Intel® oneAPI Programming Guide

Contents

Intel® oneAPI Programming Guide

1

Use this guide to learn about:

- Introduction to oneAPI Programming: A basic overview of oneAPI, Intel oneAPI Toolkits, and related resources.
- oneAPI Programming Model: An introduction to the oneAPI programming model for SYCL* and OpenMP* offload for C, C++, and Fortran.
- oneAPI Development Environment Setup: Instructions on how to set up the oneAPI application development environment.
- Compile and Run oneAPI Programs: Details about how to compile code for various accelerators (CPU, FPGA, etc.).
- API-based Programming: A brief introduction to common APIs and related libraries as well as details on buffer usage.
- Software Development Process: 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.).

Introduction to oneAPI Programming

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).

Central Processing Units (CPUs) and Graphics Processing Units (GPUs) are fundamental computing engines. But as computing demands evolve, it is not always clear what the differences are between CPUs and GPUs and which workloads are best suited to each.

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 (such as GPUs) requires different languages, libraries, and tools. That means each hardware platform requires 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 using SYCL*. SYCL 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 SYCL 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.

oneAPI also supports programming on CPUs and accelerators using the OpenMP* offload feature with existing C/C++ or Fortran code.

For more information on determining whether or not to use a CPU or GPU, see CPU vs. GPU: Making the Most of Both.

Once you have gained an understanding of the oneAPI programming model, see the oneAPI GPU Optimization Guide for information on how to optimize your software.

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

Learn more about the oneAPI initiative and programming model at oneapi.com. The site includes the oneAPI Specification, SYCL Language Guide and API Reference, and other resources.

Intel oneAPI Programming 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.

The oneAPI programming model
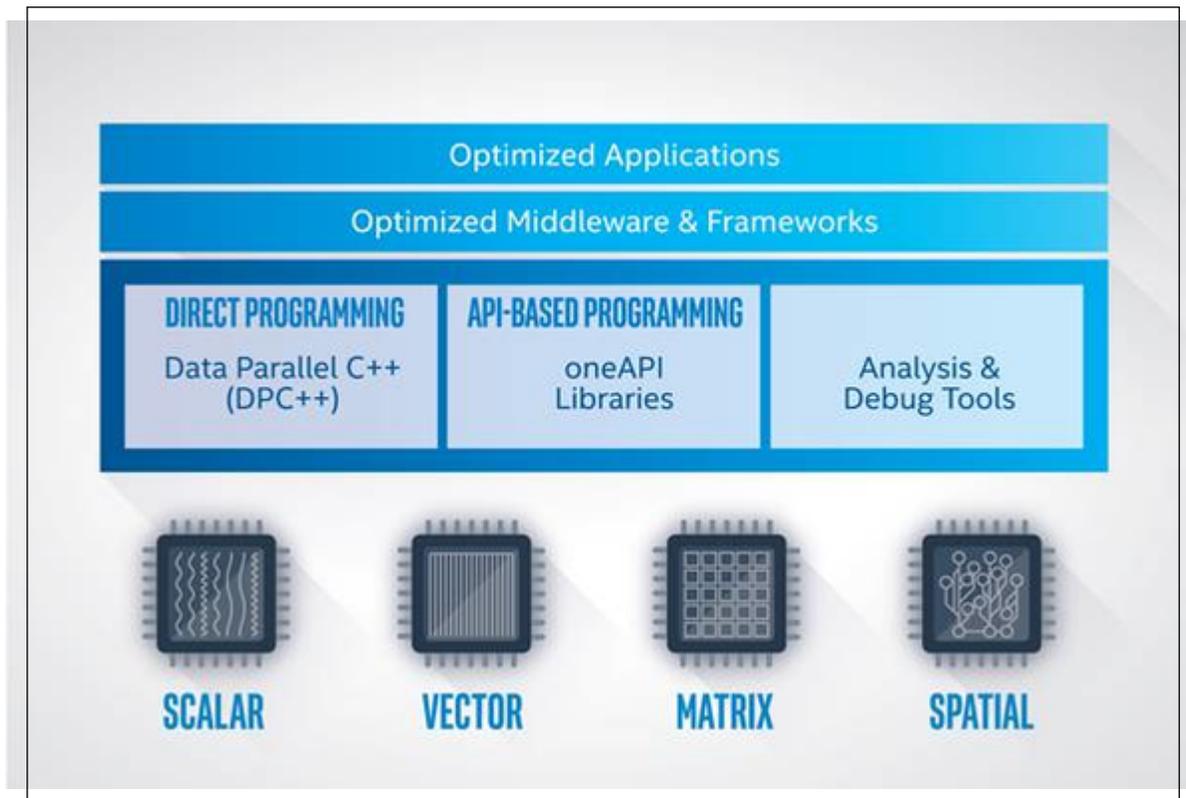
As shown in the figure above, applications that take advantage of the oneAPI programming model can run on multiple target hardware platforms ranging from CPU to FPGA. Intel offers oneAPI products as part of a set of toolkits. The Intel® oneAPI Base Toolkit, Intel® oneAPI HPC Toolkit, Intel® oneAPI IoT Toolkit, and several other toolkits feature complementary tools based on specific developer workload needs. For example, the Intel oneAPI Base Toolkit includes the Intel® oneAPI DPC++/C++ Compiler, the Intel® DPC++ Compatibility Tool, select libraries, and analysis tools.

- Developers who want to migrate existing CUDA* code to SYCL* for compilation with the DPC++ compiler can use the **Intel DPC++ Compatibility Tool** to help migrate their existing projects to SYCL* using DPC ++.
- The **Intel oneAPI DPC++/C++ Compiler** supports direct programming of code targeting accelerators. Direct programming is coding for performance when APIs are not available for the algorithms expressed in user code. 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 SYCL 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**.

The Intel oneAPI Base Toolkit is available as a free download from the Intel Developer Zone.

Users familiar with Intel® Parallel Studio and Intel® System Studio may be interested in the Intel oneAPI HPC Toolkit and Intel oneAPI IoT Toolkit respectively.

oneAPI Toolkit Distribution

oneAPI Toolkits are available via multiple distribution channels:

- Local product installation: install the oneAPI toolkits from the Intel® Developer Zone. Refer to the Installation Guides for specific install information.
- Install from containers or repositories: install the oneAPI toolkits from one of several supported containers or repositories. Instructions for each are available from the Installation Guides.
- Pre-installed in the Intel® DevCloud: use a free development sandbox for access to the latest Intel hardware and select oneAPI tools. Learn more about Intel DevCloud and sign up for free access.

Related Documentation

The following documents are useful starting points for developers getting started with oneAPI projects.

- Get started guides for select oneAPI toolkits:

    - Get Started with Intel oneAPI Base Toolkit for Linux* | Windows* | MacOS*
    - Get Started with Intel oneAPI HPC Toolkit for Linux* | Windows* | MacOS*
    - Get Started with Intel oneAPI IoT Toolkit for Linux* | Windows*
- Release notes for select oneAPI toolkits:

    - Intel oneAPI Base Toolkit
    - Intel oneAPI HPC Toolkit
    - Intel oneAPI IoT Toolkit
- Language reference material:

    - SYCL* Language Guide and API Reference
    - SYCL* Specification (PDF) 1.2.1 | 2020
    - Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL (book by James Reinders, Ben Ashbaugh, James Broadman, Michael Kinsner, John Pennycook, and Xinmin Tian, parts of this book were reused under the Creative Commons license.)
    - LLVM/OpenMP* Documentation
    - OpenMP* Specifications (examples documents recommended)

oneAPI Programming Model

In heterogenous computing, the host processor takes advantage of accelerator devices to execute code more efficiently.

The oneAPI programming model supports two important portable methods of heterogenous computing: Data Parallel C++ with SYCL* and OpenMP* for C, C++, and Fortran.

SYCL is a cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file. The DPC++ open source project is adding SYCL support to the LLVM C++ compiler. The Intel® oneAPI DPC++/C++ Compiler is available as part of the Intel oneAPI Base Toolkit.

OpenMP has been a standard programming language for over 20 years, and Intel implements version 5 of the OpenMP standard. The Intel oneAPI DPC++/C++ Compiler with OpenMP offload support is available as part of the Intel oneAPI Base Toolkit, Intel oneAPI HPC Toolkit, and Intel oneAPI IoT Toolkit. The Intel® Fortran Compiler Classic and Intel® Fortran Compiler with OpenMP offload support is available as part of the Intel oneAPI HPC Toolkit.

> **NOTE** OpenMP is not supported for FPGA devices.

The next sections briefly describe each language and provide pointers to more information.

Data Parallelism in C++ using SYCL*

Open, Multivendor, Multiarchitecture support for productive data parallel programming in C++ is accomplished via standard C++ with support for SYCL. SYCL (pronounced 'sickle') is a royalty-free, cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file. The DPC++ open source project is adding SYCL support to the LLVM C++ compiler.

Simple Sample Code Using Queue Lambda by Reference

The best way to introduce SYCL is through an example. Since SYCL is based on modern C++, this example uses several features that have been added to C++ in recent years, such as lambda functions and uniform initialization. Even if developers are not familiar with these features, their semantics will become clear from the context of the example. After gaining some experience with SYCL, these newer C++ features will become second nature.

The following application sets each element of an array to the value of its index, so that a[0] = 0, a[1] = 1, etc.

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

constexpr int num=16;
using namespace sycl;

int main() {
  auto r = range{num};
  buffer<int> a{r};

  queue{}.submit([&](handler& h) {
    accessor out{a, h};
    h.parallel_for(r, [=](item<1> idx) {
      out[idx] = idx;
    });
  });

  host_accessor result{a};
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";
}
```

The first thing to notice is that there is just one source file: both the host code and the offloaded accelerator code are combined in a single source file. The second thing to notice is that the syntax is standard C++: there aren't any new keywords or pragmas used to express the parallelism. Instead, the parallelism is expressed through C++ classes. For example, the `buffer` class on line 9 represents data that will be offloaded to the device, and the `queue` class on line 11 represents a connection from the host to the accelerator.

The logic of the example works as follows. Lines 8 and 9 create a buffer of 16 `int` elements, which have no initial value. This buffer acts like an array. Line 11 constructs a `queue`, which is a connection to an accelerator device. This simple example asks the SYCL runtime to choose a default accelerator device, but a more robust application would probably examine the topology of the system and choose a particular accelerator. Once the queue is created, the example calls the `submit()` member function to submit work to the accelerator. The parameter to this `submit()` function is a lambda function, which executes immediately on the host. The lambda function does two things. First, it creates an `accessor` on line 12, which can write elements in the buffer. Second, it calls the `parallel_for()` function on line 13 to execute code on the accelerator.

The call to `parallel_for()` takes two parameters. One parameter is a lambda function, and the other is the `range` object "r" that represents the number of elements in the buffer. SYCL arranges for this lambda to be called on the accelerator once for each index in that range, i.e. once for each element of the buffer. The lambda simply assigns a value to the buffer element by using the `out` accessor that was created on line 12. In this simple example, there are no dependencies between the invocations of the lambda, so the program is free to execute them in parallel in whatever way is most efficient for this accelerator.

After calling `parallel_for()`, the host part of the code continues running without waiting for the work to complete on the accelerator. However, the next thing the host does is to create a `host_accessor` on line 18, which reads the elements of the buffer. The SYCL runtime knows this buffer is written by the accelerator, so the `host_accessor` constructor (line 18) is blocked until the work submitted by the `parallel_for()` is complete. Once the accelerator work completes, the host code continues past line 18, and it uses the `out` accessor to read values from the buffer.

Additional Resources

This introduction to SYCL is not meant to be a complete tutorial. Rather, it just gives you a flavor of the language. There are many more features to learn, including features that allow you to take advantage of common accelerator hardware such as local memory, barriers, and SIMD. There are also features that let you submit work to many accelerator devices at once, allowing a single application to run work in parallel on many devices simultaneously.

The following resources are useful to learning and mastering SYCL using a DPC++ compiler:

- Explore SYCL with Samples from Intel provides an overview and links to simple sample applications available from GitHub*.
- The DPC++ Foundations Code Sample Walk-Through is a detailed examination of the Vector Add sample code, the DPC++ equivalent to a basic Hello World application.
- The oneapi.com site includes a Language Guide and API Reference with descriptions of classes and their interfaces. It also provides details on the four programming models - platform model, execution model, memory model, and kernel programming model.
- The DPC++ Essentials training course is a guided learning path for SYCL using Jupyter* Notebooks on Intel® DevCloud.
- Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL is a comprehensive book that introduces and explains key programming concepts and language details about SYCL and Heterogeneous programming.

C/C++ or Fortran with OpenMP* Offload Programming Model

The Intel® oneAPI DPC++/C++ Compiler and the Intel® Fortran Compiler enable software developers to use OpenMP* directives to offload work to Intel accelerators to improve the performance of applications.

This section describes the use of OpenMP directives to target computations to the accelerator. Developers unfamiliar with OpenMP directives can find basic usage information documented in the OpenMP Support sections of the Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference or Intel® Fortran Compiler for oneAPI Developer Guide and Reference.

NOTE OpenMP is not supported for FPGA devices.

Basic OpenMP Target Construct

The OpenMP target construct is used to transfer control from the host to the target device. Variables are mapped between the host and the target device. The host thread waits until the offloaded computations are complete. Other OpenMP tasks may be used for asynchronous execution on the host; use the `nowait` clause to specify that the encountering thread does not wait for the target region to complete.

C/C++

The C++ code snippet below targets a SAXPY computation to the accelerator.

```
#pragma omp target map(tofrom:fa), map(to:fb,a)
#pragma omp parallel for firstprivate(a)
for(k=0; k<FLOPS_ARRAY_SIZE; k++)
        fa[k] = a * fa[k] + fb[k]
```

Array `fa` is mapped both to and from the accelerator since `fa` is both input to and output from the calculation. Array `fb` and the variable `a` are required as input to the calculation and are not modified, so there is no need to copy them out. The variable `FLOPS_ARRAY_SIZE` is implicitly mapped to the accelerator. The loop index `k` is implicitly private according to the OpenMP specification.

Fortran

This Fortran code snippet targets a matrix multiply to the accelerator.

```
!$omp target map(to: a, b ) map(tofrom: c )
!$omp parallel do private(j,i,k)
      do j=1,n
        do i=1,n
          do k=1,n
              c(i,j) = c(i,j) + a(i,k) * b(k,j)
          enddo
        enddo
      enddo
!$omp end parallel do
!$omp end target
```

Arrays `a` and `b` are mapped to the accelerator, while array `c` is both input to and output from the accelerator. The variable `n` is implicitly mapped to the accelerator. The private clause is optional since loop indices are automatically private according to the OpenMP specification.

Map Variables

To optimize data sharing between the host and the accelerator, the target data directive maps variables to the accelerator and the variables remain in the target data region for the extent of that region. This feature is useful when mapping variables across multiple target regions.

C/C++

```
#pragma omp target data [clause[[,] clause],...]
 structured-block
```

Fortran

```
!$omp target data [clause[[,] clause],...]
structured-block
!$omp end target data
```

Clauses

The clauses can be one or more of the following. See TARGET DATA for more information.

- DEVICE (integer-expression)
- IF ([TARGET DATA:] scalar-logical-expression)
- MAP ([[map-type-modifier[,]] map-type:] list)

NOTE Map type can be one or more of the following:

- `alloc`
- `to`
- `from`
- `tofrom`
- `delete`
- `release`

- SUBDEVICE ([integer-constant ,] integer-expression [: integer-expression [: integer-expression]])
- USE_DEVICE_ADDR (list) // available only in `ifx`
- USE_DEVICE_PTR (ptr-list)

```
DEVICE (integer-expression)
IF ([TARGET DATA:] scalar-logical-expression)
MAP ([[map-type-modifier[,]] map-type: alloc | to | from | tofrom | delete | release] list)
SUBDEVICE ([integer-constant ,] integer-expression [ : integer-expression [ : integer-
expression]])
USE_DEVICE_ADDR (list) // available only in ifx
USE_DEVICE_PTR (ptr-list)
```

Use the target update directive or *always* map-type-modifier in map clause to synchronize an original variable in the host with the corresponding variable in the device.

Compile to Use OMP TARGET

The following example commands illustrate how to compile an application using OpenMP target.

C/C++

- Linux:

```
icx -fiopenmp -fopenmp-targets=spir64 code.c
```
- Windows (you can use `icx` or `icpx`):

```
icx /Qiopenmp /Qopenmp-targets=spir64 code.c
```

Fortran

- Linux:

```
ifx -fiopenmp -fopenmp-targets=spir64 code.f90
```
- Windows:

```
ifx /Qiopenmp /Qopenmp-targets=spir64 code.f90
```

Additional OpenMP Offload Resources

- Intel offers code samples that demonstrate using OpenMP directives to target accelerators at https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming. Specific samples include:

  - Matrix Multiplication is a simple program that multiplies together two large matrices and verifies the results. This program is implemented using two ways: SYCL* and OpenMP.
  - The ISO3DFD sample refers to Three-Dimensional Finite-Difference Wave Propagation in Isotropic Media. The sample is a three-dimensional stencil used to simulate a wave propagating in a 3D isotropic medium. The sample shows some of the more common challenges and techniques when targeting OMP accelerator devices in more complex applications to achieve good performance.
  - openmp_reduction is a simple program that calculates pi. This program is implemented using C++ and OpenMP for CPUs and accelerators based on Intel® Architecture.
- Get Started with OpenMP* Offload Feature
- Details on using Intel compilers with OpenMP offload, including lists of supported options and example code, is available in the compiler developer guides:

  - Intel® oneAPI DPC+/C+ Compiler Developer Guide and Reference
  - Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer Guide
- LLVM/OpenMP Runtimes describes the distinct types of runtimes available and can be helpful when debugging OpenMP offload.
- openmp.org has an examples document: https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf. Chapter 4 of the examples document focuses on accelerator devices and the target construct.
- *Using OpenMP - the Next Step* is a good OpenMP reference book. Chapter 6 covers OpenMP support for heterogeneous systems. For additional information on this book, see https://www.openmp.org/tech/using-openmp-next-step.

Device Selection

Offloading code to a device (such as a CPU, GPU, or FPGA) is available for both DPC++ and OpenMP* applications.

DPC++ Device Selection in the Host Code

Host code can explicitly select a device type. To do select a device, select a queue and initialize its device with one of the following:

- `default_selector`
- `cpu_selector`
- `gpu_selector`
- `accelerator_selector`

If `default_selector` is used, the kernel runs based on a heuristic that chooses from available compute devices (all, or a subset based on the value of the `SYCL_DEVICE_FILTER` environment variable).

If a specific device type (such as `cpu_selector` or `gpu_selector`) is used, then it is expected that the specified device type is available in the platform or included in the filter specified by `SYCL_DEVICE_FILTER`. If such a device is not available, then the runtime system throws an exception indicating that the requested device is not available. This error can be thrown in the situation where an ahead-of-time (AOT) compiled binary is run in a platform that does not contain the specified device type.

> **NOTE** While DPC++ applications can run on any supported target hardware, tuning is required to derive 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.

`SYCL_DEVICE_FILTER` is a complex environment variable that allows you to limit the runtimes, compute device types, and compute device IDs that may be used by the DPC++ runtime to a subset of all available combinations. The compute device IDs correspond to those returned by the SYCL API, `clinfo`, or `sycl-ls` (with the numbering starting at 0). They have no relation to whether the device with that ID is of a certain type or supports a specific runtime. Using a programmatic special selector (like `gpu_selector`) to request a filtered out device will cause an exception to be thrown. Refer to the environment variable description in GitHub for details on use and example values: https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md.

The `sycl-ls` tool enumerates a list of devices available in the system. It is strongly recommended to run this tool before running any SYCL or DPC++ programs to make sure the system is configured properly. As a part of enumeration, `sycl-ls` prints the `SYCL_DEVICE_FILTER` string as a prefix of each device listing. The format of the `sycl-ls` output is `[SYCL_DEVICE_FILTER] Platform_name, Device_name, Device_version [driver_version]`. In the following example, the string enclosed in the bracket ([ ]) at the beginning of each line is the `SYCL_DEVICE_FILTER` string used to designate the specific device on which the program will run.

Device Selection Example

```
$ sycl-ls
[opencl:acc:0] Intel® FPGA Emulation Platform for OpenCL™, Intel® FPGA Emulation Device 1.2
[2021.12.9.0.24_005321]
[opencl:gpu:1] Intel® OpenCL HD Graphics, Intel® UHD Graphics 630 [0x3e92] 3.0 [21.37.20939]
[opencl:cpu:2] Intel® OpenCL, Intel® Core™ i7-8700 CPU @ 3.20GHz 3.0 [2021.12.9.0.24_005321]
[level_zero:gpu:0] Intel® Level-Zero, Intel® UHD Graphics 630 [0x3e92] 1.1 [1.2.20939]
[host:host:0] SYCL host platform, SYCL host device 1.2 [1.2]
```

Additional information about device selection is available from the DPC++ Language Guide and API Reference.

OpenMP* Device Query and Selection in the Host Code

OpenMP provided a set of APIs for programmers to query and set device for running code on the device. Host code can explicitly select and set a device num. For each offloading region, a programmer can also use a **device** clause to specify the target device that is to be used for executing the offloading region.

- `int omp_get_num_procs (void)` routine returns the number of processors available to the device
- `void omp_set_default_device(int device_num)` routine controls the default target device
- `int omp_get_default_device(void)` routine returns the default target device
- `int omp_get_num_devices(void)` routine returns the number of non-host devices available for offloading code or data.
- `int omp_get_device_num(void)` routine returns the device number of the device on which the calling thread is executing.
- `int omp_is_initial_device(int device_num)` routine returns *true* if the current task is executing on the host device; otherwise, it returns *false*.
- `int omp_get_initial_device(void)` routine returns a device number that represents the host device.

A programmer can use the environment variable `LIBOMPTARGET_DEVICETYPE = [ CPU | GPU ]` to perform a device type selection. If a specific device type such as CPU or GPU is specified, then it is expected that the specified device type is available in the platform. If such a device is not available, then the runtime system throws an error that the requested device type is not available if the environment variable `OMP_TARGET_OFFLOAD=mandatory`, otherwise, the execution will have a fallback execution on its initial device. Additional information about device selection is available from the OpenMP 5.1 specification. Details about environment variables are available from GitHub: https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md.

See Also

- Device Selectors for FPGA

SYCL* Thread and Memory Hierarchy

Thread Hierarchy

The SYCL* execution model exposes an abstract view of GPU execution. The SYCL thread hierarchy consists of a 1-, 2-, or 3-dimensional grid of work-items. These work-items are grouped into equal sized thread groups called work-groups. Threads in a work-group are further divided into equal sized vector groups called sub-groups.
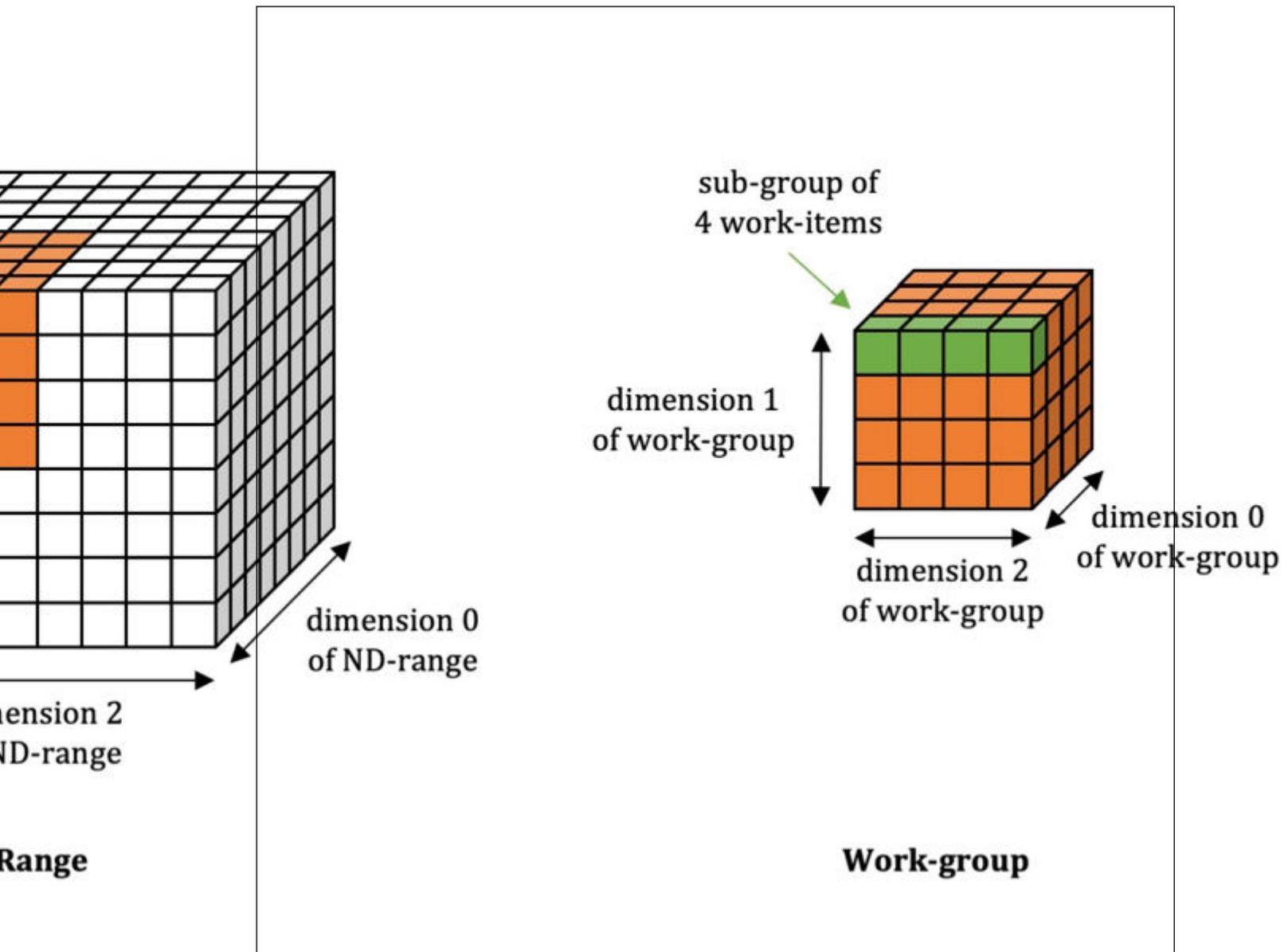
To learn more about how this hierarchy works with a GPUor a CPU with Intel® UHD Graphics, see SYCL* Thread Mapping and GPU Occupancy in the oneAPI GPU Optimization Guide.

Memory Hierarchy

The General Purpose GPU (GPGPU) compute model consists of a host connected to one or more compute devices. Each compute device consists of many GPU Compute Engines (CE), also known as Execution Units (EU) or Xe Vector Engines (XVE). The compute devices may also include caches, shared local memory (SLM), high-bandwidth memory (HBM), and so on, as shown in the figure below. Applications are then built as a combination of host software (per the host framework) and kernels submitted by the host to run on the VEs with a predefined decoupling point.
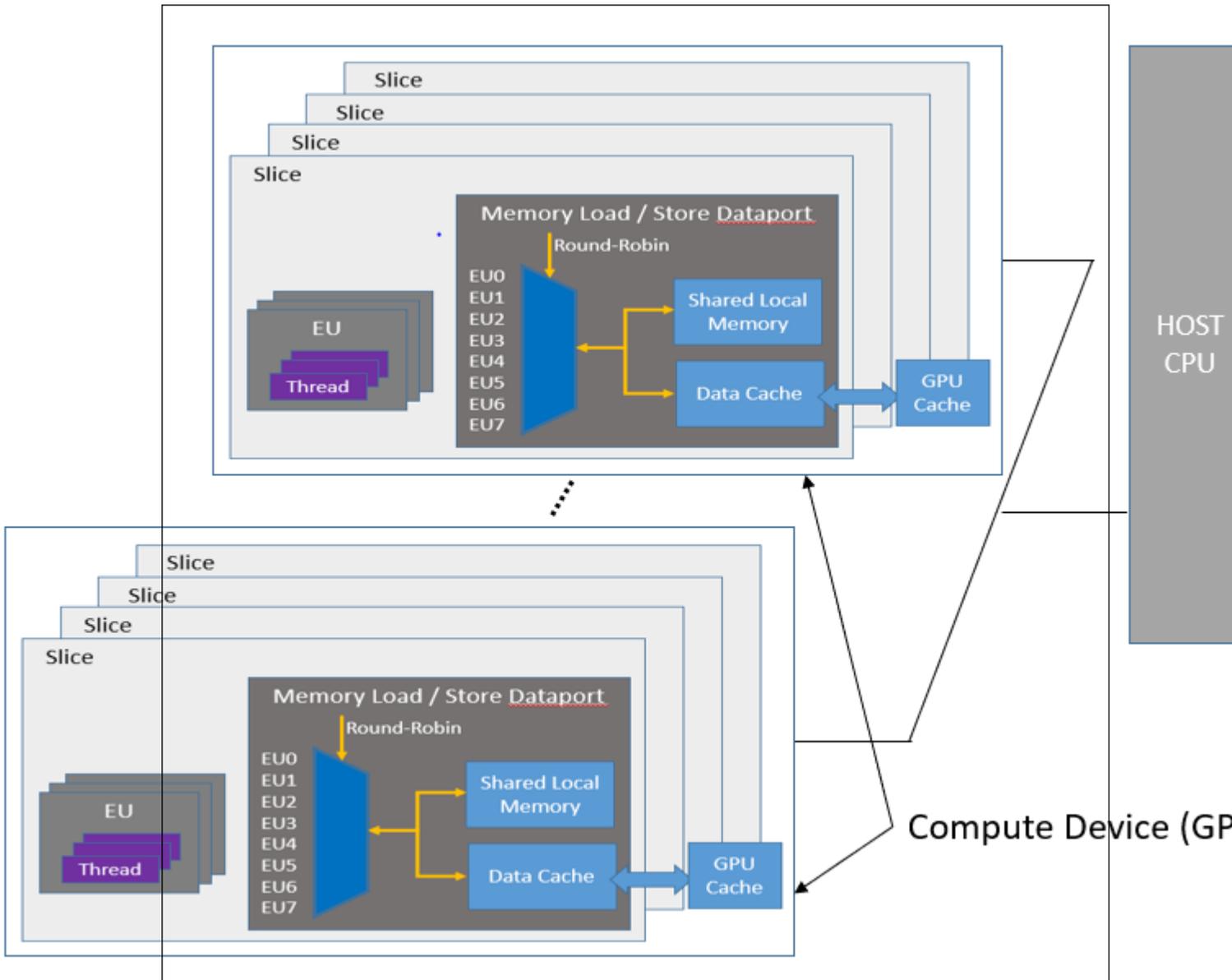
To learn more about memory hierarchy within the General Purpose GPU (GPGPU) compute model, see Execution Model Overview in the oneAPI GPU Optimization Guide.

oneAPI Development Environment Setup

The Intel® oneAPI tools are available in several convenient forms, as detailed in oneAPI Toolkit Distribution earlier in this guide. Follow the instructions in the Intel oneAPI Toolkit Installation Guide to obtain and install the tools.

Install Directories

On a Windows* system, the Intel oneAPI development tools are typically installed in the `C:\Program Files (x86)\Intel\oneAPI\` directory.

On Linux* or macOS* system, the Intel oneAPI development tools are typically installed in the `/opt/intel/oneapi/` directory.

These are the default locations; the precise location can be changed during installation.

Within the oneAPI installation directory are a collection of folders that contain the compilers, libraries, analyzers, and other tools installed on the development system. The precise list depends on the toolkit(s) installed and the options selected during installation. Most of the folders within the oneAPI installation directory have obvious names. For example, the mkl folder contains the Intel® oneAPI Math Kernel Library (Intel® oneMKL), the ipp folder contains the Intel® Integrated Performance Primitives (Intel® IPP) library, and so on.

Environment Variables

Some of the tools in the Intel oneAPI toolkits depend on environment variables to:

- Assist the compilation and link process (e.g., PATH, CPATH, INCLUDE, etc.)
- Locate debuggers, analyzers, and local help files (e.g., PATH, MANPATH)
- Identify tool-specific parameters and dynamic (shared) link libraries (e.g., LD_LIBRARY_PATH, CONDA_*, etc.)

setvars and vars Files

Every installation of the Intel oneAPI toolkits includes a single top-level "setvars" script and multiple tool-specific "vars" scripts (`setvars.sh` and `vars.sh` on Linux and macOS; `setvars.bat` and `vars.bat` on Windows). When executed (sourced), these scripts configure the local environment variables to reflect the needs of the installed Intel oneAPI development tools.

The following sections provide detailed instructions on how to use the oneAPI setvars and vars scripts to initialize the oneAPI development environment:

- Use the setvars Script with Windows*
- Use the setvars Script with Linux* or MacOS*

Modulefiles (Linux only)

Users of Environment Modules and Lmod can use the modulefiles included with the oneAPI toolkit installation to initialize their development environment variables. The oneAPI modulefile scripts are only supported on Linux and are provided as an alternative to using the setvars and vars scripts referenced above. In general, users should not mix modulefiles with the setvars environment scripts.

See Use Modulefiles with Linux* for detailed instructions on how to use the oneAPI modulefiles to initialize the oneAPI development environment.

Use the setvars Script with Windows*

Most of the oneAPI component tool folders contain an environment script named `vars.bat` that configures the environment variables needed by that component to support oneAPI development work. For example, in a default installation, the Intel® Integrated Performance Primitives (Intel® IPP) vars script on Windows is located at: `C:\Program Files (x86)\Intel\oneAPI\ipp\latest\env\vars.bat`. This pattern is shared by all oneAPI components that include an environment `vars` setup script.

These component tool `vars` scripts can be called directly or collectively. To call them collectively, a script named `setvars.bat` is provided in the oneAPI installation folder. For example, in a default installation on a Windows machine: `C:\Program Files (x86)\Intel\oneAPI\setvars.bat`.

Running the `setvars.bat` script without any arguments causes it to locate and run all `<component>\latest\env\vars.bat` scripts in the installation. Changes made to the environment by these scripts can be seen by running the Windows `set` command after running the environment setup scripts.

Visual Studio Code* developers can install a oneAPI environment extension to run the `setvars.bat` within Visual Studio Code. Learn more in Using Visual Studio Code with Intel oneAPI Toolkits.

> **NOTE** Changes to your environment made by running the `setvars.bat` script (or the individual `vars.bat` scripts) are not permanent. Those changes only apply to the `cmd.exe` session in which the `setvars.bat` environment script was executed.

Command Line Arguments

The `setvars.bat` script supports several command-line arguments, which are displayed using the `--help` option. For example:

```
"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" --help
```

The `--config=file` argument and the ability to include arguments that will be passed to the `vars.bat` scripts that are called by the `setvars.bat` script can be used to customize the environment setup.

The `--config=file` argument provides the ability to limit environment initialization to a specific set of oneAPI components. It also provides a way to initialize the environment for specific component versions. For example, to limit environment setup to just the Intel® IPP library and the Intel® oneAPI Math Kernel Library (Intel® oneMKL), pass a config file that tells the `setvars.bat` script to only call the `vars.bat` environment scripts for those two oneAPI components. More details and examples are provided in Use a Config file for setvars.bat on Windows.

Any extra arguments passed on the `setvars.bat` command line that are not described in the `setvars.bat` help message will be passed to every called `vars.bat` script. That is, if the `setvars.bat` script does not recognize an argument, it assumes the argument is meant for use by one or more component vars scripts and passes those extra arguments to every component `vars.bat` script that it calls. The most common extra arguments are `ia32` and `intel64`, which are used by the Intel compilers and the IPP, MKL, and TBB libraries to specify the application target architecture.

If more than one version of Microsoft Visual Studio* is installed on your system, you can specify which Visual Studio environment should be initialized as part of the oneAPI `setvars.bat` environment initialization by adding the `vs2017`, `vs2019`, or `vs2022` argument to the `setvars.bat` command line. By default, the most recent version of Visual Studio is located and initialized.

> **NOTE** Support for Microsoft Visual Studio* 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

Inspect the individual `vars.bat` scripts to determine which, if any, command line arguments they accept.

How to Run

```
<install-dir>\setvars.bat
```

To run `setvars.bat` or a `vars.bat` script in a PowerShell window, use the following:

```
cmd.exe "/K" '"C:\Program Files (x86)\Intel\oneAPI\setvars.bat" && powershell'
```

How to Verify

After executing `setvars.bat`, verify success by searching for the SETVARS_COMPLETED environment variable. If `setvars.bat` was successful the SETVARS_COMPLETED environment variable will have a value of 1:

```
set | find "SETVARS_COMPLETED"
```

Return value

```
SETVARS_COMPLETED=1
```

If the return value is anything other than `SETVARS_COMPLETED=1` the test failed and `setvars.bat` did not complete properly.

Multiple Runs

Because many of the individual `env\vars.bat` scripts make significant changes to PATH, CPATH, and other environment variables, the top-level `setvars.bat` script will not allow multiple invocations of itself in the same session. This is done to ensure that your environment variables do not exceed the maximum provided environment space, especially the `%PATH%` environment variable. Exceeding the available environment space results in unpredictable behavior in your terminal session and should be avoided.

This behavior can be overridden by passing `setvars.bat` the `--force` flag. In this example, the user tries to run `setvars.bat` twice. The second instance is stopped because `setvars.bat` has already been run.

```
> <install-dir>\setvars.bat
initializing environment ...
(SNIP: lot of output)
oneAPI environment initialized
```

```
> <install-dir>\setvars.bat
.. code-block:: WARNING: setvars.bat has already been run. Skipping re-execution.
   To force a re-execution of setvars.bat, use the '--force' option.
   Using '--force' can result in excessive use of your environment variables.
```

In the third instance, the user runs `<install-dir>\setvars.bat --force` and the initialization is successful.

```
> <install-dir>\setvars.bat --force
initializing environment ...
(SNIP: lot of output)
oneAPI environment initialized
```

ONEAPI_ROOT Environment Variable

The `ONEAPI_ROOT` variable is set by the top-level `setvars.bat` script when that script is sourced. If there is already a `ONEAPI_ROOT` environment variable defined, `setvars.bat` temporarily overwrites it in the `cmd.exe` session in which you ran the `setvars.bat` script. This variable is primarily used by the `oneapi-cli` sample browser and the Microsoft Visual Studio and Visual Studio Code* sample browsers to help them locate oneAPI tools and components, especially for locating the `setvars.bat` script if the `SETVARS_CONFIG` feature has been enabled. For more information about the `SETVARS_CONFIG` feature, see Automate the setvars.bat Script with Microsoft Visual Studio*.

On Windows systems, the installer adds the `ONEAPI_ROOT` variable to the environment.

Use a Config file for setvars bat on Windows

The `setvars.bat` script sets environment variables for use with the oneAPI toolkits by executing each of the `<install-dir>\latest\env\vars.bat` scripts found in the respective oneAPI folders. Unless you configure your Windows system to run the `setvars.bat` script automatically, it must be executed every time a new terminal window is opened for command line development, or prior to launching Visual Studio Code, Sublime Text, or any other C/C++ editor you use. For more information, see Configure Your System.

The procedure below describes how to use a configuration file to manage environment variables.

Versions and Configurations

Some oneAPI tools support installation of multiple versions. For those tools that do support multiple versions, the directory is organized like this (assuming a default installation and using the compiler as an example):

```
\Program Files (x86)\Intel\oneAPI\compiler\
|-- 2021.1.1
|-- 2021.2.0
`-- latest -> 2021.2.0
```

For example:
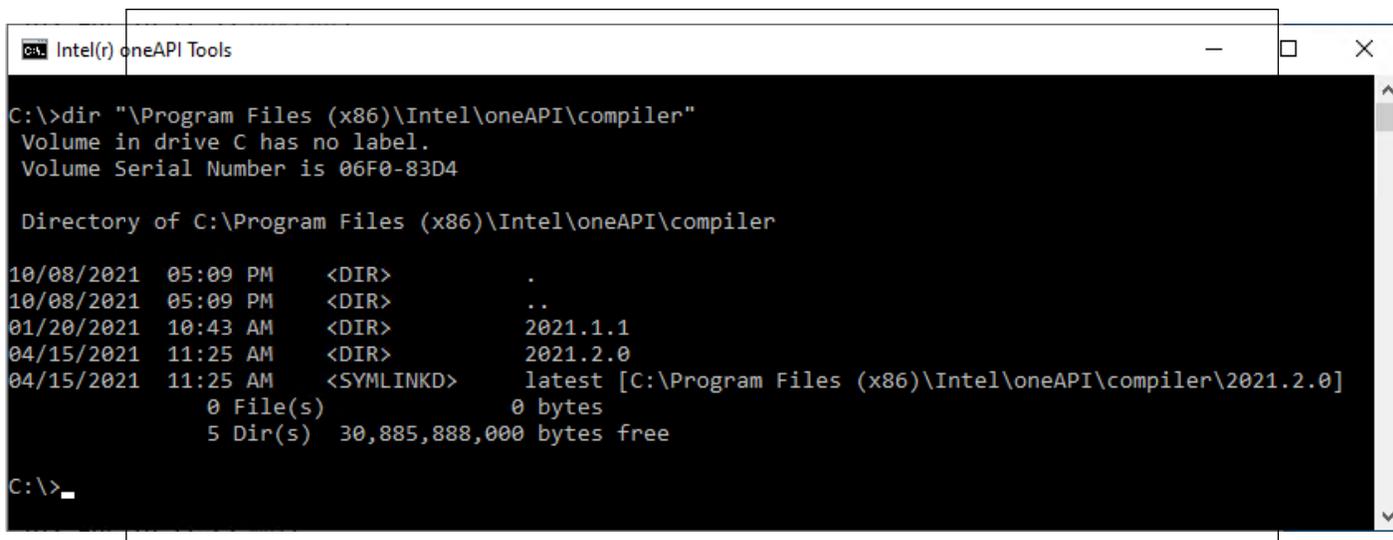
```
Intel(r) oneAPI Tools                                        —    □    ✕

C:\>dir "\Program Files (x86)\Intel\oneAPI\compiler"
 Volume in drive C has no label.
 Volume Serial Number is 06F0-83D4

 Directory of C:\Program Files (x86)\Intel\oneAPI\compiler

10/08/2021  05:09 PM    <DIR>          .
10/08/2021  05:09 PM    <DIR>          ..
01/20/2021  10:43 AM    <DIR>          2021.1.1
04/15/2021  11:25 AM    <DIR>          2021.2.0
04/15/2021  11:25 AM    <SYMLINKD>     latest [C:\Program Files (x86)\Intel\oneAPI\compiler\2021.2.0]
               0 File(s)              0 bytes
               5 Dir(s)  30,885,888,000 bytes free

C:\>_
```

For all tools, there is a symbolic link named `latest` that points to the latest installed version of that component; and the `vars.bat` script located in the `latest\env\` folder is what the `setvars.bat` executes by default.

If required, `setvars.bat` can be customized to point to a specific directory by using a configuration file.

–config Parameter

The top level `setvars.bat` script accepts a `--config` parameter that identifies your custom **config.txt** file.

