Intel® oneAPI Programming Guide

Intel Corporation

www.intel.com

Notices and Disclaimers

Contents

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.

This software and the related documents are Intel copyrighted materials, and your use of them is governed by the express license under which they were provided to you (**License**). Unless the License provides otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this software or the related documents without Intel's prior written permission.

This software and the related documents are provided as is, with no express or implied warranties, other than those that are expressly stated in the License.

Optimization Notice
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:

1 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.

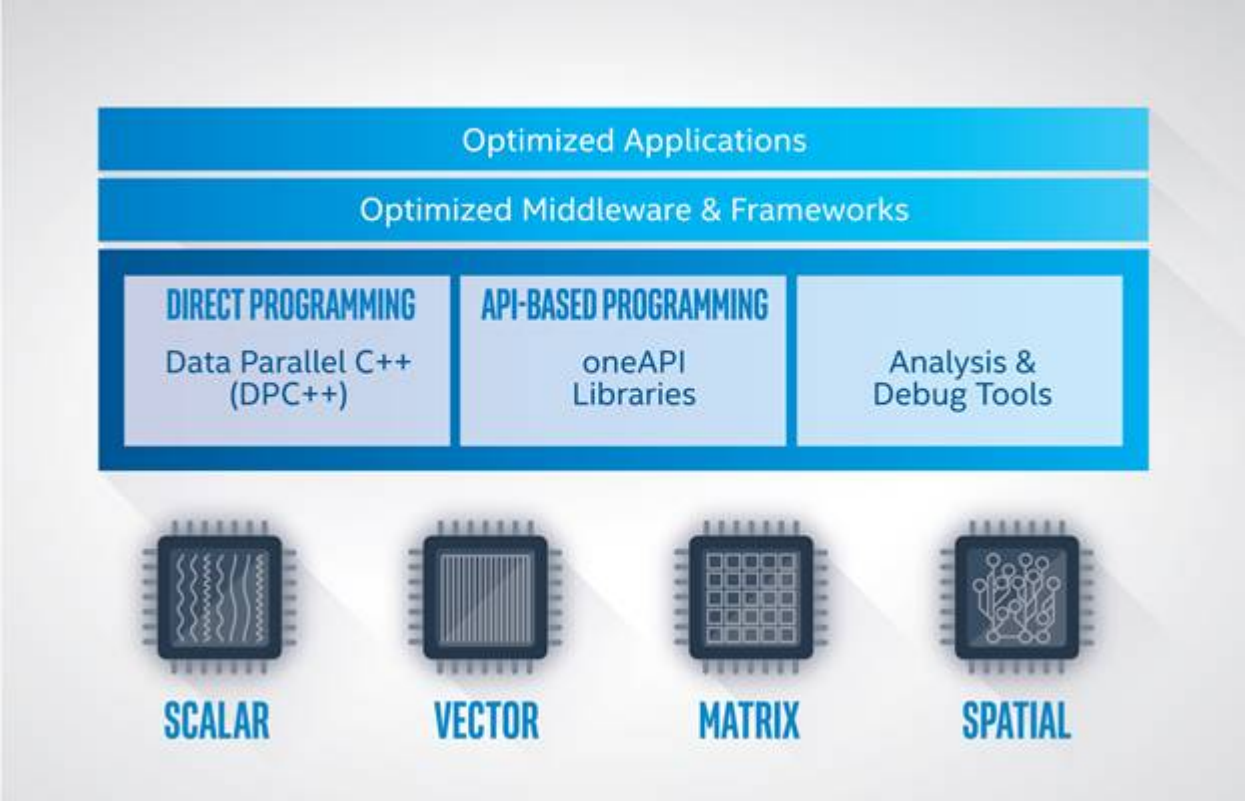
Today, coding for CPUs and accelerators requires different languages, libraries, and tools. That means each hardware platform requires completely separate software investments and provides limited application code reusability across different target architectures.

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.

oneAPI Programming Model 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. The oneAPI product is comprised of the Intel® oneAPI Base Toolkit and several add-on toolkits featuring complementary tools based on specific developer workload needs. The Intel oneAPI Base Toolkit includes the Intel® oneAPI DPC++ Compiler, the Intel® DPC++ Compatibility Tool, select libraries, and analysis tools.

- Developers who want to migrate existing CUDA* code to DPC++, can use the **Intel DPC++ Compatibility Tool** to help migrate their existing projects to DPC++.
- The **Intel oneAPI DPC++ Compiler** supports direct programming of code targeting accelerators. Direct programming is coding for performance when APIs are not available. It supports online and offline compilation for CPU and GPU targets and offline compilation for FPGA targets.
- API-based programming is supported via sets of optimized libraries. The library functions provided in the oneAPI product are pre-tuned for use with any supported target architecture, eliminating the need for developer intervention. For example, the BLAS routine available from **Intel oneAPI Math Kernel Library** is just as optimized for a GPU target as a CPU target.
- Finally, the compiled DPC++ application can be analyzed and debugged to ensure performance, stability, and energy efficiency goals are achieved using tools such as **Intel® VTune™ Profiler** or **Intel® Advisor**.

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.

oneAPI Toolkit Distribution

oneAPI Toolkits are available via multiple distribution channels:

- Local product installation: install the oneAPI toolkits from the Intel® Developer Zone.
- Install from containers or repositories: install the oneAPI toolkits from one of several supported containers or repositories.
- Pre-installed in the Intel® DevCloud: use a free development sandbox for access to the latest Intel SVMS hardware and select oneAPI tools.

About This Guide

This document provides:

- Chapter 2: An introduction to the oneAPI programming model (platform, execution, memory, and kernel programming)
- Chapter 3: Details about how to compile code for various accelerators (CPU, FPGA, etc.)
- Chapter 4: A description of the programming model with specifics about the Data Parallel C++ (DPC++) language options
- Chapter 5: A brief introduction to common APIs and related libraries
- Chapter 6: An overview of the software development process using various oneAPI tools, such as debuggers and performance analyzers, and optimizing code for a specific accelerator (CPU, FPGA, etc.)

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)

2 oneAPI 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(s).
- Execution model - Specifies the command queues and issuing commands for execution on the device(s).
- Memory model - Defines how the host and devices interact with memory.
- Kernel 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.

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;

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

  for (int i = 0; i<SIZE; ++i) {
```

```
    a[i] = i;
    b[i] = -i;
    c[i] = i;
  }

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

  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;
    }

  }

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

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

  d_queue.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];
    });

  });

  }

}
```

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 compute 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.

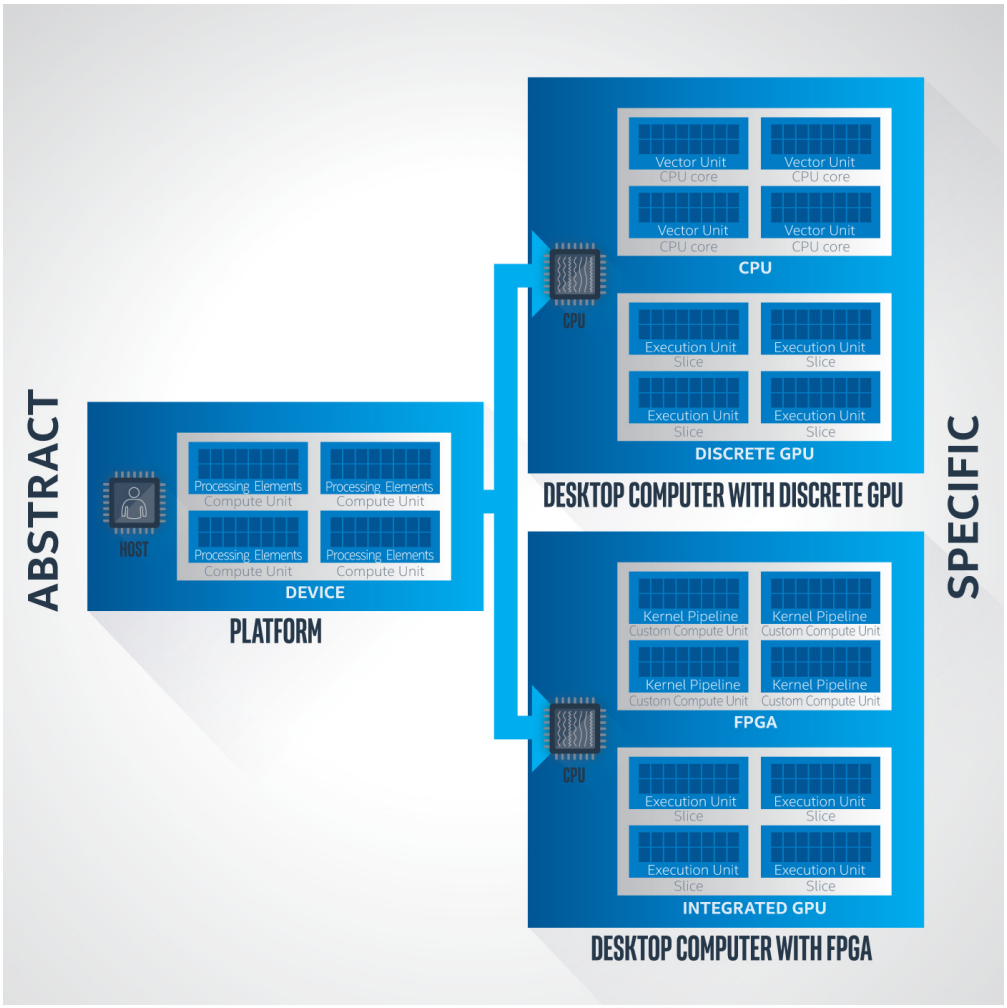
When compiled and executed, the sample program computes the 1024 element vector add in parallel on the accelerator. This assumes the accelerator has multiple compute elements capable of executing in parallel. This sample illustrates the models that the software developer will need to employ in their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

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.

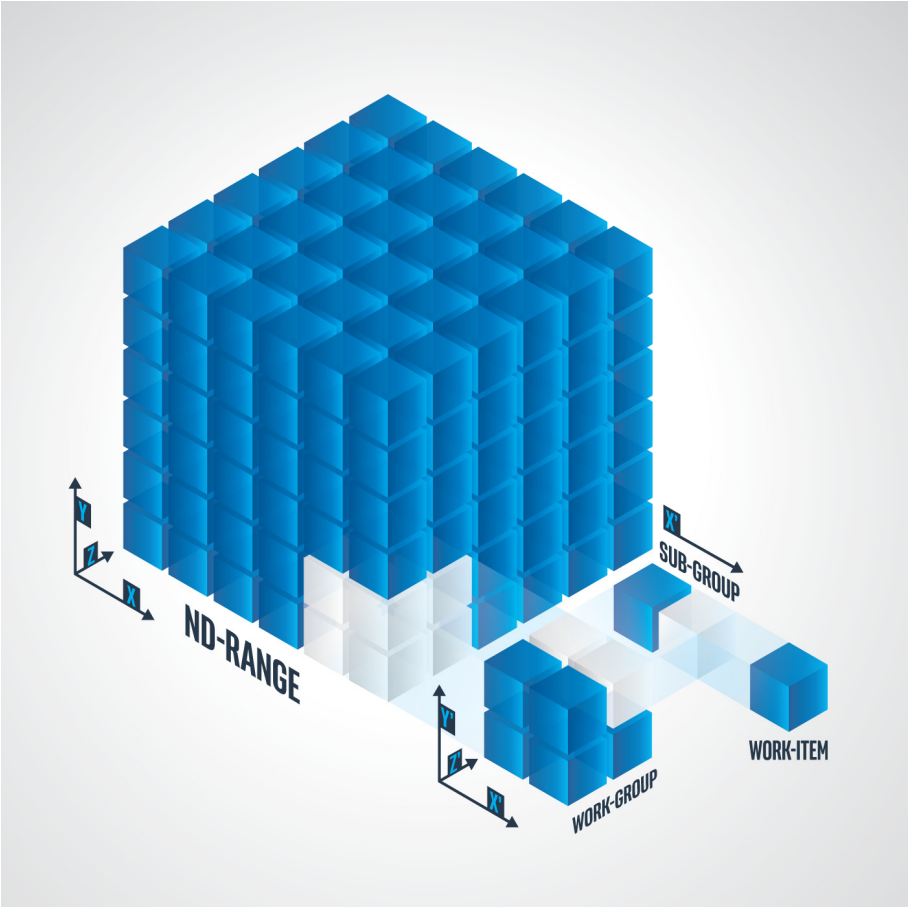
Execution Model

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-ranges, work-groups, sub-groups (Intel extension), and work-items, 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.

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));

  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.

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.

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.

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.

Target	Description
`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.

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.

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.

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;
    });
  });
  }

  {
  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);
  });
  }
}
```

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.

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);
  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;
}
}
```

3 Programming Interface

This chapter details the oneAPI compilation process across direct programming and API-based programming covering CPU, GPUs, and FPGAs. Some details about the tools employed at each stage of compilation are explained.

Single Source Compilation

The oneAPI programming model supports single source compilation. Single source compilation has several benefits compared to separate host and device code compilation. It should be noted that the oneAPI programming model also supports separate host and device code compilation as some users may prefer it. Advantages of the single source compilation model include:

- Usability – programmers need to create fewer files and can define device code right next to the call site in the host code.
- Extra safety – single source means one compiler can see the boundary code between host and device and the actual parameters generated by host compiler will match formal parameters of the kernel generated by the device compiler.
- Optimization - the device compiler can perform additional optimizations by knowing the context from which a kernel is invoked. For instance, the compiler may propagate some constants, infer pointer aliasing information across the function call.

Compiler Drivers

The Intel oneAPI DPC++ Compiler includes two compiler drivers:

- `dpcpp` is a GCC* compatible compiler driver. It recognizes GCC-style command line options (starting with "-") and can be useful for projects that share a build system across multiple operating systems.
- `dpcpp-cl` is a Microsoft* Visual C++ compatible driver. This driver is only available on Windows. It recognizes Windows command line options (starting with "/") and can be useful for Microsoft Visual Studio*-based projects.
- The examples in this guide use the `dpcpp` driver.

Example Compilation

The oneAPI tools are available in several convenient forms, as detailed in oneAPI Toolkit Distribution earlier in this guide. Follow the instructions in the Installation Guide to obtain and install the tools. Once the tools are installed and the environment is set, compile code for execution.

API-Based Code

The following code shows usage of an API call (`a * x + y`) employing the Intel oneAPI Math Kernel Library function `mkl::blas::axpy` to multiply `a` times `x` and add `y` across vectors of floating point numbers. It takes advantage of the oneAPI programming model to perform the addition on an accelerator.

