

Intel® oneAPI Programming Guide

Contents

Chapter 1: Intel® oneAPI Programming Guide

Introduction to oneAPI Programming.....	4
Intel oneAPI Programming Overview	5
Intel® oneAPI Toolkit Distribution	7
Related Documentation	7
oneAPI Programming Model	7
Data Parallelism in C++ Using SYCL*	8
C/C++ or Fortran with OpenMP* Offload Programming Model.....	10
Device Selection	13
SYCL* Execution and Memory Hierarchy	15
oneAPI Development Environment Setup	17
Use the setvars and oneapi-vars Scripts with Windows*	19
Use a Config File for setvars.bat on Windows*	24
Automate the setvars.bat Script with Microsoft Visual Studio*	28
Use the setvars and oneapi-vars Scripts with Linux*	29
Use a Config File for setvars.sh on Linux	34
Automate the setvars.sh Script with Eclipse*	38
Use Environment Modulefiles with Linux*	39
Use CMake with oneAPI Applications	45
Compile and Run oneAPI Programs	46
Single-Source Compilation.....	46
Invoke the Compiler	47
Standard Intel® oneAPI DPC++/C++ Compiler Options.....	47
Example Compilation	48
Compilation Flow Overview	50
CPU Flow	55
Traditional CPU Flow	55
CPU Offload Flow	55
GPU Flow	63
GPU Offload Flow	64
Example GPU Commands	69
Ahead-of-Time Compilation for GPU	70
FPGA Flow	70
Why is FPGA Compilation Different?.....	70
Types of SYCL* FPGA Compilation	71
API-Based Programming.....	74
Intel® oneAPI DPC++ Library (oneDPL)	75
oneDPL Library Usage.....	75
oneDPL Code Samples	75
Intel® oneAPI Math Kernel Library (oneMKL)	76
oneMKL Usage.....	77
oneMKL Code Sample	77
Intel® oneAPI Threading Building Blocks (oneTBB)	81
oneTBB Usage	81
oneTBB Code Sample	81
Intel® oneAPI Data Analytics Library (oneDAL)	82
oneDAL Usage	82
oneDAL Code Sample	83

Intel® oneAPI Collective Communications Library (oneCCL)	83
oneCCL Usage	83
oneCCL Code Sample	84
Intel® oneAPI Deep Neural Network Library (oneDNN)	84
Intel® oneAPI Deep Neural Network Library (oneDNN) Usage	84
Intel® oneAPI Deep Neural Network Library (oneDNN) Code Sample	86
Other Libraries	86
Software Development Process	87
Migrating Code to SYCL* and DPC++	87
Migrating from C++ to SYCL*	87
Migrating from CUDA* to SYCL* for the oneAPI DPC++ Compiler ..	88
Migrating from OpenCL Code to SYCL*	88
Migrating Between CPU, GPU, and FPGA	89
Composability	91
C/C++ OpenMP* and SYCL* Composability	91
OpenCL™ Code Interoperability	93
Debugging the DPC++ and OpenMP* Offload Process	93
oneAPI Debug Tools for SYCL* and OpenMP* Development	94
Trace the Offload Process	104
Debug the Offload Process	106
Optimize Offload Performance	120
Performance Tuning Cycle	122
Establish Baseline	123
Identify Kernels to Offload	123
Offload Kernels	123
Optimize Your SYCL* Applications	123
Recompile, Run, Profile, and Repeat	125
oneAPI Library Compatibility	125
SYCL* Extensions	126
Glossary	126
Notices and Disclaimers	128

Intel® oneAPI Programming Guide



Use this guide to learn about:

- [Introduction to oneAPI Programming](#): A basic overview of oneAPI, Intel® oneAPI Toolkits, and related resources.
- [oneAPI Programming Model](#): An introduction to the oneAPI programming model for SYCL* and OpenMP* offload for C, C++, and Fortran.
- [oneAPI Development Environment Setup](#): Instructions on how to set up the oneAPI application development environment.
- [Compile and Run oneAPI Programs](#): Details about how to compile code for various accelerators (CPU, FPGA, etc.).
- [API-Based Programming](#): A brief introduction to common APIs and related libraries as well as details on buffer usage.
- [Software Development Process](#): An overview of the software development process using various oneAPI tools, such as debuggers and performance analyzers, and optimizing code for a specific accelerator (CPU, FPGA, etc.).

Introduction to oneAPI Programming

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

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

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

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

The oneAPI programming model simplifies the programming of CPUs and accelerators using modern C++ features to express parallelism using SYCL*. SYCL enables code reuse for the host (such as a CPU) and accelerators (such as a GPU) using a single source language, with execution and memory dependencies

clearly communicated. Mapping within the SYCL code can be used to transition the application to run on the hardware, or set of hardware, that best accelerates the workload. A host is available to simplify development and debugging of device code, even on platforms that do not have an accelerator available.

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

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

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

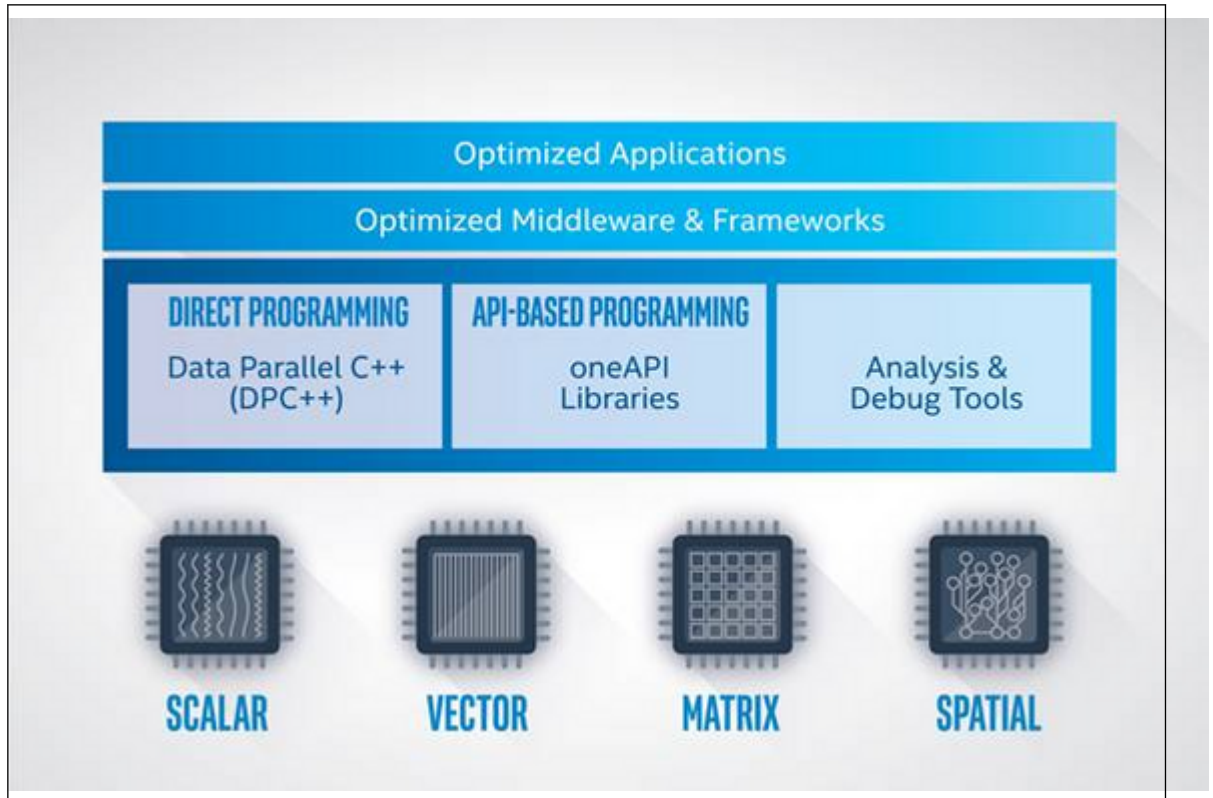
NOTE Not all programs can benefit from the single programming model offered by oneAPI. It is important to understand how to design, implement, and use the oneAPI programming model for your program.

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

Intel oneAPI Programming Overview

The oneAPI programming model provides a comprehensive and unified portfolio of developer tools that can be used across hardware targets, including a range of performance libraries spanning several workload domains. The libraries include functions custom-coded for each target architecture, so the same function call delivers optimized performance across supported architectures.

The oneAPI programming model



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

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

The Intel oneAPI Base Toolkit is available as a free download from the [Intel Developer Zone](#).

Users familiar with Intel® Parallel Studio and Intel® System Studio may be interested in the [Intel® HPC Toolkit](#).

Intel® oneAPI Toolkit Distribution

oneAPI Toolkits are available via multiple distribution channels:

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

Related Documentation

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

- Get started guides for select Intel® oneAPI toolkits:
 - Get Started with Intel® oneAPI Base Toolkit for [Linux*](#) | [Windows*](#)
 - Get Started with Intel® HPC Toolkit for [Linux*](#) | [Windows*](#)
- Release notes for select Intel® oneAPI toolkits:
 - [Intel® oneAPI Base Toolkit](#)
 - [Intel® HPC Toolkit](#)
- Language reference material:
 - [Khronos SYCL Reference](#)
 - [SYCL* Specification \(PDF\) 1.2.1 | 2020](#)
 - [Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL](#) (book by James Reinders, Ben Ashbaugh, James Broadman, Michael Kinsner, John Pennycook, and Xinmin Tian, parts of this book were reused under the [Creative Commons license](#).)
 - [LLVM/OpenMP* Documentation](#)
 - [OpenMP* Specifications](#) (examples documents recommended)

oneAPI Programming Model

In heterogenous computing, the [host](#) processor takes advantage of accelerator [devices](#) to execute code more efficiently.

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

SYCL

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

SYCL Kernel Compatibility

In accordance with the SYCL 2020 specification, DPC++ will detect uses of optional kernel features in SYCL kernels. A kernel is only considered compatible with a device if that device reports true in `device::has()` for all aspects required by the kernel. Effectively, this means that a `kernel_bundle` created with a given device will not contain kernels that have requirements that the device does not support and attempting to launch kernels on a device that they are not compatible with will result in a `sycl::exception` being thrown.

As determining the origin of a requirement in a kernel can be difficult, the user can add the `sycl::device_has` attribute to the kernels. The compiler will warn the user of any aspect requirements deduced for the kernel that do not appear in the kernel's `sycl::device_has` attribute. Note that this behavior is required by the SYCL 2020 specification.

OpenMP*

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

NOTE OpenMP is not supported for FPGA devices.

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

Data Parallelism in C++ Using SYCL*

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

Simple Sample Code Using Queue Lambda by Reference

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

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

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

constexpr int num=16;
using namespace sycl;

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

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

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

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

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

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

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

Additional Resources

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

The following resources are useful to learning and mastering SYCL with oneAPI:

- [Explore SYCL with Samples from Intel](#) provides an overview and links to simple sample applications available from GitHub*.
- The [DPC++ Foundations Code Sample Walk-Through](#) is a detailed examination of the Vector Add sample code, the DPC++ equivalent to a basic Hello World application.
- The Khronos* [SYCL Reference](#) contains information about SYCL class member functions and usage examples.
- The [DPC++ Essentials training course](#) is a guided learning path for SYCL using Jupyter* Notebooks on Intel® DevCloud.
- [Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL](#) is a comprehensive book that introduces and explains key programming concepts and language details about SYCL and Heterogeneous programming.

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

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

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

NOTE OpenMP is not supported for FPGA devices.

Basic OpenMP Target Construct

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

C/C++

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

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

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

Fortran

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

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

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

Map Variables

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

C/C++

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

Fortran

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

Clauses

The clauses can be one or more of the following. See [TARGET DATA](#) for more information.

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

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

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

- SUBDEVICE ([integer-constant ,] integer-expression [: integer-expression [: integer-expression]])

- USE_DEVICE_ADDR (list) // available only in ifx
- USE_DEVICE_PTR (ptr-list)

NOTE The SUBDEVICE clause is ignored in the following cases:

- If ZE_FLAT_DEVICE_HIERARCHY is set to FLAT or COMBINED.
 - If env LIBOMPTARGET_DEVICES is set to SUBDEVICE/SUBSUBDEVICE
 - If env ONEAPI_DEVICE_SELECTOR is used to select devices.
-

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

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

Compile to Use OpenMP TARGET

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

C/C++

- Linux:

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

- Windows (you can use icx or icpx):

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

Fortran

- Linux:

```
ifx -fopenmp -fopenmp-targets=spir64 code.f90
```

- Windows:

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

Additional OpenMP Offload Resources

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

- [Matrix Multiplication](#) is a simple program that multiplies together two large matrices and verifies the results. This program is implemented using two ways: SYCL* and OpenMP.
- The [ISO3DFD OpenMP Offload](#) sample references three-dimensional finite-difference wave propagation in isotropic media. ISO3DFD is a three-dimensional stencil to simulate a wave propagating in a 3D isotropic medium and shows some common challenges and techniques when targeting OpenMP Offload 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.
- [LLVM/OpenMP Runtimes](#) describes the distinct types of runtimes available and can be helpful when debugging OpenMP offload.
- The [oneAPI GPU Optimization Guide](#) gives extensive tips for getting the best GPU performance for oneAPI programs.
- [Offload and Optimize OpenMP* Applications with Intel Tools](#) describes how to use OpenMP* directives to add parallelism to your application.
- [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.
- There are a number of useful OpenMP books. See the listing at: <https://www.openmp.org/resources/openmp-books>
- Details on using Intel compilers with OpenMP offload, including lists of supported options and example code, is available in the compiler developer guides:
 - [Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference](#)
 - [Intel® Fortran Compiler Developer Guide and Reference](#)

Device Selection

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

DPC++ Device Selection in the Host Code

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

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

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

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

`ONEAPI_DEVICE_SELECTOR` is a complex environment variable that allows you to limit the runtimes, compute device types, and compute device IDs that may be used by the DPC++ runtime to a subset of all available combinations. The compute device IDs correspond to those returned by the SYCL API, `clinfo`, or `syctl-ls` (with the numbering starting at 0). They have no relation to whether the device with that ID is of a certain type or supports a specific runtime. Using a programmatic special selector (like `gpu_selector`) to request a

filtered out device will cause an exception to be thrown. Refer to the environment variable description in GitHub for details on use and example values: <https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md>.

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

Device Selection Example

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

Additional information about device selection is available from the Khronos* [SYCL Reference](#).

OpenMP* Device Query and Selection in the Host Code

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

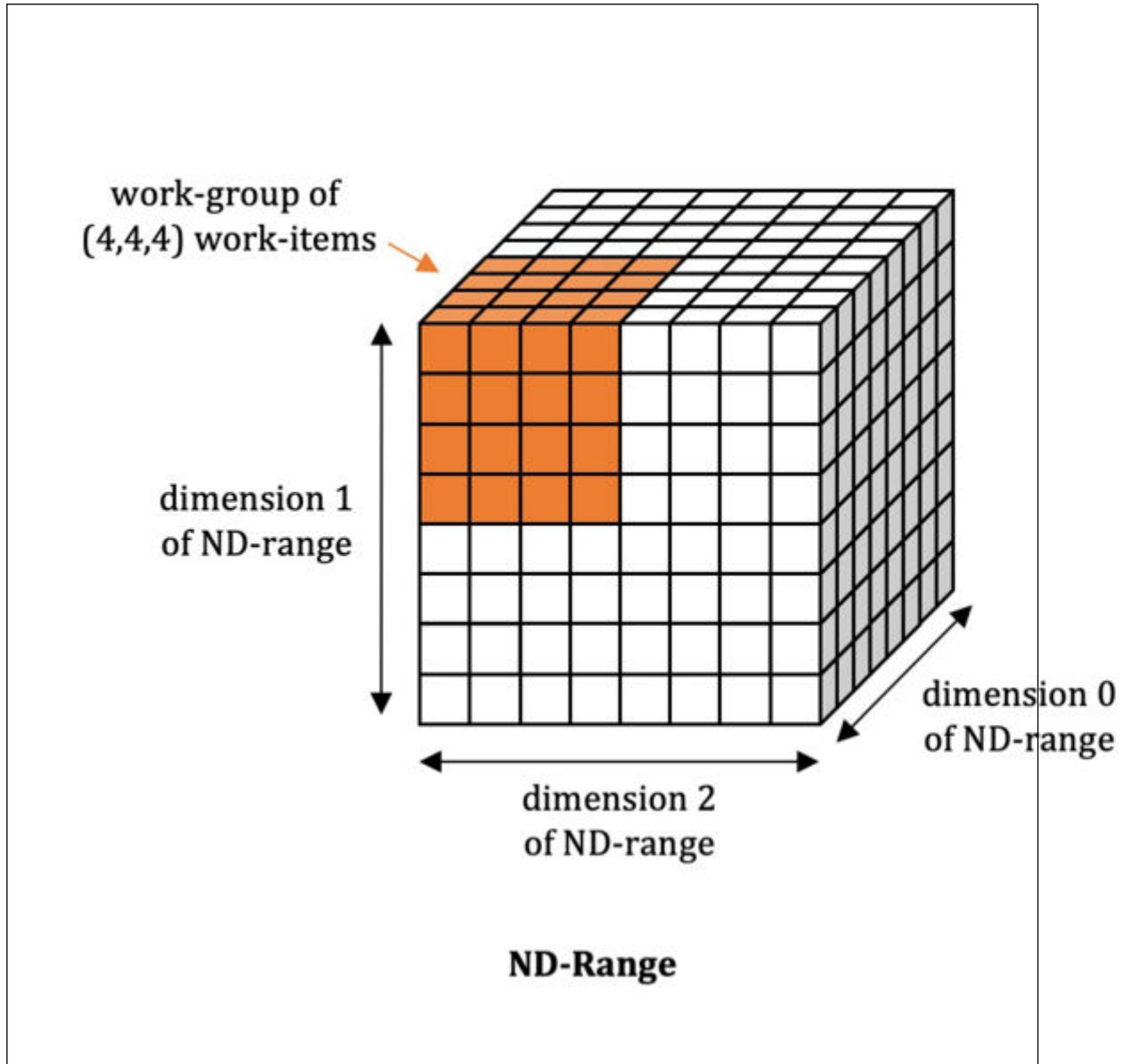
- `int omp_get_num_procs (void)` routine returns the number of processors available to the device.
- `void omp_set_default_device(int device_num)` routine controls the default target device for offloading code or data.
- `int omp_get_default_device(void)` routine returns the default target device.
- `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 also use the environment variable `LIBOMPTARGET_DEVICE_TYPE = [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` has the value `=mandatory`, otherwise, the execution will have a fallback execution on its initial device. Additional information about device selection is available from the OpenMP 5.2 specification. Details about environment variables are available from GitHub: <https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md>.

SYCL* Execution and Memory Hierarchy

Execution Hierarchy

The SYCL* execution model exposes an abstract view of GPU execution. The SYCL execution hierarchy consists of a 1-, 2-, or 3-dimensional grid of work-items. These work-items are grouped into equal-sized groups called work-groups. Work-items in a work-group are further divided into equal-sized groups called sub-groups.



To learn more about how this hierarchy works with a GPU or a CPU with Intel® UHD Graphics, see [SYCL* Mapping and GPU Occupancy](#) in the oneAPI GPU Optimization Guide.

Optimizing Memory Access

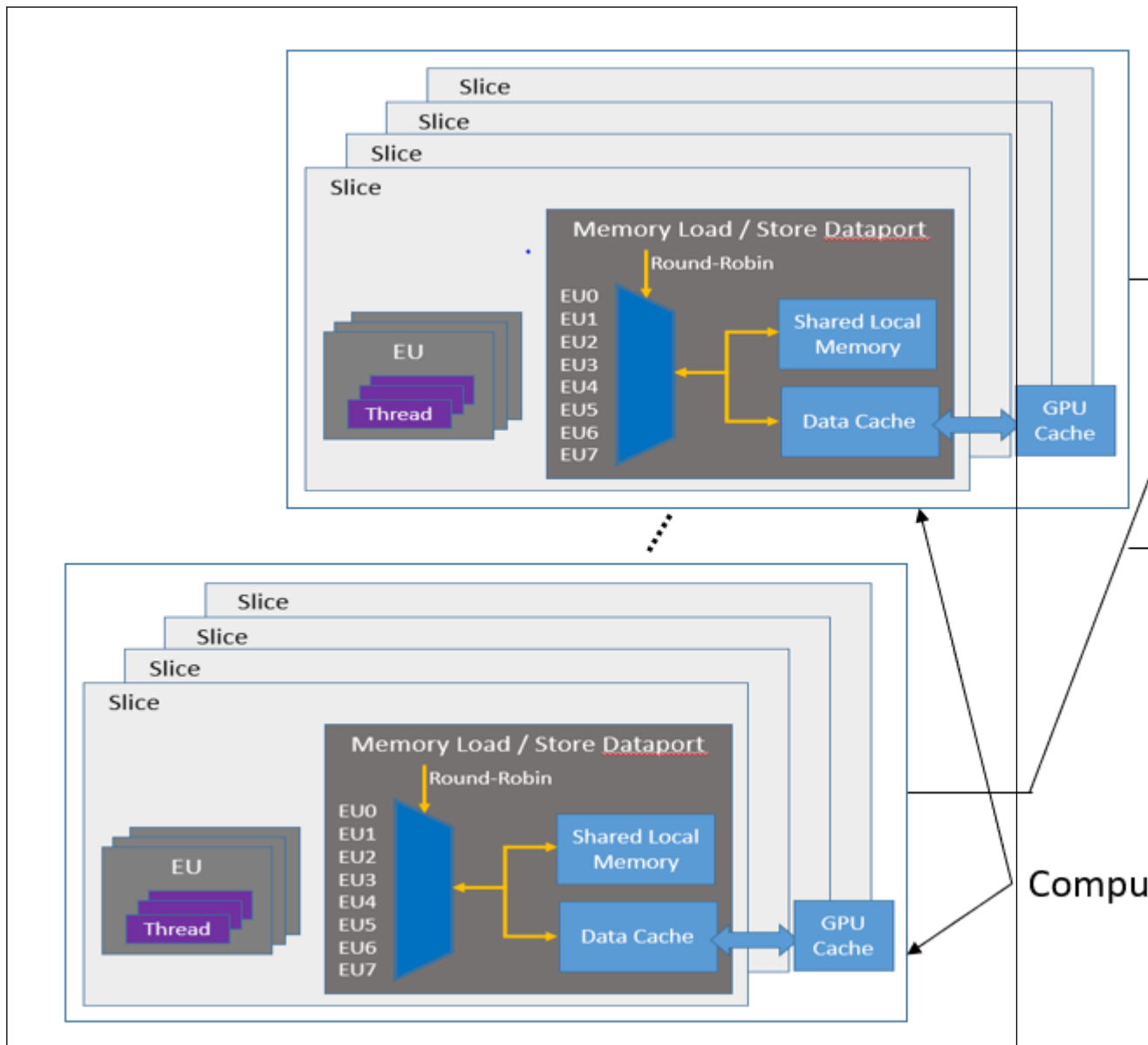
Using Intel® VTune Profiler, you can identify memory bottlenecks that are hindering performance. For more information, see the Memory Allocation APIs section of the Intel VTune User Guide.

Once the problem areas are identified, utilize the tools described in the Intel® oneAPI GPU Optimization Guide to learn how work-items in a kernel can synchronize to exchange data, update data, or cooperate with each other to accomplish a task. For more information, see these sections in the GPU Optimization Guide:

- [Synchronization among Work-items in a Kernel](#)
- [Shared Local Memory](#)
- [Unified Shared Memory Allocations](#)

Memory Hierarchy

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



In some cases, a `kernel_bundle` created with a given device will not contain kernels that are compatible, due to behavior required in the SYCL 2020 specification. For more information, see [SYCL Kernel Compatibility](#).

To learn more about memory hierarchy within the General Purpose GPU (GPGPU) compute model, see [Execution Model Overview](#) in the oneAPI GPU Optimization Guide.

Using Data Prefetching to Reduce Memory Latency in GPUs

Utilizing data prefetching can reduce the amount of write backs, reduce latency, and improve performance in Intel® GPUs.

