# Table of Contents

1. Acknowledgements ................................................................. 12
2. Introduction ............................................................................ 15
3. SYCL architecture ................................................................. 17
   3.1. Overview ........................................................................... 17
   3.2. Anatomy of a SYCL application .......................................... 17
   3.3. Normative references ......................................................... 19
   3.4. Non-normative notes and examples ...................................... 19
   3.5. The SYCL platform model ................................................ 19
   3.6. The SYCL backend model ................................................ 20
      3.6.1. Platform mixed version support ................................. 21
   3.7. SYCL execution model ....................................................... 21
      3.7.1. SYCL application execution model ............................ 21
         3.7.1.1. SYCL backend resources managed by the SYCL application 21
         3.7.1.2. SYCL command groups and execution order ............ 22
         3.7.1.3. Controlling execution order with events ................. 24
      3.7.2. SYCL kernel execution model ..................................... 24
         3.7.2.1. Basic kernels ..................................................... 24
         3.7.2.2. ND-range kernels .............................................. 24
         3.7.2.3. Backend-specific kernels .................................. 25
   3.8. Memory model ................................................................. 25
      3.8.1. SYCL application memory model .................................. 25
      3.8.2. SYCL device memory model ....................................... 28
         3.8.2.1. Access to memory ............................................ 28
      3.8.3. SYCL memory consistency model ............................... 29
         3.8.3.1. Memory ordering ............................................. 30
         3.8.3.2. Memory scope ............................................... 30
         3.8.3.3. Atomic operations .......................................... 31
         3.8.3.4. Forward progress ........................................... 31
   3.9. The SYCL programming model ......................................... 32
      3.9.1. Minimum version of C++ .......................................... 32
      3.9.2. Alignment with future versions of C++ ........................ 32
      3.9.3. Basic data parallel kernels ....................................... 32
      3.9.4. Work-group data parallel kernels ............................... 33
      3.9.5. Hierarchical data parallel kernels ............................... 33
      3.9.6. Kernels that are not launched over parallel instances ..... 34
      3.9.7. Pre-defined kernels ................................................ 34
      3.9.8. Synchronization ....................................................... 34
         3.9.8.1. Synchronization in the SYCL application ................ 34
         3.9.8.2. Synchronization in SYCL kernels .......................... 35
      3.9.9. Error handling ....................................................... 35
      3.9.10. Fallback mechanism .............................................. 35
3.9.11. Scheduling of kernels and data movement .................................. 35
3.9.12. Managing object lifetimes .................................................. 36
3.9.13. Device discovery and selection ........................................... 36
3.9.14. Interfacing with the SYCL backend API .............................. 37
3.10. Memory objects ............................................................ 37
3.11. Multi-dimensional objects and linearization ............................. 38
  3.11.1. Linearization .......................................................... 38
  3.11.2. Multi-dimensional subscript operators ................................ 38
3.12. Implementation options .................................................... 39
  3.12.1. Single source multiple compiler passes ............................ 39
  3.12.2. Single source single compiler pass ................................ 39
  3.12.3. Library-only implementation ....................................... 39
3.13. Language restrictions in kernels ........................................... 39
  3.13.1. Device copyable ...................................................... 40
  3.14. Endianness support ...................................................... 41
3.15. Example SYCL application .................................................. 41
4. SYCL programming interface ................................................... 44
  4.1. Backends ........................................................................ 44
  4.1.1. Backend macros ........................................................ 44
  4.2. Generic vs non-generic SYCL ............................................. 44
  4.3. Header files and namespaces ............................................. 45
  4.4. Class availability .......................................................... 45
  4.5. Common interface .......................................................... 45
    4.5.1. Backend interoperability .......................................... 46
      4.5.1.1. Type traits `backend_traits` ................................ 46
      4.5.1.2. Template function `get_native` .............................. 47
      4.5.1.3. Template functions `make_*` ................................ 47
    4.5.2. Common reference semantics .................................... 49
    4.5.3. Common by-value semantics ...................................... 51
    4.5.4. Properties ............................................................ 53
      4.5.4.1. Properties interface .......................................... 54
  4.6. SYCL runtime classes ....................................................... 55
    4.6.1. Device selection ...................................................... 55
      4.6.1.1. Device selector ................................................ 55
    4.6.2. Platform class ........................................................ 59
      4.6.2.1. Platform interface ............................................. 59
      4.6.2.2. Platform information descriptors ......................... 61
    4.6.3. Context class ......................................................... 61
      4.6.3.1. Context interface .............................................. 62
      4.6.3.2. Context information descriptors ......................... 64
      4.6.3.3. Context properties .......................................... 65
    4.6.4. Device class .......................................................... 65
      4.6.4.1. Device interface .............................................. 65
4.6.4.2. Device information descriptors ..................................................... 70
4.6.4.3. Device aspects .............................................................................. 81
4.6.5. Queue class ...................................................................................... 85
  4.6.5.1. Queue interface ........................................................................... 86
  4.6.5.2. Queue shortcut functions ............................................................. 93
  4.6.5.3. Queue information descriptors .................................................... 99
  4.6.5.4. Queue properties ......................................................................... 99
  4.6.5.5. Queue error handling ................................................................. 100
4.6.6. Event class ........................................................................................ 100
  4.6.6.1. Event information and profiling descriptors .................................. 104
4.7. Data access and storage in SYCL ......................................................... 106
  4.7.1. Host allocation ............................................................................... 106
    4.7.1.1. Default allocators .................................................................. 106
  4.7.2. Buffers ............................................................................................ 107
    4.7.2.1. Buffer interface ...................................................................... 108
    4.7.2.2. Buffer properties .................................................................... 126
    4.7.2.3. Buffer synchronization rules ...................................................... 127
  4.7.3. Images ............................................................................................ 129
    4.7.3.1. Unsampled image interface ....................................................... 130
    4.7.3.2. Sampled image interface ......................................................... 143
    4.7.3.3. Image properties ...................................................................... 149
    4.7.3.4. Image synchronization rules ...................................................... 151
  4.7.4. Sharing host memory with the SYCL data management classes ........ 151
    4.7.4.1. Default behavior ..................................................................... 151
    4.7.4.2. SYCL ownership of the host memory ......................................... 151
    4.7.4.3. Shared SYCL ownership of the host memory ................................ 152
  4.7.5. Synchronization primitives ............................................................. 153
  4.7.6. Accessors ......................................................................................... 153
    4.7.6.1. Data type ............................................................................... 154
    4.7.6.2. Access modes .......................................................................... 154
    4.7.6.3. Deduction tags ........................................................................ 155
    4.7.6.4. Properties ............................................................................... 155
    4.7.6.5. Read only accessors ................................................................. 157
    4.7.6.6. Accessing elements of an accessor ......................................... 157
    4.7.6.7. Container interface ................................................................. 157
    4.7.6.8. Ranged accessors .................................................................... 158
    4.7.6.9. Buffer accessor for commands ................................................ 158
      4.7.6.9.1. Interface for buffer command accessors ............................. 159
      4.7.6.9.2. Deduction tags for buffer command accessors .................. 169
      4.7.6.9.3. Read only buffer command accessors and implicit conversions 169
      4.7.6.9.4. Deprecated features of the accessor class ......................... 170
        4.7.6.9.4.1. Aliased names ............................................................. 170
        4.7.6.9.4.2. Discard access modes ................................................. 170
5. SYCL Device Compiler

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.15.2. device_event class</td>
<td>383</td>
</tr>
<tr>
<td>4.15.3. Atomic references</td>
<td>384</td>
</tr>
<tr>
<td>4.15.4. Atomic types (deprecated)</td>
<td>395</td>
</tr>
<tr>
<td>4.15.5. Interaction with host code</td>
<td>404</td>
</tr>
<tr>
<td>4.16. Stream class</td>
<td>404</td>
</tr>
<tr>
<td>4.16.1. Stream class interface</td>
<td>405</td>
</tr>
<tr>
<td>4.16.2. Synchronization</td>
<td>409</td>
</tr>
<tr>
<td>4.16.3. Implicit flush</td>
<td>409</td>
</tr>
<tr>
<td>4.16.4. Performance note</td>
<td>409</td>
</tr>
<tr>
<td>4.17. SYCL built-in functions for SYCL host and device</td>
<td>409</td>
</tr>
<tr>
<td>4.17.1. Description of the built-in types available for SYCL host and device</td>
<td>410</td>
</tr>
<tr>
<td>4.17.2. Function objects</td>
<td>414</td>
</tr>
<tr>
<td>4.17.3. Group functions</td>
<td>415</td>
</tr>
<tr>
<td>4.17.3.1. Group type trait</td>
<td>416</td>
</tr>
<tr>
<td>4.17.3.2. group_broadcast</td>
<td>416</td>
</tr>
<tr>
<td>4.17.3.3. group_barrier</td>
<td>417</td>
</tr>
<tr>
<td>4.17.4. Group algorithms library</td>
<td>417</td>
</tr>
<tr>
<td>4.17.4.1. any_of, all_of and none_of</td>
<td>418</td>
</tr>
<tr>
<td>4.17.4.2. shift_left and shift_right</td>
<td>420</td>
</tr>
<tr>
<td>4.17.4.3. permute</td>
<td>420</td>
</tr>
<tr>
<td>4.17.4.4. select</td>
<td>421</td>
</tr>
<tr>
<td>4.17.4.5. reduce</td>
<td>421</td>
</tr>
<tr>
<td>4.17.4.6. exclusive_scan and inclusive_scan</td>
<td>422</td>
</tr>
<tr>
<td>4.17.5. Math functions</td>
<td>425</td>
</tr>
<tr>
<td>4.17.6. Integer functions</td>
<td>433</td>
</tr>
<tr>
<td>4.17.7. Common functions</td>
<td>435</td>
</tr>
<tr>
<td>4.17.8. Geometric functions</td>
<td>436</td>
</tr>
<tr>
<td>4.17.9. Relational functions</td>
<td>438</td>
</tr>
<tr>
<td>5. SYCL Device Compiler</td>
<td>443</td>
</tr>
<tr>
<td>5.1. Offline compilation of SYCL source files</td>
<td>443</td>
</tr>
<tr>
<td>5.2. Naming of kernels</td>
<td>443</td>
</tr>
<tr>
<td>5.3. Compilation of functions</td>
<td>444</td>
</tr>
<tr>
<td>5.4. Language restrictions for device functions</td>
<td>444</td>
</tr>
<tr>
<td>5.5. Built-in scalar data types</td>
<td>445</td>
</tr>
<tr>
<td>5.6. Preprocessor directives and macros</td>
<td>447</td>
</tr>
<tr>
<td>5.7. Optional kernel features</td>
<td>447</td>
</tr>
<tr>
<td>5.8. Attributes for device code</td>
<td>448</td>
</tr>
<tr>
<td>5.8.1. Kernel attributes</td>
<td>449</td>
</tr>
<tr>
<td>5.8.2. Device function attributes</td>
<td>453</td>
</tr>
<tr>
<td>5.9. Address-space deduction</td>
<td>454</td>
</tr>
<tr>
<td>5.9.1. Address space assignment</td>
<td>455</td>
</tr>
<tr>
<td>5.9.2. Common address space deduction rules</td>
<td>455</td>
</tr>
<tr>
<td>5.9.3. Generic as default address space</td>
<td>456</td>
</tr>
</tbody>
</table>
5.9.4. Inferred address space ................................................. 456
5.10. SYCL offline linking ..................................................... 456
5.10.1. SYCL functions and member functions linkage ............... 456
6. SYCL Extensions .......................................................... 458
6.1. Definition of an extension .............................................. 458
6.2. Requirements for an extension ....................................... 458
6.3. Guidelines for portable extensions ................................... 459
  6.3.1. Extension namespace ............................................... 459
  6.3.2. Names for extensions to existing classes or enumerations 459
  6.3.3. Feature test macros .................................................. 459
  6.3.4. Attribute namespace ............................................... 460
  6.3.5. Include file paths .................................................... 460
  6.3.6. Optional kernel features ........................................... 460
  6.3.7. Adding a backend ................................................... 460
Appendix A: Information descriptors ....................................... 461
  A.1. Platform information descriptors ................................. 461
  A.2. Context information descriptors .................................. 461
  A.3. Device information descriptors .................................... 461
  A.4. Queue information descriptors ................................... 464
  A.5. Kernel information descriptors .................................... 464
  A.6. Event information descriptors ...................................... 465
Appendix B: Feature sets ..................................................... 466
  B.1. Full feature set .......................................................... 466
  B.2. Reduced feature set ................................................... 466
  B.3. Compatibility ............................................................ 466
  B.4. Conformance ............................................................. 466
Appendix C: OpenCL backend specification ................................. 467
  C.1. SYCL application interoperability native backend objects 467
  C.2. Kernel function interoperability native backend objects .... 467
  C.3. Destruction of interop constructed objects with reference semantics 468
  C.4. SYCL for OpenCL framework ........................................ 468
  C.5. Mapping of SYCL programming model on top of OpenCL .... 469
    C.5.1. Backend specific information descriptors .................. 469
    C.5.2. OpenCL memory model ............................................ 469
    C.5.3. OpenCL interface for buffer command accessors ........ 470
    C.5.4. OpenCL resources managed by SYCL application .......... 470
  C.6. Interoperability with the OpenCL API .............................. 470
    C.7. Programming Interface ............................................. 473
      C.7.1. Construct SYCL objects from OpenCL ones............... 473
      C.7.2. Extension query .................................................. 475
      C.7.3. Reference counting ............................................... 476
      C.7.4. Errors and limitations ......................................... 476
      C.7.5. Interoperability with kernel bundles ....................... 476
Copyright (c) 2011-2023 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. 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.

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.

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

Some parts of this Specification are purely informative and so are EXCLUDED from the Scope of this Specification.

Where this Specification uses technical terminology, defined in the Glossary or otherwise, that refer to enabling technologies that are not expressly set forth in this Specification, those enabling technologies are EXCLUDED from the Scope of this Specification. For clarity, enabling technologies not disclosed with particularity in this Specification (e.g. semiconductor manufacturing technology, hardware architecture, processor architecture or microarchitecture, memory architecture, compiler technology, object oriented technology, basic operating system technology, compression technology, algorithms, and so on) are NOT to be considered expressly set forth; only those application program interfaces and data structures disclosed with particularity are included in the Scope of this Specification.

For purposes of the Khronos Intellectual Property Rights Policy as it relates to the definition of Necessary Patent Claims, all recommended or optional features, behaviors and functionality set forth in this Specification, if implemented, are considered to be included as Compliant Portions.

Where this Specification includes normative references to external documents, only the specifically identified sections of those external documents are INCLUDED in the Scope of this Specification. If not created by Khronos, those external documents may contain contributions from non-members of Khronos not covered by the Khronos Intellectual Property Rights Policy.

This document contains extensions which are not ratified by Khronos, and as such is not a ratified Specification, though it contains text from (and is a superset of) the ratified SYCL Specification. The ratified version of the SYCL Specification can be found at https://www.khronos.org/registry/SYCL .

Khronos and Vulkan are registered trademarks, and SPIR-V is a trademark of The Khronos Group Inc. OpenCL is a trademark of Apple Inc. and OpenGL is a registered trademarks of Hewlett Packard Enterprise, all used under license by Khronos. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.
Chapter 1. Acknowledgements

Editors

• Maria Rovatsou, Codeplay
• Lee Howes, Qualcomm
• Ronan Keryell, AMD (current)

Contributors

• Eric Berdahl, Adobe
• Shivani Gupta, Adobe
• David Neto, Altera
• Carlo Bertolli, AMD
• Andrew Gozillon, AMD
• Gauthier Harnisch, AMD
• Ronan Keryell, AMD
• Yiannis Papadopoulos, AMD
• Brian Sumner, AMD
• Lin-Ya Yu, AMD
• Thomas Applencourt, Argonne National Laboratory
• Hal Finkel, Argonne National Laboratory
• Kevin Harms, Argonne National Laboratory
• Nevin Liber, Argonne National Laboratory
• Anastasia Stulova, ARM
• Balázs Keszhelyi, Broadcom
• Alexandra Crabb, Caster Communications
• Stuart Adams, Codeplay
• Verena Beckham, Codeplay
• Aidan Belton, Codeplay
• Gordon Brown, Codeplay
• Morris Hafner, Codeplay
• Alexander Johnston, Codeplay
• Marios Katsigiannis, Codeplay
• Paul Keir, Codeplay
• Steffen Larsen, Codeplay
• Victor Lomüller, Codeplay
• Tomas Matheson, Codeplay
• Duncan McBain, Codeplay
• Nicolas Miller, Codeplay
• Georgi Mirazchiyski, Codeplay
• Ralph Potter, Codeplay
• Ruyma Reyes, Codeplay
• Andrew Richards, Codeplay
• Maria Rovatsou, Codeplay
• Panagiotis Stratis, Codeplay
• Michael Wong, Codeplay
• Peter Žužek, Codeplay
• Matt Newport, EA
• Rasool Maghareh, Huawei Technologies Co. Ltd.
• Guansong Zhang, Huawei Technologies Co. Ltd.
• Ruslan Arutyunyan, Intel
• Alexey Bader, Intel
• James Brodman, Intel
• Ilya Burylov, Intel
• Jessica Davies, Intel
• Felipe de Azevedo Piovezan, Intel
• Allen Hux, Intel
• Michael Kinsner, Intel
• Greg Lueck, Intel
• John Pennycook, Intel
• Roland Schulz, Intel
• Sergey Semenov, Intel
• Jason Sewall, Intel
• James O’Riordón, Khronos
• Jon Leech, Luna Princps LLC
• Kathleen Mattson, Miller & Mattson, LLC
• Dave Miller, Miller & Mattson, LLC
• Stéphanie Even, Mercedes-Benz Research and Development NA
• Chris Gearing, Mobileye
• Seiji Nishimura, NSITEXE, Inc.
• Neil Trevett, NVIDIA
• Lee Howes, Qualcomm
• Chu-Cheow Lim, Qualcomm
• Jack Liu, Qualcomm
• Hongqiang Wang, Qualcomm
• Ruihao Zhang, Qualcomm
• Dave Airlie, Red Hat
• Hyesun Hong, Samsung Electronics
• Aksel Alpay, Self
• Dániel Berényi, Self
• Máté Nagy-Egri, Stream HPC
• Bálint Soproni, Stream HPC
• Tom Deakin, University of Bristol
• Philip Salzmann, University of Innsbruck
• Peter Thoman, University of Innsbruck
• Biagio Cosenza, University of Salerno
• Paul Preney, University of Windsor
Chapter 2. Introduction

SYCL (pronounced “sickle”) is a royalty-free, cross-platform abstraction C++ programming model for heterogeneous computing. SYCL builds on the underlying concepts, portability and efficiency of parallel API or standards like OpenCL while adding much of the ease of use and flexibility of single-source C++.

Developers using SYCL are able to write standard modern C++ code, with many of the techniques they are accustomed to, such as inheritance and templates. At the same time, developers have access to the full range of capabilities of the underlying implementation (such as OpenCL) both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly using the underneath implementation, via their APIs.

To reduce programming effort and increase the flexibility with which developers can write code, SYCL extends the concepts found in standards like OpenCL model in a few ways beyond the general use of C++ features:

- execution of parallel kernels on a heterogeneous device is made simultaneously convenient and flexible. Common parallel patterns are prioritized with simple syntax, which through a series C++ types allow the programmer to express additional requirements, such as synchronization, if needed;
- when using buffers and accessors, data access in SYCL is separated from data storage. By relying on the C++-style resource acquisition is initialization (RAII) idiom to capture data dependencies between device code blocks, the runtime library can track data movement and provide correct behavior without the complexity of manually managing event dependencies between kernel instances and without the programmer having to explicitly move data. This approach enables the data-parallel task-graphs that might be already part of the execution model to be built up easily and safely by SYCL programmers;
- Unified Shared Memory (USM) provides a mechanism for explicit data allocation and movement. This approach enables the use of pointer-based algorithms and data structures on heterogeneous devices, and allows for increased re-use of code across host and device;
- the hierarchical parallelism syntax offers a way of expressing data parallelism similar to the OpenCL device or OpenMP target device execution model in an easy-to-understand modern C++ form. It more cleanly layers parallel loops and synchronization points to avoid fragmentation of code and to more efficiently map to CPU-style architectures.

SYCL retains the execution model, runtime feature set and device capabilities inspired by the OpenCL standard. This standard imposes some limitations on the full range of C++ features that SYCL is able to support. This ensures portability of device code across as wide a range of devices as possible. As a result, while the code can be written in standard C++ syntax with interoperability with standard C++ programs, the entire set of C++ features is not available in SYCL device code. In particular, SYCL device code, as defined by this specification, does not support virtual function calls, function pointers in general, exceptions, runtime type information or the full set of C++ libraries that may depend on these features or on features of a particular host compiler. Nevertheless, these basic restrictions can be relieved by some specific Khronos or vendor extensions.

SYCL implements an SMCP design which offers the power of source integration while allowing tool-chains to remain flexible. The SMCP design supports embedding of code intended to be compiled for a device, for example a GPU, inline with host code. This embedding of code offers three primary benefits:

Simplicity

For novice programmers using frameworks like OpenCL, the separation of host and device source code in OpenCL can become complicated to deal with, particularly when similar kernel code is used for multiple different operations on different data types. A single compiler flow and integrated tool chain combined with libraries that perform a lot of simple tasks simplifies initial OpenCL programs to a minimum complexity. This reduces the learning curve for programmers new to heterogeneous programming and allows them to concentrate on parallelization techniques rather than syntax.
Reuse

C++'s type system allows for complex interactions between different code units and supports efficient abstract interface design and reuse of library code. For example, a transform or map operation applied to an array of data may allow specialization on both the operation applied to each element of the array and on the type of the data. The SMCP design of SYCL enables this interaction to bridge the host code/device code boundary such that the device code to be specialized on both of these factors directly from the host code.

Efficiency

Tight integration with the type system and reuse of library code enables a compiler to perform inlining of code and to produce efficient specialized device code based on decisions made in the host code without having to generate kernel source strings dynamically.

The use of C++ features such as generic programming, templated code, functional programming and inheritance on top of existing heterogeneous execution model opens a wide scope for innovation in software design for heterogeneous systems. Clean integration of device and host code within a single C++ type system enables the development of modern, templated generic and adaptable libraries that build simple, yet efficient, interfaces to offer more developers access to heterogeneous computing capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on open and widely implemented standard foundation like OpenCL or Vulkan.

SYCL is designed to be as close to standard C++ as possible. In practice, this means that as long as no dependence is created on SYCL’s integration with the underlying implementation, a standard C++ compiler can compile SYCL programs and they will run correctly on a host CPU. Any use of specialized low-level features can be masked using the C preprocessor in the same way that compiler-specific intrinsics may be hidden to ensure portability between different host compilers.

SYCL is designed to allow a compilation flow where the source file is passed through multiple different compilers, including a standard C++ host compiler of the developer’s choice, and where the resulting application combines the results of these compilation passes. This is distinct from a single-source flow that might use language extensions that preclude the use of a standard host compiler. The SYCL standard does not preclude the use of a single compiler flow, but is designed to not require it. SYCL can also be implemented purely as a library, in which case no special compiler support is required at all.

The advantages of this design are two-fold. First, it offers better integration with existing tool chains. An application that already builds using a chosen compiler can continue to do so when SYCL code is added. Using the SYCL tools on a source file within a project will both compile for a device and let the same source file be compiled using the same host compiler that the rest of the project is compiled with. Linking and library relationships are unaffected. This design simplifies porting of pre-existing applications to SYCL. Second, the design allows the optimal compiler to be chosen for each device where different vendors may provide optimized tool-chains.

To summarize, SYCL enables computational kernels to be written inside C++ source files as normal C++ code, leading to the concept of “single-source” programming. This means that software developers can develop and use generic algorithms and data structures using standard C++ template techniques, while still supporting multi-platform, multi-device heterogeneous execution. Access to the low level APIs of an underlying implementation (such as OpenCL) is also supported. The specification has been designed to enable implementation across as wide a variety of platforms as possible as well as ease of integration with other platform-specific technologies, thereby letting both users and implementers build on top of SYCL as an open platform for system-wide heterogeneous processing innovation.
Chapter 3. SYCL architecture

This chapter describes the structure of a SYCL application, and how the SYCL generic programming model lays out on top of a number of SYCL backends.

3.1. Overview

SYCL is an open industry standard for programming a heterogeneous system. The design of SYCL allows standard C++ source code to be written such that it can run on either an heterogeneous device or on the host.

The terminology used for SYCL inherits historically from OpenCL with some SYCL-specific additions. However SYCL is a generic C++ programming model that can be laid out on top of other heterogeneous APIs apart from OpenCL. SYCL implementations can provide SYCL backends for various heterogeneous APIs, implementing the SYCL general specification on top of them. We refer to this heterogeneous API as the SYCL backend API. The SYCL general specification defines the behavior that all SYCL implementations must expose to SYCL users for a SYCL application to behave as expected.

A function object that can execute on a device exposed by a SYCL backend API is called a SYCL kernel function.

To ensure maximum interoperability with different SYCL backend APIs, software developers can access the SYCL backend API alongside the SYCL general API whenever they include the SYCL backend interoperability headers. However, interoperability is a SYCL backend-specific feature. An application that uses interoperability does not conform to the SYCL general application model, since it is not portable across backends.

The target users of SYCL are C++ programmers who want all the performance and portability features of a standard like OpenCL, but with the flexibility to use higher-level C++ abstractions across the host/device code boundary. Developers can use most of the abstraction features of C++, such as templates, classes and operator overloading.

However, some C++ language features are not permitted inside kernels, due to the limitations imposed by the capabilities of the underlying heterogeneous platforms. These features include virtual functions, virtual inheritance, throwing/catching exceptions, and run-time type-information. These features are available outside kernels as normal. Within these constraints, developers can use abstractions defined by SYCL, or they can develop their own on top. These capabilities make SYCL ideal for library developers, middleware providers and application developers who want to separate low-level highly-tuned algorithms or data structures that work on heterogeneous systems from higher-level software development. Software developers can produce templated algorithms that are easily usable by developers in other fields.

3.2. Anatomy of a SYCL application

Below is an example of a typical SYCL application which schedules a job to run in parallel on any heterogeneous device available.

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

int main() {
  int data[1024]; // Allocate data to be worked on
  // Create a default queue to enqueue work to the default device
}```
queue myQueue;

// By wrapping all the SYCL work in a {} block, we ensure
// all SYCL tasks must complete before exiting the block,
// because the destructor of resultBuf will wait
{
// Wrap our data variable in a buffer
buffer<int, 1> resultBuf { data, range<1> { 1024 } };

// Create a command group to issue commands to the queue
myQueue.submit([&](handler &cgh)
{
// Request write access to the buffer without initialization
accessor writeResult { resultBuf, cgh, write_only, no_init };

// Enqueue a parallel_for task with 1024 work-items
cgh.parallel_for(1024, [=](id<1> idx) {
  // Initialize each buffer element with its own rank number starting at 0
  writeResult[idx] = idx;
}); // End of the kernel function

// End of our commands for this queue
}); // End of scope, so we wait for work producing resultBuf to complete

// Print result
for (int i = 0; i < 1024; i++)
  std::cout << "data[" << i << "] = " << data[i] << std::endl;

return 0;
}

At line 1, we #include the SYCL header files, which provide all of the SYCL features that will be used.

A SYCL application runs on a SYCL Platform. The application is structured in three scopes which specify the different sections; application scope, command group scope and kernel scope. The kernel scope specifies a single kernel function that will be, or has been, compiled by a device compiler and executed on a device. In this example kernel scope is defined by lines 25 to 26. The command group scope specifies a unit of work which is comprised of a SYCL kernel function and accessors. In this example command group scope is defined by lines 20 to 28. The application scope specifies all other code outside of a command group scope. These three scopes are used to control the application flow and the construction and lifetimes of the various objects used within SYCL, as explained in Section 3.9.12.

A SYCL kernel function is the scoped block of code that will be compiled using a device compiler. This code may be defined by the body of a lambda function or by the operator() function of a function object. Each instance of the SYCL kernel function will be executed as a single, though not necessarily entirely independent, flow of execution and has to adhere to restrictions on what operations may be allowed to enable device compilers to safely compile it to a range of underlying devices.

The parallel_for member function can be templated with a class. This class is used to manually name the kernel when desired, such as to avoid a compiler-generated name when debugging a kernel defined through a lambda, to provide a known name with which to apply build options to a kernel, or to ensure compatibility with multiple compiler-pass implementations.

The parallel_for member function creates an instance of a kernel, which is the entity that will be enqueued within a command group. In the case of parallel_for the SYCL kernel function will be executed over the given range from 0 to 1023. The different member functions to execute kernels can be found in Section 4.9.4.2.
A command group scope is the syntactic scope wrapped by the construction of a command group function object as seen on line 19. The command group function object may invoke only a single SYCL kernel function, and it takes a parameter of type command group handler, which is constructed by the runtime.

All the requirements for a kernel to execute are defined in this command group scope, as described in Section 3.7.1. In this case the constructor used for myQueue on line 9 is the default constructor, which allows the queue to select the best underlying device to execute on, leaving the decision up to the runtime.

In SYCL, data that is required within a SYCL kernel function must be contained within a buffer, image, or USM allocation, as described in Section 3.8. We construct a buffer on line 16. Access to the buffer is controlled via an accessor which is constructed on line 21. The buffer is used to keep track of access to the data and the accessor is used to request access to the data on a queue, as well as to track the dependencies between SYCL kernel function. In this example the accessor is used to write to the data buffer on line 26.

### 3.3. Normative references

The documents in the following list are referred to within this SYCL specification, and their content is a requirement for this document.

1. **C++17**: ISO/IEC 14882:2017 Clauses 1-19, referred to in this specification as the C++ core language. The SYCL specification refers to language in the following C++ defect reports and assumes a compiler that implements them: DR2325.


### 3.4. Non-normative notes and examples

Unless stated otherwise, text within this SYCL specification is normative and defines the required behavior of a SYCL implementation. Non-normative / informational notes are included within this specification using a “note” callout, of the form:

Information within a note callout, such as this text, is for informational purposes and does not impose requirements on or specify behavior of a SYCL implementation.

Source code examples within the specification are provided to aid with understanding, and are non-normative.

In case of any conflict between a non-normative note or source example, and normative text within the specification, the normative text must be taken to be correct.

### 3.5. The SYCL platform model

The SYCL platform model is based on the OpenCL platform model. The model consists of a host connected to one or more heterogeneous devices, called devices.

A SYCL context is constructed, either directly by the user or implicitly when creating a queue, to hold all the runtime information required by the SYCL runtime and the SYCL backend to operate on a device, or group of devices. When a group of devices can be grouped together on the same context, they have some visibility of each other's memory objects. The SYCL runtime can assume that memory is visible across all devices in the same context. Not all devices exposed from the same platform can be grouped together in the same context.

A SYCL application executes on the host as a standard C++ program. Devices are exposed through differ-
SYCL backends to the SYCL application. The SYCL application submits command group function objects to queues. Each queue enables execution on a given device.

The SYCL runtime then extracts operations from the command group function object, e.g. an explicit copy operation or a SYCL kernel function. When the operation is a SYCL kernel function, the SYCL runtime uses a SYCL backend-specific mechanism to extract the device binary from the SYCL application and pass it to the heterogeneous API for execution on the device.

A SYCL device is divided into one or more compute units (CUs) which are each divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. How computation is mapped to PEs is SYCL backend and device specific. Two devices exposed via two different backends can map computations differently to the same device.

When a SYCL application contains SYCL kernel function objects, the SYCL implementation must provide an offline compilation mechanism that enables the integration of the device binaries into the SYCL application. The output of the offline compiler can be an intermediate representation, such as SPIR-V, that will be finalized during execution or a final device ISA.

A device may expose special purpose functionality as a built-in function. The SYCL API exposes functions to query and dispatch said built-in functions. Some SYCL backends and devices may not support programmable kernels, and only support built-in functions.

### 3.6. The SYCL backend model

SYCL is a generic programming model for the C++ language that can target multiple heterogeneous APIs, such as OpenCL.

SYCL implementations enable these target APIs by implementing SYCL backends. For a SYCL implementation to be conformant on said SYCL backend, it must execute the SYCL generic programming model on the backend. All SYCL implementations must provide at least one backend.

The present document covers the SYCL generic interface available to all SYCL backends. How the SYCL generic interface maps to a particular SYCL backend is defined either by a separate SYCL backend specification document, provided by the Khronos SYCL group, or by the SYCL implementation documentation. Whenever there is a SYCL backend specification document, this takes precedence over SYCL implementation documentation.

When a SYCL user builds their SYCL application, she decides which of the SYCL backends will be used to build the SYCL application. This is called the set of active backends. Implementations must ensure that the active backends selected by the user can be used simultaneously by the SYCL implementation at runtime. If two backends are available at compile time but will produce an invalid SYCL application at runtime, the SYCL implementation must emit a compilation error.

A SYCL application built with a number of active backends does not necessarily guarantee that said backends can be executed at runtime. The subset of active backends available at runtime is called available backends. A backend is said to be available if the host platform where the SYCL application is executed exposes support for the heterogeneous API required for the SYCL backend.

It is implementation dependent whether certain backends require third-party libraries to be available in the system. Failure to have all dependencies required for all active backends at runtime will cause the SYCL application to not run.

Once the application is running, users can query what SYCL platforms are available. SYCL implementations will expose the devices provided by each backend grouped into platforms. A backend must expose at least one platform.

Under the SYCL backend model, SYCL objects can contain one or multiple references to a certain SYCL backend native type. Not all SYCL objects will map directly to a SYCL backend native type. The mapping
of SYCL objects to SYCL backend native types is defined by the SYCL backend specification document when available, or by the SYCL implementation otherwise.

To guarantee that multiple SYCL backend objects can interoperate with each other, SYCL memory objects are not bound to a particular SYCL backend. SYCL memory objects can be accessed from any device exposed by an available backend. SYCL Implementations can potentially map SYCL memory objects to multiple native types in different SYCL backends.

Since SYCL memory objects are independent of any particular SYCL backend, SYCL command groups can request access to memory objects allocated by any SYCL backend, and execute it on the backend associated with the queue. This requires the SYCL implementation to be able to transfer memory objects across SYCL backends.

USM allocations are subject to the limitations described in Section 4.8.

When a SYCL application runs on any number of SYCL backends without relying on any SYCL backend-specific behavior or interoperability, it is said to be a SYCL general application, and it is expected to run in any SYCL-conformant implementation that supports the required features for the application.

### 3.6.1. Platform mixed version support

The SYCL generic programming model exposes a number of platforms, each of them exposing a number of devices. Each platform is bound to a certain SYCL backend. SYCL devices associated with said platform are associated with that SYCL backend.

Although the APIs in the SYCL generic programming model are defined according to this specification and their version is indicated by the macro SYCL_LANGUAGE_VERSION, this does not apply to APIs exposed by the SYCL backends. Each SYCL backend provides its own document that defines its APIs, and that document tells how to query for the device and platform versions.

### 3.7. SYCL execution model

As described in Section 3.2, a SYCL application is comprised of three scopes: application scope, command group scope, and kernel scope. Code in the application scope and command group scope runs on the host and is governed by the SYCL application execution model. Code in the kernel scope runs on a device and is governed by the SYCL kernel execution model.

A SYCL device does not necessarily correspond to a physical accelerator. A SYCL implementation may choose to expose some or all of the host's resources as a SYCL device; such an implementation would execute code in kernel scope on the host, but that code would still be governed by the SYCL kernel execution model.

### 3.7.1. SYCL application execution model

The SYCL application defines the execution order of the kernels by grouping each kernel with its requirements into a command group function object. Command group function objects are submitted for execution via a queue object, which defines the device where the kernel will run. This specification sometimes refers to this as “submitting the kernel to a device”. The same command group object can be submitted to different queues. When a command group is submitted to a SYCL queue, the requirements of the kernel execution are captured. The implementation can start executing a kernel as soon as its requirements have been satisfied.

#### 3.7.1.1. SYCL backend resources managed by the SYCL application

The SYCL runtime integrated with the SYCL application will manage the resources required by the SYCL backend API to manage the heterogeneous devices it is providing access to. This includes, but is not limi-
The SYCL programming interface represents the lifetime of the resources managed by the SYCL application using RAII rules. Construction of a SYCL object will typically entail the creation of multiple SYCL backend objects, which will be properly released on destruction of said SYCL object. The overall rules for construction and destruction are detailed in Chapter 4. Those SYCL backends with a SYCL backend document will detail how the resource management from SYCL objects map down to the SYCL backend objects.

In SYCL, the minimum required object for submitting work to devices is the queue, which contains references to a platform, device and a context internally.

The resources managed by SYCL are:

1. **Platforms**: all features of SYCL backend APIs are implemented by platforms. A platform can be viewed as a given vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user which are accessible through a `sycl::platform` object.

2. **Contexts**: any SYCL backend resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Devices belonging to the same context must be able to access each other's global memory using some implementation-specific mechanism. A given context can only wrap devices owned by a single platform. A context is exposed to the user with a `sycl::context` object.

3. **Devices**: platforms provide one or more devices for executing SYCL kernels. In SYCL, a device is accessible through a `sycl::device` object.

4. **Kernels**: the SYCL functions that run on SYCL devices are defined as C++ function objects (a named function object type or a lambda function). A kernel can be introspected through a `sycl::kernel` object.

   Note that some SYCL backends may expose non-programmable functionality as pre-defined kernels.

5. **Kernel bundles**: Kernels are stored internally in the SYCL application as device images, and these device images can be grouped into a `sycl::kernel_bundle` object. These objects provide a way for the application to control the online compilation of kernels for devices.

6. **Queues**: SYCL kernels execute in command queues. The user must create a `sycl::queue` object, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. SYCL queues execute kernels on a particular device of a particular context, but can have dependencies from any device on any available SYCL backend.

The SYCL implementation guarantees the correct initialization and destruction of any resource handled by the underlying SYCL backend API, except for those the user has obtained manually via the SYCL interoperability API.

### 3.7.1.2. SYCL command groups and execution order

By default, SYCL queues execute kernel functions in an out-of-order fashion based on dependency information. Developers only need to specify what data is required to execute a particular kernel. The SYCL runtime will guarantee that kernels are executed in an order that guarantees correctness. By specifying access modes and types of memory, a directed acyclic dependency graph (DAG) of kernels is built at runtime. This is achieved via the usage of command group objects. A SYCL command group object defines a set of requisites \((R)\) and a kernel function \((k)\). A command group is submitted to a queue when using the `sycl::queue::submit` member function.

A **requisite** \((r)\) is a requirement that must be fulfilled for a kernel-function \((k)\) to be executed on a particular device. For example, a requirement may be that certain data is available on a device, or that another command group has finished execution. An implementation may evaluate the requirements of a
command group at any point after it has been submitted. The processing of a command group is the process by which a SYCL runtime evaluates all the requirements in a given \( R \). The SYCL runtime will execute \( k \) only when all \( r \) are satisfied (i.e., when all requirements are satisfied). To simplify the notation, in the specification we refer to the set of requirements of a command group named \( CG_foo \) as \( CG_foo = r_1, ..., r_n \).

The evaluation of a requisite (\( \text{Satisfied}(r_i) \)) returns the status of the requisite, which can be True or False. A satisfied requisite implies the requirement is met. \( \text{Satisfied}(r_i) \) never alters the requisite, only observes the current status. The implementation may not block to check the requisite, and the same check can be performed multiple times.

An action \( (a) \) is a collection of implementation-defined operations that must be performed in order to satisfy a requisite. The set of actions for a given command group \( A \) is permitted to be empty if no operation is required to satisfy the requirement. The notation \( a \) represents the action required to satisfy \( r_i \). Actions of different requisites can be satisfied in any order with respect to each other without side effects (i.e., given two requirements \( r_j \) and \( r_k \), \( (r_j, r_k) \equiv (r_k, r_j) \)). The intersection of two actions is not necessarily empty. Actions can include (but are not limited to): memory copy operations, mapping operations, host side synchronization, or implementation-specific behavior.

Finally, Performing an action (\( \text{Perform}(a_i) \)) executes the action operations required to satisfy the requisite \( r_i \). Note that, after \( \text{Perform}(a_i) \), the evaluation \( \text{Satisfied}(r_i) \) will return True until the kernel is executed. After the kernel execution, it is not defined whether a different command group with the same requirements needs to perform the action again, where actions of different requisites inside the same command group object can be satisfied in any order with respect to each other without side effects: Given two requirements \( r_j \) and \( r_k \), \( \text{Perform}(a_j) \) followed by \( \text{Perform}(a_k) \) is equivalent to \( \text{Perform}(a_k) \) followed by \( \text{Perform}(a_j) \).

The requirements of different command groups submitted to the same or different queues are evaluated in the relative order of submission. Command group objects whose intersection of requirement sets is not empty are said to depend on each other. They are executed in order of submission to the queue. If command groups are submitted to different queues or by multiple threads, the order of execution is determined by the SYCL runtime. Note that independent command group objects can be submitted simultaneously without affecting dependencies.

Table 1 illustrates the execution order of three command group objects \( (CG_aCG_bCG_c) \) with certain requirements submitted to the same queue. Both \( CG_a \) and \( CG_b \) only have one requirement, \( r_1 \) and \( r_2 \) respectively. \( CG_c \) requires both \( r_1 \) and \( r_2 \). This enables the SYCL runtime to potentially execute \( CG_a \) and \( CG_b \) simultaneously, whereas \( CG_c \) cannot be executed until both \( CG_a \) and \( CG_b \) have been completed. The SYCL runtime evaluates the requisites and performs the actions required (if any) for the \( CG_a \) and \( CG_b \). When evaluating the requisites of \( CG_c \), they will be satisfied once the \( CG_a \) and \( CG_b \) have finished.

### Table 1. Execution order of three command groups submitted to the same queue

<table>
<thead>
<tr>
<th>SYCL Application Enqueue Order</th>
<th>SYCL Kernel Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sycl::queue syclQueue;</code></td>
<td>( CG_a(r_1) )</td>
</tr>
<tr>
<td><code>syclQueue.submit(CG_a(r_1));</code></td>
<td>( CG_b(r_2) )</td>
</tr>
<tr>
<td><code>syclQueue.submit(CG_b(r_2));</code></td>
<td>( CG_c(r_1, r_2) )</td>
</tr>
<tr>
<td><code>syclQueue.submit(CG_c(r_1, r_2));</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 uses three separate SYCL queue objects to submit the same command group objects as before. Regardless of using three different queues, the execution order of the different command group objects is the same. When different threads enqueue to different queues, the execution order of the command group will be the order in which the submit member functions are executed. In this case, since the different command group objects execute on different devices, the actions required to satisfy the requirements may be different (e.g., the SYCL runtime may need to copy data to a different device in a separate context).
### 3.7.1.3. Controlling execution order with events

Submitting an action for execution returns an **event** object. Programmers may use these events to explicitly synchronize programs. Host code can wait for an event to complete, which will block execution on the host until the action represented by the event has completed. The **event** class is described in greater detail in Section 4.6.6.

Events may also be used to explicitly order the execution of kernels. Host code may wait for the completion of specific event, which blocks execution on the host until that event's action has completed. Events may also define requisites between **command groups**. Using events in this manner informs the runtime that one or more **command groups** must complete before another **command group** may begin executing. See Section 4.9.4.1 for greater detail.

### 3.7.2. SYCL kernel execution model

When a kernel is submitted for execution, an index space is defined. An instance of the kernel body executes for each point in this index space. This kernel instance is called a **work-item** and is identified by its point in the index space, which provides a **global id** for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary by using the work-item global id to specialize the computation.

An index space of size zero is allowed. All aspects of kernel execution proceed as normal with the exception that the kernel function itself is not executed. Note this means the command queue will still schedule this kernel after satisfying the requirements and this satisfies requirements of any dependent enqueued kernels.

#### 3.7.2.1. Basic kernels

SYCL allows a simple execution model in which a kernel is invoked over an **N**-dimensional index space defined by **range<N>**, where **N** is one, two or three. Each work-item in such a kernel executes independently.

Each work-item is identified by a value of type **item<N>**. The type **item<N>** encapsulates a work-item identifier of type **id<N>** and a **range<N>** representing the number of work-items executing the kernel.

#### 3.7.2.2. ND-range kernels

Work-items can be organized into **work groups**, providing a more coarse-grained decomposition of the index space. Each work-group is assigned a unique **work-group id** with the same dimensionality as the index space used for the work-items. Work-items are each assigned a **local id**, unique within the work-group, so that a single work-item can be uniquely identified by its global id or by a combination of its local id and work-group id. The work-items in a given work-group execute on the processing elements of a single compute unit.

When work-groups are used in SYCL, the index space is called an **nd-range**. An ND-range is an **N**-dimensional index space, where **N** is one, two or three. In SYCL, the ND-range is represented via the

### Table 2. Execution order of three command groups submitted to the different queues

<table>
<thead>
<tr>
<th>SYCL Application Enqueue Order</th>
<th>SYCL Kernel Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>sycl::queue syclQueue1;</td>
<td></td>
</tr>
<tr>
<td>sycl::queue syclQueue2;</td>
<td></td>
</tr>
<tr>
<td>sycl::queue syclQueue3;</td>
<td></td>
</tr>
<tr>
<td>syclQueue1.submit(CG(a(r1)));</td>
<td></td>
</tr>
<tr>
<td>syclQueue2.submit(CG(b(r2)));</td>
<td></td>
</tr>
<tr>
<td>syclQueue3.submit(CG(c(r1,r2))</td>
<td></td>
</tr>
</tbody>
</table>
nd_range<N> class. An nd_range<N> is made up of a global range and a local range, each represented via values of type range<N>. Additionally, there can be a global offset, represented via a value of type id<N>; this is deprecated in SYCL 2020. The types range<N> and id<N> are each \(N\)-element arrays of integers. The iteration space defined via an nd_range<N> is an \(N\)-dimensional index space starting at the ND-range’s global offset whose size is its global range, split into work-groups of the size of its local range.

Each work-item in the ND-range is identified by a value of type nd_item<N>. The type nd_item<N> encapsulates a global id, local id and work-group id, all of type id<N> (the iteration space offset also of type id<N>), but this is deprecated in SYCL 2020), as well as global and local ranges and synchronization operations necessary to make work-groups useful. Work-groups are assigned ids using a similar approach to that used for work-item global ids. Work-items are assigned to a work-group and given a local id with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group id and the local id within a work-group uniquely defines a work-item.

3.7.2.3. Backend-specific kernels

SYCL allows a SYCL backend to expose fixed functionality as non-programmable built-in kernels. The availability and behavior of these built-in kernels are SYCL backend-specific, and are not required to follow the SYCL execution and memory models. Furthermore the interface exposed utilize these built-in kernels is also SYCL backend-specific. See the relevant backend specification for details.

3.8. Memory model

Since SYCL is a single-source programming model, the memory model affects both the application and the device kernel parts of a program. On the SYCL application, the SYCL runtime will make sure data is available for execution of the kernels. On the SYCL device kernel, the SYCL backend rules describing how the memory behaves on a specific device are mapped to SYCL C++ constructs. Thus it is possible to program kernels efficiently in pure C++.

3.8.1. SYCL application memory model

The application running on the host uses SYCL buffer objects using instances of the sycl::buffer class or USM allocation functions to allocate memory in the global address space, or can allocate specialized image memory using the sycl::unsampled_image and sycl::sampled_image classes.

In the SYCL application, memory objects are bound to all devices in which they are used, regardless of the SYCL context where they reside. SYCL memory objects (namely, buffer and image objects) can encapsulate multiple underlying SYCL backend memory objects together with multiple host memory allocations to enable the same object to be shared between devices in different contexts, platforms or backends. USM allocations uniquely identify a memory allocation and are bound to a SYCL context. They are only valid on the backend used by the context.

The order of execution of command group objects ensures a sequentially consistent access to the memory from the different devices to the memory objects. Accessing a USM allocation does not alter the order of execution. Users must explicitly inform the SYCL runtime of any requirements necessary for a legal execution.

To access a memory object, the user must create an accessor object which parameterizes the type of access to the memory object that a kernel or the host requires. The accessor object defines a requirement to access a memory object, and this requirement is defined by construction of an accessor, regardless of whether there are any uses in a kernel or by the host. An accessor object specifies whether the access is via global memory, constant memory or image samplers and their associated access functions. The accessor also specifies whether the access is read-only (RO), write-only (WO) or read-write (RW). An optional no_init property can be added to an accessor to tell the system to discard any previous contents of the data the accessor refers to, so there are two additional requirement types: no-init-write-only (NWO) and no-init-read-write (NRW). For simplicity, when a requisite represents an accessor object in a
certain access mode, we represent it as MemoryObject\_AccessMode. For example, an accessor that accesses memory object \texttt{buf1} in RW mode is represented as \texttt{buf1\_rw}. A \textbf{command group} object that uses such an accessor is represented as \texttt{CG(buf1\_rw)}. The \textbf{action} required to satisfy a requisite and the location of the latest copy of a memory object will vary depending on the implementation.

\textbf{Table 3} illustrates an example where \textbf{command group} objects are enqueued to two separate SYCL queues executing in devices in different contexts. The \textbf{requisites} for the \textbf{command group} execution are the same, but the \textbf{actions} to satisfy them are different. For example, if the data is on the host before execution, \texttt{A(b1\_rw)} and \texttt{A(b2\_rw)} can potentially be implemented as copy operations from the host memory to context1 or context2 respectively. After \texttt{CG_a} and \texttt{CG_b} are executed, \texttt{A'(b1\_rw)} will likely be an empty operation, since the result of the kernel can stay on the device. On the other hand, the results of \texttt{CG_b} are now on a different context than \texttt{CG_c} is executing, therefore \texttt{A'(b2\_rw)} will need to copy data across two separate contexts using an implementation specific mechanism.

\textbf{Table 3. Actions performed when three command groups are submitted to two distinct queues}

<table>
<thead>
<tr>
<th>SYCL Application Enqueue Order</th>
<th>SYCL Kernel Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sycl::queue q1(context1);</code> <code>sycl::queue q2(context2);</code> <code>q1.submit(CG_a(b1\_rw));</code> <code>q2.submit(CG_b(b2\_rw));</code> <code>q1.submit(CG_c(b1\_rw,b2\_rw));</code></td>
<td><code>CG_a(b1\_rw)</code> <code>A(b1\_rw)</code> <code>CG_b(b2\_rw)</code> <code>A'(b1\_rw)</code> <code>CG_c(b1\_rw,b2\_rw)</code></td>
</tr>
</tbody>
</table>

\textbf{Possible implementation by a SYCL Runtime}

```
clEnqueueWriteBuffer(q1, ptr, b1, c1);
clEnqueueWriteBuffer(q2, ptr, b2, c2);
clEnqueueReadBuffer(q2, b2, c2, ptr);
clEnqueueReadBuffer(q1, b2, c1, ptr);
```

\textbf{Table 3} shows actions performed when three command groups are submitted to two distinct queues, and potential implementation in an OpenCL SYCL \textbf{backend} by a SYCL runtime. Note that in this example, each SYCL buffer (\texttt{b2,b2}) is implemented as separate cl\_mem objects per context.

Note that the order of the definition of the accessors within the \textbf{command group} is irrelevant to the requirements they define. All accessors always apply to the entire \textbf{command group} object where they are defined.

When multiple \textbf{accessors} in the same \textbf{command group} define different requisites to the same memory object these requisites must be resolved.

Firstly, any requisites with different access modes but the same access target are resolved into a single requisite with the union of the different access modes according to \textbf{Table 4}. The atomic access mode acts as if it was read-write (RW) when determining the combined requirement. The rules in \textbf{Table 4} are commutative and associative.

\textbf{Table 4. Combined requirement from two different accessor access modes within the same \textbf{command group}. The rules are commutative and associative}
The result of this should be that there should not be any requisites with the same access target.

Secondly, the remaining requisites must adhere to the following rule. Only one of the requisites may have write access (\(W\) or \(RW\)), otherwise the \textbf{SYCL runtime} must throw an exception. All requisites create a requirement for the data they represent to be made available in the specified access target, however only the requisite with write access determines the side effects of the \textbf{command group}, i.e. only the data which that requisite represents will be updated.

For example:

- \(CG(b1^G_{RW}, b1^H_{RW})\) is permitted.
- \(CG(b1^G_{RW}, b1^H_{RW})\) is \textbf{not} permitted.
- \(CG(b1^G_W, b1^C_{RW})\) is \textbf{not} permitted.

Where \(G\) and \(C\) correspond to a \texttt{target::device} and \texttt{target::constant_buffer} accessor and \(H\) corresponds to a host accessor.

A buffer created from a range of an existing buffer is called a sub-buffer. A buffer may be overlaid with any number of sub-buffers. Accessors can be created to operate on these sub-buffers. Refer to Section 4.7.2 for details on sub-buffer creation and restrictions. A requirement to access a sub-buffer is represented by specifying its range, e.g. \(CG(b1_{RW,[0,5)})\) represents the requirement of accessing the range \([0,5)\) buffer \(b1\) in read write mode.

If two accessors are constructed to access the same buffer, but both are to non-overlapping sub-buffers of the buffer, then the two accessors are said to not overlap, otherwise the accessors do overlap. Overlapping is the test that is used to determine the scheduling order of command groups. Command-groups with non-overlapping requirements may execute concurrently.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{SYCL Application Enqueue Order} & \textbf{SYCL Kernel Execution Order} \\
\hline
\begin{itemize}
\item sycl::queue q1(context1);
\item q1.submit(CG\(a\)(b1\(b\)_{RW,\([0,10)\)}));
\item q1.submit(CG\(a\)(b1\(b\)_{RW,\([10,20)\)}));
\item q1.submit(CG\(a\)(b1\(b\)\_{RW,\([5,15)\)}));
\end{itemize}
& \begin{itemize}
\item CG\(a\)\(b\_{RW,\([0,10)\)}\)
\item CG\(a\)\(b\_{RW,\([10,20)\)}\)
\item CG\(a\)\(b\_{RW,\([5,15)\)}\)
\end{itemize} \\
\hline
\end{tabular}
\end{table}

It is permissible for command groups that only read data to not copy that data back to the host or other devices after reading and for the runtime to maintain multiple read-only copies of the data on multiple...
A special case of requirement is the one defined by a **host accessor**. Host accessors are represented with
\( H(MemoryObject, AccessMode) \), e.g., \( H(b1_{\text{rw}}) \) represents a host accessor to \( b1 \) in read-write mode. Host accessors are a special type of accessor constructed from a memory object outside a command group, and require that the data associated with the given memory object is available on the host in the given pointer. This causes the runtime to block on construction of this object until the requirement has been satisfied. **Host accessor** objects are effectively barriers on all accesses to a certain memory object. Table 6 shows an example of multiple command groups enqueued to the same queue. Once the host accessor \( H(b1_{\text{rw}}) \) is reached, the execution cannot proceed until \( CG_a \) is finished. However, \( CG_b \) does not have any requirements on \( b1 \), therefore, it can execute concurrently with the barrier. Finally, \( CG_c \) will be enqueued after \( H(b1_{\text{rw}}) \) is finished, but still has to wait for \( CG_b \) to conclude for all its requirements to be satisfied. See Section 3.9.8 for details on synchronization rules.

### Table 6. Execution of command groups when using host accessors

<table>
<thead>
<tr>
<th>SYCL Application Enqueue Order</th>
<th>SYCL Kernel Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>sycl::queue q1;</td>
<td>( CG_a(b1_{\text{rw}}) )</td>
</tr>
<tr>
<td>q1.submit(CG_a(b1_{\text{rw}}));</td>
<td></td>
</tr>
<tr>
<td>q1.submit(CG_b(b2_{\text{rw}}));</td>
<td>( CG_b(b2_{\text{rw}}) )</td>
</tr>
<tr>
<td>H(b1_{\text{rw}});</td>
<td>( H(b1_{\text{rw}}) )</td>
</tr>
<tr>
<td>q1.submit(CG_c(b1_{\text{rw}}, b2_{\text{rw}}));</td>
<td>( CG_c(b1_{\text{rw}}, b2_{\text{rw}}) )</td>
</tr>
</tbody>
</table>

### 3.8.2. SYCL device memory model

The memory model for SYCL devices is based on the OpenCL 1.2 memory model. Work-items executing in a kernel have access to three distinct address spaces (memory regions) and a virtual address space overlapping some concrete address spaces:

- **Global-memory** is accessible to all work-items in all work-groups. Work-items can read from or write to any element of a global memory object. Reads and writes to global memory may be cached depending on the capabilities of the device. Global memory is persistent across kernel invocations. Concurrent access to a location in an USM allocation by two or more executing kernels where at least one kernel modifies that location is a data race; there is no guarantee of correct results unless `mempfence` and atomic operations are used.

- **Local-memory** is accessible to all work-items in a single work-group. Attempting to access local memory in one work-group from another work group results in undefined behavior. This memory region can be used to allocate variables that are shared by all work-items in a work-group. Work-group-level visibility allows local memory to be implemented as dedicated regions of the device memory where this is appropriate.

- **Private-memory** is a region of memory private to a work-item. Attempting to access private memory in one work-item from another work-item results in undefined behavior.

- **Generic-memory** is a virtual address space which overlaps the global, local and private address spaces. Therefore, an object that resides in the global, local, or private address space can also be accessed through the generic address space.

#### 3.8.2.1. Access to memory

Accessors in the device kernels provide access to the memory objects, acting as pointers to the corresponding address space.

Pointers can be passed directly as kernel arguments if an implementation supports USM. See Section 4.8.
for information on when it is legal to dereference pointers passed from the host inside kernels.

To allocate local memory within a kernel, the user can either pass a `sycl::local_accessor` object as an argument to an ND-range kernel (that has a user-defined work-group size), or can define a variable in work-group scope inside `sycl::parallel_for_work_group`.

Any variable defined inside a `sycl::parallel_for` scope or `sycl::parallel_for_work_item` scope will be allocated in private memory. Any variable defined inside a `sycl::parallel_for_work_group` scope will be allocated in local memory.

Users can create accessors that reference sub-buffers as well as entire buffers.

Within kernels, the underlying C++ pointer types can be obtained from an accessor. The pointer types will contain a compile-time deduced address space. So, for example, if a C++ pointer is obtained from an accessor to global memory, the C++ pointer type will have a global address space attribute attached to it. The address space attribute will be compile-time propagated to other pointer values when one pointer is initialized to another pointer value using a defined algorithm.

When developers need to explicitly state the address space of a pointer value, one of the explicit pointer classes can be used. There is a different explicit pointer class for each address space: `sycl::raw_local_ptr`, `sycl::raw_global_ptr`, `sycl::raw_private_ptr`, `sycl::raw_generic_ptr`, `sycl::decorated_local_ptr`, `sycl::decorated_global_ptr`, `sycl::decorated_private_ptr`, or `sycl::decorated_generic_ptr`.

The classes with the `decorated` prefix expose pointers that use an implementation-defined address space decoration, while the classes with the `raw` prefix do not. Buffer accessors with an access target `target::device` or `target::constant_buffer` and local accessors can be converted into explicit pointer classes (`multi_ptr`).

For templates that need to adapt to different address spaces, a `sycl::multi_ptr` class is defined which is templated via a compile-time constant enumerator value to specify the address space.

### 3.8.3. SYCL memory consistency model

The SYCL memory consistency model is based upon the memory consistency model of the C++ core language. Where SYCL offers extensions to classes and functions that may affect memory consistency, the default behavior when these extensions are not used always matches the behavior of standard C++.

A SYCL implementation must guarantee that the same memory consistency model is used across host and device code. Every `device compiler` must support the memory model defined by the minimum version of C++ described in Section 3.9.1; SYCL implementations supporting additional versions of C++ must also support the corresponding memory models.

Within a work-item, operations are ordered according to the `sequenced before` relation defined by the C++ core language.

Ensuring memory consistency across different work-items requires careful usage of `group barrier` operations, `mem-fence` operations and atomic operations. The ordering of operations across different work-items is determined by the `happens before` relation defined by the C++ core language, with a single relation governing all address spaces (memory regions).

On any SYCL device, local and global memory may be made consistent across work-items in a single `group` through use of a `group barrier` operation. On SYCL devices supporting acquire-release or sequentially consistent memory orderings, all memory visible to a set of work-items may be made consistent across the work-items in that set through the use of `mem-fence` and atomic operations.

Memory consistency between the host and SYCL device(s), or different SYCL devices in the same context, can be guaranteed through synchronization in the host application as defined in Section 3.9.8. On SYCL devices supporting concurrent atomic accesses to USM allocations and acquire-release or sequentially
consistent memory orderings, cross-device memory consistency can be enforced through the use of
\texttt{mem-fence} and atomic operations.

### 3.8.3.1. Memory ordering

```cpp
namespace sycl {
enum class memory_order : /* unspecified */ {
    relaxed,
    acquire,
    release,
    acq_rel,
    seq_cst
};
inline constexpr auto memory_order_relaxed = memory_order::relaxed;
inline constexpr auto memory_order_acquire = memory_order::acquire;
inline constexpr auto memory_order_release = memory_order::release;
inline constexpr auto memory_order_acq_rel = memory_order::acq_rel;
inline constexpr auto memory_order_seq_cst = memory_order::seq_cst;
} // namespace sycl
```

The memory synchronization order of a given atomic operation is controlled by a \texttt{sycl::memory_order}
parameter, which can take one of the following values:

- \texttt{sycl::memory_order::relaxed};
- \texttt{sycl::memory_order::acquire};
- \texttt{sycl::memory_order::release};
- \texttt{sycl::memory_order::acq_rel};
- \texttt{sycl::memory_order::seq_cst}.

The meanings of these values are identical to those defined in the C++ core language.

The complete set of memory orders is not guaranteed to be supported by every device, nor across all
combinations of devices within a context. The memory orders supported by a specific device and context
can be queried using functionalities of the \texttt{sycl::device} and \texttt{sycl::context} classes, respectively.

SYCL implementations are not required to support a memory order equivalent to
\texttt{std::memory_order::consume}, and using this ordering within a SYCL device kernel results
in undefined behavior. Developers are encouraged to use \texttt{sycl::memory_order::acquire}
instead.

### 3.8.3.2. Memory scope

```cpp
namespace sycl {
enum class memory_scope : /* unspecified */ {
    work_item,
    sub_group,
    work_group,
    device,
    system
}
```
The set of *work items* and devices to which the memory ordering constraints of a given atomic operation apply is controlled by a `sycl::memory_scope` parameter, which can take one of the following values:

- `sycl::memory_scope::work_item` The ordering constraint applies only to the calling work-item;
- `sycl::memory_scope::sub_group` The ordering constraint applies only to work-items in the same sub-group as the calling work-item;
- `sycl::memory_scope::work_group` The ordering constraint applies only to work-items in the same work-group as the calling work-item;
- `sycl::memory_scope::device` The ordering constraint applies only to work-items executing on the same device as the calling work-item;
- `sycl::memory_scope::system` The ordering constraint applies to any work-item or host thread in the system that is currently permitted to access the memory allocation containing the referenced object, as defined by the capabilities of buffers and USM.

The broadest scope that can be applied to an atomic operation corresponds to the set of work-items which can access the associated memory location. For example, the broadest scope that can be applied to atomic operations in work-group local memory is `sycl::memory_scope::work_group`. If a broader scope is supplied, the behavior is as-if the narrowest scope containing all work-items which can access the associated memory location was supplied.

The addition of memory scopes to the C++ memory model modifies the definition of some concepts from the C++ core language. For example: data races, the synchronizes-with relationship and sequential consistency must be defined in a way that accounts for atomic operations with differing (but compatible) scopes, in a manner similar to the OpenCL 2.0 specification. Efforts to formalize the memory model of SYCL are ongoing, and a formal memory model will be included in a future version of the SYCL specification.

### 3.8.3.3. Atomic operations

Atomic operations can be performed on memory in buffers and USM. The `sycl::atomic_ref` class must be used to provide safe atomic access to the buffer or USM allocation from device code.

### 3.8.3.4. Forward progress

This section, and any subsequent section referring to progress guarantees, uses the following terms as defined in the C++ core language: thread of execution; weakly parallel forward progress guarantees; parallel forward progress guarantees; concurrent forward progress guarantees; and block with forward progress guarantee delegation.

Each work-item in SYCL is a separate thread of execution, providing at least weakly parallel forward progress guarantees. Whether work-items provide stronger forward progress guarantees is implementation-defined.

All implementations must additionally ensure that a work-item arriving at a `group barrier` does not pre-
vent other work-items in the same group from making progress. When a work-item arrives at a group barrier acting on group \( G \), implementations must eventually select and potentially strengthen another work-item in group \( G \) that has not yet arrived at the barrier.

When a host thread blocks on the completion of a command previously submitted to a SYCL queue (for example, via the \texttt{sycl::queue::wait} function), it blocks with forward progress guarantee delegation.

SYCL commands submitted to a queue are not guaranteed to begin executing until a host thread blocks on their completion. In the absence of multiple host threads, there is no guarantee that host and device code will execute concurrently.

### 3.9. The SYCL programming model

A SYCL program is written in standard C++. Host code and device code is written in the same C++ source file, enabling instantiation of templated kernels from host code and also enabling kernel source code to be shared between host and device. The device kernels are encapsulated C++ callable types (a function object with \texttt{operator()} or a lambda function), which have been designated to be compiled as SYCL kernels.

SYCL programs target heterogeneous systems. The kernels may be compiled and optimized for multiple different processor architectures with very different binary representations.

#### 3.9.1. Minimum version of C++

The C++ features used in SYCL are based on a specific version of C++. Implementations of SYCL must support this minimum C++ version, which defines the C++ constructs that can consequently be used by SYCL feature definitions (for example, lambdas).

The minimum C++ version of this SYCL specification is determined by the normative C++ core language defined in \textbf{Section 3.3}. All implementations of this specification must support at least this core language, and features within this specification are defined using features of the core language. Note that not all core language constructs are supported within \textit{SYCL kernel functions} or code invoked by a \textit{SYCL kernel function}, as detailed by \textbf{Section 5.4}.

Implementations may support newer C++ versions than the minimum required by SYCL. Code written using newer features than the SYCL requirement, though, may not be portable to other implementations that don’t support the same C++ version.

#### 3.9.2. Alignment with future versions of C++

Some features of SYCL are aligned with the next C++ specification, as defined in \textbf{Section 3.3}.

The following features are pre-adopted by SYCL 2020 and made available in the \texttt{sycl::} namespace: \texttt{std::span}, \texttt{std::dynamic_extent}, \texttt{std::bit_cast}. The implementations of pre-adopted features are compliant with the next C++ specification, and are expected to forward directly to standard C++ features in a future version of SYCL.

The following features of SYCL 2020 use syntax based on the next C++ specification: \texttt{sycl::atomic_ref}. These features behave as described in the next C++ specification, barring modifications to ensure compatibility with other SYCL 2020 features and heterogeneous programming. Any such modifications are documented in the corresponding sections of this specification.

#### 3.9.3. Basic data parallel kernels

Data-parallel kernels that execute as multiple work-items and where no local synchronization is required are enqueued with the \texttt{sycl::parallel_for} function parameterized by a \texttt{sycl::range} parameter.
These kernels will execute the kernel function body once for each work-item in the specified range.

Functionality tied to groups of work-items, including group barriers and local memory, must not be used within these kernels.

Variables with reduction semantics can be added to basic data parallel kernels using the features described in Section 4.9.2.

### 3.9.4. Work-group data parallel kernels

Data parallel kernels can also execute in a mode where the set of work-items is divided into work-groups of user-defined dimensions. The user specifies the global range and local work-group size as parameters to the `sycl::parallel_for` function with a `sycl::nd_range` parameter. In this mode of execution, kernels execute over the nd-range in work-groups of the specified size. It is possible to share data among work-items within the same work-group by calling the `group_barrier` function. All work-groups in a given `parallel_for` will be the same size, and the global size defined in the nd-range must either be a multiple of the work-group size in each dimension, or the global size must be zero. When the global size is zero, the kernel function is not executed, the local size is ignored, and any dependencies are satisfied.

Work-groups may be further subdivided into sub-groups. The work-items that compose a sub-group are selected in an implementation-defined way, and therefore the size and number of sub-groups may differ for each kernel. Moreover, different devices may make different guarantees with respect to how sub-groups within a work-group are scheduled. The maximum number of work-items in any sub-group in a kernel is based on a combination of the kernel and its dispatch dimensions. The size of any sub-group in the dispatch is between 1 and this maximum sub-group size, and the size of an individual sub-group is invariant for the duration of a kernel's execution. Similarly to work-groups, the work-items within the same sub-group can be synchronized by calling the `group_barrier` function.

Portable device code must not assume that work-items within a sub-group execute in any particular order, that work-groups are subdivided into sub-groups in a specific way, nor that the work-items within a sub-group provide specific forward progress guarantees.

Variables with reduction semantics can be added to work-group data parallel kernels using the features described in Section 4.9.2.

### 3.9.5. Hierarchical data parallel kernels

Based on developer and implementation feedback, the hierarchical data parallel kernel feature described next is undergoing improvements to better align with the frameworks and patterns prevalent in modern programming. As this is a key part of the SYCL API and we expect to make changes to it, we temporarily recommend that new codes refrain from using this feature until the new API is finished in a near-future version of the SYCL specification, when full use of the updated feature will be recommended for use in new code. Existing codes using this feature will of course be supported by conformant implementations of this specification.

The SYCL compiler provides a way of specifying data parallel kernels that execute within work-groups via a different syntax which highlights the hierarchical nature of the parallelism. This mode is purely a compiler feature and does not change the execution model of the kernel. Instead of calling `sycl::parallel_for` the user calls `sycl::parallel_for_work_group` with a `sycl::range` value representing the number of work-groups to launch and optionally a second `sycl::range` representing the size of each work-group for performance tuning. All code within the `parallel_for_work_group` scope effectively executes once per work-group. Within the `parallel_for_work_group` scope, it is possible to call `parallel_for_work_item` which creates a new scope in which all work-items within the current work-group execute. This enables a programmer to write code that looks like there is an inner work-item loop inside an outer work-group loop,
which closely matches the effect of the execution model. All variables declared inside the `parallel_for_work_group` scope are allocated in work-group local memory, whereas all variables declared inside the `parallel_for_work_item` scope are declared in private memory. All `parallel_for_work_item` calls within a given `parallel_for_work_group` execution must have the same dimensions.

3.9.6. Kernels that are not launched over parallel instances

Simple kernels for which only a single instance of the kernel function will be executed are enqueued with the `sycl::single_task` function. The kernel enqueued takes no “work-item id” parameter and will only execute once. The behavior is logically equivalent to executing a kernel on a single compute unit with a single work-group comprising only one work-item. Such kernels may be enqueued on multiple queues and devices and as a result may be executed in task-parallel fashion.

3.9.7. Pre-defined kernels

Some SYCL backends may expose pre-defined functionality to users as kernels. These kernels are not programmable, hence they are not bound by the SYCL C++ programming model restrictions, and how they are written is implementation-defined.

3.9.8. Synchronization

Synchronization of processing elements executing inside a device is handled by the SYCL device kernel following the SYCL kernel execution model. The synchronization of the different SYCL device kernels executing with the host memory is handled by the SYCL application via the SYCL runtime.

3.9.8.1. Synchronization in the SYCL application

Synchronization points between host and device(s) are exposed through the following operations:

- **Buffer destruction**: The destructors for `sycl::buffer`, `sycl::unsampled_image` and `sycl::sampled_image` objects wait for all submitted work on those objects to complete and to copy the data back to host memory before returning. These destructors only wait if the object was constructed with attached host memory and if data needs to be copied back to the host.

  More complex forms of synchronization on buffer destruction can be specified by the user by constructing buffers with other kinds of references to memory, such as `shared_ptr` and `unique_ptr`.

- **Host Accessors**: The constructor for a host accessor waits for all kernels that modify the same buffer (or image) in any queues to complete and then copies data back to host memory before the constructor returns. Any command groups with requirements to the same memory object cannot execute until the host accessor is destroyed as shown on Table 6.

- **Command group enqueue**: The SYCL runtime internally ensures that any command groups added to queues have the correct event dependencies added to those queues to ensure correct operation. Adding command groups to queues never blocks. Instead any required synchronization is added to the queue and events of type `sycl::event` are returned by the queue’s submit function that contain event information related to the specific command group.

- **Queue operations**: The user can manually use queue operations, such as `sycl::queue::wait()` to block execution of the calling thread until all the command groups submitted to the queue have finished execution. Note that this will also affect the dependencies of those command groups in other queues.

- **SYCL event objects**: SYCL provides `sycl::event` objects which can be used for synchronization. If synchronization is required across SYCL contexts from different SYCL backends, then the SYCL runtime ensures that extra host-based synchronization is added to enable the SYCL event objects to operate between contexts correctly.

Note that the destructors of other SYCL objects (`sycl::queue`, `sycl::context`,...) do not block. Only a `sycl::buffer`, `sycl::sampled_image` or `sycl::unsampled_image` destructor might block. The rationale is that
an object without any side effect on the host does not need to block on destruction as it would impact the performance. So it is up to the programmer to use a member function to wait for completion in some cases if this does not fit the goal. See Section 3.9.12 for more information on object life time.

3.9.8.2. Synchronization in SYCL kernels

In SYCL, synchronization can be either global or local within a group of work-items. Synchronization between work-items in a single group is achieved using a group barrier.

All the work-items of a group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the group barrier must be encountered by all work-items of a group executing the kernel or by none at all. In SYCL, work-group barrier and sub-group barrier functionality is exposed via the group_barrier function.

Synchronization between work-items in different work-groups via atomic operations is possible only on SYCL devices with certain capabilities, as described in Section 3.8.3.

3.9.9. Error handling

In SYCL, there are two types of errors: synchronous errors that can be detected immediately when an API call is made, and asynchronous errors that can only be detected later after an API call has returned. Synchronous errors, such as failure to construct an object, are reported immediately by the runtime throwing an exception. Asynchronous errors, such as an error occurring during execution of a kernel on a device, are reported via an asynchronous error-handler mechanism.

Asynchronous errors are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a sycl::exception_list object, which contains a list of asynchronously-generated exception objects, on the conditions described by Section 4.13.1.1 and Section 4.13.1.2.

Asynchronous errors may be generated regardless of whether the user has specified any asynchronous error handler(s), as described in Section 4.13.1.2.

Some SYCL backends can report errors that are specific to the platform they are targeting, or that are more concrete than the errors provided by the SYCL API. Any error reported by a SYCL backend must derive from the base sycl::exception. When a user wishes to capture specifically an error thrown by a SYCL backend, she must include the SYCL backend-specific headers for said SYCL backend.

3.9.10. Fallback mechanism

A command group function object can be submitted either to a single queue to be executed on, or to a secondary queue. If a command group function object fails to be enqueued to the primary queue, then the system will attempt to enqueue it to the secondary queue, if given as a parameter to the submit function. If the command group function object fails to be queued to both of these queues, then a synchronous SYCL exception will be thrown.

It is possible that a command group may be successfully enqueued, but then asynchronously fail to run, for some reason. In this case, it may be possible for the runtime system to execute the command group function object on the secondary queue, instead of the primary queue. The situations where a SYCL runtime may be able to achieve this asynchronous fall-back is implementation-defined.

3.9.11. Scheduling of kernels and data movement

A command group function object takes a reference to a command group handler as a parameter and anything within that scope is immediately executed and takes the handler object as a parameter. The intention is that a user will perform calls to SYCL functions, member functions, destructors and constructors inside that scope. These calls will be non-blocking on the host, but enqueue operations to the queue that the command group is submitted to. All user functions within the command group scope will
be called on the host as the command group function object is executed, but any commands it invokes will be added to the SYCL queue. All commands added to the queue will be executed out-of-order from each other, according to their data dependencies.

### 3.9.12. Managing object lifetimes

A SYCL application does not initialize any SYCL backend features until a sycl::context object is created. A user does not need to explicitly create a sycl::context object, but they do need to explicitly create a sycl::queue object, for which a sycl::context object will be implicitly created if not provided by the user.

All SYCL backend objects encapsulated in SYCL objects are reference-counted and will be destroyed once all references have been released. This means that a user needs only create a SYCL queue (which will automatically create an SYCL context) for the lifetime of their application to initialize and release any SYCL backend objects safely.

There is no global state specified to be required in SYCL implementations. This means, for example, that if the user creates two queues without explicitly constructing a common context, then a SYCL implementation does not have to create a shared context for the two queues. Implementations are free to share or cache state globally for performance, but it is not required.

Memory objects can be constructed with or without attached host memory. If no host memory is attached at the point of construction, then destruction of that memory object is non-blocking. The user may use C++ standard pointer classes for sharing the host data with the user application and for defining blocking, or non-blocking behavior of the buffers and images. If host memory is attached by using a raw pointer, then the default behavior is followed, which is that the destructor will block until any command groups operating on the memory object have completed, then, if the contents of the memory object is modified on a device those contents are copied back to host and only then does the destructor return.

In the case where host memory is shared between the user application and the SYCL runtime with a std::shared_ptr, then the reference counter of the std::shared_ptr determines whether the buffer needs to copy data back on destruction, and in that case the blocking or non-blocking behavior depends on the user application.

Instead of a std::shared_ptr, a std::unique_ptr may be provided, which uses move semantics for initializing and using the associated host memory. In this case, the behavior of the buffer in relation to the user application will be non-blocking on destruction.

As said in Section 3.9.8, the only blocking operations in SYCL (apart from explicit wait operations) are:

- host accessor constructor, which waits for any kernels enqueued before its creation that write to the corresponding object to finish and be copied back to host memory before it starts processing. The host accessor does not necessarily copy back to the same host memory as initially given by the user;
- memory object destruction, in the case where copies back to host memory have to be done or when the host memory is used as a backing-store.

### 3.9.13. Device discovery and selection

A user specifies which queue to submit a command group function object and each queue is targeted to run on a specific device (and context). A user can specify the actual device on queue creation, or they can specify a device selector which causes the SYCL runtime to choose a device based on the user’s provided preferences. Specifying a device selector causes the SYCL runtime to perform device discovery. No device discovery is performed until a SYCL device selector is passed to a queue constructor. Device topology may be cached by the SYCL runtime, but this is not required.

Device discovery will return all devices from all platforms exposed by all the supported SYCL backends.
3.9.14. Interfacing with the SYCL backend API

There are two styles of developing a SYCL application:

1. writing a pure SYCL generic application;
2. writing a SYCL application that relies on some SYCL backend specific behavior.

When users follow 1., there is no assumption about what SYCL backend will be used during compilation or execution of the SYCL application. Therefore, the SYCL backend API is not assumed to be available to the developer. Only standard C++ types and interfaces are assumed to be available, as described in Section 3.9. Users only need to include the `<sycl/sycl.hpp>` header to write a SYCL generic application.

On the other hand, when users follow 2., they must know what SYCL backend APIs they are using. In this case, any header required for the normal programmability of the SYCL backend API is assumed to be available to the user. In addition to the `<sycl/sycl.hpp>` header, users must also include the SYCL backend-specific header as defined in Section 4.3. The SYCL backend-specific header provides the interoperability interface for the SYCL API to interact with native backend objects.

The interoperability API is defined in Section 4.5.1.

3.10. Memory objects

SYCL memory objects represent data that is handled by the SYCL runtime and can represent allocations in one or multiple devices at any time. Memory objects, both buffers and images, may have one or more underlying native backend objects to ensure that queues objects can use data in any device. A SYCL implementation may have multiple native backend objects for the same device. The SYCL runtime is responsible for ensuring the different copies are up-to-date whenever necessary, using whatever mechanism is available in the system to update the copies of the underlying native backend objects.

A valid mechanism for this update is to transfer the data from one SYCL backend into the system memory using the SYCL backend-specific mechanism available, and then transfer it to a different device using the mechanism exposed by the new SYCL backend.

Memory objects in SYCL fall into one of two categories: buffer objects and image objects. A buffer object stores a one-, two- or three-dimensional collection of elements that are stored linearly directly back to back in the same way C or C++ stores arrays. An image object is used to store a one-, two- or three-dimensional texture, frame-buffer or image data that may be stored in an optimized and device-specific format in memory and must be accessed through specialized operations.

Elements of a buffer object can be a scalar data type (such as an int or float), vector data type, or a user-defined structure. In SYCL, a buffer object is a templated type (`sycl::buffer`), parameterized by the element type and number of dimensions. An image object is stored in one of a limited number of formats. The elements of an image object are selected from a list of predefined image formats which are provided by an underlying SYCL backend implementation. Images are encapsulated in the `sycl::unsampled_image` or `sycl::sampled_image` types, which are templated by the number of dimensions in the image. The minimum number of elements in an image object is one. The minimum number of elements in a buffer object is zero.

The fundamental differences between a buffer and an image object are:

- elements in a buffer are stored in an array of 1, 2 or 3 dimensions and can be accessed using an accessor by a kernel executing on a device. The accessors for kernels provide a member function to get C++ pointer types, or the `sycl::global_ptr` class;
- elements of an image are stored in a format that is opaque to the user and cannot be directly accessed using a pointer. SYCL provides image accessors and samplers to allow a kernel to read from
3.11. Multi-dimensional objects and linearization

SYCL defines a number of multi-dimensional objects such as buffers and accessors. The iteration space of work-items in a kernel may also be multi-dimensional. The size of each dimension is defined by a range object of one, two or three dimensions, and an element in the multi-dimensional space can be identified using an id object with the same number of dimensions as the corresponding range.

If the size of any dimension is zero, there are zero elements in the multi-dimensional range.

3.11.1. Linearization

Some multi-dimensional objects can be viewed in a linear form. When this happens, the right-most term in the object’s range varies fastest in the linearization.

A three-dimensional element \( \text{id}(\text{id}_0, \text{id}_1, \text{id}_2) \) within a three-dimensional object of range \( \text{range}(r_0, r_1, r_2) \) has a linear position defined by:

\[
\text{id}_2 + (\text{id}_1 \cdot r_2) + (\text{id}_0 \cdot r_1 \cdot r_2)
\]

A two-dimensional element \( \text{id}(\text{id}_0, \text{id}_1) \) within a two-dimensional \( \text{range}(r_0, r_1) \) follows a similar equation:

\[
\text{id}_1 + (\text{id}_0 \cdot r_1)
\]

A one-dimensional element \( \text{id}(\text{id}_0) \) within a one-dimensional range \( \text{range}(r_0) \) is equivalent to its linear form.

3.11.2. Multi-dimensional subscript operators

Some multi-dimensional objects can be indexed using the subscript operator where consecutive subscript operators correspond to each dimension. The right-most operator varies fastest, as with standard C++ arrays. Formally, a three-dimensional subscript access \( a[\text{id}_0][\text{id}_1][\text{id}_2] \) references the element at \( \text{id}(\text{id}_0, \text{id}_1, \text{id}_2) \). A two-dimensional subscript access \( a[\text{id}_0][\text{id}_1] \) references the element at \( \text{id}(\text{id}_0, \text{id}_1) \). A one-dimensional subscript access \( a[\text{id}_0] \) references the element at \( \text{id}(\text{id}_0) \).
3.12. Implementation options

The SYCL language is designed to allow several different possible implementations. The contents of this section are non-normative, so implementations need not follow the guidelines listed here. However, this section is intended to help readers understand the possible strategies that can be used to implement SYCL.

3.12.1. Single source multiple compiler passes

With this technique, known as SMCP, there are separate host and device compilers. Each SYCL source file is compiled two times: once by the host compiler and once by the device compiler. An implementation could support more than one device compiler, in which case each SYCL source file is compiled more than two times. The host compiler in this technique could be an off-the-shelf compiler with no special knowledge of SYCL, but the device compiler must be SYCL aware. The device compiler parses the source file to identify each SYCL kernel function and any device functions it calls. SYCL is designed so that this analysis can be done statically. The device compiler then generates code only for the SYCL kernel functions and the device functions.

Typically, the device compilers generate header files which interface between the host compiler and the SYCL runtime. Therefore, the device compiler runs first, and then the host compiler consumes these header files when generating the host code.

The device compilers in this technique generate one or more device images for the SYCL kernel functions, which can be read by the SYCL runtime. Each device image could either contain native ISA for a device or it could contain an intermediate language such as SPIR-V. In the later case, the SYCL runtime must translate the intermediate language into native device ISA when the SYCL kernel function is submitted to a device.

Since this technique has separate host and device compilers, there needs to be some way to associate a SYCL kernel function (which is compiled by the device compiler) with the code that invokes it (which is compiled by the host compiler). Implementations conformant to the reduced feature set (Section B.2) can do this by using the C++ type of the SYCL kernel function. This type is specified via the kernel name template parameter if the SYCL kernel function is a lambda function, or it is obtained from the class type if the SYCL kernel function is an object. Implementations conformant to the full feature set (Section B.1) do not require a kernel name at the invocation site, so they must implement some other way to make the association.

3.12.2. Single source single compiler pass

With this technique, known as SSCP, the vendor implements a custom compiler that reads each SYCL source file only once, and that compiler generates the host code as well as the device images for the SYCL kernel functions. As in the SMCP case, each device image could either contain native device ISA or an intermediate language.

3.12.3. Library-only implementation

It is also possible to implement SYCL purely as a library, using an off-the-shelf host compiler with no special support for SYCL. In such an implementation, each kernel may run on the host system.

3.13. Language restrictions in kernels

The SYCL kernels are executed on SYCL devices and all of the functions called from a SYCL kernel are going to be compiled for the device by a SYCL device compiler. Due to restrictions of the heterogeneous devices where the SYCL kernel will execute, there are certain restrictions on the base C++ language features that can be used inside kernel code. For details on language restrictions please refer to Section 5.4.
SYCL kernels use arguments that are captured by value in the command group scope or are passed from the host to the device using accessors. Sharing data structures between host and device code imposes certain restrictions, such as using only objects that are device copyable, and in general, no pointers initialized for the host can be used on the device. SYCL memory objects, such as `sycl::buffer`, `sycl::unsampled_image`, and `sycl::sampled_image`, cannot be passed to a kernel. Instead, a kernel must interact with these objects through accessors. No hierarchical structures of these memory object classes are supported and any other data containers need to be converted to the SYCL data management classes using the SYCL interface. For more details on the rules for kernel parameter passing, please refer to Section 4.12.4.

Pointers to USM allocations may be passed to a kernel either directly as arguments or indirectly inside of other objects. Pointers to USM allocations that are passed as kernel arguments are treated as being in the global address space.

### 3.13.1. Device copyable

The SYCL implementation may need to copy data between the host and a device or between two devices. For example, this may occur when a command group has a requirement for the contents of a buffer or when the application passes certain arguments to a SYCL kernel function (as described in Section 4.12.4). Such data must have a type that is device copyable as defined below.

Any type that is trivially copyable (as defined by the C++ core language) is implicitly device copyable.

Although implementations are not required to support device code that calls library functions from the C++ core language, some implementations may provide device support for some of these functions. If the implementation provides device support for one of the following classes, that type is also implicitly device copyable:

- `std::array<T, 0>`;
- `std::array<T, N>` if `T` is device copyable;
- `std::optional<T>` if `T` is device copyable;
- `std::pair<T1, T2>` if `T1` and `T2` are device copyable;
- `std::tuple<>`;
- `std::tuple<Types...>` if all the types in the parameter pack `Types` are device copyable;
- `std::variant<>`;
- `std::variant<Types...>` if all the types in the parameter pack `Types` are device copyable;
- `std::basic_string_view<CharT, Traits>`;
- `std::span<ElementType, Extent>` (the `std::span` type has been introduced in C++20);
- `sycl::span<ElementType, Extent>`.

If the implementation provides device support for one of the classes listed above, arrays of that class and cv-qualified versions of that class are also device copyable.

The types `std::basic_string_view<CharT, Traits>` and `std::span<ElementType, Extent>` are both view types, which reference underlying data that is not contained within their type. Although these view types are device copyable, the implementation copies just the view and not the contained data when doing an inter-device copy. In order to reference the contained data after such a copy, the application must allocate the contained data in unified shared memory (USM) that is accessible on both the host and device (or on both devices in the case of a device-to-device copy).

In addition, the implementation may allow the application to explicitly declare certain class types as device copyable. If the implementation has this support, it must predefine the preprocessor macro `SYCL_DEVICE_COPYABLE` to 1, and it must not predefine this preprocessor macro if it does not have this sup-
port. When the implementation has this support, a class type $T$ is device copyable if all of the following statements are true:

- The application defines the trait `is_device_copyable_v<T>` to `true`;
- Type $T$ has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator;
- Each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is `public`;
- When doing an inter-device transfer of an object of type $T$, the effect of each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is the same as a bitwise copy of the object;
- Type $T$ has a `public` non-deleted destructor;
- The destructor has no effect when executed on the device.

When the application explicitly declares a class type to be device copyable, arrays of that type and cv-qualified versions of that type are also device copyable, and the implementation sets the `is_device_copyable_v` trait to `true` for these array and cv-qualified types.

It is unspecified whether the implementation actually calls the copy constructor, move constructor, copy assignment operator, or move assignment operator of a class declared as `is_device_copyable_v` when doing an inter-device copy. Since these operations must all be the same as a bitwise copy, the implementation may simply copy the memory where the object resides. Likewise, it is unspecified whether the implementation actually calls the destructor for such a class on the device since the destructor must have no effect on the device.

### 3.14. Endianness support

SYCL does not mandate any particular byte order, but the byte order of the host always matches the byte order of the devices. This allows data to be copied between the host and the devices without any byte swapping.

### 3.15. Example SYCL application

Below is a more complex example application, combining some of the features described above.

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

// Size of the matrices
constexpr size_t N = 2000;
constexpr size_t M = 3000;

int main() {
  // Create a queue to work on
  queue myQueue;

  // Create some 2D buffers of float for our matrices
  buffer<float, 2> a { range<2> { N, M } };
  buffer<float, 2> b { range<2> { N, M } };
  buffer<float, 2> c { range<2> { N, M } };
```
// Launch an asynchronous kernel to initialize a
myQueue.submit([&](handler & cgh) {
    // The kernel writes a, so get a write accessor on it
    accessor A { a, cgh, write_only };

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
cgh.parallel_for(ranges<2> { N, M }, [=](id<2> index) {
    A[index] = index[0] * 2 + index[1];
});

    // Launch an asynchronous kernel to initialize b
myQueue.submit([&](handler & cgh) {
    accessor B { b, cgh, write_only };

    // From the access pattern above, the SYCL runtime detects that this
    // command_group is independent from the first one and can be
    // scheduled independently
    accessor C { c, cgh, write_only };

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
cgh.parallel_for(ranges<2> { N, M }, [=](id<2> index) {
});

    // Launch an asynchronous kernel to compute matrix addition c = a + b
myQueue.submit([&](handler & cgh) {
    accessor A { a, cgh, read_only };
    accessor B { b, cgh, read_only };
    accessor C { c, cgh, write_only };

    // From these accessors, the SYCL runtime will ensure that when
    // this kernel is run, the kernels computing a and b have completed
    accessor D { d, cgh, read_only };

    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
cgh.parallel_for(ranges<2> { N, M }, [=](id<2> index) {
});

    // Ask for an accessor to read c from application scope. The SYCL runtime
    // waits for c to be ready before returning from the constructor
    host_accessor C { c, read_only };
    std::cout << std::endl << "Result:" << std::endl;
    for (size_t i = 0; i < N; i++) {
        for (size_t j = 0; j < M; j++) {
            // Compare the result to the analytic value
            if (C[i][j] != i * (2 + 2014) + j * (1 + 42)) {
                std::cout << "Wrong value " << C[i][j] << " on element " << i << " "
                          << j << std::endl;
                exit(-1);
            }
        }
    }

    std::cout << "Good computation!" << std::endl;
});
return 0;
}

Chapter 4. SYCL programming interface

The SYCL programming interface provides a common abstracted feature set to one or more SYCL backends APIs. This section describes the C++ library interface to the SYCL runtime which executes across those SYCL backends.

The entirety of the SYCL interface defined in this section is required to be available for any SYCL backends, with the exception of the interoperability interface, which is described in general terms in this document, not pertaining to any particular SYCL backend.

SYCL guarantees that all the member functions and special member functions of the SYCL classes described are thread safe.

The underlying types for all enumerations defined in this specification are implementation-defined. In addition, all enumerators within an enumeration have some implementation-defined unique value unless the specification specifically indicates a values for the enumerator.

4.1. Backends

The SYCL backends that can be supported by a SYCL implementation are identified using the enum class backend.

```cpp
namespace sycl {
  enum class backend : /* unspecified */ {
    /* see below */
  };
} // namespace sycl
```

The enum class backend is implementation-defined and must be populated with a unique identifier for each SYCL backend that the SYCL implementation can support. Note that the SYCL backends listed in the enum class backend are not guaranteed to be available in a given installation.

Each named SYCL backend enumerated in the enum class backend must be associated with a SYCL backend specification. Many sections of this specification will refer to the associated SYCL backend specification.

4.1.1. Backend macros

As the identifiers defined in enum class backend are implementation-defined, and the associated backends not guaranteed to be available, a SYCL implementation must also define a preprocessor macro for each of these identifiers. If the SYCL backend is defined by the Khronos SYCL group, the name of the macro has the form SYCL_BACKEND_<backend_name>, where backend_name is the associated identifier from backend in all upper-case. See Chapter 6 for the name of the macro if the vendor defines the SYCL backend outside of the Khronos SYCL group.

If a backend listed in the enum class backend is not available, the associated macro must be left undefined.

4.2. Generic vs non-generic SYCL

The SYCL programming API is split into two categories; generic SYCL and non-generic SYCL. Almost everything in the SYCL programming API is considered generic SYCL. However any usage of the enum class backend is considered non-generic SYCL and should only be used for SYCL backend specialized code paths, as the identifiers defined in backend are implementation-defined.
In any non-generic SYCL application code where the `backend` enum class is used, the expression must be guarded with a preprocessor `#ifdef` guard using the associated preprocessor macro to ensure that the SYCL application will compile even if the SYCL implementation does not support that `SYCL backend` being specialized for.

### 4.3. Header files and namespaces

SYCL provides one standard header file: `<sycl/sycl.hpp>`, which needs to be included in every translation unit that uses the SYCL programming API.

All SYCL classes, constants, types and functions defined by this specification should exist within the `::sycl` namespace.

For compatibility with SYCL 1.2.1, SYCL provides another standard header file: `<CL/sycl.hpp>`, which can be included in place of `<sycl/sycl.hpp>`. In that case, all SYCL classes, constants, types and functions defined by this specification should exist within the `::cl::sycl` namespace.

For consistency, the programming API will only refer to the `<sycl/sycl.hpp>` header and the `::sycl` namespace, but this should be considered synonymous with the SYCL 1.2.1 header and namespace.

Include paths starting with "sycl/ext/" and "sycl/backend/" are reserved for extensions to SYCL and for backend interop headers respectively. Other include paths starting with "sycl/" and the `sycl::detail` namespace are reserved for implementation details.

When a `SYCL backend` is defined by the Khronos SYCL group, functionality for that `SYCL backend` is available via the header "sycl/backend/<backend_name>.hpp", and all `SYCL backend`-specific functionality is made available in the namespace `sycl::<backend_name>` where `<backend_name>` is the name of the `SYCL backend` as defined in the `SYCL backend` specification.

Chapter 6 defines the allowable header files and namespaces for any extensions that a vendor may provide, including any `SYCL backend` that the vendor may define outside of the Khronos SYCL group.

Unless otherwise specified, the behavior of a SYCL program is undefined if it adds any entity to namespace `sycl` or to a namespace within namespace `sycl`.

### 4.4. Class availability

In SYCL some `SYCL runtime` classes are available to the SYCL application, some are available within a `SYCL kernel function` and some are available on both and can be passed as arguments to a `SYCL kernel function`.

Each of the following `SYCL runtime` classes: `buffer, buffer_allocator, context, device, device_image, event, exception, handler, host_accessor, host_sampled_image_accessor, host_unsampled_image_accessor, id, image_allocator, kernel, kernel_id, marray, kernel_bundle, nd_range, platform, queue, range, sampled_image, image_sampler, stream, unsampled_image` and `vec` must be available to the host application.

Each of the following `SYCL runtime` classes: `accessor, atomic_ref, device_event, group, h_item, id, item, local_accessor, marray, multi_ptr, nd_item, range, reducer, sampled_image_accessor, stream, sub_group, unsampled_image_accessor` and `vec` must be available within a `SYCL kernel function`.

### 4.5. Common interface

When a dimension template parameter is used in SYCL classes, it is defaulted as 1 in most cases.
4.5.1. Backend interoperability

Many of the SYCL runtime classes may be implemented such that they encapsulate an object unique to the SYCL backend that underpins the functionality of that class. Where appropriate, these classes may provide an interface for interoperating between the SYCL runtime object and the native backend object in order to support interoperability within an application between SYCL and the associated SYCL backend API.

There are three forms of interoperability with SYCL runtime classes: interoperability on the SYCL application with the SYCL backend API, interoperability within a SYCL kernel function with the equivalent kernel language types of the SYCL backend, and interoperability within a host task with the interop_handle.

SYCL application interoperability, SYCL kernel function interoperability and host task interoperability are provided via different interfaces and may have different behavior for the same SYCL object.

SYCL application interoperability may be provided for buffer, context, device, device_image, event, kernel, kernel_bundle, platform, queue, sampled_image, and unsampled_image.

SYCL kernel function interoperability may be provided for accessor, device_event, local_accessor, sampled_image_accessor, stream and unsampled_image_accessor inside kernel scope only and is not available outside of that scope.

host task interoperability may be provided for accessor, sampled_image_accessor, unsampled_image_accessor, queue, device, context inside the scope of a host task only, see Section 4.10.

Support for SYCL backend interoperability is optional and therefore not required to be provided by a SYCL implementation. A SYCL application using SYCL backend interoperability is considered to be non-generic SYCL.

Details on the interoperability for a given SYCL backend are available on the SYCL backend specification document for that SYCL backend.

4.5.1.1. Type traits backend_traits

```cpp
namespace sycl {

  template <backend Backend> class backend_traits {
    public:
      template <class T> using input_type = /* see below */;
      template <class T> using return_type = /* see below */;
      using errc = /* see below */;
  };

  template <backend Backend, typename SyclType>
  using backend_input_t = typename backend_traits<Backend>::template input_type<SyclType>;

  template <backend Backend, typename SyclType>
  using backend_return_t = typename backend_traits<Backend>::template return_type<SyclType>;

} // namespace sycl
```
A series of type traits are provided for SYCL backend interoperability, defined in the `backend_traits` class.

A specialization of `backend_traits` must be provided for each named SYCL backend enumerated in the enum class `backend` that is available at compile time.

- For each SYCL runtime class `T` which supports SYCL application interoperability with the SYCL backend, a specialization of `input_type` must be defined as the type of SYCL application interoperability native backend object associated with `T` for the SYCL backend, specified in the SYCL backend specification. `input_type` is used when constructing SYCL objects from backend specific native objects. See the relevant backend specification for details.

- For each SYCL runtime class `T` which supports SYCL application interoperability with the SYCL backend, a specialization of `return_type` must be defined as the type of SYCL application interoperability native backend object associated with `T` for the SYCL backend, specified in the SYCL backend specification. `return_type` is used when retrieving the backend specific native object from a SYCL object. See the relevant backend specification for details.

- For each SYCL runtime class `T` which supports kernel function interoperability with the SYCL backend, a specialization of `return_type` within `backend_traits` must be defined as the type of the kernel function interoperability native backend object associated with `T` for the SYCL backend, specified in the backend specification. See the relevant backend specification for details.

- A specialization of `errc` must be defined as the SYCL backend error code type.

The type alias `backend_input_t` is provided to enable less verbose access to the `input_type` type within `backend_traits` for a specific SYCL object of type `T`. The type alias `backend_return_t` is provided to enable less verbose access to the `return_type` type within `backend_traits` for a specific SYCL object of type `T`.

### 4.5.1.2. Template function `get_native`

```cpp
namespace sycl {

template <backend Backend, class T>
backend_return_t<Backend, T> get_native(const T& syclObject);

} // namespace sycl
```

For each SYCL runtime class `T` which supports SYCL application interoperability, a specialization of `get_-native` must be defined, which takes an instance of `T` and returns a SYCL application interoperability native backend object associated with `syclObject` which can be used for SYCL application interoperability. The lifetime of the object returned are backend-defined and specified in the backend specification.

For each SYCL runtime class `T` which supports kernel function interoperability, a specialization of `get_-native` must be defined, which takes an instance of `T` and returns the kernel function interoperability native backend object associated with `syclObject` which can be used for kernel function interoperability. The availability and behavior of these template functions is defined by the SYCL backend specification document.

The `get_native` function must throw an exception with the `errc::backend_mismatch` error code if the backend of the SYCL object doesn’t match the target backend.

### 4.5.1.3. Template functions `make_*`

```cpp
namespace sycl {

} // namespace sycl
```
4.5.1.3. Template functions

```c++
// platform
platform make_platform(const backend_input_t<Backend, platform>& backendObject);

template <backend Backend>
device make_device(const backend_input_t<Backend, device>& backendObject);

// context
template <backend Backend>
context make_context(const backend_input_t<Backend, context>& backendObject,
const async_handler asyncHandler = {});

template <backend Backend>
queue make_queue(const backend_input_t<Backend, queue>& backendObject,
const context& targetContext,
const async_handler asyncHandler = {});

template <backend Backend>
event make_event(const backend_input_t<Backend, event>& backendObject,
const context& targetContext);

// buffer
template <backend Backend, typename T, int Dimensions = 1,
typename AllocatorT = buffer_allocator<std::remove_const_t<T>>>
buffer<T, Dimensions, AllocatorT>
make_buffer(const backend_input_t<Backend, buffer<T, Dimensions, AllocatorT>>&
backendObject,
const context& targetContext, event availableEvent);

// sampled_image
template <backend Backend, int Dimensions = 1,
typename AllocatorT = sycl::image_allocator>
sampled_image<Dimensions, AllocatorT> make_sampled_image(
const backend_input_t<Backend, sampled_image<Dimensions, AllocatorT>>&
backendObject,
const context& targetContext, image_sampler imageSampler,
event availableEvent);

// unsampled_image
template <backend Backend, int Dimensions = 1,
typename AllocatorT = sycl::image_allocator>
unsampled_image<Dimensions, AllocatorT> make_unsampled_image(
const backend_input_t<Backend, unsampled_image<Dimensions, AllocatorT>>&
backendObject,
const context& targetContext, event availableEvent);

// device
template <backend Backend, int Dimensions = 1,
typename AllocatorT = sycl::image_allocator>
device make_device(const backend_input_t<Backend, device>& backendObject);
```

Chapter 4. SYCL programming interface
For each SYCL runtime class $T$ which supports SYCL application interoperability, a specialization of the appropriate template function `make_{sycl_class}` where `{sycl_class}` is the class name of $T$, must be defined, which takes a SYCL application interoperability native backend object and constructs and returns an instance of $T$. The availability and behavior of these template functions is defined by the SYCL backend specification document.

Overloads of the `make_{sycl_class}` function which take a SYCL context object as an argument must throw an exception with the `errc::backend_mismatch` error code if the backend of the provided SYCL context doesn't match the target backend.

### 4.5.2. Common reference semantics

Each of the following SYCL runtime classes: accessor, buffer, context, device, device_image, event, host_accessor, host_sampled_image_accessor, host_unsampled_image_accessor, kernel, kernel_id, kernel_bundle, local_accessor, platform, queue, sampled_image, sampled_image_accessor, stream, unsampled_image and unsampled_image_accessor must obey the following statements, where $T$ is the runtime class type:

- $T$ must be copy constructible and copy assignable on the host application and within SYCL kernel functions in the case that $T$ is a valid kernel argument. Any instance of $T$ that is constructed as a copy of another instance, via either the copy constructor or copy assignment operator, must behave as-if it were the original instance and as-if any action performed on it were also performed on the original instance and must represent the same underlying native backend object as the original instance where applicable.

- $T$ must be destructible on the host application and within SYCL kernel functions in the case that $T$ is a valid kernel argument. When any instance of $T$ is destroyed, including as a result of the copy assignment operator, any behavior specific to $T$ that is specified as performed on destruction is only performed if this instance is the last remaining host copy, in accordance with the above definition of a copy.

- $T$ must be move constructible and move assignable on the host application and within SYCL kernel functions in the case that $T$ is a valid kernel argument. Any instance of $T$ that is constructed as a move of another instance, via either the move constructor or move assignment operator, must replace the original instance rendering said instance invalid and must represent the same underlying native backend object as the original instance where applicable.

- $T$ must be equality comparable on the host application. Equality between two instances of $T$ (i.e. $a == b$) must be true if one instance is a copy of the other and non-equality between two instances of $T$ (i.e. $a != b$) must be true if neither instance is a copy of the other, in accordance with the above definition of a copy, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on $T$ must guarantee to be reflexive (i.e. $a == a$), symmetric (i.e. $a == b$
implies \( b == a \) and \( a != b \) implies \( b != a \) and transitive (i.e., \( a == b \implies b == c \) implies \( c == a \)).

- A specialization of `std::hash` for \( T \) must exist on the host application that returns a unique value such that if two instances of \( T \) are equal, in accordance with the above definition, then their resulting hash values are also equal and subsequently if two hash values are not equal, then their corresponding instances are also not equal, in accordance with the above definition.

Some SYCL runtime classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions in order to fulfill the copy, move, destruction requirements and hidden friend functions in order to fulfill the equality requirements.

A hidden friend function is a function first declared via a `friend` declaration with no additional out of class or namespace scope declarations. Hidden friend functions are only visible to ADL (Argument Dependent Lookup) and are hidden from qualified and unqualified lookup. Hidden friend functions have the benefits of avoiding accidental implicit conversions and faster compilation.

These common special member functions and hidden friend functions are described in Table 7 and Table 8 respectively.