```
#include <vector> // std::vector()
#include <cstdlib> // std::rand()
#include <CL/sycl.hpp>
#include "mkl_sycl.hpp"

int main(int argc, char* argv[]) {

    double alpha = 2.0;
    int n_elements = 1024;
```

```
int incx = 1;
std::vector<double> x;
x.resize(incx * n_elements);
for (int i=0; i<n_elements; i++)
    x[i*incx] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
    // rand value between -2.0 and 2.0

int incy = 3;
std::vector<double> y;
y.resize(incy * n_elements);
for (int i=0; i<n_elements; i++)
    y[i*incy] = 4.0 * double(std::rand()) / RAND_MAX - 2.0;
    // rand value between -2.0 and 2.0

cl::sycl::device my_dev;
try {
    my_dev = cl::sycl::device(cl::sycl::gpu_selector());
} catch (...) {
    std::cout << "Warning, failed at selecting gpu device.
     Continuing on default(host) device." << std::endl;
}

// Catch asynchronous exceptions
auto exception_handler = [] (cl::sycl::exception_list
    exceptions) {
    for (std::exception_ptr const& e : exceptions) {
        try {
            std::rethrow_exception(e);
         } catch(cl::sycl::exception const& e) {
            std::cout << "Caught asynchronous SYCL
        exception:\n"
            << e.what() << std::endl;
        }
    }
};

cl::sycl::queue my_queue(my_dev, exception_handler);

cl::sycl::buffer<double, 1> x_buffer(x.data(), x.size());
cl::sycl::buffer<double, 1> y_buffer(y.data(), y.size());

// perform y = alpha*x + y
try {
    mkl::blas::axpy(my_queue, n_elements, alpha, x_buffer,
    incx, y_buffer, incy);
}

catch(cl::sycl::exception const& e) {
    std::cout << "\t\tCaught synchronous SYCL exception:\n"
              << e.what() << std::endl;
}

// copy y_buffer data back to y vector
auto y_accessor = y_buffer.template
  get_access<cl::sycl::access::mode::read>();
for (int i=0; i<y.size(); ++i)
    y[i] = y_accessor[i];
```

```
    std::cout << "The axpy (y = alpha * x + y) computation is
        complete!" << std::endl;

    return 0;
}
```

To compile and build the application:

1. Ensure that the MKLROOT environment variable is set appropriately (`echo ${MKLROOT}`). If it is not set appropriately, source the `setvars.sh` script or run the `setvars.bat` script or set the variable to the folder that contains the `lib` and `include` folders. For more information about the `setvars.sh` script, see Get Started with Intel oneAPI Toolkits for Linux* or Get Started with Intel oneAPI Toolkits for Windows*.
2. Build the application using the following command:

```
dpcpp -I${MKLROOT}/include -Icommon  -DMKL_ILP64 -w -c axpy.cpp -o axpy.o
```

3. Link the application using the following command:

```
dpcpp -I${MKLROOT}/include -DMKL_ILP64 -w axpy.o -foffload-static-lib="${MKLROOT}/lib/intel64"/
libmkl_sycl.a -Wl,-export-dynamic -Wl,--start-group "${MKLROOT}/lib/intel64"/
libmkl_intel_ilp64.a "${MKLROOT}/lib/intel64"/libmkl_sequential.a "${MKLROOT}/lib/intel64"/
libmkl_core.a -Wl,--end-group -lsycl -lpthread -lm -ldl -o axpy.out
```

4. Run the application using the following command:

```
./axpy.out
```

Direct Programming

The vector addition sample code is employed in this example. It takes advantage of the oneAPI programming model to perform the addition on an accelerator.

The following command compiles and links the executable.

```
dpcpp vector_add.cpp
```

The components and function of the command and options are similar to those discussed in the API-Based Code section.

Execution of this command results in the creation of an executable file, which performs the vector addition when run.

Compilation Model

The command used to compile a program employing the oneAPI programming model is very similar to standard C/C++ compilation with standard compile and link steps. However, the compilation model supports code that executes on both a host and potentially several accelerators simultaneously. Thus, the commands issued by the compiler, linker, and other supporting tools as a result of compile and link steps are more complicated than standard C/C++ compilation targeting one architecture. The developer is protected from this complexity; however advanced users of the tools may want to understand these details to better target specific architectures.

A DPC++ program can consist of a set of source files, where each may have both host and device code. Compilation of the host code is somewhat straightforward as the target architecture is known. Typically, the host is an x86-based computer.

By default, the Intel oneAPI DPC++ Compiler compiles device code into a device-agnostic form that can run on any compatible devices. This is known as online compilation because the device-agnostic code gets compiled into a device-specific form at runtime, or when “online.” Additionally, DPC++ allows production of device-specific code at compile time. This process is known as offline compilation. Offline compilation for devices presents several challenges because of the need to target several architectures, some known and some unknown at the time of compilation. In addition, it may be helpful to apply aggressive optimization to a specific target. The compilation model supports these different needs. The compilation model enables:

- Target specific code - Target specific versions of a kernel function

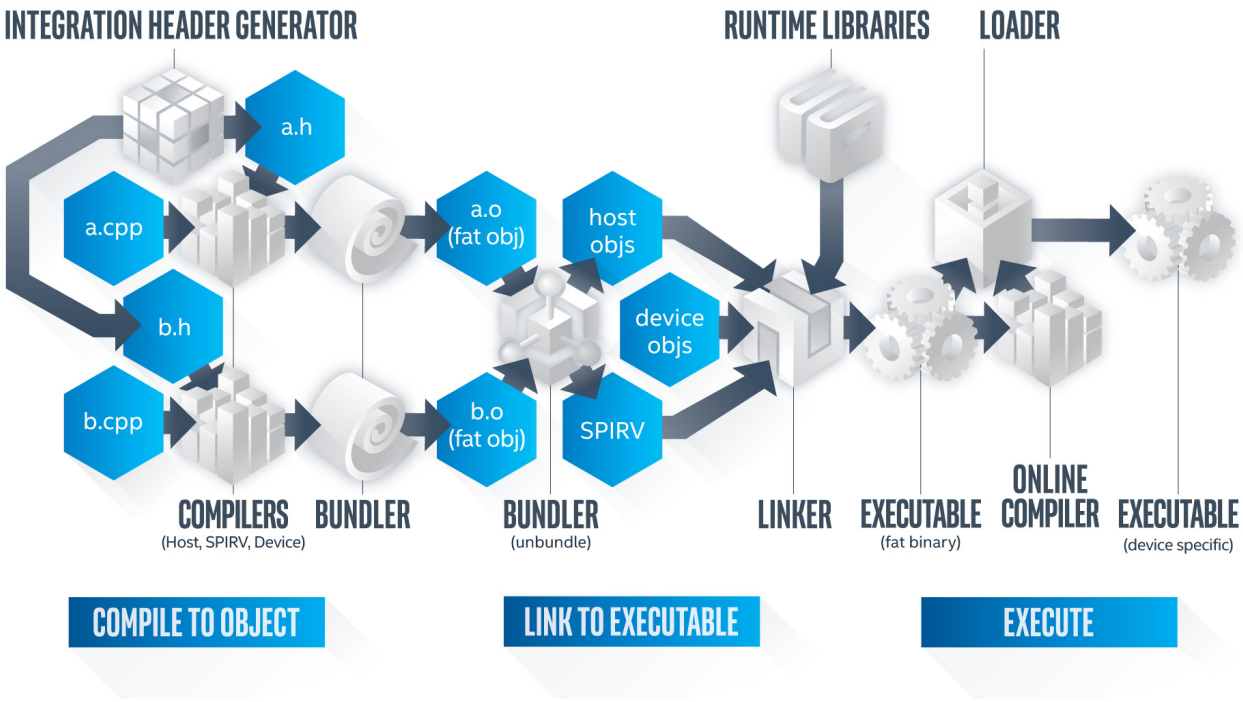
- Target specific optimization – Compiler optimization for a specific target
- General target support – Target a broad set of architectures
- Future target support - Target a new architecture. This could be the case in forward compatibility cases where a new architecture provides new optimization opportunities.

To support these needs, the code can be compiled into two main forms – a SPIR-V intermediate representation, which is the device-agnostic form, and target specific executable code.

- The SPIR-V representation enables online compilation, which is compilation to device code at execution time.
- Compilation to target specific executable code before executing the application is termed offline compilation. The Intel oneAPI DPC++ Compiler creates multiple code forms for each device.

These two forms are bundled together in an application binary known as a fat binary. The fat binary is employed by the oneAPI runtime when the application executed and form needed for the target device is determined at runtime. Device code for multiple devices co-exists in a single fat binary.

The following figure illustrates the compilation process from source code to fat binary.



The tools participating in the compilation are represented by rectangles in the figure and are described as follows:

- **Driver** – The executable name is `dpcpp` on Linux* and `dpcpp-cl` on Windows*. Invoke this file on the command line to start compilation. It orchestrates the compilation process and invokes other tools as needed.
- **Host compiler** – the DPC++ host compiler. It is possible to employ a third-party host compiler by invoking the tools manually.
- **Device compiler** – a DPC++ device compiler (there can be more than one), SPIR-V compiler by default. The output files have the extension `*.bc`. SPIR-V is produced after the device code linkage step.
- **SPIRV compiler** - **llvm-spirv** – standard LLVM tool distributed with the Intel® oneAPI Base Toolkit that converts LLVM IR bitcode to SPIR-V.
- **Bundler** – Marshalls host and device object code into a single file called a fat object.
- **Unbundler** – Unmarshalls fat objects back into their constituent host and device object code. It is a part of the bundler.
- **LLVM bitcode linker** – standard LLVM tool distributed with the Intel® oneAPI Base Toolkit that links LLVM IR bitcode files.

- **<Target> Back End** – a back end for the target <Target> used in the offline compilation scenario. Compiles SPIR-V to native device code form.
- **Platform linker** – the standard linker on the development platform – `ld` on Linux and `link.exe` on Windows.
- **Offload wrapper** - wraps device binaries (different from device object files) into a host object file linked with the final DPC++ executable (fat binary)

File Types:

- **Device object file** - device object code suitable for further linkage with other device objects. Can be LLVM bitcode or some native ISA.
- **Host object file** - usual object file containing host object code. Examples include COFF on Windows or ELF on Linux.
- **Fat object file** - a host object format file containing host and device code compiled for all targets passed via `-fsycl-targets`. Device code is inserted into special object file sections.
- **Fat binary file** - a host executable containing device binaries - either in SPIR-V or other form.

The compilation is a three-step process:

1. Compile to Object – The source code in the files `a.cpp` and `b.cpp` is parsed into an intermediate representation. Optimizations are invoked on the intermediate representation. The compiler command line specifies a list of targets for device object code generation or the default SPIR-V can be used. These objects are bundled together into a fat object. Two fat objects are produced - `a.o` and `b.o` (`a.obj` and `b.obj` on Windows)
2. Link to Executable – The fat objects are unbundled and linked together into a target specific image and a generic image. Additional optimizations may be invoked during this phase. The target specific image and generic image are grouped together into a fat binary.
3. Execute – The fat binary is loaded for execution. Once the program control flow arrives to the place where DPC++ kernel offload is invoked, the oneAPI runtime is called to determine if a target specific executable image exists. If it does, execution proceeds with the target specific executable image. If it does not, the generic image is loaded, then compiled online to produce the target specific image, then proceeds with the execution. If neither of the two exist, then the kernel executes on the host. Execution of the user application portion of the fat binary proceeds.

The starting point for any compilation involves the invocation of the compiler driver, `dpcpp`. The following compilation command compiles two source files, `a.cpp` and `b.cpp`, compiles, and links them into an executable:

```
dpcpp a.cpp b.cpp
```

This is the simplest compilation scenario where compilation and linkage are done in one step and the device code is delivered in device-agnostic form. More complex scenarios are described later.

Compile to Object Step

During the compile to object step, the source files are compiled and the output is the fat objects - one per input source. To support single source compilation, an integration header file is created for each source. In the compilation model figure above, these are named `a.h` and `b.h` respectively; however, in practice they will have a uniquely generated name to avoid conflict with any actual `a.h` or `b.h` file. The integration header enables the host code to invoke the device code and execute it on different devices with the necessary communication and coordination enabled through the oneAPI programming model.

Once the integration header is created, the compile to object step is performed by the appropriate compiler:

- Host Compiler - The host code is compiled by the host compiler. The host compiler is set by default during installation. The output is an object file that represents the host code. It includes the integration header to interoperate with the device code.
- SPIR-V device compiler – The device code is compiled by the SPIR-V compiler to generate a SPIR-V intermediate representation of the code. The output is a target-independent object file, which is a SPIR-V binary intermediate representation of the device code.
- Specific device compiler – The device code is compiled by a specific device compiler to generate a target specific object file. This is part of an offline compilation scenario. The target is specified using the `-Xsycl_targets` option.

Once the host object file and the specified target object files have been created, they are grouped together using the bundler tool into a fat object file. In the example command above, there is no specific device compiler employed; only the SPIR-V device compiler is invoked to create target independent object files. Note that when compilation and linkage are done in a single step, then no fat objects are created, thus avoiding bundling/unbundling overhead.

Link to Executable Step

The link to executable step transforms the fat object files into a fat binary. The actions taken during this step are very similar to the traditional link step of compiling for one target, such as x86:

- Object files are linked together, satisfying variable and function dependencies between the files
- Third-party library dependencies are satisfied by linking the specified libraries
- Compiler libraries and runtime routines are linked into the resulting executable.

There are some differences. Since the oneAPI programming model enables more devices to be employed and compilation to occur at more than one time, the additional steps employed during the link to executable step are as follows:

- Unbundling of the fat object files – Multiple target-specific and generic SPIR-V object files are bundled during the compile to object step. These fat object files are unbundled so that the correct target specific and generic SPIR-V object files can be grouped together for linking.
- Offline compilation – Compilation during the link to executable step can occur. This step is optional and is used in the offline compilation scenario.

  The option to request offline compilation (for example, for the Intel Processor Graphics included in 6th Generation Intel Processors) at this step is:

```
dpcpp -fsycl-targets=spir64_gen-unknown-linux-sycldevice -Xsycl-target-backend=spir64_gen-
unknown-linux-sycldevice "-device skl" src1.cpp src2.cpp
```

  The process of compiling the linked generic SPIR-V device binary into a target image is repeated for every device requested.
- Fat binary – The resulting host code and potentially multiple device images are linked together into a fat binary. This is the resulting executable that executes on the host device and then selectively executes the device kernels on the appropriate devices. A generic SPIR-V image may be included in the fat binary for use during the online compilation step (default if offline compilation is not requested).

Execute Step

During the execute step, the fat binary is obtained and loaded onto the system for execution. The fat binary is an operating system module (an executable or a dynamically loaded library) containing one or more device code binaries.

When a DPC++ application starts, its operating system modules are loaded into the process memory, and static initialization is performed for each. During this static initialization process, each module makes a call into the DPC++ runtime to register available device binary images. The DPC++ runtime bookmarks the registered binaries and uses them to resolve kernel launch requests to the appropriate device kernels.

To target new devices, the oneAPI programming model supports online compilation. If a device code binary is not available during a kernel launch request, an online compilation may be requested.

DPC++ applications may also dynamically compile and run OpenCL kernels provided in the source form. The main APIs for this feature are:

- `cl::sycl::program::build_with_source` to dynamically create a kernel from a string containing OpenCL code.
- `cl::sycl::program::get_kernel` to obtain the DPC++ kernel instance, which can be run on a device.

See the DPC++ Specification for more details about the APIs.

Online Compilation

Online compilation is the compilation of the kernel code during the execute step. This mode of execution can have significant performance benefits, as the compiler may generate more efficient code. As an example, the device kernel may take advantage of a larger SIMD width made available on the underlying hardware it executes on, or some kernel parameters, such as data tile size or loop count, may be set to specific constants to help when applying typical compiler optimizations like loop unrolling.

Online compilation results in compiling the SPIR-V binary image to native device code. SPIR-V is the default portable device code format, which can run on any compatible device.

The oneAPI runtime does not perform the online compilation. Instead, it requests underlying runtimes for the requested devices to perform the compilation job via lower-level interfaces. For example, it will ultimately use the `clCreateProgramWithIL` API on an OpenCL device to take SPIR-V as input and compile it to native code.

Offline-compiled device code is injected into the runtime system though a different API. In the OpenCL platform case, the API is `clCreateProgramWithBinary`.

Online compilation of a SPIR-V binary image embedded into the fat binary is usually triggered when one of the kernels constituting the module is requested to run. The request `cl::sycl::handler::parallel_for` always has information about the device where the kernel is to be executed; hence the runtime knows what device(s) to target. When one kernel is requested to run, the entire SPIR-V module it belongs to is compiled.

CPU Flow

Use of a CPU is recommended for use cases with branch operations, instruction pipelining, dynamic context switch, and so on.

DPC++ supports online and offline compilation modes for the CPU target. Online compilation is the same as for all other targets.

Example CPU Commands

The commands below implement the scenario when part of the device code resides in a static library.

Produce a fat object with device code:

```
dpcpp -c static_lib.cpp
```

Create a fat static library out of it using the `ar` tool:

```
ar cr libstlib.a static_lib.o
```

Compile application sources:

```
dpcpp -c a.cpp
```

Link the application with the static library:

```
dpcpp -foffload-static-lib=libstlib.a a.o -o a.exe
```

Online Compilation for CPU

No specifics for CPU target. The command below produces a fat binary with a SPIR-V image, which can be run with online compilation on any compatible device, including a CPU.