To learn more about how prefetching works with oneAPI, see [Prefetching](#) in the oneAPI GPU Optimization Guide.

oneAPI Development Environment Setup

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

Installation Directories

On a Windows* system, the Intel oneAPI development toolkits (Base, HPC, Rendering, etc.) are typically installed in the `C:\Program Files (x86)\Intel\oneAPI\` directory which is known as the Component Directory Layout. When installing a toolkit, a Unified Directory Layout is also created and linked to the Component Directory Layout in the `C:\Program Files (x86)\Intel\oneAPI\<<toolkit-version>` folder.

On a Linux* system, the Intel oneAPI development toolkits (Base, HPC, Rendering, etc.) are typically installed in the `/opt/intel/oneapi/` directory which is known as the Component Directory Layout. When installing a toolkit, a Unified Directory Layout is also created and linked to the Component Directory Layout in the `/opt/intel/oneapi/<toolkit-version>/` folder.

To learn more about the differences between the Component Directory Layout and Unified Directory Layout, see [Use the setvars and oneapi-vars scripts with Windows](#) or [Use the setvars and oneapi-vars scripts with Linux](#)

NOTE Starting with the 2024.0 release, macOS is no longer supported in Intel® oneAPI Toolkits and components. Several Intel-led open source developer tool projects will continue supporting macOS on Apple Silicon including oneAPI Threading Building Blocks (oneTBB) and Intel® Implicit SPMD Program Compiler. We welcome the opportunity to work with contributors to expand support to additional tools in the future.

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 (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, oneapi-vars, 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 `env/vars.sh` on Linux; `setvars.bat` and `env\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 Unified Directory Layout was implemented in 2024.0. It utilizes a top-level `oneapi-vars` script to initialize the common environment variables and relies on optional `etc/*/vars.sh` (on Linux) and `etc*\vars.bat` (on Windows) scripts to initialize component-specific environment variables that are not addressed by the `oneapi-vars` script.

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

- [Use the setvars and oneapi-vars Scripts with Windows*](#)
- [Use the setvars and oneapi-vars Scripts with Linux*](#)

Install GPU Drivers or Plugins (Optional)

You can develop oneAPI applications using C++ and SYCL* that will run on Intel, AMD*, or NVIDIA* GPUs.

To develop and run applications for specific GPUs you must first install the corresponding drivers or plugins:

- To use an Intel GPU, install the [latest Intel GPU drivers](#).
- To use an AMD* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for AMD GPUs plugin](#) from Codeplay (Linux only).
- To use an NVIDIA* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for NVIDIA GPUs plugin](#) from Codeplay (Linux and Windows).

Modulefiles (Linux only)

Users of [Environment Modules](#) and `Lmod` can use the modulefiles included with the oneAPI toolkit installation to initialize their development environment variables. The oneAPI modulefile scripts are only supported on Linux and are provided as an alternative to using the `setvars`, `oneapi-vars`, and `vars` scripts referenced above. In general, users should not mix modulefiles with the `setvars` or `oneapi-vars` 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 and oneapi-vars Scripts with Windows*

The Unified Directory Layout was implemented in 2024.0. If you have multiple toolkit versions installed, the Unified layout adds the ability to ensure your development environment contains the component versions that were released as part of that specific toolkit version and it shortens the PATH names to help fix problems with long PATH names.

With the new Unified Directory Layout, you will see components installed together in a collection of common folders (eg., bin, lib, include, share, etc.). These common folders are located in a top-level folder that is named for the toolkit version number. For example:

```
"C:\Program Files(x86)\Intel\oneAPI\2024.0\"
|-- bin
|-- lib
|-- include
...etc...
```

The directory layout that was used prior to 2024.0 is still supported on new and existing installations. This prior layout is called the Component Directory Layout. Now you have the option to use the Component Directory Layout or the Unified Directory Layout.

Differences in Component Directory Layout and Unified Directory Layout

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

In the Component Directory Layout, the component `env\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 Component Directory Layout installation: `C:\Program Files (x86)\Intel\oneAPI\setvars.bat`. The Unified Directory Layout does not use the `env\vars.sh` scripts to initialize the development environment. Instead, each component is “corralled” into shared folders that are common to the components. In other words, each component contributes its header files to a single common `include` folder, its library files to a single common `lib` folder, and so on.

Advantages of the Unified Directory Layout

The Unified Directory Layout makes it much easier to switch between different toolkit versions without having to build and maintain `setvars` config files or play games with installation of multiple Intel® oneAPI toolkits. It is also useful for limiting the length of environment variables, especially the PATH variable, on Windows development systems, a troublesome issue for some Windows developers.

The Unified Directory Layout environment variables can only be setup collectively. To initialize the development environment variables run the script named `oneapi-vars.bat`. In a default Unified Directory Layout installation, on a Windows machine, that script is located here: `C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\oneapi-vars.bat`. The `<toolkit-version>` corresponds to the version number of the oneAPI toolkit that you installed. For example: `C:\Program Files (x86)\Intel\oneAPI\2024.0\oneapi-vars.bat` or `C:\Program Files (x86)\Intel\oneAPI\2024.1\oneapi-vars.bat`.

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

Running the `oneapi-vars.bat` script without any arguments causes it to configure the environment for the specific toolkit version in which that `oneapi-vars.bat` script is located. It will also run any optional `C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\etc\<component>\vars.sh` scripts that are part of that Unified Directory installation. Changes made to the environment by these scripts can be seen by running the Windows `set env` command after running the `oneapi-vars.bat` script.

To learn more about how `oneapi-vars.bat` works, see [Environment Initialization in the Unified Directory Layout](#)

Visual Studio Code Extension

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

NOTE Changes to your environment made by running the `setvars.bat | oneapi-vars.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 | oneapi-vars.bat` environment script was executed.

Command-Line Arguments

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

Component Directory Layout

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

Unified Directory Layout

```
"C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\oneapi-vars.bat" --help
```

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

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

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

If more than one version of Microsoft Visual Studio* is installed on your system, you can specify which Visual Studio environment should be initialized as part of the oneAPI `setvars.bat` | `oneapi-vars.bat` environment initialization by adding the `vs2017`, `vs2019`, or `vs2022` argument to the `setvars.bat` | `oneapi-vars.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

Component Directory Layout

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

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

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

Unified Directory Layout

```
<install-dir>\<toolkit-version>\oneapi-vars.bat
```

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

```
cmd.exe "/K" "C:\Program Files (x86)\Intel\oneAPI\<toolkit-version>\oneapi-vars.bat" && powershell'
```

How to Verify

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

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

Return value

```
SETVARS_COMPLETED=1
```

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

Using the VS####INSTALLDIR environment variables

If you installed Visual Studio in a non-standard location, or you installed only the Visual Studio Build Tools (not the full Visual Studio IDE) you may experience an issue where the `setvars.bat` script cannot locate the Visual Studio `vcvarsall.bat` script. In such a case you can use the `VS####INSTALLDIR` environment variables to locate the appropriate Visual Studio installation. For example, if only the Visual Studio Build Tools are installed, you would need to do the following to make `setvars.bat` work:

```
> set "VS2022INSTALLDIR=%ProgramFiles(x86)%\Microsoft Visual Studio\2022\BuildTools"  
> "%ProgramFiles(x86)%\Intel\oneAPI\setvars.bat"
```

You can also use the `VS####INSTALLDIR` environment variables to force `setvars.bat` to configure the environment for a specific installation of Visual Studio, especially if you have multiple copies of Visual Studio installed on your system.

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` | `oneapi-vars.bat` script will not allow multiple invocations of itself in the same session. This is done to ensure that your environment variables do not

exceed the maximum provided environment space, especially the `%PATH%` environment variable. Exceeding the available environment space results in unpredictable behavior in your terminal session and should be avoided.

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

Component Directory Layout

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

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

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

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

Unified Directory Layout

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

> <install-dir>\<toolkit-version>\oneapi-vars.bat
.. code-block:: WARNING: oneapi-vars.bat has already been run. Skipping re-execution.
   To force a re-execution of oneapi-vars.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>\<toolkit-version>\oneapi-vars.bat --force` and the initialization is successful.

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

Environment Initialization in the Unified Directory Layout

Initializing the environment for the Unified Directory Layout is done by the `oneapi-vars.bat` script, not the `setvars.bat` script. The usage of `oneapi-vars` is similar to `setvars`, but there are some subtle differences.

The key difference between the `setvars` script and the `oneapi-vars` script is that the `setvars` script does not define any environment variables (other than `ONEAPI_ROOT`) and the `oneapi-vars` script defines the common environment variables.

In the Component Directory Layout, each component is responsible for defining the environment variables needed in order to function. For example, in the Component Directory Layout each component adds its linkable library folders to `LD_LIBRARY_PATH` and include headers to `CPATH`, etc. The components do this via their individual vars scripts, which are always located in:

```
%ONEAPI_ROOT%\<toolkit-version>\opt\<component-name>\latest\env\vars.bat
```

The Unified Directory Layout combines the externally facing include, lib, and bin folders into a set of common folders. In this scenario, the top-level `oneapi-vars` script defines the environment variables that are needed to locate these common folders. For example, `setvars` will define `LD_LIBRARY_PATH` as `$ONEAPI_ROOT\lib` and `CPATH` as `$ONEAPI_ROOT\include` and so on.

ONEAPI_ROOT Environment Variable

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

With the 2024.0 release, the installer does not add the `ONEAPI_ROOT` variable to the environment. To add it to your default environment, define the variable in your local shell initialization file(s) or in the system environment variables.

Customizing the Call to the Microsoft Visual Studio* vcvarsall.bat Configuration Script

The Intel oneAPI development environment includes support for working with your Visual Studio project at the command prompt. For example, see the “Windows Start Menu > All apps > Visual Studio 2022” folder which typically contains shortcuts to multiple preconfigured Visual Studio setup scripts, such as the “Developer Command Prompt for VS 2022” shortcut. These shortcuts call a Microsoft Visual Studio configuration batch file named `vcvarsall.bat` which is typically found in the “%ProgramFiles%\Microsoft Visual Studio\2022\Professional\VC\Auxiliary\Build\” directory.

If you call the Intel oneAPI environment setup script (`setvars.bat`) at a command prompt, it will call the Visual Studio `vcvarsall.bat` script as part of the oneAPI environment setup process. Normally, the `vcvarsall.bat` environment setup is configured to match the `setvars.bat` environment setup. For example, it insures that the Visual Studio `vcvarsall.bat` is setup for 64-bit application development if `setvars.bat` is setup for a 64-bit oneAPI development environment (the default case).

If `setvars.bat` (or the compiler’s `env\vars.bat`) detects that the Visual Studio `vcvarsall.bat` has already been run, the `vcvarsall.bat` will **not** be run a second time. In other words, the `setvars.bat` script will honor the pre-existing Visual Studio `vcvarsall.bat` environment and configure itself to match (64-bit with 64-bit or 32-bit with 32-bit).

Other than the 32-bit or 64-bit arguments, the `setvars.bat` and compiler’s `env\vars.bat` scripts do not pass any arguments to the `vcvarsall.bat` script. If you wish to further customize the Visual Studio environment, you must do so before running `setvars.bat` or the compiler’s `env\vars.bat` script. For example:

- run `vcvarsall.bat` directly with the necessary arguments
- run `setvars.bat` (or the compiler’s `env\vars.bat`) script

At which point, your command prompt development environment will be configured.

The available `vcvarsall.bat` arguments for a Visual Studio 2022 Professional installation can be reviewed by typing:

```
> "%ProgramFiles%\Microsoft Visual Studio\2022\Professional\VC\Auxiliary\Build\vcvarsall.bat"
help
```

at your Windows command prompt.

Using the VS2022INSTALLDIR and VS2019INSTALLDIR Environment Variables

If you see a message similar to the following, when running `setvars.bat` or `oneapi-vars.bat` or the compiler `<version>\env\vars.bat` script:

```
WARNING: Visual Studio was not found in a standard install location:
"%ProgramFiles%\Microsoft Visual Studio\<Year>\<Edition>" or
"%ProgramFiles(x86)%\Microsoft Visual Studio\<Year>\<Edition>"
Set the VS2019INSTALLDIR or VS2022INSTALLDIR
environment variable to point to your install location and try again.
```

It likely means one of the following is true:

- Microsoft Visual Studio* has not been installed.
- Microsoft Visual Studio has been installed in a non-standard location.
- Only the Microsoft Build Tools* have been installed.

In the case of the first bulleted item, install Microsoft Visual Studio. Once installed, try running the `env` scripts again.

In case of the second bulleted item, configure the `VS2022INSTALLDIR` environment variable (or `VS2019INSTALLDIR` if you are using Visual Studio 2019) to point to the non-standard location of your Microsoft Visual Studio installation, prior to running the environment setup script. For example, assume a non-standard location of Visual Studio 2022 Professional and `setvars.bat`:

```
> set "VS2022INSTALLDIR=C:\my\custom\install\path\Microsoft Visual Studio\2022\Professional"
> "%ProgramFiles(x86)%\Intel\oneAPI\setvars.bat"
```

Or, if you have installed the Visual Studio 2022 Build Tools into their standard location you must set `VS2022INSTALLDIR` to point to that install location. For example, using `setvars.bat`:

```
> set "VS2022INSTALLDIR=%ProgramFiles(x86)%\Microsoft Visual Studio\2022\BuildTools"
> "%ProgramFiles(x86)%\Intel\oneAPI\setvars.bat"
```

Use a Config File for setvars bat on Windows*

The `setvars.bat` script sets environment variables for use with the Intel® 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](#).

NOTE Configuration files can only be used with `setvars.bat` in the Component Directory Layout. The Unified Directory Layout utilizes `oneapi-vars.bat`, which does not support configuration files. To learn more about the layouts, see [Use the setvars and oneapi-vars Scripts with Windows*](#)

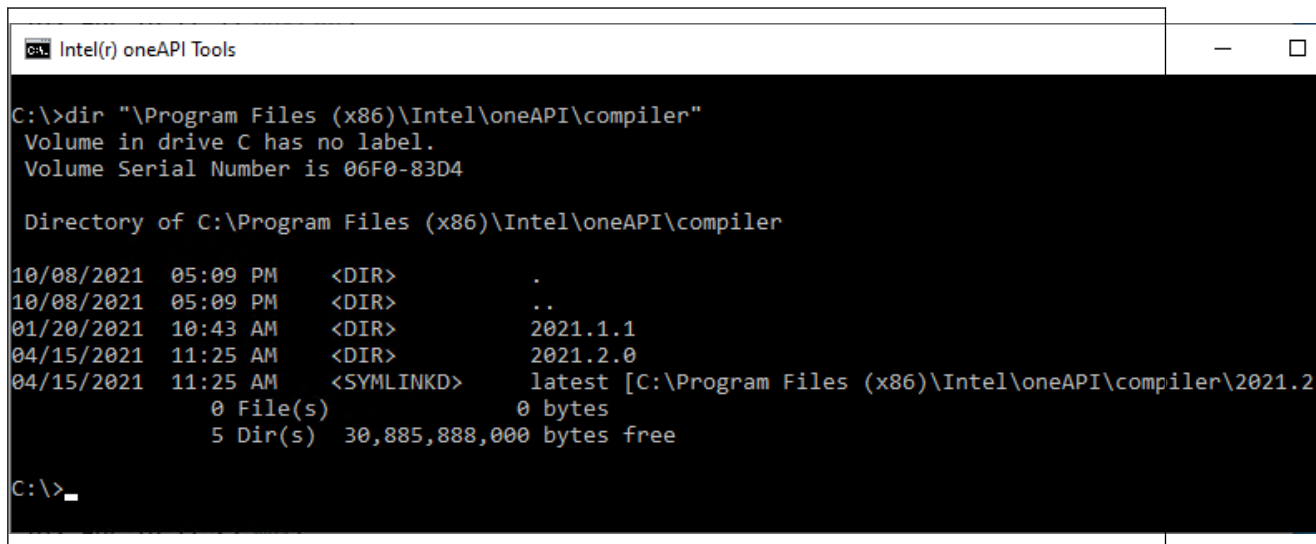
The procedure below describes how to use a configuration file to manage environment variables.

Versions and Configurations

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

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

For example:



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

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

10/08/2021  05:09 PM    <DIR>          .
10/08/2021  05:09 PM    <DIR>          ..
01/20/2021  10:43 AM    <DIR>          2021.1.1
04/15/2021  11:25 AM    <DIR>          2021.2.0
04/15/2021  11:25 AM    <SYMLINKD>    latest [C:\Program Files (x86)\Intel\oneAPI\compiler\2021.2
          0 File(s)          0 bytes
          5 Dir(s)  30,885,888,000 bytes free

C:\>_
```

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

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

-config Parameter

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

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

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

Config File Sample

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

Load Latest of Everything but...

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

Exclude Everything but...

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

The configuration text file must follow these requirements:

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

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

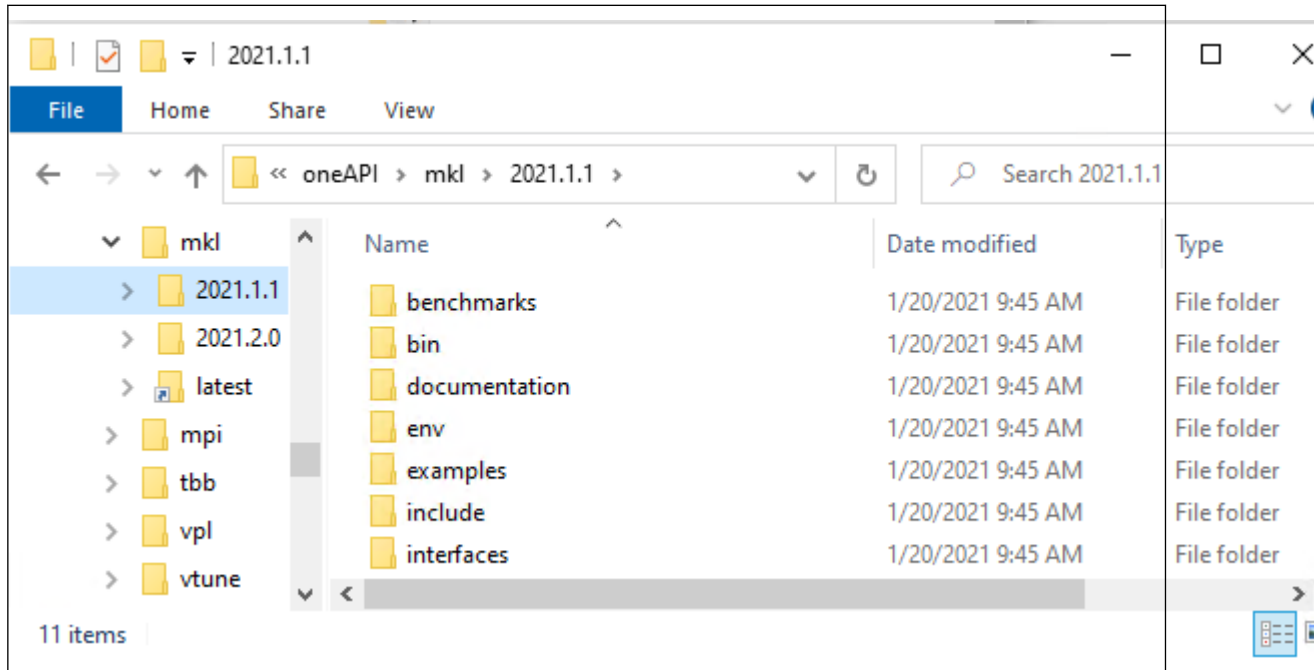
Further Customization of Config Files

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

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

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

Two versions of oneMKL and config files



Specify a Specific Version

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

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

Exclude Specific Components

To exclude a component, use the following syntax:

```
<key>=exclude
```

For example, to exclude Intel® Integrated Performance Primitives (Intel® IPP), but include the 2021.1.1 version of Intel® oneAPI Math Kernel Library (oneMKL):

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

In this example:

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

Include Specific Components

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

```
default=exclude
```

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

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

In this example:

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

Automate the setvars bat Script with Microsoft Visual Studio*

The `setvars.bat` script sets up the environment variables needed to use the Intel® 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.

To learn more about how `setvars.sh` set environment variables, see [Use the setvars and oneapi-vars Scripts with Windows*](#)

SETVARS_CONFIG Environment Variable States

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

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

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

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

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

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

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

Define the `SETVARS_CONFIG` Environment Variable

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

Use the `setvars` and `oneapi-vars` Scripts with Linux*

The Unified Directory Layout was implemented in 2024.0. If you have multiple toolkit versions installed, the Unified layout adds the ability to ensure your development environment contains the component versions that were released as part of that specific toolkit version.

With the new Unified Directory Layout, you will see components installed together in a collection of common folders (e.g., `bin`, `lib`, `include`, `share`, etc.). These common folders are located in a top-level folder that is named for the toolkit version number. For example:

```
/opt/intel/oneapi/2024.0/  
|-- bin  
|-- lib  
|-- include  
...etc...
```

The directory layout that was used prior to 2024.0 is still supported on new and existing installations. This prior layout is called the Component Directory Layout. Now you have the option to use the Component Directory Layout or the Unified Directory Layout.

Differences in Component Directory Layout and Unified Directory Layout

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

In the Component Directory Layout the component `env/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 Component Directory Layout installation on a Linux machine: `/opt/intel/oneapi/setvars.sh`. The Unified Directory Layout does not use the `env/vars.sh` scripts to initialize the development environment. Instead, each component is "corralled" into shared folders that are common to the components. In other words, each component contributes its header files to a single common `include` folder, its library files to a single common `lib` folder, and so on.

Advantages of the Unified Directory Layout

The Unified Directory Layout makes it much easier to switch between different toolkit versions without having to build and maintain `setvars` config files or play games with installation of multiple Intel® oneAPI toolkits. It is also useful for limiting the length of environment variables, especially the `PATH` variable, on Windows development systems, a troublesome issue for some Windows developers.

The Unified Directory Layout environment variables can only be setup collectively. To initialize the development environment variables a script named `oneapi-vars.sh`. In a default Unified Directory Layout installation, on a Linux machine, that script is located here: `/opt/intel/oneapi/<toolkit-version>/oneapi-vars.sh`. The `<toolkit-version>` corresponds to the version number of the oneAPI toolkit that you installed.

Example:

```
/opt/intel/oneapi/2024.0/oneapi-vars.sh
/opt/intel/oneapi/2024.1/oneapi-vars.sh
```

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

Sourcing the `oneapi-vars.sh` script without any arguments causes it to configure the environment for the specific toolkit version in which that `oneapi-vars.sh` script is located. It will also source any optional `/opt/intel/oneapi/<toolkit-version>/etc/<component>/vars.sh` scripts that are part of that Unified Directory installation. Changes made to the environment by these scripts can be seen by running the `env` command after sourcing the `oneapi-vars.sh` script.

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

To learn more about the differences in `setvars` and `oneapi-vars`, see [Environment Initialization in the Unified Directory Layout](#)

Command-Line Arguments

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

Component Directory Layout

For system-wide installations:

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

For private installations:

```
~/intel/oneapi/setvars.sh --help
```

Unified Directory Layout

For system-wide installations:

```
./opt/intel/oneapi/<version>/oneapi-vars.sh --help
```

For private installations:

```
~/intel/oneapi/<version>/oneapi-vars.sh --help
```

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

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

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

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

How to Run

Component Directory Layout

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

Unified Directory Layout

```
source <install-dir>/<toolkit-version>/oneapi-vars.sh
```

NOTE If you are using a non-POSIX shell, such as `csh`, use the following command:

Component Directory Layout

```
$ bash -c 'source <install-dir>/setvars.sh ; exec csh'
```

Unified Directory Layout

```
$ bash -c 'source <install-dir>/<toolkit-version>/oneapi-vars.sh ; exec csh'
```

If environment variables are set correctly, you will see a confirmation message similar to this:

```
:: initializing oneAPI environment ...
bash: BASH_VERSION = 4.4.20(1)-release
:: advisor -- latest
:: ccl -- latest
:: compiler -- latest
:: dal -- latest
:: debugger -- latest
:: dev-utilities -- latest
:: dnnl -- latest
:: dpcpp-ct -- latest
:: dpl -- latest
:: intelpython -- latest
:: ipp -- latest
:: ippcp -- latest
:: ipp -- latest
:: mkl -- latest
:: mpi -- latest
:: tbb -- latest
:: vpl -- latest
:: vtune -- latest
:: oneAPI environment initialized ::
ubuntu 1804:/opt/intel/oneapi$
```

If you receive an error message, troubleshoot using the Diagnostics Utility for Intel® oneAPI Toolkits, which provides system checks to find missing dependencies and permissions errors. [Learn more](#).

Alternatively, use the [modulefiles scripts](#) to set up your development environment. The modulefiles scripts work with all Linux shells.

If you wish to fine tune the list of components and the version of those components, use a [setvars config file](#) to set up your development environment.

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 / oneapi-vars` script will not allow multiple invocations of itself in the same session. This is done to ensure that your environment variables do not become too long due to redundant path references, especially the `$PATH` environment variable.

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

Component Directory Layout

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

Sourcing `setvars.sh` with the `--force` argument may lead to argument pollution with bash version 3.x and 4.x, as shown below:

```
source <install-dir>/setvars.sh --force
initializing oneAPI environment ...
(SNIP: lot of output)
oneAPI environment initialized ::
$ echo ${@}
advisor=latest ccl=latest compiler=latest dal=latest debugger=latest dev-utilities=latest
dnnl=latest dpcpp-ct=latest dpl=latest ipp=latest ippcp=latest mkl=latest mpi=latest tbb=latest
vtune=latest
```

Note: This is not an issue when `setvars.sh` is sourced with bash version 5.x, `zsh`, `ksh` or `dash`.

To work around this issue pass the shell command-line options via the `SETVARS_ARGS` environment variable.

For example:

```
$ SETVARS_ARGS="--force" source <install-dir>/setvars.sh
initializing oneAPI environment ...
(SNIP: lot of output)
oneAPI environment initialized ::
$ echo ${@}
$
```

Unified Directory Layout

```
$ source <install-dir>/<version>/oneapi-vars.sh
initializing oneAPI environment ...
(SNIP: lot of output)
oneAPI environment initialized ::
```

```
$ source <install-dir>/<toolkit-version>/oneapi-vars.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 `oneapi-vars.sh --force` and the initialization is successful.

```
$ source <install-dir>/`oneapi-vars.sh` --force
initializing oneAPI environment ...
(SNIP: lot of output)
oneAPI environment initialized ::
```

Sourcing `oneapi-vars.sh` with the `--force` argument may lead to argument pollution with bash version 3.x and 4.x, as shown below:

```
$ source <install-dir>/<toolkit-version>/oneapi-vars.sh --force
initializing oneAPI environment ...
(SNIP: lot of output)
oneAPI environment initialized ::
$ echo ${@}
advisor=latest ccl=latest compiler=latest dal=latest debugger=latest dev-utilities=latest
dnnl=latest dpcpp-ct=latest dpl=latest ipp=latest ippcp=latest mkl=latest mpi=latest tbb=latest
vtune=latest
```

Note: This is not an issue when `oneapi-vars.sh` is sourced with bash version 5.x, `zsh`, `ksh` or `dash`.