```cpp
namespace sycl {

    class T {
    ...
    
    public : T(const T& rhs);

    T(T&& rhs);

    T& operator=(const T& rhs);

    T& operator=(T&& rhs);

    ~T();

    ...

    friend bool operator==(const T& lhs, const T& rhs) { /* ... */ }

    friend bool operator!=(const T& lhs, const T& rhs) { /* ... */ }

    ...

}; // namespace sycl
```

Table 7. Common special member functions for reference semantics

<table>
<thead>
<tr>
<th>Special member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T(const T&amp; rhs)</code></td>
<td>Constructs a ( T ) instance as a copy of the RHS SYCL ( T ) in accordance with the requirements set out above.</td>
</tr>
</tbody>
</table>
### Special member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructs a SYCL $T$ instance as a move of the RHS SYCL $T$ in accordance with the requirements set out above.</td>
</tr>
</tbody>
</table>

| $T$ $\&$ operator$\Rightarrow$(const $T$ rhs) |
| Assigns this SYCL $T$ instance with a copy of the RHS SYCL $T$ in accordance with the requirements set out above. |

| $T$ $\&$ operator$\Rightarrow$(T && rhs) |
| Assigns this SYCL $T$ instance with a move of the RHS SYCL $T$ in accordance with the requirements set out above. |

| $\neg T()$ |
| Destroys this SYCL $T$ instance in accordance with the requirements set out in Section 4.5.2. On destruction of the last copy, may perform additional lifetime related operations required for the underlying native backend object specified in the SYCL backend specification document, if this SYCL $T$ instance was originally constructed using one of the backend interoperability `make_*` functions specified in Section 4.5.1.3. See the relevant backend specification for details. |

### Table 8. Common hidden friend functions for reference semantics

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool operator$\Rightarrow=$(const $T$ lhs, const $T$ rhs)</td>
<td>Returns true if this LHS SYCL $T$ is equal to the RHS SYCL $T$ in accordance with the requirements set out above, otherwise returns false.</td>
</tr>
<tr>
<td>bool operator!$\Rightarrow=$(const $T$ lhs, const $T$ rhs)</td>
<td>Returns true if this LHS SYCL $T$ is not equal to the RHS SYCL $T$ in accordance with the requirements set out above, otherwise returns false.</td>
</tr>
</tbody>
</table>

### 4.5.3. Common by-value semantics

Each of the following SYCL runtime classes: `id`, `range`, `item`, `nd_item`, `h_item`, `group`, `sub_group` and `nd_range` must follow the following statements, where $T$ is the runtime class type:

- $T$ must be default copy constructible and copy assignable on the host application (in the case where $T$ is available on the host) and within SYCL kernel functions.
- $T$ must be default destructible on the host application (in the case where $T$ is available on the host) and within SYCL kernel functions.
- $T$ must be default move constructible and default move assignable on the host application (in the case where $T$ is available on the host) and within SYCL kernel functions.
- $T$ must be equality comparable on the host application (in the case where $T$ is available on the host) and within SYCL kernel functions. Equality between two instances of $T$ (i.e. $a == b$) must be true if the value of all members are equal and non-equality between two instances of $T$ (i.e. $a != b$) must be true if the value of any members are not equal, unless either instance has become invalidated by a move operation. By extension of the requirements above, equality on $T$ must guarantee to be reflexive (i.e. $a == a$), symmetric (i.e. $a == b$ implies $b == a$ and $a != b$ implies $b != a$) and transitive (i.e. $a == b$ && $b == c$ implies $c == a$).

Some SYCL runtime classes will have additional behavior associated with copy, movement, assignment or destruction semantics. If these are specified they are in addition to those specified above unless stated otherwise.

Each of the runtime classes mentioned above must provide a common interface of special member functions and member functions in order to fulfill the copy, move, destruction and equality requirements, following the rule of five and the rule of zero.
These common special member functions and hidden friend functions are described in Table 9 and Table 10 respectively.

```cpp
namespace sycl {

    class T {

    public:
        // If any of the following five special member functions are not
        // public, inline or defaulted, then all five of them should be
        // explicitly declared (see rule of five).
        // Otherwise, none of them should be explicitly declared
        // (see rule of zero).

        // T(const T &rhs);

        // T(T &&rhs);

        // T &operator=(const T &rhs);

        // T &operator=(T &&rhs);

        // ~T();

        ... 

        friend bool operator==(const T &lhs, const T &rhs) { /* ... */ }

        friend bool operator!=(const T &lhs, const T &rhs) { /* ... */ }

    } // namespace sycl

Table 9. Common special member functions for by-value semantics

<table>
<thead>
<tr>
<th>Special member function (see rule of five and rule of zero)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(const T &amp; rhs);</td>
<td>Copy constructor.</td>
</tr>
<tr>
<td>T(T&amp;&amp; rhs);</td>
<td>Move constructor.</td>
</tr>
<tr>
<td>T &amp;operator=(const T &amp; rhs);</td>
<td>Copy assignment operator.</td>
</tr>
<tr>
<td>T &amp;operator=(T&amp;&amp; rhs);</td>
<td>Move assignment operator.</td>
</tr>
<tr>
<td>~T();</td>
<td>Destructor.</td>
</tr>
</tbody>
</table>
```
Table 10. Common hidden friend functions for by-value semantics

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool operator==(const T &amp;lhs, const T &amp;rhs)</code></td>
<td>Returns true if this LHS SYCL T is equal to the RHS SYCL T in accordance with the requirements set out above, otherwise returns false.</td>
</tr>
<tr>
<td><code>bool operator!=(const T &amp;lhs, const T &amp;rhs)</code></td>
<td>Returns true if this LHS SYCL T is not equal to the RHS SYCL T in accordance with the requirements set out above, otherwise returns false.</td>
</tr>
</tbody>
</table>

4.5.4. Properties

Each of the following SYCL runtime classes: accessor, buffer, host_accessor, host_sampled_image_accessor, host_unsampled_image_accessor, context, local_accessor, queue, sampled_image, sampled_image_accessor, stream, unsampled_image, unsampled_image_accessor and usm_allocator provide an optional parameter in each of their constructors to provide a property_list which contains zero or more properties. Each of those properties augments the semantics of the class with a particular feature. Each of those classes must also provide has_property and get_property member functions for querying for a particular property.

The listing below illustrates the usage of various buffer properties, described in Section 4.7.2.2.

The example illustrates how using properties does not affect the type of the object, thus, does not prevent the usage of SYCL objects in containers.

```cpp
1 {
2   context myContext;
3
4   std::vector<buffer<int, 1>> bufferList {
5       buffer<int, 1> { ptr, rng },
6       buffer<int, 1> { ptr, rng, property::use_host_ptr {} },
7       buffer<int, 1> { ptr, rng, property::context_bound { myContext } }
8   };
9
10  for (auto& buf : bufferList) {
11     if (buf.has_property<property::context_bound>()) {
12        auto prop = buf.get_property<property::context_bound>();
13        assert(myContext == prop.get_context());
14     }
15  }
16 }
```

Each property is represented by a unique class and an instance of a property is an instance of that type. Some properties can be default constructed while others will require an argument on construction. A property may be applicable to more than one class, however some properties may not be compatible with each other. See the requirements for the properties of the SYCL buffer class, SYCL unsampled_image class and SYCL sampled_image class in Table 41 and Table 48 respectively.

Properties can be passed to a SYCL runtime class via an instance of property_list. These properties get tied to the SYCL runtime class instance and copies of the object will contain the same properties.

A SYCL implementation or a SYCL backend may provide additional properties other than those defined here, provided they are defined in accordance with the requirements described in Section 4.3.
### 4.5.4.1. Properties interface

Each of the runtime classes mentioned above must provide a common interface of member functions in order to fulfill the property interface requirements.

A synopsis of the common properties interface, the SYCL `property_list` class and the SYCL property classes is provided below. The member functions of the common properties interface are listed in Table 12. The constructors of the SYCL `property_list` class are listed in Table 13.

```cpp
namespace sycl {

  template<typename Property> struct is_property;

  template<typename Property>
  inline constexpr bool is_property_v = is_property<Property>::value;

  template<typename Property, typename SyclObject>
  struct is_property_of;

  template<typename Property, typename SyclObject>
  inline constexpr bool is_property_of_v = is_property_of<Property, SyclObject>::value;

  class T {
    ...
    template<typename Property>
    bool has_property() const noexcept;

    template<typename Property>
    Property get_property() const;

    ...
  };

  class property_list {
    public:
    template<typename... Properties> property_list(Properties... props);
  };
}
```

<table>
<thead>
<tr>
<th>Traits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;typename Property&gt; struct is_property</td>
<td>An explicit specialization of <code>is_property</code> that inherits from <code>std::true_type</code> must be provided for each property, where <code>Property</code> is the class defining the property. This includes both standard properties described in this specification and any additional non-standard properties defined by an implementation. All other specializations of <code>is_property</code> must inherit from <code>std::false_type</code>.</td>
</tr>
</tbody>
</table>
| template<typename Property>
inline constexpr bool is_property_v | Variable containing value of `is_property<Property>`. |
### 4.6. SYCL runtime classes

#### 4.6.1. Device selection

Since a system can have several SYCL-compatible devices attached, it is useful to have a way to select a specific device or a set of devices to construct a specific object such as a `device` (see Section 4.6.4) or a `queue` (see Section 4.6.5), or perform some operations on a device subset.

Device selection is done either by already having a specific instance of a `device` (see Section 4.6.4) or by providing a `device selector` which is a ranking function that will give an integer ranking value to all the devices on the system.

#### 4.6.1.1. Device selector

The interface for a `device selector` is any object that meets the C++ named requirement `Callable`, taking a parameter of type `const device &` and returning a value that is implicitly convertible to `int`.

At any point where the SYCL runtime needs to select a SYCL `device` using a `device selector`, the system queries all root devices from all SYCL backends in the system, calls the `device selector` on each device.
and selects the one which returns the highest score. If the highest value is strictly negative no device is selected.

In places where only one device has to be picked and the high score is obtained by more than one device, then one of the tied devices will be returned, but which one is not defined and may depend on enumeration order, for example, outside the control of the SYCL runtime.

Some predefined device selectors are provided by the system as described on Table 14 in a header file with some definition similar to the following:

<table>
<thead>
<tr>
<th>SYCL device selectors</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default_selector_v</td>
<td>Select a SYCL device from any supported SYCL backend based on an implementation-defined heuristic. Since all implementations must support at least one device, this selector must always return a device.</td>
</tr>
<tr>
<td></td>
<td>Implementations may choose to return an emulated device (with aspect::emulated) as a fallback if there is no physical device available on the system.</td>
</tr>
<tr>
<td>gpu_selector_v</td>
<td>Select a SYCL device from any supported SYCL backend for which the device type is info::device_type::gpu. The SYCL class constructor using it must throw an exception with the errc::runtime error code if no device matching this requirement can be found.</td>
</tr>
<tr>
<td>accelerator_selector_v</td>
<td>Select a SYCL device from any supported SYCL backend for which the device type is info::device_type::accelerator. The SYCL class constructor using it must throw an exception with the errc::runtime error code if no device matching this requirement can be found.</td>
</tr>
<tr>
<td>cpu_selector_v</td>
<td>Select a SYCL device from any supported SYCL backend for which the device type is info::device_type::cpu. The SYCL class constructor using it must throw an exception with the errc::runtime error code if no device matching this requirement can be found.</td>
</tr>
</tbody>
</table>
The free function `aspect_selector` has several overloads, each of which returns a selector object that selects a SYCL device from any supported SYCL backend which contains all the requested aspects, i.e. for the specific device `dev` and each aspect `devAspect` from `aspectList dev.has(devAspect)` equals `true`. If no aspects are passed in, the generated selector behaves like `default_selector`.

Required aspects can be passed in as a vector, as function arguments, or as template parameters, depending on the function overload. The function overload that takes `aspectList` as a vector takes another vector argument `denyList` where the user can specify all the aspects that have to be avoided, i.e. for the specific device `dev` and each aspect `devAspect` from `denyList dev.has(devAspect)` equals `false`.

The SYCL class constructor using the generated selector must throw an exception with the `errc::runtime` error code if no device matching this requirement can be found. There are multiple overloads of this function, please refer to [header:device-selector] for full definitions and to [example:aspect-selector] for examples.

```
namespace sycl {

  // Predefined device selectors
  __unspecified__ default_selector_v;
  __unspecified__ cpu_selector_v;
  __unspecified__ gpu_selector_v;
  __unspecified__ accelerator_selector_v;

  // Predefined types for compatibility with old SYCL 1.2.1 device selectors
  using default_selector = __unspecified__;
  using cpu_selector = __unspecified__;
  using gpu_selector = __unspecified__;
  using accelerator_selector = __unspecified__;

  // Returns a selector that selects a device based on desired aspects
  __unspecified_callable__
  aspect_selector(const std::vector<aspect>& aspectList,
                  const std::vector<aspect>& denyList = {});

  template <typename... AspectList>
  __unspecified_callable__ aspect_selector(AspectList... aspectList);
  template <aspect... AspectList>
  __unspecified_callable__ aspect_selector();

} // namespace sycl
```

Typical examples of default and user-provided device selectors could be:
sycl::device my_gpu { sycl::gpu_selector_v };

sycl::queue my_accelerator { sycl::accelerator_selector_v };

int prefer_my_vendor(const sycl::device& d) {
    // Return 1 if the vendor name is "MyVendor" or 0 else.
    // 0 does not prevent another device to be picked as a second choice
    return d.get_info<sycl::info::device::vendor>() == "MyVendor";
}

// Get the preferred device or another one if not available
sycl::device preferred_device { prefer_my_vendor };

// This throws if there is no such device in the system
sycl::queue half_precision_controller {
    // Can use a lambda as a device ranking function.
    // Returns a negative number to fail in the case there is no such device
    [] (auto& d) { return d.has(sycl::aspect::fp16) ? 1 : -1; }
};

// To ease porting SYCL 1.2.1 code, there are types whose
// construction leads to the equivalent predefined device selector
sycl::queue my_old_style_gpu { sycl::gpu_selector { } };

Examples of using aspect_selector:

```
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

// Unrestrained selection, equivalent to default_selector
auto dev0 = device{aspect_selector()};

// Pass aspects in a vector
// Only accept CPUs that support half
auto dev1 = device{aspect_selector(std::vector{aspect::cpu, aspect::fp16})};

// Pass aspects without a vector
// Only accept GPUs that support half
auto dev2 = device{aspect_selector(aspect::gpu, aspect::fp16)};

// Pass aspects as compile-time parameters
// Only accept devices that can be debugged on host and support half
auto dev3 = device{aspect_selector<aspect::host_debuggable, aspect::fp16>()};

// Pass aspects in an allowlist and a denylist
// Only accept devices that support half and double floating point precision,
// but exclude emulated devices and devices of type "custom"
auto dev4 = device{aspect_selector(
    std::vector{aspect::fp16, aspect::fp64},
    std::vector{aspect::emulated, aspect::custom}
)};
```

In SYCL 1.2.1 the predefined device selectors were actually types that had to be instantiated to be used. Now they are just instances. To simplify porting code using the old type instantiations, a backward-compatible API is still provided, such as `sycl::default_selec-
The new predefined device selectors have their new names appended with "_v" to avoid conflicts, thus following the naming style used by traits in the C++ standard library. There is no requirement for the implementation to have for example `sycl::gpu_selector_v` being an instance of `sycl::gpu_selector`.

Implementation note: the SYCL API might rely on SFINAE or C++20 concepts to resolve some ambiguity in constructors with default parameters.

### 4.6.2. Platform class

The SYCL platform class encapsulates a single SYCL platform on which SYCL kernel functions may be executed. A SYCL platform must be associated with a single SYCL backend.

A SYCL platform is also associated with one or more SYCL devices associated with the same SYCL backend.

All member functions of the platform class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The execution environment for a SYCL application has a fixed number of platforms which does not vary as the application executes. The application can get a list of all these platforms via `platform::get_platforms()`, and the order of the platform objects is the same each time the application calls that function. The platform class also provides constructors, but constructing a new platform instance merely creates a new object that is a copy of one of the objects returned by `platform::get_platforms()`.

The SYCL platform class provides the common reference semantics (see Section 4.5.2).

#### 4.6.2.1. Platform interface

A synopsis of the SYCL platform class is provided below. The constructors, member functions and static member functions of the SYCL platform class are listed in Table 15, Table 16 and Table 17 respectively. The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8 respectively.

```cpp
namespace sycl {
    class platform {
        public:
            platform();

            template <typename DeviceSelector>
            explicit platform(const DeviceSelector& deviceSelector);

            /* -- common interface members -- */

            backend get_backend() const noexcept;

            std::vector<device>
                get_devices(info::device_type = info::device_type::all) const;

            template <typename Param> typename Param::return_type get_info() const;

            template <typename Param>
                typename Param::return_type get_backend_info() const;

            bool has(aspect asp) const;
    }
}
```
```cpp
bool has_extension(const std::string& extension) const; // Deprecated
static std::vector<platform> get_platforms();
```

Table 15. Constructors of the SYCL platform class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform()</td>
<td>Constructs a SYCL platform instance that is a copy of the platform which contains the device returned by default_selector_v.</td>
</tr>
<tr>
<td>template &lt;typename DeviceSelector&gt; explicit platform(const DeviceSelector&amp;)</td>
<td>Constructs a SYCL platform instance that is a copy of the platform which contains the device returned by the device selector parameter.</td>
</tr>
</tbody>
</table>

Table 16. Member functions of the SYCL platform class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend get_backend() const noexcept</td>
<td>Returns a backend identifying the SYCL backend associated with this platform.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_info() const</td>
<td>Queries this SYCL platform for information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the info parameters in Table 25 to facilitate returning the type associated with the Param parameter.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_backend_info() const</td>
<td>Queries this SYCL platform for SYCL backend-specific information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the SYCL backend specification. Must throw an exception with the errc::backend_mismatch error code if the SYCL backend that corresponds with Param is different from the SYCL backend that is associated with this platform.</td>
</tr>
<tr>
<td>bool has(aspect asp) const</td>
<td>Returns true if all of the SYCL devices associated with this SYCL platform have the given aspect.</td>
</tr>
</tbody>
</table>
Member function | Description
---|---
`bool has_extension(const std::string& extension) const` | Deprecated, use `has()` instead. Returns true if this SYCL platform supports the extension queried by the `extension` parameter. A SYCL platform can only support an extension if all associated SYCL devices support that extension.

`std::vector<device>`
`get_devices(info::device_type deviceType = info::device_type::all) const` | Returns a `std::vector` containing all the root devices associated with this SYCL platform which have the device type encapsulated by `deviceType`.

| Table 17. Static member functions of the SYCL platform class |
|----------------|-------------|
| Static member function | Description |
| `static std::vector<platform> get_platforms()` | Returns a `std::vector` containing all SYCL platforms from all SYCL backends available in the system. |

### 4.6.2.2. Platform information descriptors

A **platform** can be queried for information using the `get_info` member function of the **platform** class, specifying one of the `info` parameters in `info::platform`. The possible values for each `info` parameter and any restrictions are defined in the specification of the SYCL backend associated with the **platform**. All `info` parameters in `info::platform` are specified in Table 18 and the synopsis for `info::platform` is described in Section A.1.

**Table 18. Platform information descriptors**

<table>
<thead>
<tr>
<th>Platform descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::platform::version</code></td>
<td><code>std::string</code></td>
<td>Returns a backend-defined platform version.</td>
</tr>
<tr>
<td><code>info::platform::name</code></td>
<td><code>std::string</code></td>
<td>Returns the name of the platform.</td>
</tr>
<tr>
<td><code>info::platform::vendor</code></td>
<td><code>std::string</code></td>
<td>Returns the name of the vendor providing the platform.</td>
</tr>
<tr>
<td><code>info::platform::extensions</code></td>
<td><code>std::vector&lt;std::string&gt;</code></td>
<td>Deprecated, use <code>device::get_info()</code> with <code>info::device::aspects</code> instead. Returns the extensions supported by the platform.</td>
</tr>
</tbody>
</table>

### 4.6.3. Context class

The **context** class represents a SYCL context. A **context** represents the runtime data structures and state required by a SYCL backend API to interact with a group of devices associated with a platform.

The SYCL **context** class provides the common reference semantics (see Section 4.5.2).
### 4.6.3.1. Context interface

The constructors and member functions of the SYCL `context` class are listed in Table 19 and Table 20, respectively. The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

All member functions of the `context` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

All constructors of the SYCL `context` class will construct an instance associated with a particular SYCL backend, determined by the constructor parameters or, in the case of the default constructor, the SYCL device produced by the `default_selector_v`.

A SYCL `context` can optionally be constructed with an `async_handler` parameter. In this case the `async_handler` is used to report asynchronous SYCL exceptions, as described in Section 4.13.

Information about a SYCL `context` may be queried through the `get_info()` member function.

```cpp
namespace sycl {
    class context {
        public:
            explicit context(const property_list& propList = {});
            explicit context(async_handler asyncHandler,
                             const property_list& propList = {});
            explicit context(const device& dev, const property_list& propList = {});
            explicit context(const device& dev, async_handler asyncHandler,
                             const property_list& propList = {});
            explicit context(const std::vector<device>& deviceList,
                             const property_list& propList = {});
            explicit context(const std::vector<device>& deviceList,
                             async_handler asyncHandler,
                             const property_list& propList = {});
            /* -- property interface members -- */
            /* -- common interface members -- */
            backend get_backend() const noexcept;
            platform get_platform() const;
            std::vector<device> get_devices() const;
            template <typename Param> typename Param::return_type get_info() const;
            template <typename Param> typename Param::return_type get_backend_info() const;
    } // namespace sycl
```

Table 19. Constructors of the SYCL `context` class
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>explicit context(async_handler asyncHandler = {})</code></td>
<td>Constructs a SYCL context instance using an instance of <code>default_selector_v</code> to select the associated SYCL platform and device(s). The devices that are associated with the constructed context are implementation-defined but must contain the device chosen by the device selector. The constructed SYCL context will use the asyncHandler parameter to handle exceptions.</td>
</tr>
<tr>
<td><code>explicit context(const device&amp; dev, async_handler asyncHandler = {})</code></td>
<td>Constructs a SYCL context instance using the dev parameter as the associated SYCL device and the SYCL platform associated with the dev parameter as the associated SYCL platform. The constructed SYCL context will use the asyncHandler parameter to handle exceptions.</td>
</tr>
<tr>
<td><code>explicit context(const std::vector&lt;device&gt;&amp; deviceList, async_handler asyncHandler = {})</code></td>
<td>Constructs a SYCL context instance using the SYCL device(s) in the deviceList parameter as the associated SYCL device(s) and the SYCL platform associated with each SYCL device in the deviceList parameter as the associated SYCL platform. This requires that all SYCL devices in the deviceList parameter have the same associated SYCL platform. The constructed SYCL context will use the asyncHandler parameter to handle exceptions.</td>
</tr>
</tbody>
</table>

### Table 20. Member functions of the `context` class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backend get_backend() const noexcept</code></td>
<td>Returns a backend identifying the SYCL backend associated with this context.</td>
</tr>
<tr>
<td><code>template&lt;typename Param&gt; typename Param::return_type get_info() const</code></td>
<td>Queries this SYCL context for information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the info parameters in Table 21 to facilitate returning the type associated with the Param parameter.</td>
</tr>
</tbody>
</table>
4.6.3.2. Context information descriptors

A context can be queried for information using the get_info member function of the context class, specifying one of the info parameters in info::context. The possible values for each info parameter and any restrictions are defined in the specification of the SYCL backend associated with the context. All info parameters in info::context are specified in Table 21 and the synopsis for info::context is described in Section A.2.

Table 21. Context information descriptors

<table>
<thead>
<tr>
<th>Context Descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::context::platform</td>
<td>platform</td>
<td>Returns the platform associated with the context.</td>
</tr>
<tr>
<td>info::context::devices</td>
<td>std::vector&lt;device&gt;</td>
<td>Returns all of the devices associated with the context.</td>
</tr>
<tr>
<td>info::context::atomic_memory_order_capabilities</td>
<td>std::vector&lt;memory_order&gt;</td>
<td>Returns the set of memory orderings supported by atomic operations on all devices in the context, which is guaranteed to include relaxed. The memory ordering of the context determines the behavior of atomic operations applied to any memory that can be concurrently accessed by multiple devices in the context.</td>
</tr>
</tbody>
</table>
### Context Descriptors

<table>
<thead>
<tr>
<th>Context Descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info::context::atomic_fence_order_capabilities</td>
<td>std::vector&lt;memory_order&gt;</td>
<td>Returns the set of memory orderings supported by <code>atomic_fence</code> on all devices in the context, which is guaranteed to include relaxed, acquire, release and acq_rel. The memory ordering of the context determines the behavior of fence operations applied to any memory that can be concurrently accessed by multiple devices in the context.</td>
</tr>
<tr>
<td>info::context::atomic_memory_scope_capabilities</td>
<td>std::vector&lt;memory_scope&gt;</td>
<td>Returns the set of memory scopes supported by atomic operations on all devices in the context, which is guaranteed to include work_group.</td>
</tr>
<tr>
<td>info::context::atomic_fence_scope_capabilities</td>
<td>std::vector&lt;memory_scope&gt;</td>
<td>Returns the set of memory orderings supported by <code>atomic_fence</code> on all devices in the context, which is guaranteed to include work_group.</td>
</tr>
</tbody>
</table>

### 4.6.3.3. Context properties

The `property_list` constructor parameters are present for extensibility.

### 4.6.4. Device class

The SYCL `device` class encapsulates a single SYCL device on which kernels can be executed.

All member functions of the `device` class are synchronous and errors are handled by throwing synchronous SYCL exceptions.

The execution environment for a SYCL application has a fixed number of root devices which does not vary as the application executes. The application can get a list of all these devices via `device::getDevices()`, and the order of the device objects is the same each time the application calls that function (assuming the parameter to that function is the same for each call). The `device` class also provides constructors, but constructing a new `device` instance merely creates a new object that is a copy of one of the objects returned by `device::getDevices()`.

A SYCL `device` can be partitioned into multiple SYCL devices, by calling the `create_sub_devices()` member function template. The resulting SYCL `devices` are considered sub devices, and it is valid to partition these sub devices further. The range of support for this feature is SYCL backend and device specific and can be queried for through `get_info()`.

The SYCL `device` class provides the common reference semantics (see Section 4.5.2).

### 4.6.4.1. Device interface

A synopsis of the SYCL `device` class is provided below. The constructors, member functions and static member functions of the SYCL `device` class are listed in Table 22, Table 23 and Table 24 respectively. The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

```cpp
namespace sycl {
    class device {
```

```cpp
```
Table 22. Constructors of the SYCL device class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device()</td>
<td>Constructs a SYCL device instance that is a copy of the device returned by default_selector_v.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Description</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructs a SYCL device instance that is a copy of the device returned by</td>
<td>template &lt;typename DeviceSelector&gt; explicit device(const DeviceSelector&amp;)</td>
</tr>
<tr>
<td>the device selector parameter.</td>
<td></td>
</tr>
</tbody>
</table>

### Table 23. Member functions of the SYCL device class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend get_backend() const noexcept</td>
<td>Returns a backend identifying the SYCL backend associated with this device.</td>
</tr>
<tr>
<td>(platform get_platform() const</td>
<td>Returns the associated SYCL platform. The value returned must be equal to that returned by get_info<a href="">info::device::platform</a>().</td>
</tr>
<tr>
<td>bool is_cpu() const</td>
<td>Returns the same value as has(aspect::cpu). See Table 26.</td>
</tr>
<tr>
<td>bool is_gpu() const</td>
<td>Returns the same value as has(aspect::gpu). See Table 26.</td>
</tr>
<tr>
<td>bool is_accelerator() const</td>
<td>Returns the same value as has(aspect::accelerator). See Table 26.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_info() const</td>
<td>Queries this SYCL device for information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the info parameters in Table 25 to facilitate returning the type associated with the Param parameter.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_backend_info() const</td>
<td>Queries this SYCL device for SYCL backend-specific information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the SYCL backend specification. Must throw an exception with the errc::backend_mismatch error code if the SYCL backend that corresponds with Param is different from the SYCL backend that is associated with this device.</td>
</tr>
<tr>
<td>bool has(aspect asp) const</td>
<td>Returns true if this SYCL device has the given aspect. SYCL applications can use this member function to determine which optional features this device supports (if any).</td>
</tr>
<tr>
<td>bool has_extension(const std::string&amp; extension) const</td>
<td>Deprecated, use has() instead. Returns true if this SYCL device supports the extension queried by the extension parameter.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>template &lt;info::partition_property Prop&gt; std::vector&lt;device&gt; create_sub_devices(size_t count) const</code></td>
<td>Available only when <code>Prop</code> is <code>info::partition_property::partition_equally</code>. Returns a <code>std::vector</code> of sub devices partitioned from this SYCL device based on the <code>count</code> parameter. The returned vector contains as many sub devices as can be created such that each sub device contains <code>count</code> compute units. If the device's total number of compute units (as returned by <code>info::device::max_compute_units</code>) is not evenly divided by <code>count</code>, then the remaining compute units are not included in any of the sub devices. If this SYCL device does not support <code>info::partition_property::partition_equally</code> an exception with the <code>errc::feature_not_supported</code> error code must be thrown. If <code>count</code> exceeds the total number of compute units in the device, an exception with the <code>errc::invalid</code> error code must be thrown.</td>
</tr>
<tr>
<td><code>template &lt;info::partition_property Prop&gt; std::vector&lt;device&gt; create_sub_devices(const std::vector&lt;size_t&gt;&amp; counts) const</code></td>
<td>Available only when <code>Prop</code> is <code>info::partition_property::partition_by_counts</code>. Returns a <code>std::vector</code> of sub devices partitioned from this SYCL device based on the <code>counts</code> parameter. For each non-zero value <code>M</code> in the <code>counts</code> vector, a sub device with <code>M</code> compute units is created. If the SYCL device does not support <code>info::partition_property::partition_by_counts</code> an exception with the <code>errc::feature_not_supported</code> error code must be thrown. If the number of non-zero values in <code>counts</code> exceeds the device's maximum number of sub devices (as returned by <code>info::device::partition_max_sub_devices</code>) or if the total of all the values in the <code>counts</code> vector exceeds the total number of compute units in the device (as returned by <code>info::device::max_compute_units</code>), an exception with the <code>errc::invalid</code> error code must be thrown.</td>
</tr>
</tbody>
</table>
**Table 24. Static member functions of the SYCL device class**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;info::partition_property Prop&gt; std::vector&lt;device&gt; create_sub_devices(info::partition_affinity_domain domain) const</td>
<td>Available only when Prop is info::partition_property::partition_by_affinity_domain. Returns a std::vector of sub devices partitioned from this SYCL device by affinity domain based on the domain parameter, which must be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::numa: Split the device into sub devices comprised of compute units that share a NUMA node.</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::L4_cache: Split the device into sub devices comprised of compute units that share a level 4 data cache.</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::L3_cache: Split the device into sub devices comprised of compute units that share a level 3 data cache.</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::L2_cache: Split the device into sub devices comprised of compute units that share a level 2 data cache.</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::L1_cache: Split the device into sub devices comprised of compute units that share a level 1 data cache.</td>
</tr>
<tr>
<td></td>
<td>• info::partition_affinity_domain::next_partitionable: Split the device along the next partitionable affinity domain. The implementation shall find the first level along which the device or sub device may be further subdivided in the order numa, L4_cache, L3_cache, L2_cache, L1_cache, and partition the device into sub devices comprised of compute units that share memory subsystems at this level. The user may determine what happened via info::device::partition_type_affinity_domain.</td>
</tr>
<tr>
<td></td>
<td>If the SYCL device does not support info::partition_property::partition_by_affinity_domain or the SYCL device does not support the info::partition_affinity_domain provided, an exception with the errc::feature_not_supported error code must be thrown.</td>
</tr>
</tbody>
</table>
4.6.4.2. Device information descriptors

A `device` can be queried for information using the `get_info` member function of the `device` class, specifying one of the info parameters in `info::device`. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the `device`. All info parameters in `info::device` are specified in Table 25 and the synopsis for `info::device` is described in Section A.3.

### Table 25. Device information descriptors

<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::device_type</code></td>
<td><code>info::device_type</code></td>
<td>Returns the device type associated with the <code>device</code>. May not return <code>info::device_type::all</code>.</td>
</tr>
<tr>
<td><code>info::device::vendor_id</code></td>
<td><code>uint32_t</code></td>
<td>Returns a unique vendor device identifier.</td>
</tr>
<tr>
<td><code>info::device::max_compute_units</code></td>
<td><code>uint32_t</code></td>
<td>Returns the number of parallel compute units available to the <code>device</code>. The minimum value is 1.</td>
</tr>
<tr>
<td><code>info::device::max_work_item_dimensions</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum dimensions that specify the global and local work-item IDs used by the data parallel execution model. The minimum value is 3 if this SYCL <code>device</code> is not of device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::max_work_item_sizes&lt;1&gt;</code></td>
<td><code>range&lt;1&gt;</code></td>
<td>Returns the maximum number of work-items that are permitted in a work-group for a kernel running in a one-dimensional index space. The minimum value is (1) for <code>devices</code> that are not of device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::max_work_item_sizes&lt;2&gt;</code></td>
<td><code>range&lt;2&gt;</code></td>
<td>Returns the maximum number of work-items that are permitted in each dimension of a work-group for a kernel running in a two-dimensional index space. The minimum value is (1,1) for <code>devices</code> that are not of device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::max_work_item_sizes&lt;3&gt;</code></td>
<td><code>range&lt;3&gt;</code></td>
<td>Returns the maximum number of work-items that are permitted in each dimension of a work-group for a kernel running in a three-dimensional index space. The minimum value is (1,1,1) for <code>devices</code> that are not of device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td>Device descriptors</td>
<td>Return type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>info::device::max_work_group_size</td>
<td>size_t</td>
<td>Returns the maximum number of work-items that are permitted in a work-group executing a kernel on a single compute unit. The minimum value is 1.</td>
</tr>
<tr>
<td>info::device::max_num_sub_groups</td>
<td>uint32_t</td>
<td>Returns the maximum number of sub-groups in a work-group for any kernel executed on the device. The minimum value is 1.</td>
</tr>
<tr>
<td>info::device::sub_group_sizes</td>
<td>std::vector&lt;size_t&gt;</td>
<td>Returns a std::vector of size_t containing the set of sub-group sizes supported by the device.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_char</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_short</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_int</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_long</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_float</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_double</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::preferred_vector_width_half</td>
<td>uint32_t</td>
<td>Returns the preferred native vector width size for built-in scalar types that can be put into vectors. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_vector_width_double if the device does not have aspect::fp64 and must return 0 for info::device::preferred_vector_width_half if the device does not have aspect::fp16.</td>
</tr>
<tr>
<td>info::device::max_clock_frequency</td>
<td>uint32_t</td>
<td>Returns the maximum configured clock frequency of this SYCL device in MHz.</td>
</tr>
<tr>
<td>info::device::address_bits</td>
<td>uint32_t</td>
<td>Returns the default compute device address space size specified as an unsigned integer value in bits. Must return either 32 or 64.</td>
</tr>
</tbody>
</table>
### Device descriptors

<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::max_mem_alloc_size</code></td>
<td><code>uint64_t</code></td>
<td>Returns the maximum size of memory object allocation in bytes. The minimum value is max (1/4th of <code>info::device::global_mem_size</code>, 128<em>1024</em>1024) if this SYCL device is not of device type <code>info::device::type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::image_support</code></td>
<td><code>bool</code></td>
<td>Deprecated. Returns the same value as <code>device::has(aspect::image)</code>.</td>
</tr>
<tr>
<td><code>info::device::max_read_image_args</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum number of simultaneous image objects that can be read from by a kernel. The minimum value is 128 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::max_write_image_args</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image2d_max_width</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum width of a 2D image or 1D image in pixels. The minimum value is 8192 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image2d_max_height</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum height of a 2D image in pixels. The minimum value is 8192 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image3d_max_width</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum width of a 3D image in pixels. The minimum value is 2048 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image3d_max_height</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum height of a 3D image in pixels. The minimum value is 2048 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image3d_max_depth</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum depth of a 3D image in pixels. The minimum value is 2048 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td><code>info::device::image_max_buffer_size</code></td>
<td><code>size_t</code></td>
<td>Returns the number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if the SYCL device has <code>aspect::image</code>. Note that this information is intended for OpenCL interoperability only as this feature is not supported in SYCL.</td>
</tr>
<tr>
<td><code>info::device::maxsamplers</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum number of samplers that can be used in a kernel. The minimum value is 16 if the SYCL device has <code>aspect::image</code>.</td>
</tr>
<tr>
<td>Device descriptors</td>
<td>Return type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>info::device::max_parameter_size</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum size in bytes of the arguments that can be passed to a kernel. The minimum value is 1024 if this SYCL device is not of device type <code>info::device_type::custom</code>. For this minimum value, only a maximum of 128 arguments can be passed to a kernel.</td>
</tr>
<tr>
<td><code>info::device::mem_base_addr_align</code></td>
<td><code>uint32_t</code></td>
<td>Returns the minimum value in bits of the largest supported SYCL built-in data type if this SYCL device is not of device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::half_fp_config</code></td>
<td><code>std::vector&lt;info::fp_config&gt;</code></td>
<td>Returns a <code>std::vector</code> of <code>info::fp_config</code> describing the half precision floating-point capability of this SYCL device. The <code>std::vector</code> may contain zero or more of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::denorm</code>: denorms are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::inf_nan</code>: INF and quiet NaNs are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::round_to_nearest</code>: round to nearest even rounding mode is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::round_to_zero</code>: round to zero rounding mode is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::round_to_inf</code>: round to positive and negative infinity rounding modes are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::fma</code>: IEEE754-2008 fused multiply add is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::correctly_rounded_divide_sqrt</code>: divide and sqrt are correctly rounded as defined by the IEEE754 specification. This property is deprecated.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>info::fp_config::soft_float</code>: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.</td>
</tr>
</tbody>
</table>

If half precision is supported by this SYCL device (i.e. the `device` has `aspect::fp16` there is no minimum floating-point capability. If half support is not supported the returned `std::vector` must be empty.
<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::single_fp_config</code></td>
<td><code>std::vector&lt;info::fp_config&gt;</code></td>
<td>Returns a <code>std::vector</code> of <code>info::fp_config</code> describing the single precision floating-point capability of this SYCL device. The <code>std::vector</code> must contain one or more of the following values:</td>
</tr>
</tbody>
</table>

- `info::fp_config::denorm`: denorms are supported.
- `info::fp_config::inf_nan`: INF and quiet NaNs are supported.
- `info::fp_config::round_to_nearest`: round to nearest even rounding mode is supported.
- `info::fp_config::round_to_zero`: round to zero rounding mode is supported.
- `info::fp_config::round_to_inf`: round to positive and negative infinity rounding modes are supported.
- `info::fp_config::fma`: IEEE754-2008 fused multiply add is supported.
- `info::fp_config::correctly_rounded_divide_sqrt`: divide and `sqrt` are correctly rounded as defined by the IEEE754 specification. This property is deprecated.
- `info::fp_config::soft_float`: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.

If this SYCL device is not of type `info::device_type::custom` then the minimum floating-point capability must be: `info::fp_config::round_to_nearest` and `info::fp_config::inf_nan`.
<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::double_fp_config</code></td>
<td><code>std::vector&lt;info::fp_config&gt;</code></td>
<td>Returns a <code>std::vector</code> of <code>info::fp_config</code> describing the double precision floating-point capability of this SYCL device. The <code>std::vector</code> may contain zero or more of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::denorm</code>: denorms are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::inf_nan</code>: INF and NaNs are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::round_to_nearest</code>: round to nearest even rounding mode is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::round_to_zero</code>: round to zero rounding mode is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::round_to_inf</code>: round to positive and negative infinity rounding modes are supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::fma</code>: IEEE754-2008 fused multiply-add is supported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>info::fp_config::soft_float</code>: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software.</td>
</tr>
<tr>
<td><code>info::device::global_mem_cache_type</code></td>
<td><code>info::global_mem_cache_type</code></td>
<td>Returns the type of global memory cache supported.</td>
</tr>
<tr>
<td><code>info::device::global_mem_cache_line_size</code></td>
<td><code>uint32_t</code></td>
<td>Returns the size of global memory cache line in bytes.</td>
</tr>
<tr>
<td><code>info::device::global_mem_cache_size</code></td>
<td><code>uint64_t</code></td>
<td>Returns the size of global memory cache in bytes.</td>
</tr>
<tr>
<td><code>info::device::global_mem_size</code></td>
<td><code>uint64_t</code></td>
<td>Returns the size of global device memory in bytes.</td>
</tr>
<tr>
<td>Device descriptors</td>
<td>Return type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>info::device::max_constant_buffer_size</code></td>
<td><code>uint64_t</code></td>
<td>Deprecated in SYCL 2020. Returns the maximum size in bytes of a constant buffer allocation. The minimum value is 64 KB if this SYCL device is not of type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::max_constant_args</code></td>
<td><code>uint32_t</code></td>
<td>Deprecated in SYCL 2020. Returns the maximum number of constant arguments that can be declared in a kernel. The minimum value is 8 if this SYCL device is not of type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::local_mem_type</code></td>
<td><code>info::local_mem_type</code></td>
<td>Returns the type of local memory supported. This can be <code>info::local_mem_type::local</code> implying dedicated local memory storage such as SRAM, or <code>info::local_mem_type::global</code>. If this SYCL device is of type <code>info::device_type::custom</code> this can also be <code>info::local_mem_type::none</code>, indicating local memory is not supported.</td>
</tr>
<tr>
<td><code>info::device::local_mem_size</code></td>
<td><code>uint64_t</code></td>
<td>Returns the size of local memory arena in bytes. The minimum value is 32 KB if this SYCL device is not of type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>info::device::error_correction_support</code></td>
<td><code>bool</code></td>
<td>Returns true if the device implements error correction for all accesses to compute device memory (global and constant). Returns false if the device does not implement such error correction.</td>
</tr>
<tr>
<td><code>info::device::host_unified_memory</code></td>
<td><code>bool</code></td>
<td>Deprecated, use <code>device::has()</code> with one of the <code>aspect::usm_*</code> aspects instead. Returns true if the device and the host have a unified memory subsystem and returns false otherwise.</td>
</tr>
<tr>
<td><code>info::device::atomic_memory_order_capabilities</code></td>
<td><code>std::vector&lt;memory_order&gt;</code></td>
<td>Returns the set of memory orderings supported by atomic operations on the device, which is guaranteed to include <code>relaxed</code>.</td>
</tr>
<tr>
<td><code>info::device::atomic_fence_order_capabilities</code></td>
<td><code>std::vector&lt;memory_order&gt;</code></td>
<td>Returns the set of memory orderings supported by <code>atomic_fence</code> on the device, which is guaranteed to include <code>relaxed</code>, <code>acquire</code>, <code>release</code> and <code>acq_rel</code>.</td>
</tr>
<tr>
<td><code>info::device::atomic_memory_scope_capabilities</code></td>
<td><code>std::vector&lt;memory_scope&gt;</code></td>
<td>Returns the set of memory scopes supported by atomic operations on the device, which is guaranteed to include <code>work_group</code>.</td>
</tr>
<tr>
<td><code>info::device::atomic_fence_scope_capabilities</code></td>
<td><code>std::vector&lt;memory_scope&gt;</code></td>
<td>Returns the set of memory scopes supported by <code>atomic_fence</code> on the device, which is guaranteed to include <code>work_group</code>.</td>
</tr>
<tr>
<td>Device descriptors</td>
<td>Return type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>info::device::profiling_timer_resolution</code></td>
<td><code>size_t</code></td>
<td>Returns the resolution of device timer in nanoseconds.</td>
</tr>
<tr>
<td><code>info::device::is_endian_little</code></td>
<td><code>bool</code></td>
<td>Deprecated. Check the byte order of the host system instead. The host and device are required to have the same byte order. Returns true if this SYCL device is a little endian device and returns false otherwise.</td>
</tr>
<tr>
<td><code>info::device::is_available</code></td>
<td><code>bool</code></td>
<td>Returns true if the SYCL device is available and returns false if the device is not available.</td>
</tr>
<tr>
<td><code>info::device::is_compiler_available</code></td>
<td><code>bool</code></td>
<td>Deprecated. Returns the same value as <code>device::has(aspect::online_compiler)</code>.</td>
</tr>
<tr>
<td><code>info::device::is_linker_available</code></td>
<td><code>bool</code></td>
<td>Deprecated. Returns the same value as <code>device::has(aspect::online_linker)</code>.</td>
</tr>
<tr>
<td><code>info::device::execution_capabilities</code></td>
<td><code>std::vector&lt;info::execution_capability&gt;</code></td>
<td>Returns a std::vector of the <code>info::execution_capability</code> describing the supported execution capabilities. Note that this information is intended for OpenCL interoperability only as SYCL only supports <code>info::execution_capability::exec_kernel</code>.</td>
</tr>
<tr>
<td><code>info::device::queue_profiling</code></td>
<td><code>bool</code></td>
<td>Deprecated. Returns the same value as <code>device::has(aspect::queue_profiling)</code>.</td>
</tr>
<tr>
<td><code>info::device::built_in_kernel_ids</code></td>
<td><code>std::vector&lt;kernel_id&gt;</code></td>
<td>Returns a std::vector of identifiers for the built-in kernels supported by this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::built_in_kernels</code></td>
<td><code>std::vector&lt;std::string&gt;</code></td>
<td>Deprecated. Use <code>info::device::built_in_kernel_ids</code> instead. Returns a std::vector of built-in OpenCL kernels supported by this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::platform</code></td>
<td><code>platform</code></td>
<td>Returns the SYCL platform associated with this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::name</code></td>
<td><code>std::string</code></td>
<td>Returns the device name of this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::vendor</code></td>
<td><code>std::string</code></td>
<td>Returns the vendor of this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::driver_version</code></td>
<td><code>std::string</code></td>
<td>Returns a vendor-defined string describing the version of the underlying backend software driver.</td>
</tr>
</tbody>
</table>
### Device information descriptors

<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
</table>
| `info::device::profile`    | `std::string`   | Deprecated in SYCL 2020. Only supported when using the OpenCL backend (see Appendix C). Throws an exception with the `errc::invalid` error code if used with a device whose backend is not OpenCL. The value returned can be one of the following strings:  
  - `FULL_PROFILE` - if the device supports the OpenCL specification (functionality defined as part of the core specification and does not require any extensions to be supported).  
  - `EMBEDDED_PROFILE` - if the device supports the OpenCL embedded profile. |
| `info::device::version`    | `std::string`   | Returns a backend-defined `device` version.                                                                                                      |
| `info::device::backend_version` | `std::string` | Returns a string describing the version of the SYCL `backend` associated with the `device`. The possible values are specified in the SYCL `backend` specification of the SYCL `backend` associated with the `device`. |
| `info::device::aspects`    | `std::vector<aspect>` | Returns a `std::vector` of `aspect` values supported by this SYCL `device`.                                                                 |

SYCL 2020 rev 7
<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::extensions</code></td>
<td><code>std::vector&lt;std::string&gt;</code></td>
<td>Deprecated, use <code>info::device::aspects</code> instead. Returns a <code>std::vector</code> of extension names (the extension names do not contain any spaces) supported by this SYCL device. The extension names returned can be vendor supported extension names and one or more of the following Khronos approved extension names:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_int64_base_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_int64_extended_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_3d_image_writes</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_fp16</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_gl_sharing</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_gl_event</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_d3d10_sharing</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_dx9_media_sharing</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_d3d11_sharing</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_depth_images</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_gl_depth_images</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_gl_msaa_sharing</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_image2d_from_buffer</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_initialize_memory</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_context_abort</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_spir</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>If this SYCL device is an OpenCL device then following approved Khronos extension names must be returned by all device that support OpenCL C 1.2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_global_int32_base_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_global_int32_extended_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_local_int32_base_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_local_int32_extended_atomics</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_byte_addressable_store</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- <code>cl_khr_fp64</code> (for backward compatibility if double precision is supported)</td>
</tr>
</tbody>
</table>

Please refer to the OpenCL 1.2 Extension Specification for a detailed description of these extensions.
<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::device::printf_buffer_size</code></td>
<td><code>size_t</code></td>
<td>Deprecated in SYCL 2020. Returns the maximum size of the internal buffer that holds the output of <code>printf</code> calls from a kernel. The minimum value is 1 MB if <code>info::device::profile</code> returns true for this SYCL device.</td>
</tr>
<tr>
<td><code>info::device::preferred_interop_user_sync</code></td>
<td><code>bool</code></td>
<td>Deprecated in SYCL 2020. Only supported when using the OpenCL backend (see Appendix C). Throws an exception with the <code>errc::invalid</code> error code if used with a device whose backend is not OpenCL. Returns true if the preference for this SYCL device is for the user to be responsible for synchronization, when sharing memory objects between OpenCL and other APIs such as DirectX, false if the device/implementation has a performant path for performing synchronization of memory object shared between OpenCL and other APIs such as DirectX.</td>
</tr>
<tr>
<td><code>info::device::parent_device</code></td>
<td><code>device</code></td>
<td>Returns the parent SYCL device to which this sub-device is a child if this is a sub-device. Must throw an exception with the <code>errc::invalid</code> error code if this SYCL device is not a sub device.</td>
</tr>
<tr>
<td><code>info::device::partition_max_sub_devices</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum number of sub-devices that can be created when this SYCL device is partitioned. The value returned cannot exceed the value returned by <code>info::device::device_max_compute_units</code>.</td>
</tr>
<tr>
<td><code>info::device::partition_properties</code></td>
<td><code>std::vector&lt;info::partition_property&gt;</code></td>
<td>Returns the partition properties supported by this SYCL device; a vector of <code>info::partition_property</code>. An element is returned in this vector only if the device can be partitioned into at least two sub devices along that partition property.</td>
</tr>
<tr>
<td><code>info::device::partition_affinity_domains</code></td>
<td><code>std::vector&lt;info::partition_affinity_domain&gt;</code></td>
<td>Returns a <code>std::vector</code> of the partition affinity domains supported by this SYCL device when partitioning with <code>info::partition_property::partition_by_affinity_domain</code>. An element is returned in this vector only if the device can be partitioned into at least two sub devices along that affinity domain.</td>
</tr>
</tbody>
</table>
### Device descriptors

<table>
<thead>
<tr>
<th>Device descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
</table>
| `info::device::partition_type_property` | `info::partition_property` | Returns the partition property of this SYCL device. If this SYCL device is not a sub device then the return value must be `info::partition_property::no_partition`, otherwise it must be one of the following values:  
  - `info::partition_property::partition_equally`  
  - `info::partition_property::partition_by_counts`  
  - `info::partition_property::partition_by_affinity_domain` |
| `info::device::partition_type_affinity_domain` | `info::partition_affinity_domain` | Returns the partition affinity domain of this SYCL device. If this SYCL device is not a sub device or the sub device was not partitioned with `info::partition_type::partition_by_affinity_domain` then the return value must be `info::partition_affinity_domain::not_applicable`, otherwise it must be one of the following values:  
  - `info::partition_affinity_domain::numa`  
  - `info::partition_affinity_domain::L4_cache`  
  - `info::partition_affinity_domain::L3_cache`  
  - `info::partition_affinity_domain::L2_cache`  
  - `info::partition_affinity_domain::L1_cache` |

### 4.6.4.3. Device aspects

Every SYCL device has an associated set of aspects which identify characteristics of the device. Aspects are defined via the `enum class aspect` enumeration:

```cpp
definitions
namespace sycl {
    enum class aspect : /* unspecified */ {
        cpu,
        gpu,
        accelerator,
        custom,
        emulated,
        host_debuggable,
        fp16,
        fp64,
        atomic64,
        image,
        online_compiler,
        online_linker,
```
SYCL applications can query the aspects for a device via `device::has()` in order to determine whether the device supports any optional features. Table 26 lists the aspects that are defined in the core SYCL specification and tells which optional features correspond to each. Backends and extensions may provide additional aspects and additional optional device features. If so, the SYCL backend specification document or the extension document describes them.

### Table 26. Device aspects defined by the core SYCL specification

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>aspect::cpu</code></td>
<td>A device that runs on a CPU. Devices with this aspect have device type <code>info::device_type::cpu</code>.</td>
</tr>
<tr>
<td><code>aspect::gpu</code></td>
<td>A device that can also be used to accelerate a 3D graphics API. Devices with this aspect have device type <code>info::device_type::gpu</code>.</td>
</tr>
<tr>
<td><code>aspect::accelerator</code></td>
<td>A dedicated accelerator device, usually using a peripheral interconnect for communication. Devices with this aspect have device type <code>info::device_type::accelerator</code>.</td>
</tr>
<tr>
<td><code>aspect::custom</code></td>
<td>A dedicated accelerator that can use the SYCL API, but programmable kernels cannot be dispatched to the device, only fixed functionality is available. See Section 3.9.7. Devices with this aspect have device type <code>info::device_type::custom</code>.</td>
</tr>
<tr>
<td><code>aspect::emulated</code></td>
<td>Indicates that the device is somehow emulated. A device with this aspect is not intended for performance, and instead will generally have another purpose such as emulation or profiling. The precise definition of this aspect is left open to the SYCL implementation. As an example, a vendor might support both a hardware FPGA device and a software emulated FPGA, where the emulated FPGA has all the same features as the hardware one but runs more slowly and can provide additional profiling or diagnostic information. In such a case, an application's device selector can use <code>aspect::emulated</code> to distinguish the two.</td>
</tr>
<tr>
<td>Aspect</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>aspect::host_debuggable</td>
<td>Indicates that kernels running on this device can be debugged using standard debuggers that are normally available on the host system where the SYCL implementation resides. The precise definition of this aspect is left open to the SYCL implementation.</td>
</tr>
<tr>
<td>aspect::fp16</td>
<td>Indicates that kernels submitted to the device may use the sycl::half data type.</td>
</tr>
<tr>
<td>aspect::fp64</td>
<td>Indicates that kernels submitted to the device may use the double data type.</td>
</tr>
<tr>
<td>aspect::atomic64</td>
<td>Indicates that kernels submitted to the device may perform 64-bit atomic operations.</td>
</tr>
<tr>
<td>aspect::image</td>
<td>Indicates that the device supports images.</td>
</tr>
<tr>
<td>aspect::online_compiler</td>
<td>Indicates that the device supports online compilation of device code. Devices that have this aspect support the build() and compile() functions defined in Section 4.11.11.</td>
</tr>
<tr>
<td>aspect::online_linker</td>
<td>Indicates that the device supports online linking of device code. Devices that have this aspect support the link() functions defined in Section 4.11.11. All devices that have this aspect also have aspect::online_compiler.</td>
</tr>
<tr>
<td>aspect::queue_profiling</td>
<td>Indicates that the device supports queue profiling via property::queue::enable_profiling.</td>
</tr>
<tr>
<td>aspect::usm_device_allocations</td>
<td>Indicates that the device supports explicit USM allocations as described in Section 4.8.</td>
</tr>
<tr>
<td>aspect::usm_host_allocations</td>
<td>Indicates that the device can access USM memory allocated via usm::alloc::host. The device only supports atomic modification of a host allocation if aspect::usm_atomic_host_allocations is also supported. (See Section 4.8.)</td>
</tr>
<tr>
<td>aspect::usm_atomic_host_allocations</td>
<td>Indicates that the device supports USM memory allocated via usm::alloc::host. The host and this device may concurrently access and atomically modify host allocations. (See Section 4.8.)</td>
</tr>
<tr>
<td>aspect::usm_shared_allocations</td>
<td>Indicates that the device supports USM memory allocated via usm::alloc::shared on the same device. Concurrent access and atomic modification of a shared allocation is only supported if aspect::usm_atomic_shared_allocations is also supported. (See Section 4.8.)</td>
</tr>
<tr>
<td>Aspect:</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>aspect::usm_atomic_shared_allocations</td>
<td>Indicates that the device supports USM memory allocated via usm::alloc::shared. The host and other devices in the same context that also support this capability may concurrently access and atomically modify shared allocations. The allocation is free to migrate between the host and the appropriate devices. (See Section 4.8.)</td>
</tr>
<tr>
<td>aspect::usm_system_allocations</td>
<td>Indicates that the system allocator may be used instead of SYCL USM allocation mechanisms for usm::alloc::shared allocations on this device. (See Section 4.8.)</td>
</tr>
</tbody>
</table>

The implementation also provides two traits that the application can use to query aspects at compilation time. The traits any_device_has<aspect> and all_devices_have<aspect> are set according to the collection of devices $D$ that can possibly execute device code, as determined by the compilation environment. The trait any_device_has<aspect> inherits from std::true_type only if at least one device in $D$ has the specified aspect. The trait all_devices_have<aspect> inherits from std::true_type only if all devices in $D$ have the specified aspect.

```cpp
namespace sycl {

template <aspect Aspect> struct any_device_has;
template <aspect Aspect> struct all_devices_have;

template <aspect A>
inline constexpr bool any_device_has_v = any_device_has<A>::value;
template <aspect A>
inline constexpr bool all_devices_have_v = all_devices_have<A>::value;
}
```  // namespace sycl

Applications can use these traits to reduce their code size. The following example demonstrates one way to use these traits to avoid instantiating a templated kernel for device features that are not supported by any device.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl:" before SYCL names

constexpr int N = 512;

template <bool HasFp16> class MyKernel {
  public:
    void operator()(id<1> i) {
      if constexpr (HasFp16) {
        // Algorithm using sycl::half type
      } else {
        // Fall back code for devices that don't support sycl::half
      }
    }
}

int main() {
```
The kernel function MyKernel is templated to use a different algorithm depending on whether the device has the aspect `aspect::fp16`, and the call to `dev.has()` chooses the kernel function instantiation that matches the device's capabilities. However, the use of `any_device_has_v<aspect::fp16>` and `all_devices_have_v<aspect::fp16>` entirely avoid useless instantiations of the kernel function. For example, when the compilation environment does not support any devices with `aspect::fp16`, `any_device_has_v<aspect::fp16>` is `false`, and the kernel function is never instantiated with support for the `sycl::half` type.

Like any trait, the definitions of `any_device_has` and `all_devices_have` are uniform across all parts of a SYCL application. If an implementation uses SMCP, all compiler passes define a particular aspect's specialization of the traits the same way, regardless of whether that compiler pass' device supports the aspect. Thus, `any_device_has` and `all_devices_have` cannot be used to determine whether any particular device supports an aspect. Instead, applications must use `device::has()` or `platform::has()` for this.

An implementation could choose to provide command line options which affect the set of devices that it supports. If so, those command line options would also affect these traits. For example, if an implementation provides a command line option that disables `aspect::accelerator` devices, the trait `any_device_has<aspect::accelerator>` would inherit from `std::false_type` when that command line option was specified.

These traits only reflect the supported devices at the time the SYCL application is compiled. It's possible that unsupported devices are still visible to the application when it runs. However, if a device \(D\) is not supported when the application is compiled, the application will not be able to submit kernels to that device \(D\).

### 4.6.5. Queue class

The SYCL `queue` class encapsulates a single SYCL queue which schedules kernels on a SYCL device.

A SYCL `queue` can be used to submit command groups to be executed by the SYCL runtime using the `submit` member function.

All member functions of the `queue` class are synchronous and errors are handled by throwing synchronous SYCL exceptions. The `submit` member function synchronously invokes the provided `command group` function object (as described in Section 3.7.1.2) in the calling thread, thereby scheduling a command group for asynchronous execution. Any error in the submission of a command group is handled by throwing a synchronous SYCL exception. Any errors from the command group after it has been submitted are handled by passing asynchronous errors at specific times to an `async_handler`, as described in Section 4.13.
A SYCL queue can wait for all command groups that it has submitted by calling wait or wait_and_throw.

The default constructor of the SYCL queue class will construct a queue based on the SYCL device returned from the default_selector_v (see Section 4.6.1.1). All other constructors construct a queue as determined by the parameters provided. All constructors will implicitly construct a SYCL platform, device and context in order to facilitate the construction of the queue.

Each constructor takes as the last parameter an optional SYCL property_list to provide properties to the SYCL queue.

A SYCL queue may be destroyed even when there are uncompleted commands that have been submitted to the queue. Doing so does not block. Instead, any commands that have been submitted to the queue begin execution when their requisites are satisfied, just as they would had the queue not been destroyed. Any event objects for those commands are signaled in the normal manner when the command completes. Resources associated with the queue will be freed by the time the last command completes.

The SYCL queue class provides the common reference semantics (see Section 4.5.2).

### 4.6.5.1. Queue interface

A synopsis of the SYCL queue class is provided below. The constructors and member functions of the SYCL queue class are listed in Table 27 and Table 28 respectively. The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

Some queue member functions are shortcuts to member functions of the handler class. These are listed in Section 4.6.5.2.
const async_handler& asyncHandler,
    const property_list& propList = {});

explicit queue(const context& syclContext, const device& syclDevice,
    const async_handler& asyncHandler,
    const property_list& propList = {});

explicit queue(const context& syclContext, const device& syclDevice,
    const property_list& propList = {});

/* -- common interface members -- */

/* -- property interface members -- */

backend get_backend() const noexcept;
context get_context() const;
device get_device() const;
bool is_in_order() const;
template <typename Param> typename Param::return_type get_info() const;
template <typename Param> typename Param::return_type get_backend_info() const;
template <typename T> event submit(T cgf);
template <typename T> event submit(T cgf, const queue& secondaryQueue);
void wait();
void wait_and_throw();
void throw_asynchronous();
/* -- convenience shortcuts -- */
template <typename KernelName, typename KernelType>
    event single_task(const KernelType& kernelFunc);
template <typename KernelName, typename KernelType>
    event single_task(event depEvent, const KernelType& kernelFunc);
template <typename KernelName, typename KernelType>
    event single_task(const std::vector<event>& depEvents,
    const KernelType& kernelFunc);

// Parameter pack acts as-if: Reductions... reductions, const KernelType
// &kernelFunc
    template <typename KernelName, int Dims, typename... Rest>
    event parallel_for(range<Dims> numWorkItems, Rest&... rest);

// Parameter pack acts as-if: Reductions... reductions, const KernelType
// &kernelFunc
4.6.5.1. Queue interface

87 template <typename KernelName, int Dims, typename... Rest>
88 event parallel_for(range<Dims> numWorkItems, event depEvent, Rest... rest);
89
90 // Parameter pack acts as-if: Reductions... reductions, const KernelType
91 // &kernelFunc
92 template <typename KernelName, int Dims, typename... Rest>
93 event parallel_for(range<Dims> numWorkItems,
94       const std::vector<event>& depEvents, Rest... rest);
95
96 // Parameter pack acts as-if: Reductions... reductions, const KernelType
97 // &kernelFunc
98 template <typename KernelName, int Dims, typename... Rest>
99 event parallel_for(nd_range<Dims> executionRange, Rest... rest);
100
101 // Parameter pack acts as-if: Reductions... reductions, const KernelType
102 // &kernelFunc
103 template <typename KernelName, int Dims, typename... Rest>
104 event parallel_for(nd_range<Dims> executionRange, event depEvent,
105       Rest... rest);
106
107 // Parameter pack acts as-if: Reductions... reductions, const KernelType
108 // &kernelFunc
109 template <typename KernelName, int Dims, typename... Rest>
110 event parallel_for(nd_range<Dims> executionRange,
111       const std::vector<event>& depEvents, Rest... rest);
112
113 /* -- USM functions -- */
114
event memcpy(void* dest, const void* src, size_t numBytes);
115 event memcpy(void* dest, const void* src, size_t numBytes, event depEvent);
116 event memcpy(void* dest, const void* src, size_t numBytes,
117       const std::vector<event>& depEvents);
118
template <typename T> event copy(const T* src, T* dest, size_t count);
119 template <typename T>
120 event copy(const T* src, T* dest, size_t count, event depEvent);
121 template <typename T>
122 event copy(const T* src, T* dest, size_t count,
123       const std::vector<event>& depEvents);
124
event memset(void* ptr, int value, size_t numBytes);
125 event memset(void* ptr, int value, size_t numBytes, event depEvent);
126 event memset(void* ptr, int value, size_t numBytes,
127       const std::vector<event>& depEvents);
128
template <typename T> event fill(void* ptr, const T& pattern, size_t count);
129 template <typename T>
130 event fill(void* ptr, const T& pattern, size_t count, event depEvent);
131 template <typename T>
132 event fill(void* ptr, const T& pattern, size_t count,
133       const std::vector<event>& depEvents);
134
event prefetch(void* ptr, size_t numBytes);
135 event prefetch(void* ptr, size_t numBytes, event depEvent);
136 event prefetch(void* ptr, size_t numBytes,
137       const std::vector<event>& depEvents);
4.6.5.1. Queue interface

Table 27. Constructors of the queue class

```cpp
    template <typename T, int Dims, access_mode Mode, target Tgt,
              access_mode IsDestPlaceholder> copy(accessor<T, Dims, Mode, Tgt, IsPlaceholder> dest);
```
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>explicit queue(const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance using the device constructed from the <code>default_selector_v</code>. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>explicit queue(const async_handler&amp; asyncHandler, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance with an <code>async_handler</code> using the device constructed from the <code>default_selector_v</code>. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>template&lt;typename DeviceSelector&gt; explicit queue(const DeviceSelector&amp; deviceSelector, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance using the device returned by the <code>device_selector</code> provided. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>template&lt;typename DeviceSelector&gt; explicit queue(const DeviceSelector&amp; deviceSelector, const async_handler&amp; asyncHandler, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance with an <code>async_handler</code> using the device returned by the <code>device_selector</code> provided. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>explicit queue(const device&amp; syclDevice, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance using the <code>syclDevice</code> provided. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>explicit queue(const device&amp; syclDevice, const async_handler&amp; asyncHandler, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance with an <code>async_handler</code> using the <code>syclDevice</code> provided. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>template&lt;typename DeviceSelector&gt; explicit queue(const context&amp; syclContext, const DeviceSelector&amp; deviceSelector, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL queue instance that is associated with the <code>syclContext</code> provided, using the device returned by the <code>device_selector</code> provided. Must throw an exception with the <code>errc::invalid</code> error code if <code>syclContext</code> does not encapsulate the SYCL device returned by <code>deviceSelector</code>. Zero or more properties can be provided to the constructed SYCL queue via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
**Constructor**

```cpp
template <typename DeviceSelector>
explicit queue(const context& syclContext, const DeviceSelector& deviceSelector,
                const async_handler& asyncHandler,
                const property_list& propList = {})
```

Constructs a SYCL queue instance with an async_handler that is associated with the syclContext provided, using the device returned by the device selector provided. Must throw an exception with the errc::invalid error code if syclContext does not encapsulate the SYCL device returned by deviceSelector. Zero or more properties can be provided to the constructed SYCL queue via an instance of property_list.

```cpp
explicit queue(const context& syclContext, const device& syclDevice,
                const property_list& propList = {})
```

Constructs a SYCL queue instance using the syclDevice provided. This device must either be contained by syclContext or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code. Zero or more properties can be provided to the constructed SYCL queue via an instance of property_list.

```cpp
explicit queue(const context& syclContext, const device& syclDevice,
                const async_handler& asyncHandler,
                const property_list& propList = {})
```

Constructs a SYCL queue instance with an async_handler using the syclDevice provided. This device must either be contained by syclContext or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code. Zero or more properties can be provided to the constructed SYCL queue via an instance of property_list.

---

**Table 28. Member functions for queue class**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend get_backend() const noexcept</td>
<td>Returns a backend identifying the SYCL backend associated with this queue.</td>
</tr>
<tr>
<td>context get_context() const</td>
<td>Returns the SYCL queue's context. Reports errors using SYCL exception classes. The value returned must be equal to that returned by get_info<a href="">info::queue::context</a>().</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>device get_device() const</td>
<td>Returns the SYCL device the queue is associated with. Reports errors using SYCL exception classes. The value returned must be equal to that returned by get_info<a href="">info::queue::device</a>()</td>
</tr>
<tr>
<td>bool is_in_order() const</td>
<td>Returns true if the SYCL queue was created with the in_order property. Equivalent to has_property<a href="">property::queue::in_order</a>()</td>
</tr>
<tr>
<td>void wait()</td>
<td>Performs a blocking wait for the completion of all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions.</td>
</tr>
<tr>
<td>void wait_and_throw()</td>
<td>Performs a blocking wait for the completion of all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions. Any unconsumed asynchronous errors will be passed to the async_handler associated with the queue or enclosing context. If no user defined async_handler is associated with the queue or enclosing context, then an implementation-defined default async_handler is called to handle any errors, as described in Section 4.13.1.2.</td>
</tr>
<tr>
<td>void throw_asynchronous()</td>
<td>Checks to see if any unconsumed asynchronous errors have been produced by the queue and if so reports them by passing them to the async_handler associated with the queue or enclosing context. If no user defined async_handler is associated with the queue or enclosing context, then an implementation-defined default async_handler is called to handle any errors, as described in Section 4.13.1.2.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_info() const</td>
<td>Queries this SYCL queue for information requested by the template parameter Param. The type alias Param::return_type must be defined in accordance with the info parameters in Table 30 to facilitate returning the type associated with the Param parameter.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event submit(T cgf)</code></td>
<td>Submit a command group function object to the queue, in order to be scheduled for execution on the device.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event submit(T cgf, queue&amp; secondaryQueue)</code></td>
<td>Submit a command group function object to the queue, in order to be scheduled for execution on the device. On a kernel error, this command group function object is then scheduled for execution on the secondary queue. Returns an event, which corresponds to the queue the command group function object is being enqueued on.</td>
</tr>
<tr>
<td><code>template &lt;typename Param&gt; typename Param::return_type get_backend_info() const</code></td>
<td>Queries this SYCL queue for SYCL backend-specific information requested by the template parameter <code>Param</code>. The type alias <code>Param::return_type</code> must be defined in accordance with the SYCL backend specification. Must throw an exception with the <code>errc::backend_mismatch</code> error code if the SYCL backend that corresponds with <code>Param</code> is different from the SYCL backend that is associated with this queue.</td>
</tr>
</tbody>
</table>

### 4.6.5.2. Queue shortcut functions

Queue shortcut functions are member functions of the `queue` class that implicitly create a command group with an implicit command group handler consisting of a single command, a call to the member function of the handler object with the same signature (e.g. `queue::single_task` will call `handler::single_task` with the same arguments), and submit the command group. The main signature difference comes from the return type: member functions of the handler return `void`, whereas corresponding queue shortcut functions return an `event` object that represents the submitted command group. Queue shortcuts can additionally take a list of events to wait on, as if passing the event list to `handler::depends_on` for the implicit command group.

The full list of queue shortcuts is defined in Table 29. The list of handler member functions is defined in Table 132.

It is not allowed to capture accessors into the implicitly created command group. If a queue shortcut function launches a kernel (via `single_task` or `parallel_for`), only USM pointers are allowed inside such kernels. However, queue shortcuts that perform non-kernel operations can be provided with a valid placeholder accessor as an argument. In that case there is an additional step performed: the implicit command group handler calls `handler::require` on each accessor passed in as a function argument.

An example of using queue shortcuts is shown below.

```cpp
1 class MyKernel;
2
3 queue myQueue;
```
auto usmPtr = malloc_device<int>(1024, myQueue); // USM pointer
int* data = /* pointer to some data */;
buffer buf { data, 1024 }; // Placeholder accessor
accessor acc { buf }; // Placeholder accessor

// Queue shortcut for a kernel invocation
myQueue.single_task<MyKernel>({=}) {  // Allowed to use USM pointers,
   // not allowed to use accessors
   usmPtr[0] = 0;
};
myQueue.copy(data, acc);

Table 29. Queue shortcut functions

<table>
<thead>
<tr>
<th>Function Definition</th>
<th>Function Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename KernelName, typename KernelType&gt; event single_task(const KernelType&amp; kernelFunc)</td>
<td>Kernel function</td>
<td>Equivalent to submitting a command-group containing handler::single_task(kernelFunc).</td>
</tr>
<tr>
<td>template &lt;typename KernelName, typename KernelType&gt; event single_task(event depEvent, const KernelType&amp; kernelFunc)</td>
<td>Kernel function</td>
<td>Equivalent to submitting a command-group containing handler::depend_s_on(depEvent) and handler::single_task(kernelFunc).</td>
</tr>
<tr>
<td>template &lt;typename KernelName, typename KernelType&gt; event single_task(const std::vector&lt;event&gt;&amp; depEvents, const KernelType&amp; kernelFunc)</td>
<td>Kernel function</td>
<td>Equivalent to submitting a command-group containing handler::depend_s_on(depEvents) and handler::single_task(kernelFunc).</td>
</tr>
<tr>
<td>template &lt;typename KernelName, int Dimensions, typename Rest&gt; event parallel_for(range&lt;Dimensions&gt; numWorkItems, Rest&amp;... rest)</td>
<td>Kernel function</td>
<td>Equivalent to submitting a command-group containing handler::parallel_for(numWorkItems, rest).</td>
</tr>
<tr>
<td>template &lt;typename KernelName, int Dimensions, typename Rest&gt; event parallel_for(range&lt;Dimensions&gt; numWorkItems, event depEvent, Rest&amp;... rest)</td>
<td>Kernel function</td>
<td>Equivalent to submitting a command-group containing handler::depend_s_on(depEvent) and handler::parallel_for(numWorkItems, rest).</td>
</tr>
<tr>
<td>Function Definition</td>
<td>Function Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `template <typename KernelName, int Dimensions, typename... Rest>
  event parallel_for(range<Dimensions> numWorkItems, const std::vector<event>& depEvents, Rest... rest)` | Kernel function | Equivalent to submitting a command-group containing `handler::depend_on(depEvents)` and `handler::parallel_for(numWorkItems, rest)`. |
| `template <typename KernelName, int Dimensions, typename... Rest>
  event parallel_for(nd_range<Dimensions> executionRange, Rest... rest)` | Kernel function | Equivalent to submitting a command-group containing `handler::parallel_for(executionRange, rest)`. |
| `template <typename KernelName, int Dimensions, typename... Rest>
  event parallel_for(nd_range<Dimensions> executionRange, event depEvent, Rest... rest)` | Kernel function | Equivalent to submitting a command-group containing `handler::depend_on(depEvent)` and `handler::parallel_for(executionRange, rest)`. |
| `template <typename KernelName, int Dimensions, typename... Rest>
  event parallel_for(nd_range<Dimensions> executionRange, const std::vector<event>& depEvents, Rest... rest)` | Kernel function | Equivalent to submitting a command-group containing `handler::depend_on(depEvents)` and `handler::parallel_for(executionRange, rest)`. |
<p>| <code>event memcpy(void* dest, const void* src, size_t numBytes)</code> | USM | Equivalent to submitting a command-group containing <code>handler::memcpy(dest, src, numBytes)</code>. |
| <code>event memcpy(void* dest, const void* src, size_t numBytes, event depEvent)</code> | USM | Equivalent to submitting a command-group containing <code>handler::depend_on(depEvent)</code> and <code>handler::memcpy(dest, src, numBytes)</code>. |
| <code>event memcpy(void* dest, const void* src, size_t numBytes, const std::vector&lt;event&gt;&amp; depEvents)</code> | USM | Equivalent to submitting a command-group containing <code>handler::depend_on(depEvents)</code> and <code>handler::memcpy(dest, src, numBytes)</code>. |</p>
<table>
<thead>
<tr>
<th>Function Definition</th>
<th>Function Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename T&gt; event copy(const T* src, T* dest, size_t count)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::copy(src, dest, count)</code></td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event copy(const T* src, T* dest, size_t count, event depEvent)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::depend_on(depEvent) and handler::copy(src, dest, count)</code></td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event copy(const T* srct, T* dest, size_t count, const std::vector&lt;event&gt;&amp; depEvents)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::depend_on(depEvents) and handler::copy(src, dest, count)</code></td>
</tr>
<tr>
<td><code>event memset(void* ptr, int value, size_t numBytes)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::memset(ptr, value, numBytes)</code></td>
</tr>
<tr>
<td><code>event memset(void* ptr, int value, size_t numBytes, event depEvent)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::depend_on(depEvent) and handler::memset(ptr, value, numBytes)</code></td>
</tr>
<tr>
<td><code>event memset(void* ptr, int value, size_t numBytes, const std::vector&lt;event&gt;&amp; depEvents)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::depend_on(depEvents) and handler::memset(ptr, value, numBytes)</code></td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event fill(void* ptr, const T&amp; pattern, size_t count)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::fill(ptr, pattern, count)</code></td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; event fill(void* ptr, const T&amp; pattern, size_t count, event depEvent)</code></td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing <code>handler::depend_on(depEvent) and handler::fill(ptr, pattern, count)</code></td>
</tr>
<tr>
<td>Function Definition</td>
<td>Function Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>template &lt;typename T&gt; event fill(void* ptr, const T&amp; pattern, size_t count,</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::depends_on(depEvents) and handler::fill(ptr, pattern, count).</td>
</tr>
<tr>
<td>const std::vector&lt;event&gt;&amp; depEvents)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>event prefetch(void* ptr, size_t numBytes)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::prefetch(ptr, numBytes).</td>
</tr>
<tr>
<td>event prefetch(void* ptr, size_t numBytes, event depEvent)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::depends_on(depEvent) and handler::prefetch(ptr, numBytes).</td>
</tr>
<tr>
<td>event prefetch(void* ptr, size_t numBytes, const std::vector&lt;event&gt;&amp; depEvents)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::depends_on(depEvents) and handler::prefetch(ptr, numBytes).</td>
</tr>
<tr>
<td>event mem_advise(void* ptr, size_t numBytes, int advice)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::mem_advise(ptr, numBytes, advice).</td>
</tr>
<tr>
<td>event mem_advise(void* ptr, size_t numBytes, int advice, event depEvent)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::depends_on(depEvent) and handler::mem_advise(ptr, numBytes, advice).</td>
</tr>
<tr>
<td>event mem_advise(void* ptr, size_t numBytes, int advice, const std::vector&lt;event&gt;&amp; depEvents)</td>
<td>USM</td>
<td>Equivalent to submitting a command-group containing handler::depends_on(depEvents) and handler::mem_advise(ptr, numBytes, advice).</td>
</tr>
<tr>
<td>Function Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(src)` and `handler::copy(src, dest)`.

```cpp
template<typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt,
          access::placeholder IsSrcPlaceholder, typename DestT>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt,
          IsSrcPlaceholder> src,
          std::shared_ptr<DestT> dest);
```

| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(dest)` and `handler::copy(src, dest)`.

```cpp
template<typename SrcT, typename DestT, int DestDims,
          access_mode DestMode, target DestTgt, access::placeholder IsDestPlaceholder>
event copy(std::shared_ptr<SrcT> src,
          accessor<DestT, DestDims, DestMode, DestTgt,
          IsDestPlaceholder> dest);
```

| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(src)` and `handler::copy(src, dest)`.

```cpp
template<typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt,
          access::placeholder IsSrcPlaceholder, typename DestT>
event copy(accessor<SrcT, SrcDims, SrcMode, SrcTgt,
          IsSrcPlaceholder> src,
          DestT* dest);
```

| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(dest)` and `handler::copy(src, dest)`.

```cpp
template<typename SrcT, typename DestT, int DestDims,
          access_mode DestMode, target DestTgt, access::placeholder IsDestPlaceholder>
event copy(const SrcT* src,
          accessor<DestT, DestDims, DestMode, DestTgt,
          IsDestPlaceholder> dest);
```

| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(src)`, `handler::require(dest)` and `handler::copy(src, dest)`.

```cpp
template<typename SrcT, typename DestT, int DestDims,
          access_mode DestMode, target DestTgt,
          access::placeholder IsDestPlaceholder>
event copy(
          accessor<SrcT, SrcDims, SrcMode, SrcTgt,
          IsSrcPlaceholder> src,
          accessor<DestT, DestDims, DestMode, DestTgt,
          IsDestPlaceholder> dest);
```

| **Explicit copy** | Equivalent to submitting a command-group containing `handler::require(acc)` and `handler::update_host(acc)`.

```cpp
template<typename T, int Dims, access_mode Mode, target Tgt,
          access::placeholder IsPlaceholder>
event update_host(accessor<T, Dims, Mode, Tgt, IsPlaceholder> acc);
```
4.6.5.3. Queue information descriptors

A `queue` can be queried for information using the `get_info` member function of the `queue` class, specifying one of the info parameters in `info::queue`. The possible values for each info parameter and any restriction are defined in the specification of the SYCL backend associated with the `queue`. All info parameters in `info::queue` are specified in Table 30 and the synopsis for `info::queue` is described in Section A.4.

**Table 30. Queue information descriptors**

<table>
<thead>
<tr>
<th>Queue Descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::queue::context</code></td>
<td>context</td>
<td>Returns the SYCL <code>context</code> associated with this SYCL <code>queue</code>.</td>
</tr>
<tr>
<td><code>info::queue::device</code></td>
<td>device</td>
<td>Returns the SYCL <code>device</code> associated with this SYCL <code>queue</code>.</td>
</tr>
</tbody>
</table>

4.6.5.4. Queue properties

The properties that can be provided when constructing the SYCL `queue` class are describe in Table 31.

**Table 31. Properties supported by the SYCL `queue` class**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>property::queue::enable_profiling</code></td>
<td>The <code>enable_profiling</code> property adds the requirement that the SYCL runtime must capture profiling information for the command groups that are submitted from this SYCL <code>queue</code> and provide said information via the SYCL <code>event</code> class <code>get_profiling_info</code> member function. If the queue's associated device does not have <code>aspect::queue_profiling</code>, passing this property to the queue's constructor causes the constructor to throw a synchronous exception with the <code>errc::feature_not_supported</code> error code.</td>
</tr>
</tbody>
</table>
The in_order property adds the requirement that a SYCL queue provides in-order semantics whereby commands submitted to said queue are executed in the order in which they are submitted. Commands submitted in this fashion can be viewed as-if having an implicit dependence on the previous command submitted to that queue. Using the in_order property makes no guarantees about the ordering of commands submitted to different queues with respect to each other.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::queue::in_order</td>
<td>The in_order property adds the requirement that a SYCL queue provides in-order semantics whereby commands submitted to said queue are executed in the order in which they are submitted. Commands submitted in this fashion can be viewed as-if having an implicit dependence on the previous command submitted to that queue. Using the in_order property makes no guarantees about the ordering of commands submitted to different queues with respect to each other.</td>
</tr>
</tbody>
</table>

The constructors of the queue property classes are listed in Table 32.

Table 32. Constructors of the queue property classes

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::queue::enable_profiling::enable_profiling()</td>
<td>Constructs a SYCL enable_profiling property instance.</td>
</tr>
<tr>
<td>property::queue::in_order::in_order()</td>
<td>Constructs a SYCL in_order property instance.</td>
</tr>
</tbody>
</table>

4.6.5.5. Queue error handling

Queue errors come in two forms:

- **Synchronous Errors** are those that we would expect to be reported directly at the point of waiting on an event, and hence waiting for a queue to complete, as well as any immediate errors reported by enqueuing work onto a queue. Such errors are reported through C++ exceptions.

- **Asynchronous errors** are those that are produced or detected after associated host API calls have returned (so can't be thrown as exceptions by the API call), and that are handled by an async_handler through which the errors are reported. Handling of asynchronous errors from a queue occurs at specific times, as described by Section 4.13.

Note that if there are asynchronous errors to be processed when a queue is destructed, the handler is called and this might delay or block the destruction, according to the behavior of the handler.

4.6.6. Event class

An event in SYCL is an object that represents the status of an operation that is being executed by the SYCL runtime.

Typically in SYCL, data dependency and execution order is handled implicitly by the SYCL runtime. However, in some circumstances developers want fine grain control of the execution, or want to retrieve properties of a command that is running.

Note that, although an event represents the status of a particular operation, the dependencies of a certain event can be used to keep track of multiple steps required to synchronize said operation.

A SYCL event is returned by the submission of a command group. The dependencies of the event
returned via the submission of the command group are the implementation-defined commands associated with the \textit{command group} execution.

The SYCL \texttt{event} class provides the common reference semantics (see Section 4.5.2).

The constructors and member functions of the SYCL \texttt{event} class are listed in Table 33 and Table 34, respectively. The additional common special member functions and common member functions are listed in Table 7 and Table 8, respectively.

\begin{verbatim}
namespace sycl {
    class event {
    public:
        event();

        /* -- common interface members -- */

        backend get_backend() const noexcept;
        std::vector<event> get_wait_list();
        void wait();

        static void wait(const std::vector<event>& eventList);
        void wait_and_throw();

        static void wait_and_throw(const std::vector<event>& eventList);

        template <typename Param> typename Param::return_type get_info() const;
        template <typename Param> typename Param::return_type get_backend_info() const;
        template <typename Param> typename Param::return_type get_profiling_info() const;
    }
}
\end{verbatim}

\textit{Table 33. Constructors of the event class}
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>event()</code></td>
<td>Constructs an event that is immediately ready. The event has no dependencies and no associated commands. Waiting on this event will return immediately and querying its status will return <code>info::event_command_status::complete</code>. The event is constructed as though it was created from a default-constructed queue. Therefore, its backend is the same as the backend from the default device.</td>
</tr>
</tbody>
</table>

### Table 34. Member functions for the event class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backend get_backend() const noexcept</code></td>
<td>Returns a backend identifying the SYCL backend associated with this event.</td>
</tr>
<tr>
<td><code>std::vector&lt;event&gt; get_wait_list()</code></td>
<td>Return the list of events that this event waits for in the dependence graph. Only direct dependencies are returned, and not transitive dependencies that direct dependencies wait on. Whether already completed events are included in the returned list is implementation-defined.</td>
</tr>
<tr>
<td><code>void wait()</code></td>
<td>Wait for the event and the command associated with it to complete.</td>
</tr>
<tr>
<td><code>void wait_and_throw()</code></td>
<td>Wait for the event and the command associated with it to complete. Any unconsumed asynchronous errors from any context that the event was waiting on executions from will be passed to the async_handler associated with the context. If no user defined async_handler is associated with the context, then an implementation-defined default async_handler is called to handle any errors, as described in Section 4.13.1.2.</td>
</tr>
<tr>
<td><code>static void wait(const std::vector&lt;event&gt;&amp; eventList)</code></td>
<td>Synchronously wait on a list of events.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>static void wait_and_throw(const std::vector&lt;event&gt;&amp; eventList)</td>
<td>Synchronously wait on a list of events. Any unconsumed asynchronous errors</td>
</tr>
<tr>
<td></td>
<td>from any context that the event was waiting on executions from will be</td>
</tr>
<tr>
<td></td>
<td>passed to the async_handler associated with the context. If no user defined</td>
</tr>
<tr>
<td></td>
<td>async_handler is associated with the context, then an implementation-defined</td>
</tr>
<tr>
<td></td>
<td>default async_handler is called to handle any errors, as described in</td>
</tr>
<tr>
<td></td>
<td>Section 4.13.1.2.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_info() const</td>
<td>Queries this SYCL event for information requested by the template parameter</td>
</tr>
<tr>
<td></td>
<td>Param. The type alias Param::return_type must be defined in accordance with</td>
</tr>
<tr>
<td></td>
<td>the info parameters in Table 35 to facilitate returning the type associated</td>
</tr>
<tr>
<td></td>
<td>with the Param parameter.</td>
</tr>
<tr>
<td>template &lt;typename Param&gt; typename Param::return_type get_backend_info() const</td>
<td>Queries this SYCL event for SYCL backend-specific information requested by</td>
</tr>
<tr>
<td></td>
<td>the template parameter Param. The type alias Param::return_type must be</td>
</tr>
<tr>
<td></td>
<td>defined in accordance with the SYCL backend specification. Must throw an</td>
</tr>
<tr>
<td></td>
<td>exception with the errc::backend_mismatch error code if the SYCL backend</td>
</tr>
<tr>
<td></td>
<td>that corresponds with Param is different from the SYCL backend that is</td>
</tr>
<tr>
<td></td>
<td>associated with this event.</td>
</tr>
</tbody>
</table>
### Member function

```cpp
template<typename Param>
typename Param::return_type
get_profiling_info() const
```

### Description

Queries this SYCL event for profiling information requested by the parameter `Param`. If the requested profiling information is unavailable when `get_profiling_info` is called due to incompletion of command groups associated with the event, then the call to `get_profiling_info` will block until the requested profiling information is available. An example is asking for `info::event_profiling::command_end` when the associated command group action has yet to finish execution. Calls to `get_profiling_info` must throw an exception with the `errc::invalid` error code if the SYCL queue which submitted the command group this SYCL event is associated with was not constructed with the `property::queue::enable_profiling` property. The type alias `Param::return_type` must be defined in accordance with the info parameters in Table 37 to facilitate returning the type associated with the `Param` parameter.

### 4.6.6.1. Event information and profiling descriptors

An event can be queried for information using the `get_info` member function of the `event` class, specifying one of the info parameters in `info::event`. The possible values for each info parameter and any restrictions are defined in the specification of the SYCL backend associated with the event. All info parameters in `info::event` are specified in Table 35 and the synopsis for `info::event` is described in Section A.6.

#### Table 35. Event class information descriptors

<table>
<thead>
<tr>
<th>Event Descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::event::command_execution_status</code></td>
<td><code>info::event_command_status</code></td>
<td>Returns the event status of the command group and contained action (e.g. kernel invocation) associated with this SYCL event.</td>
</tr>
</tbody>
</table>

The `info::event::command_execution_status` query returns one of the values defined in Table 36.

#### Table 36. Event command status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::event_command_status::submitted</code></td>
<td>Indicates that the command has been submitted to the SYCL queue but has not yet started running on the device.</td>
</tr>
</tbody>
</table>
An event can be queried for profiling information using the `get_profiling_info` member function of the event class, specifying one of the profiling info parameters enumerated in `info::event_profiling`. The possible values for each info parameter and any restrictions are defined in the specification of the SYCL backend associated with the event. All info parameters in `info::event_profiling` are specified in Table 37 and the synopsis for `info::event_profiling` is described in Section A.6.

Each profiling descriptor returns a 64-bit timestamp that represents the number of nanoseconds that have elapsed since some implementation-defined timebase. All events that share the same backend are guaranteed to share the same timebase, therefore the difference between two timestamps from the same backend yields the number of nanoseconds that have elapsed between those events.

<table>
<thead>
<tr>
<th>Event information profiling descriptor</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::event_profiling::command_submit</code></td>
<td><code>uint64_t</code></td>
<td>Returns a timestamp telling when the associated command group was submitted to the queue. This is always some time after the command group function object returns and before the associated call to <code>queue::submit</code> returns.</td>
</tr>
<tr>
<td><code>info::event_profiling::command_start</code></td>
<td><code>uint64_t</code></td>
<td>Querying this profiling descriptor blocks until the event's state becomes either <code>info::event_command_status::running</code> or <code>info::event_command_status::complete</code>. The returned timestamp tells when the action associated with the command group (e.g. kernel invocation) started executing on the device. For any given event, this timestamp is always greater than or equal to the <code>info::event_profiling::command_submit</code> timestamp. Implementations are encouraged to return a timestamp that is as close as possible to the point when the action starts running on the device, but there is no specific accuracy that is guaranteed.</td>
</tr>
<tr>
<td><code>info::event_profiling::command_end</code></td>
<td><code>uint64_t</code></td>
<td>Querying this profiling descriptor blocks until the event's state becomes <code>info::event_command_status::complete</code>. The returned timestamp tells when the action associated with the command group (e.g. kernel invocation) finished executing on the device. For any given event, this timestamp is always greater than or equal to the <code>info::event_profiling::command_start</code> timestamp.</td>
</tr>
</tbody>
</table>
4.7. Data access and storage in SYCL

In SYCL, when using buffers and images, data storage and access are handled by separate classes. Buffers and images handle storage and ownership of the data, whereas accessors handle access to the data. Buffers and images in SYCL can be bound to more than one device or context, including across different SYCL backends. They also handle ownership of the data, while allowing exception handling for blocking and non-blocking data transfers. Accessors manage data transfers between the host and all of the devices in the system, as well as tracking of data dependencies.

Zero-sized buffers are permitted. In this case, the buffer still handles ownership as normal, but does not need to store the zero-sized data itself.

When using USM allocations, data storage is managed by USM allocation functions, and data access is via pointers. See Section 4.8 for greater detail.

4.7.1. Host allocation

A SYCL runtime may need to allocate temporary objects on the host to handle some operations (such as copying data from one context to another). Allocation on the host is managed using an allocator object, following the standard C++ allocator class definition. The default allocator for memory objects is implementation-defined, but the user can supply their own allocator class.

```cpp
1 {
2     buffer<int, 1, UserDefinedAllocator<int>> b(d);
3 }
```

When an allocator returns a `nullptr`, the runtime cannot allocate data on the host. Note that in this case the runtime will raise an error if it requires host memory but it is not available (e.g. when moving data across SYCL backend contexts).

In some cases, the implementation may retain a copy of the allocator object even after the buffer is destroyed. For example, this can happen when the buffer object is destroyed before commands using accessors to the buffer have completed. Therefore, the application must be prepared for calls to the allocator even after the buffer is destroyed.

If the application needs to know when the implementation has destroyed all copies of the allocator, it can maintain a reference count within the allocator.

The definition of allocators extends the current functionality of SYCL, ensuring that users can define allocator functions for specific hardware or certain complex shared memory mechanisms (e.g. NUMA), and improves interoperability with STL-based libraries (e.g., Intel’s TBB provides an allocator).

4.7.1.1. Default allocators

A default allocator is always defined by the implementation. For allocations greater than size zero, it is guaranteed to return non-`nullptr` and new memory positions every call. The default allocator for const buffers will remove the const-ness of the type (therefore, the default allocator for a buffer of type `const int` will be an `Allocator<int>`). This implies that host accessors will not synchronize with the pointer given by the user in the buffer/image constructor, but will use the memory returned by the `Allocator` itself for that purpose. The user can implement an allocator that returns the same address as the one passed in the buffer constructor, but it is the responsibility of the user to handle the potential race conditions.

Table 38. SYCL Default Allocators
### Allocators

<table>
<thead>
<tr>
<th>Allocators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>template &lt;class T&gt; buffer_allocator</strong></td>
<td>It is the default buffer allocator used by the runtime, when no allocator is defined by the user. Meets the C++ named requirement <code>Allocator</code>. A buffer of data type <code>const T</code> uses <code>buffer_allocator&lt;T&gt;</code> by default.</td>
</tr>
<tr>
<td><strong>image_allocator</strong></td>
<td>It is the default allocator used by the runtime for the SYCL <code>unsampled_image</code> and <code>sampled_image</code> classes when no allocator is provided by the user. The <code>image_allocator</code> is required to allocate in elements of <code>std::byte</code>.</td>
</tr>
</tbody>
</table>

See Section 4.7.5 for details on manual host-device synchronization.

### 4.7.2. Buffers

The `buffer` class defines a shared array of one, two or three dimensions that can be used by the SYCL `kernel` and has to be accessed using `accessor` classes. Buffers are templated on both the type of their data, and the number of dimensions that the data is stored and accessed through.

A `buffer` does not map to only one underlying backend object, and all SYCL backend memory objects may be temporary for use within a command group on a specific device.

The underlying data type of a buffer `T` must be `device copyable` as defined in Section 3.13.1. Some overloads of the `buffer` constructor initialize the buffer contents by copying objects from host memory while other overloads construct the buffer without copying objects from the host. For the overloads that do not copy host objects, the initial state of the objects in the buffer depends on whether `T` is an implicit-lifetime type (as defined in the C++ core language). If `T` is an implicit-lifetime type, objects of that type are implicitly created in the buffer with indeterminate values. For other types, these constructor overloads merely allocate uninitialized memory, and the application is responsible for constructing objects by calling `placement-new` and for destroying them later by manually calling the object’s destructor.

For the overloads that do copy objects from host memory, the `hostData` pointer must point to at least `N` bytes of memory where `N` is `sizeof(T) * bufferRange.size()`. If `N` is zero, `hostData` is permitted to be a null pointer.

A SYCL `buffer` can construct an instance of a SYCL `buffer` that reinterprets the original SYCL `buffer` with a different type, dimensionality and range using the member function `reinterpret`. The reinterpreted SYCL `buffer` that is constructed must behave as though it were a copy of the SYCL `buffer` that constructed it (see Section 4.5.2) with the exception that the type, dimensionality and range of the reinterpreted SYCL `buffer` must reflect the type, dimensionality and range specified when calling the `reinterpret` member function. By extension of this, the class member types `value_type`, `reference` and `const_reference`, and the member functions `get_range()` and `size()` of the reinterpreted SYCL `buffer` must reflect the new type, dimensionality and range. The data that the original SYCL `buffer` and the reinterpreted SYCL `buffer` manage remains unaffected, though the representation of the data when accessed through the reinterpreted SYCL `buffer` may alter to reflect the new type, dimensionality and range. It is important to note that a reinterpreted SYCL `buffer` is a copy of the original SYCL `buffer` only, and not a new SYCL `buffer`. Constructing more than one SYCL `buffer` managing the same host pointer is still undefined behavior.

The SYCL `buffer` class template provides the common reference semantics (see Section 4.5.2).
4.7.2.1. Buffer interface

The constructors and member functions of the SYCL buffer class template are listed in Table 39 and Table 40, respectively. The additional common special member functions and common member functions are listed in Table 7 and Table 8, respectively.

Each constructor takes as the last parameter an optional SYCL property_list to provide properties to the SYCL buffer.

The SYCL buffer class template takes a template parameter AllocatorT for specifying an allocator which is used by the SYCL runtime when allocating temporary memory on the host. If no template argument is provided, then the default allocator for the SYCL buffer class buffer_allocator<T> will be used (see Section 4.7.1.1).

```cpp
namespace sycl {
    namespace property {
        namespace buffer {
            class use_host_ptr {
                public:
                use_host_ptr() = default;
            };

            class use_mutex {
                public:
                use_mutex(std::mutex& mutexRef);
                std::mutex* get_mutex_ptr() const;
            };

            class context_bound {
                public:
                context_bound(context boundContext);
                context get_context() const;
            };
        } // namespace buffer
    } // namespace property

    template <typename T, int Dimensions = 1>
    typename AllocatorT = buffer_allocator<std::remove_const_t<T>>

class buffer {
    using value_type = T;
    using reference = value_type&;
    using const_reference = const value_type&;
    using allocator_type = AllocatorT;

    buffer(const range<Dimensions>& bufferRange,
           const property_list& propList = {});

    buffer(const range<Dimensions>& bufferRange, AllocatorT allocator,
           const property_list& propList = {});

    buffer(T* hostData, const range<Dimensions>& bufferRange,
           const property_list& propList = {});
```


buffer(T* hostData, const range<Dimensions>& bufferRange, 
    AllocatorT allocator, const property_list& propList = {});

buffer(const T* hostData, const range<Dimensions>& bufferRange, 
    const property_list& propList = {});

buffer(const T* hostData, const range<Dimensions>& bufferRange, 
    AllocatorT allocator, const property_list& propList = {});

/* Available only if Container is a contiguous container:
- std::data(container) and std::size(container) are well formed
- return type of std::data(container) is convertible to T*
  and Dimensions == 1 */

template<typename Container>
buffer(Container& container, AllocatorT allocator, 
    const property_list& propList = {});

/* Available only if Container is a contiguous container:
- std::data(container) and std::size(container) are well formed
- return type of std::data(container) is convertible to T*
  and Dimensions == 1 */

template<typename Container>
buffer(const std::shared_ptr<T>& hostData, 
    const range<Dimensions>& bufferRange, AllocatorT allocator, 
    const property_list& propList = {});

buffer(const std::shared_ptr<T*>& hostData, 
    const range<Dimensions>& bufferRange, AllocatorT allocator, 
    const property_list& propList = {});

buffer(const std::shared_ptr<T[]>& hostData, 
    const range<Dimensions>& bufferRange, AllocatorT allocator, 
    const property_list& propList = {});

buffer(const std::shared_ptr<T[]>& hostData, 
    const range<Dimensions>& bufferRange, 
    const property_list& propList = {});

template<class InputIterator>
buffer<T, 1>(InputIterator first, InputIterator last, AllocatorT allocator, 
    const property_list& propList = {});

template<class InputIterator>
buffer<T, 1>(InputIterator first, InputIterator last, 
    const property_list& propList = {});

buffer(buffer& b, const id<Dimensions>& baseIndex, 
    const range<Dimensions>& subRange);

/* -- common interface members -- */

/* -- property interface members -- */

range<Dimensions> get_range() const;
size_t byte_size() const noexcept;
size_t size() const noexcept;

// Deprecated
size_t get_count() const;

// Deprecated
size_t get_size() const;

AllocatorT get_allocator() const;

template <access_mode Mode = access_mode::read_write,
           target Targ = target::device>
accessor<T, Dimensions, Mode, Targ> get_access(handler& commandGroupHandler);

// Deprecated
template <access_mode Mode>
accessor<T, Dimensions, Mode, target::host_buffer> get_access();

template <access_mode Mode = access_mode::read_write,
           target Targ = target::device>
accessor<T, Dimensions, Mode, Targ>
get_access(handler& commandGroupHandler, range<Dimensions> accessRange,
           id<Dimensions> accessOffset = {});

// Deprecated
template <access_mode Mode>
accessor<T, Dimensions, Mode, target::host_buffer>
get_access(range<Dimensions> accessRange, id<Dimensions> accessOffset = {});

template <typename... Ts> auto get_access(Ts...);

template <typename... Ts> auto get_host_access(Ts...);

template <typename Destination = std::nullptr_t>
void set_final_data(Destination finalData = nullptr);

void set_write_back(bool flag = true);

bool is_sub_buffer() const;

template <typename ReinterpretT, int ReinterpretDim>
buffer<ReinterpretT, ReinterpretDim,
       typename std::allocator_traits<AllocatorT>::template rebind_alloc<
          ReinterpretT>>
reinterpret(range<ReinterpretDim> reinterpretRange) const;

// Only available when ReinterpretDim == 1
// or when (ReinterpretDim == Dimensions) &&
// (sizeof(ReinterpretT) == sizeof(T))
template <typename ReinterpretT, int ReinterpretDim = Dimensions>
buffer<ReinterpretT, ReinterpretDim,
       typename std::allocator_traits<AllocatorT>::template rebind_alloc<
          ReinterpretT>>
155  reinterpret() const;
156 );
157
158 // Deduction guides
159 template <class InputIterator, class AllocatorT>
160 buffer(InputIterator, InputIterator, AllocatorT, const property_list& = {})
161   -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1,
162            AllocatorT>;
163
164 template <class InputIterator>
165 buffer(InputIterator, InputIterator, const property_list& = {})
166   -> buffer<typename std::iterator_traits<InputIterator>::value_type, 1>;
167
168 template <class T, int Dimensions, class AllocatorT>
169 buffer(T*, const range<Dimensions>&, AllocatorT,
170         const property_list& = {}) -> buffer<T, Dimensions, AllocatorT>;
171
172 template <class T, int Dimensions>
173 buffer(T*, const range<Dimensions>&, const property_list& = {})
174   -> buffer<T, Dimensions>;
175
176 template <class Container, class AllocatorT>
177 buffer(Container&, AllocatorT, const property_list& = {})
178   -> buffer<typename Container::value_type, 1, AllocatorT>;
179
180 template <class Container>
181 buffer(Container&, const property_list& = {})
182   -> buffer<typename Container::value_type, 1>;
183
184 } // namespace sycl

Table 39. Constructors of the buffer class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer(const range&lt;Dimensions&gt;&amp; bufferRange, const property_list&amp; propList = {})</code></td>
<td>Construct a SYCL buffer instance with uninitialized memory. The constructed SYCL buffer will use a default constructed AllocatorT when allocating memory on the host. The range of the constructed SYCL buffer is specified by the bufferRange parameter provided. Data is not written back to the host on destruction of the buffer unless the buffer has a valid non-null pointer specified via the member function set_final_data(). Zero or more properties can be provided to the constructed SYCL buffer via an instance of property_list.</td>
</tr>
</tbody>
</table>
## Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer(const range&lt;Dimensions&gt;&amp; bufferRange, AllocatorT allocator, const property_list&amp; propList = { })</code></td>
<td>Construct a SYCL buffer instance with uninitialized memory. The constructed SYCL buffer will use the allocator parameter provided when allocating memory on the host. The range of the constructed SYCL buffer is specified by the bufferRange parameter provided. Data is not written back to the host on destruction of the buffer unless the buffer has a valid non-null pointer specified via the member function set_final_data(). Zero or more properties can be provided to the constructed SYCL buffer via an instance of property_list.</td>
</tr>
<tr>
<td><code>buffer(T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange, const property_list&amp; propList = { })</code></td>
<td>Construct a SYCL buffer instance with the hostData parameter provided. The buffer is initialized with the memory specified by hostData, and the buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL buffer will use a default constructed AllocatorT when allocating memory on the host. The range of the constructed SYCL buffer is specified by the bufferRange parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of property_list.</td>
</tr>
<tr>
<td><code>buffer(T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange, AllocatorT allocator, const property_list&amp; propList = { })</code></td>
<td>Construct a SYCL buffer instance with the hostData parameter provided. The buffer is initialized with the memory specified by hostData, and the buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL buffer will use the allocator parameter provided when allocating memory on the host. The range of the constructed SYCL buffer is specified by the bufferRange parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of property_list.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer(const T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange, const property_list&amp; propList = {})</code></td>
<td>Construct a SYCL buffer instance with the <code>hostData</code> parameter provided. The buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL buffer will use a default constructed AllocatorT when allocating memory on the host. The host address is <code>const T</code>, so the host accesses can be read-only. However, the typename <code>T</code> is not const so the device accesses can be both read and write accesses. Since the <code>hostData</code> is const, this buffer is only initialized with this memory and there is no write back after its destruction, unless the buffer has another valid non-null final data address specified via the member function <code>set_final_data()</code> after construction of the buffer. The range of the constructed SYCL buffer is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
Construct a SYCL buffer instance with the `hostData` parameter provided. The buffer assumes exclusive access to this memory for the duration of its lifetime.

