DPC++ Reference

Intel

Apr 25, 2020

CONTENTS

CHAPTER

ONE

INTRODUCTION

Obtaining high compute performance on today's modern computer architectures requires code that is optimized, power efficient, and scalable. The demand for high performance continues to increase due to needs in AI, video analytics, data analytics, as well as in traditional high performance computing (HPC).

Modern workload diversity has resulted in a need for architectural diversity; no single architecture is best for every workload. A mix of scalar, vector, matrix, and spatial (SVMS) architectures deployed in CPU, GPU, AI, and FPGA *accelerators* is required to extract the needed performance.

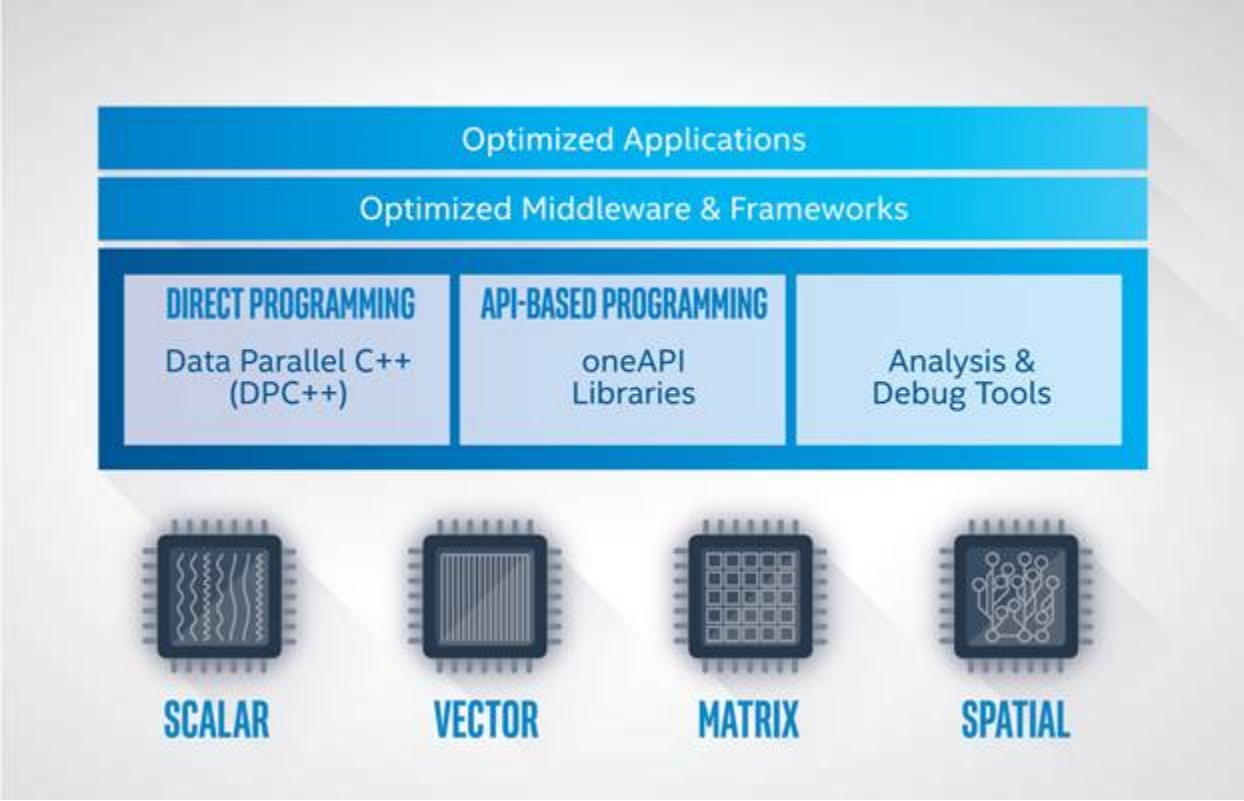
Today, coding for CPUs and accelerators requires different languages, libraries, and tools. That means each hardware platform requires completely separate software investments and provides limited application code reusability across different target architectures.

The oneAPI programming model simplifies the programming of CPUs and accelerators using modern C++ features to express parallelism with a programming language called Data Parallel C++ (DPC++). The DPC++ language enables code reuse for the host (such as a CPU) and accelerators (such as a GPU) using a single source language, with execution and memory dependencies clearly communicated. Mapping within the DPC++ code can be used to transition the application to run on the hardware, or set of hardware, that best accelerates the workload. A host is available to simplify development and debugging of device code, even on platforms that do not have an accelerator available.

Note: Not all programs can benefit from the single programming model offered by oneAPI. It is important to understand if your program can benefit and how to design, implement, and use the oneAPI programming model for your program.

1.1 Overview

The oneAPI programming model provides a comprehensive and unified portfolio of developer tools that can be used across hardware targets, including a range of performance libraries spanning several workload domains. The libraries include functions custom-coded for each target architecture, so the same function call delivers optimized performance across supported architectures. DPC++ is based on industry standards and open specifications to encourage ecosystem collaboration and innovation.



As shown in the figure above, applications that take advantage of the oneAPI programming model can execute on multiple target hardware platforms ranging from CPU to FPGA.

1.2 Data Parallel C++ (DPC++)

Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity. It is based on C++ for broad compatibility and uses common, familiar C and C++ constructs. The language seeks to deliver performance on par with other compiled languages, such as standard C++ compiled code, and uses C++ class libraries to allow the compiler to interpret the code and run on various supported architectures.

DPC++ is based on SYCL* from the Khronos* Group to support data parallelism and heterogeneous programming. In addition, Intel is pursuing extensions to SYCL with the aim of providing value to customer code and working with the standards organization for adoption. For instance, the DPC++ language includes an implementation of unified shared memory to ease memory usage between the host and the accelerators. These features are being driven into a future version of the SYCL language. For more details about SYCL, refer to version 1.2.1 of the SYCL Specification.

While DPC++ applications can run on any supported target hardware, tuning is required to gain the best performance advantage on a given target architecture. For example, code tuned for a CPU likely will not run as fast on a GPU accelerator without modification. This guide aims to help developers understand how to program using the oneAPI programming model and how to target and optimize for the appropriate architecture to achieve optimal application performance.

1.3 Related Documentation

The following documents are useful starting points for developers getting started with oneAPI projects. This document assumes you already have a basic understanding of the oneAPI programming model concepts.

Get Started with Intel oneAPI for Linux*

Get Started with Intel oneAPI for Windows*

Intel oneAPI Base Toolkit Release Notes

SYCL* Specification (for version 1.2.1)

CHAPTER

TWO

PROGRAMMING MODEL

The oneAPI programming model is based upon the SYCL* Specification. The specification presents a general heterogeneous compute capability by detailing four models. These models categorize the actions a developer needs to perform to employ one or more devices as an accelerator. Aspects of these models appear in every program that employs the oneAPI programming model. These models are summarized as:

- *Platform Model*: Specifies the *host* and *device*.
- *Execution Model*: Specifies the *command queue* and issuing commands for execution on the device(s).
- *Memory Model*: Defines how the host and devices interact with memory.
- *Kernel Programming Model*: Defines the code that executes on the device(s). This code is known as the kernel.

The programming language for oneAPI is Data Parallel C++ (DPC++) and employs modern features of the C++ language to enact its parallelism. In fact, when writing programs that employ the oneAPI programming model, the programmer routinely uses language features such as C++ lambdas, templates, parallel_for, and closures.

Tip

If you are unfamiliar with these C++11 and later language features, consult other C++ language references and gain a basic understanding before continuing.

When evaluating and learning oneAPI, keep in mind that the programming model is general enough to accommodate multiple classes of accelerators; therefore, there may be a greater number of API calls required to access the accelerators than more constrained APIs, such as those only accessing one type of accelerator.

One of the primary motivations for DPC++ is to provide a higher-level programming language than OpenCL™ C code, which it is based upon. Readers familiar with OpenCL programs will see many similarities to and differences from OpenCL code. This chapter points out similarities and differences where appropriate. This chapter also points to portions of the SYCL Specification for further information.

2.1 Sample Program

The following code sample contains a program that employs the oneAPI programming model to compute a vector addition. The program computes the formula `c = a + b` across arrays, `a` and `b`, each containing 1024 elements, and places the result in array `c`. The following discussion focuses on sections of code identified by line number in the sample. The intent with this discussion is to highlight the required functionality inherent when employing the programming model.

Note: Keep in mind that this sample code is intended to illustrate the four models that comprise the oneAPI programming model; it is not intended to be a typical program or the simplest in nature.

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

#define SIZE 1024

namespace sycl = cl::sycl;

void show_platforms() {
  auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
              << platform.get_info<sycl::info::platform::name>()
              << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
                << device.get_info<sycl::info::device::name>()
                << std::endl;
    }
  }
}

void vec_add(int *a, int *b, int *c) {
  sycl::range<1> a_size{SIZE};

  sycl::default_selector device_selector;
  sycl::queue q(device_selector);

  sycl::buffer<int, 1>  a_device(a, a_size);
  sycl::buffer<int, 1>  b_device(b, a_size);
  sycl::buffer<int, 1>  c_device(c, a_size);

  q.submit([&](sycl::handler &cgh) {
      auto c_res = c_device.get_access<sycl::access::mode::write>(cgh);
      auto a_in = a_device.get_access<sycl::access::mode::read>(cgh);
      auto b_in = b_device.get_access<sycl::access::mode::read>(cgh);

      cgh.parallel_for<class ex1>(a_size,[=](sycl::id<1> idx) {
          c_res[idx] = a_in[idx] + b_in[idx];
        });
    });
}

int main() {
  int a[SIZE], b[SIZE], c[SIZE];

  for (int i = 0; i<SIZE; ++i) {
    a[i] = i;
    b[i] = -i;
    c[i] = i;
  }

  show_platforms();
  vec_add(a, b, c);

```

(continues on next page)

(continued from previous page)

```
58   return 0;
59 }
```

A DPC++ program has the *single source* property, which means the *host code* and the *device code* can be placed in the same file so that the compiler treats them as the same compilation unit. This can potentially result in performance optimizations across the boundary between host and device code. The single source property differs from a programming model like OpenCL software technology where the host code and device code are typically in different files, and the host and device compiler are different entities, which means no optimization can occur between the host and device code boundary. Therefore, when scrutinizing a DPC++ program, the first step is to understand the delineation between host code and device code. To be more specific, DPC++ programs are delineated into different scopes similar to programming language scope, which is typically expressed via `{` and `}` in many languages.

The three types of scope in a DPC++ program include:

- *application scope*: Code that executes on the host
- *command group scope*: Code that acts as the interface between the host and device
- *kernel scope*: Code that executes on the device

In this example, command group scope comprises lines 45 through 54 and includes coordination and data passing operations required in the program to enact control and communication between the host and the device.

```
45   d_queue.submit([&](sycl::handler &cgh) {
46     auto c_res = c_device.get_access<sycl::access::mode::write>(cgh);
47     auto a_in = a_device.get_access<sycl::access::mode::read>(cgh);
48     auto b_in = b_device.get_access<sycl::access::mode::read>(cgh);
49
50     cgh.parallel_for<class ex1>(a_size,[=](sycl::id<1> idx) {
51       c_res[idx] = a_in[idx] + b_in[idx];
52     });
53
54   });
```

Kernel scope, which is nested in the command group scope, comprises lines 50 to 52. Application scope consists of all the other lines not in command group or kernel scope. Syntactically, definitions are included from the top level include file; `sycl.hpp` and `namespace` declarations can be added for convenience.

The function of each of the lines and its classification into one of the four models are detailed as follows:

- Lines 2 and 6 – `include` and `namespace` – programs employing the oneAPI programming model require the `include` of `cl/sycl.hpp`. It is recommended to employ the `namespace` statement at line 6 to save typing repeated references into the `cl::sycl` namespace.

```
2  #include <CL/sycl.hpp>
3
4  #define SIZE 1024
5
6  namespace sycl = cl::sycl;
```

- Lines 20 to 36 – Platform model – programs employing the oneAPI programming model can query the host for available platforms and can either select one to employ for execution or allow the oneAPI runtime to choose a default platform. A platform defines the relationship between the host and device(s). The platform may have a number of devices associated with it and a program can specify which device(s) to employ for execution or allow the oneAPI runtime to choose a default device.