To work around this issue pass the shell command-line options via the `SETVARS_ARGS` environment variable.

For example:

```
$ SETVARS_ARGS="--force" source <install-path>/<toolkit-version>/oneapi-vars.sh
.. code-block:: initializing oneAPI environment ...
(SNIP: lot of output)
.. code-block:: oneAPI environment initialized ::
$ echo ${@}
$
```

Environment Initialization in the Unified Directory Layout

The Unified Directory Layout was implemented with the 2024.0 release. If you are unfamiliar with the change, please see [Use the setvars and oneapi-vars Scripts with Linux](#) at the top of this page.

Initializing the environment for a “unified” directory is done by the `oneapi-vars.sh` script, not the `setvars.sh` script. The usage of `oneapi-vars` is similar to `setvars`, but there are some subtle differences.

The key difference between the `setvars` script and the `oneapi-vars` script is that the `setvars` script does not define any environment variables (other than `ONEAPI_ROOT`) and the `oneapi-vars` script defines the common environment variables.

In the Component Directory Layout, each component is responsible for defining the environment variables needed in order to function. For example, in the Component Directory Layout each component adds its linkable library folders to `LD_LIBRARY_PATH` and include headers to `CPATH`, etc. The components do this via their individual vars scripts, which are always located in:

```
$ONEAPI_ROOT/<component-name>/<component-version>/env/vars.sh
```

The Unified Directory Layout combines the externally facing `include`, `lib`, and `bin` folders into a set of common folders. In this scenario, the top-level `oneapi-vars` script defines the environment variables that are needed to locate these common folders. For example, `setvars` will define `LD_LIBRARY_PATH` as `$ONEAPI_ROOT/lib` and `CPATH` as `$ONEAPI_ROOT/include` and so on.

Modulefiles continue to be supported in the 2024.0 release, and can be used as an alternative to using `setvars.sh` to initialize an environment setup. Modulefiles scripts are only supported on Linux.

ONEAPI_ROOT Environment Variable

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

With the 2024.0 release, the installer does not add the `ONEAPI_ROOT` variable to the environment. To add it to your 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

NOTE Starting with the 2024.0 release, macOS is no longer supported in Intel® oneAPI Toolkits and components. Several Intel-led open source developer tool projects will continue supporting macOS on Apple Silicon including oneAPI Threading Building Blocks (oneTBB) and Intel® Implicit SPMD Program Compiler. We welcome the opportunity to work with contributors to expand support to additional tools in the future.

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

- Use a `setvars.sh` configuration file, as described on this page
- Use [modulefiles](#)

NOTE Configuration files can only be used with `setvars.sh` in the Component Directory Layout. The Unified Directory Layout utilizes `oneapi-vars.sh`, which does not support configuration files. To learn more about the layouts, see [Use the setvars and oneapi-vars Scripts with Linux*](#)

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

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

Versions and Configurations

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

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

For example:

Multiple versions and environmental variables

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

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

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

-config Parameter

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

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

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

Config File Sample

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

Load Latest of Everything but...

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

Exclude Everything but...

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

The configuration text file must follow these requirements:

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

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

Further Customization of Config Files

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

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

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

Two versions of oneMKL installed

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

Specify a Specific Version

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

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

Exclude Specific Components

To exclude a component, use the following syntax:

```
<key>=exclude
```

For example, to exclude Intel® Integrated Performance Primitives (Intel® IPP), but include the 2021.1.1 version of Intel® oneAPI Math Kernel Library (oneMKL):

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

In this example:

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

Include Specific Components

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

```
default=exclude
```

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

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

In this example:

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

Automate the `setvars.sh` Script with Eclipse*

The `setvars.sh` script sets up the environment variables needed to use the Intel® 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.

To learn more about how `setvars.sh` set environment variables, see [Use the `setvars` and `oneapi-vars` Scripts with Linux*](#)

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` or `oneapi-vars.sh` on Linux](#).

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

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

If you leave the `setvars` config file empty, the `setvars.sh` script will initialize your environment for *all* oneAPI tools that are installed on your system. This is equivalent to defining the `SETVARS_CONFIG` variable with an empty string. See [Use a Config File for `setvars.sh` or `oneapi-vars.sh` on Linux](#) 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 Environment Modulefiles with Linux*

Modulefiles can be used to setup your environment, allowing you to specify the precise versions of components you wish to use.

There are two methods for customizing your environment variables in Linux*:

- Use modulefiles, as described on this page
- Use a [setvars.sh configuration file](#)

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

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

- Tmod version 4.2
- Tcl version 8.4
- Lmod version 8.7.44

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

```
module --version
```

Test which version of Tcl is installed on your system with the following commands:

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

Each modulefile automatically verifies the Tcl version on your system when it runs, but it does not test the version of Tmod on your system.

If your modulefile version is not supported (older than 4.2), a workaround may be possible. See [Using Environment Modules with Intel Development Tools](#) for more details.

As of the oneAPI 2024.0 release, the `icc` modulefile has been removed because the Intel® Classic Compiler has been discontinued. Please use the Intel® oneAPI C/C++ Compiler (`icx` and `icpx`) instead. The `ifort` compiler is still available, but you are encouraged to use the equivalent Intel® oneAPI Fortran Compiler (`ifx`).

Modulefile Auto-load

The “auto-load” feature in Environment Modules (Tmod) was introduced in version 4.2 of Modulefiles as an experimental feature. The auto-load feature was elevated to a standard feature in the 5.0 release. With the 2024.0 release of Intel oneAPI toolkits, automatic loading of dependent modulefiles now relies on the auto-load feature. If you are using version 4.x of Environment Modules, the auto-load feature of dependent modulefiles (those referenced by the `prereq` command) is disabled by default. Version 5.x enables auto-load of dependent modules by default.

To control the auto-load function, the `MODULES_AUTO_HANDLING` environment variable can be used to direct modulefiles to auto-load all referenced `prereq` modulefiles. This behavior can be temporarily overridden by using the `--auto` command-line option.

- `MODULES_AUTO_HANDLING=1` enables auto-loading via the `prereq` command.
- `MODULES_AUTO_HANDLING=0` disables auto-loading via the `prereq` command.

If you experience an error message with the text `HINT: the following module must be loaded first`, you may need to either add the `MODULES_AUTO_HANDLING=1` environment variable or configure your Modules Environment to set the `auto_handling` configuration to “1” in order to enable automatic loading of `prereq` commands.

```
$ module -V
Modules Release 4.4.1 (2020-01-03)

$ module use /opt/intel/oneapi/2024.1/etc/modulefiles/
$ module load compiler
Loading compiler/2024.1.0
ERROR: compiler/2024.1.0 cannot be loaded due to missing prereq.
      HINT: the following module must be loaded first: tbb
```

This issue can be resolved using one of the methods below.

Run command to configure `auto_handling` to be 1:

```
$ module config auto_handling 1
$ module load compiler
Loading compiler/2024.1.0
Loading requirement: tbb/2021.12 compiler-rt/2024.1.0 oclfpga/2024.1.0
```

Set `MODULES_AUTO_HANDLING=1`:

```
$ export MODULES_AUTO_HANDLING=1
module load compiler
Loading compiler/2024.1.0
Loading requirement: tbb/2021.12 compiler-rt/2024.1.0 oclfpga/2024.1.0
```

Load dependencies manually:

```
$ module load tbb compiler-rt oclfpga compiler --verbose
Loading tbb/2021.12
Loading compiler-rt/2024.1.0
Loading oclfpga/2024.1.0
Loading compiler/2024.1.0
```

For more details, see [MODULES_AUTO_HANDLING](#).

Load Everything Modulefile

As of the 2024.0 release, there is a “load everything” modulefile named `oneapi` located in `intel/oneapi/2024.0`. The `oneapi` modulefile loads all available modulefiles and ensures that all of the prereq modules are also loaded in the right order without relying on the `MODULES_AUTO_HANDLING=1` environment variable described above. It should be usable with the Lmod modulefiles system. If your installation does not include the `intel/oneapi/2024.0` directory install the product with a toolkit installer, such as the Intel oneAPI Base Toolkit and/or the Intel® HPC Toolkit (use the customize option if you do not wish to install the entire toolkit). The key feature of the `oneapi` modulefile is that it only loads those modulefiles that are specific to the toolkit version (e.g., `intel/oneapi/2024.0` or `intel/oneapi/2024.1`, etc.).

Create a Custom Modulefile To Load Specific Modulefiles

To further control which modulefiles are loaded, you can create a “meta” modulefile that loads a specific set of modulefiles (and their prerequisites). For example, if you have a need for an environment that only sets up the Intel compiler, you might create something similar to this:

```
module load tbb
module load compiler-rt
module load oclfpga
module load compiler
```

Location of Modulefile Scripts

The oneAPI modulefile scripts are located in a modulefiles directory inside each component folder (similar to where the individual vars scripts are located). For example, in a default installation for 2024.0 and later oneAPI Toolkit releases, the `compiler` modulefiles script(s) are in the `/opt/intel/compiler/<component-version>/etc/modulefiles/` directory. In 2023 or earlier oneAPI Toolkit releases, the `compiler` modulefiles script(s) are in the `/opt/intel/compiler/<component-version>/modulefiles/` directory.

NOTE `<component-version>` is the version number of a component (a library or tool). You may have multiple components installed side-by-side on your development system (e.g., `compiler/2023.1`, `compiler/2023.2`, `compiler/2024.0`, etc.). The Toolkit version, which is the version of a collection of components, may be different than the component version that was delivered with that Toolkit.

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

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

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

Alternatively, on a 2024.0 or later toolkit installation, within the Unified Directory Layout, you will find a `modulefiles` folder that contains all the component modulefiles associated with that toolkit version. For example, if you install the 2024.0 Intel® oneAPI Base Toolkit into the default location, you will find a preconfigured and ready to use collection of the 2024.0 Base Toolkit modulefiles located in the `/opt/intel/oneapi/2024.0/etc/modulefiles` folder, which you can add to your modulefile setup by adding that folder to your `MODULEPATH` environment variable, or by running the `module use` command and specify that pre-configured modulefile folder. Installing the 2024.0 Intel® HPC Toolkit will add additional modulefiles to that same location, because the two toolkits are of the same version (in this case, 2024.0).

Within the pre-configured toolkit modulefiles folder there is a modulefile named `oneapi` that will load the 64-bit component modulefiles that belong to that toolkit version. The `oneapi` modulefile is a convenience modulefile, it is not required to use the individual modulefiles. Also, the `oneapi` modulefile will not work as expected within a folder created by the `modulefiles-setup.sh` script.

Creating the `modulefiles` Directory

Run the `modulefiles-setup.sh` script.

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

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

```

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

```

Update your `MODULEPATH` environment variable to include to the `modulefiles` output folder that was created by the `modulefiles-setup.sh` script or run the `module use <folder_name>` command.

Installing the Tcl Modulefiles Environment onto Your System

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

Set your environment:

```

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

```

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

```

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

```

Test the module installation by initializing the `module` alias.

```

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

```

NOTE Initialization of the Modulefiles environment in POSIX-compatible shells should work with the source command shown above. However, the precise location and name of the modulefiles init folder does vary with the Linux distribution. Most installations of Environment Modules will automatically source the modulefiles initialization script: `/etc/profile.d/modules.sh`. You can test for proper initialization of the module command by typing the `module --version` command which should result in a response similar to: `Modules Release 4.4.1 (2020-01-03)`. At this point, the system should be ready to use the module command as shown in the following section.

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

The following example assumes you have:

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

Navigate to the installation directory, and load `tbb`:

```
$ cd <oneapi-root-folder>           # cd to the oneapi_root installation 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
```

- Use the `env` command to inspect the environment and look for the changes that were made by the modulefile you loaded.

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

For example, if you loaded the `tbb` modulefile, the command will show you some of the `env` changes made by that modulefile.

- Unload `tbb`:

```
$ module unload tbb                # removes tbb/X.Y changes from the environment
$ module list                       # should no longer list the tbb/X.Y env var module
```

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

Versioning

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

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

Multiple modulefiles

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

How `modulefiles` Are Set Up in oneAPI

Symbolic links are used by the `modulefiles-setup.sh` script to gather all the available `modulefiles` into a single `modulefiles` folder. The actual `modulefile` scripts are not moved or modified. The `$(ModulesCurrentModulefile)` variable points to the symlink to each `modulefile`, not to the actual `modulefile` located in the respective installation folders. To determine the full path to the real `modulefiles`, each `modulefile` includes a statement similar to this:

```
[ file normalize ${scriptpath} ]
```

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

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

Additional Resources

For more information about `modulefiles`, see:

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

Use CMake with oneAPI Applications

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

The following components support CMake:

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

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

NOTE Starting with the 2024.0 release, macOS is no longer supported in Intel® oneAPI Toolkits and components. Several Intel-led open source developer tool projects will continue supporting macOS on Apple Silicon including oneAPI Threading Building Blocks (oneTBB) and Intel® Implicit SPMD Program Compiler and we welcome the opportunity to work with contributors to expand support to additional tools in the future.

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

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

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

Compile and Run oneAPI Programs

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

Single-Source Compilation

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

- Usability – developers need to create fewer files and can define device code right next to the call site in the host code.

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

Invoke the Compiler

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

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

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

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

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

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

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

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

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

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

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

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

Example Compilation

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

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

API-Based Code

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

```
#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 | oneapi-vars.sh` script or run the `setvars.bat | oneapi-vars.bat` script or set the variable to the folder that contains the `lib` and `include` folders.

For more information about the `setvars` and `oneapi-vars` scripts, see [oneAPI Development Environment Setup](#).

2. Build the application using the following command:

On Linux:

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

On Windows:

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

3. Link the application using the following command:

On Linux:

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

On Windows:

```
icpx -fsycl axpy.obj -fsycl-device-code-split=per_kernel ^ \  
"${MKLRROOT}\lib\intel64"\mkl_sycl.lib ^ \  
"${MKLRROOT}\lib\intel64"\mkl_intel_ilp64.lib ^ \  
"${MKLRROOT}\lib\intel64"\mkl_sequential.lib ^ \  
"${MKLRROOT}\lib\intel64"\mkl_core.lib ^ \  
sycl.lib OpenCL.lib -o axpy.exe
```

4. Run the application using the following command:

On Linux:

```
./axpy.out
```

On Windows:

```
axpy.exe
```

Direct Programming

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

The following command compiles and links the executable.

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

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

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

Compilation Flow Overview

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

NOTE In addition to Intel® processors listed in the [System Requirements](#), AMD* (Linux* only) and NVIDIA* (Linux and Windows*) GPUs may also be targeted:

- To use an AMD* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for AMD GPUs plugin](#) from Codeplay.
- To use an NVIDIA* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for NVIDIA GPUs plugin](#) from Codeplay.

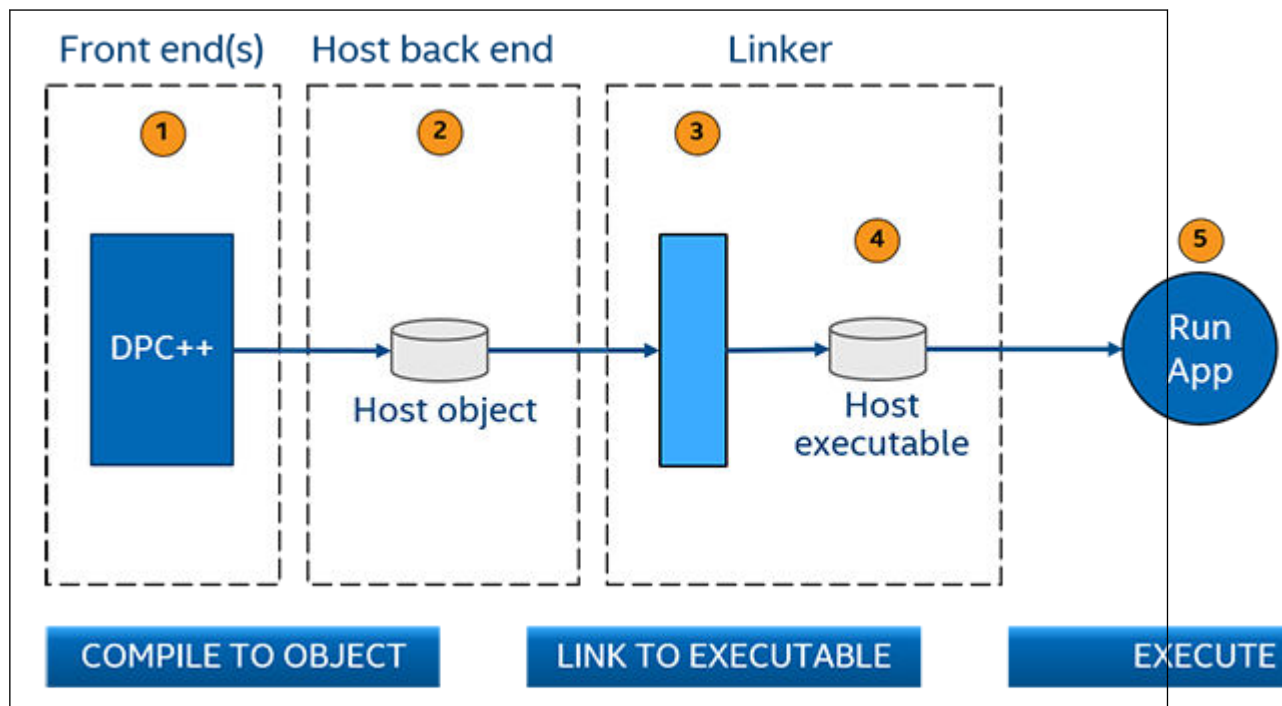
For device code, two options are available: Just-in-Time (JIT) compilation and Ahead-of-Time (AOT) compilation, with JIT being the default. This section describes how host code is compiled, and the two options for generating device code. Additional details are available in Chapter 13 of the [Data Parallel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

Traditional Compilation Flow (Host-only Application)

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

The traditional compilation phases are shown in the following diagram:

Traditional compilation phases



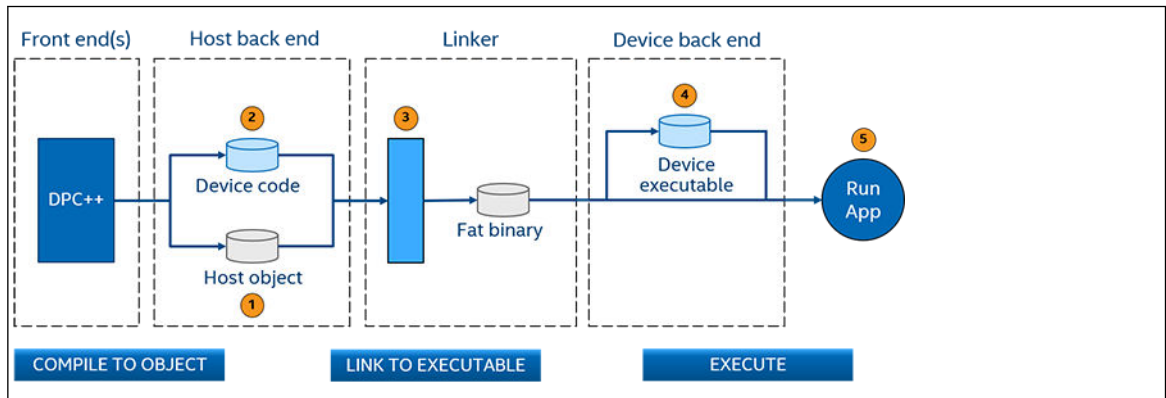
1. The front end translates the source into an intermediate representation and then passes that representation to the back end.
2. The back end translates the intermediate representation to object code and emits an object file (`host.obj` on Windows*, `host.o` on Linux*).
3. One or more object files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

Compilation Flow for SYCL Offload Code

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

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

Basic compilation phases for SYCL offload code



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

JIT Compilation Flow

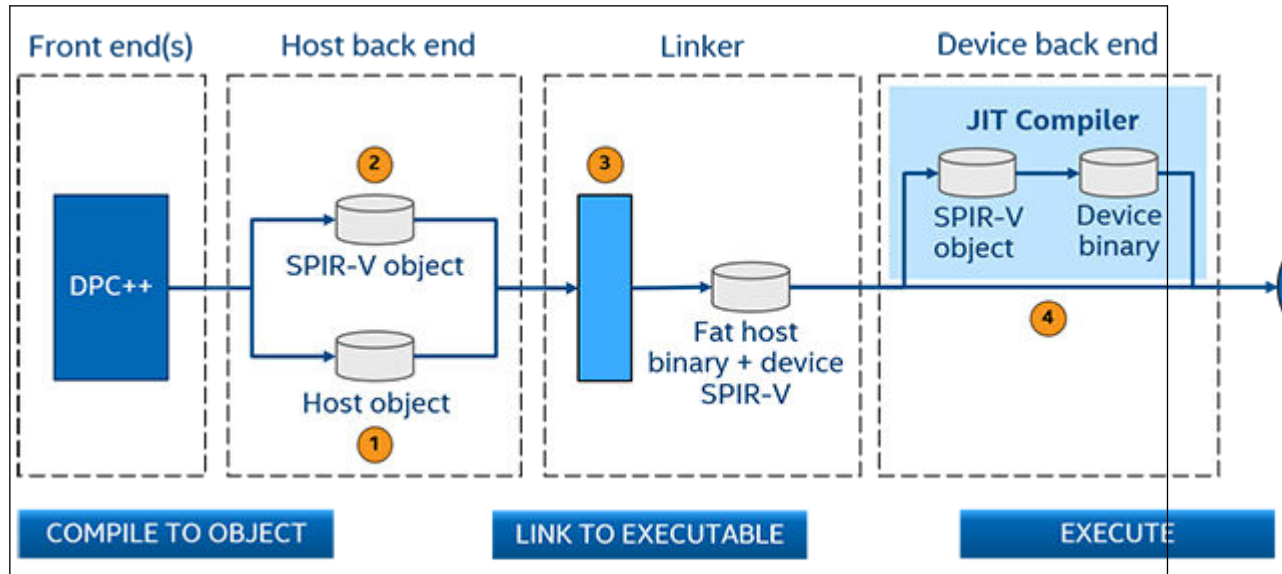
In the JIT compilation flow, the code for the device is translated to SPIR-V intermediate code by the back-end, embedded in the fat binary as SPRI-V, and translated from SPIR-V to device code by the runtime. When the application is run, the runtime determines the available devices and generates the code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

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

NOTE JIT compilation is not supported for FPGA devices.

The compilation phases are shown in the following diagram:

JIT compilation phases



1. The host code is translated to object code by the back end.
2. The device code is translated to SPIR-V.
3. The linker combines the host object code and the device SPIR-V into a fat binary containing host executable code with SPIR-V device code embedded in it.
4. At runtime:
 - a. The device runtime on the host translates the SPIR-V for the device into device binary code.
 - b. The device code is loaded onto the device.
5. The application runs on the host and device available at runtime.

AOT Compilation Flow

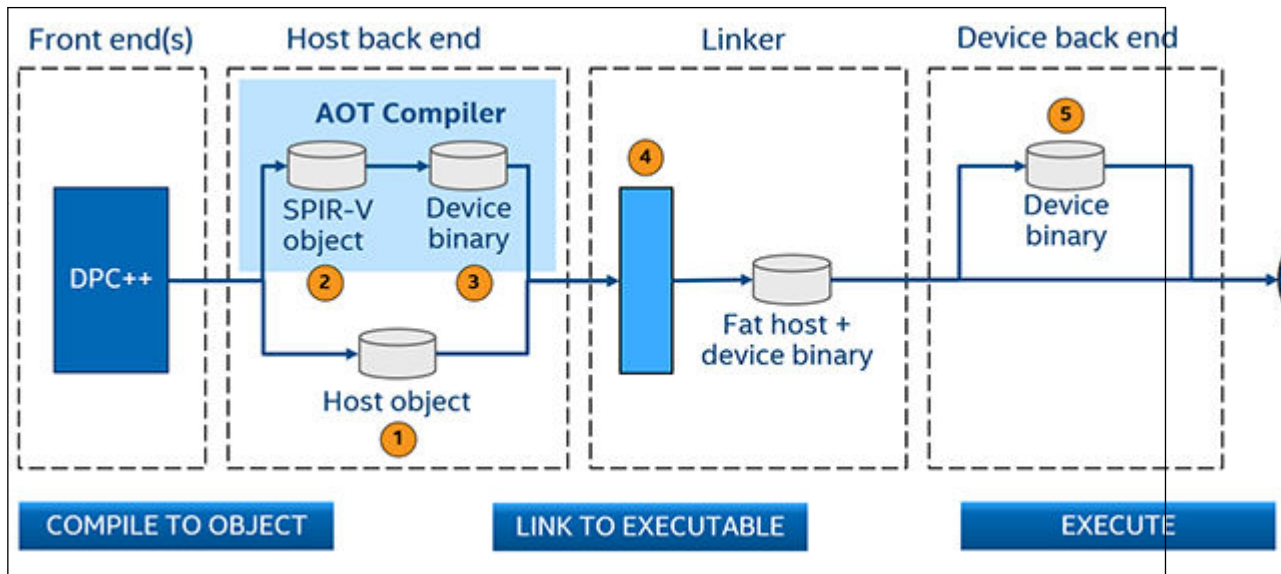
In the AOT compilation flow, the code for the device is translated to SPIR-V and then device code in the host back-end and the resulting device code is embedded in the generated fat binary. The AOT flow provides less flexibility than the JIT flow because the target device must be specified at compilation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

Tip

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

The compilation phases are shown in the following diagram:

AOT compilation phases

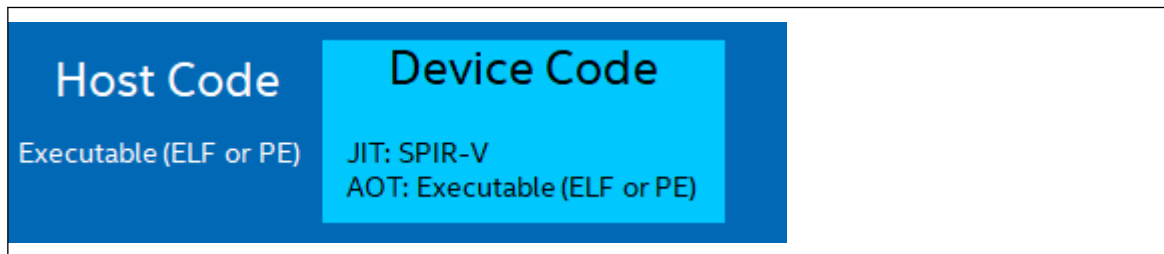


1. The host code is translated to object code by the back end.
2. The device code is translated to SPIR-V.
3. The SPIR-V for the device is translated to a device code object using the device specified by the user on the command line.
4. The linker combines the host object code and the device object code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

Fat Binary

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

FAT binary



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

- CPU: ELF (Linux), PE (Windows)
- GPU: ELF (Windows, Linux)
- FPGA: ELF (Linux), PE (Windows)

CPU Flow

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

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

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

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

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

Traditional CPU Flow

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

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

Example compilation command:

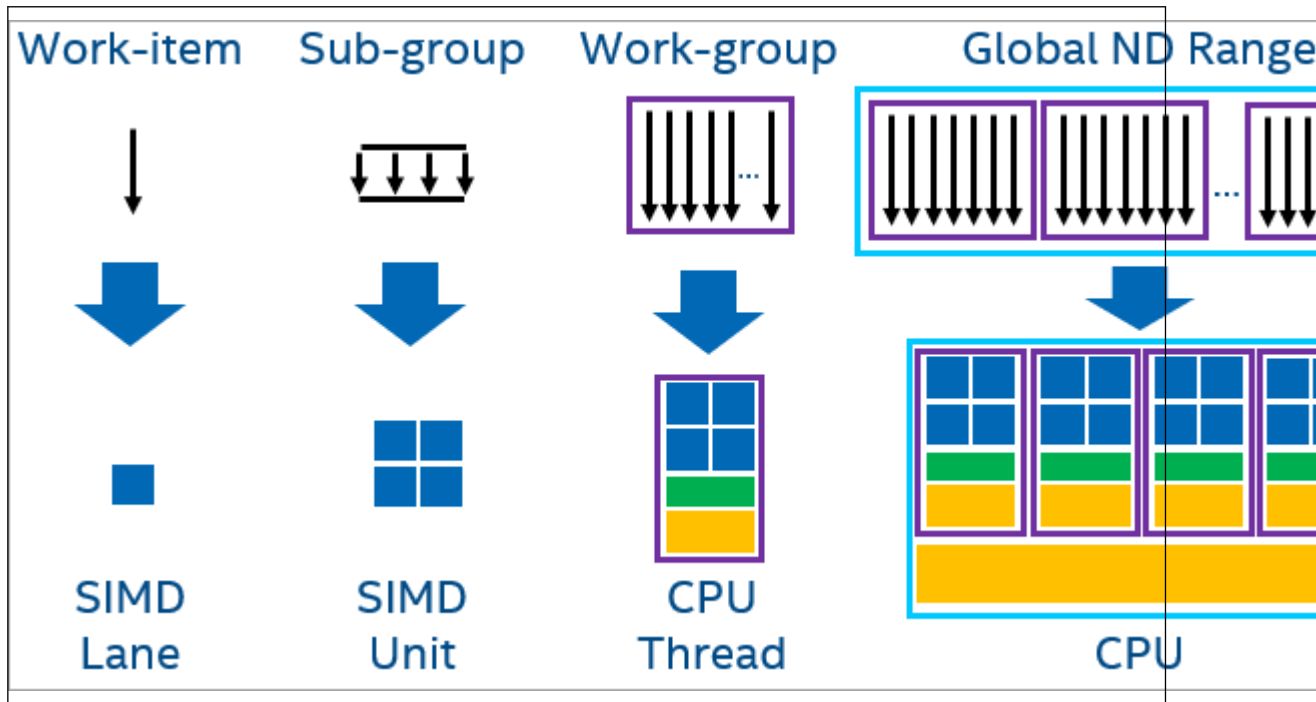
```
icpx -g -o matrix_mul_omp src/matrix_mul_omp.cpp
```

CPU Offload Flow

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

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

CPU workgroups



To learn more about CPU execution, see [Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads](#).

Set Up for CPU Offload

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

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

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

SYCL*

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

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

```
export ONEAPI_DEVICE_SELECTOR=cpu
```

SYCL code sample:

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

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

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

To compile the code sample, use:

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

Additional commands are available from [Example CPU Commands](#).

Results after compilation:

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

OpenMP*

OpenMP code sample:

```
#include<iostream>
#include<omp.h>
#define N 1024
int main(){
float *a = (float *)malloc(sizeof(float)*N);
```

```
for(int i = 0; i < N; i++)
a[i] = i;
#pragma omp target teams distribute parallel for simd map(tofrom: a[:N])
for(int i = 0; i < 1024; i++)
a[i]++;
std::cout << "Successfully completed on device.\n";
return 0;
}
```

To compile the code sample, use:

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

Setup the following environment variables before executing the binary to run the offload regions on the CPU:

```
export LIBOMPTARGET_DEVICE_TYPE=cpu
export LIBOMPTARGET_PLUGIN=opencl
```

Results after execution:

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

Offload Code to CPU

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

To find opportunities to offload your code, use the [Intel Advisor for Offload Modeling](#).

Debug Offloaded Code

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

- Check host target to verify the correctness of your code.
- Use `printf` to debug your application. Both SYCL and OpenMP offload support `printf` in kernel code.
- Use environment variables to control verbose log information.
 - For SYCL, the following debug environment variables are recommended. A full list of environment variables is available from [GitHub](#).

SYCL Recommended Debug Environment Variables

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

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

OpenMP Recommended Debug Environment Variables

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

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

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

Optimize CPU Code

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

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

Additional recommendations are available from [Optimize Offload Performance](#).

Example CPU Commands

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

NOTE Linking with a dynamic library is not supported.

Produce a fat object with device code:

```
icpx -fsycl -c static_lib.cpp
```

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

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

Compile application sources:

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

Link the application with the static library:

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

Ahead-of-Time Compilation for CPU Architectures

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

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

Supported CPU optimization flags are:

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

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

Control Binary Execution on Multiple CPU Cores

Environment Variables

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

SYCL* or OpenMP* environmental variables

Environment Variable	Description
DPCPP_CPU_CU_AFFINITY	Set thread affinity to CPU. The value and meaning is the following: <ul style="list-style-type: none">close - threads are pinned to CPU cores successively through available cores.spread - threads are spread to available cores.

Environment Variable	Description
DPCPP_CPU_SCHEDULE	<ul style="list-style-type: none"> • 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. <p>This environment variable is similar to the OMP_PROC_BIND variable used by OpenMP.</p> <p>Default: Not set</p> <p>Specify the algorithm for scheduling work-groups by the scheduler. Currently, the SYCL runtime uses Intel® oneAPI Threading Building Blocks (Intel® oneTBB) for scheduling. The value selects the petitioner used by the Intel oneTBB scheduler. The value and meaning is the following:</p> <ul style="list-style-type: none"> • dynamic - Intel oneTBB auto_partitioner. It performs sufficient splitting to balance load. • affinity - Intel oneTBB affinity_partitioner. It improves auto_partitioner's cache affinity by its choice of mapping subranges to worker threads compared to • static - Intel oneTBB static_partitioner. It distributes range iterations among worker threads as uniformly as possible. Intel oneTBB partitioner relies grain-size to control chunking. Grain-size is 1 by default, indicating every work-group can be executed independently. <p>Default: Dynamic</p>
DPCPP_CPU_NUM_CUS	<p>Set the numbers threads used for kernel execution.</p> <p>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.</p> <p>This environment variable is similar to OMP_NUM_THREADS variable used by OpenMP.</p> <p>Default: Not set. Determined by Intel oneTBB.</p>
DPCPP_CPU_PLACES	<p>Specify the places that affinities are set. The value is { sockets numa_domains cores threads }.</p> <p>This environment variable is similar to the OMP_PLACES variable used by OpenMP.</p> <p>If value is numa_domains, Intel oneTBB NUMA API will be used. This is analogous to OMP_PLACES=numa_domains in the OpenMP 5.1 Specification. Intel oneTBB task arena is bound to numa node and SYCL nd range is uniformly distributed to task arenas.</p>

Environment Variable	Description
	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.

Allocating Host Memory

When using OpenMP, you can allocate host memory so it can be shared with the device by using this API:

```
EXTERN void *llvm_omp_target_alloc_host(size_t Size, int DeviceNum)
```

For more information on memory allocation, see the [Level Zero Core Programming Guide](#).

Example 1: Hyper-threading Enabled

Assume a machine with two sockets and four physical cores per socket, where each physical core has two hyper-threads.

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

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

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

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

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

Example 2: Hyper-threading Disabled

Assume a machine with two sockets and four physical cores per socket, where each physical core has two hyper-threads.

- S<num> denotes the socket number that has eight cores specified in a list
- T<num> denotes the Intel oneTBB thread number
- “-” means unused core

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

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

GPU Flow

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

NOTE In addition to Intel® processors listed in the [System Requirements](#), AMD* and NVIDIA* GPUs may also be targeted (Linux only):

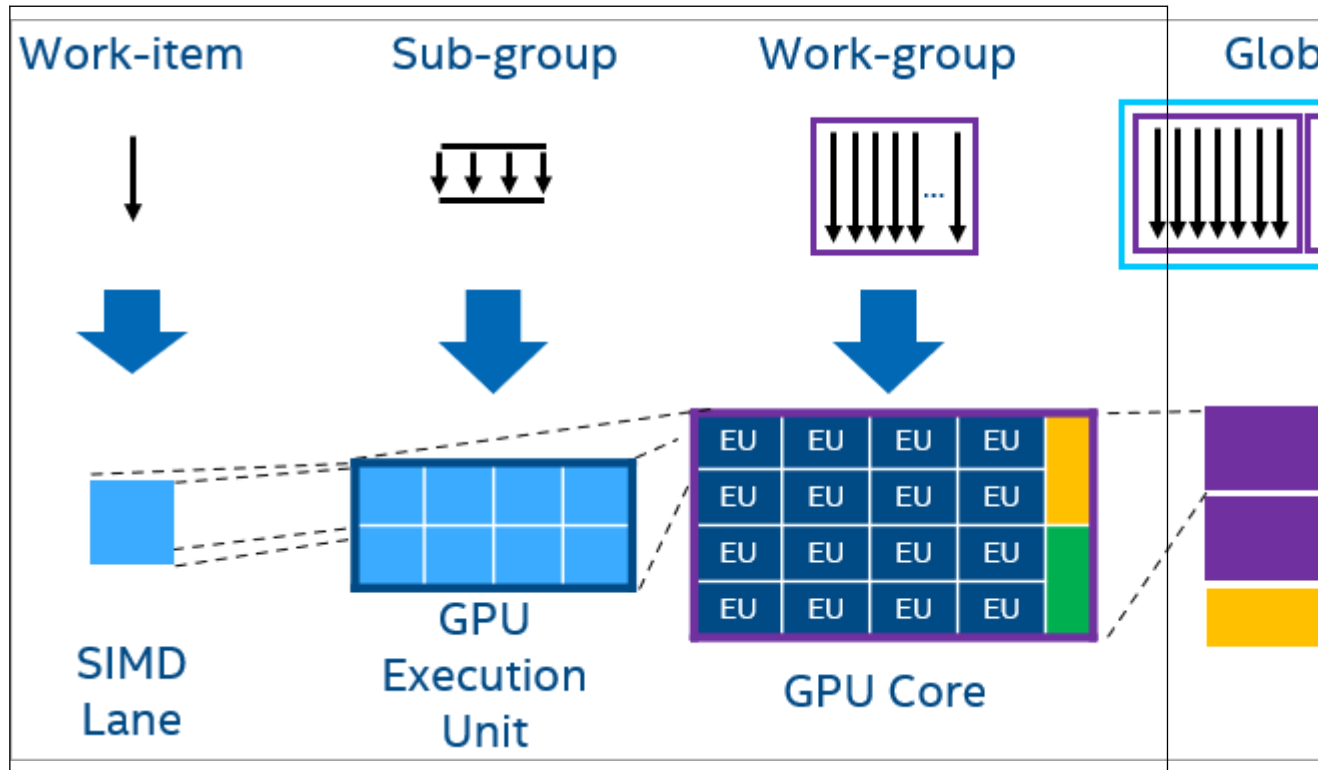
- To use an AMD* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for AMD GPUs plugin](#) from Codeplay.
 - To use an NVIDIA* GPU with the Intel® oneAPI DPC++ Compiler, install the [oneAPI for NVIDIA GPUs plugin](#) from Codeplay.
-

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

GPU Offload Flow

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

PRG Interface GPU workgroups



To learn more about GPU execution, see [Compare Benefits of CPUs, GPUs, and FPGAs for Different oneAPI Compute Workloads](#).

To learn more about Intel® Iris® Xe GPU Architecture, see the [GPU Optimization Guide](#).

Set Up for GPU Offload

1. Make sure you have followed all steps in the [oneAPI Development Environment Setup](#) section, including running the `setvars` or `oneapi-vars` script.
2. Configure your GPU system by installing drivers and add the user to the video group. See the [Get Started Guide](#) for instructions:
 - [Get Started with Intel® oneAPI Base Toolkit for Linux* | Windows*](#)
 - [Get Started with Intel® HPC Toolkit for Linux* | Windows*](#)
3. Check if you have a supported GPU and the necessary drivers installed using the `sycl-ls` command. In the following example, if you had the OpenCL and Level Zero driver installed you would see two entries for each runtime associated with the GPU:

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

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

SYCL

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

SYCL code sample:

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

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

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

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

To compile the code sample, use:

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

Results after compilation:

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

OpenMP*

OpenMP code sample:

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

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

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

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

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

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

To compile the code sample, use:

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

Results after compilation:

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

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

Offload Code to GPU

To decide which GPU hardware and what parts of the code to offload, refer to the [GPU optimization workflow guide](#).

To find opportunities to offload your code to GPU, use the [Intel Advisor for Offload Modeling](#).

Debug GPU Code

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

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

For SYCL, the following debug environment variables are recommended. A full list is available from [GitHub](#).

Debugging Tips, Offloaded Code

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

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

Recommended OpenMP Debug Environment Variables

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

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

CL_OUT_OF_RESOURCES Error

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

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

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

Or if using onetrace:

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

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

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

This will change the output to:

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

Or, if using onetrace:

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

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

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

Optimize GPU Code

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

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

Additional recommendations are available from [Optimize Offload Performance](#).

Example GPU Commands

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

NOTE Linking with a dynamic library is not supported.

Produce a fat object with device code:

```
icpx -fsycl -c static_lib.cpp
```

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

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

Compile application sources:

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

Link the application with the static library:

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

Ahead-of-Time Compilation for GPU

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

For DPC++:

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

For OpenMP*offload:

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

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

FPGA Flow

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

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

FPGA support in the Intel® [FPGA Support Package for the Intel® oneAPI DPC++/C++ Compiler](#) oneAPI DPC++/C++ Compiler requires the

For full details about the FPGA flow, refer to the [Intel® oneAPI DPC++/C++ Compiler Handbook for Intel® FPGAs](#)

Tip Learn about programming SYCL* programming for FPGA devices by reviewing the [oneAPI Samples for FPGAs](#) on GitHub.

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

Why is FPGA Compilation Different?

FPGAs differ from CPUs and GPUs in some important ways. A significant difference between compilation for CPU or GPU and compilation for FPGA is that generating a device binary for 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, and is therefore not supported.

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.

The [Intel® oneAPI DPC++/C++ Compiler Handbook for Intel® FPGAs](#) provides full details about the FPGA flow.

Types of SYCL* FPGA Compilation

SYCL supports accelerators in general. The FPGA Support Package for the Intel® oneAPI DPC++/C++ Compiler adds FPGA-specific support to the Intel® oneAPI DPC++/C++ Compiler to assist FPGA code development. This topic highlights different FPGA compilation flows that the FPGA Support Package supports.

For a hands-on lesson in the types of FPGA compilation, review the [FPGA Compile Sample](#) on GitHub.

The following table summarizes the types of FPGA compilation:

Types of FPGA Compilation

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

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

Tip To compile for FPGA emulation or FPGA simulation, and to generate the FPGA optimization report, the [FPGA Support Package for the Intel® oneAPI DPC++/C++ Compiler](#) is required. To compile on FPGA hardware, you must also install the [Quartus® Prime Software](#). Targeting a board also requires that you install the BSP for the board.

For more information, refer to the [Intel® oneAPI Toolkits Installation Guide](#) and [Intel® FPGA development flow](#) webpage.

Generating an RTL IP core requires only the [FPGA Support Package for the Intel® oneAPI DPC++/C++ Compiler](#). Simulating or integrating that IP core into your hardware design requires the Quartus® Prime Pro Edition Software.

FPGA Emulator

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

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

- **Performance is not representative.**

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

- **Undefined behavior may differ.**

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

For detailed information about emulation your kernels, refer to [Emulate Your Kernel](#) in the *Intel oneAPI FPGA Handbook*.

FPGA Optimization Report

The FPGA Optimization Report is generated in the following compilation stages:

Stages	Description	Optimization Report Information
FPGA Optimization Report image (Compilation takes minutes to complete)	<p>The SYCL device code is optimized and converted into an FPGA design specified in the Verilog Register-Transfer Level (RTL) (a low-level, native entry language for FPGAs). The intermediate compilation result is the FPGA early device image that is not an executable.</p> <p>The optimization report generated at this stage is static in nature.</p>	<p>Contains important information about how the compiler has transformed your SYCL device code into an FPGA design. The report includes the following information:</p> <ul style="list-style-type: none"> • Visualizations of structures generated on the FPGA. • Performance and expected performance bottleneck. • Estimated resource utilization.

Stages	Description	Optimization Report Information
FPGA hardware image (Compilation takes hours to complete)	The Verilog RTL specifying the design's circuit topology is mapped onto the FPGA's primitive hardware resources by the Intel® Quartus® Prime pro Edition Software . The result is an FPGA hardware binary (also referred to as a bitstream).	<p>For information about the FPGA optimization report, refer to the Review the FPGA Optimization Report in the <i>Intel® oneAPI FPGA Handbook</i>.</p> <p>Contains precise information about resource utilization and f_{MAX} numbers. For detailed information about how to analyze reports, refer to Analyze your Design in the <i>Intel® oneAPI FPGA Handbook</i>.</p> <p>For information about the FPGA hardware image, refer to the Intel oneAPI DPC++/C++ Compiler Handbook for FPGAs.</p>

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

FPGA Simulator

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

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

You can use the simulation flow in conjunction with profiling to collect additional information about your design. For more information about profiling, refer to [Intel® FPGA Dynamic Profiler for DPC++](#) in the *Intel® oneAPI FPGA Handbook*.

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

For more information [in the Intel® oneAPI FPGA Handbook](#).
 about the simulation flow,
 refer to [Evaluate Your Kernel Through Simulation](#)

FPGA Hardware

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

- Altera® FPGA device family
- Specific Altera® FPGA device part number
- Custom board with a supported BSP

For more information about the targets, refer to the [Intel® oneAPI DPC++/C++ Compiler System Requirements](#). For more information about using Intel® PAC or custom boards, refer to [FPGA Boards and Board Support Packages \(BSPs\)](#) in the *Intel® oneAPI FPGA Handbook* and the [Intel® oneAPI Toolkits Installation Guide for Linux* OS Installation Guide](#).

API-Based Programming

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

oneAPI Toolkit Libraries

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

NOTE Explore the complete list of oneAPI code samples in the **oneAPI Samples Catalog** <<https://oneapi-src.github.io/oneAPI-samples/>> (GitHub*). These samples were designed to help you develop, offload, and optimize multiarchitecture applications targeting CPUs, GPUs, and FPGAs.

Intel® oneAPI DPC++ Library (oneDPL)

The Intel® oneAPI DPC++ Library (oneDPL) aims to work with the Intel® oneAPI DPC++/C++ Compiler to provide high-productivity APIs to developers, which can minimize SYCL* programming efforts across devices for high performance parallel applications.

oneDPL consists of the following components:

- Parallel STL:
 - Parallel STL Usage Instructions
 - Macros
- An additional set of library classes and functions (referred to throughout this document as **Extension API**):
 - Parallel Algorithms
 - Iterators
 - Function Object Classes
 - Range-Based API
- Tested Standard C++ APIs
- Random Number Generator

oneDPL Library Usage

Install the [Intel® oneAPI Base Toolkit](#) to use oneDPL.

To use Parallel STL or the Extension API, include the corresponding header files in your source code. All oneDPL header files are in the `oneapi/dpl` directory. Use `#include <oneapi/dpl/...>` to include them. oneDPL uses the namespace `oneapi::dpl` for the most of its classes and functions.

To use tested C++ standard APIs, you need to include the corresponding C++ standard header files and use the `std` namespace.

oneDPL Code Samples

oneDPL sample code is available from the oneAPI GitHub repository <https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneDPL>. Each sample includes a readme with build instructions.

Intel® oneAPI Math Kernel Library (oneMKL)

The Intel® oneAPI Math Kernel Library (oneMKL) is a computing math library of highly optimized and extensively parallelized routines for applications that require maximum performance. oneMKL contains the high-performance optimizations from the full Intel® Math Kernel Library for CPU architectures (with C/Fortran programming language interfaces) and adds to them a set of SYCL* interfaces for achieving performance on various CPU architectures and Intel Graphics Technology for certain key functionalities. oneMKL provides BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

You can use OpenMP* offload to run standard oneMKL computations on Intel GPUs. Refer to [OpenMP* offload for C interfaces](#) and [OpenMP* offload for Fortran interfaces](#) for more information.

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

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

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Math Kernel Library [website](#). If you plan to use oneMKL as part of the [oneAPI Base Toolkit](#), consider that [priority support](#) is available as a paid option. For Intel community-support, visit the [oneMKL forum](#). For the open-source oneMath project, visit the [oneMath GitHub* page](#).

The table below describes the difference in these sites:

oneAPI Specification for oneMath	Defines the SYCL interfaces for performance math library functions. The oneMath specification can evolve faster and more frequently than implementations of the specification.
oneAPI Math Library (oneMath)	An open source implementation of the specification. The project goal is to demonstrate how the SYCL interfaces documented in the oneMath specification can be implemented for any math library and work for any target hardware. While the implementation provided here may not yet be the full implementation of the specification, the goal is to build it out over time. We encourage the community to contribute to this project and help to extend support to multiple hardware targets and other math libraries.
Intel(R) oneAPI Math Kernel Library (oneMKL) Product	The Intel product has SYCL interfaces that are very similar to the oneMath specification as well as similar functionality with C and Fortran interfaces, and is provided as part of Intel® oneAPI Base Toolkit. It is highly optimized for Intel CPU and Intel GPU hardware and is used for the Intel backends of the oneMath open-source project.

oneMKL Usage

When using the SYCL* interfaces, there are a few changes to consider:

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

oneMKL Two-Level Namespaces

Namespace	Description
<code>oneapi::mkl</code>	Contains common elements between various domains in oneMKL
<code>oneapi::mkl::blas</code>	Contains dense vector-vector, matrix-vector, and matrix-matrix low level operations
<code>oneapi::mkl::lapack</code>	Contains higher-level dense matrix operations like matrix factorizations and eigensolvers
<code>oneapi::mkl::rng</code>	Contains random number generators for various probability density functions
<code>oneapi::mkl::stats</code>	Contains basic statistical estimates for single and double precision multi-dimensional datasets
<code>oneapi::mkl::vm</code>	Contains vector math routines
<code>oneapi::mkl::dft</code>	Contains fast fourier transform operations
<code>oneapi::mkl::sparse</code>	Contains sparse matrix operations like sparse matrix-vector multiplication and sparse triangular solver

oneMKL Code Sample

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

NOTE The following code example requires additional code to compile and run, as indicated by the inline comments.