The constructed SYCL buffer will use the `allocator` parameter provided when allocating memory on the host.

The host address is `const T`, so the host accesses can be read-only. However, the `typename T` is not `const` so the device accesses can be both read and write accesses. Since, the `hostData` is const, this buffer is only initialized with this memory and there is no write back after its destruction, unless the buffer has another valid non-null final data address specified via the member function `set_final_data()` after construction of the buffer.

The range of the constructed SYCL buffer is specified by the `bufferRange` parameter provided.

Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`.  

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer(const T* hostData, const range&lt;Dimensions&gt;&amp; bufferRange, AllocatorT allocator, const property_list&amp; propList = {})</code></td>
<td>Construct a SYCL buffer instance with the <code>hostData</code> parameter provided. The buffer assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL buffer will use the <code>allocator</code> parameter provided when allocating memory on the host. The host address is <code>const T</code>, so the host accesses can be read-only. However, the <code>typename T</code> is not <code>const</code> so the device accesses can be both read and write accesses. Since, the <code>hostData</code> is const, this buffer is only initialized with this memory and there is no write back after its destruction, unless the buffer has another valid non-null final data address specified via the member function <code>set_final_data()</code> after construction of the buffer. The range of the constructed SYCL buffer is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
## Constructor

```cpp
template <typename Container>
buffer(Container& container,
        const property_list& propList = {}) {
```

### Description

Construct a one dimensional SYCL buffer instance from the elements starting at `std::data(container)` and containing `std::size(container)` number of elements. The buffer is initialized with the contents of `container`, and the buffer assumes exclusive access to `container` for the duration of its lifetime.

Data is written back to `container` before the completion of buffer destruction if the return type of `std::data(container)` is not `const`.

The constructed SYCL buffer will use a default constructed Allocator when allocating memory on the host.

Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`.

This constructor is only defined for a `buffer` parameterized with `Dimensions == 1`, and when `std::data(container)` is convertible to `T*`. 
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| template <typename Container>
  buffer(Container& container, AllocatorT allocator,
          const property_list& propList = {}) |
  Construct a one dimensional SYCL buffer instance from the elements starting at `std::data(container)` and containing `std::size(container)` number of elements. The buffer is initialized with the contents of `container`, and the buffer assumes exclusive access to `container` for the duration of its lifetime.
  Data is written back to `container` before the completion of buffer destruction if the return type of `std::data(container)` is not `const`.
  The constructed SYCL buffer will use the `allocator` parameter provided when allocating memory on the host.
  Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`.
  This constructor is only defined for a buffer parameterized with `Dimensions == 1`, and when `std::data(container)` is convertible to `T*`. |
### Constructor

| buffer(const std::shared_ptr<T>& hostData, const range<Dimensions>& bufferRange, const property_list& propList = {}) |

#### Description

When `hostData` is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostData` is empty, construct a SYCL buffer with uninitialized memory.

The constructed SYCL buffer will use a default constructed `AllocatorT` when allocating memory on the host. The range of the constructed SYCL buffer is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`. 
### Constructor

```cpp
buffer(const std::shared_ptr<T>& hostData,
       const range<Dimensions>& bufferRange,
       AllocatorT allocator,
       const property_list& propList = { })
```

### Description

When `hostData` is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostData` is empty, construct a SYCL buffer with uninitialized memory.

The constructed SYCL buffer will use the `allocator` parameter provided when allocating memory on the host. The range of the constructed SYCL buffer is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`. 
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>buffer&lt;</code> <code>std::shared_ptr&lt;T&gt;[]</code> <code>hostData,</code> <code>const</code> <code>range&lt;Dimensions&gt;[]</code> <code>bufferRange,</code> <code>const</code> <code>property_list[]</code> <code>propList = {}&gt;</code></td>
<td>When <code>hostData</code> is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns. When <code>hostData</code> is empty, construct a SYCL buffer with uninitialized memory. The constructed SYCL buffer will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The range of the constructed SYCL buffer is specified by the <code>bufferRange</code> parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
### Constructor

| buffer(const std::shared_ptr<T[]>& hostData,  
| const range<Dimensions>& bufferRange, 
| AllocatorT allocator,  
| const property_list& propList = {}) |

When `hostData` is not empty, construct a SYCL buffer with the contents of its stored pointer. The buffer assumes exclusive access to this memory for the duration of its lifetime. The buffer also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostData` is empty, construct a SYCL buffer with uninitialized memory.

The constructed SYCL buffer will use the `allocator` parameter provided when allocating memory on the host. The range of the constructed SYCL buffer is specified by the `bufferRange` parameter provided. Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`.

### Template

| template<typename InputIterator>  
| buffer(InputIterator first, InputIterator last,  
| const property_list& propList = {}) |

Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the `buffer` has a valid non-const iterator specified via the member function `set_final_data()`, data will be copied back to that iterator. The constructed SYCL buffer will use a default constructed `AllocatorT` when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL buffer via an instance of `property_list`. 
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| template `<typename InputIterator>
    buffer(InputIterator first, InputIterator last,  
    AllocatorT allocator = {},  
    const property_list& propList = {})` | Create a new allocated 1D buffer initialized from the given elements ranging from `first` up to one before `last`. The data is copied to an intermediate memory position by the runtime. Data is not written back to the same iterator set provided. However, if the `buffer` has a valid non-const iterator specified via the member function `set_final_data()`, data will be copied back to that iterator. The constructed SYCL `buffer` will use the `allocator` parameter provided when allocating memory on the host. Zero or more properties can be provided to the constructed SYCL `buffer` via an instance of `property_list`. |
### Constructor

| buffer(buffer & b, const id<Dimensions>& baseIndex,  
| const range<Dimensions>& subRange) |

**Description**

Create a new sub-buffer without allocation to have separate accessors later. `b` is the buffer with the real data, which must not be a sub-buffer. `baseIndex` specifies the origin of the sub-buffer inside the buffer `b`. `subRange` specifies the size of the sub-buffer. The sum of `baseIndex` and `subRange` in any dimension must not exceed the parent buffer (`b`) size (`bufferRange`) in that dimension, and an exception with the `errc::invalid` error code must be thrown if violated.

The offset and range specified by `baseIndex` and `subRange` together must represent a contiguous region of the original SYCL `buffer`.

If a non-contiguous region of a buffer is requested when constructing a sub-buffer, then an exception with the `errc::invalid` error code must be thrown.

The origin (based on `baseIndex`) of the sub-buffer being constructed must be a multiple of the memory base address alignment of each SYCL device which accesses data from the buffer. This value is retrievable via the SYCL device class `info` query `info::device::mem_base_addr_align`. Violating this requirement causes the implementation to throw an exception with the `errc::invalid` error code from the accessor constructor (if the accessor is not a placeholder) or from `handler::require()` (if the accessor is a placeholder). If the accessor is bound to a `command group` with a secondary queue, the sub-buffer's alignment must be compatible with both the primary queue's device and the secondary queue's device, otherwise this exception is thrown.

Must throw an exception with the `errc::invalid` error code if `b` is a sub-buffer.
Table 40. Member functions for the buffer class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>range&lt;Dimensions&gt; get_range() const</code></td>
<td>Return a range object representing the size of the buffer in terms of number of elements in each dimension as passed to the constructor.</td>
</tr>
<tr>
<td><code>size_t size() const noexcept</code></td>
<td>Returns the total number of elements in the buffer. Equal to <code>get_range()[0] * ... * get_range()[Dimensions-1]</code>.</td>
</tr>
<tr>
<td><code>size_t get_count() const</code></td>
<td>Returns the same value as <code>size()</code>. Deprecated.</td>
</tr>
<tr>
<td><code>size_t byte_size() const noexcept</code></td>
<td>Returns the size of the buffer storage in bytes. Equal to <code>size() * sizeof(T)</code>.</td>
</tr>
<tr>
<td><code>size_t get_size() const</code></td>
<td>Returns the same value as <code>byte_size()</code>. Deprecated.</td>
</tr>
<tr>
<td><code>AllocatorT get_allocator() const</code></td>
<td>Returns the allocator provided to the buffer.</td>
</tr>
<tr>
<td><code>template &lt;access_mode Mode = access_mode::read_write, target Targ = target::device&gt; accessor&lt;T, Dimensions, Mode, Targ&gt; get_access(handler&amp; commandGroupHandler)</code></td>
<td>Returns a valid accessor to the buffer with the specified access mode and target in the command group buffer. The value of target can be <code>target::device</code> or <code>target::constant_buffer</code>.</td>
</tr>
<tr>
<td><code>template &lt;access_mode Mode&gt; accessor&lt;T, Dimensions, Mode, target::host_buffer&gt; get_access()</code></td>
<td>Deprecated in SYCL 2020. Use <code>get_host_access()</code> instead. Returns a valid host accessor to the buffer with the specified access mode and target.</td>
</tr>
<tr>
<td><code>template &lt;access_mode Mode = access_mode::read_write, target Targ = target::device&gt; accessor&lt;T, Dimensions, Mode, Targ&gt; get_access(handler&amp; commandGroupHandler, range&lt;Dimensions&gt; accessRange, id accessOffset = {})</code></td>
<td>Returns a valid accessor to the buffer with the specified access mode and target in the command group buffer. The accessor is a ranged accessor, where the range starts at the given offset from the beginning of the buffer. The value of target can be <code>target::device</code> or <code>target::constant_buffer</code>. Throws an exception with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of the buffer in any dimension.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>template &lt;access_mode Mode&gt; accessor&lt;T, Dimensions, Mode, target::host_buffer&gt; get_access(range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset = {}})</code></td>
<td>Deprecated in SYCL 2020. Use <code>get_host_access()</code> instead. Returns a valid host accessor to the buffer with the specified access mode and target. The accessor is a ranged accessor, where the range starts at the given offset from the beginning of the buffer. The value of target can only be <code>target::host_buffer</code>. Throws an exception with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of the buffer in any dimension.</td>
</tr>
</tbody>
</table>
| `template <typename... Ts> auto get_access(Ts... args)` | Returns a valid accessor as if constructed via passing the buffer and all provided arguments to the accessor constructor. Possible implementation: 
```
return accessor{*this, args...};
``` |
| `template <typename... Ts> auto get_host_access(Ts... args)` | Returns a valid host_accessor as if constructed via passing the buffer and all provided arguments to the host_accessor constructor. Possible implementation: 
```
return host_accessor{*this, args...};
``` |
<p>| <code>template &lt;typename Destination = std::nullptr_t&gt; void set_final_data(Destination finalData = nullptr)</code> | The <code>finalData</code> points to where the outcome of all the buffer processing is going to be copied to at destruction time, if the buffer was involved with a write accessor. Destination can be either an output iterator or a <code>std::weak_ptr&lt;T&gt;</code>. Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied. In the case of a weak pointer, the output is not updated if the weak pointer has expired. If <code>Destination</code> is <code>std::nullptr_t</code>, then the copy back will not happen. |</p>
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void set_write_back(bool flag = true)</code></td>
<td>This member function allows dynamically forcing or canceling the write-back of the data of a buffer on destruction according to the value of <code>flag</code>. Forcing the write-back is similar to what happens during a normal write-back as described in Section 4.7.2.3 and Section 4.7.4. If there is nowhere to write-back, using this function does not have any effect.</td>
</tr>
<tr>
<td><code>bool is_sub_buffer() const</code></td>
<td>Returns true if this SYCL buffer is a sub-buffer, otherwise returns false.</td>
</tr>
</tbody>
</table>
| `template <typename ReinterpretT, int ReinterpretDim> buffer<ReinterpretT, ReinterpretDim, typename std::allocator_traits<AllocatorT>::template rebind_alloc<
    std::remove_const_t<ReinterpretT>>>` `reinterpret(range<ReinterpretDim> reinterpretRange) const` | Creates and returns a reinterpreted SYCL buffer with the type specified by `ReinterpretT`, dimensions specified by `ReinterpretDim` and range specified by `reinterpretRange`. The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL buffer and must throw exception with the `errc::invalid` error code if the total size in bytes represented by the type and range of the reinterpreted SYCL buffer (or sub-buffer) does not equal the total size in bytes represented by the type and range of this SYCL buffer (or sub-buffer). Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent buffer. |
### Member function

```cpp
template <typename ReinterpretT, int ReinterpretDim = Dimensions>
buffer<ReinterpretT, ReinterpretDim,
    typename std::allocator_traits<AllocatorT>::template rebind_alloc<
        std::remove_const_t<ReinterpretT>>>
reinterpret() const
```

### Description

Creates and returns a reinterpreted SYCL buffer with the type specified by `ReinterpretT` and dimensions specified by `ReinterpretDim`. Only valid when (`ReinterpretDim == 1`) or when `((ReinterpretDim == Dimensions) && (sizeof(ReinterpretT) == sizeof(T)))`. The buffer object being reinterpreted can be a SYCL sub-buffer that was created from a SYCL buffer. The implementation must throw an exception with the `errc::invalid` error code if the total size in bytes represented by this SYCL buffer (or sub-buffer) is not evenly divisible by `sizeof(ReinterpretT)`. Reinterpreting a sub-buffer provides a reinterpreted view of the sub-buffer only, and does not change the offset or size of the sub-buffer view (in bytes) relative to the parent buffer.

### 4.7.2.2. Buffer properties

The properties that can be provided when constructing the SYCL buffer class are describe in Table 41.

**Table 41. Properties supported by the SYCL buffer class**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::buffer::use_host_ptr</td>
<td>The use_host_ptr property adds the requirement that the SYCL runtime must not allocate any memory for the SYCL buffer and instead uses the provided host pointer directly. This prevents the SYCL runtime from allocating additional temporary storage on the host. This property has a special guarantee for buffers that are constructed from a hostData pointer. If a host_accessor is constructed from such a buffer, then the address of the reference type returned from the accessor’s member functions such as <code>operator[]</code> will be the same as the corresponding hostData address.</td>
</tr>
</tbody>
</table>
The use_mutex property is valid for the SYCL buffer, unsampled_image and sampled_image classes. The property adds the requirement that the memory which is owned by the SYCL buffer can be shared with the application via a std::mutex provided to the property. The mutex m is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with hostData, when the mutex is unlocked by the runtime.

The context_bound property adds the requirement that the SYCL buffer can only be associated with a single SYCL context that is provided to the property.

The constructors and special member functions of the buffer property classes are listed in Table 42 and Table 43 respectively.

Table 42. Constructors of the buffer property classes

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::buffer::use_host_ptr::use_host_ptr()</td>
<td>Constructs a SYCL use_host_ptr property instance.</td>
</tr>
<tr>
<td>property::buffer::use_mutex::use_mutex(std::mutex&amp; mutexRef)</td>
<td>Constructs a SYCL use_mutex property instance with a reference to mutexRef parameter provided.</td>
</tr>
<tr>
<td>property::buffer::context_bound::context_bound(context boundContext)</td>
<td>Constructs a SYCL context_bound property instance with a copy of a SYCL context.</td>
</tr>
</tbody>
</table>

Table 43. Member functions of the buffer property classes

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::mutex* property::buffer::use_mutex::get_mutex_ptr() const</td>
<td>Returns the std::mutex which was specified when constructing this SYCL use_mutex property.</td>
</tr>
<tr>
<td>context property::buffer::context_bound::get_context() const</td>
<td>Returns the context which was specified when constructing this SYCL context_bound property.</td>
</tr>
</tbody>
</table>

4.7.2.3. Buffer synchronization rules

Buffers are reference-counted. When a buffer value is constructed from another buffer, the two values reference the same buffer and a reference count is incremented. When a buffer value is destroyed, the reference count is decremented. Only when there are no more buffer values that reference a specific
buffer is the actual buffer destroyed and the buffer destruction behavior defined below is followed.

If any error occurs on buffer destruction, it is reported via the associated queue’s asynchronous error handling mechanism.

The basic rule for the blocking behavior of a buffer destructor is that it blocks if there is some data to write back because a write accessor on it has been created, or if the buffer was constructed with attached host memory and is still in use.

More precisely:

1. A buffer can be constructed from a range (and without a hostData pointer). The memory management for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer does not need to block, even if work on the buffer has not completed. Instead, the SYCL system frees any storage required for the buffer asynchronously when it is no longer in use in queues. The initial contents of the buffer are unspecified.

2. A buffer can be constructed from a hostData pointer. The buffer will use this host memory for its full lifetime, but the contents of this host memory are unspecified for the lifetime of the buffer. If the host memory is modified on the host or if it is used to construct another buffer or image during the lifetime of this buffer, then the results are undefined. The initial contents of the buffer will be the contents of the host memory at the time of construction.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed, then copy the contents of the buffer back to the host memory (if required) and then return.

   a. If the type of the host data is const, then the buffer is read-only; only read accessors are allowed on the buffer and no-copy-back to host memory is performed (although the host memory must still be kept available for use by SYCL). When using the default buffer allocator, the const-ness of the type will be removed in order to allow host allocation of memory, which will allow temporary host copies of the data by the SYCL runtime, for example for speeding up host accesses.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed and then return, as there is no copy of data back to host.

   b. If the type of the host data is not const but the pointer to host data is const, then the read-only restriction applies only on host and not on device accesses.

   When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed.

3. A buffer can be constructed using a shared_ptr to host data. This pointer is shared between the SYCL application and the runtime. In order to allow synchronization between the application and the runtime a mutex is used which will be locked by the runtime whenever the data is in use, and unlocked when it is no longer needed.

   The shared_ptr reference counting is used in order to prevent destroying the buffer host data prematurely. If the shared_ptr is deleted from the user application before buffer destruction, the buffer can continue securely because the pointer hasn't been destroyed yet. It will not copy data back to the host before destruction, however, as the application side has already deleted its copy.

   Note that since there is an implicit conversion of a std::unique_ptr to a std::shared_ptr, a std::unique_ptr can also be used to pass the ownership to the SYCL runtime.

4. A buffer can be constructed from a pair of iterator values. In this case, the buffer construction will copy the data from the data range defined by the iterator pair. The destructor will not copy back any data and does not need to block.

5. A buffer can be constructed from a container on which std::data(container) and std::size(cont-
The buffer may use the memory within the container for its full lifetime, and the contents of this memory are unspecified for the lifetime of the buffer. If the container memory is modified by the host during the lifetime of this buffer, then the results are undefined.

When the buffer is destroyed, the destructor will block until all work in queues on the buffer have completed. If the return type of `std::data(container)` is not `const` then the destructor will also copy the contents of the buffer to the container (if required).

If `set_final_data()` is used to change where to write the data back to, then the destructor of the buffer will block if a write accessor on it has been created.

A sub-buffer object can be created which is a sub-range reference to a base buffer. This sub-buffer can be used to create accessors to the base buffer, which have access to the range specified at time of construction of the sub-buffer. Sub-buffers cannot be created from sub-buffers, but only from a base buffer which is not already a sub-buffer.

Sub-buffers must be constructed from a contiguous region of memory in a buffer. This requirement is potentially non-intuitive when working with buffers that have dimensionality larger than one, but maps to one-dimensional SYCL backend native allocations without performance cost due to index mapping computation. For example:

```cpp
1 buffer<int, 2> parent_buffer { range<2> { 8, 8 } }; // Create 2-d buffer with 8x8 ints
2
3 // OK: Contiguous region from middle of buffer
4 buffer<int, 2> sub_buf1 { parent_buffer, /*offset*/ range<2> { 2, 0 },
5     /*size*/ range<2> { 2, 8 } };
6
7 // invalid exception: Non-contiguous regions of 2-d buffer
8 buffer<int, 2> sub_buf2 { parent_buffer, /*offset*/ range<2> { 2, 0 },
9     /*size*/ range<2> { 2, 2 } };
10 buffer<int, 2> sub_buf3 { parent_buffer, /*offset*/ range<2> { 2, 2 },
11     /*size*/ range<2> { 2, 6 } };
12
13 // invalid exception: Out-of-bounds size
14 buffer<int, 2> sub_buf4 { parent_buffer, /*offset*/ range<2> { 2, 2 },
15     /*size*/ range<2> { 2, 8 } };
```

### 4.7.3. Images

The classes `unsampled_image` (Table 44) and `sampled_image` (Table 46) define shared image data of one, two or three dimensions, that can be used by kernels in queues and have to be accessed using the image accessor classes.

The constructors and member functions of the SYCL `unsampled_image` and `sampled_image` class templates are listed in Table 44, Table 45, Table 46 and Table 47, respectively. The additional common special member functions and common member functions are listed in Table 7 and Table 8, respectively.

Where relevant, it is the responsibility of the user to ensure that the format of the data matches the format described by `image_format`.

The allocator template parameter of the SYCL `unsampled_image` and `sampled_image` classes can be any allocator type including a custom allocator, however it must allocate in units of `std::byte`.
For any image that is constructed with the range \((r_1, r_2, r_3)\) with an element type size in bytes of \(s\), the image row pitch and image slice pitch should be calculated as follows:

\[
\begin{align*}
    r_1 \cdot s \\
    r_1 \cdot r_2 \cdot s
\end{align*}
\]

The SYCL `unsampled_image` and `sampled_image` class templates provide the common reference semantics (see Section 4.5.2).

### 4.7.3.1. Unsampling image interface

Each constructor of the `unsampled_image` takes an `image_format` to describe the data layout of the image data.

Each constructor additionally takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `unsampled_image`.

The SYCL `unsampled_image` class template takes a template parameter `AllocatorT` for specifying an allocator which is used by the SYCL runtime when allocating temporary memory on the host. If no template argument is provided, the default allocator for the SYCL `unsampled_image` class `image_allocator` is used (see Section 4.7.1.1).

```cpp
namespace sycl {

    enum class image_format : /* unspecified */ {
        r8g8b8a8_unorm,
        r16g16b16a16_unorm,
        r8g8b8a8_sint,
        r16g16b16a16_sint,
        r32b32g32a32_sint,
        r8g8b8a8_uint,
        r16g16b16a16_uint,
        r32b32g32a32_uint,
        r16b16g16a16_sfloat,
        r32g32b32a32_sfloat,
        b8g8r8a8_unorm
    };

    template <int Dimensions = 1, typename AllocatorT = sycl::image_allocator>
    class unsampled_image {

        public:

        unsampled_image(image_format format, const range<Dimensions>& rangeRef,
                        const property_list& propList = {});

        unsampled_image(image_format format, const range<Dimensions>& rangeRef,
                        AllocatorT allocator, const property_list& propList = {});

        /* Available only when: Dimensions > 1 */
        unsampled_image(image_format format, const range<Dimensions>& rangeRef,
                        const range<Dimensions - 1>& pitch,
                        const property_list& propList = {});

        /* Available only when: Dimensions > 1 */
        unsampled_image(image_format format, const range<Dimensions>& rangeRef,
                        const range<Dimensions - 1>& pitch, AllocatorT allocator,
                        const property_list& propList = {});

    }
}
```

130 | Chapter 4. SYCL programming interface
unsampled_image(void* hostPointer, image_format format,
   const range<Dimensions>& rangeRef,
   const property_list& propList = {});

unsampled_image(void* hostPointer, image_format format,
   const range<Dimensions>& rangeRef, AllocatorT allocator,
   const property_list& propList = {});

/* Available only when: Dimensions > 1 */
unsampled_image(void* hostPointer, image_format format,
   const range<Dimensions>& rangeRef, AllocatorT allocator,
   const property_list& propList = {});

unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
   const range<Dimensions>& rangeRef,
   const property_list& propList = {});

unsampled_image(std::shared_ptr<void>& hostPointer, image_format format,
   const range<Dimensions>& rangeRef, AllocatorT allocator,
   const property_list& propList = {});

/* -- common interface members -- */

/* -- property interface members -- */
range<Dimensions> get_range() const;

/* Available only when: Dimensions > 1 */
range<Dimensions - 1> get_pitch() const;
size_t byte_size() const noexcept;
size_t size() const noexcept;
AllocatorT get_allocator() const;

template <typename... Ts> auto get_access(Ts... args);
```cpp
template <typename... Ts> auto get_host_access(Ts... args);

template <typename Destination = std::nullptr_t>
void set_final_data(Destination finalData = std::nullptr);

void set_write_back(bool flag = true);
}

// namespace sycl
```

### Table 44. Constructors of the `unsampled_image` class template

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `unsampled_image(image_format format,
const range<Dimensions>& rangeRef,
const property_list& propList = {})` | Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
### Constructor

```cpp
unsampled_image(image_format format,
    const range<Dimensions>& rangeRef,
    AllocatorT allocator,
    const property_list& propList = {})```

**Description**

Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use the allocator parameter provided when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`.

### Constructor

```cpp
unsampled_image(image_format format,
    const range<Dimensions>& rangeRef,
    const range<Dimensions - 1>& pitch,
    const property_list& propList = {})```

**Description**

Available only when: `Dimensions > 1`.

Construct a SYCL `unsampled_image` instance with uninitialized memory. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. 
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsampled_image(image_format format,</code></td>
<td>Available only when: $\text{Dimensions} &gt; 1$.</td>
</tr>
<tr>
<td><code>const range&lt;\text{Dimensions}&gt;\&amp; rangeRef,</code></td>
<td>Construct a SYCL <code>unsampled_image</code> instance with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, there will be no write back on destruction. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>const range&lt;\text{Dimensions} - 1&gt;\&amp; pitch,</code></td>
<td></td>
</tr>
<tr>
<td><code>AllocatorT allocator,</code></td>
<td></td>
</tr>
<tr>
<td><code>const property_list\&amp; propList = {});</code></td>
<td></td>
</tr>
<tr>
<td><code>unsampled_image(void* hostPointer, image_format format,</code></td>
<td>Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use a default constructed <code>AllocatorT</code> when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>const range&lt;\text{Dimensions}&gt;\&amp; rangeRef,</code></td>
<td></td>
</tr>
<tr>
<td><code>const property_list\&amp; propList = {});</code></td>
<td></td>
</tr>
<tr>
<td>Constructor</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>unsampled_image(void* hostPointer, image_format format, const range&lt;Dimensions&gt;&amp; rangeRef, AllocatorT allocator, const property_list&amp; propList = {})</code></td>
<td>Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
### Constructor

```
unsampled_image(void* hostPointer, image_format format,
const range<Dimensions>& rangeRef,
const range<Dimensions - 1>& pitch,
const property_list& propList = {})
```

### Description

Available only when: `Dimensions > 1`

Construct a SYCL `unsampled_image` instance with the `hostPointer` parameter provided. The `unsampled_image` assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. 
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsampled_image(void* hostPointer, image_format format, const range&lt;Dimensions&gt;&amp; rangeRef, const range&lt;Dimensions - 1&gt;&amp; pitch, AllocatorT allocator, const property_list&amp; propList = {})</code></td>
<td>Available only when: <code>Dimensions &gt; 1</code>. Construct a SYCL <code>unsampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the <code>pitch</code> parameter provided. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
When `hostPointer` is not empty, construct a SYCL `unsampled_image` with the contents of its stored pointer. The `unsampled_image` assumes exclusive access to this memory for the duration of its lifetime. The `unsampled_image` also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostPointer` is empty, construct a SYCL `unsampled_image` with uninitialized memory.

The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`.

<table>
<thead>
<tr>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsampled_image(std::shared_ptr&lt;void&gt;&amp; hostPointer, image_format format, const range&lt;Dimensions&gt;&amp; rangeRef, const property_list&amp; propList = {})</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| When `hostPointer` is not empty, construct a SYCL `unsampled_image` with the contents of its stored pointer. The `unsampled_image` assumes exclusive access to this memory for the duration of its lifetime. The `unsampled_image` also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostPointer` is empty, construct a SYCL `unsampled_image` with uninitialized memory.

The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the default size determined by the SYCL runtime. Unless the member function `set_final_data()` is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsampled_image(std::shared_ptr&lt;void&gt; hostPointer, image_format format, const range&lt;Dimensions&gt;&amp; rangeRef, AllocatorT allocator, const property_list&amp; propList = {})</code></td>
<td>When <code>hostPointer</code> is not empty, construct a SYCL <code>unsampled_image</code> with the contents of its stored pointer. The <code>unsampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The <code>unsampled_image</code> also creates its own internal copy of the <code>shared_ptr</code> that shares ownership of the <code>hostData</code> memory, which means the application can safely release ownership of this <code>shared_ptr</code> when the constructor returns. When <code>hostPointer</code> is empty, construct a SYCL <code>unsampled_image</code> with uninitialized memory. The constructed SYCL <code>unsampled_image</code> will use the <code>allocator</code> parameter provided when allocating memory on the host. The element size of the constructed SYCL <code>unsampled_image</code> will be derived from the <code>format</code> parameter. The range of the constructed SYCL <code>unsampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>unsampled_image</code> will be the default size determined by the SYCL runtime. Unless the member function <code>set_final_data()</code> is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to <code>hostPointer</code>. Zero or more properties can be provided to the constructed SYCL <code>unsampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `unsampled_image(std::shared_ptr<void>& hostPointer, image_format format, const range<Dimensions>& rangeRef, const range<Dimensions - 1>& pitch, const property_list& propList = {})` | When `hostPointer` is not empty, construct a SYCL `unsampled_image` with the contents of its stored pointer. The `unsampled_image` assumes exclusive access to this memory for the duration of its lifetime. The `unsampled_image` also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.  

When `hostPointer` is empty, construct a SYCL `unsampled_image` with uninitialized memory.  

The constructed SYCL `unsampled_image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `unsampled_image` will be derived from the `format` parameter. The range of the constructed SYCL `unsampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `unsampled_image` will be the `pitch` parameter provided. Unless the member function `set_final_data()` is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to `hostPointer`. Zero or more properties can be provided to the constructed SYCL `unsampled_image` via an instance of `property_list`. |
Constructor

```cpp
unsampled_image(std::shared_ptr<void>& hostPointer,
                 image_format format,
                 const range<Dimensions>& rangeRef,
                 const range<Dimensions - 1>& pitch,
                 AllocatorT allocator,
                 const property_list& propList = {})
```

When `hostPointer` is not empty, construct a SYCL unsampled_image with the contents of its stored pointer. The unsampled_image assumes exclusive access to this memory for the duration of its lifetime. The unsampled_image also creates its own internal copy of the shared_ptr that shares ownership of the hostData memory, which means the application can safely release ownership of this shared_ptr when the constructor returns.

When `hostPointer` is empty, construct a SYCL unsampled_image with uninitialized memory.

The constructed SYCL unsampled_image will use the allocator parameter provided when allocating memory on the host. The element size of the constructed SYCL unsampled_image will be derived from the format parameter. The range of the constructed SYCL unsampled_image is specified by the rangeRef parameter provided. The pitch of the constructed SYCL unsampled_image will be the pitch parameter provided. Unless the member function set_final_data() is called with a valid non-null pointer, any memory allocated by the SYCL runtime is written back to hostPointer. Zero or more properties can be provided to the constructed SYCL unsampled_image via an instance of property_list.

### Table 45. Member functions of the unsampled_image class template

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>range&lt;Dimensions&gt; get_range()</code> const</td>
<td>Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions - 1&gt; get_pitch()</code> const</td>
<td>Available only when: Dimensions &gt; 1. Return a range object representing the pitch of the image in bytes.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>size_t size() const noexcept</td>
<td>Returns the total number of elements in the image. Equal to ( \text{get_range()[0]} \times \ldots \times \text{get_range()[Dimensions-1]} ).</td>
</tr>
<tr>
<td>size_t byte_size() const noexcept</td>
<td>Returns the size of the image storage in bytes. The number of bytes may be greater than ( \text{size() \times element size} ) due to padding of elements, rows and slices of the image for efficient access.</td>
</tr>
<tr>
<td>AllocatorT get_allocator() const</td>
<td>Returns the allocator provided to the image.</td>
</tr>
<tr>
<td>template&lt;typename... Ts&gt; auto get_access(Ts... args)</td>
<td>Returns a valid <code>unsampled_image_accessor</code> as if constructed via passing the image and all provided arguments to the <code>unsampled_image_accessor</code> constructor. Possible implementation: <code>return unsampled_image_accessor{*this, args...};</code></td>
</tr>
<tr>
<td>template&lt;typename... Ts&gt; auto get_host_access(Ts... args)</td>
<td>Returns a valid <code>host_unsampled_image_accessor</code> as if constructed via passing the image and all provided arguments to the <code>host_unsampled_image_accessor</code> constructor. Possible implementation: <code>return host_unsampled_image_accessor{*this, args...};</code></td>
</tr>
<tr>
<td>template&lt;typename Destination = std::nullptr_t&gt; void set_final_data(Destination finalData = nullptr)</td>
<td>The <code>finalData</code> point to where the output of all the image processing is going to be copied to at destruction time, if the image was involved with a write accessor. Destination can be either an output iterator, a <code>std::weak_ptr&lt;T&gt;</code>. Note that a raw pointer is a special case of output iterator and thus defines the host memory to which the result is to be copied. In the case of a weak pointer, the output is not copied if the weak pointer has expired. If <code>Destination</code> is <code>std::nullptr_t</code>, then the copy back will not happen.</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void set_write_back(bool flag = true)</td>
<td>This member function allows dynamically forcing or canceling the write-back of the data of an image on destruction according to the value of <code>flag</code>. Forcing the write-back is similar to what happens during a normal write-back as described in Section 4.7.3.4 and Section 4.7.4. If there is nowhere to write-back, using this function does not have any effect.</td>
</tr>
</tbody>
</table>

### 4.7.3.2. Sampled image interface

Each constructor of the `sampled_image` class requires a pointer to the host data the image will sample, an `image_format` to describe the data layout and an `image_sampler` (Section 4.7.8) to describe how to sample the image data.

Each constructor additionally takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `sampled_image`.

```cpp
namespace sycl {
    enum class image_format : /* unspecified */ {
        r8g8b8a8_unorm,
        r16g16b16a16_unorm,
        r8g8b8a8_sint,
        r16g16b16a16_sint,
        r32b32g32a32_sint,
        r8g8b8a8_uint,
        r16g16b16a16_uint,
        r32b32g32a32_uint,
        r16b16g16a16_sfloat,
        r32g32b32a32_sfloat,
        b8g8r8a8_unorm
    }

    template <int Dimensions = 1, typename AllocatorT = sycl::image_allocator>
    class sampled_image {
        public:
            sampled_image(const void* hostPointer, image_format format,
                image_sampler sampler, const range<Dimensions>& rangeRef,
                const property_list& proplist = {});

            /* Available only when: Dimensions > 1 */
            sampled_image(const void* hostPointer, image_format format,
                image_sampler sampler, const range<Dimensions>& rangeRef,
                const range<Dimensions - 1>& pitch,
                const property_list& proplist = {});

            sampled_image(std::shared_ptr<const void> hostPointer, image_format format,
                image_sampler sampler, const range<Dimensions>& rangeRef,
            );
    };
}
```
const property_list& propList = {));

/* Available only when: Dimensions > 1 */
sampled_image(std::shared_ptr<const void>& hostPointer, image_format format,
               image_sampler sampler, const range<Dimensions>& rangeRef,
               const range<Dimensions - 1>& pitch,
               const property_list& propList = {});

/* -- common interface members -- */
/* -- property interface members -- */
range<Dimensions> get_range() const;

/* Available only when: Dimensions > 1 */
range<Dimensions - 1> get_pitch() const;

size_t byte_size() const;

size_t size() const;

template<typename... Ts> auto get_access(Ts... args);

template<typename... Ts> auto get_host_access(Ts... args);

};

namespace sycl

Table 46. Constructors of the `sampled_image` class template
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sampled_image(const void* hostPointer, image_format format, image_sampler sampler, const range&lt;Dimensions&gt;&amp; rangeRef, const property_list&amp; propList = {})</td>
<td>Construct a SYCL <code>sampled_image</code> instance with the <code>hostPointer</code> parameter provided. The <code>sampled_image</code> assumes exclusive access to this memory for the duration of its lifetime. The host address is <code>const</code>, so the host accesses must be read-only. Since, the <code>hostPointer</code> is <code>const</code>, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL <code>sampled_image</code> will be derived from the <code>format</code> parameter. Accessors that read the constructed SYCL <code>sampled_image</code> will use the <code>sampler</code> parameter to sample the image. The range of the constructed SYCL <code>sampled_image</code> is specified by the <code>rangeRef</code> parameter provided. The pitch of the constructed SYCL <code>sampled_image</code> will be the default size determined by the SYCL runtime. Zero or more properties can be provided to the constructed SYCL <code>sampled_image</code> via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td>Constructor</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `sampled_image(const void* hostPointer, image_format format, image_sampler sampler, const range<Dimensions>& rangeRef, const range<Dimensions - 1>& pitch, const property_list& propList = {})` | Available only when: `Dimensions > 1`.

Construct a SYCL `sampled_image` instance with the `hostPointer` parameter provided. The `sampled_image` assumes exclusive access to this memory for the duration of its lifetime. The host address is `const`, so the host accesses must be read-only. Since, the `hostPointer` is `const`, this image is only initialized with this memory and there is no write after destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. Accessors that read the constructed SYCL `sampled_image` will use the `sampler` parameter to sample the image. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the `pitch` parameter provided. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`. |
### Constructor

```cpp
class sampled_image {
public:
    sampled_image(
        std::shared_ptr<const void> hostPointer,
        image_format format,
        image_sampler sampler,
        const range<Dimensions> &rangeRef,
        const property_list &propList = {});
};
```

**Description**

When `hostPointer` is not empty, construct a SYCL `sampled_image` with the contents of its stored pointer. The `sampled_image` assumes exclusive access to this memory for the duration of its lifetime. The `sampled_image` also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostPointer` is empty, construct a SYCL `sampled_image` with uninitialized memory.

The host address is `const`, so the host accesses must be read-only. Since, the `hostPointer` is `const`, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. Accessors that read the constructed SYCL `sampled_image` will use the `sampler` parameter to sample the image. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the default size determined by the SYCL runtime. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`.
When `hostPointer` is not empty, construct a SYCL `sampled_image` with the contents of its stored pointer. The `sampled_image` assumes exclusive access to this memory for the duration of its lifetime. The `sampled_image` also creates its own internal copy of the `shared_ptr` that shares ownership of the `hostData` memory, which means the application can safely release ownership of this `shared_ptr` when the constructor returns.

When `hostPointer` is empty, construct a SYCL `sampled_image` with uninitialized memory.

The host address is `const`, so the host accesses can be read-only. Since, the `hostPointer` is `const`, this image is only initialized with this memory and there is no write after its destruction. The element size of the constructed SYCL `sampled_image` will be derived from the `format` parameter. Accessors that read the constructed SYCL `sampled_image` will use the `sampler` parameter to sample the image. The range of the constructed SYCL `sampled_image` is specified by the `rangeRef` parameter provided. The pitch of the constructed SYCL `sampled_image` will be the `pitch` parameter provided. Zero or more properties can be provided to the constructed SYCL `sampled_image` via an instance of `property_list`.

### Table 47. Member functions of the `sampled_image` class template

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>range&lt;Dimensions&gt; get_range() const</code></td>
<td>Return a range object representing the size of the image in terms of the number of elements in each dimension as passed to the constructor. Available only when: <code>Dimensions &gt; 1</code>. Return a range object representing the pitch of the image in bytes.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions - 1&gt; get_pitch() const</code></td>
<td></td>
</tr>
</tbody>
</table>
### 4.7.3.3. Image properties

The properties that can be provided when constructing the SYCL `unsampled_image` and `sampled_image` classes are described in Table 48.

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t size() const noexcept</td>
<td>Returns the total number of elements in the image. Equal to ( \text{get_range()[0]} \times \ldots \times \text{get_range()[Dimensions-1]} ).</td>
</tr>
<tr>
<td>size_t byte_size() const noexcept</td>
<td>Returns the size of the image storage in bytes. The number of bytes may be greater than <code>size()*element size</code> due to padding of elements, rows and slices of the image for efficient access.</td>
</tr>
<tr>
<td>template &lt;typename... Ts&gt; auto get_access(Ts... args)</td>
<td>Returns a valid <code>sampled_image_accessor</code> as if constructed via passing the image and all provided arguments to the <code>sampled_image_accessor</code> constructor. Possible implementation: <code>return sampled_image_accessor(*this, args...);</code></td>
</tr>
<tr>
<td>template &lt;typename... Ts&gt; auto get_host_access(Ts... args)</td>
<td>Returns a valid <code>host_sampled_image_accessor</code> as if constructed via passing the image and all provided arguments to the <code>host_sampled_image_accessor</code> constructor. Possible implementation: <code>return host_sampled_image_accessor(*this, args...);</code></td>
</tr>
</tbody>
</table>

```cpp
namespace sycl {
    namespace property {
        namespace image {
            class use_host_ptr {
                public:
                use_host_ptr() = default;
            };
            class use_mutex {
                public:
                use_mutex(std::mutex& mutexRef);
                std::mutex* get_mutex_ptr() const;
            };
            class context_bound {
                public:
                context_bound(context boundContext);
            }
        }
    }
}
```
Table 48. Properties supported by the SYCL image classes

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::image::use_host_ptr</td>
<td>The use_host_ptr property adds the requirement that the SYCL runtime must not allocate any memory for the image and instead uses the provided host pointer directly. This prevents the SYCL runtime from allocating additional temporary storage on the host.</td>
</tr>
<tr>
<td>property::image::use_mutex</td>
<td>The property adds the requirement that the memory which is owned by the SYCL image can be shared with the application via a std::mutex provided to the property. The std::mutex is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with host-Data, when the std::mutex is unlocked by the runtime.</td>
</tr>
<tr>
<td>property::image::context_bound</td>
<td>The context_bound property adds the requirement that the SYCL image can only be associated with a single SYCL context that is provided to the property.</td>
</tr>
</tbody>
</table>

The constructors and member functions of the image property classes are listed in Table 49 and Table 50

Table 49. Constructors of the image property classes

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::image::use_host_ptr::use_host_ptr()</td>
<td>Constructs a SYCL use_host_ptr property instance.</td>
</tr>
<tr>
<td>property::image::use_mutex::use_mutex(std::mutex&amp; mutexRef)</td>
<td>Constructs a SYCL use_mutex property instance with a reference to mutexRef parameter provided.</td>
</tr>
<tr>
<td>property::image::context_bound::context_bound(context boundContext)</td>
<td>Constructs a SYCL context_bound property instance with a copy of a SYCL context.</td>
</tr>
</tbody>
</table>

Table 50. Member functions of the image property classes
4.7.3.4. Image synchronization rules

The rules are similar to those described in Section 4.7.2.3.

For the lifetime of the image object, the associated host memory must be left available to the SYCL runtime and the contents of the associated host memory is unspecified until the image object is destroyed. If an image object value is copied, then only a reference to the underlying image object is copied. The underlying image object is reference-counted. Only after all image value references to the underlying image object have been destroyed is the actual image object itself destroyed.

If an image object is constructed with associated host memory, then its destructor blocks until all operations in all SYCL queues on that image object have completed. Any modifications to the image data will be copied back, if necessary, to the associated host memory. Any errors occurring during destruction are reported to any associated context's asynchronous error handler. If an image object is constructed with a storage object, then the storage object defines what synchronization or copying behavior occurs on image object destruction.

4.7.4. Sharing host memory with the SYCL data management classes

In order to allow the SYCL runtime to do memory management and allow for data dependencies, there are two classes defined, buffer and image. The default behavior for them is that a “raw” pointer is given during the construction of the data management class, with full ownership to use it until the destruction of the SYCL object.

In this section we go in greater detail on sharing or explicitly not sharing host memory with the SYCL data classes, and we will use the buffer class as an example. The same rules will apply to images as well.

4.7.4.1. Default behavior

When using a SYCL buffer, the ownership of the pointer passed to the constructor of the class is, by default, passed to SYCL runtime, and that pointer cannot be used on the host side until the buffer or image is destroyed. A SYCL application can access the contents of the memory managed by a SYCL buffer by using a host_accessor as defined in Section 4.7.6. However, there is no guarantee that the host accessor synchronizes with the original host address used in its constructor.

The pointer passed in is the one used to copy data back to the host, if needed, before buffer destruction. The memory pointed by host pointer will not be de-allocated by the runtime, and the data is copied back from the device if there is a need for it.

4.7.4.2. SYCL ownership of the host memory

In the case where there is host memory to be used for initialization of data but there is no intention of using that host memory after the buffer is destroyed, then the buffer can take full ownership of that host memory.

When a buffer owns the host pointer there is no copy back, by default. In this situation, the SYCL application may pass a unique pointer to the host data, which will be then used by the runtime internally to initialize the data in the device.
For example, the following could be used:

```c++
1 {
2    auto ptr = std::make_unique<int>(-1234);
3    buffer<int, 1> b { std::move(ptr), range { 1 } };
4    // ptr is not valid anymore.
5    // There is nowhere to copy data back
6 }
```

However, optionally the `buffer::set_final_data()` can be set to a `std::weak_ptr` to enable copying data back, to another host memory address that is going to be valid after buffer construction.

```c++
1 {
2    auto ptr = std::make_unique<int>(-42);
3    buffer<int, 1> b { std::move(ptr), range { 1 } };
4    // ptr is not valid anymore.
5    // There is nowhere to copy data back.
6    // To get copy back, a location can be specified:
7    b.set_final_data(std::weak_ptr<int> { .... })
8 }
```

### 4.7.4.3. Shared SYCL ownership of the host memory

When an instance of `std::shared_ptr` is passed to the buffer constructor, then the buffer object and the developer's application share the memory region. If the shared pointer is still used on the application's side then the data will be copied back from the buffer or image and will be available to the application after the buffer or image is destroyed.

If the `shared_ptr` is not empty, the contents of the referenced memory are used to initialize the buffer. If the `shared_ptr` is empty, then the buffer is created with uninitialized memory.

When the buffer is destroyed and the data have potentially been updated, if the number of copies of the shared pointer outside the runtime is 0, there is no user-side shared pointer to read the data. Therefore the data is not copied out, and the buffer destructor does not need to wait for the data processes to be finished, as the outcome is not needed on the application's side.

This behavior can be overridden using the `set_final_data()` member function of the buffer class, which will by any means force the buffer destructor to wait until the data is copied to wherever the `set_final_data()` member function has put the data (or not wait nor copy if set final data is `nullptr`).

```c++
1 {
2    std::shared_ptr<int> ptr { data };
3    {
4        buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
5        // update the data
6        [...]  
7    } // Data is copied back because there is an user side shared_ptr
8 }
```

```c++
1 {
2    std::shared_ptr<int> ptr { data };
3    {
4        buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
```
4.7.5. Synchronization primitives

When the user wants to use the buffer simultaneously in the SYCL runtime and their own code (e.g. a multi-threaded mechanism) and wants to use manual synchronization without using a host_accessor, a std::mutex can be passed to the buffer constructor via the right property.

The runtime promises to lock the mutex whenever the data is in use and unlock it when it no longer needs it.

```cpp
1 {
2   std::mutex m;
3   auto shD = std::make_shared<int>(42)
4   sycl::buffer b { shD, { sycl::property::buffer::use_mutex { m } } };
5   {
6     std::lock_guard lck { m };
7     // User accesses the data
8     do_something(shD);
9     /* m is unlocked when lck goes out of scope, by normal end of this
10      block but also if an exception is thrown for example */
11   }
12 }
```

When the runtime releases the mutex the user is guaranteed that the data was copied back on the shared pointer --- unless the final data destination has been changed using the member function set_final_data().

4.7.6. Accessors

Accessors provide three different capabilities: they provide access to the data managed by a buffer or image, they provide access to local memory on a device, and they define the requirements to memory objects which determine the scheduling of kernels (see Section 3.8.1).

A memory object requirement is created when an accessor is constructed, unless the accessor is a placeholder in which case the requirement is created when the accessor is bound to a command by calling handler::require().

There are several different C++ classes that implement accessors:

- The accessor class provides access to data in a buffer from within a command.
- The host_accessor class provides access to data in a buffer from host code that is outside of a command. These accessors are typically used in application scope.
- The local_accessor class provides access to device local memory from within a SYCL kernel function.
- The unsampled_image_accessor and sampled_image_accessor classes provide access to data in an unsampled_image and sampled_image from within a command.
- The host_unsampled_image_accessor and host_sampled_image_accessor classes provide access to data in an unsampled_image and sampled_image from host code that is outside of a command. These accessors are typically used in application scope.
Accessor objects must always be constructed in host code, either in command group scope or in application scope. Whether the constructor blocks waiting for data to synchronize depends on the type of accessor. Those accessors which provide access to data within a command do not block. Instead, these accessors define a requirement which influences the scheduling of the command. Those accessors which provide access to data from host code do block until the data is available on the host.

For those accessors which provide access to data within a command, the member functions which access data should only be called from within the command. Programs which call these member functions from outside of the command are ill formed. The sections below describe exactly which member functions fall into this category.

### 4.7.6.1. Data type

All accessors have a `DataT` template parameter which specifies the type of each element that the accessor accesses. For `accessor` and `host_accessor`, this type must either match the type of each element in the underlying buffer, or it must be a `const` qualified version of that type.

For the image accessors (`unsampled_image_accessor`, `sampled_image_accessor`, `host_unsampled_image_accessor`, and `host_sampled_image_accessor`), `DataT` must be one of:

- `int4 (vec<int32_t,4>)`,
- `uint4 (vec<uint32_t,4>)`,
- `float4 (vec<float,4>)`, or
- `half4 (vec<half,4>)`

For `local_accessor` see Section 4.7.6.11 for the allowable `DataT` types.

### 4.7.6.2. Access modes

Most accessors have an `AccessMode` template parameter which specifies whether the accessor can read or write the underlying data. This information is used by the runtime when defining the requirements for the associated command, and it tells the runtime whether data needs to be transferred to or from a device before data can be accessed through the accessor.

The `access_mode` enumeration, shown in Table 51, describes the potential modes of an accessor. However, not all accessor classes support all modes, so see the description of each class for more details.

```cpp
namespace sycl {

enum class access_mode : /* unspecified */ {
    read,
    write,
    read_write,
    discard_write, // Deprecated in SYCL 2020
    discard_read_write, // Deprecated in SYCL 2020
    atomic // Deprecated in SYCL 2020
};

namespace access {
    // The legacy type "access::mode" is deprecated.
    using mode = sycl::access_mode;
}
}
```
Table 51. Enumeration of access modes available to accessors

<table>
<thead>
<tr>
<th>access_mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>access_mode::read</td>
<td>Read-only access.</td>
</tr>
<tr>
<td>access_mode::write</td>
<td>Write-only access.</td>
</tr>
<tr>
<td>access_mode::read_write</td>
<td>Read and write access.</td>
</tr>
</tbody>
</table>

4.7.6.3. Deduction tags

Some accessor constructors take a TagT parameter, which is used to deduce template arguments for the constructor's class. Each of the access modes in Table 51 has an associated tag, but there are additional tags which set other template parameters in addition to the access mode. The synopsis below shows the namespace scope variables that the implementation provides as possible values for the TagT parameter.

```cpp
namespace sycl {
    namespace property {
        struct no_init {};
    } // namespace property
}
```

The precise meaning of these tags depends on the specific accessor class that is being constructed, so they are described more fully below in the section that pertains to each of the accessor types.

4.7.6.4. Properties

All accessor constructors accept a property_list parameter, which affects the semantics of the accessor. Table 52 shows the set of all possible accessor properties and tells which properties are allowed when constructing each accessor class.

```cpp
namespace sycl {
    namespace property {
        struct no_init {}; // namespace property
    }
    namespace sycl { // namespace sycl
        inline constexpr property::no_init no_init;
    } // namespace sycl
```

Table 52. Properties supported by accessors
<table>
<thead>
<tr>
<th>Property</th>
<th>Allowed with</th>
<th>Description</th>
</tr>
</thead>
</table>
| property::no_init    | accessor, host_accessor, unsampled_image_accessor, host_unsampled_image_accessor | This property is useful when an application expects to write new values to all of the accessor's elements without reading their previous values. The implementation can use this information to avoid copying the accessor's data in some cases. Following is a more formal description. 

This property is only allowed for accessors with access_mode::write or access_mode::read_write access modes. Attempting to construct an access_mode::read accessor with this property causes an exception with the errc::invalid error code to be thrown.

The usage of this property is different depending on whether the accessor's underlying data type `DataT` is an implicit-lifetime type (as defined in the C++ core language). If it is an implicit-lifetime type, the accessor implicitly creates objects of that type with indeterminate values. The application is not required to write values to each element of the accessor, but unwritten elements of the accessor's buffer or image receive indeterminate values, even if those buffer or image elements previously had defined values. If this is a ranged accessor, this applies only to the elements within the accessor's range. The values of unwritten elements outside of this range are preserved.

If `DataT` is not an implicit-lifetime type, the accessor merely allocates uninitialized memory, and the application is responsible for constructing objects in that memory (e.g. by calling placement-new). The application must create an object in each element of the accessor unless the corresponding element of the underlying buffer did not previously contain an object. If this is a ranged accessor, this applies only to the elements within the accessor’s range. The content of objects in the buffer outside of this range is preserved.

As stated above, the property::no_init property requires the application to construct an object for each accessor element when the element’s type is not an implicit-lifetime type (except in the case when the corresponding buffer element did not previously contain an object). The reason for this requirement is to avoid the possibility of overwriting a valid object with indeterminate bytes, for example, when a command using the accessor completes. This means that the implementation can unconditionally copy memory from the device back to the host when the command completes, regardless of whether the `DataT` type is an implicit-lifetime type.
The constructors of the accessor property classes are listed in Table 53.

Table 53. Constructors of the accessor property classes

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property::no_init::no_init()</td>
<td>Constructs a no_init property instance.</td>
</tr>
</tbody>
</table>

### 4.7.6.5. Read only accessors

Accessors which have an `AccessMode` template parameter can be declared as read-only by specifying `access_mode::read` for the template parameter. A read-only accessor provides read-only access to the underlying data and provides a "read" requirement for the memory object when it is constructed.

The `DataT` template parameter for a read-only accessor can optionally be `const` qualified, and the semantics of the accessor are unchanged. For example, an accessor declared with `const DataT` and `access_mode::read` has the same semantics as an accessor declared with `DataT` and `access_mode::read`.

As detailed in the sections below, some accessor types have a default value for `AccessMode`, which depends on whether the `DataT` parameter is `const` qualified. This provides a convenient way to declare a read-only accessor without explicitly specifying the access mode.

A `const` qualified `DataT` is only allowed for a read-only accessor. Programs which specify a `const` qualified `DataT` and any access mode other than `access_mode::read` are ill formed, and the implementation must issue a diagnostic in this case.

Each accessor class also provides implicit conversions between the two forms of read-only accessors. This makes it possible, for example, to assign an accessor whose type has `const DataT` and `access_mode::read` to an accessor whose type has `DataT` and `access_mode::read`, so long as the other template parameters are the same. There is also an implicit conversion from a read-write accessor to either of the forms of a read-only accessor. These implicit conversions are described in detail for each accessor class in the sections that follow.

### 4.7.6.6. Accessing elements of an accessor

Accessors of type `accessor`, `host_accessor`, and `local_accessor` can have zero, one, two, or three Dimensions. A zero dimension accessor provides access to a single scalar element via an implicit conversion operator to the underlying type of that element and via an overloaded copy/move assignment operators from the underlying type of the element.

One, two, or three dimensional specializations of these accessors provide access to the elements they contain in two ways. The first way is through a subscript operator that takes an instance of an `id` class which has the same dimensionality as the accessor. The second way is by passing a single `size_t` value to multiple consecutive subscript operators as specified in Section 3.11.2.

In all these cases, the reference to the contained element is of type `const DataT&` for read-only accessors and of type `DataT&` for other accessors.

Accessors of all types have a range that defines the set of indices that may be used to access elements. For buffer accessors, this is the range of the underlying buffer, unless it is a `ranged accessor` in which case the range comes from the accessor's constructor. For image accessors, this is the range of the underlying image. Local accessors specify the range when the accessor is constructed. Any attempt to access an element via an index that is outside of this range produces undefined behavior.

### 4.7.6.7. Container interface

Accessors of type `accessor`, `host_accessor`, and `local_accessor` meet the C++ requirement of `Reversible-Container`. The exception to this is that only `local_accessor` owns the underlying data, meaning that its
destructor destroys elements and frees the memory. The accessor and host_accessor types don't destroy any elements or free the memory on destruction. The iterator for the container interface meets the C++ requirement of LegacyRandomAccessIterator and the underlying pointers/references correspond to the address space specified by the accessor type. For multidimensional accessors the iterator linearizes the data according to Section 3.11.1.

4.7.6.8. Ranged accessors

Accessors of type accessor and host_accessor can be constructed from a sub-range of a buffer by providing a range and offset to the constructor. This limits the elements that can be accessed to the specified sub-range, which allows the implementation to perform certain optimizations such as reducing the amount of memory that needs to be copied to or from a device.

If the ranged accessor is multi-dimensional, the sub-range is allowed to describe a region of memory in the underlying buffer that is not contiguous in the linear address space. It is also legal to construct several ranged accessors for the same underlying buffer, either overlapping or non-overlapping.

A ranged accessor still creates a requisite for the entire underlying buffer, even for the portions not within the range. For example, if one command writes through a ranged accessor to one region of a buffer and a second command reads through a ranged accessor from a non-overlapping region of the same buffer, the second command must still be scheduled after the first because the requisites for the two commands are on the entire buffer, not on the sub-ranges of the ranged accessors.

Most of the accessor member functions which provide a reference to the underlying buffer elements are affected by a ranged accessor's offset and range. For example, calling operator[](0) on a one-dimensional ranged accessor returns a reference to the element at the position specified by the accessor's offset, which is not necessarily the first element in the buffer. In addition, the accessor's iterator functions iterate only over the elements that are within the sub-range.

The only exceptions are the get_pointer and get_multi_ptr member functions, which return a pointer to the beginning of the underlying buffer regardless of the accessor's offset. Applications using these functions must take care to manually add the offset before dereferencing the pointer because accessing an element that is outside of the accessor's range results in undefined behavior.

There is no change in behavior for ranged accessors with a range of zero. It still creates a requisite for the entire underlying buffer, and an attempt to access an element produces undefined behavior.

4.7.6.9. Buffer accessor for commands

The accessor class provides access to data in a buffer from within a SYCL kernel function or from within a host task. When used in a SYCL kernel function, it accesses the contents of the buffer via the device's global memory. These two forms of the accessor are distinguished by the AccessTarget template parameter as shown in Table 54. Both forms support the following values for the AccessMode template parameter: access_mode::read, access_mode::write and access_mode::read_write.

Table 54. Description of access targets for buffer accessors

<table>
<thead>
<tr>
<th>Access target</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>target::device</td>
<td>Access a buffer from a SYCL kernel function via device global memory.</td>
</tr>
<tr>
<td>target::host_task</td>
<td>Access a buffer from a host task.</td>
</tr>
</tbody>
</table>

Programs which specify the access target as target::device and then capture the accessor in a host task can only use the accessor for interoperability through the interop_handle, any other uses result in undefined behavior.

Programs which specify the access target as target::host_task and then use the accessor from a SYCL
**kernel function** result in undefined behavior.

The dimensionality of the accessor must match the underlying buffer, however, there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

Certain accessor constructors create a "placeholder" accessor. Such an accessor is bound to a buffer and its semantics such as access target and access mode are defined. However, a placeholder accessor is not yet bound to a command group. Before such an accessor can be used in a command, it must be bound by calling handler::require(). If a placeholder accessor is passed as an argument to a command without first being bound to a command group with handler::require(), the implementation throws a synchronous exception with the errc::kernel_argument error code when the command is submitted.

### 4.7.6.9.1. Interface for buffer command accessors

A synopsis of the accessor class is provided below, showing the interface when it is specialized with target::device or target::host_task. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.12. The member types are listed in Table 79 and Table 55. The constructors are listed in Table 56, and the member functions are listed in Table 80 and Table 57.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between accessor types refer to Section 4.7.6.9.3. Additionally, accessors of the same type must be equality comparable both in the host application and also in SYCL kernel functions.

```cpp
namespace sycl {

enum class target : /* unspecified */ {
    device,
    host_task,
    constant_buffer, // Deprecated
    local,          // Deprecated
    host_buffer,    // Deprecated
    global_buffer = device // Deprecated
};

namespace access {

// The legacy type "access::target" is deprecated.
using sycl::target;

enum class placeholder : /* unspecified */ { // Deprecated
    false_t,
    true_t
};
}

} // namespace sycl
```

```cpp
template <typename DataT, int Dimensions = 1,
          access_mode AccessMode =
          (std::is_const_v<DataT> ? access_mode::read
                                   : access_mode::read_write),
          target AccessTarget = target::device,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor {
```

SYCL 2020 rev 7

Chapter 4. SYCL programming interface | 159
4.7.6.9.1. Interface for buffer command accessors

```cpp
public:
    using value_type = // const DataT for read-only accessors, DataT otherwise
        __value_type__;
    using reference = value_type&;
    using const_reference = const DataT&;
    template <access::decorated IsDecorated>
        using accessor_ptr = // multi_ptr to value_type with target address space,
            __pointer_class__; // unspecified for access_mode::host_task
    using iterator = __unspecified_iterator__<value_type>;
    using const_iterator = __unspecified_iterator__<const value_type>;
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
    using difference_type =
        typename std::iterator_traits<iterator>::difference_type;
    using size_type = size_t;

    accessor();

    /* Available only when: (Dimensions == 0) */
    template <typename AllocatorT>
        accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
            const property_list& propList = {});

    /* Available only when: (Dimensions == 0) */
    template <typename AllocatorT>
        accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
            handler& commandGroupHandlerRef, const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
        accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
            const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT, typename TagT>
        accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, TagT tag,
            const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT, typename TagT>
        accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
            handler& commandGroupHandlerRef, TagT tag,
            const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
        accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
            range<Dimensions> accessRange, const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT, typename TagT>
```
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    range<Dimensions> accessRange, TagT tag,
    const property_list& propList = {});

/* Available only when: (Dimensions > 0) */

template<typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    range<Dimensions> accessRange, id<Dimensions> accessOffset,
    const property_list& propList = {});

/* Available only when: (Dimensions > 0) */

template<typename AllocatorT, typename TagT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    handler& commandGroupHandlerRef, range<Dimensions> accessRange,
    TagT tag, const property_list& propList = {});

/* Available only when: (Dimensions > 0) */

template<typename AllocatorT, typename TagT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    handler& commandGroupHandlerRef, range<Dimensions> accessRange,
    id<Dimensions> accessOffset, TagT tag,
    const property_list& propList = {});

/* -- common interface members -- */

void swap(accessor& other);

bool is_placeholder() const;

size_type byte_size() const noexcept;

size_type size() const noexcept;

size_type max_size() const noexcept;

// Deprecated
size_t get_size() const;
size_t get_count() const;

bool empty() const noexcept;

/* Available only when: (Dimensions > 0) */
range<Dimensions> get_range() const;

/* Available only when: (Dimensions > 0) */
id<Dimensions> get_offset() const;

/* Available only when: (AccessMode != access_mode::atomic && Dimensions == 0) */
operator reference() const;

/* Available only when: (AccessMode != access_mode::atomic && Dimensions == 0) */
const accessor& operator=(const value_type& other) const;

/* Available only when: (AccessMode != access_mode::atomic && Dimensions == 0) */
const accessor& operator=(value_type&& other) const;

/* Available only when: (Dimensions > 0) */
reference operator[](id<Dimensions> index) const;

/* Available only when: (Dimensions > 1) */
__unspecified__ operator[](size_t index) const;

/* Available only when: (AccessMode != access_mode::atomic && Dimensions == 1) */
reference operator[](size_t index) const;

/* Deprecated
Available only when: (AccessMode == access_mode::atomic && Dimensions == 0) */
operator cl::sycl::atomic<DataT, access::address_space::global_space>() const;

/* Deprecated
Available only when: (AccessMode == access_mode::atomic && Dimensions == 1) */
cl::sycl::atomic<DataT, access::address_space::global_space>
operator[](id<Dimensions> index) const;

std::add_pointer_t<value_type> get_pointer() const noexcept;

template <access::decorated IsDecorated>
accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;

iterator begin() const noexcept;

iterator end() const noexcept;

const_iterator cbegin() const noexcept;

const_iterator cend() const noexcept;

reverse_iterator rbegin() const noexcept;
reverse_iterator rend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;
}

} // namespace sycl

Table 55. Member types of the accessor class

<table>
<thead>
<tr>
<th>Member types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; accessor_ptr</td>
<td>If (AccessTarget == target::device): multi_ptr&lt;value_type, access::address_space::global_space, IsDecorated&gt;. The definition of this type is not specified when (AccessTarget == target::host_task).</td>
</tr>
</tbody>
</table>

Table 56. Constructors of the accessor class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessor()</td>
<td>Constructs an empty accessor which fulfills the following post-conditions:</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt; &amp; bufferRef, const property_list &amp; propList = {})</td>
<td>Available only when (Dimensions == 0). Constructs a placeholder accessor for accessing the first element of a buffer. The optional property_list provides properties for the constructed accessor.</td>
</tr>
</tbody>
</table>

Available only when (Dimensions == 0). Constructs a placeholder accessor for accessing the first element of a buffer. The optional property_list provides properties for the constructed accessor.
### Interface for buffer command accessors

**Constructor**

<table>
<thead>
<tr>
<th>Description</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available only when ((\text{Dimensions} \leq 0)). Constructs an accessor for accessing the first element of a buffer within a SYCL kernel function on the queue associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed accessor.</td>
<td></td>
</tr>
</tbody>
</table>
| \[
\text{template } \langle \text{typename } \text{AllocatorT} \rangle \\
\text{accessor} (\text{buffer} < \text{DataT}, 1, \text{AllocatorT} > \& \text{bufferRef}, \text{handler} \& \text{commandGroupHandlerRef}, \text{const} \text{property_list} \& \text{propList} = \{\})
\] |

| Available only when \((\text{Dimensions} > 0)\). Constructs a placeholder accessor for accessing a buffer. The optional `property_list` provides properties for the constructed accessor. |
| \[
\text{template } \langle \text{typename } \text{AllocatorT} \rangle \\
\text{accessor} (\text{buffer} < \text{DataT}, \text{Dimensions}, \text{AllocatorT} > \& \text{bufferRef}, \text{const} \text{property_list} \& \text{propList} = \{\})
\] |

| Available only when \((\text{Dimensions} > 0)\). Constructs a placeholder accessor for accessing a buffer. The tag is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional `property_list` provides properties for the constructed accessor. |
| \[
\text{template } \langle \text{typename } \text{AllocatorT}, \text{typename } \text{TagT} \rangle \\
\text{accessor} (\text{buffer} < \text{DataT}, \text{Dimensions}, \text{AllocatorT} > \& \text{bufferRef}, \text{TagT} \text{tag}, \text{const} \text{property_list} \& \text{propList} = \{\})
\] |

| Available only when \((\text{Dimensions} > 0)\). Constructs an accessor for accessing a buffer within a SYCL kernel function on the queue associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed accessor. |
| \[
\text{template } \langle \text{typename } \text{AllocatorT} \rangle \\
\text{accessor} (\text{buffer} < \text{DataT}, \text{Dimensions}, \text{AllocatorT} > \& \text{bufferRef}, \text{handler} \& \text{commandGroupHandlerRef}, \text{const} \text{property_list} \& \text{propList} = \{\})
\] |

| Available only when \((\text{Dimensions} > 0)\). Constructs an accessor for accessing a buffer within a SYCL kernel function on the queue associated with `commandGroupHandlerRef`. The `tag` is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional `property_list` provides properties for the constructed accessor. |
| \[
\text{template } \langle \text{typename } \text{AllocatorT}, \text{typename } \text{TagT} \rangle \\
\text{accessor} (\text{buffer} < \text{DataT}, \text{Dimensions}, \text{AllocatorT} > \& \text{bufferRef}, \text{handler} \& \text{commandGroupHandlerRef}, \text{TagT} \text{tag}, \text{const} \text{property_list} \& \text{propList} = \{\})
\] |
## Constructor

**template <typename AllocatorT>**

```cpp
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
          range<Dimensions> accessRange, const
          property_list& propList = {})
```

### Description

Available only when $(\text{Dimensions} > 0)$.

Constructs a placeholder accessor that is a **ranged accessor**, where the range starts at the beginning of the `buffer`. The optional `property_list` provides properties for the constructed accessor.

Throws an **exception** with the `errc::invalid` error code if `accessRange` exceeds the range of `bufferRef` in any dimension.

---

**template <typename AllocatorT, typename TagT>**

```cpp
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
          range<Dimensions> accessRange, TagT tag,
          const property_list& propList = {})
```

### Description

Available only when $(\text{Dimensions} > 0)$.

Constructs a placeholder accessor that is a **ranged accessor**, where the range starts at the beginning of the `buffer`. The `tag` is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional `property_list` provides properties for the constructed accessor.

Throws an **exception** with the `errc::invalid` error code if `accessRange` exceeds the range of `bufferRef` in any dimension.

---

**template <typename AllocatorT>**

```cpp
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
          range<Dimensions> accessRange, id<Dimensions>
          accessOffset, 
          const property_list& propList = {})
```

### Description

Available only when $(\text{Dimensions} > 0)$.

Constructs a placeholder accessor that is a **ranged accessor**, where the range starts at an offset from the beginning of the `buffer`. The optional `property_list` provides properties for the constructed accessor.

Throws an **exception** with the `errc::invalid` error code if the sum of `accessRange` and `accessOffset` exceeds the range of `bufferRef` in any dimension.
### Constructor

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,\nrange&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset, TagT tag,\nconst property_list&amp; propList = {})</code></td>
<td>Available only when (Dimensions &gt; 0). Constructs a placeholder accessor that is a ranged accessor, where the range starts at an offset from the beginning of the buffer. The tag is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional property_list provides properties for the constructed accessor. Throws an exception with the errc::invalid error code if the sum of accessRange and accessOffset exceeds the range of bufferRef in any dimension.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef,\nhandler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange,\nconst property_list&amp; propList = {})</code></td>
<td>Available only when (Dimensions &gt; 0). Constructs an accessor that is a ranged accessor, where the range starts at the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor. Throws an exception with the errc::invalid error code if accessRange exceeds the range of bufferRef in any dimension.</td>
</tr>
<tr>
<td>Construct</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>template &lt;typename AllocatorT, typename TagT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange, TagT tag, const property_list&amp; propList = {})</code></td>
<td>Available only when <code>(Dimensions &gt; 0)</code>. Constructs an accessor that is a ranged accessor, where the range starts at the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The tag is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional <code>property_list</code> provides properties for the constructed accessor. Throws an exception with the <code>errc::invalid</code> error code if <code>accessRange</code> exceeds the range of <code>bufferRef</code> in any dimension.</td>
</tr>
<tr>
<td><code>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, handler&amp; commandGroupHandlerRef, range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset, const property_list&amp; propList = {})</code></td>
<td>Available only when <code>(Dimensions &gt; 0)</code>. Constructs an accessor that is a ranged accessor, where the range starts at an offset from the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional <code>property_list</code> provides properties for the constructed accessor. Throws an exception with the <code>errc::invalid</code> error code if the sum of <code>accessRange</code> and <code>accessOffset</code> exceeds the range of <code>bufferRef</code> in any dimension.</td>
</tr>
</tbody>
</table>
Constructor

```cpp
template <typename AllocatorT, typename TagT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
          handler& commandGroupHandlerRef, range<Dimensions> accessRange,
          id<Dimensions> accessOffset, TagT tag,
          const property_list& propList = {})
```

### Description

Available only when (Dimensions > 0).

Constructs an accessor that is a ranged accessor, where the range starts at an offset from the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The tag is used to deduce template arguments of the accessor as described in Section 4.7.6.9.2. The optional property_list provides properties for the constructed accessor.

Throws an exception with the errc::invalid error code if the sum of accessRange and accessOffset exceeds the range of bufferRef in any dimension.

### Table 57. Member functions of the accessor class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void swap(accessor&amp; other);</code></td>
<td>Swaps the contents of the current accessor with the contents of other.</td>
</tr>
<tr>
<td><code>bool is_placeholder() const</code></td>
<td>Returns true if the accessor was constructed as a placeholder. Otherwise returns false.</td>
</tr>
<tr>
<td><code>id&lt;Dimensions&gt; get_offset() const</code></td>
<td>Available only when (Dimensions &gt; 0). If this is a ranged accessor, returns the offset that was specified when the accessor was constructed. For other accessors, returns the default constructed <code>id&lt;Dimensions&gt;{}</code>.</td>
</tr>
<tr>
<td><code>template &lt;access::decorated IsDecorated&gt; accessor_ptr&lt;IsDecorated&gt; get_multi_ptr() const noexcept</code></td>
<td>Returns a multi_ptr to the start of this accessor's underlying buffer, even if this is a ranged accessor whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty. This function may only be called from within a command.</td>
</tr>
</tbody>
</table>
4.7.6.9.2. Deduction tags for buffer command accessors

Some `accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in Table 58 along with the access mode and accessor target that they imply.

Table 58. Enumeration of tags available for `accessor` construction

<table>
<thead>
<tr>
<th>Tag value</th>
<th>Access mode</th>
<th>Accessor target</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_write</td>
<td>access_mode::read_write</td>
<td>target::device</td>
</tr>
<tr>
<td>read_only</td>
<td>access_mode::read</td>
<td>target::device</td>
</tr>
<tr>
<td>write_only</td>
<td>access_mode::write</td>
<td>target::device</td>
</tr>
<tr>
<td>read_write_host_task</td>
<td>access_mode::read_write</td>
<td>target::host_task</td>
</tr>
<tr>
<td>read_only_host_task</td>
<td>access_mode::read</td>
<td>target::host_task</td>
</tr>
<tr>
<td>write_only_host_task</td>
<td>access_mode::write</td>
<td>target::host_task</td>
</tr>
</tbody>
</table>

4.7.6.9.3. Read only buffer command accessors and implicit conversions

Table 59 shows the specializations of `accessor` with `target::device` or `target::host_task` that are read-only accessors. There is an implicit conversion between any of these specializations, provided that all other template parameters are the same.

Table 59. Specializations of `accessor` that are read-only

<table>
<thead>
<tr>
<th>Data type</th>
<th>Access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>not const-qualified</td>
<td>access_mode::read</td>
</tr>
<tr>
<td>const-qualified</td>
<td>access_mode::read</td>
</tr>
</tbody>
</table>

There is also an implicit conversion from the read-write specialization shown in Table 60 to any of the read-only specializations shown in Table 59, provided that all other template parameters are the same.

Table 60. Specializations of `accessor` that are read-write
4.7.6.9.4. Deprecated features of the accessor class

All of the features defined in this section are deprecated and will likely be removed from a future version of the specification.

4.7.6.9.4.1. Aliased names

The enumerated value `target::global_buffer` is an alias for `target::device`. It has the same type and value as its alias.

The enumerated type `access::target` is an alias for `target`, and the enumerated type `access::mode` is an alias for `access_mode`.

4.7.6.9.4.2. Discard access modes

An accessor instance specialized with access mode `access_mode::discard_write` has the same behavior as an accessor instance of mode `access_mode::write` that is constructed with the property `property::no_init`.

An accessor instance specialized with access mode `access_mode::discard_read_write` has the same behavior as an accessor instance of mode `access_mode::read_write` that is constructed with the property `property::no_init`.

4.7.6.9.4.3. Placeholder template parameter

The accessor template parameter `IsPlaceholder` is allowed to be specified, but it has no bearing on whether the accessor instance is a placeholder. This is determined solely by the constructor used to create the instance.

The associated type `access::placeholder` is also deprecated.

4.7.6.9.4.4. Additional member functions for target::device specialization

Specializations of the accessor class with `target::device` have the additional member functions described in Table 61.

Table 61. Deprecated member functions of the accessor class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t get_size() const</code></td>
<td>Returns the same value as <code>byte_size()</code></td>
</tr>
<tr>
<td><code>size_t get_count() const</code></td>
<td>Returns the same value as <code>size()</code></td>
</tr>
</tbody>
</table>

4.7.6.9.4.5. Accessor specialization with target::constant_buffer

The accessor class may be specialized with target `target::constant_buffer`, which results in an accessor that can be used within a SYCL kernel function to access the contents of a buffer through the device's constant memory.

As with other accessor specializations, the dimensionality must match the underlying buffer, however there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.
This specialization of accessor is available only for the access mode access_mode::read.

This accessor type can be constructed as a "placeholder" accessor. As with other accessor specializations that are placeholders, handler::require() must be called before passing a placeholder accessor to a command. If the application neglects to call handler::require() on a placeholder accessor, the implementation throws a synchronous exception with the errc::kernel_argument error code when the command is submitted.

A synopsis for this specialization of accessor is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.9.4.8. The member types are listed in Table 68. The constructors are listed in Table 62 and the member functions are listed in Table 69 and Table 63.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. Additionally, accessors of the same type must be equality comparable.

```cpp
namespace sycl {

template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, access::placeholder IsPlaceholder>
class accessor {

public:
    using value_type = const DataT;
    using reference = const DataT&;
    using const_reference = const DataT&;

    /* Available only when: (Dimensions == 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
             const property_list& proplist = {});

    /* Available only when: (Dimensions == 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
             handler& commandGroupHandlerRef, const property_list& proplist = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
             const property_list& proplist = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
             handler& commandGroupHandlerRef, const property_list& proplist = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
             range<Dimensions> accessRange, const property_list& proplist = {});

    /* Available only when: (Dimensions > 0) */
    template <typename AllocatorT>
    accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
             range<Dimensions> accessRange, id<Dimensions> accessOffset,
```


```cpp
const property_list& propList = {});

template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    handler& commandGroupHandlerRef, range<Dimensions> accessRange,
    const property_list& propList = {});

template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
    handler& commandGroupHandlerRef, range<Dimensions> accessRange,
    id<Dimensions> accessOffset, const property_list& propList = {});

/* -- common interface members -- */

bool is_placeholder() const;

size_t get_size() const noexcept;

size_t get_count() const noexcept;

range<Dimensions> get_range() const;

id<Dimensions> get_offset() const;

operator reference() const;

operator[](id<Dimensions> index) const;

operator[](size_t index) const;

reference operator[](size_t index) const;

constant_ptr<DataT> get_pointer() const noexcept;

}; // namespace sycl
```

### Table 62. Constructors of the deprecated constant accessor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <typename AllocatorT>
accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
    const property_list& propList = {})` | Available only when (Dimensions == 0). Constructs a placeholder accessor for accessing the first element of a buffer. The optional `property_list` provides properties for the constructed accessor. |
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef, handler&amp; commandGroupHandlerRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions == 0). Constructs an accessor for accessing the first element of a buffer within a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs a placeholder accessor for accessing a buffer. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, handler&amp; commandGroupHandlerRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs an accessor for accessing a buffer within a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, range&lt;Dimensions&gt; accessRange, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs a placeholder accessor that is a ranged accessor, where the range starts at the beginning of the buffer. The optional property_list provides properties for the constructed accessor. Throws an exception with the errc::invalid error code if accessRange exceeds the range of bufferRef in any dimension.</td>
</tr>
</tbody>
</table>
Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
range<Dimensions> accessRange, id<Dimensions> accessOffset,
const property_list& propList = {})

Available only when (Dimensions > 0).
Constructs a placeholder accessor that is a ranged accessor, where the range starts at an offset from the beginning of the buffer. The optional property_list provides properties for the constructed accessor.

Throws an exception with the errc::invalid error code if the sum of accessRange and accessOffset exceeds the range of bufferRef in any dimension.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
handler& commandGroupHandlerRef, range<Dimensions> accessRange,
const property_list& propList = {})

Available only when (Dimensions > 0).
Constructs an accessor that is a ranged accessor, where the range starts at the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor.

Throws an exception with the errc::invalid error code if accessRange exceeds the range of bufferRef in any dimension.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
handler& commandGroupHandlerRef, range<Dimensions> accessRange,
id<Dimensions> accessOffset, const property_list& propList = {})

Available only when (Dimensions > 0).
Constructs an accessor that is a ranged accessor, where the range starts at an offset from the beginning of the buffer. The accessor can only be used in a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor.

Throws an exception with the errc::invalid error code if the sum of accessRange and accessOffset exceeds the range of bufferRef in any dimension.

Table 63. Member functions of the deprecated constant accessor
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool is_placeholder() const</code></td>
<td>Returns <code>true</code> if the accessor was constructed as a placeholder and returns <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>
| `id<Dimensions> get_offset() const` | Available only when `(Dimensions > 0)`.  
If this is a ranged accessor, returns the offset that was specified when the accessor was constructed, otherwise returns the default constructed `id<Dimensions>{}`. |
| `constant_ptr<DataT> get_pointer() const noexcept` | Returns a `multi_ptr` to the start of this accessor's underlying buffer, even if this is a ranged accessor whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty. |

### 4.7.6.9.4.6. Accessor specialization with `target::host_buffer`

The `accessor` class may be specialized with target `target::host_buffer`, which results in a host accessor similar to `host_accessor`. This specialization provides access to data in a `buffer` from host code that is outside of a `command`, and constructors of this specialization block until the requested data is available on the host.

As with other accessor specializations, the dimensionality must match the underlying buffer, however there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

This specialization of `accessor` is available for all access modes except for `access_mode::atomic`.

A synopsis for this specialization of `accessor` is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.9.4.8. The member types are listed in Table 68. The constructors are listed in Table 64, and the member functions are listed in Table 69 and Table 65.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. Additionally, accessors of the same type must be equality comparable.
template <typename AllocatorT>
accessor(buffer<DataT, 1, AllocatorT>& bufferRef, const property_list& propList = {});
/* Available only when: (Dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, const property_list& propList = {});
/* Available only when: (Dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, range<Dimensions> accessRange, const property_list& propList = {});
/* Available only when: (Dimensions > 0) */
template <typename AllocatorT>
accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, range<Dimensions> accessRange, id<Dimensions> accessOffset, const property_list& propList = {});
/* -- common interface members -- */
bool is_placeholder() const;
size_t get_size() const;
size_t get_count() const;
/* Available only when: (Dimensions > 0) */
range<Dimensions> get_range() const;
/* Available only when: (Dimensions > 0) */
id<Dimensions> get_offset() const;
/* Available only when: (Dimensions == 0) */
operator reference() const;
/* Available only when: (Dimensions > 0) */
reference operator[](id<Dimensions> index) const;
/* Available only when: (Dimensions > 1) */
__unspecified__ operator[](size_t index) const;
/* Available only when: (Dimensions == 1) */
reference operator[](size_t index) const;
std::add_pointer_t<value_type> get_pointer() const noexcept;
}; // namespace sycl

Table 64. Constructors of the deprecated host buffer accessor
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions == 0). Constructs an accessor for accessing the first element of a buffer immediately on the host. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs an accessor for accessing a buffer immediately on the host. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, range&lt;Dimensions&gt; accessRange, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs an accessor that is a ranged accessor which accesses a buffer immediately on the host, where the range starts at the beginning of the buffer. The optional property_list provides properties for the constructed accessor. Throws an exception with the errc::invalid error code if accessRange exceeds the range of bufferRef in any dimension.</td>
</tr>
<tr>
<td>template &lt;typename AllocatorT&gt; accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, range&lt;Dimensions&gt; accessRange, id&lt;Dimensions&gt; accessOffset, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0). Constructs an accessor that is a ranged accessor which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The optional property_list provides properties for the constructed accessor. Throws an exception with the errc::invalid error code if the sum of accessRange and accessOffset exceeds the range of bufferRef in any dimension.</td>
</tr>
</tbody>
</table>

*Table 65. Member functions of the deprecated host buffer accessor*
### 4.7.6.9.4.7. Accessor specialization with target::local

The `accessor` class may be specialized with target `target::local`, which results in a local accessor that has the same semantics and restrictions as `local_accessor`.

This specialization of `accessor` is only available for access modes `access_mode::read_write` and `access_mode::atomic`.

A synopsis for this specialization of `accessor` is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.9.4.8. The member types are listed in Table 68, and the member functions are listed in Table 69 and Table 67.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. Additionally, accessors of the same type must be equality comparable.

```cpp
namespace sycl {

    template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, access::placeholder IsPlaceholder>
    class accessor {

        using value_type = DataT;
        using reference = DataT&;
        using const_reference = const DataT&;

        /* Available only when: (Dimensions == 0) */
        accessor(handler& commandGroupHandlerRef, const property_list& propList = {});

        /* Available only when: (Dimensions > 0) */
        accessor(range<Dimensions> allocationSize, handler& commandGroupHandlerRef, const property_list& propList = {});

        /* -- common interface members -- */
    }

};
```
Table 66. Constructors of the deprecated local accessor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| accessor(handler& commandGroupHandlerRef, const property_list& propList = {}) | Available only when (Dimensions == 0).

Constructs an accessor instance for accessing local memory of a single DataT element within a SYCL kernel function on the queue associated with commandGroupHandlerRef. The optional property_list provides properties for the constructed accessor.
Constructor

```cpp
accessor(range<Dimensions> allocationSize, handler& commandGroupHandlerRef,
         const property_list& propList = {})
```

| Description | Available only when \((\text{Dimensions} > 0)\).

Constructs an accessor instance for accessing local memory of an array of DataT elements within a SYCL kernel function on the queue associated with commandGroupHandlerRef. The number of elements in the array is defined by allocationSize. The optional property_list provides properties for the constructed accessor.

<table>
<thead>
<tr>
<th>Table 67. Member functions of the deprecated local accessor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member function</strong></td>
</tr>
</tbody>
</table>

| operator atomic<DataT, access::address_space::local_space>() const | Available only when \((\text{AccessMode} == \text{access_mode::atomic} \&\& \text{Dimensions} == 0)\). Returns an instance of atomic of type DataT providing atomic access to the element stored within the work-group’s local memory allocation that this accessor is accessing. |
| atomic<DataT, access::address_space::local_space>() operator[](id<Dimensions> index) const | Available only when \((\text{AccessMode} == \text{access_mode::atomic} \&\& \text{Dimensions} > 0)\). Returns an instance of atomic of type DataT providing atomic access to the element stored within the work-group’s local memory allocation that this accessor is accessing, at the index specified by index. |
| atomic<DataT, access::address_space::local_space>() operator[](size_t index) const | Available only when \((\text{AccessMode} == \text{access_mode::atomic} \&\& \text{Dimensions} == 1)\). Returns an instance of atomic of type DataT providing atomic access to the element stored within the work-group’s local memory allocation that this accessor is accessing, at the index specified by index. |
| local_ptr<DataT> get_pointer() const noexcept | Returns a multi_ptr to the work-group’s local memory allocation that this accessor is accessing. The return value is unspecified if the accessor is empty. |
4.7.6.9.4.8. Common members for deprecated accessors

Specializations of the accessor class with target::constant_buffer, target::host_buffer and target::local have many member types and member functions with the same name and meaning. Table 68 describes these common types and Table 69 describes the common member functions.

Table 68. Common member types of the deprecated accessors

<table>
<thead>
<tr>
<th>Member types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_type</td>
<td>If (AccessMode == access_mode::read), equal to const DataT, otherwise equal to DataT.</td>
</tr>
<tr>
<td>reference</td>
<td>Equal to value_type&amp;.</td>
</tr>
<tr>
<td>const_reference</td>
<td>Equal to const DataT&amp;.</td>
</tr>
</tbody>
</table>

Table 69. Common member functions of the deprecated accessors

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t get_size() const noexcept</td>
<td>Returns the size in bytes of the memory region this accessor may access.</td>
</tr>
</tbody>
</table>

- When AccessTarget is target::constant_buffer or target::host_buffer, the returned value is the size of the elements in the underlying buffer, unless this is a ranged accessor in which case it is the size of the elements within the accessor's range.

- When AccessTarget is target::local, the returned value is the size in bytes of the accessor's local memory allocation, per work-group.
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `size_t get_count() const noexcept` | Returns the number of `DataT` elements of the memory region this accessor may access.  
When `AccessTarget` is `target::constant_buffer` or `target::host_buffer`, the returned value is the number of elements in the underlying buffer, unless this is a ranged accessor in which case it is the number of elements within the accessor's range.  
When `AccessTarget` is `target::local`, the returned value is the number of elements in the accessor's local memory allocation, per work-group. |
| `range<Dimensions> get_range() const` | Available only when `(Dimensions > 0)`.  
Returns a range object which represents the number of elements of `DataT` per dimension that this accessor may access.  
When `AccessTarget` is `target::constant_buffer` or `target::host_buffer`, the returned value is the range of the underlying buffer, unless this is a ranged accessor in which case it is the range that was specified when the accessor was constructed.  
When `AccessTarget` is `target::local`, the returned value is the range that was specified when the accessor was constructed. |
| `operator reference() const` | When `AccessTarget` is `target::constant_buffer` or `target::host_buffer`, available only when `(Dimensions == 0)`.  
When `AccessTarget` is `target::local`, available only when `(AccessMode == access_mode::read_write && Dimensions == 0)`.  
Returns a reference to the single element that is accessed by this accessor. |
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reference operator[](id&lt;Dimensions&gt; index) const</code></td>
<td>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, available only when <code>(Dimensions &gt; 0)</code>.</td>
</tr>
<tr>
<td></td>
<td>When <code>AccessTarget</code> is <code>target::local</code>, available only when <code>(AccessMode == access_mode::read_write &amp;&amp; Dimensions &gt; 0)</code>.</td>
</tr>
<tr>
<td></td>
<td>Returns a reference to the element at the location specified by <code>index</code>. If this is a ranged accessor, the element is determined by adding <code>index</code> to the accessor’s offset.</td>
</tr>
<tr>
<td><code>__unspecified__ operator[](size_t index) const</code></td>
<td>Available only when <code>(Dimensions &gt; 1)</code>.</td>
</tr>
<tr>
<td></td>
<td>Returns an instance of an undefined intermediate type representing this accessor, with the dimensionality <code>Dimensions-1</code> and containing an implicit <code>id</code> with index <code>Dimensions</code> set to <code>index</code>. The intermediate type returned must provide all available subscript operators which take a <code>size_t</code> parameter defined by this accessor class that are appropriate for the type it represents (including this subscript operator).</td>
</tr>
<tr>
<td></td>
<td>If this is a ranged accessor, the implicit <code>id</code> in the returned instance also includes the accessor’s offset.</td>
</tr>
<tr>
<td><code>reference operator[](size_t index) const</code></td>
<td>When <code>AccessTarget</code> is <code>target::constant_buffer</code> or <code>target::host_buffer</code>, available only when <code>(Dimensions == 1)</code>.</td>
</tr>
<tr>
<td></td>
<td>When <code>AccessTarget</code> is <code>target::local</code>, available only when <code>(AccessMode == access_mode::read_write &amp;&amp; Dimensions == 1)</code>.</td>
</tr>
<tr>
<td></td>
<td>Returns a reference to the element at the location specified by <code>index</code>. If this is a ranged accessor, the element is determined by adding <code>index</code> to the accessor’s offset.</td>
</tr>
</tbody>
</table>

### 4.7.6.9.4.9. Accessor specialization with `access_mode::atomic`

The `accessor` class may be specialized with target `target::device` and access mode `access_mode::atomic`. This specialization provides additional member functions beyond those that are provided for other tar-

---

Chapter 4. SYCL programming interface | 183
The `host_accessor` class provides access to data in a `buffer` from host code that is outside of a `command` (i.e. do not use this class to access a buffer inside a host task).

As with `accessor`, the dimensionality of `host_accessor` must match the underlying buffer, however, there is a special case if the buffer is one-dimensional. In this case, the accessor may either be one-dimensional or it may be zero-dimensional. A zero-dimensional accessor has access to just the first element of the buffer, whereas a one-dimensional accessor has access to the entire buffer.

The `host_accessor` class supports the following access modes: `access_mode::read`, `access_mode::write` and `access_mode::read_write`. 
4.7.6.10.1. Interface for buffer host accessors

A synopsis of the `host_accessor` class is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.12. The member types are listed in Table 79. The constructors are listed in Table 71, and the member functions are listed in Table 80 and Table 72.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between accessor types refer to Section 4.7.6.10.3. Additionally, accessors of the same type must be equality comparable.

```cpp
namespace sycl {
  template <typename DataT, int Dimensions = 1,
            access_mode AccessMode =
            (std::is_const_v<DataT> ? access_mode::read
                                : access_mode::read_write)
  class host_accessor {
    public:
      using value_type = // const DataT for read-only accessors, DataT otherwise
        __value_type__;
      using reference = value_type&;
      using const_reference = const DataT&;
      using iterator = __unspecified_iterator<value_type>;
      using const_iterator = __unspecified_iterator<const value_type>;
      using reverse_iterator = std::reverse_iterator<iterator>;
      using const_reverse_iterator = std::reverse_iterator<const_iterator>;
      using difference_type =
        typename std::iterator_traits<iterator>::difference_type;
      using size_type = size_t;

      host_accessor();

      /* Available only when: (Dimensions == 0) */
      template <typename AllocatorT>
      host_accessor(buffer<DataT, 1, AllocatorT>& bufferRef,
                    const property_list& propList = {});

      /* Available only when: (Dimensions > 0) */
      template <typename AllocatorT>
      host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
                    const property_list& propList = {});

      /* Available only when: (Dimensions > 0) */
      template <typename AllocatorT, typename TagT>
      host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef, TagT tag,
                    const property_list& propList = {});

      /* Available only when: (Dimensions > 0) */
      template <typename AllocatorT>
      host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
                    range<Dimensions> accessRange,
                    const property_list& propList = {});

      /* Available only when: (Dimensions > 0) */
      template <typename AllocatorT, typename TagT>
      host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
                    const property_list& propList = {});
  }
}
```
template<
typename AllocatorT
>
host_accessor(
buffer<DataT, Dimensions, AllocatorT>& bufferRef,
range<Dimensions> accessRange, TagT tag,
const property_list& propList = {});

/* Available only when: (Dimensions > 0) */
host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
range<Dimensions> accessRange, id<Dimensions> accessOffset,
TagT tag, const property_list& propList = {});

/* Available only when: (Dimensions > 0) */
host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
range<Dimensions> accessRange, id<Dimensions> accessOffset,
TagT tag, const property_list& propList = {});

/* -- common interface members -- */

void swap(host_accessor& other);

size_type byte_size() const noexcept;

size_type size() const noexcept;

size_type max_size() const noexcept;

bool empty() const noexcept;

/* Available only when: (Dimensions > 0) */
range<Dimensions> get_range() const;

/* Available only when: (Dimensions > 0) */
id<Dimensions> get_offset() const;

/* Available only when: (Dimensions == 0) */
reference operator[](id<Dimensions> index) const;

/* Available only when: (Dimensions > 1) */
__unspecified__ operator[](size_t index) const;

/* Available only when: (Dimensions == 1) */
reference operator[](size_t index) const;

std::add_pointer_t<value_type> get_pointer() const noexcept;

iterator begin() const noexcept;

iterator end() const noexcept;
const_iterator cbegin() const noexcept;
const_iterator cend() const noexcept;
reverse_iterator rbegin() const noexcept;
reverse_iterator rend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

}; // namespace sycl

Table 71. Constructors of the host_accessor class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host_accessor()</td>
<td>Constructs an empty accessor which fulfills the following post-conditions:</td>
</tr>
<tr>
<td></td>
<td>• (empty() == true)</td>
</tr>
<tr>
<td></td>
<td>• All size queries return 0.</td>
</tr>
<tr>
<td></td>
<td>• The return value of get_pointer() is unspecified.</td>
</tr>
<tr>
<td></td>
<td>• Trying to access the underlying memory is undefined behavior.</td>
</tr>
<tr>
<td>template&lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, 1, AllocatorT&gt;&amp; bufferRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions == 0).</td>
</tr>
<tr>
<td></td>
<td>Constructs a host_accessor for accessing the first element of a buffer immediately on the host. The optional property_list provides properties for the constructed accessor.</td>
</tr>
<tr>
<td>template&lt;typename AllocatorT&gt; host_accessor(buffer&lt;DataT, Dimensions, AllocatorT&gt;&amp; bufferRef, const property_list&amp; propList = {})</td>
<td>Available only when (Dimensions &gt; 0).</td>
</tr>
<tr>
<td></td>
<td>Constructs a host_accessor for accessing a buffer immediately on the host. The optional property_list provides properties for the constructed accessor.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Description</th>
<th>Template</th>
</tr>
</thead>
</table>
| Available only when ($\text{Dimensions} > 0$). | \[
\text{host_accessor}(\text{buffer}<\text{DataT, Dimensions, AllocatorT}>& \text{bufferRef, TagT tag, const property_list}\& \text{propList} = \{\})
\] |

Constructs a `host_accessor` for accessing a `buffer` immediately on the host. The `tag` is used to deduce template arguments of the accessor as described in Section 4.7.6.10.2. The optional `property_list` provides properties for the constructed accessor.

<table>
<thead>
<tr>
<th>Description</th>
<th>Template</th>
</tr>
</thead>
</table>
| Available only when ($\text{Dimensions} > 0$). | \[
\text{host_accessor}(\text{buffer}<\text{DataT, Dimensions, AllocatorT}>& \text{bufferRef, range}\text{<Dimensions>} \text{accessRange, const property_list}\& \text{propList} = \{\})
\] |

Constructs a `host_accessor` that is a `ranged accessor` which accesses a `buffer` immediately on the host, where the range starts at the beginning of the `buffer`. The optional `property_list` provides properties for the constructed accessor.

Throws an `exception` with the `errc::invalid` error code if `accessRange` exceeds the range of `bufferRef` in any dimension.

<table>
<thead>
<tr>
<th>Description</th>
<th>Template</th>
</tr>
</thead>
</table>
| Available only when ($\text{Dimensions} > 0$). | \[
\text{host_accessor}(\text{buffer}<\text{DataT, Dimensions, AllocatorT}>& \text{bufferRef, range}\text{<Dimensions>} \text{accessRange, TagT tag, const property_list}\& \text{propList} = \{\})
\] |

Constructs a `host_accessor` that is a `ranged accessor` which accesses a `buffer` immediately on the host, where the range starts at the beginning of the `buffer`. The `tag` is used to deduce template arguments of the accessor as described in Section 4.7.6.10.2. The optional `property_list` provides properties for the constructed accessor.

Throws an `exception` with the `errc::invalid` error code if `accessRange` exceeds the range of `bufferRef` in any dimension.
### Constructor

```cpp
template <typename AllocatorT>
host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
               range<Dimensions> accessRange, id<Dimensions> accessOffset,
               const property_list& propList = {})  
```

Available only when \((\text{Dimensions} > 0)\).

Constructs a `host_accessor` that is a `ranged accessor` which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The optional `property_list` provides properties for the constructed accessor.

Throws an `exception` with the `errc::invalid` error code if the sum of `accessRange` and `accessOffset` exceeds the range of `bufferRef` in any dimension.

```cpp
template <typename AllocatorT, typename TagT>
host_accessor(buffer<DataT, Dimensions, AllocatorT>& bufferRef,
               range<Dimensions> accessRange, id<Dimensions> accessOffset,
               TagT tag, const property_list& propList = {})  
```

Available only when \((\text{Dimensions} > 0)\).

Constructs a `host_accessor` that is a `ranged accessor` which accesses a buffer immediately on the host, where the range starts at an offset from the beginning of the buffer. The `tag` is used to deduce template arguments of the accessor as described in Section 4.7.6.10.2. The optional `property_list` provides properties for the constructed accessor.

Throws an `exception` with the `errc::invalid` error code if the sum of `accessRange` and `accessOffset` exceeds the range of `bufferRef` in any dimension.

---

### Table 72. Member functions of the `host_accessor` class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void swap(host_accessor&amp; other);</td>
<td>Swaps the contents of the current accessor with the contents of <code>other</code>. Available only when ((\text{Dimensions} &gt; 0)). If this is a <code>ranged accessor</code>, returns the offset that was specified when the accessor was constructed. For other accessors, returns the default constructed <code>id&lt;Dimensions&gt;{}</code>.</td>
</tr>
</tbody>
</table>
4.7.6.10.2. Deduction tags for buffer host accessors

Some `host_accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in Table 73 along with the access mode that they imply.