```
20    auto platforms = sycl::platform::get_platforms();
21
22    for (auto &platform : platforms) {
23
24      std::cout << "Platform: "
25        << platform.get_info<sycl::info::platform::name>()
26        << std::endl;
27
28
29      auto devices = platform.get_devices();
30      for (auto &device : devices ) {
31        std::cout << "  Device: "
32          << device.get_info<sycl::info::device::name>()
33          << std::endl;
34      }
35
36    }
```

- Lines 39 and 45 – Execution model – programs employing the oneAPI programming model define command queues that issue command groups. The command groups control execution on the device.

```
39    sycl::queue d_queue(device_selector);
40
41    sycl::buffer<int, 1>  a_device(a.data(), a_size);
42    sycl::buffer<int, 1>  b_device(b.data(), a_size);
43    sycl::buffer<int, 1>  c_device(c.data(), a_size);
44
45    d_queue.submit([&](sycl::handler &cgh) {
```

- Lines 41 to 43 and lines 46 to 48 – Memory model – programs employing the oneAPI programming model may use buffers and accessors to manage memory access between the host and devices. In this example, the arrays, `a`, `b`, and `c` are defined and allocated on the host. Buffers, `a_device`, `b_device`, and `c_device`, are declared to hold the values from `a`, `b`, and `c` respectively so the device can compute the vector addition. The accessors, `a_in` and `b_in`, denote that `a_device` and `b_device` are to have read only access on the device. The accessor `c_res` denotes that `c_device` is to allow write access from the device.

```
41    sycl::buffer<int, 1>  a_device(a.data(), a_size);
42    sycl::buffer<int, 1>  b_device(b.data(), a_size);
43    sycl::buffer<int, 1>  c_device(c.data(), a_size);
44
45    d_queue.submit([&](sycl::handler &cgh) {
46      auto c_res = c_device.get_access<sycl::access::mode::write>(cgh);
47      auto a_in = a_device.get_access<sycl::access::mode::read>(cgh);
48      auto b_in = b_device.get_access<sycl::access::mode::read>(cgh);
```

- Line 50 to 52 – Kernel Programming Model – The C++ language `parallel_for` statement denotes that the code enclosed in its scope will execute in parallel across the *processing elements* of the device. This example code employs a C++ lambda to represent the kernel.

```
50      cgh.parallel_for<class ex1>(a_size,[=](sycl::id<1> idx) {
51        c_res[idx] = a_in[idx] + b_in[idx];
52      });
```

- Line 17 and 56 – Scope and Synchronization – Memory operations between the buffers and actual host memory

execute in an asynchronous fashion. To ensure synchronization, the command queue is placed inside another scope at line 17 and 56 which tells the runtime to synchronize before the scope is exited as part of the buffer's destructors being executed. This practice is used in many programs.

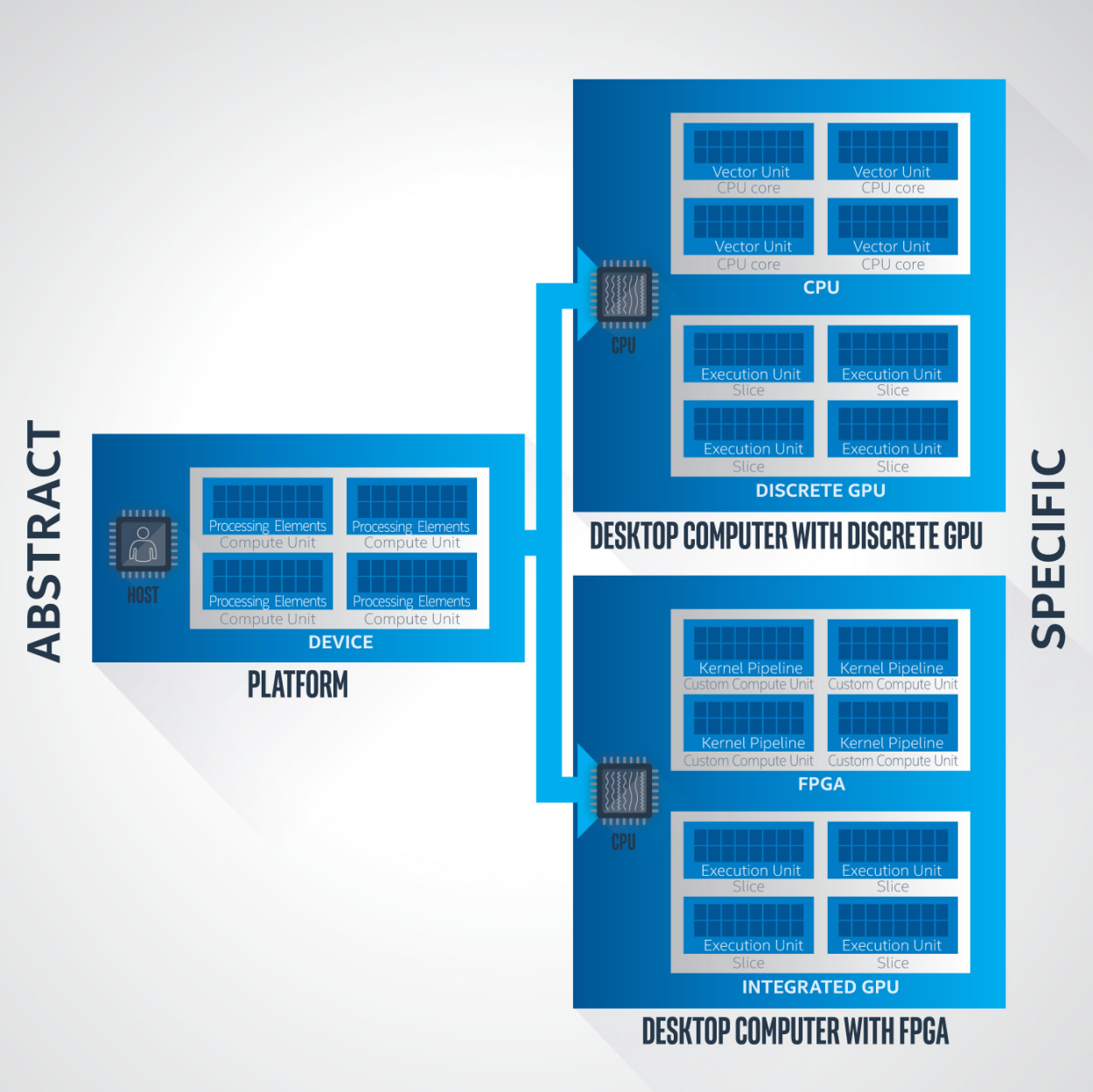
When compiled and executed, the sample program computes the 1024 element vector add in parallel on the accelerator. This assumes the accelerator has multiple compute elements capable of executing in parallel. This sample illustrates the models that the software developer will need to employ in their program. The next sections discuss in more details those four models: the Platform model, the Execution model, the Memory model, and the Kernel model.

2.2 Platform Model

The platform model for oneAPI is based upon the SYCL* platform model. It specifies a host controlling one or more devices. A host is the computer, typically a CPU-based system executing the primary portion of a program, specifically the application scope and the command group scope. The host coordinates and controls the compute work that is performed on the devices. A device is an accelerator, a specialized component containing compute resources that can quickly execute a subset of operations typically more efficiently than the CPUs in the system. Each device contains one or more compute units that can execute several operations in parallel. Each compute unit contains one or more *processing elements* that serve as the individual engine for computation.

A system can instantiate and execute several platforms simultaneously, which is desirable because a particular platform may only target a subset of the available hardware resources on a system. However, in the typical case, a system will have one platform comprised of one or more supported devices, and the compute resources made available by those devices.

The following figure provides a visual depiction of the relationships in the platform model. One host communicates with one or more devices. Each device can contain one or more compute units. Each compute unit can contain one or more processing elements.



The platform model is general enough to be mapped to several different types of devices and lends to the functional portability of the programming model. The hierarchy on the device is also general and can be mapped to several different types of accelerators from FPGAs to GPUs and ASICs as long as these devices support the minimal requirements of the oneAPI programming model. Consult the Intel oneAPI Base Toolkit System Requirements for more information.

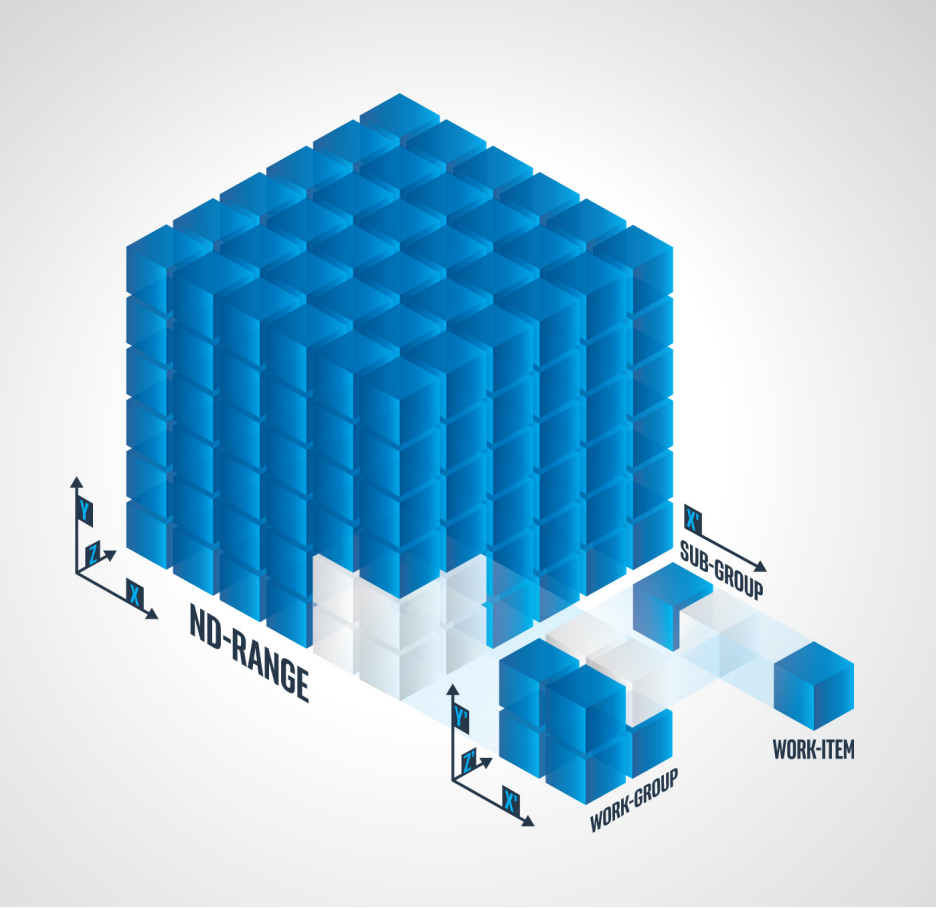
2.3 Execution Model

The execution model is based upon the SYCL* execution model. It defines and specifies how code, termed kernels, execute on the host and the devices.

The host execution model coordinates execution and data management between the host and devices via command groups. The command groups, which are groupings of commands like kernel invocation and accessors, are submitted to queues for execution. Accessors, which are formally part of the memory model, also communicate ordering requirements of execution. A program employing the execution model declares and instantiates queues. Queues can execute with an in-order or out-of-order policy controllable by the program. In-order execution is an Intel extension.

The device execution model specifies how computation is accomplished on the accelerator. Compute ranging from small one-dimensional data to large multidimensional data sets are allocated across a hierarchy of *nd-range*, *work-group*, *sub-group*, and *work-item*, which are all specified when the work is submitted to the command queue. It is important to note that the actual kernel code represents the work that is executed for one work-item. The code outside of the kernel controls just how much parallelism is executed; the amount and distribution of the work is controlled by specification of the sizes of the ND-range and work-group.

The following figure depicts the relationship between an ND-range, work-group, sub-group, and work-item. The total amount of work is specified by the ND-range size. The grouping of the work is specified by the work-group size. The example shows the ND-range size of X * Y * Z, work-group size of X' * Y' * Z', and subgroup size of X'. Therefore, there are X * Y * Z work-items. There are (X * Y * Z) / (X' * Y' * Z') work-groups and (X * Y * Z) / X' subgroups.



When kernels are executed, the location of a particular work-item in the larger ND-range, work-group, or sub-group is important. For example, if the work-item is assigned to compute on specific pieces of data, a method of specification is necessary. Unique identification of the work-item is provided via intrinsic functions such as those in the `nd_item` class (`global_id`, `work_group_id`, and `local_id`).

The following code sample launches a kernel and displays the relationships of the previously discussed `ND-range`, `work-group`, and `work-item`.

