SYCL™ Specification

SYCL™ integrates OpenCL™ devices with modern C++

Version 1.2.1

Document Revision: 6

Revision Date: May 14, 2019

Git revision: heads/travis-0-g4f9a12d-dirty

Khronos® OpenCL™ Working Group — SYCL™ subgroup

Editors: Ronan Keryell, Maria Rovatsou & Lee Howes

Contents

List of Tables

List of Figures

Acknowledgements

Editors

- Maria Rovatsou, Codeplay
- Lee Howes, Qualcomm
- Ronan Keryell, Xilinx (current)

Contributors

- Eric Berdahl, Adobe
- Shivani Gupta, Adobe
- David Neto, Altera
- Brian Sumner, AMD
- Anastasia Stulova, ARM
- Balázs Keszthelyi, Broadcom
- Gordon Brown, Codeplay
- Morris Hafner, Codeplay
- Alexander Johnston, Codeplay
- Marios Katsigiannis, Codeplay
- Paul Keir, Codeplay
- Victor Lomüller, Codeplay
- Duncan McBain, Codeplay
- Ralph Potter, Codeplay
- Ruyman Reyes, Codeplay
- Andrew Richards, Codeplay
- Maria Rovatsou, Codeplay
- Panagiotis Stratis, Codeplay
- Michael Wong, Codeplay
- Peter Žužek, Codeplay
- Matt Newport, EA
- Alexey Bader, Intel
- James Brodman, Intel
- Allen Hux, Intel
- Michael Kinsner, Intel
- Sergey Semenov, Intel
- Kathleen Mattson, Miller & Mattson, LLC
- Dave Miller, Miller & Mattson, LLC
- Lee Howes, Qualcomm
- Chu-Cheow Lim, Qualcomm
- Jack Liu, Qualcomm
- Dave Airlie, Red Hat
- Paul Preney, University of Windsor
- Andrew Gozillon, Xilinx
- Ronan Keryell, Xilinx
- Lin-Ya Yu, Xilinx

Introduction

SYCL (pronounced "sickle") is a royalty-free, cross-platform abstraction C++ programming model for OpenCL. SYCL builds on the underlying concepts, portability and efficiency of OpenCL while adding much of the ease of use and flexibility of single-source C++. Developers using SYCL are able to write standard C++ code, with many of the techniques they are accustomed to, such as inheritance and templating. At the same time developers have access to the full range of capabilities of OpenCL both through the features of the SYCL libraries and, where necessary, through interoperation with code written directly to the OpenCL APIs.

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

Simplicity For novice programmers, 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. 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 OpenCL 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.

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.

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 an OpenCL 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.

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 OpenCL, a standard C++ compiler can compile the SYCL programs and

they will run correctly on 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 retains the execution model, runtime feature set and device capabilities of the underlying OpenCL standard. The OpenCL C specification 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.

The use of C++ features such as templates and inheritance on top of the OpenCL 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 libraries that build simple, yet efficient, interfaces to offer more developers access to OpenCL capabilities and devices. SYCL is intended to serve as a foundation for innovation in programming models for heterogeneous systems, that builds on an open and widely implemented standard foundation in the form of OpenCL.

To reduce programming effort and increase the flexibility with which developers can write code, SYCL extends the underlying OpenCL model in two ways beyond the general use of C++ features:

- The hierarchical parallelism syntax offers a way of expressing the data-parallel OpenCL execution model in an easy-to-understand 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.

- 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 programming having to explicitly move data. This approach enables the data-parallel task-graphs that are already part of the OpenCL execution model to be built up easily and safely by SYCL programmers.

To summarize, SYCL enables OpenCL kernels to be written inside C++ source files. This means that software developers can develop and use generic algorithms and data structures using standard C++ template techniques, while still supporting the multi-platform, multi-device heterogeneous execution of OpenCL. 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 heterogeneous processing innovation.

SYCL Architecture

This chapter builds on the structure of the OpenCL specification's architecture chapter to explain how SYCL overlays the OpenCL specification and inherits its capabilities and restrictions as well as the additional features it provides on top of OpenCL 1.2.

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 OpenCL device or on the host.

The terminology used for SYCL inherits that of OpenCL with some SYCL-specific additions. A function object that can execute on either an OpenCL *device* or a host device is called a SYCL kernel function.

To ensure maximum backward-compatibility, a software developer can produce a program that mixes standard OpenCL C kernels and OpenCL API code with SYCL code and expect fully compatible interoperation.

The target users of SYCL are C++ programmers who want all the performance and portability features of 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 OpenCL standard. 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 applications developers who want to separate low-level highly-tuned algorithms or data structures that work on heterogeneous systems from higher-level software development. OpenCL developers can produce templated algorithms that are easily usable by developers in other fields.

Anatomy of a SYCL application

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

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  int main() {
5    using namespace cl::sycl;
6
7    int data[1024];  // Allocate data to be worked on
8
9    // By sticking all the SYCL work in a {} block, we ensure
10   // all SYCL tasks must complete before exiting the block,
11   // because the destructor of resultBuf will wait.
12   {
13     // Create a queue to enqueue work to
14     queue myQueue;
15
16     // Wrap our data variable in a buffer
17     buffer<int, 1> resultBuf { data, range<1> { 1024 } };
18
19     // Create a command_group to issue commands to the queue
20     myQueue.submit([&](handler& cgh) {
21       // request access to the buffer
22       auto writeResult = resultBuf.get_access<access::mode::discard_write>(cgh);
23
24       // Enqueue a parallel_for task
25       cgh.parallel_for<class simple_test>(range<1> { 1024 }, [=](id<1> idx) {
26         writeResult[idx] = idx[0];
27       });  // End of the kernel function
28     });    // End of our commands for this queue
29   }        // End of scope, so we wait for work producing resultBuf to complete
30
31   // Print result
32   for (int i = 0; i < 1024; i++)
33     std::cout << "data[" << i << "] = " << data[i] << std::endl;
34
35   return 0;
36 }
```

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 (see Section 3.3). 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 27. The command group scope specifies a unit of work which will comprise 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 3.6.9.

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, by the `operator()` function of a function object or by the binary `cl_kernel` entity generated from an OpenCL C string. 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` function is templated with a class, in this case called `class simple_test`. This class is used only as a name to enable the kernel (compiled with a device compiler) and the host code (possibly compiled with a different host compiler) to be linked. This is required because C++ lambda functions have no name that a linker could use to link the kernel to the host code.

The `parallel_for` method creates an instance of a kernel object. The kernel object is the entity that will be enqueued within a command group. In the case of `parallel_for` the *kernel function* will be executed over the given range from 0 to 1023. The different methods to execute kernels can be found in Section 4.8.5.

A *kernel function* can only be defined within a *command group scope*, and a *command group scope* may include only a single *kernel function*. Command group scope is the syntactic scope wrapped by the construction of a command group function object as seen on line 20. The command group function object takes as a parameter a command group `handler` which is a runtime constructed object. All the requirements for a kernel to execute are defined in this *command group scope*, as described in Section 3.4.1. In this case the constructor used for `myQueue` on line 14 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 *kernel function* must be contained within a *buffer* or *image*, as described in Section 3.5. We construct a buffer on line 17. Access to the *buffer* is controlled via an accessor which is constructed on line 22 through the `get_access` method of the buffer. 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 *kernel functions*. In this example the *accessor* is used to write to the data buffer on line 26. All *buffers* must be constructed in the application-scope, whereas all *accessors* must be constructed in the *command group scope*.

The SYCL Platform Model

The SYCL platform model is based on the OpenCL platform model, but there are a few additional abstractions available to programmers.

The model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are each divided into one or more processing elements (PEs). Computations on a device occur within the processing elements. A SYCL application runs on a host according to the standard C++ CPU execution model. The SYCL application submits command group function objects to command queues, which execute either on OpenCL devices or on the SYCL host device.

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.

When a SYCL implementation executes kernels on the host device, it is free to use whatever parallel execution facilities are available on the host, as long as it executes within the semantics of the kernel execution model defined

by OpenCL.

Platform Mixed Version Support

OpenCL is designed to support devices with different capabilities under a single platform. This includes devices which conform to different versions of the OpenCL specification and devices which support different extensions to the OpenCL specification. There are three important sets of capabilities to consider for a SYCL device: the platform version, the version of a device and the extensions supported.

The SYCL system presents the user with a set of devices, grouped into some number of platforms. The device version is an indication of the device's capabilities, as represented by the device information returned by the `cl::sycl::device::get_info()` method. Examples of attributes associated with the device version are resource limits and information about functionality beyond the core OpenCL specification's requirements. The version returned corresponds to the highest version of the OpenCL specification for which the device is conformant, but is not higher than the version of the device's platform which bounds the overall capabilities of the runtime operating the device.

In OpenCL, a device has a *language version*. In SYCL, the source language is compiled offline, so the language version is not available at runtime. Instead, the SYCL language version is available as a compile-time macro: `CL_SYCL_LANGUAGE_VERSION`.

SYCL Execution Model

The execution of a SYCL program occurs in two parts: SYCL kernel functions and a *application* that executes on the host. The SYCL kernels execution is governed by the *SYCL Kernel Execution Model*, whereas the SYCL application that executes on the host is governed by the *SYCL Application Execution Model*.

Like OpenCL, SYCL is capable of running kernels on multiple device types. However, SYCL adds functionality on top of OpenCL due to the integration into a host toolchain by providing an ability to run kernel code directly on the CPU without interacting with an OpenCL runtime. This is distinct from running on the CPU via an OpenCL device and can be used when no OpenCL platform is available on the machine.

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 object. command group objects are submitted to execution via command queue objects, which defines the device where the kernel will run. The same command group object can be submitted to different queues. When a command group is submitted to a SYCL command queue, the requirements of the kernel execution are captured. The kernels are executed as soon as their requirements have been satisfied.

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 `cl::sycl::platform` object. SYCL also provides a host platform object, which only contains a single host device.

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 `cl::sycl::context` object.

3. Devices: Platforms provide one or more devices for executing kernels. In SYCL, a device is accessible through a `cl::sycl::device` object. SYCL provides the abstract `cl::sycl::device_selector` class which the user can subclass to define how the runtime should select the best device from all available platforms for the user to use. For ease of use, SYCL provides a set of predefined concrete `device_selector` instances that select devices based on common criteria, such as type of device. SYCL, unlike OpenCL, defines a host device, which means any work that uses the host device will execute on the host and not on any OpenCL device.

4. Kernels: The SYCL functions that run on SYCL devices (i.e. either an OpenCL device, or the host device) are defined as C++ function objects (a named function object type or a lambda function). In SYCL, all kernels must have a kernel name, which must be a globally-accessible C++ type name. This is required to enable kernels compiled with one compiler to be linked to host code compiled with a different compiler.

   For named function objects, the type name of the function object is sufficient as the *kernel name*, but for C++11 lambda functions, the user must provide a user-defined type name as the *kernel name*.

5. Program objects: 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 `cl::sycl::program` class.

6. Command 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 `cl::sycl::queue` objects.

In OpenCL, queues can operate using in-order execution or out-of-order execution. In SYCL, the implementation must provide out-of-order execution ordering when possible, regardless of whether the underlying OpenCL queue is in-order or out-of-order.

SYCL Command Groups and Execution Order

In OpenCL, the user must enqueue commands on queues to transfer data or ensure different kernels execute in the correct order. OpenCL queues can be in-order (in which each command always waits for the previous one

to conclude) or out-of-order (where all commands execute as soon as they can). Events and barriers are used to synchronize host and device operations manually.

SYCL offers a higher abstraction in terms of queue ordering synchronization. All SYCL queues execute kernels in out-of-order fashion, regardless of the underlying OpenCL queues used. 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 `cl::sycl::queue::submit` method.

A **requisite** (r_i) 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_i 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 *foo* as $CG_{foo} = r_1, \ldots, r_n$.

The *evaluation of a requisite* (Satisfied(r_i)) returns the status of the requisite, which can be *True* or *False*. A *satisfied* requisite implies the requirement is met. 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_i) 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_i represents the action required to satisfy r_i. Actions of different requisites can be satisfied in any order w.r.t 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): OpenCL copy operations, mapping operations, host side synchronization, or implementation-specific behavior.

Finally, *Performing an action* (Perform(a_i)) executes the action operations required to satisfy the requisite r_j. Note that, after Perform(a_i), the evaluation Satisfied(r_j) 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 w.r.t each other without side effects: Given two requirements r_j and r_k, Perform(a_j) followed by Perform(a_k) is equivalent to Perform(a_k) followed by 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.

Figure 3.1 illustrates the execution order of three command group objects (CG_a, CG_b, CG_c) with certain requirements submitted to the same queue. Both CG_a and G_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.

Figure 3.2 uses three separate SYCL queue objects to submit the same command group objects as before. Re-

SYCL Application Enqueue Order **SYCL Kernel Execution Order**

cl::sycl::queue syclQueue;
syclQueue.submit($CG_a(r_1)$);
syclQueue.submit($CG_b(r_2)$);
syclQueue.submit($CG_c(r_1, r_2)$);

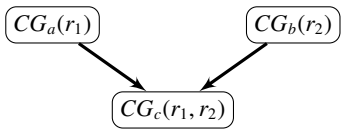


Figure 3.1: Execution order of three command groups submitted to the same queue.

gardless 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 methods is 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).

SYCL Application Enqueue Order **SYCL Kernel Execution Order**

cl::sycl::queue syclQueue1; cl::sycl::queue syclQueue2; cl::sycl::queue syclQueue3;
syclQueue1.submit($CG_a(r_1)$);
syclQueue2.submit($CG_b(r_2)$);
syclQueue3.submit($CG_c(r_1, r_2)$);

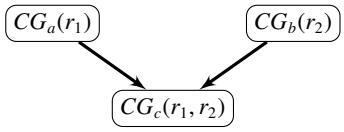


Figure 3.2: Execution order of three command groups submitted to the different queues.

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.

Work-items are organized into work-groups. The work-groups provide 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 concurrently on the processing elements of a single compute unit.

The index space supported in SYCL 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 `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> and a global offset, represented via a value of type id<N>. The types nd_range<N> and id<N> are each *N*-element arrays of integers. The iteration space defined via an range<N> is an *N*-dimensional index space starting at the ND-range's global offset and being of the size of 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>, 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.

SYCL allows a simplified execution model in which the work-group size is left unspecified. A kernel invoked over a range<N>, instead of an nd_range<N> is executed within an iteration space of unspecified workgroup size. In this case, less information is available to each work-item through the simpler item<N> class.

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, OpenCL rules are mapped to SYCL constructs to provide the same capabilities using C++ kernels.

SYCL Application Memory Model

The application running on the host uses SYCL buffer objects using instances of the cl::sycl::buffer class to allocate memory in the global address space, or can allocate specialized image memory using the cl::sycl::image class. In OpenCL, a memory object is attached to a specific context.

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 OpenCL memory objects together with multiple host memory allocations to enable the same object to be shared between devices in different contexts or platforms.

The order of execution of command group objects ensures a sequentially consistent access to the memory from the different devices to the memory objects.

To access memory objects from inside a kernel, the user must create an accessor object which parameterizes the type of access the kernel requires. The accessor object defines a requirement to access a memory object from a command group. The cl::sycl::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 *discard* flag can be added to an accessor to tell the system to discard any previous contents of the data the accessor refers to, e.g. discard write-only (DW). Atomic access can also be requested on an accessor which allows cl::sycl::atomic classes to be used via the accessor. For simplicity, when a **requisite** represents an accessor object in a certain access mode, we represent it as $\text{MemoryObject}_{\text{AccessMode}}$. For example, an accessor that accesses memory object **buf1** in **RW** mode is represented

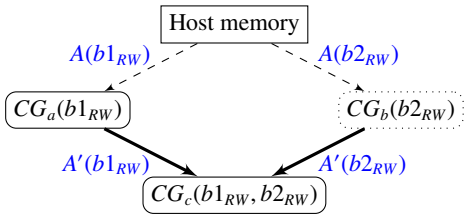
as $buf1_{RW}$. A command group object that uses such an accessor is represented as $CG(buf1_{RW})$. The **action** required to satisfy a requisite and the location of the latest copy of a memory object will vary depending on the implementation.

Figure 3.3 illustrates an example where command group objects are enqueued to two separate SYCL queues executing in devices in different contexts. The **requisites** for the command group execution are the same, but the **actions** to satisfy them are different. For example, if the data is on the host before execution, $A(b1_{RW})$ and $A(b2_{RW})$ can potentially be implemented as copy operations from the host memory to `context1` or `context2` respectively. After CG_a and CG_b are executed, $A'(b1_{RW})$ will likely be an empty operation, since the result of the kernel can stay on the device. On the other hand, the results of CG_b are now on a different context than CG_c is executing, therefore $A'(b2_{RW})$ will need to copy data across two separate OpenCL contexts using an implementation specific mechanism.

SYCL Application Enqueue Order

cl::sycl::queue q1(context1); cl::sycl::queue q2(context2); q1.submit($CG_a(b1_{RW})$); q2.submit($CG_b(b2_{RW})$); q1.submit($CG_c(b1_{RW}, b2_{RW})$);

SYCL Kernel Execution Order



Possible implementation by a SYCL Runtime

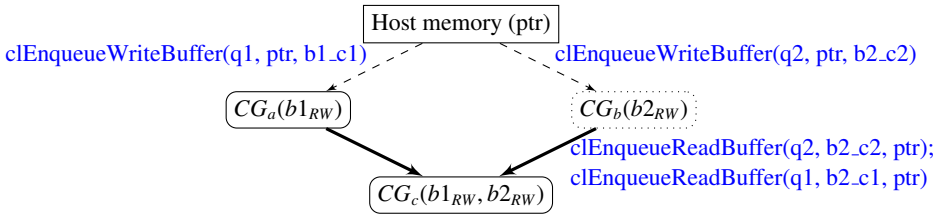


Figure 3.3: Actions performed when three command groups are submitted to two distinct queues, and possible OpenCL implementation of them by a SYCL runtime. Note that each SYCL buffer $(b1, b2)$ is implemented as separate `cl_mem` objects per context.

Note that the order of the definition of the accessors within the command group is irrelevant to the requirements they define. All accessors always apply to the entire command group object where they are defined. When multiple accessors in the same command group define requirements to the same memory object, the access mode is resolved as the union of all the different access modes, e.g. $CG(b1_R, b1_W)$ is equivalent to $CG(b1_{RW})$.

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 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.

SYCL Application Enqueue Order **SYCL Kernel Execution Order**

cl::sycl::queue q1(context1);
q1.submit($CG_a(b1_{RW,[0,10)})$);
q1.submit($CG_b(b1_{RW,[10,20)})$);
q1.submit($CG_c(b1_{RW,[5,15)})$);

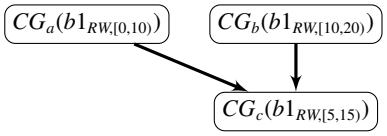


Figure 3.4: Requirements on overlapping vs non-overlapping *sub-buffer*.

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 devices.

A special case of requirement is the one defined by a **host accessor**. Host accessors are represented with $H(\text{MemoryObject}_{\text{accessMode}})$, e.g, $H(b1_{RW})$ 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. Figure 3.5 shows an example of multiple command groups enqueued to the same queue. Once the host accessor $H(b1_{RW})$ 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 $H(b1_{RW})$ is finished, but still has to wait for CG_b to conclude for all its requirements to be satisfied. See 3.6.5 for details on synchronization rules.

SYCL Application Enqueue Order **SYCL Kernel Execution Order**

cl::sycl::queue q1; q1.submit($CG_a(b1_{RW})$);
q1.submit($CG_b(b2_{RW})$);
$H(b1_{RW})$;
q1.submit($CG_c(b1_{RW}, b2_{RW})$);

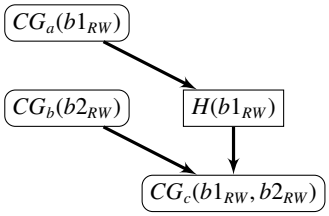


Figure 3.5: Execution of command groups when using host accessors.

SYCL Device Memory Model

The Memory Model for SYCL Devices is based on the OpenCL Memory model. Work-items executing in a kernel have access to four distinct memory regions:

- *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, however there is no guarantee that two concurrently executing kernels can simultaneously write to the same memory object and expect correct results.
- *Constant memory* is a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
- *Local memory* is a distinct memory region shared between work-items in a single work-group and inaccessible to work-items in other work-groups. 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 memory on an OpenCL device where this is appropriate.
- *Private memory* is 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.

Access to memory

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

It is not possible to pass a pointer into host memory directly as a kernel parameter because the devices may be unable to support the same address space as the host.

To allocate local memory within a kernel, the user can either pass a `cl::sycl::local_accessor` object to the kernel as a parameter, or can define a variable in workgroup scope inside `cl::sycl::parallel_for_work_group`.

Any variable defined inside a `cl::sycl::parallel_for` scope or `cl::sycl::parallel_for_work_item` scope will be allocated in private memory. Any variable defined inside a `cl::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, accessors can be implicitly cast to C++ pointer types. The pointer types will contain a compile-time deduced address space. So, for example, if an accessor to global memory is cast to a C++ pointer, 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 mechanism.

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: `cl::sycl::local_ptr`, `cl::sycl::global_ptr`, `cl::sycl::private_ptr`, or `cl::sycl::constant_ptr`. An accessor declared with one address space can be implicitly cast to an explicit pointer class for the same address space. Explicit pointer class values cannot be passed as parameters to kernels or stored in global memory.

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

Memory consistency

OpenCL uses a relaxed memory consistency model, i.e. the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. This also applies to SYCL kernels.

As in OpenCL, within a work-item memory has load/store consistency. Both local memory and global memory may be made consistent across work-items in a single work-group through use of a work-group barrier or work-group mem-fence operation with appropriate flags. There are no guarantees of memory consistency between different work-groups executing a kernel or between different kernels during their execution.

Atomic operations

Atomic operations can be performed on memory in buffers. The range of atomic operations available on a specific OpenCL device is limited by the atomic capabilities of that device. The `cl::sycl::atomic<T>` must be used for elements of a buffer to provide safe atomic access to the buffer from device code.

The SYCL programming model

SYCL programs are explicitly parallel and expose the full heterogeneous parallelism of the underlying machine model of OpenCL. This includes exposing the data-parallelism, multiple execution devices and multiple memory storage spaces of OpenCL. However, SYCL adds on top of OpenCL a higher level of abstraction allowing developers to hide much of the complexity from the source code, when a developer so chooses.

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++ function objects (a type callable with `operator()` or a lambda function), which have been designated to be compiled as SYCL kernels. SYCL will also accept OpenCL `cl_kernel` objects.

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

The C++ features used in SYCL are a subset of the C++11 standard features. Users will need to compile SYCL source code with C++ compilers which support the following C++ features:

- All C++03 features, apart from Run Time Type Information
- Exception handling
- C++11 lambda functions
- C++11 variadic templates
- C++11 template aliases

- C++11 rvalue references
- C++11 `std::function`, `std::string` and `std::vector`.

Basic data parallel kernels

Data-parallel kernels that execute as multiple work-items and where no local synchronization is required are enqueued with the `cl::sycl::parallel_for` function parameterized by a `cl::sycl::range` parameter. These kernels will execute the kernel function body once for each work-item in the range. The range passed to `cl::sycl::parallel_for` represents the global size of an OpenCL kernel and will be divided into work-groups whose size is chosen by the SYCL runtime. Barrier synchronization is not valid within these work-groups.

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 `cl::sycl::parallel_for` function with a `cl::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 in local or global memory and to synchronize between work-items in the same work-group by calling the `nd_item::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 be a multiple of the work-group size in each dimension.

Hierarchical data parallel kernels

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 `cl::sycl::parallel_for` the user calls `cl::sycl::parallel_for_work_group` with a `cl::sycl::range` value representing the number of work-groups to launch and optionally a second `cl::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 workgroup 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.

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 `cl::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, like any other OpenCL entity, be executed in task-parallel fashion.

Synchronization

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

Synchronization in the SYCL Application

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

- *Buffer destruction*: The destructors for `cl::sycl::buffer` and `cl::sycl::image` objects wait for all submitted work on those objects to complete and copies the data back to host memory before returning, if there is anything to copy back to the host or if the objects were constructed with attached host memory.

  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 (see 3.5).

- *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 `cl::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 **wait** to block execution of the caller 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.

- *Interaction with OpenCL synchronization operations*: The user can obtain OpenCL events from command groups which will enable the user to add barriers to their own queues to correctly synchronize for buffer or image data dependencies.

- *SYCL event objects*: SYCL provides `cl::sycl::event` objects which can be used for user synchronization. If synchronization is required between two different OpenCL contexts, then the SYCL runtime ensures that any 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 (`cl::sycl::queue`, `cl::sycl::context`...) do not block. Only a `cl::sycl::buffer` or `cl::sycl::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 method to wait for completion in some cases if this does not fit the goal. See Section 3.6.9 for more information on object life time.

Synchronization in SYCL Kernels

In SYCL, synchronization can be either global or local within a work-group. The SYCL implementation may need to provide extra synchronization commands and host-side synchronization in order to enable synchronization across OpenCL contexts, but this is handled internally within the SYCL runtime.

Synchronization between work-items in a single work-group is achieved using a work-group barrier. This matches the OpenCL C behavior. All the work-items of a work-group must execute the barrier before any are allowed to continue execution beyond the barrier. Note that the work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all. There is no mechanism for synchronization between work-groups. In SYCL, work-group barriers are exposed through the method on the `cl::sycl::nd_item` class, `nd_item::barrier()` which is only available inside kernels that are executed over work-groups. This ensures that developers can only use work-group barriers inside work-groups.

Error handling

In SYCL, there are two types of error: synchronous errors that can be detected immediately, and asynchronous errors that can only be detected later. 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 user-supplied asynchronous error-handlers.

A `cl::sycl::context` can be constructed with a user-supplied asynchronous error handler. If a `cl::sycl::queue` is constructed without a user-supplied context, then the user can supply an asynchronous error handler for the queue, otherwise errors on that queue will be reported to its context error handler.

Asynchronous errors are not reported immediately as they occur. The asynchronous error handler for a context or queue is called with a `cl::sycl::exception_list` object, which contains a list of asynchronously-generated exception objects, either on destruction of the context or queue that the error handler is associated with, or via an explicit `wait_and_throw` method call on an associated queue.

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.

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 has to get the handler object as a parameter. The intention is that a user will perform calls to SYCL functions, methods, 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 runtime SYCL operations will be queued.

It is worth noting that a SYCL queue does not necessarily map to only one OpenCL queue, however, the OpenCL queue that is given when interacting with the SYCL queue will retain any synchronization information that is needed for synchronization with any other OpenCL queues spawned by the system.

An OpenCL implementation can require different queues for different devices and contexts. The synchronization required to ensure order between commands in different queues varies according to whether the queues have shared contexts. A SYCL implementation must determine the required synchronization to ensure the above ordering rules above are enforced.

Managing object lifetimes

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

All OpenCL 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 OpenCL context) for the lifetime of their application to initialize and release the OpenCL context safely.

When an OpenCL object that is encapsulated in a SYCL object is copied in C++, then the underlying OpenCL object is not duplicated, but its OpenCL reference count is incremented. When the original or copied SYCL object is destroyed, then the OpenCL reference count is decremented.

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. Instead of a raw pointer, a `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. In the case where host memory is shared between the user application and the SYCL runtime, then the reference counter of the `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.

As said in Section 3.6.5, 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.

Device discovery and selection

A user specifies which queue to submit a command group function object on 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 selector causes the SYCL runtime to perform device discovery. No device discovery is performed until a SYCL 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 both OpenCL devices and platforms as well as a SYCL host platform and SYCL host device. The host device allows queue creation and running of kernels, but does not support OpenCL-specific features. It is an error for a user to request an underlying OpenCL device for the SYCL host device.

Interfacing with OpenCL

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. These constructors must retain that instance to increase the reference count of the OpenCL resource.

SYCL runtime classes which encapsulate an OpenCL opaque type (excluding the SYCL `buffer` and SYCL `image` class templates) can be queried for their encapsulated instance via a `get()` member function. These `get()` member functions must retain the instance to increase the reference count of the OpenCL resource.

The destructor for the SYCL runtime classes which encapsulate an OpenCL opaque type must release that instance to decrease the reference count of the 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 may only encapsulate a single instance of that the OpenCL type.

Note that the lifetime of a SYCL runtime class that encapsulates an OpenCL opaque type and the instance of that opaque type retrieved via the `get()` member 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.

Memory objects

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 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, float), vector data type, or a user-defined structure. In SYCL, a buffer object is a templated type (cl::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 OpenCL implementation. Images are encapsulated in the cl::sycl::image type, which is templated by the number of dimensions in the image. The minimum number of elements in a memory object is one.

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 can be converted within a kernel into C++ pointer types, or the cl::sycl::global_ptr, cl::sycl::constant_ptr classes. 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 or write to an image.

- For a buffer object the data is accessed within a kernel in the same format as it is stored in memory, but in the case of an image object the data is not necessarily accessed within a kernel in the same format as it is stored in memory.

  Image elements are always a 4-component vector (each component can be a float or signed/ unsigned integer) in a kernel. The SYCL accessor and sampler methods to read from an image convert an image element from the format that it is stored in to a 4-component vector. Similarly, the SYCL accessor methods provided to write to an image convert the image element from a 4-component vector to the appropriate image format specified such as 4 8-bit elements, for example.

Memory objects, both buffers and images, may have one or more underlying OpenCL cl_mem objects. 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.

Users may want fine-grained control of the synchronization, memory management and storage semantics of SYCL image or buffer objects. For example, a user may wish to specify the host memory for a memory object to use, but may not want the memory object to block on destruction.

Depending on the control and the use cases of the SYCL applications, well established C++ classes and patterns can be used for reference counting and sharing data between user applications and the SYCL runtime. For control over memory allocation on the host and mapping between host and device memory, pre-defined or user-defined C++ `allocator` classes are used. For better control of synchronization between a SYCL and a non SYCL application that share data, `shared_ptr` and `mutex` classes are used. In the case where the user would not like the host side to block on destruction of buffers or images, as the data given to the buffers are for initialization only, the `unique_ptr` class can be used instead of a raw pointer to data.

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. If no OpenCL implementation is available, then the SYCL implementation provides only the SYCL host device to run kernels on.
- *SYCL Device Compiler(s)*: 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.

SYCL device compiler

To enable SYCL to work on a variety of platforms, with different devices, operating systems, build systems and host compilers, SYCL provides a number of options to implementers to enable the compilation of SYCL kernels for devices, while still providing a unified programming model to the user.

Building a SYCL program

A SYCL program runs on a *host* and one or more OpenCL devices. This requires a compilation model that enables compilation for a variety of targets. There is only ever one host for the SYCL program, so the compilation of the source code for the host must happen once and only once. Both kernel and non-kernel source code is compiled for the host.

The design of SYCL enables a single SYCL source file to be passed to multiple, different compilers, using the

SMCP technique. This is an implementation option and is not required. What this option enables is for an implementer to provide a device compiler only and not have to provide a host compiler. A programmer who uses such an implementation will compile the same source file twice: once with the host compiler of their choice and once with a device compiler. This approach allows the advantages of having a single source file for both host code and kernels, while still allowing users an independent choice of host and SYCL device compilers.

Only the kernels are compiled for OpenCL devices. Therefore, any compiler that compiles only for one or more devices must not compile non-kernel source code. Kernels are contained within C++ source code and may be dependent on lambda capture and template parameters, so compilation of the non-kernel code must determine lambda captures and template parameters, but not generate device code for non-kernel code.

Compilation of a SYCL program may follow either of the following options. The choice of option is made by the implementer:

1. *Separate compilation*: One or more device compilers compile just the SYCL kernels for one or more devices. The device compilers all produce header files for interfacing between the host compiler and the SYCL runtime,which are integrated together with a tool that produces a single header file. The user compiles the source file with a normal C++ host compiler for their platform. The user must ensure that the host compiler is given the correct command-line arguments to ensure that the device compiler output header file is `#included` from inside the SYCL header files.

2. *Single-source compiler*: In this approach, a single compiler may compile an entire source file for both host and one or more devices. It is the responsibility of the single-source compiler to enable kernels to be compiled correctly for devices and enqueued from the host.

An implementer of SYCL may choose an implementation approach from the options above.

Naming of kernels

SYCL kernels are extracted from C++ source files and stored in an implementation-defined format. When the SYCL runtime needs to enqueue a SYCL kernel, it is necessary for the SYCL runtime to load the kernel and pass it to an OpenCL runtime. This requires the kernel to have a globally-visible name to enable an association between the kernel invocation and the kernel itself. The association is achieved using a kernel name, which is a C++ type name.

For a named function object, the kernel name can be the same type as the function object itself, as long as the function object type is globally accessible. For a lambda function, there is no globally-visible name, so the user must provide one. In SYCL, the name is provided as a template parameter to the kernel invocation, e.g. `parallel_for<class kernelName>`.

A device compiler should detect the kernel invocations (e.g. `parallel_for`) in the source code and compile the enclosed kernels, storing them with their associated type name. For details please refer to 6.2. The user can also extract OpenCL `cl_kernel` and `cl_program` objects for kernels by providing the type name of the kernel.

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 OpenCL 1.2 runtime and

OpenCL 1.2 capable devices, there are certain restrictions for SYCL kernels. Those restrictions can be summarized as the kernels cannot include RTTI information, exception classes, recursive code, virtual functions or make use of C++ libraries that are not compiled for the device. For more details on language restrictions please refer to 6.3.

SYCL kernels use parameters that are captured by value in the command group scope or are passed from the host to the device using the data management runtime classes of `cl::sycl::accessors`. Sharing data structures between host and device code imposes certain restrictions, such as use of only user defined classes that are *C++11 standard layout* classes for the data structures, and in general, no pointers initialized for the host can be used on the device. The only way of passing pointers to a kernel is through the `cl::sycl::accessor` class, which supports the `cl::sycl::buffer` and `cl::sycl::image` classes. No hierarchical structures of these 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 4.8.10.

Some types in SYCL vary according to pointer size or vary on the host according to the host ABI, such as `size_t` or `long`. In order for the the SYCL device compiler to ensure that the sizes of these types match the sizes on the host and to enable data of these types to be shared between host and device, the OpenCL interoperability types are defined, `cl::sycl::cl_int` and `cl::sycl::cl_size_t`.

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.

SYCL Linker

In SYCL only offline linking is supported for SYCL and OpenCL programs and libraries, however the mechanism is optional. In the case of linking C++ functions to a SYCL application, where the definitions are not available in the same translation unit of the compiler, then the macro `SYCL_EXTERNAL` has to be provided. 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 4.94.

Functions and datatypes available in kernels

Inside kernels, the functions and datatypes available are restricted by the underlying capabilities of OpenCL devices. All OpenCL C features are provided by C++ classes and functions, which are available on host and device.

Execution of kernels on the SYCL host device

SYCL enables kernels to run on either the host device or on OpenCL devices. When kernels run on an OpenCL device, then the features and behavior of that execution follows the OpenCL specification, otherwise they follow the behavior specified for the SYCL host device.

Any kernel enqueued to a host queue executes on the host device according to the same rules as the OpenCL devices.

Kernel math library functions on the host must conform to OpenCL math precision requirements. The SYCL host device needs to be queried for the capabilities it provides.

The range of image formats supported by the host device is implementation-defined, but must match the minimum requirements of the OpenCL specification.

Some of the OpenCL extensions and optional features may be available on a SYCL host device, but since these are optional features and vendor specific extensions, the user must query the host device to determine availability. A SYCL implementer must state what OpenCL device features are available on their host device implementation.

The synchronization and data movement that occurs when a kernel is executed on the host may be implemented in a variety of ways on top of OpenCL. The actual mechanism is implementation-defined.

Endianness support

SYCL supports both big-endian and little-endian systems by extension of the portability of OpenCL devices. However SYCL does not support mix-endian systems and does not support specifying the endianness of data within a SYCL kernel function. Users must be aware of the endianness of the host and the OpenCL devices they are targeting to ensure kernel arguments are processed correctly when applicable.

Example SYCL application

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

```
1  #include <CL/sycl.hpp>
2  #include <iostream>
3
4  using namespace cl::sycl;
5
6  // Size of the matrices
7  const size_t N = 2000;
8  const size_t M = 3000;
9
10 int main() {
11   // Create a queue to work on
12   queue myQueue;
13
14   // Create some 2D buffers of float for our matrices
15   buffer<float, 2> a(range<2>{N, M});
16   buffer<float, 2> b(range<2>{N, M});
17   buffer<float, 2> c(range<2>{N, M});
18
19   // Launch a first asynchronous kernel to initialize a
20   myQueue.submit( [&](handler& cgh) {
21     // The kernel write a, so get a write accessor on it
```

```
22      auto A = a.get_access<access::mode::write>(cgh);
23
24      // Enqueue a parallel kernel iterating on a N*M 2D iteration space
25      cgh.parallel_for<class init_a>(range<2> {N, M}, [=](id<2> index) {
26        A[index] = index[0] * 2 + index[1]; });
27    });
28
29    // Launch an asynchronous kernel to initialize b
30    myQueue.submit( [&](handler& cgh) {
31      // The kernel write b, so get a write accessor on it
32      auto B = b.get_access<access::mode::write>(cgh);
33      /* From the access pattern above, the SYCL runtime detect this
34         command_group is independant from the first one and can be
35         scheduled independently */
36
37      // Enqueue a parallel kernel iterating on a N*M 2D iteration space
38      cgh.parallel_for<class init_b>(range<2> {N, M}, [=](id<2> index) {
39        B[index] = index[0] * 2014 + index[1] * 42;
40      });
41    });
42
43    // Launch an asynchronous kernel to compute matrix addition c = a + b
44    myQueue.submit( [&](handler& cgh) {
45    // In the kernel a and b are read, but c is written
46    auto A = a.get_access<access::mode::read>(cgh);
47    auto B = b.get_access<access::mode::read>(cgh);
48    auto C = c.get_access<access::mode::write>(cgh);
49    // From these accessors, the SYCL runtime will ensure that when
50    // this kernel is run, the kernels computing a and b completed
51
52    // Enqueue a parallel kernel iterating on a N*M 2D iteration space
53    cgh.parallel_for<class matrix_add>(range<2> {N, M}, [=](id<2> index) {
54        C[index] = A[index] + B[index]; });
55    });
56
57    /* Ask an access to read c from the host-side.
58       This form implies access::target::host_buffer. The SYCL runtime
59       ensures that c is ready when the accessor is returned */
60    auto C = c.get_access<access::mode::read>();
61    std::cout << std::endl << "Result:" << std::endl;
62    for (size_t i = 0; i < N; i++) {
63      for (size_t j = 0; j < M; j++) {
64        // Compare the result to the analytic value
65        if (C[i][j] != i * (2 + 2014) + j * (1 + 42)) {
66          std::cout << "Wrong value " << C[i][j] << " on element " << i << " "
67                    << j << std::endl;
68          exit(-1);
69        }
70      }
71    }
72
73    std::cout << "Good computation!" << std::endl;
74    return 0;
75  }
```

SYCL Programming Interface

The SYCL programming interface provides a C++ abstraction to OpenCL 1.2 functionality and feature set. This section describes all the available classes and interfaces of SYCL, focusing on the C++ interface of the underlying runtime. In this section, we are defining all the classes and member functions for the SYCL API, which are available for SYCL host and OpenCL devices. This section also describes the synchronization rules and OpenCL API interoperability rules which guarantee that all the member functions and special member functions of the SYCL classes are thread safe.

It is assumed that the OpenCL API is also available to the developer at the same time as SYCL.

Header files and namespaces

SYCL provides one standard header file: `"CL/sycl.hpp"`, which needs to be included in every SYCL program.

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

Any SYCL classes, constants, types and functions defined as an extension to this specification should exist within the `cl::sycl::<vendor_name>` namespace.

The `cl::sycl::detail` namespace is reserved for implementation details.

Class availability

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

Each of the following SYCL runtime classes: `device_selector`, `platform`, `device`, `context`, `queue`, `program`, `kernel`, `event`, `buffer`, `image`, `sampler`, `stream`, `handler`, `nd_range`, `range`, `id`, `vec`, `buffer_allocator`, `image_allocator` and `exception` must be available to the host application.

Each of the following SYCL runtime classes: `accessor`, `sampler`, `stream`, `vec`, `multi_ptr` `device_event`, `id`, `range`, `item`, `nd_item`, `h_item`, `group` and `atomic` must be available within a SYCL kernel function.

Each of the following SYCL runtime classes: `accessor`, `sampler`, `stream`, `vec`, `id` and `range` are permitted as parameters to a SYCL kernel function.

Common interface

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

OpenCL interoperability

Many of the SYCL runtime classes encapsulate an associated OpenCL opaque type and provide facilities for interoperating between the SYCL classes and the OpenCL opaque types they encapsulate in order to allow interoperability between SYCL and OpenCL applications.

Each of the following SYCL runtime classes: `platform`, `device`, `context`, `queue`, `program`, `kernel`, `event`, `buffer`, `image`, `sampler` and `stream` must obey the following statements, where `T` is the runtime class type:

- `T` on the host application must encapsulate at least one valid instance of the associated OpenCL opaque type if the SYCL class instance is an OpenCL instance and must not encapsulate any instance of the associated OpenCL opaque type if this instance of T is a host instance.
- `T` must provide an interoperability constructor on the host application which takes as a parameter, a valid instance of the associated OpenCL opaque type, which must be retained on construction of an instance of `T`, where applicable. The constructed instance of T must be an OpenCL instance and must encapsulate only the single instance of the associated OpenCL opaque type provided during construction.
- `T` must provide a `get()` member function on the host application which returns an encapsulated instance of the associated OpenCL opaque type if this instance of `T` is an OpenCL instance, and must throw an `invalid_object_error` SYCL exception if this instance of `T` is a host instance. The instance of the associated OpenCL type must be retained before returning and must always be the same instance, where applicable.
- `T` must release each encapsulated instance of the associated OpenCL opaque type on destruction on the host application, where applicable.

The only exceptions to these rules are the SYCL `buffer`, `image` and `sampler` classes which do not require a `get()` member function.

For more details regarding these facilities and considerations for their use see section 3.6.11.

Common reference semantics

Each of the following SYCL runtime classes: `device`, `context`, `queue`, `program`, `kernel`, `event`, `buffer`, `image`, `sampler`, `accessor` and `stream` 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 if said instance is not a host object must represent and continue to represent the same underlying OpenCL objects 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 and the destructor requirements described in 4.3.1 where applicable.

- `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 if said instance is not a host object must represent and continue to represent the same underlying OpenCL objects 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 && b == c` implies `c == a`).

- A specialization of `hash_class` 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 and member functions in order to fulfil the copy, move, destruction and equality requirements.

These common special member functions and member functions are described in Tables 4.1 and 4.2 respectively.