**Table 73. Enumeration of tags available for host_accessor construction**

<table>
<thead>
<tr>
<th>Tag value</th>
<th>Access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_write</td>
<td><code>access_mode::read_write</code></td>
</tr>
<tr>
<td>read_only</td>
<td><code>access_mode::read</code></td>
</tr>
<tr>
<td>write_only</td>
<td><code>access_mode::write</code></td>
</tr>
</tbody>
</table>

4.7.6.10.3. Read only buffer host accessors and implicit conversions

Table 74 shows the specializations of `host_accessor` that are read-only accessors. There is an implicit conversion between any of these specializations, provided that all other template parameters are the same.

**Table 74. Specializations of host_accessor that are read-only**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>not const-qualified</td>
<td><code>access_mode::read</code></td>
</tr>
<tr>
<td>const-qualified</td>
<td><code>access_mode::read</code></td>
</tr>
</tbody>
</table>

There is also an implicit conversion from the read-write `host_accessor` type shown in Table 75 to any of the read-only accessors in Table 74, provided that all other template parameters are the same.

**Table 75. Specializations of host_accessor that are read-write**

<table>
<thead>
<tr>
<th>Data type</th>
<th>Access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>not const-qualified</td>
<td><code>access_mode::read_write</code></td>
</tr>
</tbody>
</table>

4.7.6.11. Local accessor

The `local_accessor` class allocates device local memory and provides access to this memory from within a SYCL kernel function. The local memory that is allocated is shared between all work items of a work-group. If multiple work-groups execute simultaneously in an implementation, each work-group receives its own independent copy of the allocated local memory.

The underlying `DataT` type can be any C++ type that the device supports. If `DataT` is an implicit-lifetime type (as defined in the C++ core language), the local accessor implicitly creates objects of that type with indeterminate values. For other types, the local accessor merely allocates uninitialized memory, and the
application is responsible for constructing objects in that memory (e.g. by calling placement-new).

A local accessor must not be used in a SYCL kernel function that is invoked via `single_task` or via the simple form of `parallel_for` that takes a range parameter. In these cases submitting the kernel to a queue must throw a synchronous exception with the `errc::kernel_argument` error code.

### 4.7.6.11.1. Interface for local accessors

A synopsis of the `local_accessor` class is provided below. Since some of the class types and member functions have the same name and meaning as other accessors, the common types and functions are described in Section 4.7.6.12. The member types are listed in Table 79 and Table 76. The constructors are listed in Table 77, and the member functions are listed in Table 80 and Table 78.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between accessor types refer to Section 4.7.6.11.2. Additionally, accessors of the same type must be equality comparable.

```
namespace sycl {

    template <typename DataT, int Dimensions = 1> class local_accessor {

    public:
        using value_type = // const DataT for read-only accessors, DataT otherwise
            __value_type__;

        using reference = value_type&;

        using const_reference = const DataT&;

        template <access::decorated IsDecorated>

        using accessor_ptr =
            multi_ptr<
                value_type, access::address_space::local_space, IsDecorated>;

        using iterator = __unspecified_iterator<
            value_type>;

        using const_iterator = __unspecified_iterator<const value_type>;

        using reverse_iterator = std::reverse_iterator<iterator>;

        using const_reverse_iterator = std::reverse_iterator<const_iterator>;

        using difference_type =
            typename std::iterator_traits<iterator>::difference_type;

        using size_type = size_t;

    local_accessor();

    /* Available only when: (Dimensions == 0) */
    local_accessor(handler& commandGroupHandlerRef,
        const property_list& propList = {});

    /* Available only when: (Dimensions > 0) */
    local_accessor(range<Dimensions> allocationSize,
        handler& commandGroupHandlerRef,
        const property_list& propList = {});

    /* -- common interface members -- */

    void swap(accessor& other);

    size_type byte_size() const noexcept;

    size_type size() const noexcept;

    size_type max_size() const noexcept;

    }

} // namespace
```
4.7.6.11. Interface for local accessors

```cpp
bool empty() const noexcept;

range<Dimensions> get_range() const;

/* Available only when: (Dimensions == 0) */
operator reference() const;

/* Available only when: (!std::is_const_v<DataT> && Dimensions == 0) */
const local_accessor& operator=(const value_type& other) const;

/* Available only when: (!std::is_const_v<DataT> && Dimensions == 0) */
const local_accessor& operator=(value_type&& other) const;

/* Available only when: (Dimensions > 0) */
reference operator[](id<Dimensions> index) const;

/* Available only when: (Dimensions > 1) */
__unspecified__ operator[](size_t index) const;

/* Available only when: (Dimensions == 1) */
reference operator[](size_t index) const;

std::add_pointer_t<value_type> get_pointer() const noexcept;

template <access::decorated IsDecorated>
accessor_ptr<IsDecorated> get_multi_ptr() const noexcept;

iterator begin() const noexcept;

iterator end() const noexcept;

const_iterator cbegin() const noexcept;

const_iterator cend() const noexcept;

reverse_iterator rbegin() const noexcept;

reverse_iterator rend() const noexcept;

const_reverse_iterator crbegin() const noexcept;

const_reverse_iterator crend() const noexcept;

} // namespace sycl
```

### Table 76. Member types of the `local_accessor` class

<table>
<thead>
<tr>
<th>Member types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; accessor_ptr</td>
<td>Equal to <code>multi_ptr&lt;value_type, access::address_space::local_space, IsDecorated&gt;</code></td>
</tr>
</tbody>
</table>

### Table 77. Constructors of the `local_accessor` class

---

192 | Chapter 4. SYCL programming interface
### Constructor

**local_accessor()**

Constructs an empty local accessor which fulfills the following post-conditions:

- `(empty() == true)`
- All size queries return 0.
- The return values of `get_pointer()` and `get_multi_ptr()` are unspecified.
- Trying to access the underlying memory is undefined behavior.

**local_accessor(handler& commandGroupHandlerRef, const property_list& propList = {})**

Available only when `(Dimensions == 0)`.

Constructs a `local_accessor` for accessing local memory of a single `DataT` element within a SYCL kernel function on the queue associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed accessor.

**local_accessor(range<Dimensions> allocationSize, handler& commandGroupHandlerRef, const property_list& propList = {})**

Available only when `(Dimensions > 0)`.

Constructs a `local_accessor` for accessing local memory of an array of `DataT` elements within a SYCL kernel function on the queue associated with `commandGroupHandlerRef`. The number of elements in the array is defined by `allocationSize`. The optional `property_list` provides properties for the constructed accessor.

### Table 78. Member functions of the `local_accessor` class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void swap(local_accessor&amp; other);</strong></td>
<td>Swaps the contents of the current accessor with the contents of <code>other</code>.</td>
</tr>
<tr>
<td><strong>template &lt;access::decorated IsDecorated&gt; accessor_ptr&lt;IsDecorated&gt; get_multi_ptr() const noexcept</strong></td>
<td>Returns a <code>multi_ptr</code> to the start of the accessor's local memory region which corresponds to the calling work-group. The return value is unspecified if the accessor is empty. This function may only be called from within a SYCL kernel function.</td>
</tr>
</tbody>
</table>
**Member function**

<table>
<thead>
<tr>
<th>const local_accessor&amp; operator=(const value_type&amp; other) const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Available only when (!std::is_const_v&lt;DataT&gt; &amp;&amp; Dimensions == 0). Assignment to the single element that is accessed by this accessor. This function may only be called from within a command.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>const local_accessor&amp; operator=(const value_type&amp;&amp; other) const</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: Available only when (!std::is_const_v&lt;DataT&gt; &amp;&amp; Dimensions == 0). Assignment to the single element that is accessed by this accessor. This function may only be called from within a command.</td>
</tr>
</tbody>
</table>

### 4.7.6.11.2. Read only local accessors and implicit conversions

Since `local_accessor` has no template parameter for the access mode, the only specialization for a read-only local accessor is by providing a `const` qualified `DataT` parameter. Specializations with a non-`const` qualified `DataT` parameter are read-write. There is an implicit conversion from the read-write specialization to the read-only specialization, provided that all other template parameters are the same.

### 4.7.6.12. Common members for buffer and local accessors

The accessor, `host_accessor`, and `local_accessor` classes have many member types and member functions with the same name and meaning. Table 79 describes these common types and Table 80 describes the common member functions.

**Table 79. Common buffer and local accessor member types**

<table>
<thead>
<tr>
<th>Member types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_type</td>
<td>If the accessor is read-only, equal to <code>const DataT</code>, otherwise equal to <code>DataT</code>. See Section 4.7.6.9.3, Section 4.7.6.10.3 and Section 4.7.6.11.2 for which accessors are considered read-only.</td>
</tr>
<tr>
<td>reference</td>
<td>Equal to <code>value_type&amp;</code>.</td>
</tr>
<tr>
<td>const_reference</td>
<td>Equal to <code>const DataT&amp;</code>.</td>
</tr>
</tbody>
</table>
### Member types

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator that can provide ranged access. Cannot be written to if the accessor is read-only. The underlying pointer is address space qualified for <code>accessor</code> specializations with <code>target::device</code> and for <code>local_accessor</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator that can provide ranged access. Cannot be written to. The underlying pointer is address space qualified for <code>accessor</code> specializations with <code>target::device</code> and for <code>local_accessor</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator adaptor that reverses the direction of <code>iterator</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterator adaptor that reverses the direction of <code>const_iterator</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to <code>typename std::iterator_traits&lt;iterator&gt;::difference_type</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to <code>size_t</code>.</td>
</tr>
</tbody>
</table>

### Table 80. Common buffer and local accessor member functions

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| Returns the size in bytes of the memory region this accessor may access. 
For a buffer accessor this is the size of the underlying buffer, unless it is a `ranged accessor` in which case it is the size of the elements within the accessor's range. 
For a local accessor this is the size of the accessor's local memory allocation, per work-group. |
### Member Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_type size() const noexcept</code></td>
<td>Returns the number of <code>DataT</code> elements of the memory region this accessor may access. For a buffer accessor this is the number of elements in the underlying buffer, unless it is a ranged accessor in which case it is the number of elements within the accessor's range. For a local accessor this is the number of elements in the accessor's local memory allocation, per workgroup.</td>
</tr>
<tr>
<td><code>size_type max_size() const noexcept</code></td>
<td>Returns the maximum number of elements any accessor of this type would be able to access.</td>
</tr>
<tr>
<td><code>bool empty() const noexcept</code></td>
<td>Returns <code>true</code> if <code>(size() == 0)</code>.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions&gt; get_range() const</code></td>
<td>Available only when <code>(Dimensions &gt; 0)</code>. Returns a <code>range</code> object which represents the number of elements of <code>DataT</code> per dimension that this accessor may access. For a buffer accessor this is the range of the underlying buffer, unless it is a ranged accessor in which case it is the range that was specified when the accessor was constructed.</td>
</tr>
<tr>
<td><code>operator reference() const</code></td>
<td>For accessor available only when <code>(AccessMode != access_mode::atomic &amp;&amp; Dimensions == 0)</code>. For <code>host_accessor</code> and <code>local_accessor</code> available only when <code>(Dimensions == 0)</code>. Returns a reference to the single element that is accessed by this accessor. For accessor and local_accessor, this function may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>reference operator[](id&lt;Dimensions&gt; index) const</td>
<td>For <strong>accessor</strong> available only when (AccessMode != access_mode::atomic &amp;&amp; Dimensions &gt; 0). For <strong>host_accessor</strong> and <strong>local_accessor</strong> available only when (Dimensions &gt; 0). Returns a reference to the element at the location specified by index. If this is a <strong>ranged accessor</strong>, the element is determined by adding index to the accessor's offset. For <strong>accessor</strong> and <strong>local_accessor</strong>, this function may only be called from within a <strong>command</strong>.</td>
</tr>
<tr>
<td><strong>unspecified</strong> operator[](size_t index) const</td>
<td>Available only when (Dimensions &gt; 1). Returns an instance of an undefined intermediate type representing this accessor, with the dimensionality Dimensions-1 and containing an implicit id with index Dimensions set to index. The intermediate type returned must provide all available subscript operators which take a size_t parameter defined by this accessor class that are appropriate for the type it represents (including this subscript operator). If this is a <strong>ranged accessor</strong>, the implicit id in the returned instance also includes the accessor's offset. For <strong>accessor</strong> and <strong>local_accessor</strong>, this function may only be called from within a <strong>command</strong>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>reference operator[](size_t index) const</code></td>
<td>For <code>accessor</code> available only when <code>(AccessMode != access_mode::atomic &amp;&amp; Dimensions == 1).</code> For <code>host_accessor</code> and <code>local_accessor</code> available only when <code>(Dimensions == 1).</code> Returns a reference to the element at the location specified by <code>index</code>. If this is a ranged accessor, the element is determined by adding <code>index</code> to the accessor's offset. For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td><code>std::add_pointer_t&lt;value_type&gt; get_pointer() const noexcept</code></td>
<td>Returns a pointer to the start of this accessor’s memory. For a buffer accessor this is a pointer to the start of the underlying buffer, even if this is a ranged accessor whose range does not start at the beginning of the buffer. The return value is unspecified if the accessor is empty. For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td><code>iterator begin() const noexcept</code></td>
<td>Returns an iterator to the first element of the memory this accessor may access. For a buffer accessor this is an iterator to the first element of the underlying buffer, unless this is a ranged accessor in which case it is an iterator to first element within the accessor's range. For <code>accessor</code> and <code>local_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iterator end() const noexcept</td>
<td>Returns an iterator to one element past the last element of the memory this accessor may access. For a buffer accessor this is an iterator to one element past the last element in the underlying buffer, unless this is a ranged accessor in which case it is an iterator to one element past the last element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
<tr>
<td>const_iterator cbegin() const noexcept</td>
<td>Returns a const iterator to the first element of the memory this accessor may access. For a buffer accessor this is a const iterator to the first element of the underlying buffer, unless this is a ranged accessor in which case it is a const iterator to first element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
<tr>
<td>const_iterator cend() const noexcept</td>
<td>Returns a const iterator to one element past the last element of the memory this accessor may access. For a buffer accessor this is a const iterator to one element past the last element in the underlying buffer, unless this is a ranged accessor in which case it is a const iterator to one element past the last element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reverse_iterator rbegin()</code> <code>const noexcept</code></td>
<td>Returns an iterator adaptor to the last element of the memory this accessor may access. For a buffer accessor this is an iterator adaptor to the last element of the underlying buffer, unless this is a ranged accessor in which case it is an iterator adaptor to the last element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
<tr>
<td><code>reverse_iterator rend()</code> <code>const noexcept</code></td>
<td>Returns an iterator adaptor to one element before the first element of the memory this accessor may access. For a buffer accessor this is an iterator adaptor to one element before the first element in the underlying buffer, unless this is a ranged accessor in which case it is an iterator adaptor to one element before the first element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
<tr>
<td><code>const_reverse_iterator crbegin()</code> <code>const noexcept</code></td>
<td>Returns a const iterator adaptor to the last element of the memory this accessor may access. For a buffer accessor this is a const iterator adaptor to the last element of the underlying buffer, unless this is a ranged accessor in which case it is a const iterator adaptor to last element within the accessor's range. For accessor and local_accessor, this function may only be called from within a command.</td>
</tr>
</tbody>
</table>
### 4.7.6.13. Unsampled image accessors

There are two classes which implement accessors for unsampled images, `unsampled_image_accessor` and `host_unsampled_image_accessor`. The former provides access from within a SYCL kernel function or from within a host task. The latter provides access from host code that is outside of a host task.

The dimensionality of an unsampled image accessor must match the dimensionality of the underlying image to which it provides access. Both unsampled image accessor classes support the `access_mode::read` and `access_mode::write` access modes. In addition, the `host_unsampled_image_accessor` class supports `access_mode::read_write`.

The `AccessTarget` template parameter dictates how the `unsampled_image_accessor` can be used: `image_target::device` means the accessor can be used in a SYCL kernel function while `image_target::host_task` means the accessor can be used in a host task. Programs which specify this template parameter as `image_target::device` and then use the `unsampled_image_accessor` from a host task are ill formed. Likewise, programs which specify this template parameter as `image_target::host_task` and then use the `unsampled_image_accessor` from a SYCL kernel function are ill formed.

#### 4.7.6.13.1. Interface for unsampled image accessors

A synopsis of the two unsampled image accessor classes is provided below. Both classes have member types with the same name, which are described in Table 81. The constructors for the two classes are described in Table 82 and Table 83. Both classes also have member functions with the same name, which are described in Table 84.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between unsampled accessor types refer to Section 4.7.6.13.3.

Two `unsampled_image_accessor` objects of the same type must be equality comparable in both the host code and in SYCL kernel functions. Two `host_unsampled_image_accessor` objects of the same type must be equality comparable in the host code.
template <typename DataT, int Dimensions, access_mode AccessMode, 
    image_target AccessTarget = image_target::device>
class unsampled_image_accessor {
    public:
        using value_type = // const DataT for read-only accessors, DataT otherwise
            __value_type__;
        using reference = value_type&;
        using const_reference = const DataT&;

        template <typename AllocatorT>
        unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef,
            handler& commandGroupHandlerRef,
            const property_list& propList = {});

        template <typename AllocatorT, typename TagT>
        unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef,
            handler& commandGroupHandlerRef, TagT tag,
            const property_list& propList = {});

    /* -- common interface members -- */
    size_t size() const noexcept;

    /* Available only when: AccessMode == access_mode::read
    if Dimensions == 1, CoordT = int
    if Dimensions == 2, CoordT = int2
    if Dimensions == 3, CoordT = int4 */
    template <typename CoordT>
    DataT read(const CoordT& coords) const noexcept;

    /* Available only when: AccessMode == access_mode::write
    if Dimensions == 1, CoordT = int
    if Dimensions == 2, CoordT = int2
    if Dimensions == 3, CoordT = int4 */
    template <typename CoordT>
    void write(const CoordT& coords, const DataT& color) const;
};

template <typename DataT, int Dimensions = 1,
    access_mode AccessMode = 
        (std::is_const_v<DataT> ? access_mode::read 
            : access_mode::read_write)>
class host_unsampled_image_accessor {
    public:
        using value_type = // const DataT for read-only accessors, DataT otherwise
            __value_type__;
        using reference = value_type&;
        using const_reference = const DataT&;

        template <typename AllocatorT>
        host_unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef,
            const property_list& propList = {});

        template <typename AllocatorT, typename TagT>
```cpp
host_unsampled_image_accessor(
    unsampled_image<Dimensions, AllocatorT>& imageRef, TagT tag,
    const property_list& propList = {});

/* -- common interface members -- */

/* -- property interface members -- */

size_t size() const noexcept;

/* Available only when: (AccessMode == access_mode::read ||
    AccessMode == access_mode::read_write) */

if Dimensions == 1, CoordT = int
if Dimensions == 2, CoordT = int2
if Dimensions == 3, CoordT = int4 */

template<typename CoordT> DataT read(const CoordT& coords) const noexcept;

/* Available only when: (AccessMode == access_mode::write ||
    AccessMode == access_mode::read_write) */

if Dimensions == 1, CoordT = int
if Dimensions == 2, CoordT = int2
if Dimensions == 3, CoordT = int4 */

template<typename CoordT> void write(const CoordT& coords, const DataT& color) const;

};

} // namespace sycl
```

### Table 81. Member types of the unsampled image classes

<table>
<thead>
<tr>
<th>Member types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_type</td>
<td>If the accessor is read-only, equal to <code>const DataT</code>, otherwise equal to <code>DataT</code>. See Section 4.7.6.13.3 for which accessors are considered read-only.</td>
</tr>
<tr>
<td>reference</td>
<td>Equal to <code>value_type&amp;</code>.</td>
</tr>
<tr>
<td>const_reference</td>
<td>Equal to <code>const DataT&amp;</code>.</td>
</tr>
</tbody>
</table>

### Table 82. Constructors of the `unsampled_image_accessor` class
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <typename AllocatorT> unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef, handler& commandGroupHandlerRef, const property_list& propList = {})` | Constructs an `unsampled_image_accessor` for accessing an `unsampled_image` within a `command` on the queue associated with `commandGroupHandlerRef`. The optional `property_list` provides properties for the constructed object. If `AccessTarget` is `image_target::device`, throws an exception with the `errc::feature_not_supported` error code if the device associated with `commandGroupHandlerRef` does not have `aspect::image`.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <typename AllocatorT, typename TagT> unsampled_image_accessor(unsampled_image<Dimensions, AllocatorT>& imageRef, handler& commandGroupHandlerRef, TagT tag, const property_list& propList = {})` | Constructs an `unsampled_image_accessor` for accessing an `unsampled_image` within a `command` on the queue associated with `commandGroupHandlerRef`. The `tag` is used to deduce template arguments of the `unsampled_image_accessor` as described in Section 4.7.6.13.2. The optional `property_list` provides properties for the constructed object. If `AccessTarget` is `image_target::device`, throws an exception with the `errc::feature_not_supported` error code if the device associated with `commandGroupHandlerRef` does not have `aspect::image`.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename AllocatorT&gt; host_unsampled_image_accessor(unsampled_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef, const property_list&amp; propList = {})</code></td>
<td>Constructs a <code>host_unsampled_image_accessor</code> for accessing an <code>unsampled_image</code> immediately on the host. The optional <code>property_list</code> provides properties for the constructed object.</td>
</tr>
</tbody>
</table>

Table 83. Constructors of the `host_unsampled_image_accessor` class
Constructs a `host_unsampled_image_accessor` for accessing an `unsampled_image` immediately on the host. The `tag` is used to deduce template arguments of the `host_unsampled_image_accessor` as described in Section 4.7.6.13.2. The optional `property_list` provides properties for the constructed object.

Table 84. Member functions of the unsampled image classes

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t size() const noexcept</code></td>
<td>Returns the number of elements of the underlying <code>unsampled_image</code> that this accessor is accessing.</td>
</tr>
<tr>
<td><code>template &lt;typename CoordT&gt; DataT read(const CoordT&amp; coords) const</code></td>
<td>Available only when ((\text{AccessMode} == \text{access_mode::read} \lor \text{AccessMode} == \text{access_mode::read_write})). Reads and returns an element of the <code>unsampled_image</code> at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>int</code> when <code>Dimensions == 1</code>, <code>int2</code> when <code>Dimensions == 2</code> and <code>int4</code> when <code>Dimensions == 3</code>. For <code>unsampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td><code>template &lt;typename CoordT&gt; void write(const CoordT&amp; coords, const DataT&amp; color) const</code></td>
<td>Available only when ((\text{AccessMode} == \text{access_mode::write} \lor \text{AccessMode} == \text{access_mode::read_write})). Writes the value specified by <code>color</code> to the element of the image at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>int</code> when <code>Dimensions == 1</code>, <code>int2</code> when <code>Dimensions == 2</code> and <code>int4</code> when <code>Dimensions == 3</code>. For <code>unsampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
</tbody>
</table>

4.7.6.13.2. Deduction tags for unsampled image accessors

Some `unsampled_image_accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in Table 85 along with the access mode and accessor target that they imply.

Table 85. Enumeration of tags available for `unsampled_image_accessor` construction
Some `host_unsampled_image_accessor` constructors also take a `TagT` parameter. The permissible values for this parameter are listed in Table 86 along with the access mode that they imply.

<table>
<thead>
<tr>
<th>Tag value</th>
<th>Access mode</th>
<th>Accessor target</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_only</code></td>
<td><code>access_mode::read</code></td>
<td><code>image_target::device</code></td>
</tr>
<tr>
<td><code>write_only</code></td>
<td><code>access_mode::write</code></td>
<td><code>image_target::device</code></td>
</tr>
<tr>
<td><code>read_only_host_task</code></td>
<td><code>access_mode::read</code></td>
<td><code>image_target::host_task</code></td>
</tr>
<tr>
<td><code>write_only_host_task</code></td>
<td><code>access_mode::write</code></td>
<td><code>image_target::host_task</code></td>
</tr>
</tbody>
</table>

Table 86. Enumeration of tags available for `host_unsampled_image_accessor` construction

4.7.6.13.3. Read only unsampled image accessors and implicit conversions

All specializations of unsampled image accessors with `access_mode::read` are read-only regardless of whether `DataT` is `const` qualified. There is an implicit conversion between the `const` qualified and non-`const` qualified specializations, provided that all other template parameters are the same.

4.7.6.14. Sampled image accessors

There are two classes which implement accessors for sampled images, `sampled_image_accessor` and `host_sampled_image_accessor`. The former provides access from within a SYCL kernel function or from within a host task. The latter provides access from host code that is outside of a host task.

The dimensionality of a sampled image accessor must match the dimensionality of the underlying image to which it provides access. Sampled image accessors are always read-only.

The `AccessTarget` template parameter dictates how the `sampled_image_accessor` can be used: `image_target::device` means the accessor can be used in a SYCL kernel function while `image_target::host_task` means the accessor can be used in a host task. Programs which specify this template parameter as `image_target::device` and then use the `sampled_image_accessor` from a host task are ill formed. Likewise, programs which specify this template parameter as `image_target::host_task` and then use the `sampled_image_accessor` from a SYCL kernel function are ill formed.

4.7.6.14.1. Interface for sampled image accessors

A synopsis of the two sampled image accessor classes is provided below. Both classes have member types with the same name, which are described in Table 87. The constructors for the two classes are described in Table 88 and Table 89. Both classes also have member functions with the same name, which are described in Table 90.

The additional common special member functions and common member functions are listed in Section 4.5.2 in Table 7 and Table 8, respectively. For valid implicit conversions between sampled accessor types refer to Section 4.7.6.14.3.

Two `sampled_image_accessor` objects of the same type must be equality comparable in both the host code and in SYCL kernel functions. Two `host_sampled_image_accessor` objects of the same type must be equality comparable in the host code.

```cpp
namespace sycl {
enum class image_target : /* unspecified */ { device, host_task };
}
```
template <typename DataT, int Dimensions, 
    image_target AccessTarget = image_target::device>
class sampled_image_accessor {
public:
    using value_type = const DataT;
    using reference = const DataT&;
    using const_reference = const DataT&;

template <typename AllocatorT>
sampled_image_accessor(sampled_image<Dimensions, AllocatorT>& imageRef, 
    handler& commandGroupHandlerRef, 
    const property_list& proplist = {});

template <typename AllocatorT, typename TagT>
sampled_image_accessor(sampled_image<Dimensions, AllocatorT>& imageRef, 
    handler& commandGroupHandlerRef, TagT tag, 
    const property_list& proplist = {}); 

/* -- common interface members -- */

/* -- property interface members -- */

size_t size() const noexcept;

/* if Dimensions == 1, CoordT = float 
   if Dimensions == 2, CoordT = float2 
   if Dimensions == 3, CoordT = float4 */

template <typename CoordT> DataT read(const CoordT& coords) const noexcept;

} // namespace sycl

Table 87. Member types of the sampled image classes
### Member types

<table>
<thead>
<tr>
<th>Description</th>
<th>Value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to <code>const DataT</code>.</td>
<td><code>value_type</code></td>
</tr>
<tr>
<td>Equal to <code>const DataT&amp;</code>.</td>
<td><code>reference</code></td>
</tr>
<tr>
<td>Equal to <code>const DataT&amp;</code>.</td>
<td><code>const_reference</code></td>
</tr>
</tbody>
</table>

#### Table 88. Constructors of the `sampled_image_accessor` class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;typename AllocatorT&gt; sampled_image_accessor(sampled_image&lt;Dimensions, AllocatorT&gt;&amp; imageRef, handler&amp; commandGroupHandlerRef, const property_list&amp; proplist = { })</code></td>
<td>Constructs a <code>sampled_image_accessor</code> for accessing a <code>sampled_image</code> within a <code>command</code> on the queue associated with <code>commandGroupHandlerRef</code>. The optional <code>property_list</code> provides properties for the constructed object. If <code>AccessTarget</code> is <code>image_target::device</code>, throws an exception with the <code>errc::feature_not_supported</code> error code if the device associated with <code>commandGroupHandlerRef</code> does not have <code>aspect::image</code>.</td>
</tr>
</tbody>
</table>

| `template<typename AllocatorT, typename TagT> sampled_image_accessor(sampled_image<Dimensions, AllocatorT>& imageRef, handler& commandGroupHandlerRef, TagT tag, const property_list& proplist = { })` | Constructs a `sampled_image_accessor` for accessing a `sampled_image` within a `command` on the queue associated with `commandGroupHandlerRef`. The `tag` is used to deduce `template` arguments of the `sampled_image_accessor` as described in Section 4.7.6.14.2. The optional `property_list` provides properties for the constructed object. If `AccessTarget` is `image_target::device`, throws an exception with the `errc::feature_not_supported` error code if the device associated with `commandGroupHandlerRef` does not have `aspect::image`. |

### Table 89. Constructors of the `host_sampled_image_accessor` class
Constructor

```
template <typename AllocatorT>
host_sampled_image_accessor(sampled_image<Dimensions, AllocatorT>& imageRef, const property_list& propList = {})
```

Description

Constructs a `host_sampled_image_accessor` for accessing a `sampled_image` immediately on the host. The optional `property_list` provides properties for the constructed object.

**Table 90. Member functions of the sampled image classes**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t size() const noexcept</code></td>
<td>Returns the number of elements of the underlying <code>sampled_image</code> that this accessor is accessing.</td>
</tr>
<tr>
<td><code>template &lt;typename CoordT&gt; DataT read(const CoordT&amp; coords) const</code></td>
<td>Reads and returns a sampled element of the <code>sampled_image</code> at the coordinates specified by <code>coords</code>. Permitted types for <code>CoordT</code> are <code>float</code> when <code>Dimensions == 1</code>, <code>float2</code> when <code>Dimensions == 2</code> and <code>float4</code> when <code>Dimensions == 3</code>. For <code>sampled_image_accessor</code>, this function may only be called from within a <code>command</code>.</td>
</tr>
</tbody>
</table>

**4.7.6.14.2. Deduction tags for sampled image accessors**

Some `sampled_image_accessor` constructors take a `TagT` parameter, which is used to deduce template arguments. The permissible values for this parameter are listed in Table 91 along with the accessor target that they imply.

**Table 91. Enumeration of tags available for `sampled_image_accessor` construction**

<table>
<thead>
<tr>
<th>Tag value</th>
<th>Accessor target</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>read_only</code></td>
<td><code>image_target::device</code></td>
</tr>
<tr>
<td><code>read_only_host_task</code></td>
<td><code>image_target::host_task</code></td>
</tr>
</tbody>
</table>

**4.7.6.14.3. Read only sampled image accessors and implicit conversions**

All specializations of sampled image accessors are read-only regardless of whether `DataT` is `const` qualified. There is an implicit conversion between the `const` qualified and non-`const` qualified specializations, provided that all other template parameters are the same.

**4.7.7. Address space classes**

In SYCL, there are five different address spaces: global, local, constant, private and generic. In a SYCL generic implementation, types are not affected by the address spaces. However, there are situations where users need to explicitly carry address spaces in the type. For example:

- For performance tuning and genericness. Even if the platform supports the representation of the generic address space, this may come at some performance sacrifice. In order to help the target compiler, it can be useful to track specifically which address space a pointer is addressing.
- When linking SYCL kernels with SYCL backend-specific functions. In this case, it might be necessary to specify the address space for any pointer parameters.
Direct declaration of pointers with address spaces is discouraged as the definition is implementation-defined. Users must rely on the `multi_ptr` class to handle address space boundaries and interoperability.

### 4.7.7.1. Multi-pointer class

The multi-pointer class is the common interface for the explicit pointer classes, defined in Section 4.7.7.2.

There are situations where a user may want to make their type address space dependent. This allows performing generic programming that depends on the address space associated with their data. An example might be wrapping a pointer inside a class, where a user may need to template the class according to the address space of the pointer the class is initialized with. In this case, the `multi_ptr` class enables users to do this in a portable and stable way.

The `multi_ptr` class exposes 3 flavors of the same interface. If the value of `access::decorated` is `access::decorated::no`, the interface exposes pointers and references type that are not decorated by an address space. If the value of `access::decorated` is `access::decorated::yes`, the interface exposes pointers and references type that are decorated by an address space. The decoration is implementation dependent and relies on device compiler extensions. The decorated type may be distinct from the non-decorated one. For interoperability with the SYCL backend, users should rely on types exposed by the decorated version. If the value of `access::decorated` is `access::decorated::legacy`, the 1.2.1 interface is exposed. This interface is deprecated.

The template traits `remove_decoration` and type alias `remove_decoration_t` retrieve the non-decorated pointer or reference from a decorated one. Using this template trait with a non-decorated type is safe and returns the same type.

It is possible to use the `void` type for the `multi_ptr` class, but in that case some functionality is disabled. `multi_ptr<void>` does not provide the `reference` or `const_reference` types, the access operators (operator*(), operator->()), the arithmetic operators or `prefetch` member function. Conversions from `multi_ptr` to `multi_ptr<void>` of the same address space are allowed, and will occur implicitly. Conversions from `multi_ptr<void>` to any other `multi_ptr` type of the same address space are allowed, but must be explicit. The same rules apply to `multi_ptr<const void>`.

An overview of the interface provided for the `multi_ptr` class follows.

```cpp
namespace sycl {
namespace access {

enum class address_space : /* unspecified */ {
    global_space,
    local_space,
    constant_space, // Deprecated in SYCL 2020
    private_space,
    generic_space
};

decorated : /* unspecified */ { no, yes, legacy }
} // namespace access

template <typename T> struct remove_decoration {
    using type = /* ... */;
};

template <typename T> using remove_decoration_t = remove_decoration<T>::type;

template <typename ElementType, access::address_space Space,
```
class multi_ptr {
    static constexpr bool is_decorated = DecorateAddress == access::decorated::yes;
    static constexpr access::address_space address_space = Space;
    using value_type = ElementType;
    using pointer = std::conditional_t<is_decorated, __unspecified__*, std::add_pointer_t<value_type>>;
    using reference = std::conditional_t<is_decorated, __unspecified__&, std::add_lvalue_reference_t<value_type>>;
    using iterator_category = std::random_access_iterator_tag;
    using difference_type = std::ptrdiff_t;
    static_assert(std::is_same_v<remove_decoration_t<pointer>, std::add_pointer_t<value_type>>);
    static_assert(std::is_same_v<remove_decoration_t<reference>, std::add_lvalue_reference_t<value_type>>);
    // Legacy has a different interface.
    static_assert(DecorateAddress != access::decorated::legacy);

    // Constructors
    multi_ptr();
    multi_ptr(const multi_ptr&);
    multi_ptr(multi_ptr&&);
    explicit multi_ptr(
        typename multi_ptr<ElementType, Space, access::decorated::yes>::pointer);
    multi_ptr(std::nullptr_t);

    // Available only when:
    //   (Space == access::address_space::global_space ||
    //    Space == access::address_space::generic_space)
    template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
        multi_ptr(
            accessor<value_type, Dimensions, Mode, target::device, IsPlaceholder>>;

    // Available only when:
    //   (Space == access::address_space::local_space ||
    //    Space == access::address_space::generic_space)
    template <int Dimensions> multi_ptr(local_accessor<ElementType, Dimensions>>;

    // Deprecated
    // Available only when:
    //   (Space == access::address_space::local_space ||
    //    Space == access::address_space::generic_space)
    template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
        multi_ptr(
            accessor<value_type, Dimensions, Mode, target::local, IsPlaceholder>>;

    // Assignment and access operators
    multi_ptr8 operator=(const multi_ptr8);
    multi_ptr8 operator=(multi_ptr8);
    multi_ptr8 operator=(std::nullptr_t);

    // Available only when:
template<access::address_space AS, access::decorated IsDecorated>
multi_ptr& operator=(const multi_ptr<value_type, AS, IsDecorated>&);

template<access::address_space AS, access::decorated IsDecorated>
multi_ptr& operator=(multi_ptr<value_type, AS, IsDecorated>&&);

reference operator[](std::ptrdiff_t) const;
reference operator*() const;
pointer operator->() const;
std::add_pointer_t<value_type> get_raw() const;
__unspecified__* get_decorated() const;

// Conversion to the underlying pointer type
// Deprecated, get() should be used instead.
operator pointer() const;

// Cast to private_ptr
// Available only when: (Space == access::address_space::generic_space)
explicit operator multi_ptr<value_type, access::address_space::private_space, DecorateAddress>();

// Cast to global_ptr
// Available only when: (Space == access::address_space::generic_space)
explicit operator multi_ptr<value_type, access::address_space::global_space, DecorateAddress>();

// Cast to local_ptr
// Available only when: (Space == access::address_space::generic_space)
explicit operator multi_ptr<value_type, access::address_space::local_space, DecorateAddress>();

// Cast to local_ptr
// Available only when: (Space == access::address_space::generic_space)
explicit operator multi_ptr<const value_type, access::address_space::local_space, DecorateAddress>();

// Implicit conversion to a multi_ptr<void>.
// Available only when: (!std::is_const_v<value_type>)
template <access::decorated IsDecorated>
operator multi_ptr<void, Space, IsDecorated>() const;

// Implicit conversion to a multi_ptr<const void>.
// Available only when: (std::is_const_v<value_type>)
template <access::decorated IsDecorated>
operator multi_ptr<const void, Space, IsDecorated>() const;

// Implicit conversion to multi_ptr<const value_type, Space>.
operator multi_ptr<const value_type, Space, IsDecorated>() const;

// Implicit conversion to the non-decorated version of multi_ptr.
// Available only when: (is_decorated == true) operator multi_ptr<value_type, Space, access::decorated::no>() const;

// Implicit conversion to the decorated version of multi_ptr.
// Available only when: (is_decorated == false) operator multi_ptr<value_type, Space, access::decorated::yes>() const;

void prefetch(size_t numElements) const;

// Arithmetic operators
friend multi_ptr8 operator++(multi_ptr8 mp) { /* ... */
} friend multi_ptr operator++(multi_ptr mp, int) { /* ... */
} friend multi_ptr8 operator--(multi_ptr8 mp) { /* ... */
} friend multi_ptr operator--(multi_ptr mp, int) { /* ... */
} friend multi_ptr8 operator+=(multi_ptr8 lhs, difference_type r) { /* ... */
} friend multi_ptr8 operator-=(multi_ptr8 lhs, difference_type r) { /* ... */
} friend multi_ptr operator+=(const multi_ptr8 lhs, difference_type r) { /* ... */
} friend multi_ptr operator-=(const multi_ptr8 lhs, difference_type r) { /* ... */
} friend reference operator*(const multi_ptr8 lhs) { /* ... */
} friend bool operator==(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
} friend bool operator!=(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
} friend bool operator<(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
} friend bool operator>(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
} friend bool operator<=(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
} friend bool operator>=(const multi_ptr8 lhs, const multi_ptr8 rhs) { /* ... */
}
4.7.7.1. Multi-pointer class

friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }

friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }

// Specialization of multi_ptr for void and const void
template <access::address_space Space, access::decorated DecorateAddress>
class multi_ptr<VoidType, Space, DecorateAddress> {
public:
static constexpr bool is_decorated =
  DecorateAddress == access::decorated::yes;
static constexpr access::address_space address_space = Space;
using value_type = VoidType;
using pointer = std::conditional_t<is_decorated, __unspecified__*,
  std::add_pointer_t<value_type>>;
using difference_type = std::ptrdiff_t;

static_assert(std::is_same_v<remove_decoration_t<pointer>,
  std::add_pointer_t<value_type>>);
// Legacy has a different interface.
static_assert(DecorateAddress != access::decorated::legacy);

// Constructors
multi_ptr();
multi_ptr(const multi_ptr&);
multi_ptr(multi_ptr&&);
explicit multi_ptr(
  typename multi_ptr<VoidType, Space, access::decorated::yes>::pointer);
multi_ptr(std::nullptr_t);

// Available only when:
// (Space == access::address_space::global_space)
template<typename ElementType, int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
  accessor<ElementType, Dimensions, Mode, target::device, IsPlaceholder>);

// Available only when:
// (Space == access::address_space::local_space)
template<typename ElementType, int Dimensions>
  multi_ptr(local_accessor<ElementType, Dimensions>);

// Deprecated
// Available only when:
// (Space == access::address_space::local_space)
template<typename ElementType, int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
  multi_ptr(accessor<ElementType, Dimensions, Mode, target::local, IsPlaceholder>);

// Assignment operators
multi_ptr& operator=(const multi_ptr&);  
multi_ptr& operator=(multi_ptr&&);  
multi_ptr& operator=(std::nullptr_t);

pointer get() const;

// Conversion to the underlying pointer type
explicit operator pointer() const;

// Explicit conversion to a multi_ptr<ElementType>
// Available only when: (std::is_const_v<ElementType> || !std::is_const_v<VoidType>)
template<typename ElementType>
  explicit operator multi_ptr<ElementType, Space, DecorateAddress>() const;

// Implicit conversion to the non-decorated version of multi_ptr.
// Available only when: (is_decorated == true)
operator multi_ptr<value_type, Space, access::decorated::no>() const;

// Implicit conversion to the decorated version of multi_ptr.
// Available only when: (is_decorated == false)
operator multi_ptr<value_type, Space, access::decorated::yes>() const;

// Implicit conversion to multi_ptr<const void, Space>
operator multi_ptr<const void, Space, DecorateAddress>() const;

friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }
friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */ }


Table 92. Constructors of the SYCL multi_ptr class template

```cpp
friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }
friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */ }

friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }
friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */ }

// Deprecated, address_space_cast should be used instead.
template <typename ElementType, access::address_space Space,
          access::decorated DecorateAddress>
multi_ptr<ElementType, Space, DecorateAddress> make_ptr(ElementType*);

template <access::address_space Space, access::decorated DecorateAddress,
          typename ElementType>
multi_ptr<ElementType, Space, DecorateAddress> address_space_cast(ElementType*);

// Deduction guides
template <Dimensions, access_mode Mode, access::placeholder IsPlaceholder,
          class T>
multi_ptr(accessor<T, Dimensions, Mode, target::device, IsPlaceholder>)
  -> multi_ptr<T, access::address_space::global_space, access::decorated::no>;

template <Dimensions, access_mode Mode, access::placeholder IsPlaceholder,
          class T>
multi_ptr(local_accessor<T, Dimensions>)
  -> multi_ptr<T, access::address_space::local_space, access::decorated::no>;

} // namespace sycl
```
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multi_ptr()</code></td>
<td>Default constructor.</td>
</tr>
<tr>
<td><code>multi_ptr(const multi_ptr&amp;)</code></td>
<td>Copy constructor.</td>
</tr>
<tr>
<td><code>multi_ptr(multi_ptr&amp;&amp;)</code></td>
<td>Move constructor.</td>
</tr>
<tr>
<td><code>explicit multi_ptr(multi_ptr&lt;ElementType, Space, yes&gt;::pointer)</code></td>
<td>Constructor that takes as an argument a decorated pointer.</td>
</tr>
<tr>
<td><code>multi_ptr(nullptr_t)</code></td>
<td>Constructor from a nullptr.</td>
</tr>
<tr>
<td><code>template &lt;int Dimensions, access_mode Mode, access::placeholder IsPlaceholder&gt; multi_ptr(accessor&lt;ElementType, Dimensions, Mode, target::device, IsPlaceholder&gt;);</code></td>
<td>Constructs a <code>multi_ptr</code> from an accessor of <code>target::device</code>. This constructor may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td><code>template &lt;int Dimensions&gt; multi_ptr(local_accessor&lt;ElementType, Dimensions&gt;)</code></td>
<td>Constructs a <code>multi_ptr</code> from a <code>local_accessor</code>. This constructor may only be called from within a <code>command</code>.</td>
</tr>
<tr>
<td><code>template &lt;int Dimensions, access_mode Mode, access::placeholder IsPlaceholder&gt; multi_ptr(accessor&lt;ElementType, Dimensions, Mode, target::local, IsPlaceholder&gt;);</code></td>
<td>Deprecated in SYCL 2020. Use the overload with <code>local_accessor</code> instead. Constructs a <code>multi_ptr</code> from an accessor of <code>target::local</code>. This constructor may only be called from within a <code>command</code>.</td>
</tr>
</tbody>
</table>
Constructor | Description
--- | ---
\[
\text{template<typename ElementType,}
\text{\hspace{0.5em}access::address_space Space,}
\text{\hspace{0.5em}access::decorated DecorateAddress>}
\text{multi_ptr<ElementType, Space, DecorateAddress>}
\text{make_ptr(ElementType* pointer)}
\]


\[
\text{template<access::address_space Space,}
\text{access::decorated DecorateAddress,}
\text{typename ElementType>}
\text{multi_ptr<ElementType, Space, DecorateAddress>}
\text{address_space_cast(ElementType* pointer)}
\]

Table 93. Operators of \text{multi_ptr} class

<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{multi_ptr&amp; operator=(const multi_ptr&amp;)}</td>
<td>Copy assignment operator.</td>
</tr>
<tr>
<td>\text{multi_ptr&amp; operator=(multi_ptr&amp;&amp;)}</td>
<td>Move assignment operator.</td>
</tr>
<tr>
<td>\text{multi_ptr&amp; operator=(std::nullptr_t)}</td>
<td>Assigns \text{nullptr} to the \text{multi_ptr}.</td>
</tr>
</tbody>
</table>
| \text{\text{template<access::address_space AS,}}
\text{access::decorated IsDecorated>}
\text{multi_ptr& operator=(const multi_ptr<value_type, AS, IsDecorated>&)} | Available only when: \(\text{Space == access::address_space::generic_space \&\& AS != access::address_space::constant_space}\). Assigns the value of the left hand side \text{multi_ptr} into the \text{generic_ptr}. |
| \text{\text{template<access::address_space AS,}}
\text{access::decorated IsDecorated>}
\text{multi_ptr& operator=(multi_ptr<value_type, AS, IsDecorated>&&)} | Available only when: \(\text{Space == access::address_space::generic_space \&\& AS != access::address_space::constant_space}\). Move the value of the left hand side \text{multi_ptr} into the \text{generic_ptr}. |
<table>
<thead>
<tr>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference operator[](std::ptrdiff_t i) const</td>
<td>Available only when: (!std::is_void_v&lt;value_type&gt;). Returns a reference to the i-th pointed value. The value i can be negative.</td>
</tr>
<tr>
<td>pointer operator-&gt;() const</td>
<td>Available only when: (!std::is_void_v&lt;value_type&gt;). Returns the underlying pointer.</td>
</tr>
<tr>
<td>reference operator*() const</td>
<td>Available only when: (!std::is_void_v&lt;value_type&gt;). Returns a reference to the pointed value.</td>
</tr>
<tr>
<td>operator pointer() const</td>
<td>Implicit conversion to the underlying pointer type. <strong>Deprecated:</strong> The member function <code>get</code> should be used instead</td>
</tr>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type, access::address_space::private_space, IsDecorated&gt;() const</td>
<td>Available only when: (<code>Space</code> == access::address_space::generic_space). Conversion from <code>generic_ptr</code> to <code>private_ptr</code>. The result is undefined if the pointer does not address the private address space.</td>
</tr>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;const value_type, access::address_space::private_space, IsDecorated&gt;() const</td>
<td>Available only when: (<code>Space</code> == access::address_space::generic_space). Conversion from <code>generic_ptr</code> to <code>private_ptr</code>. The result is undefined if the pointer does not address the private address space.</td>
</tr>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type, access::address_space::global_space, IsDecorated&gt;() const</td>
<td>Available only when: (<code>Space</code> == access::address_space::generic_space). Conversion from <code>generic_ptr</code> to <code>global_ptr</code>. The result is undefined if the pointer does not address the global address space.</td>
</tr>
<tr>
<td>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;const value_type, access::address_space::global_space, IsDecorated&gt;() const</td>
<td>Available only when: (<code>Space</code> == access::address_space::generic_space). Conversion from <code>generic_ptr</code> to <code>global_ptr</code>. The result is undefined if the pointer does not address the global address space.</td>
</tr>
</tbody>
</table>
### Operators

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type, access::address_space::local_space, IsDecorated&gt;() const</code></td>
<td>Available only when: <code>(Space == access::address_space::generic_space)</code>. Conversion from <code>generic_ptr</code> to <code>local_ptr</code>. The result is undefined if the pointer does not address the local address space.</td>
</tr>
<tr>
<td><code>template &lt;access::decorated IsDecorated&gt; explicit operator multi_ptr&lt;value_type , access::address_space::local_space, IsDecorated&gt;() const</code></td>
<td>Available only when: <code>(Space == access::address_space::generic_space)</code>. Conversion from <code>generic_ptr</code> to <code>local_ptr</code>. The result is undefined if the pointer does not address the local address space.</td>
</tr>
<tr>
<td><code>template &lt;access::decorated IsDecorated&gt; operator multi_ptr&lt;void, Space, IsDecorated&gt;() const</code></td>
<td>Available only when: <code>(is_decorated == true)</code>. Implicit conversion to the equivalent <code>multi_ptr</code> object that does not expose decorated pointers or references.</td>
</tr>
<tr>
<td><code>template &lt;access::decorated IsDecorated&gt; operator multi_ptr&lt;const value_type, Space, IsDecorated&gt;() const</code></td>
<td>Available only when: <code>(is_decorated == false)</code>. Implicit conversion to the equivalent <code>multi_ptr</code> object that exposes decorated pointers and references.</td>
</tr>
</tbody>
</table>

### Table 94. Member functions of `multi_ptr` class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pointer get() const</code></td>
<td>Returns the underlying pointer. Whether the pointer is decorated depends on the value of <code>DecorateAddress</code>.</td>
</tr>
</tbody>
</table>
**Member function**

<table>
<thead>
<tr>
<th><strong>unspecified</strong> * get_decorated() const</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Returns the underlying pointer</td>
</tr>
<tr>
<td>decorated by the address space</td>
</tr>
<tr>
<td>that it addresses. Note that the</td>
</tr>
<tr>
<td>support involves implementation-</td>
</tr>
<tr>
<td>defined device compiler extensions.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>std::add_pointer_t&lt;value_type&gt; get_raw() const</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Returns the underlying pointer,</td>
</tr>
<tr>
<td>always undecorated.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>void prefetch(size_t numElements) const</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Available only when: Space ==</td>
</tr>
<tr>
<td>access::address_space::global_space.</td>
</tr>
<tr>
<td>Prefetches a number of elements</td>
</tr>
<tr>
<td>specified by numElements into the</td>
</tr>
<tr>
<td>global memory cache. This operation is</td>
</tr>
<tr>
<td>an implementation-defined optimization</td>
</tr>
<tr>
<td>and does not effect the functional</td>
</tr>
<tr>
<td>behavior of the SYCL kernel function.</td>
</tr>
</tbody>
</table>

**Table 95. Hidden friend functions of the multi_ptr class**

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference operator*(const multi_ptr&amp; mp)</td>
<td>Available only when: (!std::is_void_v&lt;ElementType&gt;). Operator that returns a reference to the value_type of mp.</td>
</tr>
<tr>
<td>multi_ptr&amp; operator++(multi_ptr&amp; mp)</td>
<td>Available only when: (!std::is_void_v&lt;ElementType&gt;). Increments mp by 1 and returns mp.</td>
</tr>
<tr>
<td>multi_ptr operator++(multi_ptr&amp; mp, int)</td>
<td>Available only when: (!std::is_void_v&lt;ElementType&gt;). Increments mp by 1 and returns a new multi_ptr with the value of the original mp.</td>
</tr>
<tr>
<td>multi_ptr&amp; operator--(multi_ptr&amp; mp)</td>
<td>Available only when: (!std::is_void_v&lt;ElementType&gt;). Decrements mp by 1 and returns mp.</td>
</tr>
<tr>
<td>multi_ptr operator--(multi_ptr&amp; mp, int)</td>
<td>Available only when: (!std::is_void_v&lt;ElementType&gt;). Decrements mp by 1 and returns a new multi_ptr with the value of the original mp.</td>
</tr>
</tbody>
</table>
### Hidden friend function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multi_ptr&amp; operator+=(multi_ptr&amp; lhs, difference_type r)</code></td>
<td>Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code>. Moves <code>mp</code> forward by <code>r</code> and returns <code>lhs</code>.</td>
</tr>
<tr>
<td><code>multi_ptr&amp; operator-=(multi_ptr&amp; lhs, difference_type r)</code></td>
<td>Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code>. Moves <code>mp</code> backward by <code>r</code> and returns <code>lhs</code>.</td>
</tr>
<tr>
<td><code>multi_ptr operator+(const multi_ptr&amp; lhs, difference_type r)</code></td>
<td>Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code>. Creates a new <code>multi_ptr</code> that points <code>r</code> forward compared to <code>lhs</code>.</td>
</tr>
<tr>
<td><code>multi_ptr operator-(const multi_ptr&amp; lhs, difference_type r)</code></td>
<td>Available only when: <code>(!std::is_void_v&lt;ElementType&gt;)</code>. Creates a new <code>multi_ptr</code> that points <code>r</code> backward compared to <code>lhs</code>.</td>
</tr>
<tr>
<td><code>bool operator==(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>==</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator!=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>!=</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator&lt;(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator&gt;(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator&lt;=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&lt;=</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator&gt;=(const multi_ptr&amp; lhs, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&gt;=</code> for <code>multi_ptr</code> class.</td>
</tr>
<tr>
<td><code>bool operator==(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>==</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator!=(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>!=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&lt;(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&gt;(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&lt;=(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>&lt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&gt;=(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator <code>&gt;=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
</tbody>
</table>
### Hidden Friend Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool operator&gt;=(const multi_ptr&amp; lhs, std::nullptr_t)</code></td>
<td>Comparison operator (\geq) for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator==(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>==</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator!=(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>!=</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&lt;(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&lt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&gt;(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>&gt;</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&lt;=(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>\leq\</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
<tr>
<td><code>bool operator&gt;=(std::nullptr_t, const multi_ptr&amp; rhs)</code></td>
<td>Comparison operator <code>\geq\</code> for <code>multi_ptr</code> class with a <code>std::nullptr_t</code>.</td>
</tr>
</tbody>
</table>

The following is the overview of the legacy interface from 1.2.1 provided for the `multi_ptr` class.

This legacy class supports the deprecated `address_space::constant_space` address space, which can be used to represent a pointer to `constant memory`. Pointers to `constant memory` have an implementation-defined address space, and each implementation can define whether it is legal to assign such a pointer to a generic address pointer.

```cpp
namespace sycl {

// Legacy interface, inherited from 1.2.1.
// Deprecated.
template <typename ElementType, access::address_space Space>
class [[deprecated]] multi_ptr<ElementType, Space, access::decorated::legacy> {
    public:
    using element_type = ElementType;
    using difference_type = std::ptrdiff_t;

    // Implementation defined pointer and reference types that correspond to
    // SYCL/OpenCL interoperability types for OpenCL C functions.
    using pointer_t =
        multi_ptr<ElementType, Space, access::decorated::yes>::::pointer;
    using const_pointer_t =
        multi_ptr<const ElementType, Space, access::decorated::yes>::::pointer;
    using reference_t =
        multi_ptr<ElementType, Space, access::decorated::yes>::::reference;
    using const_reference_t =
        multi_ptr<const ElementType, Space, access::decorated::yes>::::reference;

    static constexpr access::address_space address_space = Space;

    // Constructors
    multi_ptr();
    multi_ptr(const multi_ptr&); 
}
```
multi_ptr(multi_ptr&&)
multi_ptr(pointer_t);
multi_ptr(ElementType*);
multi_ptr(std::nullptr_t);
~multi_ptr();

// Assignment and access operators
multi_ptr& operator=(const multi_ptr&);
multi_ptr& operator=(multi_ptr&&);
multi_ptr& operator=(pointer_t);
multi_ptr& operator=(ElementType*);
multi_ptr& operator=(std::nullptr_t);
friend ElementType& operator*(const multi_ptr& mp) { /* ... */
}
ElementType* operator->() const;

// Only if Space == global_space
template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
multi_ptr(
    accessor<ElementType, Dimensions, Mode, target::device, IsPlaceholder>);

// Only if Space == local_space
template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
multi_ptr(
    accessor<ElementType, Dimensions, Mode, target::local, IsPlaceholder>);

// Only if Space == constant_space
template <int Dimensions, access_mode Mode, access::placeholder IsPlaceholder>
multi_ptr(accessor<ElementType, Dimensions, Mode, target::constant_buffer, IsPlaceholder>);

// Returns the underlying OpenCL C pointer
pointer_t get() const;

// Implicit conversion to the underlying pointer type
operator ElementType*() const;

// Implicit conversion to a multi_ptr<void>
// Only available when ElementType is not const-qualified
operator multi_ptr<void, Space, access::decorated::legacy>() const;

// Implicit conversion to a multi_ptr<const void>
// Only available when ElementType is const-qualified
operator multi_ptr<const void, Space, access::decorated::legacy>() const;

// Implicit conversion to multi_ptr<const ElementType, Space>
operator multi_ptr<const ElementType, Space, access::decorated::legacy>() const;

// Arithmetic operators
friend multi_ptr& operator++(multi_ptr& mp) { /* ... */
} friend multi_ptr operator++(multi_ptr mp, int) { /* ... */
} friend multi_ptr& operator--(multi_ptr& mp) { /* ... */
}
friend multi_ptr operator-(multi_ptr mp, int) { /* ... */
friend multi_ptr operator+=(multi_ptr lhs, difference_type r) { /* ... */
friend multi_ptr operator-=(multi_ptr lhs, difference_type r) { /* ... */
friend multi_ptr operator+(const multi_ptr lhs, difference_type r) { /* ... */
friend multi_ptr operator-=(const multi_ptr lhs, difference_type r) { /* ... */

void prefetch(size_t numElements) const;
friend bool operator==(const multi_ptr lhs, const multi_ptr rhs) { /* ... */
friend bool operator!=(const multi_ptr lhs, const multi_ptr rhs) { /* ... */
friend bool operator<(const multi_ptr lhs, const multi_ptr rhs) { /* ... */
friend bool operator>(const multi_ptr lhs, const multi_ptr rhs) { /* ... */
friend bool operator<=(const multi_ptr lhs, const multi_ptr rhs) { /* ... */
friend bool operator>=(const multi_ptr lhs, const multi_ptr rhs) { /* ... */

friend bool operator==(const multi_ptr lhs, std::nullptr_t) { /* ... */
friend bool operator!=(const multi_ptr lhs, std::nullptr_t) { /* ... */
friend bool operator<(const multi_ptr lhs, std::nullptr_t) { /* ... */
friend bool operator>(const multi_ptr lhs, std::nullptr_t) { /* ... */
friend bool operator<=(const multi_ptr lhs, std::nullptr_t) { /* ... */
friend bool operator>=(const multi_ptr lhs, std::nullptr_t) { /* ... */

friend bool operator==(std::nullptr_t, const multi_ptr rhs) { /* ... */
friend bool operator!=(std::nullptr_t, const multi_ptr rhs) { /* ... */
friend bool operator<(std::nullptr_t, const multi_ptr rhs) { /* ... */
friend bool operator>(std::nullptr_t, const multi_ptr rhs) { /* ... */
friend bool operator<=(std::nullptr_t, const multi_ptr rhs) { /* ... */
friend bool operator>=(std::nullptr_t, const multi_ptr rhs) { /* ... */

// Legacy interface, inherited from 1.2.1.
// Deprecated.
// Specialization of multi_ptr for void and const void
// VoidType can be either void or const void

template <access::address_space Space>
class [[deprecated]] multi_ptr<VoidType, Space, access::decorated::legacy> {
public:
  using element_type = VoidType;
  using difference_type = std::ptrdiff_t;

  // Implementation defined pointer types that correspond to
  // SYCL/OpenCL interoperability types for OpenCL C functions
  using pointer_t = multi_ptr<VoidType, Space, access::decorated::yes>::pointer;
  using const_pointer_t = multi_ptr<
      const VoidType, Space, access::decorated::yes>::pointer;

  static constexpr access::address_space address_space = Space;

  // Constructors
  multi_ptr();
  multi_ptr(const multi_ptr&);
  multi_ptr(multi_ptr&&);
  multi_ptr(pointer_t);
  multi_ptr(VoidType*);
  multi_ptr(std::nullptr_t);
  ~multi_ptr();

  // Assignment operators
  multi_ptr& operator=(const multi_ptr&);
  multi_ptr& operator=(multi_ptr&&);
  multi_ptr& operator=(pointer_t);
  multi_ptr& operator=(VoidType*);
  multi_ptr& operator=(std::nullptr_t);

  // Only if Space == global_space
  template <typename ElementType, int Dimensions, access_mode Mode>
  multi_ptr<accessor<ElementType, Dimensions, Mode, target::device>>;

  // Only if Space == local_space
  template <typename ElementType, int Dimensions, access_mode Mode>
  multi_ptr<accessor<ElementType, Dimensions, Mode, target::local>>;

  // Only if Space == constant_space
  template <typename ElementType, int Dimensions, access_mode Mode>
  multi_ptr<accessor<ElementType, Dimensions, Mode, target::constant_buffer>>;

  // Returns the underlying OpenCL C pointer
  pointer_t get() const;

  // Implicit conversion to the underlying pointer type
  operator VoidType*() const;

  // Explicit conversion to a multi_ptr<ElementType>
  // If VoidType is const, ElementType must be as well
  template <typename ElementType>
  explicit
  operator multi_ptr<
      ElementType, Space, access::decorated::legacy>() const;
}
4.7.7.2. Explicit pointer aliases

SYCL provides aliases to the `multi_ptr` class template (see Section 4.7.7.1) for each specialization of `access::address_space`.

A synopsis of the SYCL `multi_ptr` class template aliases is provided below.

```cpp
namespace sycl {

friend bool operator==(const multi_ptr& lhs, const multi_ptr& rhs) const; // Implicit conversion to multi_ptr<const void, Space>
operator multi_ptr<const void, Space, access::decorated::legacy>() const;

friend bool operator!=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
}
friend bool operator<(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
}
friend bool operator>(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
}
friend bool operator<=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
}
friend bool operator>=(const multi_ptr& lhs, const multi_ptr& rhs) { /* ... */
}
friend bool operator==(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator!=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator<(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator>(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator<=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator>=(const multi_ptr& lhs, std::nullptr_t) { /* ... */
}
friend bool operator==(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}
friend bool operator!=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}
friend bool operator<(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}
friend bool operator>(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}
friend bool operator<=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}
friend bool operator>=(std::nullptr_t, const multi_ptr& rhs) { /* ... */
}

}; // namespace sycl
```
template <typename ElementType, access::address_space Space, 
        access::decorated IsDecorated>
class multi_ptr;

// Template specialization aliases for different pointer address spaces

template <typename ElementType, 
        access::decorated IsDecorated = access::decorated::legacy>
using global_ptr = multi_ptr<ElementType, access::address_space::global_space, IsDecorated>;

template <typename ElementType, 
        access::decorated IsDecorated = access::decorated::legacy>
using local_ptr = multi_ptr<ElementType, access::address_space::local_space, IsDecorated>;

// Deprecated in SYCL 2020

template <typename ElementType, 
        access::decorated IsDecorated = access::decorated::legacy>
using constant_ptr = multi_ptr<
        ElementType, access::address_space::constant_space, 
        access::decorated::legacy>;

template <typename ElementType, 
        access::decorated IsDecorated = access::decorated::legacy>
using private_ptr = multi_ptr<
        ElementType, access::address_space::private_space, IsDecorated>;

// Template specialization aliases for different pointer address spaces. 
// The interface exposes non-decorated pointer while keeping the 
// address space information internally.

template <typename ElementType>
using raw_global_ptr = multi_ptr<
        ElementType, access::address_space::global_space, 
        access::decorated::no>;

template <typename ElementType>
using raw_local_ptr = multi_ptr<
        ElementType, access::address_space::local_space, 
        access::decorated::no>;

// Template specialization aliases for different pointer address spaces. 
// The interface exposes decorated pointer.

template <typename ElementType>
using decorated_global_ptr = multi_ptr<
        ElementType, access::address_space::global_space, 
        access::decorated::yes>;

template <typename ElementType>
using decorated_local_ptr = multi_ptr<
        ElementType, access::address_space::local_space,
template<typename ElementType>
using decorated_private_ptr = multi_ptr<ElementType, access::address_space::private_space, access::decorated::yes>;

} // namespace sycl

Note that using global_ptr, local_ptr, constant_ptr or private_ptr without specifying the decoration is deprecated. The default argument is provided for compatibility with 1.2.1.

### 4.7.8. Image samplers

The SYCL image_sampler struct contains a configuration for sampling a sampled_image. The members of this struct are defined by the following tables.

```cpp
namespace sycl {
  enum class addressing_mode : /* unspecified */ {
    mirrored_repeat,
    repeat,
    clamp_to_edge,
    clamp,
    none
  };

  enum class filtering_mode : /* unspecified */ { nearest, linear };

  enum class coordinate_normalization_mode : /* unspecified */ {
    normalized,
    unnormalized
  };

  struct image_sampler {
    addressing_mode addressing;
    coordinate_mode coordinate;
    filtering_mode filtering;
  };
} // namespace sycl
```

<table>
<thead>
<tr>
<th>addressing_mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mirrored_repeat</td>
<td>Out of range coordinates will be flipped at every integer junction.</td>
</tr>
<tr>
<td></td>
<td>This addressing mode can only be used with normalized coordinates.</td>
</tr>
<tr>
<td></td>
<td>If normalized coordinates are not used, this addressing mode may generate</td>
</tr>
<tr>
<td></td>
<td>image coordinates that are undefined.</td>
</tr>
</tbody>
</table>
4.8. Unified shared memory (USM)

This section describes properties and routines for pointer-based memory management interfaces in SYCL. These routines augment, rather than replace, the buffer-based interfaces in SYCL.

Unified Shared Memory (USM) provides a pointer-based alternative to the buffer programming model. USM enables:

- Easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations.
- Fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience.
- A simpler programming model, by automatically migrating some allocations between SYCL devices
and the host.

To show the differences with the example from Section 3.2, the following source code example shows how shared memory can be used between host and device:

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

int main() {
    // Create a default queue to enqueue work to the default device
    queue myQueue;

    // Allocate shared memory bound to the device and context associated to the
    // queue Replacing malloc_shared with malloc_host would yield a correct
    // program that allocated device-visible memory on the host.
    int* data = sycl::malloc_shared<int>(1024, myQueue);

    myQueue.parallel_for(1024, [=](id<1> idx) {
        // Initialize each buffer element with its own rank number starting at 0
        data[idx] = idx;
    }); // End of the kernel function

    // Explicitly wait for kernel execution since there is no accessor involved
    myQueue.wait();

    // Print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data[" << i << "] = " << data[i] << std::endl;

    return 0;
}
```

By comparison, the following source code example uses less capable device memory, which requires an explicit copy between the device and the host:

```cpp
#include <iostream>
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

int main() {
    // Create a default queue to enqueue work to the default device
    queue myQueue;

    // Allocate shared memory bound to the device and context associated to the
    // queue
    int* data = sycl::malloc_device<int>(1024, myQueue);

    myQueue.parallel_for(1024, [=](id<1> idx) {
        // Initialize each buffer element with its own rank number starting at 0
        data[idx] = idx;
    }); // End of the kernel function

    // Explicitly wait for kernel execution since there is no accessor involved
    myQueue.wait();
```
4.8.1. Unified addressing

Unified Addressing guarantees that all devices will use a unified address space. Pointer values in the unified address space will always refer to the same location in memory. The unified address space encompasses the host and one or more devices. Note that this does not require addresses in the unified address space to be accessible on all devices, just that pointer values will be consistent.

4.8.2. Kinds of unified shared memory

USM is a capability that, when available, provides the ability to create allocations that are visible to both host and device(s). USM builds upon Unified Addressing to define a shared address space where pointer values in this space always refer to the same location in memory. USM defines three types of memory allocations described in Table 99.

**Table 99. Type of USM allocations**

<table>
<thead>
<tr>
<th>USM allocation type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>Allocations in host memory that are accessible by a device</td>
</tr>
<tr>
<td>device</td>
<td>Allocations in device memory that are not accessible by the host</td>
</tr>
<tr>
<td>shared</td>
<td>Allocations in shared memory that are accessible by both host and device</td>
</tr>
</tbody>
</table>

The following enum is used to refer to the different types of allocations inside of a SYCL program:

```cpp
namespace sycl {
    namespace usm {
        enum class alloc : /* unspecified */ {
            host,
            device,
            shared,
            unknown
        };
    }
}
```

USM is an optional feature which may not be supported by all devices, and devices that support USM may not support all types of USM allocation. A SYCL application can use the `device::has()` function to
determine the level of USM support for a device. See Table 26 in Section 4.6.4.3 for more details.

The characteristics of USM allocations are summarized in Table 100.

### Table 100. Characteristics of the different kinds of USM allocation

<table>
<thead>
<tr>
<th>Allocation Type</th>
<th>Initial Location</th>
<th>Accessible By</th>
<th>Migratable To</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>device</td>
<td>host</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>device</td>
</tr>
<tr>
<td>device</td>
<td>host</td>
<td>Yes</td>
<td>device</td>
</tr>
<tr>
<td>host</td>
<td>host</td>
<td>host</td>
<td>N/A</td>
</tr>
<tr>
<td>Any device</td>
<td>host</td>
<td>Yes</td>
<td>host</td>
</tr>
<tr>
<td>shared</td>
<td>Unspecified</td>
<td>host</td>
<td>host</td>
</tr>
<tr>
<td></td>
<td></td>
<td>device</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Another device</td>
<td>Optional (P2P)</td>
</tr>
</tbody>
</table>

Each USM allocation has an associated SYCL context, and any access to that memory must use the same context. Specifically, any SYCL kernel function that dereferences a pointer to a USM allocation must be submitted to a queue that was constructed with the same context that was used to allocate that memory. The explicit memory operation commands that take USM pointers have a similar restriction. (See Section 4.9.4.3 for details.) Violations of these requirements result in undefined behavior.

There are no similar restrictions for dereferencing a USM pointer in a host task. This is legal regardless of which queue the host task was submitted to so long as the USM pointer is accessible on the host.

Each type of USM allocation has different rules for where that memory is accessible. Attempting to dereference a USM pointer on the host or on a device in violation of these rules results in undefined behavior. Passing a USM pointer to one of the explicit memory functions where the pointer is not accessible to the device generally results in undefined behavior. See Section 4.9.4.3 for the exact rules.

Device allocations are used for explicitly managing device memory. Programmers directly allocate device memory and explicitly copy data between host memory and a device allocation. Device allocations are obtained through SYCL device USM allocation routines instead of system allocation routines like `std::malloc` or C++ `new`. Device allocations are not accessible on the host, but the pointer values remain consistent on account of Unified Addressing. The size of device allocations will be limited by the amount of memory in a device. Support for device allocations on a specific device can be queried through `aspect::usm_device_allocations`.

Device allocations must be explicitly copied between the host and a device. The member functions to copy and initialize data are found in Table 28 and Table 135, and these functions may be used on device allocations if a device supports `aspect::usm_device_allocations`.

Host allocations allow devices to directly read and write host memory inside of a kernel. This can be useful for several reasons, such as when the overhead of moving a small amount of data is not worth paying over the cost of a remote access or when the size of a data set exceeds the size of a device’s memory. Host allocations must also be obtained using SYCL routines instead of system allocation routines. While a device may remotely read and write a host allocation, the allocation does not migrate to the device - it remains in host memory. Users should take care to properly synchronize access to host allocations between host execution and kernels. The total size of host allocations will be limited by the amount of pinnable-memory on the host on most systems. Support for host allocations on a specific device can be queried through `aspect::usm_host_allocations`. Support for atomic modification of host allocations on a specific device can be queried through `aspect::usm_atomic_host_allocations`. 
Shared allocations implicitly share data between the host and devices. Data may move to where it is being used without the programmer explicitly informing the runtime. It is up to the runtime and backends to make sure that a shared allocation is available where it is used. Shared allocations must also be obtained using SYCL allocation routines instead of the system allocator. The maximum size of a shared allocation on a specific device, and the total size of all shared allocations in a context, are implementation-defined. Support for shared allocations on a specific device can be queried through `aspect::usm_shared_allocations`.

Not all devices may support concurrent access of a shared allocation with the host. If a device does not support this, host execution and device code must take turns accessing the allocation, so the host must not access a shared allocation while a kernel is executing. Host access to a shared allocation which is also accessed by an executing kernel on a device that does not support concurrent access results in undefined behavior. If a device does support concurrent access, both the host and the device may atomically modify the same data inside an allocation. Allocations, or pieces of allocations, are now free to migrate to different devices in the same context that also support this capability. Additionally, many devices that support concurrent access may support a working set of shared allocations larger than device memory. Users may query whether a device supports concurrent access with atomic modification of shared allocations through the aspect `aspect::usm_atomic_shared_allocations`. See Table 26 in Section 4.6.4.3 for more details.

Performance hints for shared allocations may be specified by the user by enqueueing `prefetch` operations on a device. These operations inform the SYCL runtime that the specified shared allocation is likely to be accessed on the device in the future, and that it is free to migrate the allocation to the device. More about `prefetch` is found in Table 28 and Table 135. If a device supports concurrent access to shared allocations, then `prefetch` operations may be overlapped with kernel execution.

Additionally, users may use the `mem_advise` member function to annotate shared allocations with `advice`. Valid `advice` is defined by the device and its associated backend. See Table 28 and Table 135 for more information.

In the most capable systems, users do not need to use SYCL USM allocation functions to create shared allocations. The system allocator (`malloc/new`) may instead be used. Likewise, `std::free` and `delete` are used instead of `sycl::free`. Note that host and device allocations are unaffected by this change and must still be allocated using their respective USM functions in order to guarantee their behavior. Users may query the device to determine if system allocations are supported for use on the device, through `aspect::usm_system_allocations`.

### 4.8.3. USM allocations

USM provides several allocation functions. These functions accept a `property_list` parameter, which is provided for future extensibility. The core SYCL specification does not yet define any USM allocation properties.

Some of the allocation functions take an explicit alignment parameter. Like `std::aligned_alloc`, these functions return `nullptr` if the alignment is not supported by the implementation. Some of the allocation functions are templated on the allocated type `T` and some are not. The following table specifies the alignment guarantees for each category.

<table>
<thead>
<tr>
<th>Category</th>
<th>Alignment guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>No alignment parameter Not templated on allocation type</td>
<td>Pointer is suitably aligned for any object with fundamental alignment whose size is less than or equal to the requested allocation size.</td>
</tr>
<tr>
<td>No alignment parameter Templated on allocation type <code>T</code></td>
<td>Pointer is suitably aligned for an object of type <code>T</code>.</td>
</tr>
</tbody>
</table>

Table 101. Alignment guarantees of USM allocation functions
### 4.8.3.1. C++ allocator interface

SYCL defines an allocator class named `usm_allocator` that satisfies the C++ named requirement `Allocator`. The `AllocKind` template parameter can be either `usm::alloc::host` or `usm::alloc::shared`, causing the allocator to make either host USM allocations or shared USM allocations.

There is no specialization for `usm::alloc::device` because an `Allocator` is required to allocate memory that is accessible on the host.

The `usm_allocator` class has a template argument `Alignment`, which specifies the minimum alignment for memory that it allocates. This alignment is used even if the allocator is rebound to a different type. Memory allocated by this allocator is suitably aligned for objects of its underlying `value_type` or at the alignment specified by `Alignment`, whichever is greater.

A synopsis of the `usm_allocator` class is provided below. The constructors are listed in Table 102.

```cpp
1 template<typename T, usm::alloc AllocKind, size_t Alignment = 0>
2 class usm_allocator {
3 public:
4   using value_type = T;
5   using propagate_on_container_copy_assignment = std::true_type;
6   using propagate_on_container_move_assignment = std::true_type;
7   using propagate_on_container_swap = std::true_type;
8
9 public:
10   template<typename U> struct rebind {
11     typedef usm_allocator<U, AllocKind, Alignment> other;
12   };
13
14   usm_allocator() = delete;
15   usm_allocator(const context& syclContext,
16                  const device& syclDevice,
17                  const property_list& propList = {});
18   usm_allocator(const queue& syclQueue,
19                  const property_list& propList = {});
20   usm_allocator(const usm_allocator& other);
21   usm_allocator(usm_allocator&&) noexcept;
22   usm_allocator& operator=(const usm_allocator&);
23   usm_allocator& operator=(usm_allocator&&);
24
25   template<class U>
26   usm_allocator(usm_allocator<U, AllocKind, Alignment> const& noexcept;
27
28   /// Allocate memory
29   T* allocate(size_t count);
30
31   /// Deallocate memory
```
```cpp
void deallocate(T* Ptr, size_t count);
```

/// Equality Comparison
/// Allocators only compare equal if they are of the same USM kind, alignment,
/// context, and device
template <class U, usm::alloc AllocKindU, size_t AlignmentU>
friend bool operator==(const usm_allocator<T, AllocKind, Alignment>&,
                       const usm_allocator<U, AllocKindU, AlignmentU>&);
```

/// Inequality Comparison
/// Allocators only compare unequal if they are not of the same USM kind, alignment,
/// context, or device
template <class U, usm::alloc AllocKindU, size_t AlignmentU>
friend bool operator!=(const usm_allocator<T, AllocKind, Alignment>&,
                       const usm_allocator<U, AllocKindU, AlignmentU>&);
```

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
</table>
| `usm_allocator(const context& syclContext, const device& syclDevice,
                 const property_list& proplist = {})` | Constructs a `usm_allocator` instance that allocates USM for the provided context and device. If `AllocKind` is `usm::alloc::host`, this constructor throws a synchronous exception with the `errc::feature_not_supported` error code if no device in `syclContext` has `aspect::usm_host_allocations`. The `syclDevice` is ignored for this allocation kind. If `AllocKind` is `usm::alloc::shared`, this constructor throws a synchronous exception with the `errc::feature_not_supported` error code if the `syclDevice` does not have `aspect::usm_shared_allocations`. The `syclDevice` must either be contained by `syclContext` or it must be a descendant device of some device that is contained by that context, otherwise this constructor throws a synchronous exception with the `errc::invalid` error code. Simplified constructor form where `syclQueue` provides the device and context. |

4.8.3.2. Device allocation functions

The functions in Table 103 allocate device USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with `sycl::free` in order to avoid a memory
leak. If there are not enough resources to allocate the requested memory, these functions return `nullptr`.

When the allocation size is zero bytes (numBytes or count is zero), these functions behave in a manner consistent with C++ `std::malloc`. The value returned is unspecified in this case, and the returned pointer may not be used to access storage. If this pointer is not null, it must be passed to `sycl::free` to avoid a memory leak.

Table 103. Device USM Allocation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void* sycl::malloc_device(size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = { })</code></td>
<td>Returns a pointer to the newly allocated memory, which is allocated on syclDevice. The allocation size is specified in bytes. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the syclDevice does not have <code>aspect::usm_device_allocations</code>. The syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>template&lt;typename T&gt; T* sycl::malloc_device(size_t count, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = { })</code></td>
<td>Returns a pointer to the newly allocated memory, which is allocated on syclDevice. The allocation size is specified in number of elements of type T. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the syclDevice does not have <code>aspect::usm_device_allocations</code>. The syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>void* sycl::malloc_device(size_t numBytes, const queue&amp; syclQueue, const property_list&amp; propList = { })</code></td>
<td>Simplified form where syclQueue provides the device and context.</td>
</tr>
<tr>
<td><code>template&lt;typename T&gt; T* sycl::malloc_device(size_t count, const queue&amp; syclQueue, const property_list&amp; propList = { })</code></td>
<td>Simplified form where syclQueue provides the device and context.</td>
</tr>
</tbody>
</table>
### 4.8.3.3. Host allocation functions

The functions in Table 104 allocate host USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with `sycl::free` in order to avoid a memory leak.

#### Table 104: Host allocation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void* sycl::aligned_alloc_device(size_t alignment, size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is allocated on <code>syclDevice</code>. The allocation is specified in bytes and aligned according to <code>alignment</code>. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the <code>syclDevice</code> does not have <code>aspect::usm_device_allocations</code>. The <code>syclDevice</code> must either be contained by <code>syclContext</code> or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::aligned_alloc_device(size_t alignment, size_t count, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is allocated on <code>syclDevice</code>. The allocation is specified in number of elements of type <code>T</code> and aligned according to <code>alignment</code>. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the <code>syclDevice</code> does not have <code>aspect::usm_device_allocations</code>. The <code>syclDevice</code> must either be contained by <code>syclContext</code> or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>void* sycl::aligned_alloc_device(size_t alignment, size_t numBytes, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where <code>syclQueue</code> provides the device and context.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::aligned_alloc_device(size_t alignment, size_t count, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where <code>syclQueue</code> provides the device and context.</td>
</tr>
</tbody>
</table>
If there are not enough resources to allocate the requested memory, these functions return `nullptr`.

### Table 104. Host USM Allocation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void* sycl::malloc_host(size_t numBytes, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory. This allocation is specified in bytes. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if no device in <code>syclContext</code> has <code>aspect::usm_host_allocations</code>.</td>
</tr>
</tbody>
</table>
| `template<typename T> T* sycl::malloc_host(size_t count, const context& syclContext, const property_list& propList = {})` | Returns a pointer to the newly allocated memory. This allocation is specified in number of elements of type `T`. Throws a synchronous exception with the `errc::feature_not_supported` error code if no device in `syclContext` has `aspect::usm_host_allocations`.
| `void* sycl::malloc_host(size_t numBytes, const queue& syclQueue, const property_list& propList = {})` | Simplified form where `syclQueue` provides the context. |
| `template<typename T> T* sycl::malloc_host(size_t count, const queue& syclQueue, const property_list& propList = {})` | Simplified form where `syclQueue` provides the context. |
| `void* sycl::aligned_alloc_host(size_t alignment, size_t numBytes, const context& syclContext, const property_list& propList = {})` | Returns a pointer to the newly allocated memory. This allocation is specified in bytes and aligned according to `alignment`. Throws a synchronous exception with the `errc::feature_not_supported` error code if no device in `syclContext` has `aspect::usm_host_allocations`.
| `template<typename T> T* sycl::aligned_alloc_host(size_t alignment, size_t count, const context& syclContext, const property_list& propList = {})` | Returns a pointer to the newly allocated memory. This allocation is specified in elements of type `T` and aligned according to `alignment`. Throws a synchronous exception with the `errc::feature_not_supported` error code if no device in `syclContext` has `aspect::usm_host_allocations`.
| `void* sycl::aligned_alloc_host(size_t alignment, size_t numBytes, const queue& syclQueue, const property_list& propList = {})` | Simplified form where `syclQueue` provides the context. |
4.8.3.4. Shared allocation functions

The functions in Table 105 allocate shared USM. On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with `sycl::free` in order to avoid a memory leak. If there are not enough resources to allocate the requested memory, these functions return `nullptr`.

Table 105. Shared USM Allocation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename T&gt; void* sycl::aligned_alloc_host(size_t alignment, size_t count, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where <code>syclQueue</code> provides the context.</td>
</tr>
<tr>
<td><code>void* sycl::malloc_shared(size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is associated with <code>syclDevice</code>. This allocation is specified in bytes. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the <code>syclDevice</code> does not have <code>aspect::usm_shared_allocations</code>. The <code>syclDevice</code> must either be contained by <code>syclContext</code> or it must be a descendend device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::malloc_shared(size_t count, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is associated with <code>syclDevice</code>. This allocation is specified in number of elements of type <code>T</code>. Throws a synchronous exception with the <code>errc::feature_not_supported</code> error code if the <code>syclDevice</code> does not have <code>aspect::usm_shared_allocations</code>. The <code>syclDevice</code> must either be contained by <code>syclContext</code> or it must be a descendend device of some device that is contained by that context, otherwise this function throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>void* sycl::malloc_shared(size_t numBytes, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where syclQueue provides the device and context.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::malloc_shared(size_t count, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where syclQueue provides the device and context.</td>
</tr>
<tr>
<td><code>void* sycl::aligned_alloc_shared(size_t alignment, size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is associated with syclDevice. This allocation is specified in bytes and aligned according to alignment. Throws a synchronous exception with the errc::feature_not_supported error code if the syclDevice does not have aspect::usm_shared_allocations. The syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::aligned_alloc_shared(size_t alignment, size_t count, const device&amp; syclDevice, const context&amp; syclContext, const property_list&amp; propList = {})</code></td>
<td>Returns a pointer to the newly allocated memory, which is associated with syclDevice. This allocation is specified in number of elements of type T and aligned aligned according to alignment. Throws a synchronous exception with the errc::feature_not_supported error code if the syclDevice does not have aspect::usm_shared_allocations. The syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td><code>void* sycl::aligned_alloc_shared(size_t alignment, size_t numBytes, const queue&amp; syclQueue, const property_list&amp; propList = {})</code></td>
<td>Simplified form where syclQueue provides the device and context.</td>
</tr>
</tbody>
</table>
4.8.3.5. Parameterized allocation functions

The functions in Table 106 take a kind parameter that specifies the type of USM to allocate. When kind is usm::alloc::device, then the allocation device must have aspect::usm_device_allocations. When kind is usm::alloc::host, at least one device in the allocation context must have aspect::usm_host_allocations. When kind is usm::alloc::shared, the allocation device must have aspect::usm_shared_allocations. If these requirements are violated, the allocation function throws a synchronous exception with the errc::feature_not_supported error code.

On success, these functions return a pointer to the newly allocated memory, which must eventually be deallocated with sycl::free in order to avoid a memory leak. If there are not enough resources to allocate the requested memory, these functions return nullptr.

Table 106. Parameterized USM Allocation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void* sycl::malloc(size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, usm::alloc kind, const property_list&amp; propList = {})</td>
<td>Returns a pointer to the newly allocated memory of type kind. This allocation size is specified in bytes. The syclDevice parameter is ignored if kind is usm::alloc::host. If kind is not usm::alloc::host, syclDevice must either be contained by syclContext or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td>template &lt;typename T&gt; T* sycl::aligned_alloc_shared(size_t alignment, size_t count, const queue&amp; syclQueue, const property_list&amp; propList = {})</td>
<td>Returns a pointer to the newly allocated memory of type kind. This allocation size is specified in number of elements of type T. The syclDevice parameter is ignored if kind is usm::alloc::host. If kind is not usm::alloc::host, syclDevice must either be contained by syclContext or it must be a descendant device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>void* sycl::malloc(size_t numBytes, const queue&amp; syclQueue, usm::alloc kind, const property_list&amp; propList = { })</td>
<td>Simplified form where syclQueue provides the context and any necessary device.</td>
</tr>
<tr>
<td>template &lt;typename T&gt; T* sycl::malloc(size_t count, const queue&amp; syclQueue, usm::alloc kind, const property_list&amp; propList = { })</td>
<td>Simplified form where syclQueue provides the context and any necessary device.</td>
</tr>
<tr>
<td>void* sycl::aligned_alloc(size_t alignment, size_t numBytes, const device&amp; syclDevice, const context&amp; syclContext, usm::alloc kind, const property_list&amp; propList = { })</td>
<td>Returns a pointer to the newly allocated memory of type kind. This allocation is specified in bytes and is aligned according to alignment. The syclDevice parameter is ignored if kind is usm::alloc::host. If kind is not usm::alloc::host, syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td>template &lt;typename T&gt; T* sycl::aligned_alloc(size_t alignment, size_t count, const device&amp; syclDevice, const context&amp; syclContext, usm::alloc kind, const property_list&amp; propList = { })</td>
<td>Returns a pointer to the newly allocated memory of type kind. This allocation is specified in number of elements of type T and is aligned according to alignment. The syclDevice parameter is ignored if kind is usm::alloc::host. If kind is not usm::alloc::host, syclDevice must either be contained by syclContext or it must be a descendent device of some device that is contained by that context, otherwise this function throws a synchronous exception with the errc::invalid error code.</td>
</tr>
<tr>
<td>void* sycl::aligned_alloc(size_t alignment, size_t numBytes, const queue&amp; syclQueue, usm::alloc kind, const property_list&amp; propList = { })</td>
<td>Simplified form where syclQueue provides the context and any necessary device.</td>
</tr>
</tbody>
</table>
### 4.8.3.6. Memory deallocation functions

#### Table 107. USM Deallocation Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename T&gt; T* sycl::aligned_alloc(size_t alignment, size_t count, const queue&amp; syclQueue, usm::alloc kind, const property_list&amp; propList = {})</code></td>
<td>Simplified form where <code>syclQueue</code> provides the context and any necessary device.</td>
</tr>
</tbody>
</table>

#### 4.8.3.6. Memory deallocation functions

**Table 107. USM Deallocation Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void sycl::free(void* ptr, const context&amp; syclContext)</code></td>
<td>Frees an allocation. The memory pointed to by <code>ptr</code> must have been allocated using one of the USM allocation routines. <code>syclContext</code> must be the same context that was used to allocate the memory. The memory is freed without waiting for commands operating on it to be completed. If commands that use this memory are in-progress or are enqueued the behavior is undefined.</td>
</tr>
<tr>
<td><code>void sycl::free(void* ptr, const queue&amp; syclQueue)</code></td>
<td>Alternate form where <code>syclQueue</code> provides the context.</td>
</tr>
</tbody>
</table>

### 4.8.4. Unified shared memory pointer queries

Since USM pointers look like raw C++ pointers, users cannot deduce what kind of USM allocation a given pointer may be from examining its type. However, two functions are defined that let users query the type of a USM allocation and, if applicable, the device on which it was allocated. These query functions are only supported on the host.

#### Table 108. USM Pointer Query Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>usm::alloc get_pointer_type(const void* ptr, const context&amp; syclContext)</code></td>
<td>Returns the USM allocation type for <code>ptr</code> if <code>ptr</code> falls inside a valid USM allocation for the context <code>syclContext</code>. Returns <code>usm::alloc::unknown</code> if <code>ptr</code> does not point within a valid USM allocation from <code>syclContext</code>.</td>
</tr>
</tbody>
</table>
Function

```c
device get_pointer_device(const void* ptr, const context& syclContext)
```

Description

Returns the `device` associated with the USM allocation. If `ptr` points within a device USM allocation or a shared USM allocation for the context `syclContext`, returns the same device that was passed when allocating the memory. If `ptr` points within a host USM allocation for the context `syclContext`, returns the first device in `syclContext`. Throws a synchronous exception with the `errc::invalid` error code if `ptr` does not point within a valid USM allocation from `syclContext`.

4.9. Expressing parallelism through kernels

4.9.1. Ranges and index space identifiers

The data parallelism of the SYCL kernel execution model requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this, SYCL exposes types to define the range of execution and to identify a given execution instance’s point in the iteration space.

The following types are defined: `range`, `nd_range`, `id`, `item`, `h_item`, `nd_item` and `group`.

When constructing multi-dimensional ids or ranges from integers, the elements are written such that the right-most element varies fastest in a linearization of the multi-dimensional space (see Section 3.11.1).

Table 109. Summary of types used to identify points in an index space, and ranges over which those points can vary

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>A point within a range</td>
</tr>
<tr>
<td>range</td>
<td>Bounds over which an id may vary</td>
</tr>
<tr>
<td>item</td>
<td>Pairing of an id (specific point) and the range that it is bounded by</td>
</tr>
<tr>
<td>nd_range</td>
<td>Encapsulates both global and local (work-group size) ranges over which work-item ids will vary</td>
</tr>
<tr>
<td>nd_item</td>
<td>Encapsulates two items, one for global id and range, and one for local id and range</td>
</tr>
<tr>
<td>h_item</td>
<td>Index point queries within hierarchical parallelism (parallel_for_work_item). Encapsulates physical global and local ids and ranges, as well as a logical local id and range defined by hierarchical parallelism</td>
</tr>
</tbody>
</table>
4.9.1.1. `range` class

`range<int Dimensions>` is a 1D, 2D or 3D vector that defines the iteration domain of either a single work-group in a parallel dispatch, or the overall Dimensions of the dispatch. It can be constructed from integers.

The SYCL `range` class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL `range` class is provided below. The constructors, member functions and non-member functions of the SYCL `range` class are listed in Table 110, Table 111 and Table 112 respectively. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```
namespace sycl {
  template <int Dimensions = 1> class range {
    public:
      static constexpr int dimensions = Dimensions;

      /* The following constructor is only available in the range class
       * specialization where: Dimensions==1 */
      range(size_t dim0);
      /* The following constructor is only available in the range class
       * specialization where: Dimensions==2 */
      range(size_t dim0, size_t dim1);
      /* The following constructor is only available in the range class
       * specialization where: Dimensions==3 */
      range(size_t dim0, size_t dim1, size_t dim2);

      /* -- common interface members -- */

      size_t get(int dimension) const;
      size_t operator[](int dimension);
      size_t operator[](int dimension) const;

      size_t size() const;

      // OP is: +, -, *, /, %, <<, >>, &,
      friend range operatorOP(const range& lhs, const range& rhs) { /* ... */
      }
      friend range operatorOP(const range& lhs, const size_t& rhs) { /* ... */
      }

      // OP is: +=, -=, *=, /=, %, <<=, >>=,
      friend range& operatorOP(range& lhs, const range& rhs) { /* ... */
      }
      friend range& operatorOP(range& lhs, const size_t& rhs) { /* ... */
    }
  }
}
```
friend range operatorOP(const size_t &lhs, const range &rhs) { /* ... */
}

friend range operatorOP(const range &rhs) { /* ... */
}

friend range operatorOP(range &rhs) { /* ... */
}

friend range operatorOP(range &lhs, int) { /* ... */
}

}; // namespace sycl

Table 110. Constructors of the range class template

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range(size_t dim0)</td>
<td>Construct a 1D range with value dim0. Only valid when the template parameter Dimensions is equal to 1.</td>
</tr>
<tr>
<td>range(size_t dim0, size_t dim1)</td>
<td>Construct a 2D range with values dim0 and dim1. Only valid when the template parameter Dimensions is equal to 2.</td>
</tr>
<tr>
<td>range(size_t dim0, size_t dim1, size_t dim2)</td>
<td>Construct a 3D range with values dim0, dim1 and dim2. Only valid when the template parameter Dimensions is equal to 3.</td>
</tr>
</tbody>
</table>

Table 111. Member functions of the range class template

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t get(int dimension) const</td>
<td>Return the value of the specified dimension of the range.</td>
</tr>
<tr>
<td>size_t &amp; operator[](int dimension)</td>
<td>Return the l-value of the specified dimension of the range.</td>
</tr>
<tr>
<td>size_t operator[](int dimension) const</td>
<td>Return the value of the specified dimension of the range.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>size_t size() const</code></td>
<td>Return the size of the range computed as dimension0*...*dimensionN.</td>
</tr>
</tbody>
</table>

**Table 112. Hidden friend functions of the SYCL range class template**

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>range operatorOP(const range&amp; lhs, const range&amp; rhs)</code></td>
<td>Where OP is: +, -, *, /, %, &lt;&lt;, &gt;&gt;, &amp;,</td>
</tr>
<tr>
<td><code>range operatorOP(const range&amp; lhs, const size_t&amp; rhs)</code></td>
<td>Where OP is: +, -, *, /, %, &lt;&lt;, &gt;&gt;, &amp;,</td>
</tr>
<tr>
<td><code>range&amp; operatorOP(range&amp; lhs, const range&amp; rhs)</code></td>
<td>Where OP is: +=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &amp;=,</td>
</tr>
</tbody>
</table>
### Hidden friend function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `range & operatorOP(range & lhs, const size_t & rhs)` | Assigns each element of `lhs range` instance with the result of an element-wise `OP` operator between each element of `lhs range` and the `rhs size_t` and returns `lhs range`. If the operator returns a `bool`, the result is the cast to `size_t`. Where `OP` is: `+=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |`.
| `range operatorOP(const size_t & lhs, const range & rhs)` | Constructs and returns a new instance of the SYCL `range` class template with the same dimensionality as the `rhs` SYCL `range`, where each element of the new SYCL `range` instance is the result of an element-wise `OP` operator between the `lhs size_t` and each element of the `rhs` SYCL `range`. If the operator returns a `bool`, the result is the cast to `size_t`. Where `OP` is: `+, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=`. |
| `range operatorOP(const range & rhs)` | Constructs and returns a new instance of the SYCL `range` class template with the same dimensionality as the `rhs` SYCL `range`, where each element of the new SYCL `range` instance is the result of an element-wise `OP` operator on the `rhs` SYCL `range`. Where `OP` is: unary `+`, unary `-`. |
| `range & operatorOP(range & rhs)` | Assigns each element of the `rhs range` instance with the result of an element-wise `OP` operator on each element of the `rhs range` and returns this `range`. Where `OP` is: prefix `++`, prefix `--`. |
| `range operatorOP(range & lhs, int)` | Make a copy of the `lhs range`. Assigns each element of the `lhs range` instance with the result of an element-wise `OP` operator on each element of the `lhs range`. Then return the initial copy of the `range`. Where `OP` is: postfix `++`, postfix `--`. |

### 4.9.1.2. `nd_range` class

```c++
namespace sycl {

// nd_range class

}```
nd_range<Dimensions> defines the iteration domain of both the work-groups and the overall dispatch. To define this the nd_range comprises two ranges: the whole range over which the kernel is to be executed, and the range of each work group.

The SYCL nd_range class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL nd_range class is provided below. The constructors and member functions of the SYCL nd_range class are listed in Table 113 and Table 114 respectively. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

Table 113. Constructors of the nd_range class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nd_range&lt;Dimensions&gt;(range&lt;Dimensions&gt; globalSize, range&lt;Dimensions&gt; localSize) id&lt;Dimensions&gt; offset = id&lt;Dimensions&gt;())</td>
<td>Construct an nd_range from the local and global constituent ranges. Supplying the option offset is deprecated in SYCL 2020. If the offset is not provided it will default to no offset.</td>
</tr>
</tbody>
</table>

Table 114. Member functions for the nd_range class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range&lt;Dimensions&gt; get_global_range() const</td>
<td>Return the constituent global range.</td>
</tr>
<tr>
<td>range&lt;Dimensions&gt; get_local_range() const</td>
<td>Return the constituent local range.</td>
</tr>
<tr>
<td>range&lt;Dimensions&gt; get_group_range() const</td>
<td>Return a range representing the number of groups in each dimension. This range would result from globalSize/localSize as provided on construction.</td>
</tr>
</tbody>
</table>
4.9.1.3. id class

`id<Dimensions>` is a vector of Dimensions that is used to represent an id into a global or local range. It can be used as an index in an accessor of the same rank. The subscript operator (`operator[](n)`) returns the component `n` as a `size_t`.

The SYCL id class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL id class is provided below. The constructors, member functions and non-member functions of the SYCL id class are listed in Table 115, Table 116 and Table 117 respectively. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```cpp
namespace sycl {
    template <int Dimensions = 1> class id {
        public:
            static constexpr int dimensions = Dimensions;

            id();

            /* The following constructor is only available in the id class
                * specialization where: Dimensions==1 */
            id(size_t dim0);

            /* The following constructor is only available in the id class
                * specialization where: Dimensions==2 */
            id(size_t dim0, size_t dim1);

            /* The following constructor is only available in the id class
                * specialization where: Dimensions==3 */
            id(size_t dim0, size_t dim1, size_t dim2);

            /* -- common interface members -- */

            id(const range<Dimensions>& range);
            id(const item<Dimensions>& item);

            size_t get(int dimension) const;
            size_t operator[](int dimension) const;

            // only available if Dimensions == 1
            operator size_t() const;

            // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
            friend id operatorOP(const id& lhs, const id& rhs) { /* ... */ }

            friend id operatorOP(const id& lhs, const size_t& rhs) { /* ... */ }

            // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
            friend id operatorOP(id& lhs, const id& rhs) { /* ... */ }

            friend id operatorOP(id& lhs, const size_t& rhs) { /* ... */ }

            // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
            friend id operatorOP(const size_t& lhs, const id& rhs) { /* ... */ }
    }
}
```
4.9.1.3. id class

```cpp
// OP is unary +, -
friend id operatorOP(const id& rhs) { /* ... */ }

// OP is prefix ++, --
friend id& operatorOP(id& rhs) { /* ... */ }

// OP is postfix ++, --
friend id operatorOP(id& lhs, int) { /* ... */ }

};

// Deduction guides
id(size_t) -> id<1>;
id(size_t, size_t) -> id<2>;
id(size_t, size_t, size_t) -> id<3>;

} // namespace sycl
```

**Table 115. Constructors of the id class template**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id()</td>
<td>Construct a SYCL id with the value 0 for each dimension.</td>
</tr>
<tr>
<td>id(size_t dim0)</td>
<td>Construct a 1D id with value dim0. Only valid when the template parameter Dimensions is equal to 1.</td>
</tr>
<tr>
<td>id(size_t dim0, size_t dim1)</td>
<td>Construct a 2D id with values dim0, dim1. Only valid when the template parameter Dimensions is equal to 2.</td>
</tr>
<tr>
<td>id(size_t dim0, size_t dim1, size_t dim2)</td>
<td>Construct a 3D id with values dim0, dim1, dim2. Only valid when the template parameter Dimensions is equal to 3.</td>
</tr>
<tr>
<td>id(const range&lt;Dimensions&gt;&amp; range)</td>
<td>Construct an id from the dimensions of range.</td>
</tr>
<tr>
<td>id(const item&lt;Dimensions&gt;&amp; item)</td>
<td>Construct an id from item.get_id().</td>
</tr>
</tbody>
</table>

**Table 116. Member functions of the id class template**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t get(int dimension) const</td>
<td>Return the value of the id for dimension Dimension.</td>
</tr>
<tr>
<td>size_t&amp; operator[](int dimension)</td>
<td>Return a reference to the requested dimension of the id object.</td>
</tr>
</tbody>
</table>
**Member function**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t</code> operator[](int dimension) const</td>
</tr>
<tr>
<td>Return the value of the requested dimension of the <code>id</code> object.</td>
</tr>
<tr>
<td><code>operator size_t()</code> const</td>
</tr>
<tr>
<td>Available only when: <code>Dimensions == 1</code></td>
</tr>
<tr>
<td>Returns the same value as <code>get(0)</code></td>
</tr>
</tbody>
</table>

### Table 117. Hidden friend functions of the `id` class template

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id operatorOP(const id&amp; lhs, const id&amp; rhs)</code></td>
<td></td>
</tr>
<tr>
<td>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as <code>lhs id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and each element of the <code>rhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td>
<td></td>
</tr>
<tr>
<td><code>id operatorOP(const id&amp; lhs, const size_t&amp; rhs)</code></td>
<td></td>
</tr>
<tr>
<td>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as <code>lhs id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and the <code>rhs size_t</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td>
<td></td>
</tr>
<tr>
<td><code>id&amp; operatorOP(id&amp; lhs, const id&amp; rhs)</code></td>
<td></td>
</tr>
<tr>
<td>Assigns each element of <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and each element of the <code>rhs id</code> and returns <code>lhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>.</td>
<td></td>
</tr>
</tbody>
</table>
### Hidden friend function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id&amp; operatorOP(id&amp; lhs, const size_t&amp; rhs)</code></td>
<td>Assigns each element of <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator between each element of <code>lhs id</code> and the <code>rhs size_t</code> and returns <code>lhs id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>. Where <code>OP</code> is: `+=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &amp;=,</td>
</tr>
<tr>
<td><code>id operatorOP(const size_t&amp; lhs, const id&amp; rhs)</code></td>
<td>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as the <code>rhs SYCL id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator between the <code>lhs size_t</code> and each element of the <code>rhs SYCL id</code>. If the operator returns a <code>bool</code> the result is the cast to <code>size_t</code>. Where <code>OP</code> is: `+, -, *, /, %, &lt;&lt;,</td>
</tr>
<tr>
<td><code>id operatorOP(const id&amp; rhs)</code></td>
<td>Constructs and returns a new instance of the SYCL <code>id</code> class template with the same dimensionality as the <code>rhs SYCL id</code>, where each element of the new SYCL <code>id</code> instance is the result of an element-wise <code>OP</code> operator on the <code>rhs SYCL id</code>. Where <code>OP</code> is: unary <code>+</code>, unary <code>-</code>.</td>
</tr>
<tr>
<td><code>id&amp; operatorOP(id&amp; rhs)</code></td>
<td>Assigns each element of the <code>rhs id</code> instance with the result of an element-wise <code>OP</code> operator on each element of the <code>rhs id</code> and returns this <code>id</code>. Where <code>OP</code> is: prefix <code>++</code>, prefix <code>--</code>.</td>
</tr>
<tr>
<td><code>id operatorOP(id&amp; lhs, int)</code></td>
<td>Make a copy of the <code>lhs id</code>. Assigns each element of the <code>lhs id</code> instance with the result of an element-wise <code>OP</code> operator on each element of the <code>lhs id</code>. Then return the initial copy of the <code>id</code>. Where <code>OP</code> is: postfix <code>++</code>, postfix <code>--</code>.</td>
</tr>
</tbody>
</table>

### 4.9.1.4. item class

`item` identifies an instance of the function object executing at each point in a range. It is passed to a `parallel_for` call or returned by member functions of `h_item`. It encapsulates enough information to identify the work-item’s range of possible values and its ID in that range. It can optionally carry the offset of the
range if provided to the `parallel_for`; note this is deprecated in SYCL 2020. Instances of the `item` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `item` class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL `item` class is provided below. The member functions of the SYCL `item` class are listed in Table 116. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```cpp
namespace sycl {

template <int Dimensions = 1, bool WithOffset = true> class item {

public:

static constexpr int dimensions = Dimensions;

item() = delete;

/* -- common interface members -- */

id<Dimensions> get_id() const;
size_t get_id(int dimension) const;
size_t operator[](int dimension) const;
range<Dimensions> get_range() const;
size_t get_range(int dimension) const;

// Deprecated in SYCL 2020.
// only available if WithOffset is true
id<Dimensions> get_offset() const;

// only available if WithOffset is false
operator item<Dimensions, true>() const;

// only available if Dimensions == 1
operator size_t() const;
size_t get_linear_id() const;

}; // namespace sycl
```

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id&lt;Dimensions&gt; get_id() const</code></td>
<td>Return the constituent <code>id</code> representing the work-item's position in the iteration space.</td>
</tr>
<tr>
<td><code>size_t get_id(int dimension) const</code></td>
<td>Return the same value as <code>get_id()[dimension]</code>.</td>
</tr>
<tr>
<td><code>size_t operator[](int dimension) const</code></td>
<td>Return the same value as <code>get_id(dimension)</code>.</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>range&lt;Dimensions&gt; get_range() const</td>
<td>Returns a range representing the dimensions of the range of possible values of the item.</td>
</tr>
<tr>
<td>size_t get_range(int dimension) const</td>
<td>Return the same value as get_range().get(dimension).</td>
</tr>
<tr>
<td>id&lt;Dimensions&gt; get_offset() const</td>
<td>Deprecated in SYCL 2020. Returns an id representing the n-dimensional offset provided to the parallel_for and that is added by the runtime to the global-ID of each work-item, if this item represents a global range. For an item converted from an item with no offset this will always return an id of all 0 values. This member function is only available if with_offset is true.</td>
</tr>
<tr>
<td>operator item&lt;Dimensions, true&gt;() const</td>
<td>Available only when: with_offset == false Returns an item representing the same information as the object holds but also includes the offset set to 0. This conversion allow users to seamlessly write code that assumes an offset and still provides an offset-less item.</td>
</tr>
<tr>
<td>operator size_t() const</td>
<td>Available only when: Dimensions == 1 Returns the same value as get_id(0).</td>
</tr>
<tr>
<td>size_t get_linear_id() const</td>
<td>Return the id as a linear index value. Calculating a linear address from the multi-dimensional index follows Section 3.11.1.</td>
</tr>
</tbody>
</table>

### 4.9.1.5. nd_item class

`nd_item<int Dimensions>` identifies an instance of the function object executing at each point in an `nd_range<int Dimensions>` passed to a `parallel_for` call. It encapsulates enough information to identify the work-item’s local and global ids, the work-group id and also provides access to the group and sub-group classes. Instances of the `nd_item<int Dimensions>` class are not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `nd_item` class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL `nd_item` class is provided below. The member functions of the SYCL `nd_item` class are listed in Table 119. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.
namespace sycl {

template <int Dimensions = 1> class nd_item {
  public:
    static constexpr int dimensions = Dimensions;

    nd_item() = delete;

    // -- common interface members -- */

    id<Dimensions> get_global_id() const;

    size_t get_global_id(int dimension) const;

    size_t get_global_linear_id() const;

    id<Dimensions> get_local_id() const;

    size_t get_local_id(int dimension) const;

    size_t get_local_linear_id() const;

    group<Dimensions> get_group() const;

    sub_group get_sub_group() const;

    size_t get_group(int dimension) const;

    size_t get_group_linear_id() const;

    range<Dimensions> get_group_range() const;

    size_t get_group_range(int dimension) const;

    range<Dimensions> get_global_range() const;

    size_t get_global_range(int dimension) const;

    range<Dimensions> get_local_range() const;

    size_t get_local_range(int dimension) const;

    // Deprecated in SYCL 2020.

    id<Dimensions> get_offset() const;

    nd_range<Dimensions> get_nd_range() const;

    template <typename DataT>
    device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
                                        decorated_global_ptr<DataT> src,
                                        size_t numElements) const;

    template <typename DataT>
    device_event async_work_group_copy(decorated_global_ptr<DataT> dest,
                                        decorated_local_ptr<DataT> src,
                                        size_t numElements) const;
}
template<typename DataT>
device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
                                   decorated_global_ptr<DataT> src,
                                   size_t numElements,
                                   size_t srcStride) const;

template<typename DataT>
device_event async_work_group_copy(decorated_global_ptr<DataT> dest,
                                   decorated_local_ptr<DataT> src,
                                   size_t numElements,
                                   size_t destStride) const;

template<typename... EventTN>
void wait_for(EventTN... events) const;
}

// namespace sycl

Table 119. Member functions for the nd_item class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id&lt;Dimensions&gt; get_global_id() const</td>
<td>Return the constituent global id representing the work-item's position in the global iteration space.</td>
</tr>
<tr>
<td>size_t get_global_id(int dimension) const</td>
<td>Return the constituent element of the global id representing the work-item's position in the nd-range in the given Dimension.</td>
</tr>
<tr>
<td>size_t get_global_linear_id() const</td>
<td>Return the constituent global id as a linear index value, representing the work-item's position in the global iteration space. The linear address is calculated from the multi-dimensional index by first subtracting the offset and then following Section 3.11.1.</td>
</tr>
<tr>
<td>id&lt;Dimensions&gt; get_local_id() const</td>
<td>Return the constituent local id representing the work-item's position within the current work-group.</td>
</tr>
<tr>
<td>size_t get_local_id(int dimension) const</td>
<td>Return the constituent element of the local id representing the work-item's position within the current work-group in the given Dimension.</td>
</tr>
<tr>
<td>size_t get_local_linear_id() const</td>
<td>Return the constituent local id as a linear index value, representing the work-item's position within the current work-group. The linear address is calculated from the multi-dimensional index following Section 3.11.1.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>group&lt;Dimensions&gt; get_group() const</code></td>
<td>Return the constituent <code>work-group</code>, <code>group</code> representing the <code>work-group</code>'s position within the overall <code>nd-range</code>.</td>
</tr>
<tr>
<td><code>sub_group get_sub_group() const</code></td>
<td>Return a <code>sub_group</code> representing the <code>sub-group</code> to which the work-item belongs.</td>
</tr>
<tr>
<td><code>size_t get_group(int dimension) const</code></td>
<td>Return the constituent element of the group id representing the <code>work-group</code>'s position within the overall <code>nd_range</code> in the given <code>Dimension</code>.</td>
</tr>
<tr>
<td><code>size_t get_group_linear_id() const</code></td>
<td>Return the group id as a linear index value. Calculating a linear address from a multi-dimensional index follows Section 3.11.1.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions&gt; get_group_range() const</code></td>
<td>Returns the number of <code>work groups</code> in the iteration space.</td>
</tr>
<tr>
<td><code>size_t get_group_range(int dimension) const</code></td>
<td>Return the number of <code>work groups</code> for <code>Dimension</code> in the iteration space.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions&gt; get_global_range() const</code></td>
<td>Returns a <code>range</code> representing the dimensions of the global iteration space.</td>
</tr>
</tbody>
</table>
| `size_t get_global_range(int dimension) const` | Return the same value as `get_global_range().get(dimension)`.
<p>| <code>range&lt;Dimensions&gt; get_local_range() const</code> | Returns a <code>range</code> representing the dimensions of the current work-group. |
| <code>size_t get_local_range(int dimension) const</code> | Return the same value as <code>get_local_range().get(dimension)</code>. |
| <code>id&lt;Dimensions&gt; get_offset() const</code> | Deprecated in SYCL 2020. Returns an <code>id</code> representing the n-dimensional offset provided to the constructor of the <code>nd_range</code> and that is added by the runtime to the <code>global id</code> of each <code>work-item</code>. |
| <code>nd_range&lt;Dimensions&gt; get_nd_range() const</code> | Returns the <code>nd_range</code> of the current execution. |</p>
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <typename DataT>
device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
decorated_global_ptr<DataT> src,
size_t numElements)
const` | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename DataT>
device_event async_work_group_copy(decorated_global_ptr<DataT> dest,
decorated_local_ptr<DataT> src,
size_t numElements) const` | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename DataT>
device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
decorated_global_ptr<DataT> src,
size_t numElements,
size_t srcStride) const` | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` with a source stride specified by `srcStride` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename DataT>
device_event async_work_group_copy(decorated_global_ptr<DataT> dest,
decorated_local_ptr<DataT> src,
size_t numElements,
size_t destStride) const` | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` with a destination stride specified by `destStride` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
| `template <typename... EventTN>
void wait_for(EventTN... events) const` | Permitted type for `EventTN` is `device_event`. Waits for the asynchronous operations associated with each `device_event` to complete. |

### 4.9.1.6. h_item class

`h_item<int Dimensions>` identifies an instance of a `group::parallel_for_work_item` function object executing at each point in a local `range<int Dimensions>` passed to a `parallel_for_work_item` call or to the corresponding `parallel_for_work_group` call if no `range` is passed to the `parallel_for_work_item` call. It encapsulates enough information to identify the `work-item`'s local and global `items` according to the information given to `parallel_for_work_group` (physical ids) as well as the `work-item`'s logical local `items` in the logical local range. All returned `items` objects are offset-less. Instances of the `h_item<int Dimensions>` class are
not user-constructible and are passed by the runtime to each instance of the function object.

The SYCL `h_item` class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL `h_item` class is provided below. The member functions of the SYCL `h_item` class are listed in Table 120. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```cpp
namespace sycl {
    template <int Dimensions> class h_item {
        public:
            static constexpr int dimensions = Dimensions;
            h_item() = delete;

            /* -- common interface members -- */
            item<Dimensions, false> get_global() const;
            item<Dimensions, false> get_local() const;
            item<Dimensions, false> get_logical_local() const;
            item<Dimensions, false> get_physical_local() const;
            range<Dimensions> get_global_range() const;
            size_t get_global_range(int dimension) const;
            id<Dimensions> get_global_id() const;
            size_t get_global_id(int dimension) const;
            range<Dimensions> get_local_range() const;
            size_t get_local_range(int dimension) const;
            id<Dimensions> get_local_id() const;
            size_t get_local_id(int dimension) const;
            range<Dimensions> get_logical_local_range() const;
            size_t get_logical_local_range(int dimension) const;
            id<Dimensions> get_logical_local_id() const;
            size_t get_logical_local_id(int dimension) const;
            range<Dimensions> get_physical_local_range() const;
            size_t get_physical_local_range(int dimension) const;
            id<Dimensions> get_physical_local_id() const;
            size_t get_physical_local_id(int dimension) const;
    }
}
```
Table 120. Member functions for the \texttt{h\_item} class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{item&lt;Dimensions, false&gt; get_global() const}</td>
<td>Return the constituent global \texttt{item} representing the work-item's position in the global iteration space as provided upon kernel invocation.</td>
</tr>
<tr>
<td>\texttt{item&lt;Dimensions, false&gt; get_local() const}</td>
<td>Return the same value as \texttt{get_logical_local()}.</td>
</tr>
<tr>
<td>\texttt{item&lt;Dimensions, false&gt; get_logical_local() const}</td>
<td>Return the constituent element of the local logical \texttt{item} work-item's position in the local iteration space as provided upon the invocation of the \texttt{group::parallel_for_work_item}. If the \texttt{group::parallel_for_work_item} was called without any logical local range then the member function returns the physical local \texttt{item}. A physical id can be computed from a logical id by getting the remainder of the integer division of the logical id and the physical range: \texttt{get_logical_local().get()} % \texttt{get_physical_local().get_range()} == \texttt{get_physical_local().get()}.</td>
</tr>
<tr>
<td>\texttt{item&lt;Dimensions, false&gt; get_physical_local() const}</td>
<td>Return the constituent element of the physical local \texttt{item} work-item's position in the local iteration space as provided (by the user or the runtime) upon the kernel invocation.</td>
</tr>
<tr>
<td>\texttt{range&lt;Dimensions&gt; get_global_range() const}</td>
<td>Return the same value as \texttt{get_global().get_range()}.</td>
</tr>
<tr>
<td>\texttt{size_t get_global_range(int dimension) const}</td>
<td>Return the same value as \texttt{get_global().get_range(dimension)}.</td>
</tr>
<tr>
<td>\texttt{id&lt;Dimensions&gt; get_global_id() const}</td>
<td>Return the same value as \texttt{get_global().get_id()}.</td>
</tr>
<tr>
<td>\texttt{size_t get_global_id(int dimension) const}</td>
<td>Return the same value as \texttt{get_global().get_id(dimension)}.</td>
</tr>
<tr>
<td>\texttt{range&lt;Dimensions&gt; get_local_range() const}</td>
<td>Return the same value as \texttt{get_local().get_range()}.</td>
</tr>
</tbody>
</table>
### 4.9.1.7. `group` class

The `group<int Dimensions>` encapsulates all functionality required to represent a particular **work-group** within a parallel execution. It is not user-constructible.