```
dpcpp a.cpp b.cpp -o app.out
```

Offline Compilation for CPU

NOTE This is an experimental feature with limited functionality.

Use this command to produce `app.out`, which only runs on an x86 device.

```
dpcpp -fsycl-targets=spir64_x86_64-unknown-linux-sycldevice a.cpp b.cpp -o app.out
```

Optimization Flags for CPU Architectures

In offline compilation mode, optimization flags can be used to produce code aimed to run better on a specific CPU architecture. Those are passed via the `-Xsycl-target-backend dpcpp` option:

```
dpcpp -fsycl-targets=spir64_x86_64-unknown-linux-sycldevice -Xsycl-target-backend=spir64_x86_64-
unknown-linux-sycldevice "<CPU optimization flags>" a.cpp b.cpp -o app.out
```

Supported CPU optimization flags are:

```
-simd=<instruction_set_arch> Set target instruction set architecture:
'sse42' for Intel(R) Streaming SIMD Extensions 4.2
'avx' for Intel(R) Advanced Vector Extensions
'avx2' for Intel(R) Advanced Vector Extensions 2
'skx' for Intel(R) Advanced Vector Extensions 512
```

NOTE The set of supported optimization flags may be changed in future releases.

Host and Kernel Interaction on CPU

Host code interacts with device code through kernel parameters and data buffers represented with `cl::sycl::accessor` objects or `cl_mem` objects for OpenCL data buffers.

GPU Flow

The GPU Flow is like the CPU flow except that different back ends and target triples are used.

Target triple for GPU offline compiler is `spir64_gen-unknown-linux-sycldevice`.

NOTE GPU offline compilation currently requires an additional option, which specifies the desired GPU architecture.

Example GPU Commands

The examples below illustrate how to create and use static libraries with device code on Linux.

Produce a fat object with device code:

```
dpcpp -c static_lib.cpp
```

Create a fat static library out of it using the ar tool:

```
ar cr libstlib.a static_lib.o
```

Compile application sources:

```
dpcpp -c a.cpp
```

Link the application with the static library:

```
dpcpp -foffload-static-lib=libstlib.a a.o -o a.exe
```

Offline Compilation for GPU

NOTE This is an experimental feature with limited functionality.

The following example command produces `app.out` for a specific GPU target:

```
dpcpp -fsycl-targets=spir64_gen-unknown-linux-sycldevice -Xsycl-target-backend=spir64_gen-
unknown-linux-sycldevice "-device skl" a.cpp b.cpp -o app.out
```

FPGA Flow

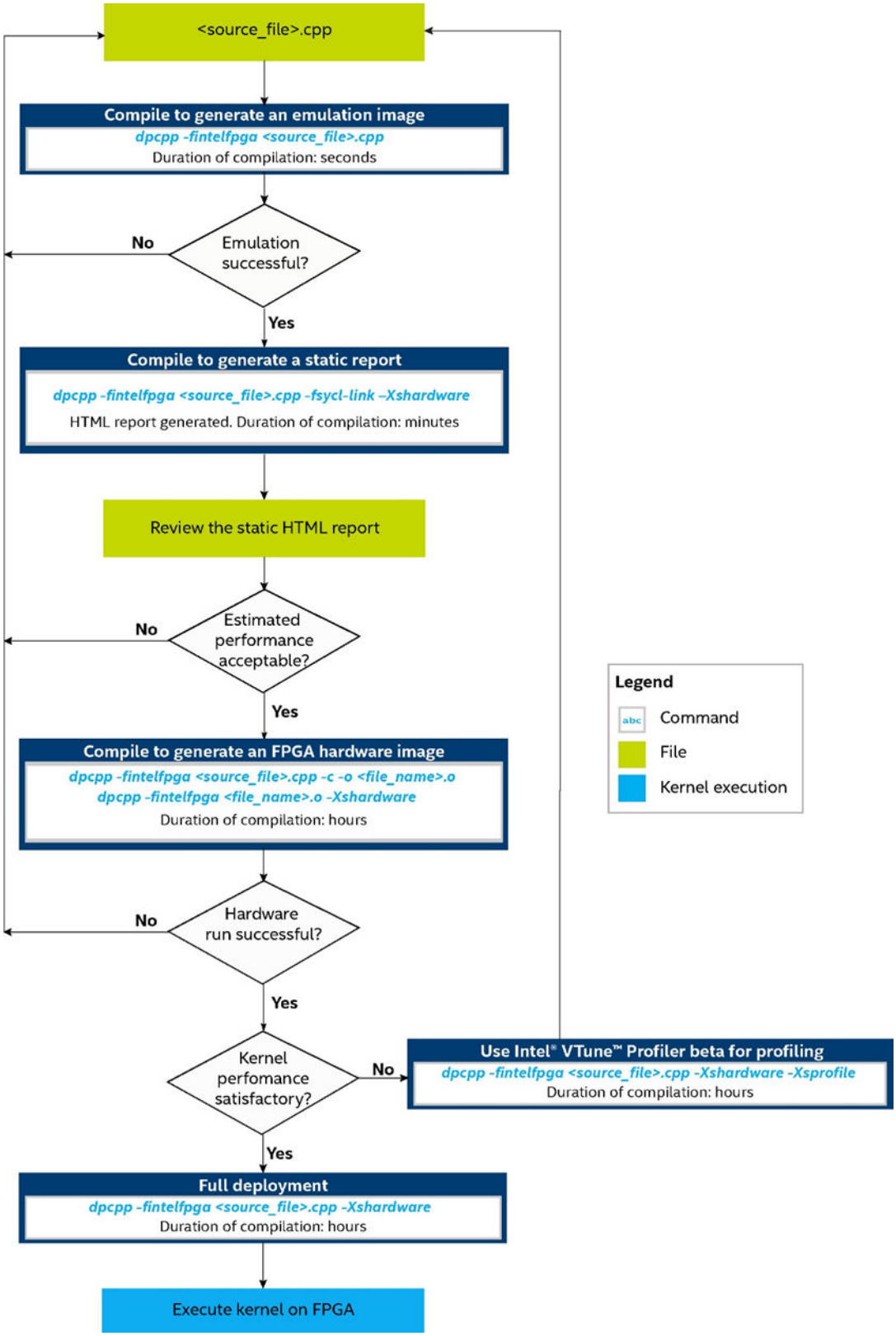
Field-programmable gate arrays (FPGAs) are integrated circuits that can be configured repeatedly to perform an infinite number of functions.

The key benefit of using FPGAs for algorithm acceleration is that they support wide, heterogeneous, and unique pipeline implementations. This characteristic contrasts with many different types of processing units such as symmetric multiprocessors, digital signal processors (a special-purpose processor), and graphics processing units (GPUs). In these types of devices, parallelism is achieved by replicating the same generic computation hardware multiple times.

For general Intel oneAPI DPC++ Compiler use, FPGA compilation, or design flow of device-specific code is special in the following ways:

- FPGAs support only the offline compilation mode and provide two device compilation stages to help iterate on a program. For more information, see FPGA Offline Compilation.
- FPGA devices support two image types, each serving a different purpose. For more information, see FPGA Device Image Types.

The following diagram shows a typical compilation flow for FPGA devices.



The emulation image verifies the code correctness and only takes seconds to complete. This is the recommended first step in compiling code. Next, the static report helps determine whether the estimated kernel performance data is acceptable. After generating a hardware image, use Intel® VTune™ Profiler to collect key profiling data from hardware runs.

For information about emulation and hardware images, see FPGA Device Image Types. For more information about the static report or collecting profiling data, see the Intel oneAPI DPC++ FPGA Optimization Guide.

Example FPGA Commands

Generating an emulation image:

```
dpcpp -fintelfpga <source_file>.cpp
```

Generating a static report:

```
dpcpp -fintelfpga <source_file>.cpp -fsycl-link -Xshardware
```

Generating an FPGA hardware image (Linux only):

```
dpcpp -fintelfpga <source_file>.cpp -c -o <file_name>.o
```

```
dpcpp -fintelfpga <file_name>.o -Xshardware
```

Running an FPGA image on the Intel® Programmable Acceleration Card (PAC) with Intel® Arria® 10 GX FPGA:

```
./<fpga_hw_image_file_name>
```

For more information about command syntax, see FPGA Offline Compilation.

Offline Compilation for FPGA

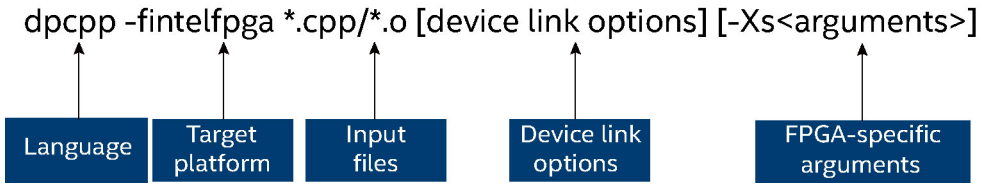
NOTE FPGA devices do not support online compilation.

NOTE While all compilation flows (emulation, report, and hardware) are supported on Linux, only the emulation flow is supported on Windows.

As shown in the previous figure, when targeting an FPGA device, an FPGA target compiler is injected into the device generation phase.

- In the first FPGA compilation stage, the FPGA target compiler synthesizes the high-level DPC++ code to RTL-level design abstraction. This process usually takes minutes to complete. At the end of this stage, the compiler generates a high-level design HTML report that contains most of the information required to analyze and iterate the design.
- In the second FPGA compilation stage, the FPGA target compiler analyzes and synthesizes the RTL-level design abstraction to an FPGA configuration bitstream. This process takes a long time (usually in hours) to complete. View accurate FPGA frequency and resource utilization results after this process completes.

The most common case for FPGA compilation is targeting a single device. Use the following syntax for compiling an FPGA device:



NOTE On Windows 7 or Windows 8.1 systems, ensure that the combined length of the source or output file name and the corresponding file path does not exceed 260 characters. If the combined length of the source or output file name and its file path exceeds 260 characters, the offline compiler generates the following error message: `Error: failed to move <work_directory> to <project_directory>` where `<work_directory>` is used for compilation and `<project_directory>` is the folder storing the reports and FPGA related files generated during the compilation. Windows 10 systems do not have the 260-character path limit.

Examples:

- Use the following command to compile for emulation:

```
dpcpp -fintelfpga *.cpp
```

NOTE Remove the project directory `<file_name>.proj` (if it exists) before executing the offline compilation command, where `<file_name>` is the name of output binary executable name.

- Linux only: Use the following command to generate the high-level design report:

```
dpcpp -fintelfpga *.cpp -fsycl-link -Xshardware
```

- Linux only: Use the following command to compile for hardware:

```
dpcpp –fintelfpga *.cpp -c
dpcpp –fintelfpga *.o -Xshardware
```

When compiling for an FPGA hardware image, a `*.d` file and a `*.prj` directory are generated.

- Run the FPGA hardware image using the `./<fpga_hw_image_file_name>` command.
- The `*.prj` directory contains reports that describe the generated hardware. Use the reports for performance tuning.
- The `*.d` file is used to map the generated code to the source to facilitate the reports.

Tip
For additional information, refer to the FPGA tutorial sample "Compile Flow" listed in the Intel oneAPI Samples Browser.

When compiling an FPGA hardware image, the Intel oneAPI DPC++ Compiler provides checkpoints to inspect errors and modify source code without performing a full compilation on each iteration. The following table summarizes the checkpoints:

Checkpoint	File Extension	Flow Type	Compile Time From Source	Capabilities Available
Object files	• *.o • *.obj	• Emulation • Hardware	In seconds	Detect compiler-parsing errors. For example, syntax.
FPGA early image object	• *.a • *.lib **NOTE** *.a and *.lib files contain all FPGA device and host code in corresponding files.	Hardware	In minutes	View the report that the Intel oneAPI DPC ++ Compiler generates.

Checkpoint	File Extension	Flow Type	Compile Time From Source	Capabilities Available
FPGA image object	• *.a • *.lib **NOTE** *.a and *.lib files contain all FPGA device and host code in corresponding files.	Hardware	In hours	The Intel oneAPI DPC ++ Compiler generates a complete FPGA image. This stage of compilation is time-consuming because mapping a fully custom computation pipeline onto FPGA's resources is a compute-intensive optimization problem. Refer to FPGA Device Link for more information.
Executable	Program executable	• Emulation • Hardware	• In seconds for emulation flow • In hours for hardware flow.	Refer to FPGA Device Image Types for more information about emulation and hardware images.

Specify Target Platform (-fintelfpga)

Use the `-fintelfpga` flag to target a single FPGA device. It sets the default compiler settings, including the optimization level, debug data, linking libraries, and so on.

FPGA-Specific Arguments (-Xs)

The `-Xs` flag passes the FPGA-specific arguments. It does not require a target name. When there are multiple targets, arguments are passed to all targets.

> **NOTE** Targeting multiple FPGAs is not supported in a single compile.

There are two ways of using the –Xs argument:

- Standalone flag to apply to the subsequent flag. For example:

```
dpcpp -fintelfpga hello.cpp -Xs -fp-relaxed -Xs -fpc
dpcpp -fintelfpga hello.cpp -Xs "-fp-relaxed -fpc"
```

- Prefix to a platform compiler flag. For example:

```
dpcpp -fintelfpga hello.cpp -Xsfp-relaxed -Xsfpc
```

where, `-fp-relaxed` and `-fpc` flags are passed to the FPGA target compiler. For more information about these flags, see the Intel oneAPI DPC++ FPGA Optimization Guide.

Use one of the following flags to specify the type of FPGA image:

FPGA Device Image Type	Arguments
Emulation (default)	N/A
Hardware	-Xshardware

FPGA Device Link

In the default case (no device link option specified), Intel oneAPI DPC++ Compiler handles the host generation, device image generation, and final executable linking as follows:

```
dpcpp -fintelfpga hello.cpp -Xshardware
```

When generating a hardware image, use a device link to only compile the device portion and choose FPGA compilation checkpoints. Input files for the device link command must contain all device parts for an FPGA image generation.

Advantages of using a device link:

- Fast development iteration cycle (minutes to generate report vs hours to generate the hardware).
- Separation of the device code compilation from the host code compilation.
  - If only the host code is changed, recompile only the host code.
  - If the DPC++ program is partitioned into separate host code and device code and modifications are made only on the host code, reuse the FPGA early image object or FPGA image object for the device code.

Tip
Use FPGA early image and FPGA image objects to link and save compile time.

Use one of the following device link options:

Device Link Option	Description
`-fsycl-link[=early]`	Default case. Generates an FPGA early image object and a HTML report. **NOTE** Since this is the default option, specifying `[=early]` is optional.
`-fsycl-link=image`	Generates a complete FPGA hardware image to use for linking.

Examples:

- Use the following command to generate an HTML report:

```
dpcpp -fintelfpga -fsycl-link[=early] -o dev_early.a
*.cpp/*.o -Xshardware
```

- Use the following command to generate an FPGA image for hardware:

```
dpcpp -fintelfpga -fsycl-link=image -o dev.a *.cpp/*.o
-Xshardware
```

- Use the following command to generate an FPGA image for hardware from an FPGA early image object:

```
dpcpp -fintelfpga -fsycl-link=image -o dev.a
dev_early.a -Xshardware
```

Example 1 - Recompiling the Host Program

A program is partitioned into multiple source files as `a.cpp`, `b.cpp`, `main.cpp`, and `util.cpp`. Only `a.cpp` and `b.cpp` files contain the device code. The following example shows how to save compile time by recompiling only the host program:

1. Generate a *.o (an FPGA image object) file by running the following command:

```
dpcpp -fintelfpga a.cpp b.cpp -fsycl-link=image
-o dev.a -Xshardware.
```

> **NOTE** This command takes hours to complete.

2. Compile and link the host part by running the following commands:

```
dpcpp -fintelfpga main.cpp -c -o main.o
dpcpp -fintelfpga util.cpp -c -o util.o
dpcpp -fintelfpga dev.a main.o util.o -o a.out
```

Where, `a.out` is the executable.

If modifications are made to only `main.cpp` and `util.cpp` files, then rerun only the commands from step 3 that complete in a few seconds.

Example 2 - Separating FPGA Compilation and Host Linking

The following example shows how to save compile time when recompiling the application with different flags or parameters, but unchanged device code:

1. Compile the source application to a *.o file.