```
1  namespace cl {
2  namespace sycl {
3
4  class T {
5    ...
6
7   public:
8    T(const T &rhs);
9
10   T(T &&rhs);
11
12   T &operator=(const T &rhs);
13
14   T &operator=(T &&rhs);
15
16   ~T();
17
18   bool operator==(const T &rhs) const;
19
20   bool operator!=(const T &rhs) const;
21
```

```
22    ...
23  };
24  }  // namespace sycl
25  }  // namespace cl
```

Special member function	Description
`T(const T &rhs)`	Constructs a `T` instance as a copy of the RHS SYCL `T` in accordance with the requirements set out above.
`T(T &&rhs)`	Constructs a SYCL `T` instance as a move of the RHS SYCL `T` in accordance with the requirements set out above.
`T &operator=(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=(T &&rhs)`	Assigns this SYCL `T` instance with a move of the RHS SYCL `T` in accordance with the requirements set out above.
`~T()`	Destroys this SYCL `T` instance in accordance with the requirements set out in 4.3.2. Must release a reference to the associated OpenCL object if this SYCL `T` instance was constructed with the interoperability constructor.
	End of table

Table 4.1: Common special member functions for reference semantics.

Member function	Description
`bool operator==(const T &rhs)const`	Returns true if this SYCL `T` is equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false.
`bool operator!=(const T &rhs)const`	Returns true if this SYCL `T` is not equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false.
	End of table

Table 4.2: Common member functions for reference semantics.

Common by-value semantics

Each of the following SYCL runtime classes: `id`, `range`, `item`, `nd_item`, `h_item`, `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 and within SYCL kernel

functions.

- `T` must be default destructible on the host application and within SYCL kernel functions.
- `T` must be default move constructible and default move assignable on the host application and within SYCL kernel functions.
- `T` must be equality comparable on the host application 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 fulfil the copy, move, destruction and equality requirements.

These common special member functions and member functions are described in Tables 4.3 and 4.4 respectively.

```
1  namespace cl {
2  namespace sycl {
3
4  class T {
5    ...
6
7   public:
8    T(const T &rhs) = default;
9
10   T(T &&rhs) = default;
11
12   T &operator=(const T &rhs) = default;
13
14   T &operator=(T &&rhs) = default;
15
16   ~T() = default;
17
18   bool operator==(const T &rhs) const;
19
20   bool operator!=(const T &rhs) const;
21
22   ...
23 };
24 }  // namespace sycl
25 }  // namespace cl
```

Special member function	Description
`T(const T &rhs)= default`	Default copy constructor.
`T(T &&rhs)= default`	Default move constructor.

Continued on next page

Table 4.3: Common special member functions for by-value semantics.

Special member function	Description
`T &operator=(const T &rhs)= default`	Default copy assignment operator.
`T &operator=(T &&rhs)= default`	Default move assignment operator.
`~T()= default`	Default destructor.
	End of table

Table 4.3: Common special member functions for by-value semantics.

Member function	Description
`bool operator==(const T &rhs)const`	Returns true if this SYCL `T` is equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false.
`bool operator!=(const T &rhs)const`	Returns true if this SYCL `T` is not equal to the RHS SYCL `T` in accordance with the requirements set out above, otherwise returns false.
	End of table

Table 4.4: Common member functions for by-value semantics.

Properties

Each of the following SYCL runtime classes: `buffer`, `image` and `queue` 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 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.

```
1  {
2    context myContext;
3
4    std::vector<buffer<int, 1>> bufferList {
5      buffer<int, 1>{},
6      buffer<int, 1>{property::use_host_ptr{}},
7      buffer<int, 1>{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 other 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 and SYCL `image` class in Table 4.33 and Table 4.38 respectively.

Any property that is provided to a SYCL runtime class via an instance of the SYCL `property_list` class must become encapsulated by that class and therefore shared between copies of that class. As a result properties must inherit the copy and move semantics of that class as described in 4.3.2.

A SYCL implementation may provide additional properties other than those defined here, provided they are defined in accordance with the requirements described in 4.1.

Properties interface

Each of the runtime classes mentioned above must provide a common interface of member functions in order to fulfil 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 4.5. The constructors of the SYCL `property_list` class are listed in Table 4.6.

```
1  namespace cl {
2  namespace sycl {
3  class T {
4    ...
5
6    template <typename propertyT>
7    bool has_property() const;
8
9    template <typename propertyT>
10   propertyT get_property() const;
11
12   ...
13 };
14
15 class property_list {
16  public:
17    template <typename... propertyTN>
18    property_list(propertyTN... props);
19 };
20 }  // namespace sycl
21 }  // namespace cl
```

Member function	Description
`template <typename propertyT>` `bool has_property()const`	Returns true if `T` was constructed with the property specified by `propertyT`. Returns false if it was not.

Continued on next page

Table 4.5: Common member functions of the SYCL property interface.

Member function	Description
`template <typename propertyT> propertyT get_property()const`	Returns a copy of the property of type `propertyT` that `T` was constructed with. Must throw an `invalid_object_error` SYCL exception if `T` was not constructed with the `propertyT` property.
	End of table

Table 4.5: Common member functions of the SYCL property interface.

Constructor	Description
`template <typename... propertyTN> property_list(propertyTN... props)`	Construct a SYCL `property_list` with zero or more properties.
	End of table

Table 4.6: Constructors of the SYCL `property_list` class.

Param traits class

The class `param_traits` is a C++ type trait for providing an alias to the return type associated with each info parameter. An implementation must provide a specialization of the `param_traits` class for every info parameter with the associated return type as defined in the info parameter tables.

```
1  namespace cl {
2  namespace sycl {
3  namespace info {
4  template <typename T, T param>
5  class param_traits {
6   public:
7
8     using return_type = __return_type__<T, param>;
9
10 };
11 }  // namespace info
12 }  // namespace sycl
13 }  // namespace cl
```

C++ Standard library classes required for the interface

The SYCL programming interfaces make extensive use of vectors, strings and function objects to carry information. Moreover, smart pointer and mutex classes allow extending the SYCL programming interface in terms of host data management. SYCL will default to using the STL string, vector, function, mutex and smart pointer classes, unless defined otherwise.

A SYCL implementation must provide aliases for the STL types that are used on the interface. These types

are exposed internally as vector_class, string_class, function_class, mutex_class, shared_ptr_class, weak_ptr_class, hash_class and exception_ptr_class.

Typically, the SYCL types will be aliases to the system STL library, as shown in the listing below:

```
#include <exception>
#include <functional>
#include <memory>
#include <mutex>
#include <string>
#include <vector>

namespace cl {
namespace sycl {

    template < class T, class Alloc = std::allocator<T> >
    using vector_class = std::vector<T, Alloc>;

    using string_class = std::string;

    template<class R, class... ArgTypes>
    using function_class = std::function<R(ArgTypes...)>;

    using mutex_class = std::mutex;

    template <class T>
    using shared_ptr_class = std::shared_ptr<T>;

    template <class T>
    using unique_ptr_class = std::unique_ptr<T>;

    template <class T>
    using weak_ptr_class = std::weak_ptr<T>;

    template <class T>
    using hash_class = std::hash<T>;

    using exception_ptr_class = std::exception_ptr;

}  // sycl
}  // cl

#include <CL/sycl.hpp>
```

However, a SYCL implementation may provide a custom implementation of any of these objects. This enables SYCL implementations to use optimized classes for specific platforms. To guarantee interoperability with the implementation types, users should use the aliases on the SYCL namespace instead of the standard types. Implementations must provide their own implicit conversion operations from the standard types into the custom defined types if they are not the same as the ones provided by the default standard template library of the system.

SYCL runtime classes

Device selection class

The SYCL device_selector class is an object which enables the SYCL runtime to choose the best device based on heuristics specified by the user, or by one of the built-in device selectors. The built-in device selectors are listed in Table 4.9.

The constructors and member functions of the SYCL device_selector class are described in Tables 4.7 and 4.8 respectively.

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

Device selector interface

The function call operator; operator() of the SYCL device_selector is an abstract member function which takes a reference to a SYCL device and returns an integer score. This abstract member function can be implemented in a derived class in order to provide a logic for selecting a SYCL device.

At any point where the SYCL runtime needs to select a SYCL device, the system will call the select_device() member functions, which will query all available SYCL devices in the system, pass each to this function call operator and select the one which returns the highest highest score. If a negative score is returned the the corresponding SYCL device will never be chosen. The SYCL devices that are provided to the SYCL device_selector can be any number of OpenCL devices but must contain a single host device.

```
1  namespace cl {
2  namespace sycl {
3  class device_selector {
4   public:
5    device_selector();
6
7    device_selector(const device_selector &rhs);
8
9    device_selector &operator=(const device_selector &rhs);
10
11   virtual ~device_selector();
12
13   device select_device() const;
14
15   virtual int operator()(const device &device) const = 0;
16 };
17 }  // namespace sycl
18 }  // namespace cl
```

Constructor	Description
device_selector()	Constructs a SYCLdevice_selector instance.
device_selector(const device_selector &rhs)	Constructs a SYCL device_selector instance from another instance.

Continued on next page

Table 4.7: Constructors of the device_selector class.

Constructor	Description
`device_selector &operator=(const device_selector & rhs)`	Assigns this SYCL `device_selector` instance with another instance.
`virtual ~device_selector()`	Destroys this SYCL `device_selector` instance.
	End of table

Table 4.7: Constructors of the `device_selector` class.

Member function	Description
`device select_device()const`	Returns a SYCL `device` that has been selected based on the highest score returned by the function call operator for all available SYCL `device`s in the system.
`virtual int operator()(const device &device)const`	Pure virtual member function, required to be implemented in a derived class to provide a logic for selecting a SYCL `device`. Returns an integer score for the `device` parameter based on the logic defined within it.
	End of table

Table 4.8: Member functions for the `device_selector` class.

Derived device selector classes

As the SYCL `device_selector` is an abstract class, it must be derived from with a valid implementation of the function call operator in order to be used by the SYCL runtime.

Any class which derives from the `device_selector`, in order to be used polymorphically, must have a valid copy constructor, copy assignment operator and destructor and it must implement the abstract function call operator.

The system provides a number of built-in derived `device_selector` types, including a selectors type which chooses a SYCL `device` based on the default behavior of the SYCL runtime, known as the `default_selector`. It is important to note that the behavior of the `default_selector` may be restricted by the platforms that the implementation chooses to target, and it must select a host device if no other suitable OpenCL device can be found. The SYCL `default_selector` is used in some cases as the default SYCL `device_selector` if one if not provided.

SYCL device selectors	Description
`default_selector`	Derived SYCL `device_selector` which selects a SYCL `device` based on an implementation defined heuristic. Must select a host device if no other suitable OpenCL device can be found.
	Continued on next page

Table 4.9: Standard device selectors included with all SYCL implementations.

SYCL device selectors	Description
gpu_selector	Derived SYCL device_selector which selects a SYCL device for which the device type is info::device::device_type::gpu. Must throw a runtime_error SYCL exception if no OpenCL device matching this requirement can be found.
accelerator_selector	Derived SYCL device_selector which selects a SYCL device for which the device type is info::device::device_type::accelerator. Must throw a runtime_error SYCL exception if no OpenCL device matching this requirement can be found.
cpu_selector	Derived SYCL device_selector which selects a SYCL device for which the device type is info::device::device_type::cpu. Must throw a runtime_error SYCL exception if no OpenCL device matching this requirement can be found.
host_selector	Derived SYCL device_selector which selects a SYCL device that is a host device. Must always return a valid SYCL device.
	End of table

Table 4.9: Standard device selectors included with all SYCL implementations.

Platform class

The SYCL platform class encapsulates a single SYCL platform on which SYCL kernel functions may be executed. A SYCL platform may be an OpenCL platform in which case it must encapsulate a valid underlying OpenCL cl_platform_id, or it may be a SYCL host platform in which case it must not.

A SYCL platform is also associated with one or more SYCL devices. These can be any number of OpenCL devices or exactly one host device.

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

The default constructor of the SYCL platform class will construct a host platform. The explicit constructor of the SYCL platform class which takes a device_selector will construct a host platform if select_device returns a host device, otherwise will construct an OpenCL platform. The OpenCL interop constructor of the SYCL platform class will construct an OpenCL platform.

The SYCL platform class provides the common reference semantics (see Section 4.3.2).

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 Tables 4.10, 4.11 and 4.12 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2 respectively.

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

  explicit platform(cl_platform_id platformID);

  explicit platform(const device_selector &deviceSelector);

  /* -- common interface members -- */

  cl_platform_id get() const;

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

  template <info::platform param>
  typename info::param_traits<info::platform, param>::return_type get_info() const;

  bool has_extension(const string_class &extension) const;

  bool is_host() const;

  static vector_class<platform> get_platforms();
};
}  // namespace sycl
}  // namespace cl
```

Constructor	Description
`platform()`	Constructs a SYCL `platform` instance as a host platform.
`explicit platform(cl_platform_id platformID)`	Constructs a SYCL `platform` instance from an OpenCL `cl_platform_id` in accordance with the requirements described in 4.3.1.
`explicit platform(const device_selector &deviceSelector)`	Constructs a SYCL `platform` instance using the the `deviceSelector` parameter. One of the SYCL `device`s that is associated with the constructed SYCL `platform` instance must be the SYCL `device` that is produced from the `deviceSelector` parameter.
	End of table

Table 4.10: Constructors of the SYCL `platform` class.

Member function	Description
`cl_platform_id get()const`	Returns a valid `cl_platform_id` instance in accordance with the requirements described in 4.3.1.
`template <info::platform param> typename info::param_traits<info::platform, param>::return_type get_info()const`	Queries this SYCL `platform` for information requested by the template parameter `param`. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.20 to facilitate returning the type associated with the `param` parameter.
`bool has_extension(const string_class & extension) const`	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 `device`s support that extension.
`bool is_host()const`	Returns true if this SYCL `platform` is a host platform.
`vector_class<device> get_devices( info::device_type = info::device_type::all)const`	Returns a vector_class containing all SYCL `device`s associated with this SYCL `platform`. The returned vector_class must contain only a single SYCL `device` that is a host device if this SYCL `platform` is a host platform. Must return an empty `vector_class` instance if there are no devices that match the given `info::device_type`.
	End of table

Table 4.11: Member functions of the SYCL `platform` class.

Static member function	Description
`static vector_class<platform> get_platforms()`	Returns a `vector_class` containing all SYCL `platform`s available in the system. The returned vector_class must contain a single SYCL `platform` that is a host platform.
	End of table

Table 4.12: Static member functions of the SYCL `platform` class.

Platform information descriptors

A SYCL `platform` can be queried for all of the following information using the `get_info` member function. All SYCL `platform`s must have valid values for every query, including a host platform. The information that can be queried is described in Table 4.13. The interface for all information types and enumerations are described in appendix A.1.

Platform descriptors	Return type	Description
info::platform::profile	string_class	Returns the OpenCL profile as a string_class, if this SYCL platform is an OpenCL platform. The value returned can be one of the following strings: • "FULL_PROFILE" — if the platform 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 platform supports the OpenCL embedded profile. Must return a string_class with the value "FULL PROFILE" if this is a host platform.
info::platform::version	string_class	Returns the OpenCL software driver version as a string_class in the form: major_number.minor_number, if this SYCL platform is an OpenCL platform. Must return a string_class with the value "1.2" if this SYCL platform is a host platform.
info::platform::name	string_class	Returns the device name of this SYCL platform.
info::platform::vendor	string_class	Returns the vendor of this SYCL platform.
info::platform::extensions	vector_class < string_class >	Returns a vector_class of extension names (the extension names do not contain any spaces) supported by this SYCL platform. An extension can only be returned here if it is supported by all associated SYCL devices.
		End of table

Table 4.13: Platform information descriptors.

Context class

The context class represents a SYCL context on which SYCL kernel functions may be executed. A SYCL context may be an OpenCL context, in which case it must encapsulate a valid underlying OpenCL `cl_context`, or it may be a SYCL host context, in which case it must not. A SYCL `context` must encapsulate a single SYCL platform and a collection of SYCL devices all of which are associated with said platform.

The SYCL `context` class provides the common reference semantics (see Section 4.3.2).

Context interface

The constructors and member functions of the SYCL `context` class are listed in Tables 4.14 and 4.15, respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, 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, excluding the interoperability constructor, will construct either an OpenCL context or a host context, determined by the constructor parameters or, in the case of the default constructor, the SYCL device produced by the default_selector. If the SYCL platform or SYCL device is a host platform or host device respectively then the constructed SYCL context is a host context. Subsequently if the constructed SYCL context is a host context, then the associated SYCL platform must be a host platform and the constructed SYCL context must have a single associated SYCL device that is a host device.

A SYCL context can optionally be constructed with an async_handler parameter. In this case the async_handler provided is passed on to a SYCL queue to be used to report asynchronous SYCL exceptions.

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

```
namespace cl {
namespace sycl {
class context {
 public:
  explicit context(async_handler asyncHandler = {});

  context(const device &dev, async_handler asyncHandler = {});

  context(const platform &plt, async_handler asyncHandler = {});

  context(const vector_class<device> &deviceList,
    async_handler asyncHandler = {});

  context(cl_context clContext, async_handler asyncHandler = {});

  /* -- common interface members -- */

  cl_context get() const;

  bool is_host() const;

  platform get_platform() const;

  vector_class<device> get_devices() const;

  template <info::context param>
  typename info::param_traits<info::context, param>::return_type get_info() const;
};
}  // namespace sycl
}  // namespace cl
```

Constructor	Description
`explicit context(async_handler asyncHandler = {})`	Constructs a SYCL `context` instance using an instance of `default_selector` to select the associated SYCL `platform` and `device`(s). One of the SYCL `device`s that is associated with this SYCL `context` must be the SYCL `device` that is produced from the `default_selector` instance. The constructed SYCL `context` will use the `asyncHandler` parameter to handle exceptions.
`context(const device &dev, async_handler asyncHandler = {})`	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.
`context(const platform &plt, async_handler asyncHandler = {})`	Constructs a SYCL `context` instance using the `plt` parameter as the associated SYCL `platform` and the SYCL `device`(s) associated with the `plt` parameter as the associated SYCL `device`(s). The constructed SYCL `context` will use the `asyncHandler` parameter to handle exceptions.
`context(const vector_class<device> & deviceList, async_handler asyncHandler = {})`	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 `device`s in the `deviceList` parameter have the same associated SYCL `platform`. The constructed SYCL `context` will use the `asyncHandler` parameter to handle exceptions.
`context (cl_context clContext, async_handler asyncHandler = {})`	Constructs a SYCL `context` instance from an OpenCL `cl_context` in accordance with the requirements described in 4.3.1.
	End of table

Table 4.14: Constructors of the SYCL `context` class.

Member function	Description
`cl_context get ()const`	Returns a valid `cl_context` instance in accordance with the requirements described in 4.3.1.
	Continued on next page

Table 4.15: Member functions of the `context` class.

Member function	Description
`bool is_host ()const`	Returns true if this SYCL `context` is a host context.
`template <info::context param> typename info::param_traits<info::context, param>::return_type get_info()const`	Queries this SYCL `context` for information requested by the template parameter `param` using the `param_traits` class template to facilitate returning the appropriate type associated with the `param` parameter.
`platform get_platform()const`	Returns the SYCL `platform` that is associated with this SYCL `context`. The value returned must be equal to that returned by `get_info<info::context::platform>()`.
`vector_class<device> get_devices()const`	Returns a `vector_class` containing all SYCL `device`s that are associated with this SYCL `context`. The value returned must be equal to that returned by `get_info<info::context::devices>()`.
	End of table

Table 4.15: Member functions of the `context` class.

Context information descriptors

A SYCL `context` can be queried for all of the following information using the `get_info` member function. All SYCL `context`s have valid devices for them, including the SYCL host context. The available information is in Table 4.16. The interface of all available context descriptors in the appendix A.2.

Context Descriptors	Return type	Description
`info::context::reference_count`	`cl_uint`	Returns the reference count of the underlying OpenCL `cl_context` if this SYCL `context` is an OpenCL context. Returns `0` if this SYCL `context` is a host context.
`info::context::platform`	`platform`	Returns the SYCL `platform` associated with this SYCL `context`.
`info::context::devices`	`vector_class<device>`	Returns a `vector_class` containing the SYCL `device`s associated with this SYCL `context`.
		End of table

Table 4.16: Context information descriptors.

Device class

The SYCL `device` class encapsulates a single SYCL device on which kernels may be executed. A SYCL device may be an OpenCL device in which case it must encapsulate a valid underlying OpenCL `cl_device_id`, or it may be a SYCL host device in which case it must not.

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

The default constructor of the SYCL device class will construct a host device. The explicit constructor of the SYCL device class which takes a device_selector will construct a host device if select_device returns a host device, otherwise will construct an OpenCL device. The OpenCL interop constructor of the SYCL device class will construct an OpenCL device.

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 implementation defined and can be queried for through get_info().

For convenience there are member functions that check the device type. The member function is_host() returns true if the SYCL device is a host device and the member functions is_cpu(), is_gpu() and is_accelerator() return true if the device type is info::device_type::cpu, info::device_type::gpu or info::device_type::accelerator respectively.

The SYCL device class provides the common reference semantics (see Section 4.3.2).

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 Tables 4.17, 4.18 and 4.19 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
1  namespace cl {
2  namespace sycl {
3  class device {
4   public:
5    device();
6
7    explicit device(cl_device_id deviceId);
8
9    explicit device(const device_selector &deviceSelector);
10
11   /* -- common interface members -- */
12
13   cl_device_id get() const;
14
15   bool is_host() const;
16
17   bool is_cpu() const;
18
19   bool is_gpu() const;
20
21   bool is_accelerator() const;
22
23   platform get_platform() const;
24
25   template <info::device param>
26   typename info::param_traits<info::device, param>::return_type
```

```
27    get_info() const;
28
29    bool has_extension(const string_class &extension) const;
30
31    // Available only when prop == info::partition_property::partition_equally
32    template <info::partition_property prop>
33    vector_class<device> create_sub_devices(size_t nbSubDev) const;
34
35    // Available only when prop == info::partition_property::partition_by_counts
36    template <info::partition_property prop>
37    vector_class<device> create_sub_devices(const vector_class<size_t> &counts) const;
38
39    // Available only when prop == info::partition_property::partition_by_affinity_domain
40    template <info::partition_property prop>
41    vector_class<device> create_sub_devices(info::affinity_domain affinityDomain) const;
42
43    static vector_class<device> get_devices(
44        info::device_type deviceType = info::device_type::all);
45  };
46  }  // namespace sycl
47  }  // namespace cl
```

Constructor	Description
`device()`	Constructs a SYCL `device` instance as a host device.
`explicit device(const device_selector & deviceSelector)`	Constructs a SYCL `device` instance using the device selected by the `deviceSelector` provided.
`explicit device(cl_device_id deviceId)`	Constructs a SYCL `device` instance from an OpenCL `cl_device_id` in accordance with the requirements described in 4.3.1.
	End of table

Table 4.17: Constructors of the SYCL `device` class.

Member function	Description
`cl_device_id get()const`	Returns a valid `cl_device_id` instance in accordance with the requirements described in 4.3.1.
`platform get_platform()const`	Returns the associated SYCL `platform`. If this SYCL `device` is an OpenCL device then the SYCL `platform` must encapsulate the OpenCL `cl_plaform_id` associated with the underlying OpenCL `cl_device_id` of this SYCL `device`. If this SYCL `device` is a host device then the SYCL `platform` must be a host platform. The value returned must be equal to that returned by `get_info<info::device::platform>()`.
	Continued on next page

Table 4.18: Member functions of the SYCL `device` class.

Member function	Description
`bool is_host()const`	Returns true if this SYCL `device` is a host device.
`bool is_cpu()const`	Returns true if this SYCL `device` is an OpenCL device and the device type is `info::device_type::cpu`.
`bool is_gpu()const`	Returns true if this SYCL `device` is an OpenCL device and the device type is `info::device_type::gpu`.
`bool is_accelerator()const`	Returns true if this SYCL `device` is an OpenCL device and the device type is `info::device_type::accelerator`.
`template <info::device param> typename info::param_traits<info::device, param>::return_type get_info()const`	Queries this SYCL `device` for information requested by the template parameter `param`. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.20 to facilitate returning the type associated with the `param` parameter.
`bool has_extension (const string_class &extension) const`	Returns true if this SYCL `device` supports the extension queried by the `extension` parameter.
`template <info::partition_property prop> vector_class<device> create_sub_devices( size_t nbSubDev)const`	Available only when prop is `info::partition_property::partition_equally`. Returns a vector_class of sub devices partitioned from this SYCL `device` equally based on the `nbSubDev` parameter. If this SYCL `device` does not support `info::partition_property::partition_equally` a `feature_not_supported` exception must be thrown.
`template <info::partition_property prop> vector_class<device> create_sub_devices( const vector_class<size_t> &counts)const`	Available only when prop is `info::partition_property::partition_by_count`. Returns a vector_class of sub devices partitioned from this SYCL `device` by count sizes based on the `counts` parameter. If the SYCL `device` does not support `info::partition_property::partition_by_count` a `feature_not_supported` exception must be thrown.

Continued on next page

Table 4.18: Member functions of the SYCL `device` class.

Member function	Description
`template <info::partition_property prop> vector_class<device> create_sub_devices( info::affinity_domain affinityDomain)const`	Available only when prop is `info::partition_property:: partition_by_affinity_domain`. Returns a vector_class of sub devices partitioned from this SYCL `device` by affinity domain based on the `affinityDomain` parameter. Partitions the device into sub devices based upon the affinity domain. If the SYCL `device` does not support `info::partition_property ::partition_by_affinity_domain` or the SYCL `device` does not support `info::affinity_domain` provided a `feature_not_supported` exception must be thrown.
	End of table

Table 4.18: Member functions of the SYCL `device` class.

Static member function	Description
`static vector_class<device> get_devices( info::device_type deviceType = info::device_type::all)`	Returns a `vector_class` containing all SYCL `device`s available in the system of the device type specified by the parameter `deviceType`. The returned `vector_class` must contain a single SYCL `device` that is a host device, permitted by the `deviceType` parameter.
	End of table

Table 4.19: Static member functions of the SYCL `device` class.

Device information descriptors

A SYCL `device` can be queried for all of the following information using the `get_info` member function. All SYCL `device`s must have valid values for every query, including a host device. The information that can be queried is described in Table 4.20. The interface for all information types and enumerations are described in appendix A.3.

Device descriptors	Return type	Description
`info::device::device_type`	`info:: device_type`	Returns the device type. Must not return `info ::device_type::all`.
`info::device::vendor_id`	`cl_uint`	Returns a unique vendor device identifier. An example of a unique device identifier could be the PCIe ID.
`info::device:: max_compute_units`	`cl_uint`	Returns the number of parallel compute units available. The minimum value is 1.
		Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device:: max_work_item_dimensions	cl_uint	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 device is not of device type info::device_type::custom.
info::device:: max_work_item_sizes	id<3>	Returns the maximum number of work-items that are permitted in each dimension of the work-group of the nd_range. The minimum value is (1, 1, 1) for devices that are not of device type info::device_type::custom.
info::device:: max_work_group_size	size_t	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.
info::device:: preferred_vector_width_char info::device:: preferred_vector_width_short info::device:: preferred_vector_width_int info::device:: preferred_vector_width_long info::device:: preferred_vector_width_float info::device:: preferred_vector_width_double info::device:: preferred_vector_width_half	cl_uint	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_width_double if the cl_khr_fp64 extension is not supported by this SYCL device and must return 0 for info::device::preferred_vector_width_half if the cl_khr_fp16 extension is not supported by this SYCL device.
info::device:: native_vector_width_char info::device:: native_vector_width_short info::device:: native_vector_width_int info::device:: native_vector_width_long info::device:: native_vector_width_float info::device:: native_vector_width_double info::device:: native_vector_width_half	cl_uint	Returns the native ISA vector width. The vector width is defined as the number of scalar elements that can be stored in the vector. Must return 0 for info::device::preferred_width_double if the cl_khr_fp64 extension is not supported by this SYCL device and must return 0 for info::device::preferred_vector_width_half if the cl_khr_fp16 extension is not supported by this SYCL device.
info::device:: max_clock_frequency	cl_uint	Returns the maximum configured clock frequency of this SYCL device in MHz.
info::device::address_bits	cl_uint	Returns the default compute device address space size specified as an unsigned integer value in bits. Must return either 32 or 64.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::max_mem_alloc_size	cl_ulong	Returns the maximum size of memory object allocation in bytes. The minimum value is max (1/4th of info::device::global_mem_size ,128*1024*1024) if this SYCL device is not of device type info::device_type::custom.
info::device::image_support	bool	Returns true if images are supported by this SYCL device and false if they are not.
info::device::max_read_image_args	cl_uint	Returns the maximum number of simultaneous image objects that can be read from by a kernel. The minimum value is 128 if info::device::image_support returns true for this SYCL device.
info::device::max_write_image_args	cl_uint	Returns the maximum number of simultaneous image objects that can be written to by a kernel. The minimum value is 8 if info::device::image_support returns true for this SYCL device.
info::device::image2d_max_width	size_t	Returns the maximum width of a 2D image or 1D image in pixels. The minimum value is 8192 if info::device::image_support returns true for this SYCL device.
info::device::image2d_max_height	size_t	Returns the maximum height of a 2D image in pixels. The minimum value is 8192 if info::device::image_support returns true for this SYCL device.
info::device::image3d_max_width	size_t	Returns the maximum width of a 3D image in pixels. The minimum value is 2048 if info::device::image_support returns true for this SYCL device.
info::device::image3d_max_height	size_t	Returns the maximum height of a 3D image in pixels. The minimum value is 2048 if info::device::image_support returns true for this SYCL device.
info::device::image3d_max_depth	size_t	Returns the maximum depth of a 3D image in pixels. The minimum value is 2048 if info::device::image_support returns true for this SYCL device.
info::device::image_max_buffer_size	size_t	Returns the number of pixels for a 1D image created from a buffer object. The minimum value is 65536 if info::device::image_support if info::device::image_support returns true for this SYCL device. Note that this information is intended for OpenCL interoperability only as this feature is not supported in SYCL.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::image_max_array_size	size_t	Returns the maximum number of images in a 1D or 2D image array. The minimum value is 2048 if info::device::image_support returns true for this SYCL device.
info::device::max_samplers	cl_uint	Returns the maximum number of samplers that can be used in a kernel. The minimum value is 16 if info::device::image_support returns true for this SYCL device.
info::device::max_parameter_size	size_t	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 info::device_type::custom. For this minimum value, only a maximum of 128 arguments can be passed to a kernel.
info::device::mem_base_addr_align	cl_uint	Returns the minimum value in bits of the largest supported SYCL built-in data type if this SYCL device is not of device type info::device_type::custom.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::half_fp_config	vector_class< info::fp_config >	Returns a vector_class of info::fp_config describing the half precision floating-point capability of this SYCL device. The vector_class may contain zero or more of the following values: • 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. • info::fp_config::soft_float: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. If half precision is supported by this SYCL device (i.e. the cl_khr_fp16 extension is supported) there is no minimum floating-point capability. If half support is not supported the returned vector_class must be empty.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::single_fp_config	vector_class< info::fp_config >	Returns a vector_class of info::fp_config describing the single precision floating-point capability of this SYCL device. The vector_class must contain one or more of the following values: • 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. • 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.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::double_fp_config	vector_class< info::fp_config >	Returns a vector_class of info::fp_config describing the single precision floating-point capability of this SYCL device. The vector_class may contain zero or more of the following values: • info::fp_config::denorm: denorms are supported. • info::fp_config::inf_nan: INF and 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::soft_float: basic floating-point operations (such as addition, subtraction, multiplication) are implemented in software. If double precision is supported by this SYCL device (i.e. the cl_khr_fp64 extension is supported) and this SYCL device is not of type info::device_type::custom then the minimum floating-point capability must be: info::fp_config::fma, info::fp_config::round_to_nearest, info::fp_config::round_to_zero, info::fp_config::round_to_inf, info::fp_config::inf_nan and info::fp_config::denorm. If double support is not supported the returned vector_class must be empty.
info::device::global_mem_cache_type	info::global_mem_cache_type	Returns the type of global memory cache supported.
info::device::global_mem_cache_line_size	cl_uint	Returns the size of global memory cache line in bytes.
info::device::global_mem_cache_size	cl_ulong	Returns the size of global memory cache in bytes.
info::device::global_mem_size	cl_ulong	Returns the size of global device memory in bytes.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::max_constant_buffer_size	cl_ulong	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 info::device_type::custom.
info::device::max_constant_args	cl_uint	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 info::device_type::custom.
info::device::local_mem_type	info::local_mem_type	Returns the type of local memory supported. This can be info::local_mem_type::local implying dedicated local memory storage such as SRAM, or info::local_mem_type::global. If this SYCL device is of type info::device_type::custom this can also be info::local_mem_type::none, indicating local memory is not supported.
info::device::local_mem_size	cl_ulong	Returns the size of local memory arena in bytes. The minimum value is 32 KB if this SYCL device is not of type info::device_type::custom.
info::device::error_correction_support	bool	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.
info::device::host_unified_memory	bool	Returns true if the device and the host have a unified memory subsystem and returns false otherwise.
info::device::profiling_timer_resolution	size_t	Returns the resolution of device timer in nanoseconds.
info::device::is_endian_little	bool	Returns true if this SYCL device is a little endian device and returns false otherwise.
info::device::is_available	bool	Returns true if the SYCL device is available and returns false if the device is not available.
info::device::is_compiler_available	bool	Returns false if the implementation does not have a compiler available to compile the program source. An OpenCL device that conforms to the OpenCL Embedded Profile may not have an online compiler available.
info::device::is_linker_available	bool	Returns false if the implementation does not have a linker available. An OpenCL device that conforms to the OpenCL Embedded Profile may not have a linker available. However, it needs to be true if info::device::is_compiler_available returns true for this SYCL device.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::execution_capabilities	vector_class<info::execution_capability>	Returns a vector_class of the info::execution_capability describing the supported execution capabilities. Note that this information is intended for OpenCL interoperability only as SYCL only supports info::execution_capability::exec_kernel.
info::device::queue_profiling	bool	Returns true if this device supports queue profiling.
info::device::built_in_kernels	vector_class<string_class>	Returns a vector_class of built-in OpenCL kernels supported by this SYCL device.
info::device::platform	platform	Returns the SYCL platform associated with this SYCL device.
info::device::name	string_class	Returns the device name of this SYCL device.
info::device::vendor	string_class	Returns the vendor of this SYCL device.
info::device::driver_version	string_class	Returns the OpenCL software driver version as a string_class in the form: major_number.minor_number, if this SYCL device is an OpenCL device. Must return a string_class with the value "1.2" if this SYCL device is a host device.
info::device::profile	string_class	Returns the OpenCL profile as a string_class, if this SYCL device is an OpenCL device. 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. Must return a string_class with the value ”FULL PROFILE” if this is a host device.
info::device::version	string_class	Returns the SYCL version as a string_class in the form: <major_version>.<minor_version>. If this SYCL device is a host device, the ¡major_version¿.¡minor_version¿ value returned must be "1.2".
info::device::opencl_c_version	string_class	Returns a string_class describing the OpenCL C version that is supported by the OpenCL C compiler of this device. Note that this information is intended for OpenCL interoperability only as SYCL kernel functions are compiled offline.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device::extensions	vector_class< string_class>	Returns a vector_class 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: • cl_khr_int64_base_atomics • cl_khr_int64_extended_atomics • cl_khr_3d_image_writes • cl_khr_fp16 • cl_khr_gl_sharing • cl_khr_gl_event • cl_khr_d3d10_sharing • cl_khr_dx9_media_sharing • cl_khr_d3d11_sharing • cl_khr_depth_images • cl_khr_gl_depth_images • cl_khr_gl_msaa_sharing • cl_khr_image2d_from_buffer • cl_khr_initialize_memory • cl_khr_context_abort • cl_khr_spir 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: • cl_khr_global_int32_base_atomics • cl_khr_global_int32_extended_atomics • cl_khr_local_int32_base_atomics • cl_khr_local_int32_extended_atomics • cl_khr_byte_addressable_store • cl_khr_fp64 (for backward compatibility if double precision is supported) Please refer to the OpenCL 1.2 Extension Specification for a detailed description of these extensions.
info::device:: printf_buffer_size	size_t	Returns the maximum size of the internal buffer that holds the output of printf calls from a kernel. The minimum value is 1 MB if info ::device::profile returns true for this SYCL device.

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
info::device:: preferred_interop_user_sync	bool	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.
info::device::parent_device	device	Returns the parent SYCL device to which this sub-device is a child if this is a sub-device. Must throw a invalid_object_error SYCL exception if this SYCL device is not a sub device.
info::device:: partition_max_sub_devices	cl_uint	Returns the maximum number of subdevices that can be created when this SYCL device is partitioned. The value returned cannot exceed the value returned by info::device:: device_max_compute_units.
info::device:: partition_properties	vector_class <info:: partition_property>	Returns the partition properties supported by this SYCL device; a vector of info:: partition_property. If this SYCL device cannot be partitioned into at least two sub devices then the returned vector must be empty.
info::device:: partition_affinity_domains	vector_class <info:: parition_affini -ty_domain>	Returns a vector_class of the partition affinity domains supported by this SYCL device when partitioning with info::partition_property ::parition_by_affinity_domain.
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 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

Continued on next page

Table 4.20: Device information descriptors.

Device descriptors	Return type	Description
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 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 • info::partition_affinity_domain::next_partitionable
info::device::reference_count	cl_uint	Returns the device reference count. If the device is not a sub-device the value returned must be 1.
		End of table

Table 4.20: Device information descriptors.

Queue class

The SYCL queue class encapsulates a single SYCL queue which schedules kernels on a SYCL device. A SYCL queue may be an OpenCL queue in which case it must encapsulate at least one valid underlying OpenCL cl_command_queue, or it may be a SYCL host queue in which case it must not. The underlying OpenCL cl_command_queue(s) may execute either in-order or out-of-order, however the SYCL queue must behave as an out-of-order queue.

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 schedules command groups asynchronously, so any errors in the submission of a command group are handled by throwing synchronous SYCL exceptions. Any exceptions from the command group after it has been submitted are handled by throwing asynchronous SYCL exceptions to an async_handler on calling throw_asynchronous or wait_and_throw, or on destruction of the SYCL queue, if one was provided when the SYCL queue was constructed.

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 (see Section 4.6.1), therefore the constructed SYCL queue could be either a host queue or a device queue. All other constructors construct a host or device queue, 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.

With the exception of the interoperability constructor, each constructor takes as the last parameter an optional SYCL property_list to provide properties to the SYCL queue.

The SYCL queue class provides the common reference semantics (see Section 4.3.2).