The local range stored in the group class is provided either by the programmer, when it is passed as an optional parameter to `parallel_for_work_group`, or by the runtime system when it selects the optimal work-group size. This allows the developer to always know how many work-items are in each executing work-group, even through the abstracted iteration range of the `parallel_for_work_item` loops.

The SYCL `group` class template provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL `group` class is provided below. The member functions of the SYCL `group` class are listed in Table 121. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```cpp
ten namespace sycl {
4template <int Dimensions = 1> class group {
5public:
6using id_type = id<Dimensions>;
7using range_type = range<Dimensions>;
8using linear_id_type = size_t;
```
static constexpr int dimensions = Dimensions;

static constexpr memory_scope fence_scope = memory_scope::work_group;

/* -- common interface members -- */

id<Dimensions> get_group_id() const;

size_t get_group_id(int dimension) const;

id<Dimensions> get_local_id() const;

size_t get_local_id(int dimension) const;

range<Dimensions> get_local_range() const;

size_t get_local_range(int dimension) const;

range<Dimensions> get_group_range() const;

size_t get_group_range(int dimension) const;

range<Dimensions> get_max_local_range() const;

size_t operator[](int dimension) const;

size_t get_group_linear_id() const;

size_t get_local_linear_id() const;

size_t get_group_linear_range() const;

size_t get_local_linear_range() const;

bool leader() const;

template <typename WorkItemFunctionT>
void parallel_for_work_item(const WorkItemFunctionT& func) const;

template <typename WorkItemFunctionT>
void parallel_for_work_item(range<Dimensions> logicalRange,
                           const WorkItemFunctionT& func) const;

template <typename DataT>
device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
                                   decorated_global_ptr<DataT> src,
                                   size_t numElements) const;

template <typename DataT>
device_event async_work_group_copy(decorated_global_ptr<DataT> dest,
                                   decorated_local_ptr<DataT> src,
                                   size_t numElements) const;

template <typename DataT>
device_event async_work_group_copy(decorated_local_ptr<DataT> dest,
                                   decorated_global_ptr<DataT> src,
                                   size_t numElements,
                                   size_t numElements,
template <typename DataT>


decorated_global_ptr<DataT> dest,

decorated_local_ptr<DataT> src,

size_t numElements,

size_t destStride) const;

template <typename DataT>

device_event async_work_group_copy(decorated_global_ptr<DataT> dest,

decorated_local_ptr<DataT> src,

size_t numElements,

size_t destStride) const;

Table 121. Member functions for the group class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id&lt;Dimensions&gt; get_group_id() const</td>
<td>Return an id representing the index of the work-group within the global nd-range for every dimension. Since the work-items in a work-group have a defined position within the global nd-range, the returned group id can be used along with the local id to uniquely identify the work-item in the global nd-range.</td>
</tr>
<tr>
<td>size_t get_group_id(int dimension) const</td>
<td>Return the same value as get_group_id()[dimension].</td>
</tr>
<tr>
<td>id&lt;Dimensions&gt; get_local_id() const</td>
<td>Return a SYCL id representing the calling work-item's position within the work-group.</td>
</tr>
<tr>
<td>size_t get_local_id(int dimension) const</td>
<td>It is undefined behavior for this member function to be invoked from within a parallel_for_work_item context.</td>
</tr>
<tr>
<td>range&lt;Dimensions&gt; get_local_range() const</td>
<td>Return a SYCL range representing all dimensions of the local range. This local range may have been provided by the programmer, or chosen by the SYCL runtime.</td>
</tr>
<tr>
<td>size_t get_local_range(int dimension) const</td>
<td>Return the same value as get_local_range()[dimension].</td>
</tr>
<tr>
<td>range&lt;Dimensions&gt; get_group_range() const</td>
<td>Return a range representing the number of work groups in the nd_range.</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t get_group_range(int dimension) const</code></td>
<td>Return the same value as <code>get_group_range()[dimension]</code>.</td>
</tr>
<tr>
<td><code>size_t operator[](int dimension) const</code></td>
<td>Return the same value as <code>get_group_id(dimension)</code>.</td>
</tr>
<tr>
<td><code>range&lt;Dimensions&gt; get_max_local_range() const</code></td>
<td>Return a <code>range</code> representing the maximum number of work-items in any <code>work-group</code> in the <code>nd_range</code>.</td>
</tr>
<tr>
<td><code>size_t get_group_linear_id() const</code></td>
<td>Get a linearized version of the <code>work-group id</code>. Calculating a linear <code>work-group id</code> from a multi-dimensional index follows Section 3.11.1.</td>
</tr>
<tr>
<td><code>size_t get_group_linear_range() const</code></td>
<td>Return the total number of <code>work-groups</code> in the <code>nd_range</code>.</td>
</tr>
<tr>
<td><code>size_t get_local_linear_id() const</code></td>
<td>Get a linearized version of the calling work-item’s <code>local id</code>. Calculating a linear <code>local id</code> from a multi-dimensional index follows Section 3.11.1. It is undefined behavior for this member function to be invoked from within a <code>parallel_for_work_item</code> context.</td>
</tr>
<tr>
<td><code>size_t get_local_linear_range() const</code></td>
<td>Return the total number of work-items in the <code>work-group</code>.</td>
</tr>
<tr>
<td><code>bool leader() const</code></td>
<td>Return true for exactly one work-item in the <code>work-group</code>, if the calling work-item is the leader of the work-group, and false for all other work-items in the work-group. The leader of the work-group is determined during construction of the work-group, and is invariant for the lifetime of the work-group. The leader of the work-group is guaranteed to be the work-item with a local id of 0.</td>
</tr>
<tr>
<td>Member function</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td></td>
</tr>
</tbody>
</table>
| template <typename WorkItemFunctionT>
void parallel_for_work_item(const WorkItemFunctionT & func) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| Launch the work-items for this work-group.

func is a function object type with a public member function `void F::operator()(h_item<Dimensions>)` representing the work-item computation.

This member function can only be invoked within a `parallel_for_work_group` context. It is undefined behavior for this member function to be invoked from within the `parallel_for_work_group` form that does not define work-group size, because then the number of work-items that should execute the code is not defined. It is expected that this form of `parallel_for_work_item` is invoked within the `parallel_for_work_group` form that specifies the size of a work-group.
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename WorkItemFunctionT&gt; void parallel_for_work_item(range&lt;Dimensions&gt; logicalRange, const WorkItemFunctionT&amp; func)</code></td>
<td>Launch the work-items for this work-group using a logical local range. The function object <code>func</code> is executed as if the kernel were invoked with <code>logicalRange</code> as the local range. This new local range is emulated and may not map one-to-one with the physical range. <code>logicalRange</code> is the new local range to be used. This range can be smaller or larger than the one used to invoke the kernel. <code>func</code> is a function object type with a public member function <code>void F::operator()(h_item&lt;Dimensions&gt;)</code> representing the work-item computation. Note that the logical range does not need to be uniform across all work-groups in a kernel. For example the logical range may depend on a work-group varying query (e.g. <code>group::get_linear_id</code>), such that different work-groups in the same kernel invocation execute different logical range sizes. This member function can only be invoked within a <code>parallel_for_work_group</code> context.</td>
</tr>
</tbody>
</table>

| template <typename DataT> device_event async_work_group_copy(decorated_local_ptr<DataT> dest, decorated_global_ptr<DataT> src, size_t numElements) | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |

| template <typename DataT> device_event async_work_group_copy(decorated_global_ptr<DataT> dest, decorated_local_ptr<DataT> src, size_t numElements) | Permitted types for `DataT` are all scalar and vector types. Asynchronously copies a number of elements specified by `numElements` from the source pointer `src` to destination pointer `dest` and returns a SYCL `device_event` which can be used to wait on the completion of the copy. |
### Member function

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;typename DataT&gt; device_event async_work_group_copy(decorated_local_ptr&lt;DataT&gt; dest, decorated_global_ptr&lt;DataT&gt; src, size_t numElements, size_t srcStride) const</td>
<td>Permitted types for DataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest with a source stride specified by srcStride and returns a SYCL device_event which can be used to wait on the completion of the copy.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;typename DataT&gt; device_event async_work_group_copy(decorated_global_ptr&lt;DataT&gt; dest, decorated_local_ptr&lt;DataT&gt; src, size_t numElements, size_t destStride) const</td>
<td>Permitted types for DataT are all scalar and vector types. Asynchronously copies a number of elements specified by numElements from the source pointer src to destination pointer dest with a destination stride specified by destStride and returns a SYCL device_event which can be used to wait on the completion of the copy.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template&lt;typename... EventTN&gt; void wait_for(EventTN... events) const</td>
<td>Permitted type for EventTN is device_event. Waits for the asynchronous operations associated with each device_event to complete.</td>
</tr>
</tbody>
</table>

### 4.9.1.8. sub_group class

The sub_group class encapsulates all functionality required to represent a particular sub_group within a parallel execution. It is not user-constructible.

The SYCL sub_group class provides the common by-value semantics (see Section 4.5.3).

A synopsis of the SYCL sub_group class is provided below. The member functions of the SYCL sub_group class are listed in Table 122. The additional common special member functions and common member functions are listed in Section 4.5.3 in Table 9 and Table 10 respectively.

```cpp
namespace sycl {
    class sub_group {
        public:
            using id_type = id<1>;
            using range_type = range<1>;
            using linear_id_type = uint32_t;
            static constexpr int dimensions = 1;
            static constexpr memory_scope fence_scope = memory_scope::sub_group;

            /* -- common interface members -- */

            id<1> get_group_id() const;
            id<1> get_local_id() const;
            range<1> get_local_range() const;
    }
}
```
Table 122. Member functions for the `sub_group` class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id&lt;1&gt; get_group_id() const</code></td>
<td>Return an id representing the index of the sub-group within the work-group. Since the work-items that compose a sub-group are chosen in an implementation defined way, the returned sub-group id cannot be used to identify a particular work-item in the global nd-range. Rather, the returned sub-group id is merely an abstract identifier of the sub-group containing this work-item.</td>
</tr>
<tr>
<td><code>id&lt;1&gt; get_local_id() const</code></td>
<td>Return a SYCL id representing the calling work-item’s position within the sub-group.</td>
</tr>
<tr>
<td><code>range&lt;1&gt; get_local_range() const</code></td>
<td>Return a range representing the size of the sub-group. This size may be less than the value returned by <code>get_max_local_range()</code>, depending on the position of the sub-group within its parent work-group and the manner in which sub-groups are constructed by the implementation.</td>
</tr>
<tr>
<td><code>range&lt;1&gt; get_group_range() const</code></td>
<td>Return a range representing the number of sub-groups in the work-group.</td>
</tr>
</tbody>
</table>
### 4.9.2. Reduction variables

All functionality related to reductions is captured by the `reducer` class and the `reduction` function.

The example below demonstrates how to write a reduction kernel that performs two reductions simultaneously on the same input values, computing both the sum of all values in a buffer and the maximum value in the buffer. For each reduction variable passed to `parallel_for`, a reference to a `reducer` object is passed as a parameter to the kernel function in the same order.

```cpp
1 buffer<int> valuesBuf { 1024 };  
2 {  
3   // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023  
4   host_accessor a { valuesBuf };  
5   std::iota(a.begin(), a.end(), 0);  
6 }  
7 8 // Buffers with just 1 element to get the reduction results  
9 int sumResult = 0;  
10 buffer<int> sumBuf { &sumResult, 1 };  
11 int maxResult = 0;  
12 buffer<int> maxBuf { &maxResult, 1 };  
```
myQueue.submit([8](handler& cgh) {
  // Input values to reductions are standard accessors
  auto inputValues = valuesBuf.get_access<access_mode::read>(cgh);

  // Create temporary objects describing variables with reduction semantics
  auto sumReduction = reduction(sumBuf, cgh, plus<>());
  auto maxReduction = reduction(maxBuf, cgh, maximum<>());

  // parallel_for performs two reduction operations
  // For each reduction variable, the implementation:
  // - Creates a corresponding reducer
  // - Passes a reference to the reducer to the lambda as a parameter
  cgh.parallel_for(range<1>({ 1024 }), sumReduction, maxReduction,
  [=](id<1> idx, auto& sum, auto& max) {
    // plus<>() corresponds to += operator, so sum can be
    // updated via += or combine()
    sum += inputValues[idx];

    // maximum<>() has no shorthand operator, so max can only
    // be updated via combine()
    max.combine(inputValues[idx]);
  });

  // sumBuf and maxBuf contain the reduction results once the kernel completes
  assert(maxBuf.get_host_access()[0] == 1023);  // sumBuf get_host_access()[0] == 523776; 

Reductions are supported for all trivially copyable types (as defined by the C++ core language). If the reduction operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic. If multiple reductions reference the same reduction variable, or a reduction variable is accessed directly during the lifetime of a reduction (e.g. via an accessor or USM pointer), the behavior is undefined.

Some of the overloads for the reduction function take an identity value and some do not. An implementation is required to compute a correct reduction even when the application does not specify an identity value. However, the implementation may be more efficient when the identity value is either provided by the application or is known by the implementation. For reductions using standard binary operators and fundamental types (e.g. plus and arithmetic types), an implementation can determine the correct identity value automatically in order to avoid performance penalties.

If an implementation can identify an identity value for a given combination of accumulator type and function object type, the value is defined as a member of the known_identity trait class. Whether this member value exists can be tested using the has_known_identity trait class.

```cpp
1 template <typename BinaryOperation, typename AccumulatorT>
2 struct known_identity {
3   static constexpr AccumulatorT value;
4 };
5
6 template <typename BinaryOperation, typename AccumulatorT>
7 inline constexpr AccumulatorT known_identity_v =
8   known_identity<BinaryOperation, AccumulatorT>::value;
```
template <typename BinaryOperation, typename AccumulatorT>
struct has_known_identity {
  static constexpr bool value;
};

template <typename BinaryOperation, typename AccumulatorT>
inline constexpr bool has_known_identity_v = has_known_identity<BinaryOperation, AccumulatorT>::value;

For each of the partial specializations listed in Table 123, known_identity exists and has the value shown.

Table 123. Known identities.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Available Only When</th>
<th>Identity</th>
</tr>
</thead>
<tbody>
<tr>
<td>sycl::plus</td>
<td>std::is_arithmetic_v&lt;AccumulatorT&gt;</td>
<td></td>
</tr>
<tr>
<td>sycl::multiplies</td>
<td>std::is_arithmetic_v&lt;AccumulatorT&gt;</td>
<td></td>
</tr>
<tr>
<td>sycl::bit_and</td>
<td>std::is_integral_v&lt;AccumulatorT&gt;</td>
<td>~AccumulatorT{ }</td>
</tr>
<tr>
<td>sycl::bit_or</td>
<td>std::is_integral_v&lt;AccumulatorT&gt;</td>
<td>AccumulatorT{}</td>
</tr>
<tr>
<td>sycl::bit_xor</td>
<td>std::is_integral_v&lt;AccumulatorT&gt;</td>
<td>AccumulatorT{}</td>
</tr>
<tr>
<td>sycl::logical_and</td>
<td>std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, bool&gt;</td>
<td>true</td>
</tr>
<tr>
<td>sycl::logical_or</td>
<td>std::is_same_v&lt;std::remove_cv_t&lt;AccumulatorT&gt;, bool&gt;</td>
<td>false</td>
</tr>
<tr>
<td>sycl::minimum</td>
<td>std::is_integral_v&lt;AccumulatorT&gt;</td>
<td>std::numeric_limits&lt;AccumulatorT&gt;::max()</td>
</tr>
<tr>
<td>sycl::minimum</td>
<td>std::is_floating_point_v&lt;AccumulatorT&gt;</td>
<td></td>
</tr>
</tbody>
</table>
The reduction interface is limited to reduction variables whose size can be determined at compile-time. As such, buffer and USM pointer arguments are interpreted by the reduction interface as describing a single variable. A reduction operation associated with a span represents an array reduction. An array reduction of size $N$ is functionally equivalent to specifying $N$ independent scalar reductions. The combination operations performed by an array reduction are limited to the extent of a USM allocation described by a span, and access to elements outside of these regions results in undefined behavior.

Since a span is one-dimensional, there is currently no way to describe an array reduction with more than one dimension. This is expected to change in a future version of the SYCL specification, but depends on the introduction of a multi-dimensional span.

### 4.9.2.1. reduction interface

The reduction interface is used to attach reduction semantics to a variable, by specifying: the reduction variable, the reduction operator and an optional identity value associated with the operator. The overloads of the interface are described in Table 124. The return value of the reduction interface is an implementation-defined object of unspecified type, which is interpreted by parallel_for to construct an appropriate reducer type as detailed in Section 4.9.2.3.

An implementation may use an unspecified number of temporary variables inside of any reducer objects it creates. If an identity value is supplied to a reduction, an implementation will use that value to initialize any such temporary variables.

Since the number of temporary variables is unspecified, supplying an identity value different to the identity value associated with the reduction operator may lead to unexpected results.

The initial value of the reduction variable is included in the reduction operation, unless the property::reduction::initialize_to_identity property was specified when the reduction interface was invoked.

The reduction variable is updated so as to contain the result of the reduction when the kernel finishes execution.

```cpp
template <typename BufferT, typename BinaryOperation>
__unspecified__ reduction(BufferT vars, handler& cgh, BinaryOperation combiner,
const property_list& propList = {});

template <typename T, typename BinaryOperation>
__unspecified__ reduction(T* var, BinaryOperation combiner,
const property_list& propList = {});
```
template <typename T, typename Extent, typename BinaryOperation>
__unspecified__ reduction(span<T, Extent> vars, BinaryOperation combiner, const property_list& propList = {});

template <typename BufferT, typename BinaryOperation>
__unspecified__ reduction(BufferT vars, handler& cgh, const BufferT::value_type& identity, BinaryOperation combiner, const property_list& propList = {});

template <typename T, typename BinaryOperation>
__unspecified__ reduction(T* var, const T& identity, BinaryOperation combiner, const property_list& propList = {});

template <typename T, typename Extent, typename BinaryOperation>
__unspecified__ reduction(span<T, Extent> vars, const T& identity, BinaryOperation combiner, const property_list& propList = {});

Table 124. Overloads of the reduction interface

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduction(BufferT, BinaryOperation&gt;(BufferT vars, handler&amp; cgh, BinaryOperation combiner, const property_list&amp; propList = {})</td>
<td>Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code>. Zero or more properties can be provided via an instance of <code>property_list</code>. Throws an exception with the <code>errc::invalid</code> error code if the range of the <code>vars</code> buffer is not 1.</td>
</tr>
<tr>
<td>reduction(T, BinaryOperation&gt;(T* var, BinaryOperation combiner, const property_list&amp; propList = {})</td>
<td>Construct an unspecified object representing a reduction of the variable described by <code>var</code> using the combination operation specified by <code>combiner</code>. Zero or more properties can be provided via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td>reduction(T, BinaryOperation&gt;(span&lt;T, Extent&gt; vars, BinaryOperation combiner, const property_list&amp; propList = {})</td>
<td>Available only when <code>Extent != sycl::dynamic_extent</code>. Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code>. Zero or more properties can be provided via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>reduction&lt;BufferT, BinaryOperation&gt;(BufferT vars, handler&amp; cgh, std::function&lt;const BufferT::value_type&amp; identity, BinaryOperation combiner&gt;, const property_list&amp; propList = {})</code></td>
<td>Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code>. The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code>. Throws an exception with the <code>errc::invalid</code> error code if the range of the <code>vars</code> buffer is not 1.</td>
</tr>
<tr>
<td><code>reduction&lt;T, BinaryOperation&gt;(T* var, const T&amp; identity, BinaryOperation combiner, const property_list&amp; propList = {})</code></td>
<td>Construct an unspecified object representing a reduction of the variable described by <code>var</code> using the combination operation specified by <code>combiner</code>. The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code>.</td>
</tr>
<tr>
<td><code>reduction&lt;T, BinaryOperation&gt;(std::span&lt;T, Extent&gt; vars, const T&amp; identity, BinaryOperation combiner, const property_list&amp; propList = {})</code></td>
<td>Available only when <code>Extent != sycl::dynamic_extent</code>. Construct an unspecified object representing a reduction of the variable(s) described by <code>vars</code> using the combination operation specified by <code>combiner</code>. The value of <code>identity</code> may be used by the implementation to initialize an unspecified number of temporary accumulation variables. Zero or more properties can be provided via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>

### 4.9.2.2. Reduction properties

The properties that can be provided when using the `reduction` interface are described in Table 125.

*Table 125. Properties supported by the `reduction` interface*
The initialize_to_identity property adds the requirement that the SYCL runtime must initialize the reduction variable to the identity value passed to the reduction interface, or to the identity value determined by the known_identity trait if no identity value was specified. If no identity value was specified and an identity value cannot be determined by the known_identity trait, the compiler must raise a diagnostic. When this property is set, the original value of the reduction variable is not included in the reduction.

The constructors of the reduction property classes are listed in Table 126.

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>property::reduction::initialize_to_identity()</code></td>
<td>Constructs an <code>initialize_to_identity</code> property instance.</td>
</tr>
</tbody>
</table>

### 4.9.2.3. reducer class

The reducer class defines the interface between a work-item and a reduction variable during the execution of a SYCL kernel, restricting access to the underlying reduction variable. The intermediate values of a reduction variable cannot be inspected during kernel execution, and the variable cannot be updated using anything other than the reduction's specified combination operation. The combination order of different reducers is unspecified, as are when and how the value of each reducer is combined with the original reduction variable.

To enable compile-time specialization of reduction algorithms, the implementation of the reducer class is unspecified, except for the functions and operators defined in Table 128 and Table 129. As such, developers should not specify the template arguments of a reducer directly, and should instead employ generic programming techniques that allow kernel functions to accept a reference to a variable of any reducer type. Kernels written as lambdas should employ `auto&` or `auto&...`, and kernels written as function objects should employ template parameters or template parameter packs.

An implementation must guarantee that it is safe for multiple work-items in a kernel to call the combine function of a reducer concurrently. An implementation is free to re-use reducer variables (e.g. across work-groups scheduled to the same compute unit) if it can guarantee that it is safe to do so.

The type aliases and constant static members of the reducer class are listed in Table 127 and its member functions are listed in Table 128. Additional shorthand operators may be made available for certain combinations of reduction variable type and combination operation, as described in Table 129.

```cpp
// Exposition only
template <typename T, typename BinaryOperation, int Dimensions,
           /* unspecified */>
class reducer {
```
public:
  using value_type = T;
  using binary_operation = BinaryOperation;
  static constexpr int dimensions = Dimensions;

  reducer(const reducer&) = delete;
  reducer(reducer&&) = delete;
  reducer& operator=(const reducer&) = delete;
  reducer& operator=(reducer&&) = delete;

  ~reducer();

  /* Only available if Dimensions == 0 */
  reducer& combine(const T& partial);

  /* Only available if Dimensions > 0 */
  __unspecified__ operator[](size_t index)

  /* Only available if identity value is known */
  T identity() const;

  /* Only available if Dimensions == 0 and either
   * BinaryOperation == plus<> or BinaryOperation == plus<T> */
  friend reducer& operator+=(reducer&, const T&)

  /* Only available if Dimensions == 0 and either
   * BinaryOperation == multiplies<> or BinaryOperation == multiplies<T> */
  friend reducer& operator*=(reducer&, const T&)

  /* Only available if Dimensions == 0, T is an integral type and either
   * BinaryOperation == bit_and<> or BinaryOperation == bit_and<T> */
  friend reducer& operator&=(reducer&, const T&)

  /* Only available if Dimensions == 0, T is an integral type and either
   * BinaryOperation == bit_or<> or BinaryOperation == bit_or<T> */
  friend reducer& operator|=(reducer&, const T&)

  /* Only available if Dimensions == 0, T is an integral type, T is not bool and
   * either BinaryOperation == plus<> or BinaryOperation == plus<T> */
  friend reducer& operator++(reducer&)

};

Table 127. Member types and constants of the reducer class
### Table 128. Member functions of the reducer class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reducer&amp; combine(const T&amp; partial)</code></td>
<td>Available only when: <code>Dimensions == 0</code>. Combine the value of <code>partial</code> with the reduction variable associated with this <code>reducer</code>. Returns <code>*this</code>.</td>
</tr>
<tr>
<td><code>__unspecified__ operator[](size_t index)</code></td>
<td>Available only when: <code>Dimensions &gt; 0</code>. Returns an instance of an undefined intermediate type representing a <code>reducer</code> of the same type as this <code>reducer</code>, with the dimensionality <code>Dimensions-1</code> and containing an implicit SYCL <code>id</code> with index <code>Dimensions</code> set to <code>index</code>. The intermediate type returned must provide all member functions and operators defined by the <code>reducer</code> class that are appropriate for the type it represents (including this subscript operator).</td>
</tr>
<tr>
<td><code>T identity() const</code></td>
<td>Return the identity value of the combination operation associated with this <code>reducer</code>. Only available if the identity value is known to the implementation.</td>
</tr>
</tbody>
</table>

### Table 129. Hidden friend operators of the reducer class

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reducer&amp; operator+=(reducer&amp; accum, const T&amp; partial)</code></td>
<td>Equivalent to calling <code>accum.combine(partial)</code>. Available only when: <code>Dimensions == 0</code> and `(std::is_same_v&lt;BinaryOperation, plus&lt;&gt;&gt;</td>
</tr>
<tr>
<td>Operator</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>&amp;operator*=</code> (<code>&amp;operator*=</code>) (reducer&amp; accum, const T&amp; partial)</td>
<td>Equivalent to calling <code>accum.combine(partial)</code>. Available only when: <code>Dimensions == 0</code> &amp; `std::is_same_v&lt;BinaryOperation, multiplies&lt;&gt;&gt;</td>
</tr>
<tr>
<td><code>&amp;operator&amp;=</code> (<code>&amp;operator&amp;=</code>) (reducer&amp; accum, const T&amp; partial)</td>
<td>Equivalent to calling <code>accum.combine(partial)</code>. Available only when: <code>Dimensions == 0</code> &amp; <code>is_integral_v&lt;T&gt;</code> &amp; `(std::is_same_v&lt;BinaryOperation, bit_and&lt;&gt;&gt;</td>
</tr>
<tr>
<td>`&amp;operator</td>
<td>=<code> (</code>&amp;operator</td>
</tr>
<tr>
<td><code>&amp;operator^=</code> (<code>&amp;operator^=</code>) (reducer&amp; accum, const T&amp; partial)</td>
<td>Equivalent to calling <code>accum.combine(partial)</code>. Available only when: <code>Dimensions == 0</code> &amp; <code>is_integral_v&lt;T&gt;</code> &amp; `(std::is_same_v&lt;BinaryOperation, bit_xor&lt;&gt;&gt;</td>
</tr>
<tr>
<td><code>&amp;operator++(reducer&amp; accum)</code></td>
<td>Equivalent to calling <code>accum.combine(1)</code>. Available only when: <code>Dimensions == 0</code> &amp; `std::is_integral_v&lt;T&gt; &amp;&amp; !std::is_same_v&lt;T, bool&gt; &amp;&amp; std::is_same_v&lt;BinaryOperation, plus&lt;&gt;&gt;</td>
</tr>
</tbody>
</table>

### 4.9.3. Command group scope

A command group scope, as defined in Section 3.7.1, may execute a single command such as invoking a kernel, copying memory, or executing a host task. It is legal for a command group scope to statically contain more than one call to a command function, but any single execution of the command group function object may execute no more than one command. If an application fails to do this, the function that submits the command group function object (i.e., `queue::submit`) must throw a synchronous exception with the `errc::invalid` error code. The statements that call commands together with the statements that define the requirements for a kernel form the command group function object. The command group function object takes as a parameter an instance of the command group handler class which encapsulates all the member functions executed in the command group scope. The member functions and objects defined in this scope will define the requirements for the kernel execution or explicit memory operation, and will be used by the SYCL runtime to evaluate if the operation is ready for execution. Host code within a command group function object (typically setting up requirements) is executed once, before the command group submit call returns. This abstraction of the kernel execution unifies the data with its processing, and consequently allows more abstraction and flexibility in the parallel program-
The command group function object and the handler class serve as an interface for the encapsulation of command group scope. A SYCL kernel function is defined as a function object. All the device data accesses are defined inside this group and any transfers are managed by the SYCL runtime. The rules for the data transfers regarding device and host data accesses are better described in Section 4.7, where buffers (Section 4.7.2) and accessor (Section 4.7.6) classes are described. The overall memory model of the SYCL application is described in Section 3.8.1.

It is possible for a command group function object to fail to enqueue to a queue, or for it to fail to execute correctly. A user can therefore supply a secondary queue when submitting a command group to the primary queue. If the SYCL runtime fails to enqueue or execute a command group on a primary queue, it can attempt to run the command group on the secondary queue. The circumstances in which it is, or is not, possible for a SYCL runtime to fall-back from primary to secondary queue are unspecified in the specification. Even if a command group is run on the secondary queue, the requirement that host code within the command group is executed exactly once remains, regardless of whether the fallback queue is used for execution.

The command group handler class provides the interface for all of the member functions that are able to be executed inside the command group scope, and it is also provided as a scoped object to all of the data access requests. The command group handler class provides the interface in which every command in the command group scope will be submitted to a queue.

### 4.9.4. Command group handler class

A command group handler object can only be constructed by the SYCL runtime. All of the accessors defined in command group scope take as a parameter an instance of the command group handler, and all the kernel invocation functions are member functions of this class.

The constructors of the SYCL handler class are described in Table 130.

It is disallowed for an instance of the SYCL handler class to be moved or copied.

```cpp
namespace sycl {

class handler {

private:

// implementation defined constructor
handler(__unspecified__);

public:

// template <typename DataT, int Dimensions, access_mode AccessMode,
// target AccessTarget, access::placeholder IsPlaceholder>
void require<
    accessor<DataT, Dimensions, AccessMode, AccessTarget, IsPlaceholder> acc);  

// template <typename T> void set_arg(int argIndex, T&& arg);

// template <typename... Ts> void set_args(Ts&&... args);  

} //----- Kernel dispatch API
```

Chapter 4. SYCL programming interface | 281
Note: In all kernel dispatch functions, the template parameter "typename KernelName" is optional.

```cpp
template <typename KernelName, typename KernelType>
void single_task(const KernelType& kernelFunc);

// Parameter pack acts as-if: Reductions... reductions, const KernelType
template <typename KernelName, int Dimensions, typename... Rest>
void parallel_for(range<Dimensions> numWorkItems, Rest&&... rest);

// Deprecated in SYCL 2020.
template <typename KernelName, typename KernelType, int Dimensions>
void parallel_for(range<Dimensions> numWorkItems,
                    id<Dimensions> workItemOffset,
                    const KernelType& kernelFunc);

// Parameter pack acts as-if: Reductions... reductions, const KernelType
template <typename KernelName, int Dimensions, typename... Rest>
void parallel_for(nd_range<Dimensions> executionRange, Rest&&... rest);

template <typename KernelName, typename WorkgroupFunctionType, int Dimensions>
void parallel_for_work_group(range<Dimensions> numWorkGroups,
                               const WorkgroupFunctionType& kernelFunc);

template <typename KernelName, typename WorkgroupFunctionType, int Dimensions>
void parallel_for_work_group(range<Dimensions> numWorkGroups,
                               range<Dimensions> workGroupSize,
                               const WorkgroupFunctionType& kernelFunc);

void single_task(const kernel& kernelObject);

template <int Dimensions>
void parallel_for(range<Dimensions> numWorkItems, const kernel& kernelObject);

template <int Dimensions>
void parallel_for(nd_range<Dimensions> ndRange, const kernel& kernelObject);

//------ USM functions

void memcpy(void* dest, const void* src, size_t numBytes);

template <typename T> void copy(const T* src, T* dest, size_t count);

void memset(void* ptr, int value, size_t numBytes);

template <typename T> void fill(void* ptr, const T& pattern, size_t count);

void prefetch(void* ptr, size_t numBytes);

void mem_advise(void* ptr, size_t numBytes, int advice);

//------ Explicit memory operation APIs
```
Table 130. Constructors of the handler class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>handler(___unspecified___)</code></td>
<td>Unspecified implementation-defined constructor.</td>
</tr>
</tbody>
</table>
4.9.4.1. SYCL functions for adding requirements

When an accessor is created from a command group handler, a requirement is implicitly added to the command group for the accessor's data. However, this does not happen when creating a placeholder accessor. In order to create a requirement for a placeholder accessor, code must call the handler::require() member function.

SYCL events may also be used to create requirements for a command group. Such requirements state that the actions represented by the events must complete before the command group may execute. Such requirements are added when code calls the handler::depends_on() member function.

Table 131. Member functions of the handler class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, access::placeholder IsPlaceholder&gt; void require(accessor&lt;DataT, Dimensions, AccessMode, AccessTarget, IsPlaceholder&gt; acc)</td>
<td>Requires access to the memory object associated with the accessor. The command group now has a requirement to gain access to the given memory object before executing the kernel. If the accessor has already been registered with the command group, calling this function has no effect. Throws exception with the errc::invalid error code if (acc.empty() == true).</td>
</tr>
<tr>
<td>void depends_on(event depEvent)</td>
<td>The command group now has a requirement that the action represented by depEvent must complete before executing this command-group's action.</td>
</tr>
<tr>
<td>void depends_on(const std::vector&lt;event&gt;&amp; depEvents)</td>
<td>The command group now has a requirement that the actions represented by each event in depEvents must complete before executing this command-group's action.</td>
</tr>
</tbody>
</table>

4.9.4.2. SYCL functions for invoking kernels

Kernels can be invoked as single tasks, basic data-parallel kernels, nd-range in work groups, or hierarchical parallelism.

Each function takes an optional kernel name template parameter. The user may optionally provide a kernel name, otherwise an implementation-defined name will be generated for the kernel.

All the functions for invoking kernels are member functions of the command group handler class (Section 4.9.4), which is used to encapsulate all the member functions provided in a command group scope. Table 132 lists all the members of the handler class related to the kernel invocation.

Table 132. Member functions of the handler class
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
</table>
| ```
template <typename T> void set_arg(int argIndex, T& arg)
``` |
This function must only be used to set arguments for a kernel that was constructed using a backend specific interoperability function or for a device built-in kernel. Attempting to use this function to set arguments for other kernels results in undefined behavior. The precise semantics of this function are defined by each SYCL backend specification. |
| ```
template <typename... Ts> void set_args(Ts&... args)
``` |
Set all arguments for a given kernel, as if each argument in `args` was passed to `set_arg` in the same order and with an increasing index starting at 0. |
| ```
template <typename KernelName, typename KernelType>
void single_task(const KernelType& kernelFunc)
``` |
Defines and invokes a SYCL kernel function as a lambda function or a named function object type. Specification of a kernel name (typename `KernelName`), as described in Section 4.9.4.2, is optional. The callable `KernelType` can optionally take a kernel_handler in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to `KernelType`. |
### Member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>

Template function as a lambda function or a named function object type, for the specified range and given an item or integral type (e.g. `int`, `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an item. Specification of a *kernel name* (`typename KernelName`), as described in Section 4.9.4.2, is optional. The `rest` parameter pack consists of 0 or more objects created by the `reduction` function, followed by a callable. For each object in `rest`, the kernel function must take an additional reference parameter corresponding to that object's `reducer` type, in the same order. The callable can optionally take a `kernel_handler` as its last parameter, in which case the *SYCL runtime* will construct an instance of `kernel_handler` and pass it to the callable.

```c++
template <typename KernelName, int Dimensions, typename...
Rest>
void parallel_for(range<Dimensions> numWorkItems, Rest&&...
rest)
```
### Member function

```
template <typename KernelName, int Dimensions, typename...
Rest>
void parallel_for(range<Dimensions> numWorkItems, id
<Dimensions> workItemOffset,
const KernelType& kernelFunc)
  // Deprecated in SYCL 2020.
```

### Description

Deprecated in SYCL 2020. Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and offset and given an item or integral type (e.g. `int`, `size_t`), if range is 1-dimensional, for indexing in the indexing space defined by range. Generic kernel functions are permitted, in that case the argument type is an item. Specification of a kernel name (`typename KernelName`), as described in Section 4.9.4.2, is optional. The rest parameter pack consists of 0 or more objects created by the reduction function, followed by a callable. For each object in `rest`, the kernel function must take an additional reference parameter corresponding to that object's reducer type, in the same order. The callable can optionally take a `kernel_handler` as its last parameter, in which case the SYCL runtime will construct an instance of `kernel_handler` and pass it to the callable.
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
</table>
| \[
\text{template } \langle \text{typename } \text{KernelName}, \text{int } \text{Dimensions}, \text{typename ... Rest}\rangle
\text{void parallel_for}(\text{nd_range<Dimensions>} \text{executionRange}, \text{Rest... rest})
\]
| Defines and invokes a **SYCL kernel function** as a lambda function or a named function object type, for the specified **nd-range** and given an **nd-item** for indexing in the indexing space defined by the **nd-range**. Generic kernel functions are permitted, in that case the argument type is an **nd-item**. Specification of a kernel name (**typename KernelName**), as described in Section 4.9.4.2., is optional. The **rest** parameter pack consists of 0 or more objects created by the **reduction** function, followed by a callable. For each object in **rest**, the kernel function must take an additional reference parameter corresponding to that object’s **reducer** type, in the same order. The callable can optionally take a **kernel_handler** as its last parameter, in which case the SYCL runtime will construct an instance of **kernel_handler** and pass it to the callable. Throws an **exception** with the **errc::nd_range** error code if the global size defined in the associated **executionRange** defines a non-zero index space which is not evenly divisible by the local size in each dimension. |
| \[
\text{template } \langle \text{typename } \text{KernelName}, \text{typename WorkgroupFunctionType}, \text{int } \text{Dimensions}\rangle
\text{void parallel_for_work_group}(\text{range<Dimensions>} \text{numWorkGroups}, \text{const WorkgroupFunctionType& kernelFunc})
\]
| Defines and invokes a hierarchical kernel as a lambda function or a named function object type, encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a **group**. May contain multiple calls to **parallel_for_work_item(..)** member functions representing the execution on each work-item. Launches **num_work_groups** work-groups of runtime-defined size. Described in detail in Section 4.9.4.2. The callable **WorkgroupFunctionType** can optionally take a **kernel_handler** as it’s last parameter, in which case the SYCL runtime will construct an instance of **kernel_handler** and pass it to **WorkgroupFunctionType**. |
### Member function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>template &lt;typename KernelName, typename WorkgroupFunctionType, int Dimensions&gt;</strong>&lt;br&gt;<code>void parallel_for_work_group(range&lt;Dimensions&gt; numWorkGroups, range&lt;Dimensions&gt; workGroupSize, const WorkgroupFunctionType&amp; kernelFunc)</code></td>
<td>Defines and invokes a hierarchical kernel as a lambda function or a named function object type, encoding the body of each work-group to launch. Generic kernel functions are permitted, in that case the argument type is a <code>group</code>. May contain multiple calls to <code>parallel_for_work_item</code> member functions representing the execution on each work-item. Launches <code>num_work_groups</code> work-groups of <code>work_group_size</code> work-items each. Described in detail in Section 4.9.4.2. The callable <code>WorkgroupFunctionType</code> can optionally take a <code>kernel_handler</code> as it's last parameter, in which case the SYCL runtime will construct an instance of <code>kernel_handler</code> and pass it to <code>WorkgroupFunctionType</code>.</td>
</tr>
<tr>
<td><strong>void single_task(const kernel&amp; kernelObject)</strong></td>
<td>This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous exception with the <code>errc::invalid</code> error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should execute exactly once. This invocation function ignores any <code>kernel_bundle</code> that was bound to this command group handler via <code>handler::use_kernel_bundle()</code> and instead implicitly uses the kernel bundle that contains the <code>kernelObject</code>. Throws an exception with the <code>errc::kernel_not_supported</code> error code if the <code>kernelObject</code> is not compatible with either the device associated with the primary queue of the <code>command group</code> or with the device associated with the secondary queue (if specified).</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| template <int Dimensions>
void parallel_for(range<Dimensions> numWorkItems, const kernel& kernelObject) | This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous exception with the errc::invalid error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should be invoked for the specified range of index values. This invocation function ignores any kernel_bundle that was bound to this command group handler via handler::use_kernel_bundle() and instead implicitly uses the kernel bundle that contains the kernelObject. Throws an exception with the errc::kernel_not_supported error code if the kernelObject is not compatible with either the device associated with the primary queue of the command group or with the device associated with the secondary queue (if specified). |
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
</table>
| `template <int Dimensions>
void parallel_for(nd_range<Dimensions> executionRange,
    const kernel& kernelObject)` | This function must only be used to invoke a kernel that was constructed using a backend specific interoperability function or to invoke a device built-in kernel. Attempting to use this function to invoke other kernels throws a synchronous exception with the `errc::invalid` error code. The precise semantics of this function are defined by each SYCL backend specification, but the intent is that the kernel should be invoked for the specified `executionRange`. Throws an exception with the `errc::nd_range` error code if the global size defined in the associated `executionRange` defines a non-zero index space which is not evenly divisible by the local size in each dimension. This invocation function ignores any `kernel_bundle` that was bound to this command group handler via `handler::use_kernel_bundle()` and instead implicitly uses the kernel bundle that contains the `kernelObject`. Throws an exception with the `errc::kernel_not_supported` error code if the `kernelObject` is not compatible with either the device associated with the primary queue of the `command group` or with the device associated with the secondary queue (if specified). |

### 4.9.4.2.1. `single_task` invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on a device. Only one instance of the kernel will be executed. This interface is useful as a primitive for more complicated parallel algorithms, as it can easily create a chain of sequential tasks on a SYCL device with each of them managing its own data transfers.

This function can only be called inside a command group using the `handler` object created by the runtime. Any accessors that are used in a kernel should be defined inside the same command group.

Local accessors are disallowed for single task invocations.

```cpp
myQueue.submit([](handler& cgh) {
    cgh.single_task()
        [=] () {
        // [kernel code]
    }
});
```
For single tasks, the kernel member function takes no parameters, as there is no need for index space classes in a unary index space.

A kernel_handler can optionally be passed as a parameter to the SYCL kernel function that is invoked by single_task for the purpose explained in Section 4.9.5.3.

```
myQueue.submit([8](handler8 cgh) {
  cgh.single_task(
    [=] (kernel_handler kh) {
      // [kernel code]
    });
});
```

### 4.9.4.2.2. parallel_for invoke

The parallel_for member function of the SYCL handler class provides an interface to define and invoke a SYCL kernel function in a command group, to execute in parallel execution over a 3 dimensional index space. There are three overloads of the parallel_for member function which provide variations of this interface, each with a different level of complexity and providing a different set of features.

For the simplest case, users need only provide the global range (the total number of work-items in the index space) via a SYCL range parameter. In this case the function object that represents the SYCL kernel function must take one of: 1) a single SYCL item parameter, 2) a single generic parameter (template parameter or auto) that will be treated as an item parameter, 3) any other type implicitly converted from SYCL item, representing the currently executing work-item within the range specified by the range parameter.

Case 3) above allows the kernel function to take an argument of type id because item is implicitly convertible to id. It also allows a 1-D kernel function to take an integral argument (e.g. int or size_t) because a 1-D item is implicitly convertible to these types. Finally, it allows the kernel function to take a user-defined argument type that can be constructed from item, enabling users to layer their own abstractions on top of SYCL.

The execution of the kernel function is the same whether the parameter to the SYCL kernel function is a SYCL id or a SYCL item. What differs is the functionality that is available to the SYCL kernel function via the respective interfaces.

Below is an example of invoking a SYCL kernel function with parallel_for using a lambda function and passing a SYCL id parameter. In this case, only the global id is available. This variant of parallel_for is designed for when it is not necessary to query the global range of the index space being executed across.

```
myQueue.submit([8](handler8 cgh) {
  accessor acc { myBuffer, cgh, write_only };
  cgh.parallel_for(range<1>(numWorkItems),
                 [=](id<1> index) { acc[index] = 42.0f; });
});
```

Below is an example of invoking a SYCL kernel function with parallel_for using a lambda function and passing a SYCL item parameter. In this case, both the global id and global range are queryable. This variant of parallel_for is designed for when it is necessary to query the global range of the index space.
being executed across.

```cpp
1 myQueue.submit([&](handler & cgh) {
2  accessor acc { myBuffer, cgh, write_only; }
3
4  cgh.parallel_for(range<1>(numWorkItems), [=](item<1> item) {
5    // kernel argument type is item
6    size_t index = item.get_linear_id();
7    acc[index] = index;
8  });
9 });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda function and passing `auto` parameter, treated as `item`. In this case, both the global id and global range are queryable. The same effect can be achieved using class with templateized `operator()`. This variant of `parallel_for` is designed for when it is necessary to query the global range within which the global id will vary.

```cpp
1 myQueue.submit([&](handler & cgh) {
2  auto acc = myBuffer.get_access<access_mode::write>(cgh);
3
4  cgh.parallel_for(range<1>(numWorkItems), [=](auto item) {
5    // kernel argument type is auto treated as an item
6    size_t index = item.get_linear_id();
7    acc[index] = index;
8  });
9 });
```

Below is an example of invoking a SYCL kernel function with `parallel_for` using a lambda function and passing an integral type parameter. This example is only valid when calling `parallel_for` with `range<1>`. In this case only the global id is available. This variant of `parallel_for` is designed for when it is not necessary to query the global range of the index space being executed across.

```cpp
1 myQueue.submit([&](handler & cgh) {
2  auto acc = myBuffer.get_access<access_mode::write>(cgh);
3
4  cgh.parallel_for(range<1>(numWorkItems), [=](size_t index) {
5    // kernel argument type is size_t
6    acc[index] = index;
7  });
8 });
```

The `parallel_for` overload without an offset can be called with either a number or a `braced-init-list` with 1-3 elements. In that case the following calls are equivalent:

- `parallel_for(N, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for(N, some_kernel)` has same effect as `parallel_for(range<1>(N), some_kernel)`
- `parallel_for((N1, N2), some_kernel)` has same effect as `parallel_for(range<2>(N1, N2), some_kernel)`
- `parallel_for((N1, N2, N3), some_kernel)` has same effect as `parallel_for(range<3>(N1, N2, N3), some_kernel)`

Below is an example of invoking `parallel_for` with a number instead of an explicit `range` object.
For SYCL kernel functions invoked via the above described overload of the \texttt{parallel\_for} member function, it is disallowed to use local accessors or to use a \texttt{work\-group\_barrier}.

The following two examples show how a kernel function object can be launched over a 3D grid, with 3 elements in each dimension. In the first case work-item ids range from 0 to 2 inclusive, and in the second case work-item ids run from 1 to 3.

The last case of a \texttt{parallel\_for} invocation enables low-level functionality of work-items and work-groups. This becomes valuable when an execution requires groups of work-items that communicate and synchronize. These are exposed in SYCL through \texttt{parallel\_for(nd\_range,...)} and the \texttt{nd\_item} class. In this case, the developer needs to define the \texttt{nd\_range} that the kernel will execute on in order to have fine grained control of the enqueueing of the kernel. This variation of \texttt{parallel\_for} expects an \texttt{nd\_range}, specifying both local and global ranges, defining the global number of work-items and the number in each cooperating work-group. The function object that represents the SYCL kernel function must take one of:

1) a single SYCL \texttt{nd\_item} parameter,
2) a single generic parameter (\texttt{template} parameter or \texttt{auto}) that will be treated as an \texttt{nd\_item} parameter,
3) any other type converted from SYCL \texttt{nd\_item}, representing the currently executing work-item within the range specified by the \texttt{nd\_range} parameter. The \texttt{nd\_item} parameter makes all information about the work-item and its position in the range available, and provides access to functions enabling the use of a \texttt{work\-group\_barrier} to synchronize between the work-items in the work-group.

Case 3) above includes user-defined types that can be constructed from \texttt{nd\_item}, enabling users to layer their own abstractions on top of SYCL.

The following example shows how sixty-four work-items may be launched in a three-dimensional grid with four in each dimension, and divided into eight work-groups. Each group of work-items synchronizes with a \texttt{work\-group\_barrier}.
In all of these cases the underlying \texttt{nd-range} will be created and the kernel defined as a function object will be created and enqueued as part of the command group scope.

Some forms of \texttt{parallel_for} accept an offset parameter of type \texttt{id<Dimensions>}, where the number of dimensions of the \texttt{id} is the same as the number of dimensions of the \texttt{range} that determines the iteration space. These forms of \texttt{parallel_for} execute the same number of iterations as the form with no offset. The difference is that the \texttt{id} or \texttt{item} parameter passed to the kernel function has the value of \texttt{offset} implicitly added. This offset parameter is deprecated in SYCL 2020.

An offset can also be passed to the forms of \texttt{parallel_for} that accept an \texttt{nd_range} via the third parameter to the \texttt{nd_range} constructor. These forms of \texttt{parallel_for} also execute the same number of iterations as if no offset was specified. The difference is that the \texttt{nd_item} parameter passed to the kernel function has the value of the offset implicitly added to the constituent \texttt{global id}. This offset parameter is deprecated in SYCL 2020.

A \texttt{kernel_handler} can optionally be passed as a parameter to the SYCL kernel function that is invoked by both variants of \texttt{parallel_for}.

4.9.4.2.3. Parallel for hierarchical invoke

The hierarchical parallel kernel execution interface provides the same functionality as is available from the \texttt{nd-range} interface, but exposed differently. To execute the same sixty-four work-items in sixteen work-groups that we saw in the previous example, we execute an outer \texttt{parallel_for_work_group} call to create the groups. The member function \texttt{handler::parallel_for_work_group} is parameterized by the number of work-groups, such that the size of each group is chosen by the runtime, or by the number of work-groups and number of work-items for users who need more control.

The body of the outer \texttt{parallel_for_work_group} call consists of a lambda function or function object. The
body of this function object contains code that is executed only once for the entire work-group. If the code has no side-effects and the compiler heuristic suggests that it is more efficient to do so, this code will be executed for each work-item.

Within this region any variable declared will have the semantics of local memory, shared between all work items in the work-group. If the device compiler can prove that an array of such variables is accessed only by a single work-item throughout the lifetime of the work-group, for example if access is derived from the id of the work-item with no transformation, then it can allocate the data in private memory or registers instead.

To guarantee use of private per-work-item memory, the private_memory class can be used to wrap the data. This class simply constructs private data for a given group across the entire group. The id of the current work-item is passed to any access to grab the correct data.

The private_memory class has the following interface:

Listing 1. Private memory class

```
namespace sycl {
    template <typename T, int Dimensions = 1> class private_memory {
    public:
        private_memory(const group<Dimensions>&); // Construct based directly off the number of work-items
        T& operator()(const h_item<Dimensions>& id); // Access the instance for the current work-item
    }; // namespace sycl
```

Table 133. Constructor of the private_memory class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>private_memory(const group&lt;Dimensions&gt;&amp;)</td>
<td>Place an object of type T in the underlying private memory of each work item. The type T must be default constructible. The underlying constructor will be called for each work-item.</td>
</tr>
</tbody>
</table>

Table 134. Member functions of the private_memory class

<table>
<thead>
<tr>
<th>Member functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp; operator()(const h_item&lt;Dimensions&gt;&amp; id)</td>
<td>Retrieve a reference to the object for the work items.</td>
</tr>
</tbody>
</table>

Private memory is allocated per underlying work-item, not per iteration of the parallel_for_work_item loop. The number of instances of a private memory object is only under direct control if a work-group size is passed to the parallel_for_work_group call. If the underlying work-group size is chosen by the runtime, the number of private memory instances is opaque to the program. Explicit private memory declarations should therefore be used with care and with a full understanding of which instances of a parallel_for_work_item loop will share the same underlying variable.

Also within the lambda body can be a sequence of calls to parallel_for_work_item. At the edges of these inner parallel executions the work-group synchronizes. As a result the pair of parallel_for_work_item calls in the code below is equivalent to the parallel execution with a work-group barrier in the earlier example.
myQueue.submit([handler & cgh] {
  cgh.parallel_for_work_group(
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {
        int myLocal; // this variable is shared between workitems
        // this variable will be instantiated for each work-item separately
        private_memory<int> myPrivate(myGroup);

        // Issue parallel work-items. The number issued per work-group is
        // determined by the work-group size range of parallel_for_work_group.
        // In this case, 8 work-items will execute the parallel_for_work_item
        // body for each of the 8 work-groups, resulting in 64 executions
        // globally/total.
        myGroup.parallel_for_work_item([=](h_item<3> myItem) {
          // [work-item code]
          myPrivate(myItem) = 0;
        });

        // Implicit work-group barrier

        // Carry private value across loops
        myGroup.parallel_for_work_item([=](h_item<3> myItem) {
          // [work-item code]
          output[myItem.get_global_id()] = myPrivate(myItem);
        });

        // [workgroup code]
      });
  });
});

It is valid to use more flexible dimensions of the work-item loops. In the following example we issue 8
work-groups but let the runtime choose their size, by not passing a work-group size to the parallel_for_work_group call. The parallel_for_work_item loops may also vary in size, with their execution ranges unrelated to the dimensions of the work-group, and the compiler generating an appropriate iteration space to fill the gap. In this case, the h_item provides access to local ids and ranges that reflect both kernel and parallel_for_work_item invocation ranges.

myQueue.submit([handler & cgh] {
  cgh.parallel_for_work_group(range<3>(2, 2, 2), [=](group<3> myGroup) {
    // Launch a set of work-items for each work-group. The number of work-items
    // is chosen by the runtime because the work-group size was not specified to
    // parallel_for_work_group and a logical range is not specified to
    // parallel_for_work_item.
    myGroup.parallel_for_work_item([=](h_item<3> myItem) {
      // [work-item code]
    });

    // Implicit work-group barrier

    // Launch 512 logical work-items that will be executed by the underlying
    // work-group size chosen by the runtime. myItem allows the logical and
    // physical work-item IDs to be queried. 512 logical work-items will
This interface offers a more intuitive way for tiling parallel programming paradigms. In summary, the hierarchical model allows a developer to distinguish the execution at work-group level and at work-item level using the `parallel_for_work_group` and the nested `parallel_for_work_item` functions. It also provides this visibility to the compiler without the need for difficult loop fission such that host execution may be more efficient.

A `kernel_handler` can optionally be passed as a parameter to the SYCL kernel function that is invoked by any variant of `parallel_for_work_group`.

4.9.4.3. SYCL functions for explicit memory operations

In addition to kernels, command group objects can also be used to perform manual operations on host and device memory by using the copy API of the command group handler. Manual copy operations can be seen as specialized kernels executing on the device, except that typically this operations will be implemented using a host API that exists as part of a backend (e.g, OpenCL enqueue copy operations).
These explicit copy operations have a source and a destination. When an accessor is the source of the operation, the destination can be a host pointer or another accessor. The source accessor must have either `access_mode::read` or `access_mode::read_write` access mode. When an accessor is the destination of the explicit copy operation, the source can be a host pointer or another accessor. The destination accessor must have either `access_mode::write`, `access_mode::read_write`, `access_mode::discard_write` or `access_mode::discard_read_write` access mode.

When an accessor is used as a parameter to one of these explicit copy operations, the target must be either `target::device` or `target::constant_buffer`.

When accessors are both the source and the destination, the operation is executed on objects controlled by the SYCL runtime. The SYCL runtime is allowed to not perform an explicit in-copy operation if a different path to update the data is available according to the SYCL application memory model.

The most recent copy of the memory object may reside on any context controlled by the SYCL runtime, or on the host in a pointer controlled by the SYCL runtime. The SYCL runtime will ensure that data is copied to the destination once the command group has completed execution.

Whenever a host pointer is used as either the source or the destination of these explicit memory operations, it is the responsibility of the user for that pointer to have at least as much memory allocated as the accessor is giving access to, e.g: if an accessor accesses a range of 10 elements of `int` type, the host pointer must at least have `10 * sizeof(int)` bytes of memory allocated.

A special case is the `update_host` member function. This member function only requires an accessor, and instructs the runtime to update the internal copy of the data in the host, if any. This is particularly useful when users use manual synchronization with host pointers, e.g. via mutex objects on the `buffer` constructors.

Table 135 describes the interface for the explicit copy operations.

**Table 135. Member functions of the handler class**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, typename DestT, access::placeholder IsPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder&gt; src, std::shared_ptr&lt;DestT&gt; dest)</code></td>
<td>Copies the contents of the memory object accessed by <code>src</code> into the memory pointed to by <code>dest</code>. <code>dest</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>src</code>.</td>
</tr>
<tr>
<td><code>template &lt;typename SrcT, typename DestT, int DestDims, access_mode DestMode, target DestTgt, access::placeholder IsPlaceholder&gt; void copy(std::shared_ptr&lt;SrcT&gt; src, accessor&lt;DestT, DestDims, DestMode, DestTgt, IsPlaceholder&gt; dest)</code></td>
<td>Copies the contents of the memory pointed to by <code>src</code> into the memory object accessed by <code>dest</code>. <code>src</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>dest</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>template &lt;typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, typename DestT, access::placeholder IsPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt, IsPlaceholder&gt; src, DestT* dest)</code></td>
<td>Copies the contents of the memory object accessed by <code>src</code> into the memory pointed to by <code>dest</code>. <code>dest</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>src</code>.</td>
</tr>
<tr>
<td><code>template &lt;typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, typename DestT, access::placeholder IsPlaceholder&gt; void copy(const SrcT* src, accessor&lt;DestT, DestDims, DestMode, DestTgt, IsPlaceholder&gt; dest)</code></td>
<td>Copies the contents of the memory pointed to by <code>src</code> into the memory object accessed by <code>dest</code>. <code>src</code> must be a host pointer and must have at least as many bytes as the range accessed by <code>dest</code>.</td>
</tr>
<tr>
<td><code>template &lt;typename SrcT, int SrcDims, access_mode SrcMode, target SrcTgt, typename DestT, int DestDims, access_mode DestMode, target DestTgt, access::placeholder IsSrcPlaceholder, IsDestPlaceholder&gt; void copy(accessor&lt;SrcT, SrcDims, SrcMode, SrcTgt, IsSrcPlaceholder&gt; src, accessor&lt;DestT, DestDims, DestMode, DestTgt, IsDestPlaceholder&gt; dest)</code></td>
<td>Copies the contents of the memory object accessed by <code>src</code> into the memory object accessed by <code>dest</code>. The size of the <code>src</code> accessor determines the number of bytes that are copied, and <code>dest</code> must have at least this many bytes. If the size of <code>dest</code> is too small, the implementation throws a synchronous exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>template &lt;typename T, int Dims, access_mode Mode, target Tgt, access::placeholder IsPlaceholder&gt; void update_host(accessor&lt;T, Dims, Mode, Tgt, IsPlaceholder&gt; acc)</code></td>
<td>The contents of the memory object accessed via <code>acc</code> on the host are guaranteed to be up-to-date after this <code>command group</code> object execution is complete.</td>
</tr>
<tr>
<td><code>template &lt;typename T, int Dims, access_mode Mode, target Tgt, access::placeholder IsPlaceholder&gt; void fill(accessor&lt;T, Dims, Mode, Tgt, IsPlaceholder&gt; dest, const T&amp; src)</code></td>
<td>Replicates the value of <code>src</code> into the memory object accessed by <code>dest</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>void memcpy(void* dest, const void* src, size_t numBytes)</code></td>
<td>Copies <code>numBytes</code> of data from the pointer <code>src</code> to the pointer <code>dest</code>. The <code>dest</code> and <code>src</code> parameters must each either be a host pointer or a pointer within a USM allocation that is accessible on the handler's device. If a pointer is to a USM allocation, that allocation must have been created from the same context as the handler's queue. For more detail on USM, please see Section 4.8.</td>
</tr>
<tr>
<td><code>template&lt;typename T&gt; void copy(const T* src, T* dest, size_t count)</code></td>
<td>Copies <code>count</code> elements of type <code>T</code> from the pointer <code>src</code> to the pointer <code>dest</code>. The <code>dest</code> and <code>src</code> parameters must each either be a host pointer or a pointer within a USM allocation that is accessible on the handler's device. If a pointer is to a USM allocation, that allocation must have been created from the same context as the handler's queue. For more detail on USM, please see Section 4.8.</td>
</tr>
<tr>
<td><code>void memset(void* ptr, int value, size_t numBytes)</code></td>
<td>Fills <code>numBytes</code> bytes of memory beginning at address <code>ptr</code> with <code>value</code>. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. Note that <code>value</code> is interpreted as an <code>unsigned char</code>. For more detail on USM, please see Section 4.8.</td>
</tr>
<tr>
<td><code>template&lt;typename T&gt; void fill(void* ptr, const T&amp; pattern, size_t count)</code></td>
<td>Replicates the provided <code>pattern</code> into the memory at address <code>ptr</code>. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. The <code>pattern</code> is filled <code>count</code> times. For more detail on USM, please see Section 4.8.</td>
</tr>
<tr>
<td><code>void prefetch(void* ptr, size_t numBytes)</code></td>
<td>Enqueues a prefetch of <code>num_bytes</code> of data starting at address <code>ptr</code>. The <code>ptr</code> must point within a USM allocation from the same context as the handler's queue, and the pointer must be accessible from the queue's device. For more detail on USM, please see Section 4.8.</td>
</tr>
</tbody>
</table>
The listing below illustrates how to use explicit copy operations in SYCL. The example copies half of the contents of a `std::vector` into the device, leaving the rest of the contents of the buffer on the device unchanged.

```cpp
const size_t nElems = 10u;

// Create a vector and fill it with values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
std::vector<int> v { nElems };
std::iota(std::begin(v), std::end(v), 0);

// Create a buffer with no associated user storage
sycl::buffer<int, 1> b { range<1>(nElems)};

// Create a queue
queue myQueue;

myQueue.submit([&](handler& cgh) {
    // Retrieve a ranged write accessor to a global buffer with access to the
    // first half of the buffer
    accessor acc { b, cgh, range<1>(nElems / 2), id<1>(0), write_only };    
    // Copy the first five elements of the vector into the buffer associated with
    // the accessor
    cgh.copy(v.data(), acc);
});
```

### 4.9.4.4. Functions for using a kernel bundle

```cpp
void use_kernel_bundle(
    const kernel_bundle<bundle_state::executable>& execBundle);
```

**Effects:** The command group associated with the `handler` will use device images of the `kernel_bundle` `execBundle` in any of its kernel invocation commands. If the `kernel_bundle` contains multiple device images that are compatible with the device to which the kernel is submitted, then the device image chosen is implementation-defined.

If the command group attempts to invoke a kernel that is not contained by a compatible device image in
execBundle, the kernel invocation command throws a synchronous exception with the `errc::kernel_not_supported` error code. If the command group has a secondary queue, then the execBundle must contain a kernel that is compatible with both the primary queue’s device and the secondary queue’s device, otherwise the kernel invocation command throws this exception.

Since the handler method for setting specialization constants is incompatible with the kernel bundle method, applications should not call this function if `handler::set_specialization_constant()` has been previously called for this same command group.

**Throws:**

- An exception with the `errc::invalid` error code if the context associated with the command group handler via its associated primary queue or the context associated with the secondary queue (if provided) is different from the context associated with the kernel bundle specified by execBundle.
- An exception with the `errc::invalid` error code if `handler::set_specialization_constant()` has been called for this command group.

### 4.9.5. Specialization constants

Device code can make use of specialization constants which represent constants whose values can be set dynamically during execution of the SYCL application. The values of these constants are fixed when a SYCL kernel function is invoked, and they do not change during the execution of the kernel. However, the application is able to set a new value for a specialization constant each time a kernel is invoked, so the values can be tuned differently for each invocation.

There are two methods for an application to use specialization constants, one method requires creating a kernel_bundle object and the other does not. The syntax for both methods is mostly the same. Both methods declare specialization constants in the same way, and kernels read their values in the same way. The main difference is whether their values are set via `handler::set_specialization_constant()` or via `kernel_bundle::set_specialization_constant()`. These two methods are incompatible with one another, so they may not both be used by the same command group.

Implementations that support online compilation of kernel bundles will likely implement both methods of specialization constants using kernel bundles. Therefore, applications should expect that there is some overhead associated with invoking a kernel with new values for its specialization constants. A typical implementation records the values of specialization constants set via `handler::set_specialization_constant()` and remembers these values until a kernel is invoked (e.g. via `parallel_for()`). At this point, the implementation determines the bundle that contains the invoked kernel. If that bundle has already been compiled for the handler’s device and compiled with the correct values for the specialization constants, the kernel is scheduled for invocation. Otherwise, the implementation compiles the bundle before scheduling the kernel for invocation. Therefore, applications that frequently change the values of specialization constants may see an overhead associated with recompilation of the kernel’s bundle.

#### 4.9.5.1. Declaring a specialization constant

Specialization constants must be declared using the `specialization_id` class with the following restrictions:

- the template parameter `T` must be a `device copyable` type;
- the `specialization_id` variable must be declared as `constexpr`;
- the `specialization_id` variable must be declared in either namespace scope or in class scope;
- if the `specialization_id` variable is declared in class scope, it must have public accessibility when referenced from namespace scope;
• the `specialization_id` variable may not be shadowed by another identifier `X` which has the same name and is declared in an `inline` namespace, such that the `specialization_id` variable is no longer accessible after the declaration of `X`;

• if the `specialization_id` variable is declared in a namespace, none of the enclosing namespace names `N` may be shadowed by another identifier `X` which has the same name as `N` and is declared in an `inline` namespace, such that `N` is no longer accessible after the declaration of `X`.

The expectation is that some implementations may conceptually insert code at the end of a translation unit which references each `specialization_id` variable that is declared in that translation unit. The restrictions listed above make this possible by ensuring that these variables are accessible at the end of the translation unit.

The following example illustrates some of these restrictions:

```cpp
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

struct Compound {
  int i;
  float f;
};

constexpr specialization_id<int> a { 1 }; // OK
constexpr specialization_id<Compound> b { 2, 3.14 }; // OK
inline constexpr specialization_id<int> c { 3 }; // OK
static constexpr specialization_id<int> d { 4 }; // OK
specialization_id<int> e { 5 }; // ILLEGAL: not constexpr

struct Bar {
  static constexpr specialization_id<int> f { 6 }; // OK
};
struct Baz {
  struct Inner {
    static constexpr specialization_id<int> g { 7 }; // OK
  };
};
class Boo {
  static constexpr specialization_id<int> h { 8 }; // ILLEGAL: not public member
};

void Func() {
  static constexpr specialization_id<int> i { 9 }; // ILLEGAL: not at namespace
  // or class scope
  /* ... */
}

constexpr specialization_id<int> same_name { 10 }; // OK
namespace foo {
  constexpr specialization_id<int> same_name { 11 }; // OK
}
namespace {
  constexpr specialization_id<int> same_name { 12 }; // OK
}
inline namespace other {
  int same_name; // ILLEGAL: shadows "specialization_id" variable with same name in
```
A synopsis of this class is shown below.

```cpp
namespace sycl {
  template <typename T> class specialization_id {
    public:
    using value_type = T;
    template <class... Args> explicit constexpr specialization_id(Args&&... args);
    specialization_id(const specialization_id& rhs) = delete;
    specialization_id(specialization_id&& rhs) = delete;
    specialization_id& operator=(const specialization_id& rhs) = delete;
    specialization_id& operator=(specialization_id&& rhs) = delete;
  };
}
```

4.9.5.1.1 Constructors

```cpp
template <class... Args> explicit constexpr specialization_id(Args&&... args);
```

**Constraints:** Available only when \texttt{std::is_constructible_v<T, Args...>} evaluates to \texttt{true}.

**Effects:** Constructs a \texttt{specialization_id} containing an instance of \texttt{T} initialized with \texttt{args...}, which represents the specialization constant's default value.

4.9.5.1.2 Special member functions

```cpp
specialization_id(const specialization_id& rhs) = delete; // (1)
specialization_id(specialization_id&& rhs) = delete; // (2)
specialization_id& operator=(const specialization_id& rhs) = delete; // (3)
specialization_id& operator=(specialization_id&& rhs) = delete; // (4)
```

1. Deleted copy constructor.
2. Deleted move constructor.
3. Deleted copy assignment operator.
4. Deleted move assignment operator.

4.9.5.2 Setting and getting the value of a specialization constant

If the application uses specialization constants without creating a \texttt{kernel_bundle} object, it can set and get their values from \texttt{command group scope} by calling member functions of the \texttt{handler} class. These member functions have a template parameter \texttt{SpecName} whose value must be a reference to a variable of type
specialization_id, which defines the type and default value of the specialization constant.

When not using a kernel bundle, the value of a specialization constant that is used in a kernel invoked from a command group is affected by calls to set its value from that same command group, but it is not affected by calls from other command groups even if those calls are from another invocation of the same command group function object.

\[
\text{template <auto& SpecName>}
\]
\[
\text{void set_specialization_constant(}
\]
\[
\text{typename std::remove_reference_t<decltype(SpecName)>value_type value;}
\]

**Effects:** Sets the value of the specialization constant whose address is SpecName for this handler's command group. If the specialization constant's value was previously set in this same command group, the value is overwritten.

This function may be called even if the specialization constant SpecName isn't used by the kernel that is invoked by this handler's command group. Doing so has no effect on the invoked kernel.

**Throws:**

- An exception with the errc::invalid error code if a kernel bundle has been bound to the handler via use_kernel_bundle().

\[
\text{template <auto& SpecName>}
\]
\[
\text{typename std::remove_reference_t<decltype(SpecName)>value_type}
\]
\[
\text{get_specialization_constant();}
\]

**Returns:** The value of the specialization constant whose address is SpecName for this handler's command group. If the value was previously set in this handler's command group, that value is returned. Otherwise, the specialization constant's default value is returned.

**Throws:**

- An exception with the errc::invalid error code if a kernel bundle has been bound to the handler via use_kernel_bundle().

### 4.9.5.3. Reading the value of a specialization constant from device code

In order to read the value of a specialization constant from device code, the SYCL kernel function must be declared to take an object of type kernel_handler as its last parameter. The SYCL runtime constructs this object, which has a member function for reading the specialization constant's value. A synopsis of this class is shown below.

```c++
1 namespace sycl {
2
3 class kernel_handler {
4  public:
5    template <auto& SpecName>
6    typename std::remove_reference_t<decltype(SpecName)>::value_type
7    get_specialization_constant();
8  };
9
10 } // namespace sycl
```
4.9.5.3.1. Member functions

```cpp
template<auto & SpecName>
typename std::remove_reference_t<decltype(SpecName)>::value_type
get_specialization_constant();
```

Returns: The value of the specialization constant whose address is `SpecName`. For a kernel invoked from a command group that was not bound to a kernel bundle, the value is the same as what would have been returned if `handler::get_specialization_constant()` was called immediately before invoking the kernel. For a kernel invoked from a command group that was bound to a kernel bundle, the value is the same as what would be returned if `kernel_bundle::get_specialization_constant()` was called on the bound bundle.

4.9.5.4. Example usage

The following example performs a convolution and uses specialization constants to set the values of the coefficients.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl;

using coeff_t = std::array<std::array<float, 3>, 3>;

// Read coefficients from somewhere.
coeff_t get_coefficients();

// Identify the specialization constant.
constexpr specialization_id<coeff_t> coeff_id;

void do_conv(buffer<float, 2> in, buffer<float, 2> out) {
    queue myQueue;

    myQueue.submit([&](handler &cgh) {
        accessor in_acc { in, cgh, read_only };
        accessor out_acc { out, cgh, write_only };

        // Set the coefficient of the convolution as constant.
        // This will build a specific kernel the coefficient available as literals.
        cgh.set_specialization_constant<coeff_id>(get_coefficients());

        cgh.parallel_for<class Convolution>(in.get_range(), [=](item<2> item_id,
                                                   kernel_handler h) {
            float acc = 0;
            coeff_t coef = h.get_specialization_constant<coeff_id>();
            for (int i = -1; i <= 1; i++) {
                for (int j = -1; j <= 1; j++) {
                    if (item_id[0] + i < 0 || item_id[0] + i >= in_acc.get_range()[0])
                        continue;
                    if (item_id[1] + j < 0 || item_id[1] + j >= in_acc.get_range()[1])
                        continue;
                    acc += coef[i + 1][j + 1] * in_acc[item_id[0] + i][item_id[1] + j];
                }
            }
        });
    });
}
```
4.10. Host tasks

4.10.1. Overview

A host task is a native C++ callable which is scheduled by the SYCL runtime. A host task is submitted to a queue via a command group by a host task command.

When a host task command is submitted to a queue it is scheduled based on its data dependencies with other commands including kernel invocation commands and asynchronous copies, resolving any requisites created by accessors attached to the command group as defined in Section 3.8.1.

Since a host task is invoked directly by the SYCL runtime rather than being compiled as a SYCL kernel function, it does not have the same restrictions as a SYCL kernel function, and can therefore contain any arbitrary C++ code.

Capturing accessors in a host task is allowed, however, capturing or using any other SYCL class that has reference semantics (see Section 4.5.2) is undefined behavior.

A host task can be enqueued on any queue and the callable will be invoked directly by the SYCL runtime, regardless of which device the queue is associated with.

A host task is enqueued on a queue via the host_task member function of the handler class. The event returned by the submission of the associated command group enters the completed state (corresponding to a status of info::event_command_status::complete) once the invocation of the provided C++ callable has returned. Any uncaught exception thrown during the execution of a host task will be turned into an asynchronous error that can be handled as described in Section 4.13.1.1.

A host task can optionally be used to interoperate with the native backend objects associated with the queue executing the host task, the context that the queue is associated with, the device that the queue is associated with and the accessors that have been captured in the callable, via an optional interop_handle parameter.

This allows host tasks to be used for two purposes: either as a task which can perform arbitrary C++ code within the scheduling of the SYCL runtime or as a task which can perform interoperability at a point within the scheduling of the SYCL runtime.

For the former use case, construct a buffer accessor with target::host_task or an image accessor with image_target::host_task. This makes the buffer or image available on the host during execution of the host task.

For the latter case, construct a buffer accessor with target::device or target::constant_buffer, or construct an image accessor with image_target::device. This makes the buffer or image available on the device that is associated with the queue used to submit the host task, so that it can be accessed via interoperability member functions provided by the interop_handle class.

Local accessors cannot be used within a host task.

```cpp
namespace sycl {
}
```
class interop_handle {
private:
  interop_handle(__unspecified__);
public:
  interop_handle() = delete;
  backend get_backend() const noexcept;
  template <backend Backend, typename DataT, int Dims, access_mode AccessMode,
            target AccessTarget, access::placeholder isPlaceholder>
    backend_return_t<Backend, buffer<DataT, Dims>>
    get_native_mem(const accessor<DataT, Dims, AccessMode, AccessTarget,
                       isPlaceholder>& bufferAccessor) const;
  template <backend Backend, typename DataT, int Dims, access_mode AccMode>
    backend_return_t<Backend, unsampled_image<Dims>>
    get_native_mem(const unsampled_image_accessor<DataT, Dims, AccMode,
                      image_target::device>& imageAcc) const;
  template <backend Backend, typename DataT, int Dims>
    backend_return_t<Backend, sampled_image<Dims>>
    get_native_mem(const sampled_image_accessor<DataT, Dims, image_target::device>& imageAcc)
    const;
  template <backend Backend>
    backend_return_t<Backend, queue>
    get_native_queue() const;
  template <backend Backend>
    backend_return_t<Backend, device>
    get_native_device() const;
  template <backend Backend>
    backend_return_t<Backend, context>
    get_native_context() const;
};

class handler {
  ...
  public :
    template <typename T>
    void host_task(T&& hostTaskCallable);
    ...
}; // namespace sycl

4.10.2. Class interop_handle

The interop_handle class is an abstraction over the queue which is being used to invoke the host task and its associated device and context. It also represents the state of the SYCL runtime dependency model at the point the host task is invoked.
The `interop_handle` class provides access to the native backend object associated with the queue, device, context and any buffers or images that are captured in the callable being invoked in order to allow a host task to be used for interoperability purposes.

An `interop_handle` cannot be constructed by user-code, only by the SYCL runtime.

```cpp
1 class interop_handle;
```

### 4.10.2.1. Constructors

```cpp
1 private:
2 interop_handle(__unspecified__); // (1)
3
4 public:
5 interop_handle() = delete; // (2)
```

1. Private implementation-defined constructor with unspecified arguments so that the SYCL runtime can construct a `interop_handle`.
2. Explicitly deleted default constructor.

### 4.10.2.2. Member functions

```cpp
1 backend get_backend() const noexcept;
```

1. Returns: Returns a `backend` identifying the SYCL backend associated with the queue associated with this `interop_handle`.

### 4.10.2.3. Template member functions `get_native_*`

```cpp
1 // SPDX-License-Identifier: MIT
2
3 template <backend Backend, typename DataT, int Dims, access_mode AccMode, 
4         target AccTarget, access::placeholder IsPlaceholder>
5 backend_return_t<Backend, buffer<DataT, Dims>>
6 get_native_mem(const accessor<DataT, Dims, AccMode, AccTarget, // (1)
7         IsPlaceholder>& bufferAcc) const;
8
9 template <backend Backend, typename DataT, int Dims, access_mode AccMode>
10 backend_return_t<Backend, unsampled_image<Dims>> get_native_mem( // (2)
11         const unsampled_image_accessor<DataT, Dims, AccMode, image_target::device>& imageAcc) const;
12
13 template <backend Backend, typename DataT, int Dims>
14 backend_return_t<Backend, sampled_image<Dims>> get_native_mem( // (3)
15         const sampled_image_accessor<DataT, Dims, image_target::device>& imageAcc) const;
16
17 template <backend Backend>
18 backend_return_t<Backend, queue> get_native_queue() const; // (4)
19
20 template <backend Backend>
21 backend_return_t<Backend, device> get_native_device() const; // (5)
```
1. **Constraints:** Available only if the optional interoperability function `get_native` taking a `buffer` is available and if `accTarget` is `target::device`.

   **Returns:** The native backend object associated with the underlying `buffer` of accessor `bufferAcc`. The native backend object returned must be in a state where it represents the memory in its current state within the SYCL runtime dependency model and is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

   **Throws:** An exception with the `errc::invalid` error code if the accessor `bufferAcc` was not registered with the command group which contained the host task. Must throw an exception with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

2. **Constraints:** Available only if the optional interoperability function `get_native` taking an `unsampled_image` is available.

   **Returns:** The native backend object associated with with the underlying `unsampled_image` of accessor `imageAcc`. The native backend object returned must be in a state where it represents the memory in its current state within the SYCL runtime dependency model and is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

   **Throws:** An exception with the `errc::invalid` error code if the accessor `imageAcc` was not registered with the command group which contained the host task.

3. **Constraints:** Available only if the optional interoperability function `get_native` taking an `sampled_image` is available.

   **Returns:** The native backend object associated with with the underlying `sampled_image` of accessor `imageAcc`. The native backend object returned must be in a state where it represents the memory in its current state within the SYCL runtime dependency model and is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

   **Throws:** An exception with the `errc::invalid` error code if the accessor `imageAcc` was not registered with the command group which contained the host task. Must throw an exception with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

4. **Constraints:** Available only if the optional interoperability function `get_native` taking a `queue` is available.

   **Returns:** The native backend object associated with the queue that the host task was submitted to. If the command group was submitted with a secondary queue and the fall-back was triggered, the queue that is associated with the `interop_handle` must be the fall-back queue. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

   **Throws:** Must throw an exception with the `errc::backend_mismatch` error code if `Backend != get_backend()`.

5. **Constraints:** Available only if the optional interoperability function `get_native` taking a `device` is available.
Returns: The native backend object associated with the device that is associated with the queue that the host task was submitted to. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

Throws: Must throw an exception with the errc::backend_mismatch error code if Backend != get_backend().

6. Constraints: Available only if the optional interoperability function get_native taking a context is available.

Returns: The native backend object associated with the context that is associated with the queue that the host task was submitted to. The native backend object returned must be in a state where it is capable of being used in a way appropriate for the associated SYCL backend. It is undefined behavior to use the native backend object outside of the scope of the host task.

Throws: Must throw an exception with the errc::backend_mismatch error code if Backend != get_backend().

4.10.3. Additions to the handler class

This section describes member functions in the command group handler class that are used with host tasks.

```cpp
1 class handler {
2   ...
3   public
4     : template <typename T>
5        void
6        host_task(T&& hostTaskCallable); // (1)
7   ...
8 }
```

1. Effects: Enqueues an implementation-defined command to the SYCL runtime to invoke hostTaskCallable exactly once. The scheduling of the invocation of hostTaskCallable in relation to other commands enqueued to the SYCL runtime must be in accordance with the dependency model described in Section 3.8.1. Initializes an interop_handle object and passes it to hostTaskCallable when it is invoked if std::is_invocable_v<T, interop_handle> evaluates to true, otherwise invokes hostTaskCallable as a nullary function.

4.11. Kernel bundles

Kernel bundles provide several features to a SYCL application. For implementations that support an online compiler, they provide fine grained control over the online compilation of device code. For example, an application can use a kernel bundle to compile its kernels at a specific time during the application's execution (such as during its initialization), rather than relying on the implementation's default behavior (which may not compile kernels until they are submitted).

Kernel bundles also provide a way for the application to set the values of specialization constants in many kernels before any of them are submitted to a device, which could potentially be more efficient in some cases.

Kernel bundles provide a way for the application to introspect its kernels. For example, an application can use a bundle to query a kernel's work-group size when it is run on a specific device.
Finally, kernel bundles provide an extension point to interoperate with backend and device specific features. Some examples of this include invocation of device specific built-in kernels, online compilation of kernel code with vendor specific options, or interoperation with kernels created with backend APIs.

4.11.1. Overview

A kernel bundle is a high-level abstraction which represents a set of kernels that are associated with a context and can be executed on a number of devices, where each device is associated with that same context. Depending on how a bundle is obtained, it could represent all of the SYCL kernel functions in the SYCL application, or a certain subset of them.

A kernel bundle is composed of one or more device images, where each device image is an indivisible unit of compilation and/or linking. When the SYCL runtime compiles or links one of the kernels represented by the device image, it must also compile or link any other kernels the device image represents. Once a device image is compiled and linked, any of the other kernels which that device image represents may be invoked without further compilation or linking.

Each SYCL kernel function a bundle represents must reside in at least one of the bundle’s device images. However, it is not necessary for each device image to contain all of the kernel functions that the bundle represents. The granularity in which kernel functions are grouped into device images is an implementation detail.

To illustrate the intent of device images, a hypothetical implementation could represent an application’s kernel functions in both the SPIR-V format and also in a native device code format. The implementation’s ahead-of-time compiler in this example produces device images with native code for certain devices and also produces SPIR-V device images for use with other devices. Note that in such an implementation, a particular kernel function could be represented in more than one device image.

An implementation could choose to have all kernel functions from all translation units grouped together in a single device image, to have each kernel function represented in its own device image, or to group kernel functions in some other way.

Each device associated with a kernel bundle must have at least one compatible device image, meaning that the implementation can either invoke the image’s kernel functions directly on the device or that the implementation can translate the device image into a format that allows it to invoke the kernel functions.

An outcome of this definition is that each kernel function in a bundle must be invocable on at least one of the devices associated with the bundle. However, it is not necessary for every kernel function in the bundle to be invocable on every associated device.

One common reason why a kernel function might not be invocable on every device associated with a bundle is if the kernel uses optional device features. It’s possible that these features are available to only some devices in the bundle.

The use of optional device features could affect how the implementation groups kernels into device images, depending on how these features are represented. For example, consider an implementation where the optional feature is represented in SPIR-V but translation of that SPIR-V into native code will fail if the target device does not support the feature. In such an implementation, kernels that use optional features should not be grouped into the same device image as kernels that do not use these features. Since a device image is an indivisible unit of compilation, doing so would cause a compilation failure if a kernel K1 is invoked on a device D1 if K1 happened to reside in the same device image as another kernel K2 that used a feature which is not supported on device D1.
A SYCL application can obtain a kernel bundle by calling one of the overloads of the `get_kernel_bundle()` free function. Certain backends may provide additional mechanisms for obtaining bundles with other representations. If this is supported, the backend specification document will describe the details.

Once a kernel bundle has been obtained there are a number of free functions for performing compilation, linking and joining. Once a bundle is compiled and linked, the application can invoke kernels from the bundle by calling `handler::use_kernel_bundle()` as described in Section 4.9.4.4.

### 4.11.2. Synopsis

```cpp	namespace sycl {
	enum class bundle_state : /* unspecified */ { input, object, executable };

class kernel_id { /* ... */
};

template <bundle_state State> class kernel_bundle { /* ... */
};

template <typename KernelName> kernel_id get_kernel_id();

std::vector<kernel_id> get_kernel_ids();

template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                           const std::vector<kernel_id>& kernelIds);

template <typename KernelName, bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                           const std::vector<device>& devs);

template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                           const std::vector<device>& devs,
                                           const std::vector<kernel_id>& kernelIds);

template <typename KernelName, bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                           const std::vector<device>& devs);

template <bundle_state State, typename Selector>
kernel_bundle<State> get_kernel_bundle(const context& ctxt, Selector selector);

template <bundle_state State, typename Selector>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                           const std::vector<device>& devs,
                                           Selector selector);
```

See Section 5.7 for more about optional device features.
template <bundle_state State> bool has_kernel_bundle(const context& ctx);

template <bundle_state State>
bool has_kernel_bundle(const context& ctx,
    const std::vector<kernel_id>& kernelIds);

template <typename KernelName, bundle_state State>
bool has_kernel_bundle(const context& ctx);

template <bundle_state State>
bool has_kernel_bundle(const context& ctx,
    const std::vector<device>& devs);

template <typename KernelName, bundle_state State>
bool has_kernel_bundle(const context& ctx,
    const std::vector<device>& devs,
    const std::vector<kernel_id>& kernelIds);

template <typename KernelName> bool is_compatible(const device& dev);

kernel_bundle<State> join(const std::vector<kernel_bundle<State>>& bundles);

kernel_bundle<bundle_state::object>
compile(const kernel_bundle<bundle_state::input>& inputBundle,
    const property_list& propList = {});

kernel_bundle<bundle_state::object>
compile(const kernel_bundle<bundle_state::input>& inputBundle,
    const std::vector<device>& devs, const property_list& propList = {});

kernel_bundle<bundle_state::executable>
link(const kernel_bundle<bundle_state::object>& objectBundle,
    const property_list& propList = {});

kernel_bundle<bundle_state::executable>
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
    const property_list& propList = {});

kernel_bundle<bundle_state::executable>
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
    const std::vector<device>& devs, const property_list& propList = {});

kernel_bundle<bundle_state::executable>
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
    const std::vector<device>& devs, const property_list& propList = {});

kernel_bundle<bundle_state::executable>
built(const kernel_bundle<bundle_state::input>& inputBundle,
    const property_list& propList = {});
4.11.3. Fixed-function built-in kernels

SYCL allows a SYCL backend to expose fixed functionality as non-programmable built-in kernels. The availability and behavior of these built-in kernels are backend specific and are not required to follow the SYCL execution and memory models. However, the basic interface is common to all backends.

4.11.4. Bundle states

A kernel bundle can be in one of three different bundle states which are represented by an enum class called bundle_state. Table 136 describes the semantics of these three states.

The states form a progression. A bundle in bundle_state::input can be translated into bundle_state::object by online compilation of the bundle. A bundle in bundle_state::object can be translated into bundle_state::executable by online linking.

Each implementation is free to define the "online compilation" and "online linking" operations as it sees fit, so long as this progression of bundle states is preserved and so long as the bundles in each state behave as specified.

There is no requirement that an implementation must expose kernels in bundle_state::input or bundle_state::object. In fact, an implementation could expose some kernels in these states but not others. For example, this behavior could be controlled by implementation specific options to the ahead-of-time compiler. Kernels that are not exposed in these states cannot be online compiled or online linked by the application.

All kernels defined in the SYCL application, however, must be exposed in bundle_state::executable because this is the only state that allows a kernel to be invoked on a device. Device built-in kernels are also exposed in bundle_state::executable.

If an application exposes a bundle in bundle_state::input for a device D, then the implementation must also provide an online compiler for device D. Therefore, an application need not explicitly test for aspect::online_compiler if it successfully obtains a bundle in bundle_state::input for that device. Likewise, an implementation must provide an online linker for device D if it exposes a bundle in bundle_state::object for device D.

<table>
<thead>
<tr>
<th>Bundle State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bundle_state::input</td>
<td>The device images in the kernel bundle have a format that must be compiled and linked before their kernels can be invoked. For example, an implementation could use this state for device images that are stored in an intermediate language format or for device images that are stored as source code strings.</td>
</tr>
<tr>
<td>bundle_state::object</td>
<td>The device images in the kernel bundle have a format that must be linked before their kernels can be invoked.</td>
</tr>
<tr>
<td>bundle_state::executable</td>
<td>The device images in the kernel bundle are in a format that allows them to be invoked on a device. For example, an implementation could use this state for device images that have been compiled into the device's native code.</td>
</tr>
</tbody>
</table>
4.11.5. Kernel identifiers

Some of the functions related to kernel bundles take an input parameter of type `kernel_id` which identifies a kernel. A synopsis of the `kernel_id` class is shown below along with a description of its member functions. Additionally, this class provides the common special member functions and common member functions that are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

As with all SYCL objects that have the common reference semantics, kernel identifiers are equality comparable. Two `kernel_id` objects compare equal if and only if they refer to the same application kernel or to the same device built-in kernel.

There is no public default constructor for this class.

```cpp
namespace sycl {

class kernel_id {

public:

kernel_id() = delete;

const char* get_name() const noexcept;

};

} // namespace sycl
```

`const char* get_name() const noexcept;`

Returns: An implementation-defined null-terminated string containing the name of the kernel. There is no guarantee that this name is unique amongst all the kernels, nor is there a guarantee that the name is stable from one run of the application to another. The lifetime of the memory containing the name is unspecified.

In practice, the lifetime of the memory containing the name will typically extend until the application terminates, unless the kernel associated with the name comes from a dynamic library. In this case, the lifetime of the memory may end if the dynamic library is unloaded.

