# **SYCL Program Structure**

- What is SYCL and Data Parallel C++?
- SYCL Classes

.

- Device
- Code: Device Selector
- Queue
- Kernel
- Parallel Kernels
  - Basic Parallel Kernels
  - ND-Range Kernels
- Memory Models
  - Code: Vector Add implementation using USM and Buffers
  - Unified Shared Memory Model
  - Buffer Memory Model
    - *Code:* Synchronization: Host Accessor
    - Code: Synchronization: Buffer Destruction
  - Code: Custom Device Selector
- Multi-GPU Selection
- Code: Complex Number Multiplication
- Lab Exercise: Vector Add

# Learning Objectives

- Explain the SYCL fundamental classes
- Use **device selection** to offload kernel workloads
- Decide when to use basic parallel kernels and ND Range Kernels
- Use Unified Shared Memory or Buffer-Accessor memory model in SYCL program
- Build a sample **SYCL application** through hands-on lab exercises

# What is SYCL and Data Parallel C++?

**SYCL** is an open standard to program for heterogeneous devices in a single source. A SYCL program is invoked on the host computer and offloads the computation to an accelerator. Programmers use familiar C++ and library constructs with added functionalities like a **queue** for work targeting, **buffer** or **Unified Shared Memory** for data management, and **parallel\_for** for parallelism to direct which parts of the computation and data should be offloaded. **Data Parallel C++ (DPC++)** is oneAPI's implementation of SYCL.

## SYCL Language and Runtime

SYCL language and runtime consists of a set of C++ classes, templates, and libraries.

## Application scope and command group scope:

- Code that executes on the host
- The full capabilities of C++ are available at application and command group scope

#### Kernel scope:

- Code that executes on the device.
- At kernel scope there are limitations in accepted C++

#### C++ SYCL Code Example

Let's look at a simple SYCL code to offload computation to GPU, the code does the following:

- 1. selects GPU device for offload
- 2. allocates memory that can be accessed on host and GPU
- 3. initializes data array on host
- 4. offloads computation to GPU
- 5. prints output on host

```
#include <sycl/sycl.hpp>
static const int N = 16;
int main(){
  sycl::queue q(sycl::gpu_device_selector_v); // <---- select GPU for offload
  int *data = sycl::malloc_shared<int>(N, q); // <---- allocate memory
  for(int i=0; i<N; i++) data[i] = i;
  q.parallel_for(N, [=] (auto i){
    data[i] *= 2; // <--- Kernel Code (executes on GPU)
  }).wait();
  for(int i=0; i<N; i++) std::cout << data[i] << "\n";
  sycl::free(data, q);
  return 0;
}</pre>
```

Programs which use SYCL requires the include of the header file sycl/sycl.hpp.

In the next few sections we will learn the basics of C++ SYCL programming.

#### SYCL Classes

Below are some important SYCL Classes that are used to write a C++ with SYCL program to offload computation to heterogeneous devices.

## Device

The **device** class represents the capabilities of the accelerators in a system utilizing Intel® oneAPI Toolkits. The device class contains member functions for querying information about the device, which is useful for SYCL programs where multiple devices are created.

- The function **get\_info** gives information about the device:
- Name, vendor, and version of the device
- The local and global work item IDs
- Width for built in types, clock frequency, cache width and sizes, online or offline

```
queue q;
device my_device = q.get_device();
std::cout << "Device: " << my device.get info<info::device::name>() << "\n";</pre>
```

## **Device Selector**

These classes enable the runtime selection of a particular device to execute kernels based upon user-provided heuristics. The following code sample shows use of the standard device selectors (**default\_selector\_v**, **cpu\_selector\_v**, **gpu\_selector\_v**, **accelerator\_selector\_v**)

```
queue q(gpu_selector_v);
//queue q(cpu_selector_v);
//queue q(accelerator_selector_v);
//queue q(default_selector_v);
//queue q;
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << "\n";</pre>
```

### Exercise #1

- 1. Inspect the code in *lab1/gpu\_sample.cpp* file showing different device selectors in use.
- On Intel Developer Cloud machine, please make sure that the environment is set: source /opt/intel/oneapi/setvars.sh
- Compile and run the code example with -fsycl option:
  - <mark>icpx -fsycl gpu\_sample.cpp -o gpu\_sample</mark>