```
<install-dir>\setvars.bat --config="path\to\your\config.txt"
```

The name of your configuration file can have any name you choose. You can create many config files to setup a variety of development or test environments. For example, you might want to test the latest version of a library with an older version of a compiler; use a setvars config file to manage such a setup.

Config File Sample

The examples below show a simple example of the config file:

Load Latest of Everything but…

```
mkl=1.1
dldt=exclude
```

Exclude Everything but…

```
default=exclude
mkl=1.0
ipp=latest
```

The configuration text file must follow these requirements:

- a newline delimited text file
- each line consists of a single "`key=value`" pair
- "`key`" names a component folder in the top-level set of oneAPI directories (the folders found in the `%ONEAPI_ROOT%` directory). If a "`key`" appears more than once in a config file, the last "`key`" wins and any prior keys with the same name are ignored.
- "`value`" names a version directory that is found at the top-level of the component directory. This includes any symbolic links (such as `latest`) that might be present at that level in the component directory.

    - OR "`value`" can be "`exclude`", which means the named key will NOT have its `vars.bat` script executed by the `setvars.bat` script.

The "`key=value`" pair "`default=exclude`" is a special case. When included, it will exclude executing ALL `env\vars.bat` scripts, except those that are listed in the config file. See the examples below.

Further Customization of Config Files

The config file can be used to exclude specific components, include specific component versions or only include specific component versions that are named after a "`default=exclude`" statement.

By default, `setvars.bat` will process the `latest` version of each `env\vars.bat` script.

The sample below shows two versions of Intel oneMKL installed: 2021.1.1 and 2021.2.0. The `latest` shortcut points to the 2021.2.0 folder because it is the latest version installed. By default, `setvars.bat` will execute the 2021.2.0 `vars.bat` script in the mkl folder because that is the folder that `latest` points to.

Two versions of Intel oneMKL and config files
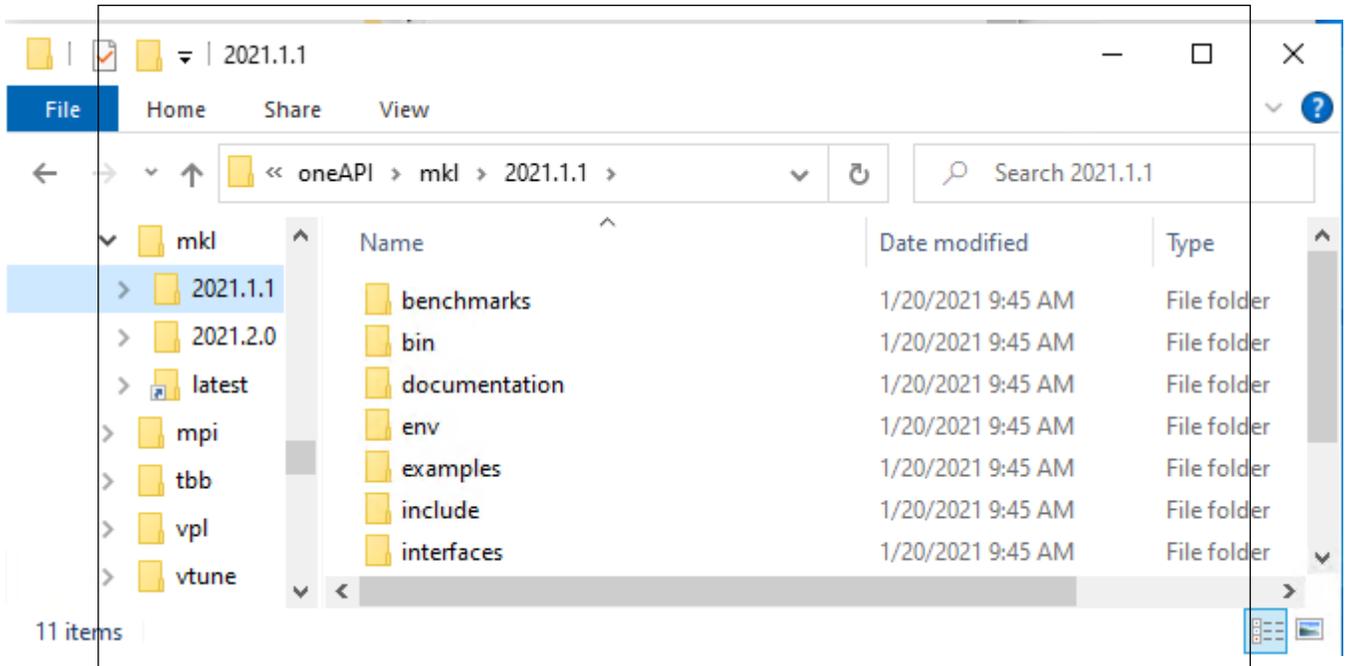
Specify a Specific Version

To direct `setvars.bat` to execute the `<install-dir>\mkl\2021.1.1\env\vars.bat` script, add `mkl=2021.1.1` to your config file.

This instructs `setvars.bat` to execute the `env\vars.bat` script located in the `2021.1.1` version folder inside the mkl directory. For other installed components, `setvars.bat` will execute the `env\vars.bat` script located in the latest version folder.

Exclude Specific Components

To exclude a component, use the following syntax:

```
<key>=exclude
```

For example, to exclude Intel IPP, but include the 2021.1.1 version of Intel oneMKL:

```
mkl=2021.1.1
ipp=exclude
```

In this example:

- `setvars.bat` WILL execute the Intel oneMKL 2021.1.1 `env\vars.bat` script
- `setvars.bat` WILL NOT execute Intel IPP `env\vars.bat` script files
- `setvars.bat` WILL execute the latest version of the remaining `env\vars.bat` script files

Include Specific Components

To execute a specific list of component `env\vars.bat` scripts, you must first exclude all `env\vars.bat` scripts. Then add back the list of components to be executed by `setvars.bat`. Use the following syntax to exclude all component `env\vars.bat` scripts from being executed:

```
default=exclude
```

For example, to have `setvars.bat` execute only the Intel oneMKL and Intel IPP component `env\vars.bat` scripts, use this config file:

```
default=exclude
mkl=2021.1.1
ipp=latest
```

In this example:

- `setvars.bat` WILL execute the Intel oneMKL 2021.1.1 `env\vars.bat` script
- `setvars.bat` WILL execute the latest version of the Intel IPP `env\vars.bat` script
- `setvars.bat` WILL NOT execute the `env\vars.bat` script for any other components

Automate the setvars bat Script with Microsoft Visual Studio*

> **NOTE** Support for Microsoft Visual Studio* 2017 is deprecated as of the Intel® oneAPI 2022.1 release, and will be removed in a future release.

The `setvars.bat` script sets up the environment variables needed to use the oneAPI toolkits. This script must be run every time a new terminal window is opened for command-line development. The setvars.bat script can also be run automatically when Microsoft Visual Studio is started. You can configure this feature to instruct the setvars.bat script to set up a specific set of oneAPI tools by using the SETVARS_CONFIG environment variable.

SETVARS_CONFIG Environment Variable States

The SETVARS_CONFIG environment variable enables automatic configuration of the oneAPI development environment when you start your instance of Microsoft Visual Studio. The variable has three conditions or states:

- Undefined (the SETVARS_CONFIG environment variable does not exist)
- Defined but empty (the value contains nothing or only whitespace)
- Defined and points to a `setvars.bat` configuration file

If SETVARS_CONFIG is undefined there will be no attempt to automatically run `setvars.bat` when Visual Studio is started. This is the default case, since the SETVARS_CONFIG variable is not defined by the oneAPI installer.

If SETVARS_CONFIG is defined and has no value (or contains only whitespace), the `setvars.bat` script will be automatically run when Visual Studio is started. In this case, the setvars.bat script initializes the environment for *all* oneAPI tools that are installed on your system. For more information about running the `setvars.bat` script, see Build and Run a Sample Project Using the Visual Studio* Command Line.

When SETVARS_CONFIG is defined with the absolute pathname to a `setvars` configuration file, the `setvars.bat` script will be automatically run when Visual Studio is started. In this case, the `setvars.bat` script initializes the environment for only those oneAPI tools that are defined in the `setvars` configuration file. For more information about how to create a setvars config file, see Using a Config File with setvars.bat.

A `setvars` configuration file can have any name and can be saved to any location on your hard disk, as long as that location and the file are accessible and readable by Visual Studio. (A plug-in that was added to Visual Studio when you installed the oneAPI tools on your Windows system performs the SETVARS_CONFIG actions; that is why Visual Studio must have access to the location and contents of the `setvars` configuration file.)

If you leave the `setvars` config file empty, the `setvars.bat` script will initialize your environment for *all* oneAPI tools that are installed on your system. This is equivalent to defining the SETVARS_CONFIG variable with an empty string. See Using a Config File with setvars.bat for details regarding what to put inside of your setvars config file.

Define the SETVARS_CONFIG Environment Variable

Since the `SETVARS_CONFIG` environment variable is not automatically defined during installation, you must add it to your environment before starting Visual Studio (per the rules above). You can define the `SETVARS_CONFIG` environment variable using the Windows SETX command or in the Windows GUI tool by typing "`rundll32.exe sysdm.cpl,EditEnvironmentVariables`" into the "`Win+R`" dialog (use "Win+R" to bring up the dialog).

Use the setvars Script with Linux* or MacOS*

Most of the component tool folders contain an environment script named `vars.sh` that configures the environment variables needed by that component to support oneAPI development work. For example, in a default installation, the Intel® Integrated Performance Primitives (Intel® IPP) vars script on Linux or macOS is located at: `/opt/intel/ipp/latest/env/vars.sh`. This pattern is shared by all oneAPI components that include an environment `vars` setup script.

These component tool `vars` scripts can be called directly or collectively. To call them collectively, a script named `setvars.sh` is provided in the oneAPI installation folder. For example, in a default installation on a Linux or macOS machine: `/opt/intel/setvars.sh`.

Sourcing the `setvars.sh` script without any arguments causes it to locate and source all `<component>/latest/env/vars.sh` scripts in the installation. Changes made to the environment by these scripts can be seen by running the `env` command after running the environment setup scripts.

> **NOTE** Changes to your environment made by sourcing the `setvars.sh` script (or the individual `vars.sh` scripts) are not permanent. Those changes only apply to the terminal session in which the `setvars.sh` environment script was sourced.

Command Line Arguments

The `setvars.sh` script supports several command-line arguments, which are displayed using the `--help` option. For example:

```
source /opt/intel/oneapi/setvars.sh --help
```

The `--config=file` argument and the ability to include arguments that will be passed to the `vars.sh` scripts that are called by the `setvars.sh` script can be used to customize the environment setup.

The `--config=file` argument provides the ability to limit environment initialization to a specific set of oneAPI components. It also provides a way to initialize the environment for specific component versions. For example, to limit environment setup to just the Intel® IPP library and the Intel® oneAPI Math Kernel Library (Intel® oneMKL), pass a config file that tells the `setvars.sh` script to only call the `vars.sh` environment scripts for those two oneAPI components. More details and examples are provided in Use a Config file for setvars.sh on Linux or macOS.

Any extra arguments passed on the `setvars.sh` command line that are not described in the `setvars.sh` help message will be passed to every called `vars.sh` script. That is, if the `setvars.sh` script does not recognize an argument, it assumes the argument is meant for use by one or more component scripts and passes those extra arguments to every component `vars.sh` script that it calls. The most common extra arguments are `ia32` and `intel64`, which are used by the Intel compilers and the IPP, MKL, and TBB libraries to specify the application target architecture.

Inspect the individual `vars.sh` scripts to determine which, if any, command line arguments they accept.

How to Run

```
source <install-dir>/setvars.sh
```

> **NOTE** If you are using a non-POSIX shell, such as csh, use the following command:
>
> ```
> $ bash -c 'source <install-dir>/setvars.sh ; exec csh'
> ```
>
> Alternatively, use the modulefiles scripts to set up your development environment. The modulefiles scripts work with all Linux shells.
>
> If you wish to fine tune the list of components and the version of those components, use a setvars config file to set up your development environment.

How to Verify

After sourcing the `setvars.sh` script, verify success by searching for the `SETVARS_COMPLETED` environment variables. If `setvars.sh` was successful, then the `SETVARS_COMPLETED` environment variable will have a value of 1:

```
env | grep SETVARS_COMPLETED
```

Return value

```
SETVARS_COMPLETED=1
```

If the return value is anything other than `SETVARS_COMPLETED=1`, then the test failed and `setvars.sh` did not complete properly.

Multiple Runs

Because many of the individual `env/vars.sh` scripts make significant changes to PATH, CPATH, and other environment variables, the top-level `setvars.sh` script will not allow multiple invocations of itself in the same session. This is done to ensure that your environment variables do not become too long due to redundant path references, especially the `$PATH` environment variable.

This behavior can be overridden by passing `setvars.sh` the `--force` flag. In this example, the user tries to run `setvars.sh` twice. The second instance is stopped because `setvars.sh` has already been run.

```
> source <install-dir>/setvars.sh
.. code-block:: initializing environment ...
(SNIP: lot of output)
.. code-block:: oneAPI environment initialized ::
```

```
> source <install-dir>/setvars.sh
 .. code-block:: WARNING: setvars.sh has already been run. Skipping re-execution.
    To force a re-execution of setvars.sh, use the '--force' option.
    Using '--force' can result in excessive use of your environment variables
```

In the third instance, the user runs `setvars.sh --force` and the initialization is successful.

```
> source <install-dir>/setvars.sh --force
 .. code-block:: initializing environment ...
(SNIP: lot of output)
 .. code-block:: oneAPI environment initialized ::
```

ONEAPI_ROOT Environment Variable

The `ONEAPI_ROOT` variable is set by the top-level `setvars.sh` script when that script is sourced. If there is already a `ONEAPI_ROOT` environment variable defined, `setvars.sh` temporarily overwrites it in the terminal session in which you sourced the `setvars.sh` script. This variable is primarily used by the `oneapi-cli` sample browser and the Eclipse* and Visual Studio Code* sample browsers to help them locate oneAPI tools and components, especially for in locating the `setvars.sh` script if the `SETVARS_CONFIG` feature has been enabled. For more information about the `SETVARS_CONFIG` feature, see Automate the setvars.sh Script with Eclipse*.

On Linux and macOS systems, the installer does not add the `ONEAPI_ROOT` variable to the environment. To add it to the default environment, define the variable in your local shell initialization file(s) or in the system's `/etc/environment` file.

Use a Config file for setvars sh on Linux or macOS

There are two methods for configuring your environment in Linux*:

* Use a setvars.sh configuration file, as described on this page
* Use modulefiles

The `setvars.sh` script sets environment variables for use with the oneAPI toolkits by sourcing each of the `<install-dir>/latest/env/vars.sh` scripts found in the respective oneAPI folders. Unless you configure your Linux system to source the `setvars.sh` script automatically, it must be sourced every time a new terminal window is opened for command line development, or prior to launching Eclipse* or any other C/C++ IDE or editor you use for C/C++ development. For more information, see Configure Your System.

The procedure below describes how to use a configuration file to manage environment variables.

Versions and Configurations

Some oneAPI tools support installation of multiple versions. For those tools that do support multiple versions, the directory is organized like this:

```
intel/oneapi/compiler/
|-- 2021.1.1
|-- 2021.2.0
`-- latest -> 2021.2.0
```

For example:

Multiple versions and environmental variables

```
$ ls -l intel/oneapi/compiler/
total 8
drwxr-xr-x 8 ubuntu ubuntu 4096 Nov  9  2020 2021.1.1/
drwxrwxr-x 8 ubuntu ubuntu 4096 Apr  9 10:06 2021.2.0/
lrwxrwxrwx 1 ubuntu ubuntu    8 Apr  9 10:06 latest -> 2021.2.0/
$
```

For all tools, there is a symlink named `latest` that points to the latest installed version of that component; and the `vars.sh` script located in the `latest/env/` folder is what the `setvars.sh` sources by default.

If required, `setvars.sh` can be customized to point to a specific directory by using a configuration file.

–config Parameter

The top level `setvars.sh` script accepts a `--config` parameter that identifies your custom **config.txt** file.

```
> source <install-dir>/setvars.sh --config="full/path/to/your/config.txt"
```

The name of your configuration file can have any name you choose. You can create many config files to setup a variety of development or test environments. For example, you might want to test the latest version of a library with an older version of a compiler; use a setvars config file to manage such a setup.

Config File Sample

The examples below show a simple example of the config file:

Load Latest of Everything but...

```
mkl=1.1
dldt=exclude
```

Exclude Everything but...

```
default=exclude
mkl=1.0
ipp=latest
```

The configuration text file must follow these requirements:

- a newline delimited text file
- each line consists of a single "`key=value`" pair
- "`key`" names a component folder in the top-level set of oneAPI directories (the folders found in the `$ONEAPI_ROOT` directory). If a "`key`" appears more than once in a config file, the last "`key`" wins and any prior keys with the same name are ignored.
- "`value`" names a version directory that is found at the top-level of the component directory. This includes any symlinks (such as `latest`) that might be present at that level in the component directory.
  - OR "`value`" can be "`exclude`", which means the named key will NOT have its `env/vars.sh` script sourced by the `setvars.sh` script.

The "`key=value`" pair "`default=exclude`" is a special case. When included, it will exclude sourcing ALL `env/vars.sh` scripts, except those that are listed in the config file. See the examples below.

Further Customization of Config Files

The config file can be used to exclude specific components, include specific component versions or only include specific component versions that are named after a "`default=exclude`" statement.

By default, `setvars.sh` will process the `latest` version of each `env/vars.sh` script.

The sample below shows two versions of Intel oneMKL installed: 2021.1.1 and 2021.2.0. The `latest` symlink points to the 2021.2.0 folder because it is the latest version. By default `setvars.sh` will source the 2021.2.0 `vars.sh` script in the `mkl` folder because that is the folder that `latest` points to.

Two versions of Intel oneMKL installed

```
$ /usr/bin/tree -dL 2 --charset=ascii intel/oneapi/mkl/
intel/oneapi/mkl/
|-- 2021.1.1
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
|-- 2021.2.0
|   |-- benchmarks
|   |-- bin
|   |-- documentation
|   |-- env
|   |-- examples
|   |-- include
|   |-- interfaces
|   |-- lib
|   |-- licensing
|   |-- modulefiles
|   `-- tools
`-- latest -> 2021.2.0
```

Specify a Specific Version

To direct `setvars.sh` to source the `<install-dir>/mkl/2021.1.1/env/vars.sh` script, add `mkl=2021.1.1` to your config file.

This instructs `setvars.sh` to source on the `env/vars.sh` script located in the `2021.1.1` version folder inside the `mkl` directory. For other installed components, `setvars.sh` will source the `env/vars.sh` script located in the latest version folder.

Exclude Specific Components

To exclude a component, use the following syntax:

```
<key>=exclude
```

For example, to exclude Intel IPP, but include the 2021.1.1 version of Intel oneMKL:

```
mkl=2021.1.1
ipp=exclude
```

In this example:

- `setvars.sh` WILL source the Intel oneMKL 2021.1.1 `env/vars.sh` script
- `setvars.sh` WILL NOT source any Intel IPP `env/vars.sh` script files
- `setvars.sh` WILL source the latest version of the remaining `env/vars.sh` script files

Include Specific Components

To source a specific list of component `env/vars.sh` scripts, you must first exclude all `env/vars.sh` scripts. Then add back the list of components to be sourced by `setvars.sh`. Use the following syntax to exclude all component `env/vars.sh` scripts from being sourced:

```
default=exclude
```

For example, to have `setvars.sh` source only the Intel oneMKL and Intel IPP component `env/vars.sh` scripts, use this config file:

```
default=exclude
mkl=2021.1.1
ipp=latest
```

In this example:

- `setvars.sh` WILL source the Intel oneMKL 2021.1.1 `env/vars.sh` script
- `setvars.sh` WILL source the latest version of the Intel IPP `env/vars.sh` script
- `setvars.sh` WILL NOT source the `env/vars.sh` script for any other components

Automate the setvars sh Script with Eclipse*

The `setvars.sh` script sets up the environment variables needed to use the oneAPI toolkits. This script must be run every time a new terminal window is opened for command-line development. The `setvars.sh` script can also be run automatically when Eclipse* is started. You can configure this feature to instruct the `setvars.sh` script to set up a specific set of oneAPI tools by using the `SETVARS_CONFIG` environment variable.

SETVARS_CONFIG Environment Variable States

The `SETVARS_CONFIG` environment variable enables automatic configuration of the oneAPI development environment when you start your instance of Eclipse IDE for C/C++ Developers. The variable has three conditions or states:

- Undefined (the `SETVARS_CONFIG` environment variable does not exist)
- Defined but empty (the value contains nothing or only whitespace)
- Defined and points to a `setvars.sh` configuration file

If `SETVARS_CONFIG` is undefined or if it exists but has no value (or contains only whitespace), the `setvars.sh` script will be automatically run when Eclipse is started. In this case, the `setvars.sh` script initializes the environment for *all* oneAPI tools that are installed on your system. For more information about running the `setvars.sh` script, see Build and Run a Sample Project Using Eclipse.

When `SETVARS_CONFIG` is defined with the absolute pathname to a `setvars` configuration file, the `setvars.sh` script will be automatically run when Eclipse is started. In this case, the `setvars.sh` script initializes the environment for only those oneAPI tools that are defined in the `setvars` configuration file. For more information about how to create a setvars config file, see Use a Config file for setvars.sh on Linux or macOS.

> **NOTE** The default `SETVARS_CONFIG` behavior in Eclipse is different than the behavior described for Visual Studio on Windows. When starting Eclipse, automatic execution of the `setvars.sh` script is always attempted. When starting Visual Studio automatic execution of the `setvars.bat` script it is only attempted if the `SETVARS_CONFIG` environment variable has been defined.

A `setvars` configuration file can have any name and can be saved to any location on your hard disk, as long as that location and the file are accessible and readable by Eclipse. (A plug-in that was added to Eclipse when you installed the oneAPI tools on your LInux system performs the `SETVARS_CONFIG` actions; that is why Eclipse must have access to the location and contents of the `setvars` configuration file.)

If you leave the `setvars` config file empty, the `setvars.sh` script will initialize your environment for *all* oneAPI tools that are installed on your system. This is equivalent to defining the `SETVARS_CONFIG` variable with an empty string. See Use a Config file for setvars.sh on Linux or macOS for details regarding what to put inside of your setvars config file.

Define the SETVARS_CONFIG Environment Variable

Since the `SETVARS_CONFIG` environment variable is not automatically defined during installation, you must add it to your environment before starting Eclipse (per the rules above). There are a variety of places to define the `SETVARS_CONFIG` environment variable:

- `/etc/environment`
- `/etc/profile`
- `~/.bashrc`
- and so on…

The list above shows common places to define environment variables on a Linux system. Ultimately, where you choose to define the SETVARS_CONFIG environment variable depends on your system and your needs.

Use Modulefiles with Linux*

There are two methods for configuring your environment in Linux*:

- Use modulefiles, as described on this page
- Use a setvars.sh configuration file

Most of the component tool folders contain one or more modulefile scripts that configure the environment variables needed by that component to support development work. Modulefiles are an alternative to using the `setvars.sh` script to set up the development environment. Because modulefiles do not support arguments, multiple modulefiles are available for oneAPI tools and libraries that support multiple configurations (such as a 32-bit configuration and a 64-bit configuration).

NOTE The modulefiles provided with the Intel oneAPI toolkits are compatible with the Tcl Environment Modules (Tmod) and Lua Environment Modules (Lmod). The following minimum versions are supported:

- Tmod 3.2.10 (compiler modulefile requires 4.1, see below)
- Tcl version 8.4
- Lmod version 8.2.10

Test which version is installed on your system using the following command:

```
module --version
```

Each modulefile automatically verifies the Tcl version on your system when it runs.

If your modulefile version is not supported, a workaround may be possible. See Using Environment Modules with Intel Development Tools for more details.

As of the oneAPI 2021.4 release you can use the `icc` modulefile to setup the `icc` and `ifort` compilers if you are using version 3.2.10 of the Tcl Environment Modules. A future oneAPI release will resolve the support for the `compiler` modulefile.

The oneAPI modulefile scripts are located in a modulefiles directory inside each component folder (similar to how the individual vars scripts are located). For example, in a default installation, the `ipp` modulefiles script(s) are in the `/opt/intel/ipp/latest/modulefiles/` directory.

Due to how oneAPI component folders are organized on the disk, it can be difficult to use the oneAPI modulefiles directly where they are installed. Therefore, a special `modulefiles-setup.sh` script is provided in the oneAPI installation folder to make it easier to work with the oneAPI modulefiles. In a default installation, that setup script is located here: `/opt/intel/oneapi/modulefiles-setup.sh`

The `modulefiles-setup.sh` script locates all modulefile scripts that are part of your oneAPI installation and organizes them into a single directory of versioned modulefiles scripts.

Each of these versioned modulefiles scripts is a symlink that points to the modulefiles located by the `modulefiles-setup.sh` script. Each component folder includes (at minimum) a "latest" version modulefile that will be selected, by default, when loading a modulefile without specifying a version label. If you use the `--ignore-latest` option when running the `modulefiles-setup.sh` script, the modulefile with the higest semver version will be loaded if no version is specified by the `module_load` command.

Creating the `modulefiles` Directory

Run the `modulefiles-setup.sh` script.

NOTE By default, the `modulefiles-setup.sh` script creates a folder named modulefiles in the oneAPI toolkit installation folder. If your oneAPI installation folder is not writeable, use the `--output-dir=<path-to-folder>` option to create the modulefiles folder in a writeable location. Run `modulefiles-setup.sh --help` for more information about this and other modulefiles-setup.sh script options.

Running the `modulefiles-setup.sh` script creates the `modulefiles` output folder, which is organized like the following example (the precise list of modulefiles depends on your installation). In this example there is one modulefile for configuring the Intel® Advisor environment and two modulefiles for configuring the compiler environment (the compiler modulefile configures the environment for all Intel compilers). If you follow the latest symlinks, they point to the highest version modulefile, per semver rules.

```
|-- advisor
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/advisor/2021.2.0/modulefiles/advisor
|    `-- latest -> /home/ubuntu/intel/oneapi/advisor/latest/modulefiles/advisor
|-- ccl
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/ccl/2021.1.1/modulefiles/ccl
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/ccl/2021.2.0/modulefiles/ccl
|    `-- latest -> /home/ubuntu/intel/oneapi/ccl/latest/modulefiles/ccl
|-- clck
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/clck/2021.1.1/modulefiles/clck
|    `-- latest -> /home/ubuntu/intel/oneapi/clck/latest/modulefiles/clck
|-- compiler
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler
|    `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler
|-- compiler-rt
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt
|    `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt
|-- compiler-rt32
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler-rt32
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler-rt32
|    `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler-rt32
|-- compiler32
|    |-- 2021.1.1 -> /home/ubuntu/intel/oneapi/compiler/2021.1.1/modulefiles/compiler32
|    |-- 2021.2.0 -> /home/ubuntu/intel/oneapi/compiler/2021.2.0/modulefiles/compiler32
|    `-- latest -> /home/ubuntu/intel/oneapi/compiler/latest/modulefiles/compiler32
```

Update your MODULEFILESPATH to include to the `modulefiles` output folder that was created by the `modulefiles-setup.sh` script or run the `moduleuse <folder_name>` command.

Installing the Tcl Modulefiles Environment onto Your System

The instructions below will help you quickly get started with the Environment Modules utility on Ubuntu*. For full details regarding installation and configuration of the `module` utility, see http://modules.sourceforge.net/.

```
$ sudo apt update
$ sudo apt install tcl
$ sudo apt install environment-modules
```

Confirm that the local copy of `tclsh` is new enough (see the beginning of this page for a list of supported versions):

```
$ echo 'puts [info patchlevel] ; exit 0' | tclsh
8.6.8
```

To test the `module` installation, initialize the `module` alias.

```
$ source /usr/share/modules/init/sh
$ module
```

> **NOTE** Initialization of the Modulefiles environment in POSIX-compatible shells should work with the `source` command shown above. Shell-specific init scripts are provided in the `/usr/share/modules/init/` folder. See that folder and the initialization section in `man module` for more details.

Source the `module` alias init script (`.../modules/init/sh`) in a global or local startup script to ensure the module command is always available. At this point, the system should be ready to use the module command as shown in the following section.

Getting Started with the `modulefiles-setup.sh` Script

The following example assumes you have:

- installed `tclsh` on to the Linux development system
- installed the Environment Modules utility (i.e., `module`) onto the system
- sourced the `.../modules/init/sh` (or equivalent) module init command
- installed the oneAPI toolkits required for your oneAPI development

```
$ cd <oneapi-root-folder>        # cd to the oneapi_root install directory
$ ./modulefiles-setup.sh         # run the modulefiles setup script
$ module use modulefiles         # use the modulefiles folder created above
$ module avail                   # will show tbb/X.Y, etc.
$ module load tbb                # loads tbb/X.Y module
$ module list                    # should list the tbb/X.Y module you just loaded
$ module unload tbb              # removes tbb/X.Y changes from the environment
$ module list                    # should no longer list the tbb/X.Y env var module
```

Before the unload step, use the `env` command to inspect the environment and look for the changes that were made by the modulefile you loaded. For example, if you loaded the `tbb` modulefile, the command will show you some of the env changes made by that `modulefile` (inspect the `modulefile` to see all of the changes it will make):

```
$ env | grep -i "intel"
```

> **NOTE** A `modulefile` is a script, but it does not need to have the 'x' (executable) permission set, because it is loaded and interpreted by the "module" interpreter that is installed and maintained by the end-user. Installation of the oneAPI toolkits do not include the `modulefile` interpreter. It must be installed separately. Likewise, `modulefiles` do not require that the 'w' permission be set, but they must be readable (ideally, the 'r' permission is set for all users).

Versioning

The oneAPI toolkit installer uses version folders to allow oneAPI tools and libraries to exist in a side-by-side layout. These versioned component folders are used by the `modulefiles-setup.sh` script to create the versioned `modulefiles`. The script organizes the symbolic links it creates in the `modulefiles` output folder as `<modulefile-name>/version`, so that each respective `modulefile` can be referenced by version when using the module command.

```
$ module avail
---------------- modulefiles -----------------
ipp/1.1  ipp/1.2  compiler/1.0  compiler32/1.0
```

Multiple modulefiles

A tool or library may provide multiple `modulefiles` within its `modulefiles` folder. Each becomes a loadable module. They will be assigned a version per the component folder from which they were extracted.

Understanding How the `modulefiles` are Written when using oneAPI

Symbolic links are used by the `modulefiles-setup.sh` script to gather all the available `modulefiles` into a single `modulefiles` folder. This means that the actual `modulefile` scripts are not moved or modified. As a consequence, the `${ModulesCurrentModulefile}` variable points to the symlink to each `modulefile`, not to the actual `modulefile` located in the respective installation folders. To determine the full path to the actual modulefiles, each modulefile starts with a statement like this:

```
[ file readlink ${ModulesCurrentModulefile} ]
```

to get a direct reference to the original `modulefile` in the product install directory. This is done because the actual install location can be customized and is, therefore, unknown at runtime and must be deduced. For that reason, the actual modulefile cannot be moved outside of the installed location, otherwise it will not be able to locate the absolute path to the library or application that it must configure.

For a better understanding, review the `modulefiles` included with the installation. Most include comments explaining how they resolve `symlink` references to a real file, as well as parsing the version number (and version directory). They also include checks to insure that the installed TCL is an appropriate version level.

Use of the `module load` Command by `modulefiles`

Several of the `modulefiles` use the `module load` command to ensure that any required dependent modules are also loaded. There is no attempt to specify the version of those dependent modulefiles. This means you have the option to load a specific version of a dependent module prior to loading the module that requires that dependent module. If you do not preload a dependent module, the latest available version is loaded.

This is by design because it gives you the flexibility to control the environment. For example, you may have installed an updated version of a library that you want to test against a previous version of the compiler. Perhaps the updated library has a bug fix and you are not interested in changing the version of any other libraries in your build. If the dependent modulefiles were hard-coded to require a specific dependent version of this library, you could not perform such a test.

> **NOTE** If a dependent `module load` cannot be satisfied, the currently loading module file will be terminated and no changes will be made to your environment.

Additional Resources

For more information about `modulefiles`, see:

- http://www.admin-magazine.com/HPC/Articles/Environment-Modules
- https://www.chpc.utah.edu/documentation/software/modules-advanced.php
- https://modules.readthedocs.io/en/latest/
- https://lmod.readthedocs.io/en/latest/

Use CMake with oneAPI Applications

The CMake packages provided with Intel oneAPI products allow a CMake project to make easy use of oneAPI libraries on Windows*, Linux*, or macOS*. Using the provided packages, the experience should be similar to how other system libraries integrate with a CMake project. There are dependency and other build variables provided to CMake project targets as desired.

The following components support CMake:

- Intel® oneAPI DPC++ Compiler - Linux, Windows
- Intel Integrated Performance Primitives (Intel IPP) and Intel Integrated Performance Primitives Cryptography (Intel IPP Cryptography) - Linux, Windows

- Intel MPI Library - Linux, Windows
- Intel oneAPI Collective Communications Library (oneCCL) - Linux, Windows
- Intel oneAPI Data Analytics Library (oneDAL) - Linux, Windows
- Intel oneAPI Deep Neural Network Library (oneDNN) - Linux, Windows
- Intel oneAPI DPC++ Library (oneDPL) - Linux, Windows
- Intel oneAPI Math Kernel Library (oneMKL) - Linux, Windows, macOS
- Intel oneAPI Threading Building Blocks (oneTBB) - Linux, Windows, macOS
- Intel oneAPI Video Processing Library (oneVPL) - Linux, Windows

Libraries that provide a CMake configuration can be identified by looking in the following locations:

- On Linux or macOS:

  - System: `/usr/local/lib/cmake` `
  - User: `~/lib/cmake` `
- On Windows: `HKEY_LOCAL_MACHINESoftwareKitwareCMakePackages\` `

To use the CMake packages, use the oneAPI libraries as you would other system libraries. For example, using `find_package(tbb)` ensures that your application's CMake package is using the oneTBB package.

Compile and Run oneAPI Programs

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 or infer pointer aliasing information across the function call.

Invoke the Compiler

The Intel® oneAPI DPC++/C++ Compiler provides multiple drivers to invoke the complier from the command line. The examples below show options for C++ and SYCL*. For a full list of driver options, see the Different Compilers and Drivers table.

For more information on the compiler, see Invoking the Compiler in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

To enable OpenMP* offloading for C++ applications, invoke the compiler with:

- `icpx -fiopenmp -fopenmp-targets=<arch>` (Linux)
- `icx /Qiopenmp /Qopenmp-targets:<arch>` (Windows).

To enable OpenMP offloading for SYCL applications, invoke the compiler with:

- `icpx –fsycl –fiopenmp –fopenmp-targets=<arch>` (Linux)
- `icx-cl –fsycl /Qiopenmp /Qopenmp-targets:<arch>` (Windows)

For more information about options, you can go to the option descriptions found in the Compiler Options section of the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

The compiler driver has different compatibilities on different OS hosts. On Linux, `icpx –fsycl` provides GCC*-style command line options. On Windows, `icx-cl` provides Microsoft Visual C++* compatibility with Microsoft Visual Studio*.

- It recognizes GCC-style command line options (starting with "-") and can be useful for projects that share a build system across multiple operating systems.
- It recognizes Windows command line options (starting with "/") and can be useful for Microsoft Visual Studio-based projects.

Standard Intel oneAPI DPC++/C++ Compiler Options

A full list of Intel oneAPI DPC++/C++ Compiler options are available from the *Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

- The Offload Compilation Options, OpenMP* Options, and Parallel Processing Options section includes options specific to SYCL* and OpenMP* offload.
- A full list of available options and a brief description of each is available in the Alphabetical List of Compiler Options.

Example Compilation

oneAPI applications can be directly programmed, API-based, which makes use of available oneAPI libraries, or a combination of directly programmed and API-based. API-based programming takes advantage of device offload using library functionality, which can save developers time when wriitng an application. In general it is easiest to start with API-based programming and use SYCL* or OpenMP* offload features where API-based programming is insufficient for your needs.

The following sections give examples of API-based code and direct programming using SYCL.

API-based Code

The following code shows usage of an API call (`a * x + y`) employing the Intel oneAPI Math Kernel Library function `oneapi::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.

```cpp
#include <vector> // std::vector()
#include <cstdlib> // std::rand()
#include <CL/sycl.hpp>
#include "oneapi/mkl/blas.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.\n";
    }


    // 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";
                std::cout << 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 {
        oneapi::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;
    }


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

```
    // print y_buffer
    auto y_accessor = y_buffer.template
      get_access<cl::sycl::access::mode::read>();
    std::cout << std::endl;
    std::cout << "y" << " = [ " << y_accessor[0] << " ]\n";
    std::cout << "    [ " << y_accessor[1*incy] << " ]\n";
    std::cout << "    [ " << "... ]\n";
    std::cout << std::endl;


    return 0;
}
```

To compile and build the application (saved as `axpy.cpp`):

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` scripts, see oneAPI Development Environment Setup.

2. Build the application using the following command:

On Linux:

```
icpx -fsycl -I${MKLROOT}/include -c axpy.cpp -o axpy.o
```

On Windows:

```
icpx -fsycl -I${MKLROOT}/include /EHsc -c axpy.cpp /Foaxpy.obj
```

3. Link the application using the following command:

On Linux:

```
icpx -fsycl axpy.o -fsycl-device-code-split=per_kernel \
"${MKLROOT}/lib/intel64"/libmkl_sycl.a -Wl,-export-dynamic -Wl,--start-group \
"${MKLROOT}/lib/intel64"/libmkl_intel_ilp64.a \
"${MKLROOT}/lib/intel64"/libmkl_sequential.a \
"${MKLROOT}/lib/intel64"/libmkl_core.a -Wl,--end-group -lsycl -lOpenCL \
-lpthread -lm -ldl -o axpy.out
```

On Windows:

```
icpx -fsycl axpy.obj -fsycl-device-code-split=per_kernel ^
"${MKLROOT}/lib/intel64"/libmkl_sycl.lib ^
"${MKLROOT}/lib/intel64"/libmkl_intel_ilp64.lib ^
"${MKLROOT}/lib/intel64"/libmkl_sequential.lib ^
"${MKLROOT}/lib/intel64"/libmkl_core.lib ^
sycl.lib OpenCL.lib -o axpy.exe
```

4. Run the application using the following command:

On Linux:

```
./axpy.out
```

On Windows:

```
axpy.exe
```

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.

```
icpx -fsycl vector_add.cpp
```

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

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

Compilation Flow Overview

When you create a program with offload, the compiler must generate code for both the host and the device. oneAPI tries to hide this complexity from the developer. The developer simply compiles a SYCL* application using the DPC++ compiler with `icpx –fsycl`, and the same compile command generates both host and device code.

For device code, two options are available: Just-in-Time (JIT) compilation and Ahead-of-Time (AOT) compilation, with JIT being the default. This section describes how host code is compiled, and the two options for generating device code. Additional details are available in Chapter 13 of the Data Parallel C++ book.

Traditional Compilation Flow (Host-only Application)

The traditional compilation flow is a standard compilation like the one used for C, C++, or other languages, used when there is no offload to a device.

The traditional compilation phases are shown in the following diagram:

Traditional compilation phases
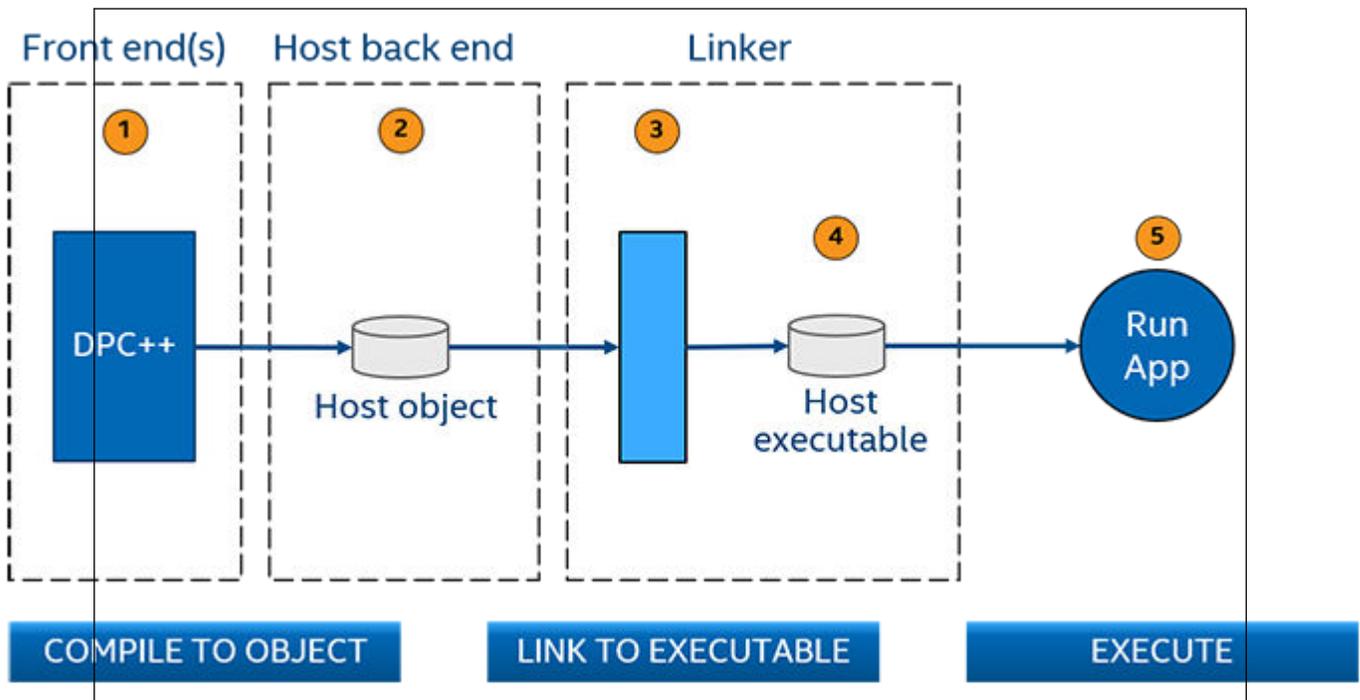
1. The front end translates the source into an intermediate representation and then passes that representation to the back end.
2. The back end translates the intermediate representation to object code and emits an object file (`host.obj` on Windows*, `host.o` on Linux*).

3. One or more object files are passed to the linker.
4. The linker creates an executable.
5. The application runs.

Compilation Flow for SYCL Offload Code

The compilation flow for SYCL offload code adds steps for device code to the traditional compilation flow, with JIT and AOT options for device code. In this flow, the developer compiles a SYCL application with `icpx –fsycl`, and the output is an executable containing both host and device code.

The basic compilation phases for SYCL offload code are shown in the following diagram:

Basic compilation phases for SYCL offload code
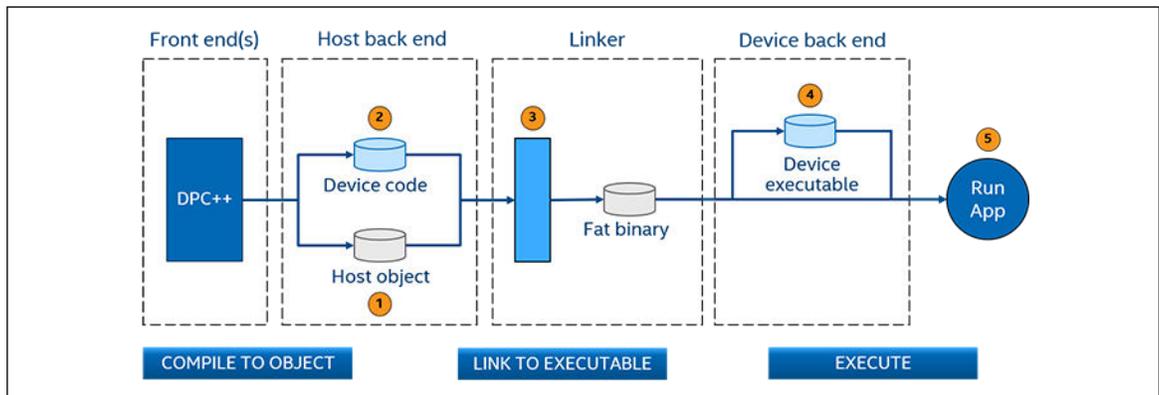
1. The host code is translated to object code by the back end.
2. The device code is translated to either a SPIR-V* or device binary.
3. The linker combines the host object code and the device code (SPIR-V or device binary) into an executable containing the host binary with the device code embedded in it. This process is known as a fat binary.
4. At runtime, the operating system starts the host application. If it has offload, the runtime loads the device code (converting the SPIR-V to device binary if needed).
5. The application runs on the host and a specified device.

JIT Compilation Flow

In the JIT compilation flow, the code for the device is translated to SPIR-V intermediate code by the back-end, embedded in the fat binary as SPRI-V, and translated from SPIR-V to device code by the runtime. When the application is run, the runtime determines the available devices and generates the code specific to that device. This allows for more flexibility in where the application runs and how it performs than the AOT flow, which must specify a device at compile time. However, performance may be worse because compilation occurs when the application runs. Larger applications with significant amounts of device code may notice performance impacts.

Tip The JIT compilation flow is useful when you do not know what the target device will be.

NOTE JIT compilation is not supported for FPGA devices.

The compilation phases are shown in the following diagram:

JIT compilation phases
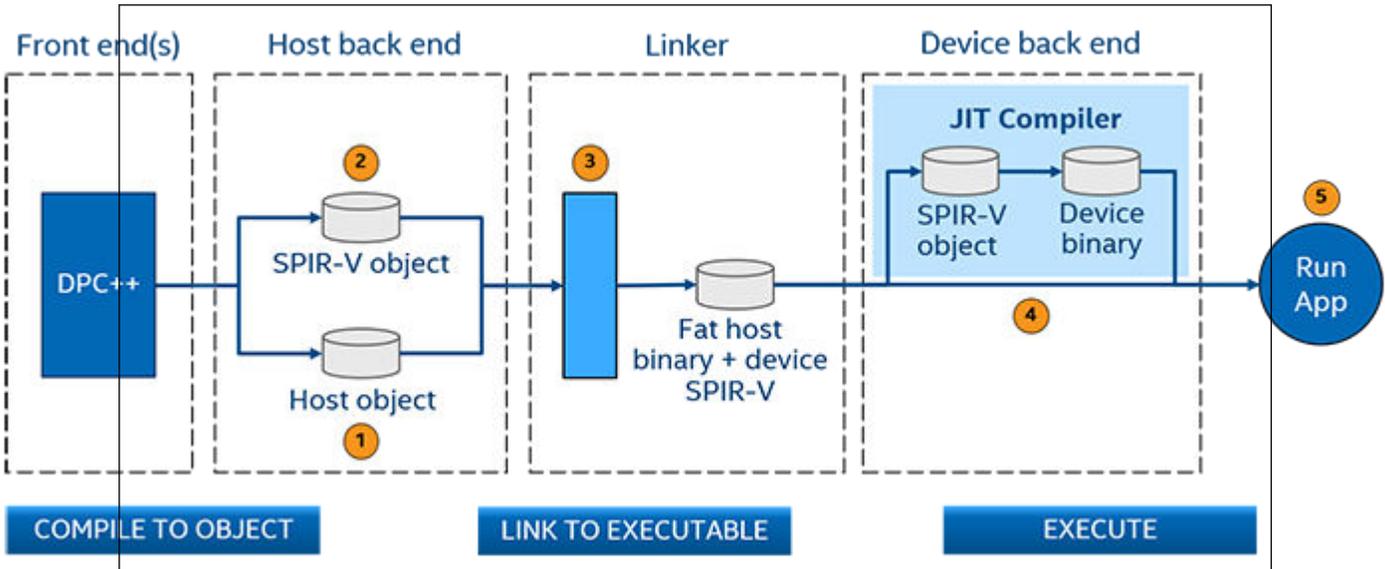
1. The host code is translated to object code by the back end.
2. The device code is translated to SPIR-V.
3. The linker combines the host object code and the device SPIR-V into a fat binary containing host executable code with SPIR-V device code embedded in it.
4. At runtime:

   a. The device runtime on the host translates the SPIR-V for the device into device binary code.
   b. The device code is loaded onto the device.
5. The application runs on the host and device available at runtime.

AOT Compilation Flow

In the AOT compilation flow, the code for the device is translated to SPIR-V and then device code in the host back-end and the resulting device code is embedded in the generated fat binary. The AOT flow provides less flexibility than the JIT flow because the target device must be specified at compilation time. However, executable start-up time is faster than the JIT flow.

> **Tip**
>
> - The AOT compilation flow is good when you know exactly which device you are targeting.
> - The AOT flow is recommended when debugging your application as it speeds up the debugging cycle.

The compilation phases are shown in the following diagram:

AOT compilation phases
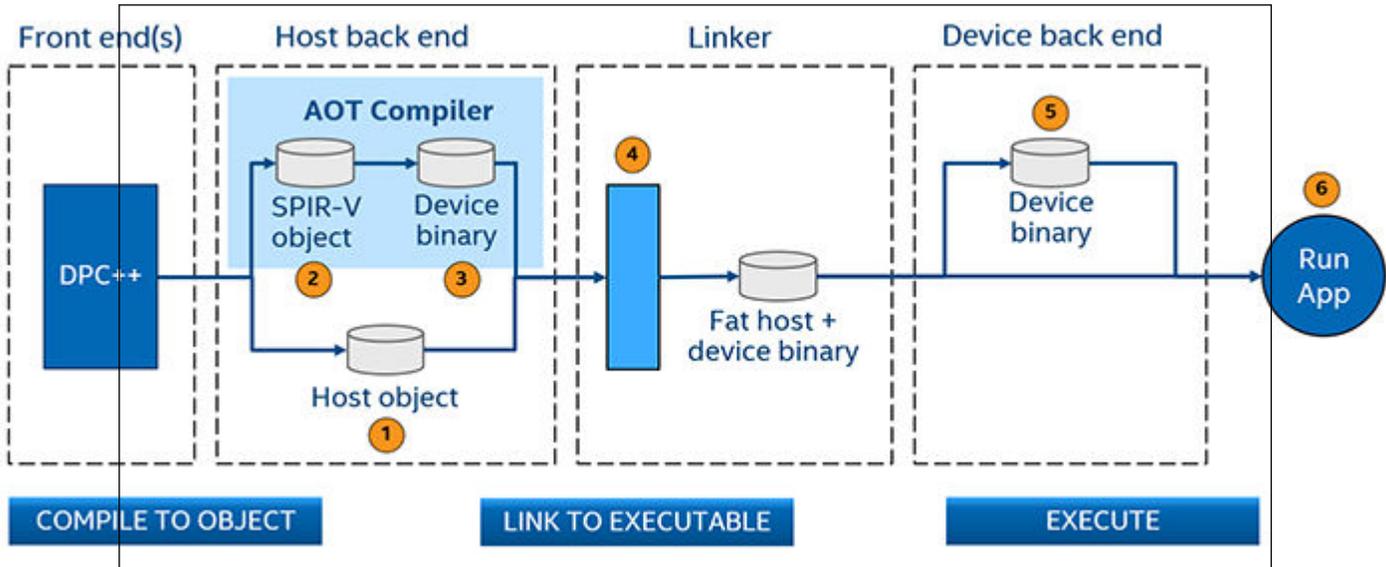
1. The host code is translated to object code by the back end.
2. The device code is translated to SPIR-V.
3. The SPIR-V for the device is translated to a device code object using the device specified by the user on the command line.
4. The linker combines the host object code and the device object code into a fat binary containing host executable code with device executable code embedded in it.
5. At runtime, the device executable code is loaded onto the device.
6. The application runs on a host and specified device.

Fat Binary

A fat binary is generated from the JIT and AOT compilation flows. It is a host binary that includes embedded device code. The contents of the device code vary based on the compilation flow.

FAT binary
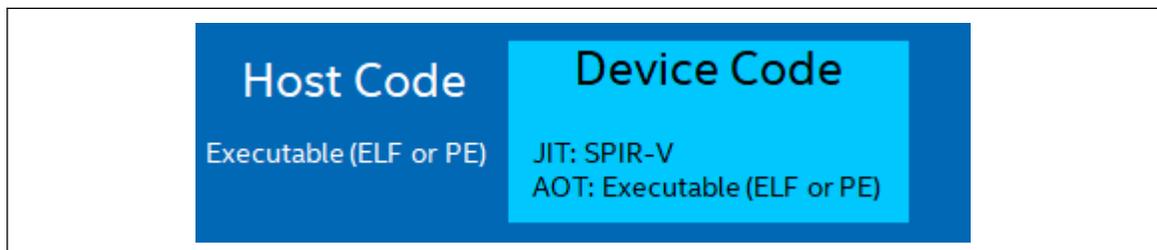
- The host code is an executable in either the ELF (Linux) or PE (Windows) format.
- The device code is a SPIR-V for the JIT flow or an executable for the AOT flow. Executables are in one of the following formats:
  - CPU: ELF (Linux), PE (Windows)
  - GPU: ELF (Windows, Linux)
  - FPGA: ELF (Linux), PE (Windows)

CPU Flow

The CPU is typically called the brain of the computer. The CPU consists of complex circuitry/algorithms that include branch predictors, memory virtualization and instruction scheduling, etc. Given this complexity, it is designed to handle a wide-range of tasks.

The SYCL* and OpenMP* offload programming model enables implementation of an application on heterogenous CPU and GPU systems. The term "devices" in SYCL and OpenMP offload can refer to both CPUs and GPUs.

Modern CPUs have many cores with hyper-threads and high SIMD width, which can be used for parallel computation. If your workloads have regions that are compute intensive and can be run in parallel, it is a good idea to offload those regions to a CPU than to a coprocessor, such as a GPU or FPGA. Also, because data does not need to be offloaded through PCIe (unlike for coprocessors or GPU), latency is reduced with minimal data transfer overhead.