4.11.6. Obtaining a kernel identifier

An application can obtain an identifier for a kernel that is defined in the application by calling one of the following free functions, or it may obtain an identifier for a device's built-in kernels by querying the device with `info::device::built_in_kernel_ids`.

```cpp
template <typename KernelName> kernel_id get_kernel_id();
```

Preconditions: The template parameter `KernelName` must be the type kernel name of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a `KernelName` in their kernel invocation command in order to obtain their identifier via this function. Applications which call `get_kernel_id()` for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

Returns: The identifier of the kernel associated with `KernelName`. 
4.11.7. Obtaining a kernel bundle

A SYCL application can obtain a kernel bundle by calling one of the overloads of the free function `get_kernel_bundle()`. The implementation may return a bundle that consists of device images that were created by the ahead-of-time compiler, or it may call the online compiler or linker to create the bundle’s device images in the requested state. A bundle may also contain device images that represent a device’s built-in kernels.

When `get_kernel_bundle()` is used to obtain a kernel bundle in `bundle_state::object` or `bundle_state::executable`, any specialization constants in the bundle will have their default values.

```
template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
            const std::vector<device>& devs);
```

**Returns:** A kernel bundle in state `State` which contains all of the kernels in the application which are compatible with at least one of the devices in `devs`. This does not include any device built-in kernels. The bundle's set of associated devices is `devs` (with any duplicate devices removed).

Since the implementation may not represent all kernels in `bundle_state::input` or `bundle_state::object`, calling this function with one of those states may return a bundle that is missing some of the application's kernels.

**Throws:**

- An exception with the `errc::invalid` error code if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a descendent device of some device in `ctxt`.
- An exception with the `errc::invalid` error code if the `devs` vector is empty.
- An exception with the `errc::invalid` error code if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::onlineCompiler`.
- An exception with the `errc::invalid` error code if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::onlineLinker`.
- An exception with the `errc::build_error` code if `State` is `bundle_state::object` or `bundle_state::executable`, if the implementation needs to perform an online compile or link, and if the online compile or link fails.

```
template <bundle_state State>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
            const std::vector<device>& devs,
            const std::vector<kernel_id>& kernelIds);
```

**Returns:** A kernel bundle in state `State` which contains all of the device images that are compatible with at least one of the devices in `devs`, further filtered to contain only those device images that contain at least one of the kernels with the given identifiers. These identifiers may represent kernels that are defined in the application, device built-in kernels, or a mixture of the two. Since the device images may group many kernels together, the returned bundle may contain additional kernels beyond those that are requested in `kernelIds`. The bundle’s set of associated devices is `devs` (with duplicate devices removed).

std::vector<kernel_id> get_kernel_ids();

**Returns:** A vector with the identifiers for all kernels defined in the SYCL application. This does not include identifiers for any device built-in kernels.
Since the implementation may not represent all kernels in `bundle_state::input` or `bundle_state::object`, calling this function with one of those states may return a bundle that is missing some of the kernels in `kernelIds`. The application can test for this via `kernel_bundle::has_kernel()`.

**Throws:**

- **An exception with the errc::invalid error code** if any of the kernels identified by `kernelIds` are incompatible with all devices in `devs`.
- **An exception with the errc::invalid error code** if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a descendant device of some device in `ctxt`.
- **An exception with the errc::invalid error code** if the `devs` vector is empty.
- **An exception with the errc::invalid error code** if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::online_compiler`.
- **An exception with the errc::invalid error code** if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::online_linker`.
- **An exception with the errc::build error code** if `State` is `bundle_state::object` or `bundle_state::executable` if the implementation needs to perform an online compile or link, and if the online compile or link fails.

```cpp
template <bundle_state State, typename Selector>
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
                                       const std::vector<device>& devs,
                                       Selector selector);
```

**Preconditions:** The `selector` must be a unary predicate whose return value is convertible to `bool` and whose parameter is `const device_image<State>&`.

**Effects:** The predicate function `selector` is called once for every device image in the application of state `State` which is compatible with at least one of the devices in `devs`. The function's return value determines whether a device image is included in the new kernel bundle. The `selector` is called only for device images that contain kernels defined in the application, not for device images that contain device built-in kernels.

**Returns:** A kernel bundle in state `State` which contains all of the device images for which the `selector` returns `true`. The bundle's set of associated devices is `devs` (with duplicate devices removed).

**Throws:**

- **An exception with the errc::invalid error code** if any of the devices in `devs` is not one of devices contained by the context `ctxt` or is not a descendant device of some device in `ctxt`.
- **An exception with the errc::invalid error code** if the `devs` vector is empty.
- **An exception with the errc::invalid error code** if `State` is `bundle_state::input` and any device in `devs` does not have `aspect::online_compiler`.
- **An exception with the errc::invalid error code** if `State` is `bundle_state::object` and any device in `devs` does not have `aspect::online_linker`.

This function is intended to be used in conjunction with backend specific APIs that allow the application to choose device images based on backend specific criteria.

This function does not call the online compiler or linker to translate device images into state `State`. If the application wants to select specific device images and also compile or link them into the desired state, it can do this by calling `compile()` or `link()` and then optionally joining several bundles together with `join()`.
template <bundle_state State> // (1)
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <bundle_state State> // (2)
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
        const std::vector<kernel_id>& kernelIds);

template <bundle_state State, typename Selector> // (3)
kernel_bundle<State> get_kernel_bundle(const context& ctxt, Selector selector);

1. Equivalent to get_kernel_bundle<State>(ctxt, ctxt.get_devices()).
2. Equivalent to get_kernel_bundle<State>(ctxt, ctxt.get_devices(), kernelIds).
3. Equivalent to get_kernel_bundle<State>(ctxt, ctxt.get_devices(), selector).

template <typename KernelName, bundle_state State> // (1)
kernel_bundle<State> get_kernel_bundle(const context& ctxt);

template <typename KernelName, bundle_state State> // (2)
kernel_bundle<State> get_kernel_bundle(const context& ctxt,
        const std::vector<device>& devs);

Preconditions: The template parameter KernelName must be the type kernel name of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a KernelName in their kernel invocation command in order to use these functions. Applications which call these functions for a KernelName that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. Equivalent to get_kernel_bundle<State>(ctxt, ctxt.get_devices(), {get_kernel_id<KernelName>()}).
2. Equivalent to get_kernel_bundle<State>(ctxt, devs, {get_kernel_id<KernelName>()}).

4.11.8. Querying if a kernel bundle exists

Most overloads of get_kernel_bundle() have a matching overload of the free function has_kernel_bundle() which checks to see if a kernel bundle with the requested characteristics exists.

template <bundle_state State>
bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs);

Returns: true only if all of the following are true:

- The application defines at least one kernel that is compatible with at least one of the devices in devs, and that kernel can be represented in a device image of state State.
- If State is bundle_state::input, all devices in devs have aspect::online_compiler.
- If State is bundle_state::object, all devices in devs have aspect::online_linker.

Throws:

- An exception with the errc::invalid error code if any of the devices in devs is not one of devices contained by the context ctxt or is not a descendant device of some device in ctxt.
- An exception with the errc::invalid error code if the devs vector is empty.
template <bundle_state State>
bool has_kernel_bundle(const context& ctxt, const std::vector<device>& devs, 
                       const std::vector<kernel_id>& kernelIds);

Returns: true only if all of the following are true:
• Each of the kernels in kernelIds can be represented in a device image of state State.
• Each of the kernels in kernelIds is compatible with at least one of the devices in devs.
• If State is bundle_state::input, all devices in devs have aspect::online_compiler.
• If State is bundle_state::object, all devices in devs have aspect::online_linker.

Throws:
• An exception with the errc::invalid error code if any of the devices in devs is not one of devices contained by the context ctxt or is not a descendent device of some device in ctxt.
• An exception with the errc::invalid error code if the devs vector is empty.

1. Equivalent to has_kernel_bundle(ctxt, ctxt.get_devices()).
2. Equivalent to has_kernel_bundle<State>(ctxt, ctxt.get_devices(), kernelIds).

Preconditions: The template parameter KernelName must be the type kernel name of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a KernelName in their kernel invocation command in order to use these functions. Applications which call these functions for a KernelName that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. Equivalent to has_kernel_bundle<State>(ctxt, {get_kernel_id<KernelName>()}).
2. Equivalent to has_kernel_bundle<State>(ctxt, devs, {get_kernel_id<KernelName>()}).

4.11.9. Querying if a kernel is compatible with a device

The following free functions allow an application to test whether a particular kernel is compatible with a device. A kernel that is defined in the application is compatible with a device unless:
• It uses optional features which are not supported on the device, as described in Section 5.7; or
• It is decorated with a [[sycl::device_has()]] C++ attribute that lists an aspect that is not supported by the device, as described in Section 5.8.1.
A device built-in kernel is only compatible with the device for which it is built-in.

```cpp
bool is_compatible(const std::vector<kernel_id>& kernelIds, const device& dev);
```

**Returns:** true if all of the kernels identified by `kernelIds` are compatible with the device `dev`.

```cpp
template <typename KernelName> bool is_compatible(const device& dev);
```

**Preconditions:** The template parameter `KernelName` must be the type kernel name of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a `KernelName` in their kernel invocation command in order to use this function. Applications which call this function for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

Equivalent to `is_compatible<State>((get_kernel_id<KernelName>()), dev)`.

### 4.11.10. Joining kernel bundles

Two or more kernel bundles of the same state may be joined together into a single composite bundle. Joining bundles together is not the same as online compiling or linking because it produces a new bundle in the same state as its inputs. Rather, joining creates the union of all the devices images from the input bundles, eliminates duplicate copies of the same device image, and creates a new bundle from the result.

```cpp
template <bundle_state State>
kernel_bundle<State> join(const std::vector<kernel_bundle<State>>& bundles);
```

**Returns:** A new kernel bundle that contains a copy of all the device images in the input `bundles` with duplicates removed. The new bundle has the same associated context and the same set of associated devices as those in `bundles`.

**Throws:**

- An exception with the errc::invalid error code if the bundles in `bundles` do not all have the same associated context or do not all have the same set of associated devices.

### 4.11.11. Online compiling and linking

If the implementation provides an online compiler or linker, a SYCL application can use the free functions defined in this section to transform a kernel bundle from `bundle_state::input` into a bundle of state `bundle_state::object` or to transform a bundle from `bundle_state::object` into a bundle of state `bundle_state::executable`.

An application can query whether the implementation provides an online compiler or linker by querying a device for `aspect::online_compiler` or `aspect::online_linker`.

All of the functions in this section accept a `property_list` parameter, which can affect the semantics of the compilation or linking operation. The core SYCL specification does not currently define any such properties, but vendors may specify these properties as an extension.

```cpp
kernel_bundle<bundle_state::object>
compile(const kernel_bundle<bundle_state::input>& inputBundle,
        const std::vector<device>& devs, const property_list& propList = {});
```
**Effects:** The device images from `inputBundle` are translated into one or more new device images of state `bundle_state::object`, and a new kernel bundle is created to contain these new device images. The new bundle represents all of the kernels in `inputBundles` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices `devs`) are not compiled and not represented in the new kernel bundle.

The new bundle has the same associated context as `inputBundle`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

**Returns:** The new kernel bundle.

**Throws:**

- An exception with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for `inputBundle` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An exception with the `errc::build` error code if the online compile operation fails.

```cpp
kernel_bundle<bundle_state::executable>
link(const std::vector<kernel_bundle<bundle_state::object>>& objectBundles,
     const std::vector<device>& devs, const property_list& propList = {});
```

**Effects:** Duplicate device images from `objectBundles` are eliminated as though they were joined via `join()`, then the remaining device images are translated into one or more new device images of state `bundle_state::executable`, and a new kernel bundle is created to contain these new device images. The new bundle represents all of the kernels in `objectBundles` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices in `devs`) are not linked and not represented in the new bundle.

The new bundle has the same associated context as those in `objectBundles`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

**Returns:** The new kernel bundle.

**Throws:**

- An exception with the `errc::invalid` error code if the bundles in `objectBundles` do not all have the same associated context.
- An exception with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for any of the bundles in `objectBundles` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An exception with the `errc::build` error code if the online link operation fails.

```cpp
kernel_bundle<bundle_state::executable>
build(const kernel_bundle<bundle_state::input>& inputBundle,
      const std::vector<device>& devs, const property_list& propList = {});
```

**Effects:** This function performs both an online compile and link operation, translating a kernel bundle of state `bundle_state::input` into a bundle of state `bundle_state::executable`. The device images from `inputBundle` are translated into one or more new device images of state `bundle_state::executable`, and a new bundle is created to contain these new device images. The new bundle represents all of the kernels in `inputBundle` that are compatible with at least one of the devices in `devs`. Any remaining kernels (those that are not compatible with any of the devices `devs`) are not compiled or linked and are not represented in the new bundle.
The new bundle has the same associated context as `inputBundle`, and the new bundle's set of associated devices is `devs` (with duplicate devices removed).

**Returns:** The new kernel bundle.

**Throws:**

- An exception with the `errc::invalid` error code if any of the devices in `devs` are not in the set of associated devices for `inputBundle` (as defined by `kernel_bundle::get_devices()`) or if the `devs` vector is empty.
- An exception with the `errc::build` error code if the online compile or link operations fail.

```cpp
kernel_bundle< bundle_state::object> // (1)
compile( const kernel_bundle< bundle_state::input>& inputBundle,
         const property_list& propList = {});

kernel_bundle< bundle_state::executable> // (2)
link( const kernel_bundle< bundle_state::object>& objectBundle,
      const std::vector<device>& devs,
      const property_list& propList = {});

kernel_bundle< bundle_state::executable> // (3)
link( const std::vector<kernel_bundle< bundle_state::object>>& objectBundles,
      const property_list& propList = {});

kernel_bundle< bundle_state::executable> // (4)
link( const kernel_bundle< bundle_state::object>& objectBundle,
      const property_list& propList = {});

kernel_bundle< bundle_state::executable> // (5)
build( const kernel_bundle< bundle_state::input>& inputBundle,
      const property_list& propList = {});
```

1. Equivalent to `compile(inputBundle, inputBundle.get_devices(), propList)`.
2. Equivalent to `link({objectBundle}, devs, propList)`.
3. Equivalent to `link(objectBundles, devs, propList)`, where `devs` is the intersection of associated devices in common for all bundles in `objectBundles`.
4. Equivalent to `link({objectBundle}, objectBundle.get_devices(), propList)`.
5. Equivalent to `build(inputBundle, inputBundle.get_devices(), propList)`.

### 4.11.12. The `kernel_bundle` class

A synopsis of the `kernel_bundle` class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

As with all SYCL objects that have the common reference semantics, kernel bundles are equality comparable. Two bundles of the same `bundle state` are considered to be equal if they are associated with the same context, have the same set of associated devices, and contain the same set of device images.

There is no public default constructor for this class.

```cpp
namespace sycl {

class kernel { /* ... */
```
template <bundle_state State> class kernel_bundle {
  public:
    using device_image_iterator = __unspecified__;

    kernel_bundle() = delete;

    bool empty() const noexcept;

    backend get_backend() const noexcept;

    context get_context() const noexcept;

    std::vector<device> get_devices() const noexcept;

    bool has_kernel(const kernel_id& kernelId) const noexcept;

    bool has_kernel(const kernel_id& kernelId, const device& dev) const noexcept;

    template <typename KernelName> bool has_kernel() const noexcept;

    template <typename KernelName> bool has_kernel(const device& dev) const noexcept;

    std::vector<kernel_id> get_kernel_ids() const;

    /* Available only when: (State == bundle_state::executable) */
    kernel get_kernel(const kernel_id& kernelId) const;

    /* Available only when: (State == bundle_state::executable) */
    template <typename KernelName> kernel get_kernel() const;

    bool contains_specialization_constants() const noexcept;

    bool native_specialization_constant() const noexcept;

    template <auto& SpecName> bool has_specialization_constant() const noexcept;

    /* Available only when: (State == bundle_state::input) */
    template <auto& SpecName>
    void set_specialization_constant(
        typename std::remove_reference_t<decltype(SpecName)>::value_type value);

    template <auto& SpecName>
    typename std::remove_reference_t<decltype(SpecName)>::value_type
    get_specialization_constant() const;

    device_image_iterator begin() const;

    device_image_iterator end() const;
};

// namespace sycl
4.11.12.1. Queries

The following member functions provide various queries for a kernel bundle.

```cpp
bool empty() const noexcept;
```

**Returns:** `true` only if the kernel bundle contains no device images.

```cpp
backend get_backend() const noexcept;
```

**Returns:** The backend that is associated with the kernel bundle.

```cpp
context get_context() const noexcept;
```

**Returns:** The context that is associated with the kernel bundle.

```cpp
std::vector<device> get_devices() const noexcept;
```

**Returns:** The set of devices that is associated with the kernel bundle.

```cpp
bool has_kernel(const kernel_id& kernelId) const noexcept; // (1)
bool has_kernel(const kernel_id& kernelId, const device& dev) const noexcept; // (2)
```

1. **Returns:** `true` only if the kernel bundle contains the kernel identified by `kernelId`.
2. **Returns:** `true` only if the kernel bundle contains the kernel identified by `kernelId` and if that kernel is compatible with the device `dev`.

```cpp
template<typename KernelName> bool has_kernel() const noexcept; // (1)
template<typename KernelName>
bool has_kernel(const device& dev) const noexcept; // (2)
```

**Preconditions:** The template parameter `KernelName` must be the type kernel name of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a `KernelName` in their kernel invocation command in order to use these functions. Applications which call these functions for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

1. **Returns:** `true` only if the kernel bundle contains the kernel identified by `KernelName`.
2. **Returns:** `true` only if the kernel bundle contains the kernel identified by `KernelName` and if that kernel is compatible with the device `dev`.

```cpp
std::vector<kernel_id> get_kernel_ids() const;
```

**Returns:** A vector of the identifiers for all kernels that are contained in the kernel bundle.

```cpp
kernel get_kernel(const kernel_id& kernelId) const;
```
**Preconditions:** This member function is only available if the kernel bundle’s state is `bundle_state::executable`.

**Returns:** A kernel object representing the kernel identified by `kernelId`, which resides in the bundle.

**Throws:**
- An exception with the `errc::invalid` error code if the kernel bundle does not contain the kernel identified by `kernelId`.

```cpp
template<typename KernelName> kernel get_kernel() const;
```

**Preconditions:** This member function is only available if the kernel bundle’s state is `bundle_state::executable`. The template parameter `KernelName` must be the type `kernel name` of a kernel that is defined in the SYCL application. Since lambda functions have no standard type name, kernels defined as lambda functions must specify a `KernelName` in their kernel invocation command in order to use this function. Applications which call this function for a `KernelName` that is not defined are ill formed, and the implementation must issue a diagnostic in this case.

**Returns:** A kernel object representing the kernel identified by `KernelName`, which resides in the bundle.

**Throws:**
- An exception with the `errc::invalid` error code if the kernel bundle does not contain the kernel identified by `KernelName`.

### 4.11.12.2. Specialization constant support

The following member functions allow an application to manipulate specialization constants that are used in the device images of a kernel bundle. Applications can set the value of specialization constants in a kernel bundle whose state is `bundle_state::input` and then online compile that bundle into `bundle_state::object` or `bundle_state::executable`. The value of the specialization constants then become fixed in the compiled bundle and cannot be changed. Specialization constants that have not had their values set by the time the bundle is compiled take their default values.

It is expected that many implementations will use an intermediate language representation for a bundle in state `bundle_state::input` such as SPIR-V, and the intermediate language will have native support for specialization constants. However, implementations that do not have such native support must still support specialization constants in some other way.

```cpp
bool contains_specialization_constants() const noexcept;
```

**Returns:** `true` only if the kernel bundle contains at least one device image which uses a specialization constant.

```cpp
bool native_specialization_constant() const noexcept;
```

**Returns:** `true` only if the kernel bundle contains at least one device image which uses a specialization constant and all specialization constants used in all of the bundle’s device images are native specialization constants.

```cpp
template<auto& SpecName> bool has_specialization_constant() const noexcept;
```
Returns: true if any device image in the kernel bundle uses the specialization constant whose address is SpecName.

```cpp
template <auto& SpecName>
void set_specialization_constant(
    typename std::remove_reference_t<decltype(SpecName)>::value_type value);
```

Preconditions: This member function is only available if the kernel bundle's state is bundle_state::input.

Effects: Sets the value of the specialization constant whose address is SpecName for this bundle. If the specialization constant's value was previously set in this bundle, the value is overwritten.

The new value applies to all device images in the bundle. It is allowed to set the value of a specialization constant even if no device image in the bundle uses it; doing so has no effect on the execution of kernels from that bundle.

```cpp
template <auto& SpecName>
typename std::remove_reference_t<decltype(SpecName)>::value_type
get_specialization_constant() const;
```

Returns: The value of the specialization constant whose address is SpecName for this kernel bundle. The value returned is as follows:

- If the value of this specialization constant was previously set in this bundle, that value is returned. Otherwise,
- If this bundle is the result of compiling, linking or joining another bundle and this specialization constant was set in that other bundle prior to compiling, linking or joining; then that value is returned. Otherwise,
- The specialization constant's default value is returned.

4.11.12.3. Device image support

The following member type and functions allow iteration over the device images contained by the kernel bundle.

```cpp
using device_image_iterator = __unspecified__;
```

An iterator type that satisfies the C++ requirements of LegacyForwardIterator. The iterator's referenced type is const device_image<State>, where State is the same state as the containing kernel_bundle.

```cpp
device_image_iterator begin() const; // (1)
device_image_iterator end() const; // (2)
```

1. Returns: An iterator to the first device image contained by the kernel bundle.
2. Returns: An iterator to one past the last device image contained by the kernel bundle.

4.11.13. The kernel class

A synopsis of the kernel class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in Section 4.5.2 in Table 7 and Table 8, respectively.
There is no public default constructor for this class.

```cpp
namespace sycl {
    class kernel {
        public:
            kernel() = delete;
            backend get_backend() const noexcept;
            context get_context() const;
            kernel_bundle<bundle_state::executable> get_kernel_bundle() const;
            template <typename Param> typename Param::return_type get_info() const;
            template <typename Param> typename Param::return_type get_info(const device& dev) const;
            template <typename Param> typename Param::return_type get_backend_info() const;
    } // namespace sycl
}
```

### 4.11.13.1. Queries

The following member functions provide various queries for a `kernel`.

- `backend get_backend() const noexcept;`
  - **Returns:** The backend associated with this kernel.

- `context get_context() const;`
  - **Returns:** The context associated with this kernel.

- `kernel_bundle<bundle_state::executable> get_kernel_bundle() const;`
  - **Returns:** The kernel bundle that contains this kernel.

- `template <typename Param> typename Param::return_type get_info() const;`
  - **Preconditions:** The `Param` must be one of the `info::kernel` descriptors defined in Table 137, and the type alias `Param::return_type` must be defined in accordance with that table.
  - **Returns:** Information about the kernel that is not specific to the device on which it is invoked.
4.11.13.2. Kernel information descriptors

A kernel can be queried for information using the `get_info()` member function, specifying one of the info parameters in `info::kernel`. All info parameters in `info::kernel` are specified in Table 137 and the synopsis for `info::kernel` is described in Section A.5.

**Table 137. Kernel class information descriptors**

<table>
<thead>
<tr>
<th>Kernel Descriptors</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::kernel::num_args</code></td>
<td><code>uint32_t</code></td>
<td>This descriptor may only be used to query a kernel that resides in a kernel bundle that was constructed using a backend specific interoperability function or to query a device built-in kernel, and the semantics of this descriptor are defined by each SYCL backend specification. Attempting to use this descriptor for other kernels throws an exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>info::kernel::attributes</code></td>
<td><code>std::string</code></td>
<td>Return any attributes specified on a kernel function (as defined in Section 5.8).</td>
</tr>
</tbody>
</table>

A kernel can also be queried for device specific information using the `get_info()` member function, specifying one of the info parameters in `info::kernel_device_specific`. All info parameters in `info::kernel_device_specific` are specified in Table 138. The synopsis for `info::kernel_device_specific` is described in Section A.5.

**Table 138. Device-specific kernel information descriptors**
### Device-specific Kernel Information Descriptors

<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Return type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::kernel_device_specific::global_work_size</code></td>
<td><code>range&lt;3&gt;</code></td>
<td>This descriptor may only be used if the device type is <code>device_type::custom</code> or if the kernel is a built-in kernel. The exact semantics of this descriptor are defined by each SYCL backend specification, but the intent is to return the kernel’s maximum global work size. Attempting to use this descriptor for other devices or kernels throws an exception with the <code>errc::invalid</code> error code.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::work_group_size</code></td>
<td><code>size_t</code></td>
<td>Returns the maximum number of work-items in a work-group that can be used to execute a kernel on a specific device.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::compile_work_group_size</code></td>
<td><code>range&lt;3&gt;</code></td>
<td>Returns the work-group size specified by the device compiler if applicable, otherwise returns <code>{0,0,0}</code>.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::referred_work_group_size_multiple</code></td>
<td><code>size_t</code></td>
<td>Returns a value, of which work-group size is preferred to be a multiple, for executing a kernel on a particular device. This is a performance hint. The value must be less than or equal to that returned by <code>info::kernel_device_specific::work_group_size</code>.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::private_mem_size</code></td>
<td><code>size_t</code></td>
<td>Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variables declared inside the kernel in the private address space.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::max_num_sub_groups</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum number of sub-groups for this kernel.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::compile_num_sub_groups</code></td>
<td><code>uint32_t</code></td>
<td>Returns the number of sub-groups specified by the kernel, or 0 (if not specified).</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::max_sub_group_size</code></td>
<td><code>uint32_t</code></td>
<td>Returns the maximum sub-group size for this kernel.</td>
</tr>
<tr>
<td><code>info::kernel_device_specific::compile_sub_group_size</code></td>
<td><code>uint32_t</code></td>
<td>Returns the required sub-group size specified by the kernel, or 0 (if not specified).</td>
</tr>
</tbody>
</table>
4.11.14. The device_image class

A synopsis of the device_image class is shown below. Additionally, this class provides the common special member functions and common member functions that are listed in Section 4.5.2 in Table 7 and Table 8, respectively.

```cpp
namespace sycl {

    template <bundle_state State> class device_image {
        public:
            device_image() = delete;

            bool has_kernel(const kernel_id& kernelId) const noexcept;
            bool has_kernel(const kernel_id& kernelId, const device& dev) const noexcept;
    }
}
```

There is no public constructor for this class.

1. `Returns: true` only if the device image contains the kernel identified by `kernelId`.
2. `Returns: true` only if the device image contains the kernel identified by `kernelId` and if that kernel is compatible with the device `dev`.

4.11.15. Example usage

This section provides some examples showing typical use cases for kernel bundles. These examples are intended to clarify the definition of the kernel bundle interfaces, but the content of this section is non-normative.

4.11.15.1. Controlling the timing of online compilation

In some cases an application may want to pre-compile its kernels before submitting them to a device. This gives the application control over when the overhead of online compilation happens, rather than relying on the default behavior (which may cause the online compilation to happen at the point when the kernel is submitted to a device). The following example shows how this can be achieved.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
    queue myQueue;
    auto myContext = myQueue.get_context();

    // This call to get_kernel_bundle() forces an online compilation of all the
    // application's kernels for the device in "myContext", unless those kernels
    // were already compiled for that device by the ahead-of-time compiler.
    auto myBundle = get_kernel_bundle<bundle_state::executable>(myContext);
}```
myQueue.submit([handler& cgh] {
    // Calling use_kernel_bundle() causes the parallel_for() below to use the
    // pre-compiled kernel from "myBundle".
    cgh.use_kernel_bundle(myBundle);
    cgh.parallel_for(range { 1024 }, [=](item index) {
        // kernel code
    });
    myQueue.wait();
})

4.11.15.2. Specialization constants

An application can use a kernel bundle to set the values of specialization constants in several kernels
before any of them are submitted for execution.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl:" before SYCL names

// Forward declare names for our two kernels.
class MyKernel1;
class MyKernel2;

extern int get_width();
extern int get_height();

// Declare specialization constants used in our kernels.
constexpr specialization_id<int> width;
constexpr specialization_id<int> height;

int main() {
    queue myQueue;
    auto myContext = myQueue.get_context();

    // Get the identifiers for our kernels, then get an input kernel bundle that
    // contains our two kernels.
    auto kernelIds = { get_kernel_id<MyKernel1>(), get_kernel_id<MyKernel2>() };
    auto inputBundle =
        get_kernel_bundle<bundle_state::input>(myContext, kernelIds);

    // Set the values of the specialization constants.
    inputBundle.set_specialization_constant(width)(get_width());
    inputBundle.set_specialization_constant(height)(get_height());

    // Build the kernel bundle into an executable form. The values of the
    // specialization constants are compiled in.
    auto exeBundle = build(inputBundle);

    myQueue.submit([handler& cgh] {
        // Use the kernel bundle we built in this command group.
        cgh.use_kernel_bundle(exeBundle);
        cgh.parallel_for<MyKernel1>();
    });
```
4.11.15.3. Kernel introspection

Applications can use kernel bundles to introspect its kernels and use that information to tune the arguments passed when invoking it.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl::" before SYCL names

class MyKernel; // Forward declare the name of our kernel.

int main() {
    size_t N = 1024;
    queue myQueue;
    auto myContext = myQueue.get_context();
    auto myDev = myQueue.get_device();

    // Get an executable kernel bundle containing our kernel.
    kernel_id kernelId = get_kernel_id<MyKernel>();
    auto myBundle =
        get_kernel_bundle<bundle_state::executable>(myContext, { kernelId });

    // Get the kernel's maximum work group size when running on our device.
    kernel myKernel = myBundle.get_kernel(kernelId);
    size_t maxWgSize =
        myKernel.get_info<info::kernel_device_specific::work_group_size>(myDev);

    // Compute a good ND-range to use for iteration in the kernel
    // based on the maximum work group size.
    std::array<size_t, 11> divisors = { 1024, 512, 256, 128, 64, 32,
                                        16,  8,  4,  2,  1 };
    size_t wgSize = *std::find_if(divisors.begin(), divisors.end(),
                                   [=](auto d) { return (d <= maxWgSize); });
    nd_range myRange { range { N }, range { wgSize } };

    myQueue.submit([&](handler &cgh) {
        // This command group uses the same kernel bundle.
        cgh.use_kernel_bundle(exeBundle);
        cgh.parallel_for<MyKernel2>(
            range { 1024 }, [
                item = item index, kernel_handler kh = kh {
                int w = kh.get_specialization_constant<width>();
                // ...
            });

        myQueue.submit([&](handler &cgh) {
            cgh.parallel_for<MyKernel2>(
                range { 1024 }, [
                    item = item index, kernel_handler kh = kh {
                int h = kh.get_specialization_constant<height>();
                // ...
            });
        }));

        myQueue.wait();
    });
}  
```
4.11.15.4. Invoking a device built-in kernel

An application can use kernel bundles to invoke a device’s built-in kernels.

```cpp
#include <sycl/sycl.hpp>
using namespace sycl; // (optional) avoids need for "sycl:" before SYCL names

int main() {
    queue myQueue;
    auto myContext = myQueue.get_context();
    auto myDevice = myQueue.get_device();
    
    const std::vector<kernel_id> builtinKernelIds =
        myDevice.get_info<info::device::built_in_kernel_ids>();

    // Get an executable kernel bundle containing all the built-in kernels
    // supported by the device.
    kernel_bundle<bundle_state::executable> myBundle =
        get_kernel_bundle(myContext, { myDevice }, builtinKernelIds);

    // Retrieve a kernel object that can be used to query for more information
    // about the built-in kernel or to submit it to a command group. We assume
    // here that the device supports at least one built-in kernel.
    kernel builtinKernel = myBundle.get_kernel(builtinKernelIds[0]);

    // Submit the built-in kernel.
    myQueue.submit([&](handler& cgh) {
        // Setting the arguments depends on the backend and the exact kernel used.
        cgh.set_args(...);
        cgh.parallel_for(range { 1024 }, builtinKernel);
    });
    
    myQueue.wait();
}
```

4.12. Defining kernels

In SYCL, functions that are executed on a SYCL device are referred to as SYCL kernel functions. A kernel containing such a SYCL kernel function is enqueued on a device queue in order to be executed on that particular device.

The return type of the SYCL kernel function is void, and all memory accesses between host and device are through accessors or through USM pointers.
There are two ways of defining kernels: as named function objects or as lambda functions. A backend may also provide interoperability interfaces for defining kernels.

4.12.1. Defining kernels as named function objects

A kernel can be defined as a named function object type. These function objects provide the same functionality as any C++ function object, with the restriction that they need to follow SYCL rules to be device copyable. The kernel function can be templated via templating the kernel function object type. For details on restrictions for kernel naming, please refer to Section 5.2.

The `operator()` member function must be const-qualified, and it may take different parameters depending on the data accesses defined for the specific kernel. If the `operator()` function writes to any of the member variables, the behavior is undefined.

The following example defines a SYCL kernel function, RandomFiller, which initializes a buffer with a random number. The random number is generated during the construction of the function object while processing the command group. The `operator()` member function of the function object receives an item object. This member function will be called for each work item of the execution range. The value of the random number will be assigned to each element of the buffer. In this case, the accessor and the scalar random number are members of the function object and therefore will be arguments to the device kernel. Usual restrictions of passing arguments to kernels apply.

```cpp
class RandomFiller {
    public:
        RandomFiller(accessor<int> ptr) :
            ptr_(ptr) {
            std::random_device hwRand;
            std::uniform_int_distribution<> r(1, 100);
            randomNum_ = r(hwRand);
        }
        void operator()(item<int> item) const { ptr_[item.get_id()] = get_random(); }
        int get_random() { return randomNum_; }
    private:
        accessor<int> ptr_; 
        int randomNum_; 
};

void workFunction(buffer<int, 1>& b, queue& q, const range<1> r) {
    myQueue.submit([&](handler& cgh) {
        accessor ptr { buf, cgh }; 
        RandomFiller filler { ptr }; 
        cgh.parallel_for(r, filler);
    });
}
```

4.12.2. Defining kernels as lambda functions

In C++, function objects can be defined using lambda functions. Kernels may be defined as lambda functions in SYCL. The name of a lambda function in SYCL may optionally be specified by passing it as a template parameter to the invoking member function, and in that case, the lambda name is a C++ typename which must be forward declarable at namespace scope. If the lambda function relies on template arguments, then if specified, the name of the lambda function must contain those template arguments which must also be forward declarable at namespace scope. The class used for the name of a lambda function...
is only used for naming purposes and is not required to be defined. For details on restrictions for kernel naming, please refer to Section 5.2.

The kernel function for the lambda function is the lambda function itself. The kernel lambda must use copy for all of its captures (i.e. `&`), and the lambda must not use the mutable specifier.

```
// Explicit kernel names can be optionally forward declared at namespace scope
class MyKernel;

myQueue.submit([8](handler8 h) {
  // Explicitly name kernel with previously forward declared type
  h.single_task<MyKernel>();
  // [kernel code]
});

// Explicitly name kernel without forward declaring type at
// namespace scope. Must still be forward declarable at
// namespace scope, even if not declared at that scope
h.single_task<class MyOtherKernel>();
  // [kernel code]
});
```

Explicit lambda naming is shown in the following code example, including an illegal case that uses a class within the kernel name which is not forward declarable (`std::complex`).

```
// Explicit kernel names can be optionally forward declared at namespace scope
class MyForwardDeclName;

template <typename T> class MyTemplatedKernelName;

// Define and launch templated kernel
template <typename T> void templatedFunction() {
  queue myQueue;

  // Launch A: No explicit kernel name
  myQueue.submit([8](handler8 h) {
    h.single_task();
    // [kernel code that depends on type T]
  });

  // Launch B: Name the kernel when invoking (this is optional)
  myQueue.submit([8](handler8 h) {
    h.single_task<MyTemplatedKernelName<T>>();
    // The provided kernel name (MyTemplatedKernelName<T>) depends on T
    // because the kernel does. T must also be forward declarable at
    // namespace scope.
    // [kernel code that depends on type T]
  });
}

int main() {
```
queue myQueue;

myQueue.submit([8](handler8 h) {
    // Declare MyKernel within this kernel invocation. Legal because
    // forward declaration at namespace scope is optional
    h.single_task<class MyKernel>[] {
        // [kernel code]
    });
});

myQueue.submit([8](handler8 h) {
    // Use kernel name that was forward declared at namespace scope
    h.single_task<MyForwardDeclName>[] {
        // [kernel code]
    });
});

templatedFunction<int>(); // OK

templatedFunction<std::complex<float>>(); // Launch A is OK, Launch B illegal
// because std::complex is not forward declarable according to C++, and was
// used in an explicit kernel name which must be forward declarable.

4.12.3. is_device_copyable type trait

namespace sycl {
    template<typename T>
    struct is_device_copyable;

    template<typename T>
    inline constexpr bool is_device_copyable_v = is_device_copyable<T>::value;
};

is_device_copyable is a user specializable class template to indicate that a type T is device copyable.

- is_device_copyable must meet the Cpp17UnaryTrait requirements.
- If is_device_copyable is specialized such that is_device_copyable_v<T> == true on a T that does not satisfy all the requirements of a device copyable type, the results are unspecified.

If the application defines a type UDT that satisfies the requirements of a device copyable type (as defined in Section 3.13.1) but the type is not implicitly device copyable as defined in that section, then the application must provide a specialization of is_device_copyable that derives from std::true_type in order to use that type in a context that requires a device copyable type. Such a specialization can be declared like this:

    template<>
    struct sycl::is_device_copyable<UDT> : std::true_type {};

It is legal to provide this specialization even if the implementation does not define SYCL_DEVICE_COPYABLE to 1, but the type cannot be used as a device copyable type in that case and the specialization is ignored.
4.12.4. Rules for parameter passing to kernels

A SYCL application passes parameters to a kernel in different ways depending on whether the kernel is a named function object or a lambda function. If the kernel is a named function object, the `operator()` member function (or other member functions that it calls) may reference member variables inside the same named function object. Any such member variables become parameters to the kernel. If the kernel is a lambda function, any variables captured by the lambda become parameters to the kernel.

Regardless of how the parameter is passed, the following rules define the allowable types for a kernel parameter:

- Any **device copyable** type is a legal parameter type.
- The following SYCL types are legal parameter types:
  - `accessor` when templated with `target::device`;
  - `accessor` when templated with any of the deprecated parameters: `target::global_buffer`, `target::constant_buffer`, or `target::local`;
  - `local_accessor`;
  - `unsampled_image_accessor` when templated with `image_target::device`;
  - `sampled_image_accessor` when templated with `image_target::device`;
  - `stream`;
  - `id`;
  - `range`;
  - `marray<T, NumElements>` when `T` is **device copyable**;
  - `vec<T, NumElements>`.
- An array of element types `T` is a legal parameter type if `T` is a legal parameter type.
- A class type `S` with a non-static member variable of type `T` is a legal parameter type if `T` is a legal parameter type and if `S` would otherwise be a legal parameter type aside from this member variable.
- A class type `S` with a non-virtual base class of type `T` is a legal parameter type if `T` is a legal parameter type and if `S` would otherwise be a legal parameter type aside from this base class.

Pointer types are trivially copyable, so they may be passed as kernel parameters. However, only the pointer value itself is passed to the kernel. Dereferencing the pointer on the kernel results in undefined behavior unless the pointer points to an address within a USM memory region that is accessible on the device.

Reference types are not trivially copyable, so they may not be passed as kernel parameters.

The `reducer` class is a special type of kernel parameter which is passed to a kernel in a different way. Section 4.9.2 describes how this parameter type is used.

4.13. Error handling

4.13.1. Error handling rules

Error handling in a SYCL application (host code) uses C++ exceptions. If an error occurs, it will be thrown by the API function call and may be caught by the user through standard C++ exception handling mechanisms.

SYCL applications are asynchronous in the sense that host and device code executions are decoupled.
from one another except at specific points. For example, device code executions often begin when dependencies in the SYCL task graph are satisfied, which occurs asynchronously from host code execution. As a result of this the errors that occur on a device cannot be thrown directly from a host API call, because the call enqueuing a device action has typically already returned by the time that the error occurs. Such errors are not detected until the error-causing task executes or tries to execute, and we refer to these as asynchronous errors.

4.13.1.1. Asynchronous error handler

The queue and context classes can optionally take an asynchronous handler object async_handler on construction, which is a callable such as a function class or lambda, with an exception_list as a parameter. Invocation of an async_handler may be triggered by the queue member functions queue::wait_and_throw() or queue::throw_asynchronous(), by the event member function event::wait_and_throw(), or automatically on destruction of a queue or context that contains unconsumed asynchronous errors. When invoked, an async_handler is called and receives an exception_list argument containing a list of exception objects representing any unconsumed asynchronous errors associated with the queue or context.

When an asynchronous error instance has been passed to an async_handler, then that instance of the error has been consumed for handling and is not reported on any subsequent invocations of the async_handler.

The async_handler may be a named function object type, a lambda function or a std::function. The exception_list object passed to the async_handler is constructed by the SYCL runtime.

4.13.1.2. Behavior without an async_handler

If an asynchronous error occurs in a queue or context that has no user-supplied asynchronous error handler object async_handler, then an implementation-defined default async_handler is called to handle the error in the same situations that a user-supplied async_handler would be, as defined in Section 4.13.1.1. The default async_handler must in some way report all errors passed to it, when possible, and must then invoke std::terminate or equivalent.

4.13.1.3. Priorities of async handlers

If the SYCL runtime can associate an asynchronous error with a specific queue, then:

- If the queue was constructed with an async_handler, that handler is invoked to handle the error.
- Otherwise if the context enclosed by the queue was constructed with an async_handler, that handler is invoked to handle the error.
- Otherwise when no handler was passed to either queue or context on construction, then a default handler is invoked to handle the error, as described by Section 4.13.1.2.
- All handler invocations in this list occur at times as defined by Section 4.13.1.1.

If the SYCL runtime cannot associate an asynchronous error with a specific queue, then:

- If the context in which the error occurred was constructed with an async_handler, then that handler is invoked to handle the error.
- Otherwise when no handler was passed to the associated context on construction, then a default handler is invoked to handle the error, as described by Section 4.13.1.2.
- All handler invocations in this list occur at times as defined by Section 4.13.1.1.

4.13.1.4. Asynchronous errors with a secondary queue

If an asynchronous error occurs when running or enqueuing a command group which has a secondary queue specified, then the command group may be enqueued to the secondary queue instead of the pri-
mary queue. The error handling in this case is also configured using the async_handler provided for both queues. If there is no async_handler given on any of the queues, then the asynchronous error handling proceeds through the contexts associated with the queues, and if they were also constructed without async_handlers, then the default handler will be used. If the primary queue fails and there is an async_handler given at this queue’s construction, which populates the exception_list parameter, then any errors will be added and can be thrown whenever the user chooses to handle those exceptions. Since there were errors on the primary queue and a secondary queue was given, then the execution of the kernel is re-scheduled to the secondary queue and any error reporting for the kernel execution on that queue is done through that queue, in the same way as described above. The secondary queue may fail as well, and the errors will be thrown if there is an async_handler and either wait_and_throw() or throw() are called on that queue. If no async_handler was specified, then the one associated with the queue’s context will be used and if the context was also constructed without an async_handler, then the default handler will be used. The command group function object event returned by that function will be relevant to the queue where the kernel has been enqueued.

Below is an example of catching a SYCL exception and printing out the error message.

```cpp
void catch_any_errors(sycl::context const & ctx) {
    try {
        do_something_to_invoke_error(ctx);
    } catch (sycl::exception const & e) {
        std::cerr << e.what();
    }
}
```

Below is an example of catching a SYCL exception with the errc::invalid error code and printing out the error message.

```cpp
void catch_invalid_errors(sycl::context const & ctx) {
    try {
        do_something_to_invoke_error(ctx);
    } catch (sycl::exception const & e) {
        if (e.code().value() == sycl::errc::invalid)
            std::cerr << "Invalid error: " << e.what();
        else
            throw;
    }
}
```

Below is an example of catching a SYCL exception, checking for the SYCL backend by inspecting the category and handling the OpenCL SYCL backend error codes if the category is that of the OpenCL SYCL backend otherwise checking the standard error code.

```cpp
void catch_backend_errors(sycl::context const & ctx) {
    try {
        do_something_to_invoke_error(ctx);
    } catch (sycl::exception const & e) {
        if (e.category() == sycl::error_category_for<sycl::backend::opencl>()) {
            switch (e.code().value()) {
                case CL_INVALID_PROGRAM:
                    std::cerr << "OpenCL invalid program error: " << e.what();
                    /* ...*/
            }
        }
    }
}
```
else {
    throw;
}
} else {
    if (e.code() == sycl::errc::invalid) {
        std::cerr << "Invalid error: " << e.what();
    } else {
        throw;
    }
}
} }

4.13.2. Exception class interface

namespace sycl {

    using async_handler = std::function<void(sycl::exception_list)>;

    class exception : public virtual std::exception {
        public:
            exception(std::error_code ec, const std::string& what_arg);
            exception(std::error_code ec, const char* what_arg);
            exception(std::error_code ec);
            exception(int ev, const std::error_category& ecat, const std::string& what_arg);
            exception(int ev, const std::error_category& ecat, const char* what_arg);
            exception(int ev, const std::error_category& ecat);
            exception(context ctx, std::error_code ec, const std::string& what_arg);
            exception(context ctx, std::error_code ec, const char* what_arg);
            exception(context ctx, std::error_code ec);
            exception(context ctx, int ev, const std::error_category& ecat, const std::string& what_arg);
            exception(context ctx, int ev, const std::error_category& ecat, const char* what_arg);
            exception(context ctx, int ev, const std::error_category& ecat);
            exception(context ctx, std::error_code& code) const noexcept;
            exception(context ctx, std::error_category& category) const noexcept;
            const char* what() const;

    bool has_context() const noexcept;
    context get_context() const;
    
    };

    class exception_list {
        // Used as a container for a list of asynchronous exceptions
        public:
            using value_type = std::exception_ptr;
            using reference = value_type&;
            using const_reference = const value_type&;
            using size_type = std::size_t;
            using iterator = /*unspecified*/;

        
    

}
using const_iterator = /*unspecified*/;

size_type size() const;  // first asynchronous exception
iterator begin() const;  // refer to past-the-end last asynchronous exception

enum class errc : /* unspecified */ {
    success = 0,
    runtime,
    kernel,
    accessor,
    nd_range,
    event,
    kernel_argument,
    build,
    invalid,
    memory_allocation,
    platform,
    profiling,
    feature_not_supported,
    kernel_not_supported,
    backend_mismatch
};

template <backend b> using errc_for = typename backend_traits<b>::errc;

std::error_code make_error_code(errc e) noexcept;

const std::error_category& sycl_category() noexcept;

template <backend b> const std::error_category& error_category_for() noexcept;

} // namespace sycl

namespace std {

template <> struct is_error_code_enum</* see below */> : true_type {};

} // namespace std

The SYCL exception_list class is also available in order to provide a list of synchronous and asynchronous exceptions.

Errors can occur both in the SYCL library and SYCL host side, or may come directly from a SYCL backend. The member functions on these exceptions provide the corresponding information. SYCL backends can provide additional exception class objects as long as they derive from sycl::exception object, or any of its derived classes.

A specialization of std::is_error_code_enum must be defined for sycl::errc and backend_traits<Backend>::errc inheriting from std::true_type for each Backend, where backend is each enumeration of the enum class backend.

Table 139. Member functions of the SYCL exception class
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exception(std::error_code ec, const std::string&amp; what_arg)</code></td>
<td>Constructs an exception. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(std::error_code ec, const char* what_arg)</code></td>
<td>Constructs an exception. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(std::error_code ec)</code></td>
<td>Constructs an exception.</td>
</tr>
<tr>
<td><code>exception(int ev, const std::error_category&amp; ecat, const std::string&amp; what_arg)</code></td>
<td>Constructs an exception with the error code <code>ev</code> and the underlying error category <code>ecat</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(int ev, const std::error_category&amp; ecat, const char* what_arg)</code></td>
<td>Constructs an exception with the error code <code>ev</code> and the underlying error category <code>ecat</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(context ctx, std::error_code ec, const std::string&amp; what_arg)</code></td>
<td>Constructs an exception with an associated SYCL context <code>ctx</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(context ctx, std::error_code ec, const char* what_arg)</code></td>
<td>Constructs an exception with an associated SYCL context <code>ctx</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(context ctx, std::error_code ec)</code></td>
<td>Constructs an exception with an associated SYCL context <code>ctx</code>.</td>
</tr>
<tr>
<td><code>exception(context ctx, int ev, const std::error_category&amp; ecat, const std::string&amp; what_arg)</code></td>
<td>Constructs an exception with an associated SYCL context <code>ctx</code>, the error code <code>ev</code> and the underlying error category <code>ecat</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
<tr>
<td><code>exception(context ctx, int ev, const std::error_category&amp; ecat, const char* what_arg)</code></td>
<td>Constructs an exception with an associated SYCL context <code>ctx</code>, the error code <code>ev</code> and the underlying error category <code>ecat</code>. The string returned by <code>what()</code> is guaranteed to contain <code>what_arg</code> as a substring.</td>
</tr>
</tbody>
</table>
Table 140. Member functions of the exception_list

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exception(context ctx, int ev, const std::error_category&amp; ecat)</td>
<td>Constructs an exception with an associated SYCL context ctx, the error code ev and the underlying error category ecat.</td>
</tr>
<tr>
<td>const std::error_code&amp; code() const noexcept</td>
<td>Returns the error code stored inside the exception.</td>
</tr>
<tr>
<td>const std::error_category&amp; category() const noexcept</td>
<td>Returns the error category of the error code stored inside the exception.</td>
</tr>
<tr>
<td>const char* what() const</td>
<td>Returns an implementation-defined non-null constant C-style string that describes the error that triggered the exception.</td>
</tr>
<tr>
<td>bool has_context() const noexcept</td>
<td>Returns true if this SYCL exception has an associated SYCL context and false if it does not.</td>
</tr>
<tr>
<td>context get_context() const</td>
<td>Returns the SYCL context that is associated with this SYCL exception if one is available. Must throw an exception with the errc::invalid error code if this SYCL exception does not have a SYCL context.</td>
</tr>
</tbody>
</table>

Table 141. Values of the SYCL errc enum

<table>
<thead>
<tr>
<th>Standard SYCL Error Codes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>success</td>
<td>The implementation never throws an exception with this error code, but it is defined to ensure that no other error code has the value zero. An application can construct an std::error_code with this code to indicate &quot;not an error&quot;.</td>
</tr>
<tr>
<td>runtime</td>
<td>Generic runtime error.</td>
</tr>
<tr>
<td>Standard SYCL Error Codes</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>kernel</td>
<td>Error that occurred before or while enqueuing the SYCL kernel.</td>
</tr>
<tr>
<td>nd_range</td>
<td>Error regarding the SYCL <code>nd_range</code> specified for the SYCL kernel.</td>
</tr>
<tr>
<td>accessor</td>
<td>Error regarding the SYCL <code>accessor</code> objects defined.</td>
</tr>
<tr>
<td>event</td>
<td>Error regarding associated SYCL <code>event</code> objects.</td>
</tr>
<tr>
<td>kernel_argument</td>
<td>The application has passed an invalid argument to a SYCL <code>kernel</code> function. This includes captured variables if the SYCL <code>kernel</code> function is a lambda function.</td>
</tr>
<tr>
<td>build</td>
<td>Error from an online compile or link operation when compiling, linking, or building a kernel bundle for a device.</td>
</tr>
<tr>
<td>invalid</td>
<td>A catchall error which is used when the application passes an invalid value as a parameter to a SYCL API function or calls a SYCL API function in some invalid way.</td>
</tr>
<tr>
<td>memory_allocation</td>
<td>Error on memory allocation on the SYCL device for a SYCL kernel.</td>
</tr>
<tr>
<td>platform</td>
<td>The SYCL platform will trigger this exception on error.</td>
</tr>
<tr>
<td>profiling</td>
<td>The SYCL runtime will trigger this error if there is an error when profiling info is enabled.</td>
</tr>
<tr>
<td>feature_not_supported</td>
<td>Exception thrown when host code uses an optional feature that is not supported by a device.</td>
</tr>
<tr>
<td>kernel_not_supported</td>
<td>Exception thrown when a kernel uses an optional feature that is not supported on the device to which it is enqueued. This exception is also thrown if a command group is bound to a kernel bundle, and the bundle does not contain the kernel invoked by the command group.</td>
</tr>
</tbody>
</table>
Standard SYCL Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backend_mismatch</td>
<td>The application has called a backend interoperability function with mismatched backend information. For example, requesting information specific to backend A from a SYCL object that comes from backend B causes this error.</td>
</tr>
</tbody>
</table>

Table 142. SYCL error code helper functions

<table>
<thead>
<tr>
<th>SYCL Error Code Helpers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const std::error_category&amp; sycl_category() noexcept;</td>
<td>Obtains a reference to the static error category object for SYCL errors. This object overrides the virtual function error_category::name() to return a pointer to the string &quot;sycl&quot;. When the implementation throws an sycl::exception object ex with this category, the error code value contained by the exception (ex.code().value()) is one of the enumerated values in sycl::errc.</td>
</tr>
<tr>
<td>std::error_code make_error_code(errc e) noexcept;</td>
<td>Constructs an error code using e and sycl_category().</td>
</tr>
</tbody>
</table>

4.14. Data types

SYCL as a C++ programming model supports the C++ core language data types, and it also provides the ability for all SYCL applications to be executed on SYCL compatible devices. The scalar and vector data types that are supported by the SYCL system are defined below. More details about the SYCL device compiler support for fundamental and backend interoperability types are found in Section 5.5.

4.14.1. Scalar data types

The fundamental C++ data types which are supported in SYCL are described in Table 184. Note these types are fundamental and therefore do not exist within the sycl namespace.

Additional scalar data types which are supported by SYCL within the sycl namespace are described in Table 143.

Table 143. Additional scalar data types supported by SYCL

<table>
<thead>
<tr>
<th>Scalar data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>An unsigned 8-bit integer. This is deprecated in SYCL 2020 since C++17 std::byte can be used instead.</td>
</tr>
</tbody>
</table>
### Scalar data type

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. This type is only supported on devices that have <code>aspect::fp16</code>. <code>std::numeric_limits</code> must be specialized for the half data type.</td>
</tr>
</tbody>
</table>

### 4.14.2. Vector types

SYCL provides a cross-platform class template that works efficiently on SYCL devices as well as in host C++ code. This type allows sharing of vectors between the host and its SYCL devices. The vector supports member functions that allow construction of a new vector from a swizzled set of component elements.

A vector type that compiles down to a SYCL backend built-in vector types on SYCL devices, where possible, and provides compatible support on the host or when it is not possible. The vec class is templated on its number of elements and its element type. The number of elements parameter, `NumElements`, can be one of: 1, 2, 3, 4, 8 or 16. Any other value shall produce a compilation failure. The element type parameter, `DataT`, must be one of the basic scalar types supported in device code.

The SYCL `vec` class template provides interoperability with the underlying vector type defined by `vector_t` which is available only when compiled for the device. The SYCL `vec` class can be constructed from an instance of `vector_t` and can implicitly convert to an instance of `vector_t` in order to support interoperability with native SYCL backend functions from a SYCL kernel function.

An instance of the SYCL `vec` class template can also be implicitly converted to an instance of the data type when the number of elements is 1 in order to allow single element vectors and scalars to be convertible with each other.

#### 4.14.2.1. Vec interface

The constructors, member functions and non-member functions of the SYCL `vec` class template are listed in Table 144, Table 145 and Table 146 respectively.

```cpp
namespace sycl {
  enum class rounding_mode : /* unspecified */ { automatic, rte, rtz, rtp, rtn }; 

  struct elem {
    static constexpr int x = 0;
    static constexpr int y = 1;
    static constexpr int z = 2;
    static constexpr int w = 3;
    static constexpr int r = 0;
    static constexpr int g = 1;
    static constexpr int b = 2;
    static constexpr int a = 3;
    static constexpr int s0 = 0;
    static constexpr int s1 = 1;
    static constexpr int s2 = 2;
    static constexpr int s3 = 3;
    static constexpr int s4 = 4;
    static constexpr int s5 = 5;
}
```
```cpp
static constexpr int s6 = 6;
static constexpr int s7 = 7;
static constexpr int s8 = 8;
static constexpr int s9 = 9;
static constexpr int sA = 10;
static constexpr int sB = 11;
static constexpr int sC = 12;
static constexpr int sD = 13;
static constexpr int sE = 14;
static constexpr int sF = 15;

};

template <typename DataT, int NumElements> class vec {
public:
    using element_type = DataT;

#ifdef __SYCL_DEVICE_ONLY__
    using vector_t = __unspecified__;
#endif

vec();
explicit constexpr vec(const DataT & arg);

template <typename ... ArgTN> constexpr vec(const ArgTN & ... args);

constexpr vec(const vec<DataT, NumElements> & rhs);

#ifdef __SYCL_DEVICE_ONLY__
vec(vector_t nativeVector);
#endif

operator vector_t() const;

// Available only when: NumElements == 1
operator DataT() const;

static constexpr size_t byte_size() noexcept;

static constexpr size_t size() noexcept;

// Deprecated
size_t get_size() const;

// Deprecated
size_t get_count() const;

template <typename ConvertT,
    rounding_mode RoundingMode = rounding_mode::automatic>
vec<ConvertT, NumElements> convert() const;

template <typename AsT> AsT as() const;

template <int ... swizzleIndexes> __swizzled_vec__ swizzle() const;

// Available only when NumElements <= 4.
```
// XYZW_ACCESS is: x, y, z, w, subject to NumElements.
__swizzled_vec__ XYZW_ACCESS() const;

// Available only NumElements == 4.
// RGBA_ACCESS is: r, g, b, a.
__swizzled_vec__ RGBA_ACCESS() const;

// INDEX_ACCESS is: s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD, sE, sF, subject to NumElements.
__swizzled_vec__ INDEX_ACCESS() const;

#define SYCL_SIMPLE_SWIZZLES
// Available only when NumElements <= 4.
// XYZW_SWIZZLE is all permutations with repetition of: x, y, z, w, subject to NumElements.
__swizzled_vec__ XYZW_SWIZZLE() const;

// Available only when NumElements == 4.
// RGBA_SWIZZLE is all permutations with repetition of: r, g, b, a.
__swizzled_vec__ RGBA_SWIZZLE() const;
#endif // #ifdef SYCL_SIMPLE_SWIZZLES

// Available only when: NumElements > 1.
__swizzled_vec__ lo() const;
__swizzled_vec__ hi() const;
__swizzled_vec__ odd() const;
__swizzled_vec__ even() const;

// load and store member functions
template <access::address_space AddressSpace, access::decorated IsDecorated>
void load(size_t offset,
  multi_ptr<const DataT, AddressSpace, IsDecorated> ptr);
template <access::address_space AddressSpace, access::decorated IsDecorated>
void store(size_t offset,
  multi_ptr<DataT, AddressSpace, IsDecorated> ptr) const;

// subscript operator
DataT& operator[](int index);
const DataT& operator[](int index) const;

// OP is: +, -, *, /, %
/* If OP is %, available only when: DataT != float && DataT != double && DataT != half. */
friend vec operatorOP(const vec& lhs, const vec& rhs) { /* ... */
}
friend vec operatorOP(const vec& lhs, const DataT& rhs) { /* ... */
}

// OP is: +=, -=, *=, /=, %=
/* If OP is %, available only when: DataT != float && DataT != double && DataT != half. */
friend vec& operatorOP(vec& lhs, const vec& rhs) { /* ... */
}
friend vec& operatorOP(vec& lhs, const DataT& rhs) { /* ... */
}
friend vec& operatorOP(vec& rhs) { /* ... */ }

friend vec operatorOP(vec lhs, int) { /* ... */ }

friend vec& operatorOP(vec& lhs, const vec& rhs) { /* ... */ }

friend vec& operatorOP(const vec& lhs, const vec& rhs) { /* ... */ }

friend vec& operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const vec& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }

friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) { /* ... */ }
friend vec<RET, NumElements> operatorOP(const vec& lhs, const DataT& rhs) {
    /* ... */
vec<RET, NumElements> operator=(const vec<DataT, NumElements>& rhs);
vec<RET, NumElements> operator=(const DataT& rhs);
    /* Available only when: DataT != float && DataT != double && DataT != half. */
friend vec operator~(const vec& v) { /* ... */
friend vec<RET, NumElements> operator!(const vec& v) { /* ... */

// OP is: +, -, *, /, %
friend vec operatorOP(const DataT& lhs, const vec& rhs) { /* ... */

// OP is: &,
friend vec<DataT, NumElements> operatorOP(const DataT& lhs, const vec& rhs) { /* ... */

// OP is: &&, ||
friend vec<DataT, NumElements> operatorOP(const DataT& lhs, const vec& rhs) { /* ... */

// OP is: <<, >>
friend vec<DataT, NumElements> operatorOP(const DataT& lhs, const vec& rhs) { /* ... */

// OP is: ==, !=, <, >, <=, >=
friend vec<DataT, NumElements> operatorOP(const DataT& lhs, const vec& rhs) { /* ... */

};

// Deduction guides
// Available only when: (std::is_same_v<T, U> && ...)
template <class T, class... U> vec<T, U...> -> vec<T, sizeof...(U) + 1>;
} // namespace sycl

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec()</td>
<td>Default construct a vector with element type DataT and with NumElements dimensions by default construction of each of its elements.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Explicit constexpr vec(const DataT&amp; arg)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a vector of element type <code>DataT</code> and <code>NumElements</code> dimensions by setting each value to <code>arg</code> by assignment.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Template&lt;typename... ArgTN&gt; constexpr vec(const ArgTN&amp;... args)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a SYCL vec instance from any combination of scalar and SYCL vec parameters of the same element type, providing the total number of elements for all parameters sum to <code>NumElements</code> of this vec specialization.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>constexpr vec(const vec&lt;DataT, NumElements&gt;&amp; rhs)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a vector of element type <code>DataT</code> and number of elements <code>NumElements</code> by copy from another similar vector.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vec(vector_t nativeVector)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available only when: compiled for the device. Constructs a SYCL vec instance from an instance of the underlying backend-native vector type defined by <code>vector_t</code>.</td>
<td></td>
</tr>
</tbody>
</table>

### Table 145. Member functions for the SYCL vec class template

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator vector_t() const</code></td>
<td>Available only when: compiled for the device. Converts this SYCL vec instance to the underlying backend-native vector type defined by <code>vector_t</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>operator DataT() const</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available only when: <code>NumElements</code> == 1. Converts this SYCL vec instance to an instance of <code>DataT</code> with the value of the single element in this SYCL vec instance.</td>
<td></td>
</tr>
</tbody>
</table>

The SYCL vec instance shall be implicitly convertible to the same data types, to which `DataT` is implicitly convertible. Note that conversion operator shall not be template to allow standard conversion sequence for implicit conversion.

<table>
<thead>
<tr>
<th>static constexpr size_t size() noexcept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the number of elements of this SYCL vec.</td>
<td></td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>size_t get_count() const</code></td>
<td>Returns the same value as <code>size()</code>. Deprecated.</td>
</tr>
<tr>
<td><code>static constexpr size_t byte_size() noexcept</code></td>
<td>Returns the size of this SYCL vec in bytes. 3-element vector size matches 4-element vector size to provide interoperability with OpenCL vector types. The same rule applies to vector alignment as described in Section 4.14.2.6.</td>
</tr>
<tr>
<td><code>size_t get_size() const</code></td>
<td>Returns the same value as <code>byte_size()</code>. Deprecated.</td>
</tr>
<tr>
<td><code>template &lt;typename ConvertT, rounding_mode RoundingMode = rounding_mode::automatic&gt; vec&lt;ConvertT, NumElements&gt; convert() const</code></td>
<td>Converts this SYCL vec to a SYCL vec of a different element type specified by <code>ConvertT</code> using the rounding mode specified by <code>RoundingMode</code>. The new SYCL vec type must have the same number of elements as this SYCL vec. The different rounding modes are described in Table 147.</td>
</tr>
<tr>
<td><code>template &lt;typename asT&gt; asT as() const</code></td>
<td>Bitwise reinterprets this SYCL vec as a SYCL vec of a different element type and number of elements specified by <code>asT</code>. The new SYCL vec type must have the same storage size in bytes as this SYCL vec.</td>
</tr>
<tr>
<td><code>template &lt;int... swizzleIndexes&gt; __swizzled_vec__ swizzle() const</code></td>
<td>Return an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4.</td>
</tr>
</tbody>
</table>
| `__swizzled_vec__ XYZW_ACCESS() const` | Available only when: `NumElements <= 4`.  
Returns an instance of the implementation-defined intermediate class template `__swizzled_vec__` representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4.  
Where `XYZW_ACCESS` is:  
x for `NumElements == 1`,  
y for `NumElements == 2`,  
z for `NumElements == 3` and  
x, y, z, w for `NumElements == 4`. |
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>swizzled_vec</strong> RGBA_ACCESS() const</td>
<td>Available only when: NumElements == 4. Returns an instance of the implementation-defined intermediate class template <strong>swizzled_vec</strong> representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. Where RGBA_ACCESS is: r, g, b, a.</td>
</tr>
<tr>
<td><strong>swizzled_vec</strong> INDEX_ACCESS() const</td>
<td>Returns an instance of the implementation-defined intermediate class template <strong>swizzled_vec</strong> representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. Where INDEX_ACCESS is: s0 for NumElements == 1, s0, s1 for NumElements == 2, s0, s1, s2 for NumElements == 3, s0, s1, s2, s3 for NumElements == 4, s0, s1, s2, s3, s4, s5, s6, s7, s8 for NumElements == 8 and s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, sA, sB, sC, sD, sE, sF for NumElements == 16.</td>
</tr>
<tr>
<td><strong>swizzled_vec</strong> XYZW_SWIZZLE() const</td>
<td>Available only when: NumElements &lt;= 4, and when the macro SYCL_SIMPLE_SWIZZLES is defined before including &lt;sycl/sycl.hpp&gt;. Returns an instance of the implementation-defined intermediate class template <strong>swizzled_vec</strong> representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. Where XYZW_SWIZZLE is all permutations with repetition, of any subset with length greater than 1, of x, y for NumElements == 2, x, y, z for NumElements == 3 and x, y, z, w for NumElements == 4. For example a four element vec provides permutations including xzwy, xyyx and xz.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>swizzled_vec</strong> RGBA_SWIZZLE() const</td>
<td>Available only when: <code>NumElements == 4</code>, and when the macro <code>SYCL_SIMPLE_SWIZZLES</code> is defined before including <code>&lt;sycl/sycl.hpp&gt;</code>. Returns an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. Where RGBA_SWIZZLE is all permutations with repetition, of any subset with length greater than 1, of r, g, b, a. For example a four element vec provides permutations including rgba, rggg and rb.</td>
</tr>
<tr>
<td><strong>swizzled_vec</strong> lo() const</td>
<td>Available only when: <code>NumElements &gt; 1</code>. Return an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence made up of the lower half of this SYCL vec which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. When <code>NumElements == 3</code>, this SYCL vec is treated as though <code>NumElements == 4</code> with the fourth element undefined.</td>
</tr>
<tr>
<td><strong>swizzled_vec</strong> hi() const</td>
<td>Available only when: <code>NumElements &gt; 1</code>. Return an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence made up of the upper half of this SYCL vec which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. When <code>NumElements == 3</code>, this SYCL vec is treated as though <code>NumElements == 4</code> with the fourth element undefined.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>__swizzled_vec__ odd()</code> const</td>
<td>Available only when: <code>NumElements &gt; 1</code>. Return an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence made up of the odd indexes of this SYCL vec which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. When <code>NumElements == 3</code>, this SYCL vec is treated as though <code>NumElements == 4</code> with the fourth element undefined.</td>
</tr>
<tr>
<td><code>__swizzled_vec__ even()</code> const</td>
<td>Available only when: <code>NumElements &gt; 1</code>. Return an instance of the implementation-defined intermediate class template <code>__swizzled_vec__</code> representing an index sequence made up of the even indexes of this SYCL vec which can be used to apply the swizzle in a valid expression as described in Section 4.14.2.4. When <code>NumElements == 3</code>, this SYCL vec is treated as though <code>NumElements == 4</code> with the fourth element undefined.</td>
</tr>
</tbody>
</table>

### Template Functions

<table>
<thead>
<tr>
<th>Template Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;access::address_space AddressSpace, access::decorated IsDecorated&gt; void load(size_t offset, multi_ptr&lt;const DataT, AddressSpace, IsDecorated&gt; ptr)</code></td>
<td>Loads the values at the address of <code>ptr</code> offset in elements of type <code>DataT</code> by <code>NumElements * offset</code>, into the components of this SYCL vec.</td>
</tr>
<tr>
<td><code>template &lt;access::address_space AddressSpace, access::decorated IsDecorated&gt; void store(size_t offset, multi_ptr&lt;DataT, AddressSpace, IsDecorated&gt; ptr) const</code></td>
<td>Stores the components of this SYCL vec into the values at the address of <code>ptr</code> offset in elements of type <code>DataT</code> by <code>NumElements * offset</code>.</td>
</tr>
</tbody>
</table>

### Other Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataT&amp; operator[](int index)</code></td>
<td>Returns a reference to the element stored within this SYCL vec at the index specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>const DataT&amp; operator[](int index) const</code></td>
<td>Returns a const reference to the element stored within this SYCL vec at the index specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>vec&amp; operator=(const vec&amp; rhs)</code></td>
<td>Assign each element of the rhs SYCL vec to each element of this SYCL vec and return a reference to this SYCL vec.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec&amp; operator=(const DataT&amp; rhs)</code></td>
<td>Assign each element of the <code>rhs</code> scalar to each element of this SYCL <code>vec</code> and return a reference to this SYCL <code>vec</code>.</td>
</tr>
</tbody>
</table>

**Table 146. Hidden friend functions of the `vec` class template**

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>. Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const DataT&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar. Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const vec8&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%=</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> arithmetic operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code> and return <code>lhs vec</code>. Where <code>OP</code> is: <code>+=</code>, <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>.</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const DataT&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%=</code> available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> arithmetic operation between each element of <code>lhs vec</code> and <code>rhs</code> scalar and return <code>lhs vec</code>. Where <code>OP</code> is: <code>+=</code>, <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>.</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 v)</code></td>
<td>Available only when: <code>DataT != bool</code>. Perform an in-place element-wise <code>OP</code> prefix arithmetic operation on each element of <code>lhs vec</code>, assigning the result of each element to the corresponding element of <code>lhs vec</code> and return <code>lhs vec</code>. Where <code>OP</code> is: <code>++</code>, <code>--</code>.</td>
</tr>
<tr>
<td><code>vec operatorOP(vec8 v, int)</code></td>
<td>Available only when: <code>DataT != bool</code>. Perform an in-place element-wise <code>OP</code> postfix arithmetic operation on each element of <code>lhs vec</code>, assigning the result of each element to the corresponding element of <code>lhs vec</code> and returns a copy of <code>lhs vec</code> before the operation is performed. Where <code>OP</code> is: <code>++</code>, <code>--</code>.</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec8 v)</code></td>
<td>Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as this SYCL <code>vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> unary arithmetic operation on each element of this SYCL <code>vec</code>. Where <code>OP</code> is: <code>+</code>, <code>-</code>.</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>. Where <code>OP</code> is: <code>&amp;</code>, `</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const DataT&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar. Where <code>OP</code> is: <code>&amp;</code>, `</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const vec8 rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs</code> SYCL <code>vec</code> and return <code>lhs vec</code>. Where <code>OP</code> is: <code>&amp;=</code>, `</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const DataT8&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> bitwise operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar and return a <code>lhs vec</code>. Where <code>OP</code> is: <code>&amp;=</code>, `</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const vec&amp; rhs)</code></td>
<td>Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>. The <code>DataT</code> template parameter of the constructed SYCL <code>vec</code>, <code>RET</code>, varies depending on the <code>DataT</code> template parameter of this SYCL <code>vec</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int8_t</code> or <code>uint8_t</code> <code>RET</code> must be <code>int8_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int16_t</code>, <code>uint16_t</code> or <code>half</code> <code>RET</code> must be <code>int16_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int32_t</code>, <code>uint32_t</code> or <code>float</code> <code>RET</code> must be <code>int32_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int64_t</code>, <code>uint64_t</code> or <code>double</code> <code>RET</code> must be <code>uint64_t</code>. Where <code>OP</code> is: <code>&amp;&amp;</code>, `</td>
</tr>
<tr>
<td><code>vec&lt;RET, NumElements&gt; operatorOP(const vec&amp; lhs, const DataT&amp; rhs)</code></td>
<td>Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as this SYCL <code>vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> logical operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar. The <code>DataT</code> template parameter of the constructed SYCL <code>vec</code>, <code>RET</code>, varies depending on the <code>DataT</code> template parameter of this SYCL <code>vec</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int8_t</code> or <code>uint8_t</code> <code>RET</code> must be <code>int8_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int16_t</code>, <code>uint16_t</code> or <code>half</code> <code>RET</code> must be <code>int16_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int32_t</code>, <code>uint32_t</code> or <code>float</code> <code>RET</code> must be <code>int32_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int64_t</code>, <code>uint64_t</code> or <code>double</code> <code>RET</code> must be <code>uint64_t</code>. Where <code>OP</code> is: <code>&amp;&amp;</code>, `</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const vec&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and each element of the <code>rhs</code> SYCL <code>vec</code>. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where <code>OP</code> is: <code>&lt;&lt;, &gt;&gt;</code>.</td>
</tr>
<tr>
<td><code>vec operatorOP(const vec&amp; lhs, const DataT&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as <code>lhs vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where <code>OP</code> is: <code>&lt;&lt;, &gt;&gt;</code>.</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const vec8 rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and the <code>rhs</code> SYCL <code>vec</code> and returns <code>lhs vec</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where <code>OP</code> is: <code>&lt;&lt;=, &gt;&gt;=</code>.</td>
</tr>
<tr>
<td><code>vec8 operatorOP(vec8 lhs, const Data8 rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> bitshift operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar and returns a reference to this SYCL <code>vec</code>. If <code>OP</code> is <code>&gt;&gt;=</code>, <code>DataT</code> is a signed type and <code>lhs vec</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where <code>OP</code> is: <code>&lt;&lt;=, &gt;&gt;=</code>.</td>
</tr>
</tbody>
</table>
### Hidden friend function

| vec<RET, NumElements> operatorOP(const vec<lhs, const vec<rhs)> |

**Description**

Construct a new instance of the SYCL vec class template with the element type `RET` with each element of the new SYCL vec instance the result of an element-wise `OP` relational operation between each element of `lhs` vec and each element of the `rhs` SYCL vec. Each element of the SYCL vec that is returned must be `-1` if the operation results in `true` and `0` if the operation results in `false`. The `==`, `!=`, `<`, `<=` and `>=` operations result in `false` if either the `lhs` element or the `rhs` element is a NaN. The `!=` operation results in `true` if either the `lhs` element or the `rhs` element is a NaN.

The `DataT` template parameter of the constructed SYCL vec, `RET`, varies depending on the `DataT` template parameter of this SYCL vec. For a SYCL vec with `DataT` of type `int8_t` or `uint8_t` `RET` must be `int8_t`. For a SYCL vec with `DataT` of type `int16_t`, `uint16_t` or `half` `RET` must be `int16_t`. For a SYCL vec with `DataT` of type `int32_t`, `uint32_t` or `float` `RET` must be `int32_t`. For a SYCL vec with `DataT` of type `int64_t`, `uint64_t` or `double` `RET` must be `uint64_t`.

Where `OP` is: `==`, `!=`, `<`, `<=`, `>=`. 

---

364 | Chapter 4. SYCL programming interface
<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vec&lt;RET, NumElements&gt; operatorOP(const vec&lt;T&gt; lhs, const DataT&lt;T&gt; rhs)</strong></td>
<td>Construct a new instance of the SYCL vec class template with the DataT parameter of <code>RET</code> with each element of the new SYCL vec instance the result of an element-wise <code>OP</code> relational operation between each element of <code>lhs vec</code> and the <code>rhs</code> scalar. Each element of the SYCL vec that is returned must be <code>-1</code> if the operation results in <code>true</code> and <code>0</code> if the operation results in <code>false</code>. The <code>==</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> element or the <code>rhs</code> is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> element or the <code>rhs</code> is a NaN. The <code>DataT</code> template parameter of the constructed SYCL vec, <code>RET</code>, varies depending on the <code>DataT</code> template parameter of this SYCL vec. For a SYCL vec with <code>DataT</code> of type <code>int8_t</code> or <code>uint8_t</code> <code>RET</code> must be <code>int8_t</code>. For a SYCL vec with <code>DataT</code> of type <code>int16_t</code>, <code>uint16_t</code> or <code>half</code> <code>RET</code> must be <code>int16_t</code>. For a SYCL vec with <code>DataT</code> of type <code>int32_t</code>, <code>uint32_t</code> or <code>float</code> <code>RET</code> must be <code>int32_t</code>. For a SYCL vec with <code>DataT</code> of type <code>int64_t</code>, <code>uint64_t</code> or <code>double</code> <code>RET</code> must be <code>uint64_t</code>. Where <code>OP</code> is: <code>==</code>, <code>!=</code>, <code>&lt;</code>, <code>&gt;</code>, <code>&lt;=</code>, <code>&gt;=</code>.</td>
</tr>
<tr>
<td><strong>vec operatorOP(const DataT&lt;T&gt; lhs, const vec&lt;T&gt; rhs)</strong></td>
<td>If <code>OP</code> is <code>%</code>, available only when: <code>DataT</code> != <code>float</code> &amp;&amp; <code>DataT</code> != <code>double</code> &amp; <code>DataT</code> != <code>half</code>. Construct a new instance of the SYCL vec class template with the same template parameters as the <code>rhs</code> SYCL vec with each element of the new SYCL vec instance the result of an element-wise <code>OP</code> arithmetic operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL vec. Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `vec operatorOP(const DataT& lhs, const vec& rhs)` | Available only when: `DataT != float && DataT != double && DataT != half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as the `rhs` SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`. Where `OP` is: `&`, `|`, `^`.

<p>| <code>vec&lt;RET, NumElements&gt; operatorOP(const DataT&amp; lhs, const vec&amp; rhs)</code> | Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>vec</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>vec</code> with each element of the new SYCL <code>vec</code> instance the result of an element-wise <code>OP</code> logical operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>vec</code>. The <code>DataT</code> template parameter of the constructed SYCL <code>vec</code>, <code>RET</code>, varies depending on the <code>DataT</code> template parameter of this SYCL <code>vec</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int8_t</code> or <code>uint8_t</code> <code>RET</code> must be <code>int8_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int16_t</code>, <code>uint16_t</code> or <code>half</code> <code>RET</code> must be <code>int16_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int32_t</code>, <code>uint32_t</code> or <code>float</code> <code>RET</code> must be <code>int32_t</code>. For a SYCL <code>vec</code> with <code>DataT</code> of type <code>int64_t</code>, <code>uint64_t</code> or <code>double</code> <code>RET</code> must be <code>int64_t</code>. Where <code>OP</code> is: <code>&amp;&amp;</code>, <code>||</code>. |</p>
<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vec</strong> operatorOP(const DataT&amp; lhs, const vec&amp; rhs)</td>
<td>Construct a new instance of the SYCL vec class template with the same template parameters as the rhs SYCL vec with each element of the new SYCL vec instance the result of an element-wise OP bit-shift operation between the lhs scalar and each element of the rhs SYCL vec. If OP is <code>&gt;&gt;</code>, DataT is a signed type and this SYCL vec has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where OP is: <code>&lt;&lt;, &gt;&gt;</code>.</td>
</tr>
</tbody>
</table>

| vec<RET, NumElements> operatorOP(const DataT& lhs, const vec& rhs) | Construct a new instance of the SYCL vec class template with the element type RET with each element of the new SYCL vec instance the result of an element-wise OP relational operation between the lhs scalar and each element of the rhs SYCL vec. Each element of the SYCL vec that is returned must be -1 if the operation results in `true` and 0 if the operation results in `false`. The `==`, `!=`, `<`, `<=`, `>`, `>=` operations result in `false` if either the lhs or the rhs element is a NaN. The `!=` operation results in `true` if either the lhs or the rhs element is a NaN. The DataT template parameter of the constructed SYCL vec, RET, varies depending on the DataT template parameter of this SYCL vec. For a SYCL vec with DataT of type int8_t or uint8_t RET must be int8_t. For a SYCL vec with DataT of type int16_t, uint16_t or half RET must be int16_t. For a SYCL vec with DataT of type int32_t, uint32_t or float RET must be int32_t. For a SYCL vec with DataT of type int64_t, uint64_t or double RET must be int64_t. Where OP is: `==, !=, <, <=, >, >=`. |
### Hidden friend function

<table>
<thead>
<tr>
<th>vec8 operator~(const vec8 v)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half.</td>
<td></td>
</tr>
<tr>
<td>Construct a new instance of the SYCL vec class template with the same template parameters as v vec with each element of the new SYCL vec instance the result of an element-wise OP bitwise operation on each element of v vec.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vec&lt;RET, NumElements&gt; operator!(const vec8 v)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct a new instance of the SYCL vec class template with the same template parameters as v vec with each element of the new SYCL vec instance the result of an element-wise OP logical operation on each element of v vec. Each element of the SYCL vec that is returned must be -1 if the operation results in true and 0 if the operation results in false or this SYCL vec is a NaN.</td>
<td></td>
</tr>
</tbody>
</table>

The DataT template parameter of the constructed SYCL vec, RET, varies depending on the DataT template parameter of this SYCL vec. For a SYCL vec with DataT of type int8_t or uint8_t RET must be int8_t. For a SYCL vec with DataT of type int16_t, uint16_t or half RET must be int16_t. For a SYCL vec with DataT of type int32_t, uint32_t or float RET must be int32_t. For a SYCL vec with DataT of type int64_t, uint64_t or double RET must be int64_t.

#### 4.14.2.2. Aliases

The SYCL programming API provides all permutations of the type alias:

```cpp
using <type><elems> = vec<<storage-type>, <elems>>
```

where <elems> is 2, 3, 4, 8 and 16, and pairings of <type> and <storage-type> for integral types are char and int8_t, uchar and uint8_t, short and int16_t, ushort and uint16_t, int and int32_t, uint and uint32_t, long and int64_t, ulong and uint64_t, and for floating point types are both half, float and double.

For example uint4 is the alias to vec<int32_t, 4> and float16 is the alias to vec<float, 16>.

#### 4.14.2.3. Swizzles

Swizzle operations can be performed in two ways. Firstly by calling the swizzle member function template, which takes a variadic number of integer template arguments between 0 and NumElements-1, speci-
fying swizzle indexes. Secondly by calling one of the simple swizzle member functions defined in Table 145 as XYZW_SWIZZLE and RGBA_SWIZZLE. Note that the simple swizzle functions are only available for up to 4 element vectors and are only available when the macro SYCL_SIMPLE_SWIZZLES is defined before including <sycl/sycl.hpp>.

In both cases the return type is always an instance of _swizzled_vec_, an implementation-defined temporary class representing a swizzle of the original SYCL vec instance. Both kinds of swizzle member functions must not perform the swizzle operation themselves, instead the swizzle operation must be performed by the returned instance of _swizzled_vec_ when used within an expression, meaning if the returned _swizzled_vec_ is never used in an expression no swizzle operation is performed.

Both the swizzle member function template and the simple swizzle member functions allow swizzle indexes to be repeated.

A series of static constexpr values are provided within the elem struct to allow specifying named swizzle indexes when calling the swizzle member function template.

4.14.2.4. Swizzled vec class

The _swizzled_vec_ class must define an unspecified temporary which provides the entire interface of the SYCL vec class template, including swizzled member functions, with the additions and alterations described below:

- The _swizzled_vec_ class template must be readable as an r-value reference on the RHS of an expression. In this case the swizzle operation is performed on the RHS of the expression and then the result is applied to the LHS of the expression.
- The _swizzled_vec_ class template must be assignable as an l-value reference on the LHS of an expression. In this case the RHS of the expression is applied to the original SYCL vec which the _swizzled_vec_ represents via the swizzle operation. Note that a _swizzled_vec_ that is used in an l-value expression may not contain any repeated element indexes.

For example: `f4.xxxx() = fx.wzyx()` would not be valid.

- The _swizzled_vec_ class template must be convertible to an instance of SYCL vec with the type DataT and number of elements specified by the swizzle member function, if NumElements > 1, and must be convertible to an instance of type DataT, if NumElements == 1.
- The _swizzled_vec_ class template must be non-copyable, non-moveable, non-user constructible and may not be bound to a l-value or escape the expression it was constructed in. For example auto x = f4.x() would not be valid.
- The _swizzled_vec_ class template should return _swizzled_vec_& for each operator inherited from the vec class template interface which would return vec<DataT, NumElements>&.

4.14.2.5. Rounding modes

The various rounding modes that can be used in the as member function template are described in Table 147.

<table>
<thead>
<tr>
<th>Rounding mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>automatic</td>
<td>Default rounding mode for the SYCL vec class element type. rtz (round toward zero) for integer types and rte (round to nearest even) for floating-point types.</td>
</tr>
</tbody>
</table>
### 4.14.2.6. Memory layout and alignment

The elements of an instance of the SYCL vec class template are stored in memory sequentially and contiguously and are aligned to the size of the element type in bytes multiplied by the number of elements:

\[
\text{sizeof}(\text{DataT}) \cdot \text{NumElements}
\]

The exception to this is when the number of element is three in which case the SYCL vec is aligned to the size of the element type in bytes multiplied by four:

\[
\text{sizeof}(\text{DataT}) \cdot 4
\]

This is true for both host and device code in order to allow for instances of the vec class template to be passed to SYCL kernel functions.

### 4.14.2.7. Performance note

The usage of the subscript operator[] may not be efficient on some devices.

### 4.14.3. Math array types

SYCL provides an marray<typename DataT, std::size_t NumElements> class template to represent a contiguous fixed-size container. This type allows sharing of containers between the host and its SYCL devices.

The marray class is templated on its element type and number of elements. The number of elements parameter, NumElements, is a positive value of the std::size_t type. The element type parameter, DataT, must be a numeric type as it is defined by C++ standard.

An instance of the marray class template can also be implicitly converted to an instance of the data type when the number of elements is 1 in order to allow single element arrays and scalars to be convertible with each other.

Logical and comparison operators for marray class template return marray<bool, NumElements>.

#### 4.14.3.1. Math array interface

The constructors, member functions and non-member functions of the SYCL marray class template are listed in Table 148, Table 149 and Table 150 respectively.

```cpp
namespace sycl {
    template <typename DataT, std::size_t NumElements>
    class marray {
    public:
```
using value_type = DataT;
using reference = DataT&;
using const_reference = const DataT&;
using iterator = DataT*;
using const_iterator = const DataT*;

marray();
explicit constexpr marray(const DataT& arg);

template<typename... ArgTN> constexpr marray(const ArgTN&... args);

constexpr marray(const marray<DataT, NumElements>& rhs);
constexpr marray(marray<DataT, NumElements>&& rhs);

// Available only when: NumElements == 1
operator DataT() const;

static constexpr std::size_t size() noexcept;

// subscript operator
reference operator[](std::size_t index);
const_reference operator[](std::size_t index) const;

marray& operator=(const marray<DataT, NumElements>& rhs);

marray& operator=(const DataT& rhs);

// iterator functions
iterator begin();
const_iterator begin() const;

iterator end();
const_iterator end() const;

// OP is: +, -, *, /, %
/* If OP is %, available only when: DataT != float && DataT != double && DataT != half. */
friend marray operatorOP(const marray& lhs, const marray& rhs) { /* ... */
}
friend marray operatorOP(const marray& lhs, const DataT& rhs) { /* ... */

/* If OP is %=, available only when: DataT != float && DataT != double && DataT != half. */
friend marray& operatorOP(marray& lhs, const marray& rhs) { /* ... */
}
friend marray& operatorOP(marray& lhs, const DataT& rhs) { /* ... */

// OP is prefix ++, --
friend marray& operatorOP(marray& v) { /* ... */
}

// OP is postfix ++, --
friend marray operatorOP(marray& v, int) { /* ... */

friend marray operatorOP(marray & v) { /* ... */}

friend marray operatorOP(const marray & lhs, const marray & rhs) { /* ... */}
friend marray operatorOP(const marray & lhs, const DataT & rhs) { /* ... */}

friend marray& operatorOP(marray & lhs, const marray & rhs) { /* ... */}
friend marray& operatorOP(marray & lhs, const DataT & rhs) { /* ... */}

friend marray< bool, NumElements > operatorOP(const marray & lhs, const marray & rhs) { /* ... */}
friend marray< bool, NumElements > operatorOP(const marray & lhs, const DataT & rhs) { /* ... */}

friend marray& operatorOP(marray & lhs, const marray & rhs) { /* ... */}
friend marray& operatorOP(marray & lhs, const DataT & rhs) { /* ... */}

friend marray< bool, NumElements > operatorOP(const marray & lhs, const marray & rhs) { /* ... */}
friend marray< bool, NumElements > operatorOP(const marray & lhs, const DataT & rhs) { /* ... */}

friend marray& operatorOP(marray & lhs, const marray & rhs) { /* ... */}
friend marray& operatorOP(marray & lhs, const DataT & rhs) { /* ... */}

friend marray< bool, NumElements > operatorOP(const marray & lhs, const marray & rhs) { /* ... */}
friend marray< bool, NumElements > operatorOP(const marray & lhs, const DataT & rhs) { /* ... */}

friend marray operator~(const marray & v) { /* ... */}
// OP is: +, -, *, /, %
/* operator% is only available when: DataT != float && DataT != double &&
* DataT != half. */
friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */ }

// OP is: &, |, ^
/* Available only when: DataT != float && DataT != double && DataT != half. */
friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */ }

// OP is: &&, ||
friend marray< bool, NumElements> operatorOP(const DataT& lhs,
const marray& rhs) { /* ... */ }

// OP is: <<, >>
/* Available only when: DataT != float && DataT != double && DataT != half. */
friend marray operatorOP(const DataT& lhs, const marray& rhs) { /* ... */ }

// OP is: ==, !=, <, >, <=, >=
friend marray< bool, NumElements> operator!(const marray& v) { /* ... */ }
};

} // namespace sycl

Table 148. Constructors of the SYCL marray class template

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>marray()</td>
<td>Default construct an array with element type DataT and with NumElements dimensions by default construction of each of its elements.</td>
</tr>
<tr>
<td>explicit constexpr marray(const DataT&amp; arg)</td>
<td>Construct an array of element type DataT and NumElements dimensions by setting each value to arg by assignment.</td>
</tr>
<tr>
<td>template &lt;typename... ArgTN&gt; constexpr marray(const ArgTN&amp;... args)</td>
<td>Construct a SYCL marray instance from any combination of scalar and SYCL marray parameters of the same element type, providing the total number of elements for all parameters sum to NumElements of this marray specialization.</td>
</tr>
</tbody>
</table>
### Constructor

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>constexpr marray(const marray&lt;DataT, NumElements&gt;&amp; rhs)</code></td>
<td>Construct an array of element type <code>DataT</code> and number of elements <code>NumElements</code> by copy from another similar vector.</td>
</tr>
<tr>
<td><code>constexpr marray(marray&lt;DataT, NumElements&gt;&amp;&amp; rhs)</code></td>
<td>Construct an array of element type <code>DataT</code> and number of elements <code>NumElements</code> by moving from another similar vector.</td>
</tr>
</tbody>
</table>

### Table 149. Member functions for the SYCL marray class template

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>operator DataT() const</code></td>
<td>Available only when: <code>NumElements == 1</code>. Converts this SYCL <code>marray</code> instance to an instance of <code>DataT</code> with the value of the single element in this SYCL <code>marray</code> instance. The SYCL <code>marray</code> instance shall be implicitly convertible to the same data types, to which <code>DataT</code> is implicitly convertible. Note that conversion operator shall not be templated to allow standard conversion sequence for implicit conversion.</td>
</tr>
<tr>
<td><code>static constexpr std::size_t size()</code></td>
<td>Returns the size of this SYCL <code>marray</code> in bytes.</td>
</tr>
<tr>
<td><code>DataT&amp; operator[](std::size_t index)</code></td>
<td>Returns a reference to the element stored within this SYCL <code>marray</code> at the index specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>const DataT&amp; operator[](std::size_t index) const</code></td>
<td>Returns a const reference to the element stored within this SYCL <code>marray</code> at the index specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>marray&amp; operator=(const marray&amp; rhs)</code></td>
<td>Assign each element of the <code>rhs</code> SYCL <code>marray</code> to each element of this SYCL <code>marray</code> and return a reference to this SYCL <code>marray</code>.</td>
</tr>
<tr>
<td><code>marray&amp; operator=(const DataT&amp; rhs)</code></td>
<td>Assign each element of the <code>rhs</code> scalar to each element of this SYCL <code>marray</code> and return a reference to this SYCL <code>marray</code>.</td>
</tr>
<tr>
<td><code>iterator begin()</code></td>
<td>Returns an iterator referring to the first element stored within the <code>v marray</code>.</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>const_iterator begin()</strong> const</td>
</tr>
<tr>
<td>Returns a const iterator referring to the first element stored within the <code>v marray</code>.</td>
</tr>
<tr>
<td><strong>iterator end()</strong></td>
</tr>
<tr>
<td>Returns an iterator referring to the one past the last element stored within the <code>v marray</code>.</td>
</tr>
<tr>
<td><strong>const_iterator end()</strong> const</td>
</tr>
<tr>
<td>Returns a const iterator referring to the one past the last element stored within the <code>v marray</code>.</td>
</tr>
</tbody>
</table>

### Table 150. Hidden friend functions of the marray class template

<table>
<thead>
<tr>
<th>Hidden friend function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>marray operatorOP(const marray&amp; lhs, const marray&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> SYCL <code>marray</code>. Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</td>
</tr>
<tr>
<td><code>marray operatorOP(const marray&amp; lhs, const DataT&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and the <code>rhs</code> scalar. Where <code>OP</code> is: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>%</code>.</td>
</tr>
<tr>
<td><code>marray&amp; operatorOP(marray&amp; lhs, const marray&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%=</code>, available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> SYCL <code>marray</code> and return <code>lhs marray</code>. Where <code>OP</code> is: <code>+=</code>, <code>-=</code>, <code>*=</code>, <code>/=</code>, <code>%=</code>.</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>marray&amp; operatorOP(marray&amp; lhs, const DataT&amp; rhs)</code></td>
<td>If <code>OP</code> is <code>%=</code> available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Perform an in-place element-wise <code>OP</code> arithmetic operation between each element of <code>lhs marray</code> and <code>rhs</code> scalar and return <code>lhs marray</code>. Where <code>OP</code> is: <code>+=, -=, *=, /=, %=</code>.</td>
</tr>
<tr>
<td><code>marray&amp; operatorOP(marray&amp; v)</code></td>
<td>Perform an in-place element-wise <code>OP</code> prefix arithmetic operation on each element of <code>v marray</code>, assigning the result of each element to the corresponding element of <code>v marray</code> and return <code>v marray</code>. Where <code>OP</code> is: <code>++, --</code>.</td>
</tr>
<tr>
<td><code>marray operatorOP(marray&amp; v, int)</code></td>
<td>Perform an in-place element-wise <code>OP</code> postfix arithmetic operation on each element of <code>v marray</code>, assigning the result of each element to the corresponding element of <code>v marray</code> and returns a copy of <code>v marray</code> before the operation is performed. Where <code>OP</code> is: <code>++, --</code>.</td>
</tr>
<tr>
<td><code>marray operatorOP(marray&amp; v)</code></td>
<td>Construct a new instance of the <code>SYCL marray</code> class template with the same template parameters as this <code>SYCL marray</code> with each element of the new <code>SYCL marray</code> instance the result of an element-wise <code>OP</code> unary arithmetic operation on each element of this <code>SYCL marray</code>. Where <code>OP</code> is: <code>+, -</code>.</td>
</tr>
<tr>
<td><code>marray operatorOP(const marray&amp; lhs, const marray&amp; rhs)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the <code>SYCL marray</code> class template with the same template parameters as <code>lhs marray</code> with each element of the new <code>SYCL marray</code> instance the result of an element-wise <code>OP</code> bitwise operation between each element of <code>lhs marray</code> and each element of the <code>rhs</code> <code>SYCL marray</code>. Where <code>OP</code> is: `&amp;,</td>
</tr>
</tbody>
</table>
## Hidden friend function

<table>
<thead>
<tr>
<th>Function Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>marray operatorOP(const marray &amp; lhs, const DataT &amp; rhs)</td>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half. Construct a new instance of the SYCL marray class template with the same template parameters as lhs marray with each element of the new SYCL marray instance the result of an element-wise OP bitwise operation between each element of lhs marray and the rhs scalar. Where OP is: &amp;,</td>
</tr>
<tr>
<td>marray&amp; operatorOP(marray&amp; lhs, const marray&amp; rhs)</td>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half. Perform an in-place element-wise OP bitwise operation between each element of lhs marray and the rhs SYCL marray and return lhs marray. Where OP is: &amp;=,</td>
</tr>
<tr>
<td>marray&amp; operatorOP(marray&amp; lhs, const DataT &amp; rhs)</td>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half. Perform an in-place element-wise OP bitwise operation between each element of lhs marray and the rhs scalar and return a lhs marray. Where OP is: &amp;=,</td>
</tr>
<tr>
<td>marray&lt;br&gt;NumElements&gt; operatorOP(const marray &amp; lhs, const marray &amp; rhs)</td>
<td>Construct a new instance of the marray class template with DataT = bool and same NumElements as lhs marray with each element of the new marray instance the result of an element-wise OP logical operation between each element of lhs marray and each element of the rhs marray. Where OP is: &amp;&amp;,</td>
</tr>
</tbody>
</table>
Hidden friend function

marray<bool, NumElements> operatorOP(const marray<bool> lhs, const DataT& rhs)

Description

Construct a new instance of the marray class template with DataT = bool and same NumElements as lhs marray with each element of the new marray instance the result of an element-wise OP logical operation between each element of lhs marray and the rhs scalar.

Where OP is: &&, ||.

marray operatorOP(const marray& lhs, const marray& rhs)

Available only when: DataT != float && DataT != double && DataT != half.

Construct a new instance of the SYCL marray class template with the same template parameters as lhs marray with each element of the new SYCL marray instance the result of an element-wise OP bit-shift operation between each element of lhs marray and each element of the rhs SYCL marray. If OP is >>, DataT is a signed type and lhs marray has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0.

Where OP is: <<, >>.

marray operatorOP(const marray& lhs, const DataT& rhs)

Available only when: DataT != float && DataT != double && DataT != half.

Construct a new instance of the SYCL marray class template with the same template parameters as lhs marray with each element of the new SYCL marray instance the result of an element-wise OP bit-shift operation between each element of lhs marray and the rhs scalar. If OP is >>, DataT is a signed type and lhs marray has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0.

Where OP is: <<, >>.
### Hidden friend function

<table>
<thead>
<tr>
<th>marray &amp; operatorOP(marray &amp; lhs, const marray &amp; rhs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half.</td>
</tr>
<tr>
<td>Perform an in-place element-wise OP bitshift operation between each element of lhs marray and the rhs SYCL marray and returns lhs marray. If OP is &gt;&gt;=, DataT is a signed type and lhs marray has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0.</td>
</tr>
<tr>
<td>Where OP is: &lt;&lt;=, &gt;&gt;=.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>marray &amp; operatorOP(marray &amp; lhs, const DataT &amp; rhs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Available only when: DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half.</td>
</tr>
<tr>
<td>Perform an in-place element-wise OP bitshift operation between each element of lhs marray and the rhs scalar and returns a reference to this SYCL marray. If OP is &gt;&gt;=, DataT is a signed type and lhs marray has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0.</td>
</tr>
<tr>
<td>Where OP is: &lt;&lt;=, &gt;&gt;=.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>marray&lt; bool, NumElements&gt; operatorOP(const marray &amp; lhs, const marray &amp; rhs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>Construct a new instance of the marray class template with DataT = bool and same NumElements as lhs marray with each element of the new marray instance is the result of an element-wise OP relational operation between each element of lhs marray and each element of the rhs marray. The ==, &lt;, &lt;= and &gt;= operations result in false if either the lhs element or the rhs element is a NaN. The != operation results in true if either the lhs element or the rhs element is a NaN.</td>
</tr>
<tr>
<td>Where OP is: ==, !=, &lt;, &lt;=, &gt;=.</td>
</tr>
</tbody>
</table>
### Hidden friend function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{marray&lt;bool, NumElements&gt; operatorOP(const marray&amp; lhs, const DataT&amp; rhs)}</td>
<td>Construct a new instance of the \texttt{marray} class template with \texttt{DataT = bool} and same \texttt{NumElements} as \texttt{lhs marray} with each element of the new \texttt{marray} instance the result of an element-wise \texttt{OP} relational operation between each element of \texttt{lhs marray} and the \texttt{rhs scalar}. The \texttt{==}, \texttt{&lt;}, \texttt{&lt;=} and \texttt{&gt;=} operations result in \texttt{false} if either the \texttt{lhs} element or the \texttt{rhs} is a NaN. The \texttt{!=} operation results in \texttt{true} if either the \texttt{lhs} element or the \texttt{rhs} is a NaN. Where \texttt{OP} is: \texttt{==, !=, &lt;, &lt;=, &gt;=}.</td>
</tr>
<tr>
<td>\texttt{marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)}</td>
<td>If \texttt{OP} is \texttt{%}, available only when: \texttt{DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half}. Construct a new instance of the SYCL marray class template with the same template parameters as the \texttt{rhs SYCL marray} with each element of the new SYCL marray instance the result of an element-wise \texttt{OP} arithmetic operation between the \texttt{lhs} scalar and each element of the \texttt{rhs SYCL marray}. Where \texttt{OP} is: \texttt{+, -, *, /, %}.</td>
</tr>
<tr>
<td>\texttt{marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)}</td>
<td>Available only when: \texttt{DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half}. Construct a new instance of the SYCL marray class template with the same template parameters as the \texttt{rhs SYCL marray} with each element of the new SYCL marray instance the result of an element-wise \texttt{OP} bitwise operation between the \texttt{lhs} scalar and each element of the \texttt{rhs SYCL marray}. Where \texttt{OP} is: \texttt{&amp;}, \texttt{</td>
</tr>
<tr>
<td>Hidden friend function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>marray&lt;bool, NumElements&gt; operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</code></td>
<td>Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>rhs marray</code> with each element of the new <code>marray</code> instance the result of an element-wise <code>OP</code> logical operation between the <code>lhs</code> scalar and each element of the <code>rhs marray</code>. Where <code>OP</code> is: `&amp;&amp;,</td>
</tr>
<tr>
<td><code>marray operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</code></td>
<td>Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as the <code>rhs</code> SYCL <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> bitshift operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> SYCL <code>marray</code>. If <code>OP</code> is <code>&gt;&gt;</code>, <code>DataT</code> is a signed type and this SYCL <code>marray</code> has a negative value any vacated bits viewed as an unsigned integer must be assigned the value 1, otherwise any vacated bits viewed as an unsigned integer must be assigned the value 0. Where <code>OP</code> is: <code>&lt;&lt;, &gt;&gt;</code>.</td>
</tr>
<tr>
<td><code>marray&lt;bool, NumElements&gt; operatorOP(const DataT&amp; lhs, const marray&amp; rhs)</code></td>
<td>Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same <code>NumElements</code> as <code>rhs marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise <code>OP</code> relational operation between the <code>lhs</code> scalar and each element of the <code>rhs</code> marray. The <code>==, &lt;, &lt;=, !=, &gt;, &gt;=</code> operations result in <code>false</code> if either the <code>lhs</code> or the <code>rhs</code> element is a NaN. The <code>!=</code> operation results in <code>true</code> if either the <code>lhs</code> or the <code>rhs</code> element is a NaN. Where <code>OP</code> is: <code>==, !=, &lt;, &gt;, &lt;=, &gt;=</code>.</td>
</tr>
</tbody>
</table>
Hidden friend function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>marray&amp; operator~(const marray&amp; v)</code></td>
<td>Available only when: <code>DataT != float &amp;&amp; DataT != double &amp;&amp; DataT != half</code>. Construct a new instance of the SYCL <code>marray</code> class template with the same template parameters as <code>v</code> <code>marray</code> with each element of the new SYCL <code>marray</code> instance the result of an element-wise OP bitwise operation on each element of <code>v</code> <code>marray</code>.</td>
</tr>
<tr>
<td><code>marray&lt;bool, NumElements&gt; operator!(const marray&amp; v)</code></td>
<td>Construct a new instance of the <code>marray</code> class template with <code>DataT = bool</code> and same NumElements as <code>v</code> <code>marray</code> with each element of the new <code>marray</code> instance the result of an element-wise logical ! operation on each element of <code>v</code> <code>marray</code>.</td>
</tr>
</tbody>
</table>

### 4.14.3.2. Aliases

The SYCL programming API provides all permutations of the type alias:

```
using m<type><elems> = marray<<storage-type>, <elems>>
```

where `<elems>` is 2, 3, 4, 8 and 16, and pairings of `<type>` and `<storage-type>` for integral types are `char` and `int8_t`, `uchar` and `uint8_t`, `short` and `int16_t`, `ushort` and `uint16_t`, `int` and `int32_t`, `uint` and `uint32_t`, `long` and `int64_t`, `ulong` and `uint64_t`, for floating point types are both `half`, `float` and `double`, and for boolean type `bool`.

