Intel® oneAPI Programming Guide

Intel Corporation

www.intel.com

Notices and Disclaimers

Contents

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.

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

1 Introduction

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

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

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

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

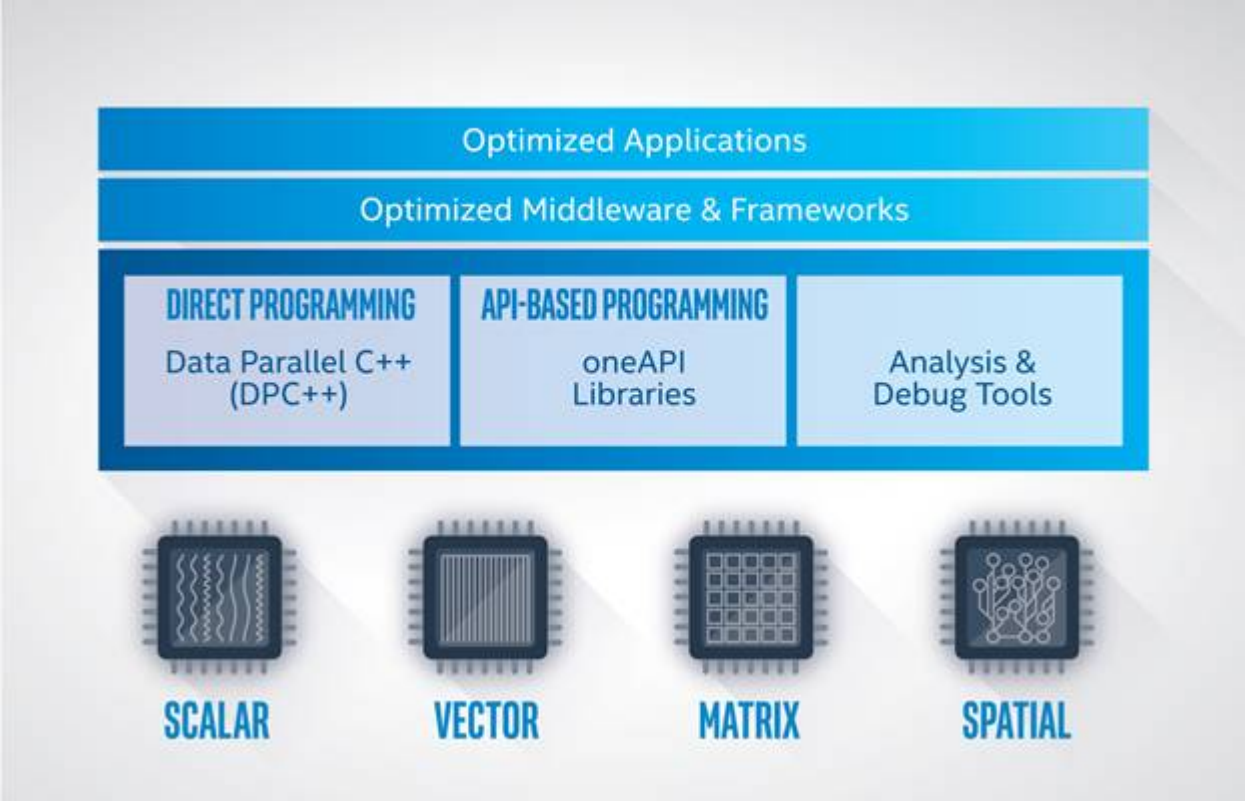
oneAPI also supports programming on CPUs and accelerators using the OpenMP* offload feature with existing C/C++ or Fortran code.

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

Learn more about the oneAPI initiative and programming model at oneapi.com. The site includes the oneAPI Specification, DPC++ Language Guide and API Reference, open source library documentation, 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. DPC++ is based on industry standards and open specifications to encourage ecosystem collaboration and innovation.



As shown in the figure above, applications that take advantage of the oneAPI programming model can execute on multiple target hardware platforms ranging from CPU to FPGA. 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 DPC++, can use the **Intel DPC++ Compatibility Tool** to help migrate their existing projects to 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 DPC++ application can be analyzed and debugged to ensure performance, stability, and energy efficiency goals are achieved using tools such as **Intel® VTune™ Profiler** or **Intel® Advisor**.

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.
- Install from containers or repositories: install the oneAPI toolkits from one of several supported containers or repositories.

- Pre-installed in the Intel® DevCloud: use a free development sandbox for access to the latest Intel SVMS hardware and select oneAPI tools.

Learn more about each of these distribution channels from the oneAPI product website on the Intel Developer Zone.

About This Guide

This document provides:

- oneAPI Programming Model: An introduction to the oneAPI programming model (platform, execution, memory, and kernel programming)
- 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
- 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.)

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:
  - DPC++ Language Guide and API Reference
  - SYCL* Specification (for version 1.2.1)
  - Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL (book)

2 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 methods of heterogenous computing: Data Parallel C++ (DPC++) and OpenMP* for C, C++, and Fortran.

DPC++ is an open source language based on modern C++ and the SYCL* language from the Khronos* Group with some additional Intel extensions. 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 5.0 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 (Beta) with OpenMP offload support is available as part of the Intel oneAPI HPC Toolkit.

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

Data Parallel C++ (DPC++)

Data Parallel C++ (DPC++) is a high-level language designed for data parallel programming productivity.

> **NOTE**
> One of the primary motivations for DPC++ is to provide a higher-level programming language than OpenCL™ C code, which it is based upon. Readers familiar with OpenCL programs will see many similarities to and differences from OpenCL code.

Simple DPC++ Sample Code

The best way to introduce DPC++ is through an example. Since DPC++ 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 DPC++, 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.

```
#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 8 represents data that will be offloaded to the device, and the `queue` class on line 10 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 DPC++ 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 is capable of writing 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. DPC++ 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 DPC++ 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. DPC++ knows this buffer is written by the accelerator, so the `host_accessor` constructor (line 18) blocks 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 DPC++ Resources

This introduction to DPC++ 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 DPC++:

- Explore DPC++ 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 DPC++ Language Guide and API Reference with descriptions of classes and their interfaces. It also provides details on the four programming models of DPC++ and SYCL - platform model, execution model, memory model, and kernel programming model.
- The DPC++ Essentials training course is a guided learning path 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 DPC++.

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

The Intel® oneAPI DPC++/C++ Compiler and the Intel® Fortran Compiler (Beta) 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.

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++

This C++ code snippet 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

```
device(scalar-integer-expression)
map(alloc | to | from | tofrom: list)
if(scalar-expr)
```

Use the target update directive 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++

```
icx -fiopenmp -fopenmp-targets=spir64 code.c
```

Fortran

```
ifx -fiopenmp -fopenmp-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: DPC++ or OpenMP.
  - The ISO3DFD sample refers to Three-Dimensional Finite-Difference Wave Propagation in Isotropic Media. It is a three-dimensional stencil to simulate a wave propagating in a 3D isotropic medium and 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 provides details on using Intel's compilers with OpenMP offload, including lists of supported options and example code.
- 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 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 this, select a queue and initialize its device with one of the following:

- `default_selector`
- `cpu_selector`
- `gpu_selector`
- `fpga_selector`

If `default_selector` is used, the kernel runs based on the value of the `SYCL_DEVICE_TYPE` environment variable. The value of this environment variable can be: `CPU`, `GPU`, `ACC`, or `HOST`.

> **NOTE** The `SYCL_DEVICE_TYPE` will be deprecated and replaced in a future release.

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. If such a device is not available, then the runtime system throws an error that the requested device type is not available. This error can be thrown in the situation where an ahead of time 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.

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.

oneAPI Development Environment Setup

3

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 might prefer to use the modulefiles included with the oneAPI toolkit installation to initialize the development environment variables. The oneAPI modulefiles are only supported on Linux and are provided as an alternative to using the setvars and vars scripts described above. In general, users should not mix the usage of 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 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 `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` 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 of the `<component>\latest\env\vars.bat` scripts installed on the system. Changes made to the environment using the Windows `set` command can be seen after running these scripts.

> **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 can displayed using the `--help` option. For example:

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

Of particular note are the `--config=file` argument and the ability to include additional arguments that will be passed to the `vars.bat` scripts that are called by the `setvars.bat` script.

The `--config=file` argument provides the ability to limit environment initialization to a specific set of oneAPI components, as well as providing a way to initialize the environment for a specific component version. For example, to limit environment setup just the Intel® Integrated Performance Primitives (Intel® IPP) library and the Intel® oneAPI Math Kernel Library (oneMKL), create a config file that tells the `setvars.bat` script to only configure the environment variables 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 the `ia32` and `intel64` arguments, which are used by the Intel compilers and the Intel IPP and oneMKL libraries to indicate the application target architecture.

If both Microsoft Visual Studio* 2017 and Visual Studio 2019 are installed on your system, you can specify which of these two Visual Studio environments should be initialized as part of the oneAPI environment initialization by adding either the `vs2017` or the `vs2019` argument to the `setvars.bat` command line. By default, the most recent version of Visual Studio is located and initialized.

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

How to Run

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

How to Verify

After executing `setvars.bat`, verify success by searching for the SETVARS_COMPLETED environment variables. If `setvars.bat` was successful, SETVARS_COMPLETED 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`, then 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 become too long due to redundant path references, especially the `%PATH%` environment variable.

This can be overridden by passing `setvars.bat` a `--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
 :: 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` overwrites it. 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 on 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 and no additional user action is required.

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.

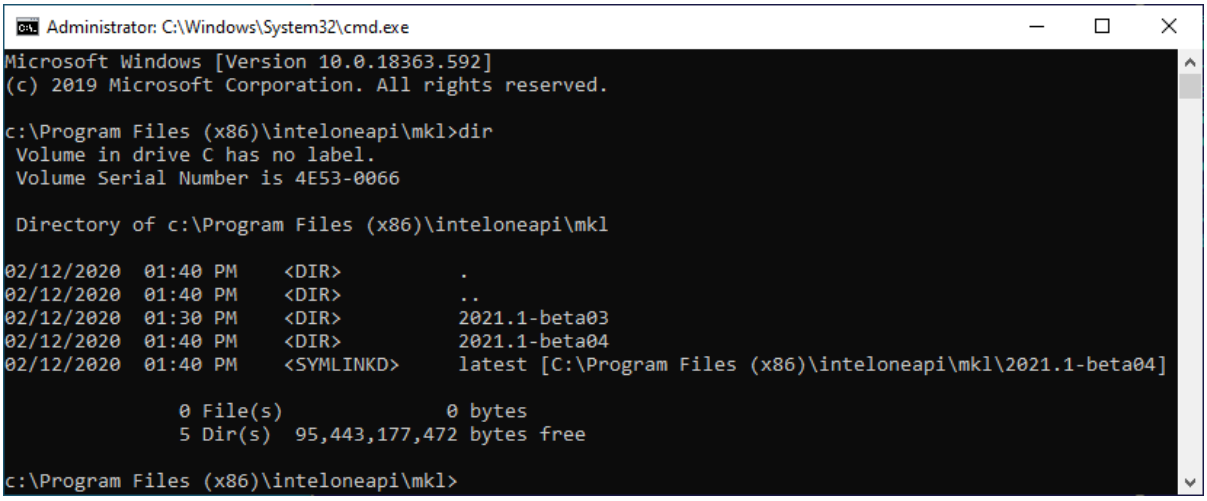
The procedure below describes how to use a configuration file to manage environment variables.