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 Tables 4.21 and 4.22 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
1  namespace cl {
2  namespace sycl {
3  class queue {
4   public:
5    explicit queue(const property_list &propList = {});
6
7    queue(const async_handler &asyncHandler,
8      const property_list &propList = {});
9
10   queue(const device_selector &deviceSelector,
11     const property_list &propList = {});
12
13   queue(const device_selector &deviceSelector,
14       const async_handler &asyncHandler, const property_list &propList = {});
15
16   queue(const device &syclDevice, const property_list &propList = {});
17
18   queue(const device &syclDevice, const async_handler &asyncHandler,
19     const property_list &propList = {});
20
21   queue(const context &syclContext, const device_selector &deviceSelector,
22     const property_list &propList = {});
23
24   queue(const context &syclContext, const device_selector &deviceSelector,
25     const async_handler &asyncHandler, const property_list &propList = {});
26
27   queue(cl_command_queue clQueue, const context& syclContext,
28     const async_handler &asyncHandler = {});
29
30   /* -- common interface members -- */
31
32   /* -- property interface members -- */
33
34   cl_command_queue get() const;
35
```

```
36    context get_context() const;
37
38    device get_device() const;
39
40    bool is_host() const;
41
42    template <info::queue param>
43    typename info::param_traits<info::queue, param>::return_type get_info() const;
44
45    template <typename T>
46    event submit(T cgf);
47
48    template <typename T>
49    event submit(T cgf, const queue &secondaryQueue);
50
51    void wait();
52
53    void wait_and_throw();
54
55    void throw_asynchronous();
56  };
57  }  // namespace sycl
58  }  // namespace cl
```

Constructor	Description
`explicit queue(const property_list &propList = {})`	Constructs a SYCL `queue` instance using the device returned by an instance of `default_selector`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.
`queue(const async_handler &asyncHandler, const property_list &propList = {})`	Constructs a SYCL `queue` instance with an `async_handler` using the device returned by an instance of `default_selector`. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.
`queue(const device_selector &deviceSelector, const property_list &propList = {})`	Constructs a SYCL `queue` instance using the device returned by the `deviceSelector` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.
`queue(const device_selector &deviceSelector, const async_handler &asyncHandler, const property_list &propList = {})`	Constructs a SYCL `queue` instance with an `async_handler` using the device returned by the `deviceSelector` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.

Continued on next page

Table 4.21: Constructors of the `queue` class.

Constructor	Description
`queue(const device &syclDevice, const property_list &propList = {})`	Constructs a SYCL `queue` instance using the `syclDevice` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.
`queue(const device &syclDevice, const async_handler &asyncHandler, const property_list &propList = {})`	Constructs a SYCL `queue` instance with an `async_handler` using the `syclDevice` provided. Zero or more properties can be provided to the constructed SYCL `queue` via an instance of `property_list`.
`queue(const context &syclContext, const device_selector &deviceSelector, const property_list &propList = {})`	Constructs a SYCL `queue` instance that is associated with the `syclContext` provided, using the device returned by the `deviceSelector` provided. Must throw an `invalid_object_error` SYCL exception 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`.
`queue(const context &syclContext, const device_selector &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 `deviceSelector` provided. Must throw an `invalid_object_error` SYCL exception 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`.
`queue(cl_command_queue clQueue, const context &syclContext, const async_handler &asyncHandler = {})`	Constructs a SYCL `queue` instance with an optional `async_handler` from an OpenCL `cl_command_queue` in accordance with the requirements described in 4.3.1.
	End of table

Table 4.21: Constructors of the `queue` class.

Member function	Description
`cl_command_queue get()const`	Returns a valid `cl_command_queue` instance in accordance with the requirements described in 4.3.1.
`context get_context ()const`	Returns the SYCL queue's context. Reports errors using SYCL exception classes. The value returned must be equal to that returned by `get_info<info::queue::context>()`.
	Continued on next page

Table 4.22: Member functions for class queue.

Member function	Description
`device get_device ()const`	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<info::queue::devices>()`.
`bool is_host ()const`	Returns whether the queue is executing on a SYCL host device.
`void wait()`	Performs a blocking wait for the completion all enqueued tasks in the queue. Synchronous errors will be reported through SYCL exceptions.
`void wait_and_throw ()`	Performs a blocking wait for the completion all enqueued tasks in the queue. Synchronous errors will be reported via SYCL exceptions. Asynchronous errors will be passed to the async_handler passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.
`void throw_asynchronous ()`	Checks to see if any asynchronous errors have been produced by the queue and if so reports them by passing them to the async_handler passed to the queue on construction. If no `async_handler` was provided then asynchronous exceptions will be lost.

`template <info::queue param>`
`typename info::param_traits`
`<info::queue, param>::return_type`
`get_info ()const`
	Queries the platform for *cl_command_queue_info*

`template <typename T>` `event submit(T cgf)`	Submit a command group function object to the queue, in order to be scheduled for execution on the device.

`template <typename T>`
`event submit(T cgf,`
`queue & secondaryQueue)`
	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.

	End of table

Table 4.22: Member functions for class queue.

Queue information descriptors

A SYCL command queue can be queried for all of the following information using the `get_info` function. All SYCL queues have valid queries for them, including the SYCL host queue. The available information is in

Table 4.23. The interface of all available device descriptors is in appendix A.4.

Queue Descriptors	Return type	Description
info::queue::context	context	Returns the SYCL context associated with this SYCL queue.
info::queue::device	device	Returns the SYCL device associated with this SYCL queue.
info::queue::reference_count	cl_uint	Return the command-queue reference count.
		End of table

Table 4.23: Queue information descriptors.

Queue Properties

The properties that can be provided when constructing the SYCL queue class are describe in Table 4.24.

Property	Description
property::queue::enable_profiling	The enable_profiling property adds the requirement that the SYCL runtime must capture profiling information for the command groups that are submitted from this SYCL queue and provide said information via the SYCL event class get_profiling_info member function, if the associated SYCL device supports queue profiling (i.e. the info::device::queue_profiling info parameter returns true).
	End of table

Table 4.24: Properties supported by the SYCL queue class.

The constructors of the queue property classes are listed in Table 4.25.

Constructor	Description
property::queue::enable_profiling::enable_profiling()	Constructs a SYCL enable_profiling property instance.
	End of table

Table 4.25: Constructors of the queue property classes.

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 returned through exceptions.

- **Asynchronous errors** are those that are produced through callback functions only. These will be stored within the queue's context until they are dispatched to the context's asynchronous error handler, the async_handler. If a queue is constructed with a user-supplied context, then it is this context's asynchronous error handler to which asynchronous errors are reported. If a queue is constructed without a user-supplied context, then the queue's constructor can be supplied with a queue-specific asynchronous error handler which will be used to construct the queue's context. To ensure that such errors are processed predictably in a known host thread, these errors are only passed to the asynchronous error handler on request when either `wait_and_throw` is called or when `throw_asynchronous` is called. If no asynchronous error handler is passed to the queue or its context on construction, then such errors go unhandled, much as they would if no callback were passed to an OpenCL context.

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

Event class

An *event* in SYCL abstracts the `cl_event` object in OpenCL. In OpenCL the events mechanism is comprised of low-level event objects that the developer uses to synchronize memory transfers, enqueues of kernels and signaling barriers.

In SYCL, events are an abstraction of the OpenCL event objects, but they retain the features and functionality of the OpenCL event mechanism. They accommodate synchronization between different contexts, devices and platforms. It is the responsibility of the SYCL implementation to ensure that when SYCL events are used in OpenCL queues, the correct synchronization points are created to allow cross-platform or cross-device synchronization.

Since data management and storage is handled by the SYCL runtime, the `event` class is used for providing the appropriate interface for OpenCL/SYCL interoperability. In the case where SYCL objects contain OpenCL memory objects created outside of the SYCL mechanism, then events can be used to provide the SYCL runtime with the initial events that it has to synchronize against. However, the events mechanism does not provide full interoperability with OpenCL during SYCL code execution. Interoperability is achieved by using the synchronization rules with the `buffer` and `image` classes.

A SYCL event can be constructed from an OpenCL event or can return an OpenCL event. A SYCL event can also be 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 actions associated with the command group execution.

The SYCL `event` class provides the common reference semantics (see Section 4.3.2).

The constructors and member functions of the SYCL `event` class are listed in Tables 4.26 and 4.27, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

```
1  namespace cl {
2  namespace sycl {
3
4  class event {
5   public:
6    event();
```

```
7
8     event(cl_event clEvent, const context& syclContext);
9
10    /* -- common interface members -- */
11
12    cl_event get();
13
14    bool is_host() const;
15
16    vector_class<event> get_wait_list();
17
18    void wait();
19
20    static void wait(const vector_class<event> &eventList);
21
22    void wait_and_throw();
23
24    static void wait_and_throw(const vector_class<event> &eventList);
25
26    template <info::event param>
27    typename info::param_traits<info::event, param>::return_type get_info() const;
28
29    template <info::event_profiling param>
30    typename info::param_traits<info::event_profiling, param>::return_type get_profiling_info()
          const;
31  };
32
33  }  // namespace sycl
34  }  // namespace cl
```

Constructor	Description
`event ()`	Constructs a ready SYCL `event`. If the constructed SYCL `event` is waited on it will complete immediately.
`event (cl_event clEvent, const context& syclContext)`	Constructs a SYCL `event` instance from an OpenCL `cl_event` in accordance with the requirements described in 4.3.1. The `syclContext` must match the OpenCL context associated with the clEvent.
	End of table

Table 4.26: Constructors of the `event` class.

Member function	Description
`cl_event get()`	Returns a valid `cl_event` instance in accordance with the requirements described in 4.3.1.
`bool is_host()const`	Returns true if this SYCL `event` is a host event.
	Continued on next page

Table 4.27: Member functions for the `event` class.

Member function	Description
`vector_class<event> get_wait_list()`	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.
`void wait()`	Wait for the event and the command associated with it to complete.
`void wait_and_throw()`	Wait for the event and the command associated with it to complete. If any uncaught asynchronous errors occurred on the context (or contexts) that the event is waiting on executions from, then will also call that context's asynchronous error handler with those errors.
`static void wait( const vector_class<event> &eventList)`	Synchronously wait on a list of events.
`static void wait_and_throw( const vector_class<event> &eventList)`	Synchronously wait on a list of events. If any uncaught asynchronous errors occurred on the context (or contexts) that the events are waiting on executions from, then will also call those contexts' asynchronous error handlers with those errors.
`template <info::event param> typename info::param_traits <info::event, param>::return_type get_info()const`	Queries this SYCL `event` for information requested by the template parameter `param` . Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.28 to facilitate returning the type associated with the `param` parameter.

Continued on next page

Table 4.27: Member functions for the `event` class.

Member function	Description
`template <info::event_profiling param> typename info::param_traits <info::event_profiling, param>::return_type get_profiling_info ()const`	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 has yet to finish execution. Calls to `get_profiling_info` must throw an `invalid_object_error` SYCL exception 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. Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.29 to facilitate returning the type associated with the `param` parameter.
	End of table

Table 4.27: Member functions for the `event` class.

Event information and profiling descriptors

A SYCL event can be queried for all of the following information using the `get_info` function. The available information is in Table 4.28. Profiling information available is in Table 4.29. The interface of all available event and profiling descriptors is in appendix A.7.

Event Descriptors	Return type	Description
`info::event::command_execution_status`	`info::event_command_status`	Returns the event status of the command group associated with this SYCL `event` or the event status of the underlying OpenCL event if this SYCL `event` instance was constructed with the interoperability constructor.
`info::event::reference_count`	`cl_uint`	Return the reference count of the event.
		End of table

Table 4.28: Event class information descriptors.

Event information profiling descriptor	Return type	Description
info::event_profiling::command_submit	cl_ulong	Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group was submitted.
info::event_profiling::command_start	cl_ulong	Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group started executing.
info::event_profiling::command_end	cl_ulong	Returns an implementation defined 64-bit value describing the time in nanoseconds when the associated command group finished executing.
		End of table

Table 4.29: Profiling information descriptors for the SYCL event class.

Data access and storage in SYCL

In SYCL, 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 are different from OpenCL buffers and images in that they can be bound to more than one device or context and they get destroyed when they go out-of-scope. 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.

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.

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

When an allocator returns a nullptr, the runtime could not create 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 OpenCL contexts).

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).

Default Allocators

A default allocator is always defined by the implementation, and 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.

Allocators	Description
`buffer_allocator`	It is the default buffer allocator used by the runtime, when no allocator is defined by the user.
`image_allocator`	It is the default allocator used by the runtime for the SYCL `image` class when no allocator is provided by the user. The `image_allocator` is required allocate in elements of `byte`.
	End of table

Table 4.30: SYCL Default Allocators.

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

Buffers

The `buffer` class defines a shared array of one, two or three dimensions that can be used by kernels in queues 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 OpenCL buffer object, and all OpenCL buffer memory objects may be temporary for use within a command group on a specific device. The only exception to this rule is when a buffer is constructed from a `cl_mem` object to interoperate with OpenCL. Use of an interoperability buffer on a queue mapping to a context other than that in which the `cl_mem` was created is an error.

Note that if no source data is provided for a buffer, the buffer uses uninitialized memory for performance reasons. So it is up to the programmer to explicitly construct the objects in this case if required.

More generally, since the value type of a buffer is required to be trivially copyable, there is no constructor or destructor called in any case.

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 sec 4.3.2) 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 get_count 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 4.3.2).

Buffer Interface

The constructors and member functions of the SYCL buffer class template are listed in Tables 4.31 and 4.32, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

Each constructor excluding the interoperability 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 will be used (see 4.7.1.1).

```
1  namespace cl {
2  namespace sycl {
3  namespace property {
4  namespace buffer {
5  class use_host_ptr {
6    public:
7      use_host_ptr() = default;
8  };
9
10 class use_mutex {
11   public:
12     use_mutex(mutex_class &mutexRef);
13
14     mutex_class *get_mutex_ptr() const;
15 };
16
17 class context_bound {
18   public:
19     context_bound(context boundContext);
20
21     context get_context() const;
22 };
23 }  // namespace buffer
24 }  // namespace property
25
26 template <typename T, int dimensions = 1,
27           typename AllocatorT = cl::sycl::buffer_allocator>
28 class buffer {
29  public:
30   using value_type = T;
```

```
31    using reference = value_type &;
32    using const_reference = const value_type &;
33    using allocator_type = AllocatorT;
34
35    buffer(const range<dimensions> &bufferRange,
36           const property_list &propList = {});
37
38    buffer(const range<dimensions> &bufferRange, AllocatorT allocator,
39           const property_list &propList = {});
40
41    buffer(T *hostData, const range<dimensions> &bufferRange,
42           const property_list &propList = {});
43
44    buffer(T *hostData, const range<dimensions> &bufferRange,
45           AllocatorT allocator, const property_list &propList = {});
46
47    buffer(const T *hostData, const range<dimensions> &bufferRange,
48           const property_list &propList = {});
49
50    buffer(const T *hostData, const range<dimensions> &bufferRange,
51           AllocatorT allocator, const property_list &propList = {});
52
53    buffer(const shared_ptr_class<T> &hostData,
54           const range<dimensions> &bufferRange, AllocatorT allocator,
55           const property_list &propList = {});
56
57    buffer(const shared_ptr_class<T> &hostData,
58           const range<dimensions> &bufferRange,
59           const property_list &propList = {});
60
61    template <class InputIterator>
62    buffer<T, 1>(InputIterator first, InputIterator last, AllocatorT allocator,
63                 const property_list &propList = {});
64
65    template <class InputIterator>
66    buffer<T, 1>(InputIterator first, InputIterator last,
67                 const property_list &propList = {});
68
69    buffer(buffer<T, dimensions, AllocatorT> b, const id<dimensions> &baseIndex,
70           const range<dimensions> &subRange);
71
72    /* Available only when: dimensions == 1. */
73    buffer(cl_mem clMemObject, const context &syclContext,
74           event availableEvent = {});
75
76    /* -- common interface members -- */
77
78    /* -- property interface members -- */
79
80    range<dimensions> get_range() const;
81
82    size_t get_count() const;
83
84    size_t get_size() const;
85
```

```
86    AllocatorT get_allocator() const;
87
88    template <access::mode mode, access::target target = access::target::global_buffer>
89    accessor<T, dimensions, mode, target> get_access(
90        handler &commandGroupHandler);
91
92    template <access::mode mode>
93    accessor<T, dimensions, mode, access::target::host_buffer> get_access();
94
95    template <access::mode mode, access::target target = access::target::global_buffer>
96    accessor<T, dimensions, mode, target> get_access(
97        handler &commandGroupHandler, range<dimensions> accessRange,
98        id<dimensions> accessOffset = {});
99
100   template <access::mode mode>
101   accessor<T, dimensions, mode, access::target::host_buffer> get_access(
102     range<dimensions> accessRange, id<dimensions> accessOffset = {});
103
104   template <typename Destination = std::nullptr_t>
105   void set_final_data(Destination finalData = std::nullptr);
106
107   void set_write_back(bool flag = true);
108
109   bool is_sub_buffer() const;
110
111   template <typename ReinterpretT, int ReinterpretDim>
112   buffer<ReinterpretT, ReinterpretDim, AllocatorT>
113   reinterpret(range<ReinterpretDim> reinterpretRange) const;
114
115 };
116 }  // namespace sycl
117 }  // namespace cl
```

Constructor	Description
`buffer(const range<dimensions> & bufferRange, const property_list &propList = {})`	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. 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 `buffer` via an instance of `property_list`.

Continued on next page

Table 4.31: Constructors of the `buffer` class.

Constructor	Description
`buffer(const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})`	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. 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 `buffer` via an instance of `property_list`.
`buffer(T* hostData, const range<dimensions> & bufferRange, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's 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`.
`buffer(T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's 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`.

Continued on next page

Table 4.31: Constructors of the `buffer` class.

Constructor	Description
`buffer(const T* hostData, const range<dimensions> & bufferRange, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's lifetime. The constructed SYCL `buffer` will use a default constructed `AllocatorT` 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 after its destruction, unless there is another final data address given 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`.
`buffer(const T* hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's 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 after its destruction, unless there is another final data address given 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`.

Continued on next page

Table 4.31: Constructors of the `buffer` class.

Constructor	Description
`buffer(const shared_ptr_class<T> &hostData, const range<dimensions> & bufferRange, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's 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`.
`buffer(const shared_ptr_class<void> &hostData, const range<dimensions> & bufferRange, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `buffer` instance with the `hostData` parameter provided. The ownership of this memory is given to the constructed SYCL `buffer` for the duration of it's 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`.
`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 written back to the same iterator set if the iterator is not a const 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`.
`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 written back to the same iterator set if the iterator is not a const 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`.

Continued on next page

Table 4.31: Constructors of the `buffer` class.

Constructor	Description
`buffer(buffer<T, dimensions, AllocatorT> &b, const id<dimensions> & baseIndex, const range<dimensions> & subRange)`	Create a new sub-buffer without allocation to have separate accessors later. `b` is the buffer with the real data. `baseIndex` specifies the origin of the sub-buffer inside the buffer `b`. `subRange` specifies the size of the sub-buffer. The offset and range specified by `baseIndex` and `subRange` together must represent a contiguous region of the original SYCL `buffer`. The total size of the sub-`buffer` being constructed must be a multiple of the memory base address alignment of each SYCL `device` that is executed on, otherwise the SYCL runtime must throw an asynchronous `invalid_object_error` SYCL exception. This value is retrievable via the SYCL `device` class info query `info::device::mem_base_addr_align`.
`buffer(cl_mem clMemObject, const context &syclContext, event availableEvent = {})`	Available only when: `dimensions == 1`. Constructs a SYCL `buffer` instance from an OpenCL `cl_mem` in accordance with the requirements described in 4.3.1. The instance of the SYCL `buffer` class template being constructed must wait for the SYCL `event` parameter, if one is provided, `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.
	End of table

Table 4.31: Constructors of the `buffer` class.

Member function	Description
`range<dimensions> get_range()const`	Return a range object representing the size of the buffer in terms of number of elements in each dimension as passed to the constructor.
`size_t get_count()const`	Returns the total number of elements in the buffer. Equal to `get_range()[0] * ... * get_range()[dimensions-1]`.
`size_t get_size()const`	Returns the size of the buffer storage in bytes. Equal to `get_count()*sizeof(T)`.
`AllocatorT get_allocator()const`	Returns the allocator provided to the buffer.
	Continued on next page

Table 4.32: Member functions for the `buffer` class.

Member function	Description
`template<access::mode mode, access::target target = access::target::global_buffer> accessor<T, dimensions, mode, target> get_access(handler &commandGroupHandler)`	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 `access::target::global_buffer` or `access::constant_buffer`.
`template<access::mode mode> accessor<T, dimensions, mode, access::target::host_buffer> get_access()`	Returns a valid host accessor to the buffer with the specified access mode and target.
`template<access::mode mode, access::target target= access::target::global_buffer> accessor<T, dimensions, mode, target> get_access(handler &commandGroupHandler, range<dimensions> accessRange, id<dimensions> accessOffset = {})`	Returns a valid accessor to the buffer with the specified access mode and target in the command group buffer. Only the values starting from the given offset and up to the given range are guaranteed to be updated. The value of target can be `access::target::global_buffer` or `access::constant_buffer`.
`template<access::mode mode> accessor<T, dimensions, mode, access::target::host_buffer> get_access(range<dimensions> accessRange, id<dimensions> accessOffset = {})`	Returns a valid host accessor to the buffer with the specified access mode and target. Only the values starting from the given offset and up to the given range are guaranteed to be updated. The value of target can only be `access::target::host_buffer`.
`template <typename Destination = std::nullptr_t> void set_final_data(Destination finalData = std::nullptr)`	The `finalData` 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 `weak_ptr_class<T>`. 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 `Destination` is `std::nullptr_t`, then the copy back will not happen.
`void set_write_back(bool flag = true)`	This method allows dynamically forcing or canceling the write-back of the data of a buffer on destruction according to the value of `flag`. Forcing the write-back is similar to what happens during a normal write-back as described in § 4.7.2.3 and 4.7.4. If there is nowhere to write-back, using this function does not have any effect.
`bool is_sub_buffer()const`	Returns true if this SYCL `buffer` is a sub-buffer, otherwise returns false.

Continued on next page

Table 4.32: Member functions for the `buffer` class.

Member function	Description
`template <typename ReinterpretT, int ReinterpretDim> buffer<ReinterpretT, ReinterpretDim, AllocatorT> 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`. Must throw an `invalid_object_error` SYCL exception if the total size in bytes represented by the type and range of the reinterpreted SYCL `buffer` does not equal the total size in bytes represented by the type and range of this SYCL `buffer`.
	End of table

Table 4.32: Member functions for the `buffer` class.

Buffer Properties

The properties that can be provided when constructing the SYCL `buffer` class are describe in Table 4.33.

Property	Description
`property::buffer::use_host_ptr`	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.
`property::buffer::use_mutex`	The `use_mutex` property is valid for the SYCL `buffer` and `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 `mutex_class` 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.
`property::buffer::context_bound`	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.
	End of table

Table 4.33: Properties supported by the SYCL `buffer` class.

The constructors and special member functions of the buffer property classes are listed in Tables 4.34 and 4.35 respectively.

Constructor	Description
property::buffer::use_host_ptr::use_host_ptr()	Constructs a SYCL use_host_ptr property instance.
property::buffer::use_mutex::use_mutex(mutex_class & mutexRef)	Constructs a SYCL use_mutex property instance with a reference to mutexRef parameter provided.
property::buffer::context_bound::context_bound(context boundContext)	Constructs a SYCL context_bound property instance with a copy of a SYCL context.
	End of table

Table 4.34: Constructors of the buffer property classes.

Member function	Description
mutex_class *property::buffer::use_mutex:: get_mutex_ptr()const	Returns the mutex_class which was specified when constructing this SYCL use_mutex property.
context property::buffer::context_bound::get_context ()const	Returns the context which was specified when constructing this SYCL context_bound property.
	End of table

Table 4.35: Member functions of the buffer property classes.

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 with just a size and using the default buffer allocator. The memory management for this type of buffer is entirely handled by the SYCL system. The destructor for this type of buffer never blocks, 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 with associated host memory and a default buffer allocator. 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 by the host, or mapped to 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 `unique_ptr_class` to a `std::shared_ptr`, a `unique_ptr_class` 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 will not block.

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.

Images

The class `image<int dimensions>` (Table 4.36) defines shared image data of one, two or three dimensions, that can be used by kernels in queues and has to be accessed using accessor classes with image accessor modes.

The constructors and member functions of the SYCL `image` class template are listed in Tables 4.36 and 4.37,

respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

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

The allocator template parameter of the SYCL `image` class can be any allocator type including a custom allocator, however it must allocate in units of `byte`.

If an image object is constructed from a `cl_mem` object, then the image is created and initialized from the OpenCL memory object. The SYCL system may copy the data to the host, but must copy it back (if modified) at the point of destruction of the image. The user must provide a `queue` and `event`. The memory object is assumed to only be available to the SYCL runtime after the event has signaled and is assumed to be currently resident on the context and device signified by the `queue`.

For any image that is constructed with the range $(r1, r2, r3)$ with a element type size in bytes of s, the image row pitch and image slice pitch should be calculated as follows:

$$r1 \cdot s \tag{4.1}$$

$$r1 \cdot r2 \cdot s \tag{4.2}$$

The SYCL `image` class template provides the common reference semantics (see Section 4.3.2).

Image Interface

Each constructor excluding the interoperability constructor takes as the last parameter an optional SYCL `property_list` to provide properties to the SYCL `image`.

The SYCL `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 `image` class `image_allocator` will be used 4.7.1.1.

```
1  namespace cl {
2  namespace sycl {
3  namespace property {
4  namespace image {
5  class use_host_ptr {
6    public:
7      use_host_ptr() = default;
8  };
9
10 class use_mutex {
11   public:
12     use_mutex(mutex_class &mutexRef);
13
14     mutex_class *get_mutex_ptr() const;
15 };
16
```

```
17  class context_bound {
18    public:
19      context_bound(context boundContext);
20
21      context get_context() const;
22  };
23  }  // namespace image
24  }  // namespace property
25
26  enum class image_channel_order : unsigned int {
27    a,
28    r,
29    rx,
30    rg,
31    rgx,
32    ra,
33    rgb,
34    rgbx,
35    rgba,
36    argb,
37    bgra,
38    intensity,
39    luminance,
40    abgr
41  }
42
43  enum class image_channel_type : unsigned int {
44    snorm_int8,
45    snorm_int16,
46    unorm_int8,
47    unorm_int16,
48    unorm_short_565,
49    unorm_short_555,
50    unorm_int_101010,
51    signed_int8,
52    signed_int16,
53    signed_int32,
54    unsigned_int8,
55    unsigned_int16,
56    unsigned_int32,
57    fp16,
58    fp32
59  }
60
61  using byte = unsigned char;
62
63  template <int dimensions = 1, typename AllocatorT = cl::sycl::image_allocator>
64  class image {
65   public:
66    image(image_channel_order order, image_channel_type type,
67          const range<dimensions> &range, const property_list &propList = {});
68
69    image(image_channel_order order, image_channel_type type,
70          const range<dimensions> &range, AllocatorT allocator,
71          const property_list &propList = {});
```

```
72
73    /* Available only when: dimensions > 1 */
74    image(image_channel_order order, image_channel_type type,
75          const range<dimensions> &range, const range<dimensions - 1> &pitch,
76          const property_list &propList = {});
77
78    /* Available only when: dimensions > 1 */
79    image(image_channel_order order, image_channel_type type,
80          const range<dimensions> &range, const range<dimensions - 1> &pitch,
81          AllocatorT allocator, const property_list &propList = {});
82
83    image(void *hostPointer, image_channel_order order,
84          image_channel_type type, const range<dimensions> &range,
85          const property_list &propList = {});
86
87    image(void *hostPointer, image_channel_order order,
88          image_channel_type type, const range<dimensions> &range,
89          AllocatorT allocator, const property_list &propList = {});
90
91    image(const void *hostPointer, image_channel_order order,
92          image_channel_type type, const range<dimensions> &range,
93          const property_list &propList = {});
94
95    image(const void *hostPointer, image_channel_order order,
96          image_channel_type type, const range<dimensions> &range,
97          AllocatorT allocator, const property_list &propList = {});
98
99    /* Available only when: dimensions > 1 */
100   image(void *hostPointer, image_channel_order order,
101         image_channel_type type, const range<dimensions> &range,
102         range<dimensions - 1> &pitch, const property_list &propList = {});
103
104   /* Available only when: dimensions > 1 */
105   image(void *hostPointer, image_channel_order order,
106         image_channel_type type, const range<dimensions> &range,
107         range<dimensions - 1> &pitch, AllocatorT allocator,
108         const property_list &propList = {});
109
110   image(shared_ptr_class<void> &hostPointer, image_channel_order order,
111         image_channel_type type, const range<dimensions> &range,
112         const property_list &propList = {});
113
114   image(shared_ptr_class<void> &hostPointer, image_channel_order order,
115         image_channel_type type, const range<dimensions> &range,
116         AllocatorT allocator, const property_list &propList = {});
117
118   /* Available only when: dimensions > 1 */
119   image(shared_ptr_class<void> &hostPointer, image_channel_order order,
120         image_channel_type type, const range<dimensions> &range,
121         const range<dimensions - 1> &pitch, const property_list &propList = {});
122
123   /* Available only when: dimensions > 1 */
124   image(shared_ptr_class<void> &hostPointer, image_channel_order order,
125         image_channel_type type, const range<dimensions> &range,
126         const range<dimensions - 1> &pitch, AllocatorT allocator,
```

```
127         const property_list &propList = {});
128
129   image(cl_mem clMemObject, const context &syclContext,
130         event availableEvent = {});
131
132   /* -- common interface members -- */
133
134   /* -- property interface members -- */
135
136   range<dimensions> get_range() const;
137
138   /* Available only when: dimensions > 1 */
139   range<dimensions - 1> get_pitch() const;
140
141   size_t get_size() const;
142
143   size_t get_count() const;
144
145   AllocatorT get_allocator() const;
146
147   template <typename dataT, access::mode accessMode>
148   accessor<dataT, dimensions, accessMode, access::target::image>
149   get_access(handler & commandGroupHandler);
150
151   template <typename dataT, access::mode accessMode>
152   accessor<dataT, dimensions, accessMode, access::target::host_image>
153   get_access();
154
155   template <typename Destination = std::nullptr_t>
156   void set_final_data(Destination finalData = std::nullptr);
157
158   void set_write_back(bool flag = true);
159
160 };
161 }  // namespace sycl
162 }  // namespace cl
```

Constructor	Description
`image(image_channel_order order, image_channel_type type, const range<dimensions> & range, const property_list &propList = {})`	Construct a SYCL `image` instance with uninitialized memory. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image(image_channel_order order, image_channel_type type, const range<dimensions> & range, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `image` instance with uninitialized memory. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
`image<dimensions>(image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> &pitch, const property_list &propList = {})`	Available only when: `dimensions > 1`. Construct a SYCL `image` instance with uninitialized memory. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image<dimensions>(image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> &pitch, AllocatorT allocator, const property_list &propList = {})`	Available only when: `dimensions > 1`. Construct a SYCL `image` instance with uninitialized memory. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
image(void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const property_list &propList = {})	Construct a SYCL image instance with the hostPointer parameter provided. The ownership of this memory is given to the constructed SYCL image for the duration of it's lifetime. The constructed SYCL image will use a default constructed AllocatorT when allocating memory on the host. The element size of the constructed SYCL image will be derived from the order and type parameters. The range of the constructed SYCL image is specified by the range parameter provided. The pitch of the constructed SYCL 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 image via an instance of property_list.
image(void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, AllocatorT allocator, const property_list &propList = {})	Construct a SYCL image instance with the hostPointer parameter provided. The ownership of this memory is given to the constructed SYCL image for the duration of it's lifetime. The constructed SYCL image 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 device accesses can be both read and write accesses. Since, the hostPointer is const, this image is only initialized with this memory and there is no write after its destruction, unless there is another final data address given after construction of the image. The element size of the constructed SYCL image will be derived from the order and type parameters. The range of the constructed SYCL image is specified by the range parameter provided. The pitch of the constructed SYCL 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 image via an instance of property_list.

Continued on next page

Table 4.36: Constructors of the image class template.

Constructor	Description
`image(const void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The host address is `const T`, so the host accesses can be read-only. However, the device accesses can be both read and write accesses. Since, the `hostPointer` is const, this image is only initialized with this memory and there is no write after its destruction, unless there is another final data address given after construction of the image. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image(const void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
`image(void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> &pitch, const property_list &propList = {})`	Available only when: `dimensions > 1`. Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image(void *hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> &pitch, AllocatorT allocator, const property_list &propList = {})`	Available only when: `dimensions > 1`. Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
`image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
`image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> & pitch, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use a default constructed `AllocatorT` when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.
`image(shared_ptr_class<void>& hostPointer, image_channel_order order, image_channel_type type, const range<dimensions> & range, const range<dimensions-1> & pitch, AllocatorT allocator, const property_list &propList = {})`	Construct a SYCL `image` instance with the `hostPointer` parameter provided. The ownership of this memory is given to the constructed SYCL `image` for the duration of it's lifetime. The constructed SYCL `image` will use the `allocator` parameter provided when allocating memory on the host. The element size of the constructed SYCL `image` will be derived from the `order` and `type` parameters. The range of the constructed SYCL `image` is specified by the `range` parameter provided. The pitch of the constructed SYCL `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 `image` via an instance of `property_list`.

Continued on next page

Table 4.36: Constructors of the `image` class template.

Constructor	Description
image(cl_mem clMemObject, const context &syclContext, event availableEvent = {})	Constructs a SYCL image instance from an OpenCL cl_mem in accordance with the requirements described in 4.3.1. The instance of the SYCL image class template being constructed must wait for the SYCL event parameter, if one is provided, 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.
	End of table

Table 4.36: Constructors of the image class template.

Member function	Description
range<dimensions> get_range()const	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.
range<dimensions-1> get_pitch()const	Available only when: dimensions > 1. Return a range object representing the pitch of the image in bytes.
size_t get_count()const	Returns the total number of elements in the image. Equal to get_range()[0] * ... * get_range()[dimensions-1].
size_t get_size()const	Returns the size of the image storage in bytes. The number of bytes may be greater than get_count()*element size due to padding of elements, rows and slices of the image for efficient access.
AllocatorT get_allocator()const	Returns the allocator provided to the image.
template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access:: target::image> get_access(handler & commandGroupHandler)	Returns a valid accessor to the image with the specified access mode and target. The only valid types for dataT are cl_int4, cl_uint4, cl_float4 and cl_half4.
template<typename dataT, access::mode accessMode> accessor<dataT, dimensions, accessMode, access:: target::host_image> get_access()	Returns a valid accessor to the image with the specified access mode and target. The only valid types for dataT are cl_int4, cl_uint4, cl_float4 and cl_half4.
	Continued on next page

Table 4.37: Member functions of the image class template.

Member function	Description
`template <typename Destination = std::nullptr_t> void set_final_data(Destination finalData = std::nullptr)`	The `finalData` points 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 `weak_ptr_class<T>`. 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 `Destination` is `std::nullptr_t`, then the copy back will not happen.
`void set_write_back(bool flag = true)`	This method allows dynamically forcing or canceling the write-back of the data of an image on destruction according to the value of `flag`. Forcing the write-back is similar to what happens during a normal write-back as described in § 4.7.3.3 and 4.7.4. If there is nowhere to write-back, using this function does not have any effect.
	End of table

Table 4.37: Member functions of the `image` class template.

Image Properties

The properties that can be provided when constructing the SYCL `image` class are describe in Table 4.38.

Property	Description
`property::image::use_host_ptr`	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.
	Continued on next page

Table 4.38: Properties supported by the SYCL `image` class.

Property	Description
property::image::use_mutex	The use_mutex property is valid for the SYCL image and image classes. The property adds the requirement that the memory which is owned by the SYCL image can be shared with the application via a mutex_class provided to the property. The mutex_class m is locked by the runtime whenever the data is in use and unlocked otherwise. Data is synchronized with hostData, when the mutex_class is unlocked by the runtime.
property::image::context_bound	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.
	End of table

Table 4.38: Properties supported by the SYCL image class.

The constructors and member functions of the image property classes are listed in Tables 4.39 and 4.40

Constructor	Description
property::image::use_host_ptr::use_host_ptr()	Constructs a SYCL use_host_ptr property instance.
property::image::use_mutex::use_mutex(mutex_class & mutexRef)	Constructs a SYCL use_mutex property instance with a reference to mutexRef parameter provided.
property::image::context_bound::context_bound(context boundContext)	Constructs a SYCL context_bound property instance with a copy of a SYCL context.
	End of table

Table 4.39: Constructors of the image property classes.

Member function	Description
mutex_class *property::image::use_mutex:: get_mutex_ptr()const	Returns the mutex_class which was specified when constructing this SYCL use_mutex property.
context property::image::context_bound::get_context ()const	Returns the context which was specified when constructing this SYCL context_bound property.
	End of table

Table 4.40: Member functions of the image property classes.

Image Synchronization Rules

The rules are similar to those described in § 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.

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.

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 use memory managed by a SYCL buffer within the buffer scope by using a `host accessor` as defined in 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.

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.

If the pointer contained in the `unique_ptr` is null, the pointer is initialized internally in the runtime but no data is

copied in. This will be the generic case of a buffer constructor that takes no host pointer.

For example, the following could be used:

```
1  {
2    cl::sycl::unique_ptr_class<int> ptr { data };
3    buffer<int, 1> b { std::move(ptr) };
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 `weak_ptr_class` to enable copying data back, to another host memory address that is going to be valid after buffer construction.

```
1  {
2    cl::sycl::unique_ptr_class<int> ptr { data };
3    buffer<int, 1> b { std::move(ptr) };
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(weak_ptr_class<int> { .... })
8  }
```

Shared SYCL ownership of the host memory

When a `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 memory pointed to by the shared object is initialized to some data, then that data is used to initialize the buffer. If the shared pointer is null, the pointer is initialized by the runtime internally (and, therefore, the user can use it afterwards in the host).

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 from OpenCL, as the outcome is not needed on the application's side.

This behavior can be overiden using the `set_final_data()` method 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()` method has put the data (or not wait nor copy if set final data is `std::nullptr`).

```
1  {
2    cl::sycl::shared_ptr_class<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  }
```

```
{
  cl::sycl::shared_ptr_class<int> ptr { data };
  {
    buffer<int, 1> b { ptr, range<2>{ 10, 10 } };
    // update the data
    [...]
    ptr.reset();
  } // Data is not copied back, there is no user side shared_ptr.
}
```

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 want to use manual synchonization without host accessors, a pointer to a `mutex_class` can be passed to the buffer constructor.

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

```
{
  cl::sycl::mutex_class m;
  auto shD = std::make_shared<int> { 42 }
  {
    buffer<int, 1> b { shD, m };

    std::lock_guard<mutex_class> lck { m };
    // User accesses the data
    do_something(shD);
    /* m is unlock when lck goes out of scope, by normal end of this
       block but also if an exception is thrown for example */
  }
}
```

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()`.

Accessors

An accessor is defined by the SYCL `accessor` class template. An `accessor` provides access to the data managed by a buffer or image, or to shared local memory allocated by the runtime. An accessor allows users to define **requirements** to memory objects (see Section 3.5.1).

The SYCL `accessor` class template takes five template parameters:

- A typename specifying the data type that the `accessor` is providing access to.
- An integer specifying the dimensionality of the accessor.
- A value of `access::mode` specifying the mode of access the `accessor` is providing.

- A value of access::target specifying the target of access the accessor is providing.
- A value of access::placeholder specifying whether the accessor is a placeholder accessor.

The parameters described above determine the data an accessor provides access to and the way in which that access is provided. This separation allows a SYCL runtime implementation to choose an efficient way to provide access to the data within an execution schedule.

Because of this the interface of the accessor will be different depending on the possible combinations of those parameters. There are three main categories of accessor; buffer accesors (see Section 4.7.6.5), local accessors (see Section 4.7.6.7) and image accessors (see Section 4.7.6.9).

Access targets

The access target of an accessor specifies what the accessor is providing access to.

The access::target enumeration, shown in Table 4.41, describes the potential targets of an accessor.

```
1  namespace cl {
2  namespace sycl {
3  namespace access {
4  enum class target {
5    global_buffer = 2014,
6    constant_buffer,
7    local,
8    image,
9    host_buffer,
10   host_image,
11   image_array
12 };
13
14 }  // namespace access
15 }  // namespace sycl
16 }  // namespace cl
```

access::target	Description
access::target::global_buffer	Access buffer via global memory.
access::target::constant_buffer	Access buffer via constant memory.
access::target::local	Access work-group local memory.
access::target::image	Access an image.
access::target::host_buffer	Access a buffer immediately in host code.
access::target::host_image	Access an image immediately in host code.
access::target::image_array	Access an array of images on a device.
	End of table

Table 4.41: Enumeration of access modes available to accessors.

Access modes

The access mode of an accessor specifies the kind of access that is being provided. This information is used by the runtime to ensure that any data dependencies are resolved by enqueuing any data transfers before or after the execution of a kernel. If a command group contains only *discard write mode* accesses to a buffer, then the previous contents of the buffer (or sub-range of the buffer, if provided) are not preserved. If a user wants to modify only certain parts of a buffer, preserving other parts of the buffer, then the user should specify the exact sub-range of modification of the buffer. Atomic access is only valid to `local`, `global_buffer` and `host_buffer` targets (see next section).

The access::mode enumeration, shown in Table 4.42, describes the potential modes of an accessor.

```
namespace cl {
namespace sycl {
namespace access {
enum class mode {
  read = 1024,
  write,
  read_write,
  discard_write,
  discard_read_write,
  atomic
};
}  // namespace access
}  // namespace sycl
}  // namespace cl
```

access::mode	Description
access::mode::read	Read-only access.
access::mode::write	Write-only access. Previous contents not discarded.
access::mode::read_write	Read and write access.
access::mode::discard_write	Write-only access. Previous contents discarded.
access::mode::discard_read_write	Read and write access. Previous contents discarded.
access::mode::atomic	Read and write atomic access.
	End of table

Table 4.42: Enumeration of access modes available to accessors.

Device and host accessors

A SYCL accessor can be a device accessor in which case it provides access to data within a SYCL kernel function, or a host accessor in which case it provides immediate access on the host.

If an accessor has the access target access::target::global_buffer, access::target::constant_buffer, access::target::local, access::target::image or access::target::image_array then it is considered a device accessor, and therefore can only be used within a SYCL kernel function and must be associated with a

command group. Creating a device accessor is a non-blocking operation which defines a requirement on the device and adds the requirement to the queue.

If an accessor has the access target access::target::host_buffer or access::target::host_image then it is considered a host accessor and can only be used on the host. Creating a host accessor is a blocking operation which defines a requirement on the host and blocks the caller until the requirement is satisfied.

A host accessor provides immediate access and continues to provide access until it is destroyed.

Placeholder accessor

A placeholder accessor can be constructed outside of a command group and then later bound to a command group. A SYCL accessor is considered a placeholder accessor if it has the access placeholder access::placeholder::true_t.

Accessors can optionally be defined as *placeholder* accessors. A *placeholder* accessor defines an accessor instance that is not bound to a specific command group. The accessor defines only the type of the accessor (target memory, access mode, base type, ...). When associated with a a command group using the appropriate handler interface, it defines a **requirement** for the command group. The same placeholder accessor can be required by multiple command groups.

The access::placeholder enumeration, shown in Table 4.43, describes the potential placeholder values of an accessor.

placeholder::mode	Description
access::placeholder::false_t	Non-placeholder accessor.
access::placeholder::true_t	Placeholder accessor.
	End of table

Table 4.43: Enumeration of placeholder values available to accessors.

Buffer accessor

A buffer accessor provides access to a SYCL buffer instance. A SYCL accessor is considered a buffer accessor if it has the access target access::target::global_buffer, access::target::constant_buffer or access::target::host_buffer.

A buffer accessor can provide access to memory managed by a SYCL buffer class via either global memory or constant memory, corresponding to the access targets access::target::global_buffer and access::target::constant_buffer respectively. A buffer accessor accessing a SYCL buffer via constant memory is restricted by the available constant memory available on the SYCL device being executed on.

Alternatively a buffer accessor can provide access to memory managed by a SYCL buffer immediately on the host, using the access target access::target::host_buffer. If the SYCL buffer this SYCL accessor is accessing was constructed with the property property::buffer::use_host_ptr the address of the memory accessed on the host must be the address the SYCL buffer was constructed with, otherwise the SYCL runtime is free to allocate temporary memory to provide access on the host.

The data type of a buffer accessor must match that of the SYCL buffer which it is accessing.

The dimensionality of a buffer accessor must match that of the SYCL buffer which it is accessing, with the exception of 0 in which case the dimensionality of the SYCL buffer must be 1.

There are three ways a SYCL accessor can provide access to the elements of a SYCL buffer. Firstly by passing a SYCL id instance of the same dimensionality as the SYCL accessor subscript operator. Secondly by passing a single size_t value to multiple consecutive subscript operators (one for each dimension of the SYCL accessor, for example acc[id1][id2][id3]). Finally, in the case of the SYCL accessor being 0 dimensions, by triggering the implicit conversion operator. Whenever a multi-dimensional index is passed to a SYCL accessor the linear index is calculated based on the index {id1, id2, id3} provided and the range of the SYCL accessor {r1, r2, r3} according to row-major ordering as follows:

$$id3 + (id2 \cdot r3) + (id1 \cdot r3 \cdot r2) \tag{4.3}$$

A buffer accessor can optionally provide access to a sub range of a SYCL buffer by providing a range and offset on construction. In this case the SYCL runtime will only guarantee the latest copy of the data is available in that given range and any modifications outside that range are considered undefined behavior. This allows the SYCL runtime to perform optimizations such as reducing copies between devices. The indexing performed when a SYCL accessor provides access to the elements of a SYCL buffer is unaffected, i.e, the accessor will continue to index from {0,0,0}. This allows the offset to be provided either manually or via the parallel_for as in 4.7.6.5.

```
1     myQueue.submit([&](handler &cgh) {
2       auto singleRange = range<3>(8, 16, 16);
3       auto offset = id<3>(8, 0, 0);
4       // We define the subset of the accessor we require for the kernel
5       accessor<int, 1, access::mode::read_write, access::target::global_buffer>
6           ptr(syclBuffer, cgh, singleRange, offset);
7       // We offset the kernel by the same value to match indexes
8       cgh.parallel_for<kernel>(singleRange, offset, [=](item<3> itemID) {
9         ptr[itemID.get_linear_id()] = 2;
10      });
11    });
```

A buffer accessor with access target access::target::global_buffer can optionally provide atomic access to a SYCL buffer, using the access mode access::mode::atomic, in which case all operators which return an element of the SYCL buffer return an instance of the SYCL atomic class.