```
dpcpp -fintelfpga -c a.cpp -o a.o
```

2. Generate a `*.a` (FPGA image object) file.

```
dpcpp -fintelfpga a.o -fsycl-link=image -o dev.a -Xshardware
```

> **NOTE** This command takes hours to complete.

3. Compile and link to the executable.

```
dpcpp -fintelfpga dev.a -o a.out
```

If linking issues are encountered in step 3, re-run only that command with proper linking flags and it will complete in few seconds.

> **Tip**
> For additional information, refer to the FPGA tutorial sample "Device Link" listed in the Intel oneAPI Samples Browser.

FPGA Device Image Types

An FPGA device image contains a program or bitstream required to run on an FPGA. Unlike other devices, FPGA devices support the following image types:

Image Type	Purpose	Toolkit Requirements	Compile Time
Emulation	Verifies the code correctness.	Use this mode if the Intel oneAPI Base Toolkit is installed.	Compilation completes in few seconds.

Image Type	Purpose	Toolkit Requirements	Compile Time
Report	Generates a static optimization report for design analysis. When completed, reports are available in `<project_name>.prj/reports/report.html`. For more information about the reports, refer to the Intel oneAPI DPC++ FPGA Optimization Guide.	Use this mode if the Intel oneAPI Base Toolkit is installed.	Compilation completes in a few minutes.
Hardware (Linux only)	Generates the actual bitstream on an FPGA device.	Use this mode if the Intel® FPGA Add-On for oneAPI Base Toolkit (Beta) is installed.	Compilation completes in few hours.

Targeting Multiple FPGAs

Not supported for oneAPI Beta release.

Other Supported Intel oneAPI DPC++ Compiler Options for FPGA

The Intel oneAPI DPC++ Compiler offers a list of options that allow you to customize the kernel compilation process.

The following table provides a summary of all options supported by the Intel oneAPI DPC++ Compiler:

Option name	Description
`-Xsv`	Generates a report on the progress of the compilation.
`-Xsemulator`	Generates an emulator device image.
`-Xshardware`	Generates a hardware device image.

For more information about the FPGA optimization flags, refer to the Intel oneAPI DPC++ FPGA Optimization Guide.

FPGA Device Selection in the Host Code

To explicitly set an FPGA emulator device or FPGA hardware device, include the following header file in the host code:

```
CL/sycl/intel/fpga_extensions.hpp
```

Declare the device selector type as one of the following:

- For the FPGA emulator, use the following:

```
intel::fpga_emulator_selector device_selector;
```

- For the FPGA hardware device, use the following:

```
intel::fpga_selector device_selector;
```

Consider the following sample code:

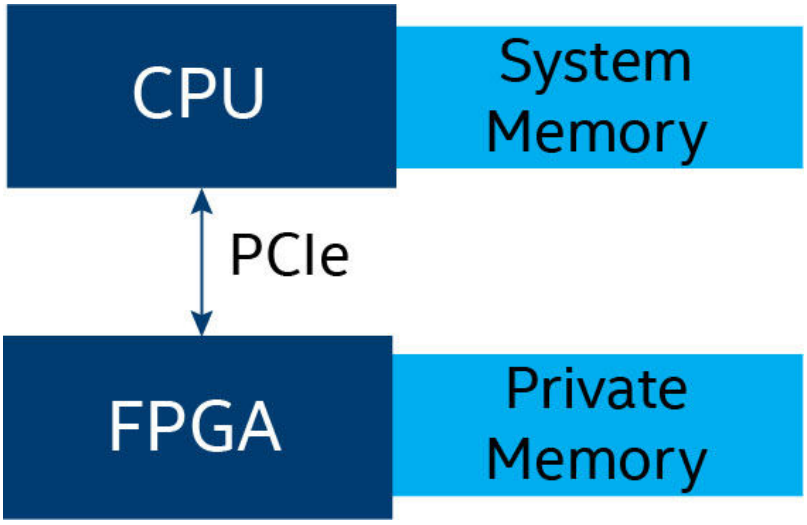
```
#include <CL/sycl/intel/fpga_extensions.hpp>
using namespace cl::sycl;
...
    #ifdef FPGA_EMULATOR
    intel::fpga_emulator_selector device_selector;
    #else
    intel::fpga_selector device_selector;
    #endif
    queue deviceQueue(device_selector);
...
```

NOTE

- If runtime cannot find `fpga_emulator_selector` or `fpga_selector` in the host code, an error message is displayed.
- To enable debugging of kernel code, optimizations are disabled by default for the FPGA emulator device. This can lead to suboptimal execution speed when emulating kernel code. Pass the `-Xsg0` compiler option to disable debugging and enable optimizations leading to faster emulator execution.

Host and Kernel Interaction on FPGA

FPGA devices typically communicate with the host (CPU) via I/O, such as PCIe.



A PCIe-connected FPGA typically has its own private DDR on which it primarily operates. Prior to executing a kernel on the FPGA reconfigurable logic, the CPU must first bulk transfer (over dynamic memory access (DMA)) all data that the kernel needs to access into the FPGA's local DDR memory. When the kernel completes its operations, it must transfer the results over DMA back to the CPU. The transfer speed is bound by the PCIe link itself and the efficiency of the DMA solution, for example, on the Intel® Arria® 10 programmable acceleration card (PAC), which has a PCIe Gen 3 x 8 link, transfers are typically 6 to 7 GB/s. For more information, refer to the Intel oneAPI DPC++ FPGA Optimization Guide, which highlights techniques for eliminating unnecessary transfers (for example, by correctly tagging read-only or write-only data).

Further, improve system efficiency by maximizing the number of concurrent operations. For example, because PCIe supports simultaneous transfers in opposite directions and PCIe transfers do not interfere with kernel execution, aim to have device-to-host transfers, host-to-device transfers, and kernel executions all executing simultaneously to maximize FPGA device utilization. For more information, refer to the Intel oneAPI DPC++ FPGA Optimization Guide, which describes techniques, such as double buffering, to help improve system efficiency.

Configuring the FPGA reconfigurable logic is a lengthy operation requiring several seconds of communication with the FPGA device. The runtime automatically manages this configuration, which might occur when a kernel launches at initialization, or might not occur if the same programming file has already been loaded onto the device. To manage this variability and when measuring performance, perform a warm-up that executes one kernel prior to taking any time measurements in a program.

Kernels on an FPGA consume reconfigurable logic and can service only one invocation of that kernel at a time. A program that launches multiple invocations of the same kernel instance requires the runtime to execute those invocations sequentially on the same kernel logic.

For more information, see:

- Intel oneAPI DPC++ FPGA Optimization Guide
- Intel® oneAPI DPC++ FPGA Workflows on Third-Party IDEs
- https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx/overview.html
- https://en.wikipedia.org/wiki/Direct_memory_access
- https://en.wikipedia.org/wiki/PCI_Express
- https://en.wikipedia.org/wiki/DDR4_SDRAM

FPGA Workflows in IDEs

The oneAPI tools integrate with third-party integrated development environments (IDEs) on Linux (Eclipse*) and Windows (Visual Studio*) to provide a seamless GUI experience for software development. See Intel oneAPI DPC++ FPGA Workflows on Third-Party IDEs for more details.

Complex Scenario: Use of Static Libraries

The Intel oneAPI DPC++ Compiler supports use of static libraries, which contain host and device compiled objects.

Create the Static Library

Prepare the compiler and compile the source as described in the Get Started Guide.

```
dpcpp app1.cpp app2.cpp -c
```

On Linux: `arcrlibapp.a app1.o app2.o`

On Windows: `lib -out:libapp.lib app1.obj app2.obj`

Use the Static Library

After creating the library, use it when creating the final application. A fat library is treated differently than a regular library. The `-foffload-static-lib` option is used to signify the necessary behavior.

On Linux: `dpcpp main.cpp -foffload-static-lib=libapp.a`

On Windows: `dpcpp main.cpp -foffload-static-lib=libapp.lib`

Standard Intel oneAPI DPC++ Compiler Options

The following standard options are available. The options are not separated by host only and device only compilation.

Option	Description
`-fsycl-targets=T1,...,Tn`	Makes Intel oneAPI DPC++ Compiler generate code for devices represented by comma-separated list of triples. Some triples can represent offline compilation.

Option	Description
`-foffload-static-lib=<lib>`	Link with fat static library. Link with `<lib>`, which is a fat static archive containing fat objects that correspond to the target device. When linking, `clang` will extract the device code from the objects contained in the library and link it with other device objects coming from the individual fat objects passed on the command line. **NOTE** Any libraries that are passed on the command line which are not specified with `–foffload-static-lib` are treated as host libraries and are only used during the final host link.
`-fsycl-device-only`	Generate a device only binary.
`-Xsycl-target-backend=T "options"`	Specifies options that are passed to the backend in the device compilation tool chain for target T. • Offline compilation: device compiler for the target T. • Online compilation: device compiler for the target T to use for runtime compilation. If needed, other tools participating in the compilation and linkage can be customized further. To do that, `backend` in the option name should be replaced with tool ID like this: `–Xsycl-target-<toolid>=T`. The additional tool IDs include: frontend — The front-end+middle-end of the SPIRV-based device compiler for target T. The middle end is the part of a SPIRV-based device compiler which generates SPIRV. This SPIRV is then passed by the clang driver to the T's back-end. linker — Device code linker for target T. Some targets may have frontend and backend in one component; in that case options are merged.
`-fsycl-link`	Perform a partial link of device binaries. This is then wrapped by the offload wrapper, allowing the device binaries to be linked by the host compiler/linker.
`-fsycl-link=<value>`	Perform a partial link of device binaries, to be used with FPGA. The <value> can be of the following: early — Generating html report at this point. User can stop here and iterate on their program. Usually takes minutes to generate. User can also resume from this point and generate FPGA image.

Option	Description
	image — Generating FPGA bitstream which is ready to be linked and used on a FPGA board. Usually takes hours to generate.
`-fintelfpga`	Option specific to performing ahead of time compilation with FPGA. Functionally equivalent to using `-fsycltargets=spir64-unknown-<os>-sycldevice`, adding compiling with dependency and debug information enabled.
`-Xs "options"-Xs<arg>`	Similar to `-Xsycl-target-backend`, passing "options" to the backend tool. The `-Xs<arg>` variant works as follows: `-Xshardware → -Xsycl-target-backend "-hardware"`

Data Parallel C++ (DPC++) Programming Language and Runtime

4

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.

C++ Version Support

The oneAPI Programming Model section 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
```

Header Files and Namespaces

The compiler header files are in a subdirectory of the Intel oneAPI DPC++ Compiler installation. For example:

```
<install_dir>/compiler/latest/<os>/lib/
```

where `<install_dir>` is the directory where the Intel oneAPI Base Toolkit is installed and `<os>` is either `windows` or `linux`.

Within the header files, the names correspond with the primary DPC++ classes for implementing the language.

The Intel-specific extensions to SYCL* that DPC++ is based upon are located in the following location:

On Windows: `<install_dir>\compiler\latest\windows\lib\clang\10.0.0\include\CL\sycl\intel`

On Linux: `<install_dir>/compiler/latest/linux/compiler/lib/clang/10.0.0/include/CL/sycl/intel`

DPC++ Classes, Class Templates, and Defines

The following table summarizes the DPC++ classes, class templates, and defines of the programming language. In addition, the specific include file for the definition of these items are mentioned along with a description of each.

Class	Include file(s)	Description
Accessor	accessor.hpp, accessor2.hpp	Enables and specifies access to buffers, images, and device local memory
Atomic	atomic.hpp	Operations and member functions to guarantee synchronized access to data values
Buffer	buffer.hpp	Encapsulates memory for communication between host and device(s)
Built-in functions	builtin.hpp	Math functions that can be employed efficiently across host and devices
Command Group Handler	handler.hpp, handler2.hpp	Encapsulates commands executed in command group scope
Context	context.hpp	Encapsulates a platform and collection of devices
Device	device.hpp	Encapsulates attributes, features, and capabilities of an accelerator
Device event	device_event.hpp	Encapsulates functions used to wait for other operations
Device selector	device_selector.hpp	Enables selection of a device for execution of command groups
Event	event.hpp	Encapsulates objects used to coordinate memory transfers and kernel execution
Exception	exception.hpp	Notifications used by the program to handle outlier occurrences such as error conditions.
Group	group.hpp	Template class encapsulates work-groups
Id	id.hpp	Encapsulates a vector of dimensions for accessing individual items in a global or local range
Image	image.hpp	Encapsulates memory for communication between host and device(s) in an optimized format
Item	item.hpp	Encapsulates a function object executing within a range
Kernel	kernel.hpp	Encapsulates a kernel object which is the function that executes on the device
Multi-pointer	multi_ptr.hpp	Abstraction around low-level pointer to enable pointer like access across host and devices

Class	Include file(s)	Description
Nd_item	nd_item.hpp	Encapsulates each work-item in an ND-range
Nd_range	nd_range.hpp	Encapsulates the index space for 1-, 2-, or 3- dimensional data
Platform	platform.hpp	Encapsulates the oneAPI host and devices on the system
Program	program.hpp	Encompasses a program that employs the oneAPI programming model, communicating if source code is compiled or linked
Property Interface	property_list.hpp	Enables passing extra parameters to buffer, image, and device classes
Queue	queue.hpp	Object and methods for executing command queues
Range	range.hpp	Encapsulates the iteration domain of a work-group in 1-, 2-, or 3-dimensions
Standard library classes	stl.hpp	Interfaces for C++ standard classes
Stream		Methods for outputting oneAPI data types
Vec and Swizzled Vec	types.hpp	Vec class representing a vector of data elements. Swizzled vec class enables selection of combinations of elements in a vec object.
Version	version.hpp	Defines compiler version

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.

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.

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.

The tables in the Memory Model section summarize the access modes and access targets allowed.

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

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

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 `multi_ptr`. 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.

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

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.

Context

A context encapsulates a platform and a collection of devices. For more information, see the Platform Model section.

The class contains multiple constructors enabling creation from platforms, devices, and context as arguments.

The class contains member functions for querying information about the instantiated context:

- `get_platform()` – obtain the platform associated with the context
- `get_device()` – obtain the device associated with the context
- `is_host()` – return true if context is a host context

Device

The `device` class represents the capabilities of the accelerators in a system, as detailed in the Execution Model section. The class contains member functions for constructing devices and obtaining information about the device. One form of constructor requires zero arguments. The constructor can also take a device selector argument that chooses which type of accelerator to employ, such as CPU, GPU, or FPGA. Lastly, construction can be via OpenCL software technology by code using `cl_device_id` for interoperability.

The device class contains member functions for querying information about the device, which is useful for DPC++ programs where multiple devices are created. Some calls return basic information, such as `is_host()`, `is_cpu()`, `is_gpu()`. For more detailed information, the function `get_info` sets a series of attributes with pertinent information about the device including:

- The local and global work item IDs
- The preferred width for built in types, native ISA types, and clock frequency
- Cache width and sizes
- Device support attributes, such as unified memory support, endianness, and (if the device is online) compiler capable
- Name, vendor, and version of the device

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.

Device Selector

The DPC++ `device_selector` class enables the runtime selection of a particular device to execute kernels based upon user-provided heuristics. Construction is either via a zero argument constructor or by providing a reference to another `device_selector` object. An instance of a `device_selector` can also be assigned to another instance.