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:

```
[Component] directory
- [1.0] version directory
- [1.1] version directory
- [1.2] version directory
- [latest -> 1.2] symlink or shortcut
```

For example:



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.

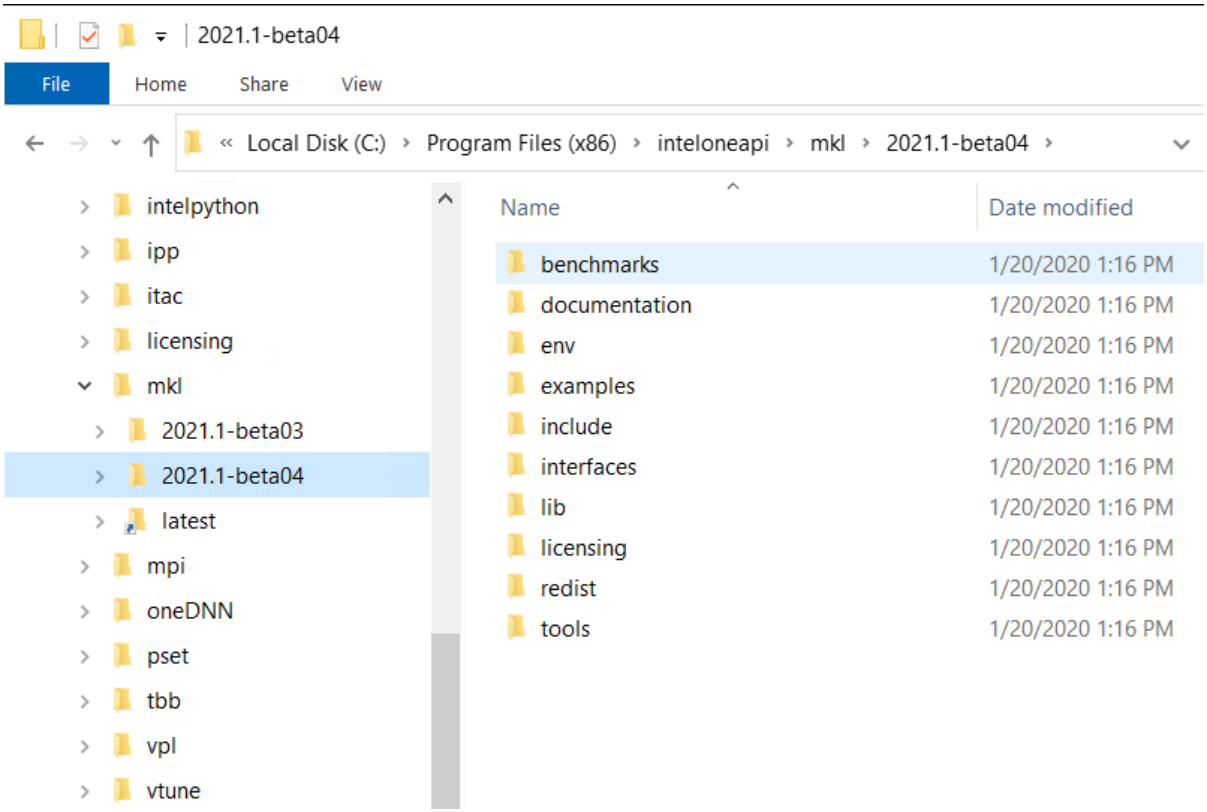
The "key=value" pair "default=exclude" is a special case. When included, it will exclude executing ALL env\vars.bat scripts, except those that are listed in the config file. See the examples below.

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.

To explain this further, the sample below shows two versions of oneMKL installed: 2021.1-beta03 and 2021.1-beta04. There is a shortcut to 2021-beta04 indicating that it is the latest version, so by default setvars.bat will execute the 2021.1-beta04 vars.bat script in the mkl folder.



Specify a Specific Version

To direct setvars.bat to execute the <install-dir>\mkl\2021.1-beta03\env\vars.bat script, add mkl=2021.1-beta03 to your config file.

This instructs `setvars.bat` to execute the `env\vars.bat` script located in the `2021.1-beta03` 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-beta03 version of oneMKL:

```
mkl=2021.1-beta03
ipp=exclude
```

In this example:

- `setvars.bat` WILL execute the oneMKL 2021.1-beta03 `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 oneMKL and Intel IPP component `env\vars.bat` scripts, use this config file:

```
default=exclude
mkl=2021.1-beta03
ipp=latest
```

In this example:

- `setvars.bat` WILL execute the oneMKL 2021.1-beta03 `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*

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

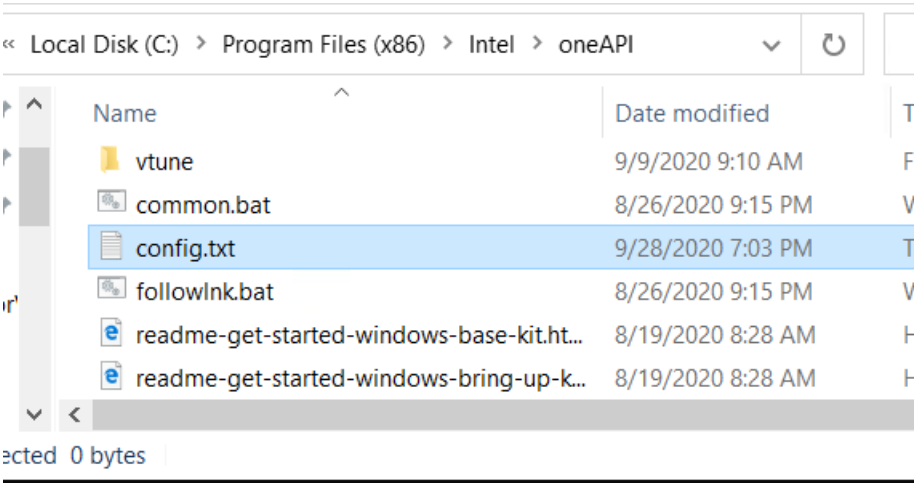
If `SETVARS_CONFIG` is undefined there will be no attempt to automatically run `setvars.bat` when Visual Studio is started. This is the default case, since the `SETVARS_CONFIG` variable is not defined by the oneAPI installer.

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.

Create a setvars Configuration File

Create a new text file that you will use as your `setvars` configuration file. In this example, the file is named `config.txt` and is located in the `C:\Program Files (x86)\Intel\oneAPI` folder.

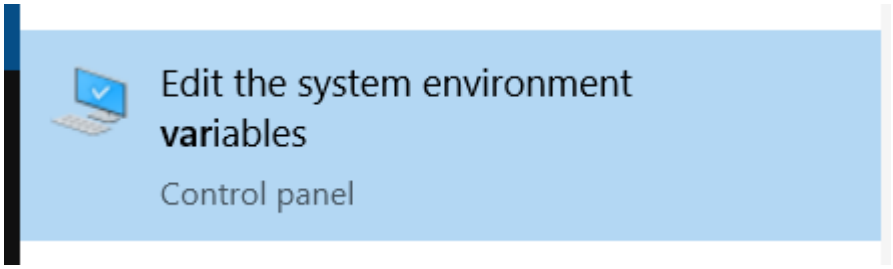


A `setvars` configuration file can have any name and can be saved to any location on your hard disk, as long as that location and the file are accessible and readable by Visual Studio. (A plug-in that was added to Visual Studio when you installed the oneAPI tools on your Windows system performs the SETVARS_CONFIG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

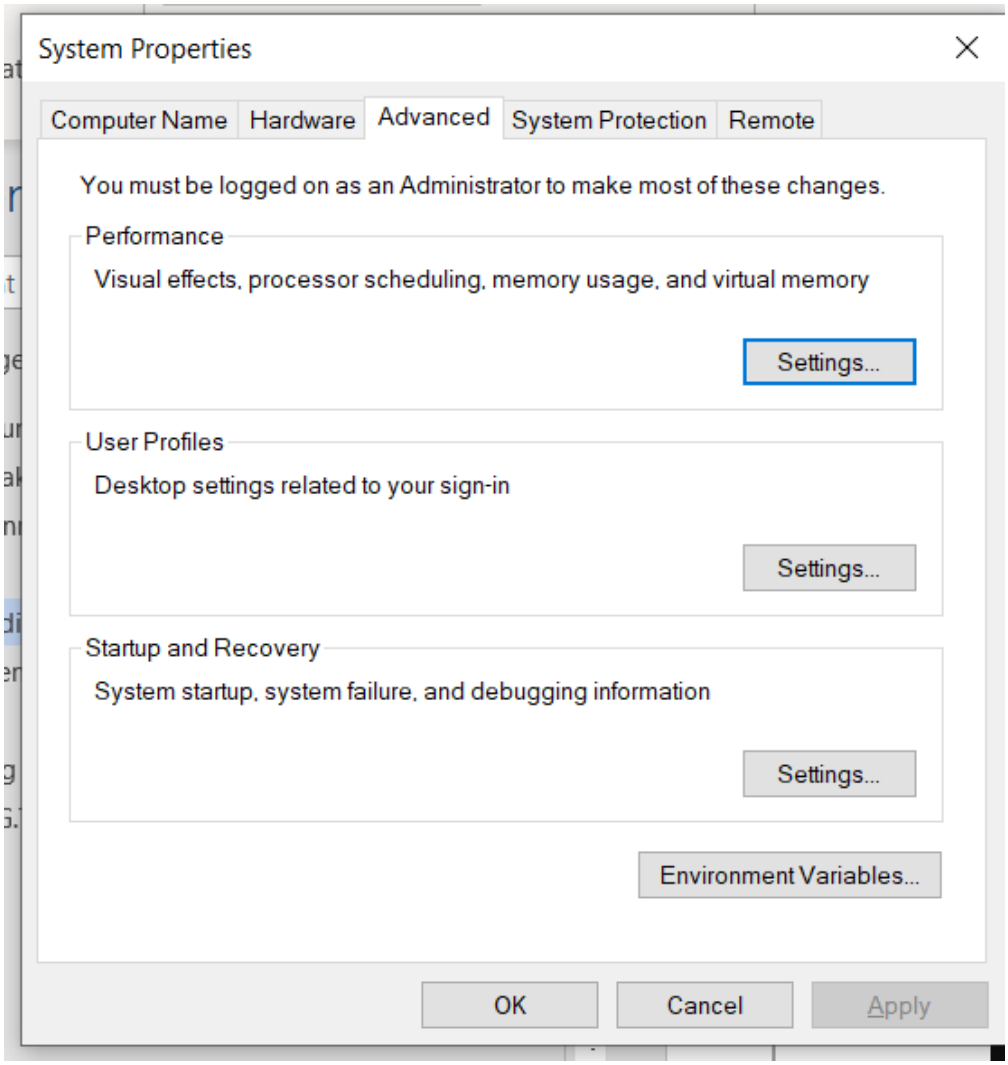
If you leave the `setvars` config file empty, the `setvars.bat` script will initialize your environment for *all* oneAPI tools that are installed on your system. This is equivalent to defining the `SETVARS_CONFIG` variable with an empty string. See Using a Config File with setvars.bat for details regarding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

Define the SETVARS_CONFIG Environment Variable

1. Press the Windows key on your keyboard, and then press the letter 's' to open the search window.
2. Enter variable in the search window.
3. Select **Edit the System Environment Variables**.