<mark>./gpu\_sample</mark>

Device: Intel(R) Data Center GPU Max 1100

- NOTE: by default you are logged in to the Intel Developer Cloud head(login) node with no GPUs available. You may allocate a node in interactive mode via following command: srun -p pvc-shared --pty bash
- Use cpu\_selector\_v instead of gpu\_selector\_v, recompile and rerun the code: icpx -fsycl gpu\_sample.cpp -o gpu\_sample ,/gpu\_sample Device: Intel(R) Xeon(R) Platinum 8480+
- 5. Use the default queue constructor and check what happens at runtime in that case.
- 6. Set the device to CPU via ONEAPI\_DEVICE\_SELECTOR\_environment variable at runtime:
  - ONEAPI\_DEVICE\_SELECTOR=opencl:cpu ./gpu\_sample

#### Queue

**Queue** submits command groups to be executed by the SYCL runtime. Queue is a mechanism where **work is submitted** to a device. A queue map to one device and multiple queues can be mapped to the same device.

```
q.submit([&](handler& h) {
    //COMMAND GROUP CODE
});
```

#### Kernel

The **kernel** class encapsulates methods and data for executing code on the device when a command group is instantiated. Kernel object is not explicitly constructed by the user and is constructed when a kernel dispatch function, such as **parallel\_for**, is called

```
q.submit([&](handler& h) {
    h.parallel_for(range<1>(N), [=](id<1> i) {
        A[i] = B[i] + C[i]);
    });
});
```

## Choosing where device kernels run

Work is submitted to queues and each queue is associated with exactly one device (e.g. a specific GPU, CPU or FPGA). You can decide which device a queue is associated with (if you want) and have as many queues as desired for dispatching work in heterogeneous systems.



## **Parallel Kernels**

**Parallel Kernel** allows multiple instances of an operation to execute in parallel. This is useful to **offload** parallel execution of a basic **for-loop** in which each iteration is completely independent and in any order. Parallel kernels are expressed using the **parallel\_for** function. A simple 'for' loop in a C++ application is written as below

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});</pre>
```

Below is how you can offload to accelerator

```
h.parallel_for(range<1>(1024), [=](id<1> i){
        A[i] = B[i] + C[i];
});
```

## **Basic Parallel Kernels**

The functionality of basic parallel kernels is exposed via **range**, **id**, and **item** classes. **Range** class is used to describe the **iteration space** of parallel execution and **id** class is used to **index** an individual instance of a kernel in a parallel execution

```
h.parallel_for(range<1>(1024), [=](id<1> i){
// CODE THAT RUNS ON DEVICE
});
```

The above example is good if all you need is the **index (id)**, but if you need the **range** value in your kernel code, then you can use **item** class instead of **id** class, which you can use to query for the **range** as shown below. **item** class represents an **individual instance** of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](item<1> item){
    auto i = item.get_id();
    auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
```

});

# ND-Range Kernels

Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level. **ND-Range kernel** is another way to expresses parallelism which enable low level performance tuning by providing access to **local memory and mapping executions** to compute units on hardware. The entire iteration space is divided into smaller groups called **work-groups**, **work-items** within a work-group are scheduled on a single compute unit on hardware.

The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution. The functionality of nd\_range kernels is exposed via **nd\_range** and **nd\_item** classes. **nd\_range** class represents a **grouped execution range** using global execution range and the local execution range of each work-group. **nd\_item** class represents an **individual instance** of a kernel function and allows to query for work-group range and index.

```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```





## Memory Models

A SYCL application can be written using one of the 2 memory models:

- Unified Shared Memory Model (USM)
- Buffer Memory Model

**Unified Shared Memory** Model is pointer-based approach to memory model, similar to C/C++ pointer-based memory allocation. Makes migrating C/C++/CUDA\* application to SYCL easier. Dependencies between multiple kernels are explicitly handled using events.

**Buffer Memory Model** allows a new memory abstraction called buffers and are accessed using accessors which allows setting read/write permissions and other properties to memory. Allows data representation in 1,2 or 3-dimentions and makes programming kernels with 2/3-dimentional data easier. Dependencies between multiple kernels are implicitly handled.

## Unified Shared Memory Model

Unified Shared Memory (USM) is a **pointer-based approach** that should be familiar to C and C++ programmers who use malloc or new to allocate data. USM **simplifies development** for the programmer when **porting existing C/C++/CUDA code** to SYCL.

### SYCL Code Anotomy - USM

Make sure that the SYCL header **sycl/sycl.hpp** is included. It is recommended (for these exercises) to employ the namespace statement to save typing repeated references into the sycl namespace.

#include <sycl/sycl.hpp>
using namespace sycl;

**SYCL programs** are standard C++. The program is invoked on the **host** computer, and offloads computation to the **accelerator**. A programmer uses SYCL's **queue and kernel abstractions** to direct which parts of the computation and data should be offloaded.

As a first step in a SYCL program we create a **queue**. We offload computation to a **device** by submitting tasks to a queue. The programmer can choose CPU, GPU, FPGA, and other devices through the **selector**. This program uses the default q here, which means SYCL runtime selects the most capable device available at runtime by using the default selector.

Device and host can either share physical **memory** or have distinct memories. When the memories are distinct, offloading computation requires **copying data between host and device**. We use USM device allocation malloc\_device to allocate memory on device and copy data between host and device using memcpy method.

In a SYCL program, we define a **kernel**, which is applied to every point in an index space. For simple programs like this one, the index space maps directly to the elements of the array. The kernel is encapsulated in a **C++ lambda function**. The lambda function is passed a point in the index space as an array of coordinates. For this simple program, the index space coordinate is the same as the array index. The **parallel\_for** in the below program applies the lambda to the index space. The index space is defined in the first argument of the parallel\_for as a 1 dimensional **range from 0 to N-1**.

The code below shows Simple Vector addition using SYCL and USM. Read through the comments addressed in step 1 through step 6.