The following code sample shows use of the standard `device_selectors` and a derived `device_selector` that employs a device selector heuristic. In the example, the selected device prioritizes a CPU device because the integer rating returned is higher than if the device is a GPU or other accelerator.

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

class my_device_selector : public device_selector {
public:
  int operator()(const device &dev) const override{
    int rating = 0;
    if (dev.is_gpu()) rating = 1;
    else if (dev.is_cpu()) rating = 2;
    return rating;
};
};
int main() {

  default_selector d_selector;
  queue Queue(d_selector);

  cpu_selector d_selector2;
  queue Queue2(d_selector2);

  std::cout << "Executing on "
    << Queue.get_device().get_info<info::device::name>()
    << std::endl;
  std::cout << "Executing on "
    << Queue2.get_device().get_info<info::device::name>()
    << std::endl;

  device device1 = Queue.get_device() ;
  device device2 = Queue2.get_device() ;

  my_device_selector h_selector;
  queue Queue3(h_selector);

  std::cout << "Executing on "
    << Queue3.get_device().get_info<info::device::name>()
    << std::endl;

  return 0;
}
```

A brief description of the key lines in the sample code is as follows:

- Lines 15-16 - use the `default_selector` to select a device to initialize a command queue

```
15    default_selector d_selector;
16    queue Queue(d_selector);
```

- Lines 18-19 - use the `cpu_selector` to select a device to initialize a command queue

```
18    cpu_selector d_selector2;
19    queue Queue2(d_selector2);
```

- Lines 4-11 and lines 31-32 - use a derived `device_selector` and a selection heuristic

```
4   class my_device_selector : public device_selector {
5   public:
6     int operator()(const device &dev) const override{
7       int rating = 0;
8       if (dev.is_gpu()) rating = 1;
9       else if (dev.is_cpu()) rating = 2;
10      return rating; };
11  };
12
13  int main() {
29    device device2 = queue2.get_device() ;
30
31    my_device_selector h_selector;
32    queue Queue3(h_selector);
```

Event

The `event` class encapsulates the `cl_event` from the OpenCL standard and is employed primarily for interoperability.

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`.

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.

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.

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

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

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`.

Multi-pointer

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.

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.

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

Platform

The DPC++ `platform` class encapsulates the host and device functionality employed by a DPC++ program.

The constructors either construct a host platform, or for backwards compatibility, an OpenCL platform. One version of the constructor takes a `device_selector` object employed to choose the particular device for execution.

Member functions for the platform class include:

- `get` – obtain an OpenCL platform from the platform
- `get_info` – obtain information on the platform
- `has_extension` – query the platform for specific support of an extension
- `is_host` – is the platform a host platform
- `get_devices` – return all devices associated with the platform

The member function `get_info` returns specific information as a string about the platform, including:

- Profile – returns if platform supports for full or embedded profile
- Version – returns version number information
- Name – returns the name of the platform
- Vendor – returns the vendor of the platform
- Extensions – returns a vector of strings that list the supported extensions

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

Queue

A DPC++ `queue` is employed to schedule and execute the command queues on the devices.

Multiple forms of constructors are available with different combinations of arguments, including device selectors, devices, contexts, and command queue. In addition, an `asyc_handler` can be passed to help communicate errors from the devices back to the host.

The command queue itself executes in a synchronous fashion and therefore errors are also synchronous in nature. The actual kernels execute asynchronous and therefore errors are handled asynchronously by the `async_handler`. Queues can synchronize by calling `wait` and `wait_and_throw_throw` member functions.

Command groups are submitted to the queue object using the submit member function.

A `property_list` can also be passed during construction, which can be used to communicate an `enable_profiling` property to the devices.

A description of a few other member functions include:

- `get` – obtain a `cl_command_queue`
- `get_context` – obtain the context associated with the queue
- `get_device` – obtain the device associated with the queue
- `is_host` – return if the queue is executing on the host

For more information, see Execution Model.

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??
```

Stream

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

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: +=, -=, *=, /=, %=, ++, --, &, |, ^, +, -, *, /, %, &&, ||, <<, >>, <<=, >>=, ==, !=, <, >, <=, >=.

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

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.

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:

- `vector_class` – `std::vector<>`
- `string_class` – `std::string`
- `function_class` – `std::function<>`
- `mutex_class` – `std::mutex`
- `shared_ptr_class` – `std::shared_ptr<>`
- `unique_ptr_class` – `std::unique_ptr<>`
- `weak_ptr_class` – `std::weak_ptr<>`
- `hash_class` – `std::hash`
- `exception_ptr_class` – `std::exception_ptr`

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.

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

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.

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

Directive	Description
SYCL_PROGRAM_BUILD_OPTIONS	Used to pass additional options for device program building.

5 API-based Programming

Several libraries are available with oneAPI toolkits that can simplify the programming process by providing specialized APIs for use in optimized applications. This chapter provides basic details about the libraries, including code samples, to help guide the decision on which library is most useful in certain use cases. Detailed information about each library, including more about the available APIs, is available in the main documentation for that library.

oneAPI Library Overview

The following libraries are available from the oneAPI toolkits:

Library	Usage
Intel oneAPI DPC++ Library	Use this library for high performance parallel applications.
Intel oneAPI Math Kernel Library	Use this library to include highly optimized and extensively parallelized math routines in an application.
Intel oneAPI Threading Building Blocks	Use this library to combine TBB-based parallelism on multicore CPUs and DPC++ device-accelerated parallelism in an application.
Intel oneAPI Data Analytics Library	Use this library to speed up big data analysis applications and distributed computation.
Intel oneAPI Collective Communications Library	Use this library for applications that focus on Deep Learning and Machine Learning workloads.
Intel oneAPI Deep Neural Network Library	Use this library for deep learning applications that use neural networks optimized for Intel Architecture Processors and Intel Processor Graphics.
Intel oneAPI Video Processing Library	Use this library to accelerate video processing in an application.

Intel oneAPI DPC++ Library (oneDPL)

The Intel oneAPI DPC++ Library (oneDPL) works with the Intel oneAPI DPC++ Compiler to provide high-productivity APIs. Using these APIs can minimize DPC++ programming efforts across devices for high performance parallel applications.

oneDPL consists of following components:

- C++ standard APIs verified for DPC++ kernels
- Parallel STL algorithms with execution policies to run on DPC++ devices
- Non-standard API extensions

Currently, Parallel STL has been integrated into oneDPL. Non-standard API extensions are planned for gradual integration in future releases. The testing results of C++ standard APIs on devices and the supported Parallel STL and non-standard API extensions are posted in the oneDPL Release Notes.

For more information, see the Intel oneAPI DPC++ Library Guide.

oneDPL Library Usage

oneDPL is a component of the Intel® oneAPI Base Toolkit.

Several C++ standard APIs have been tested and function well within DPC++ kernels. To use them, include the corresponding C++ standard header files and use the std namespace.

To use Parallel STL and non-standard API extensions, include necessary header files in the source code. All oneDPL header files are in the `dpstd` directory. Use `#include <dpstd/…>` to include them.

oneDPL has its own namespace dpstd for all its extensions, including DPC++ execution policies, non-standard algorithms, special iterators, etc.

To build the code using Parallel STL algorithms, set up environment variables for the Intel oneAPI DPC++ Compiler, Intel oneAPI Threading Building Blocks, and Parallel STL. For details, see Get Started with Parallel STL.

oneDPL Code Samples

Parallel STL for DPC++ extends the standard C++17 parallel algorithms with

- DPC++ execution policy
- `dpstd::begin`, `dpstd::end` functions

To compile the code samples:

1. Set the environment variables.
2. Run the following command:

```
dpcpp test.cpp -o test
```

DPC++ Execution Policy

The DPC++ execution policy specifies where and how a Parallel STL algorithm runs. It encapsulates a standard C++ 17 execution policy by inheritance (currently, only `parallel_unsequenced_policy` is supported), a SYCL device or queue, and an optional kernel name.

To set up the policy:

1. Add `#include <dpstd/execution>`.
2. Create a policy object providing a standard policy type and a special class (a unique kernel name; it is optional if the host code to invoke the kernel is compiled with the Intel oneAPI DPC++ Compiler) as template arguments and one of the following constructor arguments:
   - A SYCL queue
   - A SYCL device
   - A SYCL device selector
   - An existing policy providing a new kernel name

```
 using namespace dpstd::execution;
 auto policy_a = sycl_policy<parallel_unsequenced_policy, class PolicyA> {cl::sycl::queue{ }};
 std::for_each(policy_a, ...);
 auto policy_b = sycl_policy<parallel_unsequenced_policy, class PolicyB>
{cl::sycl::device{cl::sycl::gpu_selector{}}};
 std::for_each(policy_b, ...);
 auto policy_c = sycl_policy<parallel_unsequenced_policy, class PolicyC>
{cl::sycl::default_selector{}};
 std::for_each(policy_c, ...);
 auto policy_d = make_sycl_policy<class PolicyD>(sycl); // sycl is predefined object of
sycl_policy class using default kernel name
 std::for_each(policy_d, ...);
 auto policy_e = make_sycl_policy<class PolicyE>(cl::sycl::queue{});
 std::for_each(policy_e, ...);
```

dpstd::begin, dpstd::end Functions

`dpstd::begin`, `dpstd::end` are special helper functions that allow passing of SYCL buffers to Parallel STL algorithms. These functions accept a SYCL buffer and return an object of an unspecified type that satisfies the following requirements:

1. Is CopyConstructible, CopyAssignable, and comparable with operators == and !=
2. The following expressions are valid: a + n, a - n, –a - b, where a and b are objects of the type, and n is an integer value
3. Contains `get_buffer()` method that returns the SYCL buffer passed to `dpstd::begin`, `dpstd::end` functions.

To use the functions, add `#include <dpstd/iterators.h>` to your code.

The following example shows how to process a SYCL buffer with a Parallel STL algorithm:

```
#include <CL/sycl.hpp>
#include <dpstd/execution>
#include <dpstd/algorithm>
#include <dpstd/iterators.h>
int main(){
    cl::sycl::queue q;
    cl::sycl::buffer<int> buf { 1000 };
    auto buf_begin = dpstd::begin(buf);
    auto buf_end   = dpstd::end(buf);
    auto policy = dpstd::execution::make_sycl_policy<class fill>(q);
    std::fill(policy, buf_begin, buf_end, 42);
    return 0;
  }
```

NOTE Parallel STL algorithms can be called with ordinary (host-side) iterators, as in the code below. In that case, a temporary SYCL buffer is created and the data is copied to this buffer. After the processing of the temporary buffer on a device is complete, the data is copied back to the host. Working with SYCL buffers is recommended to reduce data copying between host and device.

```
#include <vector>
#include <dpstd/execution>
#include <dpstd/algorithm>
int main(){
     std::vector<int> vec( 1000000 );
     auto policy = dpstd::execution::make_sycl_policy<
     class fill>(dpstd::execution::sycl);
     std::fill(policy, vec.begin(), vec.end(), 42);
  // each element of vec will be equal to 42
     return 0;
}
```

Verified C++ Standard API

Several C++ Standard APIs can be employed in device kernels similar to how they are employed in code for a typical CPU-based platform. The following code demonstrates such use for `std::swap` function:

```
#include <CL/sycl.hpp>
#include <utility>
#include <iostream>
constexpr cl::sycl::access::mode sycl_read_write,
  cl::sycl::access::mode::read_write;
class KernelSwap;
```

```
void kernel_test() {
  cl::sycl::queue deviceQueue;
  cl::sycl::range<1> numOfItems{2};
  cl::sycl::cl_int swap_num[2] = {4, 5};
  std::cout << swap_num[0] << ", " << swap_num[1] << std::endl;
  {
    cl::sycl::buffer<cl::sycl::cl_int, 1>
      swap_buffer(swap_num, numOfItems);
    deviceQueue.submit([&](cl::sycl::handler &cgh) {
      auto swap_accessor = swap_buffer.get_access<
        sycl_read_write>(cgh);
      cgh.single_task<class KernelSwap>([=]() {
        int & num1 = swap_accessor[0];
        int & num2 = swap_accessor[1];
        std::swap(num1, num2);
      });
    });
  }
    std::cout << swap_num[0] << ", " << swap_num[1] << std::endl;
}
int main() {
  kernel_test();
  return 0;
}
```

Intel oneAPI Math Kernel Library (oneMKL)

The Intel oneAPI Math Kernel Library (oneMKL) is a computing math library of highly optimized and extensively parallelized routines for applications that require maximum performance. oneMKL contains the high-performance optimizations from the full Intel® Math Kernel Library for CPU architectures (with C/Fortran programming language interfaces) and adds to them a set of Data Parallel C++ (DPC++) programming language interfaces for achieving performance on various CPU architectures and Intel Graphics Technology for certain key functionalities.

The new DPC++ interfaces with optimizations for CPU and GPU architectures have been added for key functionality in the following major areas of computation:

- BLAS and LAPACK dense linear algebra routines
- Sparse BLAS sparse linear algebra routines
- Random number generators (RNG)
- Vector Mathematics (VM) routines for optimized mathematical operations on vectors
- Fast Fourier Transforms (FFTs)

For more information, see Get Started with Intel oneAPI Math Kernel Library for Data Parallel C++.

oneMKL Usage

When using the DPC++ programming language interfaces, there are a few changes to consider:

- oneMKL has a dependency on the Intel oneAPI DPC++ Compiler and Intel oneAPI DPC++ Library. Applications must be built with the Intel oneAPI DPC++ Compiler, the DPC++ headers made available, and the application linked with oneMKL using the DPC++ linker.
- DPC++ interfaces in oneMKL use `cl::sycl::buffer` objects for input data (vectors, matrices, etc.).
- Some DPC++ interfaces in oneMKL support the use of Unified Shared Memory (USM) pointers for input data in place of the `cl::sycl::buffer` objects.
- DPC++ interfaces in oneMKL are overloaded based on the floating point types. For example, there are several general matrix multiply APIs, accepting single precision real arguments (float), double precision real arguments (double), half precision real arguments (half), and complex arguments of different precision using the standard library types `std::complex<float>`, `std::complex<double>`.
- A two-level namespace structure for oneMKL is added for DPC++ interfaces:

Namespace	Description
`mkl`	Contains common elements between various domains in oneMKL
`mkl::blas`	Contains dense vector-vector, matrix-vector, and matrix-matrix low level operations
`mkl::lapack`	Contains higher-level dense matrix operations like matrix factorizations and eigensolvers
`mkl::rng`	Contains random number generators for various probability density functions
`mkl::vm`	Contains vector math routines
`mkl::dft`	Contains fast fourier transform operations
`mkl::sparse`	Contains sparse matrix operations like sparse matrix-vector multiplication and sparse triangular solver

oneMKL Code Sample

To demonstrate a typical workflow for the oneMKL with DPC++ interfaces, the following example source code snippets perform a double precision matrix-matrix multiplication on a GPU device.

```
// standard SYCL header
#include <CL/sycl.hpp>
// include std::exception class
#include <exception>
// declarations for Intel oneAPI Math Kernel Library SYCL apis
#include "mkl_sycl.hpp"
int main(int argc, char *argv[]) {
    //
    // User obtain data for A,B,C matrices along with setting m,n,k, ldA, ldB, ldC.
    //
    // A, B and C should be stored in containers like std::vector that contain a
    // data() and size() member function
    //
    // create gpu device
    cl::sycl::device my_device;
    try {
        my_device = cl::sycl::device(cl::sycl::gpu_selector());
    }
    catch (...) {
        std::cout << "Warning, gpu device not found! Defaulting back to host device from default
constructor. " << std::endl;
    }
    // create asynchronous exceptions handler to be attached to queue
    auto my_exception_handler = [](cl::sycl::exception_list exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
                std::rethrow_exception(e);
            }
            catch (cl::sycl::exception const& e) {
                std::cout << "Caught asynchronous SYCL exception:\n"
                    << e.what() << std::endl;
            }
            catch (std::exception const& e) {
                std::cout << "Caught asynchronous STL exception:\n"
```

```
                    << e.what() << std::endl;
            }
        }
    };
    // create execution queue on my gpu device with exception handler attached
    cl::sycl::queue my_queue(my_device, my_exception_handler);
    // create sycl buffers of matrix data for offloading between device and host
    cl::sycl::buffer<double, 1> A_buffer(A.data(), A.size());
    cl::sycl::buffer<double, 1> B_buffer(B.data(), B.size());
    cl::sycl::buffer<double, 1> C_buffer(C.data(), C.size());
    // add mkl::blas::gemm to execution queue and catch any synchronous exceptions
    try {
        mkl::blas::gemm(my_queue, mkl::transpose::nontrans, mkl::transpose::nontrans, m, n, k,
alpha, A_buffer, ldA, B_buffer,
           ldB, beta, C_buffer, ldC);
    }
    catch (cl::sycl::exception const& e) {
        std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    catch (std::exception const& e) {
        std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    // ensure any asynchronous exceptions caught are handled before proceeding
    my_queue.wait_and_throw();
    //
    // post process results
    //
    // Access data from C buffer and print out part of C matrix
    auto C_accessor = C_buffer.template get_access<cl::sycl::access::mode::read>();
    std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
        << C_accessor[1] << ", ... ]\n";
    std::cout << "\t    [ " << C_accessor[1 * ldC + 0] << ", "
        << C_accessor[1 * ldC + 1] << ",  ... ]\n";
    std::cout << "\t    [ " << "... ]\n";
    std::cout << std::endl;

    return 0;
}
```