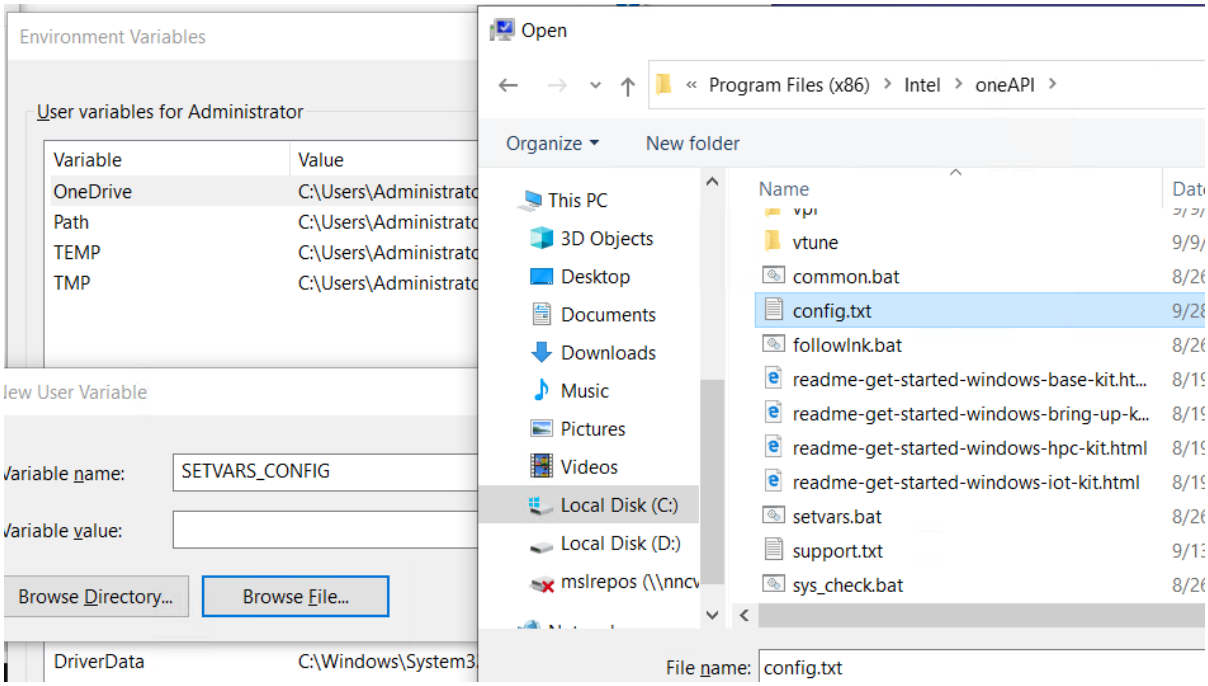


4. Click the **Environment Variables** button.



5. From the **User Variables** list, select `SETVARS_CONFIG`. Then click **Edit**.

   If `SETVARS_CONFIG` does not appear in your **User Variables** list, click **New**.
6. In the **Variable Name** field, enter `SETVARS_CONFIG`.
7. Click **Browse File**. Locate and click on the text file that you created.



8. Click **Open**.
9. Click **OK** to close the **User Variable** window.
10. Click **OK** to close the **Environment Variables** window.
11. Click **OK** to close the **System Properties** window.

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 `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` 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 of the `<component>/latest/env/vars.sh` scripts installed on the system. Changes made to the environment using the `env` command can be seen after sourcing these 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
```

Of particular note are the `--config=file` argument and the ability to include additional arguments that will be passed to the `vars.sh` scripts that are called by the `setvars.sh` script.

The `--config=file` argument provides the ability to limit environment initialization to a specific set of oneAPI components, as well as providing a way to initialize the environment for a specific component version. For example, to limit environment setup to just the Intel® Integrated Performance Primitives (Intel® IPP) library and the Intel® oneAPI Math Kernel Library (oneMKL), create a config file that tells the `setvars.sh` script to only configure the environment variables 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 the `ia32` and `intel64` arguments, which are used by the Intel compilers and the Intel IPP and oneMKL libraries to indicate 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
```

How to Verify

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

```
env | grep SETVARS_COMPLETED
```

Return value

```
SETVARS_COMPLETED=1
```

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 can be overridden by passing `setvars.sh` a `--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
:: initializing environment ...
(SNIP: lot of output)
:: oneAPI environment initialized ::
```

```
> source <install-dir>/setvars.sh
 :: 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
 :: initializing environment ...
(SNIP: lot of output)
 :: 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` overwrites it. 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 on 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

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.

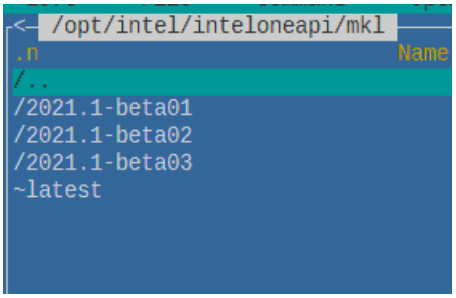
The procedure below describes how to use a configuration file to manage environment variables.

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:

```
[Component] directory
- [1.0] version directory
- [1.1] version directory
- [1.2] version directory
- [latest -> 1.2] symlink or shortcut
```

For example:



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.

To explain this further, the sample below shows two versions of oneMKL installed: 2021.1-beta03 and 2021.1-beta04. There is a symlink to 2021-beta04 indicating that it is the latest version, so by default `setvars.sh` will source the 2021.1-beta04 `vars.sh` script in the `mkl` folder.

```
/opt/intel/oneapi/mkl
|
|-- 2021.1-beta03
|   |-- env
|   |-- include
|   |-- interfaces
|   |-- lib
|   `-- tools
|
|-- 2021.1-beta04
|   |-- env
|   |-- include
|   |-- interfaces
|   |-- lib
|   `-- tools
|
`-- latest -> 2021.1-beta04
   |-- env
   |-- include
```

```
  |-- interfaces
  |-- lib
  `-- tools
```

Specify a Specific Version

To direct `setvars.sh` to source the `<install-dir>/mkl/2021.1-beta03/env/vars.sh` script, add `mkl=2021.1-beta03` to your config file.

This instructs `setvars.sh` to source on the `env/vars.sh` script located in the `2021.1-beta03` 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-beta03 version of oneMKL:

```
mkl=2021.1-beta03
ipp=exclude
```

In this example:

- `setvars.sh` WILL source the Intel one MKL 2021.1-beta03 `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 oneMKL and Intel IPP component `env/vars.sh` scripts, use this config file:

```
default=exclude
mkl=2021.1-beta03
ipp=latest
```

In this example:

- `setvars.sh` WILL source the oneMKL 2021.1-beta03 `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.

Create a setvars Configuration File

Create a new text file that you will use as your setvars configuration file. In this example, the file is named config.txt and is located in the users home folder. For example "${HOME}"/config/.txt or ~/config/.txt.

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*

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).

The oneAPI modulefile scripts are found in a `modulefiles` folder located 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 located 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. 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 the oneAPI installation and organizes them into a single directory of folders. Those folders are named for each installed oneAPI component that includes one or more modulefiles.

Each of these component-named folders contains symlinks pointing to all of the modulefiles that are provided by that oneAPI component. These symlinks are organized as versioned modulefiles. Each component folder includes (at minimum) a "latest" version modulefile that will be selected, by default, when loading a component modulefile without specifying a version label.

The default name for the top-level directory of oneAPI modulefiles is `modulefiles` and is located in the oneAPI installation folder. For example: `/opt/intel/oneapi/modulefiles`

Creating the `modulefiles` Directory

Run the `modulefiles-setup.sh` script.

> **NOTE**
> Because the `modulefiles-setup.sh` script creates a folder in the oneAPI toolkit installation folder, it may be necessary to run it as root or sudo.

After the `modulefiles-setup.sh` script has been run, an organization like the following is created in the top-level `modulefiles` folder:

```
[oneapi_root]
  - [modulefiles]
  -   - [compiler]
  -   -   - [1.0] -> [oneapi_root/compiler/1.0/modulefiles/compiler] # symlink named 1.0 points
to compiler modulefile
  -   - [compiler32]
  -   -   - [1.0] -> [oneapi_root/compiler/1.0/modulefiles/compiler32] # symlink named 1.0
points to compiler32 modulefile
  -   - [ipp]
  -   -   - [1.1] -> [oneapi_root/ipp/1.1/modulefiles/ipp]
  -   -   - [1.2] -> [oneapi_root/ipp/1.2/modulefiles/ipp]
  -   - [ipp32]
  -   -   - [1.1] -> [oneapi_root/ipp/1.1/modulefiles/ipp32]
  -   -   - [1.2] -> [oneapi_root/ipp/1.2/modulefiles/ipp32]
 etc…
```

Now, update the MODULEFILESPATH to point to this new `modulefiles` folder.

Installing the `module` Utility onto Your System

The instructions below will help 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 (as of this writing, version 8.4 or later is required):

```
$ tclsh
% puts $tcl_version
8.6
% exit
```

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

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

> **NOTE** Initialization for Bourne-compatible shells should work with the `source` command shown above. Additional 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.

Include sourcing the `module` alias init script (`. . ./modules/init/sh`) in one of the startup scripts 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`) on to 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`, this 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 a oneAPI product does 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 installer for oneAPI toolkits applies version folders to individual oneAPI tools and libraries in the form of versioned subdirectories inside the top-level directory for each tool or library. These versioned component folders are used to create the versioned `modulefiles`. This is essentially what the `modulefiles-setup.sh` script does; it organizes the symbolic links it creates in the top-level `$(ONEAPI_ROOT)/modulefiles` folder as `<modulefile-name>/version`, so the respective `modulefiles` 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 of the available tool and library `modulefiles` into a single top-level `modulefiles` folder for oneAPI. This means that the `modulefiles` for oneAPI are not moved or touched. As a consequence of this approach, the `${ModulesCurrentModulefile}` variable points to a *symlink* to each `modulefile`, not to the actual `modulefile` located in the respective installation folders. Because of this, these `modulefiles` are *not* able to reference other folders or files within the modulefile's component folder, because of the symlinks.