The full list of capabilities that buffer accessors can support is described in 4.44.

Buffer accessor interface

A synopsis of the SYCL accessor class template buffer specialization is provided below. The constructors and member functions of the SYCL accessor class template buffer specialization are listed in Tables 4.45 and 4.46 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
1  namespace cl {
2  namespace sycl {
3  template <typename dataT, int dimensions, access::mode accessmode,
4            access::target accessTarget = access::target::global_buffer,
```

Access target	Accessor type	Access modes	Data types	Dimensionalities	Placeholder modes

`global_buffer`	device	`read`
		`write`
		`read_write`
		`discard_write`
		`discard_read_write`
		`atomic`
			The data type of the SYCL buffer being accessed.	Between `0` and `3` (inclusive).	`false_t`
					`true_t`

`constant_buffer`	device	`read`	The data type of the SYCL buffer being accessed.	Between `0` and `3` (inclusive).	`false_t` `true_t`

`host_buffer`	host	`read`
		`write`
		`read_write`
		`discard_write`
		`discard_read_write`
			The data type of the SYCL buffer being accessed.	Between `0` and `3` (inclusive).	`false_t`

Table 4.44: Description of all the buffer accessor capabilities.

```
5           access::placeholder isPlaceholder = access::placeholder::false_t>
6  class accessor {
7   public:
8    using value_type = dataT;
9    using reference = dataT &;
10   using const_reference = const dataT &;
11
12   /* Available only when: ((isPlaceholder == access::placeholder::false_t &&
13   accessTarget == access::target::host_buffer) || (isPlaceholder ==
14   access::placeholder::true_t && (accessTarget == access::target::global_buffer
15   || accessTarget == access::target::constant_buffer))) && dimensions == 0 */
16   accessor(buffer<dataT, 1> &bufferRef);
17
18   /* Available only when: (isPlaceholder == access::placeholder::false_t &&
19   (accessTarget == access::target::global_buffer || accessTarget ==
20   access::target::constant_buffer)) && dimensions == 0 */
21   accessor(buffer<dataT, 1> &bufferRef, handler &commandGroupHandlerRef);
22
23   /* Available only when: ((isPlaceholder == access::placeholder::false_t &&
24   accessTarget == access::target::host_buffer) || (isPlaceholder ==
25   access::placeholder::true_t && (accessTarget == access::target::global_buffer
26   || accessTarget == access::target::constant_buffer))) && dimensions > 0 */
27   accessor(buffer<dataT, dimensions> &bufferRef);
28
29   /* Available only when: (isPlaceholder == access::placeholder::false_t &&
30   (accessTarget == access::target::global_buffer || accessTarget ==
31   access::target::constant_buffer)) && dimensions > 0 */
32   accessor(buffer<dataT, dimensions> &bufferRef,
33     handler &commandGroupHandlerRef);
34
35   /* Available only when: (isPlaceholder == access::placeholder::false_t &&
36   accessTarget == access::target::host_buffer) || (isPlaceholder ==
37   access::placeholder::true_t && (accessTarget == access::target::global_buffer
38   || accessTarget == access::target::constant_buffer)) && dimensions > 0 */
39   accessor(buffer<dataT, dimensions> &bufferRef, range<dimensions> accessRange,
40     id<dimensions> accessOffset = {});
41
```

```
42    /* Available only when: (isPlaceholder == access::placeholder::false_t &&
43    (accessTarget == access::target::global_buffer || accessTarget ==
44    access::target::constant_buffer)) && dimensions > 0 */
45    accessor(buffer<dataT, dimensions> &bufferRef,
46      handler &commandGroupHandlerRef, range<dimensions> accessRange,
47      id<dimensions> accessOffset = {});
48
49    /* -- common interface members -- */
50
51    constexpr bool is_placeholder() const;
52
53    size_t get_size() const;
54
55    size_t get_count() const;
56
57    /* Available only when: dimensions > 0 */
58    range<dimensions> get_range() const;
59
60    /* Available only when: dimensions > 0 */
61    id<dimensions> get_offset() const;
62
63    /* Available only when: (accessMode == access::mode::write || accessMode ==
64    access::mode::read_write || accessMode == access::mode::discard_write ||
65    accessMode == access::mode::discard_read_write) && dimensions == 0) */
66    operator dataT &() const;
67
68    /* Available only when: (accessMode == access::mode::write || accessMode ==
69    access::mode::read_write || accessMode == access::mode::discard_write ||
70    accessMode == access::mode::discard_read_write) && dimensions > 0) */
71    dataT &operator[](id<dimensions> index) const;
72
73    /* Available only when: (accessMode == access::mode::write || accessMode ==
74    access::mode::read_write || accessMode == access::mode::discard_write ||
75    accessMode == access::mode::discard_read_write) && dimensions == 1) */
76    dataT &operator[](size_t index) const;
77
78    /* Available only when: accessMode == access::mode::read && dimensions == 0 */
79    operator dataT() const;
80
81    /* Available only when: accessMode == access::mode::read && dimensions > 0 */
82    dataT operator[](id<dimensions> index) const;
83
84    /* Available only when: accessMode == access::mode::read && dimensions == 1 */
85    dataT operator[](size_t index) const;
86
87    /* Available only when: accessMode == access::mode::atomic && dimensions ==
88    0 */
89    operator atomic<dataT, access::address_space::global_space> () const;
90
91    /* Available only when: accessMode == access::mode::atomic && dimensions >
92    0 */
93    atomic<dataT, access::address_space::global_space> operator[](
94      id<dimensions> index) const;
95
96    /* Available only when: accessMode == access::mode::atomic && dimensions ==
```

```
97    1 */
98    atomic<dataT, access::address_space::global_space> operator[](
99      size_t index) const;
100
101   /* Available only when: dimensions > 1 */
102   __unspecified__ &operator[](size_t index) const;
103
104   /* Available only when: accessTarget == access::target::host_buffer */
105   dataT *get_pointer() const;
106
107   /* Available only when: accessTarget == access::target::global_buffer */
108   global_ptr<dataT> get_pointer() const;
109
110   /* Available only when: accessTarget == access::target::constant_buffer */
111   constant_ptr<dataT> get_pointer() const;
112 };
113 }  // namespace sycl
114 }  // namespace cl
```

Listing 4.1: Accessor class for buffers.

Constructor	Description
accessor(buffer<dataT, 1> &bufferRef)	Available only when: ((isPlaceholder == access::placeholder::false_t && accessTarget == access::target::host_buffer)|| (isPlaceholder == access::placeholder::true_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer)))&& dimensions == 0. If isPlaceholder == access::placeholder::false_t, constructs a SYCL accessor instance for accessing a single element of a SYCL buffer immediately on the host. If isPlaceholder == access::placeholder::true_t, constructs a SYCL placeholder accessor.
accessor(buffer<dataT, 1> &bufferRef, handler &commandGroupHandlerRef)	Available only when: (isPlaceholder == access::placeholder::false_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer))&& dimensions == 0. Constructs a SYCL accessor instance for accessing a single element of a SYCL buffer within a SYCL kernel function on the SYCL queue associated with commandGroupHandlerRef.

Continued on next page

Table 4.45: Constructors of the accessor class template buffer specialization.

Constructor	Description
accessor(buffer<dataT, dimensions> &bufferRef)	Available only when: ((isPlaceholder == access::placeholder::false_t && accessTarget == access::target::host_buffer)|| (isPlaceholder == access::placeholder::true_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer)))&& dimensions > 0. If isPlaceholder == access::placeholder::false_t, constructs a SYCL accessor instance for accessing a SYCL buffer immediately on the host. If isPlaceholder == access::placeholder::true_t, constructs a SYCL placeholder accessor.
accessor(buffer<dataT, dimensions> &bufferRef, handler &commandGroupHandlerRef)	Available only when: (isPlaceholder == access::placeholder::false_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer))&& dimensions > 0. Constructs a SYCL accessor instance for accessing a SYCL buffer within a SYCL kernel function on the SYCL queue associated with commandGroupHandlerRef.
accessor(buffer<dataT, dimensions> &bufferRef, range<dimensions> accessRange, id<dimensions> accessOffset = {})	Available only when: (isPlaceholder == access::placeholder::false_t && accessTarget == access::target::host_buffer)|| (isPlaceholder == access::placeholder::true_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer))&& dimensions > 0. If isPlaceholder == access::placeholder::false_t, constructs a SYCL accessor instance for accessing a range of a SYCL buffer immediately on the host. If isPlaceholder == access::placeholder::true_t, constructs a SYCL placeholder accessor.

Continued on next page

Table 4.45: Constructors of the accessor class template buffer specialization.

Constructor	Description
accessor(buffer<dataT, dimensions> &bufferRef, handler &commandGroupHandlerRef, range<dimensions> accessRange, id<dimensions> accessOffset = {})	Available only when: (isPlaceholder == access::placeholder::false_t && (accessTarget == access::target::global_buffer || accessTarget == access::target::constant_buffer))&& dimensions > 0. Constructs a SYCL accessor instance for accessing a range of SYCL buffer within a SYCL kernel function on the SYCL queue associated with commandGroupHandlerRef, specified by accessRange and accessOffset.
	End of table

Table 4.45: Constructors of the accessor class template buffer specialization.

Member function	Description
constexpr bool is_placeholder()const	Returns true if isPlaceholder == access::placeholder::true_t otherwise returns false.
size_t get_size()const	Returns the size in bytes of the SYCL buffer this SYCL accessor is accessing.
size_t get_count()const	Returns the number of elements of the SYCL buffer this SYCL accessor is accessing.
range<dimensions> get_range()const	Available only when: dimensions > 0. Returns the range of this SYCL accessor.
id<dimensions> get_offset()const	Available only when: dimensions > 0. Returns the offset of this SYCL accessor.
operator dataT &()const	Available only when: (accessMode == access::mode::write || accessMode == access::mode::read_write || accessMode == access::mode::discard_write || accessMode == access::mode::discard_read_write)&& dimensions == 0). Returns a reference to the element stored within the SYCL buffer this SYCL accessor is accessing.
	Continued on next page

Table 4.46: Member functions of the accessor class template buffer specialization.

Member function	Description
`dataT &operator[](id<dimensions> index)const`	Available only when: `(accessMode == access::mode::write || accessMode == access::mode::read_write || accessMode == access::mode::discard_write || accessMode == access::mode::discard_read_write)&& dimensions > 0`). Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
`dataT &operator[](size_t index)const`	Available only when: `(accessMode == access::mode::write || accessMode == access::mode::read_write || accessMode == access::mode::discard_write || accessMode == access::mode::discard_read_write)&& dimensions == 1`). Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
`operator dataT()const`	Available only when: `accessMode == access::mode::read && dimensions == 0`). Returns the value of the element stored within the SYCL `buffer` this SYCL `accessor` is accessing.
`dataT operator[](id<dimensions> index)const`	Available only when: `accessMode == access::mode::read && dimensions > 0`) . Returns the value of the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
`dataT operator[](size_t index)const`	Available only when: `accessMode == access::mode::read && dimensions == 1`). Returns the value of the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
Continued on next page	

Table 4.46: Member functions of the `accessor` class template buffer specialization.

Member function	Description
operator atomic<dataT, access::address_space::global_space> ()const	Available only when: accessMode == access::mode::atomic && dimensions == 0). Returns an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing.
atomic<dataT, access::address_space::global_space> operator[](id<dimensions> index)const	Available only when: accessMode == access::mode::atomic && dimensions > 0). Returns an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing at the index specified by index.
atomic<dataT, access::address_space::global_space> operator[](size_t index)const	Available only when: accessMode == access::mode::atomic && dimensions == 1). Returns an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing at the index specified by index.
__unspecified__ &operator[](size_t index)const	Available only when: dimensions > 1. Returns an instance of an undefined intermediate type representing a a SYCL accessor of the same type as this SYCL accessor, with the dimensionality dimensions-1 and containing an implicit SYCL 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 the SYCL accessor class that are appropriate for the type it represents (including this subscript operator).
dataT *get_pointer()const	Available only when: accessTarget == access::target::host_buffer. Returns a pointer to the memory this SYCL accessor memory is accessing.
global_ptr<dataT> get_pointer()const	Available only when: accessTarget == access::target::global_buffer. Returns a pointer to the memory this SYCL accessor memory is accessing.
constant_ptr<dataT> get_pointer()const	Available only when: accessTarget == access::target::global_buffer. Returns a pointer to the memory this SYCL accessor memory is accessing.
	End of table

Table 4.46: Member functions of the accessor class template buffer specialization.

Local accessor

A local accessor provides access to SYCL runtime allocated shared memory via local memory. A SYCL `accessor` is considered a local accessor if it has the access target `access::target::local`. The memory allocated by a local accessor is non-initialised so it is the user's responsibility to construct and destroy objects explicitly if required. The local memory that is allocated is shared between all work-items of a work-group.

A local accessor does not provide access on the host and the memory can not be copied back to the host.

The data type of a local accessor can be any valid SYCL kernel argument (see Section 4.8.10.

The size of memory allocated by the SYCL runtime is specified by a SYCL `range` provided on construction. The dimensionality of the SYCL `range` provided must match the SYCL `accessor`, with the exception of `0` in which case the dimensionality of the SYCL `range` must be `0`.

There are three ways that a SYCL `accessor` can provide access to the elements of the allocated memory. Firstly by passing a SYCL `id` instance of the same dimensionality as the SYCL `accessor` subscript operator. Secondly by passing a single `size_t` value to multiple consecutive subscript operators (one for each dimension of the SYCL `accessor`, for example `acc[z][y][x]`). Finally, in the case of the SYCL `accessor` having `0` dimensions, by triggering the implicit conversion operator. Whenever a multi-dimensional index is passed to a SYCL `accessor`, the linear index is calculated based on the index `{id1, id2, id3}` provided and the range of the SYCL `accessor` `{r1, r2, r3}` according to row-major ordering as follows:

$$id3 + (id2 \cdot r3) + (id1 \cdot r3 \cdot r2) \tag{4.4}$$

A local accessor can optionally provide atomic access to allocated memory, using the access mode `access::mode::atomic`, in which case all operators which return an element of the allocated memory return an instance of the SYCL `atomic` class.

Local accessors are not valid in the `single_task` or basic `parallel_for` SYCL kernel function invocations, due the fact that local work-groups are implicitly created, and the implementation is free to choose any size.

The full list of capabilities that local accessors can support is described in 4.47.

Access target	Accessor type	Access modes	Data types	Dimensionalities	Placeholder modes
`local`	device	`read_write` `atomic`	All available data types supported in a SYCL kernel function.	Between `0` and `3` (inclusive).	`false_t`

Table 4.47: Description of all the local accessor capabilities.

Local accessor interface

A synopsis of the SYCL `accessor` class template local specialization is provided below. The constructors and member functions of the SYCL `accessor` class template local specialization are listed in Tables 4.48 and 4.49 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
1 namespace cl {
```

```
2  namespace sycl {
3  template <typename dataT, int dimensions, access::mode accessmode,
4            access::target accessTarget = access::target::global_buffer,
5            access::placeholder isPlaceholder = access::placeholder::false_t>
6  class accessor {
7   public:
8    using value_type = dataT;
9    using reference = dataT &;
10   using const_reference = const dataT &;
11
12   /* Available only when: dimensions == 0 */
13   accessor(handler &commandGroupHandlerRef);
14
15   /* Available only when: dimensions > 0 */
16   accessor(range<dimensions> allocationSize, handler &commandGroupHandlerRef);
17
18   /* -- common interface members -- */
19
20   size_t get_size() const;
21
22   size_t get_count() const;
23
24   /* Available only when: accessMode == access::mode::read_write && dimensions == 0) */
25   operator dataT &() const;
26
27   /* Available only when: accessMode == access::mode::read_write && dimensions > 0) */
28   dataT &operator[](id<dimensions> index) const;
29
30   /* Available only when: accessMode == access::mode::read_write && dimensions == 1) */
31   dataT &operator[](size_t index) const;
32
33   /* Available only when: accessMode == access::mode::atomic && dimensions ==
34   0 */
35   operator atomic<dataT,access::address_space::local_space> () const;
36
37   /* Available only when: accessMode == access::mode::atomic && dimensions >
38   0 */
39   atomic<dataT, access::address_space::local_space> operator[](
40     id<dimensions> index) const;
41
42   /* Available only when: accessMode == access::mode::atomic && dimensions ==
43   1 */
44   atomic<dataT, access::address_space::local_space> operator[](
45     size_t index) const;
46
47   /* Available only when: dimensions > 1 */
48   __unspecified__ &operator[](size_t index) const;
49
50   /* Available only when: accessTarget == access::target::local */
51   local_ptr<dataT> get_pointer() const;
52 };
53 }  // namespace sycl
54 }  // namespace cl
```

Listing 4.2: Accessor class for locals.

Constructor	Description
`accessor(handler &commandGroupHandlerRef)`	Available only when: `dimensions == 0`. Constructs a SYCL `accessor` instance for accessing runtime allocated shared local memory of a single element within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`.
`accessor(range<dimensions> allocationSize, handler &commandGroupHandlerRef)`	Available only when: `dimensions > 0`. Constructs a SYCL `accessor` instance for accessing runtime allocated shared local memory of size specified by `allocationSize` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`.
	End of table

Table 4.48: Constructors of the `accessor` class template local specialization.

Member function	Description
`size_t get_size()const`	Returns the size in bytes of the SYCL `buffer` this SYCL `accessor` is accessing.
`size_t get_count()const`	Returns the number of elements of the SYCL `buffer` this SYCL `accessor` is accessing.
`operator dataT &()const`	Available only when: `accessMode == access::mode::read_write && dimensions == 0`). Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing.
`dataT &operator[](id<dimensions> index)const`	Available only when: `accessMode == access::mode::read_write && dimensions > 0`). Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
`dataT &operator[](size_t index)const`	Available only when: `accessMode == access::mode::read_write && dimensions == 1`). Returns a reference to the element stored within the SYCL `buffer` this SYCL `accessor` is accessing at the index specified by `index`.
	Continued on next page

Table 4.49: Member functions of the `accessor` class template local specialization.

Member function	Description
operator atomic<dataT, access::address_space::local_space> &()const	Available only when: accessMode == access::mode::atomic && dimensions == 0). Returns a reference to an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing.
atomic<dataT, access::address_space::local_space> & operator[](id<dimensions> index)const	Available only when: accessMode == access::mode::atomic && dimensions > 0). Returns a reference to an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing at the index specified by index.
atomic<dataT, access::address_space::local_space> & operator[](size_t index)const	Available only when: accessMode == access::mode::atomic && dimensions == 1). Returns a reference to an instance of SYCL atomic of type dataT providing atomic access to the element stored within the SYCL buffer this SYCL accessor is accessing at the index specified by index.
__unspecified__ &operator[](size_t index)const	Available only when: dimensions > 1. Returns an instance of an undefined intermediate type representing a a SYCL accessor of the same type as this SYCL accessor, with the dimensionality dimensions-1 and containing an implicit SYCL 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 the SYCL accessor class that are appropriate for the type it represents (including this subscript operator).
local_ptr<dataT> get_pointer()const	Available only when: accessTarget == access::target::local. Returns a pointer to the memory this SYCL accessor memory is accessing.
	End of table

Table 4.49: Member functions of the accessor class template local specialization.

Image accessor

An image accessor provides access to a SYCL image instance. A SYCL accessor is considered an image accessor if it has the access target access::target::image, access::target::image_array or access::target::host_image.

An image accessor can provide access to memory managed by a SYCL `image` class, using the access target `access::target::image` or `access::target::image_array`.

Alternatively an image accessor can provide access to memory managed by a SYCL `image` immediately on the host, using the access target `access::target::host_image`. If the SYCL `image` this SYCL `accessor` is accessing was constructed with the property `property::image::use_host_ptr` the address of the memory accessed on the host must be the address the SYCL `image` was constructed with, otherwise the SYCL runtime is free to allocate temporary memory to provide access on the host.

The data type of an image accessor must be either `cl_int4`, `cl_uint4`, `cl_float4` or `cl_half4`.

The dimensionality of an image accessor must match that of the SYCL `image` which it is providing access to, with the exception of when the access target is `access::target::image_array`, in which case the dimensionality of the SYCL `accessor` must be 1 less.

An image accessor with the access target `access::target::image` or `access::target::host_image` can provide access to the elements of a SYCL `image` by passing a SYCL `cl_int4` or `cl_float4` instance to the `read` or `write` member functions. The `read` member function optionally takes a SYCL `sampler` instance to perform a sampled read of the image. For example `acc.read(coords, sampler)`.

An image accessor with the access target `access::target::image_array` can provide access to a slice of an image array by passing a `size_t` value to the subscript operator. This returns an instance of `__image_array_slice__`, an unspecified type providing the interface of `accessor<dataT, dimensions, mode, access::target::image, access::placeholder::false_t>` which will provide access to a slice of the image array specified by `index`. The `__image_array_slice__` returned can then provide access via the `read` or `write` member functions as described above. For example `acc[arrayIndex].read(coords, sampler)`.

The full list of capabilities that image accessors can support is described in 4.50.

Access target	Accessor type	Access modes	Data types	Dimensionalities	Placeholder modes

`image`	device	`read`
		`write`
		`discard_write`
			`cl_int4`
			`cl_uint4`
			`cl_float4`
			`cl_half4`
				Between 1 and 3 (inclusive).	`false_t`

`image_array`	device	`read`
		`write`
		`discard_write`
			`cl_int4`
			`cl_uint4`
			`cl_float4`
			`cl_half4`
				Between 1 and 2 (inclusive).	`false_t`

`host_image`	host	`read`
		`write`
		`discard_write`
			`cl_int4`
			`cl_uint4`
			`cl_float4`
			`cl_half4`
				Between 1 and 3 (inclusive).	`false_t`

Table 4.50: Description of all the image accessor capabilities.

Image accessor interface

A synopsis of the SYCL `accessor` class template image specialization is provided below. The constructors and member functions of the SYCL `accessor` class template image specialization are listed in Tables 4.51 and 4.52 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
namespace cl {
namespace sycl {
template <typename dataT, int dimensions, access::mode accessmode,
          access::target accessTarget = access::target::global_buffer,
          access::placeholder isPlaceholder = access::placeholder::false_t>
class accessor {
 public:
  using value_type = dataT;
  using reference = dataT &;
  using const_reference = const dataT &;

  /* Available only when: accessTarget == access::target::host_image */
  template <typename AllocatorT>
  accessor(image<dimensions, AllocatorT> &imageRef);

  /* Available only when: accessTarget == access::target::image */
  template <typename AllocatorT>
  accessor(image<dimensions, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef);

  /* Available only when: accessTarget == access::target::image_array &&
  dimensions < 3 */
  template <typename AllocatorT>
  accessor(image<dimensions + 1, AllocatorT> &imageRef,
    handler &commandGroupHandlerRef);

  /* -- common interface members -- */

  size_t get_size() const;

  size_t get_count() const;

  /* Available only when: (accessTarget == access::target::image || accessTarget
  == access::target::host_image) && accessMode == access::mode::read */
  template <typename coordT>
  dataT read(const coordT &coords) const;

  /* Available only when: (accessTarget == access::target::image || accessTarget
  == access::target::host_image) && accessMode == access::mode::read */
  template <typename coordT>
  dataT read(const coordT &coords, const sampler &smpl) const;

  /* Available only when: (accessTarget == access::target::image || accessTarget
  == access::target::host_image) && accessMode == access::mode::write ||
  accessMode == access::mode::discard_write */
  template <typename coordT>
  void write(const coordT &coords, const dataT &color) const;

  /* Available only when: accessTarget == access::target::image_array &&
  dimensions < 3 */
  __image_array_slice__ operator[](size_t index) const
};
}  // namespace sycl
}  // namespace cl
```

Listing 4.3: Accessor interface for images.

Constructor	Description
`template <typename AllocatorT> accessor(image<dimensions, AllocatorT> &imageRef)`	Available only when: `accessTarget == access::target::host_image`. Constructs a SYCL `accessor` instance for accessing a SYCL `image` immediately on the host.
`template <typename AllocatorT> accessor(image<dimensions, AllocatorT> &imageRef, handler &commandGroupHandlerRef)`	Available only when: `accessTarget == access::target::image`. Constructs a SYCL `accessor` instance for accessing a SYCL `image` within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`.
`template <typename AllocatorT> accessor(image<dimensions + 1, AllocatorT> &imageRef, handler &commandGroupHandlerRef)`	Available only when: `accessTarget == access::target::image_array && dimensions < 3`. Constructs a SYCL `accessor` instance for accessing a SYCL `image` as an array of images, within a SYCL kernel function on the SYCL `queue` associated with `commandGroupHandlerRef`.
	End of table

Table 4.51: Constructors of the `accessor` class template image specialization.

Member function	Description
`size_t get_size()const`	Returns the size in bytes of the SYCL `image` this SYCL `accessor` is accessing.
`size_t get_count()const`	Returns the number of elements of the SYCL `image` this SYCL `accessor` is accessing.
`template <typename coordT> dataT read(const coordT &coords)const`	Available only when: `(accessTarget == access::target::image || accessTarget == access::target::host_image)&& accessMode == access::mode::read`. Reads and returns an element of the image at the coordinates specified by `coords`. Permitted types for `coordT` are `cl_int` and `cl_float` when `dimensions == 1`, `cl_int2` and `cl_float2` when `dimensions == 2` and `cl_int4` and `cl_float4` when `dimensions == 3`.
	Continued on next page

Table 4.52: Member functions of the `accessor` class template image specialization.

Member function	Description
`template <typename coordT> dataT read(const coordT &coords, const sampler &smpl )const`	Available only when: `(accessTarget == access::target::image || accessTarget == access::target::host_image)&& accessMode == access::mode::read.` Reads and returns a sampled element of the image at the coordinates specified by `coords` using the sampler specified by `smpl`. Permitted types for `coordT` are `cl_int` and `cl_float` when `dimensions == 1`, `cl_int2` and `cl_float2` when `dimensions == 2` and `cl_int4` and `cl_float4` when `dimensions == 3`.
`template <typename coordT> void write(const coordT &coords, const dataT &color) const`	Available only when: `accessTarget == access::target::image || accessTarget == access::target::host_image)&& accessMode == access::mode::write || accessMode == access::mode:: discard_write.` Writes the value specified by `color` to the element of the image at the coordinates specified by `coords`. Permitted types for `coordT` are `cl_int` when `dimensions == 1`, `cl_int2` when `dimensions == 2` and `cl_int4` when `dimensions == 3`.
`__image_array_slice__ operator[](size_t index)const`	Available only when: `accessTarget == access::target::image_array && dimensions < 3.` Returns an instance of `__image_array_slice__`, an unspecified type which provides the interface of `accessor< dataT, dimensions, mode , access::target::image, access:: placeholder:: false_>` which will provide access to a slice of the image array specified by `index`.
	End of table

Table 4.52: Member functions of the `accessor` class template image specialization.

Address space classes

In OpenCL, there are four different address spaces. These are: global, local, constant and private. In OpenCL C, these address spaces are manually specified using OpenCL-specific keywords. In SYCL, the device compiler is expected to auto-deduce the address space for pointers in common situations of pointer usage. However, there are situations where auto-deduction is not possible. Here are the most common situations:

- When linking SYCL kernels with OpenCL C functions. In this case, it is necessary to specify the address space for any pointer parameters when declaring an `extern "C"` function.
- When declaring data structures with pointers inside, it is not possible for the SYCL compiler to deduce at the time of declaration of the data structure what address space pointer values assigned to members of the structure will be. So, in this case, the address spaces will have to be explicitly declared by the developer.
- When a pointer is declared as a variable, but not initialized, then address space deduction is not automatic and so an explicit pointer class should be used, or the pointer should be initialized at declaration.

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.

Multi-pointer class

The multi-pointer class is the common interface for the explicit pointer classes, defined in 4.7.7.2.

There are situations where a user may want to template a data structure by an address space. Or, a user may want to write templates that adapt to the address space of a pointer. 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 order to facilitate SYCL/OpenCL C interoperability, the `pointer` type is provided. It is an implementation defined type which corresponds to the underlying OpenCL C pointer type and can be used in `extern "C"` function declarations for OpenCL functions used in SYCL kernels. `multi_ptr` class defines a `get` member function that returns the underlying OpenCL C pointer.

The `multi_ptr` class provides constructors for address space qualified and non address space qualified pointers to allow interoperability between plain C++ and OpenCL C. Implementations should reject programs that try assign a pointer with an address space not consistent with the address space represented by the `multi_ptr` specialization.

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<void>` to any other `multi_ptr` type of the same address space (and the other way around) 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.

```
1  namespace cl {
2  namespace sycl {
3  namespace access {
4  enum class address_space : int {
```

```
5     global_space,
6     local_space,
7     constant_space,
8     private_space
9   };
10  }  // namespace access
11
12  template <typename ElementType, access::address_space Space>
13  class multi_ptr {
14   public:
15    using element_type = ElementType;
16    using difference_type = std::ptrdiff_t;
17
18    // Implementation defined pointer and reference types that correspond to
19    // SYCL/OpenCL interoperability types for OpenCL C functions
20    using pointer_t = __unspecified__ ElementType*;
21    using const_pointer_t = __unspecified__ const ElementType*;
22    using reference_t = __unspecified__ ElementType&;
23    using const_reference_t = __unspecified__ const ElementType&;
24
25    static constexpr access::address_space address_space = Space;
26
27    // Constructors
28    multi_ptr();
29    multi_ptr(const multi_ptr&);
30    multi_ptr(multi_ptr&&);
31    multi_ptr(pointer_t);
32    multi_ptr(ElementType*);
33    multi_ptr(std::nullptr_t);
34    ~multi_ptr();
35
36    // Assignment and access operators
37    multi_ptr &operator=(const multi_ptr&);
38    multi_ptr &operator=(multi_ptr&&);
39    multi_ptr &operator=(pointer_t);
40    multi_ptr &operator=(ElementType*);
41    multi_ptr &operator=(std::nullptr_t);
42    ElementType& operator*() const;
43    ElementType* operator->() const;
44
45    // Only if Space == global_space
46    template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
47    multi_ptr(accessor<ElementType, dimensions, Mode, access::target::global_buffer, isPlaceholder>)
          ;
48
49    // Only if Space == local_space
50    template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
51    multi_ptr(accessor<ElementType, dimensions, Mode, access::target::local, isPlaceholder>);
52
53    // Only if Space == constant_space
54    template <int dimensions, access::mode Mode, access::placeholder isPlaceholder>
55    multi_ptr(accessor<ElementType, dimensions, Mode, access::target::constant_buffer, isPlaceholder
          >);
56
57    // Returns the underlying OpenCL C pointer
```

```
58    pointer_t get() const;
59
60    // Implicit conversion to the underlying pointer type
61    operator ElementType*() const;
62
63    // Implicit conversion to a multi_ptr<void>
64    // Only available when ElementType is not const-qualified
65    operator multi_ptr<void, Space>() const;
66
67    // Implicit conversion to a multi_ptr<const void>
68    // Only available when ElementType is const-qualified
69    operator multi_ptr<const void, Space>() const;
70
71    // Implicit conversion to multi_ptr<const ElementType, Space>
72    operator multi_ptr<const ElementType, Space>() const;
73
74    // Arithmetic operators
75    multi_ptr& operator++();
76    multi_ptr operator++(int);
77    multi_ptr& operator--();
78    multi_ptr operator--(int);
79    multi_ptr& operator+=(difference_type r);
80    multi_ptr& operator-=(difference_type r);
81    multi_ptr operator+(difference_type r) const;
82    multi_ptr operator-(difference_type r) const;
83
84    void prefetch(size_t numElements) const;
85  };
86
87  // Specialization of multi_ptr for void and const void
88  // VoidType can be either void or const void
89  template <access::address_space Space>
90  class multi_ptr<VoidType, Space> {
91   public:
92    using element_type = VoidType;
93    using difference_type = std::ptrdiff_t;
94
95    // Implementation defined pointer types that correspond to
96    // SYCL/OpenCL interoperability types for OpenCL C functions
97    using pointer_t = __unspecified__ VoidType*;
98    using const_pointer_t = __unspecified__ const VoidType*;
99
100   static constexpr access::address_space address_space = Space;
101
102   // Constructors
103   multi_ptr();
104   multi_ptr(const multi_ptr&);
105   multi_ptr(multi_ptr&&);
106   multi_ptr(pointer_t);
107   multi_ptr(VoidType*);
108   multi_ptr(std::nullptr_t);
109   ~multi_ptr();
110
111   // Assignment operators
112   multi_ptr &operator=(const multi_ptr&);
```

```
113   multi_ptr &operator=(multi_ptr&&);
114   multi_ptr &operator=(pointer_t);
115   multi_ptr &operator=(VoidType*);
116   multi_ptr &operator=(std::nullptr_t);
117
118   // Only if Space == global_space
119   template <typename ElementType, int dimensions, access::mode Mode>
120   multi_ptr(accessor<ElementType, dimensions, Mode, access::target::global_buffer>);
121
122   // Only if Space == local_space
123   template <typename ElementType, int dimensions, access::mode Mode>
124   multi_ptr(accessor<ElementType, dimensions, Mode, access::target::local>);
125
126   // Only if Space == constant_space
127   template <typename ElementType, int dimensions, access::mode Mode>
128   multi_ptr(accessor<ElementType, dimensions, Mode, access::target::constant_buffer>);
129
130   // Returns the underlying OpenCL C pointer
131   pointer_t get() const;
132
133   // Implicit conversion to the underlying pointer type
134   operator VoidType*() const;
135
136   // Explicit conversion to a multi_ptr<ElementType>
137   // If VoidType is const, ElementType must be as well
138   template <typename ElementType>
139   explicit operator multi_ptr<ElementType, Space>() const;
140
141   // Implicit conversion to multi_ptr<const void, Space>
142   operator multi_ptr<const void, Space>() const;
143 };
144
145 template <typename ElementType, access::address_space Space>
146 multi_ptr<ElementType, Space> make_ptr(multi_ptr<ElementType, Space>::pointer_t);
147
148 template <typename ElementType, access::address_space Space>
149 multi_ptr<ElementType, Space> make_ptr(ElementType*);
150
151 template <typename ElementType, access::address_space Space>
152 bool operator==(const multi_ptr<ElementType, Space>& lhs,
153                 const multi_ptr<ElementType, Space>& rhs);
154 template <typename ElementType, access::address_space Space>
155 bool operator!=(const multi_ptr<ElementType, Space>& lhs,
156                 const multi_ptr<ElementType, Space>& rhs);
157 template <typename ElementType, access::address_space Space>
158 bool operator<(const multi_ptr<ElementType, Space>& lhs,
159                const multi_ptr<ElementType, Space>& rhs);
160 template <typename ElementType, access::address_space Space>
161 bool operator>(const multi_ptr<ElementType, Space>& lhs,
162                const multi_ptr<ElementType, Space>& rhs);
163 template <typename ElementType, access::address_space Space>
164 bool operator<=(const multi_ptr<ElementType, Space>& lhs,
165                 const multi_ptr<ElementType, Space>& rhs);
166 template <typename ElementType, access::address_space Space>
167 bool operator>=(const multi_ptr<ElementType, Space>& lhs,
```

```
168                 const multi_ptr<ElementType, Space>& rhs);
169  template <typename ElementType, access::address_space Space>
170  bool operator!=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
171  template <typename ElementType, access::address_space Space>
172  bool operator!=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
173  template <typename ElementType, access::address_space Space>
174  bool operator==(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
175  template <typename ElementType, access::address_space Space>
176  bool operator==(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
177  template <typename ElementType, access::address_space Space>
178  bool operator>(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
179  template <typename ElementType, access::address_space Space>
180  bool operator>(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
181  template <typename ElementType, access::address_space Space>
182  bool operator<(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
183  template <typename ElementType, access::address_space Space>
184  bool operator<(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
185  template <typename ElementType, access::address_space Space>
186  bool operator>=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
187  template <typename ElementType, access::address_space Space>
188  bool operator>=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
189  template <typename ElementType, access::address_space Space>
190  bool operator<=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs);
191  template <typename ElementType, access::address_space Space>
192  bool operator<=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs);
193
194  }  // namespace sycl
195  }  // namespace cl
```

Constructor	Description
`template <typename ElementType, access::address_space Space> multi_ptr()`	Default constructor.
`template <typename ElementType, access::address_space Space> multi_ptr(const multi_ptr &)`	Copy constructor.
`template <typename ElementType, access::address_space Space> multi_ptr(multi_ptr&&)`	Move constructor.
`template <typename ElementType, access::address_space Space> multi_ptr(pointer_t)`	Constructor that takes as an argument a pointer of type `ElementType`.
`template <typename ElementType, access::address_space Space> multi_ptr(ElementType*)`	Constructor that takes as an argument a pointer of type `ElementType`. An implementation should reject an argument if the deduced address space is not compatible with `Space`.
`template <typename ElementType, access::address_space Space> multi_ptr(std::nullptr_t)`	Constructor from a `nullptr`.

Continued on next page

Table 4.53: Constructors of the SYCL `multi_ptr` class template.

Constructor	Description
`template <typename ElementType, access::address_space Space = access::address_space::global_space> template <int dimensions, access::mode Mode> multi_ptr( accessor<ElementType, dimensions, Mode, access::target::global_buffer>)`	Constructs a `multi_ptr<ElementType, access::address_space::global_space>` from an accessor of `access::target::global_buffer`.
`template <typename ElementType, access::address_space Space = access::address_space::local_space> template <int dimensions, access::mode Mode> multi_ptr( accessor<ElementType, dimensions, Mode, access::target::local>)`	Constructs a `multi_ptr<ElementType, access::address_space::local_space>` from an accessor of `access::target::local`.
`template <typename ElementType, access::address_space Space = access::address_space::constant_space> template <int dimensions, access::mode Mode> multi_ptr( accessor<ElementType, dimensions, Mode, access::target::constant_buffer>)`	Constructs a `multi_ptr<ElementType, access::address_space::constant_space>` from an accessor of `access::target::constant_buffer`.
`template <typename ElementType, access::address_space Space> multi_ptr<ElementType, Space> make_ptr( ElementType*)`	Global function to create a `multi_ptr` instance depending on the address space of the `pointer` type. An implementation must reject an argument if the deduced address space is not compatible with `Space`.
`template <typename ElementType, access::address_space Space> multi_ptr<ElementType, Space> make_ptr(multi_ptr <ElementType, Space>::pointer_t)`	Global function to create a `multi_ptr` instance from an OpenCL C `pointer`.
	End of table

Table 4.53: Constructors of the SYCL `multi_ptr` class template.

Member function	Description
`template <typename ElementType, access::address_space Space> multi_ptr &operator=(const multi_ptr&)`	Copy assignment operator.
`template <typename ElementType, access::address_space Space> multi_ptr &operator=(multi_ptr&&)`	Move assignment operator.
`template <typename ElementType, access::address_space Space> multi_ptr &operator=(pointer_t)`	Assigns a pointer of `ElementType` to the `multi_ptr`.
	Continued on next page

Table 4.54: Member functions of `multi_ptr` class.

Member function	Description
`template <typename ElementType, access::address_space Space> multi_ptr &operator=(ElementType*)`	Assigns a pointer of type `ElementType`. An implementation should reject an argument if the deduced address space is not compatible with `Space`.
`template <typename ElementType, access::address_space Space> multi_ptr &operator=(std::nullptr_t)`	Assigns `nullptr` to the `multi_ptr`.
`template <typename ElementType, access::address_space Space> ElementType& operator*()const`	Available only when: `!std::is_void<ElementType>::value`. Operator that returns a reference to the `ElementType` of the `multi_ptr` class.
`template <typename ElementType, access::address_space Space> ElementType* operator->()const`	Available only when: `!std::is_void<ElementType>::value`. Returns the underlying pointer.
`template <typename ElementType, access::address_space Space> pointer_t get()const`	Returns the underlying OpenCL C pointer.
`template <typename ElementType, access::address_space Space> operator ElementType*()const`	Implicit conversion to the underlying pointer type.
`template <typename ElementType, access::address_space Space> operator multi_ptr<void, Space>()const`	Available only when: `!std::is_void<ElementType>::value && !std::is_const<ElementType>::value`. Implicit conversion to a `multi_ptr` of type `void`.
`template <typename ElementType, access::address_space Space> operator multi_ptr<const void, Space>()const`	Available only when: `!std::is_void<ElementType>::value && std::is_const<ElementType>::value`. Implicit conversion to a `multi_ptr` of type `const void`.
`template <access::address_space Space> operator multi_ptr<const ElementType, Space>() const`	Implicit conversion to a `multi_ptr` of type `const ElementType`.
`template <access::address_space Space> template <typename ElementType> explicit operator multi_ptr<ElementType, Space >()const`	Available only for the `multi_ptr<void>` and `multi_ptr<const void>` specializations. Explicit conversion of a `multi_ptr<void>` or `multi_ptr<const void>` pointer object to a `multi_ptr` of type `ElementType`.
The conversion must retain the `const` qualifier. `template < typename ElementType, access::address_space Space> multi_ptr& operator++()`	Available only when: `!std::is_void<ElementType>::value`. Increments the pointer by 1.
`template <typename ElementType, access::address_space Space> multi_ptr operator++(int)`	Available only when: `!std::is_void<ElementType>::value`. Increments the pointer by 1 and returns a new `multi_ptr` with the value of the previous pointer.

Continued on next page

Table 4.54: Member functions of `multi_ptr` class.

Member function	Description
`template <typename ElementType, access::address_space Space> multi_ptr& operator--()`	Available only when: `!std::is_void<ElementType>::value`. Decrements the pointer by 1.
`template <typename ElementType, access::address_space Space> multi_ptr operator--(int)`	Available only when: `!std::is_void<ElementType>::value`. Decrements the pointer by 1 and returns a new `multi_ptr` with the value of the previous pointer.
`template <typename ElementType, access::address_space Space> multi_ptr& operator+=(difference_type r)`	Available only when: `!std::is_void<ElementType>::value`. Moves the pointer forward by `r`.
`template <typename ElementType, access::address_space Space> multi_ptr& operator-=(difference_type r)`	Available only when: `!std::is_void<ElementType>::value`. Moves the pointer backward by `r`.
`template <typename ElementType, access::address_space Space> multi_ptr operator+(difference_type r)const`	Available only when: `!std::is_void<ElementType>::value`. Creates a new `multi_ptr` that points `r` forward compared to `*this`.
`template <typename ElementType, access::address_space Space> multi_ptr operator-(difference_type r)const`	Available only when: `!std::is_void<ElementType>::value`. Creates a new `multi_ptr` that points `r` backward compared to `*this`.
`void prefetch(size_t numElements)const`	Available only when: `Space == access::address_space::global_space`. Prefetches a number of elements specified by `numElements` into the global memory cache. This operation is an implementation defined optimization and does not effect the functional behavior of the SYCL kernel function.
	End of table

Table 4.54: Member functions of `multi_ptr` class.

Non-member function	Description
`template <typename ElementType, access::address_space Space> bool operator==(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator `==` for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator!=(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator `!=` for `multi_ptr` class.
	Continued on next page

Table 4.55: Non-member functions of the `multi_ptr` class.

Non-member function	Description
`template <typename ElementType, access::address_space Space> bool operator<(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator < for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator>(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator > for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator<=(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator <= for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator>=(const multi_ptr<ElementType, Space>& lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator >= for `multi_ptr` class.
`template <typename ElementType, access::address_space Space> bool operator!=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator != for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access::address_space Space> bool operator!=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator != for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access::address_space Space> bool operator==(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator == for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access::address_space Space> bool operator==(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator == for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access::address_space Space> bool operator>(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator > for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access::address_space Space> bool operator>(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator > for `multi_ptr` class with a `std::nullptr_t`.

Continued on next page

Table 4.55: Non-member functions of the `multi_ptr` class.

Non-member function	Description
`template <typename ElementType, access:: address_space Space> bool operator<(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator < for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access:: address_space Space> bool operator<(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator < for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access:: address_space Space> bool operator>=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator >= for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access:: address_space Space> bool operator>=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator >= for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access:: address_space Space> bool operator<=(const multi_ptr<ElementType, Space>& lhs, std::nullptr_t rhs)`	Comparison operator <= for `multi_ptr` class with a `std::nullptr_t`.
`template <typename ElementType, access:: address_space Space> bool operator<=(std::nullptr_t lhs, const multi_ptr<ElementType, Space>& rhs)`	Comparison operator <= for `multi_ptr` class with a `std::nullptr_t`.
	End of table

Table 4.55: Non-member functions of the `multi_ptr` class.

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.

