Rules for Floating-Point Channel Data Type

The following rules apply for reading and writing images created with channel data type of `CL_FLOAT`.

- NaNs may be converted to a NaN value(s) supported by the device.
- Denorms can be flushed to zero.
- All other values must be preserved.

8.3.4. Conversion Rules for Signed and Unsigned Integer Channel Data Types

Calls to **read_imagei** with any of the following channel data type values return the integer values stored in the image at specified location, after sign extension to 32 bits for integer values smaller than 32 bits.

- `CL_SIGNED_INT8`
- `CL_SIGNED_INT16`
- `CL_SIGNED_INT32`

Calls to **read_imageui** with any of the following channel data type values return the integer values stored in the image at specified location, after zero extension to 32 bits for integer values smaller than 32 bits.

- `CL_UNSIGNED_INT8`
- `CL_UNSIGNED_INT16`
- `CL_UNSIGNED_INT32`
- `CL_UNSIGNED_INT10X6_EXT`
- `CL_UNSIGNED_INT12X4_EXT`
- `CL_UNSIGNED_INT14X2_EXT`

Calls to **write_imagei** will perform one of the following conversions:

32 bit signed integer → `CL_SIGNED_INT8` (8-bit signed integer)

convert_char_sat(i)

32 bit signed integer → `CL_SIGNED_INT16` (16-bit signed integer)

convert_short_sat(i)

32 bit signed integer → `CL_SIGNED_INT32` (32-bit signed integer)

no conversion is performed

Calls to **write_imageui** will perform one of the following conversions:

32 bit unsigned integer → `CL_UNSIGNED_INT8` (8-bit unsigned integer)

convert_uchar_sat(i)

32 bit unsigned integer → `CL_UNSIGNED_INT16` (16-bit unsigned integer)

convert_ushort_sat(i)

32 bit unsigned integer → `CL_UNSIGNED_INT32` (32-bit unsigned integer)

no conversion is performed

32 bit unsigned integer → `CL_UNSIGNED_INT10X6_EXT` (10-bit unsigned integer)

clamp(**convert_ushort_sat**(i), 0, 1023)

32 bit unsigned integer → `CL_UNSIGNED_INT12X4_EXT` (12-bit unsigned integer)

clamp(**convert_ushort_sat**(i), 0, 4095)

32 bit unsigned integer → `CL_UNSIGNED_INT14X2_EXT` (14-bit unsigned integer)

clamp(**convert_ushort_sat**(i), 0, 16383)

The conversions described in this section must be correctly saturated.

8.3.5. Conversion Rules for sRGBA and sBGRA Images

Standard RGB data, which roughly displays colors in a linear ramp of luminosity levels such that an average observer, under average viewing conditions, can view them as perceptually equal steps on an average display. All 0's maps to `0.0f`, and all 1's maps to `1.0f`. The sequence of unsigned integer encodings between all 0's and all 1's represent a nonlinear progression in the floating-point interpretation of the numbers between `0.0f` to `1.0f`. For more detail, see the SRGB color standard.

Conversion from sRGB space is automatically done by **read_imagef** built-in functions if the image channel order is one of the sRGB values described above. When reading from an sRGB image, the conversion from sRGB to linear RGB is performed before the filter specified in the sampler specified to read_imagef is applied. If the format has an alpha channel, the alpha data is stored in linear color space. Conversion to sRGB space is automatically done by **write_imagef** built-in functions if the image channel order is one of the sRGB values described above and the device supports writing to sRGB images.

If the format has an alpha channel, the alpha data is stored in linear color space.

The following is the conversion rule for converting a normalized 8-bit unsigned integer sRGB color value to a floating-point linear RGB color value using **read_imagef**.

```
// Convert the normalized 8-bit unsigned integer R, G and B channel values
// to a floating-point value (call it c) as per rules described in section
// 8.3.1.1.

if (c <= 0.04045),
    result = c / 12.92;
else
    result = powr((c + 0.055) / 1.055, 2.4);
```

The resulting floating-point value, if converted back to an sRGB value without rounding to a 8-bit unsigned integer value, must be within 0.5 ulp of the original sRGB value.