```
#include<CL/sycl.hpp>
#include<iostream>
#define N 6
#define M 2
using namespace cl::sycl;
int main()
{
   queue defaultqueue;
   buffer<int,2> buf(range<2>(N,N));
 defaultqueue.submit([&](handler &cgh){
   auto bufacc = buf.get_access<access::mode::read_write>(cgh);
   cgh.parallel_for<class ndim>(nd_range<2>(range<2>(N,N),
     range<2>(M,M)), [=](nd_item<2> i){
     id<2> ind = i.get_global_id();
     bufacc[ind[0]][ind[1]] = ind[0]+ind[1];
   });
 });
 auto bufacc1 = buf.get_access<access::mode::read>();
 for(int i = 0; i < N; i++){
   for(int j = 0; j < N; j++)
     std::cout<<bufacc1[i][j]<<"\t";
   std::cout<<"\n";
 }
 return 0;
}
```

ND-Range Parallelism Example

The following discusses the relationships in the use of the ND-range in the previous code sample.

- At line 12 is the `nd-range` declaration. `nd_range<2>` specifies a two-dimensional index space.
- The global range is specified by the first argument, `range<2>(N,N)`, which specifies the overall index space as 2 dimensions with size `N` by `N`.
- The second argument, `range<2>(M,M)` specifies the local work-group range as 2 dimensions with size `M` by `M`.
- Line 13 employs `nd_item<2>` to reference each work-item in the ND-range, and calls `get_global_id` to determine the index in the global buffer, `bufacc`.

The `sub-group` is an extension to the SYCL execution model and sits hierarchically between the `work_group` and `work_item`. The `sub_group` was created to align with typical hardware resources that contain a vector unit to execute several similar operations in parallel and in lock step.

2.4 Memory Model

The memory model for oneAPI is based upon the SYCL* memory model. It defines how the host and devices interact with memory. It coordinates the allocation and management of memory between the host and devices. The memory model is an abstraction that aims to generalize across and be adaptable to the different possible host and device configurations. In this model, memory resides upon and is owned by either the host or the device and is specified by declaring a memory object. There are two different types of memory objects, *buffers* and *images*. Interaction of these memory objects between the host and device is accomplished via an *accessor*, which communicates the desired location of access, such as host or device, and the particular mode of access, such as read or write.

Consider a case where memory is allocated on the host through a traditional malloc call. Once the memory is allocated on the host, a buffer object is created, which enables the host allocated memory to be communicated to the device. The `buffer` class communicates the type and number of items of that type to be communicated to the device for computation. Once a buffer is created on the host, the type of access allowed on the device is communicated via an `accessor` object, which specifies the type of access to the buffer. The general steps are summarized as:

1. Instantiate a `buffer` or `image` object.

   The host or device memory for the `buffer` or `image` is allocated as part of the instantiation or is wrapped around previously allocated memory on the host.

2. Instantiate an `accessor` object.

   The `accessor` specifies the required location of access, such as host or device, and the particular mode of access, such as read or write. It represents dependencies between uses of memory objects.

The following code sample is exercising different memory objects and accessors.

```
#include <vector>
#include <CL/sycl.hpp>
namespace sycl = cl::sycl;


#define SIZE 64


int main() {
  std::array<int, SIZE> a, c;
  std::array<sycl::float4, SIZE> b;
  for (int i = 0; i<SIZE; ++i) {
    a[i] = i;
    b[i] = (float)-i;
     c[i] = i;
  }


  {
  sycl::range<1> a_size{SIZE};


  sycl::queue d_queue;


  sycl::buffer<int>  a_device(a.data(), a_size);
  sycl::buffer<int>  c_device(c.data(), a_size);
  sycl::image<2>  b_device(b.data(),sycl::image_channel_order::rgba,
    sycl::image_channel_type::fp32, sycl::range<2>(8, 8));
```

(continues on next page)

(continued from previous page)

```
  d_queue.submit([&](sycl::handler &cgh) {
    sycl::accessor<int, 1, sycl::access::mode::discard_write,
      sycl::access::target::global_buffer> c_res(c_device, cgh);
    sycl::accessor<int, 1, sycl::access::mode::read,
      sycl::access::target::constant_buffer> a_res(a_device, cgh);
    sycl::accessor<sycl::float4, 2, sycl::access::mode::write,
      sycl::access::target::image> b_res(b_device, cgh);


    sycl::float4 init = {0.f, 0.f, 0.f, 0.f};


    cgh.parallel_for<class ex1>(a_size,[=](sycl::id<1> idx) {
      c_res[idx] = a_res[idx];
      b_res.write(sycl::int2(0,0), init);
    });


  });


  }
  return 0;
}
```

- Lines 8 and 9 contain the host allocations of arrays `a`, `b`, & `c`. The declaration of `b` is as a `float4` because it will be accessed as an image on the device side.
- Lines 27 and 28 create an accessor for `c_device` that has an access mode of `discard_write` and a target of `global_buffer`.
- Lines 29 and 30 create an accessor for `a_device` that has an access mode of `read` and a target of `constant_buffer`.
- Lines 31 and 32 create an accessor for `b_device` that has an access mode of `write` and a target of `image`.

The accessors specify where and how the kernel will access these memory objects. The runtime is responsible for placing the memory objects in the correct location. Therefore, the runtime may copy data between host and device to meet the semantics of the accessor target.

Designate accessor targets to optimize the locality of access for a particular algorithm. For example, specify that local memory should be employed if much of the kernel access would benefit from memory that resides closer to the processing elements.

If the kernel attempts to violate the communicated accessor by either attempting to write on a read accessor or read on a write accessor, a compiler diagnostic is emitted. Not all combinations of access targets and access modes are compatible. For details, see the SYCL Specification.

2.4.1 Memory Objects

Memory objects are either buffers or images.

- Buffer object - a one-, two-, or three-dimensional array of elements. Buffers can be accessed via lower level C++ pointer types. For further information on buffers, see the SYCL Specification.
- Image object - a formatted opaque memory object stored in a type specific and optimized fashion. Access occurs through built-in functions. Image objects typically pertain to pictures comprised of pixels stored in a format such as RGB (red, green, blue intensity). For further information on images, see the SYCL Specification.

2.4.2 Accessors

Accessors provide access to buffers and images in the host or inside the kernel and also communicate data dependencies between the application and different kernels. The accessor communicates the data type, the size, the target, and the access mode. To enable good performance, pay particular attention to the target because the accessor specifies the memory type from the choices in the SYCL memory model.

The targets associated with buffers are:

- global_buffer
- host_buffer
- constant_buffer
- local

The targets associated with images are:

- image
- host_image
- image_array

Image access must also specify a channel order to communicate the format of the data being read. For example, an image may be specified as a `float4`, but accessed with a channel order of RGBA.

The access mode impacts correctness as well as performance and is one of `read`, `write`, `read_write`, `discard_write`, `discard_read_write`, or `atomic`. Mismatches in access mode and actual memory operations such as a `write` to a `buffer` with access mode `read` can result in compiler diagnostics as well as erroneous program state. The `discard_write` and `discard_read_write` access modes can provide performance benefits for some implementations. For further details on accessors, see the SYCL Specification.

2.4.3 Synchronization

It is possible to access a `buffer` without employing an `accessor`, however it should be the rare case. To do so safely, a `mutex_class` should be passed when a `buffer` is instantiated. For further details on this method, see the SYCL Specification.

Access Targets

Target	Description
`host_buffer`	Access the buffer on the host.
`global_buffer`	Access the buffer through global memory on the device.
`constant_buffer`	Access the buffer from constant memory on the device. This may enable some optimization.
`local`	Access the buffer from local memory on the device.
`image`	Access the image
`image_array`	Access an array of images
`host_image`	Access the image on the host.

Access Modes

Memory Access Mode	Description
`read`	Read-only
`write`	Write-only
`read_write`	Read and write
`discard_write`	Write-only access. Previous value is discarded
`discard_read_write`	Read and write. Previous value is discarded
`atomic`	Provide atomic, one at a time, access.

2.4.4 Unified Shared Memory

An extension to the standard SYCL memory model is unified shared memory, which enables the sharing of memory between the host and device without explicit accessors in the source code. Instead, manage access and enforces dependencies with explicit functions to wait on events or signaling a `depends_on` relationship between events.

Another characteristic of unified shared memory is that it provides a C++ pointer-based alternative to the buffer programming model. Unified shared memory provides both explicit and implicit models for managing memory. In the explicit model, programmers are responsible for specifying when data should be copied between memory allocations on the host and allocations on a device. In the implicit model, the underlying runtime and device drivers are responsible for automatically migrating memory between the host and a device. Since unified shared memory does not rely on accessors, dependencies between operations must be specified using events. Programmers may either explicitly wait on event objects or use the `depends_on` method inside a command group to specify a list of events that must complete before a task may begin.

2.4.5 Memory Types

Memory Type	Description
Constant Memory	A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
Global Memory	Accessible to all work-items in all work-groups. Read/write, may be cached, persistent across kernel invocations.
Local Memory	Shared between work-items in a single work-group and inaccessible to work-items in other work-groups. Example: Shared local memory on Intel HD Graphics 530
Private Memory	A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item. Example: Register File on Intel HD Graphics 530

2.5 Kernel Programming Model

The kernel programming model for oneAPI is based upon the SYCL* kernel programming model. It enables explicit parallelism between the host and device. The parallelism is explicit in the sense that the programmer determines what code executes on the host and device; it is not automatic. The kernel code executes on the accelerator. Programs employing the oneAPI programming model support single source, meaning the host code and device code can be in the same source file. However, there are differences between the source code accepted in the host code and the device code with respect to language conformance and language features. The SYCL Specification defines in detail the required language features for host code and device code. The following is a summary that is specific to the oneAPI product.

2.5.1 C++ Language Requirements

The host code can be compiled by C++11 and later compilers and take advantage of supported C++11 and later language features. The device code requires a compiler that accepts all C++03 language features and the following C++11 features:

- Lamdba expressions
- Variadic templates
- Alias templates
- rvalue references
- std::function, std::string, std::vector

In addition, the device code cannot use the following features:

- Virtual Functions
- Virtual Inheritance
- Exceptions handling – throws and catches between host and device
- Run Time Type Information (RTTI)
- Object management employing new and delete operators

The device code is specified via one of three language constructs: lambda expression, functor, or kernel class. The separation of host code and device code via these language constructs is natural and accomplished without language extensions. These different forms of expressing kernels give the developer flexibility in enmeshing the host code and device code. For example:

- To put the kernel code in line with the host code, consider employing a lambda expression.
- To have the device code separate from the host code, but still maintain the single source property, consider employing a functor.
- To port code from OpenCL programs or to ensure a more rigid separation between host and device code, consider employing the kernel class.

The Device code inside a lambda expression, functor, or kernel object can then specify the amount of parallelism to request through several mechanisms.

- `single_task` – execute a single instance of the kernel with a single work item.
- `parallel_for` – execute a kernel in parallel across a range of processing elements. Typically, this version of parallel_for is employed on "embarrassingly parallel" workloads.
- `parallel_for_work_group` – execute a kernel in parallel across a hierarchical range of processing elements using local memory and barriers.

The following code sample shows two combinations of invoking kernels:

1. single_task and C++ lambda (lines 33-40)
2. parallel_for and functor (lines 8-20 and line 50)

These constructs enclose the aforementioned kernel scope. For details, see the SYCL Specification.

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


#define SIZE 1024


namespace sycl = cl::sycl;


template <typename T> class Vassign {
  T val;
  sycl::accessor<T, 1, sycl::access::mode::read_write,
                 sycl::access::target::global_buffer> access;


public:
  Vassign(T val_, sycl::accessor<T, 1, sycl::access::mode::read_write,
          sycl::access::target::global_buffer> &access_) : val(val_),
          access(access_) {}


  void operator()(sycl::id<1> id) { access[id] = 1; }


};


int main() {
  std::array<int, SIZE> a;


  for (int i = 0; i<SIZE; ++i) {
    a[i] = i;
  }
  {
  sycl::range<1> a_size{SIZE};
  sycl::buffer<int, 1>  a_device(a.data(), a_size);
  sycl::queue d_queue;


  d_queue.submit([&](sycl::handler &cgh) {
    auto a_in = a_device.get_access<sycl::access::mode::write>(cgh);


    cgh.single_task<class ex1>([=]() {
      a_in[0] = 2;
    });
  });
  }
```

(continues on next page)

(continued from previous page)

```
  {
  sycl::range<1> a_size{SIZE};
  sycl::buffer<int, 1>  a_device(a.data(), a_size);
  sycl::queue d_queue;
  d_queue.submit([&](sycl::handler &cgh) {
    auto a_in = a_device.get_access<sycl::access::mode::read_write,
                  sycl::access::target::global_buffer>(cgh);
    Vassign<int> F(0, a_in);
    cgh.parallel_for(sycl::range<1>(SIZE), F);
  });
  }
}
```

2.5.2 Error Handling

C++ exception handling is the basis for handling error conditions in the programming model. Some restrictions on exceptions are in place due to the asynchronous nature of host and device execution. For example, it is not possible to throw an exception in kernel scope and catch it (in the traditional sense) in application scope. Instead, there are a set of restrictions and expectations in place when performing error handling. These include:

- At application scope, the full C++ exception handling mechanisms and capability are valid as long as there is no expectation that exceptions can cross to kernel scope.
- At the command group scope, exceptions are asynchronous with respect to the application scope. During command group construction, an `async_handler` can be declared to handle any exceptions occurring during execution in the command group.