There are two options for running an application on a CPU: the traditional CPU flow that runs directly on the CPU or a CPU offload flow that runs on a CPU device. You can use CPU offload with either SYCL or OpenMP offload applications. Both OpenMP offload and SYCL offload applications use the OpenCL™ runtime and Intel® oneAPI Threading Building Blocks (Intel® oneTBB) to run on a CPU as a device.

> **Tip** Unsure whether your workload fits best on CPU, GPU, or FPGA? Compare the benefits of CPUs, GPUs, and FPGAs for different oneAPI compute workloads.

Traditional CPU Flow

The traditional CPU workflow runs on the CPU without a runtime. The compilation flow is a standard compilation used when there is no offload to a device, like the one used for C, C++, or other languages.

Traditional workloads are compiled and run on host using the Traditional Compilation Flow (Host-only Application) process described in Compilation Flow Overview.

Example compilation command:

```
icpx  -g -o matrix_mul_omp src/matrix_mul_omp.cpp
```

CPU Offload Flow

By default, if you are offloading to a CPU device, it goes through an OpenCL™ runtime, which also uses Intel oneAPI Threading Building Blocks for parallelism.

When offloading to a CPU, workgroups map to different logical cores and these workgroups can execute in parallel. Each work-item in the workgroup can map to a CPU SIMD lane. Work-items (sub-groups) execute together in a SIMD fashion.

CPU workgroups
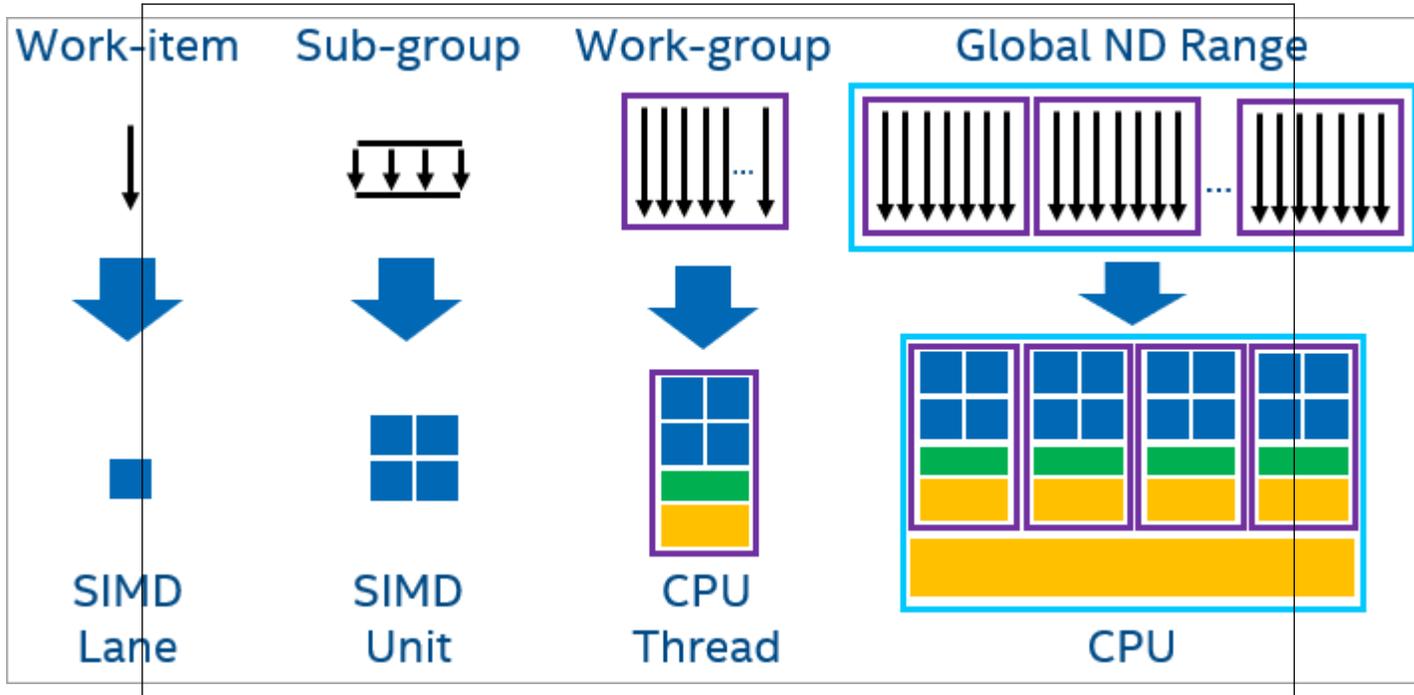
To learn more about CPU execution, see Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads.

Set Up for CPU Offload

1. Make sure you have followed all steps in the oneAPI Development Environment Setup section, including running the `setvars` script.
2. Check if you have the required OpenCL runtime associated with the CPU using the `sycl-ls` command. For example:

```
$sycl-ls
CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

3. Use one of the following code samples to verify that your code is running on the CPU. The code sample adds scalar to large vectors of integers and verifies the results.

SYCL*

To run on a CPU, SYCL provides built-in device selectors for convenience. They use `device_selector` as a base class. `cpu_selector` selects a CPU device.

Alternatively, you could also use the following environment variable when using `default_selector` to select a device according to implementation-defined heuristics.

```
export SYCL_DEVICE_FILTER=cpu
```

SYCL code sample:

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

```
using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
constexpr int value = 100000;
try{
        cpu_selector d_selector;
        queue q(d_selector);
        int *sequential = malloc_shared<int>(array_size, q);
        int *parallel = malloc_shared<int>(array_size, q);
        //Sequential iota
        for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;


        //Parallel iota in SYCL
        auto e = q.parallel_for(range{array_size}, [=](auto i) { parallel[i] = value + i; });
        e.wait();
   // Verify two results are equal.
    for (size_t i = 0; i < array_size; i++) {
      if (parallel[i] != sequential[i]) {
        cout << "Failed on device.\n";
        return -1;
      }
    }
    free(sequential, q);
    free(parallel, q);
}catch (std::exception const &e) {
    cout << "An exception is caught while computing on device.\n";
    terminate();
}
 cout << "Successfully completed on device.\n";
  return 0;
}
```

To compile the code sample, use:

```
dpcpp simple-iota-dp.cpp -o simple-iota.
```

Additional commands are available from Example CPU Commands.

Results after compilation:

```
./simple-iota
Running on device: Intel® Core™ i7-8700 CPU @ 3.20GHz
Successfully completed on device.
```

OpenMP*

OpenMP code sample:

```
#include<iostream>
#include<omp.h>
#define N 1024
int main(){
float *a = (float *)malloc(sizeof(float)*N);
for(int i = 0; i < N; i++)
a[i] = i;
#pragma omp target teams distribute parallel for simd map(tofrom: a[:N])
for(int i = 0; i < 1024; i++)
```

```
a[i]++;
std::cout<<a[100]<<"\n";
return 0;
}
```

Use the following environment variable to compile for running on a CPU:

```
export LIBOMPTARGET_DEVICETYPE=cpu
```

To compile the code sample, use:

```
icpx simple-ompoffload.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-ompoffload
```

Results after compilation:

```
./simple-ompoffload
Successfully completed on device
```

Offload Code to CPU

When offloading your application, it is important to identify the bottlenecks and which code will benefit from offloading. If you have a code that is compute intensive or a highly data parallel kernel, offloading your code would be something to look into.

To find opportunities to offload your code, use the Intel Advisor for Offload Modeling.

Debug Offloaded Code

The following list has some basic debugging tips for offloaded code.

- Check host target to verify the correctness of your code.
- Use `printf` to debug your application. Both SYCL and OpenMP offload support `printf` in kernel code.
- Use environment variables to control verbose log information.

  - For SYCL, the following debug environment variables are recommended. A full list of environment variables is available from GitHub.

  SYCL Recommended Debug Environment Variables

Name	Value	Description
SYCL_DEVICE_FILTER	backend:device_type:device_num	GitHub description
SYCL_PI_TRACE	1|2|-1	1: print out the basic trace log of the SYCL/DPC++ runtime plugin
		2: print out all API traces of SYCL/DPC++ runtime plugin
		-1: all of "2" including more debug messages

- For OpenMP, the following debug environment variables are recommended. A full list is available from the LLVM/OpenMP documentation.

  OpenMP Recommended Debug Environment Variables

Name	Value	Description
LIBOMPTARGET_DEVICETYPE	cpu|gpu|host	Select

Name	Value	Description
LIBOMPTARGET_DEBUG	1	Print out verbose debug information
LIBOMPTARGET_INFO	Values available in LLVM/ OpenMP documentation	Allows the user to request different types of runtime information from libomptarget

- Use Ahead of Time (AOT) to move Just-in-Time (JIT) compilations to AOT compilation issues. For more information, see Ahead-of-Time Compilation for CPU Architectures.

See Debugging the SYCL and OpenMP Offload Process for more information on debug techniques and debugging tools available with oneAPI.

Optimize CPU Code

There are many factors that can affect the performance of CPU offload code. The number of work-items, workgroups, and amount of work done depends on the number of cores in your CPU.

- If the amount of work being done by the core is not compute-intensive, then this could hurt performance. This is because of the scheduling overhead and thread context switching.
- On a CPU, there is no need for data transfer through PCIe, resulting in lower latency because the offload region does not have to wait long for the data.
- Based on the nature of your application, thread affinity could affect the performance on CPU. For details, see Control Binary Execution on Multiple Cores.
- Offloaded code uses JIT compilation by default. Use AOT compilation (offline compilation) instead. With offline compilation, you could target your code to specific CPU architecture. Refer to Optimization Flags for CPU Architectures for details.

Additional recommendations are available from Optimize Offload Performance.

Example CPU Commands

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

> **NOTE** Linking with a dynamic library is not supported.

Produce a fat object with device code:

```
icpx -fsycl -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:

```
icpx -fsycl -c a.cpp
```

Link the application with the static library:

```
icpx -fsycl -foffload-static-lib=libstlib.a a.o -o a.exe
```

Ahead-of-Time Compilation for CPU Architectures

In ahead-of-time (AOT) compilation mode, optimization flags can be used to produce code aimed to run better on a specific CPU architecture.

```
icpx -fsycl -fsycl-targets=spir64_x86_64  -Xs "-device <CPU optimization flags>" a.cpp b.cpp -o
app.out
```

Supported CPU optimization flags are:

```
-march=<instruction_set_arch> Set target instruction set architecture:
'sse42' for Intel® Streaming SIMD Extensions 4.2
'avx2' for Intel® Advanced Vector Extensions 2
'avx512' for Intel® Advanced Vector Extensions 512
```

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

Control Binary Execution on Multiple CPU Cores

Environment Variables

The following environment variables control the placement of SYCL* or OpenMP* threads on multiple CPU cores during program execution. Use these variables if you are using the OpenCL™ runtime CPU device to offload to a CPU.

SYCL* or OpenMP* evnironmental variables

Environment Variable	Description
DPCPP_CPU_CU_AFFINITY	Set thread affinity to CPU. The value and meaning is the following:
	- close - threads are pinned to CPU cores successively through available cores. - spread - threads are sread to available cores. - master - threads are put in the same cores as master. If DPCPP_CPU_CU_AFFINITY is set, master thread is pinned as well, otherwise master thread is not pinned.
	This environment variable is similar to the OMP_PROC_BIND variable used by OpenMP.
	Default: Not set
DPCPP_CPU_SCHEDULE	Specify the algorithm for scheduling work-groups by the scheduler. Currently, the SYCL runtime uses Intel® oneAPI Threading Building Blocks (Intel® oneTBB) for scheduling. The value selects the petitioner used by the Intel oneTBB scheduler. The value and meaning is the following:
	- dynamic - Intel oneTBB auto_partitioner. It performs sufficient splitting to balance load. - affinity - Intel oneTBB affinity_partitioner. It improves auto_partitioner's cache affinity by its choice of mapping subranges to worker threads compared to

Environment Variable	Description
	• static - Intel oneTBB static_partitioner. It distributes range iterations among worker threads as uniformly as possible. Intel oneTBB partitioner relies grain-size to control chunking. Grain-size is 1 by default, indicating every work-group can be executed independently.
	Default: Dynamic
DPCPP_CPU_NUM_CUS	Set the numbers threads used for kernel execution.
	To avoid over subscription, maximum value of DPCPP_CPU_NUM_CUS should be the number of hardware threads. If DPCPP_CPU_NUM_CUS is 1, all the workgroups are executed sequentially by a single thread and this is useful for debugging.
	This environment variable is similar to OMP_NUM_THREADS variable used by OpenMP.
	Default: Not set. Determined by Intel oneTBB.
DPCPP_CPU_PLACES	Specify the places that affinities are set. The value is { sockets | numa_domains | cores | threads }.
	This environment variable is similar to the OMP_PLACES variable used by OpenMP.
	If value is numa_domains, Intel oneTBB NUMA API will be used. This is analogous to OMP_PLACES=numa_domains in the OpenMP 5.1 Specification. Intel oneTBB task arena is bound to numa node and SYCL nd range is uniformly distributed to task arenas.
	DPCPP_CPU_PLACES is suggested to be used together with DPCPP_CPU_CU_AFFINITY.
	Default: cores

See the Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference for more information about all supported environment variables.

Example 1: Hyper-threading Enabled

Assume a machine with 2 sockets, 4 physical cores per socket, and each physical core has 2 hyper threads.

- `S<num>` denotes the socket number that has 8 cores specified in a list
- `T<num>` denotes the Intel® oneAPI Threading Building Blocks (Intel® oneTBB) thread number
- "-" means unused core

```
DPCPP_CPU_NUM_CUS=16
   export DPCPP_CPU_PLACES=sockets
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0 T1 T2 T3 T4 T5 T6 T7]      S1:[T8 T9 T10 T11 T12 T13
T14 T15]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0 T2 T4 T6 T8 T10 T12 T14]   S1:[T1 T3 T5 T7 T9 T11
T13 T15]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3 T4 T5 T6 T7]      S1:[T8 T9 T10 T11 T12 T13
T14 T15]
```

```
   export DPCPP_CPU_PLACES=cores
   DPCPP_CPU_CU_AFFINITY=close :   S0:[T0 T8 T1 T9 T2 T10 T3 T11]    S1:[T4 T12 T5 T13 T6 T14
T7 T15]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0 T8 T2 T10 T4 T12 T6 T14]   S1:[T1 T9 T3 T11 T5 T13 T7
T15]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3 T4 T5 T6 T7]      S1:[T8 T9 T10 T11 T12 T13
T14 T15]


   export DPCPP_CPU_PLACES=threads
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0 T1 T2 T3 T4 T5 T6 T7]      S1:[T8 T9 T10 T11 T12 T13
T14 T15]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0 T2 T4 T6 T8 T10 T12 T14]   S1:[T1 T3 T5 T7 T9 T11 T13
T15]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3 T4 T5 T6 T7]      S1:[T8 T9 T10 T11 T12 T13
T14 T15]


export DPCPP_CPU_NUM_CUS=8
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close close:   S0:[T0 - T1 - T2 - T3 -]    S1:[T4 - T5 - T6 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close spread:  S0:[T0 - T2 - T4 - T6 -]    S1:[T1 - T3 - T5 - T7 -]
   DPCPP_CPU_CU_AFFINITY=close master:  S0:[T0 T1 T2 T3 T4 T5 T6 T7] S1:[]
```

Example 2: Hyper-threading Disabled

Assume a machine with 2 sockets, 4 physical cores per socket, and each physical core has 2 hyper threads.

- `S<num>` denotes the socket number that has 8 cores specified in a list
- `T<num>` denotes the Intel oneTBB thread number
- "-" means unused core

```
export DPCPP_CPU_NUM_CUS=8
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0 T1 T2 T3]    S1:[T4 T5 T6 T7]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0 T2 T4 T6]    S1:[T1 T3 T5 T7]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3]    S1:[T4 T5 T6 T7]


export DPCPP_CPU_NUM_CUS=4
   DPCPP_CPU_PLACES=sockets, cores and threads have the same bindings:
   DPCPP_CPU_CU_AFFINITY=close:    S0:[T0  - T1  - ]    S1:[T2  - T3  - ]
   DPCPP_CPU_CU_AFFINITY=spread:   S0:[T0  - T2  - ]    S1:[T1  - T3  - ]
   DPCPP_CPU_CU_AFFINITY=master:   S0:[T0 T1 T2 T3]     S1:[ -   -   -   - ]
```

GPU Flow

GPUs are special-purpose compute devices that can be used to offload a compute intensive portion of your application. GPUs usually consists of many smaller cores and are therefore known for massive throughput. There are some tasks better suited to a CPU and others that may be better suited to a GPU.

Tip Unsure whether your workload fits best on CPU, GPU, or FPGA? Compare the benefits of CPUs, GPUs, and FPGAs for different oneAPI compute workloads.

GPU Offload Flow

Offloading a program to a GPU defaults to the level zero runtime. There is also an option to switch to the OpenCL™ runtime. In SYCL* and OpenMP* offload, each work item is mapped to a SIMD lane. A subgroup maps to SIMD width formed from work items that execute in parallel and subgroups are mapped to GPU EU thread. Work-groups, which include work-items that can synchronize and share local data, are assigned for execution on compute units (that is, streaming multiprocessors or Xe core, also known as sub-slices). Finally, the entire global NDRange of work-items maps to the entire GPU.

PRG Interface GPU workgroups
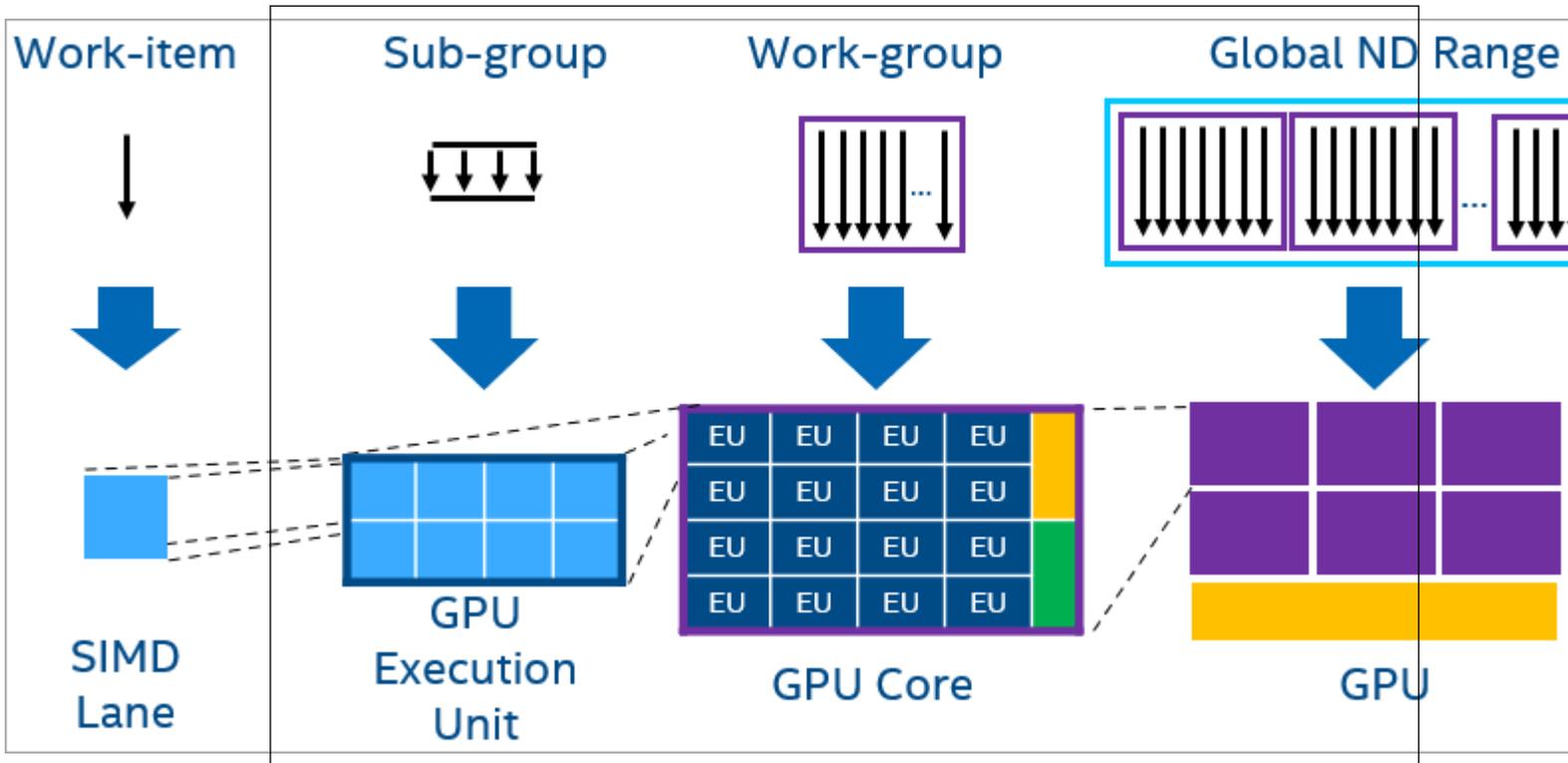
To learn more about GPU execution, see Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads.

Set Up for GPU Offload

1. Make sure you have followed all steps in the oneAPI Development Environment Setup section, including running the `setvars` script.
2. Configure your GPU system by installing drivers and add the user to the video group. See the Get Started Guide for instructions:

    *   Get Started with Intel oneAPI Base Toolkit for Linux* | Windows* | MacOS*
    *   Get Started with Intel oneAPI HPC Toolkit for Linux* | Windows* | MacOS*
    *   Get Started with Intel oneAPI IoT Toolkit for Linux* | Windows*
3. Check if you have a supported GPU and the necessary drivers installed using the `sycl-ls` command. In the following example, if you had the OpenCL and Level Zero driver installed you would see two entries for each runtime associated with the GPU:

```
CPU : OpenCL 2.1 (Build 0)[ 2020.11.12.0.14_160000 ]
GPU : OpenCL 3.0 NEO [ 21.33.20678 ]
GPU : 1.1[ 1.2.20939 ]
```

4. Use one of the following code samples to verify that your code is running on the GPU. The code sample adds scalar to large vectors of integers and verifies the results.

SYCL

To run on a GPU, SYCL provides built-in device selectors using `device_selector` as a base class. `gpu_selector` selects a GPU device. You can also create your own custom selector. For more information, see the Choosing Devices section in Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL (book).

SYCL code sample:

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


using namespace sycl;
using namespace std;
constexpr size_t array_size = 10000;
int main(){
constexpr int value = 100000;
try{
        //
// The default device selector will select the most performant device.
        default_selector d_selector;
        queue q(d_selector);


    //Allocating shared memory using USM.
        int *sequential = malloc_shared<int>(array_size, q);
        int *parallel = malloc_shared<int>(array_size, q);
        //Sequential iota
        for (size_t i = 0; i < array_size; i++) sequential[i] = value + i;


        //Parallel iota in SYCL
        auto e = q.parallel_for(range{array_size}, [=](auto i) { parallel[i] = value + i; });
        e.wait();
   // Verify two results are equal.
    for (size_t i = 0; i < array_size; i++) {
      if (parallel[i] != sequential[i]) {
        cout << "Failed on device.\n";
        return -1;
      }
    }
    free(sequential, q);
    free(parallel, q);
}catch (std::exception const &e) {
    cout << "An exception is caught while computing on device.\n";
    terminate();
}
 cout << "Successfully completed on device.\n";
  return 0;
}
```

To compile the code sample, use:

```
icpx -fsycl simple-iota-dp.cpp -o simple-iota
```

Results after compilation:

```
./simple-iota
Running on device: Intel® UHD Graphics 630 [0x3e92]
Successfully completed on device.
```

OpenMP*

OpenMP code sample:

```
#include <stdlib.h>
#include <omp.h>
#include <iostream>
constexpr size_t array_size = 10000;
#pragma omp requires unified_shared_memory
int main(){
constexpr int value = 100000;
// Returns the default target device.
int deviceId = (omp_get_num_devices() > 0) ? omp_get_default_device() : omp_get_initial_device();
int *sequential = (int *)omp_target_alloc_host(array_size, deviceId);
int *parallel = (int *)omp_target_alloc(array_size, deviceId);


        for (size_t i = 0; i < array_size; i++)
                sequential[i] = value + i;


        #pragma omp target parallel for
        for (size_t i = 0; i < array_size; i++)
                parallel[i] = value + i;


        for (size_t i = 0; i < array_size; i++) {
         if (parallel[i] != sequential[i]) {
           std::cout << "Failed on device.\n";
           return -1;
         }
        }


        omp_target_free(sequential, deviceId);
        omp_target_free(parallel, deviceId);


        std::cout << "Successfully completed on device.\n";
         return 0;


}
```

To compile the code sample, use:

```
icpx -fsyclsimple-iota-omp.cpp -fiopenmp -fopenmp-targets=spir64 -o simple-iota
```

Results after compilation:

```
 ./simple-iota
Successfully completed on device.
```

> **NOTE** If you have an offload region present and no accelerator, the kernel falls back to traditional host compilation (without the OpenCL runtime) unless you are using the environment variable `OMP_TARGET_OFFLOAD=mandatory`.

Offload Code to GPU

To decide which GPU hardware and what parts of the code to offload, refer to the GPU optimization workflow guide.

To find opportunities to offload your code to GPU, use the Intel Advisor for Offload Modeling.

Debug GPU Code

The following list has some basic debugging tips for offloaded code.

- Check CPU or host/target or switch runtime to OpenCL to verify the correctness of code.
- You could use printf to debug your application. Both SYCL and OpenMP offload support printf in kernel code.
- Use environment variables to control verbose log information.

For SYCL, the following debug environment variables are recommended. A full list is available from GitHub.

Debugging Tips, Offloaded Code

Name	Value	Description
SYCL_DEVICE_FILTER	`backend:device_type:device_num`	GitHub description
SYCL_PI_TRACE	**1|2|-1**	**1**: print out the basic trace log of the DPC++ runtime plugin **2**: print out all API traces of DPC++ runtime plugin **-1**: all of "2" including more debug messages
ZE_DEBUG	Variable defined with any value - enabled	This environment variable enables debug output from the Level Zero backend when used with the DPC++ runtime. It reports: * Level Zero APIs called * Level Zero event information

For OpenMP, the following debug environment variables are recommended. A full list is available from the LLVM/OpenMP documentation.

Recommended OpenMP Debug Environment Variables

Name	Value	Description
LIBOMPTARGET_DEVICETYPE	cpu | gpu	Select
LIBOMPTARGET_DEBUG	**1**	Print out verbose debug information
LIBOMPTARGET_INFO	Values available in LLVM/OpenMP documentation	Allows the user to request different types of runtime information from `libomptarget`

Use Ahead of Time (AOT) to move Just-in-Time (JIT) compilations to AOT compilation issues.

`CL_OUT_OF_RESOURCES` Error

The **CL_OUT_OF_RESOURCES** error can occur when a kernel uses more `__private` or `__local` memory than the emulator supports by default.

When this occurs, you will see an error message similar to this:

```
$  ./myapp
:
Problem size: c(150,600) = a(150,300) * b(300,600)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
  what():  Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

Or if using onetrace:

```
$ onetrace -c ./myapp
:
>>>> [6254070891] zeKernelSuggestGroupSize: hKernel = 0x263b7a0 globalSizeX = 163850 globalSizeY
= 1 globalSizeZ = 1 groupSizeX = 0x7fff94e239f0 groupSizeY = 0x7fff94e239f4 groupSizeZ =
0x7fff94e239f8
<<<< [6254082074] zeKernelSuggestGroupSize [922 ns] ->
ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY(0x1879048195)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
  what():  Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

To see how much memory was being copied to shared local memory and the actual hardware limit, set debug keys:

```
export PrintDebugMessages=1
export NEOReadDebugKeys=1
```

This will change the output to:

```
$ ./myapp
:
Size of SLM (656384) larger than available (131072)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
  what():  Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

Or, if using onetrace:

```
$ onetrace -c ./myapp
:
>>>> [317651739] zeKernelSuggestGroupSize: hKernel = 0x2175ae0 globalSizeX = 163850 globalSizeY
= 1 globalSizeZ = 1 groupSizeX = 0x7ffd9caf0950 groupSizeY = 0x7ffd9caf0954 groupSizeZ =
0x7ffd9caf0958
Size of SLM (656384) larger than available (131072)
<<<< [317672417] zeKernelSuggestGroupSize [10325 ns] ->
ZE_RESULT_ERROR_OUT_OF_DEVICE_MEMORY(0x1879048195)
terminate called after throwing an instance of 'cl::sycl::runtime_error'
  what():  Native API failed. Native API returns: -5 (CL_OUT_OF_RESOURCES) -5
```

```
(CL_OUT_OF_RESOURCES)
Aborted (core dumped)
$
```

See Debugging the DPC++ and OpenMP Offload Process for more information on debug techniques and debugging tools available with oneAPI.

Optimize GPU Code

There are multiple ways to optimize offloaded code. The following list provides some starting points. Review the oneAPI GPU Optimization Guide for additional information.

- Reduce overhead of memory transfers between host and device.
- Have enough work to keep the cores busy and reduce the data transfer overhead cost.
- Use GPU memory hierarchy like GPU caches, shared local memory for faster memory accesses.
- Use AOT compilation (offline compilation) instead of JIT compilation. With offline compilation, you could target your code to specific GPU architecture. Refer to Offline Compilation for GPU for details.
- The Intel® GPU Occupancy Calculator allows you to compute the occupancy of an Intel® GPU for a given kernel and work group parameters.

Additional recommendations are available from Optimize Offload Performance.

Example GPU Commands

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

NOTE Linking with a dynamic library is not supported.

Produce a fat object with device code:

```
icpx -fsycl -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:

```
icpx -fsycl -c a.cpp
```

Link the application with the static library:

```
icpx -fsycl -foffload-static-lib=libstlib.a a.o -o a.exe
```

Ahead-of-Time Compilation for GPU

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

For DPC++:

```
icpx -fsycl-targets=spir64_gen -Xs "-device <device name>" a.cpp b.cpp -o app.out
```

For OpenMP*offload:

```
icpx -fiopenmp -fopenmp-targets=spir64_gen -Xopenmp-target-backend "-device <device name>" a.cpp
b.cpp -o app.out
```

A list of allowed values for the device name are available from the Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference.

FPGA Flow

Field-programmable gate arrays (FPGAs) are configurable integrated circuits that you can program to implement arbitrary circuit topologies. Classified as spatial compute architectures, FPGAs differ significantly from fixed Instruction Set Architecture (ISA) devices such as CPUs and GPUs. FPGAs offer a different set of optimization trade-offs from these traditional accelerator devices.

While you can compile SYCL* code for CPU, GPU or FPGA, the compiling process for FPGA development is somewhat different than that for CPU or GPU development.

The following table summarizes terminologies used in describing the FPGA flow:

FPGA Flow-specific Terminology

Term	Definition
Device code	SYCL source code that executes on a SYCL device rather than the host. Device code is specified via lambda expression, functor, or kernel class.
Host code	SYCL source code that is compiled by the host compiler and executes on the host rather than the device.
Device image	The result of compiling the device code to a binary (or intermediate) representation. The device image is combined with the host binary, within a (fat) object or executable file. See Compilation Flow Overview.
FPGA emulator image	The device image resulting from compiling for the FPGA emulator. See FPGA Emulator.
FPGA early image	The device image resulting from the early image compilation stage. See FPGA Optimization Report.
FPGA hardware image	The device image resulting from the hardware image compilation stage. See FPGA Optimization Report and FPGA Hardware.

> **Tip** You can also learn about programming for FPGA devices in detail from the *Data Parallel C++* book available at https://link.springer.com/chapter/10.1007/978-1-4842-5574-2_17.

Why is FPGA Compilation Different?

FPGAs differ from CPUs and GPUs in some ways. A significant difference compared to CPU or GPU is generating a device binary for FPGA hardware, which is a computationally intensive and time-consuming process. It is normal for an FPGA compile to take several hours to complete. For this reason, only ahead-of-time (or *offline*) kernel compilation mode is supported for FPGA. The long compile time for FPGA hardware makes just-in-time (or *online*) compilation impractical.

Longer compile times are detrimental to developer productivity. The Intel® oneAPI DPC++/C++ Compiler provides several mechanisms that enable you to target FPGA and iterate quickly on your designs. By circumventing the time-consuming process of full FPGA compilation wherever possible, you can benefit from the faster compile times that you are familiar with for CPU and GPU development.

Types of SYCL* FPGA Compilation

SYCL supports accelerators in general. The Intel® oneAPI DPC++/C++ Compiler implements additional FPGA-specific support to assist FPGA code development. This topic highlights different FPGA compilation flows that the Intel oneAPI Base Toolkit supports.

The following table summarizes the types of FPGA compilation:

Types of FPGA Compilation

Device Image Type	Time to Compile	Description
FPGA Emulator	Seconds	Compiles the FPGA device code to the CPU. Use the Intel® FPGA Emulation Platform for OpenCL™ software to verify your SYCL code's functional correctness.
Optimization Report	Minutes	Partially compiles the FPGA device code for hardware. The compiler generates an optimization report that describes the structures generated on the FPGA, identifies performance bottlenecks, and estimates resource utilization. When your compilation targets an FPGA device family or part number, this stage also give you RTL files for the IP component in your code. You can then use Intel® Quartus® Prime software to integrate your IP components into a larger design.
FPGA Simulator	Minutes	Compiles the FPGA device code to the CPU. Use the Questa*-Intel® FPGA Edition simulator to debug your code.
FPGA Hardware Image	Hours	When your compilation targets an FPGA acceleration board, this stage generates the real FPGA bitstream to execute on the target FPGA platform. When your compilation targets an FPGA device family or part number, this stage also gives you RTL files for the IP component in your code. You can then use Intel® Quartus® Prime software to integrate your IP components into a larger design.

A typical FPGA development workflow is to iterate in the emulation, optimization report, and simulation stages, refining your code using the feedback provided by each stage. Intel® recommends relying on emulation and the FPGA optimization report whenever possible.

For details about how these stages apply when developing IP components, refer to FPGA IP Authoring Flow

---

Tip To compile for FPGA emulation or FPGA simulation, generate the FPGA optimization report, you require only the Intel® oneAPI DPC++/C++ Compiler that is part of the Intel® oneAPI Base Toolkit. An FPGA hardware compile requires installing the Intel® Quartus® Prime software separately. Targeting a board also requires that you install the BSP for the board.

For more information, refer to the Intel® oneAPI Toolkits Installation Guide and Intel® FPGA development flow webpage.

Also, generating RTL code for an IP component requires only the Intel® oneAPI DPC++/C++ Compiler that is part of the Intel® oneAPI Base Toolkit. However, for simulating or integrating that IP component into your hardware design requires installing the Intel® Quartus® Prime Pro Edition software.

FPGA Emulator

The FPGA emulator (Intel® FPGA Emulation Platform for OpenCL™ software) is the fastest method to verify the correctness of your code. It executes the SYCL device code on the CPU. The emulator is similar to the SYCL host device, but unlike the host device, the FPGA emulator device supports FPGA extensions such as FPGA pipes and `fpga_reg`. For more information, refer to Pipes Extension and Kernel Variables topics in the *FPGA Optimization Guide for Intel® oneAPI Toolkits*.

The following are some important caveats to remember when using the FPGA emulator:

- **Performance is not representative**.

  Never draw inferences about FPGA performance from the FPGA emulator. The FPGA emulator's timing behavior is not correlated to that of the physical FPGA hardware. For example, an optimization that yields a 100x performance improvement on the FPGA may not impact the emulator performance. The emulator might show an unrelated increase or decrease.
- **Undefined behavior may differ**.

  If your code produces different results when compiled for the FPGA emulator versus FPGA hardware, your code most likely exercises undefined behavior. By definition, undefined behavior is not specified by the language specification and might manifest differently on different targets.

For detailed information about emulation for full-stack acceleration kernels, refer to Emulate Your Kernel.

For information about emulation of IP components, refer to For more details, refer to Emulate and Debug Your IP Component.

FPGA Optimization Report

A full FPGA compilation occurs in the following stages, and optimization reports are generated after both stages:

FPGA Optimization Report

Stages	Description	Optimization Report Information
FPGA early image (Compilation takes minutes to complete)	The SYCL device code is optimized and converted into an FPGA design specified in the Verilog Register-Transfer Level (RTL) (a low-level, native entry language for FPGAs). The intermediate compilation result is the FPGA early device image that is not an executable.	Contains important information about how the compiler has transformed your SYCL device code into an FPGA design. The report includes the following information: - Visualizations of structures generated on the FPGA.

Stages	Description	Optimization Report Information
	The optimization report generated at this stage is static in nature.	• Performance and expected performance bottleneck. • Estimated resource utilization. For information about the FPGA optimization report, refer to the FPGA Optimization Guide for Intel® oneAPI Toolkits.
FPGA hardware image (Compilation takes hours to complete)	The Verilog RTL specifying the design's circuit topology is mapped onto the FPGA's primitive hardware resources by the Intel® Quartus® Prime pro Edition Software. The result is an FPGA hardware binary (also referred to as a bitstream).	Contains precise information about resource utilization and f_{MAX} numbers. For detailed information about how to analyze reports, refer to Analyze your Design section in the *FPGA Optimization Guide for Intel® oneAPI Toolkits*. For information about the FPGA hardware image, refer to the FPGA Optimization Guide for Intel® oneAPI Toolkits.

When your compilation targets an FPGA device or part number, this stage gives you RTL files for the IP component in your code. You can then use Intel® Quartus® Prime software to integrate your IP components into a larger design.

FPGA Simulator

The simulation flow allows you to use the Questa*-Intel® FPGA Edition simulator software to simulate the exact behavior of the synthesized kernel. Like emulation, you can run simulation on a system that does not have a target FPGA board installed. The simulator models a kernel much more accurately than the emulator, but it is much slower than the emulator.

The simulation flow is cycle-accurate and bit-accurate. It exactly models the behavior of a kernel's datapath and the results of operations on floating-point data types. However, simulation cannot accurately model variable-latency memories or other external interfaces. Intel recommends that you simulate your design with a small input dataset because simulation is much slower than running on FPGA hardware or emulator.

You can use the simulation flow in conjunction with profiling to collect additional information about your design. For more information about profiling, refer to Intel® FPGA Dynamic Profiler for DPC++ in the *FPGA Optimization Guide for Intel® oneAPI Toolkits*.

> **NOTE** You cannot debug kernel code compiled for simulation using the GNU Project Debugger (GDB)*, Microsoft* Visual Studio*, or any standard software debugger.

For more information about the simulation flow, refer to one of the following topics:

- Evaluate Your Kernel Through Simulation
- Evaluate Your IP Component Through Simulation

FPGA Hardware

An FPGA hardware compile requires the Intel® Quartus® Prime software (installed separately). This is a full compilation stage through to the FPGA hardware image where you can target one of the following:

- Intel® FPGA device family
- Specific Intel® FPGA device part number
- Custom board with a supported BSP
- Intel® Programmable Acceleration Card (PAC) (deprecated)

For more information about the targets, refer to the Intel® oneAPI DPC++/C++ Compiler System Requirements. For more information about using Intel® PAC or custom boards, refer to the FPGA BSPs and Boards section and and the Intel® oneAPI Toolkits Installation Guide for Linux* OS Installation Guide.

FPGA Compilation Flags

FPGA compilation flags control the FPGA image type the Intel® oneAPI DPC++/C++ Compiler targets.

The following are examples of Intel® oneAPI DPC++/C++ Compiler commands that target the FPGA image types:

```
# FPGA emulator image
icpx -fsycl -fintelfpga fpga_compile.cpp -o fpga_compile.fpga_emu


# FPGA simulator image: FPGA device family
# Note that in this release, you can target Intel Agilex® 7 devices in any of the following
formats:
# -Xstarget=Agilex  | -Xstarget=Agilex7 | -Xstarget="Agilex 7" | -Xstarget=agilex7 | -
Xstarget="agilex 7" |
# The following command is an example that shows one of the supported formats:
icpx -fsycl -fintelfpga fpga_compile.cpp -Xssimulation -Xstarget=Agilex7 -Xsghdl -o
fpga_compile.fpga_sim


# FPGA simulator image: FPGA part number
icpx -fsycl -fintelfpga fpga_compile.cpp -Xssimulation -Xstarget=AGFB014R24A3EV -Xsghdl  -o
fpga_compile.fpga_sim


# FPGA simulator image: explicit board
icpx -fsycl -fintelfpga fpga_compile.cpp -Xssimulation -Xstarget=intel_s10sx_pac:pac_s10 -o
fpga_compile.fpga_sim


# FPGA early image (with optimization report): FPGA device family
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -fsycl-link=early -Xstarget=Stratix10 -o
fpga_compile_report.a


# FPGA early image (with optimization report): FPGA part number
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -fsycl-link=early -
Xstarget=1SG280LU3FS0E3VG -o fpga_compile_report.a


# FPGA early image (with optimization report): default board
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -fsycl-link=early -o fpga_compile_report.a


# FPGA early image (with optimization report): explicit board
```

```
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -fsycl-link=early -
Xstarget=intel_s10sx_pac:pac_s10 -o fpga_compile_report.a


# FPGA hardware image: FPGA device family
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -Xstarget=Arria10 -o fpga_compile.fpga


# FPGA hardware image: FPGA part number
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -Xstarget=10AX115S2F45I1SG -o
fpga_compile.fpga


# FPGA hardware image: default board
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -o fpga_compile.fpga


# FPGA hardware image: explicit board
icpx -fsycl -fintelfpga fpga_compile.cpp -Xshardware -Xstarget=intel_s10sx_pac:pac_s10 -o
fpga_compile.fpga
```

> **NOTE** When you specify `-o` flag, the output directory is named based on what you specify with the `-o` flag. For example, if you specify `-o fpga_compile.fpga`, the output directory would be `fpga_compile.prj`. However, when you do not specify the `-o` flag, the output directory is always `a.prj`.

The following table explains the compiler flags used in the above example commands:

FPGA Compilation Flags

Flag	Explanation
`-fintelfpga`	Performs ahead-of-time (offline) compilation for FPGA.
`-Xshardware`	Instructs the compiler to target FPGA hardware. If you omit this flag, the compiler targets the FPGA emulator. > **NOTE** Using the prefix `-Xs` causes an argument to be passed to the FPGA backend.
`-Xsemulator`	Generates an emulator device image. This is the default behavior.
`-Xssimulation`	Generates a simulator device image.
`-fsycl-link=early`	Instructs the compiler to stop after creating the FPGA early image (and associated optimization report).

`-Xstarget=<FPGA device family>`
`-Xstarget=<FPGA part number>`
`-Xstarget=<bsp:variant>`
	[Optional] Instructs the compiler to target an FPGA device family, an FPGA part number, or an FPGA board as follows:
	
	• `-Xstarget=<FPGA device family>` specifies the target FPGA device family. Use this target to facilitate initial development.

Flag	Explanation
	For more accurate reports, use the `-Xstarget=<FPGA part number>` or `-Xstarget=<bsp:variant>` to target specific FPGA devices or acceleration platforms. Valid values are `CycloneV`, `Cyclone10GX`, `Agilex7`, `Arria10`, and `Stratix10`. These values target the following FPGA device part numbers (OPNs): • `CycloneV`: 5CGXFC7C7F23C8 • `Cyclone10GX`: 10CX220YF780I5G • `Agilex7`: AGFB014R24A2E2V • `Arria10`: 10AX115U1F45I1SG • `Stratix10`: 1SG280LU3F50I2VG --- **NOTE** In this release, you can target an Intel Agilex® 7 device in any of the following formats: • `-Xstarget=Agilex` • `-Xstarget=Agilex7` • `-Xstarget="Agilex 7"` • `-Xstarget=agilex7` • `-Xstarget="agilex 7"` --- • `-Xstarget=<FPGA part number>` specifies the target FPGA part number (sometimes called an OPN). You can specify any valid Cyclone® V, Intel® Cyclone® 10 GX, Intel Agilex® 7, Intel® Arria® 10, or Intel® Stratix® 10 part number. • `-Xstarget=<bsp:variant>` specifies the FPGA board variant and BSP. Refer to the FPGA BSPs and Boards section for additional details. If you omit the `-Xstarget` flag, the compiler chooses the default FPGA board variant `pac_a10` from the `intel_a10gx_pac` BSP (equivalent to `-Xstarget=intel_a10gx_pac:pac_a10`).

Warning The output of an `icpx` compile command overwrites the output of previous compiles that used the same output name. Therefore, Intel® recommends using unique output names (specified with `-o`). This is especially important for FPGA compilation since a lost hardware image may take hours to regenerate.

In addition to the compiler flags demonstrated by the commands above, there are flags to control the verbosity of the `icpx` command's output, the number of parallel threads to use during compilation, and so on. The following section briefly describes those flags.

Other SYCL* FPGA Flags Supported by the Compiler

The Intel® oneAPI DPC++/C++ Compiler offers several options that allow you to customize the kernel compilation process. The following table summarizes other options supported by the compiler:

Other Supported FPGA Flags

Option name	Description
`-fsycl-help=fpga`	Prints out FPGA-specific options for the `icpx` command.

`-fsycl-link=early`
 	
`-fsycl-link=image`
	• `-fsycl-link=early` is synonymous with `-fsycl-link`. Both instruct the compiler to stop after creating the FPGA early image (and the associated optimization report).
	• `-fsycl-link=image` is used in the device link compilation flow to instruct the compiler to generate the FPGA hardware image. Refer to the Fast Recompile for FPGA section for additional information.

`-reuse-exe=<exe_file>`	Instructs the compiler to extract the compiled FPGA hardware or simulation image from the existing executable if the kernel has not changed and package it into the new executable, saving the device or simulation compilation time. This option is not applicable when compiling for emulation. Refer to the Fast Recompile for FPGA section for additional information.
`-Xsv`	FPGA backend generates a verbose output describing the progress of the compilation.
`-Xsghdl[=<depth>]`	Causes the simulation flow to log signals to Siemens EDA *(formerly Mentor Graphics)* Questa* waveform files. Use the optional `<depth>` attribute to specify how many levels of hierarchy are logged. If you do not specify a value for the `<depth>` attribute, a depth of 1 is used by default.
`-Xsparallel=<num_threads>`	Sets the degree of parallelism used in the FPGA bitstream compilation. The `<num_threads>` value specifies the number of parallel threads you want to use. The maximum recommended value is the number of available cores. Setting this flag is optional. The default behavior is for the Intel® Quartus® Prime software to compile in parallel on all available cores.
`-Xsseed=<value>`	Sets the seed used by Intel® Quartus® Prime software when generating the FPGA bitstream. The value must be an unsigned integer, and by default, the value is 1.
`-Xsfast-compile`	Runs FPGA bitstream compilation with reduced effort. This option allows faster compile time but at the cost of reduced performance of the compiled

Option name	Description
	FPGA hardware image. Use this flag only for faster development time. It is not intended for production-quality results. The `-Xsfast-compile` flag sets the Intel Quartus Prime Pro Edition software into the compile mode that is dominated by the Fast Functional Test. **Warning** When compiling your SYCL kernel using the `-Xsfast-compile` flag, you might see functional failures due to timing violations in your design. In such cases, either avoid using the `-Xsfast-compile` flag or try compiling your kernel with different seeds.

For more information about FPGA optimization flags, refer to the Optimization Flags section in the *FPGA Optimization Guide for Intel® oneAPI Toolkits*.

Emulate and Debug Your Design

The Intel FPGA Emulation Platform for OpenCL software (also referred to as the emulator or the FPGA emulator) is installed as part of the Intel® oneAPI Base Toolkit. It assesses the functionality of your kernel. The emulator supports 64-bit Windows and Linux operating systems. On Linux systems, the GNU C Library (glibc) version 2.15 or later is required.

> **NOTE**
> - You cannot use the execution time of an emulated design to estimate its execution time on an FPGA. Furthermore, running an emulated design is not a substitute for natively running a functionally equivalent C/C++ implementation on an x86-64 host.
> - Emulation does not support cross-compilation to ARM® processor. To run emulation on a design that targets an ARM SoC device, emulate on a non-SoC board (for example, `intel_a10gx_pac` or `intel_s10sx_pac`). When satisfied with the emulation results, you can target your design on an SoC board for subsequent optimization steps.
> - To enable debugging of kernel code, optimizations are disabled by default for the FPGA emulator. This can lead to sub-optimal execution speed when emulating kernel code. You can pass the `-g0` flag to the `icpx` compile command to disable debugging and enable optimizations. This enables faster emulator execution.
> - When targeting the FPGA emulator device, use the `-O2` compiler flag to turn on optimizations and speed up the emulation. To turn off optimizations (for example, to facilitate debugging), pass `-O0`. With Windows Visual C++ debugger, specify `/Od`.
> - For information about debugging with Intel® Distribution for GDB*, refer to the following:
>   - Debugging with Intel® Distribution for GDB* on Linux* OS Host
>   - Get Started with Intel® Distribution for GDB* on Linux* OS Host
>   - Get Started with Intel® Distribution for GDB* on Windows* OS Host

Refer to the following topics for additional information:

- Emulator Environment Variables
- Emulate Pipe Depth
- Emulate Applications with a Pipe That Reads or Writes to an I/O Pipe

- Compile and Emulate Your Design
- Limitations of the Emulator
- Discrepancies in Hardware and Emulator Results
- Emulator Known Issues

Emulator Environment Variables

The following table lists environment variables that you can use to modify the behavior of the emulator:

Emulator Environment Variables

Environment Variable	Description
CL_CONFIG_CPU_EMULATE_DEVICES	Controls the number of identical emulator devices provided by the emulator platform. If not set, a single emulator device is available. Therefore, set this variable only if you want to emulate multiple devices.
DPCPP_CPU_NUM_CUS	Indicates a maximum number of threads that the emulator can use. The default value is 32, and the maximum value is 255. Each thread can run a single kernel. If the application requires several kernels to be executing simultaneously, you must set the DPCPP_CPU_NUM_CUS environment variable appropriately to the number of kernels used or a higher value.
CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE	Set the amount of available local memory with units. For example: 8MB, 256KB, or 1024B.
CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE	Set the amount of available private memory with units. For example: 8MB, 256KB, or 1024B. **NOTE** On Windows, the FPGA emulator can silently fail by running out of memory. As a workaround to catch this error, write your kernel code using the try-catch syntax.
CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE	When you compile your kernel for emulation, the pipe depth is different from the pipe depth generated when your kernel is compiled for hardware. You can change this behavior with the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable. For details, see Emulate Pipe Depth.

Emulate Pipe Depth

When you compile your kernel for emulation, the default pipe depth is different from the default pipe depth generated when your kernel is compiled for hardware. You can change this behavior when you compile your kernel for emulation with the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable.

> **Important** For pipes, you must set the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable before running the host program.

The CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable accepts the following values:

CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE values

Environment Variable	Description
`ignoredepth`	All pipes are given a pipe depth chosen to provide the fastest execution time for your kernel emulation. Any explicitly set pipe depth attribute is ignored.
`default`	Pipes with an explicit depth attribute have their specified depth. Pipes without a specified depth are given a default pipe depth that is chosen to provide the fastest execution time for your kernel emulation.
`strict`	All pipe depths in the emulation are given a depth that matches the depth given for the FPGA compilation. If the specified depth is not given, the depth will be 1. This value is used by default if the CL_CONFIG_CHANNEL_DEPTH_EMULATION_MODE environment variable is not set.

Emulate Applications with a Pipe That Reads or Writes to an I/O Pipe

The Intel® FPGA Emulation Platform for OpenCL™ software emulates kernel-to-kernel pipes. However, it does not support interacting directly with the hardware I/O pipes on your target board. Nevertheless, you can emulate the behavior of I/O pipes using the following procedures:

For Input I/O Pipes

1. Store input data to be transferred to the pipe in a file with a name matching the `id` specialization of the pipe. Consider the following example:

```
// Specialize a pipe type
struct read_io_pipe {
  static constexpr unsigned id = 0;
};
using read_iopipe = sycl::ext::intel::kernel_readable_io_pipe<read_io_pipe, unsigned, 4>;
```

2. Create a file named `0`.
3. Store the test input data in the file `0`.

For Output I/O Pipes

Output data is automatically written to a file with a name matching the `id` specialization of the output pipe.

Compile and Emulate Your Design

To compile and emulate your FPGA kernel design, perform the following steps:

1. Modify the host part of your program to declare the `sycl::ext::intel::fpga_emulator_selector_v` device selector. Use this selector when instantiating a device queue for enqueuing your FPGA device kernel. For more information, refer to Device Selectors for FPGA.
2. Compile your design by including the `-fintelfpga` option in your `icpx` command to generate an executable.
3. Run the resulting executable:
   - For Windows:

1. Define the number of emulated devices by invoking the following command:

```
set CL_CONFIG_CPU_EMULATE_DEVICES=<number_of_devices>
```

2. Run the executable.
3. Invoke the following command to unset the variable:

```
set CL_CONFIG_CPU_EMULATE_DEVICES=
```

- For Linux, invoke the following command:

```
env CL_CONFIG_CPU_EMULATE_DEVICES=<number_of_devices> <executable_filename>
```

This command specifies the number of identical emulation devices that the emulator must provide.

Tip If you want to use only one emulator device, you need not set the `CL_CONFIG_CPU_EMULATE_DEVICES` environment variable.

NOTE

- The emulator is built with GCC 7.4.0 as part of the Intel® oneAPI DPC++/C++ Compiler. When running the executable for an emulated FPGA device, the version of `libstdc++.so` must be at least that of GCC 7.4.0. In other words, the `LD_LIBRARY_PATH` environment variable must ensure that the correct version of `libstdc++.so` is found.

  If the correct version of `libstdc++.so` is not found, the call to `clGetPlatformIDs` function fails to load the FPGA emulator platform and returns `CL_PLATFORM_NOT_FOUND_KHR` (error code –1001). Depending on which version of `libstdc++.so` is found, the call to `clGetPlatformIDs` may succeed, but a later call to the `clCreateContext` function may fail with `CL_DEVICE_NOT_AVAILABLE` (error code –2).

  If the `LD_LIBRARY_PATH` does not point to a compatible `libstdc++.so`, use the following syntax to invoke the host program:

  ```
  env LD_LIBRARY_PATH=<path to compatible libstdc++.so>:$LD_LIBRARY_PATH <executable>
  [executable arguments]
  ```

Limitations of the Emulator

The Intel® FPGA Emulation Platform for OpenCL™ software has the following limitations:

- **Concurrent execution**

  Modeling of concurrent kernel executions has limitations. During execution, the emulator is not guaranteed to run interacting work items in parallel. Therefore, some concurrent execution behaviors, such as different kernels accessing global memory without a barrier for synchronization, might generate inconsistent emulation results between executions.

- **Same address space execution**

  The emulator executes the host runtime and kernels in the same address space. Certain pointer or array use in your host application might cause the kernel program to fail and vice versa. Example uses include indexing externally allocated memory and writing to random pointers.

  To analyze your program, you may use memory leak detection tools, such as Valgrind. However, the host might encounter a fatal error caused by out-of-bounds write operations in your kernel and vice versa.

- **Conditional pipe operations**

  Emulation of pipe behavior has limitations, especially for conditional pipe operations where the kernel does not call the pipe operation in every loop iteration. In these cases, the emulator might execute pipe operations in a different order than on the hardware.

- **GCC version**

You must run the emulator host programs on Linux with a version of `libstdc++.so` from GCC 7.4.0 or later. You can achieve this either by installing GCC 7.4.0 or later on your system or setting the `LD_LIBRARY_PATH` environment variable such that a compatible `libstdc++.so` is identified.

Troubleshooting Discrepancies in Hardware and Emulator Results

When you emulate a kernel, your kernel might produce results different from the kernel compiled for hardware. You can further debug your kernel before you compile for hardware by running your kernel through simulation.

Warning These discrepancies usually occur when the Intel® FPGA Emulation Platform for OpenCL™ is unable to model some aspects of the hardware computation accurately or when your program relies on undefined behavior.

The most common reasons for differences in emulator and hardware results are as follows:

- Your kernel code is using the ivdep attribute. The emulator does not model your kernel when the `ivdep` attribute breaks a true dependence. During a full hardware or simulator compilation, you observe this as an incorrect result.
- Your kernel code relies on uninitialized data. Examples of uninitialized data include uninitialized variables and uninitialized or partially initialized global buffers, and arrays.
- Your kernel code behavior depends on the precise results of floating-point operations. The emulator uses floating-point computation hardware of the CPU, whereas the hardware run uses floating-point cores implemented as FPGA cores.

NOTE The SYCL* standard allows one or more least significant bits of floating-point computations to differ between platforms while still being considered correct on both such platforms.

- Your kernel code behavior depends on the order of pipe accesses in different kernels. The emulation of pipe behavior has limitations, especially for conditional pipe operations where the kernel does not call the pipe operation in every loop iteration. In such cases, the emulator might execute pipe operations in an order different from that of the hardware. For instance, if you have two kernels connected by pipes, and each kernel contains a loop containing a `read()` or `write()` function that does not happen every loop iteration (for example, if it is gated by an if-statement), the emulator might interleave the `read()` or `write()` calls differently than the hardware.
- Your kernel or host code is accessing global memory buffers out-of-bounds.

NOTE

- Uninitialized memory read and write behaviors are platform-dependent. Verify the sizes of your global memory buffers when using all addresses within kernels.
- You can use software memory leak detection tools, such as Valgrind, on the emulated version of your kernel to analyze memory-related problems. The absence of warnings from such tools does not mean the absence of issues. It only means that the tool could not detect any problem. In such a scenario, Intel recommends manual verification of your kernel or host code.

- Your kernel code is accessing local variables out-of-bounds. For example, accessing a local array out-of-bounds or accessing a variable after it has gone out of scope.

> **NOTE** In software terms, these issues are stack corruption issues because accessing variables out of bounds usually affects unrelated variables located close to the variable being accessed on a stack. Emulated kernels are implemented as regular CPU functions and have an actual stack that can be corrupted. When targeting hardware, no stack exists. Hence, the stack corruption issues are guaranteed to manifest differently. When you suspect a stack corruption, use memory leak analyzer tools, such as Valgrind. However, stack-related issues are usually difficult to identify. Intel recommends manual verification of your kernel code to debug a stack-related issue.

- Your kernel code uses shifts that are larger than the type being shifted. For example, shifting a 64-bit integer by 65 bits. According to the SYCL specification version 1.0, the behavior of such shifts is undefined.
- When you compile your kernel for emulation, the default pipe depth is different from the default pipe depth generated when your kernel is compiled for hardware. This difference in pipe depths might lead to scenarios where execution on the hardware hangs while kernel emulation works without any issue. Refer to Emulate Pipe Depth for information about fixing the pipe depth difference.
- In terms of ordering the printed lines, the output of the `cout stream` function might be ordered differently on the emulator and hardware. This is because, in the hardware, `cout stream` data is stored in a global memory buffer and flushed from the buffer only when the kernel execution is complete or when the buffer is full. In the emulator, the `cout stream` function uses the x86 `stdout`.
- The hardware and emulator might produce different results if you perform an unaligned load/store through upcasting of types. A load/store of this type is undefined in the C99 specification.
- When debugging unknown behaviors that differ between emulation and simulation/hardware, Intel recommends using the `-Weverything` diagnostic command option for emulation. The `-Weverything` option turns on all warnings allowing you to utilize available diagnostics and expose risky coding patterns, which you might be inadvertently using in your design.

Emulator Known Issues

A few known issues might affect your use of the emulator. Review these issues to avoid possible problems when using the emulator.

Discrepancy in the Results for Task Sequence Functions

For task sequence functions that have multiple `async()` calls before the first `get()` call, the order the `get()` calls return results may not be the order in which the `async()` calls were called. To overcome this issue, use the simulator to verify that correct results are being returned in this scenario.

Compiler Diagnostics

Some compiler diagnostics are not yet implemented for the emulator.

`CL_OUT_OF_RESOURCES` Error Returned When Launching a Kernel

This can occur when a kernel uses more `__private` or `__local` memory than the emulator supports by default.

Once you have determined the amount of memory needed, try setting larger values for the `CL_CONFIG_CPU_FORCE_PRIVATE_MEM_SIZE` or the `CL_CONFIG_CPU_FORCE_LOCAL_MEM_SIZE` environment variable, as described in Emulator Environment Variables.

> **NOTE** On Windows, the FPGA emulator can silently fail by running out of memory. As a workaround to catch this error, write your kernel code using the try-catch syntax.

FPGA Runtime Compatibility With Emulation Binaries

The oneAPI FPGA runtime does not support emulation binaries built using an earlier version of oneAPI. You must recompile emulation binaries with the current oneAPI release.

`libstdc++.so` and GCC Version

The emulator is built with GCC 7.4.0 as part of the Intel® oneAPI DPC++/C++ Compiler. When running the executable for an emulated FPGA device, the version of `libstdc++.so` must be at least that of GCC 7.4.0. In other words, the `LD_LIBRARY_PATH` environment variable must ensure that the correct version of `libstdc++.so` is found.

If the correct version of `libstdc++.so` is not found, the call to `clGetPlatformIDs` function fails to load the FPGA emulator platform and returns `CL_PLATFORM_NOT_FOUND_KHR` (error code –1001). Depending on which version of `libstdc++.so` is found, the call to `clGetPlatformIDs` may succeed, but a later call to the `clCreateContext` function may fail with `CL_DEVICE_NOT_AVAILABLE` (error code –2).

If the `LD_LIBRARY_PATH` does not point to a compatible `libstdc++.so`, use the following syntax to invoke the host program:

```
env LD_LIBRARY_PATH=<path to compatible libstdc++.so>:$LD_LIBRARY_PATH <executable> [executable
arguments]
```

Evaluate Your Kernel Through Simulation

The Questa*-Intel® FPGA Edition and Questa*-Intel® FPGA Starter Edition software assess the functionality of your kernel.

The simulator flow generates a simulation binary file that runs on the host. The hardware portion of your code is evaluated in an RTL simulator, and the host portion is executed natively on the processor. This feature allows you to simulate the functionality of your kernel and iterate on your design without needing to compile your kernel to hardware and run on the FPGA each time.

> **NOTE** The performance of the simulator is very slow when compared to that of hardware. So, Intel recommends using a smaller data set for testing.

Use the simulator when you want an insight into the dynamic performance of your kernel and more information about the functional correctness of your kernel than emulation or the reporting tools provide.

The simulator is cycle accurate and bit-accurate. It has a netlist identical to the generated hardware and can provide full waveforms for debugging. View the waveforms with Siemens* EDA (*formerly Mentor Graphics*) Questa* software.

- Simulation Prerequisites
- Installing the Questa*-Intel FPGA Edition Software
- Set Up the Simulation Environment
- Compile a Kernel for Simulation
- Simulate Your Kernel
- Viewing Simulation Waveforms
- Troubleshoot Simulator Issues

Simulation Prerequisites

To use the FPGA simulation flow, you must download the following prerequisite software:

- **Intel Quartus Prime Pro Edition software**: Download this package from the FPGA Software Download Center download page.
- **Compatible simulation software (Questa*-Intel® FPGA Edition and Questa*-Intel® FPGA Starter Edition)**: Obtain them from the FPGA Software Download Center.

NOTE

- The Questa*-Intel® FPGA Edition requires a license. However, Questa*-Intel® FPGA Starter Edition is free but requires a zero-cost license. For additional details, refer to the Licensing chapter of the Intel FPGA Software Installation and Licensing.
- You can also use your licensed version of Siemens* EDA ModelSim* SE or Siemens* EDA Questa Advanced Simulator software. For information about all ModelSim* and Questa* software versions that your Intel® Quartus® Prime Pro Edition software supports, refer to the EDA Interface Information section of the *Intel® Quartus® Prime Pro Edition: <version_number> Software and Device Support Release Notes*.
- On Linux systems, you must install Red Hat* development tools to work with Questa*-Intel® FPGA Edition and Questa*-Intel® FPGA Starter Edition software.

Installing the Questa*-Intel FPGA Edition Software

For installation instructions, refer to the Questa*-Intel® FPGA Edition Quick-Start: Intel® Quartus® Prime Pro Edition guide.

NOTE From within the oneAPI environment, you can determine the Intel® Quartus® Prime software installation location by inspecting the `QUARTUS_ROOTDIR_OVERRIDE` environment variable.

Set Up the Simulation Environment

You must add directories containing the Intel® Quartus® Prime and Questa* simulation software binaries to your `PATH` environment variable.

NOTE Commands listed in this topic assume that you have installed the Questa* simulation software alongside the Intel® Quartus® Prime Pro Edition software, as mentioned in the Simulation Prerequisites. If you installed the Questa* simulation software elsewhere, you must modify the `PATH` environment variable appropriately.

For Intel® Quartus® Prime Software (Simulation flow only)

For the FPGA simulation flow only, you must explicitly add the Intel® Quartus® Prime software binary directory to your `PATH` environment variable using the following command:

- **Linux**

```
$ export PATH=$PATH:<quartus_installdir>/quartus/bin
```

- **Windows**

```
set "PATH=%PATH%;<quartus_installdir>\quartus\bin64"
```

Additionally, you must also set the `OCL_ICD_FILENAMES` variable to specify the Installable Client Driver (ICD) to load.

```
set "OCL_ICD_FILENAMES=%OCL_ICD_FILENAMES%;alteracl_icd.dll"
```

For Questa*-Intel® FPGA Starter Edition Software

For the free Questa*-Intel® FPGA Starter Edition software, run the following command:

- **Linux**

```
$ export PATH=$PATH:<quartus_installdir>/questa_fse/bin
```

- **Windows**

```
set "PATH=%PATH%;<quartus_installdir>\questa_fse\win64"
```

For Questa*-Intel® FPGA Edition Software

For the licensed Questa*-Intel® FPGA Edition software, run the following command:

- **Linux**

```
$ export PATH=$PATH:<quartus_installdir>/questa_fe/bin
```
- **Windows**

```
set "PATH=%PATH%;<quartus_installdir>\questa_fe\win64"
```

You should now be able to successfully compile for simulation.

Compile a Kernel for Simulation

Before performing simulation, you must ensure that you have installed the Intel® Quartus Prime Pro Edition software on your system. For more information, refer to the Intel® oneAPI Toolkits Installation Guide and Intel® FPGA development flow webpage.

To compile a kernel for simulation, include the `-Xssimulation` option in your `icpx` command as shown in the following:

```
icpx -fsycl -fintelfpga -Xssimulation fpga_compile.cpp
```

To enable collecting the waveform during the simulation, include the `-Xsghdl[=<depth>]` option in your `icpx` command, where the optional `<depth>` attribute specifies how many levels of hierarchy are logged. If you do not specify a value for the `<depth>` attribute, a depth of 1 is used by default. To log all waveforms, specify a depth of 0 (`-Xsghdl=0`).

When simulating on Windows systems, you need the Microsoft linker and additional compilation time libraries. Verify the following settings:

- The `PATH` environment variable setting must include the path to the `LINK.EXE` file in Microsoft Visual Studio.
- `LIB` environment variable setting includes the path to the Microsoft compile-time libraries. The compile-time libraries are available with Microsoft Visual Studio.

Simulate Your Kernel

The simulation runtime creates a simulation device based on the automatically discovered or specified `board_spec.xml`/`ipinterfaces.xml` file. You can apply two simulation environment variables to control the `.xml` file searching mechanism.

- `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA`: When you set this environment variable to 1, the simulation runtime attempts to automatically search for a `board_spec.xml`/`ipinterfaces.xml` file to use, based on the following rules:
  - If there is only one `.prj` directory in the current working directory, the compiler uses the `board_spec.xml`/`ipinterfaces.xml` file in that directory.
  - If there are multiple `.prj` directories in the current working directory, the compiler uses the `board_spec.xml`/`ipinterfaces.xml` file in the directory with the name matching the current executable name. When comparing the name, the compiler strips the extensions (`.exe`, `.bin`, `.elf`, and `.out` for Linux and `.exe` for Windows).

NOTE When the automatic search fails, the runtime emits a `CL_DEVICE_NOT_AVAILABLE` error.

- `INTELFPGA_SIM_DEVICE_SPEC_DIR`: Use this environment variable to set the path to the directory containing the target `.xml` file when the automatic search mechanism fails or when the automatically found `.xml` file is not desired. This environment variable forces the value of the `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA` variable to 1 so that you can avoid specifying both variables to run a design in simulation.

If you want to use the simulation flow and view the waveforms generated during simulation, you must have either the Siemens EDA* Questa Simulator or ModelSim SE installed and available.

To run your SYCL library through the simulator:

1. Set the `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA` or `INTELFPGA_SIM_DEVICE_SPEC_DIR` environment variable to enable the simulation device:

    - **Linux**

```
export CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
                    or
export INTELFPGA_SIM_DEVICE_SPEC_DIR=<prj dir>
```

    - **Windows**

```
set CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1
                  or
set INTELFPGA_SIM_DEVICE_SPEC_DIR=<prj dir>
```

    > **NOTE** When you set any of the simulation environment variables, only the simulation devices are available. That is, access to physical boards is disabled. To unset the environment variable, run the following command:
    >
    > - **Linux**
    >
    > ```
    > unset CL_CONTEXT_MPSIM_DEVICE_INTELFPGA
    >                   or
    > unset INTELFPGA_SIM_DEVICE_SPEC_DIR
    > ```
    >
    > - **Windows**
    >
    > ```
    > set CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=
    >                   or
    > set INTELFPGA_SIM_DEVICE_SPEC_DIR=
    > ```

    You might need to set `CL_CONTEXT_COMPILER_MODE_INTELFPGA=3` if the host program cannot find the simulator device.
2. Run your host program. On Linux systems, you can use GDB or Eclipse to debug your host. If necessary, you can inspect the simulation waveforms for your kernel code to verify the functionality of the generated hardware.

    If you compiled with the `-Xsghdl` flag, running your compiled program produces a waveform file (`vsim.wlf`) that you can view in the Questa*-Intel FPGA Edition software as your host code executes. The `vsim.wlf` file appears in the `.prj` directory.

Viewing Simulation Waveforms

By default, the Intel oneAPI DPC++/C++ Compiler instructs the simulator not to log any signal because logging signals slows the simulation, and the waveform files can be enormous. However, you can configure the compiler to save these waveforms for debugging purposes.

To enable signal logging in the simulator, invoke the `icpx` command with the `-Xsghdl` option, as follows:

```
icpx -fsycl -fintelfpga -Xssimulation -Xsghdl[=<depth>] <input files> -o <project_name>
```

Specify the `<depth>` attribute to indicate the number of hierarchy levels logged. A depth value of 1 logs only the top-level signals. A depth of 1 is used as the default if you do not specify the `<depth>` attribute. To log all waveforms, specify a depth of 0 (`-Xsghdl=0`).

After running the simulation, you can view the generated waveform files by invoking the appropriate script as follows:

- **Linux**

```
bash <project_directory>/view_waveforms.sh
```
- **Windows**

```
<project_directory>\view_waveforms.cmd
```

> **NOTE** The `<project_directory>` is commonly `<project_name>.prj`, where `<project_name>` is the name specified with the `-o` argument to the `icpx` command.

Troubleshoot Simulator Issues

Review this section to troubleshoot simulator problems you might have when attempting to run a simulation.

Windows Compilation or Run Fails

On Windows, simulation might fail at compilation time or run time if you are running from a directory with a very long path. Use the `-o` compiler option to output your compilation results to a shorter path.

A `socket=-11` Error Is Logged to `transcript.log`

If you receive the following error message, you might be mixing resources from multiple simulators, such as Questa*-Intel FPGA Edition and ModelSim* SE:

```
Message: "src/hls_cosim_ipc_socket.cpp:202: void IPCSocketMaster::connect():
Assertion `sockfd != -1 && "IPCSocketMaster::connect() call to accept() failed"' failed."
```

An example of mixing simulator resources is compiling a device with ModelSim* SE and running the host program in Questa*-Intel FPGA Starter Edition.

Compatibility with Questa*-Intel FPGA Starter Edition Software

Questa*-Intel FPGA Starter Edition software has limitations on design size that prevent it from simulating large designs. When trying to launch a simulation using Questa*-Intel FPGA Starter Edition software, you may encounter the following error message:

```
Error: The simulator's process ended unexpectedly.
```

Instead, simulate the designs with Questa*-Intel FPGA Edition or ModelSim* SE software.

Environment Variable Not Set

When you forget setting the `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1` environment variable, you might encounter an error message.

On Linux

```
terminate called after throwing an instance of 'sycl::_V1::runtime_error'
what():  No device of requested type available. Please check https://www.intel.com/
content/www/us/en/developer/articles/system-requirements/intel-oneapi-dpcpp-system-
requirements.html -1 (PI_ERROR_DEVICE_NOT_FOUND)
Aborted (core dumped)
```

On Windows

The simulator might crash and you might see the following message in the debugger:

```
Unhandled exception at 0x00007FFD92DFDE4E (ucrtbase.dll) in <your exe>.exe: Fatal program exit
requested.
```

To overcome this issue, define the `CL_CONTEXT_MPSIM_DEVICE_INTELFPGA=1` environment variable.

Device Selectors for FPGA

Depending on whether you are targeting the FPGA emulator, simulator, or hardware, you must use the correct SYCL* device selector in the host code. You can use the FPGA hardware device selector for simulation also. The following host code snippet demonstrates how you can use a selector to specify the target device at compile time:

```
// FPGA device selectors are defined in this utility header, along with
// all FPGA extensions such as pipes and fpga_reg
#include <sycl/ext/intel/fpga_extensions.hpp>

int main() {
  // Select either:
  //  - the FPGA emulator device (CPU emulation of the FPGA)
  //  - the FPGA simulator
  //  - the FPGA device (a real FPGA)
  #if FPGA_SIMULATOR
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
  #elif FPGA_HARDWARE
    auto selector = sycl::ext::intel::fpga_selector_v;
  #else  // #if FPGA_EMULATOR
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
  #endif
  queue q(selector);
  ...
}
```

NOTE
- The FPGA emulator and the FPGA are different target devices. Intel® recommends using a preprocessor `define` to choose between the emulator and FPGA selectors. This makes it easy to switch between targets using only command-line flags. For example, you can compile the above code snippet for the FPGA emulator by passing the flag `-DFPGA_EMULATOR` to the `icpx` command.
- Since FPGAs support only the ahead-of-time compilation method, dynamic selectors (such as the `default_selector`) are less useful that explicit selectors when targeting FPGAs.

Caution When targeting the FPGA emulator or FPGA hardware, you must pass correct compiler flags and use the correct device selector in the host code. Otherwise, you might experience runtime failures. Refer to the fpga_compile tutorial in the Intel® oneAPI Samples Browser to get started with compiling SYCL code for FPGA.

FPGA IP Authoring Flow

In the FPGA IP Authoring flow, the compiler uses your SYCL* code to generate IP components that you can integrate into a custom Intel® Quartus® Prime project. Use the IP authoring flow by targeting your compilation to a supported Intel® FPGA device family or part number instead of a specific acceleration platform.

Use this flow to help speed your IP development by letting you compile your SYCL* code to IP components on different targets that you can deploy into your systems.

For details about getting started with the IP component development flow, refer to Getting Started with Intel® oneAPI Toolkits and Intel® Quartus® Prime Software.

The typical design flow when you author IP components consists of the following stages:

1. **Creating your IP component and testbench.**

   Write a complete SYCL application that contains both your IP component code and your testbench code. The SYCL device code (kernel code) corresponds to your IP component, and the SYCL host code serves as the testbench for the emulation and simulation flows.

   For information about writing SYCL code, refer to Data Parallelism in C++ using SYCL*.

   Also, refer to Code IP Components in SYCL* for additional information specific to writing IP components in SYCL*.

2. **Verify the functionality of your IP component algorithm and testbench through emulation.**

   Verify the functionality of your IP component and refine the algorithms in your IP by compiling your design to an x86-64 executable and running the executable. For details, see Emulate and Debug Your IP Component.

3. **Optimize and refine the FPGA performance of your component.**

   Optimize the FPGA performance of your component by compiling your design for an FPGA device family or part number target with the `-Xstarget=<FPGA device family>` or `-Xstarget=<FPGA part number>` compiler option along with `-Xssimulation` or `-Xshardware` option and reviewing the FPGA Optimization Report to see where you can optimize your component. This step generates RTL code for your component.

   For details, refer to Analyze Your Design in the *FPGA Optimization Guide for Intel® oneAPI Toolkits*.

   After completing some initial optimization based on the contents of the FPGA Optimization Report, you can see where to further refine your component by simulating it.

   For details, see Evaluate Your IP Component Through Simulation.

4. **Synthesize your component with an FPGA hardware image compilation.**

   When you use the `-Xstarget=<FPGA device family>` or `-Xstarget=<FPGA part number>` compiler option, the Intel® oneAPI DPC++/C++ Compiler ties the inputs and outputs of your component to virtual pins and compiles the design to provide a more accurate estimate of your component's area and f_{MAX}. The generated output is not deployable to a board because the compilation occurred without a board support package.

   For details, refer to Synthesizing Your Component IP with Intel® Quartus® Prime Software.

   Synthesizing your component generates accurate quality-of-results (QoR) metrics like FPGA area utilization and f_{MAX}.

5. **Integrate your IP into a system with Intel® Quartus® Prime or Platform Designer.**

   For details, refer to Integrating Your IP Into a System.

When you are satisfied with the predicted performance of your component, use Intel® Quartus® Prime software to synthesize your component. Synthesis also generates accurate area and performance (f_{MAX}) estimates for your design. However, your design is not expected to cleanly close timing in the Intel® Quartus® Prime reports.

You can expect timing closure warnings in the Intel® Quartus® Prime logs because the generated project targets a clock speed of 1000 MHz to achieve the best possible placement for your design. The f_{MAX} value presented in the FPGA optimization report estimates the maximum clock rate that your component can cleanly close timing for.

After the Intel® Quartus® Prime compilation is completed, the summary section of the FPGA optimization report shows the area and performance data for your components.

These estimates are more accurate than estimates generated when you compile your IP component for simulation only.

Typically, Intel® Quartus® Prime compilation times can take minutes to hours depending on the size and complexity of your IP components.

To synthesize your component IP and generate quality of results (QoR) data, instruct the compiler to run the Intel® Quartus® Prime compilation flow automatically after synthesizing the components. Include the `–Xstarget=<FPGA device family>` or `-Xstarget=<FPGA part number>` options in your `icpx` command:

- `icpx -fsycl -fintelfpga -Xshardware -Xstarget=<FPGA device family>...`
- `icpx -fsycl -fintelfpga -Xshardware -Xstarget=<FPGA part number>...`

The following flowchart shows a coarse-grained progression through the stages of a typical IP component authoring design flow.

Overview of Procedure for Synthesizing IP for Intel® FPGA Products
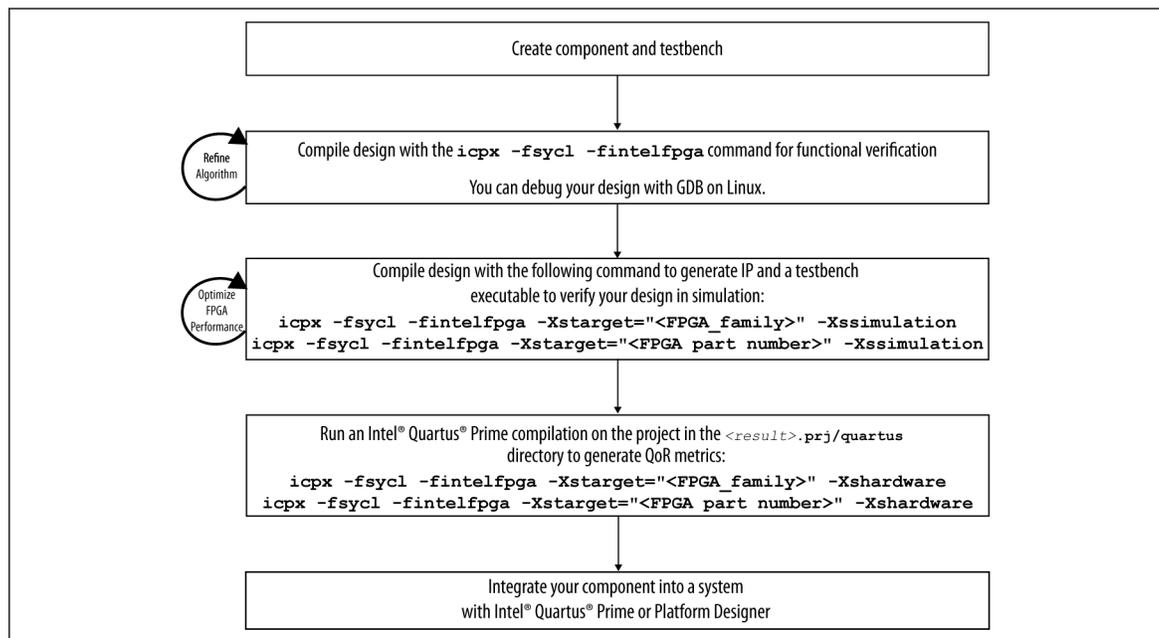
- Code IP Components in SYCL*

  - Customize RTL Interfaces
  - Suggested Coding Styles

    - Lambda Coding Style Example
    - Functor Coding Style Example
  - Memory-Mapped Host Interfaces

    - Addresses in Memory-Mapped Host Interfaces
  - Host Pipes
  - Agent IP Component Kernels

    - Example Register Map File
  - Streaming IP Component Kernels

    - Limitations of Streaming IP Component Kernels
  - Kernel Argument Interfaces
  - Pipelined Kernels
  - Stable Arguments
  - IP Component Reset Behavior
  - The `printf` Command
- Emulate and Debug Your IP Component
- Evaluate Your IP Component Through Simulation

  - Debug During Verification

- FPGA IP Component Performance Optimization
- Synthesizing Your Component IP with Intel® Quartus® Prime Software
- Integrating Your IP Into a System

    - Adding IP into an Intel® Quartus® Prime Project
    - Adding IP into a Platform Designer System
- Encrypt IP Components for Distribution

Code IP Components in SYCL*

When you write IP components in SYCL, consider these additional requirements and techniques.

- Customize RTL Interfaces
- Suggested Coding Styles

    - Lambda Coding Style Example
    - Functor Coding Style Example
- Memory-Mapped Host Interfaces

    - Addresses in Memory-Mapped Host Interfaces
- Host Pipes
- Agent IP Component Kernels

    - Example Register Map File
- Streaming IP Component Kernels

    - Limitations of Streaming IP Component Kernels
- Kernel Argument Interfaces
- Pipelined Kernels
- Stable Arguments
- IP Component Reset Behavior
- The `printf` Command

Customize RTL Interfaces

The compiler generates a component interface for integrating your RTL component into a larger system. An IP component has two basic interface types: the component invocation interface and the data interface.

IPs are generated by default using a control-and-status register (CSR) agent interface for consuming inputs. The Streaming IP Component Kernels section demonstrates how to use a streaming interface instead.

You can pass data into a kernel using the kernel arguments, host pipes, or through memory (using accessors or USM). You can pass items by value in the capture list of the lambda expression (often called a *lambda*) or by using an accessor or a Unified Shared Memory (USM) pointer to create an Avalon memory mapped host interface on your IP.

Your IP can produce output only through an accessor, USM pointer, or pipe. The CSR interface cannot capture output from an IP component generated from the Intel oneAPI DCP++/C++ compiler.

Suggested Coding Styles

For creating your IP, use one of the following recommended general coding styles:

- Lambda Coding Style Example: The lambda coding style is typically used in most full-system SYCL programs.
- Functor Coding Style Example: You can write your IP component (kernel) code out-of-line from the host code with the functor coding style.

Lambda Coding Style Example

```cpp
#include <iostream>
#include <vector>

// oneAPI headers
#include <sycl/sycl.hpp>
#include <sycl/ext/intel/fpga_extensions.hpp>

using namespace sycl;

// Forward declare the kernel name in the global scope. This is an FPGA best
// practice that reduces name mangling in the optimization reports.
class VectorAddID;

void VectorAdd(const int *vec_a_in, const int *vec_b_in, int *vec_c_out,
               int len) {
  for (int idx = 0; idx < len; idx++) {
    int a_val = vec_a_in[idx];
    int b_val = vec_b_in[idx];
    int sum = a_val + b_val;
    vec_c_out[idx] = sum;
  }
}

constexpr int kVectSize = 256;

int main() {
  bool passed = true;
  try {
    // Use compile-time macros to select either:
    //  - the FPGA emulator device (CPU emulation of the FPGA)
    //  - the FPGA device (a real FPGA)
    //  - the simulator device
#if FPGA_SIMULATOR
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
#elif FPGA_HARDWARE
    auto selector = sycl::ext::intel::fpga_selector_v;
#else  // #if FPGA_EMULATOR
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
#endif

    // create the device queue
    sycl::queue q(selector);

    // make sure the device supports USM host allocations
    auto device = q.get_device();

    std::cout << "Running on device: "
              << device.get_info<sycl::info::device::name>().c_str()
              << std::endl;

    if (!device.has(sycl::aspect::usm_host_allocations)) {
      std::terminate();
    }


    // declare arrays and fill them
    // allocate in shared memory so the kernel can see them
```

```
    int *vec_a = malloc_shared<int>(kVectSize, q);
    int *vec_b = malloc_shared<int>(kVectSize, q);
    int *vec_c = malloc_shared<int>(kVectSize, q);
    for (int i = 0; i < kVectSize; i++) {
      vec_a[i] = i;
      vec_b[i] = (kVectSize - i);
    }

    std::cout << "add two vectors of size " << kVectSize << std::endl;

    q.single_task<VectorAddID>([=]() {
        VectorAdd(vec_a, vec_b, vec_c, kVectSize);
    })
    .wait();

    // verify that vec_c is correct
    for (int i = 0; i < kVectSize; i++) {
      int expected = vec_a[i] + vec_b[i];
      if (vec_c[i] != expected) {
        std::cout << "idx=" << i << ": result " << vec_c[i] << ", expected ("
                  << expected << ") A=" << vec_a[i] << " + B=" << vec_b[i]
                  << std::endl;
        passed = false;
      }
    }

    std::cout << (passed ? "PASSED" : "FAILED") << std::endl;

    free(vec_a, q);
    free(vec_b, q);
    free(vec_c, q);
  } catch (sycl::exception const &e) {
    // Catches exceptions in the host code.
    std::cerr << "Caught a SYCL host exception:\n" << e.what() << "\n";

    // Most likely the runtime couldn't find FPGA hardware!
    if (e.code().value() == CL_DEVICE_NOT_FOUND) {
      std::cerr << "If you are targeting an FPGA, please ensure that your "
                   "system has a correctly configured FPGA board.\n";
      std::cerr << "Run sys_check in the oneAPI root directory to verify.\n";
      std::cerr << "If you are targeting the FPGA emulator, compile with "
                   "-DFPGA_EMULATOR.\n";
    }
    std::terminate();
  }
  return passed ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Functor Coding Style Example

With this style, you can specify all the interfaces in one location and make a call to your IP component from your SYCL* host program.

```
#include <iostream>

// oneAPI headers
#include <sycl/ext/intel/fpga_extensions.hpp>
#include <sycl/sycl.hpp>
```

```
// Forward declare the kernel name in the global scope. This is an FPGA best
// practice that reduces name mangling in the optimization reports.
class VectorAddID;

struct VectorAdd {
  int *const vec_a_in;
  int *const vec_b_in;
  int *const vec_c_out;
  int len;

  void operator()() const {
    for (int idx = 0; idx < len; idx++) {
      int a_val = vec_a_in[idx];
      int b_val = vec_b_in[idx];
      int sum = a_val + b_val;
      vec_c_out[idx] = sum;
    }
  }
};

constexpr int kVectSize = 256;

int main() {
  bool passed = true;
  try {
    // Use compile-time macros to select either:
    //  - the FPGA emulator device (CPU emulation of the FPGA)
    //  - the FPGA device (a real FPGA)
    //  - the simulator device
#if FPGA_SIMULATOR
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
#elif FPGA_HARDWARE
    auto selector = sycl::ext::intel::fpga_selector_v;
#else  // #if FPGA_EMULATOR
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
#endif

    // create the device queue
    sycl::queue q(selector);

   // make sure the device supports USM host allocations
    auto device = q.get_device();

    std::cout << "Running on device: "
              << device.get_info<sycl::info::device::name>().c_str()
              << std::endl;

    if (!device.has(sycl::aspect::usm_host_allocations)) {
      std::terminate();
    }

    // declare arrays and fill them
    // allocate in shared memory so the kernel can see them
    int *vec_a = sycl::malloc_shared<int>(kVectSize, q);
    int *vec_b = sycl::malloc_shared<int>(kVectSize, q);
    int *vec_c = sycl::malloc_shared<int>(kVectSize, q);
    for (int i = 0; i < kVectSize; i++) {
```

```
      vec_a[i] = i;
      vec_b[i] = (kVectSize - i);
    }

    std::cout << "add two vectors of size " << kVectSize << std::endl;

    q.single_task<VectorAddID>(VectorAdd{vec_a, vec_b, vec_c, kVectSize})
        .wait();

    // verify that vec_c is correct
    for (int i = 0; i < kVectSize; i++) {
      int expected = vec_a[i] + vec_b[i];
      if (vec_c[i] != expected) {
        std::cout << "idx=" << i << ": result " << vec_c[i] << ", expected ("
                  << expected << ") A=" << vec_a[i] << " + B=" << vec_b[i]
                  << std::endl;
        passed = false;
      }
    }

    std::cout << (passed ? "PASSED" : "FAILED") << std::endl;

    sycl::free(vec_a, q);
    sycl::free(vec_b, q);
    sycl::free(vec_c, q);
  } catch (sycl::exception const &e) {
    // Catches exceptions in the host code.
    std::cerr << "Caught a SYCL host exception:\n" << e.what() << "\n";

    // Most likely the runtime couldn't find FPGA hardware!
    if (e.code().value() == CL_DEVICE_NOT_FOUND) {
      std::cerr << "If you are targeting an FPGA, please ensure that your "
                   "system has a correctly configured FPGA board.\n";
      std::cerr << "Run sys_check in the oneAPI root directory to verify.\n";
      std::cerr << "If you are targeting the FPGA emulator, compile with "
                   "-DFPGA_EMULATOR.\n";
    }
    std::terminate();
  }
  return passed ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Memory-Mapped Host Interfaces

Each external memory interface is uniquely identified by a "buffer location" identifier. Buffer location identifiers can be applied to arguments annotated with the `mmhost` macros or to accessors. A kernel argument with a specified buffer location is sometimes referred to as an *annotated pointer argument*. An *unannotated pointer argument* can be any of the of the following items:

- Accessor arguments without a specified buffer location
- Kernel arguments not annotated with any `mmhost` macro

For each unique buffer location among the annotated pointer arguments of your kernels, the compile infers one memory-mapped host interface. If you have any unannotated pointer arguments in your kernels, one additional global memory-mapped interface is inferred (with buffer location `0`).

You can attribute a buffer location to a kernel argument in the following ways:

- Memory-Mapped Host Interfaces Using Unified Shared Memory

USM pointers allow you to customize the memory-mapped host interface.

In most cases, the compiler encodes the virtual address space information automatically. However, in some cases, you might need to encode it yourself. For details, refer to Memory-Mapped Interface Unified-Shared-Memory Virtual Address Space.
* Memory-Mapped Host Interfaces Using Accessors

Using accessors allows the runtime and BSP to manage the copying of the memory between the host and device.

Accessor kernel arguments result in four arguments for each accessor argument.

Addresses in Memory-Mapped Host Interfaces

Memory-mapped (MM) host interfaces issue byte addresses, not word addresses.

For example, if you dereference a pointer to a 4-byte wide datatype, the address issued by the MM host interface is base address of the pointer plus the array index multiplied by 4.

Assuming a base address of `0x0000` for `mm_a`, the following code example results in `0x0018` on the interface because `0x0018 = 0x0006 x 4`:

```
uint32_t value = mm_a[0x0006];
```

If `mm_a` has a base address of `0x1000`, the resulting address is `0x1018` instead of `0x0018`.

If you dereference a pointer to a 1-byte wide datatype, the address issued by the MM Host interface is the base address of the pointer plus the array index multiplied by 1.

Assuming a base address of `0x0000` for `mm_a`, the following code example results in `0x0006` on the interface because `0x0006 = 0x0006 x 1`:

```
uint8_t value = mm_a[0x0006];
```

If `mm_a` has a base address of `0x1000`, the resulting address is `0x1006` instead of `0x0006`.

Memory-Mapped Host Interfaces Using Unified Shared Memory

You can customize memory-mapped interfaces of your IP component if the component uses a unified shared memory (USM) host pointer to access data.

To customize the interface, use a functor to specify the component and use one of the macros described here. To use the macros, include the header file `sycl/ext/intel/prototype/interfaces.hpp` in your code.

Use the following flag when you compiler your kernel to ensure that the header file is on the include path:

* On Linux: `-I/$INTELFPGAOCLSDKROOT/include`
* On Windows: `-I %INTELFPGAOCLSDKROOT%\include`

For memory-mapped host interfaces, the testbench (or host program) must allocate the USM memory using the `sycl::malloc_shared` or `sycl::malloc_host` function that specifies the buffer location as a property. The function should be passed the buffer location property as an argument if a buffer location has been specified using the macros on the kernel argument in the functor.

For IP component kernels, allocating and using USM device memory with the `sycl::malloc_device` API is not supported.

The following macros create a memory-mapped host interface:

* mmhost() macro
* register_map_mmhost() macro
* conduit_mmhost() macro

The `mmhost()` macro (or no macro specified)

If no macro is specified for the pointer kernel argument or if the `mmhost()` macro is specified, the argument inherits the same style of interface as the kernel invocation interface.

The default for kernel invocation interfaces is a register map-based interface.

To override this argument behavior, use the `register_map_mmhost()` and `conduit_mmhost()` macros.

The macros that control the kernel invocation interface are described in Agent IP Component Kernels and Streaming IP Component Kernels.

```
mmhost(
  1,       // buffer location
  28,      // address width
  64,      // data width
  0,       // latency. Setting 0 specifies variable  latency interface
  0,       // read_write_mode, 0: Read/Write, 1:  Read-only, 2: Write-
only
  1,       // maxburst
  0,       // align
  1        // waitrequest, 0: false, 1: true
  ) int *x;
```

The `register_map_mmhost()` macro

The base pointer is passed in through the register map.

When you use `register_map_mmhost()` macro, only the address width number of bits are consumed by the kernel even though a 64-bit wide register is created to store the address.

```
register_map_mmhost(
  1,       // buffer location
  28,      // address width
  64,      // data width
  0,       // latency. Setting 0 specifies variable  latency interface
  0,       // read_write_mode, 0: Read/Write, 1:  Read-only, 2: Write-
only
  1,       // maxburst
  0,       // align
  1        // waitrequest, 0: false, 1: true
  ) int *x;
```

The `conduit_mmhost()` macro

The base pointer passed in the through a conduit interface.

When you use the `conduit_mmhost()`, the port created for the pointer argument is sized to the address width that is specified for it.

```
conduit_mmhost(
  1,       // buffer location
  28,      // address width
  64,      // data width
  0,       // latency. Setting 0 specifies variable  latency interface
  0,       // read_write_mode, 0: Read/Write, 1:  Read-only, 2: Write-
only
  1,       // maxburst
  0,       // align
  1        // waitrequest, 0: false, 1: true
 ) int *x;
```

As an example, the following kernels have register-map-based argument interfaces:

```
// This struct defines the IP that will be generated
struct MyIPComponent1{
  // struct members are kernel arguments
  int* a; // no macro specified

  // operator()() defines the device/IP code
  // no macro specified for the operator()()
  void operator()() const { ... }
};

struct MyIPComponent2{
  mmhost(...) int* a;
  ...
  // no kernel invocation macros specified for the operator()()
  void operator()() const { ... }
};
```

You can customize the following properties:

MM Host Macro Properties :header-rows: 1 :class: longtable

Property	Description	Default	Valid Values
Buffer location	A literal that specifies a unique identifier for an external memory. It must be a compile-time constant value.	N/A	See description.
	Buffer locations must be sequential integers starting from `0` unless there is an unannotated pointer kernel argument defined for any kernel.		
	If there is an unannotated pointer kernel argument, then the buffer location must start at `1` because `0` is reserved for the external memory that is inferred whenever there is an unannotated pointer kernel argument present.		
	The total number of distinct buffer locations in the entire design (across all kernels) must be less than 64.		
Address width	Width of the memory-mapped address bus, in bits.	41	Integer value in the range 1–41

	If the address width exceeds the maximum valid value, the compile issues a compile-time error.		
Data width	Width of the memory-mapped data bus, in bits.	64	8, 16, 32, 64, 128, 256, 512, 1024
Latency	The guaranteed latency from when a read command exits the component until the external memory returns valid read data. If this latency is variable (such as when accessing DRAM), not known at compile time, or if you are accessing a shared agent interface, set it to 0.	1	Non-negative integer value
Read/Write Mode	The port direction of the interface.	0	0 (Read/write) 1 (Read-only) 2 (Write-only)
Maxburst	The maximum number of data transfers that can be associated with a read or write transaction. This value controls the width of the `burstcount` signal. For fixed latency interfaces, this value must be set to 1. For more details, review information about burst signals and the `burstcount` signal role in "Avalon® Memory-Mapped Interface Signal Roles" in Avalon® Interface Specifications.	1	1–1024
Alignment	The alignment of the argument pointer address in bytes.	1	0, 1, 2, 4, 8, 16, 32, 64, 128

Important You must ensure that all the values that the pointer takes are divisible by the specified alignment otherwise functional failures might occur.

The alignment setting allows the compiler to generate optimized hardware that can issue wider loads/stores by combining multiple loads/stores. For example, if you want to transact four 32-bit integers, set the data width to 128 bits and the alignment to 16 bytes. This means that up to 16 contiguous bytes (or four 32-bit integers) can be loaded or stored as a coalesced memory word per clock cycle.

Specifying an alignment value of `0` is the same as specifying an alignment value of `1`.

Waitrequest	Directive for adding the `waitrequest` signal that is asserted by the agent when it is unable to respond to a read or write request. For more information about the `waitrequest` signal, refer to "Avalon® Memory-Mapped Interface Signal Roles" in Avalon® Interface Specifications. **Important** Do not specify waitrequest to 1 when specifying a fixed latency interface (Latency=0).	0	0: Disable `waitrequest` signal 1: Enable `waitrequest` signal.

The following code example creates two customized memory-mapped host interfaces. The host program allocates two USM shared pointers and initializes the memory that they point to with values 5 and 6, respectively. It enqueues the kernel for execution with these pointers as kernel arguments, checks the returned data, and then frees the USM allocated memory.

