**Data Parallel C++ (DPC++)** = **oneAPI's implementation of SYCL. Based on modern C++ productivity benefits** and **familiar constructs** | Incorporates the **SYCL** standard for data parallelism and heterogeneous programming.

**SYCL (pronounced** sickle) = **single source** where **host code** and **heterogeneous accelerator kernels** can be **mixed in same source files**. 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** for ***data management***, and **parallel\_for** for ***parallelism*** to **direct which parts of the computation and data should be offloaded**.

**device class = the capabilities of the accelerators in a system utilizing Intel® oneAPI Toolkits. Contains members for querying information about the device (useful for SYCL programs where multiple devices are created:)**

* 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";

The **device\_selector** class = **enables** 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, cpu\_selector, gpu\_selector…**) and a derived device\_selector

default\_selector selector;

*// host\_selector selector;* *// cpu\_selector selector; // gpu\_selector selector;*

queue q(selector);

std:cout **<<** "Device: " **<<** q.get\_device().get\_info**<**info::device::name**>**()**<<** "\n";

The SYCL code below shows different device selectors: Inspect code, there are no modifications necessary:

#include <CL/sycl.hpp>

using namespace cl::sycl;

int main() {

//# Create a device queue with device selector

gpu\_selector selector;

//cpu\_selector selector;

//default\_selector selector;

//host\_selector selector;

queue q(selector);

//# Print the device name

std::cout << "Device: " << q.get\_device().get\_info<info::device::name>() << "\n";

return 0;

}

**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 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]);**

**});**

**});**

A picture containing text, sky, outdoor, sign

Description automatically generated**Work is submitted to queues** and **each queue is associated with exactly one device** (e.g., a specific GPU 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 system**s.

|  |  |
| --- | --- |
| Command | Target Device |
| queue() | Create queue targeting any device: |
| queue(cpu\_selector{});  queue(gpu\_selector{});  queue(INTEL::fpga\_selector{}); queue(accelerator\_selector{});  queue(host\_selector{}); | Create queue targeting a pre-configured classis of devices: |
| class custom\_selector : public device\_selector {int operator()(…… // Any logic you want! … queue(custom\_selector{}); | Create queue targeting specific device (custom criteria): |

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++**

A picture containing graphical user interface

Description automatically generatedA screenshot of a computer

Description automatically generated with low confidence**Parallel Kernels**



**Basic Parallel Kernels**

Exposed via range, id, and item classes.

**Range class** = used to **describe the iteration space** of parallel execution

Text

Description automatically generated with medium confidence**ID class** = used to **index an individual instance** **of a kernel** in a parallel execution



Text

Description automatically generatedIf we need the **range** value in our kernel, then we can use an **ITEM class** instead of an id class. It can be used to query for the **range**. **Item class** = an **individual instance of a kernel function,** **exposes additional functions to query** **properties** of the **execution range**

**ND RANGE KERNELS**

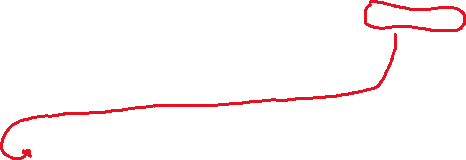
Basic Parallel Kernels **DOES NOT ALLOW PERFORMANCE OPTIMIZATION AT HARDWARE LEVEL**. ND-Range kernel is another way to expresses parallelism which **enable low level performance tuning** by **providing access to local memory and mapping executions to compute units on hardware**. The **entire iteration 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*** = a **grouped execution range using global execution** **range** and **the local execution range of each work-group**.

Diagram

Description automatically generated***nd\_item class*** = an **individual instance of a kernel function** and **allows to query for work-group range and index**.



Buffer Model:

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

**SYCL Code Anatomy**

Programs which utilize oneAPI **require the include of cl/sycl.hpp**. It **is recommended to employ the namespace statement** to **save typing repeated references** into the cl::sycl *namespace*.

#include <CL/sycl.hpp>

using namespace cl::sycl;

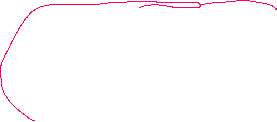
**SYCL PROGRAMS ARE STANDARD C++**. The **program** is **invoked on the host computer**, and **offloads computation to the accelerator**. A programmer uses SYCL’s queue, buffer, device, and kernel abstractions to direct which parts of the computation and data should be offloaded.

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**. ***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**.

Graphical user interface, text, application

Description automatically generatedIn 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**.



Diagram

Description automatically generated**Implicit dependency with Accessors**



* Accessors create **data dependencies** in the **SYCL graph** that **order kernel executions**.
* If **two kernels use the same buffer**, the **second kernel needs** to **wait for the completion** **of the first kernel to avoid race conditions**.

**Host Accessors =** 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.

**Synchronization: Host Accessor = 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**.

***Synchronization: Buffer Destruction***

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**.

**Custom Device Selector**

The **selected device prioritizes a GPU device** because **the integer rating returned is higher than for CPU or other accelerator.**