For further details on error handling, see the SYCL Specification.

2.5.3 Fall Back

Typically, a command group is submitted and executed on the designated command queue; however, there may be cases where the command queue is unable to execute the group. In these cases, it is possible to specify a fall back command queue for the command group to be executed upon. This capability is handled by the runtime. This fallback mechanism is detailed in the SYCL Specification.

The following code fails due to the size of the workgroup when executed on Intel Processor Graphics, such as Intel HD Graphics 530. The SYCL specification allows specifying a secondary queue as a parameter to the submit function and this secondary queue is used if the device kernel runs into issues with submission to the first device.

```
#include<CL/sycl.hpp>
#include<iostream>
#define N 1024
#define M 32
using namespace cl::sycl;
int main(){
{
  cpu_selector cpuSelector;
  queue cpuQueue(cpuSelector);
queue defaultqueue;
buffer<int,2> buf(range<2>(N,N));
defaultqueue.submit([&](handler &cgh){
  auto bufacc = buf.get_access<access::mode::read_write>(cgh);
```

(continues on next page)

(continued from previous page)

```
  cgh.parallel_for<class ndim>(nd_range<2>(range<2>(N,N),
    range<2>(M,M)), [=](nd_item<2> i){
    id<2> ind = i.get_global_id();
    bufacc[ind[0]][ind[1]] = ind[0]+ind[1];
  });
},cpuQueue);
auto bufacc1 = buf.get_access<access::mode::read>();
for(int i = 0; i < N; i++){
  for(int j = 0; j < N; j++){
    if(bufacc1[i][j] != i+j){
      std::cout<<"Wrong result\n";
      return 1;
    }
  }
}
std::cout<<"Correct results\n";
return 0;
}
}
```

CHAPTER

THREE

LANGUAGE

Todo: C++ version mininum

Programming Model and subsections documented the C++ language features accepted in code at application scope and command group scope in a DPC++ program. Application scope and command group scope includes the code that executes on the host. That section also documented the C++ language features accepted in code at kernel scope in a DPC++ program. Kernel scope is the code that executes on the device. In general, the full capabilities of C++ are available at application and command group scope. At kernel scope there are limitations in accepted C++ due to the more limited, but focused, capabilities of accelerators.

Compilers from different vendors have small eccentricities or differences in their conformance to the C++ standard. The Intel oneAPI DPC++ Compiler is a LLVM-based compiler and therefore drafts the specific behavior of the LLVM-based compilers in accepting and creating executables from C++ source code. To determine the specific LLVM version that the Intel oneAPI DPC++ Compiler is based upon, use the `--version` option.

```
dpcpp --version
```

For example:

```
DPC++ Compiler 2021.1 (2019.8.x.0.1010)
Target: x86_64-pc-windows-msvc
Thread model: posix
InstalledDir: c:\PROGRA~2\INTELO~1\compiler\latest\windows\bin
```

3.1 Keywords

One of the design goals of DPC++ and SYCL is to not add keywords to the language. The motivation is to enable easier compiler vendor adoption. Whereas OpenCL C code and other accelerator-targeted languages require proprietary keywords, DPC++ does not.

3.2 Preprocessor Directives and Macros

Standard C++ preprocessing directives and macros are supported by the compiler. In addition, the SYCL Specification defines the SYCL specific preprocessor directives and macros.

The following preprocessor directives and macros are supported by the compiler.

Directive	Description
`SYCL_DUMP_IMAGES`	If true, instructs runtime to dump the device image
`SYCL_USE_KERNEL_SPV=<device binary>`	Employ device binary to fulfill kernel launch request
`SYCL_PROGRAM_BUILD_OPTIONS`	Used to pass additional options for device program building.

3.3 Standard Library Classes Required for the Interface

Programming for oneAPI employs a variety of vectors, strings, functions, and pointer objects common in STL programming.

The SYCL specification documents a facility to enable vendors to provide custom optimized implementations. Implementations require aliases for several STL interfaces. These are summarized as follows:

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

Template Parameters

- **T** –
- **allocatorT** –

CHAPTER

FOUR

PROGRAMMING INTERFACE

The Data Parallel C++ (DPC++) programming language and runtime consists of a set of C++ classes, templates, and libraries used to express a DPC++program. This chapter provides a summary of the key classes, templates, and runtime libraries used to program.

4.1 Header File

4.2 Namespaces

4.3 Class Descriptions

The following sections provide further details on these items. These sections do not provide the exhaustive details found in the SYCL Specification. Instead, these sections provide:

- A summary that includes a description and the purpose
- Comments on the different constructors, if applicable
- Member function information, if applicable
- Special cases to consider with the DPC++ implementation compared to the SYCL Specification

For further details on SYCL, see the SYCL Specification.

4.3.1 Runtime Classes

4.3.1.1 `device_selector`

```
class device_selector();
```

Member functions

(constructor)	
(destructor)	
select_device	

Non-member functions

operator()	

(constructor)

```
device_selector(const device_selector &rhs);
device_selector &operator=(const device_selector &rhs);
```

(destructor)

```
virtual ~device_selector();
```

select_device

```
device select_device() const;
```

Return value

operator()

```
virtual int operator()(const device &device) const = 0;
```

Return value

4.3.1.2 platform

```
class platform;
```

Abstraction for SYCL platform.

Member functions

(constructor)	constructs a platform
(destructor)	destroys a platform
get	returns OpenCL platform ID
get_devices	returns devices bound to the platform
get_info	queries properties
has_extension	checks if platform has an extension
is_host	checks if platform has a SYCL host device

Non-member functions

get_platforms	returns platforms bound to the system

Example

Demonstrates several methods for `platform`

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

namespace sycl = cl::sycl;

int main() {
  auto platforms = sycl::platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
              << platform.get_info<sycl::info::platform::name>()
              << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
                << device.get_info<sycl::info::device::name>()
                << std::endl;
    }
  }

  return 0;
}
```

Output:

```
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
  Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: Intel(R) CPU Runtime for OpenCL(TM) Applications
  Device: Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz
Platform: SYCL host platform
  Device: SYCL host device
```

(constructor)

```
platform();
explicit platform(cl_platform_id platformID); [1]
explicit platform(const device_selector &deviceSelector); [2]
```

Constructs a platform handle.

Arguments

platformID	OpenCL platform ID
deviceSelector	Platform must contain the selected device

get

```
cl_platform_id get() const;
```

Returns OpenCL platform id used in the constructor.

get_devices

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

Returns vector of devices of the requested type

Arguments

device_type	limits type of device returned

Return value

`vector` containing devices of the specified type bound to the platform.

Example

See *Example*.

[1] Constructs a SYCL platform that retains an OpenCL id
[2] Selects a platform that contains the desired device

get_info

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

Returns information about the platform, as specified by `param`.

Return value

Requested information

Example

See *Example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Checks if the platform has the requested extension.

Arguments

extension	

Return value

`true` if the platform has `extension`

is_host

```
bool is_host() const;
```

Checks if the platform contains a SYCL *host device*

Return value

`true` if the platform contains a host device

get_platforms

```
static vector_class<platform> get_platforms();
```

Returns vector of platforms

Return value

vector_class containing SYCL platforms bound to the system

Example

See *Example*.

4.3.1.3 context

```
class context;
```

Member functions

(constructor)	constructs a context
get	returns OpenCL conext ID
is_host	checks if contains a SYCL host device
get_platform	
get_devices	returns devices bound to the context
get_info	queries properties

(constructor)

```
explicit context(const property_list &propList = {});
context(async_handler asyncHandler,
        const property_list &propList = {});
context(const device &dev, const property_list &propList = {});
context(const device &dev, async_handler asyncHandler,
        const property_list &propList = {});
context(const platform &plt, const property_list &propList = {});
context(const platform &plt, async_handler asyncHandler,
        const property_list &propList = {});
context(const vector_class<device> &deviceList,
        const property_list &propList = {});
context(const vector_class<device> &deviceList,
        async_handler asyncHandler, const property_list &propList = {});
context(cl_context clContext, async_handler asyncHandler = {});
```

Arguments

propList	
asyncHandler	
dev	
plt	
deviceList	

get

```
cl_context get() const;
```

Return value

is_host

```
bool is_host() const;
```

Return value

get_platform

```
platform get_platform() const;
```

Return value

get_devices

```
vector_class<device> get_devices() const;
```

Return value

get_info

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

Return value

4.3.1.4 device

```
class device;
```

Member functions

(constructor)	
(destructor)	
get	
is_host	
is_cpu	
is_gpu	
is_accelerator	
get_platform	
get_info	
has_extension	
create_sub_devices	

Non-member functions

get_devices	

(constructor)

```
device();
explicit device(cl_device_id deviceId);
explicit device(const device_selector &deviceSelector);
```

Arguments

deviceID	
deviceSelector	

get

```
cl_device_id get() const;
```

Return value

is_host

```
bool is_host() const;
```

Return value

is_cpu

```
bool is_cpu() const;
```

Return value

is_gpu

```
bool is_gpu() const;
```

Return value

is_accelerator

```
bool is_accelerator() const;
```

Return value

get_platform

```
platform get_platform() const;
```

Return value

get_info

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

Return value

Example

See *Example*.

has_extension

```
bool has_extension(const string_class &extension) const;
```

Arguments

extension	

Return value

create_sub_devices

```
template <info::partition_property prop>
vector_class<device> create_sub_devices(size_t nbSubDev) const; [4]

template <info::partition_property prop>
vector_class<device> create_sub_devices(const vector_class<size_t> &counts)␣
↪const; [5]

template <info::partition_property prop>
vector_class<device> create_sub_devices(info::affinity_domain affinityDomain)␣
↪const; [6]
```

Arguments

nbSubDev	
counts	
affinityDomain	

[4] Available only when prop == info::partition_property::partition_equally
[5] Available only when prop == info::partition_property::partition_by_counts
[6] Available only when prop == info::partition_property::partition_by_affinity_domain

Return value

get_devices

```
static vector_class<device> get_devices(
    info::device_type deviceType = info::device_type::all);
```

Return value

4.3.1.5 queue

```
class queue;
```

Member functions

(constructor)	
(destructor)	
get	
get_context	
get_device	
get_info	
is_host	
submit	
wait	
wait_and_throw	
throw_asynchronous	

(constructor)

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

(continues on next page)

(continued from previous page)

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

get

```
cl_command_queue get() const;
```

Return value

get_context

```
context get_context() const;
```

Return value

get_device

```
device get_device() const;
```

Return value

is_host

```
bool is_host() const;
```

Return value

get_info

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

Return value

submit

```
template <typename T>
event submit(T cgf);

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

Arguments

cgf	
secondaryQueue	

Return value

wait

```
void wait();
```

wait_and_throw

```
void wait_and_throw();
```

throw_asynchronous

```
void throw_asynchronous();
```

4.3.1.6 event

```
class event;
```

Member functions

(constructor)	
(destructor)	
cl_event_get	
is_host	
get_wait_list	
wait	
wait_and_throw	
get_info	
get_profiling_info	

(constructor)

```
event();
event(cl_event clEvent, const context& syclContext);
```

cl_event_get

```
cl_event get();
```

Return value

is_host

```
bool is_host() const;
```

Return value

get_wait_list

```
vector_class<event> get_wait_list();
```

Return value

wait

```
void wait();

static void wait(const vector_class<event> &eventList);
```

wait_and_throw

```
void wait_and_throw();

static void wait_and_throw(const vector_class<event> &eventList);
```

get_info

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

Return value

get_profiling_info

```
template <info::event_profiling param>
typename info::param_traits<info::event_profiling, param>::return_type get_profiling_
↪info() const;
```

Return value

4.3.2 Data Access

4.3.2.1 accessor

```
template<
    typename dataT,
    int dimensions,
    access::mode accessmode,
    access::target accessTarget = access::target::global_buffer,
    access::placeholder isPlaceholder = access::placeholder::false_t
> accessor;
```

A DPC++ `accessor` encapsulates reading and writing memory objects which can be buffers, images, or device local memory. Creating an accessor requires a method to reference the desired access target. Construction also requires the type of the memory object, the dimensionality of the memory object, the access mode, and a placeholder argument.