```cpp
#include <sycl/sycl.hpp>
#include <sycl/ext/intel/fpga_extensions.hpp>
#include <sycl/ext/intel/prototype/interfaces.hpp>

using namespace sycl;
using ext::intel::experimental::property::usm::buffer_location;

constexpr int BL1 = 0;
constexpr int BL2 = 1;

struct MyIP {
  register_map_mmhost(
    BL1,      // buffer location
    28,       // address width
    64,       // data width
    0,        // latency. Setting 0 specifies variable latency interface
    0,        // read_write_mode, 0: Read/Write, 1: Read-only, 2: Write-only
    1,        // maxburst
    0,        // align
    1         // waitrequest, 0: false, 1: true
  ) int *x;
  register_map_mmhost(
    BL2,      // buffer location
    28,       // address width
    64,       // data width
    0,        // latency. Setting 0 specifies variable latency interface
    0,        // read_write_mode, 0: Read/Write, 1: Read-only, 2: Write-only
    1,        // maxburst
    0,        // align
    1         // waitrequest, 0: false, 1: true
  ) int *y;

  register_map_interface
  void operator()() const {
    *x = 5;
    *y = 6;
  }
};

void Test() {

#if FPGA_SIMULATOR
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
#elif FPGA_HARDWARE
    auto selector = sycl::ext::intel::fpga_selector_v;
#else  // #if FPGA_EMULATOR
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
#endif

  sycl::queue q(selector);
  int *HostA = malloc_shared<int>(1, q, property_list{buffer_location(BL1)});
  *HostA = 0;
  int *HostB = malloc_shared<int>(1, q, property_list{buffer_location(BL2)});
  *HostB = 0;
```

```
   q.single_task(MyIP{HostA, HostB}).wait();

   if (*HostA == 6 && *HostB == 5) std::cout << "PASSED\n";
   else std::cout << "FAILED\n";

   sycl::free(HostA, q);
   sycl::free(HostB, q);
}

int main() {

   Test();

   if (*HostA == 6 && *HostB == 5) std::cout << "PASSED\n";
   else std::cout << "FAILED\n";

   return 0;
}
```

Memory-Mapped Interface Unified-Shared-Memory Virtual Address Space

The compiler encodes certain information regarding the virtual address space in the top bits of a 64-bit pointer address as follows:

Pointer-Address Bit-Range Descriptions

Bit Range	Description
40:0	Used for addressing within the memory system
63:41	Stores the virtual address space information that is derived from the buffer location

In some cases, the compiler cannot determine which buffer location a pointer corresponds to and it creates logic in the generated RTL that inspects the top bits of the pointer at runtime to detect the buffer location and route the memory transaction to the correct external memory interface.

You do not need to encode the buffer location information yourself in most cases. Exceptions are outlined in Manual Buffer Location Encoding Use Case.

The compiler automatically generates logic to embed this information from the buffer location specified on the pointer kernel argument in the source file.

Manual Buffer Location Encoding Use Case

If the kernel has at least one kernel argument where the buffer location is specified (annotated argument) and at least one argument where the buffer location is not specified (unannotated argument), then you must embed the buffer location information in the top bits of any unannotated kernel arguments.

Consider the following code example:

```
// This struct defines the IP that will be generated
struct MyIPComponent{
  // struct members are kernel arguments
  int* a; // no buffer location specified
  mmhost(1, ...) int *b; // buffer location 1
  mmhost(2, ...) int *c; // buffer location 2
  mmhost(3, ...) int *d; // buffer location 3

  // operator()() defines the device/IP code
```

```
  void operator()() const {
    *a *= 2;
    *b *= 2;
    *c *= 2;
    *d *= 2;
  }
};
```

In this example, the compiler does not know what external memory pointer `a` will point to, so the compiler creates logic to check the top bits of the pointer to determine, at runtime, which buffer location to access. Therefore, you must set those top bits for argument `a` (but not for the other kernel arguments).

In such cases, if the unannotated pointer argument has a conduit interface then the port is 64 bits wide. And, if the interface is register-map–based, all 64 bits are passed to the kernel.

> **NOTE**
> **Simulation Exception**
> When you simulate your kernel in the OneAPI simulation flow, you do not need to write any host code (even in this use case) to embed information in the pointer bits. The buffer locations are all taken care of by the runtime stack that allocates the pointers in the host code.

When the compiler can deduce which buffer location that a pointer argument points to (for example, when there is only one `mm_host` interface), the compiler embeds the buffer location automatically. Buffer locations need to be specified manually only when your kernels have a mix of annotated and unannotated kernel arguments.

The compiler infers one global memory (with inferred buffer location `0`) whenever there is any unannotated pointer kernel argument. In the following code example, because there are only unannotated pointer arguments, the compiler infers only one global memory and so it can embed the correct information in the top bits of the pointer kernel arguments.

```
// This struct defines the IP that will be generated
struct MyIPComponent{
    // struct members are kernel arguments
    int* a; // no buffer location specified
    int* b; // no buffer location specified
    // no other annotated kernel argument is present

    // operator()() defines the device/IP code
    void operator()() const {
        *a = ...
        *b = ...
    }
};
```

Determining Virtual Address Space Information

If you need to embed the virtual address space information in the top bits of the pointer kernel arguments because the compiler cannot do so, get the information to embed from the HTML reports:

1. If you have not already done so, compile your kernel to obtain the HTML reports
2. Open the HTML reports.
3. Go to **Views > System Viewer**
4. In the left pane, expand **System** and then expand **Global memory**

   Under **Global memory**, you see entries for all external memories for your kernel.
5. Click on a memory to display that memory in the **System Viewer** pane.

6. In the **System Viewer** pane, find the box that represents the memory and click the node inside the box.

This node represents the "interface" of that global memory.

7. In the **Details** pane, find the **Start Address** of the memory.

The top bits of your unannotated pointer argument must match the top bits of this start address if you want the pointer to access this buffer location.

The following screen capture shows an example of determining the top bit of the start address needed to address a buffer location `1`.

Memory-Mapped Host Interfaces Using Accessors

The following example shows how to create multiple memory-mapped (`mm_host`) interfaces using the `buffer_location` property in SYCL*:

```
#include <sycl/sycl.hpp>
#include <iostream>
#include <sycl/ext/intel/fpga_extensions.hpp>
#include <vector>
```

```
using namespace sycl;

// Forward declare the kernel name in the global scope.
// This is an FPGA best practice that reduces name mangling in the
// optimization reports.
class SimpleVAdd;


// The members of the functor serve as inputs and outputs to your IP.
// The code inside the operator()() function describes your IP.
template <class AccA, class AccB, class AccC>
struct SimpleVAddKernel {
    AccA A;
    AccB B;
    AccC C;
    int count;

    void operator()() const {
        for (int i = 0; i < count; i++) {
            C[i] = A[i] + B[i];
        }
    }
};

constexpr int VECT_SIZE = 4;

int main() {

  #if FPGA_SIMULATOR
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
  #elif FPGA_HARDWARE
    auto selector = sycl::ext::intel::fpga_selector_v;
  #else  // #if FPGA_EMULATOR
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
  #endif

    queue q(selector);

    int count = VECT_SIZE; // pass array size by value

    // declare arrays and fill them
    std::vector<int> VA;
    std::vector<int> VB;
    std::vector<int> VC(count);
    for (int i = 0; i < count; i++) {
        VA.push_back(i);
        VB.push_back(count - i);
    }

    std::cout << "add two vectors of size " << count << std::endl;

    buffer bufferA{VA};
    buffer bufferB{VB};
    buffer bufferC{VC};

    q.submit([&](handler &h) {
        accessor accessorA{bufferA, h, read_only};
```

```
        accessor accessorB{bufferB, h, read_only};
        accessor accessorC{bufferC, h, read_write, no_init};

        h.single_task<SimpleVAdd>(SimpleVAddKernel<decltype(accessorA), decltype(accessorB),
decltype(accessorC)>{accessorA, accessorB, accessorC, count});
    });

    // verify that VC is correct
    bool passed = true;
    for (int i = 0; i < count; i++) {
        int expected = VA[i] + VB[i];
        std::cout << "idx=" << i << ": result " << VC[i] << ", expected (" << expected << ")
VA=" << VA[i] << " + VB=" << VB[i] << std::endl;
        if (VC[i] != expected) {
            passed = false;
        }
    }

    std::cout << (passed ? "PASSED" : "FAILED") << std::endl;
    return passed ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Host Pipes

Pipes are a first-in first-out (FIFO) buffer construct that provide links between elements of a design. Pipes that connect a host and a device are referred to as *host pipes*. Host pipe support is enabled by including the following include statement in your design:

```
#include <sycl/ext/intel/experimental/pipes.hpp>
```

For information about declaring and using host pipes, refer to Host Pipes in *FPGA Optimization Guide for Intel® oneAPI Toolkits*.

> **Important** For multiarchitecture binary kernels (sometimes referred to as "fat binaries" or "full stack"), the number of non-CSR host pipes in your design is limited by your BSP.

Agent IP Component Kernels

SYCL* kernels generate an interface that can control the kernel and pass in the kernel arguments to the IP component.

By default, the Intel® oneAPI DPC++/C++ Compiler generates an Avalon agent interface to control the kernel and pass in the kernel arguments. The compiler also generates a header file that provides the addresses of various registers in the agent memory map. A top-level header named `register_map_offsets.hpp` is created for each device image that you can include if you are interfacing with the SYCL* device image.

An additional header is generated for each of your kernels within the `.prj` directory. The `register_map_offsets.hpp` header file includes these files, but contain the addresses and offsets for each of the kernels.

Example Register Map File

```
/* Status register contains all the control bits to control kernel execution */
/********************************************************************************/
/* Memory Map Summary                                                        */
/********************************************************************************/
```

```
/*
Address | Access | Register    | Argument              | Description
--------|--------|-------------|-----------------------|-------------------------------
   0x0  |  R/W   |  reg0[63:0] |          Status[63:0] |  * Read/Write the status bits
        |        |             |                       |          that are described below
--------|--------|-------------|-----------------------|-------------------------------
   0x8  |  R/W   |  reg1[31:0] |           Start[31:0] |        * Write 1 to initiate a
        |        |             |                       |                    kernel start
--------|--------|-------------|-----------------------|-------------------------------
  0x30  |    R   |  reg6[31:0] |  FinishCounter[31:0]  | * Read to get number of kernel
        |        |  reg6[63:32]|  FinishCounter[31:0]  |          finishes, note that this
        |        |             |                       |          register will clear on read
--------|--------|-------------|-----------------------|-------------------------------
  0x80  |    W   | reg16[63:0] |    arg_input_a[63:0]  |
--------|--------|-------------|-----------------------|-------------------------------
  0x88  |  R/W   | reg17[63:0] |    arg_input_b[63:0]  |
--------|--------|-------------|-----------------------|-------------------------------
  0x90  |  R/W   | reg18[63:0] |    arg_input_c[63:0]  |
--------|--------|-------------|-----------------------|-------------------------------
  0x98  |  R/W   | reg19[31:0] |           arg_n[31:0] |
*/


/*************************************************************************/
/* Register Address Macros                                               */
/*************************************************************************/

/* Status Register Bit Offsets (Bits) */
/* Note: Bits In Status Registers Are Marked As Read-Only or Read-Write
   Please Do Not Write To Read-Only Bits */
#define KERNEL_REGISTER_MAP_DONE_OFFSET (1) // Read-only
#define KERNEL_REGISTER_MAP_BUSY_OFFSET (2) // Read-only
#define KERNEL_REGISTER_MAP_STALLED_OFFSET (3) // Read-only
#define KERNEL_REGISTER_MAP_UNSTALL_OFFSET (4) // Read-write
#define KERNEL_REGISTER_MAP_VALID_IN_OFFSET (14) // Read-only
#define KERNEL_REGISTER_MAP_STARTED_OFFSET (15) // Read-only

/* Status Register Bit Masks (Bits) */
#define KERNEL_REGISTER_MAP_DONE_MASK (0x2)
#define KERNEL_REGISTER_MAP_BUSY_MASK (0x4)
#define KERNEL_REGISTER_MAP_STALLED_MASK (0x8)
#define KERNEL_REGISTER_MAP_UNSTALL_MASK (0x10)
#define KERNEL_REGISTER_MAP_VALID_IN_MASK (0x4000)
#define KERNEL_REGISTER_MAP_STARTED_MASK (0x8000)
```

While the default option for kernels are agent kernels, there is a `register_map_interface` macro to mark a function as an agent kernel. This is shown in the following example:

```
#include <sycl/ext/intel/prototype/interfaces.hpp>
using namespace sycl;

struct MyIP {
  int *input_a, *input_b, *input_c;
  int n;

  register_map_interface void operator()() const {
    for (int i = 0; i < n; i++) {
```

```
      input_c[i] = input_a[i] + input_b[i];
    }
  }
};
```

Streaming IP Component Kernels

You can also choose to have the Intel® oneAPI DPC++/C++ Compiler implement the IP component invocation interface with a "ready-valid" handshake, like in an Avalon Streaming (ST) interface.

To have the compiler implement the IP component invocation interface with a "ready-valid" handshake:

1. Implement the IP kernel as a functor.
2. Include the following header file:

   `sycl/ext/intel/prototype/interfaces.hpp`
3. Add one of the following options to the compiler command (`icpx -fsycl`):

   - Linux: `-I/$INTELFPGAOCLSDKROOT/include`
   - Windows: `/I %INTELFPGAOCLSDKROOT%\include`
4. Add the `streaming_interface` macro to the functor `operator()`.

The following code shows an example of implementing a streaming interface:

```
#include <sycl/ext/intel/prototype/interfaces.hpp>
using namespace sycl;

struct MyIP {
  int *input_a, *input_b, *input_c;
  int n;
  MyIP(int *a, int *b, int *c, int N_)
      : input_a(a), input_b(b), input_c(c), n(N_) {}
  streaming_interface void operator()() const {
    for (int i = 0; i < n; i++) {
      input_c[i] = input_a[i] + input_b[i];
    }
  }
};
```

The resulting IP component kernel is invoked with a "ready-valid" handshake. Compiling the example code generates the `start` signal, the `done` signal, the `ready_in` signal, and `ready_out` signals as conduits. The compilation of the example code also generates conduits for the base addresses of the three pointers as well the value of `N`.

The streaming handshaking follows the Avalon Streaming (ST) protocol. The IP kernel consumes the arguments on the clock cycle that the `start` and `ready_out` signals are asserted. The IP component kernel invocation is finished on the clock cycle that the `done` and `ready_in` signals are asserted.

Limitations of Streaming IP Component Kernels

The following actions are not supported when using a streaming IP component kernel:

- Using streaming kernels as SYCL NDRange kernels.
- Profiling of streaming kernels.
- Using agent kernel arguments in streaming kernels.

Kernel Argument Interfaces

By default, the kernel arguments are passed to your component through the same interface as the `start` signal (that is, either through the IP component CSR or through inputs synchronized to a ready/valid handshake). However, you can override this behavior. For example, you can select a register-mapped invocation interface with arguments passed through conduits, or you can select a streaming invocation interface with arguments passed through the register map.

The following code snippet demonstrates how to place the invocation interface in the CSR, and pass kernel arguments through conduit interfaces:

```
#include <sycl/ext/intel/prototype/interfaces.hpp>
using namespace sycl;

struct MyIP {
  conduit int *input_a, *input_b, *input_c;
  conduit int n;
  register_map_interface void operator()() const {
    for (int i = 0; i < n; i++) {
      input_c[i] = input_a[i] + input_b[i];
    }
  }
};
```

The following code snippet demonstrates how to configure the kernel with a handshake invocation interface, and pass kernel arguments through the CSR:

```
#include <sycl/ext/intel/prototype/interfaces.hpp>
using namespace sycl;

struct MyIP {
  register_map int *input_a, *input_b, *input_c;
  register_map int n;
  streaming_interface void operator()() const {
    for (int i = 0; i < n; i++) {
      input_c[i] = input_a[i] + input_b[i];
    }
  }
};
```

Pipelined Kernels

By default, SYCL* task kernels are not pipelined. You must wait for the previous invocation to finish before invoking the kernel again.

However, streaming kernels can be optionally pipelined by using the `streaming_pipelined_interface macro`, as shown in the following example:

```
struct MyIP {
  conduit int *input;
  MyIP(int *inp_a_) : input(inp_a_) {}
  streaming_pipelined_interface void operator()() const {
    int temp = *input;
    *input = something_complicated(temp);
  }
};
/* To exercise the pipelined nature of the kernel in simulation,
   you must queue up multiple instances of the functions before you
   call the wait() function. The following code example shows how to
   exercise a pipelined kernel: */
for (int i = 0; i < kN; i++) {
```

```
   q.single_task(MyIP{&input_array[i]});
}
q.wait();
```

Stable Arguments

By default, the Intel® oneAPI DPC++/C++ Compiler assumes that the values of kernel arguments change during kernel executions.

For pipelined kernels, you can mark a kernel argument as stable if the kernel and kernel argument satisfy the following conditions:

- The kernel argument does not change during the execution of any invocation of this kernel.
- The kernel is not launched with a different kernel argument value when a different invocation of this kernel is still running.

Declare a streaming (conduit) kernel argument to be stable with the `stable_conduit` attribute.

Changing the value of a stable kernel argument results in undefined behavior.

You might save some FPGA area in your kernel design when you declare a streaming (conduit) kernel argument as stable.

If all the kernel arguments do not change while the kernel is executing, you can include the `-Xsno-hardware-kernel-invocation-queue` option in your `icpx` command.

Changing the value of a kernel argument on a kernel compiled with the `-Xsno-hardware-kernel-invocation-queue` option results in undefined behavior.

IP Component Reset Behavior

For your IP component, the reset assertion can be asynchronous but the reset deassertion must be synchronous.

The reset assertion and deassertion behavior can be generated from an asynchronous reset input by using reset synchronization. Add reset synchronization to your component with Platform Designer when you integrate your IP into a system.

For information about integrating your IP into a system, refer to Integrating Your IP Into a System.

For an example of adding reset synchronization, refer to the Platform Designer Sample example design.

Synchronizers are implemented to minimize synchronization failures due to metastability in asynchronous signal transfers. For more information about metastability, refer to "Managing Metastability with the Intel Quartus Prime Software" in Intel Quartus Prime Pro Edition User Guide: Design Recommendations.

When the reset is asserted, the component holds its busy signal high and its done signal low. After the reset is deasserted, the component holds its busy signal high until the component is ready to accept the next invocation. All component interfaces (agents, hosts, and streams) are ready only after the component busy signal is low.

Example Waveform Showing IP Component Reset Behavior
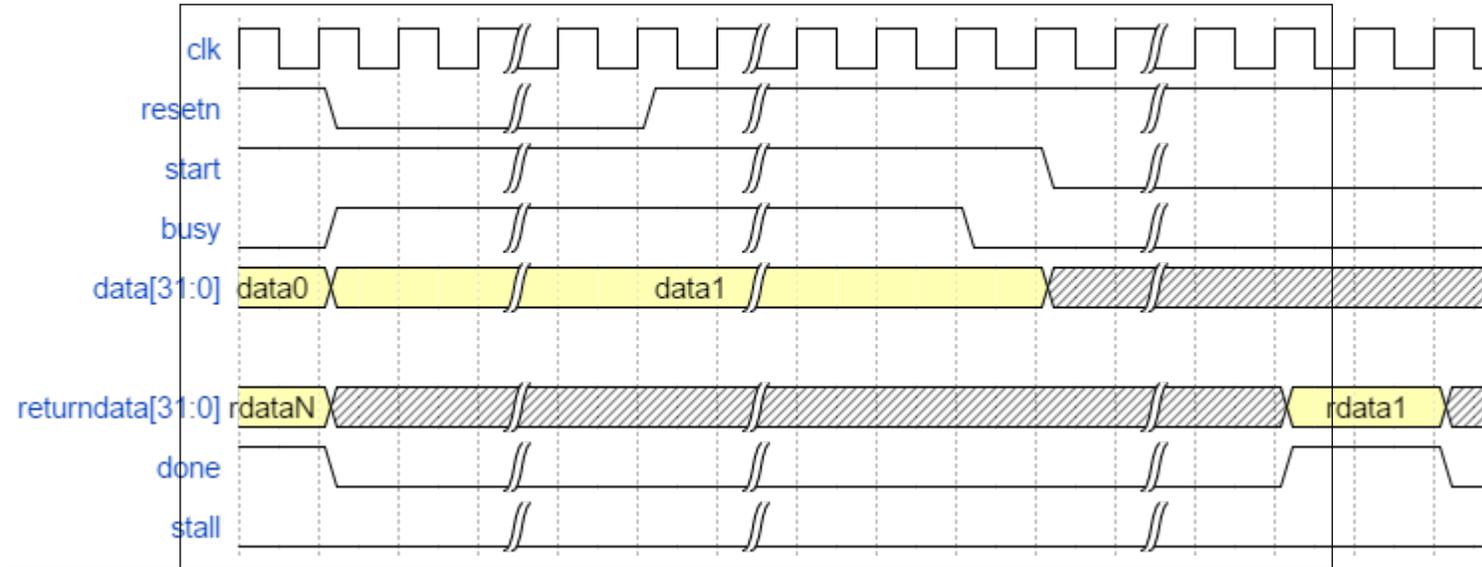
The printf Command

The `sycl::oneapi::experimental::printf()` function is currently not supported in IP components.

Emulate and Debug Your IP Component

Verify the functionality of your design by compiling your component and testbench to an x86-64 FPGA emulation executable that you can debug with a oneAPI debugger. This process is sometimes referred to as debugging through emulation.

Compiling your design to an x86-64 executable is faster than generating and simulating RTL. Shorter compilation time allows you to debug and refine your component quickly before verifying how your component is implemented in hardware.

No additional software is required to emulate your IP component, and no modifications to your host code are required.

For details, refer to Emulate and Debug Your Design.

Evaluate Your IP Component Through Simulation

When you compile your component to an Intel® FPGA device family or part number with the `-Xstarget` compiler option, the Intel oneAPI DPC++/C++ Compiler links your design C++ testbench with an RTL-compiled version of your component that runs in an RTL simulator.

Use Siemens® EDA Questa® software to perform the simulation. You must have Questa® simulation software installed when authoring IP components with the Intel oneAPI Base Toolkit.

Verifying the functionality of your design in this way is sometimes called debugging through simulation.

To verify the design functionality from your design simulation, use the following debugging techniques:

- Run the executable that the compiler generates by targeting the FPGA device.
- Write variable values to output pipes or `mm_host` interfaces at certain points in your code.
- Review the waveforms generated when running your design.

  The compiler does not log signals by default when you compile your design. To enable signal logging in simulation, refer to Debug During Verification.

For more information about simulation, refer to Evaluate Your Kernel Through Simulation.

- Debug During Verification

Debug During Verification

By default, the compiler instructs the simulator not to log any signals because logging signals slows the simulation, and waveform files can be extremely large. However, you can configure the compiler to save these waveforms for debugging purposes.

To enable signal logging in the simulator, invoke the `icpx -fsycl` command with the `-Xsghdl` option command as follows:

```
icpx -fsycl -fintelfpga -Xssimulation -Xstarget=<family_or_part_number> -Xsghdl <input files>
```

> **NOTE** After you compile your component and testbench with the `-Xsghdl` option, run the resulting executable to run the simulation and generate the waveform.

When the simulation finishes, open the `vsim.wlf` file inside the current directory to view the waveform.

To view the waveform after the simulation finishes:

1. In the Questa® simulator, open the `vsim.wlf` file inside the `<project name>.prj` directory.
2. Right-click the **<IP_component_name>_inst** block and select Add Wave.

   You can now view the top-level component signals: start, done, ready_in, ready_out, parameters, and outputs. Use the waveform to see how the component interacts with its interfaces.

> **Tip** When you view the simulation waveform in the Questa® simulator, the simulation clock period is set to a default value of 1000 picoseconds (ps). To synchronize the Time axis to show one cycle per tick mark, change the time resolution from picoseconds (ps) to nanoseconds (ns):
>
> 1. Right-click the timeline and select **Grid, Timeline & Cursor Control**.
> 2. Under **Timeline Configuration**, set the **Time** units to `ns`.

FPGA IP Component Performance Optimization

The Intel® oneAPI DPC++/C++ Compiler provides tools that you can use to find areas for improvement and a variety of flags, attributes, and extensions to control design and compiler behavior.

For more information about optimizing your design, refer to the FPGA Optimization Guide for Intel® oneAPI Toolkits.

Synthesizing Your Component IP with Intel® Quartus® Prime Software

When you are satisfied with the predicted performance of your component, use Intel® Quartus® Prime software to synthesize your component. Synthesis also generates accurate area and performance (f_{MAX}) estimates for your design. However, your design is not expected to cleanly close timing in the Intel® Quartus® Prime reports.

You can expect to see timing closure warnings in the Intel® Quartus® Prime logs because the generated project targets a clock speed of 1000 MHz to achieve the best possible placement for your design. The f_{MAX} value presented in the FPGA optimization report estimates the maximum clock rate your component can cleanly close timing for.

After the Intel® Quartus® Prime compilation is completed, the summary section of the FPGA optimization report shows the area and performance data for your components. These estimates are more accurate than estimates generated when you compile your IP component for simulation only.

Typically, Intel® Quartus® Prime compilation times can take minutes to hours, depending on the size and complexity of your IP components.

To synthesize your component IP and generate quality of results (QoR) data, instruct the compiler to run the Intel® Quartus® Prime compilation flow automatically after synthesizing the components. Include the **–Xshardware** option in your **icpx -fsycl** command:

```
icpx -fsycl -fintelfpga -Xshardware -Xstarget="<FPGA device family or part number>"...
```

Integrating Your IP Into a System

To integrate your IP component into a system with the Intel® Quartus® Prime software, you must be familiar with Intel® Quartus® Prime software, including Platform Designer.

The Intel® oneAPI DPC++/C++ Compiler generates a project directory (`<result>.prj/`) and a set of IP files per IP component (a set of kernels that are part of the same system). You can control this with the `-fsycl-device-code-split=<off|per_source|per_kernel>` option.

The `<result>.prj/` directory generated by the compiler contains all the files that you need to include your IP component in an Intel® Quartus® Prime project, including the following files:

- `<project_name>_di.ip`

  An ip format file that you can add to your Intel Quartus Prime projects.
- `<project_name>_di_hw.tcl`

  A script that describes your IP component interfaces for Platform Designer.
- `<project_name>_di_inst.v`

  An example of how to instantiate the IP into other Verilog modules.

- Adding IP into an Intel® Quartus® Prime Project
- Adding IP into a Platform Designer System

Adding IP into an Intel® Quartus® Prime Project

To use the IP component generated by the Intel® oneAPI DPC++/C++ compiler in an Intel® Quartus® Prime project, you must first add the `.ip` file to the project.

The `.ip` file contains information to add to all the necessary HDL files for the component. It also applies to any component-specific Intel® Quartus® Prime Settings File (`.qsf`) settings that are necessary for IP synthesis.

Follow these steps:

1. Create an Intel Quartus Prime Pro Edition project.
2. Open the Platform Designer and select your IP from the oneAPI folder.

   For your IP to be in the oneAPI folder, either create the project in the same directory that contains the generated IP project or add the file path.
3. Create the rest of your Intel Quartus Prime software project.

   For an example of how to instantiate the IP component top-level module, examine the `<result>.prj/<project_name>_di_inst.v file.`

Adding IP into a Platform Designer System

To use the IP component generated by the Intel® oneAPI DPC++/C++ compiler in a Platform Designer system, you must first add the directory to the IP search path or the IP Catalog.

In Platform Designer, if your IP component does not appear in the IP Catalog, perform the following tasks:

1. In the Intel® Quartus® Prime software, click **Tools > Options**.
2. In the **Options** dialog box, under **Category**, expand **IP Settings** and click **IP Catalog Search Locations**.
3. In the **IP Catalog Search Locations** dialog box, add the path to the directory that contains the `_hw.tcl file` to IP Search Paths as `<result>.prj/<project_name>`.
4. In the **IP Catalog**, add your IP to the Platform Designer system by selecting it from the oneAPI project directory.

For more information about Platform Designer, refer to Creating a System with Platform Designer in *Intel® Quartus® Prime Pro Edition User Guide: Platform Designer*.

For an example of adding an IP component into a system with Platform Designer, see the Platform Designer Sample tutorial.

Encrypt IP Components for Distribution

If you are a member of the Intel® FPGA Design Solutions Network, you have access to tools to encrypt your IP design files and generate a license for it. Your IP users can use the encrypted IP only in ways specified by the generated license.

This license is compatible with the FlexLM licensing technology used by Intel® Quartus® Prime software.

If you have the Intel-provided IP encryption and licensing infrastructure installed, you can also generate encrypted IP with the Intel® oneAPI DCP++/C++ Compiler.

Your encrypted IP can then be used by your customers in Intel® Quartus® Prime software, licensed by the file that your users added to their Intel® Quartus® Prime license search path. For more details, refer to the documentation that Intel provided you when you joined the Intel® FPGA Design Solutions Network.

If you want to support simulation with your encrypted IP, you must create a separately-encrypted version of your IP for simulation. For simulation, an IEEE 1735 compliant encryption scheme is used.

To generate encrypted IP for use in Intel® Quartus® Prime software, use the following command:

```
icpx -fsycl -fintelfpga -Xshardware -Xstarget=<FPGA device or part number> -Xsencryption-key=<key> -Xsencryption-id=<product_id> -Xsencryption-release-date=<yyyy.mm>
```

> **Important** Before you run this command, you must create a license file for the IP and add the license file to your `$LM_LICENSE_FILE` environment variable.

To generate encrypted IP for use in simulation, use the following command:

```
icpx -fsycl -fintelfpga -Xssimulation -Xstarget=<FPGA device or part number> -DFPGA_SIMULATOR -I/
$INTELFPGAOCLSDKROOT/include -Xsencryption-key=<key> -Xsencryption-id=<product_id> -Xsencryption-release-date=<yyyy.mm>
```

FPGA Compilation Flags for IP Encryption

Option name	Description
`-Xsencryptionkey`	Specifies the encryption key used to encrypt the source file.
	The key must be a 48-digit hexadecimal value.
`-Xsencryption-id`	Specifies the product ID for the IP.

Option name	Description
	This ID must be a 4-digit hexadecimal value.
`-Xsencryption-release-date`	Sets the release date in the format `yyyy.mm`.
`-Xsno-encryption`	If you have created an alias to your `icpx -fsycl` command that encrypts your IP, use this option on your alias command to temporarily disable encryption.

Fast Recompile for FPGA

The Intel® oneAPI DPC++/C++ Compiler supports only the ahead-of-time (AoT) compilation for FPGA hardware and simulation, which means that an FPGA device image is generated at compile time. The FPGA device image generation process can take hours to complete. If you make a change exclusive to the host code, then recompile only your host code by reusing the existing FPGA device image and circumventing the time-consuming device compilation process. Compiling for the simulator is much faster, but you can save time by using the `-reuse-exe` flag.

The Intel® oneAPI DPC++/C++ Compiler provides the following mechanisms to separate device code and host code compilation:

- Passing the `-reuse-exe=<exe_name>` flag to instruct the compiler to attempt to reuse the existing FPGA device image.
- Separating the host and device code into separate files. When a code change applies only to host-only files, the FPGA device image is not regenerated.
- Separating the device code using the compiler option `-fsycl-device-code-split`.

The following sections explain these these mechanisms in detail.

Using the `-reuse-exe` Flag

If the device code and options affecting the device have not changed since the previous compilation, passing the `-reuse-exe=<exe_name>` flag instructs the compiler to extract the compiled FPGA hardware or simulation image from the existing executable and package it into the new executable, saving the device compilation time.

Sample use:

```
# Initial compilation
icpx -fsycl -fintelfpga -Xshardware <files.cpp> -o out.fpga
```

The initial compilation generates an FPGA device image, which takes several hours. Suppose you now make some changes to the host code.

```
# Subsequent recompilation
icpx -fsycl <files.cpp> -o out.fpga -reuse-exe=out.fpga -Xshardware -fintelfpga
```

One of the following actions are taken by the command:

- If the `out.fpga` file does not exist, the `-reuse-exe` flag is ignored, and the FPGA device image is regenerated. This is always the case the first time you compile a project.
- If the `out.fpga` file is found, the compiler verifies no change that affects the FPGA device code is made since the last compilation. If no change is detected in the device code, the compiler then reuses the existing FPGA device image and recompiles only the host code. The recompilation process takes a few minutes to complete.
- If the `out.fpga` file is found, but the compiler cannot prove that the FPGA device code will yield a result identical to the last compilation, a warning is printed, and the FPGA device code is fully recompiled. Since the compiler checks must be conservative, spurious recompilations can sometimes occur when using the `-reuse-exe` flag.

Using the Device Link Method

Suppose the program is separated into two files, `main.cpp` and `kernel.cpp`, where only the `kernel.cpp` file contains the device code.
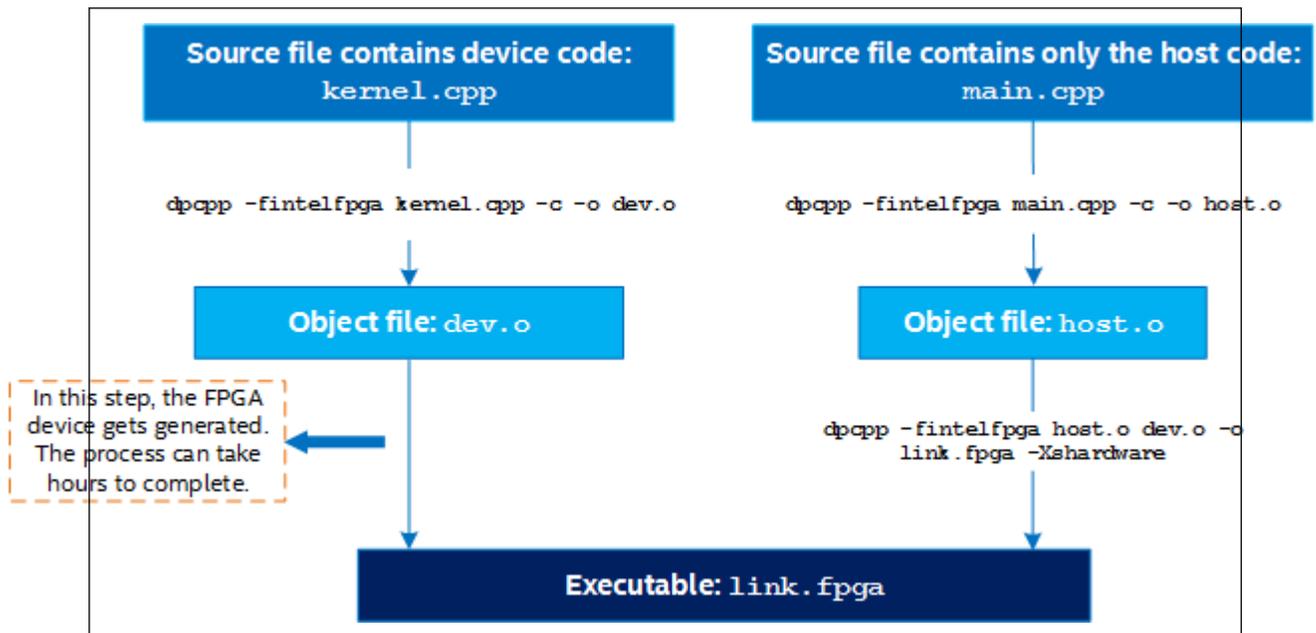
In the normal compilation process, FPGA device image generation happens at link time.

```
# normal compile command
icpx -fsycl -fintelfpga -Xshardware main.cpp kernel.cpp -o link.fpga
```

As a result, any change to either the main.cpp or `kernel.cpp` triggers the regeneration of an FPGA hardware image.

The following graph depicts this compilation process:

Compilation Process

If you want to iterate on the host code and avoid a long compile time for your FPGA device, consider using a device link to separate the device and host compilation:

```
# device link command
icpx -fsycl -fintelfpga -fsycl-link=image <input files> [options]
```

The compilation is a three-step process as listed in the following:

1. Compile the device code.

```
icpx -fsycl -fintelfpga -Xshardware -fsycl-link=image kernel.cpp -o dev_image.a
```

Input files must include all files that contain the device code. This step might take several hours to complete.

2. Compile the host code.

```
icpx -fsycl -fintelfpga main.cpp -c -o host.o
```

Input files should include all source files that contain only the host code. These files must not contain any source code that executes on the device but may contain setup and tear-down code, for example, parsing command-line options and reporting results. This step takes seconds to complete.

3. Create the device link.

```
icpx -fsycl -fintelfpga host.o dev_image.a -o fast_recompile.fpga
```

This step takes seconds to complete. The input should include one or more host object files (`.o`) and exactly one device image file (`.a`). When linking a static library (`.a` file), always include the static library after its use. Otherwise, the library's functions are discarded. For additional information about static library linking, refer to Library order in static linking.

NOTE You only need to perform steps 2 and 3 when modifying host-only files.

The following diagram illustrates the device link process:

FPGA Device Link Process
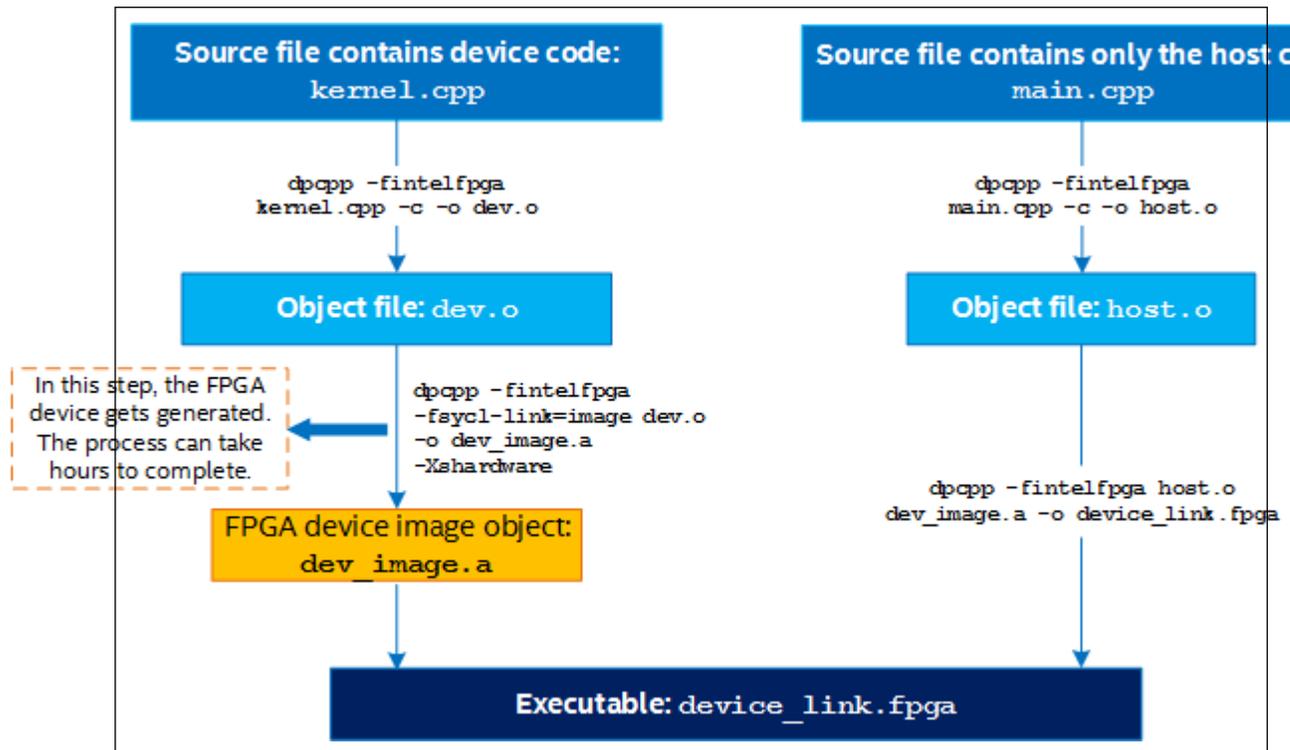
Refer to the fast_recompile tutorial in the Intel® oneAPI Samples Browser for an example using the device link method.

Using the `-fsycl-device-code-split[=value]` Option

The `-fsycl-device-code-split[=value]` option informs the compiler how to separate your design into device code modules. This option supports the following modes:

- `auto`: This is the default mode and the same as the `-fsycl-device-code-split` option without any value. The compiler uses a heuristic to select the best way of splitting device code.
- `off`: Creates a single module for all kernels.
- `per_kernel`: Creates a separate device code module for each kernel. Each device code module contains a kernel and dependencies, such as called functions and user variables.
- `per_source`: Creates a separate device code module for each source (translation unit). Each device code module contains a bunch of kernels grouped on a per-source basis and all their dependencies, such as all used variables and called functions, including the `SYCL_EXTERNAL` macro-marked functions from other translation units.

> **Attention** For FPGA, each split must not share device resources, such as memory, across it. Furthermore, kernel pipes must have their source and sink within the same split.

For additional information about this option, refer to the fsycl-device-code-split topic in *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

Which Mechanism to Use?

Of the mechanisms described above, the `-reuse-exe` flag mechanism is easier to use than the device link mechanism. The flag also allows you to keep your host and device code as a single source, which is preferred for small programs. For larger and more complex projects, the device link method gives you more control over the compiler's behavior.

However, there are some drawbacks of the `-reuse-exe` flag when compared to compiling separate files. Consider the following when using the `-reuse-exe` flag:

- The compiler must spend time partially recompiling and then analyzing the device code to ensure that it is unchanged. This takes several minutes for larger designs. Compiling separate files does not incur this extra time.
- You might occasionally encounter a false positive where the compiler incorrectly believes it must recompile your device code. In a single source file, the device and host code are coupled, so certain changes to the host code can change the compiler's view of the device code. The compiler always behaves conservatively and triggers a full recompilation if it cannot prove that reusing the previous FPGA binary is safe. Compiling separate files eliminates this possibility.

Generate Multiple FPGA Images (Linux only)

Use this feature of the Intel® oneAPI DPC++/C++ Compiler when you want to split your FPGA compilation into different FPGA images. This feature is particularly useful when your design does not fit on a single FPGA. You can use it to split your very large design into multiple smaller images, which you can use to partially reconfigure your FPGA device.

You can split your design using one of the following approaches, each giving you different benefits:

- Dynamic Linking Flow
- Dynamic Loading Flow

Between the two flows, dynamic linking is easier to implement than dynamic loading. However, dynamic linking can require more memory on the host device as all of the device images must be loaded into memory. Dynamic loading addresses these limitations but introduces the need for some extra source-level changes. The following comparison table highlights the differences between the flows:

Dynamic Linking vs. Dynamic Loading Flow

	Dynamic Linking	**Dynamic Loading**
Can dynamically change FPGA Image at runtime?	Yes	Yes
Defining the type and number of FPGA images	At compile time	At runtime
Host-program memory footprint	All FPGA images are stored in memory at runtime.	Only explicitly loaded FPGA images are stored in memory.
Calling host code	Call function in the dynamic library directly.	Explicitly load the dynamic library and functions to call.

Dynamic Linking Flow

This flow allows you to split your design into different source files and map them into a separate FPGA image. Intel® recommends this flow for designs with a small number of FPGA images.

To use this flow, perform the following steps:

1. Split your source code such that for each FPGA image you want, you create a separate `.cpp` file that submits various kernels. Separate the host code into one or more `.cpp` files that can then interface with functions in the kernel files.

Consider that you now have the following three files:

- `main.cpp` containing your host code. For example:

```
// main.cpp
int main() {
  queue queueA;
  add(queueA);
  mul(queueA);
}
```

- `vector_add.cpp` containing a function that submits the `vector_add` kernel. For example:

```
// vector_add.cpp
extern "C"{
  void add(queue queueA) {
    queue.submit(
      // Kernel Code
    );
  }
}
```

- `vector_mul.cpp` containing a function that submits the `vector_mul` kernel. For example:

```
// vector_mul.cpp
extern "C"{
  void mul(queue queueA) {
    queue.submit(
      // Kernel Code
    );
  }
}
```

2. Compile the source files using the following commands:

```
icpx -fsycl -fPIC -fintelfpga -c vector_add.cpp -o vector_add.o
icpx -fsycl -fPIC -fintelfpga -c vector_mul.cpp -o vector_mul.o

// FPGA image compiles take a long time to complete
icpx -fsycl -fPIC -shared -fintelfpga vector_add.o -o vector_add.so -Xshardware -Xstarget=pac_a10
icpx -fsycl -fPIC -shared -fintelfpga vector_mul.o -o vector_mul.so -Xshardware -Xstarget=pac_a10

// Final link step
icpx -fsycl -o main.exe main.cpp vector_add.so vector_mul.so
```

With this flow, the long FPGA compile steps are split into separate commands that you can potentially run on different systems or only when you change the files.

Dynamic Loading Flow

Use this flow to avoid loading all of the different FPGA images into memory at once. Similar to dynamic linking flow, this flow also requires you to split your code. However, for this flow, you must load the .so (shared object) files in the host program. The advantage of this flow is that you can load large FPGA image files dynamically as necessary instead of linking all image files at compile time.