Thus, each modulefile for oneAPI uses a statement such as:

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

to get a reference to the original `modulefile` located in the respective install directory.

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 and include checks to prevent you from inadvertently loading two versions of the same `modulefile`.

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 that a specific dependent module can be manually preloaded 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 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://support.pawsey.org.au/documentation/display/US/Sample+modulefile+for+Environment+Modules
- https://www.chpc.utah.edu/documentation/software/modules-advanced.php

Compile and Run oneAPI Programs

4

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 is invoked using `dpcpp` for DPC++ applications and `icpx -fiopenmp` (Linux*) or `icx /Qiopenmp` (Windows*) for C++ applications with OpenMP* offload. For more information, see Invoking the Compiler in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

The `dpcpp` driver has different compatibilities on different OS hosts. Linux provides GCC*-style command line options and Windows provides Microsoft* Visual C++ compatibility with 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 and OpenMP* Options and Parallel Processing Options section includes options specific to DPC++ 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 DPC++ 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 DPC++.

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.

```
#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:

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

   On Windows:

```
dpcpp -I${MKLROOT}/include /EHsc -c axpy.cpp /Foaxpy.obj
```
3. Link the application using the following command:

   On Linux:

```
dpcpp 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:

```
dpcpp 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.

```
dpcpp 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

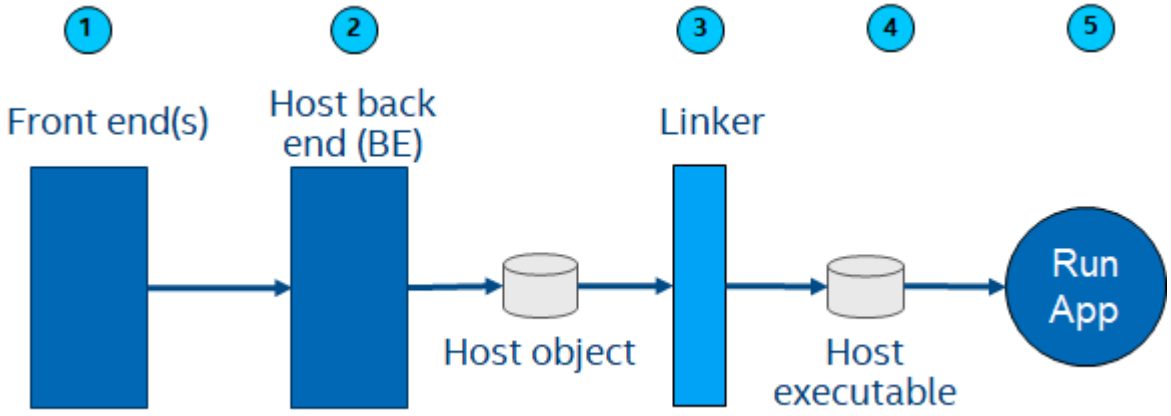
When you create a program with offload, the compiler must generate code for both the host and the device. oneAPI tries to hide this complexity from the developer. The developer simply compiles a DPC++ application with `dpcpp`, and the same compile command generates host and device code. Behind the scenes, `dpcpp` generates the host code, generates the device code, and then embeds the device code in the host binary. At runtime, the operating system starts the host application. If it has offload, the DPC++ runtime loads the device code into the device and then starts the application.

For device code, two options are available: Just-in-Time (JIT) compilation and ahead-of-time (AOT), 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

The traditional compilation flow is a standard compilation like the one used for C, C++, or other languages.

The compilation phases are shown in the following diagram:



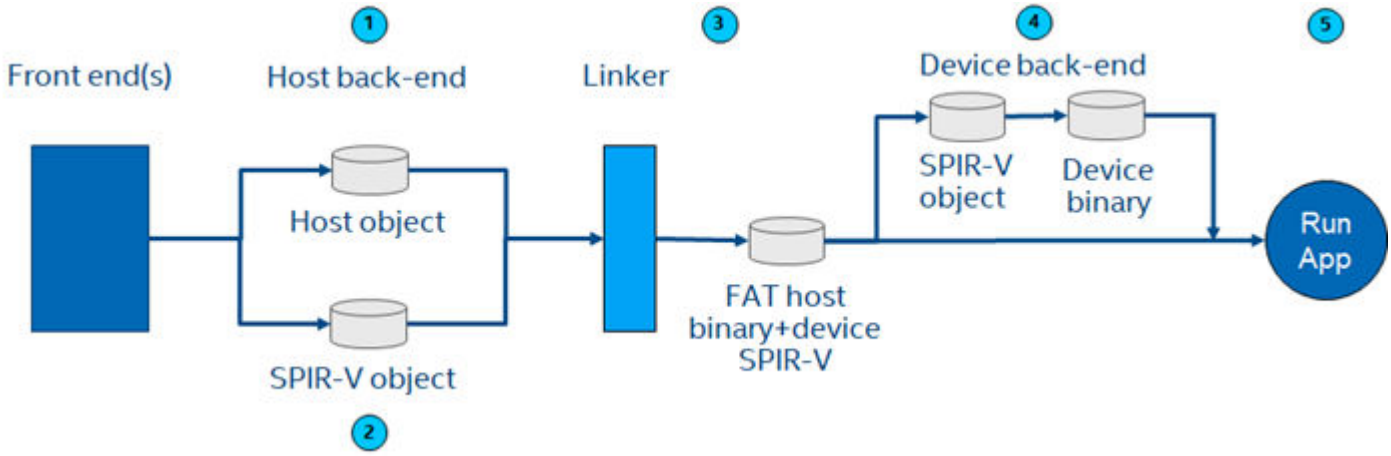
1. The front-end translates the source into an intermediate representation and then passes that representation to the back-end.
2. The back-end translates the intermediate representation to object code and emits an object file (`host.obj` on Windows*, `host.o` on Linux*).
3. One or more object files are passed to the linker.
4. The linker creates an executable.
5. The application runs.

JIT Compilation Flow

Just-in-Time (JIT) compilation begins like a standard compilation, such as the one used for C, C++, or other languages, but also includes steps for device code. 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. However, performance may be worse because compilation occurs when the application runs. Larger applications with significant amounts of device code may notice performance impacts.

> **NOTE**
> JIT compilation is not supported for FPGA devices.

The compilation phases are shown in the following diagram:



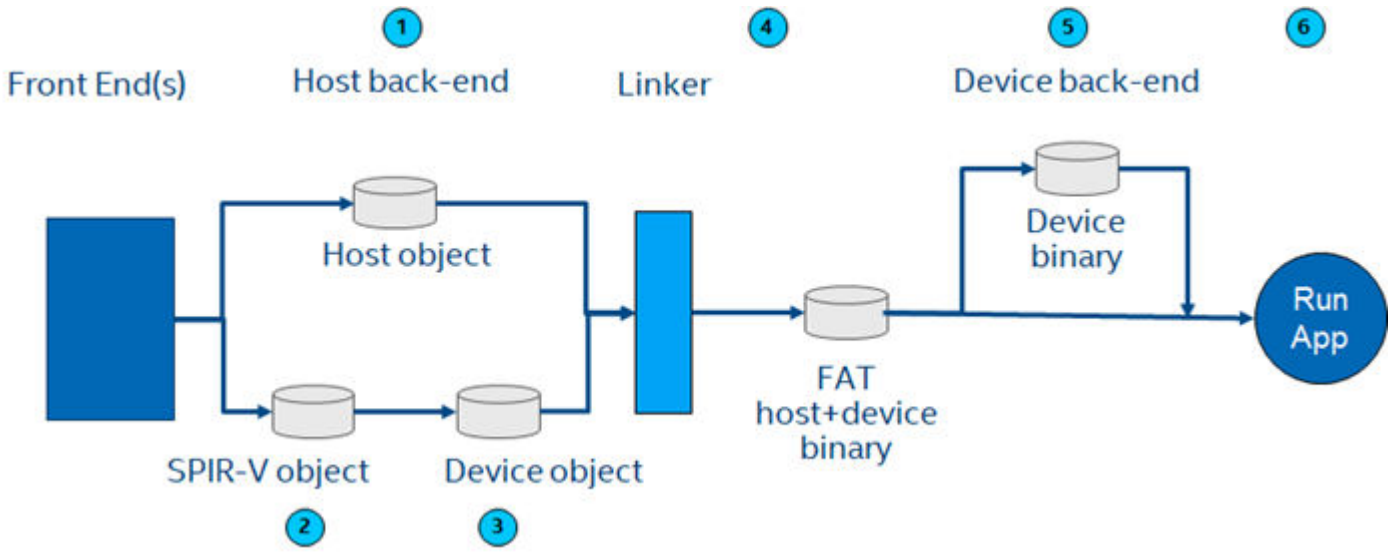
1. The host code is translated to object code by the back-end.
2. The device code is translated to SPIR-V.
3. The linker combines the host object code and the device SPIR-V into a fat binary containing host executable code with SPIR-V device code embedded in it.
4. At runtime:
   a. The device runtime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

AOT compilation begins like a standard compilation, such as the one used for C, C++, or other languages, but also includes steps for device code. Unlike the JIT flow, the AOT flow translates the device code before the executable is created. 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 flow is recommended when debugging your application as it speeds up the debugging cycle.

The compilation phases are shown in the following diagram:

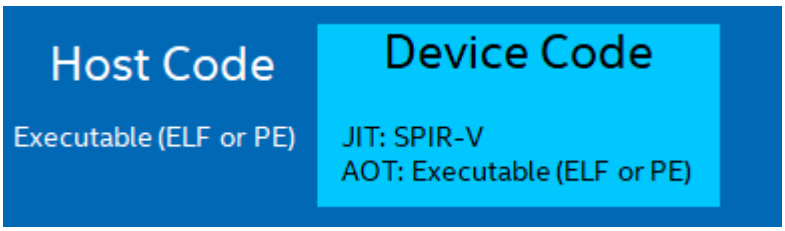


1. The host code is translated to object code by the back-end.
2. The device code is translated to SPIR-V (`SPIR-V.obj`).
3. The SPIR-V for the device is translated to a device code object using the device specified by the user on the command-line.
4. The linker combines the host object code and the 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.

> **NOTE**
> JIT compilation is not supported for FPGA devices.



- 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

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

Online Compilation for CPU

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

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

Offline Compilation for CPU

> **NOTE** This is an experimental feature with limited functionality.

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

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

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:

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

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

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

Compile application sources:

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

Link the application with the static library:

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

Optimization Flags for CPU Architectures

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

On Linux:

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

On Windows:

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

Supported CPU optimization flags are:

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

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

Host and Kernel Interaction on CPU

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

Control Binary Execution on Multiple CPU Cores

Environment Variables

The following environment variables control the placement of DPC++ threads on multiple CPU cores during program execution.