Template parameters

`dataT`	type of buffer element
`dimensions`	dimensionality of buffer
`accessmode`	type of access
`accessTarget`	type of memory
`isPlaceholder`	placeholder

Member types

`value_type`	dataT
`reference`	dataT&
`const_reference`	const dataT&

Member functions

(constructor)	constructs an accessor
(destructor)	destroys the accessor
is_placeholder	
get_size	
get_count	
get_range	
get_offset	

`get_size`

```
size_t get_size() const
```

Description

`get_count`

```
size_t get_size() const [1]
size_t get_size(int b) const [2]
size_t get_size(int c, int d) const [3]
```

Description

A common method of construction can employ the `get_access` method of the memory object to specify the object and infer the other parameters from that memory object.

See *Accessors* for access modes and targets.

Placeholder accessors are those created independent of a command group and then later associated with a particular memory object. Designation of a placeholder accessor is communicated via the placeholder argument set to `access::placeholder::true_t` if so and `access::placeholder::false_t` otherwise.

Once an accessor is created, query member functions to review accessor information. These member functions include:

- `is_placeholder` – return true if accessor is a placeholder, not yet associated with a memory object, false otherwise
- `get_size` – obtain the size (in bytes) of the memory object
- `get_count` – obtain the number of elements of the memory object
- `get_range` – obtain the range of the memory object, where range is a range class
- `get_offset` – obtain the offset of the memory object

[1] No arguments
[2] single argument
[3] 2 arguments

An accessor can reference a subset of a memory object; this is the offset of the accessor into the memory object.

4.3.2.2 Atomic

The DPC++ `atomic` class encapsulates operations and member functions to guarantee synchronized access to data values. Construction of an atomic object requires a reference to a *Address Spaces*. A `multi_ptr` is an abstraction on top of a low-level pointer that enables efficient access across the host and devices.

The atomic member functions are modeled after the C++ standard atomic functions. They are documented more fully in the SYCL Specification and include the following:

- `Store` – store a value
- `Load` – load a value
- `Exchange` – swap two values
- `Compare_exchange_strong` - compares two values for equality and exchanges based on result
- `Fetch_add` – add a value to the value pointed to by a `multi_ptr`
- `Fetch_sub` - subtract a value from the value pointed to by a `multi_ptr`
- `Fetch_and` – bitwise and a value from the value pointed to by a `multi_ptr`
- `Fetch_or` - bitwise or a value from the value pointed to by a `multi_ptr`
- `Fetch_xor` - bitwise xor a value from the value pointed to by a `multi_ptr`
- `Fetch_min` – compute the minimum between a value and the value pointed to by a `multi_ptr`
- `Fetch_max` - compute the maximum between a value and the value pointed to by a `multi_ptr`

In addition to the member functions above, a set of functions with the same capabilities are available acting on atomic types. These functions are similarly named with the addition of "`atomic_`" prepended.

enum class memory_order

enumerator relaxed

template<typename **T**, access::*address_space* **addressSpace** = access::*address_space*::*global_space*>
class atomic

tparam T

tparam addressSpace

template<typename **pointerT**>
atomic(*multi_ptr*<*pointerT*, *addressSpace*> *ptr*)

Template Parameters **pointerT** –

Parameters **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*)

Parameters

- **operand** –
- **memoryOrder** –

bool **compare_exchange_strong**(*T &expected*, *T desired*, *memory_order successMemoryOrder* = *memory_order*::*relaxed*, *memory_order failMemoryOrder* = *memory_order*::*relaxed*)

Parameters

- **expected** –
- **desired** –
- **successMemoryOrder** –
- **failMemoryOrder** –

Available only when: *T* != float

T **fetch_add**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_sub**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_and**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_or**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_xor**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_min**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

T **fetch_max**(*T operand*, *memory_order memoryOrder* = *memory_order*::*relaxed*)

Parameters

- **operand** –
- **memoryOrder** –

Available only when: *T* != float

4.3.2.3 Buffer

A DPC++ `buffer` encapsulates a 1-, 2-, or 3-dimensional array that is shared between host and devices. Creating a buffer requires the number of dimensions of the array as well as the type of the underlying data.

The class contains multiple constructors with different combinations of ranges, allocators, and property lists.

- The memory employed by the buffer is already existing in host memory. In this case, a pointer to the memory is passed to the constructor.
- Temporary memory is allocated for the buffer by employing the constructors that do not include a `hostData` parameter.
- An allocator object is passed, which provides an alternative memory allocator to be used for allocating the temporary memory for the buffer. Special arguments, termed properties, can be provided to the constructor for cases where host memory use is desired (`use_host_ptr`), use of the mutex_class is desired (`use_mutex`), and single context only (`context_bound`) is desired.

Once a buffer is allocated, query member functions to learn more. These member functions include:

- `get_range` – obtain the range object associated with the buffer
- `get_count` – obtain the number of elements in the buffer
- `get_size` – obtain the size of the buffer in bytes
- `get_allocator` – obtain the allocator that was provided in creating the buffer
- `is_sub_buffer` – return if buffer is a sub-buffer or not

template<typename **T**, int **dimensions** = 1, typename **AllocatorT** = cl::sycl::buffer_allocator>

class buffer

Template Parameters

- **T** – buffer element type
- **dimensions** – dimensionality of the data
- **AllocatorT** – STL allocator

using value_type = *T*

using reference = *value_type*&

using const_reference = **const** *value_type*&

using allocator_type = *AllocatorT*

Standard types

buffer(**const** *range*<*dimensions*> &*bufferRange*, **const** *property_list* &*propList* = {})

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

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

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

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

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

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

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

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

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

buffer(*buffer*<*T*, *dimensions*, *AllocatorT*> *b*, **const** *id*<*dimensions*> &*baseIndex*, **const** *range*<*dimensions*> &*subRange*)

Constructors.

Parameters

- **bufferRange** – Dimensions of buffer
- **propList** – Properties for buffer
- **hostData** – Home for data
- **allocator** – STL allocator for data
- **first** – first and last are STL iterators used to initialize the buffer
- **last** – first and last are STL iterators used to initialize the buffer
- **b** – Use buffer *b* to initialize
- **baseIndex** – Index in *b*
- **subRange** – Range of data in *b*

buffer(cl_mem *clMemObject*, **const** context &*syclContext*, event *availableEvent* = {})

Parameters

- **clMemObject** –
- **syclContext** –
- **availableEvent** –

Simplified buffer constructor when *dimensions* == 1.

range<*dimensions*> **get_range**() **const**

Returns

size_t **get_count**() **const**

Returns

size_t **get_size**() **const**

Returns

AllocatorT **get_allocator**() **const**

Returns

template<access::mode **mode**, access::target **target** = access::target::global_buffer>
accessor<*T*, *dimensions*, *mode*, *target*> **get_access**(*handler* &*commandGroupHandler*)

template<access::mode **mode**>
accessor<*T*, *dimensions*, *mode*, access::target::host_buffer> **get_access**()

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* = {})

template<access::mode **mode**>
accessor<*T*, *dimensions*, *mode*, access::target::host_buffer> **get_access**(*range*<*dimensions*> *accessRange*, *id*<*dimensions*> *accessOffset* = {})

Template Parameters

- **mode** –
- **target** –

Parameters

- **handler** –
- **accessRange** –
- **accessOffset** –

Returns

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

Template Parameters **Destination** –

Parameters **finalData** –

void **set_write_back**(bool *flag* = true)

Parameters **flag** –

bool **is_sub_buffer**() **const**

Returns

template<typename **ReinterpretT**, int **ReinterpretDim**>
buffer<*ReinterpretT*, *ReinterpretDim*, *AllocatorT*> **reinterpret**(*range*<*ReinterpretDim*> *reinterpretRange*) **const**

Template Parameters

- **ReinterpretT** –
- **ReinterpretDim** –

Parameters **reinterpretRange** –

Returns

class use_host_ptr

use_host_ptr()

class use_mutex

use_mutex(*mutex_class* &*mutexRef*)

Parameters **mutexRef** –

mutex_class ***get_mutex_ptr**() **const**

Returns

class context_bound

context_bound(context *boundContext*)

Parameters **boundContext** –

context **get_context**() **const**

Returns

4.3.2.4 Image

A DPC++ `image` encapsulates a 1-, 2-, or 3-dimensional set of data shared between host and devices. Creating an image requires the number of dimensions of the array as well as the order and type of the underlying data.

The class contains multiple constructors with different combinations of orders, types, ranges, allocators, and property lists.

- The memory employed by the image is already existing host memory. In this case, a pointer to the memory is passed to the constructor.
- Temporary memory is allocated for the image by employing the constructors that do not include a `hostPointer` parameter.
- An allocator object is passed, which provides an alternative memory allocator to be used for allocating the temporary memory for the buffer.
- When host memory use is desired (`use_host_ptr`), use of the mutex_class is desired (`use_mutex`), and if a single context only (`context_bound`) is desired, special arguments, termed properties, are provided to the constructor.

Once a buffer is allocated, query member functions to learn more. These member functions include

- `get_range` – obtain the range object associated with the image
- `get_pitch` - obtain the range associated with a one-dimensional image
- `get_count` – obtain the number of elements in the image
- `get_size` – obtain the size of the image (in bytes)
- `get_allocator` – obtain the allocator that was provided in creating the image
- `get_access` – obtain an accessor to the image with the specified access mode and target

4.3.2.5 Address Spaces

The DPC++ `multi-pointer` class encapsulates lower level pointers that point to abstract device memory.

Constructors for the `multi-pointer` class enable explicit mention of the address space of the memory. The following lists the address space with the appropriate identifier:

- Global memory – `global_space`
- Local memory – `local_space`
- Constant memory – `constant_space`
- Private memory – `private_space`

The constructors can also be called in an unqualified fashion for cases where the location will be known later.

Member functions include standard pointer operations such as ++ (increment), – (decrement), + (plus), and – (minus). A prefetch function is also specified to aid in optimization and is implementation defined.