For example `muint4` is the alias to `marray<uint32_t, 4>` and `mfloat16` is the alias to `marray<float, 16>`.

### 4.14.3.3. Memory layout and alignment

The elements of an instance of the `marray` class template as if stored in `std::array<DataT, NumElements>`.

### 4.15. Synchronization and atomics

The available features are:

- **Accessor classes**: Accessor classes specify acquisition and release of buffer and image data structures to provide points at which underlying queue synchronization primitives must be generated.
- **Atomic operations**: SYCL devices support a restricted subset of C++ atomics and SYCL uses the library syntax from the next C++ specification to make this available.
- **Fences**: Fence primitives are made available to order loads and stores. They are exposed through the `atomic_fence` function. Fences can have acquire semantics, release semantics or both.
- **Barriers**: Barrier primitives are made available to synchronize sets of work-items within individual groups. They are exposed through the `group_barrier` function.
- **Hierarchical parallel dispatch**: In the hierarchical parallelism model of describing computations, synchronization within the work-group is made explicit through multiple instances of the `parallel_for_work_item` function call, rather than through the use of explicit work-group barrier operations.
- **Device event**: they are used inside SYCL kernel functions to wait for asynchronous operations within...
4.15.1. Barriers and fences

A group barrier or mem-fence provides memory ordering semantics over both the local address space and global address space. A mem-fence provides control over the re-ordering of memory load and store operations, subject to the associated memory order and memory scope, when paired with synchronization through an atomic object.

```cpp
namespace sycl {
    void atomic_fence(memory_order order, memory_scope scope);
}
```

The effects of a call to atomic_fence depend on the value of the order parameter:

- `memory_order::relaxed`: No effect
- `memory_order::acquire`: Acquire fence
- `memory_order::release`: Release fence
- `memory_order::acq_rel`: Both an acquire fence and a release fence
- `memory_order::seq_cst`: A sequentially consistent acquire and release fence

A group barrier acts as both an acquire fence and a release fence: all work-items in the group execute a release fence prior to synchronizing at the barrier, and all work-items in the group execute an acquire fence afterwards. A group barrier provides implicit atomic synchronization as if through an internal atomic object, such that the acquire and release fences associated with the barrier synchronize with each other, without an explicit atomic operation being required on an atomic object to synchronize the fences.

4.15.2. device_event class

The SYCL device_event class encapsulates a single SYCL device event which is available only within SYCL kernel functions and can be used to wait for asynchronous operations within a SYCL kernel function to complete.

All member functions of the device_event class must not throw a SYCL exception.

A synopsis of the SYCL device_event class is provided below. The constructors and member functions of the SYCL device_event class are listed in Table 152 and Table 151 respectively.

```cpp
namespace sycl {
    class device_event {
        device_event(__unspecified__);  
    public:
        void wait() noexcept;
    }
}
```

Table 151. Member functions of the SYCL device_event class
4.15.3. Atomic references

The `sycl::atomic_ref` class provides the ability to perform atomic operations in device code with a syntax similar to the C++ standard `std::atomic_ref`. The `sycl::atomic_ref` class must not be used in host code.

Unlike `std::atomic_ref`, `sycl::atomic_ref` does not provide a default memory ordering for its operations. Instead, the application must specify a default ordering via the `DefaultOrder` template parameter. This ordering is used as a default for most of the atomic operations, but most member functions also provide an optional parameter that allows the application to override this default. The set of supported orderings is specific to a device, but every device is guaranteed to support at least `memory_order::relaxed`. If the default order is set to `memory_order::relaxed`, all memory order arguments default to `memory_order::relaxed`. If the default order is set to `memory_order::acq_rel`, memory order arguments default to `memory_order::acquire` for load operations, `memory_order::release` for store operations and `memory_order::acq_rel` for read-modify-write operations. If the default order is set to `memory_order::seq_cst`, all memory order arguments default to `memory_order::seq_cst`.

The `sycl::atomic_ref` class has a template parameter `DefaultScope`, which allows the application to define a default memory scope for the atomic operations. Most member functions also provide an optional parameter that allows the application to override this default.

The `sycl::atomic_ref` class also has a template parameter `AddressSpace`, which allows the application to make an assertion about the address space of the object of type `T` that it references. The default value for this parameter is `access::address_space::generic_space`, which indicates that the object could be in either the global or local address spaces. If the application knows the address space, it can set this template parameter to either `access::address_space::global_space` or `access::address_space::local_space` as an assertion to the implementation. Specifying the address space via this template parameter may allow the implementation to perform certain optimizations. Specifying an address space that does not match the object’s actual address space results in undefined behavior.

The template parameter `T` must be one of the following types:

- `int`,
- `unsigned int`,
- `long`,
- `unsigned long`,
- `long long`,
- `unsigned long long`,
- `float`, or
- `double`.

In addition, the type `T` must satisfy one of the following conditions:

---

### Table 152. Constructors of the `device_event` class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device_event(___unspecified___)</code></td>
<td>Unspecified implementation-defined constructor.</td>
</tr>
</tbody>
</table>
• `sizeof(T) == 4`, or
• `sizeof(T) == 8` and the code containing this `atomic_ref` was submitted to a device that has `aspect::atomic64`.

For floating-point types, the member functions of the `atomic_ref` class may be emulated, and they may use a different floating-point environment from those defined by `info::device::single_fp_config` and `info::device::double_fp_config` (i.e. floating-point atomics may use different rounding modes and may have different exception behavior).

The atomic types are defined as follows.

```cpp
namespace sycl {
    // Exposition only
    template <memory_order ReadModifyWriteOrder>
    struct memory_order_traits;

    template <>
    struct memory_order_traits<memory_order::relaxed> {
        static constexpr memory_order read_order = memory_order::relaxed;
        static constexpr memory_order write_order = memory_order::relaxed;
    };

    template <>
    struct memory_order_traits<memory_order::acq_rel> {
        static constexpr memory_order read_order = memory_order::acquire;
        static constexpr memory_order write_order = memory_order::release;
    };

    template <>
    struct memory_order_traits<memory_order::seq_cst> {
        static constexpr memory_order read_order = memory_order::seq_cst;
        static constexpr memory_order write_order = memory_order::seq_cst;
    };

    template <typename T, memory_order DefaultOrder, memory_scope DefaultScope, access::address_space AddressSpace = access::address_space::generic_space>
    class atomic_ref {
    public:
        using value_type = T;
        static constexpr size_t required_alignment = /* implementation-defined */;
        static constexpr bool is_always_lock_free = /* implementation-defined */;
        static constexpr memory_order default_read_order = memory_order_traits<DefaultOrder>::read_order;
        static constexpr memory_order default_write_order = memory_order_traits<DefaultOrder>::write_order;
        static constexpr memory_order default_read_modify_write_order = DefaultOrder;
        static constexpr memory_scope default_scope = DefaultScope;

        bool is_lock_free() const noexcept;

        explicit atomic_ref(T&);
        atomic_ref(const atomic_ref&) noexcept;
        atomic_ref& operator=(const atomic_ref&) = delete;

        void store(T operand, memory_order order = default_write_order, memory_scope scope = default_scope) const noexcept;

        T operator=(T desired) const noexcept;
    }
}`
T load(memory_order order = default_read_order,
        memory_scope scope = default_scope) const noexcept;

operator T() const noexcept;

T exchange(T operand, memory_order order = default_read_modify_write_order,
            memory_scope scope = default_scope) const noexcept;

bool compare_exchange weak(T expected, T desired, memory_order success,
                            memory_order failure,
                            memory_scope scope = default_scope) const noexcept;

bool compare_exchange_weak(T expected, T desired,
                            memory_order order = default_read_modify_write_order,
                            memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(T expected, T desired,
                              memory_order order = default_read_modify_write_order,
                              memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(T expected, T desired,
                              memory_order order = default_read_modify_write_order,
                              memory_scope scope = default_scope) const noexcept;

/* All other members from atomic_ref<T> are available */

using difference_type = value_type;

Integral fetch_add(Integral operand,
                     memory_order order = default_read_modify_write_order,
                     memory_scope scope = default_scope) const noexcept;

Integral fetch_sub(Integral operand,
                    memory_order order = default_read_modify_write_order,
                    memory_scope scope = default_scope) const noexcept;

Integral fetch_and(Integral operand,
                   memory_order order = default_read_modify_write_order,
                   memory_scope scope = default_scope) const noexcept;

Integral fetch_or(Integral operand,
                  memory_order order = default_read_modify_write_order,
                  memory_scope scope = default_scope) const noexcept;

Integral fetch_xor(Integral operand,
                   memory_order order = default_read_modify_write_order,
                   memory_scope scope = default_scope) const noexcept;
Integral fetch_min(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral fetch_max(Integral operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Integral operator++(int) const noexcept;
Integral operator--(int) const noexcept;
Integral operator++() const noexcept;
Integral operator--() const noexcept;
Integral operator+=(Integral) const noexcept;
Integral operator-=(Integral) const noexcept;
Integral operator&=(Integral) const noexcept;
Integral operator|=(Integral) const noexcept;
Integral operator^=(Integral) const noexcept;
};

// Partial specialization for floating-point types
template<typename T,
    memory_order DefaultOrder, memory_scope DefaultScope,
    access::address_space AddressSpace = access::address_space::generic_space>
class atomic_ref<Floating, DefaultOrder, DefaultScope, AddressSpace> { }

/* All other members from atomic_ref<T> are available */
using difference_type = value_type;

Floating fetch_add(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_sub(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_min(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating fetch_max(Floating operand,
    memory_order order = default_read_modify_write_order,
    memory_scope scope = default_scope) const noexcept;

Floating operator+=(Floating) const noexcept;
Floating operator-=(Floating) const noexcept;
};

// Partial specialization for pointers
template<typename T,
    memory_order DefaultOrder, memory_scope DefaultScope,
    access::address_space AddressSpace = access::address_space::generic_space>
class atomic_ref<T*, DefaultOrder, DefaultScope, AddressSpace> { }

using value_type = T*;
using difference_type = ptrdiff_t;
static constexpr size_t required_alignment = /* implementation-defined */;
static constexpr bool is_always_lock_free = /* implementation-defined */;
static constexpr memory_order default_read_order =
    memory_order_traits<DefaultOrder>::read_order;
static constexpr memory_order default_write_order =
    memory_order_traits<DefaultOrder>::write_order;
static constexpr memory_order default_read_modify_write_order = DefaultOrder;
static constexpr memory_scope default_scope = DefaultScope;

bool is_lock_free() const noexcept;
explicit atomic_ref(T*&) noexcept;
atomic_ref(const atomic_ref&) noexcept = delete;

void store(T* operand, memory_order order = default_write_order,
            memory_scope scope = default_scope) const noexcept;
T* operator=(T* desired) const noexcept;
T* load(memory_order order = default_read_order,
         memory_scope scope = default_scope) const noexcept;
operator T*() const noexcept;
T* exchange(T* operand, memory_order order = default_read_modify_write_order,
            memory_scope scope = default_scope) const noexcept;
bool compare_exchange_weak(T*& expected, T* desired, memory_order success,
                            memory_order failure,
                            memory_scope scope = default_scope) const noexcept;
bool compare_exchange_weak(T*& expected, T* desired,
                            memory_order order = default_read_modify_write_order,
                            memory_scope scope = default_scope) const noexcept;

bool compare_exchange_strong(T*& expected, T* desired, memory_order success,
                            memory_order failure,
                            memory_scope scope = default_scope) const noexcept;
bool compare_exchange_strong(T*& expected, T* desired,
                            memory_order order = default_read_modify_write_order,
                            memory_scope scope = default_scope) const noexcept;

T* fetch_add(difference_type,
              memory_order order = default_read_modify_write_order,
              memory_scope scope = default_scope) const noexcept;
T* fetch_sub(difference_type,
              memory_order order = default_read_modify_write_order,
              memory_scope scope = default_scope) const noexcept;
T* operator++(int) const noexcept;
The constructors and member functions for instances of the SYCL `atomic_ref` class using any compatible type are listed in Table 153 and Table 154 respectively. Additional member functions for integral, floating-point and pointer types are listed in Table 155, Table 156 and Table 157 respectively.

The static member `required_alignment` describes the minimum required alignment in bytes of an object that can be referenced by an `atomic_ref<T>`, which must be at least `alignof(T)`.

The static member `is_always_lock_free` is true if all atomic operations for type `T` are always lock-free. A SYCL implementation is not guaranteed to support atomic operations that are not lock-free.

The static members `default_read_order`, `default_write_order` and `default_read_modify_write_order` reflect the default memory order values for each type of atomic operation, consistent with the `DefaultOrder` template.

The atomic operations and member functions behave as described in the C++ specification, barring the restrictions discussed above.

Care must be taken when using atomics for work-item coordination, because work-items are not required to provide stronger than weakly parallel forward progress guarantees. Operations that block a work-item, such as continuously checking the value of an atomic variable until some condition holds, or using atomic operations that are not lock-free, may prevent overall progress.

### Table 153. Constructors of the SYCL `atomic_ref` class template

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atomic_ref(T &amp; ref)</code></td>
<td>Constructs an instance of SYCL <code>atomic_ref</code> which is associated with the reference <code>ref</code>.</td>
</tr>
</tbody>
</table>

### Table 154. Member functions available on any object of type `atomic_ref<T>`

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool is_lock_free()</code> const</td>
<td>Return <code>true</code> if the atomic operations provided by this <code>atomic_ref</code> are lock-free.</td>
</tr>
<tr>
<td><code>void store(T operand, memory_order order = default_write_order, memory_scope scope = default_scope) const</code></td>
<td>Atomically stores <code>operand</code> to the object referenced by this <code>atomic_ref</code>. The memory order of this atomic operation must be <code>memory_order::relaxed</code>, <code>memory_order::release</code> or <code>memory_order::seq_cst</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>T operator=(T desired) const</td>
<td>Equivalent to store(desired). Returns desired.</td>
</tr>
<tr>
<td>T load(memory_order order = default_read_order memory_scope scope = default_scope) const</td>
<td>Atomically loads the value of the object referenced by this atomic_ref. The memory order of this atomic operation must be memory_order::relaxed, memory_order::acquire, or memory_order::seq_cst. This function is only supported for 64-bit data types on devices that have aspect::atomic64.</td>
</tr>
<tr>
<td>operator T() const</td>
<td>Equivalent to load().</td>
</tr>
<tr>
<td>T exchange(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</td>
<td>Atomically replaces the value of the object referenced by this atomic_ref with value operand and returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have aspect::atomic64.</td>
</tr>
<tr>
<td>bool compare_exchange_weak(T&amp; expected, T desired, memory_order success, memory_order failure, memory_scope scope = default_scope) const</td>
<td>Atomically compares the value of the object referenced by this atomic_ref against the value of expected. If the values are equal, attempts to replace the value of the referenced object with the value of desired; otherwise assigns the original value of the referenced object to expected. Returns true if the comparison operation and replacement operation were successful. The failure memory order of this atomic operation must be memory_order::relaxed, memory_order::acquire or memory_order::seq_cst.</td>
</tr>
<tr>
<td>bool compare_exchange_weak(T&amp; expected, T desired, default_read_modify_write_order, memory_scope scope = default_scope) const</td>
<td>Equivalent to compare_exchange_weak(expected, desired, order, order, scope).</td>
</tr>
</tbody>
</table>
Member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
</table>
| Atomically compares the value of the object referenced by this *atomic_ref* against the value of `expected`. If the values are equal, replaces the value of the referenced object with the value of `desired`; otherwise assigns the original value of the referenced object to `expected`.

Returns `true` if the comparison operation was successful. The failure memory order of this atomic operation must be `memory_order::relaxed`, `memory_order::acquire` or `memory_order::seq_cst`.

This function is only supported for 64-bit data types on devices that have `aspect::atomic64`.

![Table](image)

**Table 155. Additional member functions available on an object of type `atomic_ref<T>` for integral T**

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>bool compare_exchange_strong(T&amp; expected, T desired,</strong> memory_order success, <strong>memory_order failure,</strong> memory_scope scope = default_scope) const**</td>
<td></td>
</tr>
</tbody>
</table>
| **bool compare_exchange_strong(T& expected, T desired,** memory_order order = default_read_modify_write_order) const** | Equivalent to `compare_exchange_strong(expected, desired, order, order, scope)`.

| T fetch_add(T operand,** memory_order order = default_read_modify_write_order,** memory_scope scope = default_scope) const |
| T operator+=(T operand) const |
| T operator++(int) const |
| T operator++() const |

Equivalent to `fetch_add(operand) + operand`.

Equivalent to `fetch_add(1)`.

Equivalent to `fetch_add(1) + 1`.
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T fetch_sub(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</code></td>
<td>Atomically subtracts <code>operand</code> from the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><code>T operator-=(T operand) const</code></td>
<td>Equivalent to <code>fetch_sub(operand) - operand</code>.</td>
</tr>
<tr>
<td><code>T operator--(int) const</code></td>
<td>Equivalent to <code>fetch_sub(1)</code>.</td>
</tr>
<tr>
<td><code>T operator--() const</code></td>
<td>Equivalent to <code>fetch_sub(1) - 1</code>.</td>
</tr>
<tr>
<td><code>T fetch_and(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</code></td>
<td>Atomically performs a bitwise AND between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><code>T operator&amp;=(T operand) const</code></td>
<td>Equivalent to <code>fetch_and(operand) &amp; operand</code>.</td>
</tr>
<tr>
<td><code>T fetch_or(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</code></td>
<td>Atomically performs a bitwise OR between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td>`T operator</td>
<td>=(T operand) const`</td>
</tr>
</tbody>
</table>
### Member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically performs a bitwise XOR between <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

```cpp
T fetch_xor(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to <code>fetch_xor(operand) ^ operand</code>.</td>
</tr>
</tbody>
</table>

```cpp
T operator^=(T operand) const
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically computes the minimum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

```cpp
T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically computes the maximum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

```cpp
T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const
```

### Table 156. Additional member functions available on an object of type `atomic_ref<T>` for floating-point `T`

<table>
<thead>
<tr>
<th>Member function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Atomically adds <code>operand</code> to the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

```cpp
T fetch_add(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const
```

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to <code>fetch_add(operand) + operand</code>.</td>
</tr>
</tbody>
</table>

```cpp
T operator+=(T operand) const
```
### Member function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically subtracts <code>operand</code> from the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

| T fetch_sub(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to <code>fetch_sub(operand) - operand</code>.</td>
</tr>
</tbody>
</table>

| T operator-=(T operand) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically computes the minimum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

| T fetch_min(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically computes the maximum of <code>operand</code> and the value of the object referenced by this <code>atomic_ref</code>, and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

| T fetch_max(T operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const |

---

### Table 157. Additional member functions available on an object of type `atomic_ref<T*>`

<table>
<thead>
<tr>
<th>Member function</th>
</tr>
</thead>
<tbody>
<tr>
<td>T* fetch_add(ptrdiff_t operand, memory_order order = default_read_modify_write_order, memory_scope scope = default_scope) const</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomically adds <code>operand</code> to the value of the object referenced by this <code>atomic_ref</code> and assigns the result to the value of the referenced object. Returns the original value of the referenced object. This function is only supported for 64-bit pointers on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

| T* operator+=(ptrdiff_t operand) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to <code>fetch_add(operand) + operand</code>.</td>
</tr>
</tbody>
</table>

| T* operator++(int) const |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent to <code>fetch_add(1)</code>.</td>
</tr>
</tbody>
</table>
### 4.15.4. Atomic types (deprecated)

The atomic types and operations on atomic types provided by SYCL 1.2.1 are deprecated in SYCL 2020, and will be removed in a future version of SYCL. The types and operations are made available in the `cl::sycl::` namespace for backwards compatibility.

The constructors and member functions for the `cl::sycl::atomic` class are listed in Table 158 and Table 159 respectively.

```cpp
namespace cl {
namespace sycl {

/* Deprecated in SYCL 2020 */
enum class memory_order : /* unspecified */ { relaxed };

/* Deprecated in SYCL 2020 */
template <typename T, access::address_space AddressSpace = access::address_space::global_space>
class atomic {
    public:
        template <typename PointerT, access::decorated IsDecorated>
        atomic(multi_ptr<PointerT, AddressSpace, IsDecorated> ptr);

        void store(T operand, memory_order memoryOrder = memory_order::relaxed);
        T load(memory_order memoryOrder = memory_order::relaxed) const;
        T exchange(T operand, memory_order memoryOrder = memory_order::relaxed);

        /* Available only when: T != float */
        bool compare_exchange_strong(T& expected, T& desired,}
```
The global functions are as follows and described in Table 160.

```cpp
namespace cl {
namespace sycl {
    void atomic_store(atomic<T, AddressSpace> object, T operand,
                       memory_order memoryOrder = memory_order::relaxed);

    T atomic_load(atomic<T, AddressSpace> object,
                   memory_order memoryOrder = memory_order::relaxed);

    T atomic_exchange(atomic<T, AddressSpace> object, T operand,
                      memory_order memoryOrder = memory_order::relaxed);

    bool atomic_compare_exchange_strong(
        atomic<T, AddressSpace> object, T& expected, T desired,
        memory_order successMemoryOrder = memory_order::relaxed,
        memory_order failMemoryOrder = memory_order::relaxed);
};
} // namespace sycl
} // namespace cl
```
template <typename T, access::address_space AddressSpace>
T atomic_fetch_add(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_sub(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_and(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_or(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_xor(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_min(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

template <typename T, access::address_space AddressSpace>
T atomic_fetch_max(atomic<T, AddressSpace> object, T operand,
    memory_order memoryOrder = memory_order::relaxed);

namespace sycl
} // namespace sycl
} // namespace cl

Table 158. Constructors of the cl::sycl::atomic class template

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;typename pointerT&gt; atomic(multi_ptr&lt;pointerT, AddressSpace&gt; ptr)</td>
<td>Deprecated in SYCL 2020. Permitted data types for pointerT are any valid scalar data type which is the same size in bytes as T. Constructs an instance of SYCL atomic which is associated with the pointer ptr, converted to a pointer of data type T.</td>
</tr>
</tbody>
</table>

Table 159. Member functions available on an object of type cl::sycl::atomic<T>
<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void store(T operand, memory_order memoryOrder = memory_order::relaxed)</strong></td>
<td>Deprecated in SYCL 2020.</td>
</tr>
<tr>
<td></td>
<td>Atomically stores the value <code>operand</code> at the address of the <code>multi_ptr</code> associated with this SYCL</td>
</tr>
<tr>
<td></td>
<td><code>atomic</code>. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This</td>
</tr>
<tr>
<td></td>
<td>function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><strong>T load(memory_order memoryOrder = memory_order::relaxed)</strong></td>
<td>Deprecated in SYCL 2020.</td>
</tr>
<tr>
<td></td>
<td>Atomically loads the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code></td>
</tr>
<tr>
<td></td>
<td>Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before</td>
</tr>
<tr>
<td></td>
<td>the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This</td>
</tr>
<tr>
<td></td>
<td>function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><strong>T exchange(T operand, memory_order memoryOrder = memory_order::relaxed)</strong></td>
<td>Deprecated in SYCL 2020.</td>
</tr>
<tr>
<td></td>
<td>Atomically replaces the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code></td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>bool compare_exchange_strong(&lt;br&gt;  T&amp; expected, T desired,&lt;br&gt;  memory_order successMemoryOrder = memory_order::relaxed,&lt;br&gt;  memory_order failMemoryOrder = memory_order::relaxed)</td>
<td>Deprecated in SYCL 2020.&lt;br&gt;Available only when: T != float.&lt;br&gt;Atomically compares the value at the address of the multi_ptr associated with this SYCL atomic against the value of expected. If the values are equal, replaces value at address of the multi_ptr associated with this SYCL atomic with the value of desired; otherwise assigns the original value at the address of the multi_ptr associated with this SYCL atomic to expected. Returns true if the comparison operation was successful. The memory order of this atomic operation must be memory_order::relaxed for both success and fail. This function is only supported for 64-bit data types on devices that have aspect::atomic64.</td>
</tr>
<tr>
<td>T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed)</td>
<td>Deprecated in SYCL 2020.&lt;br&gt;Available only when: T != float.&lt;br&gt;Atomically adds the value operand to the value at the address of the multi_ptr associated with this SYCL atomic and assigns the result to the value at the address of the multi_ptr associated with this SYCL atomic. Returns the value at the address of the multi_ptr associated with this SYCL atomic before the call. The memory order of this atomic operation must be memory_order::relaxed. This function is only supported for 64-bit data types on devices that have aspect::atomic64.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically subtracts the value <code>operand</code> to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><code>T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically performs a bitwise AND between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically performs a bitwise OR between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><code>T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically performs a bitwise XOR between the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td>Member function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically computes the minimum of the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
<tr>
<td><code>T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Available only when: <code>T != float</code>. Atomically computes the maximum of the value <code>operand</code> and the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> and assigns the result to the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code>. Returns the value at the address of the <code>multi_ptr</code> associated with this SYCL <code>atomic</code> before the call. The memory order of this atomic operation must be <code>memory_order::relaxed</code>. This function is only supported for 64-bit data types on devices that have <code>aspect::atomic64</code>.</td>
</tr>
</tbody>
</table>

Table 160. Global functions available on atomic types

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_load(atomic&lt;T, AddressSpace&gt; object, memory_order memoryOrder = memory_order::relaxed)</code></td>
<td>Deprecated in SYCL 2020. Equivalent to calling <code>object.load(memoryOrder)</code>.</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Description</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td><code>template &lt;typename T, access::address_space AddressSpace&gt; void atomic_store(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
</tr>
<tr>
<td>Equivalent to calling <code>object.store(operand, memoryOrder)</code></td>
<td><code>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_exchange(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
</tr>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td><code>template &lt;typename T, access::address_space AddressSpace&gt; bool atomic_compare_exchange_strong(atomic&lt;T, AddressSpace&gt; object, T&amp; expected, T desired, memory_order successMemoryOrder = memory_order::relaxed, memory_order failMemoryOrder = memory_order::relaxed)</code></td>
</tr>
<tr>
<td>Equivalent to calling <code>object.compare_exchange_strong(expected, desired, successMemoryOrder, failMemoryOrder)</code></td>
<td><code>template &lt;typename T, access::address_space AddressSpace&gt; T atomic_fetch_add(atomic&lt;T, AddressSpace&gt; object, T operand, memory_order memoryOrder = memory_order::relaxed)</code></td>
</tr>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td>Equivalent to calling <code>object.fetch_add(operand, memoryOrder)</code></td>
</tr>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td>Equivalent to calling <code>object.fetch_sub(operand, memoryOrder)</code></td>
</tr>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td>Equivalent to calling <code>object.fetch_and(operand, memoryOrder)</code></td>
</tr>
<tr>
<td>Deprecated in SYCL 2020.</td>
<td>Equivalent to calling <code>object.fetch_or(operand, memoryOrder)</code></td>
</tr>
</tbody>
</table>
### 4.15.5. Interaction with host code

When a kernel runs on a device that has either `aspect::usm_atomic_host_allocations` or `aspect::usm_atomic_shared_allocations`, the device code and the host code can concurrently access the same memory. This has a ramification on the atomic operations because it is possible for device code and host code to perform atomic operations on the same object `M` in this shared memory. It also has a ramification on the fence operations because the C++ core language defines the semantics of these fence operations in relation to atomic operations on some shared object `M`. The following paragraphs specify the guarantees that the SYCL implementation provides when the application performs atomic or fence operations in device code using the memory scope `memory_scope::system`.

Atomic operations in device code using `sycl::atomic_ref` on an object `M` are guaranteed to be atomic with respect to atomic operations in host code using `std::atomic_ref` on that same object `M`.

Fence operations in device code using `sycl::atomic_fence` synchronize with fence operations in host code using `std::atomic_thread_fence` if the fence operations shared the same atomic object `M` and follow the rules for fence synchronization defined in the C++ core language.

Fence operations in device code using `sycl::atomic_fence` synchronize with atomic operations in host code using `std::atomic_ref` if the operations share the same atomic object `M` and follow the rules for fence synchronization defined in the C++ core language.

Atomic operations in device code using `sycl::atomic_ref` synchronize with fence operations in host code using `std::atomic_thread_fence` if the operations share the same atomic object `M` and follow the rules for fence synchronization defined in the C++ core language.

### 4.16. Stream class

The SYCL `stream` class is a buffered output stream that allows outputting the values of built-in, vector and SYCL types to the console. The implementation of how values are streamed to the console is left as an implementation detail.

The way in which values are output by an instance of the SYCL `stream` class can also be altered using a...
range of manipulators.

There are two limits that are relevant for the stream class. The totalBufferSize limit specifies the maximum size of the overall character stream that can be output during a kernel invocation, and the workItemBufferSize limit specifies the maximum size of the character stream that can be output within a work item before a flush must be performed. Both of these limits are specified in bytes. The totalBufferSize limit must be sufficient to contain the characters output by all stream statements during execution of a kernel invocation (the aggregate of outputs from all work items), and the workItemBufferSize limit must be sufficient to contain the characters output within a work item between stream flush operations.

If the totalBufferSize or workItemBufferSize limits are exceeded, it is implementation-defined whether the streamed characters exceeding the limit are output, or silently ignored/discarded, and if output it is implementation-defined whether those extra characters exceeding the workItemBufferSize limit count toward the totalBufferSize limit. Regardless of this implementation defined behavior of output exceeding the limits, no undefined or erroneous behavior is permitted of an implementation when the limits are exceeded. Unused characters within workItemBufferSize (any portion of the workItemBufferSize capacity that has not been used at the time of a stream flush) do not count toward the totalBufferSize limit, in that only characters flushed count toward the totalBufferSize limit.

The SYCL stream class provides the common reference semantics (see Section 4.5.2).

### 4.16.1. Stream class interface

The constructors and member functions of the SYCL stream class are listed in Table 163, Table 164, and Table 165 respectively. The additional common special member functions and common member functions are listed in Table 7 and Table 8, respectively.

The operand types that are supported by the SYCL stream class operator<<() operator are listed in Table 161.

The manipulators that are supported by the SYCL stream class operator<<() operator are listed in Table 162.

```cpp
namespace sycl {

enum class stream_manipulator : /* unspecified */ {
  flush,
  dec,
  hex,
  oct,
  noshowbase,
  showbase,
  noshowpos,
  showpos,
  endl,
  fixed,
  scientific,
  hexfloat,
  defaultfloat
};

const stream_manipulator flush = stream_manipulator::flush;
const stream_manipulator dec = stream_manipulator::dec;
const stream_manipulator hex = stream_manipulator::hex;
```
const stream_manipulator oct = stream_manipulator::oct;
const stream_manipulator noshowbase = stream_manipulator::noshowbase;
const stream_manipulator showbase = stream_manipulator::showbase;
const stream_manipulator noshowpos = stream_manipulator::noshowpos;
const stream_manipulator showpos = stream_manipulator::showpos;
const stream_manipulator endl = stream_manipulator::endl;
const stream_manipulator fixed = stream_manipulator::fixed;
const stream_manipulator scientific = stream_manipulator::scientific;
const stream_manipulator hexfloat = stream_manipulator::hexfloat;
const stream_manipulator defaultfloat = stream_manipulator::defaultfloat;
__precision_manipulator__ setprecision(int precision);
__width_manipulator__ setw(int width);

class stream {
public:
    stream(size_t totalBufferSize, size_t workItemBufferSize,
           handler& cgh, const property_list& propList = {});
    size_t size() const noexcept;
    size_t get_size() const;
    size_t get_work_item_buffer_size() const;
    size_t get_max_statement_size() const;
};

template<typename T> const stream& operator<<(const stream& os, const T& rhs);

} // namespace sycl

Table 161. Operand types supported by the stream class
## Stream operand type

| Description                                                                                   |
|---------------------------------------------------------------------------------------------|---|
| char, signed char, unsigned char, int, unsigned int, short, unsigned short, long int, unsigned long int | Outputs the value as a stream of characters. |
| float, double, half                                                                         | Outputs the value according to the precision of the current statement as a stream of characters. |
| char*, const char*                                                                           | Outputs the string. |
| T*, const T*, multi_ptr                                                                      | Outputs the address of the pointer as a stream of characters. |
| vec                                                                                         | Outputs the value of each component of the vector as a stream of characters. |
| id, range, item, nd_item, group, nd_range, h_item                                             | Outputs the value of each component of each id or range as a stream of characters. |

### Table 162. Manipulators supported by the stream class

<table>
<thead>
<tr>
<th>Stream manipulator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flush</td>
<td>Triggers a flush operation, which synchronizes the work item stream buffer with the global stream buffer, and then empties the work item stream buffer. After a flush, the full <code>workItemBufferSize</code> is available again for subsequent streaming within the work item.</td>
</tr>
<tr>
<td>endl</td>
<td>Outputs a new-line character and then triggers a flush operation.</td>
</tr>
<tr>
<td>dec</td>
<td>Outputs any subsequent values in the current statement in decimal base.</td>
</tr>
<tr>
<td>hex</td>
<td>Outputs any subsequent values in the current statement in hexadecimal base.</td>
</tr>
<tr>
<td>oct</td>
<td>Outputs any subsequent values in the current statement in octal base.</td>
</tr>
<tr>
<td>noshowbase</td>
<td>Outputs any subsequent values without the base prefix.</td>
</tr>
<tr>
<td>showbase</td>
<td>Outputs any subsequent values with the base prefix.</td>
</tr>
<tr>
<td>Stream manipulator</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>noshowpos</td>
<td>Outputs any subsequent values without a plus sign if the value is positive.</td>
</tr>
<tr>
<td>showpos</td>
<td>Outputs any subsequent values with a plus sign if the value is positive.</td>
</tr>
<tr>
<td>setw(int)</td>
<td>Sets the field width of any subsequent values in the current statement.</td>
</tr>
<tr>
<td>setprecision(int)</td>
<td>Sets the precision of any subsequent values in the current statement.</td>
</tr>
<tr>
<td>fixed</td>
<td>Outputs any subsequent floating-point values in the current statement in fixed notation.</td>
</tr>
<tr>
<td>scientific</td>
<td>Outputs any subsequent floating-point values in the current statement in scientific notation.</td>
</tr>
<tr>
<td>hexfloat</td>
<td>Outputs any subsequent floating-point values in the current statement in hexadecimal notation.</td>
</tr>
<tr>
<td>defaultfloat</td>
<td>Outputs any subsequent floating-point values in the current statement in the default notation.</td>
</tr>
</tbody>
</table>

Table 163. Constructors of the stream class

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream(size_t totalBufferSize, size_t workItemBufferSize, handler&amp; cgh, const property_list&amp; propList = {})</code></td>
<td>Constructs a SYCL stream instance associated with the command group specified by cgh, with a maximum buffer size in bytes per kernel invocation specified by the parameter <code>totalBufferSize</code>, and a maximum stream size that can be buffered by a work item between stream flushes specified by the parameter <code>workItemBufferSize</code>. Zero or more properties can be provided to the constructed SYCL stream via an instance of <code>property_list</code>.</td>
</tr>
</tbody>
</table>

Table 164. Member functions of the stream class

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t size() const noexcept</code></td>
<td>Returns the total buffer size, in bytes.</td>
</tr>
</tbody>
</table>
### 4.16.2. Synchronization

An instance of the SYCL `stream` class is required to synchronize with the host, and must output everything that is streamed to it via the `operator<<()` operator before a flush operation (that doesn't exceed the `workItemBufferSize` or `totalBufferSize` limits) within a SYCL kernel function by the time that the event associated with a command group submission enters the completed state. The point at which this synchronization occurs and the member function by which this synchronization is performed are implementation-defined. For example it is valid for an implementation to use `printf()`.

The SYCL `stream` class is required to output the content of each stream, between flushes (up to `workItemBufferSize`), without mixing with content from the same stream in other work items. There are no other output order guarantees between work items or between streams. The stream flush operation therefore delimits the unit of output that is guaranteed to be displayed without mixing with other work items, with respect to a single stream.

### 4.16.3. Implicit flush

There is guaranteed to be an implicit flush of each stream used by a kernel, at the end of kernel execution, from the perspective of each work item. There is also an implicit flush when the `endl` stream manipulator is executed. No other implicit flushes are permitted in an implementation.

### 4.16.4. Performance note

The usage of the `stream` class is designed for debugging purposes and is therefore not recommended for performance critical applications.

### 4.17. SYCL built-in functions for SYCL host and device

SYCL kernels may execute on any SYCL device, which requires the functions used in the kernels to be compiled and linked for both device and host. In the SYCL programming model, the built-ins are available for the entire SYCL application within the `sycl` namespace, although their semantics may be different. This section follows the OpenCL 1.2 specification document ch. 6.12 - except that for SYCL, all functions are located within the `sycl` namespace - and describes the behavior of these functions for SYCL host and device. The expected precision and any other semantic requirements are defined in the back-end specification.
The SYCL built-in functions are available throughout the SYCL application, and depending on where they execute, they are either implemented using their host implementation or the device implementation. The SYCL system guarantees that all of the built-in functions fulfill the same requirements for both host and device.

### 4.17.1. Description of the built-in types available for SYCL host and device

All of the OpenCL built-in types are available in the namespace `sycl`. For the purposes of this document we use generic type names for describing sets of valid SYCL types. The generic type names themselves are not valid SYCL types, but they represent a set of valid types, as defined in Table 166. Each generic type within a section is comprised of a combination of scalar, SYCL `vec` and/or `marray` class specializations. The letters `{n}` and `{N}` define valid sizes for class specializations, where `{n}` means 2,3,4,8,16 and `{N}` means any positive value of size_t type. Note that any reference to the base type refers to the type of a scalar or the element type of a SYCL `vec` or `marray` specialization.

*Table 166. Generic type name description, which serves as a description for all valid types of parameters to kernel functions.*

<table>
<thead>
<tr>
<th>Generic type name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>floatn</code></td>
<td><code>float{n}</code>, <code>mfloat{n}</code>, <code>marray&lt;{N},float&gt;</code></td>
</tr>
<tr>
<td><code>genfloatf</code></td>
<td><code>float, floatn</code></td>
</tr>
<tr>
<td><code>doublen</code></td>
<td><code>double{n}</code>, <code>mdouble{n}</code>, <code>marray&lt;{N},double&gt;</code></td>
</tr>
<tr>
<td><code>genfloatd</code></td>
<td><code>double, doublen</code></td>
</tr>
<tr>
<td><code>halfn</code></td>
<td><code>half{n}, mhalf{n}, marray&lt;{N},half&gt;</code></td>
</tr>
<tr>
<td><code>genfloath</code></td>
<td><code>half, halfn</code></td>
</tr>
<tr>
<td><code>genfloat</code></td>
<td><code>genfloatf, genfloatd, genfloath</code></td>
</tr>
<tr>
<td><code>sgenfloat</code></td>
<td><code>float, double, half</code></td>
</tr>
<tr>
<td><code>vgenfloat</code></td>
<td><code>float{n}, double{n}, half{n}</code></td>
</tr>
<tr>
<td><code>mgenfloat</code></td>
<td><code>marray&lt;float,{N}&gt;, marray&lt;double,{N}&gt;, marray&lt;half,{N}&gt;</code></td>
</tr>
<tr>
<td><code>gengeofloat</code></td>
<td><code>float, float2, float3, float4, mfloat2, mfloat3, mfloat4</code></td>
</tr>
<tr>
<td><code>gengeodouble</code></td>
<td><code>double, double2, double3, double4, mdouble2, mdouble3, mdouble4</code></td>
</tr>
<tr>
<td>Generic type name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>charn</td>
<td>char{n}, mchar{n}, marray{N}, char</td>
</tr>
<tr>
<td>scharn</td>
<td>schar{n}, mschar{n}, marray{N}, signed char</td>
</tr>
<tr>
<td>ucharn</td>
<td>uchar{n}, muchar{n}, marray{N}, unsigned char</td>
</tr>
<tr>
<td>igenchar</td>
<td>signed char, scharn</td>
</tr>
<tr>
<td>ugenchar</td>
<td>unsigned char, ucharn</td>
</tr>
<tr>
<td>genchar</td>
<td>char, charn, igenchar, ugenchar</td>
</tr>
<tr>
<td>shortn</td>
<td>short{n}, mshort{n}, marray{N}, short</td>
</tr>
<tr>
<td>genshort</td>
<td>short, shortn</td>
</tr>
<tr>
<td>ushortn</td>
<td>ushort{n}, mushort{n}, marray{N}, unsigned short</td>
</tr>
<tr>
<td>ugenshort</td>
<td>unsigned short, ushortn</td>
</tr>
<tr>
<td>uintn</td>
<td>uint{n}, muint{n}, marray{N}, unsigned int</td>
</tr>
<tr>
<td>ugenint</td>
<td>unsigned int, uintn</td>
</tr>
<tr>
<td>intn</td>
<td>int{n}, mint{n}, marray{N}, int</td>
</tr>
<tr>
<td>genint</td>
<td>int, intn</td>
</tr>
<tr>
<td>ulongn</td>
<td>ulong{n}, mulong{n}, marray{N}, unsigned long int</td>
</tr>
<tr>
<td>ugenlong</td>
<td>unsigned long int, ulongn</td>
</tr>
<tr>
<td>longn</td>
<td>long{n}, mlong{n}, marray{N}, long int</td>
</tr>
<tr>
<td>genlong</td>
<td>long int, longn</td>
</tr>
<tr>
<td>Generic type name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ulonglongn</td>
<td>ulonglong(n), mulonglong(n), marray(&lt;N), unsigned long long int&gt;</td>
</tr>
<tr>
<td>ugenlonglong</td>
<td>unsigned long long int, ulonglongn</td>
</tr>
<tr>
<td>longlongn</td>
<td>longlong(n), mlonglong(n), marray(&lt;N), long long int&gt;</td>
</tr>
<tr>
<td>genlonglong</td>
<td>long long int, longlongn</td>
</tr>
<tr>
<td>igenlonginteger</td>
<td>genlong, genlonglong</td>
</tr>
<tr>
<td>ugenlonginteger</td>
<td>ugenlong, ugenlonglong</td>
</tr>
<tr>
<td>geninteger</td>
<td>genchar, genshort, ugenshort, genint, ugenint, igenlonginteger, ugenlonginteger</td>
</tr>
<tr>
<td>genintegerNbit</td>
<td>All types within geninteger whose base type are (N) bits in size, where (N = 8, 16, 32, 64).</td>
</tr>
<tr>
<td>igeninteger</td>
<td>igenchar, genshort, genint, igenlonginteger</td>
</tr>
<tr>
<td>igenintegerNbit</td>
<td>All types within igeninteger whose base type are (N) bits in size, where (N = 8, 16, 32, 64).</td>
</tr>
<tr>
<td>sigeninteger</td>
<td>signed char, short, int, long int, long long int</td>
</tr>
<tr>
<td>vigeninteger</td>
<td>schar(n), short(n), int(n), long(n), longlong(n)</td>
</tr>
<tr>
<td>migeninteger</td>
<td>marray\langle signed char,(N\rangle, marray\langle short,(N\rangle, marray\langle int,(N\rangle, marray\langle long,(N\rangle, marray\langle long long,(N\rangle\rangle</td>
</tr>
<tr>
<td>ugeninteger</td>
<td>ugenchar, ugenshort, ugenint, ugenlonginteger</td>
</tr>
<tr>
<td>ugenintegerNbit</td>
<td>All types within ugeninteger whose base type are (N) bits in size, where (N = 8, 16, 32, 64).</td>
</tr>
<tr>
<td>sugeninteger</td>
<td>unsigned char, unsigned short, unsigned int, unsigned long int, unsigned long long int</td>
</tr>
<tr>
<td>vugeninteger</td>
<td>uchar(n), ushort(n), uint(n), ulong(n), ulonglong(n)</td>
</tr>
<tr>
<td>Generic type name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mugeninteger</td>
<td>marray&lt;unsigned char,{N}&gt;, marray&lt;unsigned short,{N}&gt;, marray&lt;unsigned int,{N}&gt;, marray&lt;unsigned long int,{N}&gt;, marray&lt;unsigned long long int,{N}&gt;</td>
</tr>
<tr>
<td>sgeninteger</td>
<td>char, sigeninteger, sugeninteger</td>
</tr>
<tr>
<td>vgeninteger</td>
<td>char(n), vigeninteger, vugeninteger</td>
</tr>
<tr>
<td>mgeninteger</td>
<td>marray&lt;char,{N}&gt;, migeninteger, mugeninteger</td>
</tr>
<tr>
<td>gentype</td>
<td>genfloat, geninteger</td>
</tr>
<tr>
<td>sgentype</td>
<td>sgenfloat, sgeninteger</td>
</tr>
<tr>
<td>vgentype</td>
<td>vgenfloat, vgeninteger</td>
</tr>
<tr>
<td>mgentype</td>
<td>mgenfloat, mgeninteger</td>
</tr>
<tr>
<td>genfloatptr</td>
<td>All permutations of <code>multi_ptr&lt;DataT, AddressSpace, IsDecorated&gt;</code> where DataT is all types within genfloat, AddressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space and IsDecorated is access::decorated::yes and access::decorated::no.</td>
</tr>
<tr>
<td>genintptr</td>
<td>All permutations of <code>multi_ptr&lt;DataT, AddressSpace, IsDecorated&gt;</code> where DataT is all types within genint, AddressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space and IsDecorated is access::decorated::yes and access::decorated::no.</td>
</tr>
<tr>
<td>booln</td>
<td>marray&lt;{N},bool&gt;</td>
</tr>
<tr>
<td>genbool</td>
<td>bool, booln</td>
</tr>
</tbody>
</table>
4.17.2. Function objects

SYCL provides a number of function objects in the `sycl` namespace on host and device. All function objects obey C++ conversion and promotion rules. Each function object is additionally specialized for `void` as a transparent function object that deduces its parameter types and return type.

```cpp
namespace sycl {
    template <typename T = void> struct plus {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct multiplies {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct bit_and {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct bit_or {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct bit_xor {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct logical_and {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct logical_or {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct minimum {
        T operator()(const T& x, const T& y) const;
    };

    template <typename T = void> struct maximum {
        T operator()(const T& x, const T& y) const;
    };
} // namespace sycl
```

<table>
<thead>
<tr>
<th>Table 167. Member functions for the <code>plus</code> function object</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member function</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the sum of its arguments, equivalent to <code>x + y</code>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 168. Member functions for the <code>multiplies</code> function object</th>
<th></th>
</tr>
</thead>
</table>
Table 169. Member functions for the bit_and function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the bitwise AND of its arguments, equivalent to <code>x &amp; y</code>.</td>
</tr>
</tbody>
</table>

Table 170. Member functions for the bit_or function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the bitwise OR of its arguments, equivalent to `x</td>
</tr>
</tbody>
</table>

Table 171. Member functions for the bit_xor function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the bitwise XOR of its arguments, equivalent to <code>x ^ y</code>.</td>
</tr>
</tbody>
</table>

Table 172. Member functions for the logical_and function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the logical AND of its arguments, equivalent to <code>x &amp;&amp; y</code>.</td>
</tr>
</tbody>
</table>

Table 173. Member functions for the logical_or function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the logical OR of its arguments, equivalent to `x</td>
</tr>
</tbody>
</table>

Table 174. Member functions for the minimum function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the smaller value. Returns the first argument when the arguments are equivalent.</td>
</tr>
</tbody>
</table>

Table 175. Member functions for the maximum function object

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>T operator()(const T&amp; x, const T&amp; y) const</code></td>
<td>Returns the larger value. Returns the first argument when the arguments are equivalent.</td>
</tr>
</tbody>
</table>

4.17.3. Group functions

SYCL provides a number of functions that expose functionality tied to groups of work-items (such as group barriers and collective operations). These group functions act as synchronization points and must be encountered in converged control flow by all work-items in the group. If one work-item in a group
calls a group function, then all work-items in that group must call exactly the same function under the
same set of conditions — calling the same function under different conditions (e.g. in different iterations
of a loop, or different branches of a conditional statement) results in undefined behavior. Additionally,
restrictions may be placed on the arguments passed to each function in order to ensure that all work-
items in the group agree on the operation that is being performed. Any such restrictions on the argu-
ments passed to a function are defined within the descriptions of those functions. Violating these restric-
tions results in undefined behavior.

All group functions are supported for the fundamental scalar types supported by SYCL (see Table 184)
and instances of the SYCL vec and marray classes.

Using a group function inside of a kernel may introduce additional limits on the resources available to
user code inside the same kernel. The behavior of these limits is implementation-defined, but must be
reflected by calls to kernel querying functions (such as kernel::get_info) as described in Section
4.11.13.1.

It is undefined behavior for any group function to be invoked within a parallel_for_work_group or paral-
lel_for_work_item context.

### 4.17.3.1. Group type trait

```cpp
namespace sycl {
    template <class T> struct is_group;
    template <class T> inline constexpr bool is_group_v = is_group<T>::value;
} // namespace sycl
```

The is_group type trait is used to determine which types of groups are supported by group functions, and
to control when group functions participate in overload resolution.

is_group<T> inherits from std::true_type if T is the type of a standard SYCL group (group or sub_group)
and it inherits from std::false_type otherwise. A SYCL implementation may introduce additional spe-
cializations of is_group<T> for implementation-defined group types, if the interface of those types sup-
ports all member functions and static members common to the group and sub_group classes.

### 4.17.3.2. group_broadcast

The group_broadcast function communicates a value held by one work-item to all other work-items in the
group.

```cpp
template <typename Group, typename T> T group_broadcast(Group g, T x); // (1)
template <typename Group, typename T>
T group_broadcast(Group g, T x, Group::linear_id_type local_linear_id); // (2)
template <typename Group, typename T>
T group_broadcast(Group g, T x, Group::id_type local_id); // (3)
```

1. **Constraints:** Available only if sycl::is_group_v<std::decay_t<Group>> is true and T is a trivially copy-
able type.

   Returns: The value of x from the work-item with the smallest linear id within group g.

2. **Constraints:** Available only if sycl::is_group_v<std::decay_t<Group>> is true and T is a trivially copy-
able type.
Preconditions: local_linear_id must be the same for all work-items in the group and must be in the range \([0, \text{get_local_linear_range()}]\).

Returns: The value of \(x\) from the work-item with the specified linear id within group \(g\).

3. Constraints: Available only if \(\text{sycl::is_group_v<std::decay_t<Group>>}\) is true and \(T\) is a trivially copyable type.

Preconditions: local_id must be the same for all work-items in the group, and its dimensionality must match the dimensionality of the group. The value of local_id in each dimension must be greater than or equal to 0 and less than the value of get_local_range() in the same dimension.

Returns: The value of \(x\) from the work-item with the specified id within group \(g\).

4.17.3.3. group_barrier

The group_barrier function synchronizes all work-items in a group, using a group barrier.

```cpp
1 template <typename Group>
2 void group_barrier(Group g, 
3      memory_scope fence_scope = Group::fence_scope);  // (1)
```

1. Constraints: Available only if \(\text{sycl::is_group_v<\text{std::decay_t<Group>}>}\) is true.

Effects: Synchronizes all work-items in group \(g\). The current work-item will wait at the barrier until all work-items in group \(g\) have reached the barrier. In addition, the barrier performs mem-fence operations ensuring that memory accesses issued before the barrier are not re-ordered with those issued after the barrier: all work-items in group \(g\) execute a release fence prior to synchronizing at the barrier, all work-items in group \(g\) execute an acquire fence afterwards, and there is an implicit synchronization of these fences as if provided by an explicit atomic operation on an atomic object.

By default, the scope of these fences is set to the narrowest scope including all work-items in group \(g\) (as reported by Group::fence_scope). This scope may be optionally overridden with a broader scope, specified by the fence_scope argument.

4.17.4. Group algorithms library

SYCL provides an algorithms library based on the functions described in Section 28 of the C++17 specification. The first argument to each function is a group, and data ranges can be described using pointers, iterators or instances of the multi_ptr class. The functions defined in this section are free functions available in the sycl namespace.

Any restrictions from the standard algorithms library apply. Some of the functions in the SYCL algorithms library introduce additional restrictions in order to maximize portability across different devices and to minimize the chances of encountering unexpected behavior.

All algorithms are supported for the fundamental scalar types supported by SYCL (see Table 184) and instances of the SYCL vec and marray classes.

The group argument to a SYCL algorithm denotes that it should be performed collaboratively by the work-items in the specified group. All algorithms act as group functions (as defined in Section 4.17.3), inheriting all restrictions of group functions. Unless the description of a function says otherwise, how the elements of a range are processed by the work-items in a group is undefined.

SYCL provides separate functions for algorithms which use the work-items in a group to execute an operation over a range of iterators and algorithms which are applied to data held directly by the work-items in a group. An example of the usage of these functions is given below:
Listing 2. Using the group algorithms library to perform a work-group reduce

```cpp
1 buffer<int> inputBuf { 1024 };  
2 buffer<int> outputBuf { 2 };  
3 {  
4   // Initialize buffer on the host with 0, 1, 2, 3, ..., 1023  
5   host_accessor a { inputBuf };  
6   std::iota(a.begin(), a.end(), 0);  
7 }  
8 myQueue.submit([&](handler &cgh)  
9 {  
10   accessor inputValues { inputBuf, cgh, read_only };  
11   accessor outputValues { outputBuf, cgh, write_only, no_init };  
12   cgh.parallel_for(nd_range<1>(range<1>(16), range<1>(16)), [=](nd_item<1> it) {  
13     int* first = inputValues.get_pointer();  
14     int* last = first + 1024;  
15     int sum = joint_reduce(it.get_group(), first, last, plus<>());  
16     outputValues[0] = sum;  
17     // Apply a group algorithm to a set of values held directly by work-items.  
18     // The work-group reduces a number of values equal to the size of the group  
19     // and each work-item provides one value.  
20     int partial_sum = reduce_over_group(  
21       it.get_group(), inputValues[it.get_global_linear_id()], plus<>());  
22     outputValues[1] = partial_sum;  
23   });  
24   host_accessor a { outputBuf };  
25   assert(a[0] == 523776 && a[1] == 120);  
26 });
```

4.17.4.1. any_of, all_of and none_of

The `any_of`, `all_of` and `none_of` functions from standard C++ test whether Boolean conditions hold for any of, all of or none of the values in a range, respectively.

SYCL provides two sets of similar algorithms:

1. `joint_any_of`, `joint_all_of` and `joint_none_of` use the work-items in a group to execute the corresponding algorithm in parallel.

2. `any_of_group`, `all_of_group` and `none_of_group` test Boolean conditions applied to data held directly by the work-items in a group.

```cpp
1 template <typename Group, typename Ptr, typename Predicate>  
2 bool joint_any_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)  
3  
4 template <typename Group, typename T, typename Predicate>  
5 bool any_of_group(Group g, T x, Predicate pred); // (2)  
6  
7 template <typename Group> bool any_of_group(Group g, bool pred); // (3)
```
1. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `Ptr` is a pointer.

   **Preconditions:** `first` and `last` must be the same for all work-items in group `g`, and `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

   **Returns:** true if `pred` returns true when applied to the result of dereferencing any iterator in the range `[first, last)`.

2. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

   **Preconditions:** `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

   **Returns:** true if `pred(x)` returns true for any work-item in group `g`.

3. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

   **Returns:** true if `pred` is true for any work-item in group `g`.

---

```cpp
1 template <typename Group, typename Ptr, typename Predicate>
2 bool joint_all_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)
3
4 template <typename Group, typename T, typename Predicate>
5 bool all_of_group(Group g, T x, Predicate pred); // (2)
6
7 template <typename Group> bool all_of_group(Group g, bool pred); // (3)
```

---

1. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true and `Ptr` is a pointer.

   **Preconditions:** `first` and `last` must be the same for all work-items in group `g`, and `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

   **Returns:** true if `pred` returns true when applied to the result of dereferencing all iterators in the range `[first, last)`. 

2. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

   **Preconditions:** `pred` must be an immutable callable with the same type and state for all work-items in group `g`.

   **Returns:** true if `pred(x)` returns true for all work-items in group `g`.

3. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true.

   **Returns:** true if `pred` is true for all work-items in group `g`.

```cpp
1 template <typename Group, typename Ptr, typename Predicate>
2 bool joint_none_of(Group g, Ptr first, Ptr last, Predicate pred); // (1)
3
4 template <typename Group, typename T, typename Predicate>
5 bool none_of_group(Group g, T x, Predicate pred); // (2)
6
7 template <typename Group> bool none_of_group(Group g, bool pred); // (3)
```
4.17.4.2. shift_left and shift_right

The `shift_left` and `shift_right` functions from standard C++ move values in a range down (to the left) or up (to the right) respectively.

SYCL provides similar algorithms compatible with the `sub_group` class:

1. `shift_group_left` and `shift_group_right` move values held by the work-items in a group directly to another work-item in group `g`, by shifting values a fixed number of work-items to the left or right.

   ```cpp
   1 template<typename Group, typename T>
   2 T shift_group_left(Group g, T x, Group::linear_id_type delta = 1); // (1)
   3
   4 template<typename Group, typename T>
   5 T shift_group_right(Group g, T x, Group::linear_id_type delta = 1); // (2)
   ```

1. **Constraints**: Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

   **Preconditions**: `delta` must be the same for all work-items in the group.

   **Returns**: the value of `x` from the work-item whose group local id (`id`) is `delta` larger than that of the calling work-item. `id + delta` may be greater than or equal to the group's linear size, but the value returned in this case is unspecified.

2. **Constraints**: Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

   **Preconditions**: `delta` must be the same for all work-items in the group.

   **Returns**: the value of `x` from the work-item whose group local id (`id`) is `delta` smaller than that of the calling work-item. `id - delta` may be less than 0, but the value returned in this case is unspecified.

4.17.4.3. permute

SYCL provides an algorithm to permute the values held by work-items in a sub-group:

1. `permute_group_by_xor` permutes values by exchanging values held by pairs of work-items identified by computing the bitwise exclusive OR of the work-item id and some fixed mask.

   ```cpp
   1 template<typename Group, typename T>
   2 T permute_group_by_xor(Group g, T x, Group::linear_id_type mask); // (1)
   ```
1. **Constraints:** Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

    **Preconditions:** mask must be the same for all work-items in the group.

    **Returns:** the value of `x` from the work-item whose group local id is equal to the bitwise exclusive OR of the calling work-item's group local id and `mask`. The result of the exclusive OR may be greater than or equal to the group's linear size, but the value returned in this case is unspecified.

### 4.17.4.4. **select**

SYCL provides an algorithm to directly exchange the values held by work-items in a sub-group:

1. **select_from_group** allows work-items to obtain a copy of a value held by any other work-item in group `g`.

   ```
   template<typename Group, typename T>
   T select_from_group(Group g, T x, Group::id_type remote_local_id); // (1)
   ```

   **Constraints:** Available only if `std::is_same_v<std::decay_t<Group>, sub_group>` is true and `T` is a trivially copyable type.

   **Returns:** the value of `x` from the work-item with the group local id specified by `remote_local_id`. The value of `remote_local_id` may be outside of the group, but the value returned in this case is unspecified.

### 4.17.4.5. **reduce**

The **reduce** function from standard C++ combines the values in a range in an unspecified order using a binary operator.

SYCL provides two similar algorithms that compute the same generalized sum as defined by standard C++:

1. **joint_reduce** uses the work-items in a group to execute a **reduce** operation in parallel.
2. **reduce_over_group** combines values held directly by the work-items in a group.

The result of a call to these functions is non-deterministic if the binary operator is not commutative and associative. Only the binary operators defined in **Section 4.17.2** are supported by the **reduce** functions in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

```
1 template <typename Group, typename Ptr, typename BinaryOperation>
2 std::iterator_traits<Ptr>::value_type
3 joint_reduce(Group g, Ptr first, Ptr last, BinaryOperation binary_op); // (1)
4
5 template <typename Group, typename Ptr, typename T, typename BinaryOperation>
6 T joint_reduce(Group g, Ptr first, Ptr last, T init,
7       BinaryOperation binary_op); // (2)
8
9 template <typename Group, typename T, typename BinaryOperation>
10 T reduce_over_group(Group g, T x, BinaryOperation binary_op); // (3)
11
12 template <typename Group, typename V, typename T, typename BinaryOperation>
13 T reduce_over_group(Group g, V x, T init, BinaryOperation binary_op); // (4)
```
damental type, and **BinaryOperation** is a SYCL function object type.

**Mandates:** \( \text{binary_op(*first, *first)} \) must return a value of type `std::iterator_traits<Ptr>::value_type`.

**Preconditions:** `first, last` and the type of `binary_op` must be the same for all work-items in group \( g \). `binary_op` must be an instance of a SYCL function object.

**Returns:** The result of combining the values resulting from dereferencing all iterators in the range \([first, last)\) using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

2. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `Ptr` is a pointer to a fundamental type, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

**Mandates:** `binary_op(init, *first)` must return a value of type `T`.

**Preconditions:** `first, last, init` and the type of `binary_op` must be the same for all work-items in group \( g \). `binary_op` must be an instance of a SYCL function object.

**Returns:** The result of combining the values resulting from dereferencing all iterators in the range \([first, last)\) and the initial value `init` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

3. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `T` is a fundamental type and `BinaryOperation` is a SYCL function object type.

**Mandates:** `binary_op(x, x)` must return a value of type `T`.

**Preconditions:** `binary_op` must be an instance of a SYCL function object.

**Returns:** The result of combining all the values of `x` specified by each work-item in group \( g \) using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

4. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `V` and `T` are fundamental types, and `BinaryOperation` is a SYCL function object type.

**Mandates:** `binary_op(init, x)` must return a value of type `T`.

**Preconditions:** `binary_op` must be an instance of a SYCL function object.

**Returns:** The result of combining all the values of `x` specified by each work-item in group \( g \) and the initial value `init` using the operator `binary_op`, where the values are combined according to the generalized sum defined in standard C++.

### 4.17.4.6. exclusive_scan and inclusive_scan

The **exclusive_scan** and **inclusive_scan** functions in standard C++ compute a prefix sum using a binary operator. For a scan of elements \( [x_0, \ldots, x_n] \), the \( i \)th result in an exclusive scan is the generalized noncommutative sum of all elements preceding \( x_i \) (excluding \( x_i \) itself), whereas the \( i \)th result in an inclusive scan is the generalized noncommutative sum of all elements preceding \( x_i \) (including \( x_i \) itself).

SYCL provides two similar sets of algorithms that compute the same prefix sums using the generalized noncommutative sum as defined by standard C++:

1. **joint_exclusive_scan** and **joint_inclusive_scan** use the work-items in a group to execute the corresponding algorithm in parallel, and intermediate partial prefix sums are written to memory as in standard C++.

2. **exclusive_scan_over_group** and **inclusive_scan_over_group** perform a scan over values held directly by
the work-items in a group, and the result returned to each work-item represents a partial prefix sum.

The result of a call to a scan is non-deterministic if the binary operator is not associative. Only the binary operators defined in Section 4.17.2 are supported by the scan functions in SYCL 2020, but the standard C++ syntax is used for forward compatibility with future SYCL versions.

```cpp
1 template <typename Group, typename InPtr, typename OutPtr,
2          typename BinaryOperation>
3 OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
4                              BinaryOperation binary_op); // (1)
5
6 template <typename Group, typename InPtr, typename OutPtr, typename T,
7          typename BinaryOperation>
8 OutPtr joint_exclusive_scan(Group g, InPtr first, InPtr last, OutPtr result,
9                              T init, BinaryOperation binary_op); // (2)
10
11 template <typename Group, typename T, typename BinaryOperation>
12 T exclusive_scan_over_group(Group g, T x, BinaryOperation binary_op); // (3)
13
14 template <typename Group, typename V, typename T, typename BinaryOperation>
15 T exclusive_scan_over_group(Group g, V x, T init,
16                              BinaryOperation binary_op); // (4)
```

1. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, and `BinaryOperation` is a SYCL function object type.

   **Mandates:** `binary_op(*first, *first)` must return a value of type `std::iterator_traits<OutPtr>::value_type`.

   **Preconditions:** `first, last, result` and the type of `binary_op` must be the same for all work-items in group `g`. `binary_op` must be an instance of a SYCL function object.

   Note that `first` may be equal to `result`.

   **Effects:** The value written to `result + i` is the exclusive scan of the values resulting from dereferencing the first `i` values in the range `[first, last)` and the identity value of `binary_op` (as identified by `sycl::known_identity`), using the operator `binary_op`. The scan is computed using a generalized non-commutative sum as defined in standard C++.

   **Returns:** A pointer to the end of the output range.

2. **Constraints:** Available only if `sycl::is_group_v<std::decay_t<Group>>` is true, `InPtr` and `OutPtr` are pointers to fundamental types, `T` is a fundamental type, and `BinaryOperation` is a SYCL function object type.

   **Mandates:** `binary_op(init, *first)` must return a value of type `T`.

   **Preconditions:** `first, last, result, init` and the type of `binary_op` must be the same for all work-items in group `g`. `binary_op` must be an instance of a SYCL function object.

   Note that `first` may be equal to `result`.

   **Effects:** The value written to `result + i` is the exclusive scan of the values resulting from dereferencing the first `i` values in the range `[first, last)` and an initial value specified by `init`, using the operator `binary_op`. The scan is computed using a generalized noncommutative sum as defined in standard C++.
Returns: A pointer to the end of the output range.

3. Constraints: Available only if sycl::is_group_v<std::decay_t<Group>> is true, T is a fundamental type, and BinaryOperation is a SYCL function object type.

Mandates: binary_op(x, x) must return a value of type T.

Preconditions: binary_op must be an instance of a SYCL function object.

Returns: The value returned on work-item i is the exclusive scan of the first i values in group g and the identity value of binary_op (as identified by sycl::known_identity), using the operator binary_op. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group g is determined by their linear id.

4. Constraints: Available only if sycl::is_group_v<std::decay_t<Group>> is true, V and T are fundamental types, and BinaryOperation is a SYCL function object type.

Mandates: binary_op(init, x) must return a value of type T.

Preconditions: binary_op must be an instance of a SYCL function object.

Returns: The value returned on work-item i is the exclusive scan of the first i values in group g and an initial value specified by init, using the operator binary_op. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group g is determined by their linear id.
Returns: A pointer to the end of the output range.

2. Constraints: Available only if sycl::is_group_v<std::decay_t<Group>> is true, InPtr and OutPtr are pointers to fundamental types, BinaryOperation is a SYCL function object type, and T is a fundamental type.

Mandates: binary_op(init, *first) must return a value of type T.

Preconditions: first, last, result, init and the type of binary_op must be the same for all work-items in group g. BinaryOperation must be an instance of a SYCL function object.

Effects: The value written to result + i is the inclusive scan of the values resulting from dereferencing the first i values in the range [first, last) and an initial value specified by init, using the operator binary_op. The scan is computed using a generalized noncommutative sum as defined in standard C++.

Returns: A pointer to the end of the output range.

3. Constraints: Available only if sycl::is_group_v<std::decay_t<Group>> is true, T is a fundamental type, and BinaryOperation is a SYCL function object type.

Mandates: binary_op(x, x) must return a value of type T.

Preconditions: binary_op must be an instance of a SYCL function object.

Returns: The value returned on work-item i is the inclusive scan of the first i values in group g, using the operator binary_op. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group g is determined by their linear id.

4. Constraints: Available only if sycl::is_group_v<std::decay_t<Group>> is true, V is a fundamental type, BinaryOperation is a SYCL function object type, and T is a fundamental type.

Mandates: binary_op(init, x) must return a value of type T.

Preconditions: binary_op must be an instance of a SYCL function object.

Returns: The value returned on work-item i is the inclusive scan of the first i values in group g and an initial value specified by init, using the operator binary_op. The scan is computed using a generalized noncommutative sum as defined in standard C++. For multi-dimensional groups, the order of work-items in group g is determined by their linear id.

4.17.5. Math functions

In SYCL the OpenCL math functions are available in the namespace sycl on host and device with the same precision guarantees as defined in the OpenCL 1.2 specification document ch. 7 for host and device. For a SYCL platform the numerical requirements for host need to match the numerical requirements of the OpenCL math built-in functions. The built-in functions can take as input float or optionally double and their vec and marray counterparts, for all supported dimensions including dimension 1.

The built-in functions available for SYCL host and device, with the same precision requirements for both host and device, are described in Table 176.

Table 176. Math functions which work on SYCL host and device. They correspond to Table 6.8 of the OpenCL 1.2 specification.
<table>
<thead>
<tr>
<th>Math Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genfloat acos(genfloat x)</code></td>
<td>Inverse cosine function.</td>
</tr>
<tr>
<td><code>genfloat acosh(genfloat x)</code></td>
<td>Inverse hyperbolic cosine.</td>
</tr>
<tr>
<td><code>genfloat acospi(genfloat x)</code></td>
<td>Compute $\frac{\text{arccos}(x)}{\pi}$.</td>
</tr>
<tr>
<td><code>genfloat asin(genfloat x)</code></td>
<td>Inverse sine function.</td>
</tr>
<tr>
<td><code>genfloat asinh(genfloat x)</code></td>
<td>Inverse hyperbolic sine.</td>
</tr>
<tr>
<td><code>genfloat asinpi(genfloat x)</code></td>
<td>Compute $\frac{\text{arcsin}(x)}{\pi}$.</td>
</tr>
<tr>
<td><code>genfloat atan(genfloat y_over_x)</code></td>
<td>Inverse tangent function.</td>
</tr>
<tr>
<td><code>genfloat atan2(genfloat y, genfloat x)</code></td>
<td>Compute $\arctan(\xi)$.</td>
</tr>
<tr>
<td><code>genfloat atanh(genfloat x)</code></td>
<td>Hyperbolic inverse tangent.</td>
</tr>
<tr>
<td><code>genfloat atanpi(genfloat x)</code></td>
<td>Compute $\frac{(y, x)}{\pi}$.</td>
</tr>
<tr>
<td><code>genfloat atan2pi(genfloat y, genfloat x)</code></td>
<td>Compute $\frac{(y, x)}{\pi}$.</td>
</tr>
<tr>
<td><code>genfloat cbrt(genfloat x)</code></td>
<td>Compute cube-root.</td>
</tr>
<tr>
<td><code>genfloat ceil(genfloat x)</code></td>
<td>Round to integral value using the</td>
</tr>
<tr>
<td></td>
<td>round to positive infinity rounding</td>
</tr>
<tr>
<td></td>
<td>mode.</td>
</tr>
<tr>
<td><code>genfloat copiesign(genfloat x, genfloat y)</code></td>
<td>Returns $x$ with its sign changed to</td>
</tr>
<tr>
<td></td>
<td>match the sign of $y$.</td>
</tr>
<tr>
<td><code>genfloat cos(genfloat x)</code></td>
<td>Compute cosine.</td>
</tr>
<tr>
<td><code>genfloat cosh(genfloat x)</code></td>
<td>Compute hyperbolic cosine.</td>
</tr>
<tr>
<td><code>genfloat cospi(genfloat x)</code></td>
<td>Compute $\cos(\pi x)$.</td>
</tr>
<tr>
<td><code>genfloat erfc(genfloat x)</code></td>
<td>Complementary error function.</td>
</tr>
<tr>
<td>Math Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>genfloat erf(genfloat x)</td>
<td>Error function encountered in integrating the normal distribution.</td>
</tr>
<tr>
<td>genfloat exp(genfloat x)</td>
<td>Compute the base-e exponential of x.</td>
</tr>
<tr>
<td>genfloat exp2(genfloat x)</td>
<td>Exponential base 2 function.</td>
</tr>
<tr>
<td>genfloat exp10(genfloat x)</td>
<td>Exponential base 10 function.</td>
</tr>
<tr>
<td>genfloat expm1(genfloat x)</td>
<td>Compute $e^x - 1.0$.</td>
</tr>
<tr>
<td>genfloat fabs(genfloat x)</td>
<td>Compute absolute value of a floating-point number.</td>
</tr>
<tr>
<td>genfloat fdim(genfloat x, genfloat y)</td>
<td>$x - y$ if $x&gt;y$, $+0$ if $x$ is less than or equal to $y$.</td>
</tr>
<tr>
<td>genfloat floor(genfloat x)</td>
<td>Round to integral value using the round to negative infinity rounding mode.</td>
</tr>
<tr>
<td>genfloat fma(genfloat a, genfloat b, genfloat c)</td>
<td>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.</td>
</tr>
<tr>
<td>genfloat fmax(genfloat x, genfloat y)</td>
<td>Returns y if $x&lt;y$, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.</td>
</tr>
<tr>
<td>genfloat fmax(genfloat x, sgenfloat y)</td>
<td>Returns y if $x&gt;y$, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.</td>
</tr>
<tr>
<td>genfloat fmin(genfloat x, genfloat y)</td>
<td>Returns y if $y&lt;x$, otherwise it returns x. If one argument is a NaN, fmin() returns the other argument. If both arguments are NaNs, fmin() returns a NaN.</td>
</tr>
<tr>
<td>genfloat fmod(genfloat x, genfloat y)</td>
<td>Modulus. Returns $x - y \cdot \text{trunc}(x/y)$.</td>
</tr>
<tr>
<td>genfloat fract(genfloat x, genfloatptr iptr)</td>
<td>Returns fmin(x - floor(x), nextafter(genfloat(1.0), genfloat(0.0)) ). floor(x) is returned in iptr.</td>
</tr>
</tbody>
</table>
### Math Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genfloat frexp(genfloat x, genintptr exp)</code></td>
<td>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 (\times 2^{exp}).</td>
</tr>
<tr>
<td><code>genfloat hypot(genfloat x, genfloat y)</code></td>
<td>Compute the square root of (x^2 + y^2) without undue overflow or underflow.</td>
</tr>
<tr>
<td><code>genint ilogb(genfloat x)</code></td>
<td>Compute the integral part of (\log_r(x)) and return the result as an integer, where r is the value returned by <code>std::numeric_limits&lt;genfloat&gt;::radix</code>.</td>
</tr>
<tr>
<td><code>genfloat ldexp(genfloat x, genint k)</code></td>
<td>Multiply x by (2^k).</td>
</tr>
<tr>
<td><code>genfloat lgamma(genfloat x)</code></td>
<td>Log gamma function. Returns the natural logarithm of the absolute value of the gamma function.</td>
</tr>
<tr>
<td><code>genfloat lgamma_r(genfloat x, genintptr signp)</code></td>
<td>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 <code>lgamma_r</code>.</td>
</tr>
<tr>
<td><code>genfloat log(genfloat x)</code></td>
<td>Compute natural logarithm.</td>
</tr>
<tr>
<td><code>genfloat log2(genfloat x)</code></td>
<td>Compute a base 2 logarithm.</td>
</tr>
<tr>
<td><code>genfloat log10(genfloat x)</code></td>
<td>Compute a base 10 logarithm.</td>
</tr>
<tr>
<td><code>genfloat log1p(genfloat x)</code></td>
<td>Compute (\log_e(1.0 + x)).</td>
</tr>
<tr>
<td><code>genfloat logb(genfloat x)</code></td>
<td>Compute the integral part of (\log_r(x)), where r is the value returned by <code>std::numeric_limits&lt;genfloat&gt;::radix</code>.</td>
</tr>
<tr>
<td><code>genfloat mad(genfloat a, genfloat b, genfloat c)</code></td>
<td><code>mad</code> approximates (a \times b + c). Whether or how the product of (a \times b) is rounded and how supernormal or subnormal intermediate products are handled is not defined. <code>mad</code> is intended to be used where speed is preferred over accuracy.</td>
</tr>
<tr>
<td>Math Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>genfloat maxmag(genfloat x, genfloat y)</td>
<td>Returns x if</td>
</tr>
<tr>
<td>genfloat minmag(genfloat x, genfloat y)</td>
<td>Returns x if</td>
</tr>
<tr>
<td>genfloat modf(genfloat x, genfloatptr iptr)</td>
<td>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.</td>
</tr>
<tr>
<td>genfloatf nan(ugenint nancode)</td>
<td>genfloatd nan(ugenlonginteger nancode)</td>
</tr>
<tr>
<td>genfloat nextafter(genfloat x, genfloat y)</td>
<td>Computes the next representable single-precision 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.</td>
</tr>
<tr>
<td>genfloat pow(genfloat x, genfloat y)</td>
<td>Compute x to the power y.</td>
</tr>
<tr>
<td>genfloat pown(genfloat x, genint y)</td>
<td>Compute x to the power y, where y is an integer.</td>
</tr>
<tr>
<td>genfloat powr(genfloat x, genfloat y)</td>
<td>Compute x to the power y, where x ≥ 0.</td>
</tr>
<tr>
<td>genfloat remainder(genfloat x, genfloat y)</td>
<td>Compute the value r such that r = x - n*</td>
</tr>
<tr>
<td>genfloat remquo(genfloat x, genfloat y, genintptr quo)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Math Function

<table>
<thead>
<tr>
<th>Math Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genfloat rint(genfloat x)</td>
<td>Round to integral value (using round to nearest even rounding mode) in floating-point format. Refer to section 7.1 of the OpenCL 1.2 specification document for description of rounding modes.</td>
</tr>
<tr>
<td>genfloat rootn(genfloat x, genint y)</td>
<td>Compute x to the power 1/y.</td>
</tr>
<tr>
<td>genfloat round(genfloat x)</td>
<td>Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction.</td>
</tr>
<tr>
<td>genfloat rsqrt(genfloat x)</td>
<td>Compute inverse square root.</td>
</tr>
<tr>
<td>genfloat sin(genfloat x)</td>
<td>Compute sine.</td>
</tr>
<tr>
<td>genfloat sincos(genfloat x, genfloatptr cosval)</td>
<td>Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in <code>cosval</code>.</td>
</tr>
<tr>
<td>genfloat sinh(genfloat x)</td>
<td>Compute hyperbolic sine.</td>
</tr>
<tr>
<td>genfloat sinpi(genfloat x)</td>
<td>Compute $\sin(\pi x)$.</td>
</tr>
<tr>
<td>genfloat sqrt(genfloat x)</td>
<td>Compute square root.</td>
</tr>
<tr>
<td>genfloat tan(genfloat x)</td>
<td>Compute tangent.</td>
</tr>
<tr>
<td>genfloat tanh(genfloat x)</td>
<td>Compute hyperbolic tangent.</td>
</tr>
<tr>
<td>genfloat tanpi(genfloat x)</td>
<td>Compute $\tan(\pi x)$.</td>
</tr>
<tr>
<td>genfloat tgamma(genfloat x)</td>
<td>Compute the gamma function.</td>
</tr>
<tr>
<td>genfloat trunc(genfloat x)</td>
<td>Round to integral value using the round to zero rounding mode.</td>
</tr>
</tbody>
</table>

In SYCL the implementation-defined precision math functions are defined in the namespace `sycl::native`. The functions that are available within this namespace are specified in Table 177.

**Table 177. Native math functions**
### Native Math Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genfloatf <code>cos(genfloatf x)</code></td>
<td>Compute cosine over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>divide(genfloatf x, genfloatf y)</code></td>
<td>Compute x / y over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>exp(genfloatf x)</code></td>
<td>Compute the base-e exponential of x over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>exp2(genfloatf x)</code></td>
<td>Compute the base-2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>exp10(genfloatf x)</code></td>
<td>Compute the base-10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>log(genfloatf x)</code></td>
<td>Compute natural logarithm over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>log2(genfloatf x)</code></td>
<td>Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>log10(genfloatf x)</code></td>
<td>Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>powr(genfloatf x, genfloatf y)</code></td>
<td>Compute x to the power y, where x ≥ 0. The range of x and y are implementation-defined. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>recip(genfloatf x)</code></td>
<td>Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>rsqrt(genfloatf x)</code></td>
<td>Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
<tr>
<td>genfloatf <code>sin(genfloatf x)</code></td>
<td>Compute sine over an implementation-defined range. The maximum error is implementation-defined.</td>
</tr>
</tbody>
</table>
Native Math Function | Description
--- | ---
`genfloatf sqrt(genfloatf x)` | Compute square root over an implementation-defined range. The maximum error is implementation-defined.

`genfloatf tan(genfloatf x)` | Compute tangent over an implementation-defined range. The maximum error is implementation-defined.

In SYCL the half precision math functions are defined in `sycl::half_precision`. The functions that are available within this namespace are specified in Table 178. These functions are implemented with a minimum of 10-bits of accuracy i.e. the maximum error is less than or equal to 8192 ulp.

**Table 178. Half precision math functions**

<table>
<thead>
<tr>
<th>Half Math function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genfloatf cos(genfloatf x)</code></td>
<td>Compute cosine. x must be in the range -216 to +216.</td>
</tr>
<tr>
<td><code>genfloatf divide(genfloatf x, genfloatf y)</code></td>
<td>Compute x / y.</td>
</tr>
<tr>
<td><code>genfloatf exp(genfloatf x)</code></td>
<td>Compute the base- e exponential of x.</td>
</tr>
<tr>
<td><code>genfloatf exp2(genfloatf x)</code></td>
<td>Compute the base- 2 exponential of x.</td>
</tr>
<tr>
<td><code>genfloatf exp10(genfloatf x)</code></td>
<td>Compute the base- 10 exponential of x.</td>
</tr>
<tr>
<td><code>genfloatf log(genfloatf x)</code></td>
<td>Compute natural logarithm.</td>
</tr>
<tr>
<td><code>genfloatf log2(genfloatf x)</code></td>
<td>Compute a base 2 logarithm.</td>
</tr>
<tr>
<td><code>genfloatf log10(genfloatf x)</code></td>
<td>Compute a base 10 logarithm.</td>
</tr>
<tr>
<td><code>genfloatf powr(genfloatf x, genfloatf y)</code></td>
<td>Compute x to the power y, where x ≥ 0.</td>
</tr>
<tr>
<td><code>genfloatf recip(genfloatf x)</code></td>
<td>Compute reciprocal.</td>
</tr>
<tr>
<td><code>genfloatf rsqrt(genfloatf x)</code></td>
<td>Compute inverse square root.</td>
</tr>
<tr>
<td><code>genfloatf sin(genfloatf x)</code></td>
<td>Compute sine. x must be in the range -216 to +216.</td>
</tr>
<tr>
<td><code>genfloatf sqrt(genfloatf x)</code></td>
<td>Compute square root.</td>
</tr>
</tbody>
</table>
4.17.6. Integer functions

Integer math functions are available in SYCL in the namespace `sycl` on host and device. The built-in functions can take as input `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `long int`, `unsigned long long int` and their `vec` and `marray` counterparts. The supported integer math functions are described in Table 179.

Table 179. Integer functions which work on SYCL host and device, are available in the `sycl` namespace

<table>
<thead>
<tr>
<th>Integer Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>geninteger abs(geninteger x)</td>
<td>Returns</td>
</tr>
<tr>
<td>ugeninteger abs_diff(geninteger x, geninteger y)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger add_sat(geninteger x, geninteger y)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger hadd(geninteger x, geninteger y)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger rhadd(geninteger x, geninteger y)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger clamp(geninteger x, geninteger minval, geninteger maxval)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger clamp(geninteger x, sgeninteger minval, sgeninteger maxval)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger clz(geninteger x)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger ctz(geninteger x)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger mad_hi(geninteger a, geninteger b, geninteger c)</td>
<td>Returns</td>
</tr>
<tr>
<td>geninteger mad_sat(geninteger a, geninteger b, geninteger c)</td>
<td>Returns</td>
</tr>
<tr>
<td>Integer Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><code>geninteger max(geninteger x, geninteger y)</code></td>
<td>Returns y if x &lt; y, otherwise it returns x.</td>
</tr>
<tr>
<td><code>geninteger min(geninteger x, geninteger y)</code></td>
<td>Returns y if y &lt; x, otherwise it returns x.</td>
</tr>
<tr>
<td><code>geninteger mul_hi(geninteger x, geninteger y)</code></td>
<td>Computes x * y and returns the high half of the product of x and y.</td>
</tr>
<tr>
<td><code>geninteger rotate(geninteger v, geninteger i)</code></td>
<td>For each element in v, the bits are shifted left by the number of bits given by the corresponding element in i (subject to usual shift modulo rules described in the OpenCL 1.2 specification section 6.3). Bits shifted off the left side of the element are shifted back in from the right.</td>
</tr>
<tr>
<td><code>geninteger sub_sat(geninteger x, geninteger y)</code></td>
<td>Returns x - y and saturates the result.</td>
</tr>
<tr>
<td><code>ugeninteger16bit upsample(ugeninteger8bit hi, ugeninteger8bit lo)</code></td>
<td>result[i] = ((ushort)hi[i] &lt;&lt; 8)</td>
</tr>
<tr>
<td><code>igeninteger16bit upsample(igeninteger8bit hi, ugeninteger8bit lo)</code></td>
<td>result[i] = ((short)hi[i] &lt;&lt; 8)</td>
</tr>
<tr>
<td><code>ugeninteger32bit upsample(ugeninteger16bit hi, ugeninteger16bit lo)</code></td>
<td>result[i] = ((uint)hi[i] &lt;&lt; 16)</td>
</tr>
<tr>
<td><code>igeninteger32bit upsample(igeninteger16bit hi, ugeninteger16bit lo)</code></td>
<td>result[i] = ((int)hi[i] &lt;&lt; 16)</td>
</tr>
<tr>
<td><code>ugeninteger64bit upsample(ugeninteger32bit hi, ugeninteger32bit lo)</code></td>
<td>result[i] = ((ulonglong)hi[i] &lt;&lt; 32)</td>
</tr>
<tr>
<td><code>igeninteger64bit upsample(igeninteger32bit hi, ugeninteger32bit lo)</code></td>
<td>result[i] = ((longlong)hi[i] &lt;&lt; 32)</td>
</tr>
<tr>
<td><code>geninteger popcount(geninteger x)</code></td>
<td>Returns the number of non-zero bits in x.</td>
</tr>
</tbody>
</table>
### Integer Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>geninteger32bit mad24(geninteger32bit x, geninteger32bit y, geninteger32bit z)</code></td>
<td>Multiply 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 <code>mul24</code> to see how the 24-bit integer multiplication is performed.</td>
</tr>
<tr>
<td><code>geninteger32bit mul24(geninteger32bit x, geninteger32bit y)</code></td>
<td>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. <code>mul24</code> 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.</td>
</tr>
</tbody>
</table>

### 4.17.7. Common functions

In SYCL the OpenCL common functions are available in the namespace `sycl` on host and device as defined in the OpenCL 1.2 specification document par. 6.12.4. They are described here in Table 180. The built-in functions can take as input `float` or optionally `double` and their `vec` and `marray` counterparts.

**Table 180. Common functions which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification.**

<table>
<thead>
<tr>
<th>Common Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genfloat clamp(genfloat x, genfloat minval, genfloat maxval)</code></td>
<td>Returns <code>fmin(fmax(x, minval), maxval)</code>. Results are undefined if <code>minval &gt; maxval</code>.</td>
</tr>
<tr>
<td><code>genfloatf clamp(genfloatf x, float minval, float maxval)</code></td>
<td>Converts radians to degrees, i.e. $\frac{180}{\pi} \times \text{radians}$.</td>
</tr>
<tr>
<td><code>genfloat degrees(genfloat radians)</code></td>
<td>Returns y if $x &lt; y$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.</td>
</tr>
<tr>
<td><code>genfloat max(genfloat x, genfloat y)</code></td>
<td>Returns y if $x &lt; y$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.</td>
</tr>
<tr>
<td><code>genfloatf max(genfloatf x, float y)</code></td>
<td>Returns the linear blend of x and y implemented as: $x + (y - x) \times a$. a must be a value in the range $0.0 \ldots 1.0$. If a is not in the range $0.0 \ldots 1.0$, the return values are undefined.</td>
</tr>
<tr>
<td><code>genfloatd max(genfloatd x, double y)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloat min(genfloat x, genfloat y)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatf min(genfloatf x, float y)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatd min(genfloatd x, double y)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloat mix(genfloat x, genfloat y, genfloat a)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatf mix(genfloatf x, float y, float a)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatd mix(genfloatd x, genfloatd y, double a)</code></td>
<td></td>
</tr>
</tbody>
</table>
### Common Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>genfloat radians(genfloat degrees)</code></td>
<td>Converts degrees to radians, i.e. $(\pi / 180) \times degrees$.</td>
</tr>
<tr>
<td><code>genfloat step(genfloat edge, genfloat x)</code></td>
<td>Returns 0.0 if $x &lt; edge$, otherwise it returns 1.0.</td>
</tr>
<tr>
<td><code>genfloatf step(float edge, genfloatf x)</code></td>
<td>Returns 0.0 if $x &lt; edge0$ and 1.0 if $x \geq edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 &lt; x &lt; edge1$. This is useful in cases where you would want a threshold function with a smooth transition.</td>
</tr>
<tr>
<td><code>genfloatd step(double edge, genfloatd x)</code></td>
<td>This is equivalent to:</td>
</tr>
<tr>
<td><code>gentype t;</code></td>
<td>Results are undefined if $edge0 &gt; edge1$ or if $x$, $edge0$ or $edge1$ is a NaN.</td>
</tr>
<tr>
<td><code>t = clamp((x - edge0) / (edge1 - edge0), 0, 1);</code></td>
<td>Returns 1.0 if $x &gt; 0$, -0.0 if $x = -0.0$, +0.0 if $x = +0.0$, or -1.0 if $x &lt; 0$.</td>
</tr>
<tr>
<td><code>return t * t * (3 - 2 * t);</code></td>
<td>Returns 0.0 if $x$ is a NaN.</td>
</tr>
<tr>
<td><code>genfloat smoothstep (genfloat edge0, genfloat edge1, genfloat x)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatf smoothstep (float edge0, float edge1, genfloatf x)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloatd smoothstep (double edge0, double edge1, genfloatd x)</code></td>
<td></td>
</tr>
<tr>
<td><code>genfloat sign(genfloat x)</code></td>
<td></td>
</tr>
</tbody>
</table>

### 4.17.8. Geometric functions

In SYCL the OpenCL geometric functions are available in the namespace `sycl` on host and device as defined in the OpenCL 1.2 specification document par. 6.12.5. The built-in functions can take as input float or optionally double and their `vec` and `marray` counterparts, for dimensions 2, 3 and 4. On the host the vector types use the `vec` class and on an SYCL device use the corresponding native SYCL backend vector types. All of the geometric functions use round-to-nearest-even rounding mode. Table 181 contains the definitions of supported geometric functions.

**Table 181.** Geometric functions which work on SYCL host and device, are available in the `sycl` namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification.

<table>
<thead>
<tr>
<th>Geometric Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>float4 cross(float4 p0, float4 p1)</code></td>
<td>Returns the cross product of p0.xyz and p1.xyz. The $w$ component of <code>float4</code> result returned will be 0.0.</td>
</tr>
<tr>
<td><code>float3 cross(float3 p0, float3 p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>double4 cross(double4 p0, double4 p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>double3 cross(double3 p0, double3 p1)</code></td>
<td></td>
</tr>
<tr>
<td>Geometric Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td><code>mfloat4 cross(mfloat4 p0, mfloat4 p1)</code></td>
<td>Returns the cross product of first 3 components of p0 and p1. The 4th component of result returned will be 0.0.</td>
</tr>
<tr>
<td><code>mfloat3 cross(mfloat3 p0, mfloat3 p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>mdouble4 cross(mdouble4 p0, mdouble4 p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>mdouble3 cross(mdouble3 p0, mdouble3 p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>float dot(gengeofloat p0, gengeofloat p1)</code></td>
<td>Compute dot product.</td>
</tr>
<tr>
<td><code>double dot(gengeodouble p0, gengeodouble p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>float distance(gengeofloat p0, gengeofloat p1)</code></td>
<td>Returns the distance between p0 and p1. This is calculated as ( \text{length}(p0 - p1). )</td>
</tr>
<tr>
<td><code>double distance(gengeodouble p0, gengeodouble p1)</code></td>
<td></td>
</tr>
<tr>
<td><code>float length(gengeofloat p)</code></td>
<td>Return the length of vector p, i.e., ( \sqrt{p.x^2 + p.y^2 + \ldots} ).</td>
</tr>
<tr>
<td><code>double length(gengeodouble p)</code></td>
<td></td>
</tr>
<tr>
<td><code>gengeofloat normalize(gengeofloat p)</code></td>
<td>Returns a vector in the same direction as p but with a length of 1.</td>
</tr>
<tr>
<td><code>gengeodouble normalize(gengeodouble p)</code></td>
<td></td>
</tr>
<tr>
<td><code>float fast_distance(gengeofloat p0, gengeofloat p1)</code></td>
<td>Returns ( \text{fast_length}(p0 - p1). )</td>
</tr>
<tr>
<td><code>float fast_length(gengeofloat p)</code></td>
<td>Returns the length of vector p computed as: ( \sqrt{(\text{half})(\text{pow}(p.x, 2) + \text{pow}(p.y, 2) + \ldots)} ).</td>
</tr>
</tbody>
</table>
**Geometric Function**

| gengeofloat fast_normalize(gengeofloat p) |

**Description**

Returns a vector in the same direction as \( p \) but with a length of 1.

\[ \text{fast\_normalize} \text{ is computed as:} \]

\[ p \ast \text{rsqrt}((\text{half})(\text{pow}(p.x,2) + \text{pow}(p.y,2) + \ldots))) \]

The result shall be within 8192 ulps error from the infinitely precise result of

\[
\begin{align*}
\text{if} (\text{all}(p == 0.0f)) \\
\text{result} &= p; \\
\text{else} \\
\text{result} &= p / \text{sqrt}(\text{pow}(p.x, 2) + \text{pow}(p.y, 2) + \ldots); \\
\end{align*}
\]

with the following exceptions:

1. If the sum of squares is greater than \( \text{FLT\_MAX} \) then the value of the floating-point values in the result vector are undefined.
2. If the sum of squares is less than \( \text{FLT\_MIN} \) 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 \( \text{sqrt}(\text{FLT\_MIN}) \) may be flushed to zero before proceeding with the calculation.

---

### 4.17.9. Relational functions

The following functions are defined in the `sycl` namespace and are available on both host and device. These functions perform various relational comparisons on `vec`, `marray`, and scalar types.

The comparisons performed by `isequal`, `isgreater`, `isgreaterequal`, `isless`, `islessequal`, and `islessgreater` are false when one or both operands are NaN. The comparison performed by `isnotequal` is true when one or both operands are NaN.

The `vec` versions of these functions follow the definitions in the OpenCL 1.2 specification document [par. 6.12.6](#). Unless otherwise specified, these functions return a vector component of -1 (i.e. all bits set) when the comparison is true and 0 when the comparison is false.

**Table 182** shows the functions that are available for the `vec` type.

**Table 182. Relational functions for the `vec` template class.**
### Relational Function

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isequal(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isequal(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isequal(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isnotequal(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isnotequal(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isnotequal(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isgreater(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isgreater(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isgreater(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isgreaterequal(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isgreaterequal(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isgreaterequal(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isless(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isless(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isless(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>islessequal(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>islessequal(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>islessequal(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>islessgreater(half { n } x, half { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>islessgreater(float { n } x, float { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>islessgreater(double { n } x, double { n } y)</td>
</tr>
<tr>
<td><code>vec&lt;int16_t, { n }&gt;</code></td>
<td>isfinite(half { n } x)</td>
</tr>
<tr>
<td><code>vec&lt;int32_t, { n }&gt;</code></td>
<td>isfinite(float { n } x)</td>
</tr>
<tr>
<td><code>vec&lt;int64_t, { n }&gt;</code></td>
<td>isfinite(double { n } x)</td>
</tr>
<tr>
<td><code>int64_t</code></td>
<td>isfinite(half { n } x)</td>
</tr>
<tr>
<td><code>int32_t</code></td>
<td>isfinite(float { n } x)</td>
</tr>
<tr>
<td><code>int16_t</code></td>
<td>isfinite(double { n } x)</td>
</tr>
</tbody>
</table>

#### Description

- Returns the component-wise compare of `x == y`.  
- Returns the component-wise compare of `x != y`.  
- Returns the component-wise compare of `x > y`.  
- Returns the component-wise compare of `x >= y`.  
- Returns the component-wise compare of `x < y`.  
- Returns the component-wise compare of `x <= y`.  
- Returns the component-wise compare of `(x < y) || (x > y)`.  
- Test for finite value.
### Relational Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isinf</code></td>
<td>Test for infinity value (positive or negative).</td>
</tr>
<tr>
<td><code>isnan</code></td>
<td>Test for a NaN.</td>
</tr>
<tr>
<td><code>isnormal</code></td>
<td>Test for a normal value.</td>
</tr>
<tr>
<td><code>isordered</code></td>
<td>Test if arguments are ordered. Takes arguments (x, y), and returns the result (\text{isordered}(x, y) \land \text{isequal}(x, y)).</td>
</tr>
<tr>
<td><code>isunordered</code></td>
<td>Test if arguments are unordered. Takes arguments (x, y), returning non-zero if (x) or (y) is NaN, and zero otherwise.</td>
</tr>
<tr>
<td><code>signbit</code></td>
<td>Test for sign bit. Returns the following for each component in (x): -1 (i.e. all bits set) if the sign bit in the component value is set else returns 0.</td>
</tr>
<tr>
<td><code>any</code></td>
<td>Returns 1 if the most significant bit in any component of (x) is set; otherwise returns 0.</td>
</tr>
<tr>
<td><code>all</code></td>
<td>Returns 1 if the most significant bit in all components of (x) is set; otherwise returns 0.</td>
</tr>
<tr>
<td><code>bitselect</code></td>
<td>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).</td>
</tr>
<tr>
<td><code>select</code></td>
<td>For each component of a vector type: (\text{result}[i] = (\text{MSB of } c[i] \text{ is set}) ? b[i] : a[i]). (\text{vigeninteger}) and (\text{vugeninteger}) must have the same number of elements and bits as (\text{vgentype}).</td>
</tr>
</tbody>
</table>
Table 183 shows the functions that are available for the `marray` type and for scalar data types.

### Table 183. Relational functions for the marray template class and for scalar data types.

<table>
<thead>
<tr>
<th>Relational Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool isequal(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isequal(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x == y</code>.</td>
</tr>
<tr>
<td><code>bool isnotequal(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isnotequal(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x != y</code>.</td>
</tr>
<tr>
<td><code>bool isgreater(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isgreater(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x &gt; y</code>.</td>
</tr>
<tr>
<td><code>bool isgreaterequal(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isgreaterequal(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x &gt;= y</code>.</td>
</tr>
<tr>
<td><code>bool isless(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isless(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x &lt; y</code>.</td>
</tr>
<tr>
<td><code>bool islessequal(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>islessequal(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of <code>x &lt;= y</code>.</td>
</tr>
<tr>
<td><code>bool islessgreater(sgenfloat x, sgenfloat y)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>islessgreater(mgenfloat x, mgenfloat y)</code> Returns the component-wise compare of `(x &lt; y)</td>
</tr>
<tr>
<td><code>bool isfinite(sgenfloat x)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isfinite(mgenfloat x)</code> Test for finite value.</td>
</tr>
<tr>
<td><code>bool isinf(sgenfloat x)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isinf(mgenfloat x)</code> Test for infinity value (positive or negative).</td>
</tr>
<tr>
<td><code>bool isnan(sgenfloat x)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isnan(mgenfloat x)</code> Test for a NaN.</td>
</tr>
<tr>
<td><code>bool isnormal(sgenfloat x)</code></td>
<td><code>marray&lt; bool, { N }&gt;</code> <code>isnormal(mgenfloat x)</code> Test for a normal value.</td>
</tr>
</tbody>
</table>
| `bool isordered(sgenfloat x, sgenfloat y)` | `marray< bool, { N }>` `isordered(mgenfloat x, mgenfloat y)` Test if arguments are ordered. `isordered()` takes arguments `x` and `y`, and returns the result `isequal(x, x) && isequal(y, y)`.
<table>
<thead>
<tr>
<th>Relational Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bool isunordered(sgenfloat x, sgenfloat y)</code>&lt;br&gt;<code>marray&lt;bool, { N }&gt; isunordered(mgenfloat x, mgenfloat y)</code></td>
<td>Test if arguments are unordered. <code>isunordered()</code> takes arguments <code>x</code> and <code>y</code>, returning <code>true</code> if <code>x</code> or <code>y</code> is NaN, and <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>bool signbit(sgenfloat x)</code>&lt;br&gt;<code>marray&lt;bool, { N }&gt; signbit(mgenfloat x)</code></td>
<td>Test for sign bit, returning <code>true</code> if the sign bit in <code>x</code> is set, and <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>bool any(sigeninteger x)</code>&lt;br&gt;<code>bool any(migeninteger x)</code></td>
<td>Returns <code>true</code> if the most significant bit in any component of <code>x</code> is set; otherwise returns <code>false</code>.</td>
</tr>
<tr>
<td><code>bool all(sigeninteger x)</code>&lt;br&gt;<code>bool all(migeninteger x)</code></td>
<td>Returns <code>true</code> if the most significant bit in all components of <code>x</code> is set; otherwise returns <code>false</code>.</td>
</tr>
<tr>
<td><code>sgentype bitselect(sgentype a, sgentype b, sgentype c)</code>&lt;br&gt;<code>mgentype bitselect(mgentype a, mgentype b, mgentype c)</code></td>
<td>Each bit of the result is the corresponding bit of <code>a</code> if the corresponding bit of <code>c</code> is 0. Otherwise it is the corresponding bit of <code>b</code>.</td>
</tr>
<tr>
<td><code>sgentype select(sgentype a, sgentype b, bool c)</code>&lt;br&gt;<code>mgentype select(mgentype a, mgentype b, marray&lt;bool, { N }&gt;) c)</code></td>
<td>Returns the component-wise result = <code>c</code> ? <code>b</code> : <code>a</code>.</td>
</tr>
</tbody>
</table>
Chapter 5. SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying SYCL backend capabilities of target devices and limiting the requirements of device code to ensure portability.

5.1. Offline compilation of SYCL source files

There are two alternatives for a SYCL device compiler: a single-source device compiler and a device compiler that supports the technique of SMCP.

A SYCL device compiler takes in a C++ source file, extracts only the SYCL kernels and outputs the device code in a form that can be enqueued from host code by the associated SYCL runtime. How the SYCL runtime invokes the kernels is implementation-defined, but a typical approach is for a device compiler to produce a header file with the compiled kernel contained within it. By providing a command-line option to the host compiler, it would cause the implementation's SYCL header files to #include the generated header file. The SYCL specification has been written to allow this as an implementation approach in order to allow SMCP. However, any of the mechanisms needed from the SYCL compiler, the SYCL runtime and build system are implementation-defined, as they can vary depending on the platform and approach.