To use this flow, perform the following steps:

1. Split your source code in the same manner as done in step 1 of the dynamic linking flow.

2. Modify the main.cpp file to appear as follows:

```
// main.cpp
#include <dlfcn.h>


int main() {
  queue queueA;
  bool runAdd, runMul;
  // Assuming runAdd and runMul are set dynamically at runtime
  if (runAdd) {
    auto add_lib = dlopen("./vector_add.so", RTLD_NOW);
    auto add = (void (*)(queue))dlsym(add_lib, "add");
    add(queueA);
  }
  if (runMul) {
    auto mul_lib = dlopen("./vector_mul.so", RTLD_NOW);
    auto mul = (void (*)(queue))dlsym(mul_lib, "mul");
    mul(queueA);
  }
}
```

3. Compile the source files using the following commands:

NOTE You do not have to link the .so files at compile time since they are loaded dynamically at runtime.

```
icpx -fsycl -fPIC -fintelfpga -c vector_add.cpp -o vector_add.o
icpx -fsycl -fPIC -fintelfpga -c vector_mul.cpp -o vector_mul.o

// FPGA Image compiles take a long time to complete
icpx -fsycl -fPIC -shared -fintelfpga vector_add.o -o vector_add.so -Xshardware -Xstarget=pac_a10
icpx -fsycl -fPIC -shared -fintelfpga vector_mul.o -o vector_mul.so -Xshardware -Xstarget=pac_a10

icpx -fsycl -o main.exe main.cpp
// Before running the design, add the path containing the .so files to LD_LIBRARY_PATH
// e.g., export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

With this approach, you can arbitrarily load many .so files at runtime. This is useful when you have a large library of FPGA images, and you want to select a subset of files from it.

FPGA BSPs and Boards

As mentioned earlier in Types of FPGA Compilation, generating an FPGA hardware image requires Intel® Quartus® Prime software, to map your design from RTL to the FPGA's primitive hardware resources. For BSPs necessary to compile to FPGA hardware, refer to the Intel® FPGA development flow webpage.

What is a Board?

Like a GPU, an FPGA is an integrated circuit that must be mounted onto a card or a board to interface with a server or a desktop computer. In addition to the FPGA, the board provides memory, power, and thermal management, and physical interfaces to allow the FPGA to communicate with other devices.

What is a BSP?

A BSP consists of software layers and an FPGA hardware scaffold design that makes it possible to target the FPGA through the Intel® oneAPI DPC++/C++ Compiler. The FPGA design generated by the compiler is stitched into the framework provided by the BSP.

What is Board Variant?

A BSP can provide multiple board variants that support different functionality. For example, the `intel_s10sx_pac` BSP contains two variants that differ in their support for Unified Shared Memory (USM). For additional information about USM, refer to the Unified Shared Memory and USM Interfaces topics in the SYCL Reference Documentation.

> **NOTE** A board can be supported by more than one BSP and a BSP might support more than one board variant.

The Intel® FPGA Add-On for oneAPI Base Toolkit provides BSPs for two boards and board variants provided by these BSPs can be selected using the following flags in your `icpx -fsycl` command:

Flags in dpcpp command

Board	BSP	Flag	USM Support
Intel® Programmable Acceleration Card (PAC) with Intel® Arria® 10 GX FPGA	`intel_a10gx_pac`	`-Xstarget=intel_a10gx_pac:pac_a10`	Explicit USM
Intel® FPGA Programmable Acceleration Card (PAC) D5005 (previously known as Intel® PAC with Intel® Stratix® 10 SX FPGA)	`intel_s10sx_pac`	`-Xstarget=intel_s10sx_pac:pac_s10`	Explicit USM
Intel® FPGA Programmable Acceleration Card (PAC) D5005 (previously known as Intel® PAC with Intel® Stratix® 10 SX FPGA)	`intel_s10sx_pac`	`-Xstarget=intel_s10sx_pac:pac_s10_usm`	Explicit USM / Restricted USM

NOTE

- The (part of the Intel® oneAPI Base Toolkit) provides partial BSPs sufficient for generating the FPGA early image and optimization report. In contrast, the Intel® FPGA Add-On for oneAPI Base Toolkit provides full BSPs, which are necessary for generating the FPGA hardware image.
- When running an executable on an FPGA board, you must ensure that you have initialized the FPGA board for the board variant that the executable is targeting. For information about initializing an FPGA board, refer to FPGA Board Initialization.
- For information about FPGA optimizations possible with Restricted USM, refer to Prepinning and Zero-Copy Memory Access topics in the FPGA Optimization Guide for Intel® oneAPI Toolkits.

FPGA Board Initialization

Before you run an executable containing an FPGA hardware image, you must initialize the FPGA board using the following command:

```
aocl initialize <board id> <board variant>
```

where:

FPGA Board initialization parameters

Parameter	Description
`<board_id>`	Board ID obtained from the aocl diagnose command. For example, `acl0`, `acl1`, and so on.
`<board variant>`	Name of the board variant as specified by the `–Xstarget` flag the executable was compiled with. For example, `pac_s10_usm`.

For example, consider that you have a single Intel® Programmable Acceleration Card (PAC) D5005 (previously known as *Intel® Programmable Acceleration Card (PAC) with Intel® Stratix® 10 SX)* on your system, and you compile the executable using the following compiler command:

```
icpx -fsycl -fintelfpga -Xshardware -Xstarget=intel_s10sx_pac:pac_s10_usm kernel.cpp
```

In this case, you must initialize the board using the following command:

```
aocl initialize acl0 pac_s10_usm
```

Once this is complete, you can run the executable without initializing the board again, unless you are doing one of the following:

- Running a SYCL*-compiled workload for the first time after power cycling the host.
- Running a SYCL-compiled workload after running a non-SYCL workload on the FPGA.
- Running a SYCL compiled workload compiled with a different board variant in `–Xstarget` flag.

Obtain FPGA Hardware Image Information

The `aocl binedit` utility allows you to extract the following useful information about the compiled binary:

- Compilation environment details, such as: * Compiler version * Compile command used * Intel® Quartus® Prime software version
- `board_spec.xml` from the BSP used for compiling
- Kernel f_{MAX} (Quartus-compiled f_{MAX})
- BSP and board used for compiling

Syntax

Use the `aocl binedit` utility with the following command:

```
aocl binedit <oneapi_binary> <list/get/print/exists> [<section_name> [output_file]]
```

The following are the list of available actions:

- `list` : Lists all available `<section_name>` in the given binary.
- `print` : Writes contents of the existing named section to the standard output stream for each package file in the binary.
- `get` : Writes contents of the existing named section to the output file.
- `exists` : Verifies if the section exists in the package files in the binary. The non-zero exit code indicates the section does not exist.

For example, if you have a binary compiled in the simulator flow, then the following command outputs **SimulatorDevice**:

```
aocl binedit <oneapi_binary> print .acl.board
```

You can also identify the BSP versions using the following command:

```
aocl binedit <oneapi_binary> print .acl.board_package
```

Targeting Multiple Homogeneous FPGA Devices

The Intel® oneAPI DPC++/C++ Compiler supports targeting multiple homogeneous FPGA devices from a single host CPU. This allows to improve your design's throughput by parallelizing the execution of your program on multiple FPGAs.

Intel® recommends creating a single context with multiple device queues because, with multi-context, buffers at OpenCL layer must be copied between contexts, which introduces overhead and impacts overall performance. However, you can use multi-context if your design is simple and the overhead does not affect the overall performance.

Follow one of the following methods to target multiple FPGA devices:

Create a Single Context with Multiple Device Queues

Perform the following steps to target multiple FPGA devices with a single context:

1. Create a single SYCL* context to encapsulate a collection of FPGA devices of the same platform.

```
context ctxt(deviceList, &m_exception_handler);
```
2. Create a SYCL queue for each FPGA device.

```
std::vector<queue> queueList;
for (unsigned int i = 0; i < ctxt.get_devices().size(); i++) {
  queue newQueue(ctxt, ctxt.get_devices()[i], &m_exception_handler);
  queueList.push_back(newQueue);
}
```
3. Submit either the same or different device codes to all available FPGA devices. If you want to target a subset of all available devices, then you must first perform device selection to filter out unwanted devices.

```
for (unsigned int i = 0; i < queueList.size(); i++) {
  queueList[i].submit([&](handler& cgh) {...});
}
```

Create a Context For Each Device Queue (Multi-Context)

Perform the following steps to target multiple FPGA devices with multiple contexts:

1. Obtain a list of all available FPGA devices. Optionally, you can select a device based on the device member or device property. For device properties such as device name, use the member function `get_info()const` with the desired device property.

```
std::vector<device> deviceList = device::get_devices();
```

2. Create a SYCL queue for each FPGA device.

```
std::vector<queue> queueList;
for (unsigned int i = 0; i < deviceList.size(); i++) {
  queue newQueue(deviceList[i], &m_exception_handler);
  queueList.push_back(newQueue);
}
```

3. Submit either the same or different device codes to all available FPGA devices. If you want to target a subset of all available devices, then you must first perform device selection to filter out unwanted devices.

```
for (unsigned int i = 0; i < queueList.size(); i++) {
  queueList[i].submit([&](handler& cgh) {...});
}
```

Limitations

Consider the following limitations when targeting multiple FPGA devices:

- All FPGA devices use the same FPGA bitstream.
- All FPGA devices used must be of the same FPGA card (same -Xstarget target)

Targeting Multiple Platforms

To compile a design that targets multiple target device types (using different device selectors), you can run the following commands:

Emulation Compile

For compiling your SYCL* code for the FPGA emulator target, execute the following commands:

```
# For Linux:
icpx -fsycl jit_kernel.cpp -c -o jit_kernel.o

icpx -fsycl -fintelfpga -fsycl-link=image fpga_kernel.cpp -o fpga_kernel.a

icpx -fsycl -fintelfpga main.cpp jit_kernel.o fpga_kernel.a
```

```
# For Windows:
icx-cl -fsycl jit_kernel.cpp -c -o jit_kernel.o

icx-cl -fsycl -fintelfpga -fsycl-link=image fpga_kernel.cpp -o fpga_kernel.lib

icx-cl -fsycl -fintelfpga main.cpp jit_kernel.o fpga_kernel.lib
```

The design uses libraries and includes an FPGA kernel (AOT flow) and a CPU kernel (JIT flow).

Specifically, there should be a main function residing in the `main.cpp` file and two kernels for both CPU (`jit_kernel.cpp`) and FPGA (`fpga_kernel.cpp`).

Sample `jit_kernel.cpp` file:

```
sycl::cpu_selector device_selector;
queue deviceQueue(device_selector);
deviceQueue.submit([&](handler &cgh) {
  // CPU Kernel function
});
```

Sample `fpga_kernel.cpp` file:

```
#if FPGA_SIMULATOR
  auto selector = sycl::ext::intel::fpga_simulator_selector_v;
#elif FPGA_HARDWARE
  auto selector = sycl::ext::intel::fpga_selector_v;
#else  // #if FPGA_EMULATOR
  auto selector = sycl::ext::intel::fpga_emulator_selector_v;
#endif
queue deviceQueue(device_selector);
deviceQueue.submit([&](handler &cgh) {
  // FPGA Kernel Function
});
```

FPGA Hardware Compile

To compile for the FPGA hardware target, add the `-Xshardware` flag and remove the `-DFPGA_EMULATOR` flag, as follows:

```
# For Linux:
icpx -fsycl jit_kernel.cpp -c -o jit_kernel.o

//Hardware compilation command. Takes a long time to complete.
icpx -fsycl -fintelfpga -fsycl-link=image -Xshardware fpga_kernel.cpp -o fpga_kernel.a

icpx -fsycl -fintelfpga main.cpp jit_kernel.o fpga_kernel.a
```

```
# For Windows:
icx-cl -fsycl jit_kernel.cpp -c -o jit_kernel.o

//Hardware compilation command. Takes a long time to complete.
icx-cl -fsycl -fintelfpga -fsycl-link=image -Xshardware fpga_kernel.cpp -o fpga_kernel.lib

icx-cl -fsycl -fintelfpga main.cpp jit_kernel.o fpga_kernel.lib
```
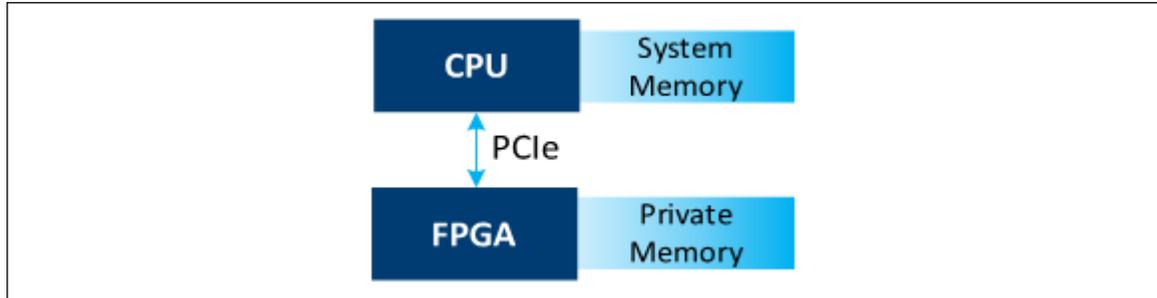
FPGA-CPU Interaction

One of the main influences on the overall performance of an FPGA design is how kernels executing on the FPGA interact with the host on the CPU.

Host and Kernel Interaction

FPGA devices typically communicate with the host (CPU) via PCIe.

FPGA Device Communication with the Host

This is an important factor influencing the performance of SYCL* programs targeting FPGAs. Furthermore, the first time you run a particular SYCL program, you must configure the FPGA with its hardware bitstream, and this may require several seconds.

Data Transfer

Typically, the FPGA board has its own private Double Data Rate (DDR) memory on which it primarily operates. The CPU must bulk transfer or direct memory access (DMA) all data that the kernel needs to access into the FPGA's local DDR memory. After the kernel completes its operations, results must be transferred 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, the Intel® PAC with Intel® Arria® 10 GX FPGA has a PCIe Gen 3 x 8 link, and transfers are typically limited to 6-7 GB/s.

The following are the techniques to manage these data transfer times:

- SYCL allows buffers to be tagged as read-only or write-only, which eliminates some unnecessary transfers.
- Improve the overall system efficiency by maximizing the number of concurrent operations. Since PCIe supports simultaneous transfers in opposite directions and PCIe transfers do not interfere with kernel execution, you can apply techniques such as double buffering. Refer to the Double Buffering Host Utilizing Kernel Invocation Queue topic in the FPGA Optimization Guide for Intel® oneAPI Toolkits and the double_buffering tutorial for additional information about these techniques.
- Improve data transfer throughput by prepinning system memory on board variants that support Restricted USM. Refer to the Prepinning topic in the FPGA Optimization Guide for Intel® oneAPI Toolkits for additional information.

Configuration Time

You must program the hardware bitstream on the FPGA device in a process called configuration. Configuration is a lengthy operation requiring several seconds of communication with the FPGA device. The SYCL runtime manages configuration for you automatically. The runtime decides when the configuration occurs. For example, the configuration might be triggered when a kernel is first launched, but subsequent launches of the same kernel may not trigger configuration since the bitstream has not changed. Therefore, during development, Intel® recommends to time the execution of the kernel after the FPGA has been configured, for example, by performing a warm-up execution of the kernel before timing kernel execution. You must remove this warm-up execution in the production code.

Multiple Kernel Invocations

If a SYCL program submits the same kernel to a SYCL queue multiple times (for example, by calling `single_task` within a loop), only one kernel invocation is active at a time. Each subsequent invocation of the kernel waits for the previous run of the kernel to complete.

FPGA Performance Optimization

The preceding FPGA flow covered the basics of compiling for FPGA, but there is still much to learn about improving the performance of your designs. The Intel® oneAPI DPC++/C++ Compiler provides tools that you can use to find areas for improvement and a variety of flags, attributes, and extensions to control design and compiler behavior. You can find this information in the FPGA Optimization Guide for Intel® oneAPI Toolkits, which should be your main reference if you want to understand how to optimize your design.

Use of RTL Libraries for FPGA

An RTL library is a file that contains one or more functions. You can create an RTL library file using register transfer level (RTL) code. You can then include this library file and use the functions inside your SYCL* kernels.
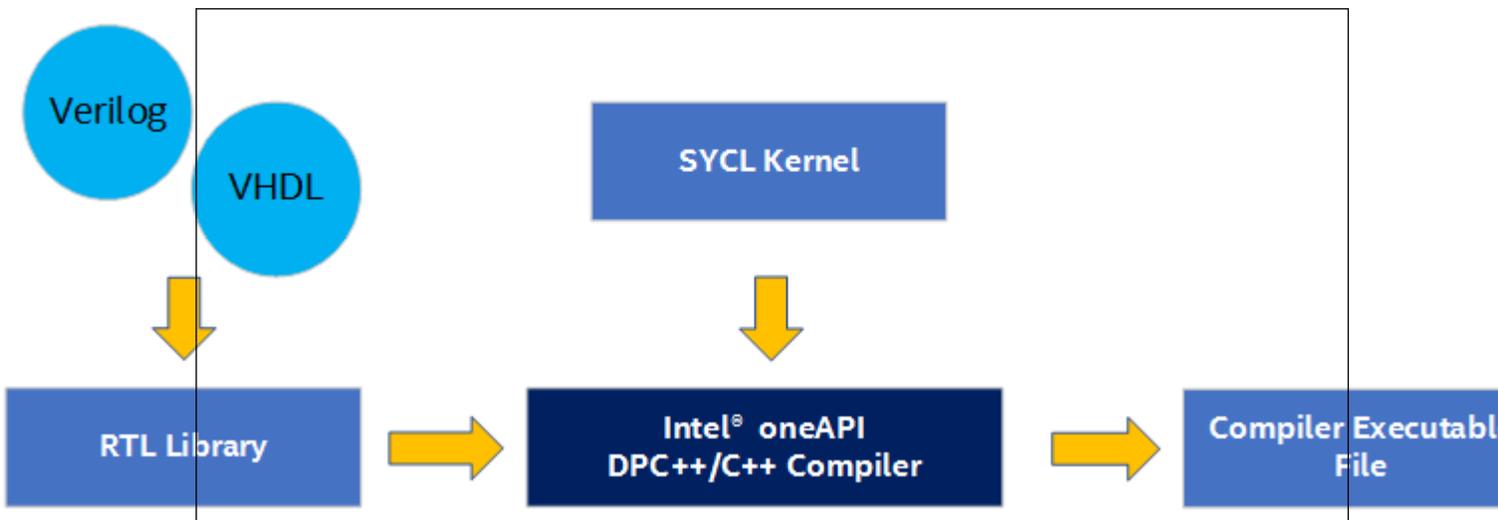
To generate libraries that you can use with SYCL, you need to create the following files:

Generating RTL Libraries for Use with SYCL for FPGA

File or Component	Description
RTL Library Files	
RTL source files	Verilog, System Verilog, or VHDL files that define the RTL component.
	Additional files such as Intel® Quartus® Prime IP File (`.qip`), Synopsys Design Constraints File (`.sdc`), and Tcl Script File (`.tcl`) are not allowed. For information about the file syntax, see Object Manifest File Syntax of an RTL Library.
eXtensible Markup Language File (`.xml`)	Describes the properties of the RTL component. The Intel® oneAPI DPC++/C++ Compiler uses these properties to integrate the RTL component into the SYCL pipeline. For information about the XML attributes, see XML Elements for ATTRIBUTES.
Header file (`.hpp`)	A header file containing valid SYCL kernel language and declares the signatures of functions implemented by the RTL component.

File or Component	Description
Emulation model file (SYCL-based)	Provides a C++ model for the RTL component that is used only for emulation. Full hardware compilations use the RTL source files.
SYCL Functions	
SYCL source files (`.cpp`)	Contains definitions of the SYCL functions. These functions are used during emulation and full hardware compilations.
Header file (`.hpp`)	A header file describing the functions to be called from SYCL in the SYCL syntax.

The format of the library files is determined by which operating system you compile your source code on, with additional sections that carry additional library information.

- On Linux* platforms, a library is a `.a` archive file that contains `.o` object files.
- On Windows* platforms, a library is a `.lib` archive file that contains `.obj` object files.

You can call the functions in the library from your kernel without the need to know the hardware design or the implementation details of the underlying functions in the library. Add the library to the `icpx` command line when you compile your kernel.

Creating a library is a two-step process:

1. Each object file is created from an input source file using the `fpga_crossgen` command.

   - An object file is effectively an intermediate representation of your source code with both a CPU representation and an FPGA representation of your code.
   - An object can be targeted for use with only one Intel® high-level design product. If you want to target more than one high-level design product, you must generate a separate object for each target product.
2. Object files are combined into a library file using the `fpga_libtool` command. Objects created from different types of source code can be combined into a library, provided all objects target the same high-level design product.

   A library is automatically assigned a toolchain version number and can be used only with the targeted high-level design product with the same version number.

Library Toolchain Creation Process
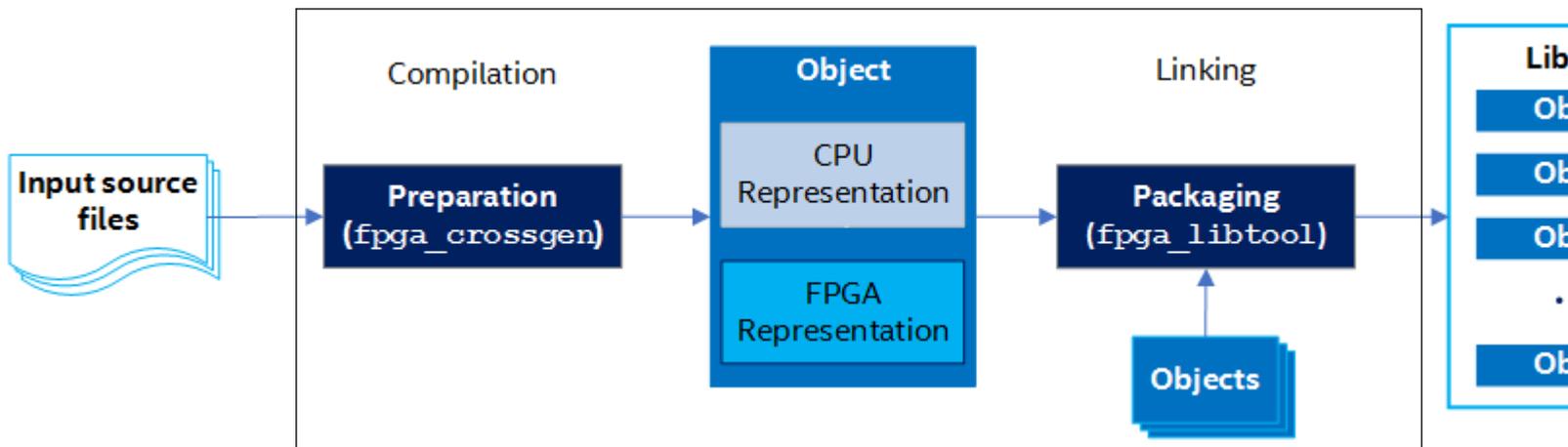
Create Library Objects From Source Code

You can create a library from object files from your source code. A SYCL-based object file includes code for CPU and hardware execution of CPU capturing for use in host and emulation of the kernel.

Create an Object File From Source Code

Use the `fpga_crossgen` command to create library objects from your source code. An object created from your source code contains information required both for emulating the functions in the object and synthesizing the hardware for the object functions.

The `fpga_crossgen` command creates one object file from one input source file. The object created can be used only in libraries that target the same high-level design tool. Also, objects are versioned. Each object is assigned a compiler version number and be used only with high-level design tools with the same version number.

Create a library object using the following command:

```
fpga_crossgen <rtl_spec>.xml --emulation_model <emulation_model>.cpp --target sycl -o
<object_file>
```

The following table describes the parameters:

FPGA crossgen parameters

Parameter	Description
`<rtl_spec>`	An XML file name that specifies the details about your RTL library.
`--target`	Targets an Intel® high-level design tool `sycl` for the library. The objects are combined as object files into a SYCL library archive file using the `fpga_libtool`.
`-o`	Optional flag. This option helps you specify an object file name. If you do not specify this option, the object file name defaults to be the same name as the source code file name but with an object file suffix (`.o` or `.obj`).

Example command:

```
fpga_crossgen lib_rtl_spec.xml --emulation_model lib_rtl_model.cpp --source sycl --target sycl -
o lib_rtl.o
```

Packaging Object Files into a Library File

Gather the object files into a library file so that others can incorporate the library into their projects and call the functions that are contained in the objects in the library. To package object files into a library, use the `fpga_libtool` command.

Before you package object files into a library, ensure that you have the path information for all of the object files that you want to include in the library.

All objects you want to package into a library must have the same version number. The `fpga_libtool` command creates libraries encapsulated in operating-system-specific archive files (`.a` on Linux* and `.lib` on Windows*). You cannot use libraries created on one operating system with an Intel® high-level design product running on a different operating system.

Create a library file using the following command:

```
fpga_libtool file1 file2 ... fileN --target sycl --create <library_name>
```

The command parameters are defined as follows:

Library File Command Parameters

Parameter	Description
`file1 file2 ... fileN`	You can specify one or more object files to include in the library.
`--target sycl`	Target this library for kernels developed. When you mention the `sycl` option, `--target` prepares the library for use with the Intel® oneAPI DPC++/C++ Compiler.
`--create <library_name>`	Allows you to specify the name of the library archive file. Specify the file extension of the library file as `.a` for Linux-platform libraries.

Example command:

```
fpga_libtool lib_rtl.o  --target sycl --create lib.a
```

where, the command packages objects created from RTL source code into a SYCL library called `lib.a`.

> **NOTE** For additional information, refer to the FPGA tutorial sample "Use Library" listed in the Intel® oneAPI Samples Browser on Linux* or Windows*, or access the code sample on Github.

Using Static Libraries

You can include static libraries in your compiler command along with your source files, as shown in the following command:

```
icpx -fsycl -fintelfpga main.cpp lib.a
```

> **NOTE** For the functions you implemented in RTL to be usable, you must declare them in your source code so that the compiler can dynamically link the functions. For example:
>
> ```
> SYCL_EXTERNAL extern "C" void foo()
> ```

Object Manifest File Syntax of an RTL Library

This section provides the syntax of a simple object manifest file for an RTL library that implements double-precision square root function. The RTL library is implemented in VHDL with a Verilog wrapper.

The following object manifest file is for an RTL library named `my_fp_sqrt_double` (line 2) that implements a SYCL helper function named `my_sqrtfd` (line 2):

```
<RTL_SPEC>
  <FUNCTION name="my_sqrtfd" module="my_fp_sqrt_double">
    <ATTRIBUTES>
      <IS_STALL_FREE value="yes"/>
      <IS_FIXED_LATENCY value="yes"/>
      <EXPECTED_LATENCY value="31"/>
      <CAPACITY value="1"/>
      <HAS_SIDE_EFFECTS value="no"/>
      <ALLOW_MERGING value="yes"/>
    </ATTRIBUTES>
    <INTERFACE>
```

```
    <AVALON port="clock" type="clock"/>
    <AVALON port="resetn" type="resetn"/>
    <AVALON port="ivalid" type="ivalid"/>
    <AVALON port="iready" type="iready"/>
    <AVALON port="ovalid" type="ovalid"/>
    <AVALON port="oready" type="oready"/>
    <INPUT port="datain" width="64"/>
    <OUTPUT port="dataout" width="64"/>
  </INTERFACE>
  <C_MODEL>
    <FILE name="c_model.cl" />
  </C_MODEL>
  <REQUIREMENTS>
    <FILE name="my_fp_sqrt_double_s5.v" />
    <FILE name="fp_sqrt_double_s5.vhd" />
  </REQUIREMENTS>
  <RESOURCES>
    <ALUTS value="2057"/>
    <FFS value="3098"/>
    <RAMS value="15"/>
    <MLABS value="43"/>
    <DSPS value="1.5"/>
  </RESOURCES>
  </FUNCTION>
</RTL_SPEC>
```

Elements and Attributes in the Object Manifest File

XML Element	Description
RTL_SPEC	Top-level element in the object manifest file. There can only be one such top-level element in the file. In the above example, the name RTL_SPEC is historic and carries no file-specific meaning.
FUNCTION	Element that defines the SYCL function that the RTL library implements. The name attribute within the FUNCTION element specifies the function's name.
	You can have multiple FUNCTION elements, each declaring a different function that you can call from the SYCL kernel. The same RTL library can implement multiple functions by specifying different parameters.
ATTRIBUTES	Element containing other XML elements that describe various characteristics (for example, latency) of the RTL library. The example RTL library takes one PARAMETER setting named WIDTH, which has a value of 32. Refer to XML Elements for ATTRIBUTES section below for more details about other ATTRIBUTES-specific elements.
	NOTE If you create multiple SYCL helper functions for different libraries, or use the same RTL library with different PARAMETER settings, you must create a separate FUNCTION element for each function.

XML Element	Description
INTERFACE	Element containing other XML elements that describe the RTL library's interface. The example object manifest file shows the Avalon® streaming interface signals that every RTL library must provide (that is, `clock`, `resetn`, `ivalid`, `iready`, `ovalid`, and `oready`). The `resetn` signal is active low. Its synchronicity depends on the target device: • **Intel® Arria® 10, Cyclone® V, Intel® Cyclone® 10 GX, and Cyclone® 10 LP**: The `resetn` signal is asynchronous to the clock signal. • **Intel® Stratix® 10 and Intel Agilex® 7**: The `resetn` signal is synchronous to the clock signal. For more information about reset signal timing, see Intel® Stratix® 10 and Intel Agilex® 7 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Libraries. **NOTE** The signal names must match the ones specified in the `.xml` file. An error occurs during library creation if a signal name is inconsistent.
C_MODEL	Element specifying one or more files that implement SYCL model for the function. The model is used only during emulation. However, the `C_MODEL` element and the associated file(s) must be present when you create the library file.
REQUIREMENTS	Element specifying one or more RTL resource files (that is, `.v`, `.sv`, `.vhd`, `.hex`, and `.mif`). The specified paths to these files are relative to the location of the object manifest file. Each RTL resource file becomes part of the associated Platform Designer component that corresponds to the entire SYCL system. **NOTE** The SYCL library feature does not support `.qip` files. The Intel® oneAPI DPC++/C++ Compiler error occurs if you compile a SYCL kernel while using a library that includes an unsupported resource file type.
RESOURCES	Optional element specifying the FPGA resources that the RTL library uses. If you do not specify this element, the FPGA resources that the RTL library uses defaults to zero.

XML Elements for ATTRIBUTES

In the object manifest file of the RTL library, there are XML elements under ATTRIBUTES that you can specify to set the library's characteristics.

Attention Except for `IS_STALL_FREE` and `EXPECTED_LATENCY`, all elements have safe values. If you are unsure which value you should specify for an attribute, set it to the safe value. Compiling your kernel with a library that uses safe values results in functional hardware. However, the hardware might be larger than the actual size.

XML Elements Associated with the `ATTRIBUTES` Element in the Object Manifest File of an RTL Library

XML Element	Description
`IS_STALL_FREE`	Instructs the compiler properly handle all stall and valid signals. In this case, the compiler can save some area by not generating stall logic around your RTL libarary.
	Set `IS_STALL_FREE` to `"yes"` to indicate that the library neither generates stalls internally nor can it properly handle incoming stalls. The library simply ignores its stall input. If you set `IS_STALL_FREE` to `"no"`, the library must properly handle all stall and valid signals.
	NOTE If you set `IS_STALL_FREE` to `"yes"`, you must also set `IS_FIXED_LATENCY` to `"yes"`. Also, if the RTL library has an internal state, it must properly handle `ivalid=0` inputs. An incorrect `IS_STALL_FREE` setting leads to incorrect results in hardware.
`IS_FIXED_LATENCY`	Indicates whether the RTL library has a fixed latency.
	Set `IS_FIXED_LATENCY` to `"yes"` if the RTL library always takes a known number of clock cycles to compute its output. The value you assign to the `EXPECTED_LATENCY` element specifies the number of clock cycles.
	The safe value for `IS_FIXED_LATENCY` is `"no"`. When you set `IS_FIXED_LATENCY="no"`, the `EXPECTED_LATENCY` value must be at least 1.
	NOTE For a given library, you may set `IS_FIXED_LATENCY` to `"yes"` and `IS_STALL_FREE` to `"no"`. Such a library produces its output in a fixed number of clock cycles and handles stall signals properly.
`EXPECTED_LATENCY`	Specifies the expected latency of the RTL library.

XML Element	Description
	If you set IS_FIXED_LATENCY to "yes", the EXPECTED_LATENCY value indicates the number of pipeline stages inside the library. In this case, you must set this value to be the exact latency of the library. Otherwise, the compiler generates incorrect hardware.
	For a library with variable latency, the compiler balances the pipeline around this library to the EXPECTED_LATENCY value that you specify. For libraries that can stall and require use of signals such as iready, set the EXPECTED_LATENCY value to at least 1. The specified value and the actual latency might differ, which might affect the number of stalls inside the pipeline. However, the resulting hardware is correct.
CAPACITY	Specifies the number of multiple inputs that the library can process simultaneously. You must specify a value for CAPACITY if you also set IS_STALL_FREE="no" and IS_FIXED_LATENCY="no". Otherwise, you do not need to specify a value for CAPACITY.
	If CAPACITY is strictly less than EXPECTED_LATENCY, the compiler automatically inserts capacity-balancing FIFO buffers after this library when necessary.
	The safe value for CAPACITY is 1.
HAS_SIDE_EFFECTS	Indicates whether the RTL library has side effects. Libraries that have internal states or communicate with external memories are examples of libraries with side effects.
	Set HAS_SIDE_EFFECTS to "yes" to indicate that the library has side effects. Specifying HAS_SIDE_EFFECTS to "yes" ensures that optimization efforts do not remove calls to libraries with side effects.
	Stall-free libraries with side effects (that is, IS_STALL_FREE="yes" and HAS_SIDE_EFFECTS="yes") must properly handle ivalid=0 input cases because the library might receive invalid data occasionally.
	The safe value for HAS_SIDE_EFFECTS is "yes".
ALLOW_MERGING	Instructs the compiler to merge multiple instances of the RTL library. Set ALLOW_MERGING to "yes" to allow merging of multiple instances of the library. Intel® recommends setting ALLOW_MERGING to "yes".

XML Element	Description
	The safe value for `ALLOW_MERGING` is `"no"`.
	NOTE Marking the library with `HAS_SIDE_EFFECTS="yes"` does not prevent merging.
`PARAMETER`	Specifies the value of an RTL library parameter.
	`PARAMETER` attributes:
	• `name`: Specifies the name of the RTL library parameter.
	• `value`: Specifies a decimal numeric value for the parameter.
	• `type`: Specifies a system parameter the value of which is used as the RTL library parameter value. You can use the `bspaddresswidth` parameter that specifies the Avalon® memory bus width required to address the memory range configured for SYCL global memory in the board support package.
	NOTE You can specify the value for an RTL library parameter using either a `value` or a `type` attribute.

XML Elements for INTERFACE

In the object manifest file of the RTL library within a SYCL library, there are XML elements under `INTERFACE` that you can define to specify aspects of the RTL library's interface (for example, Avalon® streaming interface).

Mandatory XML Elements Associated with the `INTERFACE` Element in the Object Manifest File of an RTL Library

XML Element	Description
`INPUT`	Specifies the RTL library input parameter.
	`INPUT` attributes:
	• `port`: Specifies the RTL library port name.
	• `width`: Specifies the port width in bits.
	The input parameters are concatenated to form the input stream.
	NOTE Aggregate data structures, such as structs and arrays, are not supported as input parameters.
`OUTPUT`	Specifies the RTL library output parameter.

XML Element	Description
	`OUTPUT` attributes: • `port`: Specifies the port name of the RTL library. • `width`: Specifies the port width in bits. The return value from the input stream is sent out via the output parameter on the output stream. **NOTE** Aggregate data structures, such as structs and arrays, are not supported as input parameters.

XML Elements for RESOURCES

In the object manifest file of the RTL library within a SYCL library, there are optional elements under `RESOURCES` that you can define to specify the FPGA resource utilization of the library. If you do not specify a particular element, it has a default value of zero.

Additional XML Elements to Support External Memory Access

XML Element	Description
`ALUTS`	Specifies the number of combinational adaptive look-up tables (ALUTs) that the library uses.
`FFS`	Specifies the number of dedicated logic registers that the library uses.
`RAMS`	Specifies the number of block RAMs that the library uses.
`DSPS`	Specifies the number of digital signal processing (DSP) blocks that the library uses.
`MLABS`	Specifies the number of memory logic arrays (MLABs) that the library uses. This value is equal to the number of adaptive logic modules (ALMs) that is used for memory divided by 10 because each MLAB consumes 10 ALMs.

Restrictions and Limitations in RTL Support

When creating your RTL module for use inside SYCL kernels, ensure that the RTL module operates within the following restrictions:

• An RTL module must use a single input Avalon® streaming interface. A single pair of ready and valid logic must control all the inputs. You have the option to provide the necessary Avalon® streaming interface ports but declare the RTL module as stall-free. In this case, you do not have to implement proper stall behavior because the Intel® oneAPI DPC++/C++ Compiler creates a wrapper for your module. Refer to Object Manifest File Syntax of an RTL Module for additional information.

> **NOTE** You must handle `ivalid` signals properly if your RTL module has an internal state. Refer to Stall-Free RTL for more information.

• The RTL module must work correctly regardless of the kernel clock frequency.

- RTL modules cannot connect to external I/O signals. All input and output signals must come from a SYCL kernel.
- An RTL module must have a `clock` port, a `resetn` port, and Avalon® streaming interface input and output ports (that is, `ivalid`, `ovalid`, `iready`, `oready`). Name the ports as specified here.
- RTL modules cannot act as stand-alone SYCL kernels. RTL modules can only be helper functions and be integrated into a SYCL kernel during kernel compilation.
- Every function call corresponding to RTL module instantiation is independent of other instantiations. There is no hardware sharing.
- Do not incorporate kernel code into a SYCL library file. Incorporating kernel code into the library file causes the offline compiler to issue an error message. You may incorporate helper functions into the library file.
- An RTL component must receive all its inputs at the same time. A single `ivalid` input signifies that all inputs contain valid data.
- You can only set RTL module parameters in the `<RTL module description file name>.xml` specification file and not in the SYCL kernel source file. To use the same RTL module with multiple parameters, create a separate `FUNCTION` tag for each parameter combination.
- You can only pass data inputs to an RTL module by value via the SYCL kernel code. Do not pass data inputs to an RTL module via pass-by-reference, structs, or channels. In the case of channel data, pass the extracted scalar data.

> **NOTE** Passing data inputs to an RTL module via pass-by-reference or structs causes a fatal error in the offline compiler.

- The debugger (for example, GDB for Linux) cannot step into a library function during emulation if the library is built without the debug information. However, irrespective of whether the library is built with or without the debug data, optimization and area reports are not mapped to the individual code line numbers inside a library.
- Names of RTL module source files cannot conflict with the file names of Intel® oneAPI DPC++/C++ Compiler IP. Both the RTL module source files and the compiler IP files are stored in the `<kernel file name>/system/synthesis/submodules` directory. Naming conflicts cause existing compiler IP files in the directory to be overwritten by the RTL module source files.
- The compiler does not support `.qip` files. You must manually parse nested .qip files to create a flat list of RTL files.

> **Tip** It is challenging to debug an RTL module that works correctly on its own but works incorrectly as part of a SYCL kernel. Double-check all parameters under the ATTRIBUTES element in the `<RTL object manifest file name>.xml` file.

- All compiler area estimation tools assume that the RTL module area is 0. The compiler does not currently support specifying an area model for RTL modules.

Intel® Stratix® 10 and Intel Agilex® 7 Design-Specific Reset Requirements for Stall-Free and Stallable RTL Libraries

When creating an RTL library for Intel® Stratix® 10 SYCL and and Intel Agilex® 7 designs, ensure that the library satisfies specific logic reset requirements.

Reset Requirements for Stall-Free RTL Libraries

A stall-free RTL library is a fixed-latency library for which the Intel® oneAPI DPC++/C++ Compiler can optimize away stall logic. It accepts valid data, and does not have a `ready_in` signal at the end.

- When creating a stall-free RTL library for an Intel® Stratix® 10 design, use synchronous clear signals only.
- After **deassertion** of the reset signal to the stall-free RTL library, the library must be operational within 15 clock cycles. If the reset signal is pipelined within the library, this requirement limits the reset pipelining to no more than 15 stages.

## Reset Requirements for Stallable RTL Libraries

A stallable RTL library has a variable latency, and it relies on backpressured input and output interfaces to function correctly.

- When creating a stallable RTL library for an Intel® Stratix® 10 and and Intel Agilex® 7 designs, use synchronous clear signals only.
- After assertion of the reset signal to the stallable RTL library, the library must deassert its `oready` and `ovalid` interface signals within 40 clock cycles.
- After deassertion of the reset signal to the stallable RTL library, the library must be fully operational within 40 clock cycles. The library signals its readiness by asserting the `oready` interface signal.

Use SYCL Shared Library With Third-Party Applications

Use the Intel® oneAPI DPC++/C++ Compiler to compile your SYCL code to a C-standard shared library (`.so` file on Linux and `.dll` file on Windows). You can then call this library from other third-party code to access a broad base of accelerated functions from your preferred programming language.

SYCL Functions Packaged into a Shared Library File For Use in Third-party Applications
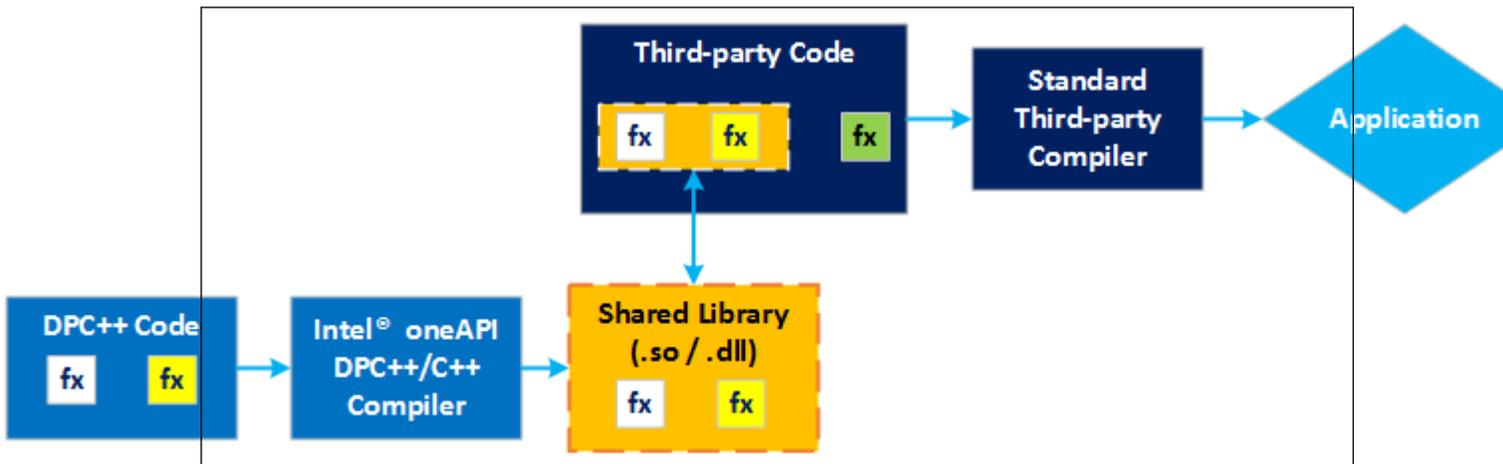
To use a shared library with a third-party application, perform these steps:

1. Define the Shared Library Interface
2. Generate the Library File in Linux or Windows
3. Use the Shared Library

Define the Shared Library Interface

Intel® recommends defining an interface between the C-standard shared library and your SYCL code. The interface must include functions you want to export and how those functions interface with your SYCL code. Prefix the functions that you want to include in the shared library with `extern "C"`.

NOTE If you do not prefix with `extern "C"`, then the functions appear with mangled names in the shared library.

Consider the following example code of the `vector_add` function:

```
extern "C" int vector_add(int *a, int *b, int **c, size_t vector_len) {
  // Create device selector for the device of your interest.
  #if FPGA_EMULATOR
    // Intel extension: FPGA emulator selector on systems without FPGA card.
```

```
    auto selector = sycl::ext::intel::fpga_emulator_selector_v;
  #elif FPGA_SIMULATOR
    // Intel extension: FPGA simulator selector on systems without FPGA card.
    auto selector = sycl::ext::intel::fpga_simulator_selector_v;
  #elif FPGA_HARDWARE
    // Intel extension: FPGA selector on systems with FPGA card.
    auto selector = sycl::ext::intel::fpga_selector_v;
  #else
    // The default device selector will select the most performant device.
    auto selector = default_selector_v;
  #endif

  try {
    queue q(d_selector, exception_handler);
    // SYCL code interface:
    // Vector addition in SYCL
    VectorAddKernel(q, a, b, c, vector_len);
  } catch (exception const &e) {
      std::cout << "An exception is caught for vector add.\n";
      return -1;
  }
  return 0;
}
```

Generate the Shared Library File in Linux

If you are using a Linux system, then perform these steps to generate the shared library file:

1. Compile the device code separately.

```
icpx -fsycl -fPIC –fintelfpga -fsycl-link=image [kernel src files] –o <hw image name> -Xshardware
```

Where:

- `fPIC`: Determines whether the compiler generates position-independent code for the host portion of the device image. Option `–fPIC` specifies full symbol preemption. Global symbol definitions and global symbol references get default (preemptable) visibility unless explicitly specified otherwise. You must use this option when building shared objects. You can also specify this option as `–fpic`.

NOTE PIC is required so that pointers in the shared library reference global addresses and not local addresses.

- `fintelfpga`: Targets FPGA devices.
- `fsycl-link=image`: Informs the Intel® oneAPI DPC++/C++ Compiler to partially link device binaries for use with FPGA.
- `Xshardware`: Compiles for hardware instead of the emulator.

2. Compile the host code separately.

```
icpx -fsycl –fPIC -fintelfpga <host src files> -o <host image name> -c -DFPGA=1
```

Where:

- `DFPGA=1`: Sets a compiler macro, `FPGA`, equal to 1. It is used in the device selector to change between target devices (requires corresponding host code to support this). This is optional as you can also set your device selector to FPGA. For more information, refer to Device Selectors for FPGA.

3. Link the host and device images and create the binary.