```
// Standard SYCL header
#include <CL/sycl.hpp>
// STL classes
#include <exception>
#include <iostream>
// Declarations for Intel oneAPI Math Kernel Library SYCL/DPC++ APIs
#include "oneapi/mkl.hpp"
int main(int argc, char *argv[]) {
    //
    // User obtains data here for A, B, C matrices, along with setting m, n, k, ldA, ldB, ldC.
    //
    // For this example, A, B and C should be initially stored in a std::vector,
    // or a similar container having data() and size() member functions.
    //

    // Create GPU device
    sycl::device my_device;
    try {
        my_device = sycl::device(sycl::gpu_selector());
    }
    catch (...) {
        std::cout << "Warning: GPU device not found! Using default device instead." << std::endl;
    }
    // Create asynchronous exceptions handler to be attached to queue.
    // Not required; can provide helpful information in case the system isn't correctly
    configured.
    auto my_exception_handler = [](sycl::exception_list exceptions) {
        for (std::exception_ptr const& e : exceptions) {
            try {
                std::rethrow_exception(e);
            }
            catch (sycl::exception const& e) {
                std::cout << "Caught asynchronous SYCL exception:\n"
                    << e.what() << std::endl;
            }
            catch (std::exception const& e) {
                std::cout << "Caught asynchronous STL exception:\n"
                    << e.what() << std::endl;
            }
        }
    };
    // create execution queue on my gpu device with exception handler attached
    sycl::queue my_queue(my_device, my_exception_handler);
    // create sycl buffers of matrix data for offloading between device and host
    sycl::buffer<double, 1> A_buffer(A.data(), A.size());
    sycl::buffer<double, 1> B_buffer(B.data(), B.size());
    sycl::buffer<double, 1> C_buffer(C.data(), C.size());
    // add oneapi::mkl::blas::gemm to execution queue and catch any synchronous exceptions
    try {
        using oneapi::mkl::blas::gemm;
```

```

        using oneapi::mkl::transpose;
        gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha, A_buffer, ldA,
B_buffer,
        ldB, beta, C_buffer, ldC);
    }
    catch (sycl::exception const& e) {
        std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    catch (std::exception const& e) {
        std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
            << e.what() << std::endl;
    }
    // ensure any asynchronous exceptions caught are handled before proceeding
    my_queue.wait_and_throw();
    //
    // post process results
    //
    // Access data from C buffer and print out part of C matrix
    auto C_accessor = C_buffer.template get_access<sycl::access::mode::read>();
    std::cout << "\t" << C << " = [ " << C_accessor[0] << ", "
        << C_accessor[1] << ", ... ]\n";
    std::cout << "\t    [ " << C_accessor[1 * ldC + 0] << ", "
        << C_accessor[1 * ldC + 1] << ", ... ]\n";
    std::cout << "\t    [ " << "... ]\n";
    std::cout << std::endl;

    return 0;
}

```

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

$$C = \alpha * A * B + \beta * C$$

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

```

// Standard SYCL header
#include <CL/sycl.hpp>
// STL classes
#include <exception>
#include <iostream>
// Declarations for Intel oneAPI Math Kernel Library SYCL/DPC++ APIs
#include "oneapi/mkl.hpp"

```

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

```

// Create GPU device
sycl::device my_device;
try {
    my_device = sycl::device(sycl::gpu_selector());
}
catch (...) {
    std::cout << "Warning: GPU device not found! Using default device instead." << std::endl;
}

```

```

}
// Create asynchronous exceptions handler to be attached to queue.
// Not required; can provide helpful information in case the system isn't correctly configured.
auto my_exception_handler = [] (sycl::exception_list exceptions) {
    for (std::exception_ptr const& e : exceptions) {
        try {
            std::rethrow_exception(e);
        }
        catch (sycl::exception const& e) {
            std::cout << "Caught asynchronous SYCL exception:\n"
                << e.what() << std::endl;
        }
        catch (std::exception const& e) {
            std::cout << "Caught asynchronous STL exception:\n"
                << e.what() << std::endl;
        }
    }
};

```

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

```

// create execution queue on my gpu device with exception handler attached
sycl::queue my_queue(my_device, my_exception_handler);
// create sycl buffers of matrix data for offloading between device and host
sycl::buffer<double, 1> A_buffer(A.data(), A.size());
sycl::buffer<double, 1> B_buffer(B.data(), B.size());
sycl::buffer<double, 1> C_buffer(C.data(), C.size());
// add oneapi::mkl::blas::gemm to execution queue and catch any synchronous exceptions
try {
    using oneapi::mkl::blas::gemm;
    using oneapi::mkl::transpose;
    gemm(my_queue, transpose::nontrans, transpose::nontrans, m, n, k, alpha, A_buffer, ldA,
B_buffer,
        ldB, beta, C_buffer, ldC);
}
catch (sycl::exception const& e) {
    std::cout << "\t\tCaught synchronous SYCL exception during GEMM:\n"
        << e.what() << std::endl;
}
catch (std::exception const& e) {
    std::cout << "\t\tCaught synchronous STL exception during GEMM:\n"
        << e.what() << std::endl;
}
}

```

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

```

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

```

```
std::cout << std::endl;

return 0;
```

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

Intel® oneAPI Threading Building Blocks (oneTBB)

Intel® oneAPI Threading Building Blocks (oneTBB) is a widely used C++ library for task-based, shared memory parallel programming on the host. The library provides features for parallel programming on CPUs beyond those currently available in SYCL* and ISO C++, including:

- Generic parallel algorithms
- Concurrent containers
- A scalable memory allocator
- Work-stealing task scheduler
- Low-level synchronization primitives

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

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Threading Building Blocks Library [website](#). If you plan to use oneTBB as part of the [oneAPI Base Toolkit](#), consider that [priority support](#) is available as a paid option. For Intel community-support, visit the [oneTBB forum](#). For the community-supported open-source version, visit the [oneTBB GitHub* page](#).

oneTBB Usage

oneTBB can be used with the Intel® oneAPI DPC++/C++ Compiler in the same way as with any other C++ compiler. For more details, see the [oneTBB documentation](#).

Currently, oneTBB does not directly use any accelerators. However, it can be combined with SYCL*, OpenMP* offload, and other oneAPI libraries to build a program that efficiently uses all available hardware resources.

oneTBB Code Sample

Two basic oneTBB code samples are available within the oneAPI GitHub repository <https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneTBB>. Both samples are prepared for CPU and GPU.

- `tbb-async-sycl`: illustrates how computational kernel can be split for execution between CPU and GPU using oneTBB Flow Graph asynchronous node and functional node. The Flow Graph asynchronous node uses SYCL* to implement calculations on GPU while the functional node does CPU part of calculations.
- `tbb-task-sycl`: illustrates how two oneTBB tasks can execute similar computational kernels with one task executing SYCL code and another one the oneTBB code.
- `tbb-resumable-tasks-sycl`: illustrates how a computational kernel can be split for execution between a CPU and GPU using oneTBB resumable task and `parallel_for`. The resumable task uses SYCL to implement calculations on GPU while `parallel_for` does the CPU portion of calculations.

Intel® oneAPI Data Analytics Library (oneDAL)

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

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

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

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

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Data Analytics Library [website](#). If you plan to use oneDAL as part of the [oneAPI Base Toolkit](#), consider that [priority support](#) is available as a paid option. For Intel community-support, visit the [oneDAL forum](#). For the community-supported open-source version, visit the [oneDAL GitHub* page](#).

oneDAL Usage

Information about dependencies needed to build and link your application with oneDAL are available from the [oneDAL System Requirements](#).

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

- extracting SYCL* buffers from numeric tables and pass them to a custom kernel
- creating numeric tables from SYCL buffers

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

oneDAL Code Sample

oneDAL code samples are available from the oneDAL GitHub. The following code sample is a recommended starting point: <https://github.com/oneapi-src/oneDAL/tree/master/examples/oneapi/dpc/source/svm>

Intel® oneAPI Collective Communications Library (oneCCL)

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

oneCCL features include:

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

For the complete list of features, documentation, code samples, and downloads, visit the official Intel® oneAPI Collective Communications Library [website](#). If you plan to use oneCCL as part of the [oneAPI Base Toolkit](#), consider that [premium support](#) is available as a paid option. For the community-supported open-source version, visit the [oneCCL GitHub* page](#).

oneCCL Usage

Refer to the [Intel® oneAPI Collective Communications Library System Requirements](#) for a full list of hardware and software dependencies, such as MPI and Intel® oneAPI DPC++/C++ Compiler.

SYCL*-aware API is an optional feature of oneCCL. There is a choice between CPU and SYCL back ends when creating the oneCCL stream object.

- For CPU backend: Specify `ccl_stream_host` as the first argument.
- For SYCL backend: Specify `ccl_stream_cpu` or `ccl_stream_gpu` depending on the device type.
- For collective operations that operate on the SYCL stream:
 - For C API, oneCCL expects communication buffers to be `sycl::buffer` objects casted to `void*`.
 - For C++ API, oneCCL expects communication buffers to be passed by reference.

Additional usage details are available from <https://oneapi-src.github.io/oneCCL/>.

oneCCL Code Sample

oneCCL code samples are available from the oneAPI GitHub repository <https://github.com/oneapi-src/oneAPI-samples/tree/master/Libraries/oneCCL>.

A Getting Started sample with instructions to build and run the code is available from within the same GitHub repository.

Intel® oneAPI Deep Neural Network Library (oneDNN)

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

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

oneDNN continues to support features currently available with Intel® Deep Neural Network Library (Intel® DNNL), including C and C++ interfaces, OpenMP*, Intel® oneAPI Threading Building Blocks, and OpenCL™ runtimes. oneDNN introduces SYCL*/DPC++ API and runtime support for the oneAPI programming model.

For the complete list of features, documentation, code samples, and downloads, visit the official Intel oneAPI Deep Neural Network Library [website](#). If you plan to use oneDNN as part of the [oneAPI Base Toolkit](#), consider that [premium support](#) is available as a paid option. For the community-supported open-source version, visit the [oneDNN GitHub*](#) page.

Intel® oneAPI Deep Neural Network Library (oneDNN) Usage

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

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

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

Package Availability by Operating System

Configuration	Dependency
cpu_dpccpp_gpu_dpccpp	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 Intel® oneAPI toolkits or third-party tools.

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

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

The interoperability API is provided for two scenarios:

- Construction of oneDNN objects based on existing SYCL objects
- Accessing SYCL objects for existing oneDNN objects

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

oneDNN and SYCL Object Mapping 1

oneDNN Objects	SYCL Objects
Engine	cl::sycl::device and cl::sycl::context
Stream	cl::sycl::queue
Memory	cl::sycl::buffer<uint8_t, 1> or Unified Shared Memory (USM) pointer

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

oneDNN and SYCL Object Mapping 2

oneDNN Object	Constructing from SYCL Object
Engine	<code>dnnl::sycl_interop::make_engine(sycl_dev, sycl_ctx)</code>
Stream	<code>dnnl::sycl_interop::make_stream(engine, sycl_queue)</code>
Memory	USM based: <code>dnnl::memory(memory_desc, engine, usm_ptr)</code> Buffer based: <code>dnnl::sycl_interop::make_memory(memory_desc, engine, sycl_buf)</code>

oneDNN and SYCL Object Mapping 3

oneDNN Object	Extracting SYCL Object
Engine	<code>dnnl::sycl_interop::get_device(engine)</code> <code>dnnl::sycl_interop::get_context(engine)</code>
Stream	<code>dnnl::sycl_interop::get_queue(stream)</code>
Memory	USM pointer: <code>dnnl::memory::get_data_handle()</code> Buffer: <code>dnnl::sycl_interop::get_buffer(memory)</code>

NOTE

- Building applications with oneDNN requires a compiler. The Intel® oneAPI DPC++/C++ Compiler is available as part of the Intel oneAPI Base Toolkit.
- You must include `dnnl_sycl.hpp` to enable the SYCL-interop API.
- Because OpenMP does not rely on the passing of runtime objects, it does not require an interoperability API to work with oneDNN.

Intel® oneAPI Deep Neural Network Library (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.

Other Libraries

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

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

Software Development Process

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

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

Migrating Code to SYCL* and DPC++

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

Migrating from C++ to SYCL*

SYCL is a single-source style programming model based on C++. It builds on features of C++17 and C++20 to offer an open, multivendor, multiarchitecture solution for heterogeneous programming.

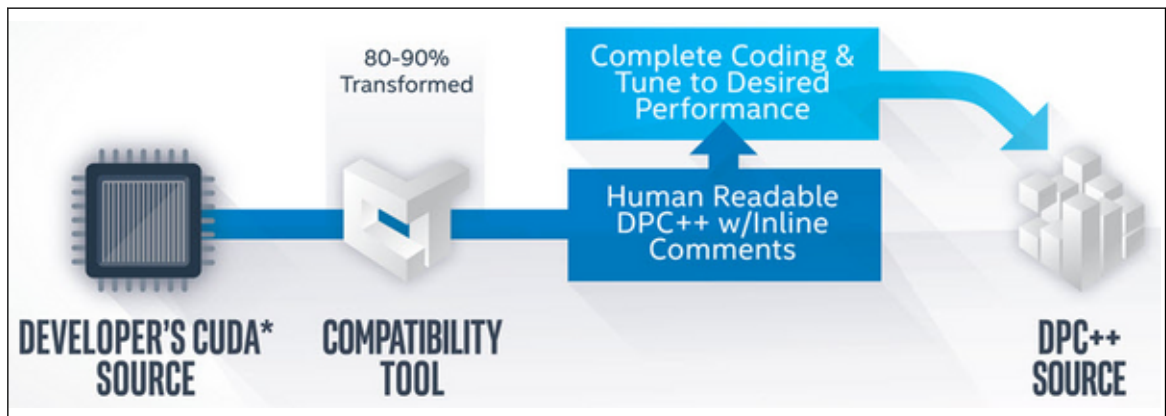
The Intel® oneAPI DPC++ Compiler project is bringing SYCL* to an LLVM C++ compiler, with high performance implementations for multiple vendors and architectures.

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

Migrating from CUDA* to SYCL* for the oneAPI DPC++ Compiler

The Intel® DPC++ Compatibility Tool is part of the Intel® oneAPI Base Toolkit. The goal of this tool is to assist in the migration of an existing program that is written in NVIDIA* CUDA* to a program written in SYCL* and compiled with the Intel® oneAPI DPC++ Compiler. This tool generates SYCL code as much as it can. However, it will not migrate all code and manual changes may be required. The tool provides help with IDE plugins, a [user guide](#), and embedded comments in the code to complete the migration to be compiled with DPC++. After completing any manual changes, use a oneAPI DPC++ Compiler to create executables.

Migrating CUDA to SYCL Using the Intel® DPC++ Compatibility Tool



- Additional details, including examples of migrated code and download instructions for the tool, are available from the [Intel® DPC++ Compatibility Tool website](#).
- Full usage information is available from the [Intel® DPC++ Compatibility Tool User Guide](#)

Migrating from OpenCL Code to SYCL*

The SYCL runtime for the DPC++ project uses OpenCL and other means to enact the parallelism. SYCL typically requires fewer lines of code to implement kernels and also fewer calls to essential API functions and methods. It enables creation of OpenCL programs by embedding the device source code in line with the host source code.

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

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

Migrating Between CPU, GPU, and FPGA

Programming with SYCL* and using the Intel® oneAPI DPC++ Compiler, a platform consists of a host device connected to zero or more devices, such as CPU, GPU, FPGA, or other kinds of accelerators and processors.

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

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

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

```

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

```

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

```

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

```

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

Composability

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

C/C++ OpenMP* and SYCL* Composability

The oneAPI programming model provides a unified compiler based on LLVM/Clang with support for OpenMP* offload. This allows seamless integration that allows the use of OpenMP constructs to either parallelize host side applications or offload to a target device. The Intel® oneAPI DPC++/C++ Compiler, available with the Intel® oneAPI Base Toolkit, supports OpenMP and SYCL composability with a set of restrictions. A single application can offload execution to available devices using OpenMP target regions or SYCL constructs in different parts of the code, such as different functions or code segments.

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

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

Restrictions

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

- OpenMP directives cannot be used inside SYCL kernels that run in the device. Similarly, SYCL code cannot be used inside the OpenMP target regions. However, it is possible to use SYCL constructs within the OpenMP code that runs on the host CPU.
- OpenMP and SYCL device parts of the program cannot have cross dependencies. For example, a function defined in the SYCL part of the device code cannot be called from the OpenMP code that runs on the device and vice versa. OpenMP and SYCL device parts are linked independently and they form separate binaries that become a part of the resulting fat binary that is generated by the compiler.
- The direct interaction between OpenMP and SYCL runtime libraries are not supported at this time. For example, a device memory object created by OpenMP API is not accessible by SYCL code. That is, using the device memory object created by OpenMP in SYCL code results unspecified execution behavior.

Example

The following code snippet uses SYCL and OpenMP offloading constructs in the same application.

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

float computePi(unsigned N) {
    float Pi;
#pragma omp target map(from : Pi)
#pragma omp parallel for reduction(+ : Pi)
    for (unsigned I = 0; I < N; ++I) {
        float T = (I + 0.5f) / N;
        Pi += 4.0f / (1.0 + T * T);
    }
    return Pi / N;
}

void iota(float *A, unsigned N) {
    cl::sycl::range<1> R(N);
    cl::sycl::buffer<float, 1> AB(A, R);
    cl::sycl::queue().submit([&](cl::sycl::handler &cgh) {
        auto AA = AB.template get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Iota>(R, [=](cl::sycl::id<1> I) {
            AA[I] = I;
        });
    });
}

int main() {
    std::array<float, 1024u> Vec;
    float Pi;

#pragma omp parallel sections
    {
```

```
#pragma omp section
    iota(Vec.data(), Vec.size());
#pragma omp section
    Pi = computePi(8192u);
}

std::cout << "Vec[512] = " << Vec[512] << std::endl;
std::cout << "Pi = " << Pi << std::endl;
return 0;
}
```

The following command is used to compile the example code: `icpx -fsycl -fiopenmp -fopenmp-targets=spir64 offloadOmp_dpcpp.cpp`

where

- `-fsycl` option enables SYCL
- `-fiopenmp -fopenmp-targets=spir64` option enables OpenMP* offload

The following shows the program output from the example code.

```
./a.out
Vec[512] = 512
Pi = 3.14159
```

NOTE If the code does not contain OpenMP offload, but only normal OpenMP code, use the following command, which omits `-fopenmp-targets`: `icpx -fsycl -fiopenmp omp_dpcpp.cpp`

OpenCL™ Code Interoperability

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

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

Debugging the DPC++ and OpenMP* Offload Process

When writing, debugging, and optimizing code for a host platform, the process of improving your code is simple: deal with language errors when you build, catch and root-cause crashes/incorrect results during execution with a debugger, then identify and fix performance issues using a profiling tool.

Improving code can become considerably more complicated in applications where part of the execution is offloaded to another device using either DPC++ or OpenMP* offload.

- Incorrect use of the DPC++ or OpenMP* offload languages may not be exposed until after just-in-time compilation occurs. These issues can be exposed earlier with ahead-of-time (AOT) compilation.
- Crashes due to logic errors may arise as unexpected behavior on the host, on the offload device, or in the software stack that ties the various computing devices together. To root cause these issues, you need to:
 - Debug what is happening in your code on the host using a standard debugger, such as Intel® Distribution for GDB*.
 - Debug problems on the offload device using a device-specific debugger. Note, however, that the device may have a different architecture, conventions for representing compute threads, or assembly than the host.
 - To debug problems that show up in the intermediate software stack only when kernels and data are being exchanged with the device, you need to monitor the communication between device and host and any errors that are reported during the process.
- Besides the usual performance issues that can occur on the host and offload devices, the patterns by which the host and offload device work together can have a profound impact on application performance. This is another case where you need to monitor the communications between the host and offload device.

This section discusses the various debugging and performance analysis tools and techniques available to you for the entire lifecycle of the offload program.