Conversion operations are also available to convert between the raw underlying pointer and an OpenCL program's C pointer for interoperability. Consult the SYCL Specification for complete details.

enum class address_space

enumerator global_space
enumerator local_space
enumerator constant_space
enumerator private_space

template<typename **ElementType**, access::*address_space* **Space**>
class multi_ptr

using element_type = *ElementType*
using difference_type = std::ptrdiff_t
using pointer_t = *ElementType**
using const_pointer_t = **const** *ElementType**
using reference_t = *ElementType*&
using const_reference_t = **const** *ElementType*&
Implementation defined pointer and reference types that correspond to SYCL/OpenCL interoperability types for OpenCL C functions

static constexpr access::*address_space* **address_space** = *Space*

multi_ptr()
multi_ptr(**const** *multi_ptr*&)

multi_ptr(*multi_ptr*&&)

multi_ptr(*pointer_t*)

multi_ptr(*ElementType**)

multi_ptr(std::nullptr_t)

template<int **dimensions**, access::mode **Mode**, access::placeholder **isPlaceholder**>
multi_ptr(accessor<*ElementType*, *dimensions*, *Mode*, access::target::global_buffer, *isPlaceholder*>)

template<int **dimensions**, access::mode **Mode**, access::placeholder **isPlaceholder**>
multi_ptr(accessor<*ElementType*, *dimensions*, *Mode*, access::target::local, *isPlaceholder*>)

template<int **dimensions**, access::mode **Mode**, access::placeholder **isPlaceholder**>
multi_ptr(accessor<*ElementType*, *dimensions*, *Mode*, access::target::constant_buffer, *isPlaceholder*>)

~multi_ptr()

Assignment and access operators

multi_ptr &**operator=**(**const** *multi_ptr*&)

multi_ptr &**operator=**(*multi_ptr*&&)

multi_ptr &**operator=**(*pointer_t*)

multi_ptr &**operator=**(*ElementType**)

multi_ptr &**operator=**(std::nullptr_t)

friend *ElementType* &**operator***(**const** *multi_ptr* &*mp*)

ElementType ***operator->**() **const**

pointer_t **get**() **const**

Returns underlying OpenCL C pointer

operator ElementType*() **const**
Implicit conversion to the underlying pointer type

operator multi_ptr<void, Space>() **const**

operator multi_ptr<const void, Space>() **const**

operator multi_ptr<const ElementType, Space>() **const**
Implicit conversion to a multi_ptr

void **prefetch**(size_t *numElements*) **const**

Arithmetic operators

friend *multi_ptr* &**operator++**(*multi_ptr* &*mp*)

friend *multi_ptr* **operator++**(*multi_ptr* &*mp*, int)

friend *multi_ptr* &**operator--**(*multi_ptr* &*mp*)

friend *multi_ptr* **operator--**(*multi_ptr* &*mp*, int)

friend *multi_ptr* &**operator+=**(*multi_ptr* &*lhs*, *difference_type r*)

friend *multi_ptr* &**operator-=**(*multi_ptr* &*lhs*, *difference_type r*)

friend *multi_ptr* **operator+**(**const** *multi_ptr* &*lhs*, *difference_type r*)

friend *multi_ptr* **operator-**(**const** *multi_ptr* &*lhs*, *difference_type r*)

Relational operators

friend bool **operator==**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator!=**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator<**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator>**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator<=**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator>=**(**const** *multi_ptr* &*lhs*, **const** *multi_ptr* &*rhs*)
friend bool **operator==**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator!=**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator<**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator>**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator<=**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator>=**(**const** *multi_ptr* &*lhs*, std::nullptr_t)
friend bool **operator==**(std::nullptr_t, **const** *multi_ptr* &*rhs*)
friend bool **operator!=**(std::nullptr_t, **const** *multi_ptr* &*rhs*)
friend bool **operator<**(std::nullptr_t, **const** *multi_ptr* &*rhs*)
friend bool **operator>**(std::nullptr_t, **const** *multi_ptr* &*rhs*)
friend bool **operator<=**(std::nullptr_t, **const** *multi_ptr* &*rhs*)
friend bool **operator>=**(std::nullptr_t, **const** *multi_ptr* &*rhs*)

template<typename **ElementType**>
using global_ptr = *multi_ptr*<*ElementType*, access::*address_space*::*global_space*>

template<typename **ElementType**>
using local_ptr = *multi_ptr*<*ElementType*, access::*address_space*::*local_space*>

template<typename **ElementType**>
using constant_ptr = *multi_ptr*<*ElementType*, access::*address_space*::*constant_space*>

template<typename **ElementType**>
using private_ptr = *multi_ptr*<*ElementType*, access::*address_space*::*private_space*>

4.3.2.6 Samplers

4.3.3 Expressing Parallelism

4.3.3.1 Range

The DPC++ `range` class encapsulates the iteration domain of a work-group or the entire kernel dispatch. Constructors for a range object take one, two, or three arguments of `size_t` dependent on the dimensionality of the range, either one, two, or three dimensions respectively.

Member functions include:

- `get` – obtain the specified dimension
- `size` – obtain the size of the range

Additional functions allow construction of new ranges from old ranges with additional operations on the range. For example:

```
Range<2> +() const??
```

template<int **dimensions** = 1>
class range

range (size_t *dim0*)

range (size_t *dim0*, size_t *dim1*)

range (size_t *dim0*, size_t *dim1*, size_t *dim2*)

Parameters

- **dim0** –
- **dim1** –
- **dim2** –

size_t **get** (int *dimension*) **const**

Parameters **dimension** –

Returns

size_t &**operator[]** (int *dimension*)

size_t **operator[]** (int *dimension*) **const**

Parameters **dimension** –

Returns

size_t **size** () **const**

friend *range* **operator+** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator-** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator*** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator/** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator%** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator<<** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator>>** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator&** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator|** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator^** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator&&** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator||** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator<** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator>** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator<=** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator>=** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator+** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator-** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator*** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator/** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator%** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator<<** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator>>** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator&** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator|** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator^** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator&&** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator||** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator<** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator>** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator<=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator>=** (**const** *range* &*lhs*, **const** size_t &*rhs*)

friend *range* **operator+** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator-** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator*** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator/** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator%** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator<<** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator>>** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator&** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator|** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator^** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator&&** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator||** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator<** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator>** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator<=** (**const** size_t &*lhs*, **const** *range* &*rhs*)
friend *range* **operator>=** (**const** size_t &*lhs*, **const** *range* &*rhs*)

friend *range* **operator+=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator-=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator*=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator/=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator%=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator<<=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator>>=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator&=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator|=** (**const** *range* &*lhs*, **const** *range* &*rhs*)
friend *range* **operator^=** (**const** *range* &*lhs*, **const** *range* &*rhs*)

friend *range* **operator+=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator-=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator*=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator/=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator%=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator<<=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator>>=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator&=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator|=** (**const** *range* &*lhs*, **const** size_t &*rhs*)
friend *range* **operator^=** (**const** *range* &*lhs*, **const** size_t &*rhs*)

4.3.3.2 Nd_range

The DPC++ `nd_range` class encapsulates the iteration domain of the work-groups and kernel dispatch. It is the entire iteration space of data that a kernel may operation upon. The constructor for an `nd_range` object take the global range, local range, and an optional offset.

Member functions include:

- `get_global_range` – obtain the global range
- `get_local_range` – obtain the local range
- `get_group_range` – obtain the number of groups in each dimension of the `nd_range`
- `get_offset` – obtain the offset

template<int **dimensions** = 1>
class nd_range

nd_range(*range*<*dimensions*> *globalSize*, *range*<*dimensions*> *localSize*, *id*<*dimensions*> *offset* = *id*<*dimensions*>())

range<*dimensions*> **get_global_range**() **const**

range<*dimensions*> **get_local_range**() **const**

range<*dimensions*> **get_group_range**() **const**

id<*dimensions*> **get_offset**() **const**

4.3.3.3 ID

The `id` class encapsulates a vector of dimensions that identify an index into a global or local range. Constructors for the class take one to three integer arguments representing a one, two, and three dimension ID. Each integer argument specifies the size of the dimension. ID objects can also be constructed as a placeholder where the dimension is unspecified and set to zero by default. Construction can also be based upon the dimension of an existing range or item.

The class supports operations such as + (plus), - (minus), and many more. Consult the SYCL Specification for complete details.

template<int **dimensions** = 1>
class id

tparam dimensions

id()

id(size_t *dim0*)

id(size_t *dim0*, size_t *dim1*)

id(size_t *dim0*, size_t *dim1*, size_t *dim2*)

id(**const** *range*<*dimensions*> &*range*)

id(**const** *item*<*dimensions*> &*item*)

Parameters

- **dim0** –
- **dim1** –
- **dim2** –
- **range** –

- **item** –

size_t **get**(int *dimension*) **const**

Returns

size_t &**operator[]**(int *dimension*)

Returns

size_t **operator[]**(int *dimension*) **const**

Returns

friend *id* **operator+**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator-**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator*** (**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator/**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator%**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<<**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>>**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator&**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator|**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator^**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator&&**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator||**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<=**(**const** *id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>=**(**const** *id* &*lhs*, **const** *id* &*rhs*)

friend *id* **operator+**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator-**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator*** (**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator/**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator%**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator<<**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator>>**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator&**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator|**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator^**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator&&**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator||**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator<**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator>**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator<=**(**const** *id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator>=**(**const** *id* &*lhs*, **const** size_t &*rhs*)

friend *id* **operator+**(**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator-**(**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator*** (**const** size_t &*lhs*, **const** *id* &*rhs*)

friend *id* **operator/** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator%** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<<** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>>** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator&** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator|** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator^** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator&&** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator||** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<=** (**const** size_t &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>=** (**const** size_t &*lhs*, **const** *id* &*rhs*)

friend *id* **operator+=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator*=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator/=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator%=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator<<=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator>>=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator&=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator|=** (*id* &*lhs*, **const** *id* &*rhs*)
friend *id* **operator^=** (*id* &*lhs*, **const** *id* &*rhs*)

friend *id* **operator+=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator*=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator/=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator%=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator<<=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator>>=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator&=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator|=** (*id* &*lhs*, **const** size_t &*rhs*)
friend *id* **operator^=** (*id* &*lhs*, **const** size_t &*rhs*)

4.3.3.4 Item

A DPC++ `item` encapsulates a function object executing on an individual data point in a DPC++ range. When a kernel is executed, it is associated with an individual item in a range and acts upon it. This association is accomplished implicitly, by the runtime. Therefore, there are no user callable constructors; a DPC++ item is created when a kernel is instantiated.

The member functions of the item class pertain to determining the relationship between the item and the enclosing range:

- `get_id` – obtain the position of the work item in the iteration space
- `get_range` – obtain the range associated with the item
- `get_offset` – obtain the position of the item in the n-dimensional space

- `get_linear_id` – obtain the position of the item converting the n-dimensional space into one

template<int **dimensions** = 1, bool **with_offset** = true>
class item

tparam dimensions

tparam with_offset

id<*dimensions*> **get_id**() **const**

size_t **get_id**(int *dimension*) **const**

Parameters **dimension** –

Returns

size_t **operator[]**(int *dimension*) **const**

range<*dimensions*> **get_range**() **const**

size_t **get_range**(int *dimension*) **const**

Parameters **dimension** –

Returns

id<*dimensions*> **get_offset**() **const**

Returns

operator item<dimensions, true>() **const**

size_t **get_linear_id**() **const**

4.3.3.5 Nd_item

A DPC++ `nd_item` encapsulates a function object executing on an individual data point in a DPC++ `nd_range`. When a kernel is executed, it is associated with an individual item in a range and acts upon it. This association is accomplished implicitly, by the runtime. Therefore, there are no user callable constructors; a DPC++ `nd_item` is created when a kernel is instantiated.

The member functions of the `nd_item` class pertain to determining the relationship between the `nd_item` and the enclosing range:

- `get_global_id` – obtain the position of the work item in the iteration space
- `get_global_linear_id` – obtain the position of the work item in a linear representation of the global iteration space
- `get_local_id` – obtain the position of the item in the current work-group
- `get_local_linear_id` – obtain the position of the item in a linear representation of the current work-group
- `get_group` – obtain the position of the item in the overall `nd_range`
- `get_group_range` – obtain the number of work-groups in the iteration space
- `get_global_range` – obtain the range representing the dimensions of the global iteration space
- `get_local_range` – obtain the range representing the dimension of the current work-group
- `get_offset` – obtain an id that represents the offset between a work-item representation between local and global iteration space
- `get_nd_range` – obtain the `nd_range` from the `nd_item`

The class also includes a member function, `async_work_group_copy`, which can copy a range of items asynchronously.

template<int **dimensions** = 1>
class nd_item

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**

nd_range<*dimensions*> **get_nd_range**() **const**

void **barrier**(access::fence_space *accessSpace* = access::fence_space::global_and_local) **const**

template<access::mode **accessMode** = access::mode::read_write>
void **mem_fence**(access::fence_space *accessSpace* = access::fence_space::global_and_local) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*local_ptr*<*dataT*> *dest*, *global_ptr*<*dataT*> *src*, size_t *numElements*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*global_ptr*<*dataT*> *dest*, *local_ptr*<*dataT*> *src*, size_t *numElements*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*local_ptr*<*dataT*> *dest*, *global_ptr*<*dataT*> *src*, size_t *numElements*, size_t *srcStride*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*global_ptr*<*dataT*> *dest*, *local_ptr*<*dataT*> *src*, size_t *numElements*, size_t *destStride*) **const**

template<typename ...**eventTN**>
void **wait_for**(*eventTN*... *events*) **const**

4.3.3.6 Group

The `group` class encapsulates work-group functionality. Constructors are not user-callable; objects are created as a by-product of a call to `parallel_for_work_group`.