```
void SYCL code(int* a, int* b, int* c, int N) {
  //Step 1: create a device queue
  //(developer can specify a device type via device selector or use default selector)
  queue q;
  //Step 2: create USM device allocation
  auto a device = malloc device<int>(N, q);
  auto b_device = malloc_device<int>(N, q);
  auto c device = malloc device<int>(N, q);
  //Step 3: copy memory from host to device
  q.memcpy(a_device, a, N*sizeof(int));
  q.memcpy(b device, b, N*sizeof(int));
  q.wait();
  //Step 4: send a kernel (lambda) for execution
  q.parallel_for(N, [=](auto i){
    //Step 5: write a kernel
    //Kernel invocations are executed in parallel
    //Kernel is invoked for each element of the range
    //Kernel invocation has access to the invocation id
    c_device[i] = a_device[i] + b_device[i];
  }).wait();
  //Step 6: copy the result back to host
  q.memcpy(c, c device, N*sizeof(int)).wait();
```

## Buffer Memory Model

**Buffers encapsulate** data in a SYCL application across both devices and host. **Accessors** is the mechanism to access buffer data.

As explained earlier in USM model section, offloading computation requires **copying data between host and device**. In Buffer Memory model SYCL does not require the programmer to manage the data copies. By creating **Buffers and Accessors**, SYCL ensures that the data is available to host and device without any programmer effort. SYCL also allows the programmer explicit control over data movement when it is necessary to achieve best performance. The code below shows Simple Vector addition using SYCL and Buffers. Read through the comments addressed in step 1 through step 6.

```
void SYCL code(int* a, int* b, int* c, int N) {
  //Step 1: create a device queue
  //(developer can specify a device type via device selector or use default selector)
  queue q;
  //Step 2: create buffers (represent both host and device memory)
 buffer buf a(a, range<1>(N));
 buffer buf b(b, range<1>(N));
  buffer buf c(c, range<1>(N));
  //Step 3: submit a command for (asynchronous) execution
  q.submit([&](handler &h){
    //Step 4: create buffer accessors to access buffer data on the device
    accessor A(buf a,h,read only);
    accessor B(buf b,h,read only);
    accessor C(buf_c,h,write_only);
    //Step 5: send a kernel (lambda) for execution
    h.parallel for(N, [=](auto i){
      //Step 6: write a kernel
      //Kernel invocations are executed in parallel
      //Kernel is invoked for each element of the range
      //Kernel invocation has access to the invocation id
      C[i] = A[i] + B[i];
    });
  });
}
```

#### Vector Add implementation using USM and Buffers

The SYCL code below shows vector add computation implemented using USM and Buffers memory model:

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

```
using namespace sycl;
```

```
// kernel function to compute vector add using Unified Shared memory model (USM)
void kernel usm(int* a, int* b, int* c, int N) {
  //Step 1: create a device queue
  queue q;
  //Step 2: create USM device allocation
  auto a device = malloc device<int>(N, q);
  auto b device = malloc device<int>(N, q);
  auto c device = malloc device<int>(N, q);
  //Step 3: copy memory from host to device
 q.memcpy(a device, a, N*sizeof(int));
 q.memcpy(b device, b, N*sizeof(int));
  q.wait();
 //Step 4: send a kernel (lambda) for execution
  q.parallel for(N, [=](auto i){
    //Step 5: write a kernel
    c_device[i] = a_device[i] + b_device[i];
```

}