```
icpx -fsycl –fPIC -fintelfpga –shared <host image name> <hw image name> -o lib<library name>.so
```

Where:

- `shared`: Outputs a shared library (`.so` file).
- Output file name: Prefix with `lib` for the GCC type of compilers. For additional information, see Shared libraries with GCC on Linux. For example:

```
gcc -Wall -fPIC -L. -o out.a -l<library name>.so
```

NOTE Instead of the above multi-step process, you can also perform a single-step compilation to generate the shared library. However, you must perform a full compile if you want to build the executable for testing purposes (for example, `a.out`) or if you make changes in the SYCL code or C interface.

Generate the Shared Library File in Windows

If you are using a Windows system, then perform these steps to generate the library file:

NOTE

- Intel® recommends creating a new configuration in the same project properties. If you want to build the application, you can avoid changing the configuration type for your project.
- Creating a Windows library with the default Intel® oneAPI Base Toolkit and Intel® Programmable Acceleration Card (PAC) with Intel® Arria® 10 GX FPGA or Intel® FPGA PAC D5005 (previously known as Intel® PAC with Intel® Stratix® 10 SX FPGA) are supported only for FPGA emulation. For custom platforms, contact your board vendor for Windows support for FPGA hardware compiles.

---

1. In Microsoft Visual Studio*, navigate to **Project > Properties**. The **Property Pages** dialog is displayed for your project.

2. Under the **Configuration Properties > General > Project Defaults > Configuration Type** option, select **Dynamic Library (``.dll``)** from the drop-down list.

Project Properties Dialog
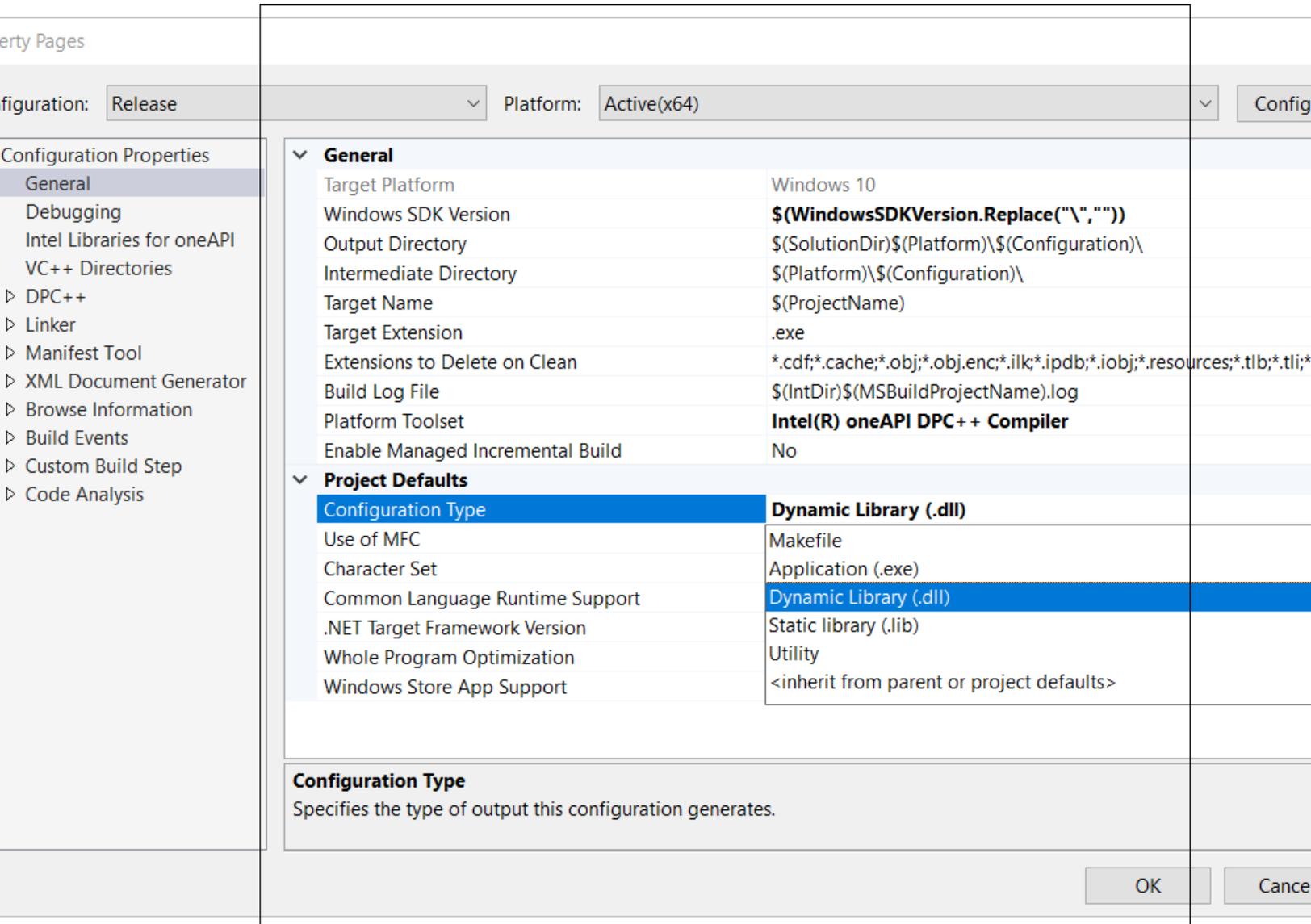
3. Click **OK** to close the dialog.

The project automatically builds to create a dynamic library (.dll)

Use the Shared Library

These steps may vary depending on the language or compiler you decide to use. Consult the specifications for your desired language for more details. See Shared libraries with GCC on Linux for an example.

Generally, follow these steps to use the shared library:

1. Use the shared library function call in your third-party host code.
2. Link your host code with the shared library during the compilation.
3. Ensure that the library file is discoverable. For example:

```
export LD_LIBRARY_PATH=<lib file location>:$LD_LIBRARY_PATH
```

The following is an example illustration of using the shared library:

Example Use of the Shared Library
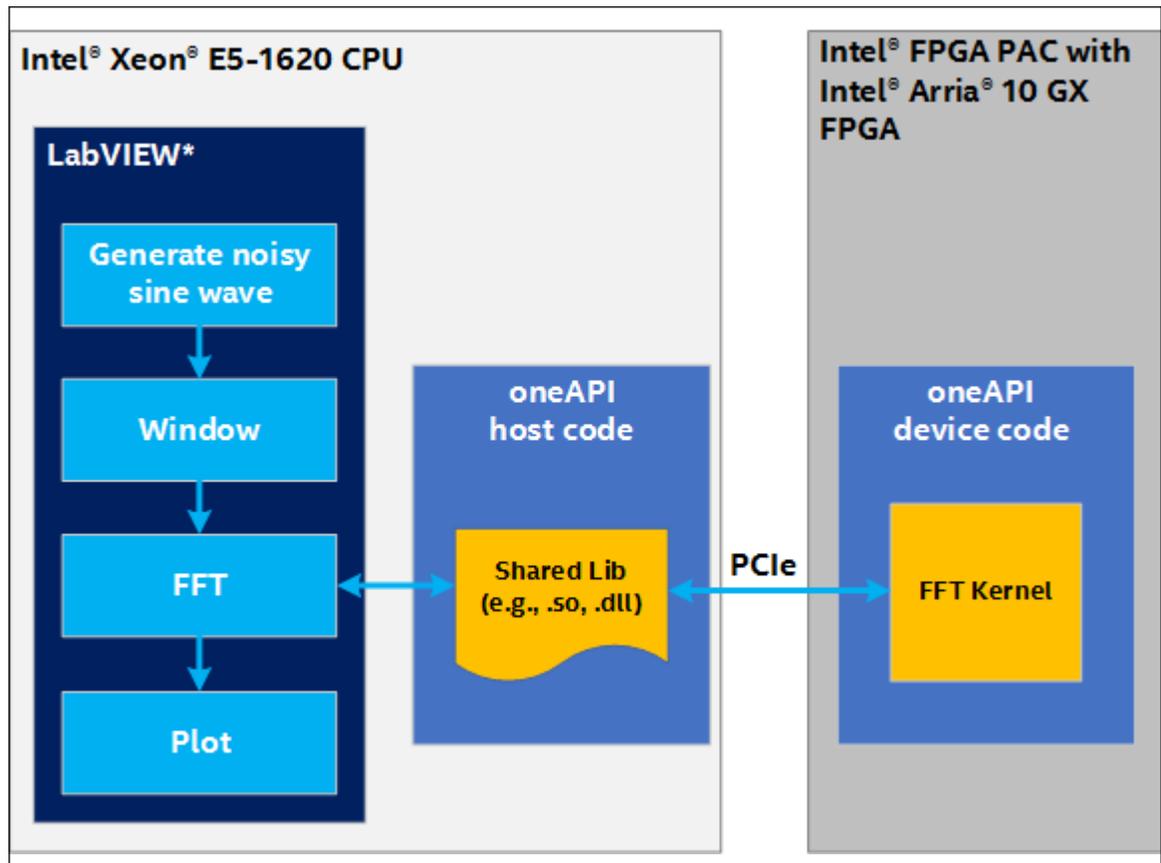
FPGA Workflows in IDEs

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 FPGA Workflows on Third-Party IDEs for Intel® oneAPI Toolkits for more details.

For FPGA development with Visual Studio Code on Linux*, refer to FPGA Development for Intel® oneAPI Toolkits with Visual Studio Code on Linux.

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 Toolkit Libraries

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 SYCL* 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) aims to work with the Intel® oneAPI DPC++/C++ Compiler to provide high-productivity APIs to developers, which can minimize SYCL* programming efforts across devices for high performance parallel applications.

oneDPL consists of the following components:

- Parallel STL:

  - Parallel STL Usage Instructions
  - Macros
- An additional set of library classes and functions (referred to throughout this document as **Extension API**):

  - Parallel Algorithms
  - Iterators
  - Function Object Classes
  - Range-Based API
- Tested Standard C++ APIs
- Random Number Generator

oneDPL Library Usage

Install the Intel® oneAPI Base Toolkit to use oneDPL.

To use Parallel STL or the Extension API, include the corresponding header files in your source code. All oneDPL header files are in the `oneapi/dpl` directory. Use `#include <oneapi/dpl/...>` to include them. oneDPL uses the namespace `oneapi::dpl` for the most of its classes and functions.

To use tested C++ standard APIs, you need to include the corresponding C++ standard header files and use the `std` namespace.

oneDPL Code Sample

oneDPL sample code is available from the oneAPI GitHub repository https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDPL. Each sample includes a readme with build instructions.

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 SYCL* interfaces for achieving performance on various CPU architectures and Intel Graphics Technology for certain key functionalities. oneMKL provides BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

You can use OpenMP* offload to run standard oneMKL computations on Intel GPUs. Refer to OpenMP* offload for C interfaces and OpenMP* offload for Fortran interfaces for more information.

The new SYCL 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 the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Math Kernel Library website. If you plan to use oneMKL as part of the oneAPI Base Toolkit, consider that priority support is available as a paid option. For Intel community-support, visit the oneMKL forum. For the open-source oneMKL Interfaces project, visit the oneMKL GitHub* page.

The table below describes the difference in these oneMKL sites:

oneAPI Specification for oneMKL	Defines the DPC++ interfaces for performance math library functions. The oneMKL specification can evolve faster and more frequently than implementations of the specification.
oneAPI Math Kernel Library (oneMKL) Interfaces Project	An open source implementation of the specification. The project goal is to demonstrate how the DPC++ interfaces documented in the oneMKL specification can be implemented for any math library and work for any target hardware. While the implementation provided here may not yet be the full implementation of the specification, the goal is to build it out over time. We encourage the community to contribute to this project and help to extend support to multiple hardware targets and other math libraries.
Intel(R) oneAPI Math Kernel Library (oneMKL) Product	The Intel product implementation of the specification (with DPC++ interfaces) as well as similar functionality with C and Fortran interfaces, and is provided as part of Intel® oneAPI Base Toolkit. It is highly optimized for Intel CPU and Intel GPU hardware.

oneMKL Usage

When using the SYCL* interfaces, there are a few changes to consider:

- oneMKL has a dependency on the Intel oneAPI DPC++/C++ Compiler and Intel oneAPI DPC++ Library. Applications must be built with the Intel oneAPI DPC++/C++ Compiler, the SYCL headers made available, and the application linked with oneMKL using the DPC++ linker.
- SYCL interfaces in oneMKL use device-accessible Unified Shared Memory (USM) pointers for input data (vectors, matrices, etc.).
- Many SYCL interfaces in oneMKL also support the use of `sycl::buffer` objects in place of the device-accessible USM pointers for input data.
- SYCL 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 SYCL interfaces:

oneMKL Two-level Namespaces

Namespace	Description
`oneapi::mkl`	Contains common elements between various domains in oneMKL
`oneapi::mkl::blas`	Contains dense vector-vector, matrix-vector, and matrix-matrix low level operations
`oneapi::mkl::lapack`	Contains higher-level dense matrix operations like matrix factorizations and eigensolvers
`oneapi::mkl::rng`	Contains random number generators for various probability density functions
`oneapi::mkl::stats`	Contains basic statistical estimates for single and double precision multi-dimensional datasets
`oneapi::mkl::vm`	Contains vector math routines
`oneapi::mkl::dft`	Contains fast fourier transform operations
`oneapi::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 SYCL* interfaces, the following example source code snippets perform a double precision matrix-matrix multiplication on a GPU device.

> **NOTE** The following code example requires additional code to compile and run, as indicated by the inline comments.

```
// Standard SYCL header
#include <CL/sycl.hpp>
// STL classes
#include <exception>
#include <iostream>
// Declarations for Intel oneAPI Math Kernel Library SYCL/DPC++ APIs
#include "oneapi/mkl.hpp"
```

```
int main(int argc, char *argv[]) {
    //
    // User obtains data here for A, B, C matrices, along with setting m, n, k, ldA, ldB, ldC.
    //
    // For this example, A, B and C should be initially stored in a std::vector,
    //   or a similar container having data() and size() member functions.
    //


    // Create GPU device
    sycl::device my_device;
    try {
        my_device = sycl::device(sycl::gpu_selector());
    }
    catch (...) {
        std::cout << "Warning: GPU device not found! Using default device instead." << std::endl;
    }
    // Create asynchronous exceptions handler to be attached to queue.
    // Not required; can provide helpful information in case the system isn't correctly
configured.
    auto my_exception_handler = [](sycl::exception_list exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
                std::rethrow_exception(e);
            }
            catch (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
    sycl::queue my_queue(my_device, my_exception_handler);
    // create sycl buffers of matrix data for offloading between device and host
    sycl::buffer<double, 1> A_buffer(A.data(), A.size());
    sycl::buffer<double, 1> B_buffer(B.data(), B.size());
    sycl::buffer<double, 1> C_buffer(C.data(), C.size());
    // add oneapi::mkl::blas::gemm to execution queue and catch any synchronous exceptions
    try {
        using oneapi::mkl::blas::gemm;
        using oneapi::mkl::transpose;
        gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha, A_buffer, ldA,
B_buffer,
            ldB, beta, C_buffer, ldC);
    }
    catch (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<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/DPC++ specific header that declares the desired `mkl::blas::gemm` API:

```
// Standard SYCL header
#include <CL/sycl.hpp>
// STL classes
#include <exception>
#include <iostream>
// Declarations for Intel oneAPI Math Kernel Library SYCL/DPC++ APIs
#include "oneapi/mkl.hpp"
```

Next, load or instantiate the matrix data on the host machine as usual and then create the GPU 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
sycl::device my_device;
try {
    my_device = sycl::device(sycl::gpu_selector());
}
catch (...) {
    std::cout << "Warning: GPU device not found! Using default device instead." << std::endl;
}
// Create asynchronous exceptions handler to be attached to queue.
// Not required; can provide helpful information in case the system isn't correctly configured.
auto my_exception_handler = [](sycl::exception_list exceptions) {
    for (std::exception_ptr const& e : exceptions) {
        try {
            std::rethrow_exception(e);
        }
        catch (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;
        }
    }
};
```

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 execution queue on my gpu device with exception handler attached
sycl::queue my_queue(my_device, my_exception_handler);
// create sycl buffers of matrix data for offloading between device and host
sycl::buffer<double, 1> A_buffer(A.data(), A.size());
sycl::buffer<double, 1> B_buffer(B.data(), B.size());
sycl::buffer<double, 1> C_buffer(C.data(), C.size());
// add oneapi::mkl::blas::gemm to execution queue and catch any synchronous exceptions
try {
    using oneapi::mkl::blas::gemm;
    using oneapi::mkl::transpose;
    gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha, A_buffer, ldA,
B_buffer,
        ldB, beta, C_buffer, ldC);
}
catch (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;
}
```

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 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 if necessary. In this case, the accessor is used to print 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<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;
```

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 a widely used C++ library for task-based, shared memory parallel programming on the host. The library provides features for parallel programming on CPUs beyond those currently available in SYCL* and ISO C++, including:

- Generic parallel algorithms
- Concurrent containers
- A scalable memory allocator
- Work-stealing task scheduler
- Low-level synchronization primitives

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.

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Threading Building Blocks Library website. If you plan to use oneTBB as part of the oneAPI Base Toolkit, consider that priority support is available as a paid option. For Intel community-support, visit the oneTBB forum. For the community-supported open-source version, visit the oneTBB GitHub* page.

oneTBB Usage

oneTBB can be used with the Intel oneAPI DPC++/C++ Compiler in the same way as with any other C++ compiler. For more details, see the oneTBB documentation.

Currently, oneTBB does not directly use any accelerators. However, it can be combined with SYCL*, OpenMP* offload, and other oneAPI libraries to build a program that efficiently uses all available hardware resources.

oneTBB Code Sample

Two basic oneTBB code samples are available within the oneAPI GitHub repository https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneTBB. Both samples are prepared for CPU and GPU.

- `tbb-async-sycl`: illustrates how computational kernel can be split for execution between CPU and GPU using oneTBB Flow Graph asynchronous node and functional node. The Flow Graph asynchronous node uses SYCL* to implement calculations on GPU while the functional node does CPU part of calculations.
- `tbb-task-sycl`: illustrates how two oneTBB tasks can execute similar computational kernels with one task executing SYCL code and another one the oneTBB code.
- `tbb-resumable-tasks-sycl`: illustrates how a computational kernel can be split for execution between a CPU and GPU using oneTBB resumable task and parallel_for. The resumable task uses SYCL to implement calculations on GPU while parallel_for does the CPU portion of calculations.

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++ SYCL 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 the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Data Analytics Library website. If you plan to use oneDAL as part of the oneAPI Base Toolkit, consider that priority support is available as a paid option. For Intel community-support, visit the oneDAL forum. For the community-supported open-source version, visit the oneDAL GitHub* page.

oneDAL Usage

Information about dependencies needed to build and link your application with oneDAL are available from the oneDAL System Requirements.

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

- extracting SYCL* buffers from numeric tables and pass them to a custom kernel
- creating numeric tables from SYCL buffers

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

oneDAL Code Sample

oneDAL code samples are available from the oneDAL GitHub. The following code sample is a recommended starting point: https://github.com/oneapi-src/oneDAL/tree/master/examples/oneapi/dpc/source/svm

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

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Collective Communications Library website. If you plan to use oneCCL as part of the oneAPI Base Toolkit, consider that premium support is available as a paid option. For the community-supported open-source version, visit the oneCCL GitHub* page.

oneCCL Usage

Refer to the Intel oneAPI Collective Communications Library System Requirements for a full list of hardware and software dependencies, such as MPI and Intel oneAPI DPC++/C++ Compiler.

SYCL*-aware API is an optional feature of oneCCL. There is a choice between CPU and SYCL back ends when creating the oneCCL stream object.

- For CPU backend: Specify `ccl_stream_host` as the first argument.
- For SYCL backend: Specify `ccl_stream_cpu` or `ccl_stream_gpu` depending on the device type.
- For collective operations that operate on the SYCL stream:

  - For C API, oneCCL expects communication buffers to be `sycl::buffer` objects casted to `void*`.
  - For C++ API, oneCCL expects communication buffers to be passed by reference.

Additional usage details are available from https://oneapi-src.github.io/oneCCL/.

oneCCL Code Sample

oneCCL code samples are available from the oneAPI GitHub repository https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneCCL.

A Getting Started sample with instructions to build and run the code is available from within the same GitHub repository.

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 SYCL*/DPC++ API and runtime support for the oneAPI programming model.

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Deep Neural Network Library website. If you plan to use oneDNN as part of the oneAPI Base Toolkit, consider that premium support is available as a paid option. For the community-supported open-source version, visit the oneDNN GitHub* page.

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.

Package Availability by Operating System

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

One of the possible scenarios is executing a SYCL 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 SYCL objects
- Accessing SYCL objects for existing oneDNN objects

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

oneDNN and SYCL Object Mapping 1

oneDNN Objects	SYCL Objects
Engine	cl::sycl::device and cl::sycl::context
Stream	cl::sycl::queue
Memory	cl::sycl::buffer<uint8_t, 1> or Unified Shared Memory (USM) pointer

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 and SYCL Object Mapping 2

oneDNN Object	Constructing from SYCL Object
Engine	`dnnl::sycl_interop::make_engine(sycl_dev, sycl_ctx)`
Stream	`dnnl::sycl_interop::make_stream(engine, sycl_queue)`
Memory	USM based: `dnnl::memory(memory_desc, engine, usm_ptr)`
	Buffer based: `dnnl::sycl_interop::make_memory(memory_desc, engine, sycl_buf)`

oneDNN and SYCL Object Mapping 3

oneDNN Object	Extracting SYCL Object
Engine	`dnnl::sycl_interop::get_device(engine)`
	`dnnl::sycl_interop::get_context(engine)`
Stream	`dnnl::sycl_interop::get_queue(stream)`
Memory	USM pointer: `dnnl::memory::get_data_handle()`
	Buffer: `dnnl::sycl_interop::get_buffer(memory)`

NOTE

- Building applications with oneDNN requires a compiler. The Intel oneAPI DPC++/C++ Compiler is available as part of the Intel oneAPI Base Toolkit.
- You must include dnnl_sycl.hpp to enable the SYCL-interop API.
- Because OpenMP does not rely on the passing of runtime objects, it does not require an interoperability API to work with oneDNN.

---

oneDNN Code Sample

oneDNN sample code is available from the Intel oneAPI Base Toolkit GitHub repository https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDNN. The Getting Started sample is targeted to new users and includes a readme file with example build and run commands.

Intel oneAPI Video Processing Library (oneVPL)

Intel® oneAPI Video Processing Library (oneVPL) is a programming interface for video decoding, encoding, and processing to build portable media pipelines on CPUs, GPUs, and other accelerators. The oneVPL API is used to develop quality, performant video applications that can leverage Intel® hardware accelerators. It provides device discovery and selection in media centric and video analytics workloads, and API primitives for zero-copy buffer sharing. oneVPL is backward compatible with Intel® Media SDK and cross-architecture compatible to ensure optimal execution on current and next generation hardware without source code changes.

oneVPL is an open specification API.

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Video Processing Library website. If you plan to use oneVPL as part of the oneAPI Base Toolkit, consider that priority support is available as a paid option. For Intel community-support, visit the oneVPL forum. For the community-supported open-source version, visit the oneVPL GitHub* page.

oneVPL Usage

Applications can use oneVPL to program video decoding, encoding, and image processing components. oneVPL provides a default CPU implementation that can be used as a reference design before using other accelerators.

oneVPL applications follow a basic sequence in the programming model:

1. The oneVPL dispatcher automatically finds all available accelerators during runtime.
2. Dispatcher uses the selected accelerator context to initialize a session.
3. oneVPL configures the video component at the start of the session.
4. oneVPL processing loop is launched. The processing loop handles work asynchronously.
5. If the application chooses to let oneVPL manage working memory, then memory allocation will be implicitly managed by the video calls in the processing loop.
6. After work is done, oneVPL uses a clear call to clean up all resources.

The oneVPL API is defined using a classic C style interface and is compatible with C++ and SYCL*.

oneVPL Code Sample

oneVPL provides rich code samples to show how to use the oneVPL API. The code samples are included in the release package and are also available from the oneAPI-samples repository on GitHub*.

For example, the hello-decode sample shows a simple decode operation of HEVC input streams and demonstrates the basic steps in the oneVPL programming model.

The sample can be broken down into the following key steps in the code:

> **NOTE** The snippets below may not reflect the latest version of the sample. Refer to the release package or sample repository for the latest version of this example.

1. Initialize oneVPL session with dispatcher:

```
mfxLoader loader = NULL;
mfxConfig cfg = NULL;


loader = MFXLoad();


cfg = MFXCreateConfig(loader);
ImplValue.Type = MFX_VARIANT_TYPE_U32;
ImplValue.Data.U32 = MFX_CODEC_HEVC;
sts = MFXSetConfigFilterProperty(cfg,
(mfxU8*)"mfxImplDescription.mfxDecoderDescription.decoder.CodecID", ImplValue);


sts = MFXCreateSession(loader, 0, &session);
```

Here, `MFXCreateConfig()` creates the dispatcher internal configuration. Once the dispatcher is configured, the application uses `MFXSetConfigFilterProperty()` to set its requirements including codec ID and accelerator preference. After the application sets the desired requirements, the session is created.

2. Start the decoding loop:

```
while(is_stillgoing) {
    sts = MFXVideoDECODE_DecodeFrameAsync(session,
            (isdraining) ? NULL : &bitstream,
            NULL,
            &pmfxOutSurface,
            &syncp);
......
}
```

After preparing the input stream, the stream has the required context and the decoding loop is started immediately.

`MFXVideoDECODE_DecodeFrameAsync()` takes the bit stream as the second parameter. When the bit stream becomes NULL, oneVPL drains the remaining frames from the input and completes the operation. The third parameter is the working memory; the NULL input shown in the example means the application wants oneVPL to manage working memory.

3. Evaluate results of a decoding call:

```
while(is_stillgoing) {
    sts = MFXVideoDECODE_DecodeFrameAsync(...);


    switch(sts) {
        case MFX_ERR_MORE_DATA:
            ......
                ReadEncodedStream(bitstream, codec_id, source);
                ......
            }
            break;
```

```
        case MFX_ERR_NONE:
            do {
                sts = pmfxOutSurface->FrameInterface->Synchronize(pmfxOutSurface,
WAIT_100_MILLSECONDS);
                if( MFX_ERR_NONE == sts ) {
                    sts = pmfxOutSurface->FrameInterface->Map(pmfxOutSurface, MFX_MAP_READ);


                    WriteRawFrame(pmfxOutSurface, sink);


                    sts = pmfxOutSurface->FrameInterface->Unmap(pmfxOutSurface);


                    sts = pmfxOutSurface->FrameInterface->Release(pmfxOutSurface);


                    framenum++;
                }
            } while( sts == MFX_WRN_IN_EXECUTION );
            break;


        default:
            break;
    }
```

For each `MFXVideoDECODE_DecodeFrameAsync()` call, the application continues to read the input bit stream until oneVPL completes a new frame with `MFX_ERR_NONE`, indicating the function successfully completed its operation. For each new frame, the application waits until the output memory `(surface)` is ready and then outputs and releases the output frame.

The `Map()` call is used to map the memory from the discrete graphic memory space to the host memory space.

4. Exit and do cleanup:

```
MFXUnload(loader);
free(bitstream.Data);
    fclose(sink);
    fclose(source);
```

Finally, `MFXUnload()` is called to reclaim the resources from oneVPL. This is the only call that the application must execute to reclaim the oneVPL library resources.

NOTE This example explains the key steps in the oneVPL programming model. It does not explain utility functions for input and output.

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

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

Migrating Code to SYCL* and DPC++

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

Migrating from C++ to SYCL*

SYCL is a single-source style programming model based on C++. It builds on features of C++17 and C++20 to offer an open, multivendor, multiarchitecture solution for heterogeneous programming.

The DPC++ compiler project is bringing SYCL* to an LLVM C++ compiler, with high performance implementations for multiple vendors and architectures.

When accelerating an existing C++ application, 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 SYCL* for the DPC++ Compiler

The Intel© DPC++ Compatibility Tool is part of the Intel© oneAPI Base Toolkit. The goal of this tool is to assist in the migration of an existing program that is written in NVIDIA* CUDA* to a program written in SYCL* and compiled with the DPC++ compiler. This tool generates SYCL code as much as it can. However, 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 be compiled with DPC++. After completing any manual changes, use a DPC++ compiler to create executables.

Migrating CUDA to SYCL Using the DPC++ Compatibility Tool
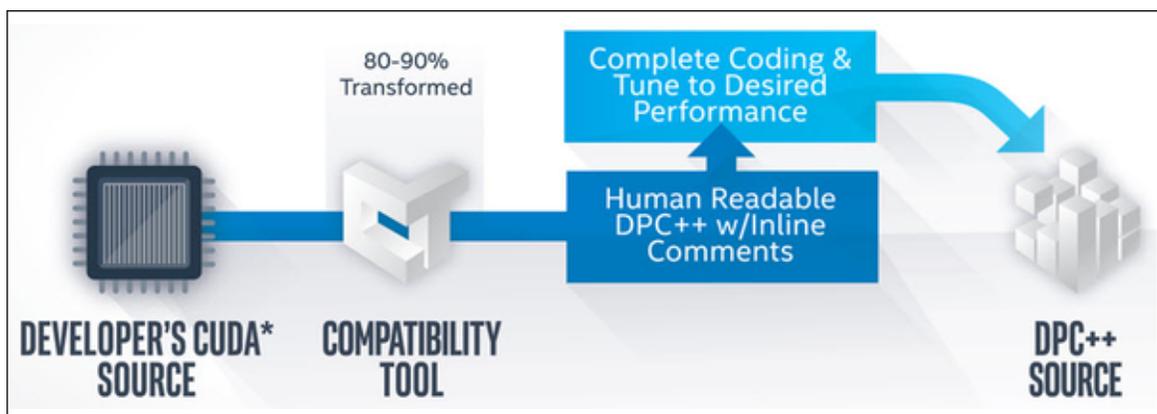
- Additional details, including examples of migrated code and download instructions for the tool, are available from the Intel® DPC++ Compatibility Tool website.
- Full usage information is available from the Intel® DPC++ Compatibility Tool User Guide

Migrating from OpenCL Code to SYCL*

The SYCL runtime for the DPC++ project uses OpenCL and other means to enact the parallelism. SYCL 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.

OpenCL application developers are keenly aware of the somewhat verbose setup code that goes with offloading kernels on devices. Using SYCL, 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

Programming with SYCL* and using the DPC++ compiler, 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 `icpx -fsycl 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.

C/C++ OpenMP* and SYCL* Composability

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. Both the Intel® oneAPI DPC++/C++ Compiler, available with the Intel® oneAPI Base Toolkit, and Intel® C++ Compiler Classic, available with the Intel® oneAPI HPC Toolkit or the Intel® oneAPI IoT Toolkit, support OpenMP and SYCL composability with a set of restrictions. A single application can offload execution to available devices using OpenMP target regions or SYCL constructs in different parts of the code, such as different functions or code segments.

OpenMP and SYCL offloading constructs may be used in separate files, in the same file, or in the same function with some restrictions. OpenMP and SYCL offloading code can be bundled together in executable files, in static libraries, in dynamic libraries, or in various combinations.

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

Restrictions

There are some restrictions to be considered when mixing OpenMP and SYCL constructs in the same application.

- OpenMP directives cannot be used inside SYCL kernels that run in the device. Similarly, SYCL code cannot be used inside the OpenMP target regions. However, it is possible to use SYCL constructs within the OpenMP code that runs on the host CPU.
- OpenMP and SYCL device parts of the program cannot have cross dependencies. For example, a function defined in the SYCL part of the device code cannot be called from the OpenMP code that runs on the device and vice versa. OpenMP and SYCL device parts are linked independently and they form separate binaries that become a part of the resulting fat binary that is generated by the compiler.
- The direct interaction between OpenMP and SYCL runtime libraries are not supported at this time. For example, a device memory object created by OpenMP API is not accessible by SYCL code. That is, using the device memory object created by OpenMP in SYCL code results unspecified execution behavior.

Example

The following code snippet uses SYCL and OpenMP offloading constructs in the same application.

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


float computePi(unsigned N) {
  float Pi;
#pragma omp target map(from : Pi)
#pragma omp parallel for reduction(+ : Pi)
  for (unsigned I = 0; I < N; ++I) {
    float T = (I + 0.5f) / N;
    Pi += 4.0f / (1.0 + T * T);
  }
  return Pi / N;
}


void iota(float *A, unsigned N) {
  cl::sycl::range<1> R(N);
  cl::sycl::buffer<float, 1> AB(A, R);
  cl::sycl::queue().submit([&](cl::sycl::handler &cgh) {
    auto AA = AB.template get_access<cl::sycl::access::mode::write>(cgh);
    cgh.parallel_for<class Iota>(R, [=](cl::sycl::id<1> I) {
      AA[I] = I;
    });
  });
}


int main() {
  std::array<float, 1024u> Vec;
  float Pi;


#pragma omp parallel sections
  {
#pragma omp section
    iota(Vec.data(), Vec.size());
#pragma omp section
    Pi = computePi(8192u);
  }
```

```
    std::cout << "Vec[512] = " << Vec[512] << std::endl;
    std::cout << "Pi = " << Pi << std::endl;
    return 0;
}
```

The following command is used to compile the example code: `icpx -fsycl -fiopenmp -fopenmp-targets=spir64 offloadOmp_dpcpp.cpp`

where

- `-fsycl` option enables SYCL
- `-fiopenmp -fopenmp-targets=spir64` option enables OpenMP* offload

The following shows the program output from the example code.

```
./a.out
Vec[512] = 512
Pi = 3.14159
```

> **NOTE** If the code does not contain OpenMP offload, but only normal OpenMP code, use the following command, which omits `-fopenmp-targets`: `icpx -fsycl -fiopenmp omp_dpcpp.cpp`

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.

Debugging the DPC++ and OpenMP* Offload Process

When writing, debugging, and optimizing code for a host platform, the process of improving your code is simple: deal with language errors when you build, catch and root-cause crashes/incorrect results during execution with a debugger, then identify and fix performance issues using a profiling tool.

Improving code can become considerably more complicated in applications where part of the execution is offloaded to another device using either DPC++ or OpenMP* offload.

- Incorrect use of the DPC++ or OpenMP* offload languages may not be exposed until after just-in-time compilation occurs. These issues can be exposed earlier with ahead-of-time (AOT) compilation.
- Crashes due to logic errors may arise as unexpected behavior on the host, on the offload device, or in the software stack that ties the various computing devices together. To root cause these issues, you need to:
  - Debug what is happening in your code on the host using a standard debugger, such as Intel Distribution for GDB*.
  - Debug problems on the offload device using a device-specific debugger. Note, however, that the device may have a different architecture, conventions for representing compute threads, or assembly than the host.
  - To debug problems that show up in the intermediate software stack only when kernels and data are being exchanged with the device, you need to monitor the communication between device and host and any errors that are reported during the process.

- Besides the usual performance issues that can occur on the host and offload devices, the patterns by which the host and offload device work together can have a profound impact on application performance. This is another case where you need to monitor the communications between the host and offload device.

This section discusses the various debugging and performance analysis tools and techniques available to you for the entire lifecycle of the offload program.

To troubleshoot your applications that use OpenMP* or the SYCL* API with extensions to offload resources, see the Troubleshoot Highly Parallel Applications tutorial.

oneAPI Debug Tools for SYCL* and OpenMP* Development

The following tools are available to help with debugging the SYCL* and OpenMP* offload process.

Tools to debug SYCL* and OpenMP* offload process

Tool	When to Use
Environment variables	Environment variables allow you to gather diagnostic information from the OpenMP and SYCL runtimes at program execution with no modifications to your program.
The onetrace tool from Profiling Tools Interfaces for GPU (PTI for GPU)	When using the oneAPI Level Zero and OpenCL™ backends for SYCL and OpenMP Offload, this tool can be used to debug backend errors and for performance profiling on both the host and device. Learn more: • Onetrace tool GitHub • PTI for GPU GitHub
Intercept Layer for OpenCL™ Applications	When using the OpenCL™ backend for SYCL and OpenMP Offload, this library can be used to debug backend errors and for performance profiling on both the host and device (has wider functionality comparing with onetrace).
Intel® Distribution for GDB*	Used for source-level debugging of the application, typically to inspect logical bugs, on the host and any devices you are using (CPU, GPU, FPGA emulation).
Intel® Inspector	This tool helps to locate and debug memory and threading problems, including those that can cause offloading to fail. **NOTE** Intel Inspector is included in the Intel oneAPI HPC Toolkit or the Intel oneAPI IoT Toolkit.
In-application debugging	In addition to these tools and runtime based approaches, the developer can locate problems using other approaches. For example: • Comparing kernel output to expected output • Sending intermediate results back by variables they create for debugging purposes • Printing results from within kernels **NOTE** Both SYCL and OpenMP allow printing to stdout from within an offload region - be sure to note which SIMD lane or thread is providing the output.
SYCL Exception Handler	Some DPC++ programming errors are returned as exceptions by the SYCL runtime during program execution. They can help you diagnose errors in your code that are flagged at runtime. For more details and examples, refer to <link> Using SYCL Exceptions </link>. For Samples

Tool	When to Use
	that demonstrate SYCL Exceptions, refer to: * Guided Matrix Multiplication Exception * Guided Matrix Multiplication Invalid Contexts * Guided Matrix Multiplication Race Condition
Intel® Advisor	Use to ensure Fortran, C, C++, OpenCL™, and SYCL applications realize full performance potential on modern processors.
Intel® VTune ™ Profiler	Use to gather performance data either on the native system or on a remote system.

Debug Environment Variables

Both the OpenMP* and SYCL offload runtimes, as well as Level Zero, OpenCL, and the Shader Compiler, provide environment variables that help you understand the communication between the host and offload device. The variables also allow you to discover or control the runtime chosen for offload computations.

OpenMP* Offload Environment Variables

There are several environment variables that you can use to understand how OpenMP Offload works and control which backend it uses.

> **NOTE** OpenMP is not supported for FPGA devices.

OpenMP* Offload Environment Variables

Environment Variable	Description
LIBOMPTARGET_DEBUG=<Num>	Controls whether or not debugging information will be displayed. See details in Runtimes This environment variable enables debug output from the OpenMP Offload runtime. It reports: • The available runtimes detected and used (1,2) • When the chosen runtime is started and stopped (1,2) • Details on the offload device used (1,2) • Support libraries loaded (1,2) • Size and address of all memory allocations and deallocations (1,2) • Information on every data copy to and from the device, or device mapping in the case of unified shared memory (1,2) • When each kernel is launched and details on the launch (arguments, SIMD width, group information, etc.) (1,2) • Which Level Zero/OpenCL API functions are invoked (function name, arguments/parameters) (2) Values: `<Num>=0`: Disabled

Environment Variable	Description
	`<Num>=1`: Displays basic debug information from the plugin actions such as device detection, kernel compilation, memory copy operations, kernel invocations, and other plugin-dependent actions.
	`<Num>=2`: Additionally displays which GPU runtime API functions are invoked with which arguments/parameters.
	Default: 0
LIBOMPTARGET_INFO=<Num>	This variable controls whether basic offloading information will be displayed from the offload runtime. Allows the user to request different types of runtime information from libomptarget. See details in Runtimes
	- Prints all data arguments upon entering an OpenMP device kernel (1) - Indicates when a mapped address already exists in the device mapping table (2) - Dumps the contents of the device pointer map if target offloading fails (4) - Indicates when an entry is changed in the device mapping table (8) - Indicates when data is copied to and from the device (32)
	Values: (0, 1, 2, 4, 8, 32)
	Default: 0
LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]	This variable enables the display of performance data for offloaded OpenMP code. It displays:
	- Total data transfer times (read and write) - Data allocation times - Module build times (just-in-time compile) - The execution time of each kernel.
	Values:
	`F` - disabled `T` - enabled with timings in milliseconds `T,usec` - enabled with timings in microseconds
	Default: `F`
	Example: `export LIBOMPTARGET_PLUGIN_PROFILE=T,usec`
	``` <Enable> := 1 | T <Unit> := usec | unit_usec ```
	Enables basic plugin profiling and displays the result when program finishes. Microsecond is the default unit if `<Unit>` is not specified.

Environment Variable	Description
LIBOMPTARGET_PLUGIN=<Name>	This environment variable allows you to choose the backend used for OpenMP offload execution. **NOTE** The Level Zero backend is only supported for GPU devices. ```<Name> := LEVEL0 | OPENCL | CUDA | X86_64 | NIOS2 |` ` level0 | opencl | cuda | x86_64 | nios2 |``` Designates offload plugin name to use. Offload runtime does not try to load other RTLs if this option is used. Values: • `LEVEL0` or `LEVEL_ZERO` - uses the Level Zero backend • `OPENCL` - uses the OpenCL™ backend Default: • For GPU offload devices: `LEVEL0` • For CPU or FPGA offload devices: `OPENCL`
LIBOMPTARGET_PROFILE=<FileName>	Allows libomptarget to generate time profile output similar to Clang's `-ftime-trace` option. See details in Runtimes
LIBOMPTARGET_DEVICES=<DeviceKind>	```<DeviceKind> := DEVICE | SUBDEVICE | SUBSUBDEVICE | ALL |` ` device | subdevice | subsubdevice | all``` Controls how subdevices are exposed to users. `DEVICE/device`: Only top-level devices are reported as OpenMP devices, and `subdevice` clause is supported. `SUBDEVICE/subdevice`: Only 1st-level subdevices are reported as OpenMP devices, and `subdevice` clause is ignored. `SUBSUBDEVICE/subsubdevice`: Only 2nd-level subdevices are reported as OpenMP devices, and `subdevice` clause is ignored. On Intel GPU using Level Zero backend, limiting the `subsubdevice` to a single compute slice within a tile also requires setting additional GPU compute runtime environment variable `CFESingleSliceDispatchCCSMode=1`.

Environment Variable	Description
	`ALL/all`: All top-level devices and their subdevices are reported as OpenMP devices, and `subdevice` clause is ignored. This is not supported on Intel GPU and is being deprecated. **Default**: Equivalent to `<DeviceKind>`=device
LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option>	`<Option> := 0	<PoolInfoList>` `<PoolInfoList> := <PoolInfo>[,<PoolInfoList>]` `<PoolInfo> :=` `<MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]]` `<MemType> := all	device	host	shared` `<AllocMax> := positive integer or empty,` `max allocation size in MB` `<Capacity> := positive integer or empty,` `number of allocations from a` ` single block` `<PoolSize> := positive integer or empty,` `max pool size in MB`  Controls how reusable memory pool is configured. Pool is a list of memory blocks that can serve at least `<Capacity>` allocations of up to `<AllocMax>` size from a single block, with total size not exceeding `<PoolSize>`.  **Default**: Equivalent to `<Option>`=device,1,4,256,host,1,4,256,shared,8,4,256
LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE=<Num>	Sets the staging buffer size to `<Num>` KB. Staging buffer is used in copy operations between host and device as a temporary storage for two-step copy operation. The buffer is only used for discrete devices. **Default**: 16
LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH=<Value>	`<Value> := <Type>[,<Count>]` `<Type> := none	NONE	copy	COPY	compute	COMPUTE` `<Count> := maximum number of commands to batch`  Enables command batching for a target region.  `<Type>`=none|NONE: Disables command batching. `<Type>`=copy|COPY: Enables command batching for a target region for data transfer. `<Type>`=compute|COMPUTE: Enables command batching for a target region for data transfer and compute, disabling use of copy engine.  If `<Type>` is either `copy` or `compute` (enabled) and `<Count>` is not specified, batching is performed for all eligible commands for the target region.

Environment Variable	Description
LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_COMMAND_LIST=<Bool>	**Default**: `<Type>=none` (Disabled)
	`<Bool> := 1	T	t	0	F	f`
	Enables/disables using immediate command list for kernel submission.
OMP_TARGET_OFFLOAD=MANDATORY	**Default**: Disabled
	This is defined by the OpenMP Standard : https://www.openmp.org/spec-html/5.1/openmpse74.html#x340-5150006.17

SYCL* and DPC++ Environment Variables

The DPC++ compiler supports all standard SYCL environment variables. The full list is available from GitHub. Of interest for debugging are the following SYCL environment variables, plus an additional Level Zero environment variable.

SYCL* and DPC++ Environment Variables

Environment Variable	Description
SYCL_DEVICE_FILTER	This complex environment variable allows you to limit the runtimes, compute device types, and compute device IDs used by the runtime to a subset of all available combinations.
	The compute device IDs correspond to those returned by the SYCL API, `clinfo`, or `sycl-ls` (with the numbering starting at 0) and have no relation to whether the device with that ID is of a certain type or supports a specific runtime. Using a programmatic special selector (like `gpu_selector`) to request a device filtered out by `SYCL_DEVICE_FILTER` will cause an exception to be thrown.
	Refer to the Environment Variables descriptions in GitHub for additional details: https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md
	Example values include:
	• `opencl:cpu` - use only the OpenCL™ runtime on all available CPU devices
	• `opencl:gpu` - use only the OpenCL runtime on all available GPU devices
	• `opencl:gpu:2` - use only the OpenCL runtime on only the third device, which also has to be a GPU
	• `level_zero:gpu:1` - use only the Level Zero runtime on only the second device, which also has to be a GPU

Environment Variable	Description
	• `opencl:cpu,level_zero` - use only the OpenCL runtime on the CPU device, or the Level Zero runtime on any supported compute device
	Default: use all available runtimes and devices
ONEAPI_DEVICE_SELECTOR	This device selection environment variable can be used to limit the choice of devices available when the SYCL-using application is run. Useful for limiting devices to a certain type (like GPUs or accelerators) or backends (like Level Zero or OpenCL). This device selection mechanism is replacing SYCL_DEVICE_FILTER . The ONEAPI_DEVICE_SELECTOR syntax is shared with OpenMP and also allows devices to be chosen. See oneAPI DPC++ Compiler documentation for a full description.
SYCL_PI_TRACE	This environment variable enables debug output from the runtime.
	Values:
	• 1 - report SYCL plugins and devices discovered and used
	• 2 - report SYCL API calls made, including arguments and result values
	• -1 - provides all available tracing
	Default:disabled
ZE_DEBUG	This environment variable enables debug output from the Level Zero backend when used with the runtime. It reports:
	• Level Zero APIs called
	• Level Zero event information
	Value: variable defined with any value - enabled
	Default: disabled

Environment Variables that Produce Diagnostic Information for Support

The Level Zero backend provides a few environment variables that can be used to control behavior and aid in diagnosis.

• Level Zero Specification, core programming guide: https://spec.oneapi.com/level-zero/latest/core/PROG.html#environment-variables
• Level Zero Specification, tool programming guide: https://spec.oneapi.com/level-zero/latest/tools/PROG.html#environment-variables

An additional source of debug information comes from the Intel® Graphics Compiler, which is called by the Level Zero or OpenCL backends (used by both the OpenMP Offload and SYCL/DPC++ Runtimes) at runtime or during Ahead-of-Time (AOT) compilation. Intel Graphics Compiler creates the appropriate executable code for the target offload device. The full list of these environment variables can be found at https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md. The two that are most often needed to debug performance issues are:

- `IGC_ShaderDumpEnable=1 (default=0)` causes all LLVM, assembly, and ISA code generated by the Intel® Graphics Compiler to be written to `/tmp/IntelIGC/<application_name>`
- `IGC_DumpToCurrentDir=1 (default=0)` writes all the files created by `IGC_ShaderDumpEnable` to your current directory instead of `/tmp/IntelIGC/<application_name>`. Since this is potentially a lot of files, it is recommended to create a temporary directory just for the purpose of holding these files.

If you have a performance issue with your OpenMP offload or SYCL offload application that arises between different versions of Intel® oneAPI, when using different compiler options, when using the debugger, and so on, then you may be asked to enable `IGC_ShaderDumpEnable` and provide the resulting files. For more information on compatibility, see oneAPI Library Compatibility.

Offload Intercept Tools

In addition to debuggers and diagnostics built into the offload software itself, it can be quite useful to monitor offload API calls and the data sent through the offload pipeline. For Level Zero, if your application is run as an argument to the onetrace and ze_tracer tools, they will intercept and report on various aspects of Level Zero made by your application. For OpenCL™, you can add a library to `LD_LIBRARY_PATH` that will intercept and report on all OpenCL calls, and then use environment variables to control what diagnostic information to report to a file. You can also use onetrace or cl_tracer to report on various aspects of OpenCL API calls made by your application. Once again, your application is run as an argument to the onetrace or cl_tracer tool.

Intercept Layer for OpenCL™ Applications

This library collects debugging and performance data when OpenCL is used as the backend to your SYCL or OpenMP offload program. When OpenCL is used as the backend to your SYCL or OpenMP offload program, this tool can help you detect buffer overwrites, memory leaks, mismatched pointers, and can provide more detailed information about runtime error messages (allowing you to diagnose these issues when either CPU, FPGA, or GPU devices are used for computation). Note that you will get nothing useful if you use ze_tracer on a program that uses the OpenCL backend, or the Intercept Layer for OpenCL Applications library and cl_tracer on a program that uses the Level Zero backend.

Additional resources:

- Extensive information on building and using the Intercept Layer for OpenCL Applications is available from https://github.com/intel/opencl-intercept-layer.

  NOTE For best results, run `cmake` with the following flags: `–DENABLE_CLIPROF=TRUE –DENABLE_CLILOADER=TRUE`

- Information about a similar tool (CLIntercept) is available from https://github.com/gmeeker/clintercept and https://sourceforge.net/p/clintercept/wiki/Home/.
- Information on the controls for the Intercept Layer for OpenCL Applications can be found at https://github.com/intel/opencl-intercept-layer/blob/master/docs/controls.md.
- Information about optimizing for GPUs is available from the Intel oneAPI GPU Optimization Guide.

Profiling Tools Interfaces for GPU (onetrace, cl_tracer, and ze_trace)

Like the Intercept Layer for OpenCL™ Applications, these tools collect debugging and performance data from applications that use the OpenCL and Level Zero offload backends for offload via OpenMP* or SYCL. Note that Level Zero can only be used as the backend for computations that happen on the GPU (there is no Level Zero backend for the CPU or FPGA at this time). The onetrace tool is part of the Profiling Tools Interfaces for GPU (PTI for GPU) project, found at https://github.com/intel/pti-gpu. This project also contains the ze_tracer and cl_tracer tools, which trace just activity from the Level Zero or OpenCL offload backends respectively. The ze_tracer and cl_tracer tools will produce no output if they are used with the application using the other backend, while onetrace will provide output no matter which offload backend you use.

The onetrace tool is distributed as source. Instructions for how to build the tool are available from https://github.com/intel/pti-gpu/tree/master/tools/onetrace. The tool provides the following features:

- Call logging: This mode allows you to trace all standard Level Zero (L0) and OpenCL™ API calls along with their arguments and return values annotated with time stamps. Among other things, this can give you supplemental information on any failures that occur when a host program tries to make use of an attached compute device.
- Host and device timing: These provide the duration of all API calls, the duration of each kernel, and application runtime for the entire application.
- Device Timeline mode: Gives time stamps for each device activity. All the time stamps are in the same (CPU) time scale.
- Chrome Call Logging mode: Dumps API calls to JSON format that can be opened in chrome://tracing browser tool.

These data can help debug offload failures or performance issues.

Additional resources:

- Profiling Tools Interfaces for GPU (PTI for GPU) GitHub project
- Onetrace tool GitHub

Intel® Distribution for GDB*

The Intel Distribution for GDB* is an application debugger that allows you to inspect and modify the program state. With the debugger, both the host part of your application and kernels that are offloaded to a device can be debugged seamlessly in the same debug session. The debugger supports the CPU, GPU, and FPGA-emulation devices. Major features of the tool include:

- Automatically attaching to the GPU device to listen to debug events
- Automatically detecting JIT-compiled, or dynamically loaded, kernel code for debugging
- Defining breakpoints (both inside and outside of a kernel) to halt the execution of the program
- Listing the threads; switching the current thread context
- Listing active SIMD lanes; switching the current SIMD lane context per thread
- Evaluating and printing the values of expressions in multiple thread and SIMD lane contexts
- Inspecting and changing register values
- Disassembling the machine instructions
- Displaying and navigating the function call-stack
- Source- and instruction-level stepping
- Non-stop and all-stop debug mode
- Recording the execution using Intel Processor Trace (CPU only)

For more information and links to full documentation for Intel Distribution for GDB, see *Get Started with Intel Distribution for GDB on* Linux* Host | Windows* Host.

Intel® Inspector for Offload

Intel® Inspector is a dynamic memory and threading error checking tool for users developing serial and multithreaded applications. It can be used to verify correctness of the native part of the application as well as dynamically generated offload code.

Unlike the tools and techniques above, Intel Inspector cannot be used to catch errors in offload code that is communicating with a GPU or an FPGA. Instead, Intel Inspector requires that the SYCL or OpenMP runtime needs to be configured to execute kernels on CPU target. In general, it requires definition of the following environment variables prior to an analysis run.

- To configure a SYCL application to run kernels on a CPU device