The following are the conversion rules for converting a linear RGB floating-point color value (call it *c*) to a normalized 8-bit unsigned integer sRGB value using **write_imagef**.

```
if (c is NaN)
    c = 0.0;
if (c > 1.0)
    c = 1.0;
else if (c < 0.0)
    c = 0.0;
else if (c < 0.0031308)
    c = 12.92 * c;
else
    c = 1.055 * powr(c, 1.0/2.4) - 0.055;

scaled_reference_result = c * 255
channel_component = floor(scaled_reference_result + 0.5);
```

The precision of the above conversion should be such that

$$|\texttt{generated_channel_component} - \texttt{scaled_reference_result}| \leq 0.6$$

where `generated_channel_component` is the actual value that the implementation produces and being checked for conformance.

8.4. Selecting an Image From an Image Array

Let (*u*, *v*, *w*) represent the unnormalized image coordinate values for reading from and/or writing to a 2D image in a 2D image array.

When read using a sampler, the 2D image layer selected is computed as:

$$\text{layer} = \textbf{clamp}(\textbf{rint}(w), 0, d_t - 1)$$

otherwise the layer selected is computed as:

$$\text{layer} = w$$

(since *w* is already an integer) and the result is undefined if *w* is not one of the integers 0, 1, ... d_t - 1.

Let (*u*, *v*) represent the unnormalized image coordinate values for reading from and/or writing to a 1D image in a 1D image array.

When read using a sampler, the 1D image layer selected is computed as:

layer = **clamp**(**rint**(*v*), 0, h_t - 1)

otherwise the layer selected is computed as:

layer = *v*

(since *v* is already an integer) and the result is undefined if *v* is not one of the integers 0, 1, ... h_t - 1.

Chapter 9. Normative References

1. “ISO/IEC 9899:1999 - Programming languages - C”, with technical corrigenda TC1 and TC2, https://www.iso.org/standard/29237.html . References are to sections of this specific version, referred to as the “C99 Specification”, although other versions exist.
2. “ISO/IEC 9899:2011 - Information technology - Programming languages - C”, https://www.iso.org/standard/57853.html . References are to sections of this specific version, referred to as the “C11 Specification”, although other versions exist.
3. “The OpenCL Specification, Version 3.0, Unified”, https://www.khronos.org/registry/OpenCL/ . References are to sections and tables of this specific version, although other versions exists.
4. “Device Queries” are defined in the OpenCL Specification for **clGetDeviceInfo**, and the individual queries are defined in the “OpenCL Device Queries” table (4.3) of that Specification.
5. “Image Channel Order” is defined in the OpenCL Specification in the “Image Format Descriptor” section (5.3.1.1), and the individual channel orders are defined in the “List of supported Image Channel Order Values” table (5.6) of that Specification.
6. “Image Channel Data Type” is defined in the OpenCL Specification in the “Image Format Descriptor” section (5.3.1.1), and the individual channel data types are defined in the "`List of supported Image Channel Data Types" table (5.7) of that Specification.
7. “IEC 61966-2-1:1999 Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB”, https://webstore.iec.ch/publication/6169 .
8. “ISO/IEC TR 18037:2008 Programming languages - C - Extensions to support embedded processors”, https://www.iso.org/standard/51126.html . References are to sections of this specific version, referred to as the “Embedded C Specification”, although other versions exist.

Appendix A: Changes to OpenCL

Changes to the OpenCL C specifications between successive versions are summarized below.

Summary of Changes to OpenCL 3.0

The first non-experimental version of the OpenCL 3.0 specifications was **v3.0.5**.