```
1  namespace cl {
2  namespace sycl {
3
4  template <typename ElementType, access::address_space Space>
5  class multi_ptr;
6
7  // Template specialization aliases for different pointer address spaces
8
9  template <typename ElementType>
10 using global_ptr = multi_ptr<ElementType, access::address_space::global_space>;
11
12 template <typename ElementType>
13 using local_ptr = multi_ptr<ElementType, access::address_space::local_space>;
14
```

```
15  template <typename ElementType>
16  using constant_ptr =
17      multi_ptr<ElementType, access::address_space::constant_space>;
18
19  template <typename ElementType>
20  using private_ptr =
21      multi_ptr<ElementType, access::address_space::private_space>;
22
23  }  // namespace sycl
24  }  // namespace cl
```

Samplers

The SYCL `sampler` class encapsulates a configuration for sampling an image `accessor`. A SYCL `sampler` may be an OpenCL sampler, in which case it must encapsulate a valid underlying OpenCL `cl_sampler`, or it may be a host sampler, in which case it must not.

The constructors and member functions of the SYCL `sampler` class are listed in Tables 4.59 and 4.60, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

The members of the `sampler` class that provide information on the sampler (`get_addressing_mode()`, `get_filtering_mode()`, `get_coordinate_normalization_mode()`) are callable from host code. Invoking these queries within device kernel code produces undefined results.

```
1   namespace cl {
2   namespace sycl {
3
4   enum class addressing_mode: unsigned int {
5     mirrored_repeat,
6     repeat,
7     clamp_to_edge,
8     clamp,
9     none
10  };
11
12  enum class filtering_mode: unsigned int {
13    nearest,
14    linear
15  };
16
17  enum class coordinate_normalization_mode : unsigned int {
18    normalized,
19    unnormalized
20  };
21
22  class sampler {
23   public:
24    sampler(coordinate_normalization_mode normalizationMode, addressing_mode addressingMode,
25            filtering_mode filteringMode);
26
```

```
27    sampler(cl_sampler clSampler, const context &syclContext);
28
29    /* -- common interface members -- */
30
31    addressing_mode get_addressing_mode() const;
32
33    filtering_mode get_filtering_mode() const;
34
35    coordinate_normalization_mode get_coordinate_normalization_mode() const;
36  };
37  }  // namespace sycl
38  }  // namespace cl
```

addressing_mode	Description
mirrored_repeat	Out of range coordinates will be flipped at every integer junction. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.
repeat	Out of range image coordinates are wrapped to the valid range. This addressing mode can only be used with normalized coordinates. If normalized coordinates are not used, this addressing mode may generate image coordinates that are undefined.
clamp_to_edge	Out of range image coordinates are clamped to the extent.
clamp	Out of range image coordinates will return a border color.
none	For this addressing mode the programmer guarantees that the image coordinates used to sample elements of the image refer to a location inside the image; otherwise the results are undefined.
	End of table

Table 4.56: Addressing modes description.

filtering_mode	Description
nearest	Chooses a color of nearest pixel.
linear	Performs a linear sampling of adjacent pixels.
	End of table

Table 4.57: Filtering modes description.

coordinate_normalization_mode	Description
normalized	Normalizes image coordinates.
unnormalized	Does not normalize image coordinates.
	End of table

Table 4.58: Coordinate normalization modes description.

Constructor	Description
sampler(coordinate_normalization_mode normalizationMode, addressing_mode addressingMode, filtering_mode filteringMode)	Constructs a SYCL sampler instance with address mode, filtering mode and coordinate normalization mode specified by the respective parameters. It is not valid to construct a SYCL sampler within a SYCL kernel function.
sampler(cl_sampler clSampler, const context &syclContext)	Constructs a SYCL sampler instance from an OpenCL cl_sampler in accordance with the requirements described in 4.3.1.
	End of table

Table 4.59: Constructors the sampler class.

Member function	Description
addressing_mode get_addressing_mode()const	Return the addressing mode used to construct this SYCL sampler.
filtering_mode get_filtering_mode()const	Return the filtering mode used to construct this SYCL sampler.
coordinate_normalization_mode get_coordinate_normalization_mode()const	Return the coordinate normalization mode used to construct this SYCL sampler.
	End of table

Table 4.60: Member functions for the sampler class.

Expressing parallelism through kernels

Ranges and index space identifiers

The data parallelism of the OpenCL execution model and its exposure through SYCL requires instantiation of a parallel execution over a range of iteration space coordinates. To achieve this we expose types to define the range of execution and to identify a given execution instance's point in the iteration space.

To achieve this we expose seven types: range, nd_range, id, item, h_item, nd_item and group.

When constructing ids or ranges from integers, the elements are written in row-major format.

Type	Description
id	A point within a range
range	Bounds over which an id may vary
item	Pairing of an id (specific point) and the range that it is bounded by
nd_range	Encapsules both global and local (work-group size) ranges over which work-item ids will vary
nd_item	Encapsulates two items, one for global id and range, and one for local id and range
h_item	Index point queries within hierarchical parallelism (parallel_for_work_item). Encapsulates physical global and local ids and ranges, as well as a logical/flexible local id and range defined by hierarchical parallelism
group	Work-group queries within hierarchical parallelism (parallel_for_work_group), and exposes the parallel_for_work_item construct that identifies code to be executed by each work-item. Encapsulates work-group ids and ranges
	End of table

Table 4.61: Summary of types used to identify points in an index space, and ranges over which those points can vary.

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 4.3.3).

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 Tables 4.67, 4.63 and 4.64 respectively. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions = 1>
4  class range {
5  public:
6    /* The following constructor is only available in the range class specialization where:
         dimensions==1 */
7    range(size_t dim0);
8    /* The following constructor is only available in the range class specialization where:
         dimensions==2 */
9    range(size_t dim0, size_t dim1);
10   /* The following constructor is only available in the range class specialization where:
```

```
        dimensions==3 */
11    range(size_t dim0, size_t dim1, size_t dim2);
12
13     /* -- common interface members -- */
14
15    size_t get(int dimension) const;
16    size_t &operator[](int dimension);
17    size_t operator[](int dimension) const;
18
19    size_t size() const;
20
21    // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
22    range<dimensions> operatorOP(const range<dimensions> &rhs) const;
23    range<dimensions> operatorOP(const size_t &rhs) const;
24
25    // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
26    range<dimensions> &operatorOP(const range<dimensions> &rhs);
27    range<dimensions> &operatorOP(const size_t &rhs);
28  };
29
30  // OP is: +, -, *, /, %, <<, >>, &, |, ^
31  template <int dimensions>
32  range<dimensions> operatorOP(const size_t &lhs, const range<dimensions> &rhs);
33  }  // sycl
34  }  // cl
```

Constructor	Description
`range(size_t dim0)`	Construct a 1D range with value dim0. Only valid when the template parameter `dimensions` is equal to 1.
`range(size_t dim0, size_t dim1)`	Construct a 2D range with values dim0 and dim1. Only valid when the template parameter `dimensions` is equal to 2.
`range(size_t dim0, size_t dim1, size_t dim2)`	Construct a 3D range with values dim0, dim1 and dim2. Only valid when the template parameter `dimensions` is equal to 3.
	End of table

Table 4.62: Constructors of the `range` class template.

Member function	Description
`size_t get(int dimension)const`	Return the value of the specified dimension of the `range`.
`size_t &operator[](int dimension)`	Return the l-value of the specified dimension of the `range`.
`size_t operator[](int dimension)const`	Return the value of the specified dimension of the `range`.
`size_t size()const`	Return the size of the range computed as dimension0*...*dimensionN.
	Continued on next page

Table 4.63: Member functions of the `range` class template.

Member function	Description
range<dimensions> operatorOP(const range<dimensions> &rhs)const	Where OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=. Constructs and returns a new instance of the SYCL range class template with the same dimensionality as this SYCL range, where each element of the new SYCL range instance is the result of an element-wise OP operator between each element of this SYCL range and each element of the rhs range. If the operator returns a bool the result is the cast to size_t.
range<dimensions> operatorOP(const size_t &rhs)const	Where OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=. Constructs and returns a new instance of the SYCL range class template with the same dimensionality as this SYCL range, where each element of the new SYCL range instance is the result of an element-wise OP operator between each element of this SYCL range and the rhs size_t. If the operator returns a bool the result is the cast to size_t .
range<dimensions> &operatorOP(const range<dimensions > &rhs)	Where OP is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=. Assigns each element of this SYCL range instance with the result of an element-wise OP operator between each element of this SYCL range and each element of the rhs range and returns a reference to this SYCL range. If the operator returns a bool the result is the cast to size_t.
range<dimensions> &operatorOP(const size_t &rhs)	Where OP is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=. Assigns each element of this SYCL range instance with the result of an element-wise OP operator between each element of this SYCL range and the rhs size_t and returns a reference to this SYCL range. If the operator returns a bool the result is the cast to size_t.
	End of table

Table 4.63: Member functions of the range class template.

Non-member function	Description
`template <int dimensions> range<dimensions> operatorOP(const size_t &lhs, const range<dimensions> &rhs)`	Where `OP` is: +, -, *, /, %, <<, >>, &, |, ^. 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`.
	End of table

Table 4.64: Non-member functions of the SYCL `range` class template.

nd_range class

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions = 1>
4  struct nd_range {
5
6     /* -- common interface members -- */
7
8    nd_range(range<dimensions> globalSize, range<dimensions> localSize,
9             id<dimensions> offset = id<dimensions>());
10
11   range<dimensions> get_global_range() const;
12   range<dimensions> get_local_range() const;
13   range<dimensions> get_group_range() const;
14   id<dimensions> get_offset() const;
15 };
16 }  // namespace sycl
17 }  // namespace cl
```

`nd_range<int 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 4.3.3).

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 Tables 4.65 and 4.66 respectively. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

Constructor	Description

`nd_range<dimensions>(`
`  range<dimensions> globalSize,`
`  range<dimensions> localSize)`
`  id<dimensions> offset = id<dimensions>())`
	Construct an `nd_range` from the local and global constituent ranges as well as an optional offset. If the offset is not provided it will default to no offset.

	End of table

Table 4.65: Constructors of the `nd_range` class.

Member function	Description
`range<dimensions> get_global_range()const`	Return the constituent global range.
`range<dimensions> get_local_range()const`	Return the constituent local range.
`range<dimensions> get_group_range()const`	Return a range representing the number of groups in each dimension. This range would result from `globalSize/localSize` as provided on construction.
`id<dimensions> get_offset()const`	Return the constituent offset.
	End of table

Table 4.66: Member functions for the `nd_range` class.

`id` class

`id<int` dimensions> is a vector of dimensions that is used to represent an *index* into a global or local `range`. It can be used as an index in an accessor of the same rank. The `[n]` operator returns the component `n` as an `size_t`.

The SYCL `id` class template provides the common by-value semantics (see Section 4.3.3).

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 Tables 4.67, 4.68 and 4.69 respectively. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions = 1>
4  struct id {
5    id();
6
7    /* The following constructor is only available in the id class
8     * specialization where: dimensions==1 */
9    id(size_t dim0);
10   /* The following constructor is only available in the id class
11    * specialization where: dimensions==2 */
12   id(size_t dim0, size_t dim1);
13   /* The following constructor is only available in the id class
14    * specialization where: dimensions==3 */
15   id(size_t dim0, size_t dim1, size_t dim2);
16
17    /* -- common interface members -- */
18
```

```
19    id(const range<dimensions> &range);
20    id(const item<dimensions> &item);
21
22    size_t get(int dimension) const;
23    size_t &operator[](int dimension);
24    size_t operator[](int dimension) const;
25
26    // OP is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=
27    id<dimensions> operatorOP(const id<dimensions> &rhs) const;
28    id<dimensions> operatorOP(const size_t &rhs) const;
29
30    // OP is: +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=
31    id<dimensions> &operatorOP(const id<dimensions> &rhs);
32    id<dimensions> &operatorOP(const size_t &rhs);
33  };
34
35  // OP is: +, -, *, /, %, <<, >>, &, |, ^
36  template <int dimensions>
37  id<dimensions> operatorOP(const size_t &lhs, const id<dimensions> &rhs);
38  }  // namespace sycl
39  }  // namespace cl
```

Constructor	Description
`id()`	Construct a SYCL `id` with the value `0` for each dimension.
`id(size_t dim0)`	Construct a 1D `id` with value dim0. Only valid when the template parameter `dimensions` is equal to 1.
`id(size_t dim0, size_t dim1)`	Construct a 2D `id` with values dim0, dim1. Only valid when the template parameter `dimensions` is equal to 2.
`id(size_t dim0, size_t dim1, size_t dim2)`	Construct a 3D `id` with values dim0, dim1, dim2. Only valid when the template parameter `dimensions` is equal to 3.
`id(const range<dimensions> &range)`	Construct an `id` from the dimensions of `r`.
`id(const item<dimensions> &item)`	Construct an `id` from `item.get_id()`.
	End of table

Table 4.67: Constructors of the `id` class template.

Member function	Description
`size_t get(int dimension)const`	Return the value of the `id` for dimension `dimension`.
`size_t &operator[](int dimension)`	Return a reference to the requested dimension of the `id` object.
`size_t operator[](int dimension)const`	Return the value of the requested dimension of the `id` object.
	Continued on next page

Table 4.68: Member functions of the `id` class template.

Member function	Description
`id<dimensions> operatorOP(const id<dimensions> &rhs) const`	Where `OP` is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=. Constructs and returns a new instance of the SYCL `id` class template with the same dimensionality as this SYCL `id`, where each element of the new SYCL `id` instance is the result of an element-wise `OP` operator between each element of this SYCL `id` and each element of the `rhs` `id`. If the operator returns a `bool` the result is the cast to `size_t`.
`id<dimensions> operatorOP(const size_t &rhs)const`	Where `OP` is: +, -, *, /, %, <<, >>, &, |, ^, &&, ||, <, >, <=, >=. Constructs and returns a new instance of the SYCL `id` class template with the same dimensionality as this SYCL `id`, where each element of the new SYCL `id` instance is the result of an element-wise `OP` operator between each element of this SYCL `id` and the `rhs` `size_t`. If the operator returns a `bool` the result is the cast to `size_t`.
`id<dimensions> &operatorOP(const id<dimensions> &rhs )`	Where `OP` is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=. Assigns each element of this SYCL `id` instance with the result of an element-wise `OP` operator between each element of this SYCL `id` and each element of the `rhs` `id` and returns a reference to this SYCL `id`. If the operator returns a `bool` the result is the cast to `size_t`.
`id<dimensions> &operatorOP(const size_t &rhs)`	Where `OP` is: +=, -=,*=, /=, %=, <<=, >>=, &=, |=, ^=. Assigns each element of this SYCL `id` instance with the result of an element-wise `OP` operator between each element of this SYCL `id` and the `rhs` `size_t` and returns a reference to this SYCL `id`. If the operator returns a `bool` the result is the cast to `size_t`.
	End of table

Table 4.68: Member functions of the `id` class template.

Non-member function	Description
`template <int dimensions> id<dimensions> operatorOP(const size_t &lhs, const id<dimensions> &rhs`	Where `OP` is: +, -, *, /, %, <<, >>, &, |, ^. Constructs and returns a new instance of the SYCL `id` class template with the same dimensionality as the `rhs` SYCL `id`, where each element of the new SYCL `id` instance is the result of an element-wise `OP` operator between the `lhs` `size_t` and each element of the `rhs` SYCL `id`. If the operator returns a `bool` the result is the cast to `size_t`.
	End of table

Table 4.69: Non-member functions of the `id` class template.

`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`. 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 4.3.3).

Item interface

A synopsis of the SYCL `item` class is provided below. The member functions of the SYCL `item` class are listed in Table 4.68. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions = 1, bool with_offset = true>
4  struct item {
5    item() = delete;
6
7     /* -- common interface members -- */
8
9    id<dimensions> get_id() const;
10
11   size_t get_id(int dimension) const;
12
13   size_t operator[](int dimension) const;
14
15   range<dimensions> get_range() const;
16
17   size_t get_range(int dimension) const;
18
19   // only available if with_offset is true
```

```
20    id<dimensions> get_offset() const;
21
22    // only available if with_offset is false
23    operator item<dimensions, true>() const;
24
25    size_t get_linear_id() const;
26  };
27  }  // namespace sycl
28  }  // namespace cl
```

Member function	Description
`id<dimensions> get_id()const`	Return the constituent `id` representing the work-item's position in the iteration space.
`size_t get_id(int dimension)const`	Return the requested dimension of the constituent `id` representing the work-item's position in the iteration space.
`size_t operator[](int dimension)const`	Return the constituent `id` value representing the work-item's position in the iteration space in the given `dimension`.
`range<dimensions> get_range()const`	Returns a `range` representing the dimensions of the range of possible values of the `item`.
`size_t get_range(int dimension)const`	Return the same value as get_range().get(dimension)
`id<dimensions> get_offset()const`	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`.
`operator item<dimensions, true>()const`	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`. This member function is only available if `with_offset` is `false`.
`size_t get_linear_id()const`	Return the id as a linear index value. Calculating a linear address from the multi-dimensional index follow the equation 4.3.
	End of table

Table 4.70: Member functions for the `item` class.

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 the barrier and mem_fence member functions, for performing work-group barrier and work-group mem-fence operations respectively. 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 4.3.3).

A synopsis of the SYCL nd_item class is provided below. The member functions of the SYCL nd_item class are listed in Table 4.71. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

```
namespace cl {
namespace sycl {
template <int dimensions = 1>
struct nd_item {
  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;

  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;

  id<dimensions> get_offset() const;
```

```
40
41    nd_range<dimensions> get_nd_range() const;
42
43    void barrier(access::fence_space accessSpace =
44      access::fence_space::global_and_local) const;
45
46    template <access::mode accessMode = access::mode::read_write>
47    void mem_fence(access::fence_space accessSpace =
48      access::fence_space::global_and_local) const;
49
50    template <typename dataT>
51    device_event async_work_group_copy(local_ptr<dataT> dest,
52      global_ptr<dataT> src, size_t numElements) const;
53
54    template <typename dataT>
55    device_event async_work_group_copy(global_ptr<dataT> dest,
56      local_ptr<dataT> src, size_t numElements) const;
57
58    template <typename dataT>
59    device_event async_work_group_copy(local_ptr<dataT> dest,
60      global_ptr<dataT> src, size_t numElements, size_t srcStride) const;
61
62    template <typename dataT>
63    device_event async_work_group_copy(global_ptr<dataT> dest,
64      local_ptr<dataT> src, size_t numElements, size_t destStride) const;
65
66    template <typename... eventTN>
67    void wait_for(eventTN... events) const;
68  };
69  }  // namespace sycl
70  }  // namespace cl
```

Member function	Description
`id<dimensions> get_global_id()const`	Return the constituent global id representing the work-item's position in the global iteration space.
`size_t get_global_id(int dimension)const`	Return the constituent element of the global id representing the work-item's position in the nd-range in the given `dimension`.
`size_t get_global_linear_id()const`	Return the flattened id of the current work-item after subtracting the offset. Calculating a linear id from a multi-dimensional index follows the equation 4.3.
`id<dimensions> get_local_id()const`	Return the constituent local id representing the work-item's position within the current work-group.
`size_t get_local_id(int dimension)const`	Return the constituent element of the local id representing the work-item's position within the current work-group in the given `dimension`.

Continued on next page

Table 4.71: Member functions for the `nd_item` class.

Member function	Description
`size_t get_local_linear_id()const`	Return the flattened id of the current work-item within the current work-group. Calculating a linear address from a multi-dimensional index follows the equation 4.3.
`group<dimensions> get_group()const`	Return the constituent work-group, `group` representing the work-group's position within the overall nd-range.
`size_t get_group(int dimension)const`	Return the constituent element of the group `id` representing the work-group's position within the overall `nd_range` in the given `dimension`.
`size_t get_group_linear_id()const`	Return the group id as a linear index value. Calculating a linear address from a multi-dimensional index follows the equation 4.3.
`range<dimensions> get_group_range()const`	Returns the number of work-groups in the iteration space.
`size_t get_group_range(int dimension)const`	Return the number of work-groups for `dimension` in the iteration space.
`range<dimensions> get_global_range()const`	Returns a `range` representing the dimensions of the global iteration space.
`size_t get_global_range(int dimension)const`	Return the same value as get_global_range().get(dimension)
`range<dimensions> get_local_range()const`	Returns a `range` representing the dimensions of the current work-group.
`size_t get_local_range(int dimension)const`	Return the same value as get_local_range().get(dimension)
`id<dimensions> get_offset()const`	Returns an id representing the n-dimensional offset provided to the constructor of the `nd_range` and that is added by the runtime to the global id of each work-item.
`nd_range<dimensions> get_nd_range()const`	Returns the `nd_range` of the current execution.

`void barrier(`
`  access::fence_space accessSpace =`
`  access::fence_space::global_and_local)const`
	Executes a work-group barrier with memory ordering on the local address space, global address space or both based on the value of `accessSpace`. The current work-item will wait at the barrier until all work-items in the current work-group have reached the barrier. In addition the barrier performs a fence operation ensuring that all memory accesses in the specified address space issued before the barrier complete before those issued after the barrier.

Continued on next page

Table 4.71: Member functions for the `nd_item` class.

Member function	Description
`template <access::mode accessMode = access::mode::read_write> void mem_fence(access::fence_space accessSpace = access::fence_space::global_and_local)const`	Available only when: `accessMode == access::mode::read_write || accessMode == access::mode::read || accessMode == access::mode::write`. Executes a work-group mem-fence with memory ordering on the local address space, global address space or both based on the value of `accessSpace`. If `accessMode == access::mode::read_write` the current work-item will ensure that all load and store memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem-fence. If `accessMode == access::mode::read` the current work-item will ensure that all load memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem-fence. If `accessMode == access::mode::write` the current work-item will ensure that all store memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem- fence.
`template <typename dataT> device_event async_work_group_copy( local_ptr<dataT> dest, 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( global_ptr<dataT> dest, 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( local_ptr<dataT> dest, 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.

Continued on next page

Table 4.71: Member functions for the `nd_item` class.

Member function	Description
`template <typename dataT> device_event async_work_group_copy( global_ptr<dataT> dest, 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.
	End of table

Table 4.71: Member functions for the `nd_item` class.

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 flexible 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 4.3.3).

A synopsis of the SYCL `h_item` class is provided below. The member functions of the SYCL `h_item` class are listed in Table 4.72. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions>
4  struct h_item {
5    h_item() = delete;
6
7    /* -- common interface members -- */
8
9    item<dimensions, false> get_global() const;
10
11   item<dimensions, false> get_local() const;
12
13   item<dimensions, false> get_logical_local() const;
14
15   item<dimensions, false> get_physical_local() const;
16
17   range<dimensions> get_global_range() const;
18
19   size_t get_global_range(int dimension) const;
```

```
20
21    id<dimensions> get_global_id() const;
22
23    size_t get_global_id(int dimension) const;
24
25    range<dimensions> get_local_range() const;
26
27    size_t get_local_range(int dimension) const;
28
29    id<dimensions> get_local_id() const;
30
31    size_t get_local_id(int dimension) const;
32
33    range<dimensions> get_logical_local_range() const;
34
35    size_t get_logical_local_range(int dimension) const;
36
37    id<dimensions> get_logical_local_id() const;
38
39    size_t get_logical_local_id(int dimension) const;
40
41    range<dimensions> get_physical_local_range() const;
42
43    size_t get_physical_local_range(int dimension) const;
44
45    id<dimensions> get_physical_local_id() const;
46
47    size_t get_physical_local_id(int dimension) const;
48
49  };
50  }  // namespace sycl
51  }  // namespace cl
```

Member function	Description
`item<dimensions, false> get_global()const`	Return the constituent global item representing the work-item's position in the global iteration space as provided upon kernel invocation.
`item<dimensions, false> get_local()const`	Return the same value as `get_logical_local()`.

Continued on next page

Table 4.72: Member functions for the `h_item` class.

Member function	Description
item<dimensions, false> get_logical_local()const	Return the constituent element of the logical local item work-item's position in the local iteration space as provided upon the invocation of the group::parallel_for_work_item. If the group::parallel_for_work_item was called without any flexible local range then the member function returns the physical local item. A physical id can be computed from a logical id by getting the reminder of the integer division of the logical id and the physical range: get_logical_local().get()% get_physical_local.get_range()== get_physical_local().get().
item<dimensions, false> get_physical_local()const	Return the constituent element of the physical local item work-item's position in the local iteration space as provided (by the user or the runtime) upon the kernel invocation.
range<dimensions> get_global_range()const	Return the same value as get_global().get_range()
size_t get_global_range(int dimension)const	Return the same value as get_global().get_range(dimension)
id<dimensions> get_global_id()const	Return the same value as get_global().get_id()
size_t get_global_id(int dimension)const	Return the same value as get_global().get_id(dimension)
range<dimensions> get_local_range()const	Return the same value as get_local().get_range()
size_t get_local_range(int dimension)const	Return the same value as get_local().get_range(dimension)
id<dimensions> get_local_id()const	Return the same value as get_local().get_id()
size_t get_local_id(int dimension)const	Return the same value as get_local().get_id(dimension)
range<dimensions> get_logical_local_range()const	Return the same value as get_logical_local().get_range()
size_t get_logical_local_range(int dimension)const	Return the same value as get_logical_local().get_range(dimension)
id<dimensions> get_logical_local_id()const	Return the same value as get_logical_local().get_id()
size_t get_logical_local_id(int dimension)const	Return the same value as get_logical_local().get_id(dimension)
range<dimensions> get_physical_local_range()const	Return the same value as get_physical_local().get_range()

Continued on next page

Table 4.72: Member functions for the h_item class.

Member function	Description
`size_t get_physical_local_range(int dimension)const`	Return the same value as `get_physical_local().get_range(dimension)`
`id<dimensions> get_physical_local_id()const`	Return the same value as `get_physical_local().get_id()`
`size_t get_physical_local_id(int dimension)const`	Return the same value as `get_physical_local().get_id(dimension)`
	End of table

Table 4.72: Member functions for the `h_item` class.

group class

The `group<int dimensions>` encapsulates all functionality required to represent a particular work-group within a parallel execution. It is not user-constructable.

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 concurrent work-items are active 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 4.3.3).

A synopsis of the SYCL `group` class is provided below. The member functions of the SYCL `group` class are listed in Table 4.73. The additional common special member functions and common member functions are listed in 4.3.3 in Tables 4.3 and 4.4 respectively.

The `group` class also provides the `mem_fence` member function for performing a work-group mem-fence operation.

```
1  namespace cl {
2  namespace sycl {
3  template <int dimensions = 1>
4  struct group {
5
6    /* -- common interface members -- */
7
8   id<dimensions> get_id() const;
9
10  size_t get_id(int dimension) const;
11
12  range<dimensions> get_global_range() const;
13
14  size_t get_global_range(int dimension) const;
15
16  range<dimensions> get_local_range() const;
17
18  size_t get_local_range(int dimension) const;
19
20  range<dimensions> get_group_range() const;
```

```
21
22    size_t get_group_range(int dimension) const;
23
24    size_t operator[](int dimension) const;
25
26    size_t get_linear_id() const;
27
28    template<typename workItemFunctionT>
29    void parallel_for_work_item(workItemFunctionT func) const;
30
31    template<typename workItemFunctionT>
32    void parallel_for_work_item(range<dimensions> flexibleRange,
33      workItemFunctionT func) const;
34
35    template <access::mode accessMode = access::mode::read_write>
36    void mem_fence(access::fence_space accessSpace =
37      access::fence_space::global_and_local) const;
38
39    template <typename dataT>
40    device_event async_work_group_copy(local_ptr<dataT> dest,
41      global_ptr<dataT> src, size_t numElements) const;
42
43    template <typename dataT>
44    device_event async_work_group_copy(global_ptr<dataT> dest,
45      local_ptr<dataT> src, size_t numElements) const;
46
47    template <typename dataT>
48    device_event async_work_group_copy(local_ptr<dataT> dest,
49      global_ptr<dataT> src, size_t numElements, size_t srcStride) const;
50
51    template <typename dataT>
52    device_event async_work_group_copy(global_ptr<dataT> dest,
53      local_ptr<dataT> src, size_t numElements, size_t destStride) const;
54
55    template <typename... eventTN>
56    void wait_for(eventTN... events) const;
57  };
58  }  // sycl
59  }  // cl
```

Member function	Description
`id<dimensions> get_id()const`	Return an id representing the index of the work-group within the nd-range for every dimension.
`size_t get_id(int dimension)const`	Return the index of the work-group in the given `dimension`.
`range<dimensions> get_global_range()const`	Return a SYCL `range` representing all dimensions of the global range.
`size_t get_global_range(int dimension)const`	Return the dimension of the global range specified by the `dimension` parameter.
`range<dimensions> get_local_range()const`	Return a SYCL `range` representing all dimensions of the local range.

Continued on next page

Table 4.73: Member functions for the `group` class.

Member function	Description
`size_t get_local_range(int dimension)const`	Return the dimension of the local range specified by the `dimension` parameter.
`range<dimensions> get_group_range()const`	Return a `range` representing the dimensions of the current group. This local range may have been provided by the programmer, or chosen by the SYCL runtime.
`size_t get_group_range(int dimension)const`	Return element `dimension` from the constituent group range.
`size_t operator[](int dimension)const`	Return the index of the group in the given `dimension` within the `nd_range`.
`size_t get_linear_id()const`	Get a linearized version of the work-group id. Calculating a linear work-group id from a multi-dimensional index follows the equation 4.3.
`template <typename workItemFunctionT> void parallel_for_work_item(workItemFunctionT func)const`	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 specifiies the size of a work-group.

Continued on next page

Table 4.73: Member functions for the `group` class.

Member function	Description
`template <typename workItemFunctionT> void parallel_for_work_item(range<dimensions> flexibleRange, workItemFunctionT func)const`	Launch the work-items for this work-group using a logical local range. The function object `func` is executed as if the kernel where invoked with `flexibleRange` as the local range. This new local range is emulated and may not map one-to-one with the physical range. `flexibleRange` is the new local range to be used. This range can be smaller or larger than the one used to invoke the kernel. `func` is a function object type with a public member function `void F::operator()(h_item<dimensions>)` representing the work-item computation. Note that the flexible range does not need to be uniform across all work-groups in a kernel. For example the flexible range may depend on a work-group varying query (e.g. `group::get_linear_id`), such that different work-groups in the same kernel invocation execute different flexible range sizes. This member function can only be invoked within a `parallel_for_work_group` context.
`template <access::mode accessMode = access::mode::read_write> void mem_fence(access::fence_space accessSpace = access::fence_space::global_and_local)const`	Available only when: `accessMode == access::mode::read_write || accessMode == access::mode::read || accessMode == access::mode::write`. Executes a work-group mem-fence with memory ordering on the local address space, global address space or both based on the value of `accessSpace`. If `accessMode == access::mode::read_write` the current work-item will ensure that all load and store memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem-fence. If `accessMode == access::mode::read` the current work-item will ensure that all load memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem-fence. If `accessMode == access::mode::write` the current work-item will ensure that all store memory accesses in the specified address space issued before the mem-fence complete before those issued after the mem- fence.

Continued on next page

Table 4.73: Member functions for the `group` class.

Member function	Description
`template <typename dataT> device_event async_work_group_copy( local_ptr<dataT> dest, 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( global_ptr<dataT> dest, 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( local_ptr<dataT> dest, 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( global_ptr<dataT> dest, 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.
	End of table

Table 4.73: Member functions for the `group` class.

`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. A SYCL device event may be an OpenCL device event, in which case it must encapsulate a valid underlying OpenCL `event_t`, or it may be a SYCL host device event, in which case it must not.

All member functions of the `device_event` class must not throw a SYCL exception.

Device event interface

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 4.75 and 4.74 respectively.

```
namespace cl {
namespace sycl {
class device_event {

  device_event(__unspecified__);

 public:
  void wait();
};
}  // namespace sycl
}  // namespace cl
```

Member function	Description
`void wait()`	Waits for the asynchronous operation associated with this SYCL `device_event` to complete.
	End of table

Table 4.74: Member functions of the SYCL `device_event` class.

Constructor	Description
`device_event(___unspecified___)`	Unspecified implementation defined constructor.
	End of table

Table 4.75: Constructors of the `device_event` class.

Command group scope

A command group scope in SYCL, as it is defined in Section 3.4.1, consists of a single kernel or explicit memory operation (`handler` methods such as `copy`, `update_host`, `fill`), together with its **requirements**. The commands that enqueue a kernel or explicit memory operation and the requirements for its execution 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 methods 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 programming models that can be implemented on top of SYCL.

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 the data management section (4.7), where buffers (4.7.2) and accessor (4.7.6) classes are described. The overall memory model of the SYCL application is described in Section 3.5.1.

It is possible to obtain events for the start of the command group function object, the kernel starting, and the command group completing. These events are most useful for profiling, because safe synchronization in SYCL requires synchronization on buffer availability, not on kernel completion. This is because the memory that data is stored in upon kernel completion is not rigidly specified. The events are provided at the submission of the command group function object to the queue to be executed on.

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.

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 4.76.

It is disallowed for an instance of the SYCL `handler` class to be moved or copied.

```
1  namespace cl {
2  namespace sycl {
3
4  class handler {
5   private:
6
7    // implementation defined constructor
8    handler(___unspecified___);
9
10  public:
11
12   template <typename dataT, int dimensions, access::mode accessMode,
13     access::target accessTarget>
14   void require(accessor<dataT, dimensions, accessMode, accessTarget,
15                access::placeholder::true_t> acc);
16
```

```
17    //----- OpenCL interoperability interface
18    //
19    template <typename T>
20    void set_arg(int argIndex, T && arg);
21
22    template <typename... Ts>
23    void set_args(Ts &&... args);
24
25    //------ Kernel dispatch API
26    //
27    // Note: In all Kernel dispatch functions,
28    // when using a functor with a globally visible name
29    // the template parameter:"typename kernelName" can be ommitted
30    // and the kernelType can be used instead.
31    //
32    template <typename KernelName, typename KernelType>
33    void single_task(KernelType kernelFunc);
34
35    template <typename KernelName, typename KernelType, int dimensions>
36    void parallel_for(range<dimensions> numWorkItems, KernelType kernelFunc);
37
38    template <typename KernelName, typename KernelType, int dimensions>
39    void parallel_for(range<dimensions> numWorkItems,
40                      id<dimensions> workItemOffset, KernelType kernelFunc);
41
42    template <typename KernelName, typename KernelType, int dimensions>
43    void parallel_for(nd_range<dimensions> executionRange, KernelType kernelFunc);
44
45    template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
46    void parallel_for_work_group(range<dimensions> numWorkGroups,
47                                 WorkgroupFunctionType kernelFunc);
48
49    template <typename KernelName, typename WorkgroupFunctionType, int dimensions>
50    void parallel_for_work_group(range<dimensions> numWorkGroups,
51                                 range<dimensions> workGroupSize,
52                                 WorkgroupFunctionType kernelFunc);
53
54    void single_task(kernel syclKernel);
55
56    template <int dimensions>
57    void parallel_for(range<dimensions> numWorkItems, kernel syclKernel);
58
59    template <int dimensions>
60    void parallel_for(range<dimensions> numWorkItems,
61                      id<dimensions> workItemOffset, kernel syclKernel);
62
63    template <int dimensions>
64    void parallel_for(nd_range<dimensions> ndRange, kernel syclKernel);
65
66    //------ Explicit memory operation APIs
67    //
68    template <typename T, int dim, access::mode mode, access::target tgt>
69    void copy(accessor<T, dim, mode, tgt> src, shared_ptr_class<T> dest);
70
71    template <typename T, int dim, access::mode mode, access::target tgt>
```

```
72    void copy(shared_ptr_class<T> src, accessor<T, dim, mode, tgt> dest);
73
74    template <typename T, int dim, access::mode mode, access::target tgt>
75    void copy(accessor<T, dim, mode, tgt> src, T * dest);
76
77    template <typename T, int dim, access::mode mode, access::target tgt>
78    void copy(const T * src, accessor<T, dim, mode, tgt> dest);
79
80    template <typename T, int dim, access::mode mode, access::target tgt>
81    void copy(accessor<T, dim, mode, tgt> src, accessor<T, dim, mode, tgt> dest);
82
83    template <typename T, int dim, access::mode mode, access::target tgt>
84    void update_host(accessor<T, dim, mode, tgt> acc);
85
86    template<typename T, int dim, access::mode mode, access::target tgt>
87    void fill(accessor<T, dim, mode, tgt> dest, const T& src);
88
89  };
90  }  // namespace sycl
91  }  // namespace cl
```

Constructor	Description
`handler(___unspecified___)`	Unspecified implementation defined constructor.
	End of table

Table 4.76: Constructors of the `handler` class.

SYCL functions for adding requirements

Requirements for execution of SYCL kernels can be specified directly using handler methods.

Member function	Description
`template <typename dataT, int dimensions, access::mode accessMode, access::target accessTarget >` `void require(accessor<dataT, dimensions, accessMode, accessTarget, placeholder::true_t> acc)`	Requires access to the memory object associated with the placeholder accessor. The command group now has a **requirement** to gain access to the given memory object before executing the kernel.
	End of table

Table 4.77: Member functions of the `handler` class.

SYCL functions for invoking kernels

Kernels can be invoked as *single tasks*, basic *data-parallel kernels*, OpenCL-style *nd-range* in *work-groups*, or SYCL *hierarchical parallelism*.

Each function takes a kernel name template parameter. The kernel name must be a datatype that is unique for each kernel invocation. If a kernel is a named function object, and its type is globally visible, then the kernel's function object type will be automatically used as the kernel name and so the user does not need to supply a name. If the kernel function is a C++11 lambda function, then the user must manually provide a kernel name to enable linking between host and device code to occur.

All the functions for invoking kernels are member functions of the command group `handler` class 4.8.3, which is used to encapsulate all the member functions provided in a command group scope. Table 4.78 lists all the members of the `handler` class related to the kernel invocation.

Member function	Description
`template <typename T>` `void set_arg(int argIndex, T &&arg)`	Set a kernel argument for an OpenCL kernel through the SYCL/OpenCL interoperability interface. The `index` value specifies which parameter of the OpenCL kernel is being set and `arg` specifies the kernel argument. Index 0 is the first parameter. The argument can be either a SYCL accessor, a SYCL sampler or a trivially copyable and standard-layout C++ type.
`template <typename... Ts>` `void set_args(Ts &&... args)`	Set all the given kernel `args` arguments for an OpenCL kernel, as if `set_arg()` was used with each of them in the same order and increasing index always starting at 0.
`template <typename KernelName, typename KernelType>` `void single_task(KernelType kernelFunc)`	Defines and invokes a SYCL kernel function as a lambda function or a named function object type. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.8.5.

`template <typename KernelName, typename KernelType, int dimensions>`
`void parallel_for(`
`range<dimensions> numWorkItems, KernelType kernelFunc)`
	Defines and invokes a SYCL kernel function as a lambda function or a named function object type, for the specified range and given an id or item for indexing in the indexing space defined by range. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.8.5.

Continued on next page

Table 4.78: Member functions of the `handler` class.