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 spread 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, DPC++ uses TBB for scheduling. The value selects the petitioner used by the TBB scheduler. The value and meaning is the following: • dynamic - TBB auto_partitioner. It performs sufficient splitting to balance load. • affinity - TBB affinity_partitioner. It improves auto_partitioner's cache affinity by its choice of mapping subranges to worker threads compared to • static - TBB static_partitioner. It distributes range iterations among worker threads as uniformly as possible. TBB 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.

Environment Variable	Description
	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 TBB.
`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, TBB NUMA API will be used. This is analogous to OMP_PLACES=numa_domains in the OpenMP 5.1 Specification. TBB 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 TBB 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 TBB 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

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

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

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

See Also

Intel® oneAPI GPU Optimization Guide

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:

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

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

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

Compile application sources:

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

Link the application with the static library:

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

Offline Compilation for GPU

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

On Linux:

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

On Windows:

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

FPGA Flow

Field-programmable gate arrays (FPGAs) are 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, and 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. For example, kernel code.
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.

Notice You can also learn about programming for FPGAs 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. One important difference when compared to CPU or GPU is generating a device binary for an FPGA hardware 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 DPC++ FPGA Compilation

The following table summarizes the types of FPGA compilation:

Types of DPC++ FPGA Compilation

Device Image Type	Time to Compile	Description
FPGA Emulator	Seconds	The FPGA device code is compiled to the CPU. Use FPGA emulator to verify your SYCL code's functional correctness.
FPGA Early Image	Minutes	The FPGA device code is partially compiled for hardware. The compiler generates an optimization report that describes the structures generated on the FPGA, identifies performance bottlenecks, and estimates resource utilization.
FPGA Hardware Image	Hours	Generates the real FPGA bitstream to execute on the target FPGA platform.

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

> **Tip**
> To compile for FPGA emulation or to generate the FPGA optimization report, you need only the Intel® oneAPI Base Toolkit. However, an FPGA hardware compile requires the Intel® FPGA Add-on for oneAPI Base Toolkit. Refer to the Intel® oneAPI Toolkits Installation Guide for more information about installing this add-on.

FPGA Emulator

The FPGA emulator is the fastest method to verify the correctness of your code. It executes device code on the CPU, but supports all FPGA extensions such as FPGA pipes and `fpga_reg`. For more information, refer to Pipes Extension and Kernel Variables topics in the *Intel® oneAPI DPC++ FPGA Optimization Guide*.

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

- **Performance is not representative**

  You should not be using FPGA emulator to evaluate performance as it is not representative of the behavior of the FPGA device. For example, an optimization that yields a 100x performance improvement on the FPGA may show no impact on the emulator performance, or the emulator may 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, it is likely that your code is exhibiting undefined behavior. By definition, undefined behavior is not specified by the language specification, and might manifest differently on different targets.

Tip
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`.

FPGA Optimization Report

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

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, design entry language for FPGAs). The result is an FPGA early image that is not an executable. The optimization report generated at this stage is sometimes referred to as the *static report*.	Contains significant information about how the compiler has transformed your SYCL device code into an FPGA design. The report contains the following information: • Visualizations of structures generated on the FPGA • Performance and expected performance bottleneck • Estimated resource utilization
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 Software. The Intel® Quartus® Prime Software is included in the Intel® FPGA Add-On for oneAPI Base Toolkit, which is required for this compilation stage. The result is an FPGA hardware image that contains the FPGA bitstream (or binary).	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 *Intel® oneAPI DPC++ FPGA Optimization Guide*.

FPGA Hardware

This is a full compile through to the FPGA hardware image. You can target the Intel® Programmable Acceleration Card (PAC) with Intel Arria® 10 GX FPGA, the Intel® FPGA PAC D5005 (previously known as *Intel® PAC with Intel® Stratix® 10 SX FPGA*), or a custom board.

For more information about using Intel® PAC or custom boards, refer to the FPGA BSPs and Boards section.

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 three FPGA image types:

```
# FPGA emulator image
dpcpp -fintelfpga fpga_compile.cpp

# FPGA early image (with optimization report): default board
dpcpp -fintelfpga -Xshardware -fsycl-link fpga_compile.cpp

# FPGA early image (with optimization report): explicit board
dpcpp -fintelfpga -Xshardware -fsycl-link -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp

# FPGA hardware image: default board
dpcpp -fintelfpga -Xshardware fpga_compile.cpp -c -o fpga_compile.fpga
```

```
# FPGA hardware image: explicit board
dpcpp -fintelfpga -Xshardware -Xsboard=intel_s10sx_pac:pac_s10 fpga_compile.cpp
```

NOTE
Using the prefix `-Xs` causes an argument to be passed to the FPGA backend.

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

FPGA Compilation Flags

Flag	Explanation
`-fintelfpga`	Performs (offline) compilation for FPGA.
`-Xshardware`	Instructs the compiler to target FPGA hardware. If this flag is omitted, the compiler targets the default FPGA target, which is FPGA emulator.
`-fsycl-link`	Instructs the compiler to stop after creating the FPGA early image (and associated optimization report).
`-Xsboard=<bsp:variant>`	[Optional argument] Specifies the FPGA board variant and BSP. If omitted, the compiler chooses the default FPGA board variant `pac_a10` from the `intel_a10gx_pac BSP`. Refer to the FPGA BSPs and Boards section for additional details.

Warning
The output of a `dpcpp` 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 `dpcpp` 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 a list of 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 `dpcpp` 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>` [Linux only]	Instructs the compiler to extract the compiled FPGA hardware image from the existing executable and package it into the new executable, saving the device compilation time. This option is useful only when compiling for hardware. Refer to the Fast Recompile for FPGA section for additional information.

Option name	Description
`-Xsv`	FPGA backend generates a verbose output describing the progress of the compilation.
`-Xsemulator`	Generates an emulator device image. This is the default behavior.
`-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 a cost of reduced performance of the compiled FPGA hardware image. Use this flag only for faster development time. It is not intended for production quality results.

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

Device Selectors for FPGA

You must use the correct SYCL device selector in the host code, depending on whether you are targeting the FPGA emulator or FPGA hardware. 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 <CL/sycl/INTEL/fpga_extensions.hpp>

int main() {
  // Select either:
  //  - the FPGA emulator device (CPU emulation of the FPGA)
  //  - the FPGA device (a real FPGA)
#if defined(FPGA_EMULATOR)
  INTEL::fpga_emulator_selector device_selector;
#else
  INTEL::fpga_selector device_selector;
#endif

  queue q(device_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, the above code snippet can be compiled for the FPGA emulator by passing the flag `-DFPGA_EMULATOR` to `dpcpp`.
- Since FPGAs support only the ahead-of-time compilation method, dynamic selectors (such as the `default_selector`) cannot be used when targeting FPGA.

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.

Fast Recompile for FPGA

The Intel® oneAPI DPC++/C++ Compiler supports only the ahead-of-time (AoT) compilation for FPGA hardware, which means that an FPGA hardware image is generated at compile time. The FPGA hardware image generation process can take hours to complete. If you make a change that is exclusive to the host code, then recompile only your host code by reusing the existing FPGA hardware image and circumventing the time-consuming device compilation process.

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 (Linux only) to instruct the compiler to attempt to reuse the existing FPGA hardware image.
- Separating the host and device code into separate files. When a code change applies only to host-only files, the FPGA hardware image is not regenerated.

The following sections explain these two mechanisms in detail.

[Linux only] 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 image from the existing executable and package it into the new executable, saving the device compilation time.

Sample use:

```
# Initial compilation
dpcpp -fintelfpga -Xshardware <files.cpp> -o out.exe
```

The initial compilation generates an FPGA hardware image, which takes several hours. Suppose you now make some changes to the host code.

```
# Subsequent recompilation
dpcpp <files.cpp> -o out.exe -reuse-exe=out.exe -Xshardware -fintelfpga
```

One of the following actions are taken by the command:

- If `out.exe` does not exist, the `-reuse-exe` flag is ignored and the FPGA hardware image is regenerated. This is always the case the first time you compile a project.
- If `out.exe` is found, the compiler verifies no change that affect 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 hardware image and recompiles only the host code. The recompilation process takes a few minutes to complete.

NOTE
The device code is partially recompiled to check whether the FPGA hardware image can safely be reused.

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 hardware image generation happens at link time.

```
# normal compile command
dpcpp -fintelfpga -Xshardware main.cpp kernel.cpp -o fpga_compile.exe
```

As a result, any change to either the `main.cpp` or `kernel.cpp` triggers the regeneration of an FPGA hardware image.

If you want to iterate on the host code and avoid long compile time for your FPGA device, consider using a device link to separate the device and host compilation. The device link is a three-step process as listed in the following:

1. Compile the device code.

   ```
   dpcpp -fintelfpga -Xshardware -fsycl-link=image kernel.cpp -o dev_image.a
   ```

   Input files should include all files that contain source code, which executes on the device (for example, kernel code). This step might take several hours to complete.
2. Compile the host code.

   ```
   dpcpp -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.

   ```
   dpcpp -fintelfpga host.o dev_image.a -o fast_recompile.exe
   ```

   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 https://eli.thegreenplace.net/2013/07/09/library-order-in-static-linking.

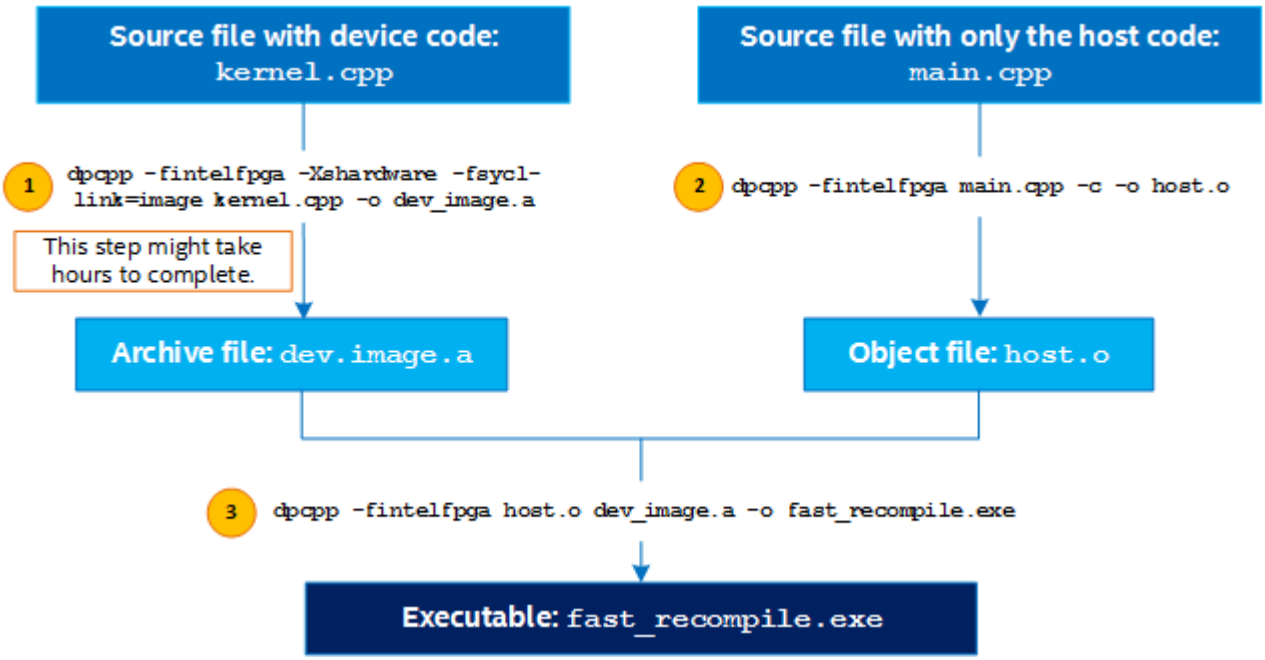
Tip

- After performing these three steps at least once, step 1 can be omitted unless you modify the device code.
- You can combine steps 2 and 3 into a single step, as follows:

  ```
  dpcpp -fintelfpga main.cpp dev_image.a -o fast_recompile.exe
  ```

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.

Which Mechanism to Use?

Of the two 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 has the advantage of giving 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.
- You might occasionally encounter a *false positive* where the compiler incorrectly believes that 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.

FPGA BSPs and Boards

As mentioned earlier in Types of DPC++ FPGA Compilation, generating an FPGA hardware image requires the Intel® FPGA Add-On for oneAPI Base Toolkit, which provides the Intel® Quartus® Prime Software that maps your design from RTL to the FPGA's primitive hardware resources. Additionally, this add-on package provides basic Board Support Packages (BSPs) that can be used to compile to FPGA hardware.

What is a Board?

Similar to 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 topic in the *Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference*.

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 `dpcpp` command:

Board	BSP	Flag	USM Support
Intel® Programmable Acceleration Card (PAC) with Intel Arria® 10 GX FPGA	`intel_a10gx_pac`	`-Xsboard=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`	`-Xsboard=intel_s10sx_pac:pac_s10`	Explicit USM
		`-Xsboard=intel_s10sx_pac:pac_s10_usm`	Explicit USM Restricted USM

NOTE

- The Intel® oneAPI DPC++/C++ Compiler (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 a DPC++ 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 Intel® oneAPI DPC++ FPGA Optimization Guide.

Custom BSPs

In addition to the BSPs described above, you can also use custom BSPs provided by a board vendor. Follow these steps to use the Intel® oneAPI DPC++/C++ Compiler with a custom BSP:

1. Install the Intel® FPGA Add-On for Intel® Custom Platform package that contains the version of Intel® Quartus® Prime Software required by the custom BSP. Refer to the detailed instructions in the Intel oneAPI Toolkits Installation Guide.
2. Place the full custom BSP in the `intelfpgadpcpp/latest/board` directory of the add-on installation.
3. Follow all installation instructions provided by the BSP vendor.

FPGA Board Initialization

Before you run the DPC++ executable, you must initialize the FPGA board using the following command:

```
aoclinitialize<board id> <board variant>
```

where:

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 `-Xsboard` flag when the DPC++ 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 DPC++ executable using the following command:

```
dpcpp -fintelfpga -Xshardware -Xsboard=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 DPC++ executable without initializing the board again, unless you are doing one of the following:

- Running a DPC++ workload for the first time after power cycling the host.
- Running a DPC++ workload after running a non-DPC++ workload on the FPGA.
- Running a DPC++ workload compiled with a different board variant in `-Xsboard` flag.

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 FPGA emulator target, execute the following commands:

```
dpcpp -fsycl -fsycl-targets=spir64-unknown-unknown-sycldevice \
jit_kernel.cpp -c -o jit_kernel.o

dpcpp -DFPGA_EMULATOR -fsycl
-fsycl-targets=spir64_fpga-unknown-unknown-sycldevice fpga_kernel.cpp \
-c -o fpga_kernel.o
```

```
dpcpp -DFPGA_EMULATOR -fsycl \
-fsycl-targets=spir64_fpga-unknown-unknown-sycldevice,spir64-unknown-unknown-sycldevice main.cpp
jit_kernel.o fpga_kernel.o -o emulator.out
```

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:

```
#ifdef FPGA_EMULATOR
INTEL::fpga_emulator_selector device_selector;
#else
INTEL::fpga_selector device_selector;
#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:

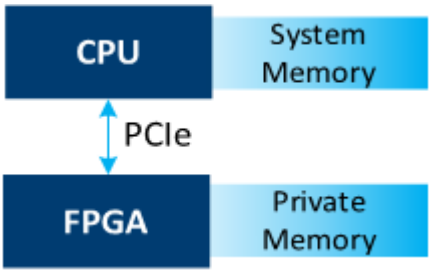
```
dpcpp -fsycl -fsycl-targets=spir64-unknown-unknown-sycldevice \
jit_kernel.cpp -c -o jit_kernel.o

dpcpp -fsycl -fsycl-targets=spir64_fpga-unknown-unknown-sycldevice \
fpga_kernel.cpp -c -o fpga_kernel.o -Xshardware

dpcpp -fintelfpga main.cpp jit_kernel.o fpga_kernel.o -Xshardware
```

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.



This is an important factor influencing the performance of SYCL programs targeting FPGAs. The first time a particular DPC++ program is run, the FPGA must be configured with its hardware bitstream and this may require several seconds.

Data Transfer

Typically, the FPGA board has its own private DDR memory on which it primarily operates. The CPU must bulk transfer or dynamic 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, as well as 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:

- DPC++ allows buffers to be tagged as read-only or write-only, which allows some unnecessary transfers to be eliminated.
- 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, techniques such as double buffering can be applied. Refer to the Double Buffering Host Utilizing Kernel Invocation Queue topic in the *Intel oneAPI DPC++ FPGA Optimization Guide* 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 *Intel® oneAPI DPC++ FPGA Optimization Guide* 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, 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.

See Also

Intel® oneAPI DPC++ FPGA Optimization Guide
Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA
Intel® FPGA Programmable Acceleration Card (PAC) D5005

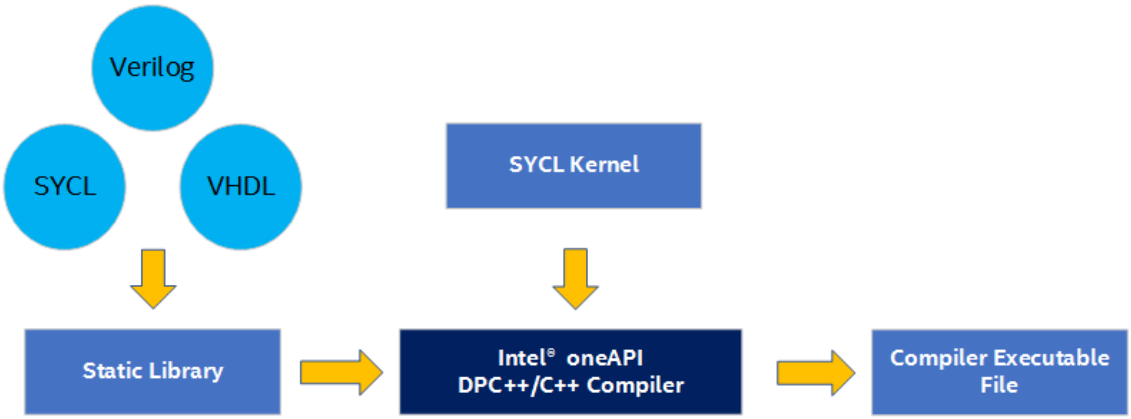
FPGA Performance Optimization

The preceding FPGA flow covered the basics of compiling DPC++ 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 all these information in the Intel® oneAPI DPC++ FPGA Optimization Guide, which should be your main reference if you want to understand how to optimize your design.

Use of Static Library for FPGA

A static library is a single file that contains multiple functions. You can create a static library file using SYCL, OpenCL, High-Level Synthesis (HLS) sources, or register transfer level (RTL). You can then include this library file and use the functions inside your SYCL kernels.

If you develop your library function in HLS, then for more information, refer to Intel High Level Synthesis Compiler: Reference Manual.



You may use a third-party library or create your own library. To use a static library, you do not require in-depth knowledge in hardware design or in the implementation of library primitives. To generate libraries that you can use with SYCL, you need to create the following files:

Generating Libraries for Use with SYCL

File or Component	Description
RTL Modules	
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.
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.
Header file (`.hpp`)	A header file that contains valid SYCL kernel language and declares the signatures of functions that are implemented by the RTL component.
Emulation model file (OpenCL or HLS-based)	Provides C or C++ model for the RTL component that is used only for emulation. Full hardware compilations use the RTL source files.

File or Component	Description
	Restriction You cannot write the emulation model in SYCL.
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.
OpenCL Functions	
OpenCL source files (`.cl`)	Contains definitions of the OpenCL 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.
HLS Functions	
HLS source files (`.cpp`)	Contains definitions of the HLS 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.

NOTE
There is no difference in the header file used for RTL and other library functions. A single library can contain any of the supported sources. You can create a library from mixed sources (SYCL, OpenCL, HLS or RTL) and target these products:

- Intel® oneAPI DPC++/C++ Compiler
- Intel® FPGA SDK for OpenCL Offline Compiler
- Intel® HLS Compiler

The library files use the same format as the operating system that you compile your source code on, with additional sections that carry additional library information.

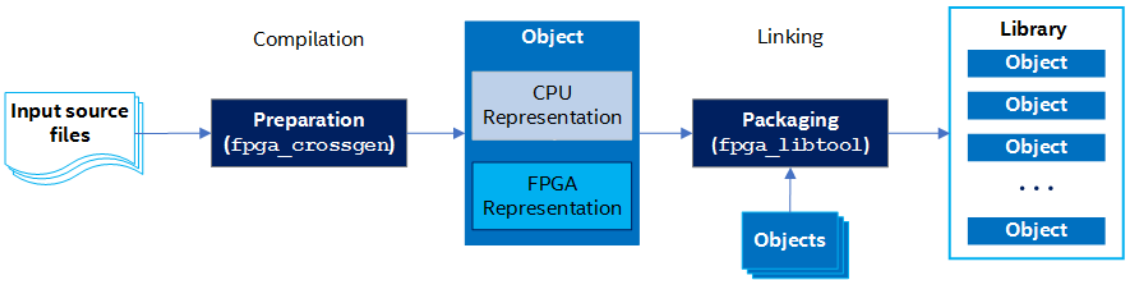
- On Linux* platforms, a library is a `.a` archive file that contains `.o` object files.
- On Windows* platforms, a library is a `.lib` archive file that contains `.obj` object files.

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 `dpcpp` command line when your 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.



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 as well as hardware execution of CPU capturing for use in host and emulation of kernel.

A library can contain multiple object files. You can create object files for use in different Intel high-level design tools from the same source code. Depending on the target high-level design tool, your source code might require adjustments to support tool-specific data types or constructs.

Product	Description
Intel® oneAPI DPC++/C++ Compiler Intel® FPGA SDK for OpenCL Offline Compiler	No additional work is needed in your SYCL and OpenCL source code when you use the code to create objects for the compiler libraries.
Intel® HLS Compiler	SYCL supports language constructs that are not natively supported by C++. Your component might need modifications to support those constructs. It is always preferred to use SYCL data types as library function call parameters. **Restriction** You cannot use systems of tasks in components intended for use in a SYCL library object.

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.

NOTE
Importing HLS libraries for use in SYCL by using the `fpga_crossgen` command is supported only on Linux*.

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 Intel high-level design tool. Also, objects are versioned. That is, each object is assigned a compiler version number and be used only with Intel high-level design tools with the same version number.

Create a library object using the following command:

```
fpga_crossgen <source_file> [--source (sycl|ocl|hls)]
--target (sycl|ocl|hls) [-o <object_file>]
```

The following table describes the parameters:

Parameter	Description
`<source_file>`	You can use SYCL, OpenCL, HLS, and RTL source code files.
`--source`	Optional flag. It supports `sycl`, `hls`, and `ocl` options. • When the source file ends in `.cpp`, the flag defaults to `hls`. • When the source file ends in `.cl`, the flag defaults to `ocl`. • For RTL source file, the `--source` flag refers to the emulation model.
`--target`	Targets a Intel® high-level design tool (`sycl`, `ocl`, and `hls`) for the library. The objects are combined as object files into a SYCL library archive file using the `fpga_libtool`.
`-o`	Optional flag. This options 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 commands:

```
fpga_crossgen lib_hls.cpp --source hls --target sycl -o lib_hls.o
```

```
fpga_crossgen lib_ocl.cl --source ocl --target sycl -o lib_ocl.o
```

```
fpga_crossgen lib_rtl_spec.xml --emulation_model lib_rtl_model.cpp
--target sycl -o lib_rtl.o
```

```
fpga_crossgen lib_sycl.cpp --source sycl --target sycl -o lib_sycl.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 that 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|ocl|hls) --create <library_name>
```

The command parameters are defined as follows:

Parameter	Description
file1 file2 ... fileN	You can specify one or more object files to include in the library.
`--target (sycl|ocl|hls)`	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.

Parameter	Description
`--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_hls.o lib_ocl.o lib_rtl.o lib_sycl.o --target sycl --create lib.a
```

where, the command packages objects created from HLS, OpenCL, RTL, and SYCL source code into a SYCL library called `lib.a`.

> **Tip**
> For additional information, refer to the FPGA tutorial sample "Use Library" listed in the Intel® oneAPI Samples Browser on Linux* or Intel® oneAPI Samples Browser on Windows*.

Using Static Libraries

You can include static libraries in your compiler command along with your source files, as shown in the following command:

```
dpcpp -fintelfpga main.cpp lib.a
```

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. That is, 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 that communicate with external memory must have Avalon® memory-mapped interface port parameters that match the corresponding Custom Platform parameters. The Intel® oneAPI DPC++/C++ Compiler does not perform any width or burst adaptation.
- RTL modules that communicate with external memory must behave as follows:
  - They cannot burst across the burst boundary.
  - They cannot make requests every clock cycle and stall the hardware by monopolizing the arbitration logic. An RTL module must pause its requests regularly to allow other load or store units to execute their operations.
- 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 that corresponds to RTL module instantiation is completely 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 to occur 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 causes 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 very difficult 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 RTL module area is 0. The compiler does not currently support the capability of specifying an area model for RTL modules.

FPGA Workflows in IDEs

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

For FPGA development with Visual Studio Code on Linux*, refer to Intel® oneAPI DPC++ FPGA Development with Sample Browser Extension for Visual Studio Code on Linux.

5 API-based Programming

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

oneAPI Library Overview

The following libraries are available from the oneAPI toolkits:

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

Intel oneAPI DPC++ Library (oneDPL)

The Intel® oneAPI DPC++ Library (oneDPL) aims to work with the Intel® oneAPI DPC++/C++ Compiler to provide high-productivity APIs to developers, which can minimize Data Parallel C++ (DPC++) 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

See Also

Intel oneAPI DPC++ Library Guide

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 Data Parallel C++ (DPC++) programming language interfaces for achieving performance on various CPU architectures and Intel Graphics Technology for certain key functionalities.

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

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

See Also

Get Started with Intel oneAPI Math Kernel Library for Data Parallel C++
Intel oneAPI Math Kernel Library - Data Parallel C++ Developer Reference
Developer Guide for Intel oneAPI Math Kernel Library for Linux* OS
Developer Guide for Intel oneAPI Math Kernel Library for Windows* OS

oneMKL Usage

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

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

Namespace	Description
`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 DPC++ interfaces, the following example source code snippets perform a double precision matrix-matrix multiplication on a GPU device.

```
// Standard SYCL header
#include <CL/sycl.hpp>
// STL classes
#include <exception>
#include <iostream>
// Declarations for Intel oneAPI Math Kernel Library DPC++ APIs
#include "mkl_sycl.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 mkl::blas::gemm to execution queue and catch any synchronous exceptions
    try {
        mkl::blas::gemm(my_queue, mkl::transpose::nontrans, mkl::transpose::nontrans, m, n, k,
alpha, A_buffer, ldA, B_buffer,
           ldB, beta, C_buffer, ldC);
    }
    catch (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 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 DPC++ APIs
#include "mkl_sycl.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;
            }
        }
    };
    // create execution queue on my gpu device with exception handler attached
    sycl::queue my_queue(my_device, my_exception_handler);
```

The matrix data is now loaded into the DPC++ 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 mkl::blas::gemm to execution queue and catch any synchronous exceptions
    try {
        mkl::blas::gemm(my_queue, mkl::transpose::nontrans, mkl::transpose::nontrans, m, n, k,
alpha, A_buffer, ldA, B_buffer,
           ldB, beta, C_buffer, ldC);
    }
    catch (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;
```

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.

See Also

Get Started with Intel oneAPI Threading Building Blocks

Intel oneAPI Threading Building Blocks Documentation

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 the DPC++ language and other oneAPI libraries to build a program that efficiently utilizes all available hardware resources.

oneTBB Code Sample

Two basic oneTBB code samples are available 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.

Intel oneAPI Data Analytics Library (oneDAL)

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

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

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

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

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

See Also

Get Started with the Intel oneAPI Data Analytics Library

Intel oneAPI Data Analytics Library Documentation

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

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

oneDAL Code Sample

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

See Also

Get Started with Intel oneAPI Collective Communications Library
Intel oneAPI Collective Communications Library Documentation

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 DPC++ API and runtime support for the oneAPI programming model.

For more information, see https://github.com/oneapi-src/oneDNN.

See Also

Get Started with Intel oneAPI Deep Neural Network Library
Intel oneAPI Deep Neural Network Library Documentation

oneDNN Usage

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

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

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

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

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

The interoperability API is provided for two scenarios:

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

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

oneDNN Objects	DPC++ Objects
Engine	cl::sycl::device and cl::sycl::context
Stream	cl::sycl::queue
Memory	cl::sycl::buffer<uint8_t, 1> 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 Object	Constructing from DPC++ 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 Object	Extracting DPC++ Object
Engine	`dnnl::sycl_interop::get_device(engine)` `dnnl::sycl_interop::get_context(engine)`

oneDNN Object	Extracting DPC++ Object
Stream	`dnnl::sycl_interop::get_queue(stream)`
Memory	USM pointer: `dnnl::memory::get_data_handle()` Buffer: `dnnl::sycl_interop::get_buffer(memory)`

Notes

- 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.

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.

See Also

Get Started with oneVPL

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 DPC++.

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

6 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 DPC++

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

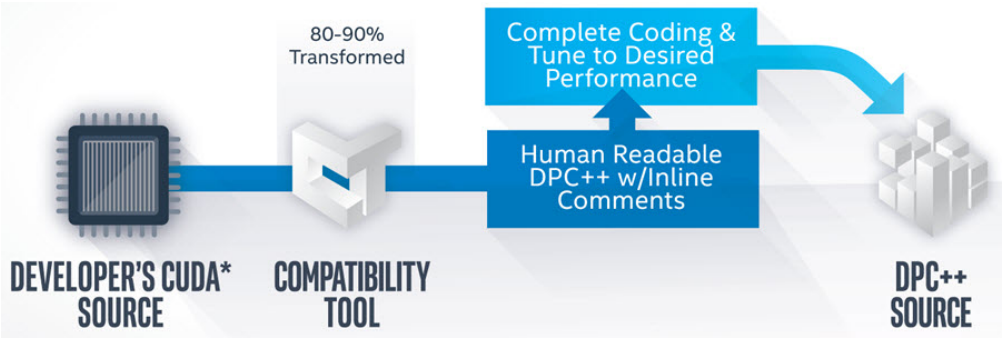
Migrating from C++ to SYCL/DPC++

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

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

Migrating from CUDA* to DPC++

The Intel DPC++ Compatibility Tool is part of the Intel oneAPI Base Toolkit. The goal of this tool is to assist in the migration of an existing program that is written in NVIDIA* CUDA* to a program written in DPC++. This tool generates DPC++ code as much as possible. 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 DPC++. After completing any manual changes, use a DPC++ compiler to create executables.



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 DPC++

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

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

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

Migrating Between CPU, GPU, and FPGA

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

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

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

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

```
          std::cout <<
          "SYCL exception caught: " << e.what() << '\n';
          return 2;
      }
      return 0;
}
```

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

```
#include <stdio.h>
#include <vector>
#include <CL/sycl.hpp>
using namespace cl::sycl;
using namespace std;
int main()
{
       size_t N = 1024;
       vector<float> a(N, 10.0);
       vector<float> b(N, 10.0);
       vector<float> c_add(N, 0.0);
       vector<float> c_mul(N, 0.0);
   {
       buffer<float, 1> abuffer(a.data(), range<1>(N),
         { property::buffer::use_host_ptr() });
       buffer<float, 1> bbuffer(b.data(), range<1>(N),
         { property::buffer::use_host_ptr() });
       buffer<float, 1> c_addbuffer(c_add.data(), range<1>(N),
        { property::buffer::use_host_ptr() });
       buffer<float, 1> c_mulbuffer(c_mul.data(), range<1>(N),
         { property::buffer::use_host_ptr() });
    try {
              gpu_selector gpuSelector;
              auto queue = cl::sycl::queue(gpuSelector);
              queue.submit([&](cl::sycl::handler& cgh) {
                     auto a_acc = abuffer.template
                       get_access<access::mode::read>(cgh);
                     auto b_acc = bbuffer.template
                       get_access<access::mode::read>(cgh);
                     auto c_acc_add = c_addbuffer.template
                       get_access<access::mode::write>(cgh);
                     cgh.parallel_for<class VectorAdd>
                      (range<1>(N), [=](id<1> it) {
                          //int i = it.get_global();
                              c_acc_add[it] = a_acc[it] + b_acc[it];
                                   });
                                 });
              cpu_selector cpuSelector;
              auto queue1 = cl::sycl::queue(cpuSelector);
              queue1.submit([&](cl::sycl::handler& cgh) {
                     auto a_acc = abuffer.template
                         get_access<access::mode::read>(cgh);
                     auto b_acc = bbuffer.template
                         get_access<access::mode::read>(cgh);
                     auto c_acc_mul = c_mulbuffer.template
                         get_access<access::mode::write>(cgh);
```