Consider that (double precision valued) matrices A(of size m-by-k), B(of size k-by-n) and C(of size m-by-n) are stored in some arrays on the host machine with leading dimensions ldA, ldB, and ldC, respectively. Given scalars (double precision) alpha and beta, compute the matrix-matrix multiplication (`mkl::blas::gemm`):

C = alpha * A * B + beta * C

Include the standard SYCL headers and the oneMKL SYCL specific header that declares the desired `mkl::blas::gemm` API:

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

// declarations for oneAPI MKL DPC++ BLAS apis
#include "mkl_blas_sycl.hpp"
```

Next, load or instantiate the matrix data on the host machine as usual and then create the sycl device, create an asynchronous exception handler, and finally create the queue on the device with that exception handler. Exceptions that occur on the host can be caught using standard C++ exception handling mechanisms; however, exceptions that occur on a device are considered asynchronous errors and stored in an exception list to be processed later by this user-provided exception handler.

```
// create gpu device
cl::sycl::device my_device;
try {
  my_device = cl::sycl::device(cl::sycl::gpu_selector());
} catch (...) {
  std::cout << "Warning, gpu device not found! Defaulting back to host device from default
constructor. " << std::endl;
}

// create asynchronous exceptions handler to be attached to queue
auto my_exception_handler = [](cl::sycl::exception_list exceptions) {
  for (std::exception_ptr const& e : exceptions) {
    try {
      std::rethrow_exception(e);
    }
    catch (cl::sycl::exception const& e) {
      std::cout << "Caught asynchronous SYCL exception:\n"
                << e.what() << std::endl;
    }
  }
};

// create execution queue on my gpu device with exception handler attached
cl::sycl::queue my_queue(my_device, my_exception_handler);
```

The matrix data is now loaded into the sycl buffers, which enables offloading to desired devices and then back to host when complete. Finally, the `mkl::blas::gemm` API is called with all the buffers, sizes, and transpose operations, which will enqueue the matrix multiply kernel and data onto the desired queue.

```
// create sycl buffers of matrix data for offloading between device and host
cl::sycl::buffer<double, 1> A_buffer(A.data(), A.size());
cl::sycl::buffer<double, 1> B_buffer(B.data(), B.size());
cl::sycl::buffer<double, 1> C_buffer(C.data(), C.size());

// add mkl::blas::gemm to execution queue and catch any synchronous exceptions
try {
  mkl::blas::gemm(my_queue, mkl::transpose::nontrans, mkl::transpose::nontrans,
                  m, n, k, alpha, A_buffer, ldA, B_buffer, ldB, beta, C_buffer, ldC);
}
catch (cl::sycl::exception const& e) {
  std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
    << e.what() << std::endl;
}
```

At some time after the gemm kernel has been enqueued, it will be executed. The queue is asked to wait for all kernels to execute and then pass any caught asynchronous exceptions to the exception handler to be thrown. The SYCL runtime will handle transfer of the buffer's data between host and GPU device and back. By the time an accessor is created for the C_buffer, the buffer data will have been silently transferred back to the host machine. In this case, the accessor is used to prints out a 2x2 submatrix of C_buffer.

```
// Access data from C buffer and print out part of C matrix
auto C_accessor = C_buffer.template get_access<cl::sycl::access::mode::read>();
std::cout << "\t" << C << " = [ " << C_accessor[0]           << ", "
                                                 << C_accessor[1]           << ", ... ]\n";
std::cout << "\t    [ "                      << C_accessor[1 * ldC + 0] << ", "
```

```
                                                  << C_accessor[1*ldC + 1]   << ",  ... ]\n";
std::cout << "\t    [ "                        << "... ]\n";
std::cout << std::endl;
```

Note that the resulting data is still in the C_buffer object and, unless it is explicitly copied elsewhere (like back to the original C container), it will only remain available through accessors until the C_buffer is out of scope.

Intel oneAPI Threading Building Blocks (oneTBB)

Intel oneAPI Threading Building Blocks (oneTBB) is an open source library to write parallel C++ programs that take full advantage of multicore performance, are portable and composable, and have future-proof scalability. Through community involvement, oneTBB is a feature-rich and popular solution for developing shared memory parallel applications. It has been used in production for data analytics, creative content creation, and many other applications.

oneTBB is compiler-independent and is available on a variety of processors and operating systems. It is used by other oneAPI libraries (Intel oneAPI Math Kernel Library, Intel oneAPI Deep Neural Network Library, etc.) to express multithreading parallelism for CPUs.

oneTBB Usage

oneTBB can be used with the Intel oneAPI DPC++ Compiler in the same way as with any other C++ Compiler. For more details, see the documentation at https://software.intel.com/en-us/tbb.

Currently, oneTBB does not directly use any accelerators. However, it can be combined with the DPC++ language and other oneAPI libraries to build a program that efficiently utilizes all available hardware resources.

oneTBB Code Sample

Two basic oneTBB code samples are available from the Intel oneAPI Base Toolkit GitHub repository https://github.com/intel/BaseKit-code-samples. The examples are illustrative of typical use cases and can serve as a starting point for developers who want to experiment with using both oneTBB and DPC++ in the same application.

Intel oneAPI Data Analytics Library (oneDAL)

Intel oneAPI Data Analytics Library (oneDAL) is a library that helps speed up big data analysis by providing highly optimized algorithmic building blocks for all stages of data analytics (preprocessing, transformation, analysis, modeling, validation, and decision making) in batch, online, and distributed processing modes of computation.

The library optimizes data ingestion along with algorithmic computation to increase throughput and scalability. It includes C++ and Java* APIs and connectors to popular data sources such as Spark* and Hadoop*. Python* wrappers for oneDAL are part of Intel Distribution for Python.

In addition to classic features, oneDAL provides DPC++ API extensions to the traditional C++ interface and enables GPU usage for some algorithms.

The library is particularly useful for distributed computation. It provides a full set of building blocks for distributed algorithms that are independent from any communication layer. This allows users to construct fast and scalable distributed applications using user-preferable communication means.

For general information, visit the oneDAL GitHub* page. The complete list of features and documentation are available at the official Intel oneAPI Data Analytics Library website. Free and open-source community-supported versions are available, as well as paid versions with premium support.

oneDAL Usage

The following dependencies are needed to build and link your application with oneDAL:

1. Intel oneAPI DPC++ Compiler
2. OpenCL™ Runtime 1.2 or later

A oneDAL-based application can seamlessly execute algorithms on CPU or GPU by picking the proper device selector. New capabilities also allow:

- extracting DPC++ buffers from numeric tables and pass them to a custom kernel
- creating numeric tables from DPC++ buffers

Algorithms are optimized to reuse DPC++ buffers to keep GPU data and remove overload from repeatedly copying data between GPU and CPU.

oneDAL Code Sample

The following code sample demonstrates oneDAL-specific features:

```
#include "daal_sycl.h"
#include <iostream>
using namespace daal;
using namespace daal::algorithms;
using namespace daal::data_management;
int main(int argc, char const *argv[])
{
  // Set the desired execution context
  cl::sycl::queue queue { cl::sycl::gpu_selector() };
  services::SyclExecutionContext ctx(queue);
  services::Environment::getInstance()->setDefaultExecutionContext(ctx);
  float input[6] = { 1.5f, 2.7f, 3.0f, 6.0f, 2.0f, 4.0f };
  cl::sycl::buffer<float, 1> a { input, cl::sycl::range<1>(6) };
  auto data = SyclHomogenNumericTable<>::create(a, 2, 3);
  covariance::Batch<> algorithm;
  algorithm.input.set(covariance::data, data);
  algorithm.parameter.outputMatrixType = covariance::correlationMatrix;
  algorithm.compute();
  NumericTablePtr table = algorithm.getResult()->get(covariance::correlation);
  // Get the DPC++ buffer from table
  BlockDescriptor<float> block;
  const size_t startRowIndex = 0;
  const size_t numberOfRows = table->getNumberOfRows();
  table->getBlockOfRows(startRowIndex, numberOfRows, readOnly, block);
  cl::sycl::buffer<float, 1> buffer = block.getBuffer().toSycl();
  table->releaseBlockOfRows(block);
  // Printing result to the console
  auto accessor = buffer.get_access<cl::sycl::access::mode::read>();
  for (int row = 0; row < table->getNumberOfRows(); row++)
  {
    for (int col = 0; col < table->getNumberOfColumns(); col++)
    {
      std::cout << accessor[row*table->getNumberOfColumns()+col] << ", ";
    }
    std::cout << std::endl;
  }
  return 0;
}
```

A typical command line to build a oneDAL application is:

```
dpcpp main.cpp -ldaal_core -ldaal_thread -foffload-static-lib=$DAALROOT/lib/intel64/
libdaal_sycl.a
```

Intel oneAPI Collective Communications Library (oneCCL)

Intel oneAPI Collective Communications Library (oneCCL) is a scalable and high-performance communication library for Deep Learning (DL) and Machine Learning (ML) workloads. It develops the ideas that originated in Intel® Machine Learning Scaling Library and expands the design and API to encompass new features and use cases.

oneCCL features include:

- Built on top of lower-level communication middleware – MPI and libfabrics
- Optimized to drive scalability of communication patterns by enabling the productive trade-off of compute for communication performance
- Enables a set of DL-specific optimizations, such as prioritization, persistent operations, out of order execution, etc.
- DPC++-aware API to run across various hardware targets, such as CPUs and GPUs
- Works across various interconnects: Intel® Omni-Path Architecture (Intel® OPA), InfiniBand*, and Ethernet

oneCCL Usage

Dependencies:

- MPI
- libfabrics
- Intel oneAPI DPC++ Compiler

SYCL-aware API is an optional feature of oneCCL. The choice between CPU and SYCL back ends is performed by specifying `ccl_stream_type` value when creating the oneCCL stream object. For a GPU backend, specify `ccl_stream_sycl` as the first argument. For collective operations that operate on SYCL stream, oneCCL expects communication buffers to be `sycl::buffer` objects casted to `void*`.

oneCCL Code Sample

oneCCL sample code, such as `ccl_sample.cpp`, is available from the Intel oneAPI Base Toolkit GitHub repository https://github.com/intel/BaseKit-code-samples or from the Intel oneAPI DL Framework Developer Toolkit GitHub repository https://github.com/intel/DLFDKit-code-samples.

Use the following command to compile the code:

```
dpcpp -I$CCL_ROOT/include -L$CCL_ROOT/lib/ -lccl ./ccl_sample.cpp -o ccl_sample
```

Intel oneAPI Deep Neural Network Library (oneDNN)

Intel oneAPI Deep Neural Network Library (oneDNN) is an open-source performance library for deep learning applications. The library includes basic building blocks for neural networks optimized for Intel Architecture Processors and Intel Processor Graphics. oneDNN is intended for deep learning applications and framework developers interested in improving application performance on Intel Architecture Processors and Intel Processor Graphics. Deep learning practitioners should use one of the applications enabled with oneDNN.

oneDNN is distributed as part of Intel® oneAPI DL Framework Developer Toolkit, the Intel oneAPI Base Toolkit, and is available via apt and yum channels.

oneDNN continues to support features currently available with DNNL, including C and C++ interfaces, OpenMP*, Intel oneAPI Threading Building Blocks, and OpenCL™ runtimes. oneDNN introduces DPC++ API and runtime support for the oneAPI programming model.

For more information, see https://github.com/intel/mkl-dnn.

oneDNN Usage

oneDNN supports systems based on Intel 64 architecture or compatible processors. A full list of supported CPU and graphics hardware is available from the Intel oneAPI Deep Neural Network Library System Requirements.

oneDNN detects the instruction set architecture (ISA) in the runtime and uses online generation to deploy the code optimized for the latest supported ISA.

Several packages are available for each operating system to ensure interoperability with CPU or GPU runtime libraries used by the application.

Configuration	Dependency
`cpu_dpcpp_gpu_dpcpp`	DPC++ runtime
`cpu_iomp`	OpenMP* runtime
`cpu_gomp`	GNU* OpenMP runtime
`cpu_vcomp`	Microsoft* Visual C++ OpenMP runtime
`cpu_tbb`	Intel oneAPI Threading Building Blocks

The packages do not include library dependencies and these need to be resolved in the application at build time with oneAPI toolkits or third-party tools.

When used in the DPC++ environment, oneDNN relies on the DPC++ runtime to interact with CPU or GPU hardware. oneDNN may be used with other code that uses DPC++. To do this, oneDNN provides API extensions to interoperate with underlying SYCL objects.

One of the possible scenarios is executing a DPC++ kernel for a custom operation not provided by oneDNN. In this case, oneDNN provides all necessary APIs to seamlessly submit a kernel, sharing the execution context with oneDNN: using the same device and queue.

The interoperability API is provided for two scenarios:

- Construction of oneDNN objects based on existing DPC++ objects
- Accessing DPC++ objects for existing oneDNN objects

The mapping between oneDNN and DPC++objects is summarized in the tables below.

oneDNN Objects	DPC++ Objects
Engine	cl::sycl::device and cl::sycl::context
Stream	cl::sycl::queue
Memory	cl::sycl::buffer<uint8_t, 1>

NOTE Internally, library memory objects use 1D uint8_t SYCL buffers, however SYCL buffers of a different type can be used to initialize and access memory. In this case, buffers will be reinterpreted to the underlying type `cl::sycl::buffer<uint8_t, 1>`.

oneDNN Object	Constructing from DPC++ Object	Extracting DPC++ Object
Engine	`mkldnn::engine(kind, sycl_dev, sycl_ctx)`	`mkldnn::engine::get_sycl_device()` `mkldnn::engine::get_sycl_context()`

oneDNN Object	Constructing from DPC++ Object	Extracting DPC++ Object
Stream	`mkldnn::stream(engine, sycl_queue)`	`mkldnn::stream::get_sycl_queue()`
Memory	`mkldnn::memory(memory_desc, engine, sycl_buf)`	`mkldnn::memory::get_sycl_buffer()`

Building applications with oneDNN requires a compiler. The Intel oneAPI DPC++ Compiler is available as part of the Intel oneAPI Base Toolkit and the Intel C++ Compiler is available as part of the Intel oneAPI HPC Toolkit.

oneDNN Code Sample

oneDNN sample code is available installed with the product at `onednn/cpu_dpcpp_gpu_dpcpp/examples/sycl_interop.cpp`.

Intel oneAPI Video Processing Library (oneVPL)

Intel oneAPI Video Processing Library (oneVPL) provides cross platform support and vendor independent support for accelerated video processing. oneVPL is designed to simplify interacting with video content in applications where accelerators with independent storage are leveraged.

oneVPL Usage

oneVPL can use any of multiple accelerators. As a fallback it can use purely CPU based operations; however, to use other accelerator based hardware, the relevant drivers must be installed. Required drivers currently include:

- Intel® Graphics Driver (for support of Graphics Controller)

The oneVPL API is defined using a classic C++ style interface, which allows the caller to select how the accelerator should be used at runtime. However, oneVPL is expected to be used in programs that also use other data parallel operations, and as such it is designed to be used in a SYCL/DPC++ program.

The main interface between oneVPL and data parallel algorithms is the vplMemory library. vplMemory is used for handling frames to simplify transporting memory buffers. vplMemory provides reference counted buffers that can be moved between differing hardware contexts for parallel processing on demand.

oneVPL Code Sample

oneVPL exposes a minimal basic interface suitable for most core video processing use cases. This interface consists of a class that exposes three core methods: SetConfig (use for configuring options), GetState (use to control the main read/write loop), and DecodeFrame (use to read/write data).