To troubleshoot your applications that use OpenMP* or the SYCL* API with extensions to offload resources, see the [Troubleshoot Highly Parallel Applications](#) tutorial.

oneAPI Debug Tools for SYCL* and OpenMP* Development

The following tools are available to help with debugging the SYCL* and OpenMP* offload process.

Tools to debug SYCL* and OpenMP* offload process

Tool	When to Use
Environment variables	Environment variables allow you to gather diagnostic information from the OpenMP and SYCL runtimes at program execution with no modifications to your program.
The onetrace tool from Profiling Tools Interfaces for GPU (PTI for GPU)	When using the Intel® oneAPI Level Zero and OpenCL™ backends for SYCL and OpenMP Offload, this tool can be used to debug backend errors and for performance profiling on both the host and device. Learn more: <ul style="list-style-type: none"> • Onetrace tool GitHub • PTI for GPU GitHub • GPU Compute/Media Hotspots Analysis
Intercept Layer for OpenCL™ Applications	When using the OpenCL™ backend for SYCL and OpenMP Offload, this library can be used to debug backend errors and for performance profiling on both the host and device (has wider functionality comparing with onetrace).
Intel® Distribution for GDB*	Used for source-level debugging of the application, typically to inspect logical bugs, on the host and any devices you are using (CPU, GPU, FPGA emulation).
Intel® Inspector	This tool helps to locate and debug memory and threading problems, including those that can cause offloading to fail.

Tool	When to Use
	<hr/> <p>NOTE Intel Inspector is included in the Intel® HPC Toolkit.</p> <hr/>
In-application debugging	<p>In addition to these tools and runtime based approaches, the developer can locate problems using other approaches. For example:</p> <ul style="list-style-type: none"> • Comparing kernel output to expected output • Sending intermediate results back by variables they create for debugging purposes • Printing results from within kernels <hr/> <p>NOTE Both SYCL and OpenMP allow printing to stdout from within an offload region - be sure to note which SIMD lane or thread is providing the output.</p> <hr/>
SYCL Exception Handler	<p>Some DPC++ programming errors are returned as exceptions by the SYCL runtime during program execution. They can help you diagnose errors in your code that are flagged at runtime. For more details and examples, refer to Using SYCL Exceptions. For Samples that demonstrate SYCL Exceptions, refer to: * Guided Matrix Multiplication Exception * Guided Matrix Multiplication Invalid Contexts * Guided Matrix Multiplication Race Condition</p>
Intel® Advisor	<p>Use to ensure Fortran, C, C++, OpenCL™, and SYCL applications realize full performance potential on modern processors.</p>
Intel® VTune™ Profiler	<p>Use to gather performance data either on the native system or on a remote system.</p>
OpenMP* directives	<p>Offload and Optimize OpenMP* Applications with Intel Tools describes how to use OpenMP* directives to add parallelism to your application.</p>

Debug Environment Variables

Both the OpenMP* and SYCL offload runtimes, as well as Level Zero, OpenCL, and the Shader Compiler, provide environment variables that help you understand the communication between the host and offload device. The variables also allow you to discover or control the runtime chosen for offload computations.

OpenMP* Offload Environment Variables

There are several environment variables that you can use to understand how OpenMP Offload works and control which backend it uses.

NOTE OpenMP is not supported for FPGA devices.

OpenMP* Offload Environment Variables

Environment Variable	Description
LIBOMPTARGET_DEBUG=<Num>	<p>Controls whether or not debugging information will be displayed. See details in Runtimes This environment variable enables debug output from the OpenMP Offload runtime. It reports:</p> <ul style="list-style-type: none"> • The available runtimes detected and used (1,2)

Environment Variable	Description
LIBOMPTARGET_INFO=<Num>	<ul style="list-style-type: none"> • When the chosen runtime is started and stopped (1,2) • Details on the offload device used (1,2) • Support libraries loaded (1,2) • Size and address of all memory allocations and deallocations (1,2) • Information on every data copy to and from the device, or device mapping in the case of unified shared memory (1,2) • When each kernel is launched and details on the launch (arguments, SIMD width, group information, etc.) (1,2) • Which Level Zero/OpenCL API functions are invoked (function name, arguments/parameters) (2) <p>Values:</p> <p><Num>=0 : Disabled</p> <p><Num>=1 : Displays basic debug information from the plugin actions such as device detection, kernel compilation, memory copy operations, kernel invocations, and other plugin-dependent actions.</p> <p><Num>=2 : Additionally displays which GPU runtime API functions are invoked with which arguments/parameters.</p> <p>Default: 0</p> <p>This variable controls whether basic offloading information will be displayed from the offload runtime. Allows the user to request different types of runtime information from libomptarget. See details in Runtimes</p>
LIBOMPTARGET_PLUGIN_PROFILE=<Enable>[,<Unit>]	<ul style="list-style-type: none"> • Prints all data arguments upon entering an OpenMP device kernel (1) • Indicates when a mapped address already exists in the device mapping table (2) • Dumps the contents of the device pointer map if target offloading fails (4) • Indicates when an entry is changed in the device mapping table (8) • Indicates when data is copied to and from the device (32) <p>Values: (0, 1, 2, 4, 8, 32)</p> <p>Default: 0</p> <p>This variable enables the display of performance data for offloaded OpenMP code. It displays:</p> <ul style="list-style-type: none"> • Total data transfer times (read and write) • Data allocation times

Environment Variable	Description
LIBOMPTARGET_PLUGIN=<Name>	<ul style="list-style-type: none"> Module build times (just-in-time compile) The execution time of each kernel. <p>Values:</p> <ul style="list-style-type: none"> F - disabled T - enabled with timings in milliseconds T,usec - enabled with timings in microseconds <p>Default: F</p> <p>Example: export LIBOMPTARGET_PLUGIN_PROFILE=T,usec</p> <pre><Enable> := 1 T <Unit> := usec unit_usec</pre> <p>Enables basic plugin profiling and displays the result when program finishes. Microsecond is the default unit if <Unit> is not specified.</p> <p>This environment variable allows you to choose the backend used for OpenMP offload execution.</p> <hr/> <p>NOTE The Level Zero backend is only supported for GPU devices.</p> <hr/> <pre><Name> := LEVEL0 OPENCL CUDA X86_64 NIOS2 level0 opencl cuda x86_64 nios2 </pre> <p>Designates offload plugin name to use. Offload runtime does not try to load other RTLs if this option is used.</p> <p>Values:</p> <ul style="list-style-type: none"> LEVEL0 or LEVEL_ZERO - uses the Level Zero backend OPENCL - uses the OpenCL™ backend <p>Default:</p> <ul style="list-style-type: none"> For GPU offload devices: LEVEL0 For CPU or FPGA offload devices: OPENCL
LIBOMPTARGET_PROFILE=<FileName>	<p>Allows libomptarget to generate time profile output similar to Clang's -ftime-trace option. See details in Runtimes</p>
LIBOMPTARGET_DEVICES=<DeviceKind>	<pre><DeviceKind> := DEVICE SUBDEVICE SUBSUBDEVICE ALL device subdevice subsubdevice all</pre> <p>Controls how subdevices are exposed to users.</p>

Environment Variable	Description
	<p>DEVICE/device: Only top-level devices are reported as OpenMP devices, and <code>subdevice</code> clause is supported.</p> <p>SUBDEVICE/subdevice: Only 1st-level subdevices are reported as OpenMP devices, and <code>subdevice</code> clause is ignored.</p> <p>SUBSUBDEVICE/subsubdevice: Only 2nd-level subdevices are reported as OpenMP devices, and <code>subdevice</code> clause is ignored. On Intel GPU using Level Zero backend, limiting the <code>subsubdevice</code> to a single compute slice within a stack also requires setting additional GPU compute runtime environment variable <code>CFESingleSliceDispatchCCSMODE=1</code>.</p> <p>ALL/all: All top-level devices and their subdevices are reported as OpenMP devices, and <code>subdevice</code> clause is ignored. This is not supported on Intel GPU and is being deprecated.</p> <p>Default: Equivalent to <code><DeviceKind>=device</code></p>
<p>LIBOMPTARGET_LEVEL0_MEMORY_POOL=<Option> ></p>	<pre data-bbox="824 978 1451 1365"> <Option> := 0 <PoolInfoList> <PoolInfoList> := <PoolInfo>[,<PoolInfoList>] <PoolInfo> := <MemType>[,<AllocMax>[,<Capacity>[,<PoolSize>]]] <MemType> := all device host shared <AllocMax> := positive integer or empty, max allocation size in MB <Capacity> := positive integer or empty, number of allocations from a single block <PoolSize> := positive integer or empty, max pool size in MB </pre> <p>Controls how reusable memory pool is configured. Pool is a list of memory blocks that can serve at least <code><Capacity></code> allocations of up to <code><AllocMax></code> size from a single block, with total size not exceeding <code><PoolSize></code>.</p> <p>Default: Equivalent to <code><Option>=device,1,4,256,host,1,4,256,shared,8,4,256</code></p>
<p>LIBOMPTARGET_LEVEL0_STAGING_BUFFER_SIZE= <Num></p>	<p>Sets the staging buffer size to <code><Num></code> KB. Staging buffer is used in copy operations between host and device as a temporary storage for two-step copy operation. The buffer is only used for discrete devices.</p> <p>Default: 16</p>

Environment Variable	Description
LIBOMPTARGET_LEVEL_ZERO_COMMAND_BATCH= <Value>	<pre data-bbox="826 268 1438 1073"> <Value> := <Type>[,<Count>] <Type> := none NONE copy COPY compute COMPUTE <Count> := maximum number of commands to batch Enables command batching for a target region. ``<Type>=none NONE``: Disables command batching. ``<Type>=copy COPY``: Enables command batching for a target region for data transfer. ``<Type>=compute COMPUTE``: Enables command batching for a target region for data transfer and compute, disabling use of copy engine. If ``<Type>`` is either ``copy`` or ``compute`` (enabled) and ``<Count>`` is not specified, batching is performed for all eligible commands for the target region. **Default**:: ``<Type>=none`` (Disabled) </pre>
LIBOMPTARGET_LEVEL_ZERO_USE_IMMEDIATE_CO MMAND_LIST=<Value>	<pre data-bbox="826 1108 1455 1255"> <True> := 1 T t <False>:= 0 F f <Bool>:= <True> <False> <Value> := <Bool> compute COMPUTE copy COPY all ALL </pre> <p data-bbox="826 1276 1455 1539"> compute: Enables immediate command list for kernel submission copy: Enables immediate command list for memory copy operations all: Enables immediate command list for kernel submission and memory copy operations <True>: Equivalent to compute<False>: Immediate command list is disabled. </p> <p data-bbox="826 1560 992 1587">Default: "all"</p>
OMP_TARGET_OFFLOAD=MANDATORY	<p data-bbox="826 1619 1430 1707">This is defined by the OpenMP Standard : https://www.openmp.org/spec-html/5.1/openmpse74.html#x340-5150006.17</p>
ONEAPI_DEVICE_SELECTOR	<p data-bbox="826 1738 1455 1923">This device selection environment variable can be used to limit the choice of devices available when the OpenMP application is run. Useful for limiting devices to a certain type (like GPUs or accelerators) or backends (like Level Zero or OpenCL). The ONEAPI_DEVICE_SELECTOR syntax is shared with</p>

Environment Variable	Description
	OpenMP and also allows devices to be chosen. See oneAPI DPC++ Compiler documentation for a full description. See oneAPI DPC++ Compiler documentation for a full description.

SYCL* and DPC++ Environment Variables

The oneAPI DPC++ Compiler supports all standard SYCL environment variables. The full list is available from [GitHub](#). Of interest for debugging are the following SYCL environment variables, plus an additional Level Zero environment variable.

SYCL* and DPC++ Environment Variables

Environment Variable	Description
ONEAPI_DEVICE_SELECTOR	<p>This complex environment variable allows you to limit the runtimes, compute device types, and compute device IDs used by the runtime to a subset of all available combinations.</p> <p>The compute device IDs correspond to those returned by the SYCL API, <code>clinfo</code>, or <code>sycl-ls</code> (with the numbering starting at 0) and have no relation to whether the device with that ID is of a certain type or supports a specific runtime. Using a programmatic special selector (like <code>gpu_selector</code>) to request a device filtered out by <code>ONEAPI_DEVICE_SELECTOR</code> will cause an exception to be thrown.</p> <p>Refer to the Environment Variables descriptions in GitHub for additional details: https://github.com/intel/llvm/blob/sycl/sycl/doc/EnvironmentVariables.md</p> <p>Example values include:</p> <ul style="list-style-type: none"> • <code>opencl:cpu</code> - use only the OpenCL™ runtime on all available CPU devices • <code>opencl:gpu</code> - use only the OpenCL runtime on all available GPU devices • <code>opencl:gpu:2</code> - use only the OpenCL runtime on only the third device, which also has to be a GPU • <code>level_zero:gpu:1</code> - use only the Level Zero runtime on only the second device, which also has to be a GPU • <code>opencl:cpu,level_zero</code> - use only the OpenCL runtime on the CPU device, or the Level Zero runtime on any supported compute device <p>Default: use all available runtimes and devices</p>
ONEAPI_DEVICE_SELECTOR	<p>This device selection environment variable can be used to limit the choice of devices available when the SYCL-using application is run. Useful for limiting</p>

Environment Variable	Description
SYCL_UR_TRACE	<p>devices to a certain type (like GPUs or accelerators) or backends (like Level Zero or OpenCL). This device selection mechanism is replacing ONEAPI_DEVICE_SELECTOR . The ONEAPI_DEVICE_SELECTOR syntax is shared with OpenMP and also allows devices to be chosen. Refer to oneAPI DPC++ Compiler documentation for a full description: https://intel.github.io/llvm/EnvironmentVariables.html</p> <p>This environment variable enables debug output from the runtime.</p> <p>Values:</p> <ul style="list-style-type: none"> • 1 - report SYCL plugins and devices discovered and used • 2 - report SYCL API calls made, including arguments and result values • -1 - provides all available tracing <p>Default: disabled</p>
ZE_DEBUG	<p>This environment variable enables debug output from the Level Zero backend when used with the runtime. It reports:</p> <ul style="list-style-type: none"> • Level Zero APIs called • Level Zero event information <p>Value: variable defined with any value - enabled</p> <p>Default: disabled</p>

Environment Variables that Produce Diagnostic Information for Support

The Level Zero backend provides a few environment variables that can be used to control behavior and aid in diagnosis.

- [Level Zero Specification Core Programming Guide](#)
- [Level Zero Specification Tool Programming Guide](#)

An additional source of debug information comes from the Intel® Graphics Compiler, which is called by the Level Zero or OpenCL backends (used by both the OpenMP Offload and SYCL/DPC++ Runtimes) at runtime or during Ahead-of-Time (AOT) compilation. Intel® Graphics Compiler creates the appropriate executable code for the target offload device. The full list of these environment variables can be found at https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md. The two that are most often needed to debug performance issues are:

- `IGC_ShaderDumpEnable=1` (default=0) causes all LLVM, assembly, and ISA code generated by the Intel® Graphics Compiler to be written to `/tmp/IntelIGC/<application_name>`
- `IGC_DumpToCurrentDir=1` (default=0) writes all the files created by `IGC_ShaderDumpEnable` to your current directory instead of `/tmp/IntelIGC/<application_name>`. Since this is potentially a lot of files, it is recommended to create a temporary directory just for the purpose of holding these files.

If you have a performance issue with your OpenMP offload or SYCL offload application that arises between different versions of Intel® oneAPI, when using different compiler options, when using the debugger, and so on, then you may be asked to enable `IGC_ShaderDumpEnable` and provide the resulting files. For more information on compatibility, see [oneAPI Library Compatibility](#).

Offload Intercept Tools

In addition to debuggers and diagnostics built into the offload software itself, it can be quite useful to monitor offload API calls and the data sent through the offload pipeline. For Level Zero, if your application is run as an argument to the `onetrace` and `ze_tracer` tools, they will intercept and report on various aspects of Level Zero made by your application. For OpenCL™, you can add a library to `LD_LIBRARY_PATH` that will intercept and report on all OpenCL calls, and then use environment variables to control what diagnostic information to report to a file. You can also use `onetrace` or `cl_tracer` to report on various aspects of OpenCL API calls made by your application. Once again, your application is run as an argument to the `onetrace` or `cl_tracer` tool.

Intercept Layer for OpenCL™ Applications

This library collects debugging and performance data when OpenCL is used as the backend to your SYCL or OpenMP offload program. When OpenCL is used as the backend to your SYCL or OpenMP offload program, this tool can help you detect buffer overwrites, memory leaks, mismatched pointers, and can provide more detailed information about runtime error messages (allowing you to diagnose these issues when either CPU, FPGA, or GPU devices are used for computation). Note that you will get nothing useful if you use `ze_tracer` on a program that uses the OpenCL backend, or the Intercept Layer for OpenCL™ Applications library and `cl_tracer` on a program that uses the Level Zero backend.

Additional resources:

- Extensive information on building and using the Intercept Layer for OpenCL Applications is available from <https://github.com/intel/openccl-intercept-layer>.

NOTE For best results, run `cmake` with the following flags: `-DENABLE_CLIPROF=TRUE -DENABLE_CLILOADER=TRUE`

- Information about a similar tool (CLIntercept) is available from <https://github.com/gmeeker/clintercept> and <https://sourceforge.net/p/clintercept/wiki/Home/>.
- Information on the controls for the Intercept Layer for OpenCL™ Applications can be found at <https://github.com/intel/openccl-intercept-layer/blob/master/docs/controls.md>.
- Information about optimizing for GPUs is available from the [Intel oneAPI GPU Optimization Guide](#).

Profiling Tools Interfaces for GPU (`onetrace`, `cl_tracer`, and `ze_trace`)

Like the Intercept Layer for OpenCL™ Applications, these tools collect debugging and performance data from applications that use the OpenCL and Level Zero offload backends for offload via OpenMP* or SYCL. Note that Level Zero can only be used as the backend for computations that happen on the GPU (there is no Level Zero backend for the CPU or FPGA at this time). The `onetrace` tool is part of the Profiling Tools Interfaces for GPU (PTI for GPU) project, found at <https://github.com/intel/pti-gpu>. This project also contains the `ze_tracer` and `cl_tracer` tools, which trace just activity from the Level Zero or OpenCL offload backends respectively. The `ze_tracer` and `cl_tracer` tools will produce no output if they are used with the application using the other backend, while `onetrace` will provide output no matter which offload backend you use.

The `onetrace` tool is distributed as source. Instructions for how to build the tool are available from <https://github.com/intel/pti-gpu/tree/master/tools/onetrace>. The tool provides the following features:

- Call logging: This mode allows you to trace all standard Level Zero (L0) and OpenCL™ API calls along with their arguments and return values annotated with time stamps. Among other things, this can give you supplemental information on any failures that occur when a host program tries to make use of an attached compute device.

- Host and device timing: These provide the duration of all API calls, the duration of each kernel, and application runtime for the entire application.
- Device Timeline mode: Gives time stamps for each device activity. All the time stamps are in the same (CPU) time scale.
- Chrome Call Logging mode: Dumps API calls to JSON format that can be opened in <chrome://tracing> browser tool.

These data can help debug offload failures or performance issues.

Additional resources:

- [Profiling Tools Interfaces for GPU \(PTI for GPU\) GitHub project](#)
- [Onetrace tool GitHub](#)

Intel® Distribution for GDB*

The Intel® Distribution for GDB* is an application debugger that allows you to inspect and modify the program state. With the debugger, both the host part of your application and kernels that are offloaded to a device can be debugged seamlessly in the same debug session. The debugger supports the CPU, GPU, and FPGA-emulation devices. Major features of the tool include:

- Automatically attaching to the GPU device to listen to debug events
- Automatically detecting JIT-compiled, or dynamically loaded, kernel code for debugging
- Defining breakpoints (both inside and outside of a kernel) to halt the execution of the program
- Listing the threads; switching the current thread context
- Listing active SIMD lanes; switching the current SIMD lane context per thread
- Evaluating and printing the values of expressions in multiple thread and SIMD lane contexts
- Inspecting and changing register values
- Disassembling the machine instructions
- Displaying and navigating the function call-stack
- Source- and instruction-level stepping
- Non-stop and all-stop debug mode
- Recording the execution using Intel Processor Trace (CPU only)

For more information and links to full documentation for Intel Distribution for GDB, see *Get Started with Intel® Distribution for GDB onLinux* Host|Windows* Host*.

Intel® Inspector for Offload

Intel® Inspector is a dynamic memory and threading error checking tool for users developing serial and multithreaded applications. It can be used to verify correctness of the native part of the application as well as dynamically generated offload code.

Unlike the tools and techniques above, Intel Inspector cannot be used to catch errors in offload code that is communicating with a GPU or an FPGA. Instead, Intel Inspector requires that the SYCL or OpenMP runtime needs to be configured to execute kernels on CPU target. In general, it requires definition of the following environment variables prior to an analysis run.

- To configure a SYCL application to run kernels on a CPU device

```
export ONEAPI_DEVICE_SELECTOR=opencl:cpu
```

- To configure an OpenMP application to run kernels on a CPU device

```
export OMP_TARGET_OFFLOAD=MANDATORY
export LIBOMPTARGET_DEVICETYPE=cpu
```

- To enable code analysis and tracing in JIT compilers or runtimes

```
export CL_CONFIG_USE_VTUNE=True
export CL_CONFIG_USE_VECTORIZER=false
```

Use one of the following commands to start analysis from the command line. You can also start from the Intel Inspector graphical user interface.

- Memory: `inspxe-cl -c mi3 -- <app> [app_args]`
- Threading: `inspxe-cl -c ti3 -- <app> [app_args]`

View the analysis result using the following command: `inspxe-cl -report=problems -report-all`

If your SYCL or OpenMP Offload program passes bad pointers to the OpenCL™ backend, or passes the wrong pointer to the backend from the wrong thread, Intel Inspector should flag the issue. This may make the problem easier to find than trying to locate it using the intercept layers or the debugger.

Additional details are available from the *Intel Inspector User Guide for Linux* OS | Windows* OS*.

Trace the Offload Process

When a program that offloads computation to a GPU is started, there are a lot of moving parts involved in program execution. Machine-independent code needs to be compiled to machine-dependent code, data and binaries need to be copied to the device, results returned, etc. This section will discuss how to trace all this activity using the tools described in the [oneAPI Debug Tools](#) section.

Kernel Setup Time

Before offload code can run on the device, the machine-independent version of the kernel needs to be compiled for the target device, and the resulting code needs to be copied to the device. This can complicate/skew benchmarking if this kernel setup time is not considered. Just-in-time compilation can also introduce a noticeable delay when debugging an offload application.

If you have an OpenMP* offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the amount of time required to build the offload code “ModuleBuild”, which you can compare to the overall execution time of your program.

Kernel setup time is more difficult to determine if you have a SYCL* offload program.

- If Level Zero or OpenCL™ is your backend, you can derive kernel setup time from the Device Timing and Device Timeline returned by `onetrace` or `ze_tracer`.
- If OpenCL™ is your backend, you may also be able to derive the information by setting the `BuildLogging`, `KernelInfoLogging`, `CallLogging`, `CallLoggingElapsedTime`, `KernelInfoLogging`, `HostPerformanceTiming`, `HostPerformanceTimeLogging`, `ChromeCallLogging`, or `CallLoggingElapsedTime` flags when using the Intercept Layer for OpenCL™ Applications to get similar information. You can also derive kernel setup time from the Device Timing and Device Time-line returned by `onetrace` or `cl_tracer`.

You can also use these tools to supplement the information returned by `LIBOMPTARGET_PLUGIN_PROFILE=T`.

For details on how Intel® VTune™ Profiler can analyze kernel setup time, see [Enable Linux* Kernel Analysis](#)

Monitoring Buffer Creation, Sizes, and Copies

Understanding when buffers are created, how many buffers are created, and whether they are reused or constantly created and destroyed can be key to optimizing the performance of your offload application. This may not always be obvious when using a high-level programming language like OpenMP or SYCL, which can hide a lot of the buffer management from the user.

At a high level, you can track buffer-related activities using the `LIBOMPTARGET_DEBUG` and `SYCL_UR_TRACE` environment variables when running your program. `LIBOMPTARGET_DEBUG` gives you more information than `SYCL_UR_TRACE` - it reports the addresses and sizes of the buffers created. By contrast, `SYCL_UR_TRACE` just reports the API calls, with no information you can easily tie to the location or size of individual buffers.