```
export SYCL_DEVICE_FILTER=opencl:cpu
```

- To configure an OpenMP application to run kernels on a CPU device

```
export OMP_TARGET_OFFLOAD=MANDATORY
export LIBOMPTARGET_DEVICETYPE=cpu
```
- To enable code analysis and tracing in JIT compilers or runtimes

```
export CL_CONFIG_USE_VTUNE=True
export CL_CONFIG_USE_VECTORIZER=false
```

Use one of the following commands to start analysis from the command line. You can also start from the Intel Inspector graphical user interface.

- Memory: `inspxe-cl -c mi3 -- <app> [app_args]`
- Threading: `inspxe-cl -c ti3 -- <app> [app_args]`

View the analysis result using the following command: `inspxe-cl -report=problems -report-all`

If your SYCL or OpenMP Offload program passes bad pointers to the OpenCL™ backend, or passes the wrong pointer to the backend from the wrong thread, Intel Inspector should flag the issue. This may make the problem easier to find than trying to locate it using the intercept layers or the debugger.

Additional details are available from the *Intel Inspector User Guide for*Linux* OS|Windows* OS.

Trace the Offload Process

When a program that offloads computation to a GPU is started, there are lot of moving parts involved in program execution. Machine-independent code needs to be compiled to machine-dependent code, data and binaries need to be copied to the device, results returned, etc. This section will discuss how to trace all this activity using the tools described in the oneAPI Debug Tools section.

Kernel Setup Time

Before offload code can run on the device, the machine-independent version of the kernel needs to be compiled for the target device, and the resulting code needs to be copied to the device. This can complicate/skew benchmarking if this kernel setup time is not considered. Just-in-time compilation can also introduce a noticeable delay when debugging an offload application.

If you have an OpenMP* offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the amount of time required to build the offload code "ModuleBuild", which you can compare to the overall execution time of your program.

Kernel setup time is more difficult to determine if you have a SYCL* offload program.

- If Level Zero or OpenCL™ is your backend, you can derive kernel setup time from the Device Timing and Device Timeline returned by onetrace or ze_tracer.
- If OpenCL™ is your backend, you may also be able to derive the information by setting the `BuildLogging`, `KernelInfoLogging`, `CallLogging`, `CallLoggingElapsedTime`, `KernelInfoLogging`, `HostPerformanceTiming`, `HostPerformanceTimeLogging`, `ChromeCallLogging`, or `CallLoggingElapsedTime` flags when using the Intercept Layer for OpenCL Applications to get similar information. You can also derive kernel setup time from the Device Timing and Device Time- line returned by onetrace or cl_tracer.

You can also use these tools to supplement the information returned by `LIBOMPTARGET_PLUGIN_PROFILE=T`.

For details on how Intel© VTune™ Profiler can analyze kernel setup time, see Enable Linux* Kernel Analysis

Monitoring Buffer Creation, Sizes, and Copies

Understanding when buffers are created, how many buffers are created, and whether they are reused or constantly created and destroyed can be key to optimizing the performance of your offload application. This may not always be obvious when using a high-level programming language like OpenMP or SYCL, which can hide a lot of the buffer management from the user.

At a high level, you can track buffer-related activities using the `LIBOMPTARGET_DEBUG` and `SYCL_PI_TRACE` environment variables when running your program. `LIBOMPTARGET_DEBUG` gives you more information than `SYCL_PI_TRACE` - it reports the addresses and sizes of the buffers created. By contrast, `SYCL_PI_TRACE` just reports the API calls, with no information you can easily tie to the location or size of individual buffers.

At a lower level, if you are using Level Zero or OpenCL™ as your backend, the Call Logging mode of onetrace or ze_tracer will give you information on all API calls, including their arguments. This can be useful because, for example, a call for buffer creation (such as `zeMemAllocDevice`) will give you the size of the resulting buffer being passed to and from the device. onetrace and ze_tracer also allows you to dump all the Level Zero device-side activities (including memory transfers) in Device Timeline mode. For each activity one can get append (to command list), submit (to queue), start and end times.

If you are using OpenCL as your backend, setting the `CallLogging`, `CallLoggingElapsedTime`, and `ChromeCallLogging` flags when using the Intercept Layer for OpenCL Applications should give you similar information. The Call Logging mode of onetrace or cl_tracer will give you information on all OpenCL API calls, including their arguments. As was the case above, onetrace and cl_tracer also allow you to dump all the OpenCL device-side activities (including memory transfers) in Device Timeline mode.

Total Transfer Time

Comparing total data transfer time to kernel execution time can be important for determining whether it is profitable to offload a computation to a connected device.

If you have an OpenMP offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the amount of time required to build ("DataAlloc"), read ("DataRead"), and write data ("DataWrite") to the offload device (although only in aggregate).

Data transfer times can be more difficult to determine if you have a C++ program using SYCL.

- If Level Zero or OpenCL™ is your backend, you can derive total data transfer time from the Device Timing and Device Timeline returned by onetrace or ze_tracer.
- If OpenCL is your backend, you can use onetrace or cl_tracer, or alternatively you may also be able to derive the information by setting the `BuildLogging`, `KernelInfoLogging`, `CallLogging`, `CallLoggingElapsedTime`, `KernelInfoLogging`, `HostPerformanceTiming`, `HostPerformanceTimeLogging`, `ChromeCallLogging`, or `CallLoggingElapsedTime` flags when using the Intercept Layer for OpenCL Applications.

For details on how Intel® VTune™ Profiler can analyze transfer setup time, see these sections of the Intel® VTune™ Profiler User Guide: GPU Offload AnalysisGPU Compute/Media Hotspots ViewHotspots Report

Kernel Execution Time

If you have an OpenMP offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the total execution time of every offloaded kernel ("Kernel#…").

For programs using SYCL to offload kernels:

- If Level Zero or OpenCL™ is your backend, the Device Timing mode of onetrace or ze_tracer will give you the device-side execution time for every kernel.

- If OpenCL is your backend , you can use onetrace or cl_tracer, or alternatively you may be able to derive the information by setting the `CallLoggingElapsedTime`, `DevicePerformanceTiming`, `DevicePerformanceTimeKernelInfoTracking`, `DevicePerformanceTimeLWSTracking`, `DevicePerformanceTimeGWSTracking`, `ChromePerformanceTiming`, `ChromePerformanceTimingInStages` flags when using the Intercept Layer for OpenCL Applications.

For details on how Intel® VTune™ Profiler can analyze kernel execution time, see Accelerators Analysis Group

When Device Kernels are Called and Threads are Created

On occasion, offload kernels are created and transferred to the device a long time before they actually start executing (usually only after all data required by the kernel has also been transferred, along with control).

You can set a breakpoint in a device kernel using the Intel® Distribution for GDB* and a compatible GPU. From there, you can query kernel arguments, monitor thread creation and destruction, list the current threads and their current positions in the code (using "info thread"), and so on.

Debug the Offload Process

Run with Different Runtimes or Compute Devices

When an offload program fails to run correctly or produces incorrect results, a relatively quick sanity check is to run the application on a different runtime (OpenCL™ vs. Level Zero) or compute device (CPU vs. GPU) using `LIBOMPTARGET_PLUGIN` and `OMP_TARGET_OFFLOAD` for OpenMP* applications, and `SYCL_DEVICE_FILTER` for SYCL* applications. Errors that reproduce across runtimes mostly eliminate the runtime as being a problem. Errors that reproduce on all available devices mostly eliminates bad hardware as the problem.

Debug CPU Execution

Offload code has two options for CPU execution: either a "host" implementation, or the CPU version of OpenCL. A "host" implementation is a truly native implementation of the offloaded code, meaning it can be debugged like any of the non-offloaded code. The CPU version of OpenCL, while it goes through the OpenCL runtime and code generation process, eventually ends up as normal parallel code running under a TBB runtime. Again, this provides a familiar debugging environment with familiar assembly and parallelism mechanisms. Pointers have meaning through the entire stack, and data can be directly inspected. There are also no memory limits beyond the usual limits for any operating system process.

Finding and fixing errors in CPU offload execution may solve errors seen in GPU offload execution with less pain, and without requiring use of a system with an attached GPU or other accelerator.

For OpenMP applications, to get a "host" implementation, remove the "target" or "device" constructs, replacing them with normal host OpenMP code. If `LIBOMPTARGET_PLUGIN=OPENCL` and offload to the GPU is disabled, then the offloaded code runs under the OpenMP runtime with TBB providing parallelism.

For SYCL applications, with `SYCL_DEVICE_FILTER=host` the "host" device is actually single-threaded, which may help you determine if threading issues, such as data races and deadlocks, are the source of execution errors. Setting `SYCL_DEVICE_FILTER=opencl:cpu` uses the CPU OpenCL runtime, which also uses TBB for parallelism.

Debug GPU Execution Using Intel® Distribution for GDB* on compatible GPUs

Intel® Distribution for GDB* is extensively documented in *Get Started with Intel Distribution for GDB on* Linux* Host|Windows* Host. Useful commands are briefly described in the *Intel Distribution for GDB* Reference Sheet. However, since debugging applications with GDB* on a GPU differs slightly from the process on a host (some commands are used differently and you might see some unfamiliar output), some of those differences are summarized here.

The Debugging with Intel Distribution for GDB on Linux OS Host Tutorial shows a sample debug session where we start a debug session of a SYCL program, define a breakpoint inside the kernel, run the program to offload to the GPU, print the value of a local variable, switch to the SIMD lane 5 of the current thread, and print the variable again.

As in normal GDB*, for a command `<CMD>`, use the `help <CMD>` command of GDB to read the information text for `<CMD>`. For example:

```
(gdb) help info threads
Display currently known threads.
Usage: info threads [OPTION]... [ID]...
If ID is given, it is a space-separated list of IDs of threads to display.
Otherwise, all threads are displayed.


Options:
  -gid
    Show global thread IDs.
```

Inferiors, Threads, and SIMD Lanes Referencing in GDB*

The threads of the application can be listed using the debugger. The printed information includes the thread ids and the locations that the threads are currently stopped at. For the GPU threads, the debugger also prints the active SIMD lanes.

In the example referenced above, you may see some unfamiliar formatting used when threads are displayed via the GDB "info threads" command:

```
  Id              Target Id           Frame
  1.1             Thread <id omitted>  <frame omitted>
  1.2             Thread <id omitted>  <frame omitted>
* 2.1:1           Thread 1073741824    <frame> at array-transform.cpp:61
  2.1:[3 5 7]     Thread 1073741824    <frame> at array-transform.cpp:61
  2.2:[1 3 5 7]   Thread 1073741888    <frame> at array-transform.cpp:61
  2.3:[1 3 5 7]   Thread 1073742080    <frame> at array-transform.cpp:61
```

Here, GDB is displaying the threads with the following format:
`<inferior_number>.<thread_number>:<SIMD Lane/s>`

So, for example, the thread id "`2.3:[1 3 5 7]`" refers to SIMD lanes 1, 3, 5, and 7 of thread 3 running on inferior 2.

An "inferior" in the GDB terminology is the process that is being debugged. In the debug session of a program that offloads to the GPU, there will typically be two inferiors; one "native" inferior representing a host part of the program (inferior 1 above), and another "remote" inferior representing the GPU device (inferior 2 above). Intel Distribution for GDB automatically creates the GPU inferior - no extra steps are required.

When you print the value of an expression, the expression is evaluated in the context of the current thread's current SIMD lane. You can switch the thread as well as the SIMD lane to change the context using the "thread" command such as "`thread 3:4 `", "`thread :6 `", or "`thread 7 `". The first command makes a switch to the thread 3 and SIMD lane 4. The second command switches to SIMD lane 6 within the current thread. The third command switches to thread 7. The default lane selected will either be the previously selected lane, if it is active, or the first active lane within the thread.

The "thread apply command" may be similarly broad or focused (which can make it easier to limit the output from, for example, a command to inspect a variable). For more details and examples about debugging with SIMD lanes, see the Debugging with Intel Distribution for GDB on Linux OS Host Tutorial.

More information about threads and inferiors in GDB can be found from https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html and https://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-Connections-and-Programs.html#Inferiors-Connections-and-Programs.

Controlling the Scheduler

By default, when a thread hits a breakpoint, the debugger stops all the threads before displaying the breakpoint hit event to the user. This is the all-stop mode of GDB. In the non-stop mode, the stop event of a thread is displayed while the other threads run freely.

In all-stop mode, when a thread is resumed (for example, to resume normally with the `continue` command, or for stepping with the `next` command), all the other threads are also resumed. If you have some breakpoints set in threaded applications, this can quickly get confusing, as the next thread that hits the breakpoint may not be the thread you are following.

You can control this behavior using the `set scheduler-locking` command to prevent resuming other threads when the current thread is resumed. This is useful to avoid intervention of other threads while only the current thread executes instructions. Type `help set scheduler-locking` for the available options, and see https://sourceware.org/gdb/current/onlinedocs/gdb/Thread-Stops.html for more information. Note that SIMD lanes cannot be resumed individually; they are resumed together with their underlying thread.

In non-stop mode, by default, only the current thread is resumed. To resume all threads, pass the "`-a`" flag to the `continue` command.

Dumping Information on One or More Threads/Lanes (Thread Apply)

Commands for inspecting the program state are typically executed in the context of the current thread's current SIMD lane. Sometimes it is desired to inspect a value in multiple contexts. For such needs, the `thread apply` command can be used. For instance, the following executes the `print element` command for the SIMD lanes 3-5 of Thread 2.5:

```
(gdb) thread apply 2.5:3-5 print element
```

Similarly, the following runs the same command in the context of SIMD lane 3, 5, and 6 of the current thread:

```
(gdb) thread apply :3 :5 :6 print element
```

Stepping GPU Code After a Breakpoint

To stop inside the kernel that is offloaded to the GPU, simply define a breakpoint at a source line inside the kernel. When a GPU thread hits that source line, the debugger stops the execution and shows the breakpoint hit. To single-step a thread over a source-line, use the `step` or `next` commands. The `step` commands steps into functions while `next` steps over calls. Before stepping, we recommend to `set scheduler-locking step` to prevent intervention of other threads.

Building a SYCL Executable for Use with Intel® Distribution for GDB*

Much like when you want to debug a host application, you need to set some additional flags to create a binary that can be debugged on the GPU. See Get Started with Intel Distribution for GDB on Linux* Host for details.

For a smooth debug experience when using the just-in-time (JIT) compilation flow, enable debug information emission from the compiler via the `-g` flag, and disable optimizations via the `-O0` flag for both a host and JIT-compiled kernel of the application. The flags for the kernel are taken during link time. For example:

- Compile your program using: `icpx -fsycl -g -O0 -c myprogram.cpp`
- Link your program using: `icpx -fsycl -g -O0 myprogram.o`

If you are using CMake to configure the build of your program, use the `Debug` type for the `CMAKE_BUILD_TYPE`, and append `-O0` to the `CMAKE_CXX_FLAGS_DEBUG` variable. For example: `set (CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0")`

Applications that are built for debugging may take a little longer to start up than when built with the usual "release" level of optimization. Thus, your program may appear to run a little more slowly when started in the debugger. If this causes problems, developers of larger applications may want to use ahead-of-time (AOT) compilation to JIT the offload code when their program is built, rather than when it is run (warning, this may also take longer to build when using `-g -O0`). For more information, see Compilation Flow Overview.

When doing ahead-of-time compilation for GPU, you must use a device type that fits your target device. Run the following command to see the available GPU device options on your current machine: `ocloc compile --help`

Additionally, the debug mode for the kernel must be enabled. The following example AoT compilation command targets the KBL device:

```
dpcpp -g -O0 -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xs "-device kbl -internal_options -cl-kernel-debug-enable -options -cl-opt-disable"
myprogram.cpp
```

Building an OpenMP* Executable for use with Intel® Distribution for GDB*

Compile and link your program using the `-g -O0` flags. For example:

```
icpx -fiopenmp -O0 -fopenmp-targets=spir64 -c -g myprogram.cpp
icpx -fiopenmp -O0 -fopenmp-targets=spir64 -g myprogram.o
```

Set the following environment variables to disable optimizations and enable debug info for the kernel:

```
export LIBOMPTARGET_OPENCL_COMPILATION_OPTIONS="-g -cl-opt-disable"
export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-g -cl-opt-disable"
```

Debugging GPU Execution

A common issue with offload programs is that they may to fail to run at all, instead giving a generic OpenCL™ error with little additional information. The Intercept Layer for OpenCL Applications along with onetrace, ze_tracer, and cl_tracer can be used to get more information about these errors, often helping the developer identify the source of the problem.

Intercept Layer for OpenCL Applications

Using this library, in particular the `Buildlogging`, `ErrorLogging`, and `USMChecking=1` options, you can often find the source of the error.

1. Create a `clintercept.conf` file in the home directory with the following content:

```
SimpleDumpProgramSource=1
CallLogging=1
LogToFile=1
//KernelNameHashTracking=1
BuildLogging=1
ErrorLogging=1
USMChecking=1
//ContextCallbackLogging=1
// Profiling knobs
KernelInfoLogging=1
DevicePerformanceTiming=1
DevicePerformanceTimeLWSTracking=1
DevicePerformanceTimeGWSTracking=1
```

2. Run the application with cliloader as follows:

```
<OCL_Intercept_Install_Dir>/bin/cliloader/cliloader -d ./<app_name> <app_args>
```

3. Review the following results in the ~CLIntercept_Dump/<app_name> directory:

- clintercept_report.txt: Profiling results
- clintercept_log.txt: Log of OpenCL™ calls used to debug OpenCL issues

The following snippet is from an example log file generated by a program that returned the runtime error:
CL_INVALID_ARG_VALUE (-50)

```
...
<<<< clSetKernelArgMemPointerINTEL -> CL_SUCCESS
>>>>
clGetKernelInfo( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257cS2_S2_S2_S2_S2_S2
_S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197->45clES6_EUlNS4_7nd_itemILi3EEEE225->13 ):
param_name = CL_KERNEL_CONTEXT (1193)
<<<< clGetKernelInfo -> CL_SUCCESS
>>>>
clSetKernelArgMemPointerINTEL( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257cS2_
S2_S2_S2_S2_S2_S2_fS2_S2_S2_S2_iENKUlRN2cl4sycl7handlerEE197-
>45clES6_EUlNS4_7nd_itemILi3EEEE225->13 ): kernel = 0xa2d51a0, index = 3, value = 0x41995e0
mem pointer 0x41995e0 is an UNKNOWN pointer and no device support shared system pointers!
ERROR! clSetKernelArgMemPointerINTEL returned CL_INVALID_ARG_VALUE (-50)
<<<< clSetKernelArgMemPointerINTEL -> CL_INVALID_ARG_VALUE
```

In this example, the following values help with debugging the error:

- `ZTSZZ10outer_coreiP5mesh`
- `index = 3, value = 0x41995e0`

Using this data, you can identify which kernel had the problems, what argument was problematic, and why.

onetrace, ze_tracer, and cl_tracer

Similar to Intercept Layer for OpenCL Applications, the onetrace, ze_tracer and cl_tracer tools can help find the source of errors detected by the Level Zero and OpenCL™ runtimes.

To use the onetrace or ze_tracer tools to root-cause Level Zero issues (cl_tracer would be used the same way to root-cause OpenCL issues):

1. Use Call Logging mode to run the application. Redirecting the tool output to a file is optional, but recommended.

```
./onetrace -c ./<app_name> <app_args> [2> log.txt]
```

The command for ze_tracer is the same - just substitute "ze_tracer" for "onetrace".

1. Review the call trace to figure out the error (log.txt). For example:

```
>>>> [102032049] zeKernelCreate: hModule = 0x55a68c762690 desc = 0x7fff865b5570 {29 0 0 GEMM}
phKernel = 0x7fff865b5438 (hKernel = 0)
<<<< [102060428] zeKernelCreate [28379 ns] hKernel = 0x55a68c790280 -> ZE_RESULT_SUCCESS (0)
...
>>>> [102249951] zeKernelSetGroupSize: hKernel = 0x55a68c790280 groupSizeX = 256 groupSizeY = 1
groupSizeZ = 1
<<<< [102264632] zeKernelSetGroupSize [14681 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [102278558] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 0 argSize = 8
pArgValue = 0x7fff865b5440
<<<< [102294960] zeKernelSetArgumentValue [16402 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [102308273] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 1 argSize = 8
pArgValue = 0x7fff865b5458
<<<< [102321981] zeKernelSetArgumentValue [13708 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
```

```
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0)
...
```

The example log data shows:

- A level zero API call that causes the problem (`zeKernelSetArgumentValue`)
- The problem reason (`ZE_RESULT_ERROR_INVALID_ARGUMENT`)
- The argument index (`argIndex = 1`)
- An invalid value location (`pArgValue = 0x7fff865b5458`)
- A kernel handle (`hKernel = 0x55a68c790280`), which provides the name of the kernel for which this issue is observed (GEMM)

More information could be obtained by omitting the "redirection to file" option and dumping all the output (application output + tool output) into one stream. Dumping to one stream may help determine the source of the error in respect to application output (for example, you can find that the error happens between application initialization and the first phase of computations):

```
Level Zero Matrix Multiplication (matrix size: 1024 x 1024, repeats 4 times)
Target device: Intel® Graphics [0x3ea5]
...
>>>> [104131109] zeKernelCreate: hModule = 0x55af5f39ca10 desc = 0x7ffe289c7f80 {29 0 0 GEMM}
phKernel = 0x7ffe289c7e48 (hKernel = 0)
<<<< [104158819] zeKernelCreate [27710 ns] hKernel = 0x55af5f3ca600 -> ZE_RESULT_SUCCESS (0)
...
>>>> [104345820] zeKernelSetGroupSize: hKernel = 0x55af5f3ca600 groupSizeX = 256 groupSizeY = 1
groupSizeZ = 1
<<<< [104360082] zeKernelSetGroupSize [14262 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104373679] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 0 argSize = 8
pArgValue = 0x7ffe289c7e50
<<<< [104389443] zeKernelSetArgumentValue [15764 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104402448] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 1 argSize = 8
pArgValue = 0x7ffe289c7e68
<<<< [104415871] zeKernelSetArgumentValue [13423 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0)
...
Matrix multiplication time: 0.0427564 sec
Results are INCORRECT with accuracy: 1
...
Matrix multiplication time: 0.0430995 sec
Results are INCORRECT with accuracy: 1
...
Total execution time: 0.381558 sec
```

Correctness

Offload code is often used for kernels that can efficiently process large amounts of information on the attached compute device, or to generate large amounts of information from some input parameters. If these kernels are running without crashing, this can often mean that you learn that they are not producing the correct results much later in program execution.

In these cases, it can be difficult to identify which kernel is producing incorrect results. One technique for finding the kernel producing incorrect data is to run the program twice, once using a purely host-based implementation, and once using an offload implementation, capturing the inputs and outputs from every kernel (often to individual files). Now compare the results and see which kernel call is producing unexpected results (within a certain epsilon - the offload hardware may have a different order of operation or native precision that causes the results to differ from the host code in the last digit or two).

Once you know which kernel is producing incorrect results, and you are working with a compatible GPU, use Intel Distribution for GDB to determine the reason. See the Debugging with Intel Distribution for GDB on Linux OS Host Tutorial for basic information and links to more detailed documentation.

Both SYCL and OpenMP* also allow for the use of standard language print mechanisms (`printf` for SYCL and C++ OpenMP offload, `print *, ...` for Fortran OpenMP offload) within offloaded kernels, which you can use to verify correct operation while they run. Print the thread and SIMD lane the output is coming from and consider adding synchronization mechanisms to ensure printed information is in a consistent state when printed. Examples for how to do this in SYCL using the stream class can be found in the Intel oneAPI GPU Optimization Guide. You could use a similar approach to the one described for SYCL for OpenMP offload.

Tip Using `printf` can be verbose in SYCL kernels. To simplify, add the following macro:

```
#ifdef __SYCL_DEVICE_ONLY__
  #define CL_CONSTANT __attribute__((opencl_constant))
#else
  #define CL_CONSTANT
#endif
#define PRINTF(format, ...) { \
          static const CL_CONSTANT char _format[] = format; \
          sycl::ONEAPI::experimental::printf(_format, ## __VA_ARGS__); }
```

Usage example: `PRINTF("My integer variable:%d\n", (int) x);`

Failures

Just-in-time (JIT) compilation failures that occur at runtime due to incorrect use of the SYCL or OpenMP* offload languages will cause your program to exit with an error.

In the case of SYCL, if you cannot find these using ahead-of-time compilation of your SYCL code, selecting the OpenCL backend, setting SimpleDumpProgramSource and BuildLogging, and using the Intercept Layer for OpenCL Applications may help identify the kernel with the syntax error.

Logic errors can also result in crashes or error messages during execution. Such issues can include:

- Passing a buffer that belongs to the wrong context to a kernel
- Passing the "this" pointer to a kernel rather than a class element
- Passing a host buffer rather than a device buffer
- Passing an uninitialized pointer, even if it is not used in the kernel

Using the Intel® Distribution for GDB* (or even the native GDB), if you watch carefully, you can record the addresses of all contexts created and verify that the address being passed to an offload kernel belongs to the correct context. Likewise, you can verify that the address of a variable passed matches that of the variable itself, and not its containing class.

It may be easier to track buffers and addresses using the Intercept Layer for OpenCL™ allocation or onetrace/ cl_tracer and choosing the appropriate backend. When using the OpenCL backend, setting `CallLogging`, `BuildLogging`, `ErrorLogging`, and `USMChecking` and running your program should produce output that explains what error in your code caused the generic OpenCL error to be produced.

Using onetrace or ze_tracer's Call Logging or Device Timeline should give additional enhanced error information to help you better understand the source of generic errors from the Level Zero backend. This can help locate many of the logic errors mentioned above.

If the code is giving an error when offloading to a device using the Level Zero backend, try using the OpenCL backend. If the program works, report an error against the Level Zero backend. If the error reproduces in the OpenCL backend to the device, try using the OpenCL CPU backend. In OpenMP offload, this can be specified by setting `OMP_TARGET_OFFLOAD` to `CPU`. For SYCL, this can be done by setting `SYCL_DEVICE_FILTER=opencl:cpu`. Debugging with everything on the CPU can be easier, and removes complications caused by data copies and translation of the program to a non-CPU device.

As an example of a logic issue that can get you in trouble, consider what is captured by the lambda function used to implement the `parallel_for` in this SYCL code snippet.

```
class MyClass {
private:
  int *data;
  int factor;
  :
void run() {
  :
  auto data2 = data;
  auto factor2 = factor;
  {
    dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh)
    {
       auto dpct_global_range = grid * block;
       auto dpct_local_range = block;
       cgh.parallel_for<dpct_kernel_name<class kernel_855a44>>(
          cl::sycl::nd_range<1>(
            cl::sycl::range<1> dpct_global_range.get(0)),
            cl::sycl::range<1>( dpct_local_range.get(0))),
            [=](cl::sycl::nd_item<3> item_ct1)
       {
          kernel(data, b, factor, LEN, item_ct1);    // This blows up
       });
     });
   }
} // run
} // MyClass
```

In the above code snippet, the program crashes because `[=]` will copy by value all variables used inside the lambda. In the example it may not be obvious that "`factor`" is really "`this->factor`" and "`data`" is really "`this->data`," so "`this`" is the variable that is captured for the use of "`data`" and "`factor`" above. OpenCL or Level Zero will crash with an illegal arguments error in the "`kernel(data, b, factor, LEN, item_ct1)`" call.

The fix is the use of local variables `auto data2` and `auto factor2`. "`auto factor2 = factor`" becomes "`int factor2 = this->factor`" so using `factor2` inside the lambda with `[=]` would capture an "`int`". We would rewrite the inner section as "`kernel(data2, b, factor2, LEN, item_ct1);`".

> **NOTE** This issue is commonly seen when migrating CUDA* kernels. You can also resolve the issue by keeping the same CUDA kernel launch signature and placing the command group and lambda inside the kernel itself.

Using the Intercept Layer for OpenCL™ allocation or onetrace or ze_tracer, you would see that the kernel was called with two identical addresses, and the extended error information would tell you that you are trying to copy a non-trivial data structure to the offload device.

Note that if you are using unified shared memory (USM), and "`MyClass`" is allocated in USM, the above code will work. However, if only "`data`" is allocated in USM, then the program will crash for the above reason.

In this example, note that you can also re-declare the variables in local scope with the same name so that you don't need to change everything in the kernel call.

Intel® Inspector can also help diagnose these sorts of failures. If you set the following environment variables and then run Memory Error Analysis on offload code using the CPU device, Intel Inspector will flag many of the above issues:

- OpenMP*

  - `export OMP_TARGET_OFFLOAD=CPU`
  - `export OMP_TARGET_OFFLOAD=MANDATORY`
  - `export LIBOMPTARGET_PLUGIN=OPENCL`
- SYCL

  - `export SYCL_DEVICE_FILTER=opencl:cpu`
  - Or initialize your queue with a CPU selector to force use of the OpenCL CPU device: `cl::sycl::queue Queue(cl::sycl::cpu_selector{});`
- Both

  - `export CL_CONFIG_USE_VTUNE=True`
  - `export CL_CONFIG_USE_VECTORIZER=false`

> **NOTE** A crash can occur when optimizations are turned on during the compilation process. If turning off optimizations causes your crash to disappear, use `-g -[optimization level]` for debugging. For more information, see the Intel oneAPI DPC++/C++ Compiler Developer Guide and Reference.

Using the SYCL* Exception Handler

As explained in the book Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL:

The C++ exception features are designed to cleanly separate the point in a program where an error is detected from the point where it may be handled, and this concept fits very well with both synchronous and asynchronous errors in SYCL.

Using the methods from this book, C++ exception handling can help terminate a program when an error is encountered instead of allowing the program to silently fail.

Note: the *italicized* text in this section is copied directly from Chapter 5 "Error Handling" in the book *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL.* In some places, text has been removed for brevity. See the book for full details.

Ignoring Error Handling

C++ and SYCL are designed to tell us that something went wrong even when we don't handle errors explicitly. The default result of unhandled synchronous or asynchronous errors is abnormal program termination which an operating system should tell us about. The following two examples mimic the behavior that will occur if we do not handle a synchronous and an asynchronous error, respectively.

The figure below shows the result of an unhandled C++ exception, which could be an unhandled SYCL synchronous error, for example. We can use this code to test what a particular operating system will report in such a case.

Figure: *Unhandled exception in C++*

```
#include <iostream>

class something_went_wrong {};

int main() {
  std::cout << "Hello\n";

  throw(something_went_wrong{});
}

Example output in Linux:

Hello
terminate called after throwing an instance of 'something_went_wrong'

Aborted (core dumped)
```

The next figure shows example output from std::terminate being called, which will be the result of an unhandled SYCL asynchronous error in our application. We can use this code to test what a particular operating system will report in such a case.

Although we probably should handle errors in our programs, since uncaught errors will be caught and the program terminated, we do not need to worry about a program silently failing!

Figure: `std::terminate` *is called when a SYCL asynchronous exception isn't handled*

```
#include <iostream>

int main() {
    std::cout << "Hello\n";

    std::terminate();
}

Example output in Linux:
Hello
terminate called without an active exception
Aborted (core dumped)
```

The book details reasons for why synchronous errors can be handled by the C++ exceptions. However, to handle asynchronous errors at controlled points in an application, one must be aware of the situations where a SYCL throw is invoked, and accordingly, SYCL exceptions must be used.

Synchronous errors defined by SYCL are a derived class from std::exception of type ``sycl::exception``, which allows us to catch the SYCL errors specifically though a try-catch structure such as what we see in the figure below.

Figure. *Pattern to catch* `sycl::exception` *specifically*

```
try{
  // Do some SYCL work
} catch (sycl::exception &e) {
  // Do something to output or handle the exception
  std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
  return 1;
}
```

*On top of the C++ error handling mechanisms, SYCL adds a * ``sycl::exception`` *type for the exceptions thrown by the runtime. Everything else is standard C++ exception handling, so will be familiar to most developers. A slightly more complete example is provided in the figure below, where additional classes of exception are handled, as well as the program being ended by returning from main(). On top of the C++ error handling mechanisms, SYCL adds a * ``sycl::exception`` *type for the exceptions thrown by the runtime. Everything else is standard C++ exception handling, so will be familiar to most developers. A slightly more complete example is provided in the figure below, where additional classes of exception are handled, as well as the program being ended by returning from main().*

Figure. *Pattern to catch exceptions from a block of code*

```
try{
  buffer<int> B{ range{16} };

  // ERROR: Create sub-buffer larger than size of parent buffer
  // An exception is thrown from within the buffer constructor
  buffer<int> B2(B, id{8}, range{16});

} catch (sycl::exception &e) {
  // Do something to output or handle the exception
  std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
  return 1;
} catch (std::exception &e) {
  std::cout << "Caught std exception: " << e.what() << "\n";
  return 2;
} catch (...) {
  std::cout << "Caught unknown exception\n";
  return 3;
}

return 0;

Example output:
Caught sync SYCL exception: Requested sub-buffer size exceeds the
size of the parent buffer -30 (CL_INVALID_VALUE)
```

Asynchronous Error Handling

Asynchronous errors are detected by the SYCL runtime (or an underlying backend), and the errors occur independently of execution of commands in the host program. The errors are stored in lists internal to the SYCL runtime and only released for processing at specific points that the programmer can control. There are two topics that we need to discuss to cover handling of asynchronous errors

1. **The asynchronous handler***that is invoked when there are outstanding asynchronous errors to process*

2. **When***the asynchronous handler is invoked The Asynchronous Handle*

*The asynchronous handler is a function that the application defines, which is registered with SYCL contexts and/or queues. At the times defined by the next section, if there are any unprocessed asynchronous exceptions that are available to be handled, then the asynchronous handler is invoked by the SYCL runtime and passed a list of these exceptions. The asynchronous handler is passed to a context or queue constructor as a*std::function*and can be defined in ways such as a regular function, lambda, or functor, depending on our preference. The handler must accept a*sycl::exception_list*argument, such as in the example handler shown in the figure below*

Figure. *Example asynchronous handler implementation defined as a lambda*

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
  for (auto &e : elist) {
    try{ std::rethrow_exception(e); }
    catch ( sycl::exception& e ) {
      std::cout << "ASYNC EXCEPTION!!\n";
      std::cout << e.what() << "\n";
```

```
    }
  }
};
```

In the figure above, the `std::rethrow_exception` *followed by catch of a specific exception type provides filtering of the type of exception, in this case to the only* `sycl::exception`*. We can also use alternative filtering approaches in C++ or just choose to handle all exceptions regardless of the type The handler is associated with a queue or context (low-level detail covered more in* Chapter 6*) at construction time. For example, to register the handler defined in the figure above with a queue that we are creating, we could write* `queue my_queue{ gpu_selector{}, handle_async_error }` *Likewise, to register the handler defined in the figure above with a context that we are creating, we could write* `context my_context{ handle_async_error }` *Most applications do not need contexts to be explicitly created or managed (they are created behind the scenes for us automatically), so if an asynchronous handler is going to be used, most developers should associate such handlers with queues that are being constructed for specific devices (and not explicit contexts).*

NOTE: *In defining asynchronous handlers, most developers should define them on queues unless already explicitly managing contexts for other reasons.*

If an asynchronous handler is not defined for a queue or the queue's parent context and an asynchronous error occurs on that queue (or in the context) that must be processed, then the default asynchronous handler is invoked. The default handler operates as if it was coded as shown in the figure below.

Figure. *Example of how the default asynchronous handler behaves*

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
  for (auto &e : elist) {
    try{ std::rethrow_exception(e); }
    catch ( sycl::exception& e ) {
        // Print information about the asynchronous exception
    }
  }

  // Terminate abnormally to make clear to user
  // that something unhandled happened
  std::terminate();
};
```

The default handler should display some information to the user on any errors in the exception list and then will terminate the application abnormally, which should also cause the operating system to report that termination was abnormal.

What we put within an asynchronous handler is up to us. It can range from logging of an error to application termination to recovery of the error condition so that an application can continue executing normally.

The common case is to report any details of the error available by calling `sycl::exception::what()`*, followed by termination of the application. Although it's up to us to decide what an asynchronous handler does internally, a common mistake is to print an error message (that may be missed in the noise of other messages from the program), followed by completion of the handler function. Unless we have error management principles in place that allow us to recover known program state and to be confident that it's safe to continue execution, we should consider terminating the application within our asynchronous handler function(s).*

This reduces the chance that incorrect results will appear from a program where an error was detected, but where the application was inadvertently allowed to continue with execution regardless. In many programs, abnormal termination is the preferred result once we have experienced asynchronous exceptions.

Example: SYCL Throw on Zero Sized Object

The source code below shows how the SYCL handler will produce an error when a zero-sized object is passed.

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

template <bool non_empty>
static void fill(sycl::buffer<int> buf, sycl::queue & q) {
    q.submit([&](sycl::handler & h) {
        auto acc  = sycl::accessor { buf, h, sycl::read_write };
        h.single_task([=]() {
            if constexpr(non_empty) {
                acc[0] = 1;
            }
        }
        );
    }
    );
    q.wait();



}

int main(int argc, char *argv[]) {
    sycl::queue q;
    sycl::buffer<int, 1> buf_zero ( 0 );

    fprintf(stderr, "buf_zero.count() = %zu\n", buf_zero.get_count());
    fill<false>(buf_zero, q);
    fprintf(stdout, "PASS\n");

    return 0;
}
```

When the application encounters the zero-sized object at runtime, the program aborts and produces an error message:

```
$ dpcpp zero.cpp
$ ./a.out
buf_zero.count() = 0
submit...
terminate called after throwing an instance of 'cl::sycl::invalid_object_error'
  what():  SYCL buffer size is zero. To create a device accessor, SYCL buffer size must be
greater than zero. -30 (CL_INVALID_VALUE)
Aborted (core dumped)
```

The programmer can then locate the programming error by catching the exception in the debugger and looking at the backtrace for the source line that triggered the error.

Example: SYCL Throw on Illegal Null Pointer

Consider code that does the following:

```
deviceQueue.memset(mdlReal, 0, mdlXYZ \* sizeof(XFLOAT));

deviceQueue.memcpy(mdlImag, 0, mdlXYZ \* sizeof(XFLOAT)); // coding
error
```

The compiler will not flag the bad (null pointer) value specified in `deviceQueue.memcpy`. This error will not be caught until runtime.

```
terminate called after throwing an instance of 'cl::sycl::runtime_error'

what(): NULL pointer argument in memory copy operation. -30
(CL_INVALID_VALUE)

Aborted (core dumped)
```

The example code that follows shows a way the user can control the format of the exception output when it is detected at runtime on a given queue, implemented in a standalone program that demonstrates the null pointer error.

```
#include "stdlib.h"
#include "stdio.h"
#include <cmath>
#include <signal.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <CL/sycl.hpp>

#define XFLOAT float
#define mdlXYZ 1000
#define MEM_ALIGN 64

int main(int argc, char *argv[])
{
      XFLOAT *mdlReal, *mdlImag;

      cl::sycl::property_list propList =
cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};
cl::sycl::queue deviceQueue(cl::sycl::gpu_selector { }, [&](cl::sycl::exception_list eL)
      {
            bool error = false;
            for (auto e : eL)
            {
                  try
                  {
                        std::rethrow_exception(e);
                  } catch (const cl::sycl::exception& e)
                  {
                        auto clError = e.get_cl_code();
                        bool hascontext = e.has_context();
                        std::cout << e.what() << "CL ERROR CODE : " << clError << std::endl;
                        error = true;
                        if (hascontext)
                        {
                              std::cout << "We got a context with this exception" << std::endl;
                        }
                  }
            }
            if (error) {
                  throw std::runtime_error("SYCL errors detected");
            }
      }, propList);

      mdlReal   = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);
```

```
    mdlImag   = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);

    deviceQueue.memset(mdlReal, 0, mdlXYZ * sizeof(XFLOAT));
    deviceQueue.memcpy(mdlImag, 0, mdlXYZ * sizeof(XFLOAT));    // coding error

    deviceQueue.wait();

    exit(0);
}
```

Resources

For a a guided approach to debugging SYCL exceptions from incorrect use of the SYCL* API, see the Guided Matrix Multiplication Exception Sample.

To troubleshoot your applications that use OpenMP* or the SYCL* API with extensions to offload resources, see Troubleshoot Highly Parallel Applications.

Optimize Offload Performance

Offload performance optimization basically boils down to three tasks:

1. Minimize the number and size of data transfers to and from the device while maximizing execution time of the kernel on the device.
2. When possible, overlap data transfers to/from the device with computation on the device.
3. Maximize the performance of the kernel on the device.

While it is possible to take explicit control of data transfers in both OpenMP* offload and SYCL*, you also can allow this to happen automatically. In addition, because the host and offload device operate mostly asynchronously, even if you try to take control over data transfers, the transfers may not happen in the expected order, and may take longer than anticipated. When data used by both the device and the host is stored in unified shared memory (USM), there is another transparent layer of data transfers happening that also can affect performance.

Resources:

• Intel oneAPI GPU Optimization Guide
• Intel oneAPI FPGA Optimization Guide

Buffer Transfer Time vs Execution Time

Transferring any data to or from an offload device is relatively expensive, requiring memory allocations in user space, system calls, and interfacing with hardware controllers. Unified shared memory (USM) adds to these costs by requiring that some background process keeps memory being modified on either the host or offload device in sync. Furthermore, kernels on the offload device must wait to run until all the input or output buffers they need to run are set up and ready to use.

All this overhead is roughly the same no matter how much information you need to transfer to or from the offload device in a single data transfer. Thus, it is much more efficient to transfer 10 numbers in bulk rather than one at a time. Still, every data transfer is expensive, so minimizing the total number of transfers is also very important. If, for example, you have some constants that are needed by multiple kernels, or during multiple invocations of the same kernel, transfer them to the offload device once and reuse them, rather than sending them with every kernel invocation. Finally, as might be expected, single large data transfers take more time than single small data transfers.

The number and size of buffers sent is only part of the equation. Once the data is at the offload device, consider how long the resulting kernel executes. If it runs for less time than it takes to transfer the data to the offload device, it may not be worthwhile to offload the data in the first place unless the time to do the same operation on the host is longer than the combined kernel execution and data transfer time.

Finally, consider how long the offload device is idle between the execution of one kernel and the next. A long wait could be due data transfer or just the nature of the algorithm on the host. If the former, it may be worthwhile to overlap data transfer and kernel execution, if possible.

In short, execution of code on the host, execution of code on the offload device, and data transfer is quite complex. The order and time of such operations isn't something you can gain through intuition, even in the simplest code. You need to make use of tools like those listed below to get a visual representation of these activities and use that information to optimize your offload code.

Intel® VTune™ Profiler

In addition to giving you detailed performance information on the host, VTune can also provide detailed information about performance on a connected GPU. Setup information for GPUs is available from the Intel VTune Profiler User Guide.

Intel VTune Profiler's GPU Offload view gives you an overview of the hotspots on the GPU, including the amount of time spent for data transfer to and from each kernel. The GPU Compute/Media Hotspots view allows you to dive more deeply into what is happening to your kernels on the GPU, such as by using the **Dynamic Instruction Count** to view a micro analysis of the GPU kernel performance. With these profiling modes, you can observe how data transfer and compute occur over time, determine if there is enough work for a kernel to run effectively, learn how your kernels use the GPU memory hierarchy, and so on.

Additional details about these analysis types is available from the Intel VTune Profiler User Guide. A detailed look at optimizing for GPU using VTune Profiler is available from the Optimize Applications for Intel GPUs with Intel VTune Profiler page.

You can also use Intel VTune Profiler to capture kernel execution time. The following commands provide light-weight profiling results:

- Collect

  - Level zero backend: `vtune -collect-with runss -knob enable-gpu-level-zero=true -finalization-mode=none -app-working-dir <app_working_dir> <app>`
  - OpenCL™ backend: `vtune -collect-with runss -knob collect-programming-api=true -finalization-mode=none -r <result_dir_name> -app-working-dir <app_working_dir> <app>`
- Report: `vtune --report hotspots --group-by=source-computing-task --sort-desc="Total Time" -r <result_dir_name>`

Intel® Advisor

Intel® Advisor provides two features that can help you get the improved performance when offloading computation to GPU:

- Offload Modeling can watch your host OpenMP* program and recommend parts of it that would be profitably offloaded to the GPU. It also allows you to model a variety of different target GPUs, so that you can learn if offload will be profitable on some but not others. Offload Advisor gives detailed information on what factors may bound offload performance.
- GPU Roofline analysis can watch your application when it runs on the GPU, and graphically show how well each kernel is making use of the memory subsystem and compute units on the GPU. This can let you know how well your kernel is optimized for the GPU.

To run these modes on an application that already does some offload, you need to set up your environment to use the OpenCL™ device on the CPU for analysis. Instructions are available from the Intel Advisor User Guide.

Offload modeling does not require that you have already modified your application to use a GPU - it can work entirely on host code.

Resources:

- Intel Advisor Cookbook: GPU Offload
- Get Started with Offload Modeling
- Get Started with GPU Roofline

Offload API call Timelines

If you do not want to use Intel® VTune™ Profiler to understand when data is being copied to the GPU, and when kernels run, onetrace, ze_tracer, cl_tracer, and the Intercept Layer for OpenCL™ Applications give you a way to observe this information /(although, if you want a graphical timeline, you'll need to write a script to visualize the output/). For more information, see oneAPI Debug Tools, Trace the Offload Process, and Debug the Offload Process.

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 Modeling perspective of the Intel® Advisor.

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

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

See Get Started: Identify High-impact Opportunities to Offload to GPU for quick steps to ramp up with the Offload Modeling perspective. For more resources about modeling performance of your application on GPU platforms, see Offload Modeling Resources for Intel® Advisor Users.

Offload Kernels

After identifying kernels that are suitable for offload, employ SYCL* or OpenMP* to offload the kernel onto the device. Consult the previous chapters as an information resource.

Optimize Your SYCL* Applications

oneAPI enables functional code that can execute on multiple accelerators such as CPU, GPU, and FPGA. 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 to 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 FPGA Optimization Guide for Intel© oneAPI Toolkits. 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. SYCL-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 SYCL-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 simple exit condition loops – these are loops that 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.
- Consider employing the `[[intel::kernel_args_restrict]]` attribute to a kernel. The attribute allows the compiler to ignore dependencies between accessor arguments in the kernel. In turn, ignoring accessor argument dependencies allows the compiler to perform more aggressive optimizations and potentially improve the performance of the kernel.

SYCL-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 set the work-group size. The compiler can take advantage of this information 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.

oneAPI Library Compatibility

oneAPI applications may include dynamic libraries at runtime that require compatibility across release versions of Intel tools. Intel oneAPI Toolkits and component products use semantic versioning to support compatibility.

The following policies apply to APIs and ABIs delivered with Intel oneAPI Toolkits.

NOTE oneAPI applications are supported on 64-bit target devices.

- New Intel oneAPI device drivers, oneAPI dynamic libraries, and oneAPI compilers will not break previously deployed applications built with oneAPI tools. Current APIs will not be removed or modified without notice and an iteration of the major version.

- Developers of oneAPI applications should ensure that the header files and libraries have the same release version. For example, an application should not use 2021.2 Intel® oneAPI Math Kernel Library header files with 2021.1 Intel oneAPI Math Kernel Library.
- New dynamic libraries provided with the Intel compilers will work with applications built by older versions of the compilers (this is commonly referred to as *backward compatibility*). However, the converse is not true: newer versions of the oneAPI dynamic libraries may contain routines that are not available in earlier versions of the library.
- Older dynamic libraries provided with the oneAPI Intel compilers will not work with newer versions of the oneAPI compilers.

Developers of oneAPI applications should ensure that thorough application testing is conducted to ensure that a oneAPI application is deployed with a compatible oneAPI library.

SYCL* Extensions

SYCL extensions provide the ability to rapidly experiment, innovate, develop, and establish a virtuous cycle into open standards bodies—like SYCL from the Khronos Group to facilitate cross-architecture systems. To access extensions that work with the Intel® oneAPI DPC++ Compiler, see SYCL Extensions on GitHub.

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.

DPC++

An open source project is adding SYCL* support to the LLVM C++ compiler.

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.

SYCL

A standard for a cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file.

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.

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

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