Once a group object has been instantiated, query various properties of the object by calling several member functions including:

- `get_id` – obtains the index of the work-group
- `get_global_range` – obtain a range that represents the work-items across the index space
- `get_local_range` – obtain a range that represents the work-items in a work-group
- `get_group_range` – obtain a range representing the dimensions of the current work-group
- `get_linear_id` – obtain a linear version of the work-group id

Definitions of global range and local range are in the SYCL Specification glossary. In brief, a global range is the overall number of work-items in the index space. A local range is the number of work-items in a work-group.

template<int **dimensions** = 1>
class group

id<*dimensions*> **get_id**() **const**

size_t **get_id**(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**

range<*dimensions*> **get_group_range**() **const**

size_t **get_group_range**(int *dimension*) **const**

size_t **operator[]**(int *dimension*) **const**

size_t **get_linear_id**() **const**

template<typename **workItemFunctionT**>
void **parallel_for_work_item**(*workItemFunctionT func*) **const**

template<typename **workItemFunctionT**>
void **parallel_for_work_item**(*range*<*dimensions*> *logicalRange*, *workItemFunctionT func*) **const**

template<access::mode **accessMode** = access::mode::read_write>
void **mem_fence**(access::fence_space *accessSpace* = access::fence_space::global_and_local) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*local_ptr*<*dataT*> *dest*, *global_ptr*<*dataT*> *src*, size_t *numElements*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*global_ptr*<*dataT*> *dest*, *local_ptr*<*dataT*> *src*, size_t *numElements*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*local_ptr*<*dataT*> *dest*, *global_ptr*<*dataT*> *src*, size_t *numElements*, size_t *srcStride*) **const**

template<typename **dataT**>
device_event **async_work_group_copy**(*global_ptr*<*dataT*> *dest*, *local_ptr*<*dataT*> *src*, size_t *numElements*, size_t *destStride*) **const**

template<typename ...**eventTN**>
void **wait_for**(*eventTN... events*) **const**

4.3.3.7 Device Event

The DPC++ `device_event` class encapsulates wait objects within kernels. The `device_event` objects are used to coordinate asynchronous operations in kernels. The constructor and its arguments are unspecified and implementation dependent. The wait member function causes execution to stop until the operation associated with the wait is complete.

class device_event

void **wait**()

4.3.3.8 Command Group Handler

The command group handler class encapsulates the actions of the command group, namely the marshaling of data and launching of the kernels on the devices.

There are no user callable constructors; construction is accomplished by the oneAPI runtime. Consider the example code below:

```
d_queue.submit([&](sycl::handler &cgh) {
         auto c_res = c_device.get_access<sycl::access::mode::write>(cgh);
         cgh.parallel_for<class ex1>(a_size,[=](sycl::id<1> idx) {
           c_res[idx] =0;
         });
       });
```

In the example, the accessor, `c_res`, is obtained from the device and takes a command group handler as a parameter, in this case `cgh`. The kernel dispatch itself is a member function of the command group handler. In this case, a `parallel_for` is called. The kernel dispatch API has multiple calls including `parallel_for`, `parallel_for_work_group`, and `single_task`.

There is a `set_args` function employed for passing arguments to an OpenCL™ kernel for interoperability.

class handler

template<typename **dataT**, int **dimensions**, access::mode **accessMode**, access::target **accessTarget**>
void **require**(accessor<*dataT*, *dimensions*, *accessMode*, *accessTarget*, access::placeholder::true_t>
acc)

Template Parameters

- **dataT** –
- **dimensions** –
- **accessMode** –
- **accessTarget** –

OpenCL interoperability interface

template<typename **T**>
void **set_arg**(int *argIndex*, *T* &&*arg*)

Template Parameters **T** –

Parameters

- **argIndex** –
- **arg** –

template<typename ...**Ts**>
void **set_args**(*Ts*&&... *args*)

Kernel dispatch API

Note: Note: In all Kernel dispatch functions, when using a functor with a globally visible name the template parameter:"typename kernelName" can be ommitted and the kernelType can be used instead.

---

template<typename **KernelName**, typename **KernelType**>
void **single_task**(*KernelType kernelFunc*)

void **single_task**(kernel *syclKernel*)

Template Parameters

- **KernelName** –
- **KernelType** –

Parameters

- **kernelFunc** –
- **syclKernel** –

template<typename **KernelName**, typename **KernelType**, int **dimensions**>
void **parallel_for**(*range*<*dimensions*> *numWorkItems*, *KernelType kernelFunc*)

template<typename **KernelName**, typename **KernelType**, int **dimensions**>
void **parallel_for**(*range*<*dimensions*> *numWorkItems*, *id*<*dimensions*> *workItemOffset*, *KernelType kernelFunc*)

template<typename **KernelName**, typename **KernelType**, int **dimensions**>
void **parallel_for**(*nd_range*<*dimensions*> *executionRange*, *KernelType kernelFunc*)

template<int **dimensions**>
void **parallel_for**(*range*<*dimensions*> *numWorkItems*, kernel *syclKernel*)

template<int **dimensions**>
void **parallel_for**(*range*<*dimensions*> *numWorkItems*, *id*<*dimensions*> *workItemOffset*, kernel *syclKernel*)

template<int **dimensions**>
void **parallel_for**(*nd_range*<*dimensions*> *ndRange*, kernel *syclKernel*)

Template Parameters

- **KernelName** –
- **KernelType** –
- **dimensions** –

Parameters

- **numWorkItems** –
- **workItemOffset** –
- **executionRange** –
- **ndRange** –
- **kernelFunc** –
- **syclKernel** –

template<typename **KernelName**, typename **WorkgroupFunctionType**, int **dimensions**>
void **parallel_for_work_group**(*range*<*dimensions*> *numWorkGroups*, *WorkgroupFunctionType* *kernelFunc*)

template<typename **KernelName**, typename **WorkgroupFunctionType**, int **dimensions**>
void **parallel_for_work_group**(*range*<*dimensions*> *numWorkGroups*, *range*<*dimensions*> *workGroupSize*, *WorkgroupFunctionType* *kernelFunc*)

Template Parameters

- **KernelName** –
- **WorkgroupFunctionType** –
- **dimensions** –

Parameters

- **numWorkGroups** –
- **kernelFunc** –

Explicit memory operation APIs

template<typename **T_src**, int **dim_src**, access::mode **mode_src**, access::target **tgt_src**, access::placeholder **isPlaceh**
void **copy**(accessor<*T_src*, *dim_src*, *mode_src*, *tgt_src*, *isPlaceholder*> *src*, *shared_ptr_class*<*T_dest*> *dest*)

template<typename **T_src**, typename **T_dest**, int **dim_dest**, access::mode **mode_dest**, access::target **tgt_dest**, access:
void **copy**(*shared_ptr_class*<*T_src*> *src*, accessor<*T_dest*, *dim_dest*, *mode_dest*, *tgt_dest*, *isPlaceholder*> *dest*)

template<typename **T_src**, int **dim_src**, access::mode **mode_src**, access::target **tgt_src**, access::placeholder **isPlaceh**
void **copy**(accessor<*T_src*, *dim_src*, *mode_src*, *tgt_src*, *isPlaceholder*> *src*, *T_dest* **dest*)

template<typename **T_src**, typename **T_dest**, int **dim_dest**, access::mode **mode_dest**, access::target **tgt_dest**, access:
void **copy**(**const** *T_src* **src*, accessor<*T_dest*, *dim_dest*, *mode_dest*, *tgt_dest*, *isPlaceholder*> *dest*)

template<typename **T_src**, int **dim_src**, access::mode **mode_src**, access::target **tgt_src**, access::placeholder **isPlaceh**
void **copy**(accessor<*T_src*, *dim_src*, *mode_src*, *tgt_src*, *isPlaceholder_src*> *src*, accessor<*T_dest*, *dim_dest*, *mode_dest*, *tgt_dest*, *isPlaceholder_dest*> *dest*)

Template Parameters

- **T_src** –
- **dim_src** –
- **mode_src** –
- **tgt_src** –
- **isPlaceholder** –
- **isPlaceholder_src** –

- **T_dest** –
- **dim_dest** –
- **mode_dest** –
- **tgt_dest** –
- **isPlaceholder_dest** –

Parameters

- **src** –
- **dest** –

template<typename **T**, int **dim**, access::mode **mode**, access::target **tgt**, access::placeholder **isPlaceholder**> void **update_host**(accessor<*T*, *dim*, *mode*, *tgt*, *isPlaceholder*> *acc*)

Template Parameters

- **T** –
- **dim** –
- **mode** –
- **tgt** –
- **isPlaceholder** –

Parameters **acc** –

template<typename **T**, int **dim**, access::mode **mode**, access::target **tgt**, access::placeholder **isPlaceholder**> void **fill**(accessor<*T*, *dim*, *mode*, *tgt*, *isPlaceholder*> *dest*, **const** *T* &*src*)

Template Parameters

- **T** –
- **dim** –
- **mode** –
- **tgt** –
- **isPlaceholder** –

Parameters

- **dest** –
- **src** –

4.3.3.9 Kernel

The DPC++ `kernel` class encapsulates methods and data for executing code on the device when a command group is instantiated. In many cases, the runtime creates the kernel objects when a command queue is instantiated.

Typically, a kernel object is not explicitly constructed by the user; instead it is constructed when a kernel dispatch function, such as `parallel_for`, is called. The sole case where a kernel object is constructed is when constructing a kernel object from an OpenCL application's `cl_kernel`. To compile the kernel ahead of time for use by the command queue, use the program class.

Member functions of the class return related objects and attributes regarding the kernel object including:

- `get` – obtains a `cl_kernel` if associated

- `Is_host` – obtains if the kernel is for the host
- `get_context` – obtains the context to which the kernel is associated
- `get_program` – obtains the program the kernel is contained in
- `get_info` – obtain details on the kernel and return in `info::kernel_info` descriptor
- `get_work_group_info` – obtain details on the work group and return in `info::kernel_work_group` descriptor

The `get_info` member function obtains kernel information such as `function_name`, `num_args`, `context`, `program`, `reference_count`, and `attributes`.

```
enum class kernel

    enumerator function_name
    enumerator num_args
    enumerator context
    enumerator program
    enumerator reference_count
    enumerator attributes
enum class kernel_work_group

    enumerator global_work_size
    enumerator work_group_size
    enumerator compile_work_group_size
    enumerator preferred_work_group_size_multiple
    enumerator private_mem_size
```

4.3.3.10 Program

A DPC++ `program` class encapsulates a program, either a host program or an OpenCL program. The `program` object is employed when compilation or linkage of the program is desired.

Constructors for a `program` object require a context at a minimum.

Member functions of the `program` class include:

- `get` – obtain an OpenCL `program` object from the program
- `is_host` – determines if the program is targeted for the host
- `compile_with_kernel_type` – enables compilation of a kernel
- `compile_with_source` – compiles OpenCL kernel
- `build_with_kernel_type` – builds kernel function
- `build_with_source` – builds kernel function from source
- `link` – link the object files
- `has_kernel` – determines if the program has a valid kernel function
- `get_kernel` – obtains the kernel from the program
- `get_binaries` – obtain a vector of compiled binaries for each device in the program

- `get_context` – obtain the context the program was built with
- `get_devices` – obtain a vector of the compiled binary sizes for each device

enum class program

enumerator context
enumerator devices
enumerator reference_count

4.3.4 Error Handling

4.3.4.1 Exception

The DPC++ `exception` class encapsulates objects to communicate error conditions from the DPC++ program. Errors during DPC++ program execution are either scheduling or device related.

Execution between host and device is asynchronous in nature, therefore any events or errors are asynchronous. To catch exceptions that occur on the device, employ an `asynch_handler`, which is provided during command queue construction. During execution of the kernel, if any exceptions occur, these are placed on the `async_handler` list for processing once the command group function returns and the host handles the exceptions through the `async_handler` list. The `exception_ptr_class` is used to store the exception and can contain exceptions representing different types of errors such as `device_error_ compile_program_error`, `link_program_error`, `invalid_object_error`, `memory_allocation_error`, `platform_error`, `profiling_error`, and `feature_not_supported`.