At a lower level, if you are using Level Zero or OpenCL™ as your backend, the Call Logging mode of `onetrace` or `ze_tracer` will give you information on all API calls, including their arguments. This can be useful because, for example, a call for buffer creation (such as `zeMemAllocDevice`) will give you the size of the resulting buffer being passed to and from the device. `onetrace` and `ze_tracer` also allows you to dump all the Level Zero device-side activities (including memory transfers) in Device Timeline mode. For each activity one can get `append` (to command list), `submit` (to queue), `start` and `end` times.

If you are using OpenCL as your backend, setting the `CallLogging`, `CallLoggingElapsedTime`, and `ChromeCallLogging` flags when using the Intercept Layer for OpenCL™ Applications should give you similar information. The Call Logging mode of `onetrace` or `cl_tracer` will give you information on all OpenCL API calls, including their arguments. As was the case above, `onetrace` and `cl_tracer` also allow you to dump all the OpenCL device-side activities (including memory transfers) in Device Timeline mode.

Total Transfer Time

Comparing total data transfer time to kernel execution time can be important for determining whether it is profitable to offload a computation to a connected device.

If you have an OpenMP offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the amount of time required to build ("DataAlloc"), read ("DataRead"), and write data ("DataWrite") to the offload device (although only in aggregate).

Data transfer times can be more difficult to determine if you have a C++ program using SYCL.

- If Level Zero or OpenCL™ is your backend, you can derive total data transfer time from the Device Timing and Device Timeline returned by `onetrace` or `ze_tracer`.
- If OpenCL is your backend, you can use `onetrace` or `cl_tracer`, or alternatively you may also be able to derive the information by setting the `BuildLogging`, `KernelInfoLogging`, `CallLogging`, `CallLoggingElapsedTime`, `KernelInfoLogging`, `HostPerformanceTiming`, `HostPerformanceTimeLogging`, `ChromeCallLogging`, or `CallLoggingElapsedTime` flags when using the Intercept Layer for OpenCL Applications.

For details on how Intel® VTune™ Profiler can analyze transfer setup time, see these sections of the Intel® VTune™ Profiler User Guide: [GPU Offload Analysis](#)[GPU Compute/Media Hotspots](#) [ViewHotspots Report](#)

Kernel Execution Time

If you have an OpenMP offload program, setting `LIBOMPTARGET_PLUGIN_PROFILE=T[,usec]` explicitly reports the total execution time of every offloaded kernel ("Kernel#...").

For programs using SYCL to offload kernels:

- If Level Zero or OpenCL™ is your backend, the Device Timing mode of `onetrace` or `ze_tracer` will give you the device-side execution time for every kernel.

- If OpenCL is your backend, you can use `onetrace` or `cl_tracer`, or alternatively you may be able to derive the information by setting the `CallLoggingElapsedTime`, `DevicePerformanceTiming`, `DevicePerformanceTimeKernelInfoTracking`, `DevicePerformanceTimeLWSTracking`, `DevicePerformanceTimeGWSTracking`, `ChromePerformanceTiming`, `ChromePerformanceTimingInStages` flags when using the Intercept Layer for OpenCL™ Applications.

For details on how Intel® VTune™ Profiler can analyze kernel execution time, see [Accelerators Analysis Group](#)

When Device Kernels Are Called and Threads Are Created

On occasion, offload kernels are created and transferred to the device a long time before they actually start executing (usually only after all data required by the kernel has also been transferred, along with control).

You can set a breakpoint in a device kernel using the Intel® Distribution for GDB* and a compatible GPU. From there, you can query kernel arguments, monitor thread creation and destruction, list the current threads and their current positions in the code (using “info thread”), and so on.

Debug the Offload Process

Run with Different Runtimes or Compute Devices

When an offload program fails to run correctly or produces incorrect results, a relatively quick sanity check is to run the application on a different runtime (OpenCL™ vs. Level Zero) or compute device (CPU vs. GPU) using `LIBOMPTARGET_PLUGIN` and `OMP_TARGET_OFFLOAD` for OpenMP* applications, and `ONEAPI_DEVICE_SELECTOR` for SYCL* applications. Errors that reproduce across runtimes mostly eliminate the runtime as being a problem. Errors that reproduce on all available devices mostly eliminates bad hardware as the problem.

Debug CPU Execution

Offload code has two options for CPU execution: either a “host” implementation, or the CPU version of OpenCL. A “host” implementation is a truly native implementation of the offloaded code, meaning it can be debugged like any of the non-offloaded code. The CPU version of OpenCL, while it goes through the OpenCL runtime and code generation process, eventually ends up as normal parallel code running under a TBB runtime. Again, this provides a familiar debugging environment with familiar assembly and parallelism mechanisms. Pointers have meaning through the entire stack, and data can be directly inspected. There are also no memory limits beyond the usual limits for any operating system process.

Finding and fixing errors in CPU offload execution may solve errors seen in GPU offload execution with less pain, and without requiring use of a system with an attached GPU or other accelerator.

For OpenMP applications, to get a “host” implementation, remove the “target” or “device” constructs, replacing them with normal host OpenMP code. If `LIBOMPTARGET_PLUGIN=OPENCL` and offload to the GPU is disabled, then the offloaded code runs under the OpenMP runtime with TBB providing parallelism.

For SYCL applications, with `ONEAPI_DEVICE_SELECTOR=host` the “host” device is actually single-threaded, which may help you determine if threading issues, such as data races and deadlocks, are the source of execution errors. Setting `ONEAPI_DEVICE_SELECTOR=opencl:cpu` uses the CPU OpenCL runtime, which also uses TBB for parallelism.

Debug GPU Execution Using Intel® Distribution for GDB* on Compatible GPUs

Intel® Distribution for GDB* is extensively documented in *Get Started with Intel® Distribution for GDB on Linux* Host|Windows* Host*. Useful commands are briefly described in the *Intel® Distribution for GDB Reference Sheet*. However, since debugging applications with GDB* on a GPU differs slightly from the process on a host (some commands are used differently and you might see some unfamiliar output), some of those differences are summarized here.

The [Debugging with Intel® Distribution for GDB on Linux OS Host Tutorial](#) shows a sample debug session where we start a debug session of a SYCL program, define a breakpoint inside the kernel, run the program to offload to the GPU, print the value of a local variable, switch to the SIMD lane 5 of the current thread, and print the variable again.

As in normal GDB*, for a command <CMD>, use the `help <CMD>` command of GDB to read the information text for <CMD>. For example:

```
(gdb) help info threads
Display currently known threads.
Usage: info threads [OPTION]... [ID]...
If ID is given, it is a space-separated list of IDs of threads to display.
Otherwise, all threads are displayed.

Options:
-gid
  Show global thread IDs.
```

Inferiors, Threads, and SIMD Lanes Referencing in GDB*

The threads of the application can be listed using the debugger. The printed information includes the thread ids and the locations that the threads are currently stopped at. For the GPU threads, the debugger also prints the active SIMD lanes.

In the example referenced above, you may see some unfamiliar formatting used when threads are displayed via the GDB “info threads” command:

Id	Target Id	Frame
1.1	Thread <id omitted>	<frame omitted>
1.2	Thread <id omitted>	<frame omitted>
* 2.1:1	Thread 1073741824	<frame> at array-transform.cpp:61
2.1:[3 5 7]	Thread 1073741824	<frame> at array-transform.cpp:61
2.2:[1 3 5 7]	Thread 1073741888	<frame> at array-transform.cpp:61
2.3:[1 3 5 7]	Thread 1073742080	<frame> at array-transform.cpp:61

Here, GDB is displaying the threads with the following format:

```
<inferior_number>.<thread_number>:<SIMD Lane/s>
```

So, for example, the thread id “2.3:[1 3 5 7]” refers to SIMD lanes 1, 3, 5, and 7 of thread 3 running on inferior 2.

An “inferior” in the GDB terminology is the process that is being debugged. In the debug session of a program that offloads to the GPU, there will typically be two inferiors; one “native” inferior representing a host part of the program (inferior 1 above), and another “remote” inferior representing the GPU device (inferior 2 above). Intel® Distribution for GDB automatically creates the GPU inferior - no extra steps are required.

When you print the value of an expression, the expression is evaluated in the context of the current thread’s current SIMD lane. You can switch the thread as well as the SIMD lane to change the context using the “thread” command such as “thread 3:4”, “thread :6”, or “thread 7”. The first command makes a

switch to the thread 3 and SIMD lane 4. The second command switches to SIMD lane 6 within the current thread. The third command switches to thread 7. The default lane selected will either be the previously selected lane, if it is active, or the first active lane within the thread.

The “thread apply command” may be similarly broad or focused (which can make it easier to limit the output from, for example, a command to inspect a variable). For more details and examples about debugging with SIMD lanes, see the [Debugging with Intel® Distribution for GDB on Linux OS Host Tutorial](#).

More information about threads and inferiors in GDB can be found from <https://sourceware.org/gdb/current/onlinedocs/gdb/Threads.html> and <https://sourceware.org/gdb/current/onlinedocs/gdb/Inferiors-Connections-and-Programs.html#Inferiors-Connections-and-Programs>.

Controlling the Scheduler

By default, when a thread hits a breakpoint, the debugger stops all the threads before displaying the breakpoint hit event to the user. This is the all-stop mode of GDB. In the non-stop mode, the stop event of a thread is displayed while the other threads run freely.

In all-stop mode, when a thread is resumed (for example, to resume normally with the `continue` command, or for stepping with the `next` command), all the other threads are also resumed. If you have some breakpoints set in threaded applications, this can quickly get confusing, as the next thread that hits the breakpoint may not be the thread you are following.

You can control this behavior using the `set scheduler-locking` command to prevent resuming other threads when the current thread is resumed. This is useful to avoid intervention of other threads while only the current thread executes instructions. Type `help set scheduler-locking` for the available options, and see <https://sourceware.org/gdb/current/onlinedocs/gdb/Thread-Stops.html> for more information. Note that SIMD lanes cannot be resumed individually; they are resumed together with their underlying thread.

In non-stop mode, by default, only the current thread is resumed. To resume all threads, pass the “-a” flag to the `continue` command.

Dumping Information on One or More Threads/Lanes (Thread Apply)

Commands for inspecting the program state are typically executed in the context of the current thread’s current SIMD lane. Sometimes it is desired to inspect a value in multiple contexts. For such needs, the `thread apply` command can be used. For instance, the following executes the `print element` command for the SIMD lanes 3-5 of Thread 2.5:

```
(gdb) thread apply 2.5:3-5 print element
```

Similarly, the following runs the same command in the context of SIMD lane 3, 5, and 6 of the current thread:

```
(gdb) thread apply :3 :5 :6 print element
```

Stepping GPU Code After a Breakpoint

To stop inside the kernel that is offloaded to the GPU, simply define a breakpoint at a source line inside the kernel. When a GPU thread hits that source line, the debugger stops the execution and shows the breakpoint hit. To single-step a thread over a source-line, use the `step` or `next` commands. The `step` commands steps into functions while `next` steps over calls. Before stepping, we recommend to `set scheduler-locking step` to prevent intervention of other threads.

Building a SYCL Executable for Use with Intel® Distribution for GDB*

Much like when you want to debug a host application, you need to set some additional flags to create a binary that can be debugged on the GPU. See [Get Started with Intel® Distribution for GDB on Linux* Host](#) for details.

For a smooth debug experience when using the just-in-time (JIT) compilation flow, enable debug information emission from the compiler via the `-g` flag, and disable optimizations via the `-O0` flag for both a host and JIT-compiled kernel of the application. The flags for the kernel are taken during link time. For example:

- Compile your program using: `icpx -fsycl -g -O0 -c myprogram.cpp`
- Link your program using: `icpx -fsycl -g -O0 myprogram.o`

If you are using CMake to configure the build of your program, use the `Debug` type for the `CMAKE_BUILD_TYPE`, and append `-O0` to the `CMAKE_CXX_FLAGS_DEBUG` variable. For example: `set (CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -O0")`

Applications that are built for debugging may take a little longer to start up than when built with the usual “release” level of optimization. Thus, your program may appear to run a little more slowly when started in the debugger. If this causes problems, developers of larger applications may want to use ahead-of-time (AOT) compilation to JIT the offload code when their program is built, rather than when it is run (warning, this may also take longer to build when using `-g -O0`). For more information, see [Compilation Flow Overview](#).

When doing ahead-of-time compilation for GPU, you must use a device type that fits your target device. Run the following command to see the available GPU device options on your current machine: `ocloc compile --help`

Additionally, the debug mode for the kernel must be enabled. The following example AoT compilation command targets the KBL device:

```
dpcpp -g -O0 -fsycl-targets=spir64_gen-unknown-unknown-sycldevice \
-Xs "-device kbl -internal_options -cl-kernel-debug-enable -options -cl-opt-disable"
myprogram.cpp
```

Building an OpenMP* Executable for use with Intel® Distribution for GDB*

Compile and link your program using the `-g -O0` flags. For example:

```
icpx -fiopenmp -O0 -fopenmp-targets=spir64 -c -g myprogram.cpp
icpx -fiopenmp -O0 -fopenmp-targets=spir64 -g myprogram.o
```

Set the following environment variables to disable optimizations and enable debug info for the kernel:

```
export LIBOMPTARGET_OPENCL_COMPILATION_OPTIONS="-g -cl-opt-disable"
export LIBOMPTARGET_LEVEL0_COMPILATION_OPTIONS="-g -cl-opt-disable"
```

Debugging GPU Execution

A common issue with offload programs is that they may fail to run at all, instead giving a generic OpenCL™ error with little additional information. The Intercept Layer for OpenCL™ Applications along with `onetrace`, `ze_tracer`, and `cl_tracer` can be used to get more information about these errors, often helping the developer identify the source of the problem.

Intercept Layer for OpenCL™ Applications

Using this library, in particular the `BuildLogging`, `ErrorLogging`, and `USMChecking=1` options, you can often find the source of the error.

1. Create a `clintercept.conf` file in the home directory with the following content:

```
SimpleDumpProgramSource=1
CallLogging=1
LogToFile=1
//KernelNameHashTracking=1
BuildLogging=1
ErrorLogging=1
USMChecking=1
```

```
//ContextCallbackLogging=1
// Profiling knobs
KernelInfoLogging=1
DevicePerformanceTiming=1
DevicePerformanceTimeLWSTracking=1
DevicePerformanceTimeGWSTracking=1
```

2. Run the application with cliloader as follows:

```
<OCL_Intercept_Install_Dir>/bin/cliloader/cliloader -d ./<app_name> <app_args>
```

3. Review the following results in the ~CLIntercept_Dump/<app_name> directory:

- clintercept_report.txt: Profiling results
- clintercept_log.txt: Log of OpenCL™ calls used to debug OpenCL issues

The following snippet is from an example log file generated by a program that returned the runtime error: CL_INVALID_ARG_VALUE (-50)

```
...
<<<< clSetKernelArgMemPointerINTEL -> CL_SUCCESS
>>>>
clGetKernelInfo( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257cS2_S2_S2_S2_S2_S2_S2_S2_S2_iENKU1RN2cl4sycl7handlerEE197->45clES6_EU1NS4_7nd_itemILi3EEEE225->13 ):
param_name = CL_KERNEL_CONTEXT (1193)
<<<< clGetKernelInfo -> CL_SUCCESS
>>>>
clSetKernelArgMemPointerINTEL( _ZTSZZ10outer_coreiP5mesh_i16dpct_type_1c0e3516dpct_type_60257cS2_S2_S2_S2_S2_S2_S2_S2_S2_iENKU1RN2cl4sycl7handlerEE197->45clES6_EU1NS4_7nd_itemILi3EEEE225->13 ): kernel = 0xa2d51a0, index = 3, value = 0x41995e0
mem pointer 0x41995e0 is an UNKNOWN pointer and no device support shared system pointers!
ERROR! clSetKernelArgMemPointerINTEL returned CL_INVALID_ARG_VALUE (-50)
<<<< clSetKernelArgMemPointerINTEL -> CL_INVALID_ARG_VALUE
```

In this example, the following values help with debugging the error:

- ZTSZZ10outer_coreiP5mesh
- index = 3, value = 0x41995e0

Using this data, you can identify which kernel had the problems, what argument was problematic, and why.

onetrace, ze_tracer, and cl_tracer

Similar to Intercept Layer for OpenCL™ Applications, the onetrace, ze_tracer and cl_tracer tools can help find the source of errors detected by the Level Zero and OpenCL™ runtimes.

To use the onetrace or ze_tracer tools to root-cause Level Zero issues (cl_tracer would be used the same way to root-cause OpenCL issues):

- 1. Use Call Logging mode to run the application. Redirecting the tool output to a file is optional, but recommended.**

```
./onetrace -c ./<app_name> <app_args> [2> log.txt]
```

The command for ze_tracer is the same - just substitute "ze_tracer" for "onetrace".

- 1. Review the call trace to figure out the error (log.txt). For example:**

```
>>>> [102032049] zeKernelCreate: hModule = 0x55a68c762690 desc = 0x7fff865b5570 {29 0 0 GEMM}
phKernel = 0x7fff865b5438 (hKernel = 0)
<<<< [102060428] zeKernelCreate [28379 ns] hKernel = 0x55a68c790280 -> ZE_RESULT_SUCCESS (0)
...
>>>> [102249951] zeKernelSetGroupSize: hKernel = 0x55a68c790280 groupSizeX = 256 groupSizeY = 1
groupSizeZ = 1
<<<< [102264632] zeKernelSetGroupSize [14681 ns] -> ZE_RESULT_SUCCESS (0)
```

```

>>>> [102278558] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 0 argSize = 8
pArgValue = 0x7fff865b5440
<<<< [102294960] zeKernelSetArgumentValue [16402 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [102308273] zeKernelSetArgumentValue: hKernel = 0x55a68c790280 argIndex = 1 argSize = 8
pArgValue = 0x7fff865b5458
<<<< [102321981] zeKernelSetArgumentValue [13708 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0)
...

```

The example log data shows:

- A level zero API call that causes the problem (`zeKernelSetArgumentValue`)
- The problem reason (`ZE_RESULT_ERROR_INVALID_ARGUMENT`)
- The argument index (`argIndex = 1`)
- An invalid value location (`pArgValue = 0x7fff865b5458`)
- A kernel handle (`hKernel = 0x55a68c790280`), which provides the name of the kernel for which this issue is observed (GEMM)

More information could be obtained by omitting the “redirection to file” option and dumping all the output (application output + tool output) into one stream. Dumping to one stream may help determine the source of the error in respect to application output (for example, you can find that the error happens between application initialization and the first phase of computations):

```

Level Zero Matrix Multiplication (matrix size: 1024 x 1024, repeats 4 times)
Target device: Intel® Graphics [0x3ea5]
...
>>>> [104131109] zeKernelCreate: hModule = 0x55af5f39ca10 desc = 0x7ffe289c7f80 {29 0 0 GEMM}
phKernel = 0x7ffe289c7e48 (hKernel = 0)
<<<< [104158819] zeKernelCreate [27710 ns] hKernel = 0x55af5f3ca600 -> ZE_RESULT_SUCCESS (0)
...
>>>> [104345820] zeKernelSetGroupSize: hKernel = 0x55af5f3ca600 groupSizeX = 256 groupSizeY = 1
groupSizeZ = 1
<<<< [104360082] zeKernelSetGroupSize [14262 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104373679] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 0 argSize = 8
pArgValue = 0x7ffe289c7e50
<<<< [104389443] zeKernelSetArgumentValue [15764 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104402448] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 1 argSize = 8
pArgValue = 0x7ffe289c7e68
<<<< [104415871] zeKernelSetArgumentValue [13423 ns] -> ZE_RESULT_ERROR_INVALID_ARGUMENT
(2013265924)
>>>> [104428764] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 2 argSize = 8
pArgValue = 0x7ffe289c7e60
<<<< [104442529] zeKernelSetArgumentValue [13765 ns] -> ZE_RESULT_SUCCESS (0)
>>>> [104455176] zeKernelSetArgumentValue: hKernel = 0x55af5f3ca600 argIndex = 3 argSize = 4
pArgValue = 0x7ffe289c7e2c
<<<< [104468472] zeKernelSetArgumentValue [13296 ns] -> ZE_RESULT_SUCCESS (0)
...
Matrix multiplication time: 0.0427564 sec
Results are INCORRECT with accuracy: 1
...
Matrix multiplication time: 0.0430995 sec

```

```
Results are INCORRECT with accuracy: 1
...
Total execution time: 0.381558 sec
```

Correctness

Offload code is often used for kernels that can efficiently process large amounts of information on the attached compute device, or to generate large amounts of information from some input parameters. If these kernels are running without crashing, this can often mean that you learn that they are not producing the correct results much later in program execution.

In these cases, it can be difficult to identify which kernel is producing incorrect results. One technique for finding the kernel producing incorrect data is to run the program twice, once using a purely host-based implementation, and once using an offload implementation, capturing the inputs and outputs from every kernel (often to individual files). Now compare the results and see which kernel call is producing unexpected results (within a certain epsilon - the offload hardware may have a different order of operation or native precision that causes the results to differ from the host code in the last digit or two).

Once you know which kernel is producing incorrect results, and you are working with a compatible GPU, use Intel Distribution for GDB to determine the reason. See the [Debugging with Intel® Distribution for GDB on Linux OS Host Tutorial](#) for basic information and links to more detailed documentation.

Both SYCL and OpenMP* also allow for the use of standard language print mechanisms (`printf` for SYCL and C++ OpenMP offload, `print *`, `...` for Fortran OpenMP offload) within offloaded kernels, which you can use to verify correct operation while they run. Print the thread and SIMD lane the output is coming from and consider adding synchronization mechanisms to ensure printed information is in a consistent state when printed. Examples for how to do this in SYCL using the stream class can be found in the [Intel oneAPI GPU Optimization Guide](#). You could use a similar approach to the one described for SYCL for OpenMP offload.

For more information about using OpenMP directives to add parallelism to your application, see **Offload and Optimize OpenMP* Applications with Intel Tools** <<https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/offload-optimize-openmp-applications.html>>`_

Tip Using `printf` can be verbose in SYCL kernels. To simplify, add the following macro:

```
#ifdef __SYCL_DEVICE_ONLY__
#define CL_CONSTANT __attribute__((opencl_constant))
#else
#define CL_CONSTANT
#endif
#define PRINTF(format, ...) { \
    static const CL_CONSTANT char _format[] = format; \
    sycl::ONEAPI::experimental::printf(_format, ## __VA_ARGS__); }
```

Usage example: `PRINTF("My integer variable:%d\n", (int) x);`

Failures

Just-in-time (JIT) compilation failures that occur at runtime due to incorrect use of the SYCL or OpenMP* offload languages will cause your program to exit with an error.

In the case of SYCL, if you cannot find these using ahead-of-time compilation of your SYCL code, selecting the OpenCL backend, setting `SimpleDumpProgramSource` and `BuildLogging`, and using the Intercept Layer for OpenCL™ Applications may help identify the kernel with the syntax error.

Logic errors can also result in crashes or error messages during execution. Such issues can include:

- Passing a buffer that belongs to the wrong context to a kernel
- Passing the “this” pointer to a kernel rather than a class element
- Passing a host buffer rather than a device buffer
- Passing an uninitialized pointer, even if it is not used in the kernel

Using the Intel® Distribution for GDB* (or even the native GDB), if you watch carefully, you can record the addresses of all contexts created and verify that the address being passed to an offload kernel belongs to the correct context. Likewise, you can verify that the address of a variable passed matches that of the variable itself, and not its containing class.

It may be easier to track buffers and addresses using the Intercept Layer for OpenCL™ allocation or `onetrace/cl_tracer` and choosing the appropriate backend. When using the OpenCL backend, setting `CallLogging`, `BuildLogging`, `ErrorLogging`, and `USMChecking` and running your program should produce output that explains what error in your code caused the generic OpenCL error to be produced.

Using `onetrace` or `ze_tracer`’s Call Logging or Device Timeline should give additional enhanced error information to help you better understand the source of generic errors from the Level Zero backend. This can help locate many of the logic errors mentioned above.