```
#include "vpl/vpl.hpp"
#define BUFFER_SIZE 1024 * 1024
int main(int argc, char* argv[]) {
  if (argc != 2) {
    printf("usage: %s [h264 input file]\n", argv[0]);
    printf("example: %s content/cars_1280x720.h264\n", argv[0]);
    return 1;
  }
  // Create decoder, default device is GPU
  vpl::Decode decoder(VPL_FOURCC_H264);
  // Set output color format
  decoder.SetConfig(VPL_PROP_DST_FORMAT, VPL_FOURCC_RGBA);
  // Set output resolution
  VplVideoSurfaceResolution output_size = {352, 288};
```

```
  decoder.SetConfig(VPL_PROP_OUTPUT_RESOLUTION, output_size);
  // initialize
  uint8_t *pbs=new uint8_t[BUFFER_SIZE];
  FILE* fInput = fopen(argv[1], "rb");
  if (!fInput) {
    printf("could not open input file '%s'\n", argv[1]);
    return 1;
  }
  VplFile* fOutput = vplOpenFile("out_352x288.rgba", "wb");
  vplm_mem* image = nullptr;
  bool bdrain_mode = false;
  // -----------
  // MAIN LOOP
  // ----------
  // Loop until done.  Decode state of END_OF_OPERATION or
  // ERROR indicates loop exit.
  vplWorkstreamState decode_state = VPL_STATE_READ_INPUT;
  for (; decode_state != VPL_STATE_END_OF_OPERATION &&
         decode_state != VPL_STATE_ERROR;
       decode_state = decoder.GetState()) {
    // read more input if state indicates buffer space
    // is available
    uint32_t bs_size = 0;
    if ((decode_state == VPL_STATE_READ_INPUT) && (!bdrain_mode)) {
      bs_size = (uint32_t)fread(pbs, 1, BUFFER_SIZE, fInput);
    }
    if (bs_size == 0 || decode_state ==
      VPL_STATE_INPUT_BUFFER_FULL) {
      bdrain_mode = true;
    }
    // Attempt to decode a frame. If more data is needed read again
    if (bdrain_mode)
      image = decoder.DecodeFrame(nullptr, 0);
    else
      image = decoder.DecodeFrame(pbs, bs_size);
    if (!image) continue;
    // If decode resulted in a frame of output write it to file
    vplWriteData(fOutput, image);
    printf(".");
    fflush(stdout);
  }
  printf("\n");
  // cleanup
  fclose(fInput);
  vplCloseFile(fOutput);
  delete[] pbs;
  return 0;
}
```

To compile the code, use one of the following commands:

- On Linux:

```
gcc <filename> -lvpl -lvplmemory -lstdc++
```

- On Windows:

```
cl <filename> /link vpl.lib /link vplmemory.lib
```

DecodeFrame accepts any amount of data from one byte up to the size of a large internal buffer and return a single frame if one can be decoded.

The important states for normal operation are:

State	Description
VPL_STATE_READY_FOR_INPUT	Data provided will be read (input will be ignored in other states).
VPL_STATE_ERROR	An error has occurred.
VPL_STATE_INPUT_BUFFER_FULL	There is no room to read more input, recommendation is to read out frames before writing more.
VPL_STATE_END_OF_OPERATION	Indicates all available frames have been output (End of input was reached while the End Of Stream parameter was true.)

Other Libraries

Other libraries are included in various oneAPI toolkits. For more information about each of the libraries listed, consult the official documentation for that library.

- Intel® Integrated Performance Primitives (IPP)
- Intel® MPI Library
- Intel® Open Volume Kernel Library

Software Development Process

6

The software development process using the oneAPI programming model is based upon standard development processes. Since the programming model pertains to employing an accelerator to improve performance, this chapter details steps specific to that activity. These include:

- The performance tuning cycle
- Debugging of code
- Migrating code that targets other accelerators
- Composability of code

Performance Tuning Cycle

The goal of the performance tuning cycle is to improve the time to solution whether that be interactive response time or elapsed time of a batch job. In the case of a heterogeneous platform, there are compute cycles available on the devices that execute independently from the host. Taking advantage of these resources offers a performance boost.

The performance tuning cycle includes the following steps detailed in the next sections:

1. Establish a baseline
2. Identify kernels to offload
3. Offload the kernels
4. Optimize
5. Repeat until objectives are met

Establish Baseline

Establish a baseline that includes a metric such as elapsed time, time in a compute kernel, or floating point operations per second that can be used to measure the performance improvement and that provides a means to verify the correctness of the results.

A simple method is to employ the chrono library routines in C++, placing timer calls before and after the workload executes.

Identify Kernels to Offload

To best utilize the compute cycles available on the devices of a heterogeneous platform, it is important to identify the tasks that are compute intensive and that can benefit from parallel execution. Consider an application that executes solely on a CPU, but there may be some tasks suitable to execute on a GPU. This can be determined using the offload performance prediction capabilities of Intel Advisor.

Intel Advisor can create performance characterizations of the workload as it may execute on an accelerator. It consumes the information from profiling the workload and provides performance estimates, bottleneck characterization, and offload data transfer estimates.

Typically, kernels with high compute, a large dataset, and limited memory transfers are best suited for offload to a device.

Offload Kernels

After identifying kernels that are suitable for offload, employ DPC++ to offload the kernel onto the device. Consult the previous chapters as an information resource.

Optimize

oneAPI enables functional code that can execute on multiple accelerators; however, the code may not be the most optimal across the accelerators. A three-step optimization strategy is recommended to meet performance needs:

1. Pursue general optimizations that apply across accelerators.
2. Optimize aggressively for the prioritized accelerators.
3. Optimize the host code in conjunction with step 1 and 2.

Optimization is a process of eliminating bottlenecks, i.e. the sections of code that are taking more execution time relative other sections of the code. These sections could be executing on the devices or the host. During optimization, employ a profiling tool such as Intel VTune Profiler to find these bottlenecks in the code.

This section discusses the first step of the strategy - Pursue general optimizations that apply across accelerators. Device specific optimizations and best practices for specific devices (step 2) and optimizations between the host and devices (step 3) are detailed in device-specific optimization guides, such as the Intel oneAPI DPC++ FPGA Optimization Guide. This section assumes that the kernel to offload to the accelerator is already determined. It also assumes that work will be accomplished on one accelerator. This guide does not speak to division of work between host and accelerator or between host and potentially multiple and/or different accelerators.

General optimizations that apply across accelerators can be classified into four categories:

1. High-level optimizations
2. Loop-related optimizations
3. Memory-related optimizations
4. DPC++-specific optimizations

The following sections summarize these optimizations only; specific details on how to code most of these optimizations can be found online or in commonly available code optimization literature. More detail is provided for the DPC++ specific optimizations.

High-level Optimization Tips

- Increase the amount of parallel work. More work than the number of processing elements is desired to help keep the processing elements more fully utilized.
- Minimize the code size of kernels. This helps keep the kernels in the instruction cache of the accelerator, if the accelerator contains one.
- Load balance kernels. Avoid significantly different execution times between kernels as the long-running kernels may become bottlenecks and affect the throughput of the other kernels.
- Avoid expensive functions. Avoid calling functions that have high execution times as they may become bottlenecks.

Loop-related Optimizations

- Prefer well-structured, well-formed, and simply exit condition loops – these are loops have a single exit and a single condition when comparing against an integer bound.
- Prefer loops with linear indexes and constant bounds – these are loops that employ an integer index into an array, for example, and have bounds that are known at compile-time.
- Declare variables in deepest scope possible. Doing so can help reduce memory or stack usage.
- Minimize or relax loop-carried data dependencies. Loop-carried dependencies can limit parallelization. Remove dependencies if possible. If not, pursue techniques to maximize the distance between the dependency and/or keep the dependency in local memory.
- Unroll loops with pragma unroll.

Memory-related Optimizations

- When possible, favor greater computation over greater memory use. The latency and bandwidth of memory compared to computation can become a bottleneck.

- When possible, favor greater local and private memory use over global memory use.
- Avoid pointer aliasing.
- Coalesce memory accesses. Grouping memory accesses helps limit the number of individual memory requests and increases utilization of individual cache lines.
- When possible, store variables and arrays in private memory for high-execution areas of code.
- Beware of loop unrolling effects on concurrent memory accesses.
- Avoid a write to a global that another kernel reads. Use a pipe instead.

DPC++-specific Optimizations

- When possible, specify a work-group size. The attribute, `[[cl::reqd_work_group_size(X, Y, Z)]]`, where X, Y, and Z are integer dimension in the ND-range, can be employed to limit the maximum possible size. The compiler can take advantage of these limits to optimize more aggressively.
- Consider use of the `-Xsfp-relaxed` option when possible. This option relaxes the order of arithmetic floating-point operations.
- Consider use of the `-Xsfpc` option when possible. This option removes intermediary floating-point rounding operations and conversions whenever possible and carries additional bits to maintain precision.
- Consider use of the `-Xsno-accessor-aliasing` option. This option ignores dependencies between accessor arguments in a SYCL* kernel.

Recompile, Run, Profile, and Repeat

Once the code is optimized, it is important to measure the performance. The questions to be answered include:

- Did the metric improve?
- Is the performance goal met?
- Are there any more compute cycles left that can be used?

Confirm the results are correct. If you are comparing numerical results, the numbers may vary depending on how the compiler optimized the code or the modifications made to the code. Are any differences acceptable? If not, go back to optimization step.

Debugging

Debugging a DPC++ application can take advantage of the GDB*. The debugger is based on GDB, the GNU Debugger. For full GDB documentation, see https://www.gnu.org/software/gdb/documentation/.

Debugger Features

GDB is based on GDB 8.3 with multi-target extensions that adds support for Intel accelerator targets. It supports all GDB features that are applicable to the respective target and allows debugging the host application plus all supported devices within the same debug session.

Device code is presented as one or more additional inferiors. Inferiors are GDB's internal representation of each program execution. GDB automatically detects offloads to a supported device and creates a new inferior to debug device functions offloaded to that device. This feature is implemented using GDB's Python* scripting extension. It is important that the correct -data-directory is specified when invoking GDB.

The first version adds support for current Intel Graphics Technology and thus allows debugging device functions offloaded to Host, CPU, GPU, and FPGA emulation devices.

GDB plugs into Eclipse on Linux* host and Microsoft* Visual Studio* on Windows* host.

SIMD Support

NOTE This feature is only supported on GPU devices via GDB command line interface.

The debugger enables debugging of SIMD device code. Some commands, (for example `info register`, `list`, or stepping commands) operate on the underlying thread and therefore on all SIMD lanes at the same time. Other commands (for example `print`) operate on the currently selected SIMD lane.

The `info threads` command groups similar active SIMD lanes. The currently selected lane, however, is always shown in a separate row. If all SIMD lanes of a thread are inactive, the whole thread is marked as (inactive). The output appears as follows:

```
(gdb) info threads
  Id           Target Id           Frame
  1.1          Thread <id omitted>     <frame omitted>
  1.2          Thread <id omitted>     <frame omitted>
  2.1          Thread 1610612736    (inactive)
* 2.2:1        Thread 1073741824    <frame> at array-transform.cpp:54
  2.2:[3 5 7]  Thread 1073741824    <frame> at array-transform.cpp:54
  2.3:[1 3 5 7] Thread 1073741888    <frame> at array-transform.cpp:54
  2.4:[1 3 5 7] Thread 1073742080    <frame> at array-transform.cpp:54
  2.5:[1 3 5 7] Thread 1073742144    <frame> at array-transform.cpp:54
  2.6:[1 3 5 7] Thread 1073742336    <frame> at array-transform.cpp:54
  2.7:[1 3 5 7] Thread 1073745920    <frame> at array-transform.cpp:54
  2.8:[1 3 5 7] Thread 1073746176    <frame> at array-transform.cpp:54
  2.9:[1 3 5 7] Thread 1073746432    <frame> at array-transform.cpp:54 [...]
```

When a thread stops, such as after hitting a breakpoint, the debugger chooses the SIMD lane. Use the `thread` command to switch to a different lane.

> **NOTE** SIMD lane switching is only supported via GDB command line. When stopped at a breakpoint in Visual Studio, GDB sets the correct SIMD lane, but there is not a way to change the lane using Visual Studio.

The SIMD lane is specified by an optional lane number separated by ':' from the thread number. To switch to lane 2 of the current thread or to switch to lane 5 in thread 3 in inferior 2, use:

```
(gdb) thread :2
(gdb) thread 2.3:5
```

respectively. When not specifying a SIMD lane, GDB preserves the previously selected lane or, lacking a previously selected lane, chooses one. When single-stepping a thread, the debugger also tries to preserve the currently selected SIMD lane.

When using the `thread apply` command, the specified command is applied to all active lanes of a thread sequentially. For example, the step command in `thread apply` is issued as many times as the number of active SIMD lanes in the thread. If `thread 2` has three active lanes, `thread apply 2 step` results in `thread 2` making the step three times.

Operating System Differences

On Linux, the GDB debug engine is used to debug both the host process and all device code. Once Eclipse has been configured to launch GDB, the debug experience should be similar to debugging a client/server application with standard GDB. GDB (Python) scripts have access to both the host and the device part.

On Windows, the Microsoft debug engine is used to debug the host process and GDB is used to debug the device code. The parts are combined in Visual Studio and presented in a single debug session. GDB (Python) scripts only have access to the device part.

Environment Setup

There are some required steps to set up the Linux or Windows environment for application debugging. Detailed instructions are available from Get Started with Debugging Data Parallel C++ for Linux OS Host or Get Started with Debugging Data Parallel C++ for Windows OS Host.

Breakpoints

Unless specified otherwise, breakpoints are set for all threads in all inferiors. GDB takes care to automatically insert and remove breakpoints into device code as new device modules are created and new device functions are launched.

Breakpoint conditions are evaluated inside the debugger. The use of conditional breakpoints may incur a noticeable performance overhead as threads may frequently be stopped and resumed again to evaluate the breakpoint condition.

Evaluations and Data Races

The debugger may be configured to stop all other threads in an inferior (all-stop mode) on an event or to leave other threads running (non-stop mode). It does not stop other inferiors. Data that is shared between host and device, between two devices, or that is shared between threads (when the debugger is in non-stop mode) may be modified while the debugger is trying to access it, for example, in an expression evaluation.

To guarantee that the debugger's data accesses do not race with debugger accesses, all threads that may access that data need to be stopped for the duration of this command.

NOTE Non-stop mode is not supported on GPU devices.

Linux Sample Session

```
dpcpp -g -lsycl -o sample sample.cpp
export SYCL_PROGRAM_BUILD_OPTIONS="-g -cl-opt-disable"
gdb ./ sample

GNU gdb (GDB) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.; (C) 2019 Intel Corp.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.