A SYCL single-source device compiler takes in a C++ source file and compiles both host and device code at the same time. This specification specifies how a SYCL single-source device compiler sees and outputs device code for kernels, but does not specify the host compilation.

5.2. Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation-defined format. In the case of the shared-source compilation model, the kernels have to be uniquely identified by both host and device compiler. This is required in order for the host runtime to be able to load the kernel by using a backend-specific host runtime interface.

From this requirement the following rules apply for naming the kernels:

- The kernel name is a C++ typename.
- The kernel name must be forward declarable at namespace scope (including global namespace scope) and may not be forward declared other than at namespace scope. If it isn't forward declared but is specified as a template argument in a kernel invoking interface, as described in Section 4.9.4.2, then it may not conflict with a name in any enclosing namespace scope.

The requirement that a kernel name be forward declarable makes some types for kernel names illegal, such as anything declared in the std namespace (adding a declaration to namespace std leads to undefined behavior).

- If the kernel is defined as a named function object type, the name can be the typename of the function object as long as it is either declared at namespace scope, or does not conflict with any name in an enclosing namespace scope.
- If the kernel is defined as a lambda, a typename can optionally be provided to the kernel invoking interface as described in Section 4.9.4.2, so that the developer can control the kernel name for purposes such as debugging or referring to the kernel when applying build options.
- If a kernel function relies on template parameters, then those template parameters must be contained by the kernel name. If such a kernel name is specified as a template argument in a kernel invoking interface, then the template parameters on which the kernel depends must be forward declarable at namespace scope.
In both single-source and shared-source implementations, a device compiler should detect the kernel invocations (e.g. `parallel_for<kernelname>`) in the source code and compile the enclosed kernels, storing them with their associated type name.

The format of the kernel and the compilation techniques are details of an implementation and not specified. The interface between the compiler and the runtime for extracting and executing SYCL kernels on the device is a detail of an implementation and not specified.

### 5.3. Compilation of functions

The SYCL device compiler parses an entire C++ source file supplied by the user, including any header files referenced via `#include` directives. From this source file, the SYCL device compiler must compile kernels for the device, as well as any functions that the kernels call.

The device compiler identifies kernels by looking for calls to **Kernel invocation commands** such as `parallel_for`. One of the parameters is a function object which is known as a **SYCL kernel function**, and this function must always return `void`. Any function called by the **SYCL kernel function** is also compiled for the device, and these functions together with the **SYCL kernel functions** are known as **device functions**. The device compiler searches recursively for any functions called from a **device function**, and these functions are also compiled for the device and known as **device functions**.

To illustrate, the following source code shows three functions and a kernel invoke with comments explaining which functions need to be compiled for the device.

```cpp
1 void f(handler& cgh) {
2     // Function "f" is not compiled for device
3
4     cgh.single_task [=] {
5         // This code is compiled for device
6         g(); // This line forces "g" to be compiled for device
7     };
8 }
9
10 void g() {
11     // Called from kernel, so "g" is compiled for device
12 }
13
14 void h() {
15     // Not called from a device function, so not compiled for device
16 }
```

In order for the SYCL device compiler to correctly compile **device functions**, all functions in the source file, whether **device functions** or not, must be syntactically correct functions according to this specification. A syntactically correct function adheres to at least the minimum required C++ version defined in **Section 3.9.1**.

### 5.4. Language restrictions for device functions

**Device functions** must abide by certain restrictions. The full set of C++ features are not available to these functions. Following is a list of these restrictions:

- Pointers and objects containing pointers may be shared. However, when a pointer is passed between SYCL devices or between the host and a SYCL device, dereferencing that pointer on the device produces undefined behavior unless the device supports **USM** and the pointer is an address within a **USM** memory region (see **Section 4.8**).
• Memory storage allocation is not allowed in kernels. All memory allocation for the device is done on the host using accessor classes or using USM as explained in Section 4.8. Consequently, the default allocation `operator new` overloads that allocate storage are disallowed in a SYCL kernel. The placement `new` operator and any user-defined overloads that do not allocate storage are permitted.

• Kernel functions must always have a `void` return type. A kernel lambda trailing-return-type that is not `void` is therefore illegal, as is a return statement (that would return from the kernel function) with an expression that does not convert to `void`.

• The odr-use of polymorphic classes and classes with virtual inheritance is allowed. However, no virtual member functions are allowed to be called in a `device function`.

• No function pointers or references are allowed to be called in a `device function`.

• RTTI is disabled inside `device functions`.

• No variadic functions are allowed to be called in a `device function`.

• Exception-handling cannot be used inside a `device function`. `noexcept` is allowed.

• Recursion is not allowed in a `device function`.

• Variables with thread storage duration (`thread_local` storage class specifier) are not allowed to be odr-used in a `device function`.

• Variables with static storage duration that are odr-used inside a `device function`, must be `const` or `constexpr` and zero-initialized or constant-initialized.

  Amongst other things, this restriction makes it illegal for a `device function` to access a global variable that isn’t `const` or `constexpr`.

• The rules for kernels apply to both the kernel function objects themselves and all functions, operators, member functions, constructors and destructors called by the kernel. This means that kernels can only use library functions that have been adapted to work with SYCL. Implementations are not required to support any library routines in kernels beyond those explicitly mentioned as usable in kernels in this spec. Developers should refer to the SYCL built-in functions in Section 4.17 to find functions that are specified to be usable in kernels.

• Interacting with a special SYCL runtime class (e.g. SYCL `accessor` or `stream`) that is stored within a C++ union is undefined behavior.

• Any variable or function that is odr-used from a `device function` must be defined in the same translation unit as that use. However, a function may be defined in another translation unit if the implementation defines the `SYCL_EXTERNAL` macro as described in Section 5.10.1.

5.5. Built-in scalar data types

In a SYCL device compiler, the device definition of all standard C++ fundamental types from Table 184 must match the host definition of those types, in both size and alignment. A device compiler may have this preconfigured so that it can match them based on the definitions of those types on the platform, or there may be a necessity for a device compiler command-line option to ensure the types are the same.

The standard C++ fixed width types, e.g. `int8_t`, `int16_t`, `int32_t`, `int64_t`, should have the same size as defined by the C++ standard for host and device.

Table 184. Fundamental data types supported by SYCL
<table>
<thead>
<tr>
<th>Fundamental data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>A conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0.</td>
</tr>
<tr>
<td>char</td>
<td>A signed or unsigned 8-bit integer, as defined by the C++ core language</td>
</tr>
<tr>
<td>signed char</td>
<td>A signed 8-bit integer, as defined by the C++ core language</td>
</tr>
<tr>
<td>unsigned char</td>
<td>An unsigned 8-bit integer, as defined by the C++ core language</td>
</tr>
<tr>
<td>short int</td>
<td>A signed integer of at least 16-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>An unsigned integer of at least 16-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>int</td>
<td>A signed integer of at least 16-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>unsigned int</td>
<td>An unsigned integer of at least 16-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>long int</td>
<td>A signed integer of at least 32-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>An unsigned integer of at least 32-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>long long int</td>
<td>An integer of at least 64-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td>An unsigned integer of at least 64-bits, as defined by the C++ core language</td>
</tr>
<tr>
<td>float</td>
<td>A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.</td>
</tr>
<tr>
<td>double</td>
<td>A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format. This type is only supported on devices that have <code>aspect::fp64</code>.</td>
</tr>
</tbody>
</table>
5.6. Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported. The following preprocessor macros must be defined by all conformant implementations:

- `SYCL_LANGUAGE_VERSION` substitutes an integer reflecting the version number and revision of the SYCL language being supported by the implementation. The version of SYCL defined in this document will have `SYCL_LANGUAGE_VERSION` substitute the integer `2020`, composed with the general SYCL version followed by 2 digits representing the revision number;

- `SYCL_DEVICE_COPYABLE` is defined to 1 if the implementation supports explicitly specified device copyable types as described in Section 3.13.1. Otherwise, the implementation’s definition of device copyable falls back to C++ trivially copyable and `sycl::is_device_copyable` is ignored;

- `__SYCL_DEVICE_ONLY__` is defined to 1 if the source file is being compiled with a SYCL device compiler which does not produce host binary;

- `__SYCL_SINGLE_SOURCE__` is defined to 1 if the source file is being compiled with a SYCL single-source compiler which produces host as well as device binary;

- `SYCL_FEATURE_SET_FULL` is defined to 1 if the SYCL implementation supports the full feature set and is not defined otherwise. For more details see Appendix B;

- `SYCL_FEATURE_SET_REDUCED` is defined to 1 if the SYCL implementation supports the reduced feature set and not the full feature set, otherwise it is not defined. For more details see Appendix B;

- `SYCL_EXTERNAL` is an optional macro which enables external linkage of SYCL functions and member functions to be included in a SYCL kernel. The macro is only defined if the implementation supports external linkage. For more details see Section 5.10.1.

In addition, for each SYCL backend supported, the preprocessor macros described in Section 4.1 must be defined by all conformant implementations.

5.7. Optional kernel features

A number of kernel features defined by this SYCL specification are optional; they may be supported on some devices but not on other devices. As described in Section 4.6.4.3, an application can test whether a device supports these features by testing whether the device has an associated aspect. The following aspects are those that correspond to optional kernel features:

- fp16
- fp64
- atomic64

In addition, the following C++ attributes from Section 5.8.1 also correspond to optional kernel features because they force the kernel to be compiled in a way that might not run on all devices:

- reqd_work_group_size()
- reqd_sub_group_size()

In order to guarantee source code portability of SYCL applications that use optional kernel features, all SYCL implementations must be able to compile device code that uses these optional features regardless of whether the implementation supports the features on any of its devices.

Of course, applications that make use of optional kernel features should ensure that a kernel using such a feature is submitted only to a device that supports the feature. If the application submits a command group using a secondary queue, then any kernel submitted from the command group should use only features that are supported by both the primary queue’s device and the secondary queue’s device. If an application fails to do this, the implementation must throw a synchronous exception with the `errc::ker-`
**nel_not_supported** error code from the **kernel invocation command** (e.g. `parallel_for()`).

It is legal for a SYCL application to define several kernels in the same translation unit even if they use different optional features, as shown in the following example:

```cpp
queue q1(dev1);
if (dev1.has(aspect::fp16)) {
    q1.submit([&](handler& cgh) {
        cgh.parallel_for<KernelA>(range { N }, [=](id i) {
            half fpShort = 1.0;
            /* ... */
        });
    });
}
queue q2(dev2);
if (dev2.has(aspect::atomic64)) {
    q2.submit([&](handler& cgh) {
        cgh.parallel_for<KernelB>(range { N }, [=](id i) {
            /* ... */
        },
        sycl::atomic_ref longAtomic(longValue);
        longAtomic.fetch_add(1);
    });
}
```

An implementation may not raise a compile time diagnostic or a run time exception merely due to speculative compilation of a kernel for a device when the application does not actually submit the kernel to that device. To illustrate using the example above, assume that device `dev1` does not have `aspect::atomic64` and device `dev2` does not have `aspect::fp16`. An implementation cannot raise a diagnostic due to compilation of `KernelA` for device `dev2` or for compilation of `KernelB` for device `dev1` because the application does not submit these kernels to those devices.

It is expected that this requirement will have an impact on the way an implementation bundles kernels into device images. For example, naively bundling `KernelA` and `KernelB` into the same device image could run afoul of this requirement if the implementation compiles the entire device image when `KernelA` is submitted to device `dev1`.

### 5.8. Attributes for device code

C++ attributes may be used to decorate kernels and device functions in order to influence the code generated by the device compiler. These attributes are all defined in the `[[sycl::]]` namespace.

If one of the attributes defined in this section is applied to a kernel or device function, it must be applied to the first declaration of that kernel or device function in the translation unit. Programs which fail to do this are ill formed and the compiler must issue a diagnostic. Redeclarations of the kernel or device function in the same translation unit may optionally have the same attribute applied (so long as the attribute arguments are the same between the declarations), but this is not required. The attribute remains in effect regardless of whether it appears in the redeclaration.

Unless an attribute's description specifically allows it, a kernel or device function may not be declared with the more than one instance of the same attribute unless all instances have the same attribute arguments. The compiler must issue a diagnostic for programs which violate this requirement. When two or more instances of the same attribute appear on the declaration of a kernel or device function, the effect is as though a single instance appeared (assuming that all instances have the same attribute arguments).
If a kernel or device function is declared with an attribute in one translation unit and the same kernel or
device function is declared without the same attribute (and its same attribute arguments) in another
translation unit, the program is ill formed and no diagnostic is required.

If any of these attributes are applied to a device function that is also compiled for the host, they have no
effect when the function is compiled for the host.

Applying these attributes to any language construct other than those specified in this section has imple-
mentation-defined effect.

5.8.1. Kernel attributes

The attributes listed in Table 185 have a different position depending on whether the kernel is defined as
a lambda function or as a named function object. If the kernel is a named function object, the attribute is
applied to the declarator-id in the function declaration. However, if the kernel is a lambda function, the
attribute is applied to the lambda declarator.

The reason for the different positions is because the C++ core language does not cur-
rently define a position for attributes to appertain to the lambda's corresponding func-
tion operator or operator template, only to the corresponding type of the function opera-
tor or operator template. This is expected to be remedied in a future version of the C++
core language specification.

The example below demonstrates these attribute positions using the `[[sycl::reqd_work_group_size(16)]]`
attribute. Note that the C++ core language allows two possible positions for kernels that are defined as a
named function object.

```cpp
// Kernel defined as a lambda
myQueue.submit([@](handler& h) {
    h.parallel_for(range<1>(16),
        [=](item<1> it) [[sycl::reqd_work_group_size(16)]] {
            // [kernel code]
        });
});

// Kernel defined as a named function object
class KernelFunctor1 {
public:
    [[sycl::reqd_work_group_size(16)]] void operator()(item<1> it) const {
        // [kernel code]
    }
};

// Kernel defined as a named function object
class KernelFunctor2 {
public:
    void operator() [[sycl::reqd_work_group_size(16)]] (item<1> it) const {
        // [kernel code]
    }
};
```

Table 185. Attributes for kernel functions
### SYCL attribute

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicates that the kernel must be launched with the specified work-group size. The number of arguments must match the dimensionality of the work-group used to invoke the kernel, and the order of the arguments matches the order of the dimension extents to the <code>range</code> constructor. Each argument must be an integral constant expression.</td>
</tr>
</tbody>
</table>

Kernels that are decorated with this attribute may not call functions that are defined in another translation unit via the `SYCL_EXTERNAL` macro.

Each device may have limitations on the work group sizes that it supports. If a kernel is decorated with this attribute and then submitted to a device that does not support the work group size, the implementation must throw a synchronous exception with the `errc::kernel_not_supported` error code. If the kernel is submitted to a device that does support the work group size, but the application provides an `nd_range` that does not match the size from the attribute, then the implementation must throw a synchronous exception with the `errc::nd_range` error code.

<table>
<thead>
<tr>
<th>Work group size hint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides a hint to the compiler about the work-group size most likely to be used when launching the kernel at runtime. The number of arguments must match the dimensionality of the work-group used to invoke the kernel, and the order of the arguments matches the order of the dimension extents to the <code>range</code> constructor. Each argument must be an integral constant expression. The effect of this attribute, if any, is implementation-defined.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SYCL attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reqd_work_group_size(dim0)</code></td>
</tr>
<tr>
<td><code>reqd_work_group_size(dim0, dim1)</code></td>
</tr>
<tr>
<td><code>reqd_work_group_size(dim0, dim1, dim2)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Work group size hint</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>work_group_size_hint(dim0)</code></td>
</tr>
<tr>
<td><code>work_group_size_hint(dim0, dim1)</code></td>
</tr>
<tr>
<td><code>work_group_size_hint(dim0, dim1, dim2)</code></td>
</tr>
<tr>
<td>SYCL attribute</td>
</tr>
<tr>
<td>-----------------------------------</td>
</tr>
<tr>
<td>vec_type_hint(&lt;type&gt;)</td>
</tr>
<tr>
<td>reqd_sub_group_size(dim)</td>
</tr>
</tbody>
</table>
### SYCL attribute

<table>
<thead>
<tr>
<th>device_has(aspect, ...)</th>
<th>Description</th>
</tr>
</thead>
</table>

This attribute may be used to decorate either the declaration of a kernel function that is defined in the current translation unit or to decorate the declaration of a non-kernel device function. The following description applies when the attribute decorates a kernel function.

The parameter list to the `device_has()` attribute consists of zero or more integral constant expressions, where each integer is interpreted as one of the enumerated values in the `aspect` enumeration type.

Specifying this attribute on a kernel has two effects. First, it causes the kernel invocation command to throw a synchronous exception with the `errc::kernel_not_supported` error code if the kernel is submitted to a device that does not have one of the listed aspects. (This includes the device associated with the secondary queue if the kernel is submitted from a command group that has a secondary queue.) Second, it causes the compiler to issue a diagnostic if the kernel (or any of the functions it calls) uses an optional feature that is associated with an aspect that is not listed in the attribute.

The value of each parameter to this attribute must be equal to one of the values in the `aspect` enumeration type (including any extended values the implementation may provide). If it does not, the program is ill formed and the compiler must issue a diagnostic.

See Listing 3 for an example of this attribute.

### Listing 3. Example of the `device_has()` attribute

```cpp
class KernelFunctor {
public:
    [[device_has(aspect::fp16)]] void operator()(item<1> it) const {
        foo();
    }
};
```
5.8.2. Device function attributes

The attributes in Table 186 are applied to the declaration of a non-kernel device function. The position of the attribute is the same as for the kernel function attributes defined above in Section 5.8.1.

Table 186. Attributes for non-kernel device functions
<table>
<thead>
<tr>
<th>SYCL attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device_has(aspect, ...)</code></td>
<td>This attribute may be used to decorate either the declaration of a kernel function that is defined in the current translation unit or to decorate the declaration of a non-kernel device function. The following description applies when the attribute decorates a non-kernel device function declaration. The syntax of this attribute's parameter list is the same as the syntax for the form of <code>sycl::device_has()</code> that is specified on a kernel function (see Table 185). This attribute is required when a non-kernel device function that uses optional device features is called in one translation unit and defined in another translation unit via the <code>SYCL_EXTERNAL</code> macro. When this attribute appears in a translation unit that calls the decorated device function, it is an assertion that the device function uses optional features that correspond to the aspects listed in the attribute. The program is ill formed if the called device function uses optional features that do not correspond to any of the aspects listed in the attribute, or if the function uses optional features and the attribute is not specified. No diagnostic is required in this case. When this attribute appears in a translation unit that defines the decorated device function, it causes the compiler to issue a diagnostic if the device function (or any of the functions it calls) uses an optional feature that is associated with an aspect that is not listed in the attribute.</td>
</tr>
</tbody>
</table>

5.9. Address-space deduction

C++ has no type-level support to represent address spaces. As a consequence, the SYCL generic programming model does not directly affect the C++ type of unannotated pointers and references.
Source level guarantees about address spaces in the SYCL generic programming model can only be achieved using pointer classes (instances of `multi_ptr`), which are regular classes that represent pointers to data stored in the corresponding address spaces.

In SYCL, the address space of pointer and references are derived from:

- Accessors that give access to shared data. They can be bound to a memory object in a command group and passed into a kernel. Accessors are used in scheduling of kernels to define ordering. Accessors to buffers have a compile-time address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) holds a pointer which is known to be addressing the address space represented by the `access::address_space`. This allows the compiler to determine whether the pointer references global, local, constant or private memory and generate code accordingly.
- Raw C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, explicit pointer classes, or accessors.

5.9.1. Address space assignment

In order to understand where data lives, the device compiler is expected to assign address spaces while lowering types for the underlying target based on the context. Depending on the SYCL backends and mode, address space deducing rules differ slightly.

If the target of the SYCL backend can represent the generic address space, then the "common address space deduction rules" in Section 5.9.2 and the "generic as default address space rules" in Section 5.9.3 apply. If the target of the SYCL backend cannot represent the generic address space, then the "common address space deduction rules" in Section 5.9.2 and the "inferred address space rules" in Section 5.9.4 apply.

SYCL address space does not affect the type, address space shall be understood as memory segment in which data is allocated. For instance, if `int i;` is allocated to the global address space, then `decltype(&i)` shall evaluate to `int*`.

5.9.2. Common address space deduction rules

The variable declarations get assigned to an address space depending on their scope and storage class:

- Namespace scope
  - If the type is `const`, the address space the declaration is assigned to is implementation-defined. If the target of the SYCL backend can represent the generic address space, then the assigned address space must be compatible with the generic address space.
    
    Namespace scope `const` declarations cannot be used within a kernel, as restricted in Section 5.4. This means that `const` global variables cannot be accessed by any device kernel or code called by the device kernel.

- Block scope and function parameter scope
  - Declarations with static storage duration are treated the same way as variables in namespace scope
  - Otherwise the declaration is assigned to the local address space if declared in a hierarchical context
  - Otherwise the declaration is assigned to the private address space

- Class scope
  - Static data members are treated the same way as for variable in namespace scope
The result of a prvalue-to-xvalue conversion is assigned to the local address space if it happens in a hierarchical context or to the private address space otherwise.

### 5.9.3. Generic as default address space

For SYCL backends that can represent the generic address space (see Section 5.9.1), unannotated pointers and references are considered to be pointing to the generic address space.

### 5.9.4. Inferred address space

**Note for this version**
The address space deduction feature described next is inherited from the SYCL 1.2.1 specifications. This section will be changed in a future version to better align with addition of generic address space and generic as default address space.

For SYCL backends that cannot represent the generic address space (see Section 5.9.1), inside kernels the SYCL device compiler will need to auto-deduce the memory region of unannotated pointer and reference types during the lowering of types from C++ to the underlying representation.

If a kernel function or device function contains a pointer or reference type, then the address space deduction must be attempted using the following rules:

- If an explicit pointer class is converted into a C++ pointer value, then the C++ pointer value will point to same address space as the one represented by the explicit pointer class.
- If a variable is declared as a pointer type, but initialized in its declaration to a pointer value with an already-deduced address space, then that variable will have the same address space as its initializer.
- If a function parameter is declared as a pointer type, and the argument is a pointer value with a deduced address space, then the function will be compiled as if the parameter had the same address space as its argument. It is legal for a function to be called in different places with different address spaces for its arguments: in this case the function is said to be “duplicated” and compiled multiple times. Each duplicated instance of the function must compile legally in order to have defined behavior.
- If a function return type is declared as a pointer type and return statements use address space deduced expressions, then the function will be compiled as if the return type had the same address space. To compile legally, all return expressions must deduce to the same address space.
- The rules for pointer types also apply to reference types. i.e. a reference variable takes its address space from its initializer. A function with a reference parameter takes its address space from its argument.
- If no other rule above can be applied to a declaration of a pointer, then it is assumed to be in the private address space.

It is illegal to assign a pointer value addressing one address space to a pointer variable addressing a different address space.

### 5.10. SYCL offline linking

#### 5.10.1. SYCL functions and member functions linkage

By default, any function that is odr-used from a device function must be defined in the same translation unit as that use. However, this restriction is relaxed if both of the following conditions are met:

- The implementation defines the `SYCL_EXTERNAL` macro;
- The translation unit that calls the function declares the function with `SYCL_EXTERNAL` as described
When a function is declared with `SYCL_EXTERNAL`, that macro must be used on the first declaration of that function in the translation unit. Redeclarations of the function in the same translation unit may optionally use `SYCL_EXTERNAL`, but this is not required.

When a function is declared with `SYCL_EXTERNAL`, that function must also be defined in some translation unit, where the function is declared with `SYCL_EXTERNAL`.

A function may only be declared with `SYCL_EXTERNAL` if it has external linkage by normal C++ rules.

A function declared with `SYCL_EXTERNAL` may be called from both host and device code. The macro has no effect when the function is called from host code.

In order to declare a function with `SYCL_EXTERNAL`, the macro name `SYCL_EXTERNAL` must appear before the function declaration. If the function is also decorated with C++ attributes that appear before the declaration, the `SYCL_EXTERNAL` may appear before, after, or between these attributes. The following example demonstrates the use of `SYCL_EXTERNAL`.

```cpp
#include <sycl/sycl.hpp>

SYCL_EXTERNAL void Foo();

SYCL_EXTERNAL void Bar() { /* ... */ }

SYCL_EXTERNAL extern void Baz();

[[nodiscard]] SYCL_EXTERNAL void Important();

SYCL_EXTERNAL [[nodiscard]] void AlsoImportant();
```

Functions that are declared using `SYCL_EXTERNAL` have the following additional restrictions beyond those imposed on other device functions:

- If the SYCL backend does not support the generic address space then the function cannot use raw pointers as parameter or return types. Explicit pointer classes must be used instead;
- The function cannot call `group::parallel_for_work_item`;
- The function cannot be called from a `parallel_for_work_group` scope.
Chapter 6. SYCL Extensions

This chapter describes the mechanism by which the core SYCL specification can be extended. Some parts of this chapter are requirements that all implementations must follow if they extend the core SYCL specification, while other parts of the chapter are merely guidelines. Unless a requirement is specifically stated as normative, all content in this chapter is a non-normative guideline.

An extension can be either of two flavors: an extension ratified by the Khronos SYCL group or a vendor supplied extension. In both cases, an extension is an optional feature set which an implementation need not implement in order to be conformant with the core SYCL specification.

Vendors may choose to define extensions in order to expose custom features or to gather feedback on an API that is not yet ready for inclusion in the core SYCL specification. Once a vendor extension has stabilized, the vendor is encouraged to promote it to a future version of the core SYCL specification or to a ratified Khronos extension. Thus, vendor extensions can be viewed as a pipeline of features for consideration in future SYCL versions.

The Khronos SYCL group may define extensions for features that are not yet ready for the core SYCL specification but are implemented by more than one vendor. These extensions also may be considered for inclusion in a future version of the core SYCL specification.

This chapter does not describe any particular extension to SYCL. Rather, it describes the mechanism for defining an extension. Each extension is defined by its own separate document. If an extension is ratified by the Khronos SYCL group, that group will release a document describing the extension. If a vendor defines an extension, the vendor is responsible for releasing its documentation.

### 6.1. Definition of an extension

An extension can take many possible forms. Some examples include:

- adding new types or free functions to the SYCL runtime;
- modifying existing SYCL classes, structs, or enumeration types by adding new members, member functions, or enumerated values;
- adding new overloads for existing free functions or member functions;
- defining new specializations for existing SYCL templates;
- adding new C++ attributes;
- adding new predefined macros;
- adding new keywords to the language;
- adding a new backend.

An extension may also broaden the definition of existing functions defined in the core SYCL specification by defining semantics for cases that are left unspecified by the core SYCL specification.

### 6.2. Requirements for an extension

This section is normative. All vendors which provide an extension must abide by the requirements described here.

An extension may not change the definition of existing functions defined by the core SYCL specification in a way that changes their specified behavior. Also, an extension may not remove any feature defined by the core SYCL specification.

The vendor must choose at least one `<vendorstring>` which uniquely identifies the vendor's SYCL imple-
The Khronos SYCL group does not provide any registry of the strings, so each vendor is responsible for choosing its own. One way to choose a unique string is to use the vendor's company name or a marketing name that is associated with the vendor's implementation. Ultimately, it is each vendor's responsibility to choose a string that is unique. The strings "khr" and "KHR" are reserved for the Khronos SYCL group for its own extensions, so vendors may not use these as a `<vendorstring>`.

The implementation must predefine at least one macro of the form `SYCL_IMPLEMENTATION_<vendorstring>` which allows applications to test whether they are being compiled with that vendor's implementation. For example, the Acme vendor could predefine a macro whose name is `SYCL_IMPLEMENTATION_ACME`.

### 6.3. Guidelines for portable extensions

Vendors who want to ensure that their extension does not collide with other vendors' extensions or with future versions of the core SYCL specification should follow the additional rules specified in this section. However, this is not a requirement for conformance.

#### 6.3.1. Extension namespace

If an extension adds new types or free functions, it should avoid adding these directly in the `sycl::` namespace since future versions of the core SYCL specification may also add new identifiers in this namespace. The namespace `sycl::ext::<vendorstring>` is reserved for use by extensions. For example, the Acme vendor could define extended types and free functions in the namespace `sycl::ext::acme`, and this would guarantee that they will not collide with definitions in other vendors' extensions or with future versions of the core SYCL specification.

#### 6.3.2. Names for extensions to existing classes or enumerations

An extension may add new members or member functions to existing SYCL classes or new values to existing SYCL enumeration types. To ensure these extensions do not collide, vendors are encouraged to name them with the prefix `ext_<vendorstring>`. For example, the Acme vendor could add a new member function to the `sycl::device` class named `device::ext_acme_fancy()` or a new value to the `sycl::aspect` enumeration named `aspect::ext_acme_fancier`.

In some cases, an extension does not have the freedom to choose a specific function name. For example, this could happen if the extension adds a new constructor overload for an existing SYCL class. In cases like this, the extension should ensure that one of the function parameters has a type that is defined in the extension's namespace. For example, the Acme vendor could add a new constructor for `sycl::context` with the signature `context(ext::acme::frobber&)`.

A similar situation can occur if an existing SYCL template is specialized with an extended enumerated value. Obviously, the extension cannot rename the template in this case. Instead, it is sufficient that the template is specialized with an extended enumerated value, and this guarantees that the extended specialization will not collide.

Vendors are encouraged to use the `ext_<vendorstring>` prefix form when possible for additions to existing SYCL classes because this form makes the extension's vendor name apparent. People reading application code will immediately know that a member function is an extension, and they will immediately know which vendor's documentation to consult.

#### 6.3.3. Feature test macros

Vendors are encouraged to group a related set of extensions together into a "feature" and to predefine a feature-test macro when the implementation supports the extensions in that feature. The feature-test macro should have the following form to ensure it is unique: `SYCL_EXT_<vendorstring>_<featurename>`. For
example, the Acme vendor might define a feature-test macro named `SYCL_EXT_ACME_FANCYFEATURE`. This allows applications to protect code using the extension with `#ifdef`, so that the code is skipped when compiled with an implementation that doesn't support the feature.

Since the interface to an extension might change from one release to another, vendors are also encouraged to predefine the macro's value to the version of the extension. Vendors should use a numerical value that monotonically increases for each revision of the extension API.

Of course, an extension may also predefine other macros. In order to ensure that these macro names do not collide with other extensions or future versions of the core SYCL specification, the name should start with the prefix `SYCL_EXT_<vendorstring>` or `SYCL_IMPLEMENTATION_<vendorstring>`.

### 6.3.4. Attribute namespace

An extension may define new C++ attributes. The attribute namespace `sycl::` is reserved for the core SYCL specification, so vendors should choose a different namespace for any attributes they add.

### 6.3.5. Include file paths

An extension may define new `#include` files under the "sycl" path. The path prefix "sycl/ext/<vendorstring>" is reserved for this purpose. For example, the Acme vendor could add a header file "sycl/ext/acme/fancy.h" and be guaranteed that it would not conflict with other extensions or with future versions of the core SYCL specification.

### 6.3.6. Optional kernel features

An extension may also add new optional kernel features—features which are supported on some devices but not on others. Vendors are encouraged to follow the same mechanism outlined in Section 5.7. Therefore, an extended optional kernel feature should have a matching extension to the `sycl::aspect` enumerated type.

### 6.3.7. Adding a backend

An extension may also add a new backend. If it does, the naming of the backend APIs follows the normal guidelines for extensions and also follows the naming pattern for backends that are defined in the core SYCL specification. To illustrate:

- The extension should add a new value to the `sycl::backend` enumeration type using a naming scheme like `ext_<vendorstring>_<backendname>`. For example, if the Acme vendor adds a backend named "foo", it would add an enumerated value named `sycl::backend::ext_acme_foo`.
- The extension should define the backend's interop API in a namespace named `sycl::ext::<vendorstring>::<backendname>`. For our hypothetical Acme example, this would be a namespace named `sycl::ext::acme::foo`.
- If the backend interop API is available through a separate header file, that header should be named "sycl/ext/<vendorstring>/backend/<backendname>.hpp". For our hypothetical Acme example this would be "sycl/ext/acme/backend/foo.hpp".
- The extension should predefine a macro for the backend when it is "active". The name of this macro should be `SYCL_EXT_<vendorstring>_BACKEND_<backendname>`. For our hypothetical Acme example this would be `SYCL_EXT_ACME_BACKEND_FOO`.
Appendix A: Information descriptors

This appendix contains the definitions of all the SYCL information descriptors introduced in Chapter 4.

A.1. Platform information descriptors

The following interface includes all the information descriptors for the `platform` class as described in Table 18.

```cpp
namespace sycl {
    namespace info {
        namespace platform {
            struct profile;
            struct version;
            struct name;
            struct vendor;
            struct extensions; // Deprecated
        } // namespace platform
    } // namespace info
} // namespace sycl
```

A.2. Context information descriptors

The following interface includes all the information descriptors for the `context` class as described in Table 21.

```cpp
namespace sycl {
    namespace info {
        namespace context {
            struct platform;
            struct devices;
            struct atomic_memory_order_capabilities;
            struct atomic_fence_order_capabilities;
            struct atomic_memory_scope_capabilities;
            struct atomic_fence_scope_capabilities;
        } // namespace context
    } // namespace info
} // namespace sycl
```

A.3. Device information descriptors

The following interface includes all the information descriptors for the `device` class as described in Table 25.

```cpp
namespace sycl {
    namespace info {
        namespace device {
        }
    } // namespace info
} // namespace sycl
```
A.3. Device information descriptors

```c
struct device_type;
struct vendor_id;
struct max_compute_units;
struct max_work_item_dimensions;

// Template
template <int Dimensions = 3> struct max_work_item_sizes;
struct max_work_group_size;
struct preferred_vector_width_char;
struct preferred_vector_width_short;
struct preferred_vector_width_int;
struct preferred_vector_width_long;
struct preferred_vector_width_float;
struct preferred_vector_width_double;
// Deprecated
struct native_vector_width_char;
struct native_vector_width_short;
struct native_vector_width_int;
struct native_vector_width_long;
struct native_vector_width_double;
struct native_vector_width_half;
struct max_clock_frequency;
struct address_bits;
struct max_mem_alloc_size;
struct image_support; // Deprecated
struct max_read_image_args;
struct max_write_image_args;
struct image2d_max_height;
struct image2d_max_width;
struct image3d_max_height;
struct image3d_max_width;
struct image3d_max_depth;
struct image_max_buffer_size;
struct max_sampler_size;
struct mem_base_addr_align;
struct half_fp_config;
struct single_fp_config;
struct double_fp_config;
struct global_mem_cache_type;
struct global_mem_cache_line_size;
struct global_mem_cache_size;
struct global_mem_size;
struct max_constant_buffer_size; // Deprecated
struct max_constant_args; // Deprecated
struct local_mem_type;
struct local_mem_size;
struct error_correction_support;
struct host_unified_memory;
struct atomic_memory_order_capabilities;
struct atomic_fence_order_capabilities;
struct atomic_memory_scope_capabilities;
struct atomic_fence_scope_capabilities;
struct profiling_timer_resolution;
struct is_endian_little;
struct is_available;
```
struct is_compiler_available;  // Deprecated
struct is_linker_available;  // Deprecated
struct execution_capabilities;
struct queue_profiling;  // Deprecated
struct built_in_kernels;  // Deprecated
struct built_in_kernel_ids;
struct platform;
struct name;
struct vendor;
struct driver_version;
struct profile;
struct version;
struct backend_version;
struct aspects;
struct extensions;  // Deprecated
struct printf_buffer_size;
struct preferred_interop_user_sync;
struct parent_device;
struct partition_max_sub_devices;
struct partition_properties;
struct partition_affinity_domains;
struct partition_type_property;
struct partition_type_affinity_domain;

}  // namespace device

enum class device_type : /* unspecified */ {
cpu,       // Maps to OpenCL CL_DEVICE_TYPE_CPU
gpu,      // Maps to OpenCL CL_DEVICE_TYPE_GPU
accelerator, // Maps to OpenCL CL_DEVICE_TYPE_ACCELERATOR
custom,    // Maps to OpenCL CL_DEVICE_TYPE_CUSTOM
automatic, // Maps to OpenCL CL_DEVICE_TYPE_DEFAULT
host,
all // Maps to OpenCL CL_DEVICE_TYPE_ALL
};

enum class partition_property : /* unspecified */ {
no_partition,
partition_equally,
partition_by_counts,
partition_by_affinity_domain
};

enum class partition_affinity_domain : /* unspecified */ {
not_applicable,
numa,
L4_cache,
L3_cache,
L2_cache,
L1_cache,
next_partitionable
};

enum class local_mem_type : /* unspecified */ { none, local, global }

enum class fp_config : /* unspecified */ {
A.4. Queue information descriptors

The following interface includes all the information descriptors for the queue class as described in Table 30.

```c
namespace sycl {
namespace info {
namespace queue {

  struct context;
  struct device;

} // namespace queue
} // namespace info
} // namespace sycl
```

A.5. Kernel information descriptors

The following interface includes all the information descriptors that apply to kernels as described in Table 137.

```c
namespace sycl {
namespace info {
namespace kernel {

  struct num_args;
  struct attributes;

} // namespace kernel
} // namespace info
} // namespace sycl
```
A.6. Event information descriptors

The following interface includes all the information descriptors for the `event` class as described in Table 35 and Table 37.
Appendix B: Feature sets

As of SYCL 2020 there are now two distinct feature sets which a SYCL implementation can conform to, in order to better fit the requirements of different domains, such as embedded, mobile, and safety critical, which may have limitations because of the toolchains used.

A SYCL implementation can choose to conform to either the full feature set or the reduced feature set.

B.1. Full feature set

The full feature set includes all features specified in the core SYCL specification with no exceptions.

B.2. Reduced feature set

The reduced feature set makes certain features optional or restricted to specific forms. The following list defines all the differences between the reduced feature set and the full feature set.

1. **Un-named SYCL kernel functions**: SYCL kernel functions which are defined using a lambda expression and therefore have no standard name are required to be provided a name via the kernel name template parameter of kernel invocation functions such as `parallel_for`. This overrides the core SYCL specification rules for SYCL kernel function naming as specified in Section 4.9.4.2.

2. **Address space mode**: The address space assignment mode used in the reduced feature set is not required to be generic address space, regardless of SYCL backend in use. Instead the inferred address space mode may always be used.

3. **Declarations**: In addition to the requirements specified in Section 5.9.2, the reduced feature set does not require support for odr-use inside device functions of variables declared `const` or `constexpr` with static storage duration.

B.3. Compatibility

In order to avoid introducing any kind of divergence the reduced and full feature sets are defined such that the full feature set is a subsumption of the reduced feature set. This means that any applications which are developed for the reduced feature set will be compatible with both a SYCL reduced implementation and a SYCL full implementation.

B.4. Conformance

One of the reasons for having this be defined in the specification is that hardware vendors which wish to support SYCL on their platform(s) want to be able to demonstrate their support for it by passing conformance. However, if passing conformance means adopting features which they do not believe to be necessary at an additional development effort then this may deter them.

Each feature set has its own route for passing conformance allowing adopters of SYCL to specify the feature set they wish to test conformance against. The conformance test suite would then alter or disable the tests within the test suite according to how the feature sets are differentiated above.
Appendix C: OpenCL backend specification

This chapter describes how the SYCL general programming model is mapped on top of OpenCL, and how the SYCL generic interoperability interface must be implemented by vendors providing SYCL for OpenCL implementations to ensure SYCL applications written for the OpenCL backend are interoperable.

C.1. SYCL application interoperability native backend objects

For each SYCL runtime class which supports SYCL application interoperability, specializations of backend_traits::input_type and backend_traits::return_type must be defined as the type of SYCL application interoperability native backend object associated with SyclType for the SYCL backend.

The types of the native backend objects for SYCL application interoperability are described in Table 191.

C.2. Kernel function interoperability native backend objects

For each SYCL runtime class which supports kernel function interoperability, a specialization of backend_traits::return_type must be defined as the type of kernel function interoperability native backend object associated with SyclType for the SYCL backend.

The types of the native backend objects for kernel function interoperability are described in Table 187.

### Table 187. Types of native backend objects kernel function interoperability

<table>
<thead>
<tr>
<th>SyclType</th>
<th>backend::return_t&lt;backend::opencl, SyclType&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessor&lt;T, Dims, Mode, target::device&gt;</td>
<td>__global T*</td>
</tr>
<tr>
<td>accessor&lt;T, Dims, Mode, target::constant_buffer&gt;</td>
<td>__constant T*</td>
</tr>
<tr>
<td>accessor&lt;T, Dims, Mode, target::local&gt;</td>
<td>__local T*</td>
</tr>
<tr>
<td>local_accessor&lt;T, Dims&gt;</td>
<td>__local T*</td>
</tr>
<tr>
<td>sampled_image_accessor&lt;T, 1, Mode, image_target::device&gt;</td>
<td>sampler_1dimage_pair_t</td>
</tr>
<tr>
<td>sampled_image_accessor&lt;T, 2, Mode, image_target::device&gt;</td>
<td>sampler_2dimage_pair_t</td>
</tr>
<tr>
<td>sampled_image_accessor&lt;T, 3, Mode, image_target::device&gt;</td>
<td>sampler_3dimage_pair_t</td>
</tr>
<tr>
<td>unsampled_image_accessor&lt;T, 1, Mode, image_target::device&gt;</td>
<td>image1d_t</td>
</tr>
<tr>
<td>unsampled_image_accessor&lt;T, 2, Mode, image_target::device&gt;</td>
<td>image2d_t</td>
</tr>
<tr>
<td>unsampled_image_accessor&lt;T, 3, Mode, image_target::device&gt;</td>
<td>image3d_t</td>
</tr>
<tr>
<td>stream</td>
<td>__global cl_char*</td>
</tr>
<tr>
<td>device_event</td>
<td>event_t</td>
</tr>
</tbody>
</table>

The sampler_1dimage_pair_t, sampler_1dimage_pair_t and sampler_1dimage_pair_t types must be implemented as described below.

```c
struct sampler_1dimage_pair_t {
  sampler_t sampler;
  image1d_t image;
};
```

```c
struct sampler_2dimage_pair_t {
  sampler_t sampler;
};
```
C.3. Destruction of interop constructed objects with reference semantics

On destruction of the last copy of an instance of a SYCL class which is specified to have reference semantics as described in Section 4.5.2 that was constructed using one of the SYCL backend interoperability make_* functions specified in Section 4.5.1.3 additional lifetime related operations may be performed which are required for the underlying native backend object.

The additional behavior performed by the OpenCL SYCL backend for each SYCL class is described in Table 188.

Table 188. Destructor behavior of interop constructed objects with reference semantics

<table>
<thead>
<tr>
<th>SYCL object</th>
<th>Destructor behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>accessor</td>
<td>No additional behavior is performed.</td>
</tr>
<tr>
<td>buffer</td>
<td><code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.</td>
</tr>
<tr>
<td>context</td>
<td><code>clReleaseContext</code> will be called on the native <code>cl_context</code> object provided during construction.</td>
</tr>
<tr>
<td>device</td>
<td><code>clReleaseDevice</code> will be called on the native <code>cl_device</code> object provided during construction.</td>
</tr>
<tr>
<td>event</td>
<td><code>clReleaseEvent</code> will be called on the native <code>cl_event</code> object provided during construction.</td>
</tr>
<tr>
<td>kernel</td>
<td><code>clReleaseKernel</code> will be called on the native <code>cl_kernel</code> objects provided during construction.</td>
</tr>
<tr>
<td>kernel_bundle</td>
<td><code>clReleaseProgram</code> will be called on the native <code>cl_program</code> objects provided during construction.</td>
</tr>
<tr>
<td>platform</td>
<td>No additional behavior is performed.</td>
</tr>
<tr>
<td>queue</td>
<td><code>clReleaseCommandQueue</code> will be called on the native <code>cl_command_queue</code> object provided during construction.</td>
</tr>
<tr>
<td>sampled_image</td>
<td><code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.</td>
</tr>
<tr>
<td>unsampled_image</td>
<td><code>clReleaseMemObject</code> will be called on the native <code>cl_mem</code> object provided during construction.</td>
</tr>
</tbody>
</table>

C.4. SYCL for OpenCL framework

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

- **SYCL C++ template library:** The template library provides a set of C++ templates and classes which
provide the programming model to the user. It enables the creation of runtime classes such as SYCL queues, buffers and images, as well as access to some underlying OpenCL runtime object, such as contexts, platforms, devices and program objects.

- **SYCL runtime**: The SYCL runtime interfaces with the underlying OpenCL implementations and handles scheduling of commands in queues, moving of data between host and devices, manages contexts, programs, kernel compilation and memory management.

- OpenCL Implementation(s): The SYCL system assumes the existence of one or more OpenCL implementations available on the host machine.

- **SYCL device compilers**: The SYCL device compilers compile SYCL C++ kernels into a format which can be executed on an OpenCL device at runtime. There may be more than one SYCL device compiler in a SYCL implementation. The format of the compiled SYCL kernels is not defined. A SYCL device compiler may, or may not, also compile the host parts of the program.

The OpenCL backend is enabled using the `sycl::backend::opencl` value of `enum class backend`. That means that when the OpenCL backend is active, the value of `sycl::is_backend_active<sycl::backend::opencl>::value` will be `true`.

### C.5. Mapping of SYCL programming model on top of OpenCL

The SYCL programming model was originally designed as a high-level model for the OpenCL API, hence the mapping of SYCL on the OpenCL API is mostly straightforward.

When the OpenCL backend is active on a SYCL application, all visible OpenCL platforms are exported as SYCL platforms.

When a SYCL implementation executes kernels on an OpenCL device, it achieves this by enqueuing OpenCL commands to execute computations on the processing elements within a device. The processing elements within an OpenCL compute unit may execute a single stream of instructions as ALUs within a SIMD unit (which execute in lockstep with a single stream of instructions), as independent SPMD units (where each PE maintains its own program counter) or as some combination of the two.

#### C.5.1. Backend specific information descriptors

Some of the SYCL information descriptors are backend-defined. For the OpenCL backend these information descriptors map directly to OpenCL properties as described in the table below:

<table>
<thead>
<tr>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>info::platform::version</code></td>
<td><code>CL_PLATFORM_VERSION</code></td>
</tr>
<tr>
<td><code>info::device::version</code></td>
<td><code>CL_DEVICE_VERSION</code></td>
</tr>
</tbody>
</table>

#### C.5.2. OpenCL memory model

The memory model for SYCL devices running on OpenCL platforms follows the memory model of the OpenCL version they conform to.

In addition to global memory, local memory and private memory memory, the OpenCL backend permits the use of constant memory space in SYCL:

- **Constant-memory** is a region of memory that remains constant during the execution of a kernel. A
pointer to the generic address space cannot represent an address to this memory region.

Work-items executing in a kernel have access to four distinct memory regions, with the mapping between SYCL and OpenCL described in Table 190.

Table 190. Mapping of SYCL memory regions into OpenCL memory regions

<table>
<thead>
<tr>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>Global memory</td>
</tr>
<tr>
<td>Constant</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Local</td>
<td>Local memory</td>
</tr>
<tr>
<td>Private</td>
<td>Private memory</td>
</tr>
</tbody>
</table>

C.5.3. OpenCL interface for buffer command accessors

The enumerator target::constant_buffer is deprecated, but will remain a part of the OpenCL backend as an extension. This enables SYCL kernel functions to access the contents of a buffer through the OpenCL device's constant memory.

C.5.4. OpenCL resources managed by SYCL application

In OpenCL, a developer must create a context to be able to execute commands on a device. Creating a context involves choosing a platform and a list of devices. In SYCL, contexts, platforms and devices all exist, but the user can choose whether to specify them or have the SYCL implementation create them automatically. The minimum required object for submitting work to devices in SYCL is the queue, which contains references to a platform, device and context internally.

The resources managed by SYCL are:

1. **Platforms**: all features of OpenCL are implemented by platforms. A platform can be viewed as a given hardware vendor's runtime and the devices accessible through it. Some devices will only be accessible to one vendor's runtime and hence multiple platforms may be present. SYCL manages the different platforms for the user. In SYCL, a platform resource is accessible through a sycl::platform object.

2. **Contexts**: any OpenCL resource that is acquired by the user is attached to a context. A context contains a collection of devices that the host can use and manages memory objects that can be shared between the devices. Data movement between devices within a context may be efficient and hidden by the underlying OpenCL runtime while data movement between contexts may involve the host. A given context can only wrap devices owned by a single platform. In SYCL, a context resource is accessible through a sycl::context object.

3. **Devices**: platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a sycl::device object.

4. **Kernel bundles**: OpenCL objects that store implementation data for the SYCL kernels. These objects are only required for advanced use in SYCL and are encapsulated in the sycl::kernel_bundle class.

5. **Queues**: SYCL kernels execute in command queues. The user must create a queue, which references an associated context, platform and device. The context, platform and device may be chosen automatically, or specified by the user. In SYCL, command queues are accessible through sycl::queue objects.

C.6. Interoperability with the OpenCL API

The OpenCL backend for SYCL ensures maximum compatibility between SYCL and OpenCL kernels and API. This includes supporting devices with different capabilities and support for different versions of the OpenCL C language, in addition to supporting SYCL kernels written in C++.
SYCL runtime classes which encapsulate an OpenCL opaque type such as SYCL context or SYCL queue must provide an interoperability constructor taking an instance of the OpenCL opaque type. When the OpenCL object supports reference counting, these constructors must retain that instance to increase the reference count of the OpenCL resource. Likewise, the destructor for the SYCL runtime classes which encapsulate a reference counted OpenCL opaque type must release that instance to decrease the reference count of the OpenCL resource. Since the OpenCL platform_id is not reference counted, the encapsulating SYCL platform class neither retains nor releases this OpenCL resource.

Note that an instance of a SYCL runtime class which encapsulates an OpenCL opaque type can encapsulate any number of instances of the OpenCL type, unless it was constructed via the interoperability constructor, in which case it can encapsulate only a single instance of the OpenCL type.

The lifetime of a SYCL runtime class that encapsulates an OpenCL opaque type and the instance of that opaque type retrieved via the get_native() free function are not tied in either direction given correct usage of OpenCL reference counting. For example if a user were to retrieve a cl_command_queue instance from a SYCL queue instance and then immediately destroy the SYCL queue instance, the cl_command_queue instance is still valid. Or if a user were to construct a SYCL queue instance from a cl_command_queue instance and then immediately release the cl_command_queue instance, the SYCL queue instance is still valid.

Note that a SYCL runtime class that encapsulates an OpenCL opaque type is not responsible for any incorrect use of OpenCL reference counting outside of the SYCL runtime. For example if a user were to retrieve a cl_command_queue instance from a SYCL queue instance and then release the cl_command_queue instance more than once without any prior retain then the SYCL queue instance that the cl_command_queue instance was retrieved from is now undefined.

Note that an instance of the SYCL buffer or SYCL image class templates constructed via the interoperability constructor is free to copy from the cl_mem into another memory allocation within the SYCL runtime to achieve normal SYCL semantics, for as long as the SYCL buffer or SYCL image instance is alive.

Table 191 relates SYCL objects to their OpenCL native type in the SYCL application.

<table>
<thead>
<tr>
<th>SyclType</th>
<th>backend_input_t&lt;backend::opencl, SyclType&gt;</th>
<th>backend_return_t&lt;backend::opencl, SyclType&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>platform</td>
<td>cl_platform_id</td>
<td>cl_platform_id</td>
<td>A SYCL platform object encapsulates an OpenCL platform ID.</td>
</tr>
<tr>
<td>device</td>
<td>cl_device_id</td>
<td>cl_device_id</td>
<td>A SYCL device object encapsulates an OpenCL device ID.</td>
</tr>
<tr>
<td>context</td>
<td>cl_context</td>
<td>cl_context</td>
<td>A SYCL context object encapsulates an OpenCL context object.</td>
</tr>
<tr>
<td>queue</td>
<td>cl_command_queue</td>
<td>cl_command_queue</td>
<td>A SYCL queue object encapsulates an OpenCL queue object.</td>
</tr>
<tr>
<td>kernel</td>
<td>cl_kernel</td>
<td>cl_kernel</td>
<td>A SYCL kernel object encapsulates an OpenCL kernel object.</td>
</tr>
</tbody>
</table>
### Table 192

**SYCL kernel native types in OpenCL**

<table>
<thead>
<tr>
<th>SYCL kernel native types in OpenCL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>multi_ptr::get_decorated()</code></td>
<td>Returns a pointer in the OpenCL address space corresponding to the type of multi pointer object</td>
</tr>
</tbody>
</table>

When a buffer or image is allocated on more than one OpenCL device, if these devices are on separate contexts then multiple `cl_mem` objects may be allocated for the memory object, depending on whether the object has actively been used on these devices yet or not.

The OpenCL C function qualifier `__kernel` and the access qualifiers: `__read_only`, `__write_only` and `__read_write` are not exposed in SYCL via keywords, but are instead encapsulated in SYCL's parameter passing system inside accessors. Users wishing to achieve the OpenCL equivalent of these qualifiers in SYCL should instead use SYCL accessors with equivalent semantics.

Any OpenCL C function included in a pre-built OpenCL library can be defined as an `extern "C"` function and the OpenCL program has to be linked against any SYCL program that contains kernels using the external function. In this case, the data types used have to comply with the interoperability aliases defined in Table 194.
C.7. Programming interface

The following section describes the OpenCL-specific API.

C.7.1. Construct SYCL objects from OpenCL ones

The OpenCL backend provides the following specializations of the `make_{sycl_class}` template functions which are defined in Section 4.5.1.3. These functions are in the `sycl` namespace.

<table>
<thead>
<tr>
<th>OpenCL interoperability function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>context make_context(const cl_context&amp; clContext, const async_handler&amp; asyncHandler = {})</code></td>
<td>Constructs a SYCL <code>context</code> instance from an OpenCL <code>cl_context</code> in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td><code>event make_event(const std::vector&lt;cl_event&gt;&amp; clEvents, const context&amp; syclContext)</code></td>
<td>Constructs a SYCL <code>event</code> instance from a vector of OpenCL <code>cl_event</code> objects in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td><code>device make_device(const cl_device_id&amp; clDeviceId)</code></td>
<td>Constructs a SYCL <code>device</code> instance from an OpenCL <code>cl_device_id</code> in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td><code>platform make_platform(const cl_platform_id&amp; clPlatformId)</code></td>
<td>Constructs a SYCL <code>platform</code> instance from an OpenCL <code>cl_platform_id</code> in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td><code>queue make_queue(const cl_command_queue&amp; clQueue, const context&amp; syclContext, const async_handler&amp; asyncHandler = {})</code></td>
<td>Constructs a SYCL <code>queue</code> instance with an optional <code>async_handler</code> from an OpenCL <code>cl_command_queue</code> in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td><code>template &lt;typename T, int Dimensions = 1, typename AllocatorT = buffer_allocator&lt;std::remove_const_t&lt;T&gt;&gt;, buffer&lt;T, Dimensions, AllocatorT&gt; make_buffer(const cl_mem&amp; clMemObject, const context&amp; syclContext, event availableEvent)</code></td>
<td>Available only when: <code>Dimensions == 1</code>. Constructs a SYCL <code>buffer</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in Section 4.5.1. The instance of the SYCL <code>buffer</code> class template being constructed must wait for the SYCL <code>event</code> parameter, <code>availableEvent</code> to signal that the <code>cl_mem</code> instance is ready to be used. The SYCL <code>context</code> parameter <code>syclContext</code> is the context associated with the memory object.</td>
</tr>
</tbody>
</table>
### OpenCL interoperability function

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available only when: ( \text{Dimensions} = 1 ).</td>
</tr>
</tbody>
</table>

Constructs a SYCL buffer instance from an OpenCL cl_mem in accordance with the requirements described in Section 4.5.1.

**Template**

```cpp
template <typename T, int Dimensions = 1, typename AllocatorT = buffer_allocator<remove_const_t<T>>>(
    buffer<T, Dimensions, AllocatorT> make_buffer(clMemObject, const context& syclContext)
```

**Available when:**

- \( \text{Dimensions} = 1 \).

**Description:**

Constructs a SYCL sampled_image instance from an OpenCL cl_mem in accordance with the requirements described in Section 4.5.1. The instance of the SYCL image class template being constructed must wait for the SYCL event parameter, availableEvent to signal that the cl_mem instance is ready to be used. The SYCL context parameter syclContext is the context associated with the memory object.

**Template**

```cpp
template <int Dimensions = 1, typename AllocatorT = image_allocator>
    sampled_image<Dimensions, AllocatorT> make_sampled_image(clMemObject, const context& syclContext, image_sampler syclImageSampler, event availableEvent)
```

**Available when:**

- \( \text{Dimensions} = 1 \).

**Description:**

Constructs a SYCL unsampled_image instance from an OpenCL cl_mem in accordance with the requirements described in Section 4.5.1. The instance of the SYCL image class template being constructed must wait for the SYCL event parameter, availableEvent to signal that the cl_mem instance is ready to be used. The SYCL context parameter syclContext is the context associated with the memory object.

**Template**

```cpp
template <int Dimensions = 1, typename AllocatorT = image_allocator>
    unsampled_image<Dimensions, AllocatorT> make_unsampled_image(clMemObject, const context& syclContext, image_sampler syclImageSampler, event availableEvent)
```

**Available when:**

- \( \text{Dimensions} = 1 \).
### OpenCL interoperability function

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template &lt;int Dimensions = 1, typename AllocatorT = image_allocator&gt; unsampled_image&lt;Dimensions, AllocatorT&gt; make_unsampled_image(const cl_mem&amp; clMemObject, const context&amp; syclContext)</td>
<td>Constructs a SYCL <code>unsampled_image</code> instance from an OpenCL <code>cl_mem</code> in accordance with the requirements described in Section 4.5.1.</td>
</tr>
<tr>
<td>kernel make_kernel(const cl_kernel&amp; clKernel, const context&amp; syclContext);</td>
<td>Constructs a SYCL <code>kernel</code> instance from an OpenCL kernel object.</td>
</tr>
</tbody>
</table>
| template <bundle_state State> kernel_bundle<State> make_kernel_bundle(const cl_program& clProgram, const context& syclContext) | Constructs a SYCL `kernel_bundle` instance from an OpenCL `cl_program` for the devices in `syclContext` in accordance with the requirements described in Section 4.5.1. The SYCL `context` must represent the same underlying OpenCL context associated with the OpenCL program object. The `state` specifies the expected `kernel_bundle` state. The mapping between the `kernel_bundle` state and OpenCL program state (CL_PROGRAM_BINARY_TYPE) is as follows:

1. `bundle_state::input` - CL_PROGRAM_BINARY_TYPE_NONE
2. `bundle_state::object` - CL_PROGRAM_BINARY_TYPE_COMPILED_OBJECT or CL_PROGRAM_BINARY_TYPE_INTERMEDIATE or CL_PROGRAM_BINARY_TYPE_LIBRARY.
3. `bundle_state::executable` - CL_PROGRAM_BINARY_TYPE_EXECUTABLE

If the internal state of the OpenCL program doesn’t match `state`, the kernel bundle will be compiled and linked as necessary. If the OpenCL program is already an executable binary, but the specified `state` is not `bundle_state::executable`, an exception with the errc::invalid error code is thrown. If the specified `state` is `bundle_state::input`, but the OpenCL program already has a binary associated with it, an exception with the errc::invalid error code is thrown.

Throws an exception with the errc::invalid error code if any error is produced by the SYCL backend. |

---

### C.7.2. Extension query

Platforms and devices with an OpenCL backend may support extensions. For convenience, the extensions supported by a platform or device can be queried through the following functions provided in the `sycl::opencl` namespace.
C.7.3. Reference counting

Most OpenCL objects are reference counted. The SYCL general programming model doesn’t require that native objects are reference counted. However, for convenience, the following function is provided in the `sycl::opencl` namespace.

<table>
<thead>
<tr>
<th>Reference counting</th>
<th>Description</th>
</tr>
</thead>
</table>
| template<typename openCLT>
  cl_uint get_reference_count(
    openCLT obj) | Returns the reference count of the given object |

C.7.4. Errors and limitations

If there is an OpenCL error associated with an exception triggered, then the OpenCL error code can be obtained by the free function `cl_int sycl::opencl::get_error_code(sycl::exception&)`. In the case where there is no OpenCL error associated with the exception triggered, the OpenCL error code will be `CL_SUCCESS`.

C.7.5. Interoperability with kernel bundles

In OpenCL any kernel function that is enqueued over an nd-range is represented by a `cl_kernel` and must be compiled and linked via a `cl_program` using `clBuildProgram`, `clCompileProgram` and `clLinkProgram`.

For OpenCL SYCL backend this detail is abstracted away by kernel bundles and a `kernel_bundle` object containing all SYCL kernel functions is retrieved by calling the free function `get_kernel_bundle`.

The OpenCL SYCL backend specification provides additional free functions which provide convenience functions for constructing kernel bundles from OpenCL specific objects.
1. **Preconditions:** The `context` specified by `ctxt` must be associated with the OpenCL SYCL backend. All devices in `devs` must be associated with `ctxt`. All OpenCL programs in `clPrograms` must be associated with `ctxt`.

**Effects:** Constructs a kernel bundle in the specified `bundle_state` from the provided list of OpenCL programs and associated with the `context` specified by `syclContext` by invoking the necessary OpenCL APIs. Follows the same rules as calling `make_kernel_bundle` on a single OpenCL program, except that the rules apply to all OpenCL programs in `clPrograms`. Multiple programs will be linked together into a single one if required by the requested `State`. The constructed `kernel_bundle` will retain all provided OpenCL programs and will also release them on destruction.

**Throws:** An exception with the `errc::build` error code if any error is produced by invoking the OpenCL APIs.

1. **Preconditions:** The `context` specified by `ctxt` must be associated with the OpenCL SYCL backend. All devices in `devs` must be associated with `ctxt`. All OpenCL kernels in `clKernels` must be associated with `ctxt`.

**Effects:** Constructs an executable kernel bundle from the provided list of OpenCL kernels and associated with the `context` specified by `syclContext` by invoking the necessary OpenCL APIs. `cl_kernel` objects might be associated with different `cl_program` objects, the kernel bundle will encapsulate all of them.

**Throws:** An exception with the `errc::build` error code if any error is produced by invoking the OpenCL APIs.

### C.7.6. Interoperability with kernels

A `kernel_bundle` object contains one or multiple OpenCL programs and one or multiple OpenCL kernels. Calling `kernel_bundle::get_kernel` returns a kernel object which can be invoked by any of kernel invocation commands such as `parallel_for` which take a kernel but not SYCL kernel function.

Calling `make_kernel` must trigger a call to `clRetainKernel` and the resulting `kernel` object must call `clReleaseKernel` on destruction.

It is also possible to construct a `kernel_bundle` from previously created OpenCL `cl_kernel` objects by calling the free function `create_bundle` as described in Section C.7.5.

The kernel arguments for the OpenCL C kernel kernel can either be set prior to creating the `kernel` object or by calling `set_arg` or `set_args` member functions of the `handler` class.

If kernel arguments are set prior to creating the `kernel` object the SYCL runtime is not responsible for managing the data of these arguments.
C.7.7. OpenCL kernel conventions and SYCL

OpenCL and SYCL use opposite conventions for the unit stride dimension. SYCL aligns with C++ conventions, which is important to understand from a performance perspective when porting code to SYCL. The unit stride dimension, at least for data, is implicit in the linearization equations in SYCL (Section 3.11.1) and OpenCL. SYCL aligns with C++ array subscript ordering `arr[a][b][c]`, in that range constructor dimension ordering used to launch a kernel (e.g. `range<3> R(a,b,c)`) and range and ID queries within a kernel, are ordered in the same way as the C++ multi-dimensional subscript operators (unit stride on the right).

When specifying a `range` as the global or local size in a `parallel_for` that invokes an OpenCL interop kernel (through `cl_kernel` interop), the highest dimension of the range in SYCL will map to the lowest dimension within the OpenCL kernel. That statement applies to both an underlying enqueue operation such as `clEnqueueNDRangeKernel` in OpenCL, and also ID and size queries within the OpenCL kernel. For example, a 3D global range specified in SYCL as:

```cpp
range<3> R { r0, r1, r2 };
```

maps to an `clEnqueueNDRangeKernel global_work_size` argument of:

```cpp
size_t cl_interop_range[3] = { r2, r1, r0 };
```

Likewise, a 2D global range specified in SYCL as:

```cpp
range<2> R { r0, r1 };
```

maps to an `clEnqueueNDRangeKernel global_work_size` argument of:

```cpp
size_t cl_interop_range[2] = { r1, r0 };
```

The mapping of highest dimension in SYCL to lowest dimension in OpenCL applies to all operations where a multi-dimensional construct must be mapped, such as when mapping SYCL explicit memory operations to OpenCL APIs like `clEnqueueCopyBufferRect`.

Work-item and work-group ID and range queries have the same reversed convention for unit stride dimension between SYCL and OpenCL. For example, with three, two, or one dimensional SYCL global ranges, OpenCL and SYCL kernel code queries relate to the range as shown in Table 193. The "SYCL kernel query" column applies for SYCL-defined kernels, and the "OpenCL kernel query" column applies for kernels defined through OpenCL interop.

<table>
<thead>
<tr>
<th>SYCL kernel query</th>
<th>OpenCL kernel query</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>With enqueued 3D SYCL global <code>range</code> of <code>range&lt;3&gt; R{r0,r1,r2}</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nd_item::get_global_range(0) / item::get_range(0)</code></td>
<td><code>get_global_size(2)</code></td>
<td><code>r0</code></td>
</tr>
<tr>
<td><code>nd_item::get_global_range(1) / item::get_range(1)</code></td>
<td><code>get_global_size(1)</code></td>
<td><code>r1</code></td>
</tr>
<tr>
<td><code>nd_item::get_global_range(2) / item::get_range(2)</code></td>
<td><code>get_global_size(0)</code></td>
<td><code>r2</code></td>
</tr>
<tr>
<td><code>nd_item::get_global_id(0) / item::get_id(0)</code></td>
<td><code>get_global_id(2)</code></td>
<td>Value in range 0...(r0-1)</td>
</tr>
</tbody>
</table>

Table 193. Example range mapping from SYCL enqueued three dimensional global `range` to OpenCL and SYCL queries
SYCL kernel query & OpenCL kernel query & Returned Value
nd_item::get_global_id(1) / item::get_id(1) & get_global_id(1) & Value in range 0..(r1-1)}
nd_item::get_global_id(2) / item::get_id(2) & get_global_id(0) & Value in range 0..(r2-1)}
With enqueued 2D SYCL global range of range<2> R{r0,r1}
nd_item::get_global_range(0) / item::get_range(0) & get_global_size(1) & r0
nd_item::get_global_range(1) / item::get_range(1) & get_global_size(0) & r1
nd_item::get_global_id(0) / item::get_id(0) & get_global_id(1) & Value in range 0..(r0-1})
nd_item::get_global_id(1) / item::get_id(1) & get_global_id(0) & Value in range 0..(r1-1})
With enqueued 1D SYCL global range of range<1> R{r0}
nd_item::get_global_range(0) / item::get_range(0) & get_global_size(0) & r0
nd_item::get_global_id(0) / item::get_id(0) & get_global_id(0) & Value in range 0..(r0-1})

C.7.8. Data types

The OpenCL C language standard Section 6.11 defines its own built-in scalar data types, and these have additional requirements in terms of size and signedness on top of what is guaranteed by ISO C++. For the purpose of interoperability and portability, SYCL defines a set of aliases to C++ types within the sycl::opencl namespace using the cl_prefix. These aliases are described in Table 194.

Table 194. Scalar data type aliases supported by SYCL OpenCL backend

<table>
<thead>
<tr>
<th>Scalar data type alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_bool</td>
<td>Alias to a conditional data type which can be either true or false. The value true expands to the integer constant 1 and the value false expands to the integer constant 0.</td>
</tr>
<tr>
<td>cl_char</td>
<td>Alias to a signed 8-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_uchar</td>
<td>Alias to an unsigned 8-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_short</td>
<td>Alias to a signed 16-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_ushort</td>
<td>Alias to an unsigned 16-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_int</td>
<td>Alias to a signed 32-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_uint</td>
<td>Alias to an unsigned 32-bit integer, as defined by the C++ core language.</td>
</tr>
</tbody>
</table>
### Scalar data type alias

<table>
<thead>
<tr>
<th>Scalar data type alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cl_long</td>
<td>Alias to a signed 64-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_ulong</td>
<td>Alias to an unsigned 64-bit integer, as defined by the C++ core language.</td>
</tr>
<tr>
<td>cl_float</td>
<td>Alias to a 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.</td>
</tr>
<tr>
<td>cl_double</td>
<td>Alias to a 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.</td>
</tr>
<tr>
<td>cl_half</td>
<td>Alias to a 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. Kernels using this type are only supported on devices that have aspect::fp16, as described in Section 5.7.</td>
</tr>
</tbody>
</table>

### C.8. Preprocessor directives and macros

- **SYCL_BACKEND_OPENCL** substitutes to 1 if the OpenCL SYCL backend is active while building the SYCL application.

### C.8.1. Offline linking with OpenCL C libraries

SYCL supports linking SYCL kernel functions with OpenCL C libraries during offline compilation or during online compilation by the SYCL runtime within a SYCL application.

Linking with OpenCL C kernel functions offline is an optional feature and is unspecified. Linking with OpenCL C kernel functions online is performed by using the SYCL kernel_bundle class to compile and link an OpenCL C source; using the compile_with_source or build_with_source member functions.

OpenCL C functions that are linked with, using either offline or online compilation, must be declared as extern "C" function declarations. The function parameters of these function declarations must be defined as the OpenCL C interoperability aliases; pointer of the multi_ptr class template, vector_t of the vec class template and scalar data type aliases described in Table 194.

### C.9. SYCL support of non-core OpenCL features

In addition to the OpenCL core features, SYCL also provides support for OpenCL extensions which provide features in OpenCL via khr extensions.

Some extensions are natively supported within the SYCL interface, however some can only be used via the OpenCL interoperability interface. The SYCL interface required for native extensions must be available. However, if the respective extension is not supported by the executing SYCL device, the SYCL runtime must throw an exception with the errc::feature_not_supported or errc::kernel_not_supported error codes.

The OpenCL backend exposes some khr extensions to SYCL applications through the sycl::aspect enumerated type. Therefore, applications can query for the existence of these khr extensions by calling the device::has() or platform::has() member functions.

All OpenCL extensions are available through the OpenCL interoperability interface, but some can also be used through core SYCL APIs. Table 195 shows which these are. Table 195 also shows the mapping from each OpenCL extension name to its associated SYCL device aspect when one is available.

**Table 195. SYCL support for OpenCL 1.2 extensions**
C.9.1. Half precision floating-point

The half scalar data type: `half` and the half vector data types: `half1`, `half2`, `half3`, `half4`, `half8` and `half16` must be available at compile-time. However, a kernel using these types is only supported on devices that have `aspect::fp16`, as described in Section 5.7.

The conversion rules for half precision types follow the same rules as in the OpenCL 1.2 extensions specification par. 9.5.1.

The math functions for half precision types follow the same rules as in the OpenCL 1.2 extensions specification par. 9.5.2, 9.5.3, 9.5.4, 9.5.5. The allowed error in ULP(Unit in the Last Place) is less than 8192, corresponding to Table 6.9 of the OpenCL 1.2 specification.

C.9.2. Writing to 3D image memory objects

The `unsampled_image_accessor` class in SYCL supports member functions for writing 3D image memory objects, but this functionality is only allowed on a device if the extension `cl_khr_3d_image_writes` is supported on that device.

C.9.3. Interoperability with OpenGL

Interoperability between SYCL and OpenGL is not directly provided by the SYCL interface, however can be achieved via the SYCL OpenCL interoperability interface.

C.10. Correspondence of some OpenCL features to SYCL

This section describes the correspondence between some OpenCL features and features in the core SYCL specification that provide similar functionality. All content in this section is non-normative.

C.10.1. Work-item functions

The OpenCL 1.2 specification document ch. 6.12.1 in Table 6.7 defines work-item functions that tell various information about the currently executing work item in an OpenCL kernel. SYCL provides equivalent functionality through the item and group classes that are defined in Section 4.9.1.4, Section 4.9.1.5 and Section 4.9.1.7.

C.10.2. Vector data load and store functions

The functionality from the OpenCL functions as defined in the OpenCL 1.2 specification document par. 6.12.7 is available in SYCL through the `vec` class in Section 4.14.2.
C.10.3. Synchronization functions

In SYCL the OpenCL synchronization functions are available through the `nd_item` class (Section 4.9.1.5), as they are applied to work-items for local or global address spaces. Please see Table 119.

C.10.4. `printf` function

The functionality of the `printf` function is covered by the `stream` class (Section 4.16), which has the capability to print to standard output all of the SYCL classes and primitives, and covers the capabilities defined in the OpenCL 1.2 specification document par. 6.12.13.
Appendix D: What has changed from previous versions

D.1. What has changed from SYCL 1.2.1 to SYCL 2020

The SYCL runtime moved from namespace `cl::sycl` provided by `#include <CL/sycl.hpp>` to namespace `sycl` provided by `#include <sycl/sycl.hpp>` as explained in Section 4.3. The old header file is still available for compatibility with SYCL 1.2.1 applications.

The SYCL specification is now based on the core language of C++17, as described in Section 3.9.1. Features of C++17 are now enabled within the specification, such as deduction guides for class template argument deduction.

Naming of lambda functions passed to kernel invocations is now optional.

Changes to buffers, images and accessors:

- The `image` class has been removed. There are now new classes `unsampled_image` and `sampled_image` which represent sampled and unsampled images. The `sampler` class has been removed and replaced with the new `image_sampler` structure.
- Support for image arrays has been removed.
- The type name `access::target` has been deprecated and replaced with the type `target`.
- The type name `access::mode` has been deprecated and replaced with the type `access_mode`.
- The name of the accessor target `target::global_buffer` has been deprecated and replaced with `target::device`.
- Support for the accessor target `target::host_buffer` has been deprecated. There is now a new accessor class `host_accessor` which provides equivalent functionality.
- The `buffer` member functions which return an accessor of type `target::host_buffer` have been deprecated. A new member function `get_host_access()` has been added which returns a `host_accessor`.
- The `buffer` class has a new variadic overload of the `get_access()` member function which allows construction of an accessor with various parameters.
- Support for the accessor target `target::local` has been deprecated. There is now a new accessor class `local_accessor` which provides equivalent functionality.
- Support for the accessor targets `target::image` and `target::host_image` have been removed. There are now new accessor classes for sampled and unsampled images: `sampled_image_accessor`, `host_sampled_image_accessor`, `unsampled_image_accessor` and `host_unsampled_image_accessor`.
- A new accessor target `target::host_task` has been added, which allows access to a `buffer` from a `host_task`.
- Support for the accessor modes `access_mode::discard_write` and `access_mode::discard_read_write` has been deprecated. Accessors can now be constructed with a property list, and the new property `property::no_init` provides equivalent functionality.
- Support for the accessor mode `access_mode::atomic` and the member functions that return an instance of the `atomic` class have been deprecated in favor of using the new `atomic_ref` class instead.
- Support for the accessor template parameter `isPlaceholder` has been deprecated, and the value of this parameter no longer has any bearing on whether the accessor is a placeholder. The enumerated type `access::placeholder` is also deprecated. A placeholder accessor can now be constructed by calling the appropriate constructor, without regard to the template parameter.
- The return type of `accessor::is_placeholder()` is no longer `constexpr`. 
The member function `handler::require()` may now be called on any `accessor` with target `target::device`, `target::constant_buffer` or `target::host_task`, regardless of whether it is a placeholder.

New `accessor` constructors have been added which take a type tag parameter, which allows the class template parameters to be inferred via C++ class template argument deduction (CTAD).

The `buffer` member function `get_access()` now has a default value for the `target` template parameter, so it is no longer necessary to provide any template parameters in order to get a `access_mode::read_write` accessor.

The `accessor` template parameters `Dimensions` and `AccessMode` now have default values, so the only required template parameter is `DataT`. Moreover, the default access mode is either `access_mode::read_write` or `access_mode::read`, depending on the constness of the `DataT` type. This makes it possible to declare a read-only accessor by simply using a `const` qualified type.

Implicit conversions have been added between the two forms of read-only accessor (one form has `const DataT` and `access_mode::read` and the other has non-const `DataT` and `access_mode::read`). There is also an implicit conversion from a read-write accessor to either of the read-only forms.

Member functions of `accessor` which return a reference to an element have been changed to return a `const` reference for read-only accessors. The `get_pointer()` member function has also been changed to return a `const` pointer for read-only accessors. The `value_type` and `reference` member types of `accessor` have been changed to be `const` types for read-only accessors.

The `accessor` member function `get_pointer()` now returns a raw pointer. The `get_multi_ptr()` member function was introduced which returns the `multi_ptr` class to the appropriate space.

The `accessor` class now meets the C++ requirement of `ReversibleContainer`. This includes (but is not limited to) returning `begin` and `end` iterators, specifying a default constructible accessor that can be passed to a kernel but not dereferenced, and making them equality comparable.

Many of the `accessor` member functions have been marked `noexcept`.

A `ranged accessor` is no longer allowed to read elements that are outside of its range; attempting to do so produces undefined behavior.

The semantics of the subscript operator have been changed for a `ranged accessor` which has an offset. Calling `operator[](0)` now returns a reference to the first element in the range, rather than a reference to the first element in the underlying buffer.

The behavior of buffers and accessors with a zero-sized range has been clarified.

Constant memory no longer appears in the SYCL device memory model in SYCL 2020.

The C++ attributes that decorate kernels are now better described, and their position has changed so that they are applied directly to the kernel function. (Previously, they were applied to a device function that the kernel calls, and the implementation needed to propagate the information up to the enclosing kernel.) The old C++ attribute form is no longer included in the SYCL specification.

The list of built-in integer math functions was extended with `ctz()` in Table 179. Specification of `clz()` was extended with the case of argument is 0.

The classes `vector_class`, `string_class`, `function_class`, `mutex_class`, `shared_ptr_class`, `weak_ptr_class`, `hash_class` and `exception_ptr_class` have been removed from the API and the standard classes `std::vector`, `std::string`, `std::function`, `std::mutex`, `std::shared_ptr`, `std::weak_ptr`, `std::hash` and `std::exception_ptr` are used instead.

The specific `sycl::buffer` API taking `std::unique_ptr` has been removed. The behavior is the same as in SYCL 1.2.1 but with a simplified API. Since there is still the API taking `std::shared_ptr` and there is an implicit conversion from a `std::unique_ptr` prvalue to a `std::shared_ptr`, the API can still be used as before with a `std::unique_ptr` to give away memory ownership.

Offsets to `parallel_for`, `nd_range`, `nd_item` and `item` classes have been deprecated. As such, the parallel iteration spaces all begin at `(0,0,0)` and developers are now required to handle any offset arithmetic
themselves. The behavior of \texttt{nd\_item.get\_global\_linear\_id()} and \texttt{nd\_item.get\_local\_linear\_id()} has been clarified accordingly.

Unified Shared Memory (USM), in Section 4.8, has been added as a pointer-based strategy for data management. It defines several types of allocations with various accessibility rules for host and devices. USM is meant to complement buffers, not replace them.

The \texttt{queue} class received a new \texttt{property} that requires in-order semantics for a queue where operations are executed in the order in which they are submitted.

The \texttt{queue} class received several new member functions to invoke kernels directly on a queue objects instead of inside a command group handler in the \texttt{submit} member function.

The \texttt{queue} constructor overloads that accept both a \texttt{context} and a \texttt{device} parameter have been broadened to allow the device to be either a device that is in the context or a descendant device of a device that is in the context.

The \texttt{program} class has been removed and replaced with a new class \texttt{kernel\_bundle}, which provides similar functionality in a type-safe and thread-safe way. The \texttt{kernel} class has changed, and some member functions have been removed.

Support has been added for \texttt{specialization-constants}, which allow a SYCL kernel function to use constant variables whose values aren't known until the kernel is invoked. A SYCL kernel function can now take an optional parameter of type \texttt{kernel\_handler}, which allows the kernel to read the values of \texttt{specialization-constants}.

The constructors for SYCL \texttt{context} and \texttt{queue} are made \texttt{explicit} to prevent ambiguities in the selected constructor resulting from implicit type conversion.

The requirement for C++ standard layout for data shared between host and devices has been relaxed. SYCL now requires data shared between host and devices to be \texttt{device\_copyable} as defined Section 3.13.1.

The concept of a \texttt{group} of \texttt{work\_items} was generalized to include \texttt{work\_groups} and \texttt{sub\_groups}. A \texttt{work\_group} is represented by the \texttt{sycl::group} class as in SYCL 1.2.1, and a \texttt{sub\_group} is represented by the new \texttt{sycl::sub\_group} class.

The \texttt{host\_task} member function for the \texttt{queue} has been introduced for en-queueing \texttt{host\_tasks} on a \texttt{queue} to schedule the SYCL runtime to invoke native C++ functions, conforming to the SYCL memory model. \texttt{Host\_tasks} also support interoperability with the native SYCL \texttt{backend} objects associated at that point in the DAG using the optional \texttt{interop\_handle} class.

A library of algorithms based on the C++17 algorithms library was introduced in Section 4.17.4. These algorithms provide a simple way for developers to apply common parallel algorithms using the work-items of a group.

The definition of the \texttt{sycl::group} class was modified to support the new group functions in Section 4.17.3. New member types and variables were added to enable generic programming, and member functions were updated to encapsulate all functionality tied to \texttt{work\_groups} in the \texttt{sycl::group} class. See Table 121 for details.

The \texttt{barrier} and \texttt{mem\_fence} member functions of the \texttt{nd\_item} class have been removed. The \texttt{barrier} member function has been replaced by the \texttt{group\_barrier()} function, which can be used to synchronize either \texttt{work\_groups} or \texttt{sub\_groups}. The \texttt{mem\_fence} member function has been replaced by the \texttt{atomic\_fence} function, which is more closely aligned with \texttt{std::atomic\_thread\_fence} and offers control over memory ordering and scope.

Changes in the SYCL \texttt{vec} class described in Section 4.14.2:
• `operator[]` was added;
• unary `operator+()` and `operator-()` were added;

The device selection now relies on a simpler API based on ranking functions used as device selectors described in Section 4.6.1.1.

A new device selector utility has been added to Section 4.6.1.1, the `aspect_selector`, which returns a selector object that only selects devices that have all the requested aspects.

The device query `info::fp_config::correctly_rounded_divide_sqrt` has been deprecated.

A new reduction library consisting of the `reduction` function and `reducer` class was introduced to simplify the expression of variables with reduction semantics in SYCL kernels. See Section 4.9.2.

The `atomic` class from SYCL 1.2.1 was deprecated in favor of a new `atomic_ref` interface.

A new reduction library consisting of the `reduction` function and `reducer` class was introduced to simplify the expression of variables with reduction semantics in SYCL kernels. See Section 4.9.2.

The SYCL exception class hierarchy has been condensed into a single exception type: `exception` now derives from `std::exception`. The variety of errors are now provided via error codes, which aligns with the C++ error code mechanism.

The new error code mechanism now also generalizes the previous `get_cl_code` interface to provide a generic interface way for querying backend-specific error codes.

Default asynchronous error handling behavior is now defined, so that asynchronous errors will cause abnormal program termination even if a user-defined asynchronous handler function is not defined. This prevents asynchronous errors from being silently lost during early stages of application development.

Kernel invocation functions, such as `parallel_for`, now take kernel functions by `const` reference. Kernel functions must now have a `const`-qualified `operator()`, and are allowed to be copied zero or more times by an implementation. These clarifications allow implementations to have flexibility for specific devices, and define what users should expect with kernel functors. Specifically, kernel functors can not be marked as `mutable`, and sharing of data between work-items should not be attempted through state stored within a kernel functor.

A new concept called device `aspects` has been added, which tells the set of optional features a device supports. This new mechanism replaces the `has_extension()` function and some uses of `get_info()`.

There is a new Chapter 6 which describes how extensions to the SYCL language can be added by vendors and by the Khronos Group.

A `queue` constructor has been added that takes both a `device` and `context`, to simplify interfacing with libraries.

The `parallel_for` interface has been simplified in some forms to accept a braced initializer list in place of a `range`, and to always take `item` arguments. Kernel invocation functions have also been modified to accept generic lambda expressions. Implicit conversions from one-dimensional `item` and one-dimensional `id` to scalar types have been defined. All of these modifications lead to simpler SYCL code in common use cases.

The behaviour of executing a kernel over a `range` or `nd_range` with index space of zero has been clarified.

Some device-specific queries have been renamed to more clearly be “device-specific kernel” `get_info` queries (`info::kernel_device_specific`) instead of “work-group” (`get_workgroup_info`) and sub-group (`get_sub_group_info`) queries.

A new math array type `marray` has been defined to begin disambiguation of the multiple possible interpretations of how `sycl::vec` should be interpreted and implemented.
Changes in SYCL address spaces:

- the address space meaning has been significantly improved;
- the generic address space was introduced;
- the constant address space was deprecated;
- behavior of unannotated pointer/reference (raw pointer/reference) is now dependent on the compilation mode. The compiler can either interpret unannotated pointer/reference has addressing the generic address space or to be deduced;
- some ambiguities in the address space deduction were clarified. Notably that deduced type does not affect the user-provided type.

Changes in multi_ptr interface:

- addition of access::address_space::generic_space to represent the generic address space;
- deprecation of access::address_space::constant_space;
- an extra template parameter to allow to select a flavor of the multi_ptr interface. There are now 3 different interfaces:
  - interface exposing undecorated types. Returned pointer and reference are not annotated by an address space;
  - interface exposing decorated types. Returned pointer and reference are annotated by an address space;
  - legacy 1.2.1 interface (deprecated);
- deprecation of the 1.2.1 interface;
- deprecation of constant_ptr;
- global_ptr, local_ptr and private_ptr alias take the new extra parameter;
- addition of the address_space_cast free function to cast undecorated pointer to multi_pointer;
- addition of construction/conversion operator for the generic address space;
- removal of the constructor and assignment operator taking an unannotated pointer;
- implicit conversion to a pointer is now deprecated. get should be used instead;
- the return type of the member function get now depends on the selected interface.
- addition of the member function get_raw which returns the underlying pointer as an unannotated pointer;
- addition of the member function get_decorated which returns the underlying pointer as an annotated pointer;
- addition of the subscript operator providing random access.

The cl::sycl::byte has been deprecated and now the C++17 std::byte should be used instead.

A SYCL implementation is no longer required to provide a host device. Instead, an implementation is only required to provide at least one device. Implementations are still allowed to provide devices that are implemented on the host, but it is no longer required. The specification no longer defines any special semantics for a "host device" and APIs specific to the host device have been removed.

The default constructors for the device and platform classes have been changed to construct a copy of the default device and a copy of the platform containing the default device. Previously, they returned a copy of the host device and a copy of the platform containing the host device. The default constructor for the event class has also been changed to construct an event that comes from a default-constructed queue. Previously, it constructed an event that used the host backend.

Explicit copy functions of the handler class have also been introduced to the queue class as shortcuts for
the handler ones. This is enabled by the improved placeholder accessors to help reduce code verbosity in certain cases because the shortcut functions implicitly create a command group and call handler::require.

Information query descriptors have been changed to structures under namespaces named accordingly. param_traits has been removed and the return type of an information query is now contained in the descriptor. The sycl::info::device::max_work_item_sizes is now a template that takes a dimension parameter corresponding to the number of dimensions of the work-item size maxima.

Changes to retrieving size information:

- all get_size() member functions have been deprecated and replaced with byte_size(), which is marked noexcept;
- all get_count() member functions have been deprecated and replaced with size(), which is marked noexcept;
- in the vec class the functions byte_size() and size() are now static member functions;
- in the stream class get_size() has been deprecated in favor of size(), whereas stream::byte_size() is not available;
- accessors for sampled and unsampled images only define size() and not byte_size().

The device descriptors info::device::max_constant_buffer_size and info::device::max_constant_args are deprecated in SYCL 2020.

The buffer_allocator is now templated on the data type and follows the C++ named requirement Allocator.

The SYCL id and range have now unary + and - operations, prefix ++ and -- operations, postfix ++ and -- operations which were forgotten in SYCL 1.2.1.

In SYCL 1.2.1, the handler::copy() overload with two accessor parameters did not clearly specify which accessor's size determines the amount of memory that is copied. The spec now clarifies that the src accessor's size is used.
Appendix E: References


accessor
An accessor is a class which allows a command to access data managed by a buffer or image class or allows a SYCL kernel function to access local memory on a device. Accessors are also used to express the dependencies among the different command groups. For the full description please refer to Section 4.7.6

application scope
The application scope starts with the construction first SYCL runtime class object and finishes with the destruction of the last one. Application refers to the C++ SYCL application and not the SYCL runtime.

aspect
A characteristic of a device which determines whether it supports some optional feature. Aspects are always boolean, so a device either has or does not have an aspect.

asynchronous error
A SYCL asynchronous error is an error occurring after the host API call invoking the error causing action has returned, such that the error cannot be thrown as a typical C++ exception from a host API call. Such errors are typically generated from device kernel invocations which are executed when SYCL task graph dependencies are satisfied, which occur asynchronously from host code execution. For the full description and associated asynchronous error handling mechanisms, please refer to Section 4.13.

async_handler
An asynchronous error handler object is a function class instance providing necessary code for handling all the asynchronous errors triggered from the execution of command groups on a queue, within a context or an associated event. For the full description please refer to Section 4.13.2.

barrier
A barrier is either a command queue barrier, or a kernel execution group barrier depending on whether it is a synchronization point on the command queue or on a group of work-items in a kernel execution.

blocking accessor
A blocking accessor is an accessor which provides immediate access and continues to provide access until it is destroyed. For the full description please refer to Section 4.7.6

buffer
The buffer class manages data for the SYCL C++ host application and the SYCL device kernels. The buffer class may acquire ownership of some host pointers passed to its constructors according to the constructor kind.

The buffer class, together with the accessor class, is responsible for tracking memory transfers and guaranteeing data consistency among the different kernels. The SYCL runtime manages the memory allocations on both the host and the device within the lifetime of the buffer object. For the full description please refer to Section 4.7.2.

bundle state
A SYCL bundle state represents the state of a kernel bundle and therefore its capabilities in the SYCL programming API. Possible states are input, object or executable.

command
A request to execute work that is submitted to a queue such as the invocation of a SYCL kernel function, the invocation of a host task or an asynchronous copy.
command group

In SYCL, the operations required to process data on a device are represented using a command group function object. Each command group function object is given a unique command group handler object to perform all the necessary work required to correctly process data on a device using a kernel. In this way, the group of commands for transferring and processing data is enqueued as a command group on a device for execution. A command group is submitted atomically to a SYCL queue.

command group function object

A type which is callable with operator() that takes a reference to a command group handler, that defines a command group which can be submitted by a queue. The function object can be a named type, lambda function or std::function.

command group handler

The command group handler class provides the interface for the commands that can be executed inside the command group scope. It is provided as a scoped object to all of the data access requests within the command group scope. For the full description please refer to Section 4.9.4.

command group scope

The command group scope is the function scope defined by the command group function object. The command group command group handler object lifetime is restricted to the command group scope. For more details see Section 4.9.3.

command queue barrier

The SYCL API provides two variants for functions that force synchronization on a SYCL command queue. The sycl::queue::wait() and sycl::queue::wait_and_throw() functions force the SYCL command queue to wait for the execution of the command group function object before it is able to continue executing.

constant memory

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

context

A context represents the runtime data structures and state required by a SYCL backend API to interact with a group of devices associated with a platform. The context is defined as the sycl::context class, for further details please see Section 4.6.3.

control flow

When all work-items in a group are executing the same sequence of statements, they are said to be executing under converged control flow. Control flow diverges when different work-items in a group execute a different sequence of statements, typically as a result of evaluating conditions differently (e.g. in selection statements or loops).

core SYCL specification

The text of the SYCL language specification (this document), excluding the text of any backend specifications and excluding the text for any extensions.

descendent device

The descendent devices of device $D$ include all of the sub-devices of $D$, all of the sub-devices of those devices, etc.

device

A SYCL device is an abstraction of a piece of hardware that can execute SYCL kernels.

device compiler

A SYCL device compiler is a compiler that produces device binaries from a valid SYCL application. For
the full description please refer to Chapter 5.

device copyable
Data that is shared between the host and the devices must generally have a type that abides by the restrictions listed in Section 3.13.1 for a device copyable type.

device function
A device function is any function in a SYCL application that can be run on a device. This includes SYCL kernel functions and, recursively, functions they call.

device image
A device image is a representation of one or more kernels in an implementation-defined format. A device image could be a compiled version of the kernels in an intermediate language representation which needs to be translated at runtime into a form that can be invoked on a device, it could be a compiled version of the kernels in a native code format that is ready to be invoked without further translation, or it could be a source code representation which needs to be compiled before it can be invoked. Other representations are possible too.

device selector
A way to select a device used in various places. This is a callable object taking a device reference and returning an integer rank. One of the device with the highest non-negative value is selected. See Section 4.6.1.1 for more details.

event
A SYCL object that represents the status of an operation that is being executed by the SYCL runtime.

executable
A state which a kernel bundle can be in, representing SYCL kernel functions as an executable.

generic memory
Generic memory is a virtual memory region which can represent global memory, local memory and private memory region.

global id
As in OpenCL, a global ID is used to uniquely identify a work-item and is derived from the number of global work items specified when executing a kernel. A global ID is a one, two or three-dimensional value that starts at 0 per dimension.

global memory
Global memory is a memory region accessible to all work items executing on a device.

group
A group of work-items within the index space of a SYCL kernel execution, such as a work-group or sub-group.

group barrier
A synchronization function within a group of work items. All the work items of a group must execute the barrier construct before any work-item continues execution beyond the barrier. Additionally all work-items in the group execute a release mem-fence prior to synchronizing at the barrier, all work-items in the group execute an acquire mem-fence after synchronizing at the barrier, and there is an implicit synchronization between these acquire and release fences as if through an atomic operation on an atomic object internal to the barrier implementation.

h-item
A unique identifier representing a single work-item within the index space of a SYCL kernel hierarchical execution. Can be one, two or three dimensional. In the SYCL interface a h-item is represented by
the \texttt{h\_item} class (see Section 4.9.1.6).

**host**
Host is the system that executes the C++ application including the SYCL API.

**host pointer**
A pointer to memory on the host. Cannot be accessed directly from a \texttt{device}.

**host task**
A \texttt{command} which invokes a native C++ callable, scheduled conforming to SYCL dependency rules.

**host task command**
A type of command that can be used inside a \texttt{command group} in order to schedule a native C++ function.

**id**
It is a unique identifier of an item in an index space. It can be one, two or three dimensional index space, since the SYCL kernel execution model is an \texttt{nd-range}. It is one of the index space classes. For the full description please refer to Section 4.9.1.3.

**image**
Images in SYCL, like buffers, are abstractions of multidimensional structured arrays. Image can refer to \texttt{unsampled\_image} and \texttt{sampled\_image}. For the full description please refer to Section 4.7.3.

**implementation-defined**
Behavior that is explicitly allowed to vary between conforming implementations of SYCL. A SYCL implementer is required to document the implementation-defined behavior.

**index space classes**
Like in OpenCL, the kernel execution model defines an \texttt{nd-range} index space. The \texttt{SYCL runtime} class that defines an \texttt{nd-range} is the \texttt{cl::nd\_range}, which takes as input the sizes of global and local work-items, represented using the \texttt{cl::range} class. The kernel library classes for indexing in the defined \texttt{nd-range} are the following classes:

- \texttt{cl::id}: The basic index class representing an \texttt{id};
- \texttt{cl::item}: The \texttt{item} index class that contains the \texttt{global id} and \texttt{local id};
- \texttt{cl::nd\_item}: The \texttt{nd\_item} index class that contains the \texttt{global id}, \texttt{local id} and the \texttt{work\_group id};
- \texttt{cl::group}: The \texttt{group} class that contains the \texttt{work\_group id} and the member functions on a \texttt{work\_group}.

**input**
A state which a \texttt{kernel bundle} can be in, representing \texttt{SYCL kernel functions} as a source or intermediate representation

**item**
An item id is an interface used to retrieve the \texttt{global id}, \texttt{work\_group id} and \texttt{local id}. For further details see Section 4.9.1.4.

**kernel**
A kernel represents a \texttt{SYCL kernel function} that has been compiled for a device, including all of the \texttt{device functions} it calls. A kernel is implicitly created when a \texttt{SYCL kernel function} is submitted to a device via a \texttt{kernel invocation command}. However, a kernel can also be created manually by pre-compiling a \texttt{kernel bundle} (see Section 4.11).

**kernel bundle**
A kernel bundle is a collection of \texttt{device images} that are associated with the same \texttt{context} and with a
set of devices. Kernel bundles have one of three states: input, object or executable. Kernel bundles in the executable state are ready to be invoked on a device, whereas bundles in the other states need to be translated into the executable state before they can be invoked.

**kernel handler**
A representation of a SYCL kernel function being invoked that is available to the kernel scope.

**kernel invocation command**
A type of command that can be used inside a command group in order to schedule a SYCL kernel function, includes single_task, all variants of parallel_for and parallel_for_workgroup.

**kernel name**
A kernel name is a class type that is used to assign a name to the kernel function, used to link the host system with the kernel object output by the device compiler. For details on naming kernels please see Section 5.2.

**kernel scope**
The function scope of the operator() on a SYCL kernel function. Note that any function or member function called from the kernel is also compiled in kernel scope. The kernel scope allows C++ language extensions as well as restrictions to reflect the capabilities of devices. The extensions and restrictions are defined in the SYCL device compiler specification.

**local id**
A unique identifier of a work-item among other work-items of a work-group.

**local memory**
Local memory is a memory region associated with a work-group and accessible only by work-items in that work-group.

**native backend object**
An opaque object defined by a specific backend that represents a high-level SYCL object on said backend. There is no guarantee of having native backend objects for all SYCL types.

**native-specialization constant**
A specialization constant in a device image whose value can be used by an online compiler as an immediate value during the compilation.

**nd-item**
A unique identifier representing a single work-item and work-group within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a nd-item is represented by the nd_item class (see Section 4.9.1.5).

**nd-range**
A representation of the index space of a SYCL kernel execution, the distribution of work items within work groups. Contains a range specifying the number of global work items, a range specifying the number of local work items and a id specifying the global offset. Can be one, two or three dimensional. The minimum size of range within the nd-range is 0 per dimension; where any dimension is set to zero, the index space in all dimensions will be zero. In the SYCL interface an nd-range is represented by the nd_range class (see Section 4.9.1.2).

**mem-fence**
A memory fence provides control over re-ordering of memory load and store operations when coupled with an atomic operation that synchronizes two fences with each other (or when the fences are part of a group barrier in which case there is implicit synchronization as if an atomic operation has synchronized the fences). The sycl::atomic_fence function acts as a fence across all work-items and devices specified by a memory_scope argument.
object
A state which a kernel bundle can be in, representing SYCL kernel functions as a non-executable object.

platform
A collection of devices managed by a single backend.

private memory
A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item. The sycl::private_memory class provides access to the work-item's private memory for the hierarchical API as it is described at Listing 1.

queue
A SYCL command queue is an object that holds command groups to be executed on a SYCL device. SYCL provides a heterogeneous platform integration using device queue, which is the minimum requirement for a SYCL application to run on a SYCL device. For the full description please refer to Section 4.6.5.

range
A representation of a number of work items or work-group within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a work-group is represented by the group class (see Section 4.9.1.7).

ranged accessor
A ranged accessor is a host or buffer accessor that was constructed with a non-zero offset into the data buffer or with an access range smaller than the range of the data buffer, or both. Please refer to Section 4.7.6.8 for more info.

reduction
An operation that produces a single value by combining multiple values in an unspecified order using a binary operator. If the operator is non-associative or non-commutative, the behavior of a reduction may be non-deterministic.

root device
A device that is not a sub-device. The function device::get_devices() returns a vector of all the root devices.

rule of five
For a given class, if at least one of the copy constructor, move constructor, copy assignment operator, move assignment operator or destructor is explicitly declared, all of them should be explicitly declared.

rule of zero
For a given class, if the copy constructor, move constructor, copy assignment operator, move assignment operator and destructor would all be inlined, public and defaulted, none of them should be explicitly declared.

SMCP
The single-source multiple compiler-passes (SMCP) technique allows a single-source file to be parsed by multiple compilers for building native programs per compilation target. For example, a standard C++ CPU compiler for targeting host will parse the SYCL file to create the C++ SYCL application which offloads parts of the computation to other devices. A SYCL device compiler will parse the same source file and target only SYCL kernels. For the full description please refer to Section 3.12.1. See SSCP for another approach.
specialization constant
A constant variable where the value is not known until compilation of the SYCL kernel function.

specialization id
An identifier which represents a reference to a specialization constant both in the SYCL application for setting the value prior to the compilation of a kernel bundle and in a SYCL kernel function for retrieving the value during invocation.

SSCP
The single-source single compiler-pass (SSCP) technique allows a single-source file to be parsed only once by a single compiler. For example, the SYCL compiler will parse the SYCL file once. Then, from this single intermediate representation, for each kind of device architecture a compilation flow will generate the binary for each kernel and another compilation flow will generate the host code of the C++ SYCL application. For the full description please refer to Section 3.12.2. See SMCP for another approach.

string kernel name
The name of a SYCL kernel function in string form, this can be the name of a kernel function created via interop or a string form of a type kernel name.

sub-group
The SYCL sub-group (sycl::sub_group class) is a representation of a collection of related work-items within a work-group. For further details for the sycl::sub_group class see Section 4.9.1.8.

sub-group barrier
A group barrier for all work items in a sub-group.

sub-group mem-fence
A mem-fence for all work items in a sub-group.

SYCL application
A SYCL application is a C++ application which uses the SYCL programming model in order to execute kernels on devices.

SYCL backend
An implementation of the SYCL programming model using an heterogeneous programming API. A SYCL backend exposes one or multiple SYCL platforms. For example, the OpenCL backend, via the ICD loader, can expose multiple OpenCL platforms.

SYCL backend API
The exposed API for writing SYCL code against a given SYCL backend.

SYCL C++ template library
The template library is a set of C++ templated classes which provide the programming interface to the SYCL developer.

SYCL file
A SYCL C++ source file that contains SYCL API calls.

SYCL kernel function
A type which is callable with operator() that takes an id, item, nd-item or work-group, and an optional kernel_handler as its last parameter. This type can be passed to kernel enqueue member functions of the command group handler. A SYCL kernel function defines an entry point to a kernel. The function object can be a named device copyable type or lambda function.
SYCL runtime

A SYCL runtime is an implementation of the SYCL API specification. The SYCL runtime manages the different platforms, devices, contexts as well as memory handling of data between host and SYCL backend contexts to enable semantically correct execution of SYCL programs.

type kernel name

The name of a SYCL kernel function in type form, this can be either a kernel name provided to a kernel invocation command or the type of a function object use as a SYCL kernel function.

USM

Unified Shared Memory (USM) provides a pointer-based alternative to the buffer programming model. USM enables:

- easier integration into existing code bases by representing allocations as pointers rather than buffers, with full support for pointer arithmetic into allocations;
- fine-grain control over ownership and accessibility of allocations, to optimally choose between performance and programmer convenience;
- a simpler programming model, by automatically migrating some allocations between SYCL devices and the host.

See Section 4.8

work-group

The SYCL work-group (sycl::group class) is a representation of a collection of related work items that execute on a single compute unit. The work items in the group execute the same kernel-instance and share local memory and work-group functions. For further details for the sycl::group class see Section 4.9.1.7.

work-group barrier

A group barrier for all work items in a work-group.

work-group mem-fence

A mem-fence for all work items in a work-group.

work-group id

As in OpenCL, SYCL kernels execute in work groups. The group ID is the ID of the work-group that a work-item is executing within. A group ID is an one, two or three dimensional value that starts at 0 per dimension.

work-group range

A group range is the size of the work-group for every dimension.

work-item

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