Member function	Description
`template <typename KernelName, typename KernelType, int dimensions> void parallel_for( range<dimensions> numWorkItems, id<dimensions> workItemOffset, KernelType kernelFunc)`	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 id or item for indexing in the indexing space defined by range. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.8.5.
`template <typename KernelName, typename KernelType, int dimensions> void parallel_for( nd_range<dimensions> executionRange, KernelType kernelFunc)`	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. If it is a named function object and the function object type is globally visible there is no need for the developer to provide a kernel name (`typename KernelName`) for it, as described in 4.8.5.
`template <typename KernelName, typename WorkgroupFunctionType, int dimensions> void parallel_for_work_group( range<dimensions> numWorkGroups, WorkgroupFunctionType kernelFunc)`	Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. May contain multiple calls to `parallel_for_work_item(..)` methods representing the execution on each work-item. Launches `num_work_groups` work-groups of runtime-defined size. Described in detail in 4.8.5.
`template <typename KernelName, typename WorkgroupFunctionType, int dimensions> void parallel_for_work_group( range<dimensions> numWorkGroups, range<dimensions> workGroupSize, WorkgroupFunctionType kernelFunc)`	Hierarchical kernel invocation method of a kernel defined as a lambda encoding the body of each work-group to launch. May contain multiple calls to `parallel_for_work_item` methods representing the execution on each work-item. Launches `num_work_groups` work-groups of `work_roup_size` work-items each. Described in detail in 4.8.5.
`void single_task(kernel syclKernel)`	Defines and invokes a SYCL kernel function as a lambda function or a named function object type, executes exactly once.
`template <int dimensions> void parallel_for( range<dimensions> numWorkItems, kernel syclKernel)`	Kernel invocation method of a pre-compiled kernel defined by SYCL `sycl-kernel-function` instance, for the specified range and given an id or item for indexing in the indexing space defined by range, described in detail in 4.8.5

Continued on next page

Table 4.78: Member functions of the `handler` class.

Member function	Description
`template <int dimensions> void parallel_for( range<dimensions> numWorkItems, id<dimensions> workItemOffset, kernel syclKernel )`	Kernel invocation method of a pre-compiled kernel defined by SYCL `sycl-kernel-function` instance, for the specified range and offset and given an id or item for indexing in the indexing space defined by range, described in detail in 4.8.5
`template <int dimensions> void parallel_for( nd_range<dimensions> ndRange, kernel syclKernel)`	Kernel invocation method of a pre-compiled kernel defined by SYCL `kernel` instance, for the specified `ndrange` and given an `nd_item` for indexing in the indexing space defined by the `nd_range`, described in detail in 4.8.5
	End of table

Table 4.78: Member functions of the `handler` class.

`single_task` invoke

SYCL provides a simple interface to enqueue a kernel that will be sequentially executed on an OpenCL 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 an OpenCL 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.

```
1  myQueue.submit([&](handler & cgh) {
2  cgh.single_task<class kernel_name>(
3      [=] () {
4        // [kernel code]
5      }));
6  });
```

For single tasks, the kernel method takes no parameters, as there is no need for index space classes in a unary index space.

`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, and the SYCL runtime will select a local range (the number of work-items in each

work-group). The local range chosen by the SYCL runtime is entirely implementation defined. In this case the function object that represents the SYCL kernel function must take either a single SYCL `id` parameter, or a single SYCL `item` parameter, representing the currently executing work-item within the range specified by the `range` parameter.

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, or the local (work-group) size chosen by the implementation.

```
1  myQueue.submit([&](handler & cgh) {
2      auto acc = myBuffer.get_access<access::mode::write>(cgh);
3
4      cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
5                                       [=] (id<1> index) {
6          acc[index] = 42.0f;
7      });
8  });
```

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 within which the global id will vary. No information is queryable on the local (work-group) size chosen by the implementation.

```
1  myQueue.submit([&](handler & cgh) {
2      auto acc = myBuffer.get_access<access::mode::write>(cgh);
3
4      cgh.parallel_for<class myKernel>(range<1>(numWorkItems),
5                                       [=] (item<1> item) {
6          size_t index = item.get_linear_id();
7          acc[index] = 42.0f;
8      });
9  });
```

For SYCL kernel functions invoked via the above described overload of the `parallel_for` member function, it is disallowed to use local accessors or to use a work-group barrier or work-group mem-fence operation.

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.

```
1  myQueue.submit([&](handler & cgh) {
2   cgh.parallel_for<class example_kernel1>(
3     range<3>(3,3,3), // global range
4       [=] (item<3> it) {
5         //[kernel code]
6       });
7  });
```

```
8  myQueue.submit([&](handler & cgh) {
9   cgh.parallel_for<class example_kernel2>(
10     range<3>(3,3,3), // global range
11     id<3>(1,1,1), // offset
12       [=] (item<3> it) {
13         //[kernel code]
14       });
15 });
```

The last case of a 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 `parallel_for (nd_range,...)` and the `nd_item` class, which provides all the functionality of OpenCL for an nd-range. In this case, the developer needs to define the `nd_range` that the kernel will execute on in order to have fine grained control of the enqueing of the kernel. This variation of parallel_for expects an `nd_range`, specifying both local and global ranges, defining the global number of work-items and the number in each cooperating work-group. The resulting function object is passed an `nd_item` instance making all the information available, as well as work-group barrier and work-group mem-fence operations to synchronize or guarantee memory consistency between the work-items in the work-group.

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 work-group barrier.

```
1  myQueue.submit([&](handler & cgh) {
2    cgh.parallel_for<class example_kernel>(
3        nd_range<3>(range<3>(4, 4, 4), range<3>(2, 2, 2)), [=](nd_item<3> item) {
4          //[kernel code]
5          // Internal synchronization
6          item.barrier(access::fence_space::global_space);
7          //[kernel code]
8        });
9  });
```

Optionally, in any of these variations of parallel_for invocations, the developer may also pass an offset. An offset is an instance of the `id` class added to the identifier for each point in the range.

In all of these cases the underlying *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.

Parallel For hierarchical invoke

The hierarchical parallel kernel execution interface provides the same functionality as is available from the 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 `parallel_for_work_group` call to create the groups. The member function `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 `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 very 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:

```
1  namespace cl {
2  namespace sycl {
3  template <typename T, int Dimensions = 1>
4  class private_memory {
5   public:
6    // Construct based directly off the number of work-items
7    private_memory(const group<Dimensions> &);
8
9    // Access the instance for the current work-item
10   T &operator()(const h_item<Dimensions> &id);
11 };
12 }
13 }
```

Constructor	Description
`private_memory(const group<Dimensions> &)`	Place an object of type `T` in the underlying private memory of each work-items. The type `T` must be default constructible. The underlying constructor will be called for each work-item.
	End of table

Table 4.79: Constructor of the `private_memory` class.

Member functions	Description
`T &operator()(const h_item<Dimensions> &id)`	Retrieve a reference to the object for the work-items.
	End of table

Table 4.80: Member functions of the `private_memory` class.

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) {
  // Issue 8 work-groups of 8 work-items each
  cgh.parallel_for_work_group<class example_kernel>(
      range<3>(2, 2, 2), range<3>(2, 2, 2), [=](group<3> myGroup) {

    //[workgroup code]
    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) {
  // Issue 8 work-groups.  The work-group size is chosen by the runtime because unspecified
  cgh.parallel_for_work_group<class example_kernel>(
      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) {
```

```
10        //[work-item code]
11      });
12
13      // Implicit work-group barrier
14
15      // Launch 512 logical work-items that will be executed by the underlying work-group size
16      // chosen by the runtime.  myItem allows the logical and physical work-item IDs to be
17      // queried.  512 logical work-items will execute for each work-group, and the parallel_for
18      // body will therefore be executed 8*512 = 4096 times globally/total.
19      myGroup.parallel_for_work_item(range<3>(8, 8, 8), [=](h_item<3> myItem) {
20        //[work-item code]
21      });
22      //[workgroup code]
23    });
24  });
```

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.

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 the OpenCL host API (e.g, enqueue copy operations).

The SYCL memory objects involved in a copy operation are specified using accessors. 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 can have either **read** or **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 can have either **write**, **read_write**, **discard_write**, **discard_read_write** access modes.

When accessors are both the origin and the destination, the operation is executed on objects controlled by the SYCL runtime. The SYCL runtime is allowed to not perfom 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 host 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` method. This method 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 4.81 describes the interface for the explicit copy operations.

Member function	Description
`template <typename T, int dim, access::mode mode, access::target tgt> void copy(accessor<T, dim, mode, tgt> src, shared_ptr_class<T> dest)`	Copies the contents of the memory pointed to by src into the memory object accessed by dest. src must have at least as many bytes as the range accessed by dest.
`template <typename T, int dim, access::mode mode, access::target tgt> void copy(shared_ptr_class<T> src accessor<T, dim, mode, tgt> dest)`	Copies the contents of the memory object accessed via src into the memory pointed to by dest. dest must have at least as many bytes as the range accessed by src.
`template <typename T, int dim, access::mode mode, access::target tgt> void copy(accessor<T, dim, mode, tgt> src, T * dest)`	Copies the contents of the memory pointed to by src into the memory object accessed by dest. src must have at least as many bytes as the range accessed by dest.
`template <typename T, int dim, access::mode mode, access::target tgt> void copy(const T * src accessor<T, dim, mode, tgt> dest)`	Copies the contents of the memory object accessed via src into the memory pointed to by dest. dest must have at least as many bytes as the range accessed by src.
`template <typename T, int dim, access::mode mode, access::target tgt> void copy(accessor<T, dim, mode, tgt> src accessor<T, dim, mode, tgt> dest)`	Copies the contents of the memory object accessed by src into the memory object accessed by dest. src must have at least as many bytes as the range accessed by dest.
`template <typename T, int dim, access::mode mode, access::target tgt> void update_host(accessor<T, dim, mode, tgt> acc )`	The contents of the memory object accessed via acc on the host are guaranteed to be up-to-date after this command group object execution is complete.
`template <typename T, int dim, access::mode mode, access::target tgt> void fill(accessor<T, dim, mode, tgt> dest, const T& src)`	Replicates the value of src into the memory object accessed by dest. T must be an integral scalar value or a SYCL vector type.
	End of table

Table 4.81: Member functions of the handler class.

The listing below illustrates how to use explicit copy operations in SYCL. The example copies half of the contents of a vector_class into the device, leaving the rest of the contents of the buffer on the device unchanged.

```
1  const size_t nElems = 10u;
2
3  // Create a vector and fill it with values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
4  std::vector<int> v(nElems);
5  std::iota(std::begin(v), std::end(v), 0);
6
7  // Create a buffer with no associated user storage
8  cl::sycl::buffer<int, 1> b{range<1>(nElems)};
9
10 // 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<int, 1, access::mode::write, access::target::global_buffer>
      acc(b, range<1>(nElems / 2), id<1>(0));
  // Copy the first five elements of the vector into the buffer associated with
  // the accessor
  cgh.copy(v.data(), acc);
});
```

Kernel class

The kernel class is an abstraction of a kernel object in SYCL. In the most common case the kernel object will contain the compiled version of a kernel invoked inside a command group using one of the parallel interface functions as described in 4.8.5. The *SYCL runtime* will create a kernel object, when it needs to enqueue the kernel on a command queue.

In the case where a developer would like to pre-compile a kernel or compile and link it with an existing program, then the kernel object will be created and contain that kernel using the program class, as defined in 4.8.8. In both of the above cases, the developer cannot instantiate a kernel object but can instantiate a named function object type that they could use, or create a function object from a kernel method using C++11 features. The kernel class object needs a `parallel_for(...)` invocation or an explicitly built SYCL `kernel` instance, for this compilation of the kernel to be triggered.

The SYCL `kernel` class provides the common reference semantics (see Section 4.3.2).

The kernel class also provides the interface for getting information from a kernel object. The kernel information descriptor interface is described in A.5 and the description is in the Table 4.84.

The constructors and member functions of the SYCL `kernel` class are listed in Tables 4.82 and 4.83, respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

```
namespace cl {
namespace sycl {
class kernel {
 private:
  friend class program;

  // The default object is not valid because there is no
  // program or cl_kernel associated with it
  kernel();

 public:

  kernel(cl_kernel clKernel, const context& syclContext);

  /* -- common interface members -- */

```

```
17    cl_kernel get() const;
18
19    bool is_host() const;
20
21    context get_context() const;
22
23    program get_program() const;
24
25    template <info::kernel param>
26    typename info::param_traits<info::kernel, param>::return_type
27    get_info() const;
28
29    template <info::kernel_work_group param>
30    typename info::param_traits<info::kernel_work_group, param>::return_type
31    get_work_group_info(const device &dev) const;
32  };
33  }  // namespace sycl
34  }  // namespace cl
```

Constructor	Description
`kernel (cl_kernel clKernel, const context& syclContext)`	Constructs a SYCL `kernel` instance from an OpenCL `cl_kernel` in accordance with the requirements described in 4.3.1. The SYCL `context` must represent the same underlying OpenCL context associated with the OpenCL kernel object.

End of table

Table 4.82: Constructors of the SYCL `kernel` class.

Member functions	Description
`cl_kernel get()const`	Returns a valid `cl_kernel` instance in accordance with the requirements described in 4.3.1.
`bool is_host()const`	Returns true if this SYCL `kernel` is a host kernel.
`context get_context()const`	Return the context that this kernel is defined for. The value returned must be equal to that returned by `get_info<info::kernel ::context>()`.
`program get_program()const`	Return the program that this kernel is part of. The value returned must be equal to that returned by `get_info<info::kernel:: program>()`.
`template <info::kernel param> typename info::param_traits< info::kernel, param>::return_type get_info()const`	Query information from the kernel object using the `info::kernel_info` descriptor.

Continued on next page

Table 4.83: Member functions of the `kernel` class.

Member functions	Description
`template <info::kernel_work_group param> typename info::param_traits< info::kernel_work_group, param>::return_type get_work_group_info(const device &dev)const`	Query information from the work-group from a kernel using the `info::kernel_work_group` descriptor for a specific device
	End of table

Table 4.83: Member functions of the `kernel` class.

Kernel Descriptors	Return type	Description
`info::kernel::function_name`	`string_class`	Return the kernel function name.
`info::kernel::num_args`	`cl_uint`	Return the number of arguments to the extracted OpenCL C kernel.
`info::kernel::context`	`context`	Return the SYCL `context` associated with this SYCL `kernel`.
`info::kernel::program`	`program`	Return the SYCL `program` associated with this SYCL `kernel`.
`info::kernel::reference_count`	`cl_uint`	Returns the reference count of the encapsulated SYCL `cl_kernel`, if this SYCL `kernel` is an OpenCL program. Must thrown a `invalid_object_error` SYCL exception if this SYCL `kernel` is a host kernel.
`info::kernel::attributes`	`string_class`	Return any attributes specified using the `__attribute__` qualifier with the kernel function declaration in the program source.
		End of table

Table 4.84: Kernel class information descriptors.

Kernel Work-group Information Descriptors	Return type	Description
`info::kernel_work_group:: global_work_size`	`range<3>`	Returns the maximum global work size. Only valid if device is of device_type custom or the kernel is a built-in OpenCL kernel.
`info::kernel_work_group:: work_group_size`	`size_t`	Returns the maximum work-group size that can be used to execute a kernel on a specific device.
`info::kernel_work_group:: compile_work_group_size`	`range<3>`	Returns the work-group size specified by the device compiler if applicable, otherwise returns $(0, 0, 0)$
`info::kernel_work_group:: preferred_work_group_size_multiple`	`size_t`	Returns the preferred work-group size for executing a kernel on a particular device.
		Continued on next page

Table 4.85: Kernel work-group information descriptors.

Kernel Work-group Information Descriptors	Return type	Description
`info::kernel_work_group::private_mem_size`	`cl_ulong`	Returns the minimum amount of private memory, in bytes, used by each work-item in the kernel. This value may include any private memory needed by an implementation to execute the kernel, including that used by the language built-ins and variables declared inside the kernel in the private address space.
		End of table

Table 4.85: Kernel work-group information descriptors.

Program class

The SYCL `program` class encapsulates a single SYCL program. A SYCL program may be an OpenCL program, in which case it must encapsulate a valid underlying OpenCL `cl_program`, depending on it's state, or it may be a SYCL host program, in which case it must not.

A SYCL `program` can be used to compile and link both SYCL programs and OpenCL programs.

A SYCL `program` instance can be in one of three states defined by `program_state`:

- A SYCL `program` in the `program_state::none` state must have no encapsulated `cl_program`.
- A SYCL `program` in the `program_state::compiled` state must encapsulate a `cl_program` that has been compiled but not yet linked, if that SYCL `program` is an OpenCL program. It must have no encapsulated `cl_program` if that SYCL `program` is a host program.
- A SYCL `program` in the `program_state::linked` state must encapsulate a `cl_program` that has been either compiled and linked or built, if that SYCL `program` is an OpenCL program. It must have no encapsulated `cl_program` if that SYCL `program` is a host program.

A SYCL `program` host program must follow the same state changes as an OpenCL program, however the transitions are implementation defined.

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

There is no default constructor for the SYCL `program` as all constructors require a SYCL `context` instance to be provided. The only exception is a constructor taking a `vector_class` containing SYCL `program` instances. This constructor links them together into a new SYCL `program`.

The encapsulated `cl_program` of an OpenCL program can contain either SYCL kernel functions or OpenCL C kernel functions. When a `program` instance is constructed using a non-OpenCL interoperability constructor, it is in the `program_state::none` state and should then be compiled or built by specifying the SYCL kernel name (either the type of the function object or the explicit kernel name type specified when defining the SYCL kernel function). When a `program` instance is constructed using an OpenCL interoperability constructor, it can be in either the `program_state::compiled` or `program_state::linked` state and should not be compiled or built, only linked.

The compiler options that can be provided are described in the OpenCL specification [1, p. 145, § 5.6.4] and the linker options that can be provided are described in [1, p. 148,§ 5.6.5].

The SYCL `program` class provides the common reference semantics (see Section 4.3.2).

Program interface

A synopsis of the SYCL `program` class is provided below. The constructors and member functions of the SYCL `program` class are listed in Tables 4.86 and 4.87 respectively. The additional common special member functions and common member functions are listed in 4.3.2 in Tables 4.1 and 4.2, respectively.

```
namespace cl {
namespace sycl {
enum class program_state {
  none,
  compiled,
  linked
};

class program {
 public:
  program() = delete;

  explicit program(const context &context);

  program(const context &context, vector_class<device> deviceList);

  program(vector_class<program> programList, string_class linkOptions = "");

  program(const context &context, cl_program clProgram);

  /* -- common interface members -- */

  cl_program get() const;

  bool is_host() const;

  template <typename kernelT>
  void compile_with_kernel_type(string_class compileOptions = "");

  void compile_with_source(string_class kernelSource, string_class compileOptions = "");

  template <typename kernelT>
  void build_with_kernel_type(string_class buildOptions = "");

  void build_with_source(string_class kernelSource, string_class buildOptions = "");

  void link(string_class linkOptions = "");

  template <typename kernelT>
  bool has_kernel<kernelT>() const;

  bool has_kernel(string_class kernelName) const;
```

```
43
44    template <typename kernelT>
45    kernel get_kernel<kernelT>() const;
46
47    kernel get_kernel(string_class kernelName) const;
48
49    template <info::program param>
50    typename info::param_traits<info::program, param>::return_type
51      get_info() const;
52
53    vector_class<vector_class<char>> get_binaries() const;
54
55    context get_context() const;
56
57    vector_class<device> get_devices() const;
58
59    string_class get_compile_options() const;
60
61    string_class get_link_options() const;
62
63    string_class get_build_options() const;
64
65    program_state get_state() const;
66  };
67  }  // namespace sycl
68  }  // namespace cl
```

Constructor	Description
`program ()= delete`	Default constructor is deleted.
`explicit program ( const context &context)`	Constructs an instance of SYCL `program` in the `program_state::none` state, associated with the `context` provides and the SYCL `device`s that are associated with the `context` .
`program ( const context &context, vector_class<device> deviceList)`	Constructs an instance of SYCL `program` in the `program_state::none` state, associated with the `context` provides and `deviceList` .
`program ( vector_class<program> programList, string_class linkOptions = "")`	Constructs an instance of SYCL `program` in the `program_state::linked` by linking together each SYCL `program` instance in `programList`. Each SYCL `program` in `programList` must be in the `program_state ::compiled` state and must be associated with the same SYCL `context`. Otherwise must throw an `invalid_object_error` SYCL exception.

Continued on next page

Table 4.86: Constructors of the SYCL `program` class.

Constructor	Description
`program ( const context &context, cl_program clProgram)`	Constructs a SYCL `program` instance from an OpenCL `cl_program` in accordance with the requirements described in 4.3.1. The state of the constructed SYCL `program` can be either `program_state::compiled` or `program_state::linked`, depending on the state of the `clProgram`. Otherwise must throw an `invalid_object_error` SYCL exception.
	End of table

Table 4.86: Constructors of the SYCL `program` class.

Member functions	Description
`cl_program get()const`	Returns a valid `cl_program` instance in accordance with the requirements described in 4.3.1. Must throw an `invalid_object_error` SYCL exception if this `program` is in the `program_state::none` state.
`bool is_host()const`	Returns true if this SYCL `program` is a host program.
`template<typename kernelT> void compile_with_kernel_type( string_class compileOptions = "")`	Compiles the SYCL kernel function defined by the type `kernelT` into the encapsulated `cl_program` with the compiler options specified by `compileOptions`, if this SYCL `program` is an OpenCL program. Sets the state of this SYCL `program` to `program_state::compiled`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state ::none` state when called. Must return a `compile_program_error` SYCL exception if the compilation fails.
`void compile_with_source(string_class kernelSource, string_class compileOptions = "")`	Compiles the OpenCL C kernel function defined by `kernelSource` into the encapsulated `cl_program` with the compiler options specified by `compileOptions`, if this SYCL `program` is an OpenCL program. Sets the state of this SYCL `program` to `program_state::compiled`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state ::none` state when called. Must return a `compile_program_error` SYCL exception if the compilation fails.
	Continued on next page

Table 4.87: Member functions of the SYCL `program` class.

Member functions	Description
`template<typename kernelT> void build_with_kernel_type( string_class buildOptions = "")`	Builds the SYCL kernel function defined by the type `kernelT` into the encapsulated `cl_program` with the compiler options specified by `buildOptions`, if this SYCL `program` is an OpenCL program. Sets the state of this SYCL `program` to `program_state::linked`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state::none` state when called. Must return a `compile_program_error` SYCL exception if the compilation fails.
`void build_with_source(string_class kernelSource, string_class buildOptions = "")`	Builds the OpenCL C kernel function defined by `kernelSource` into the encapsulated `cl_program` with the compiler options specified by `buildOptions`, if this SYCL `program` is an OpenCL program. Sets the state of this SYCL `program` to `program_state::linked`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state::none` state when called. Must return a `compile_program_error` SYCL exception if the compilation fails.
`void link(string_class linkOptions = "")`	Links the encapsulated `cl_program` with the compiler options specified by `linkOptions`, if this SYCL `program` is an OpenCL program. Sets the state of this SYCL `program` to `program_state::linked`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state::compiled` state when called. Must return a `compile_program_error` SYCL exception if the linking fails.
`template <typename kernelT> bool has_kernel<kernelT>()const`	Returns true if the SYCL kernel function defined by the type `kernelT` is an available kernel, either within the the encapsulated `cl_program`, if this SYCL `program` is an OpenCL program, or on the host if this SYCL `program` is a host program, otherwise returns false. Must throw an `invalid_object_error` SYCL exception if this SYCL `program` is in the `program_state::none` state.

Continued on next page

Table 4.87: Member functions of the SYCL `program` class.

Member functions	Description
`bool has_kernel(string_class kernelName)const`	Returns true if the OpenCL C kernel function defined by the `string_class kernelName` is an available kernel within the encapsulated `cl_program` and this SYCL `program` is not a host program, otherwise returns false. Must throw an `invalid_object_error` SYCL exception if this SYCL `program` is in the `program_state ::none` state.
`template <typename kernelT> kernel get_kernel<kernelT>()const`	Returns a SYCL `kernel` OpenCL kernel instance encapsulating a `cl_kernel` for the SYCL kernel function defined by the type `kernelT`, if this SYCL `program` is an OpenCL program. Returns a SYCL `kernel` host kernel if this SYCL `program` is a host program. Must throw an `invalid_object_error` SYCL exception if this SYCL `program` is in the `program_state ::none` state or if the SYCL kernel function specified by `kernelT` is not available in this SYCL `program`.
`kernel get_kernel(string_class kernelName)const`	Returns a SYCL `kernel` OpenCL kernel instance encapsulating a `cl_kernel` for the OpenCL C kernel function defined by the `string_class kernelName`, if this SYCL `program` is an OpenCL program. Must throw an `invalid_object_error` SYCL exception if this SYCL `program` is a host program, this SYCL `program` is in the `program_state ::none` state or the `cl_program` encapsulated by this SYCL `program` does not contain the OpenCL C kernel function specified by `kernelName`. Returns a SYCL `kernel` host kernel if this SYCL `program` is a host program.
`template<info::program param> typename info::param_traits< info::program, param>::return_type get_info()const`	Queries this SYCL `program` for information requested by the template parameter `param` . Specializations of `info::param_traits` must be defined in accordance with the info parameters in Table 4.88 to facilitate returning the type associated with the `param` parameter.

Continued on next page

Table 4.87: Member functions of the SYCL `program` class.

Member functions	Description
`vector_class<vector_class<char>> get_binaries()const`	Returns a `vector_class` of `vector_class<char>` representing the compiled binaries for each associated SYCL `device`. Must throw an `invalid_object_error` SYCL exception if this `program` was not in the `program_state::compiled` or `program_state::linked` states when called.
`context get_context()const`	Returns the SYCL `context` that this SYCL `program` was constructed with. The value returned must be equal to that returned by `get_info<info::program::context>()`.
`vector_class<device> get_devices()const`	Returns a `vector_class` containing all SYCL `device`s that are associated with this SYCL `program`. The value returned must be equal to that returned by `get_info<info::program::devices>()`.
`string_class get_compile_options()const`	Returns the compile options that were provided when the encapsulated `cl_program` was explicitly compiled. If the program was built instead of explicitly compiled, if the program has not yet been compiled, or if the program has been compiled for only the host device (which does not have an underlying `cl_program`), then an empty string is returned. If the program was constructed from a `cl_program`, then an empty string is returned unless the `cl_program` was explicitly compiled, in which case the compile options used in the explicit compile are returned.
`string_class get_link_options()const`	Returns the link options that were provided to the most recent invocation of `program::link`. If the program has not been explicitly linked using `program::link`, constructed with an explicitly linking constructor, or if the program has been linked for only the host device, then an empty string is returned. If the program was constructed from a `cl_program`, then an empty string is returned unless the `cl_program` was explicitly linked, in which case the link options used in that explicit link are returned. If the program object was constructed using a constructor form that links a vector of programs (and leaves the program in `program_state::linked`), then the link options passed to this constructor are returned.

Continued on next page

Table 4.87: Member functions of the SYCL `program` class.

Member functions	Description
string_class get_build_options()const	Returns the compile, link, or build options, from whichever of those operations was performed most recently on the encapsulated cl_program. If no compile, link, or build operations have been performed on this SYCL program object, or if the program only includes the host device in its deviceList, then an empty string is returned.
program_state get_state()const	Returns the current state of this SYCL program.
	End of table

Table 4.87: Member functions of the SYCL program class.

Program information descriptors

A SYCL program can be queried for all of the following information using the get_info member function. All SYCL programs must have valid values for every query, including a host program. The information that can be queried is described in Table 4.88. The interface for all information types and enumerations are described in appendix A.6.

Program Descriptor	Return type	Description
info::program::reference_count	cl_uint	Returns the reference count of the encapsulated SYCL cl_program, if this SYCL program is an OpenCL program. Must throw an invalid_object_error SYCL exception if this SYCL program is a host program.
info::program::context	context	Returns the SYCL context associated with this program.
info::program::devices	vector_class< device>	Returns a vector_class containing the SYCL devices that this program has been compiled for.
		End of table

Table 4.88: Program class information descriptors.

Defining kernels

In SYCL functions that are executed in parallel on a SYCL device are referred to as *kernel functions*. A *kernel* containing such a *kernel function* is enqueued on a device queue in order to be executed on that particular device. The return type of the *kernel function* is void, and all kernel accesses between host and device are defined using the accessor class 4.7.6.

There are three ways of defining kernels, defining them as named function objects, lambda functions or as OpenCL cl_kernel objects. However, in the case of OpenCL kernels, the developer is expected to have created the kernel and set the kernel arguments.

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 C++11 standard layout rules. The kernel function can be templated via templating the kernel function object type. The `operator()` function may take different parameters depending on the data accesses defined for the specific kernel. For details on restrictions for kernel naming, please refer to 6.2.

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 method 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 parameters to the device kernel. Usual restrictions of passing parameters to kernels apply.

```
1  class RandomFiller {
2   public:
3    RandomFiller(accessor<int, 1, access::mode::read_write,
4                          access::target::global_buffer> ptr)
5        : ptr_ { ptr } {
6      std::random_device hwRand;
7      std::uniform_int_distribution<> r{1, 100};
8      randomNum_ = r(hwRand);
9    }
10   void operator()(item<1> item) { ptr_[item.get_id()] = get_random(); }
11   int get_random() { return randomNum_; }
12
13  private:
14   accessor<int, 1, access::mode::read_write, access::target::global_buffer>
15       ptr_;
16   int randomNum_;
17 };
18
19 void workFunction(buffer<int, 1>& b, queue& q, const range<1> r) {
20     myQueue.submit([&](handler& cgh) {
21       auto ptr = buf.get_access<access::mode::read_write>(cgh);
22       RandomFiller filler { ptr };
23
24       cgh.parallel_for(r, filler);
25     });
26 }
```

Defining kernels as lambda functions

In C++11, function objects can be defined using lambda functions. We allow lambda functions to define kernels in SYCL, but we have an extra requirement to *name lambda functions* in order to enable the linking of the SYCL device kernels with the host code to invoke them. The name of a lambda function in SYCL is a C++ class. If the lambda function relies on template arguments, then the name of the lambda function must contain those template arguments. 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 6.2.

To invoke a C++11 lambda, the kernel name must be included explicitly by the user as a template parameter to the kernel invoke function.

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. [=]).

```
class MyKernel;

myQueue.submit([&](handler& cmdGroup) {
  cmdgroup.single_task<class MyKernel>([=]() {
    // [kernel code]
  });
});
```

Defining kernels using program objects

In case the developer needs to specify compiler flags or special linkage options for a kernel, then a kernel object can be used, as described in 4.8.8. The SYCL kernel function is defined as a named function object 4.8.9.1 or lambda function 4.8.9.2. The user can obtain a program object for the kernel with the `get_kernel` method. This method is templated by the *kernel name*, so that the user can specify the kernel whose associated kernel they wish to obtain.

In the following example, the kernel is defined as a lambda function. The example obtains the program object for the lambda function kernel and then passes it to the `parallel_for`.

```
class MyKernel;  // Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;
cl::sycl::program myProgram(myQueue.get_context());

/* use the name of the kernel to obtain the associated program */
myProgram.build_from_name<MyKernel>();

myQueue.submit([&](handler& commandGroup) {
  commandgroup.parallel_for<class MyKernel>(
      cl::sycl::nd_range<2>(range<2>(4, 4),range<2>(1,1)),
      MyProgram.get_kernel<MyKernel>(),  // execute the kernel as compiled in MyProgram
      ([=](cl::sycl::nd_item<2> index) {
        //[kernel code]
      }));
});
```

In the above example, the *kernel function* is defined in the `parallel_for` invocation as part of a lambda function which is named using the type of the forward declared class "myKernel". The type of the function object and the program object enable the compilation and linking of the kernel in the program class, *a priori* of its actual invocation as a kernel object. For more details on the SYCL device compiler please refer to chapter 6.

In the next example, a SYCL kernel is linked with an existing pre-compiled OpenCL C program object to create a combined program object, which is then called in a `parallel_for`.

```
class MyKernel;  // Forward declaration of the name of the lambda functor

cl::sycl::queue myQueue;

// obtain an existing OpenCL C program object
cl_program myClProgram = ...;

// Create a SYCL program object from a cl_program object
cl::sycl::program myExternProgram(myQueue.get_context(), myClProgram);

// Release the program if we no longer need it as
// SYCL program retained a reference to it
clReleaseProgram(myClProgram);

// Add in the SYCL program object for our kernel
cl::sycl::program mySyclProgram(myQueue.get_context());
mySyclProgram.compile_with_kernel_type<MyKernel>("-my-compile-options");

// Link myClProgram with the SYCL program object
mySyclProgram.link(myExternProgram,"-my-link-options");

myQueue.submit([&](handler& commandgroup) {
  commandgroup.parallel_for<class MyKernel>(
      cl::sycl::range<2>(4, 4),
      myLinkedProgram.get_kernel<MyKernel>(),  // execute the kernel as compiled in MyProgram
      ([=](cl::sycl::item<2> index) {
        //[kernel code]
      }));
});
```

Defining kernels using OpenCL C kernel objects

In OpenCL C [1] program and kernel objects can be created using the OpenCL C API, which is available in the SYCL system. Interoperability of OpenCL C kernels and the SYCL system is achieved by allowing the creation of a *SYCL kernel* object from an *OpenCL kernel* object.

The constructor using kernel objects from 4.82:

```
kernel::kernel(cl_kernel kernel, const context& syclContext)
```

creates a `kernel` which can be enqueued using all of the `parallel_for` functions that can enqueue a kernel object. This way of defining kernels assumes that the developer is using OpenCL C to create the kernel and to set the kernel arguments. The system assumes that the developer has already set kernel arguments when they are trying to enqueue the kernel. Buffers do give ownership to their accessors on specific contexts and the developer can enqueue OpenCL kernels in the same way as enqueuing SYCL kernels. However, the system is not responsible for data management at this point. Note that like all constructors from OpenCL API objects, constructing a `kernel` from a `cl_kernel` will retain a reference to the kernel and the user code should call `clReleaseKernel` if the `cl_kernel` is no longer needed in the calling context.

Rules for parameter passing to kernels

In a case where a kernel is a named function object or a lambda function, any member variables encapsulated within the function object or variables captured by the lambda function must be treated according to the following rules:

- Any accessor must be passed as an argument to the device kernel in a form that allows the device kernel to access the data in the specified way. For OpenCL 1.0–1.2 class devices, this means that the argument must be passed via `clSetKernelArg` and be compiled as a kernel parameter of the valid reference type. For global shared data access, the parameter must be an OpenCL `global` pointer. For an accessor that specifies OpenCL `constant` access, the parameter must be an OpenCL `constant` pointer. For images, the accessor must be passed as an `image_t` and/or sampler.
- The SYCL runtime and compiler(s) must produce the necessary conversions to enable accessor arguments from the host to be converted to the correct type of parameter on the device.
- A local accessor provides access to work-group-local memory. The accessor is not constructed with any buffer, but instead constructed with a size and base data type. The runtime must ensure that the work-group-local memory is allocated per work-group and available to be used by the kernel via the local accessor.
- C++ standard layout values must be passed by value to the kernel.
- C++ non-standard layout values must not be passed as arguments to a kernel that is compiled for a device.
- It is illegal to pass a buffer or image (instead of an accessor class) as an argument to a kernel. Generation of a compiler error in this illegal case is optional.
- Sampler objects (`sampler`) can be passed as parameters to kernels.
- It is illegal to pass a pointer or reference argument to a kernel. Generation of a compiler error in this illegal case is optional.
- Any aggregate types such as structs or classes should follow the rules above recursively. It is not necessary to separate struct or class members into separate OpenCL kernel parameters if all members of the aggregate type are unaffected by the rules above.

Error handling

Error Handling Rules

Error handling in SYCL uses exceptions. If an error occurs, it can be propagated at the point of a function call. An exception will be thrown and may be caught by the user using standard C++ exception handling mechanisms. For example, any exception which is triggered from code executed on the host is able to be propagated at the call site and it will follow the standard C++ exception handling mechanisms.

SYCL applications are asynchronous in the sense that host and device code executions are executed asynchronously. As a result of this, the errors that occur on a device cannot be propagated directly from the call site, and they will not be detected until the error-causing task executes or tries to execute. We refer to those errors as asynchronous errors. A good example of an asynchronous error is an out-of-bounds access error. In this case,

if the kernel is enqueued on a SYCL OpenCL device, then the out-of-bounds error is asynchronous with respect to the SYCL host application, because it is executed on the device. The standard exception mechanisms will not be available as this is an asynchronous error.

SYCL queues are by default asynchronous, as they schedule tasks on SYCL devices. The queue constructor can optionally get an asynchronous handler object `async_handler`, which is a function class instance. If waiting and exception handling member functions are used on queues, the async_handler receives a list of C++ exception objects.

If an asynchronous error occurs in a queue that has no user-supplied asynchronous error handler object `async_handler`, then no exception is thrown and the error is not available to the user in any specified way. Implementations may provide extra debugging information to users to trap and handle asynchronous errors. If a synchronous error occurs in a SYCL application and it is not handled, the application will exit abnormally.

If an 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 primary 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 no asynchronous error reporting is done and no exceptions are thrown. 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. The command group function object event returned by that function will be relevant to the queue where the kernel has been enqueued.

Exception Class Interface

```
namespace cl {
namespace sycl {

using async_handler = function_class<void(cl::sycl::exception_list)>;

class exception {
 public:
  const char *what() const;

  bool has_context() const;

  context get_context() const;

  cl_int get_cl_code() const;
};

class exception_list {
  // Used as a container for a list of asynchronous exceptions
 public:
  using value_type = exception_ptr_class;
  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;
  iterator begin() const;  // first asynchronous exception
  iterator end() const;    // refer to past-the-end last asynchronous exception
};

class runtime_error : public exception;

class kernel_error : public runtime_error;

class accessor_error : public runtime_error;

class nd_range_error : public runtime_error;

class event_error : public runtime_error;

class invalid_parameter_error : public runtime_error;

class device_error : public exception;

class compile_program_error : public device_error;

class link_program_error : public device_error;

class invalid_object_error : public device_error;

class memory_allocation_error : public device_error;

class platform_error : public device_error;

class profiling_error : public device_error;

class feature_not_supported : public device_error;

}  // namespace sycl
}  // namespace cl
```

The SYCL `exception_ptr_class` class is used to store SYCL `exception` objects and allows exception objects to be transferred between threads. It is equivalent to the `exception_ptr_class` class. The SYCL `exception_list` class is also available in order to provide a list of synchronous and asynchronous exceptions.

There are two categories of errors, the `runtime_error` that refers to the scheduling errors that may happen during execution, and the `device_error` that refers to the execution errors on a SYCL device.

Errors can occur both in the SYCL library and SYCL host side, as well as the OpenCL runtime and device side. The member functions on these exceptions provide the corresponding information. If there is an OpenCL error associated with the exception triggered, then the OpenCL error code will be given by the method `get_cl_code()`. In the case where there is no OpenCL error associated with the exception triggered, the OpenCL error code will be 0.

The asynchronous handler object async_handler is a function_class with an exception_list as a parameter. The asynchronous handler is an optional parameter to a constructor of the queue class and it is the only way to handle asynchronous errors occurring on a SYCL device. The asynchronous handler may be a named function object type, a lambda function or a function_class, that can be given to the queue and will be executed on error. The exception_list object is constructed from the SYCL runtime and is populated with the errors caught during the execution of all the kernels running on the same queue.

Member function	Description
const char *what()const	Returns an implementation defined non-null constant C-style string that describes the error that triggered the exception.
bool has_context()const	Returns true if this SYCL exception has an associated SYCL context and false if it does not.
context get_context()const	Returns the SYCL context that is associated with this SYCL exception if one is available. Must throw an invalid_object_error SYCL exception if this SYCL exception does not have a SYCL context.
cl_int get_cl_code()const	Returns the OpenCL error code if the exception was thrown as an OpenCL error, otherwise returns CL_SUCCESS.
	End of table

Table 4.89: Member functions of the SYCL exception class.

Member function	Description
size_t size()const	Returns the size of the list
iterator begin()const	Returns an iterator to the beginning of the list of asynchronous exceptions.
iterator end()const	Returns an iterator to the end of the list of asynchronous exceptions.
	End of table

Table 4.90: Member functions of the exception_list.

Runtime Error Exception Type	Description
kernel_error	Error that occurred before or while enqueuing the SYCL kernel.
nd_range_error	Error regarding the SYCL nd_range specified for the SYCL kernel
accessor_error	Error regarding the SYCL accessor objects defined.
event_error	Error regarding associated SYCL event objects.
	Continued on next page

Table 4.91: Exceptions types that derive from the runtime_error class.

Runtime Error Exception Type	Description
`invalid_parameter_error`	Error regarding parameters to the SYCL kernel, it may apply to any captured parameters to the kernel lambda.
	End of table

Table 4.91: Exceptions types that derive from the `runtime_error` class.

Device Error Exception Type	Description
`compile_program_error`	Error while compiling the SYCL kernel to a SYCL device.
`link_program_error`	Error while linking the SYCL kernel to a SYCL device.
`invalid_object_error`	Error regarding any memory objects being used inside the kernel
`memory_allocation_error`	Error on memory allocation on the SYCL device for a SYCL kernel.
`platform_error`	The SYCL platform will trigger this exception on error.
`profiling_error`	The SYCL runtime will trigger this error if there is an error when profiling info is enabled.
`feature_not_supported`	Exception thrown when an optional feature or extension is used in a kernel but its not available on the device the SYCL kernel is being enqueued on.
	End of table

Table 4.92: Exception types that derive from the SYCL `device_error` class.

Data types

SYCL as a C++11 programming model supports the C++11 ISO standard data types, and it also provides the ability for all SYCL applications to be executed on SYCL compatible devices, OpenCL and host 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 OpenCL interoperability types are found in 6.5.

Scalar data types

The fundamental C++ data types which are supported in SYCL are described in Table 6.1. Note these types are fundamental and therefore do not exist within the `cl::sycl` namespace.

Additional scalar data types which are supported by SYCL within the `cl::sycl` namespace are described in Table 4.93.

Scalar data type	Description
`byte`	A signed or unsigned 8-bit integer, as defined by the C++11 ISO Standard.
	End of table

Table 4.93: Additional scalar data types supported by SYCL.

The OpenCL C language standard [1, §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 `cl::sycl` namespace using the `cl_` prefix. These aliases are described in Table 4.94

Scalar data type alias	Description
`cl_bool`	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.
`cl_char`	Alias to a signed 8-bit integer, as defined by the C++11 ISO Standard.
`cl_uchar`	Alias to an unsigned 8-bit integer, as defined by the C++11 ISO Standard.
`cl_short`	Alias to a signed 16-bit integer, as defined by the C++11 ISO Standard.
`cl_ushort`	Alias to an unsigned 16-bit integer, as defined by the C++11 ISO Standard.
`cl_int`	Alias to a signed 32-bit integer, as defined by the C++11 ISO Standard.
`cl_uint`	Alias to an unsigned 32-bit integer, as defined by the C++11 ISO Standard.
`cl_long`	Alias to a signed 64-bit integer, as defined by the C++11 ISO Standard.
`cl_ulong`	Alias to an unsigned 64-bit integer, as defined by the C++11 ISO Standard.
`cl_float`	Alias to a 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
`cl_double`	Alias to a 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.
`cl_half`	Alias to a 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. A SYCL `feature_not_supported` exception must be thrown if the `half` type is used in a SYCL kernel fucntion which executes on a SYCL `device` that does not support the extension `khr_fp16`.
	End of table

Table 4.94: Scalar data type aliases supported by SYCL.

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 methods that allow construction of a new vector from a swizzled set of component elements.

`vec<typename dataT, int numElements>` is a vector type that compiles down to the OpenCL built-in vector types on OpenCL devices, where possible, and provides compatible support on the host. 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 should 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 OpenCL 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 OpenCL C 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.

Vec interface

The constructors, member functions and non-member functions of the SYCL `vec` class template are listed in Tables 4.95, 4.96 and 4.97 respectively.