If the code is giving an error when offloading to a device using the Level Zero backend, try using the OpenCL backend. If the program works, report an error against the Level Zero backend. If the error reproduces in the OpenCL backend to the device, try using the OpenCL CPU backend. In OpenMP offload, this can be specified by setting `OMP_TARGET_OFFLOAD` to `CPU`. For SYCL, this can be done by setting `ONEAPI_DEVICE_SELECTOR=opencl:cpu`. Debugging with everything on the CPU can be easier, and removes complications caused by data copies and translation of the program to a non-CPU device.

As an example of a logic issue that can get you in trouble, consider what is captured by the lambda function used to implement the `parallel_for` in this SYCL code snippet.

```
class MyClass {
private:
    int *data;
    int factor;
    :
void run() {
    :
    auto data2 = data;
    auto factor2 = factor;
    {
        dpct::get_default_queue_wait().submit([&](cl::sycl::handler &cgh)
        {
            auto dpct_global_range = grid * block;
            auto dpct_local_range = block;
            cgh.parallel_for<dpct_kernel_name<class kernel_855a44>>(
                cl::sycl::nd_range<1>(
                    cl::sycl::range<1> dpct_global_range.get(0)),
                    cl::sycl::range<1>( dpct_local_range.get(0))),
                [=](cl::sycl::nd_item<3> item_ct1)
                {
                    kernel(data, b, factor, LEN, item_ct1);    // This blows up
                });
            });
    });
};
```

```
}  
} // run  
} // MyClass
```

In the above code snippet, the program crashes because [=] will copy by value all variables used inside the lambda. In the example it may not be obvious that "factor" is really "this->factor" and "data" is really "this->data," so "this" is the variable that is captured for the use of "data" and "factor" above. OpenCL or Level Zero will crash with an illegal arguments error in the "kernel(data, b, factor, LEN, item_ct1)" call.

The fix is the use of local variables `auto data2` and `auto factor2`. "auto factor2 = factor" becomes "int factor2 = this->factor" so using factor2 inside the lambda with [=] would capture an "int". We would rewrite the inner section as "kernel(data2, b, factor2, LEN, item_ct1);".

NOTE This issue is commonly seen when migrating CUDA* kernels. You can also resolve the issue by keeping the same CUDA kernel launch signature and placing the command group and lambda inside the kernel itself.

Using the Intercept Layer for OpenCL™ allocation or onetrace or ze_tracer, you would see that the kernel was called with two identical addresses, and the extended error information would tell you that you are trying to copy a non-trivial data structure to the offload device.

Note that if you are using unified shared memory (USM), and "MyClass" is allocated in USM, the above code will work. However, if only "data" is allocated in USM, then the program will crash for the above reason.

In this example, note that you can also re-declare the variables in local scope with the same name so that you don't need to change everything in the kernel call.

Intel® Inspector can also help diagnose these sorts of failures. If you set the following environment variables and then run Memory Error Analysis on offload code using the CPU device, Intel Inspector will flag many of the above issues:

- OpenMP*
 - `export OMP_TARGET_OFFLOAD=CPU`
 - `export OMP_TARGET_OFFLOAD=MANDATORY`
 - `export LIBOMPTARGET_PLUGIN=OPENCL`
- SYCL
 - `export ONEAPI_DEVICE_SELECTOR=opencl:cpu`
 - Or initialize your queue with a CPU selector to force use of the OpenCL CPU device: `cl::sycl::queue Queue(cl::sycl::cpu_selector{});`
- Both
 - `export CL_CONFIG_USE_VTUNE=True`
 - `export CL_CONFIG_USE_VECTORIZER=false`

NOTE A crash can occur when optimizations are turned on during the compilation process. If turning off optimizations causes your crash to disappear, use `-g -[optimization level]` for debugging. For more information, see the [Intel® oneAPI DPC++/C++ Compiler Developer Guide and Reference](#).

Using the SYCL* Exception Handler

As explained in the book [Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL](#):

The C++ exception features are designed to cleanly separate the point in a program where an error is detected from the point where it may be handled, and this concept fits very well with both synchronous and asynchronous errors in SYCL.

Using the methods from this book, C++ exception handling can help terminate a program when an error is encountered instead of allowing the program to silently fail.

Note: the *italicized* text in this section is copied directly from Chapter 5 “Error Handling” in the book *Data Parallel C++ Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. In some places, text has been removed for brevity. See the book for full details.

Ignoring Error Handling

C++ and SYCL are designed to tell us that something went wrong even when we don't handle errors explicitly. The default result of unhandled synchronous or asynchronous errors is abnormal program termination which an operating system should tell us about. The following two examples mimic the behavior that will occur if we do not handle a synchronous and an asynchronous error, respectively. The figure below shows the result of an unhandled C++ exception, which could be an unhandled SYCL synchronous error, for example. We can use this code to test what a particular operating system will report in such a case.

Unhandled exception in C++

```
#include <iostream>
class something_went_wrong {};
int main() {
    std::cout << "Hello\n";
    throw(something_went_wrong{});
}
Example output in Linux:
Hello
terminate called after throwing an instance of 'something_went_wrong'
Aborted (core dumped)
```

The next figure shows example output from `std::terminate` being called, which will be the result of an unhandled SYCL asynchronous error in our application. We can use this code to test what a particular operating system will report in such a case. Although we probably should handle errors in our programs, since uncaught errors will be caught and the program terminated, we do not need to worry about a program silently failing!

std::terminate is called when a SYCL asynchronous exception isn't handled

```
#include <iostream>
int main() {
    std::cout << "Hello\n";
    std::terminate();
}
Example output in Linux:
Hello
terminate called without an active exception
Aborted (core dumped)
```

The book details reasons for why synchronous errors can be handled by the C++ exceptions. However, to handle asynchronous errors at controlled points in an application, one must be aware of the situations where a SYCL throw is invoked, and accordingly, SYCL exceptions must be used.

Synchronous errors defined by SYCL are a derived class from `std::exception` of type ```sycl::exception```, which allows us to catch the SYCL errors specifically though a try-catch structure such as what we see in the figure below.

Pattern to catch `sycl::exception`s specifically

```
try{
  // Do some SYCL work
} catch (sycl::exception &e) {
  // Do something to output or handle the exception
  std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
  return 1;
}
```

On top of the C++ error handling mechanisms, SYCL adds a `* `sycl::exception` *type` for the exceptions thrown by the runtime. Everything else is standard C++ exception handling, so will be familiar to most developers. A slightly more complete example is provided in the figure below, where additional classes of exception are handled, as well as the program being ended by returning from `main()`. On top of the C++ error handling mechanisms, SYCL adds a `* `sycl::exception` *type` for the exceptions thrown by the runtime. Everything else is standard C++ exception handling, so will be familiar to most developers. A slightly more complete example is provided in the figure below, where additional classes of exception are handled, as well as the program being ended by returning from `main()`.

Pattern to catch exceptions from a block of code

```
try{
  buffer<int> B{ range{16} };
  // ERROR: Create sub-buffer larger than size of parent buffer
  // An exception is thrown from within the buffer constructor
  buffer<int> B2(B, id{8}, range{16});
} catch (sycl::exception &e) {
  // Do something to output or handle the exception
  std::cout << "Caught sync SYCL exception: " << e.what() << "\n";
  return 1;
} catch (std::exception &e) {
  std::cout << "Caught std exception: " << e.what() << "\n";
  return 2;
} catch (...) {
  std::cout << "Caught unknown exception\n";
  return 3;
}
return 0;
Example output:
Caught sync SYCL exception: Requested sub-buffer size exceeds the
size of the parent buffer -30 (CL_INVALID_VALUE)
```

Asynchronous Error Handling

Asynchronous errors are detected by the SYCL runtime (or an underlying backend), and the errors occur independently of execution of commands in the host program. The errors are stored in lists internal to the SYCL runtime and only released for processing at specific points that the programmer can control. There are two topics that we need to discuss to cover handling of asynchronous errors 1. **The asynchronous handler** that is invoked when there are outstanding asynchronous errors to process 2. **When the asynchronous handler is invoked** The Asynchronous HandleThe asynchronous handler is a function that the application defines, which is registered with SYCL contexts and/or queues. At the times defined by the next section, if there are any unprocessed asynchronous exceptions that are available to be handled, then the asynchronous handler is invoked by the SYCL runtime and passed a list of these exceptions. The asynchronous handler is passed to a context or queue constructor as `astd::function` and can be defined in ways such as a regular function, lambda, or functor, depending on our preference. The handler must accept `asycl::exception_list` argument, such as in the example handler shown in the figure below

Example asynchronous handler implementation defined as a lambda

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
    for (auto &e : elist) {
        try{ std::rethrow_exception(e); }
        catch ( sycl::exception& e ) {
            std::cout << "ASYNC EXCEPTION!!\n";
            std::cout << e.what() << "\n";
        }
    }
};
```

In the figure above, the `std::rethrow_exception` followed by `catch` of a specific exception type provides filtering of the type of exception, in this case to the only `sycl::exception`. We can also use alternative filtering approaches in C++ or just choose to handle all exceptions regardless of the type. The handler is associated with a queue or context (low-level detail covered more in [Chapter 6](#)) at construction time. For example, to register the handler defined in the figure above with a queue that we are creating, we could write `queue my_queue{ gpu_selector{}, handle_async_error }`. Likewise, to register the handler defined in the figure above with a context that we are creating, we could write `context my_context{ handle_async_error }`. Most applications do not need contexts to be explicitly created or managed (they are created behind the scenes for us automatically), so if an asynchronous handler is going to be used, most developers should associate such handlers with queues that are being constructed for specific devices (and not explicit contexts).

NOTE: In defining asynchronous handlers, most developers should define them on queues unless already explicitly managing contexts for other reasons. If an asynchronous handler is not defined for a queue or the queue's parent context and an asynchronous error occurs on that queue (or in the context) that must be processed, then the default asynchronous handler is invoked. The default handler operates as if it was coded as shown in the figure below.

Example of how the default asynchronous handler behaves

```
// Our simple asynchronous handler function
auto handle_async_error = [](exception_list elist) {
    for (auto &e : elist) {
        try{ std::rethrow_exception(e); }
        catch ( sycl::exception& e ) {
            // Print information about the asynchronous exception
        }
    }
    // Terminate abnormally to make clear to user
    // that something unhandled happened
    std::terminate();
};
```

The default handler should display some information to the user on any errors in the exception list and then will terminate the application abnormally, which should also cause the operating system to report that termination was abnormal. What we put within an asynchronous handler is up to us. It can range from logging of an error to application termination to recovery of the error condition so that an application can continue executing normally. The common case is to report any details of the error available by calling `sycl::exception::what()`, followed by termination of the application. Although it's up to us to decide what an asynchronous handler does internally, a common mistake is to print an error message (that may be missed in the noise of other messages from the program), followed by completion of the handler function. Unless we have error management principles in place that allow us to recover known program state and to be confident that it's safe to continue execution, we should consider terminating the application within our asynchronous handler function(s). This reduces the chance that incorrect results will appear from a

program where an error was detected, but where the application was inadvertently allowed to continue with execution regardless. In many programs, abnormal termination is the preferred result once we have experienced asynchronous exceptions.

Example: SYCL Throw on Zero Sized Object

The source code below shows how the SYCL handler will produce an error when a zero-sized object is passed.

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

template <bool non_empty>
static void fill(sycl::buffer<int> buf, sycl::queue & q) {
    q.submit([&](sycl::handler & h) {
        auto acc = sycl::accessor { buf, h, sycl::read_write };
        h.single_task( [= ] () {
            if constexpr(non_empty) {
                acc[0] = 1;
            }
        });
    });
    q.wait();
}

int main(int argc, char *argv[]) {
    sycl::queue q;
    sycl::buffer<int, 1> buf_zero ( 0 );

    fprintf(stderr, "buf_zero.count() = %zu\n", buf_zero.get_count());
    fill<false>(buf_zero, q);
    fprintf(stdout, "PASS\n");

    return 0;
}
```

When the application encounters the zero-sized object at runtime, the program aborts and produces an error message:

```
$ dpcpp zero.cpp
$ ./a.out
buf_zero.count() = 0
submit...
terminate called after throwing an instance of 'cl::sycl::invalid_object_error'
  what():  SYCL buffer size is zero. To create a device accessor, SYCL buffer size must be
greater than zero. -30 (CL_INVALID_VALUE)
Aborted (core dumped)
```

The programmer can then locate the programming error by catching the exception in the debugger and looking at the backtrace for the source line that triggered the error.

Example: SYCL Throw on Illegal Null Pointer

Consider code that does the following:

```
deviceQueue.memset(mdlReal, 0, mdlXYZ * sizeof(XFLOAT));

deviceQueue.memcpy(mdlImag, 0, mdlXYZ * sizeof(XFLOAT)); // coding
error
```

The compiler will not flag the bad (null pointer) value specified in `deviceQueue.memcpy`. This error will not be caught until runtime.

```
terminate called after throwing an instance of 'cl::sycl::runtime_error'

what(): NULL pointer argument in memory copy operation. -30
(CL_INVALID_VALUE)

Aborted (core dumped)
```

The example code that follows shows a way the user can control the format of the exception output when it is detected at runtime on a given queue, implemented in a standalone program that demonstrates the null pointer error.

```
#include "stdlib.h"
#include "stdio.h"
#include <cmath>
#include <signal.h>
#include <fstream>
#include <iostream>
#include <vector>
#include <CL/sycl.hpp>

#define XFLOAT float
#define mdlXYZ 1000
#define MEM_ALIGN 64

int main(int argc, char *argv[])
{
    XFLOAT *mdlReal, *mdlImag;

    cl::sycl::property_list propList =
cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};
cl::sycl::queue deviceQueue(cl::sycl::gpu_selector { }, [&](cl::sycl::exception_list eL)
    {
        bool error = false;
        for (auto e : eL)
        {
            try
            {
                std::rethrow_exception(e);
            } catch (const cl::sycl::exception& e)
            {
                auto clError = e.get_cl_code();
                bool hascontext = e.has_context();
                std::cout << e.what() << "CL ERROR CODE : " << clError << std::endl;
                error = true;
                if (hascontext)
                {
                    std::cout << "We got a context with this exception" << std::endl;
                }
            }
        }
    })
}
```

```
    }
    if (error) {
        throw std::runtime_error("SYCL errors detected");
    }
}, propList);

mdlReal = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);
mdlImag = sycl::malloc_device<XFLOAT>(mdlXYZ, deviceQueue);

deviceQueue.memset(mdlReal, 0, mdlXYZ * sizeof(XFLOAT));
deviceQueue.memcpy(mdlImag, 0, mdlXYZ * sizeof(XFLOAT)); // coding error

deviceQueue.wait();

exit(0);
}
```

Resources

For a guided approach to debugging SYCL exceptions from incorrect use of the SYCL* API, see the [Guided Matrix Multiplication Exception Sample](#).

To troubleshoot your applications that use OpenMP* or the SYCL* API with extensions to offload resources, see [Troubleshoot Highly Parallel Applications](#).

Optimize Offload Performance

Offload performance optimization basically boils down to three tasks:

1. Minimize the number and size of data transfers to and from the device while maximizing execution time of the kernel on the device.
2. When possible, overlap data transfers to/from the device with computation on the device.
3. Maximize the performance of the kernel on the device.

While it is possible to take explicit control of data transfers in both OpenMP* offload and SYCL*, you also can allow this to happen automatically. In addition, because the host and offload device operate mostly asynchronously, even if you try to take control over data transfers, the transfers may not happen in the expected order, and may take longer than anticipated. When data used by both the device and the host is stored in unified shared memory (USM), there is another transparent layer of data transfers happening that also can affect performance.

Resources:

- [oneAPI GPU Optimization Guide](#)
- [Intel® oneAPI FPGA Optimization Guide](#)

Buffer Transfer Time Versus Execution Time

Transferring any data to or from an offload device is relatively expensive, requiring memory allocations in user space, system calls, and interfacing with hardware controllers. Unified shared memory (USM) adds to these costs by requiring that some background process keeps memory being modified on either the host or offload device in sync. Furthermore, kernels on the offload device must wait to run until all the input or output buffers they need to run are set up and ready to use.

All this overhead is roughly the same no matter how much information you need to transfer to or from the offload device in a single data transfer. Thus, it is much more efficient to transfer 10 numbers in bulk rather than one at a time. Still, every data transfer is expensive, so minimizing the total number of transfers is also very important. If, for example, you have some constants that are needed by multiple kernels, or during multiple invocations of the same kernel, transfer them to the offload device once and reuse them, rather than sending them with every kernel invocation. Finally, as might be expected, single large data transfers take more time than single small data transfers.

The number and size of buffers sent is only part of the equation. Once the data is at the offload device, consider how long the resulting kernel executes. If it runs for less time than it takes to transfer the data to the offload device, it may not be worthwhile to offload the data in the first place unless the time to do the same operation on the host is longer than the combined kernel execution and data transfer time.

Finally, consider how long the offload device is idle between the execution of one kernel and the next. A long wait could be due data transfer or just the nature of the algorithm on the host. If the former, it may be worthwhile to overlap data transfer and kernel execution, if possible.

In short, execution of code on the host, execution of code on the offload device, and data transfer is quite complex. The order and time of such operations isn't something you can gain through intuition, even in the simplest code. You need to make use of tools like those listed below to get a visual representation of these activities and use that information to optimize your offload code.

Intel® VTune™ Profiler

In addition to giving you detailed performance information on the host, VTune can also provide detailed information about performance on a connected GPU. Setup information for GPUs is available from the [Intel VTune Profiler User Guide](#).

Intel VTune Profiler's GPU Offload view gives you an overview of the hotspots on the GPU, including the amount of time spent for data transfer to and from each kernel. The GPU Compute/Media Hotspots view allows you to dive more deeply into what is happening to your kernels on the GPU, such as by using the **Dynamic Instruction Count** to view a micro analysis of the GPU kernel performance. With these profiling modes, you can observe how data transfer and compute occur over time, determine if there is enough work for a kernel to run effectively, learn how your kernels use the GPU memory hierarchy, and so on.

Additional details about these analysis types is available from the [Intel VTune Profiler User Guide](#). A detailed look at optimizing for GPU using VTune Profiler is available from the [Optimize Applications for Intel GPUs with Intel VTune Profiler](#) page.

You can also use Intel VTune Profiler to capture kernel execution time. The following commands provide light-weight profiling results:

- **Collect**
 - **Level zero backend:** `vtune -collect-with runss -knob enable-gpu-level-zero=true -finalization-mode=none -app-working-dir <app_working_dir> <app>`
 - **OpenCL™ backend:** `vtune -collect-with runss -knob collect-programming-api=true -finalization-mode=none -r <result_dir_name> -app-working-dir <app_working_dir> <app>`
- **Report:** `vtune --report hotspots --group-by=source-computing-task --sort-desc="Total Time" -r <result_dir_name>`

Intel® Advisor

Intel® Advisor provides two features that can help you get the improved performance when offloading computation to GPU:

- Offload Modeling can watch your host OpenMP* program and recommend parts of it that would be profitably offloaded to the GPU. It also allows you to model a variety of different target GPUs, so that you can learn if offload will be profitable on some but not others. Offload Advisor gives detailed information on what factors may bound offload performance.
- GPU Roofline analysis can watch your application when it runs on the GPU, and graphically show how well each kernel is making use of the memory subsystem and compute units on the GPU. This can let you know how well your kernel is optimized for the GPU.

To run these modes on an application that already does some offload, you need to set up your environment to use the OpenCL™ device on the CPU for analysis. Instructions are available from the [Intel Advisor User Guide](#).

Offload modeling does not require that you have already modified your application to use a GPU - it can work entirely on host code.

Resources:

- [Intel Advisor Cookbook: GPU Offload](#)
- [Get Started with Offload Modeling](#)
- [Get Started with GPU Roofline](#)

Offload API Call Timelines

If you do not want to use Intel® VTune™ Profiler to understand when data is being copied to the GPU, and when kernels run, onetrace, ze_tracer, cl_tracer, and the Intercept Layer for OpenCL™ Applications give you a way to observe this information /(although, if you want a graphical timeline, you'll need to write a script to visualize the output/). For more information, see [oneAPI Debug Tools](#), [Trace the Offload Process](#), and [Debug the Offload Process](#).

Performance Tuning Cycle

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

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

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

Establish Baseline

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

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

Identify Kernels to Offload

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

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

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

See [Get Started: Identify High-impact Opportunities to Offload to GPU](#) for quick steps to ramp up with the Offload Modeling perspective. For more resources about modeling performance of your application on GPU platforms, see [Offload Modeling Resources for Intel® Advisor Users](#).

Offload Kernels

After identifying kernels that are suitable for offload, employ SYCL* or OpenMP* to offload the kernel onto the device. Consult the previous chapters as an information resource.

Optimize Your SYCL* Applications

oneAPI enables functional code that can execute on multiple accelerators such as CPU, GPU, and FPGA. However, the code may not be the most optimal across the accelerators. A three-step optimization strategy is recommended to meet performance needs:

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

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

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

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

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

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

High-Level Optimization Tips

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

Loop-Related Optimizations

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

Memory-Related Optimizations

- When possible, favor greater computation over greater memory use. The latency and bandwidth of memory compared to computation can become a bottleneck.
- When possible, favor greater local and private memory use over global memory use.
- Avoid pointer aliasing.

- Coalesce memory accesses. Grouping memory accesses helps limit the number of individual memory requests and increases utilization of individual cache lines.
- When possible, store variables and arrays in private memory for high-execution areas of code.
- Beware of loop unrolling effects on concurrent memory accesses.
- Avoid a write to a global that another kernel reads. Use a pipe instead.
- Consider employing the `[[intel::kernel_args_restrict]]` attribute to a kernel. The attribute allows the compiler to ignore dependencies between accessor arguments in the kernel. In turn, ignoring accessor argument dependencies allows the compiler to perform more aggressive optimizations and potentially improve the performance of the kernel.

SYCL-Specific Optimizations

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

Recompile, Run, Profile, and Repeat

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

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

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

oneAPI Library Compatibility

oneAPI applications may include dynamic libraries at runtime that require compatibility across release versions of Intel tools. Intel® oneAPI Toolkits and component products use [semantic versioning](#) to support compatibility.

The following policies apply to APIs and ABIs delivered with Intel oneAPI Toolkits.

NOTE oneAPI applications are supported on 64-bit target devices.

- New Intel oneAPI device drivers, oneAPI dynamic libraries, and oneAPI compilers will not break previously deployed applications built with oneAPI tools. Current APIs will not be removed or modified without notice and an iteration of the major version.
- Developers of oneAPI applications should ensure that the header files and libraries have the same release version. For example, an application should not use 2021.2 Intel® oneAPI Math Kernel Library header files with 2021.1 Intel oneAPI Math Kernel Library.
- New dynamic libraries provided with the Intel compilers will work with applications built by older versions of the compilers (this is commonly referred to as *backward compatibility*). However, the converse is not true: newer versions of the oneAPI dynamic libraries may contain routines that are not available in earlier versions of the library.
- Older dynamic libraries provided with the oneAPI Intel compilers will not work with newer versions of the oneAPI compilers.

Developers of oneAPI applications should ensure that thorough application testing is conducted to ensure that a oneAPI application is deployed with a compatible oneAPI library.

SYCL* Extensions

SYCL extensions allow developers to quickly experiment, innovate, and create new solutions. They help foster a continuous improvement cycle within open standards organizations, such as the Khronos Group, which support the development of cross-architecture systems.

For a list of SYCL extensions supported by the Intel® oneAPI DPC++ Compiler, refer to [SYCL Extensions](#).

Glossary

Accelerator

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

See also: Device

Accessor

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

Application Scope

Code that executes on the host.

Buffers

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

Command Group Scope

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

Command Queue

Issues command groups concurrently.

Compute Unit

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

Device

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

See also: Accelerator

Device Code

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

DPC++

An open source project is adding SYCL* support to the LLVM C++ compiler.

Fat Binary

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

Fat Library

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

Fat Object

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

Host

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

Host Code

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

Images

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

Kernel Scope

Code that executes on the device.

ND-Range

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

Processing Element

Individual engine for computation that makes up a compute unit.

Single Source

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

SPIR-V

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

SYCL

A standard for a cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file.

Work-Groups

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

Work-Item

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

Notices and Disclaimers

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Product and Performance Information

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Notice revision #20201201

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

Copyright Intel Corporation

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.