```
                    cgh.parallel_for<class VectorMul>
                     (range<1>(N), [=](id<1> it) {
                          c_acc_mul[it] = a_acc[it] * b_acc[it];
                                  });
                           });
              }
              catch (cl::sycl::exception e) {
/* In the case of an exception being throw, print the
error message and
                     * return 1. */
                     std::cout << e.what();
                     return 1;
              }
       }
       for (int i = 0; i < 8; i++) {
              std::cout << c_add[i] << std::endl;
              std::cout << c_mul[i] << std::endl;
       }
       return 0;
}
```

Composability

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

C/C++ OpenMP* and DPC++ 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 DPC++ composability with a set of restrictions. A single application can offload execution to available devices using OpenMP target regions or DPC++/SYCL constructs in different parts of the code, such as different functions or code segments.

OpenMP and DPC++ offloading constructs may be used in separate files, in the same file, or in the same function with some restrictions. OpenMP and DPC++ offloading code can be bundled together in executable files, in static libraries, in dynamic libraries, or in various combinations.

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

Restrictions

There are some restrictions to be considered when mixing OpenMP and DPC++/SYCL constructs in the same application.

- OpenMP directives cannot be used inside DPC++/SYCL kernels that run in the device. Similarly, DPC++/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 DPC++/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 DPC++ code. That is, using the device memory object created by OpenMP in DPC++/SYCL code results unspecified execution behavior.

Example

The following code snippet uses DPC++/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 DPC++
- `-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

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

Debugger Features

Intel® Distribution for GDB* is based on GDB 10.0 with multi-target extensions that adds support for Intel accelerator targets. It supports all GDB features that are applicable to the respective target and allows debugging the host application plus all supported devices within the same debug session.

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

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

Intel® Distribution for GDB* plugs into Eclipse on Linux* host and Microsoft* Visual Studio* on Windows* host.

SIMD Support

NOTE This feature is only supported on GPU devices via the Intel® Distribution for GDB* command line interface.

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

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

```
  Id             Target Id            Frame
  1.1            Thread <id omitted>  <frame omitted>
  1.2            Thread <id omitted>  <frame omitted>
  2.1            Thread 1610612736    (inactive)
* 2.2:1          Thread 1073741824       <frame> at array-transform.cpp:61
  2.2:[3 5 7]    Thread 1073741824       <frame> at array-transform.cpp:61
  2.3:[1 3 5 7]  Thread 1073741888       <frame> at array-transform.cpp:61
  2.4:[1 3 5 7]  Thread 1073742080       <frame> at array-transform.cpp:61
  2.5:[1 3 5 7]  Thread 1073742144       <frame> at array-transform.cpp:61
  2.6:[1 3 5 7]  Thread 1073742336       <frame> at array-transform.cpp:61
  2.7:[1 3 5 7]  Thread 1073745920       <frame> at array-transform.cpp:61
  2.8:[1 3 5 7]  Thread 1073746176       <frame> at array-transform.cpp:61
  2.9:[1 3 5 7]  Thread 1073746432       <frame> at array-transform.cpp:61
```

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

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

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

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

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

When using the `thread apply` command, the specified command is applied in the context of the default lane of a thread. To apply the command in contexts of several lanes, specify them after the colon (:). For example, `thread apply 2.2:3-6 print element` prints the value of element variable for all active lanes from the range 3-6 in thread 2.2.

Operating System Differences for Debugging oneAPI Code

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

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

Environment Setup

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

Breakpoints

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

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

Evaluations and Data Races

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

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

Linux Sample Session

The following sample session is from the array-transform sample code. The latest version is available at the oneAPI samples on GitHub.

For step-by-step instructions on basic debugging on CPU and GPU, refer to the Debugging tutorial for Linux* OS host or Debugging tutorial for Windows* OS host.

```
$ gdb-oneapi -q --args array-transform gpu
Reading symbols from array-transform...
(gdb) break 57
Breakpoint 1 at 0x4194d8: file /path/to/array-transform.cpp, line 57.
(gdb) run
Starting program: /path/to/array-transform gpu
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7ffff2bb4700 (LWP 24725)]
[New inferior 2]
Added inferior 2 on connection 1 (native)
Intelgt auto-attach: a gdbserver-gt will be attached using:
inferior 2 and host's pid 24721.
[Switching to inferior 2 [<null>] (<noexec>)]
Attached; pid = 24721
Remote debugging using stdio
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00000000ffffd020 in ?? ()
Intelgt auto-attach: inferior 2 created locally to listen to GT debug
events.
[Switching to inferior 1 [process 24721] (array-transform)]
[Switching to thread 1.1 (Thread 0x7ffff7fb9480 (LWP 24721))]
#0  0x00007ffff21ab4b0 in igfxdbgxchgDebuggerHook () from libigfxdbgxchg64.so
[SYCL] Using device: [Intel(R) Gen9 HD Graphics NEO] from [Intel(R) OpenCL HD Graphics]
```

```
[New Thread 1073741824]
Reading default.gtelf from remote target...
warning: File transfers from remote targets can be slow. Use "set sysroot" to access files
locally instead.
Reading /tmp/kernel_4723361564299735604.dbgelf from remote target...
[New Thread 1073746432]
[New Thread 1073746176]
[New Thread 1073745920]
[New Thread 1073742336]
[New Thread 1073742144]
[New Thread 1073742080]
[New Thread 1073741888]
[Switching to Thread 1073741824 lane 0]

Thread 2.2 hit Breakpoint 1,  with SIMD lanes [0-7], main::
$_1::operator()<cl::sycl::handler>(cl::sycl::handler&) const::
{lambda(cl::sycl::id<1>)#1}::operator()(cl::sycl::id<1>) const (this=0x7fffe1b1b740,
index=cl::sycl::id<1> = {...}) at array-transform.cpp:57
57            int result = element + 50;
(gdb) info inferiors
  Num  Description       Connection                                Executable
  1    process 24721     1 (native)                                array-transform
* 2    Remote target     2 (remote gdbserver-gt --attach - 24721)
(gdb) list
52
53          // kernel-start
54          h.parallel_for(data_range, [=](id<1> index) {
55            int id0 = GetDim(index, 0);
56            int element = in[index];  // breakpoint-here
57            int result = element + 50;
58            if (id0 % 2 == 0) {
59              result = result + 50;  // then-branch
60            } else {
61              result = -1;  // else-branch
(gdb) print element
$1 = 100
(gdb) thread 2.3:5
[Switching to thread 2.3:5 (Thread 1073746432 lane 5)]
#0  main::$_1::operator()<cl::sycl::handler>(cl::sycl::handler&) const::
{lambda(cl::sycl::id<1>)#1}::operator()(cl::sycl::id<1>) const (this=0x7fffe29c5480,
    index=cl::sycl::id<1> = {...}) at array-transform.cpp:57
57            int result = element + 50;
(gdb) print element
$2 = 145
(gdb) thread apply 2-3:3-5 -q print index
$3 = cl::sycl::id<1> = {3}
$4 = cl::sycl::id<1> = {4}
$5 = cl::sycl::id<1> = {5}
$6 = cl::sycl::id<1> = {43}
$7 = cl::sycl::id<1> = {44}
$8 = cl::sycl::id<1> = {45}
(gdb) disassemble $pc, +36
Dump of assembler code from 0xfffe7e00 to 0xfffe7e24:
=> 0x00000000fffe7e00
<_ZTSZZ4mainENKUlRT_E49_14clIN2cl4sycl7handlerEEEDaS0_EUlNS4_2idILi1EEEE54_34(int*,
cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>, int*, cl::sycl::range<1>,
cl::sycl::range<1>, cl::sycl::id<1>)+269824>:    (W)      send (16|M0)             r26:uw
r25     0xA         0x22C154C  //  wr:1h+?, rd:2, Scratch Block Read from scratch offset 0xa980
```

```
   0x00000000fffe7e10
<_ZTSZZ4mainENKUlRT_E49_14clIN2cl4sycl7handlerEEEDaS0_EUlNS4_2idILi1EEEE54_34(int*,
cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>, int*, cl::sycl::range<1>,
cl::sycl::range<1>, cl::sycl::id<1>)+269840>:             send (8|M0)              r26:d
r26:uq  0xC         0x41401FF  //  wr:2+?, rd:1, A64 Scattered Read msc:1, to global memory
   0x00000000fffe7e20
<_ZTSZZ4mainENKUlRT_E49_14clIN2cl4sycl7handlerEEEDaS0_EUlNS4_2idILi1EEEE54_34(int*,
cl::sycl::range<1>, cl::sycl::range<1>, cl::sycl::id<1>, int*, cl::sycl::range<1>,
cl::sycl::range<1>, cl::sycl::id<1>)+269856>:             add (8|M0)               r26.0<1>:d
r26.0<8;8,1>:d    50:w
End of assembler dump.
(gdb)
```

Performance Tuning Cycle

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

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

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

Establish Baseline

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

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

Identify Kernels to Offload

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

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

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

See Intel Advisor User Guide for functionality description.

Offload Kernels

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

Optimize

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

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

Optimization is a process of eliminating bottlenecks, i.e. the sections of code that are taking more execution time relative 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 Intel oneAPI DPC++ FPGA Optimization Guide. This section assumes that the kernel to offload to the accelerator is already determined. It also assumes that work will be accomplished on one accelerator. This guide does not speak to division of work between host and accelerator or between host and potentially multiple and/or different accelerators.

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

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

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

High-level Optimization Tips

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

Loop-related Optimizations

- Prefer well-structured, well-formed, and 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 DPC++ kernel. In turn, ignoring accessor argument dependencies allows the compiler to perform more aggressive optimizations and potentially improve the performance of the kernel.

DPC++-specific Optimizations

- When possible, specify a work-group size. The attribute, `[[cl::reqd_work_group_size(X, Y, Z)]]`, where X, Y, and Z are integer dimension in the ND-range, can be employed to 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.

Glossary

Accelerator

Specialized component containing compute resources that can quickly execute a subset of operations. Examples include CPU, FPGA, GPU.

See also: Device

Accessor

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

Application Scope

Code that executes on the host.

Buffers

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

Command Group Scope

Code that acts as the interface between the host and device.

Command Queue

Issues command groups concurrently.

Compute Unit

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

Device

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

See also: Accelerator

Device Code

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

Fat Binary

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

Fat Library

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

Fat Object

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

Host

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

Host Code

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

Images

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

Kernel Scope

Code that executes on the device.

ND-range

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

Processing Element

Individual engine for computation that makes up a compute unit.

Single Source

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

SPIR-V

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

Sub-groups

Sub-groups are an Intel extension.

Work-groups

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

Work-item

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