Changes from **v3.0.5**:

- Clarified that `memory_scope_all_devices` is supported only for OpenCL C 3.0 or newer.
- Defined ULP overflow leniency.
- Removed a confusing phrase about kernel argument pointer types.
- Clarified usage of feature test macros pre-OpenCL C 3.0.
- Clarified relationship between optional core features and extensions.
- Deprecated the `__OPENCL_C_VERSION__` predefined macro and clarified possible values of the macro for different versions of OpenCL.

Changes from **v3.0.6**:

- Clarified the argument to **vec_step** is not evaluated.
- Improved description for pipe specifier.
- Fixed parameter name in **work_group_broadcast** description.
- Clarified that the size of a pipe is implementation-defined.
- Moved descriptions of the identify value for exclusive scans.
- Fixed several bugs and formatting in the fast math ULP tables.
- Clarified the behavior of **work_group_broadcast**.
- Clarified the minimum OpenCL C version for the `opencl_unroll_hint` attribute.

Changes from **v3.0.7**:

- Clarified optionality support for double-precision literals.

Changes from **v3.0.14**:

- Improved capitalization and hyphenation consistency throughout the specs, see #902.
- Clarified that the **nextafter** built-in function works with all floating-point types, see #953.
- Clarified that the async copy and wait group events built-in functions must be called within converged control flow, see #1015.
- Removed unnecessary rounding mode text from the descriptions of the geometric and common functions, see #1027.

Changes from **v3.0.15**:

- 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.
- Fixed the derived formula for `atanh`, see #1048.
- Removed an incorrect statement about geometric functions operating component-wise, see #1137.
- Added new extension:
  - `cl_khr_kernel_clock` (experimental)

Changes from **v3.0.16**:

- Documented the error bounds for a non-derived `atan2` implementation with unsafe math optimizations, see #1073.
- Fixed a typo affecting `EPSILON` macros, see #1225.

Changes from **v3.0.17**:

- Fixed formatting issues in the builtins table for `ldexp` and `mix`, see #1290.
- Updated the ULP requirements for the half-precision divide and reciprocal, see #1293.
- Fixed a typo in the address space conversion example, see #1322.
- Clarified the unsafe math accuracy requirements for the embedded profile, see #1318.
- Clarified the data types that are supported for explicit conversions, see #1340.
- Clarified image conversion rules for signed and unsigned image channel data types, see #1348.

Changes from **v3.0.18**:

- Fixed an incorrect argument type for the **fast_distance** function, see #1374.
- Updated the ULP requirements for the half-precision square root, see #1386.
- Updated the ULP requirements for the half-precision tangent, see #1387.

Summary of Changes from OpenCL 3.0 to OpenCL 3.1

OpenCL 3.1 adds the OpenCL 3.1 C kernel language.

Other changes in OpenCL 3.1:

- Added support for the **z** and **t** `printf` length specifiers, see #654.
- Removed a few references to SPIR-V that do not belong in OpenCL C spec, see #1517.
- Fixed a typo for `fmin` for the `half` type, see #1519.
- Removed the Extending Attribute Qualifiers section, see #1532.
- Moved the documentation of cluster size restrictions to improve readability, see #1535.
- Clarified behavior for NaN to integer saturated conversions, see #1546.
- Promoted the following extensions to the core API:

    - `cl_khr_extended_bit_ops`
    - `cl_khr_integer_dot_product`
    - `cl_khr_subgroup_extended_types`
    - `cl_khr_subgroup_rotate`
    - `cl_khr_subgroup_shuffle`
    - `cl_khr_subgroup_shuffle_relative`

Changes from **v3.1.0** to **v3.1.1**:

- Clarified that the version requirements in this OpenCL C spec are for the OpenCL C language version, not the OpenCL device or platform version, see #1557.
- Clarified the description of the `andy` and `all` functions, see #1561.
- `cl_khr_fp16`
    - Clarified that the `halfn` vector data type is not a reserved data type when the extension is supported, see #1548.