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

class exception

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

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*

4.3.5 Stream

The DPC++ `stream` class is employed for outputting values of SYCL built-in, vector, and other types to the console.

enum class stream_manipulator

enumerator flush
enumerator dec
enumerator hex
enumerator oct
enumerator noshowbase
enumerator showbase
enumerator noshowpos
enumerator showpos
enumerator endl
enumerator fixed
enumerator scientific
enumerator hexfloat
enumerator defaultfloat

const *stream_manipulator* **flush** = *stream_manipulator*::*flush*

const *stream_manipulator* **dec** = *stream_manipulator*::*dec*

const *stream_manipulator* **hex** = *stream_manipulator*::*hex*

const *stream_manipulator* **oct** = *stream_manipulator*::*oct*

const *stream_manipulator* **noshowbase** = *stream_manipulator*::*noshowbase*

const *stream_manipulator* **showbase** = *stream_manipulator*::*showbase*

const *stream_manipulator* **noshowpos** = *stream_manipulator*::*noshowpos*

const *stream_manipulator* **showpos** = *stream_manipulator*::*showpos*

const *stream_manipulator* **endl** = *stream_manipulator*::*endl*

const *stream_manipulator* **fixed** = *stream_manipulator*::*fixed*

const *stream_manipulator* **scientific** = *stream_manipulator*::*scientific*

const *stream_manipulator* **hexfloat** = *stream_manipulator*::*hexfloat*

const *stream_manipulator* **defaultfloat** = *stream_manipulator*::*defaultfloat*

__precision_manipulator__ **setprecision** (int *precision*)

__width_manipulator__ **setw** (int *width*)

class stream

stream (size_t *totalBufferSize*, size_t *workItemBufferSize*, *handler* &*cgh*)

size_t **get_size** () **const**

size_t **get_work_item_buffer_size** () **const**

size_t **get_max_statement_size** () **const**
get_max_statement_size() has the same functionality as *get_work_item_buffer_size()*, and is provided for backward compatibility. get_max_statement_size() is a deprecated query.

4.3.6 Vec and Swizzled Vec

The DPC++ Vec and Swizzled Vec class templates are designed to represent vectors between host and devices.

To instantiate a `vec` class template, provide the type and an integer representing the number of elements. The number of elements can be 1, 2, 3, 4, 8, or 16; any other integer results in a compile error. The type provided must be a basic scalar type, such as `int` or `float`.

Member functions once an object is created include:

- `get_count` – obtains the number of elements of the vec
- `get_size` – obtain the size of the vec (in bytes)
- `lo` – obtain the lower half of the vec
- `hi` – obtain the higher half of the vec
- `odd` – obtain the odd index elements of the vec
- `even` – obtain the even index elements of the vec
- `load` – copy the pointed to values into a vec
- `store` – copy the vec into the pointed to location

The `__swizzled_vec__` class is employed to reposition elements of a `vec` object. A good motivation for employing is to obtain every odd or even element of a vector. In this case, employ the odd or even member function of the class. There are member functions associated with the `__swizzled_vec__` class for converting a `vec` into a new `vec`, such as one in RGBA format.

Various operators on the `vec` class include: +=, -=, *=, /=, %=, ++, –, &, |, ^, +, -, *, /, %, &&, ||, <<, >>, <<=, >>=, ==, !=, <, >, <=, >=.

enum class rounding_mode

enumerator automatic
enumerator rte
enumerator rtz
enumerator rtp
enumerator 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

using element_type = *dataT*
vec()
explicit vec(**const** *dataT* &*arg*)
template<typename ...**argTN**>

vec(**const** *argTN*&... *args*)

vec(**const** *vec*<*dataT*, *numElements*> &*rhs*)

template<typename **convertT**, *rounding_mode* **roundingMode** = *rounding_mode*::*automatic*>
vec<*convertT*, *numElements*> **convert**() **const**

operator dataT() **const**
Available only when: numElements == 1

size_t **get_count**() **const**

size_t **get_size**() **const**

template<typename **asT**>
asT **as**() **const**

template<int... **swizzleIndexes**>
__swizzled_vec__ **swizzle**() **const**

__swizzled_vec__ **x_ACCESS**() **const**

__swizzled_vec__ **y_ACCESS**() **const**

__swizzled_vec__ **z_ACCESS**() **const**

__swizzled_vec__ **w_ACCESS**() **const**
Available only when numElements <= 4

__swizzled_vec__ **r**() **const**

__swizzled_vec__ **g**() **const**

__swizzled_vec__ **b**() **const**

__swizzled_vec__ **a**() **const**
Available only numElements == 4

__swizzled_vec__ **s0**() **const**

__swizzled_vec__ **s1**() **const**

__swizzled_vec__ **s2**() **const**

__swizzled_vec__ **s3**() **const**

__swizzled_vec__ **s4**() **const**

__swizzled_vec__ **s5**() **const**

__swizzled_vec__ **s6**() **const**

__swizzled_vec__ **s7**() **const**

__swizzled_vec__ **s8**() **const**

__swizzled_vec__ **s9**() **const**

__swizzled_vec__ **sA**() **const**

__swizzled_vec__ **sB**() **const**

__swizzled_vec__ **sC**() **const**

__swizzled_vec__ **sD**() **const**

__swizzled_vec__ **sE**() **const**

__swizzled_vec__ **sF**() **const**

__swizzled_vec__ **lo**() **const**

__swizzled_vec__ **hi**() **const**

__swizzled_vec__ **odd**() **const**

__swizzled_vec__ **even**() **const**

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**

load and store member functions

friend *vec* **operatorOP**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operatorOP**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)
OP is: +, -, *, /, %

friend *vec* &**operatorOP**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operatorOP**(*vec* &*lhs*, **const** *dataT* &*rhs*)
OP is: +=, -=, *=, /=, %=

friend *vec* &**operatorOP**(*vec* &*lhs*)

friend *vec* **operatorOP**(*vec* &*lhs*, int)
OP is: ++, –

friend *vec* **operator&**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator&**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* **operator|**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator|**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* **operator^**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator^**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* &**operator&=**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operator&=**(*vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* &**operator|=**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operator|=**(*vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* &**operator^=**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operator^=**(*vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec*<RET, *numElements*> **operator&&**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator&&**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec*<RET, *numElements*> **operator||**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator||**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* **operator<<**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator<<**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* **operator>>**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator>>**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* &**operator<<=**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operator<<=**(*vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec* &**operator>>=**(*vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec* &**operator>>=**(*vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec*<RET, *numElements*> **operator==**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator==**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec*<RET, *numElements*> **operator!=**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator!=**(**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

friend *vec*<RET, *numElements*> **operator<**(**const** *vec* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator<** (**const** *vec* &*lhs*, **const** *dataT* &*rhs*)
friend *vec*<RET, *numElements*> **operator>** (**const** *vec* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator>** (**const** *vec* &*lhs*, **const** *dataT* &*rhs*)
friend *vec*<RET, *numElements*> **operator<=** (**const** *vec* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator<=** (**const** *vec* &*lhs*, **const** *dataT* &*rhs*)
friend *vec*<RET, *numElements*> **operator>=** (**const** *vec* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator>=** (**const** *vec* &*lhs*, **const** *dataT* &*rhs*)

vec<*dataT*, *numElements*> &**operator=** (**const** *vec*<*dataT*, *numElements*> &*rhs*)
vec<*dataT*, *numElements*> &**operator=** (**const** *dataT* &*rhs*)

friend *vec* **operator~** (**const** *vec* &*v*)
friend *vec*<RET, *numElements*> **operator!** (**const** *vec* &*v*)

friend *vec* **operatorOP** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
OP is: +, -, *, /, %

friend *vec* **operator&** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec* **operator|** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec* **operator^** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator&&** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator||** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)

friend *vec* **operator<<** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec* **operator>>** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)

friend *vec*<RET, *numElements*> **operator==** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator!=** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator<** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator>** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator<=** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)
friend *vec*<RET, *numElements*> **operator>=** (**const** *dataT* &*lhs*, **const** *vec* &*rhs*)

4.3.7 Built-in Types & Functions

The DPC++ built-in functions provide low level capabilities that can execute on the host and device with some level of compatibility. Section 4.13 of the SYCL Specification details all the various built-in types and functions available.

One task taken care of by the implementation is the mapping of C++ fundamental types such as int, short, long such that the types agree between the host and the device.

The built-in scalar data types are summarized in the SYCL Specification. In general, the built-in types cover floating point, double, half precision, char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, long long, unsigned long long, and signed integer. Lastly, the built-in types can be post fixed with 2, 3, 4, 8, or 16, which indicated a vector of the post fixed type. For example, `float3` indicates a type consisting of three floating point elements addressed as one object. A `float3` is common in image processing algorithms for representing RGB data.

The built-in functions are either defined as part of lower level classes or part of the basic math functions. These built-in functions enable vendors to provide differentiated built-in functions specific to the architecture while also enabling basic functionality for generic implementations.

The categories of built-in functions are summarized as:

- Work-item functions – pertaining to `nd_item` and `group` classes
- Basic Math functions – low level math and comparison functions
- General math functions – Transcendental, trigonomic, and geometric functions
- Vector load and store – reading and writing `vec` class
- Synchronization – `nd_item` related barriers
- Output – `stream` class for output

4.3.8 Property Interface

The DPC++ property interface is employed with the `buffer`, `image`, and `queue` classes to provide extra information to those classes without affecting the type. These classes provide an additional `has_property` and `get_property` member function to test for and obtain a particular property.

template<typename **propertyT**>
struct is_property

template<typename **propertyT**, typename **syclObjectT**>
struct is_property_of

class T

template<typename **propertyT**>
bool **has_property**() **const**

template<typename **propertyT**>
propertyT **get_property**() **const**

class property_list

template<typename ...**propertyTN**>
property_list(*propertyTN*... *props*)

4.3.9 Version

The include file, `version.h`, includes a definition of `__SYCL_COMPILER_VERSION` based upon the date of the compiler. It can be used to control compilation based upon the specific version of the compiler.

CHAPTER

FIVE

GLOSSARY

accelerator Specialized component containing compute resources that can quickly execute a subset of operations. Examples include CPU, FPGA, GPU. See also: *device*

accessor Communicates the desired location (host, device) and mode (read, write) of access.

application scope Code that executes on the host.

buffers Memory object that communicates the type and number of items of that type to be communicated to the device for computation.

command group scope Code that acts as the interface between the host and device.

command queue Issues command groups concurrently.

compute unit A grouping of processing elements into a 'core' that contains shared elements for use between the processing elements and with faster access than memory residing on other compute units on the device.

device An accelerator or specialized component containing compute resources that can quickly execute a subset of operations. A CPU can be employed as a device, but when it is, it is being employed as an accelerator. Examples include CPU, FPGA, GPU. See also: *accelerator*

device code Code that executes on the device rather than the host. Device code is specified via lambda expression, functor, or kernel class.

fat binary Application binary that contains device code for multiple devices. The binary includes both the generic code (SPIR-V representation) and target specific executable code.

fat library Archive or library of object code that contains object code for multiple devices. The fat library includes both the generic object (SPIR-V representation) and target specific object code.

fat object File that contains object code for multiple devices. The fat object includes both the generic object (SPIR-V representation) and target specific object code.

host A CPU-based system (computer) that executes the primary portion of a program, specifically the application scope and command group scope.

host device A SYCL device that is always present and usually executes on the host CPU.

host code Code that is compiled by the host compiler and executes on the host rather than the device.

images Formatted opaque memory object that is accessed via built-in function. Typically pertains to pictures comprised of pixels stored in format like RGB.

kernel scope Code that executes on the device.

nd-range Short for N-Dimensional Range, a group of kernel instances, or work item, across one, two, or three dimensions.

processing element Individual engine for computation that makes up a compute unit.

single source Code in the same file that can execute on a host and accelerator(s).

SPIR-V Binary intermediate language for representing graphical-shader stages and compute kernels.

sub-group Sub-groups are an Intel extension.

work-group Collection of work-items that execute on a compute unit.

work-item Basic unit of computation in the oneAPI programming model. It is associated with a kernel which executes on the processing element.

CHAPTER

SIX

NOTICES AND DISCLAIMERS

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Intel, the Intel logo, Intel Atom, Intel Core, Intel Xeon Phi, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Java is a registered trademark of Oracle and/or its affiliates.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804

Unless stated otherwise, the code examples in this document are provided to you under an MIT license, the terms of which are as follows:

INDEX

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W