**[CUDA And SYCK – Programming Model Comparison](https://www.intel.com/content/www/us/en/docs/dpcpp-compatibility-tool/developer-guide-reference/2023-2/cuda-and-sycl-programming-model-comparison.html)**

**Execution Model**

**Kernel Function**

In CUDA, a kernel function is defined using the **\_\_global\_\_** declaration specifier and is executed concurrently across multiple threads on the GPU. Functions that are called by a CUDA kernel must be qualified with the **\_\_device\_\_** specifier. The **\_\_host\_\_** declaration specifier is used to qualify functions that can be called from host code running on the CPU.

In contracts, a SYCL kernel is a function that is executed on SYCL-capable devices, such as CPSs, GPUs, or FPGAs. These kernels are launched from host code and are executed concurrently on SYCL devices. Unlike CUDA, SYCL kernel functions do not require special declaration specifiers and are defined using standard C++ syntax.

The following table shows CUDA and SYCL equivalencies for defining kernel functions:

A screenshot of a computer program

Description automatically generated

If the macro **\_\_CUDA\_ARCH\_\_** is used to differentiate the code path in a CUDA **\_\_host\_\_** **\_\_device\_\_** function, you can use macro **\_\_SYCL\_DEVICE\_ONLY\_\_** in SYCL to achieve similar functionality.

The following table shows CUDA and SYCL equivalencies specifying device functions:

A screenshot of a computer program

Description automatically generated

**Execution Hierarchy**

In both CUDA and SYCL programming models, the kernel execution instances are organized hierarchically to exploit parallelism effectively. In CUDA, these instances are called threads; in SYCL, they are called work-items. CUDA threads can be organized into blocks, which in turn can be organized into grids. SYCL work-items can be organized into work-groups, which in turn can be organized into ND-ranges.

From a hardware perspective, when a CUDA kernel is executed on a GPU, the Streaming Multiprocessor (SMs) create, manage, schedule, and execute threads in groups of 32 threads, known as warps. In comparison, in SYCL, a sub-group represents a collection of related work-items within a work-groups that execute concurrently.

To migrate CUDA code to SYCL code, the CUDA execution hierarchy can be mapped to the SYCL hierarchy as shown in the following table:

A screenshot of a computer

Description automatically generated

**Thread Indexing**

As previously discussed, CUDA threads and SYCL work-items are organized in a hierarchical manner.

CUDA contains built-in variables to support threads:

* **Thread ID**: threadIDx.x/y/z
* **Block ID**: blockIdx.x/y/z
* **Block dimension**: blockDim.x/y/z
* **Grid dimension**: gridDim.x/y/z

SYCL contains equivalent built-in variables:

* **Thread ID**: sycl::nd\_item.get\_local\_id(0/1/2)
* **Work-group ID**: sycl::nd\_item.get\_group(0/1/2)
* **Work-group dimension**: sycl::nd\_item.get\_local\_range().get(0/1/2)
* **NR-range dimensions**: sycl::nd\_item.get\_group\_range(0/1/2)

According to the execution space index linearization sections of the [**CUDA C++ Programming Guide**](https://docs.nvidia.com/cuda/cuda-c-programming-guide/) and the [**SYCL 2020 specification**](https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html), given a block or work-group with shape (dx, dy, dz) and an element with id (x, y, z), then the index **is x + y \* Dx + z \* Dx \* Dy** for **CUDA**, and **z + y \* Dz + x \* Dz \* Dy** for **SYCL**. This discrepancy is partly due to SYCL’s better alignment with C++’s multidimensional indexing. Due to the difference in index computation methods, the CUDA execution space right-most dimension z is mapped to the SYCL left-most dimension x, as shown in the following table:

A screenshot of a computer program

Description automatically generated

**Kernel Lauch**

CUDA uses the **<<< … >>>** execution configuration syntax and the **dim3** type to specify the dimensions and sizes of grids and blocks. The function call operator, in conjunction with the execution configuration syntax, is used to submit a kernel function to a stream for execution. When a kernel is submitted, an index space is defined, with each thread and block receiving a unique thread ID and block ID. These IDs are used to compute the index within the index space and are accessible within the kernel through built-in variables.

SYCL uses the **parallel\_for** member function of **sycl::queue** and **sycl::range** to specify the dimensions and sizes of ND-ranges and work-groups. Users can also apply the SYCL kernel attribute **[[sycl::reqd\_sub\_group\_size(dim)]]** to indicate that a kernel must be compiled and executed with the specified sub-group size. Each device supports only specific sub-group sizes, as defined by **info::device::sub\_group\_sizes**.

The following table shows the original CUDA code for a kernel launch example migrated to SYCL:

|  |  |
| --- | --- |
| **Original CUDA Code** | **Migrated SYCL Code** |
| \_\_global\_\_ void foo() {  int a = threadIdx.x;  }  int main() {  dim3 size\_1(100, 200, 300);  dim3 size\_2(5, 10, 20);  foo<<<size\_1, size\_2>>>();  } | void foo(sycl::nd\_item<3> item) {  int a = item.get\_local\_id(2);  }  int main() {  sycl::queue q;  sycl::range<3> size\_1(300, 200, 100);  sycl::range<3> size\_2(20, 10, 5);  q.parallel\_for(  sycl::nd\_range<3>(size\_1 \* size\_2, size\_2),  [=](sycl::nd\_item<3> item) [[sycl::reqd\_sub\_group\_size(32)]] {  foo(item);  });  } |

Make a note of the following details in the migrated SYCL code:

* In the constructor of **sycl::nd\_range**, the first parameter **global size** is a work-item instead of a work-group. Thus, **global size** should be the product of size\_1 and size\_2 to align with CUDA.
* The Thread Indexing section describes that the CUDA execution space right-most dimension z should be mapped to the SYCL left-most dimension x. Thus, in this example, the dimension size needs to be reversed.

**Memory Model**

**Shared Memory**