```
namespace cl {
namespace sycl {

enum class rounding_mode {
  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;
```

```
  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 vec(const dataT &arg);

  template <typename... argTN>
  vec(const argTN&... args);

  vec(const vec<dataT, numElements> &rhs);

#ifdef __SYCL_DEVICE_ONLY__
  vec(vector_t openclVector);

  operator vector_t() const;
#endif

  // Available only when: numElements == 1
  operator dataT() const;

  size_t get_count() const;

  size_t get_size() const;

  template <typename convertT, rounding_mode roundingMode>
  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;

#ifdef 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>
  void load(size_t offset, multi_ptr<const dataT, addressSpace> ptr);
  template <access::address_space addressSpace>
  void store(size_t offset, multi_ptr<dataT, addressSpace> ptr) const;

  // OP is: +, -, *, /, %
  /* When OP is % available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> operatorOP(const vec<dataT, numElements> &rhs) const;
  vec<dataT, numElements> operatorOP(const dataT &rhs) const;

  // OP is: +=, -=, *=, /=, %=
  /* When OP is %= available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs);
  vec<dataT, numElements> &operatorOP(const dataT &rhs);

  // OP is: ++, --
  vec<dataT, numElements> &operatorOP();
  vec<dataT, numElements> operatorOP(int);

  // OP is: &, |, ^
  /* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> operatorOP(const vec<dataT, numElements> &rhs) const;
  vec<dataT, numElements> operatorOP(const dataT &rhs) const;
```

```
  // OP is: &=, |=, ^=
  /* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs);
  vec<dataT, numElements> &operatorOP(const dataT &rhs);

  // OP is: &&, ||
  vec<RET, numElements> operatorOP(const vec<dataT, numElements> &rhs) const;
  vec<RET, numElements> operatorOP(const dataT &rhs) const;

  // OP is: <<, >>
  /* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> operatorOP(const vec<dataT, numElements> &rhs) const;
  vec<dataT, numElements> operatorOP(const dataT &rhs) const;

  // OP is: <<=, >>=
  /* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs);
  vec<dataT, numElements> &operatorOP(const dataT &rhs);

  // OP is: ==, !=, <, >, <=, >=
  vec<RET, numElements> operatorOP(const vec<dataT, numElements> &rhs) const;
  vec<RET, numElements> operatorOP(const dataT &rhs) const;

  vec<dataT, numElements> &operator=(const vec<dataT, numElements> &rhs);
  vec<dataT, numElements> &operator=(const dataT &rhs);

  /* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
  vec<dataT, numElements> operator~() const;

  vec<RET, numElements> operator!() const;
};

// OP is: +, -, *, /, %
/* operator% is only available when: dataT != cl_float && dataT != cl_double &&
dataT != cl_half. */
template <typename dataT, int numElements>
vec<dataT, numElements> operatorOP(const dataT &lhs,
  const vec<dataT, numElements> &rhs);

// OP is: &, |, ^
/* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
template <typename dataT, int numElements>
vec<dataT, numElements> operatorOP(const dataT &lhs,
  const vec<dataT, numElements> &rhs);

// OP is: &&, ||
template <typename dataT, int numElements>
vec<RET, numElements> operatorOP(const dataT &lhs,
  const vec<dataT, numElements> &rhs);
```

```
// OP is: <<, >>
/* Available only when: dataT != cl_float && dataT != cl_double
  && dataT != cl_half. */
template <typename dataT, int numElements>
vec<dataT, numElements> operatorOP(const dataT &lhs,
    const vec<dataT, numElements> &rhs);

// OP is: ==, !=, <, >, <=, >=
template <typename dataT, int numElements>
vec<RET, numElements> operatorOP(const dataT &lhs,
    const vec<dataT, numElements> &rhs);
}  // namespace sycl
}  // namespace cl
```

Constructor	Description
`vec()`	Default construct a vector with element type `dataT` and with `numElements` dimensions by default construction of each of its elements.
`explicit vec(const dataT &arg)`	Construct a vector of element type `dataT` and `numElements` dimensions by setting each value to `arg` by assignment.
`template <typename... argTN> vec(const argTN&... args)`	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 `numElements` of this `vec` specialization.
`vec(const vec<dataT, numElements> &rhs)`	Construct a vector of element type `dataT` and number of elements `numElements` by copy from another similar vector.
`vec(vector_t openclVector)`	Available only when: compiled for the device. Constructs a SYCL `vec` instance from an instance of the underlying OpenCL vector type defined by `vector_t`.
	End of table

Table 4.95: Constructors of the SYCL `vec` class template.

Member function	Description
`operator vector_t()const`	Available only when: compiled for the device. Converts this SYCL `vec` instance to the underlying OpenCL vector type defined by `vector_t`.
`operator dataT()const`	Available only when: `numElements == 1`. Converts this SYCL `vec` instance to an instance of `dataT` with the value of the single element in this SYCL `vec` instance.
	Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`size_t get_count()const`	Returns the number of elements of this SYCL `vec`.
`size_t get_size()const`	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 4.10.2.6.
`template<typename convertT, rounding_mode roundingMode>` `vec<convertT, numElements> convert()const`	Converts this SYCL `vec` to a SYCL `vec` of a different element type specified by `convertT` using the rounding mode specified by `roundingMode`. The new SYCL `vec` type must have the same number of elements as this SYCL `vec`. The different rounding modes are described in Table 4.98.
`template<typename asT>` `asT as()const`	Bitwise reinterprets this SYCL `vec` as a SYCL `vec` of a different element type and number of elements specified by `asT`. The new SYCL `vec` type must have the same storage size in bytes as this SYCL `vec`.
`template<int... swizzleIndexes>` `__swizzled_vec__ swizzle()const`	Return 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 4.10.2.4.
`__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 4.10.2.4. Where `XYZW_ACCESS` is: `x` for `numElements == 1`, `x, y` for `numElements == 2`, `x, y, z` for `numElements == 3` and `x, y, z, w` for `numElements == 4`.
`__swizzled_vec__ RGBA_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 4.10.2.4. Where `RGBA_ACCESS` is: `r, g, b, a`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`__swizzled_vec__ INDEX_ACCESS()const`	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 4.10.2.4. 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`.
`__swizzled_vec__ XYZW_SWIZZLE()const`	Available only when `numElements <= 4`, and when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `cl/sycl.hpp`. 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 4.10.2.4. Where XYZW_SWIZZLE is all permutations with repetition of `x, y` for `numElements == 2`, `x, y, z` for `numElements == 3` and `x, y, z, w` for `numElements == 4`. For example `xzyw` and `xyyy`.
`__swizzled_vec__ RGBA_SWIZZLE()const`	Available only when `numElements == 4`, and when the macro `SYCL_SIMPLE_SWIZZLES` is defined before including `cl/sycl.hpp`. 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 4.10.2.4. Where RGBA_SWIZZLE is all permutations with repetition of `r, g, b, a`. For example `bgra` and `rrba`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`__swizzled_vec__ lo()const`	Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` 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 4.10.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined.
`__swizzled_vec__ hi()const`	Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` 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 4.10.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined.
`__swizzled_vec__ odd()const`	Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` 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 4.10.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined.
`__swizzled_vec__ even()const`	Available only when: `numElements > 1`. Return an instance of the implementation defined intermediate class template `__swizzled_vec__` 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 4.10.2.4. When `numElements == 3` this SYCL `vec` is treated as though `numElements == 4` with the fourth element undefined.
`template <access::address_space addressSpace> void load(size_t offset, multi_ptr<const dataT, addressSpace> ptr)`	Loads the values at the address of `ptr` offset in elements of type `dataT` by `numElements * offset`, into the components of this SYCL `vec`.
`template <access::address_space addressSpace> void store(size_t offset, multi_ptr<dataT, addressSpace> ptr)const`	Stores the components of this SYCL `vec` into the values at the address of `ptr` offset in elements of type `dataT` by `numElements * offset`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<dataT, numElements> operatorOP(const vec<dataT, numElements> &rhs)const`	When `OP` is `%` available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` arithmetic operation between each element of this SYCL `vec` and each element of the `rhs` SYCL `vec`. Where `OP` is: `+`, `-`, `*`, `/`, `%`.
`vec<dataT, numElements> operatorOP(const dataT &rhs)const`	When `OP` is `%` available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` arithmetic operation between each element of this SYCL `vec` and the `rhs` scalar. Where `OP` is: `+`, `-`, `*`, `/`, `%`.
`vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs)`	When `OP` is `%=` available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` arithmetic operation between each element of this SYCL `vec` and each element of the `rhs` SYCL `vec` and return a reference to this SYCL `vec`. Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`.
`vec<dataT, numElements> &operatorOP(const dataT &rhs)`	When `OP` is `%=` available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` arithmetic operation between each element of this SYCL `vec` and `rhs` scalar and return a reference to this SYCL `vec`. Where `OP` is: `+=`, `-=`, `*=`, `/=`, `%=`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<dataT, numElements> &operatorOP()`	Perform an in-place element-wise `OP` prefix arithmetic operation on each element of this SYCL `vec`, assigning the result of each element to the corresponding element of this SYCL `vec` and return a reference to this SYCL `vec`. Where `OP` is: `++`, `--`.
`vec<dataT, numElements> operatorOP(int)`	Perform an in-place element-wise `OP` postfix arithmetic operation on each element of this SYCL `vec`, assigning the result of each element to the corresponding element of this SYCL `vec` and returns a copy of this SYCL `vec` before the operation is performed. Where `OP` is: `++`, `--`.
`vec<dataT, numElements> operatorOP(` `const vec<dataT, numElements> &rhs)const`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between each element of this SYCL `vec` and each element of the `rhs` SYCL `vec`. Where `OP` is: `&`, `|`, `^`.
`vec<dataT, numElements> operatorOP(` `const dataT &rhs)const`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation between each element of this SYCL `vec` and the `rhs` scalar. Where `OP` is: `&`, `|`, `^`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs)`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` bitwise operation between each element of this SYCL `vec` and the `rhs` SYCL `vec` and return a reference to this SYCL `vec`. Where `OP` is: `&=`, `|=`, `^=`.
`vec<dataT, numElements> &operatorOP(const dataT &rhs)`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` bitwise operation between each element of this SYCL `vec` and the `rhs` scalar and return a reference to this SYCL `vec`. Where `OP` is: `&=`, `|=`, `^=`.
`vec<RET, numElements> operatorOP(const vec<dataT, numElements> &rhs)const`	Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation between each element of this SYCL `vec` and each element of the `rhs` SYCL `vec`. 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 `cl_char` or `cl_uchar` `RET` must be `cl_char`. For a SYCL `vec` with `dataT` of type `cl_short`, `cl_ushort` or `cl_half` `RET` must be `cl_short`. For a SYCL `vec` with `dataT` of type `cl_int`, `cl_uint` or `cl_float` `RET` must be `cl_int`. For a SYCL `vec` with `dataT` of type `cl_long`, `cl_ulong` or `cl_double` `RET` must be `cl_long`. Where `OP` is: `&&`, `||`.
Continued on next page	

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<RET, numElements> operatorOP(` `  const dataT &rhs)const`	Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation between each element of this SYCL `vec` and the `rhs` scalar. 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 `cl_char` or `cl_uchar` `RET` must be `cl_char`. For a SYCL `vec` with `dataT` of type `cl_short`, `cl_ushort` or `cl_half` `RET` must be `cl_short`. For a SYCL `vec` with `dataT` of type `cl_int`, `cl_uint` or `cl_float` `RET` must be `cl_int`. For a SYCL `vec` with `dataT` of type `cl_long`, `cl_ulong` or `cl_double` `RET` must be `cl_long`. Where `OP` is: `&&`, `||`.
`vec<dataT, numElements> operatorOP(` `  const vec<dataT, numElements> &rhs)const`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitshift operation between each element of this SYCL `vec` and each element of the `rhs` SYCL `vec`. If `OP` is `>>`, `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: `<<`, `>>`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<dataT, numElements> operatorOP(const dataT &rhs)const`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitshift operation between each element of this SYCL `vec` and the `rhs` scalar. If `OP` is >>, `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: <<, >>.
`vec<dataT, numElements> &operatorOP(const vec<dataT, numElements> &rhs)`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` bitshift operation between each element of this SYCL `vec` and the `rhs` SYCL `vec` and returns a reference to this SYCL `vec`. If `OP` is >>=, `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: <<=, >>=.
`vec<dataT, numElements> &operatorOP(const dataT &rhs)`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Perform an in-place element-wise `OP` bitshift operation between each element of this SYCL `vec` and the `rhs` scalar and returns a reference to this SYCL `vec`. If `OP` is >>=, `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: <<=, >>=.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
vec<RET, numElements> operatorOP(const vec<dataT, numElements> &rhs)const	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 this SYCL 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 or either this SYCL vec or the rhs SYCL vec 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 cl_char or cl_uchar RET must be cl_char. For a SYCL vec with dataT of type cl_short, cl_ushort or cl_half RET must be cl_short. For a SYCL vec with dataT of type cl_int, cl_uint or cl_float RET must be cl_int. For a SYCL vec with dataT of type cl_long, cl_ulong or cl_double RET must be cl_long. Where OP is: ==, !=, <, >, <=, >=.
Continued on next page	

Table 4.96: Member functions for the SYCL vec class template.

Member function	Description
`vec<RET, numElements> operatorOP(const dataT &rhs) const`	Construct a new instance of the SYCL `vec` class template with the `dataT` parameter of `RET` with each element of the new SYCL `vec` instance the result of an element-wise `OP` relational operation between each element of this SYCL `vec` and the `rhs` scalar. 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 either this SYCL `vec` or the `rhs` SYCL `vec` 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 `cl_char` or `cl_uchar` `RET` must be `cl_char`. For a SYCL `vec` with `dataT` of type `cl_short`, `cl_ushort` or `cl_half` `RET` must be `cl_short`. For a SYCL `vec` with `dataT` of type `cl_int`, `cl_uint` or `cl_float` `RET` must be `cl_int`. For a SYCL `vec` with `dataT` of type `cl_long`, `cl_ulong` or `cl_double` `RET` must be `cl_long`. Where `OP` is: ==, !=, <, >, <=, >=.
`vec<dataT, numElements> &operator=( const vec<dataT, numElements> &rhs)`	Assign each element of the `rhs` SYCL `vec` to each element of this SYCL `vec` and return a reference to this SYCL `vec`.
`vec<dataT, numElements> &operator=( const dataT &rhs)`	Assign each element of the `rhs` scalar to each element of this SYCL `vec` and return a reference to this SYCL `vec`.
`vec<dataT, numElements> operator~()`	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`. Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` bitwise operation on each element of this SYCL `vec`.

Continued on next page

Table 4.96: Member functions for the SYCL `vec` class template.

Member function	Description
`vec<RET, numElements> operator!()`	Construct a new instance of the SYCL `vec` class template with the same template parameters as this SYCL `vec` with each element of the new SYCL `vec` instance the result of an element-wise `OP` logical operation on each element of this 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` or this SYCL `vec` 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 `cl_char` or `cl_uchar` `RET` must be `cl_char`. For a SYCL `vec` with `dataT` of type `cl_short`, `cl_ushort` or `cl_half` `RET` must be `cl_short`. For a SYCL `vec` with `dataT` of type `cl_int`, `cl_uint` or `cl_float` `RET` must be `cl_int`. For a SYCL `vec` with `dataT` of type `cl_long`, `cl_ulong` or `cl_double` `RET` must be `cl_long`.
	End of table

Table 4.96: Member functions for the SYCL `vec` class template.

Non-member function	Description

`vec<dataT, numElements> operatorOP(`
`  const dataT &lhs,`
`  const vec<dataT, numElements> &rhs)`
	When `OP` is `%` available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_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` arithmetic operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`.
	
	Where `OP` is: `+`, `-`, `*`, `/`, `%`.

	Continued on next page

Table 4.97: Non-member functions of the `vec` class template.

Non-member function	Description
vec<dataT, numElements> operatorOP(const dataT &lhs, const vec<dataT, numElements> &rhs)	Available only when: dataT != cl_float && dataT != cl_double && dataT != cl_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: &, |, ^.
vec<RET, numElements> operatorOP(const dataT &lhs, const vec<dataT, numElements> &rhs)	Available only when: dataT != cl_float && dataT != cl_double && dataT != cl_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 logical operation between the lhs scalar and each element of the rhs SYCL vec. 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 cl_char or cl_uchar RET must be cl_char. For a SYCL vec with dataT of type cl_short, cl_ushort or cl_half RET must be cl_short. For a SYCL vec with dataT of type cl_int, cl_uint or cl_float RET must be cl_int. For a SYCL vec with dataT of type cl_long, cl_ulong or cl_double RET must be cl_long. Where OP is: &&, ||.
Continued on next page	

Table 4.97: Non-member functions of the vec class template.

Non-member function	Description

`vec<dataT, numElements> operatorOP(`
`  const dataT &lhs,`
`  const vec<dataT, numElements> &rhs)`
	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` bitshift operation between the `lhs` scalar and each element of the `rhs` SYCL `vec`. If `OP` is `>>`, `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: `<<`, `>>`.

`vec<RET, numElements> operatorOP(`
`  const dataT &lhs,`
`  const vec<dataT, numElements> &rhs)`
	Available only when: `dataT != cl_float && dataT != cl_double && dataT != cl_half`.
	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` or either this SYCL `vec` or the `rhs` SYCL `vec` 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 `cl_char` or `cl_uchar` `RET` must be `cl_char`. For a SYCL `vec` with `dataT` of type `cl_short`, `cl_ushort` or `cl_half` `RET` must be `cl_short`. For a SYCL `vec` with `dataT` of type `cl_int`, `cl_uint` or `cl_float` `RET` must be `cl_int`. For a SYCL `vec` with `dataT` of type `cl_long`, `cl_ulong` or `cl_double` `RET` must be `cl_long`.
	
	Where `OP` is: `==`, `!=`, `<`, `>`, `<=`, `>=`.

	End of table

Table 4.97: Non-member functions of the `vec` class template.

Aliases

SYCL provides aliases for vec<dataT, numElements> as <dataT><numElements> for the data types: char, short, int, long, float, double, half, cl_char, cl_uchar, cl_short, cl_ushort, cl_int, cl_uint, cl_long, cl_ulong, cl_float, cl_double and cl_half and the data types: signed char , unsigned char, unsigned short, unsigned int, unsigned long, long long and unsigned long long represented with the short hand schar, uchar, ushort, uint, ulong, longlong and ulonglong respectively, for number of elements: 2, 3, 4, 8, 16. For example the alias to vec<float, 4> would be float4.

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, specifying swizzle indexes. Secondly by calling one of the simple swizzle member functions defined in 4.96 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 CL/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.

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 inhetired from the `vec` class template interface which would return `vec<dataT, numElements> &`.

Rounding modes

The various rounding modes that can be used in the `as` member function template are described in Table 4.98.

Rounding mode	Description
`automatic`	Default rounding mode for the SYCL `vec` class element type element type, `rtz` (round toward zero) for integer types and `rte` (round to nearest even) for floating-point types.
`rte`	Round to nearest even.
`rtz`	Round toward zero.
`rtp`	Round toward positive infinity.
`rtn`	Round toward negative infinity.
	End of table

Table 4.98: Rounding modes for the SYCL `vec` class template.

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:

$$\texttt{sizeof(dataT)} \cdot \texttt{numElements} \tag{4.5}$$

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:

$$\texttt{sizeof(dataT)} \cdot 4 \tag{4.6}$$

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.

Considerations for endianness

As SYCL supports both big-endian and little-endian on OpenCL devices, users must take care to ensure kernel arguments are processed correctly. This is particularly true for SYCL `vec` arguments as the order in which a SYCL `vec` is loaded differs between big-endian and little-endian.

Users should consult vendor documentation for guidance on how to handle kernel arguments in these situations.

Synchronization and atomics

The SYCL specification offers the same set of synchronization operations that are available to OpenCL C programs, for compatibility and portability across OpenCL devices. 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: OpenCL 1.2 devices only support the equivalent of relaxed C++ atomics and SYCL uses the C++11 library syntax to make this available. This is provided for forward compatibility with future SYCL versions.
- Barriers: Barrier primitives are made available to synchronize sets of work-items within individual work-groups. They are exposed through the `nd_item` class that abstracts the current point in the overall iteration space.
- 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.

A work-group barrier or work-group mem-fence may provide ordering semantics over the local address space, global address space or both. All memory operations initiated before the work-group barrier or work-group mem-fence operation in the specified address space(s) will be completed before any memory operation after the operation. Address spaces are specified using the `fence_space` enum class:

```
namespace cl {
namespace sycl {
namespace access {
enum class fence_space : char {
  local_space,
  global_space,
  global_and_local
};  // enum class fence_space
}  // namepaces access
}  // namespace sycl
}  // namespace cl
```

The SYCL specification provides atomic operations based on the C++11 library syntax. The only available ordering, due to constraints of the OpenCL 1.2 memory model, is `memory_order_relaxed`. No default order is supported because a default order would imply sequential consistency. The SYCL atomic library may map directly to the underlying C++11 library in host code, and must interact safely with the host C++11 atomic library when used in host code. The SYCL library must be used in device code to ensure that only the limited subset of functionality is available. SYCL 1.2.1 device compilers should give a compilation error on use of the `std::atomic` classes and functions in device code.

The template parameter `addressSpace` is permitted to be `access::address_space::global_space` or `access::address_space::local_space`.

The data type `T` is permitted to be `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long` and `float`. Though `float` is only available for the `store`, `load` and `exchange` member functions. For any data type `T` which is 64bit, the member functions of the `atomic` class are required to compile however are only guaranteed to execute if the 64bit atomic extension `cl_khr_int64_base_atomics` or `cl_khr_int64_extended_atomics` (depending on which extension provides support for each given member function) is supported by the SYCL `device` which is executing the SYCL kernel function. If a member function is called with a 64bit data type and the necessary extension is not supported by the SYCL `device` which is executing the SYCL kernel function, the SYCL runtime must throw a SYCL `feature_not_supported` exception. For more detail see Section 5.2.

The atomic types are defined as follows, the constructors and member functions for the SYCL `atomic` class are listed in Tables 4.99 and 4.100 respectively.

```
namespace cl {
namespace sycl {
enum class memory_order : int {
  relaxed
};

template <typename T, access::address_space addressSpace =
  access::address_space::global_space>
class atomic {
 public:
  template <typename pointerT>
  atomic(multi_ptr<pointerT, addressSpace> 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,
    memory_order successMemoryOrder = memory_order::relaxed,
    memory_order failMemoryOrder = memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_add(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_sub(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_and(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_or(T operand, memory_order memoryOrder =
    memory_order::relaxed);
```

```
  /* Available only when: T != float */
  T fetch_xor(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_min(T operand, memory_order memoryOrder =
    memory_order::relaxed);

  /* Available only when: T != float */
  T fetch_max(T operand, memory_order memoryOrder =
    memory_order::relaxed);
};
}  // namespace sycl
}  // namespace cl
```

As well as the member functions, a matching set of operations on atomic types is provided by the SYCL library. As in the previous case, the only available memory order is `memory_order::relaxed`. The global functions are as follows and described in Table 4.101.

```
namespace cl {
namespace sycl {
template <typename T, access::address_space addressSpace>
void atomic_store(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

template <typename T, access::address_space addressSpace>
T atomic_load(atomic<T, addressSpace> object, memory_order memoryOrder =
  memory_order::relaxed);

template <typename T, access::address_space addressSpace>
T atomic_exchange(atomic<T, addressSpace> object, T operand, memory_order memoryOrder =
  memory_order::relaxed);

template <typename T, access::address_space addressSpace>
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);

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 cl
```

The atomic operations and member functions behave as described in the C++11 specification, barring the restrictions discussed above. Note that care must be taken when using `compare_exchange_strong` to perform many of the operations that would be expected of it in standard CPU code due to the lack of forward progress guarantees between work-items in SYCL. No work-item may be dependent on another work-item to make progress if the code is to be portable.

Constructor	Description
`template <typename pointerT> atomic(multi_ptr<pointerT, addressSpace> ptr)`	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`.
	End of table

Table 4.99: Constructors of the SYCL `atomic` class template.

Member function	Description
`void store(T operand, memory_order memoryOrder = memory_order::relaxed)`	Atomically stores the value `operand` at the address of the `multi_ptr` associated with this SYCL `atomic`. The memory order of this atomic operation must be `memory_order::relaxed`.
`T load(memory_order memoryOrder = memory_order::relaxed)const`	Atomically loads 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`.
	Continued on next page

Table 4.100: Member functions available on an object of type `atomic<T>`.

Member function	Description
`T exchange(T operand, memory_order memoryOrder = memory_order::relaxed)`	Atomically replaces the value at the address of the `multi_ptr` associated with this SYCL `atomic` with value `operand` and 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`.
`bool compare_exchange_strong(T &expected, T desired, memory_order successMemoryOrder = memory_order::relaxed, memory_order failMemoryOrder = memory_order::relaxed)`	Available only when: `T != float`. 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.
`T fetch_add(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. 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`.
`T fetch_sub(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. Atomically subtracts 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`.

Continued on next page

Table 4.100: Member functions available on an object of type `atomic<T>`.

Member function	Description
`T fetch_and(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. Atomically performs a bitwise AND between the value `operand` and 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`.
`T fetch_or(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. Atomically performs a bitwise OR between the value `operand` and 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`.
`T fetch_xor(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. Atomically performs a bitwise XOR between the value `operand` and 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`.
`T fetch_min(T operand, memory_order memoryOrder = memory_order::relaxed)`	Atomically computes the minimum of the value `operand` and 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`.

Continued on next page

Table 4.100: Member functions available on an object of type `atomic<T>`.

Member function	Description
`T fetch_max(T operand, memory_order memoryOrder = memory_order::relaxed)`	Available only when: `T != float`. Atomically computes the maximum of the value `operand` and 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`.

End of table

Table 4.100: Member functions available on an object of type `atomic<T>`.

Functions	Description
`template <typename T, access::address_space addressSpace> T atomic_load(atomic<T, addressSpace> object, memory_order memoryOrder = memory_order::relaxed)`	Equivalent to calling `object.load(memoryOrder)`.
`template <typename T, access::address_space addressSpace> void atomic_store(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed)`	Equivalent to calling `object.store(operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_exchange(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed)`	Equivalent to calling `object.exchange(operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> 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)`	Equivalent to calling `object.compare_exchange_strong(expected, desired, successMemoryOrder, failMemoryOrders)`.

Continued on next page

Table 4.101: Global functions available on atomic types.

Functions	Description
`template <typename T, access::address_space addressSpace> T atomic_fetch_add(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_add( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_sub(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_sub( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_and(atomic<T> operand, T object, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_add( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_or(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_or( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_xor(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_xor( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_min(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_min( operand, memoryOrder)`.
`template <typename T, access::address_space addressSpace> T atomic_fetch_max(atomic<T, addressSpace> object, T operand, memory_order memoryOrder = memory_order::relaxed )`	Equivalent to calling `object.fetch_max( operand, memoryOrder)`.
	End of table

Table 4.101: Global functions available on atomic types.

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 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.

An instance of the SYCL `stream` class has a maximum buffer size that specifies maximum size of the character stream that can be output in bytes and a maximum statement size that specifies the maximum size of the character stream that can be output in a single statement in bytes.

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

The SYCL `stream` class provides the common reference semantics (see Section 4.3.2).

Stream class interface

The constructors and member functions of the SYCL `stream` class are listed in Tables 4.104, 4.105, and 4.106 respectively. The additional common special member functions and common member functions are listed in Tables 4.1 and 4.2, respectively.

The operand types that are supported by the SYCL `stream` class `operator<<()` operator are listed in Table 4.102.

The manipulators that are supported by the SYCL `stream` class `operator<<()` operator are listed in Table 4.103.

```
namespace cl {
namespace sycl {

enum class stream_manipulator {
  dec,
  hex,
  oct,
  noshowbase,
  showbase,
  noshowpos,
  showpos,
  endl,
  fixed,
  scientific,
  hexfloat,
  defaultfloat
};

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 bufferSize, size_t maxStatementSize, handler& cgh);

  /* -- common interface members -- */

  size_t get_size() const;

  size_t get_max_statement_size() const;
};

template <typename T>
const stream& operator<<(const stream& os, const T &rhs);