For information about how to find Technical Support, Product Updates,
User Forums, FAQs, tips and tricks, and other support information, please visit:
<http://www.gnu.org/software/gdb/bugs/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sample...
(gdb) break 34
Breakpoint 1 at 0x40da58: file sample.cpp, line 34.
(gdb) run
Starting program: ./sample
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff5127700 (LWP 1695)]

Thread 1 "sample" hit Breakpoint -20, 0x00007ffff7fe7670 in isDebuggerActive () from
libigfxdbgxchg64.so
[New inferior 2]
Added inferior 2 on target 1 (native)
Intelgt auto-attach: a gdbserver will be attached using host`s inferior pid, i.e. 1695.
```

```
[Switching to inferior 2 [<null>] (<noexec>)]
Attached; pid = 1695
Remote debugging using stdio
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000ffffd020 in ?? ()
[Switching to inferior 1 [process 1695] (sample)]
[Switching to thread 1.1 (Thread 0x7ffff7fdbec0 (LWP 1695))]
#0  0x00007ffff7fe7670 in isDebuggerActive () from libigfxdbgxchg64.so
Intelgt auto-attach: a new inferior (Num: 2) has been added and an Intel GT gdbserver has been
created to listen to GT debug events.

Thread 1.1 "sample" hit Breakpoint -20, 0x00007ffff7fe7670 in isDebuggerActive () from
libigfxdbgxchg64.so
[New Thread 1073741824]
Reading default.gtelf from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files
locally instead.
Reading /tmp/_ZTSN2cl4sycl6kernelE.dbgelf from remote target...
[New Thread 1073741840]
[New Thread 1073741888]
[New Thread 1073742080]
[New Thread 1073742096]
[New Thread 1073742144]
[New Thread 1073742336]
[New Thread 1073742352]
[New Thread 1073742400]
[New Thread 1073745920]
[New Thread 1073745936]
[New Thread 1073745984]
[New Thread 1073746176]
[New Thread 1073746240]
[New Thread 1073746432]
[New Thread 1073746496]
[Switching to Thread 1073741824 lane 0]

Thread 2.2:0 hit Breakpoint 1, main::$_0::operator()(cl::sycl::handler&) const::
{lambda(cl::sycl::id<1>)#1}::operator()(cl::sycl::id<1>) const (this=0x0, wiID=...) at
sample.cpp:34
34              accessorOut[wiID] = in_elem + 100;
(gdb) info inferiors
Num  Description       Connection                                Executable
1    process 1695        1 (native)                                sample
* 2    Remote target     2 (remote gdbserver-gt --attach - 1695)
 (gdb) list
29
30            cgh.parallel_for<class kernel> (dataRange, [=] (id<1> wiID)
31            {
32              int dim0 = wiID[0];
33              int in_elem = accessorIn[wiID];
34              accessorOut[wiID] = in_elem + 100;
35            });
36          });
37        }
38
(gdb) print in_elem
$1 = 0
 (gdb) disassemble $pc, +36
```

```
Dump of assembler code from 0xfffdd570 to 0xfffdd594:
=> 0x00000000fffdd570 <_ZTSN2cl4sycl6kernelE(int*, cl::sycl::range<1>, cl::sycl::range<1>,
cl::sycl::id<1>, int*, cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>)+83312>:
(W)      send (16|M0)              r8:uw    r75     0xA         0x22C1103  {Breakpoint} //    wr:
1h+?, rd:2, Scratch Block Read (2grfs from 0x103)
   0x00000000fffdd580 <_ZTSN2cl4sycl6kernelE(int*, cl::sycl::range<1>, cl::sycl::range<1>,
cl::sycl::id<1>, int*, cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>)
+83328>:               send (8|M0)               r8:d     r8:uq   0xC         0x41401FF  //    wr:
2+?, rd:1, A64 Scattered Read msc:1, to 0x0
   0x00000000fffdd590 <_ZTSN2cl4sycl6kernelE(int*, cl::sycl::range<1>, cl::sycl::range<1>,
cl::sycl::id<1>, int*, cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>)
+83344>:               add (8|M0)                r8.0<1>:d     r8.0<8;8,1>:d     100:w
End of assembler dump.
```

Migrating Code to DPC++

Code written in other programming languages, such as C++ or OpenCL™, can be migrated to DPC++ code for use on multiple devices. The steps used to complete the migration vary based on the original language.

Migrating from C++ to SYCL/DPC++

SYCL is a single-source style programming model based on C++. It builds on features of C++11 and has additional support for C++14 and enables C++17 Parallel STL programs to be accelerated on OpenCL™ devices. Some of the supported C++ features include templates, classes, operator overloading, lambda, and static polymorphism.

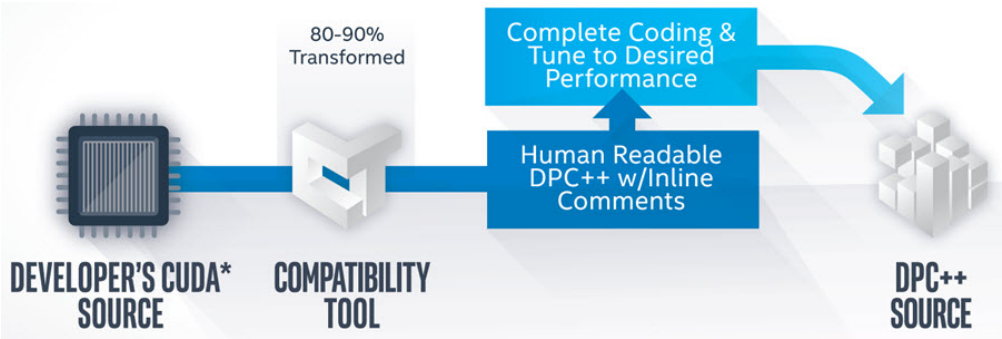
When accelerating an existing C++ application on OpenCL devices, SYCL provides seamless integration as most of the C++ code remains intact. Refer to sections within oneAPI Programming Model for SYCL constructs to enable device side compilation.

Migrating from CUDA* to DPC++

The Intel DPC++ Compatibility Tool is part of the Intel oneAPI Base Toolkit. The goal of this tool is to assist developers employing NVIDIA* CUDA or other languages in the future to migrate their applications to benefit from DPC++. This tool generates DPC++ code as much as possible. However, depending on the complexity of the code, it will not migrate all code and manual changes may be required. The tool provides help with IDE plug-ins, a user guide, and embedded comments in the code to complete the migration to DPC++.

Source File Workflow

The following workflow uses a command line with multiple source files:



1. Run the Intel DPC++ Compatibility Tool using the following inputs:
   - Original source code
   - Options, macros, and settings

- Applicable source language header files, including header files for the libraries used in the sources

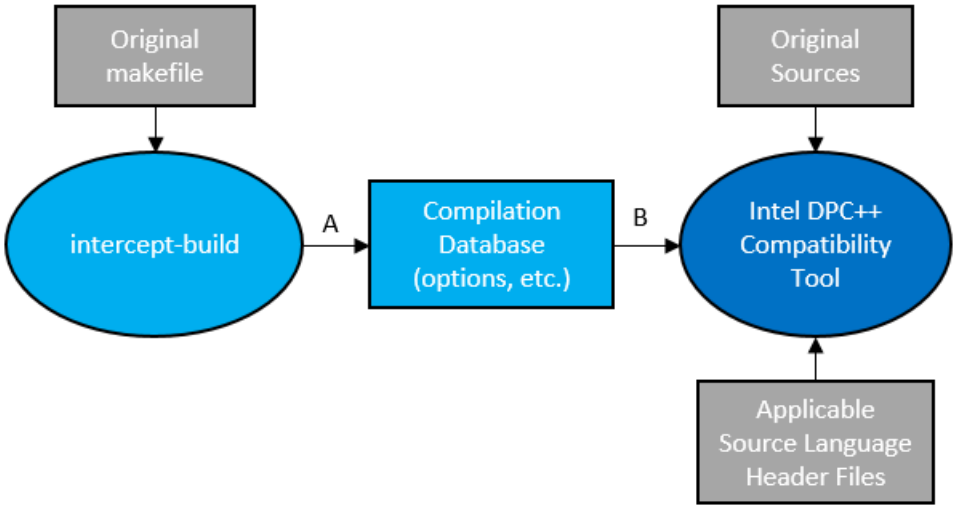
Command line example for the Intel DPC++ Compatibility Tool:

```
dpct --in-root=./foo --out-root=./result ./foo/main.cu ./foo/bar/util.cu --extra-arg="-I./foo/
bar/"
```

2. After the DPC++ code is output, identify any areas that require additional attention with the help of embedded comments and modify the code to create the final DPC++ code.

Makefile Workflow

The following workflow uses a command line with a makefile project:



Command line example for intercept-build (A):

```
intercept-build make
```

Command line example for Intel DPC++ Compatibility Tool (B):

```
dpct -p compile_commands.json --in-root=../.. --out-root=output program.cu
```

Additional information is available from Get Started with the Intel® DPC++ Compatibility Tool and the Intel® DPC++ Compatibility Tool User Guide.

Migrating from OpenCL Code to DPC++

In the current version of DPC++, the runtime employs OpenCL code to enact the parallelism. DPC++ typically requires fewer lines of code to implement kernels and also fewer calls to essential API functions and methods. It enables creation of OpenCL programs by embedding the device source code in line with the host source code.

Most of the OpenCL application developers are aware of the somewhat verbose setup code that goes with offloading kernels on devices. Using DPC++, it is possible to develop a clean, modern C++ based application without most of the setup associated with OpenCL C code. This reduces the learning effort and allows for focus on parallelization techniques.

However, OpenCL application features can continue to be used via the SYCL API. The updated code can use as much or as little of the SYCL interface as desired.

Migrating Between CPU, GPU, and FPGA

In DPC++, a platform consists of a host device connected to zero or more devices, such as CPU, GPU, FPGA, or other kinds of accelerators and processors.

When a platform has multiple devices, design the application to offload some or most of the work to the devices. There are different ways to distribute work across devices in the oneAPI programming model:

1. Initialize device selector – SYCL provides a set of classes called selectors that allow manual selection of devices in the platform or let oneAPI runtime heuristics choose a default device based on the compute power available on the devices.
2. Splitting datasets – With a highly parallel application with no data dependency, explicitly divide the datasets to employ different devices. The following code sample is an example of dispatching workloads across multiple devices. Use `dpcpp snippet.cpp` to compile the code.

```
int main() {
   int data[1024];
   for (int i = 0; i < 1024; i++)
       data[i] = i;
       try {
           cpu_selector cpuSelector;
           queue cpuQueue(cpuSelector);
           gpu_selector gpuSelector;
           queue gpuQueue(gpuSelector);
           buffer<int, 1> buf(data, range<1>(1024));
           cpuQueue.submit([&](handler& cgh) {
               auto ptr =
               buf.get_access<access::mode::read_write>(cgh);
               cgh.parallel_for<class divide>(range<1>(512),
                   [=](id<1> index) {
                   ptr[index] -= 1;
               });
           });
           gpuQueue.submit([&](handler& cgh1) {
               auto ptr =
               buf.get_access<access::mode::read_write>(cgh1);
               cgh1.parallel_for<class offset1>(range<1>(1024),
                   id<1>(512), [=](id<1> index) {
                       ptr[index] += 1;
               });
           });
           cpuQueue.wait();
           gpuQueue.wait();
      }
      catch (exception const& e) {
          std::cout <<
          "SYCL exception caught: " << e.what() << '\n';
          return 2;
      }
      return 0;
}
```

3. Target multiple kernels across devices – If the application has scope for parallelization on multiple independent kernels, employ different queues to target devices. The list of SYCL supported platforms can be obtained with the list of devices for each platform by calling `get_platforms()` and `platform.get_devices()` respectively. Once all the devices are identified, construct a queue per device and dispatch different kernels to different queues. The following code sample represents dispatching a kernel on multiple SYCL devices.

```
#include <stdio.h>
#include <vector>
#include <CL/sycl.hpp>
using namespace cl::sycl;
using namespace std;
int main()
{
```

```
      size_t N = 1024;
      vector<float> a(N, 10.0);
      vector<float> b(N, 10.0);
      vector<float> c_add(N, 0.0);
      vector<float> c_mul(N, 0.0);
  {
      buffer<float, 1> abuffer(a.data(), range<1>(N),
        { property::buffer::use_host_ptr() });
      buffer<float, 1> bbuffer(b.data(), range<1>(N),
        { property::buffer::use_host_ptr() });
      buffer<float, 1> c_addbuffer(c_add.data(), range<1>(N),
       { property::buffer::use_host_ptr() });
      buffer<float, 1> c_mulbuffer(c_mul.data(), range<1>(N),
        { property::buffer::use_host_ptr() });
   try {
            gpu_selector gpuSelector;
            auto queue = cl::sycl::queue(gpuSelector);
            queue.submit([&](cl::sycl::handler& cgh) {
                  auto a_acc = abuffer.template
                    get_access<access::mode::read>(cgh);
                  auto b_acc = bbuffer.template
                    get_access<access::mode::read>(cgh);
                  auto c_acc_add = c_addbuffer.template
                    get_access<access::mode::write>(cgh);
                  cgh.parallel_for<class VectorAdd>
                   (range<1>(N), [=](id<1> it) {
                       //int i = it.get_global();
                           c_acc_add[it] = a_acc[it] + b_acc[it];
                                });
                             });
            cpu_selector cpuSelector;
            auto queue1 = cl::sycl::queue(cpuSelector);
            queue1.submit([&](cl::sycl::handler& cgh) {
                  auto a_acc = abuffer.template
                      get_access<access::mode::read>(cgh);
                  auto b_acc = bbuffer.template
                      get_access<access::mode::read>(cgh);
                  auto c_acc_mul = c_mulbuffer.template
                      get_access<access::mode::write>(cgh);
                  cgh.parallel_for<class VectorMul>
                   (range<1>(N), [=](id<1> it) {
                        c_acc_mul[it] = a_acc[it] * b_acc[it];
                                });
                              });
             }
             catch (cl::sycl::exception e) {
/* In the case of an exception being throw, print the
error message and
                    * return 1. */
                    std::cout << e.what();
                    return 1;
             }
      }
      for (int i = 0; i < 8; i++) {
            std::cout << c_add[i] << std::endl;
            std::cout << c_mul[i] << std::endl;
      }
      return 0;
}
```

Composability

The oneAPI programming model enables an ecosystem with support for the entire development toolchain. It includes compilers and libraries, debuggers, and analysis tools to support multiple accelerators like CPU, GPUs, FPGA, and more.

Compatibility with Other Compilers

The oneAPI programming model provides DPC++ and C++ programming support for single source heterogeneous programming. However, the compiler works with a variety of existing and new C++ compilers on the host and layers over OpenCL 1.2 code from diverse vendors.

OpenMP* Offload Interoperability

The oneAPI programming model provides a unified compiler based on LLVM/Clang with support for OpenMP* offload. This allows seamless integration that allows the use of OpenMP constructs to either parallelize host side applications or offload to a target device.

DPC++ is based on TBB runtime when executing device code on the CPU; hence, using both OpenMP and DPC++ on a CPU can lead to oversubscribing of threads. Performance analysis of workloads executing on the system could help determine if this is occurring.

OpenCL™ Code Interoperability

The oneAPI programming model enables developers to continue using all OpenCL code features via different parts of the SYCL API. The OpenCL code interoperability mode provided by SYCL helps reuse the existing OpenCL code while keeping the advantages of higher programming model interfaces provided by SYCL. There are 2 main parts in the interoperability mode:

1. To create SYCL objects from OpenCL code objects. For example, a SYCL buffer can be constructed from an OpenCL `cl_mem` or SYCL queue from a `cl_command_queue`.
2. To get OpenCL code objects from SYCL objects. For example, launching an OpenCL kernel that uses an implicit `cl_mem` associated to a SYCL accessor.

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 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-groups

Sub-groups are an Intel extension.

Work-groups

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.