```
}).wait();
  //Step 6: copy the result back to host
 q.memcpy(c, c_device, N*sizeof(int)).wait();
  //Step 7: free device allocation
 free(a_device, q);
  free(b_device, q);
  free(c_device, q);
}
// kernel function to compute vector add using Buffer memory model
void kernel buffers(int* a, int* b, int* c, int N) {
  //Step 1: create a device queue
 queue q;
  //Step 2: create buffers
 buffer buf a(a, range<1>(N));
 buffer buf b(b, range<1>(N));
 buffer buf c(c, range<1>(N));
  //Step 3: submit a command for (asynchronous) execution
  q.submit([&](handler &h){
    //Step 4: create buffer accessors to access buffer data on the device
    accessor A(buf a, h, read only);
    accessor B(buf b, h, read only);
    accessor C(buf c, h, write only);
    //Step 5: send a kernel (lambda) for execution
    h.parallel for (N, [=] (auto i) {
      //Step 6: write a kernel
      C[i] = A[i] + B[i];
    });
  });
}
int main() {
  // initialize data arrays on host
  constexpr int N = 256;
  int a[N], b[N], c[N];
  for (int i=0; i<N; i++) {</pre>
    a[i] = 1;
   b[i] = 2;
  }
  // initialize c = 0 and offload computation using USM, print output
  for (int i=0; i<N; i++) c[i] = 0;</pre>
  kernel usm(a, b, c, N);
  std::cout << "Vector Add Output (USM): \n";</pre>
  for (int i=0; i<N; i++)std::cout << c[i] << " ";std::cout << "\n";</pre>
  // initialize c = 0 and offload computation using USM, print output
  for (int i=0; i<N; i++) c[i] = 0;</pre>
  std::cout << "Vector Add Output (Buffers): \n";</pre>
  kernel buffers(a, b, c, N);
  for (int i=0; i<N; i++)std::cout << c[i] << " ";std::cout << "\n";</pre>
```

}

#### Exercise #2

- Inspect the code in lab1/vector\_add\_usm\_buffers.cpp file showing vector add computation implemented using USM and Buffers memory model.
- Compile and run this code: icpx -fsycl vector\_add\_usm\_buffers.cpp -o vector\_add\_usm\_buffers ./vector\_add\_usm\_buffers
- Set the environment variable <u>SYCL\_PI\_TRACE</u> to enable the tracing of plugins/devices discovery. You should be able to check on which device you are running: SYCL\_PI\_TRACE=1 ./vector\_add\_usm\_buffers

- Set ONEAPI\_DEVICE\_SELECTOR to select a different backend, e.g. OpenCL: SYCL\_PI\_TRACE=1 ONEAPI\_DEVICE\_SELECTOR=opencl:gpu ./vector\_add\_usm\_buffers
- Now you run on Intel<sup>®</sup> Data Center GPU Max 1100 with OpenCL backend: SYCL\_PI\_TRACE[all]: platform: Intel(R) OpenCL Graphics SYCL\_PI\_TRACE[all]: device: Intel(R) Data Center GPU Max 1100

#### Synchronization: Host Accessor

The Host Accessor is an accessor which uses host buffer access target. It is created outside of the scope of the command group and the data that this gives access to will be available on the host. These are used to synchronize the data back to the host by constructing the host accessor objects. Buffer destruction is the other way to synchronize the data back to the host.

Buffer takes ownership of the data stored in vector. Creating host accessor is a **blocking call** and will only return after all enqueued SYCL kernels that modify the same buffer in any queue completes execution and the data is available to the host via this host accessor.

The SYCL code below demonstrates Synchronization with Host Accessor:

```
#include <sycl/sycl.hpp>
using namespace sycl;
int main() {
 constexpr int N = 16;
  auto R = range < 1 > (N);
 std::vector<int> v(N, 10);
 queue q;
  // Buffer takes ownership of the data stored in vector.
 buffer buf(v);
  q.submit([&](handler& h) {
   accessor a(buf,h);
   h.parallel for(R, [=](auto i) { a[i] -= 2; });
  });
  // Creating host accessor is a blocking call and will only return after all
  // enqueued SYCL kernels that modify the same buffer in any queue completes
  // execution and the data is available to the host via this host accessor.
  host accessor b(buf,read only);
  for (int i = 0; i < N; i++) std::cout << b[i] << " ";</pre>
  return 0;
}
```

#### Exercise #3

Inspect the code in lab1/host\_accessor\_sample.cpp file
 Compile and run this code

 icpx -fsycl host\_accessor\_sample.cpp -o host\_accessor\_sample
 ./host\_accessor\_sample

 Comment the host accessor creation and print vectors v values. What happens in that case?

#### Synchronization: Buffer Destruction

In the below example Buffer creation happens within a separate function scope. When execution advances beyond this **function scope**, buffer destructor is invoked which relinquishes the ownership of data and copies back the data to the host memory.

The SYCL code below demonstrates Synchronization with Buffer Destruction:

```
#include <sycl/sycl.hpp>
constexpr int N = 16;
using namespace sycl;
```

```
// Buffer creation happens within a separate function scope.
void SYCL code(std::vector<int> &v, queue &q) {
  auto R = range < 1 > (N);
 buffer buf(v);
  q.submit([&](handler &h) {
   accessor a(buf,h);
   h.parallel for(R, [=](auto i) { a[i] -= 2; });
 });
}
int main() {
 std::vector<int> v(N, 10);
 queue q;
 SYCL code (v, q);
  // When execution advances beyond this function scope, buffer destructor is
  // invoked which relinquishes the ownership of data and copies back the data to
  // the host memory.
  for (int i = 0; i < N; i++) std::cout << v[i] << " ";</pre>
  return 0;
}
```

#### Exercise #4

- 1. Inspect the code in lab1/buffer\_destruction.cpp file showing the synchronization with buffer destruction
- Compile and run this code icpx -fsycl buffer\_destruction.cpp -o buffer\_destruction ./buffer\_destruction
- Add -### option to understand which commands are executed under the hood once you invoke icpx compiler driver: icpx -fsycl buffer\_destruction.cpp -o buffer\_destruction -###

## **Custom Device Selector**

The following code shows custom device selector using your own logic. The selected device prioritizes a GPU device because the integer rating returned is higher than for CPU or other accelerator.

```
Example of custom device selector with specific vendor name
// Return 1 if the vendor name is "Intel" or 0 else.
// 0 does not prevent another device to be picked as a second choice
int custom device selector(const sycl::device& d ) {
  return d.get info<sycl::info::device::vendor>() == "Intel";
3
sycl::device preferred device { custom device selector };
sycl::queue q(preferred device);
Example of custom device selector with specific GPU device name
// Return 1 if device is GPU and name has "Intel"
int custom device selector(const sycl::device& d ) {
  return dev.is gpu() & (dev.get info<info::device::name>().find("Intel") !=
std::string::npos);
3
sycl::device preferred device { custom device selector };
sycl::queue q(preferred device);
Example of custom device selector with priority based on device
```

```
// Highest priority for Xeon device, then any GPU, then any CPU.
int custom_device_selector(const sycl::device& d ) {
    int rating = 0;
```

```
if (d.get_info<info::device::name>().find("Xeon") != std::string::npos)) rating = 3;
else if (d.is_gpu()) rating = 2;
else if (d.is_cpu()) rating = 1;
return rating;
}
sycl::device preferred_device { custom_device_selector };
sycl::queue q(preferred_device);
```

The SYCL code below demonstrates Custom Device Selector:

```
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
class my device selector {
public:
    my device selector(std::string vendorName) : vendorName (vendorName) {};
   int operator() (const device& dev) const {
    int rating = 0;
   //We are querying for the custom device specific to a Vendor and if it is a GPU device we
   //are giving the highest rating as 3.
   //The second preference is given to any GPU device and the third preference is given to
   //CPU device.
   if (dev.is gpu() & (dev.get info<info::device::name>().find(vendorName) !=
std::string::npos))
        rating = 3;
    else if (dev.is gpu()) rating = 2;
    else if (dev.is_cpu()) rating = 1;
    return rating;
    };
private:
    std::string vendorName ;
};
int main() {
    //pass in the name of the vendor for which the device you want to query
    std::string vendor name = "Intel";
    //std::string vendor name = "AMD";
    //std::string vendor name = "Nvidia";
    my device selector selector (vendor name);
    queue q(selector);
    std::cout << "Device: "</pre>
    << q.get device().get info<info::device::name>() << "\n";</pre>
    return 0;
}
```

#### Exercise #5

- Inspect the code in lab1/custom\_device\_sample.cpp file showing the usage of custom device selector with your own logic.
- Compile and run this code: *icpx -fsycl custom\_device\_sample.cpp -o custom\_device\_sample ./custom\_device\_sample*
- Compile this code in Ahead of Time (AOT) compilation mode by adding -fsycl-targets=intel\_gpu\_pvc compiler option:

```
icpx -fsycl -fsycl-targets=intel_gpu_pvc custom_device_sample.cpp -o custom_device_sample
```

4. Do you observe some additional output during the compilation? Check what is going under the hood by analyzing the output of the same command with -### option: icpx -fsycl -fsycl-targets=intel\_gpu\_pvc custom\_device\_sample.cpp -o custom\_device\_sample -###

## **Multi-GPU Selection**

To submit job to a single GPU, we use sycl::device class with sycl::gpu\_selector\_v to find GPU device on the system and then create sycl::queue with this device as shown below:

```
auto gpu = sycl::device(sycl::gpu_selector_v);
sycl::queue q(gpu);
```

To find multiple GPU device in the system, sycl::platform class is used to query all devices in a system, sycl::gpu\_selector\_v is used to filter only GPU devices, the get\_devices() method will create a vector of GPU devices found.

```
auto gpus = sycl::platform(sycl::gpu_selector_v).get_devices();
sycl::queue q_gpu1(gpus[0]);
sycl::queue q_gpu2(gpus[1]);
```

Once we have found all the GPU devices, we create sycl::queue for each GPU device and submit job for GPU devices.

The code below shows how to find multiple GPU devices on a system and submit different kernels to different GPU devices

```
// Get all GPU device in platform
 auto gpus = sycl::platform(sycl::gpu selector v).get devices();
 // create a vector for queue
 std::vector<sycl::queue> q;
 for (auto &gpu : gpus) {
   // create queue for each device and add to vector
    q.push back(queue(gpu));
 }
 // Submit kernels to multiple GPUs
 if (gpus.size() >= 2) {
   q[0].parallel for(N, [=](auto i){
     //...
    });
    q[1].parallel for(N, [=](auto i){
     //...
    });
 }
```

# **Code Sample: Complex Number Multiplication**

The following is the definition of a custom class type that represents complex numbers:

- The file Complex.hpp defines the Complex2 class.
- The Complex2 Class got two member variables "real" and "imag" of type int.
- The Complex2 class got a member function for performing complex number multiplication. The function complex\_mul returns the object of type Complex2 performing the multiplication of two complex numbers.
- We are going to call complex\_mul function from our SYCL code.

#### Exercise #6

- Inspect the code in lab1/complex\_mult.cpp file
- Compile using AOT compilation and run this code icpx -fsycl -fsycl-targets=intel\_gpu\_pvc complex\_mult.cpp -o complex\_mult ./ complex\_mult
- What happens if you try to run on CPU device:

ONEAPI\_DEVICE\_SELECTOR=opencl:cpu ./complex\_mult

4. It is possible to pass additional options to the device compiler via -Xsycl-target-backend option, e.g. icpx -fsycl -fsycl-targets=intel\_gpu\_pvc -Xsycl-target-backend "-options -ze-intel-enable-auto-large-GRFmode" complex\_mult.cpp -o complex\_mult Find more details on general-purpose register (GRF) modes available in Intel® Data Center GPU Max Series here.

## Exercise #7

Complete the coding exercise using SYCL Buffer and Accessor concepts.

1. Complete the code in lab1/vector\_add.cpp file by writing the missing code (look for comments)

- vector1 is initialized on host
- The kernel code increments the vector1 by 1.
- Create a new second vector2 and initialize to value 20.
- Create sycl buffers for the above second vector
- In the kernel code, create a second accessor for the second vector buffer
- Modify the vector increment to vector add, by adding vector2 to vector1
- Compile and run the code

icpx -fsycl vector\_add.cpp -o vector\_add

<mark>./vector\_add</mark>

Note that the solution is available in the source file vector\_add\_solution.cpp

# Summary

In this module you learned:

- The fundamental SYCL Classes
- How to select the device to offload to kernel workloads
- How to write a SYCL program using Buffers, Accessors, Command Group handler, and kernel
- How to use the Host accessors and Buffer destruction to do the synchronization