}  // namespace sycl
}  // namespace cl
```

Stream operand type	Description
char, signed char, unsigned char, int, unsigned int, short, unsigned short, long int, unsigned long int, long long int, unsigned long long int, cl_char, cl_uchar, cl_int, cl_uint, cl_short, cl_ushort, cl_long, cl_ulong, byte	Outputs the value as a stream of characters.
float, double, half, cl_float, cl_double, cl_half	Outputs the value according to the precision of the current statement as a stream of characters.
char *, const char *	Outputs the string.

Continued on next page

Table 4.102: Operand types supported by the stream class.

Stream operand type	Description
`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.
	End of table

Table 4.102: Operand types supported by the `stream` class.

Stream manipulator	Description
`endl`	Outputs a new-line character.
`dec`	Outputs any subsequent values in the current statement in decimal base.
`hex`	Outputs any subsequent values in the current statement in hexadecimal base.
`oct`	Outputs any subsequent values in the current statement in octal base.
`noshowbase`	Outputs any subsequent values without the base prefix.
`showbase`	Outputs any subsequent values with the base prefix.
`noshowpos`	Outputs any subsequent values without a plus sign if the value is positive.
`showpos`	Outputs any subsequent values with a plus sign if the value is positive.
`setw(int)`	Sets the field width of any subsequent values in the current statement.
`setprecision(int)`	Sets the precision of any subsequent values in the current statement.
`fixed`	Outputs any subsequent floating-point values in the current statement in fixed notation.
`scientific`	Outputs any subsequent floating-point values in the current statement in scientific notation.
`hexfloat`	Outputs any subsequent floating-point values in the current statement in hexadecimal notation.
`defaultfloat`	Outputs any subsequent floating-point values in the current statement in the default notation.
	End of table

Table 4.103: Manipulators supported by the `stream` class.

Constructor	Description
`stream(size_t bufferSize, size_t maxStatementSize, handler& cgh)`	Constructs a SYCL `stream` instance associated with the command group specified by `cgh`, with a maximum buffer size specified by the parameter `bufferSize` and a maximum statement size specified by the parameter `maxStatementSize`.
	End of table

Table 4.104: Constructors of the `stream` class.

Member function	Description
`size_t get_size()const`	Returns the maximum buffer size.
`size_t get_max_statement_size()const`	Returns the maximum statement size.
	End of table

Table 4.105: Member functions of the `stream` class.

Global function	Description
`template <typename T> const stream& operator<<(const stream& os, const T &rhs)`	Outputs any valid values (see 4.102) as a stream of characters and applies any valid manipulator (see 4.103) to the current statements.
	End of table

Table 4.106: Global functions of the `stream` class.

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 within a SYCL kernel function by the time that the kernel function finishes execution. The point at which this synchronization occurs and the method by which this synchronization is performed are implementation defined. For example it is valid for an implementation to use `printf()`.

In the case that an instance of the SYCL `stream` class is used across multiple work items concurrently, there is no guarantee of ordering of outputs.

If an instance of the SYCL `stream` class is used on a SYCL kernel function executed on a OpenCL context, there is no guarantee that statements are output in their entirety. If an instance of the SYCL `stream` class is used on a SYCL kernel function executed on a host context, then the SYCL `stream` class is required to output each statement in full without mixing with statements of other work items.

Performance note

The usage of the `stream` class is designed for debugging purposes and is therefore not recommended for performance critical applications.

SYCL built-in functions for SYCL host and device

SYCL kernels may execute on any SYCL device, specifically an OpenCL device or SYCL host, which requires the functions used in the kernels to be compiled and linked for both device and host. In the SYCL system the OpenCL built-ins are available for the SYCL host and device within the `cl::sycl` namespace, although, their semantics may be different. This section follows the OpenCL 1.2 specification document [1, ch. 6.12] and describes the behavior of these functions for SYCL host and device.

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.

Description of the built-in types available for SYCL host and device

All of the OpenCL built-in types are available in the namespace *cl::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 Tables 4.107. Each generic type within a section is comprised of a combination of scalar and/or SYCL `vec` class specializations. Note that any reference to the base type refers to the type of a scalar or the element type of a SYCL `vec` specialization.

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the item and group classes see sections 4.8.1.4, 4.8.1.6 and 4.8.1.8.

Generic type name	Description
floatn	float2, float3, float4, float8, float16
genfloatf	float, floatn
doublen	double2, double3, double4, double8, double16
genfloatd	double, doublen
halfn	half2, half3, half4, half8, half16
genfloath	half, halfn
genfloat	genfloatf, genfloatd, genfloath
sgenfloat	float, double, half
gengeofloat	float, float2, float3, float4
gengeodouble	double, double2, double3, double4

Continued on next page

Table 4.107: Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1].

Generic type name	Description
charn	char2, char3, char4, char8, char16
scharn	schar2, schar3, schar4, schar8, schar16
ucharn	uchar2, uchar3, uchar4, uchar8, uchar16
igenchar	signed char, scharn
ugenchar	unsigned char, ucharn
genchar	char, charn, igenchar, ugenchar
shortn	short2, short3, short4, short8, short16
genshort	short, shortn
ushortn	ushort2, ushort3, ushort4, ushort8, ushort16
ugenshort	unsigned short, ushortn
uintn	uint2, uint3, uint4, uint8, uint16
ugenint	unsigned int, uintn
intn	int2, int3, int4, int8, int16
genint	int, intn
ulongn	ulong2, ulong3, ulong4, ulong8, ulong16
ugenlong	unsigned long int, ulongn
longn	long2, long3, long4, long8, long16
genlong	long int, longn
ulonglongn	ulonglong2, ulonglong3, ulonglong4, ulonglong8, ulonglong16
ugenlonglong	unsigned long long int, ulonglongn
longlongn	longlong2, longlong3, longlong4, longlong8, longlong16
genlonglong	long long int, longlongn
igenlonginteger	genlong, genlonglong
ugenlonginteger	ugenlong, ugenlonglong
geninteger	genchar, genshort, ugenshort, genint, ugenint, igenlonginteger, ugenlonginteger
genintegerNbit	All types within geninteger whose base type are N bits in size, where N = 8, 16, 32, 64.
igeninteger	igenchar, genshort, genint, igenlonginteger
igenintegerNbit	All types within igeninteger whose base type are N bits in size, where N = 8, 16, 32, 64.
ugeninteger	ugenchar, ugenshort, ugenint, ugenlonginteger
ugenintegerNbit	All types within ugeninteger whose base type are N bits in size, where N = 8, 16, 32, 64.

Continued on next page

Table 4.107: Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1].

Generic type name	Description
sgeninteger	char, signed char, unsigned char, short, unsigned short, int, unsigned int, long int, unsigned long int, long long int, unsigned long long int
gentype	genfloat, geninteger
genfloatptr	All permutations of multi_ptr<dataT, addressSpace> where dataT is all types within genfloat and addressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space.
genintptr	All permutations of multi_ptr<dataT, addressSpace> where dataT is all types within genint and addressSpace is access::address_space::global_space, access::address_space::local_space and access::address_space::private_space.
	End of table

Table 4.107: Generic type name description, which serves as a description for all valid types of parameters to kernel functions [1].

Work-item functions

In the OpenCL 1.2 specification document [1, ch. 6.12.1] in Table 6.7 the work-item functions are defined where they provide the size of the enqueued kernel NDRange. These functions are available in SYCL through the nd_item and group classes see section 4.8.1.6 and 4.8.1.8.

Math functions

In SYCL the OpenCL math functions are available in the namespace *cl::sycl* on host and device with the same precision guarantees as defined in the OpenCL 1.2 specification document [1, 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* counterparts, for dimensions 1, 2, 3, 4, 8 and 16. On the host the vector types use the *vec* class and on an OpenCL device use the corresponding OpenCL vector types.

The built-in functions available for SYCL host and device, with the same precision requirements for both host and device, are described in Table 4.108.

Math Function	Description
genfloat acos (genfloat x)	Inverse cosine function.
genfloat acosh (genfloat x)	Inverse hyperbolic cosine.
	Continued on next page

Table 4.108: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1].

Math Function	Description
genfloat acospi (genfloat x)	Compute $acosx/\pi$
genfloat asin (genfloat x)	Inverse sine function.
genfloat asinh (genfloat x)	Inverse hyperbolic sine.
genfloat asinpi (genfloat x)	Compute $asinx/\pi$
genfloat atan (genfloat y_over_x)	Inverse tangent function.
genfloat atan2 (genfloat y, genfloat x)	Compute atan(y / x).
genfloat atanh (genfloat x)	Hyperbolic inverse tangent.
genfloat atanpi (genfloat x)	Compute atan (x) / π.
genfloat atan2pi (genfloat y, genfloat x)	Compute atan2 (y, x) / π.
genfloat cbrt (genfloat x)	Compute cube-root.
genfloat ceil (genfloat x)	Round to integral value using the round to positive infinity rounding mode.
genfloat copysign (genfloat x, genfloat y)	Returns x with its sign changed to match the sign of y.
genfloat cos (genfloat x)	Compute cosine.
genfloat cosh (genfloat x)	Compute hyperbolic cosine.
genfloat cospi (genfloat x)	Compute cos (πx).
genfloat erfc (genfloat x)	Complementary error function.
genfloat erf (genfloat x)	Error function encountered in integrating the normal distribution.
genfloat exp (genfloat x)	Compute the base-e exponential of x.
genfloat exp2 (genfloat x)	Exponential base 2 function.
genfloat exp10 (genfloat x)	Exponential base 10 function.
genfloat expm1 (genfloat x)	Compute exp (x) − 1.0.
genfloat fabs (genfloat x)	Compute absolute value of a floating-point number.
genfloat fdim (genfloat x, genfloat y)	$x - y$ if $x > y$,+0 if x is less than or equal to y.
genfloat floor (genfloat x)	Round to integral value using the round to negative infinity rounding mode.
genfloat fma (genfloat a, genfloat b, genfloat c)	Returns the correctly rounded floating-point representation of the sum of c with the infinitely precise product of a and b. Rounding of intermediate products shall not occur. Edge case behavior is per the IEEE 754-2008 standard.
genfloat fmax (genfloat x, genfloat y) genfloat fmax (genfloat x, sgenfloat y)	Returns y if $x < y$, otherwise it returns x. If one argument is a NaN, fmax() returns the other argument. If both arguments are NaNs, fmax() returns a NaN.
genfloat fmin (genfloat x, genfloat y) genfloat fmin (genfloat x, sgenfloat y)	Returns y if $y < x$, otherwise it returns x. If one argument is a NaN, fmin() returns the other argument. If both arguments are NaNs, fmin() returns a NaN.
genfloat fmod (genfloat x, genfloat y)	Modulus. Returns $xy * trunc(x/y)$.

Continued on next page

Table 4.108: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1].

Math Function	Description
`genfloat fract (genfloat x, genfloatptr iptr)`	Returns fmin(x − floor (x), nextafter(genfloat(1.0), genfloat(0.0))). floor(x) is returned in iptr.
`genfloat frexp (genfloat x, genintptr exp)`	Extract mantissa and exponent from x. For each component the mantissa returned is a float with magnitude in the interval [1/2, 1) or 0. Each component of x equals mantissa returned * 2exp.
`genfloat hypot (genfloat x, genfloat y)`	Compute the value of the square root of x2+ y2 without undue overflow or underflow.
`genint ilogb (genfloat x)`	Return the exponent as an integer value.
`genfloat ldexp (genfloat x, genint k)` `genfloat ldexp (genfloat x, int k)`	Multiply x by 2 to the power k.
`genfloat lgamma (genfloat x)`	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.
`genfloat lgamma_r (genfloat x, genintptr signp)`	Log gamma function. Returns the natural logarithm of the absolute value of the gamma function. The sign of the gamma function is returned in the signp argument of lgamma_r.
`genfloat log (genfloat x)`	Compute natural logarithm.
`genfloat log2 (genfloat x)`	Compute a base 2 logarithm.
`genfloat log10 (genfloat x)`	Compute a base 10 logarithm.
`genfloat log1p (genfloat x)`	Compute $loge(1.0 + x)$.
`genfloat logb (genfloat x)`	Compute the exponent of x, which is the integral part of logr (|x|).
`genfloat mad (genfloat a,genfloat b, genfloat c)`	mad approximates a * b + c. Whether or how the product of a * b is rounded and how supernormal or subnormal intermediate products are handled is not defined. mad is intended to be used where speed is preferred over accuracy.
`genfloat maxmag (genfloat x, genfloat y)`	Returns x if $\|x\| > \|y\|$, y if $\|y\| > \|x\|$, otherwise fmax(x, y).
`genfloat minmag (genfloat x, genfloat y)`	Returns x if $\|x\| < \|y\|$, y if $\|y\| < \|x\|$, otherwise fmin(x, y).
`genfloat modf (genfloat x, genfloatptr iptr)`	Decompose a floating-point number. The modf function breaks the argument x into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part in the object pointed to by iptr.
`genfloatf nan (ugenint nancode)` `genfloatd nan (ugenlonginteger nancode)`	Returns a quiet NaN. The nancode may be placed in the significand of the resulting NaN.

Continued on next page

Table 4.108: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1].

Math Function	Description
genfloat nextafter (genfloat x, genfloat y)	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.
genfloat pow (genfloat x, genfloat y)	Compute x to the power y.
genfloat pown (genfloat x, genint y)	Compute x to the power y, where y is an integer.
genfloat powr (genfloat x, genfloat y)	Compute x to the power y, where $x >= 0$.
genfloat remainder (genfloat x, genfloat y)	Compute the value r such that r = x - n*y, where n is the integer nearest the exact value of x/y. If there are two integers closest to x/y, n shall be the even one. If r is zero, it is given the same sign as x.
genfloat remquo (genfloat x, genfloat y, genintptr quo)	The remquo function computes the value r such that r = x - k*y, where k is the integer nearest the exact value of x/y. If there are two integers closest to x/y, k shall be the even one. If r is zero, it is given the same sign as x. This is the same value that is returned by the remainder function. remquo also calculates the lower seven bits of the integral quotient x/y, and gives that value the same sign as x/y. It stores this signed value in the object pointed to by quo.
genfloat rint (genfloat x)	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 [1] for description of rounding modes.
genfloat rootn (genfloat x, genint y)	Compute x to the power 1/y.
genfloat round (genfloat x)	Return the integral value nearest to x rounding halfway cases away from zero, regardless of the current rounding direction.
genfloat rsqrt (genfloat x)	Compute inverse square root.
genfloat sin (genfloat x)	Compute sine.
genfloat sincos (genfloat x, genfloatptr cosval)	Compute sine and cosine of x. The computed sine is the return value and computed cosine is returned in cosval.
genfloat sinh (genfloat x)	Compute hyperbolic sine.
genfloat sinpi (genfloat x)	Compute sin (π x).
genfloat sqrt (genfloat x)	Compute square root.
genfloat tan (genfloat x)	Compute tangent.
genfloat tanh (genfloat x)	Compute hyperbolic tangent.
genfloat tanpi (genfloat x)	Compute tan (π x).
genfloat tgamma (genfloat x)	Compute the gamma function.
	Continued on next page

Table 4.108: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1].

Math Function	Description
`genfloat trunc (genfloat x)`	Round to integral value using the round to zero rounding mode.
	End of table

Table 4.108: Math functions which work on SYCL Host and device. They correspond to Table 6.7 of the OpenCL 1.2 specification [1].

In SYCL the implementation defined precision math functions are defined in the namespace *cl::sycl::native*. The functions that are available within this namespace are specified in Tables 4.109.

Native Math Function	Description
`genfloatf cos (genfloatf x)`	Compute cosine over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf divide (genfloatf x, genfloatf y)`	Compute x / y over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf exp (genfloatf x)`	Compute the base- e exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf exp2 (genfloatf x)`	Compute the base- 2 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf exp10 (genfloatf x)`	Compute the base- 10 exponential of x over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf log (genfloatf x)`	Compute natural logarithm over an implementation defined range. The maximum error is implementation-defined.
`genfloatf log2 (genfloatf x)`	Compute a base 2 logarithm over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf log10 (genfloatf x)`	Compute a base 10 logarithm over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf powr (genfloatf x, genfloatf y)`	Compute x to the power y, where $x >= 0$. The range of x and y are implementation-defined. The maximum error is implementation-defined.
`genfloatf recip (genfloatf x)`	Compute reciprocal over an implementation-defined range. The maximum error is implementation-defined.
`genfloatf rsqrt (genfloatf x)`	Compute inverse square root over an implementation-defined range. The maximum error is implementation-defined.
	Continued on next page

Table 4.109: Native math functions.

Native Math Function	Description
`genfloatf sin (genfloatf x)`	Compute sine over an implementation-defined range. The maximum error is implementation-defined.
`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.
	End of table

Table 4.109: Native math functions.

In SYCL the half precision math functions are defined in *cl::sycl::half_precision*. The functions that are available within this namespace are specified in Tables 4.110. These functions are implemented with a minimum of 10-bits of accuracy i.e. an ULP value is less than or equal to 8192 ulp.

Half Math function	Description
`genfloatf cos (genfloatf x)`	Compute cosine. x must be in the range -216 to +216.
`genfloatf divide (genfloatf x, genfloatf y)`	Compute x / y.
`genfloatf exp (genfloatf x)`	Compute the base- e exponential of x.
`genfloatf exp2 (genfloatf x)`	Compute the base- 2 exponential of x.
`genfloatf exp10 (genfloatf x)`	Compute the base- 10 exponential of x.
`genfloatf log (genfloatf x)`	Compute natural logarithm.
`genfloatf log2 (genfloatf x)`	Compute a base 2 logarithm.
`genfloatf log10 (genfloatf x)`	Compute a base 10 logarithm.
`genfloatf powr (genfloatf x, genfloatf y)`	Compute x to the power y, where $x >= 0$.
`genfloatf recip (genfloatf x)`	Compute reciprocal.
`genfloatf rsqrt (genfloatf x)`	Compute inverse square root.
`genfloatf sin (genfloatf x)`	Compute sine. x must be in the range -216 to +216.
`genfloatf sqrt (genfloatf x)`	Compute square root.
`genfloatf tan (genfloatf x)`	Compute tangent. x must be in the range -216 to +216.
	End of table

Table 4.110: Half precision math functions.

Integer functions

In SYCL the OpenCL integer math functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.3]. 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* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types use the *vec* class and on an OpenCL device use the corresponding OpenCL vector types. The supported integer math functions are described in Table 4.111.

Integer Function	Description
`ugeninteger abs (geninteger x)`	Returns $\|x\|$.
`ugeninteger abs_diff (geninteger x, geninteger y)`	Returns $\|x - y\|$ without modulo overflow.
`geninteger add_sat (geninteger x, geninteger y)`	Returns $x + y$ and saturates the result.
`geninteger hadd (geninteger x, geninteger y)`	Returns $(x+y) >> 1$. The intermediate sum does not modulo overflow.
`geninteger rhadd (geninteger x, geninteger y)`	Returns $(x + y + 1) >> 1$. The intermediate sum does not modulo overflow.
`geninteger clamp (geninteger x, geninteger minval, geninteger maxval)` `geninteger clamp (geninteger x, sgeninteger minval, sgeninteger maxval)`	Returns min(max(x, minval), maxval). Results are undefined if minval > maxval.
`geninteger clz (geninteger x)`	Returns the number of leading 0-bits in x, starting at the most significant bit position.
`geninteger mad_hi (` `geninteger a, geninteger b, geninteger c)`	Returns mul_hi(a, b) + c.
`geninteger mad_sat (geninteger a,` `geninteger b, geninteger c)`	Returns a * b + c and saturates the result.
`geninteger max (geninteger x, geninteger y)` `geninteger max (geninteger x, sgeninteger y)`	Returns y if $x < y$, otherwise it returns x.
`geninteger min (geninteger x, geninteger y)` `geninteger min (geninteger x, sgeninteger y)`	Returns y if $y < x$, otherwise it returns x.
`geninteger mul_hi (geninteger x, geninteger y)`	Computes x * y and returns the high half of the product of x and y.
`geninteger rotate (geninteger v, geninteger i)`	For each element in v, the bits are shifted left by the number of bits given by the corresponding element in i (subject to usual shift modulo rules described in section 6.3). Bits shifted off the left side of the element are shifted back in from the right.
`geninteger sub_sat (geninteger x, geninteger y)`	Returns $x - y$ and saturates the result.
`ugeninteger16bit upsample (ugeninteger8bit hi, ugeninteger8bit lo)`	`result[i] = ((ushort)hi[i] << 8)| lo[i]`
`igeninteger16bit upsample (igeninteger8bit hi, ugeninteger8bit lo)`	`result[i] = ((short)hi[i] << 8)| lo[i]`
`ugeninteger32bit upsample (ugeninteger16bit hi, ugeninteger16bit lo)`	`result[i] = ((uint)hi[i] << 16)| lo[i]`
`igeninteger32bit upsample (igeninteger16bit hi, ugeninteger16bit lo)`	`result[i] = ((int)hi[i] << 16)| lo[i]`
`ugeninteger64bit upsample (ugeninteger32bit hi, ugeninteger32bit lo)`	`result[i] = ((ulonglong)hi[i] << 32)| lo[i]`
`igeninteger64bit upsample (igeninteger32bit hi, ugeninteger32bit lo)`	`result[i] = ((longlong)hi[i] << 32)| lo[i]`
`geninteger popcount (geninteger x)`	Returns the number of non-zero bits in x.

Continued on next page

Table 4.111: Integer functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1].

Integer Function	Description
`geninteger32bit mad24 (geninteger32bit x, geninteger32bit y, geninteger32bit z)`	Multipy two 24-bit integer values x and y and add the 32-bit integer result to the 32-bit integer z. Refer to definition of mul24 to see how the 24-bit integer multiplication is performed.
`geninteger32bit mul24 (geninteger32bit x, geninteger32bit y)`	Multiply two 24-bit integer values x and y. x and y are 32-bit integers but only the low 24-bits are used to perform the multiplication. mul24 should only be used when values in x and y are in the range [- 223, 223-1] if x and y are signed integers and in the range [0, 224-1] if x and y are unsigned integers. If x and y are not in this range, the multiplication result is implementation-defined.
	End of table

Table 4.111: Integer functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.10 of the OpenCL 1.2 specification [1].

Common functions

In SYCL the OpenCL *common* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.4]. They are described here in Table 4.112. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2, 3, 4, 8 and 16. On the host the vector types use the *vec* class and on an OpenCL device use the corresponding OpenCL vector types.

Common Function	Description

`genfloat clamp (genfloat x, genfloat minval, genfloat maxval)`
`genfloatf clamp (genfloatf x, float minval, float maxval)`
`genfloatd clamp (genfloatd x, double minval, double maxval)`
	Returns fmin(fmax(x, minval), maxval). Results are undefined if *minval* > *maxval*.

`genfloat degrees (genfloat radians)`	Converts radians to degrees, i.e.$(180/\pi) * radians$.

`genfloat max (genfloat x, genfloat y)`
`genfloatf max (genfloatf x, float y)`
`genfloatd max (genfloatd x, double y)`
	Returns y if $x < y$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.

`genfloat min (genfloat x, genfloat y)`
`genfloatf min (genfloatf x, float y)`
`genfloatd min (genfloatd x, double y)`
	Returns y if $y < x$, otherwise it returns x. If x or y are infinite or NaN, the return values are undefined.

`genfloat mix (genfloat x, genfloat y, genfloat a)`
`genfloatf mix (genfloatf x, genfloatf y, float a)`
`genfloatd mix (genfloatd x, genfloatd y, double a)`
	Returns the linear blend of x&y implemented as: $x+(y-x)*a$. a must be a value in the range 0.0 ... 1.0. If a is not in the range 0.0 ... 1.0, the return values are undefined.

	Continued on next page

Table 4.112: Common functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1].

Common Function	Description
`genfloat radians (genfloat degrees)`	Converts degrees to radians, i.e. $(\pi/180) * degrees$.

`genfloat step (genfloat edge, genfloat x)`
`genfloatf step (float edge, genfloatf x)`
`genfloatd step (double edge, genfloatd x)`
	Returns 0.0 if $x < edge$, otherwise it returns 1.0.

`genfloat smoothstep (genfloat edge0, genfloat edge1, genfloat x)`
`genfloatf smoothstep (float edge0, float edge1, genfloatf x)`
`genfloatd smoothstep (double edge0, double edge1, genfloatd x)`
	Returns 0.0 if $x <= edge0$ and 1.0 if $x >= edge1$ and performs smooth Hermite interpolation between 0 and 1 when $edge0 < x < edge1$. This is useful in cases where you would want a threshold function with a smooth transition.
	This is equivalent to:
	`gentype t;`
	`t = clamp ((x <= edge0)/ (edge1 >= edge0), 0, 1);`
	`return t * t * (3 - 2 * t);`
	
	Results are undefined if $edge0 >= edge1$ or if x, edge0 or edge1 is a NaN.

`genfloat sign (genfloat x)`	Returns 1.0 if $x > 0$, -0.0 if $x = -0.0$, $+0.0$ if $x = +0.0$, or -1.0 if $x < 0$. Returns 0.0 if x is a NaN.
	End of table

Table 4.112: Common functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.12 of the OpenCL 1.2 specification [1].

Geometric Functions

In SYCL the OpenCL *geometric* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.5]. The built-in functions can take as input float or optionally double and their *vec* counterparts, for dimensions 2, 3 and 4. On the host the vector types use the *vec* class and on an OpenCL device use the corresponding OpenCL vector types. All of the geometric functions use round-to-nearest-even rounding mode. Table 4.113 contains the definitions of supported geometric functions.

Geometric Function	Description

`float4 cross (float4 p0, float4 p1)`
`float3 cross (float3 p0, float3 p1)`
`double4 cross (double4 p0, double4 p1)`
`double3 cross (double3 p0, double3 p1)`
	Returns the cross product of p0.xyz and p1.xyz. The *w* component of float4 result returned will be 0.0.

`float dot (gengeofloat p0, gengeofloat p1)` `double dot (gengeodouble p0, gengeodouble p1)`	Compute dot product.
`float distance (gengeofloat p0, gengeofloat p1)` `double distance (gengeodouble p0, gengeodouble p1)`	Returns the distance between p0 and p1. This is calculated as `length(p0 - p1)`.
`float length (gengeofloat p)` `double length (gengeodouble p)`	Return the length of vector p, i.e., $\sqrt{p.x^2 + p.y^2 + ...}$
	Continued on next page

Table 4.113: Geometric functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1].

Geometric Function	Description
`gengeofloat normalize (gengeofloat p)` `gengeodouble normalize (gengeodouble p)`	Returns a vector in the same direction as p but with a length of 1.
`float fast_distance (gengeofloat p0, gengeofloat p1)`	Returns `fast_length(p0 - p1)`.
`float fast_length (gengeofloat p)`	Returns the length of vector p computed as: `sqrt((half)(pow(p.x,2)+ pow(p.y,2) + ...))`

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

	End of table

Table 4.113: Geometric functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.13 of the OpenCL 1.2 specification [1].

Relational functions

In SYCL the OpenCL *relational* functions are available in the namespace *cl::sycl* on host and device as defined in the OpenCL 1.2 specification document [1, par. 6.12.6]. The built-in functions can take as input char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float or optionally double and their *vec* counterparts, for dimensions 2,3,4,8, and 16. On the host the vector types use the *vec* class and on an OpenCL device use the corresponding OpenCL vector types. The relational operators are available on both host and device. The relational functions are provided in addition to the the operators and will return 0 if the conditional is *false* and 1 otherwise. The available built-in functions are described in Tables 4.114

Relational Function	Description
igeninteger32bit isequal (genfloatf x, genfloatf y) igeninteger64bit isequal (genfloatd x, genfloatd y)	Returns the component-wise compare of $x == y$.
igeninteger32bit isnotequal (genfloatf x, genfloatf y) igeninteger64bit isnotequal (genfloatd x, genfloatd y)	Returns the component-wise compare of $x! = y$.
igeninteger32bit isgreater (genfloatf x, genfloatf y) igeninteger64bit isgreater (genfloatd x, genfloatd y)	Returns the component-wise compare of $x > y$.
igeninteger32bit isgreaterequal (genfloatf x, genfloatf y) igeninteger64bit isgreaterequal (genfloatd x, genfloatd y)	Returns the component-wise compare of $x >= y$.
igeninteger32bit isless (genfloatf x, genfloatf y) igeninteger64bit isless (genfloatd x, genfloatd y)	Returns the component-wise compare of $x < y$.
igeninteger32bit islessequal (genfloatf x, genfloatf y) igeninteger64bit islessequal (genfloatd x, genfloatd y)	Returns the component-wise compare of $x <= y$.
igeninteger32bit islessgreater (genfloatf x, genfloatf y) igeninteger64bit islessgreater (genfloatd x, genfloatd y)	Returns the component-wise compare of $(x < y)\|\|(x > y)$.
igeninteger32bit isfinite (genfloatf x) igeninteger64bit isfinite (genfloatd x)	Test for finite value.
igeninteger32bit isinf (genfloatf x) igeninteger64bit isinf (genfloatd x)	Test for infinity value (positive or negative) .
igeninteger32bit isnan (genfloatf x) igeninteger64bit isnan (genfloatd x)	Test for a NaN.
igeninteger32bit isnormal (genfloatf x) igeninteger64bit isnormal (genfloatd x)	Test for a normal value.
igeninteger32bit isordered (genfloatf x, genfloatf y) igeninteger64bit isordered (genfloatd x, genfloatd y)	Test if arguments are ordered. isordered() takes arguments x and y, and returns the result isequal(x, x) && isequal(y, y).
igeninteger32bit isunordered (genfloatf x, genfloatf y) igeninteger64bit isunordered (genfloatd x, genfloatd y)	Test if arguments are unordered. isunordered() takes arguments x and y, returning non-zero if x or y is NaN, and zero otherwise.
igeninteger32bit signbit (genfloatf x) igeninteger64bit signbit (genfloatd x)	Test for sign bit. The scalar version of the function returns a 1 if the sign bit in the float is set else returns 0. The vector version of the function returns the following for each component in *floatn*: -1 (i.e all bits set) if the sign bit in the float is set else returns 0.

Continued on next page

Table 4.114: Relational functions which work on SYCL Host and device, are available in the cl::sycl namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1].

Relational Function	Description
`int any (igeninteger x)`	Returns 1 if the most significant bit in any component of x is set; otherwise returns 0.
`int all (igeninteger x)`	Returns 1 if the most significant bit in all components of x is set; otherwise returns 0.
`gentype bitselect (gentype a, gentype b, gentype c)`	Each bit of the result is the corresponding bit of a if the corresponding bit of c is 0. Otherwise it is the corresponding bit of b.

`geninteger select (geninteger a, geninteger b, igeninteger c)`
`geninteger select (geninteger a, geninteger b, ugeninteger c)`
`genfloatf select (genfloatf a, genfloatf b, genint c)`
`genfloatf select (genfloatf a, genfloatf b, ugenint c)`
`genfloatd select (genfloatd a, genfloatd b, igeninteger64 c)`
`genfloatd select (genfloatd a, genfloatd b, ugeninteger64 c)`
	For each component of a vector type:
	`result[i] = (MSB of c[i] is set)? b[i] : a[i].`
	For a scalar type:
	`result = c ? b : a.`
	`geninteger` must have the same number of elements and bits as `gentype`.

	End of table

Table 4.114: Relational functions which work on SYCL Host and device, are available in the `cl::sycl` namespace. They correspond to Table 6.14 of the OpenCL 1.2 specification [1].

Vector data load and store functions

The functionality from the OpenCL functions as defined in the OpenCL 1.2 specification document [1, par. 6.12.7] is available in SYCL through the *vec* class in section 4.10.2.

Synchronization Functions

In SYCL the OpenCL *synchronization* functions are available through the `nd_item` class 4.8.1.6, as they are applied to work-items for local or global address spaces. Please see 4.71.

`printf` function

The functionality of the `printf` function is covered by the `stream` class 4.12, 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 [1, par. 6.12.13].

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 a `feature_not_supported` exception.

A SYCL `platform` or SYCL `device` can be queried for extension support by calling the `has_extension` member function.

The level of support for the OpenCL extensions are listed in Table 5.1.

Extension	SYCL interface	OpenCL interoperability interface
`cl_khr_int64_base_atomics`	`Yes`	Yes
`cl_khr_int64_extended_atomics`	`Yes`	Yes
`cl_khr_fp16`	`Yes`	Yes
`cl_khr_3d_image_writes`	`Yes`	Yes
`cl_khr_gl_sharing`	`No`	Yes
`cl_apple_gl_sharing`	`No`	Yes
`cl_khr_d3d10_sharing`	`No`	Yes
`cl_khr_d3d11_sharing`	`No`	Yes
`cl_khr_dx9_media_sharing`	`No`	Yes
		End of table

Table 5.1: SYCL support for OpenCL 1.2 extensions.

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 if any of the above types are used in a SYCL kernel function, executing on a device which does not support the extension `khr_fp16`, the SYCL runtime must throw a `feature_not_supported` exception.

The conversion rules for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [2, par. 9.5.1].

The math functions for half precision types follow the same rules as in the OpenCL 1.2 extensions specification [2, 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 [1].

64 Bit Atomics

The SYCL `atomic` class can support 64 bit data types if the 64 bit extensions `cl_khr_int64_base_atomics` and `cl_khr_int64_extended_atomics` are supported by the SYCL `device` which is executing the SYCL kernel function.

The extension `cl_khr_int64_base_atomics` adds support for the `load`, `store`, `fetch_add`, `fetch_sub`, `exchange` and `compare_exchange_strong` member functions with 64 bit data types.

The extension `cl_khr_int64_extended_atomics` adds support for the `fetch_min`, `fetch_max`, `fetch_and`, `fetch_or` and `fetch_xor` member functions with 64 bit data types.

Writing to 3D image memory objects

The `accessor` class for target `access::target::image` in SYCL support 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*.

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.

SYCL Device Compiler

This section specifies the requirements of the SYCL device compiler. Most features described in this section relate to underlying OpenCL capabilities of target devices and limiting the requirements of device code to ensure portability.

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.

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 the OpenCL host runtime interface.

From this requirement the following rules apply for naming the kernels:

- The kernel name is a *C++ typename*.
- The kernel needs to have a *globally-visible* name. In the case of a named function object type, the name can be the typename of the function object, as long as it is globally-visible. In the case where it isn't, a globally-visible name has to be provided, as template parameter to the kernel invoking interface, as described in 4.8.5. In C++11, lambdas[1] do not have a globally-visible name, so a globally-visible typename has to be provided

[1] C++14 lambdas have the same naming rules as C++11 lambdas.

in the kernel invoking interface, as described in 4.8.5.

- The kernel name has to be a unique identifier in the program.

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 implementation defined. The interface between the compiler and the runtime for extracting and executing SYCL kernels on the device is implementation defined.

Language restrictions for kernels

The extracted SYCL kernels need to be compiled by an OpenCL online or offline compiler and be executed by the OpenCL 1.2 runtime. The extracted kernels need to be OpenCL 1.2 compliant kernels and as such there are certain restrictions that apply to them.

The following restrictions are applied to device functions and kernels:

- Structures containing pointers may be shared but the value of any pointer passed between SYCL devices or between the host and a SYCL device is undefined.
- Memory storage allocation is not allowed in kernels. All memory allocation for the device is done on the host using accessor classes. 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.
- 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 SYCL kernel or any functions called by the kernel.
- No function pointers or references are allowed to be called in a SYCL kernel or any functions called by the kernel.
- RTTI is disabled inside kernels.
- No variadic functions are allowed to be called in a SYCL kernel or any functions called by the kernel.
- Exception-handling cannot be used inside a SYCL kernel or any code called from the kernel. But of course `noexcept` is allowed.
- Recursion is not allowed in a SYCL kernel or any code called from the kernel.
- Variables with thread storage duration are not allowed to be odr-used in kernel code
- Variables with static storage duration that are odr-used inside a kernel must be `const` or `constexpr` and zero-initialized or constant-initialized.
- 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 4.13 to find functions that are specified to be usable in kernels.

- Interacting with a special SYCL runtime class (i.e. SYCL `accessor`, `sampler` or `stream`) that is stored within a C++ union is undefined behavior.

Compilation of functions

The SYCL device compiler parses an entire C++ source file supplied by the user. This also includes C++ header files, using `#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.

In SYCL, kernels are invoked using a kernel invoke function (e.g. `parallel_for`). The kernel invoke functions are templated by their kernel parameter, which is a function object. The code inside the function object that is invoked as a kernel is called the "kernel function". The "kernel function" must always return void. Any function called by the kernel function is compiled for device and called a "device function". Recursively, any function called by a device function is itself compiled as a device function.

For example, this source code shows three functions and a kernel invoke with comments explaining which functions need to be compiled for device.

```
void f ()
{
    // function "f" is not compiled for device

    single_task<class kernel_name>([=] ()
        {
            // This code compiled for device
            g (); // this line forces "g" to be compiled for device
        });
}

void g ()
{
    // called from kernel, so "g" is compiled for device
}

void h ()
{
    // not called from a device function, so not compiled for device
}
```

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 is a function that matches at least the C++11 specification, plus any extensions from the C++14 specification.

Built-in scalar data types

In a SYCL device compiler, the standard C++ fundamental types, including `int`, `short`, `long`, `long long int` need to be configured so that the device definitions of those types match the host definitions of those types. 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.

Fundamental data type	Description
`bool`	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.
`char`	A signed or unsigned 8-bit integer, as defined by the C++11 ISO Standard
`signed char`	A signed 8-bit integer, as defined by the C++11 ISO Standard
`unsigned char`	An unsigned 8-bit integer, as defined by the C++11 ISO Standard
`short int`	A signed integer of at least 16-bits, as defined by the C++11 ISO Standard
`unsigned short int`	An unsigned integer of at least 16-bits, as defined by the C++11 ISO Standard
`int`	A signed integer of at least 16-bits, as defined by the C++11 ISO Standard
`unsigned int`	An unsigned integer of at least 16-bits, as defined by the C++11 ISO Standard
`long int`	A signed integer of at least 32-bits, as defined by the C++11 ISO Standard
`unsigned long int`	An unsigned integer of at least 32-bits, as defined by the C++11 ISO Standard
`long long int`	An integer of at least 64-bits, as defined by the C++11 ISO Standard
`unsigned long long int`	An unsigned integer of at least 64-bits, as defined by the C++11 ISO Standard
`size_t`	An unsigned integer type which is the result of the `sizeof` operator on host.
`float`	A 32-bit floating-point. The float data type must conform to the IEEE 754 single precision storage format.
`double`	A 64-bit floating-point. The double data type must conform to the IEEE 754 double precision storage format.

Continued on next page

Table 6.1: Fundamental data types supported by SYCL.

Fundamental data type	Description
half	A 16-bit floating-point. The half data type must conform to the IEEE 754-2008 half precision storage format. A SYCL feature_not_supported exception must be thrown if the half type is used in a SYCL kernel fucntion which executes on a SYCL device that does not support the extension khr_fp16.
	End of table

Table 6.1: Fundamental data types supported by SYCL.

Preprocessor directives and macros

The standard C++ preprocessing directives and macros are supported.

- CL_SYCL_LANGUAGE_VERSION substitutes an integer reflecting the version number of the SYCL language being supported by the device compiler. The version of SYCL defined in this document will have CL_SYCL_LANGUAGE_VERSION substitute the integer 121;
- __FAST_RELAXED_MATH__ is used to determine if the -cl-fast-relaxed-math optimization option is specified in the build options given to the SYCL device compiler. This is an integer constant of 1 if the option is specified and unspecified otherwise;
- __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_EXTERNAL is an optional macro which enables external linkage of SYCL functions and methods to be included in a SYCL kernel. The macro is only defined if the implementation supports external linkage. For more details see 6.9.1

Attributes

The attribute syntax defined in the OpenCL C specification is supported by the SYCL device compiler. The C++11 attribute specifier can be used in which case these attributes are available in the cl namespace. For instance __attribute__((vec_type_hint(int2))) and [[cl::vec_type_hint(int2)]] are equivalent.

The vec_type_hint, work_group_size_hint and reqd_work_group_size kernel attributes in OpenCL C apply to kernel functions, but this is not syntactically possible in SYCL. In SYCL, these attributes are legal on device functions and their specification is propagated down to any caller of those device functions, such that the kernel attributes are the sum of all the kernel attributes of all device functions called. If there are any conflicts between different kernel attributes, then the behavior is undefined.

Address-space deduction

In SYCL, there are several different types of pointer, or reference:

- Accessors 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 OpenCL address space based on their access mode.
- Explicit pointer classes (e.g. `global_ptr`) contain an OpenCL address space. This allows the compiler to determine whether the pointer references global, local, constant or private memory.
- C++ pointer and reference types (e.g. `int*`) are allowed within SYCL kernels. They can be constructed from the address of local variables, from explicit pointer classes, or from accessors. In all cases, a SYCL device compiler will need to auto-deduce the address space.

Inside kernels, explicit pointer classes and C++ pointers are allowed as long as they reference the same data type and have compatible qualifiers and address spaces.

If a kernel function or device function contains a pointer or reference type, then 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 have the address space of 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. This default assumption is expected to change to be the `generic` address space for OpenCL versions that support the generic address space.

It is illegal to assign a pointer value of one address space to a pointer variable of a different address space.

SYCL offline linking

SYCL functions and methods linkage

The default behavior in SYCL applications is that all the definitions and declarations of the functions and methods are available to the SYCL compiler, in the same translation unit. When this is not the case, all the symbols that need to be exported to a SYCL library or from a C++ library to a SYCL application need to be defined using the macro: `SYCL_EXTERNAL`.

The `SYCL_EXTERNAL` macro will only be defined if the implementation supports offline linking. The macro is implementation-defined, but the following restrictions apply:

- `SYCL_EXTERNAL` can only be used on functions;
- the function cannot use raw pointers as parameter or return types. Explicit pointer classes must be used instead;
- externally defined functions cannot call a `cl::sycl::parallel_for_work_item` method;
- externally defined functions cannot be called from a `cl::sycl::parallel_for_work_group` scope.

The SYCL linkage mechanism is optional and implementation defined.

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 `program` 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 an extern "C" function declarations. The function parameters of these function declarations must be defined as the OpenCL C interoperability aliases; `pointer_t` and `const_pointer_t`, of the `multi_ptr` class template, `vector_t` of the `vec` class template and scalar data type aliases described in Table 4.94.

For example:

```
extern "C" cl::sycl::global_ptr<cl::sycl::cl_int>::pointer_t my_func(
  cl::sycl::cl_float4::vector_t x, cl::sycl::cl_double y);
```

Information Descriptors

The purpose of this chapter is to include all the headers of the Memory Object Descriptors, which are described in detail in Chapter 3, for platform, context, device, and queue.

Platform Information Descriptors

The following interface includes all the information descriptors for the `platform` class as described in Table 4.13.

```
namespace cl {
namespace sycl {
namespace info {
enum class platform : unsigned int {
  profile,
  version,
  name,
  vendor,
  extensions
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

Context Information Descriptors

The following interface includes all the information descriptors for the `context` class as described in Table 4.16.

```
namespace cl {
namespace sycl {
namespace info {
enum class context : int {
  reference_count,
  platform,
  devices
};
}  // info
}  // sycl
}  // cl
```

Device Information Descriptors

The following interface includes all the information descriptors for the `device` class as described in Table 4.20.

```
namespace cl {
namespace sycl {
namespace info {

enum class device : int {
  device_type,
  vendor_id,
  max_compute_units,
  max_work_item_dimensions,
  max_work_item_sizes,
  max_work_group_size,
  preferred_vector_width_char,
  preferred_vector_width_short,
  preferred_vector_width_int,
  preferred_vector_width_long,
  preferred_vector_width_float,
  preferred_vector_width_double,
  preferred_vector_width_half,
  native_vector_width_char,
  native_vector_width_short,
  native_vector_width_int,
  native_vector_width_long,
  native_vector_width_float,
  native_vector_width_double,
  native_vector_width_half,
  max_clock_frequency,
  address_bits,
  max_mem_alloc_size,
  image_support,
  max_read_image_args,
  max_write_image_args,
  image2d_max_height,
  image2d_max_width,
  image3d_max_height,
  image3d_max_width,
  image3d_max_depth,
  image_max_buffer_size,
  image_max_array_size,
  max_samplers,
  max_parameter_size,
  mem_base_addr_align,
  half_fp_config,
  single_fp_config,
  double_fp_config,
  global_mem_cache_type,
  global_mem_cache_line_size,
  global_mem_cache_size,
  global_mem_size,
  max_constant_buffer_size,
```

```
  max_constant_args,
  local_mem_type,
  local_mem_size,
  error_correction_support,
  host_unified_memory,
  profiling_timer_resolution,
  is_endian_little,
  is_available,
  is_compiler_available,
  is_linker_available,
  execution_capabilities,
  queue_profiling,
  built_in_kernels,
  platform,
  name,
  vendor,
  driver_version,
  profile,
  version,
  opencl_c_version,
  extensions,
  printf_buffer_size,
  preferred_interop_user_sync,
  parent_device,
  partition_max_sub_devices,
  partition_properties,
  partition_affinity_domains,
  partition_type_property,
  partition_type_affinity_domain,
  reference_count
};

enum class device_type : unsigned int {
  cpu,
  gpu,
  accelerator,
  custom,
  automatic,
  host,
  all
};

enum class partition_property : int {
  no_partition,
  partition_equally,
  partition_by_counts,
  partition_by_affinity_domain
};

enum class partition_affinity_domain : int {
  not_applicable,
  numa,
  L4_cache,
  L3_cache,
  L2_cache,
```

```
  L1_cache,
  next_partitionable
};

enum class local_mem_type : int { none, local, global };

enum class fp_config : int {
  denorm,
  inf_nan,
  round_to_nearest,
  round_to_zero,
  round_to_inf,
  fma,
  correctly_rounded_divide_sqrt,
  soft_float
};

enum class global_mem_cache_type : int { none, read_only, write_only };

enum class execution_capability : unsigned int {
  exec_kernel,
  exec_native_kernel
};

}  // namespace info
}  // namespace sycl
}  // namespace cl
```

Queue Information Descriptors

The following interface includes all the information descriptors for the queue class as described in Table 4.23.

```
namespace cl {
namespace sycl {
namespace info {
enum class queue : int {
  context,
  device,
  reference_count
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

Kernel Information Descriptors

The following interface includes all the information descriptors for the kernel class as described in Table 4.84.

```
namespace cl {
namespace sycl {
namespace info {
enum class kernel: int {
  function_name,
  num_args,
  context,
  program,
  reference_count,
  attributes
};

enum class kernel_work_group: int {
 global_work_size,
 work_group_size,
 compile_work_group_size,
 preferred_work_group_size_multiple,
 private_mem_size
};

}  // namespace info
}  // namespace sycl
}  // namespace cl
```

Program Information Descriptors

The following interface includes all the information descriptors for the `program` class as described in Table 4.88.

```
namespace cl {
namespace sycl {
namespace info {
enum class program: int {
    context,
    devices,
    reference_count
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

Event Information Descriptors

The following interface includes all the information descriptors for the `event` class as described in Table 4.84 and Table 4.29.

```
namespace cl {
namespace sycl {
```

```
namespace info {
enum class event: int {
  command_execution_status,
  reference_count
};

enum class event_command_status : int {
  submitted,
  running,
  complete
};

enum class event_profiling : int {
  command_submit,
  command_start,
  command_end
};
}  // namespace info
}  // namespace sycl
}  // namespace cl
```

References

[1] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2.19*, 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[2] ——, *The OpenCL Extension Specification, version 1.2.22*, 2012. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2-extensions.pdf

Glossary

accessor An accessor is a class which allows a SYCL kernel function to access data managed by a buffer or image class. Accessors are used to express the dependencies among the different command groups. For the full description please refer to section [4.7.6]. 17, 18, 23, 24, 81, 82, 93, 110

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. 17

async_handler An asynchronous error handler object is a function class instance providing necessary code for handling all the asynchronous exceptions 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 [4.9.2]. 71, 74, 77, 192

barrier A barrier is either a command queue barrier or a kernel execution work-group barrier depending on whether it is a synchronization point on the command queue or on the work-groups of a kernel execution. 29

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]. 18, 23, 33, 110, 259

command queue barrier The SYCL API provides two variants for functions that force synchronization on a SYCL command queue. The `cl::sycl::queue::wait()` and `cl::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. 259

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 [4.8.3]. 164, 165, 175, 259, 260, 262

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. 19, 21, 23, 24, 71, 76, 77, 78, 80, 112, 113, 167, 175, 176, 259

command 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]. 18, 19, 20, 75, 259

command group function object A type which is callable with 'operator()' that takes a reference to acommand group handler, that defines a command group which can be submitted by a command queue. The function object can be a named type, lambda function or `function_class`. 18, 30, 32, 74, 164, 165, 192, 259, 260

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 please see [4.8.2]. 17, 36, 164, 165, 259

constant memory A region of global memory that remains constant during the execution of a kernel. The SYCL runtime allocates and initializes memory objects placed into constant memory. 111, 113

context A SYCL context is an encapsulation of an OpenCL context. In OpenCL, any OpenCL resource is attached to a context. A context contains a collection of devices of the same platform. The context is defined as the `cl::sycl::context` class, for further details please see [4.6.3]. 20, 53, 54

device A SYCL device encapsulates an OpenCL device or the SYCL host device, which can run SYCL kernels on host. 16, 17, 20, 53, 75, 259, 260, 261, 262

device compiler A SYCL device compiler is a compiler that produces OpenCL device binaries from a valid SYCL application. For the full description please refer to section [6]. 17, 34, 35, 245

global memory Global memory is a memory region accessible to all work-items executing on a device. 111, 113, 135

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. 22, 153, 260, 261

group A unique identifier representing a single work-group within the index space of a SYCL kernel execution. Can be one, two or three dimensional. In the SYCL interface a group is represented by the `group` class (see Section 4.8.1.8). 262

host Host is the system that executes the C++ application including the SYCL API. 16, 19, 35, 113, 122, 126, 261

host pointer A pointer to memory on the host. Cannot be accessed directly from a device. 108

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 nd-range. It is one of the index space classes. For the full description please refer to section 4.8.1.3. 151, 153, 160, 260, 261, 262

image Images in SYCL, like buffers, are abstractions of the OpenCL equivalent. As in OpenCL, an image stores a two- or three-dimensional structured array. The SYCL runtime will make available images in OpenCL contexts in order to execute semantically correct kernels in different OpenCL contexts. For the full description please refer to section [4.8.1.3]. 23, 33, 110, 259

index space classes The OpenCL Kernel Execution Model defines an nd-range index space. The SYCL runtime class that defines an nd-range is the `cl::sycl::nd_range`, which takes as input the sizes of global and local work-items, represented using the `cl::sycl::range` class. The kernel library classes for indexing in the defined nd-range are the following classes:

- `cl::sycl::id` : The basic index class representing a id.
- `cl::sycl::item` : The index class that contains the global id and local id.
- `cl::sycl::nd_item` : The index class that contains the global id, local id and the work-group id.
- `cl::sycl::group` : The group class that contains the work-group id and the methods on a work-group.

170

item An item id is an interface used to retrieve the global id, work-group id and local id. For further details see [4.8.1.4]. 150, 156, 157, 262

kernel A SYCL kernel which can be executed on a device, including the SYCL host device. Is created implicitly when defining a SYCL kernel function (See 4.8) but can also be created manually in order to pre-compile SYCL kernel functions. 19, 20, 35, 167, 168, 175, 177, 262

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 [6.2]. 20, 35, 167, 168

kernel scope The function scope of the `operator()` on a SYCL kernel function. Note that any function or method 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 OpenCL devices. The extensions and restrictions are defined in the SYCL device compiler specification. 17

local memory Local memory is a memory region associated with a work-group and accessible only by work-items in that work-group. 110, 111, 122, 173

local id A unique identifier of a work-item among other work-items of a work-group. 22, 153, 260, 261

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. A nd-item is capable of performing work-group barriers and work-group mem-fences. In the SYCL interface a nd-item is represented by the `nd_item` class (seeSection 4.8.1.6). 262

nd-range A representation of the index space of a SYCL kernel execution, the distribution of work-items within into 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 each range within the nd-range is 1 per dimension. In the SYCL interface an nd-range is represented by the `nd_range` class (see Section 4.8.1.2). 22, 153, 160, 168, 172, 261

platform The host together or a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform. A SYCL application can target one or multiple OpenCL platforms provided by OpenCL device vendors [1]. 20, 53, 260

private memory A region of memory private to a work-item. Variables defined in one workitems private memory are not visible to another work-item. [1]. The `cl::sycl::private_memory` class provides access to the work-item's private memory for the hierarchical API as it is described at [4.8.5.3]. 173

program object A program object in SYCL is an OpenCL program object encapsulated in A SYCL class. It contains OpenCL kernels and functions compiled to execute on OpenCL devices. A program object can be generated from SYCL C++ kernels by the SYCL runtime, or obtained from an OpenCL implementation. For furthen details on the `cl::sycl::program` class see [4.8.8]. 20

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 group is represented by the `group` class (see Section 4.8.1.8). 261

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. 14, 34, 245

SYCL File A SYCL C++ source file that contains SYCL API calls. 261

SYCL C++ Template Library The template library is a set of C++ templated classes which provide the programming interface to the SYCL developer. 34

SYCL runtime A SYCL runtime is an implementation of the SYCL API specification. The SYCL runtime manages the different OpenCL platforms, devices, contexts as well as memory handling of data between host and OpenCL contexts to enable semantically correct execution of SYCL programs. 29, 31, 32, 33, 34, 35, 39, 40, 41, 42, 43, 44, 45, 47, 48, 49, 71, 76, 77, 81, 83, 85, 91, 93, 94, 97, 106, 108, 110, 111, 113, 114, 122, 126, 160, 164, 165, 170, 177, 191, 193, 195, 217, 243, 245, 247, 251, 259, 260, 261

SYCL host device The SYCL host device is a native C++ implementation of a device. It does not have an OpenCL `cl_device_id` and it will only appear in the available SYCL devices, as it is not an OpenCL device. It has full SYCL capabilities and reports them through the SYCL information retrieval interface. The SYCL host device is mandatory for every SYCL implementation and is always available, but may not achieve the same performance as an OpenCL CPU device. Any C++ application debugger can be used for debugging SYCL kernels executing on a SYCL host device. 18, 34, 36, 261

SYCL application A SYCL application is a C++ application which uses the SYCL programming model in order to execute kernels on devices. 17, 259, 260, 261

SYCL kernel function A type which is callable with 'operator()' that takes a id, item, nd-item or group which 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 standard layout type or lambda function. 16, 17, 19, 39, 164, 168, 188, 189, 251, 259, 261, 262

work-group mem-fence The work-group mem-fence guarantees that any access on the corresponding memory address space before the barrier, must complete before continuing to process any data from that memory space after the barrier. 27, 151, 153, 159, 160, 171, 172, 217, 262

work-group barrier The work-group barrier, cl::sycl::nd_item::barrier, is a synchronization function within a work-group. All the work-items of a work-group must execute the barrier construct before any work-item

continues execution beyond the barrier. Additionally the work- group barrier performs a work-group mem-fence. Note: In OpenCL 1.2 there is no synchronization between different work- groups. 27, 30, 151, 153, 171, 172, 174, 217, 259

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. 22, 151, 160, 260, 261

work-group The SYCL work-group (`cl::sycl::group` class) is a representation of an OpenCL work group. A collection of related work-items that execute on a single compute unit. The work-items in the group execute the same kernel-instance and share local memory and workgroup functions [1]. For further details for the `cl::sycl::group` class see [4.8.1.8]. 22, 29, 122, 153, 159, 172, 173, 261, 262

work-item The SYCL work-item (`cl::sycl::nd_item` class) is a representation of an OpenCL work item. One of a collection of parallel executions of a kernel invoked on a device by a command. A work-item is executed by one or more processing elements as part of a work-group executing on a compute unit. A work-item is distinguished from other work-items by its global ID or the combination of its work-group ID and its local ID within a work-group [1]. 22, 122, 151, 153, 156, 172, 173, 260, 261, 262