Execution Model

Execution model

In this chapter, we describe both CUDA and SYCL execution models, and we highlight their similarities and differences as we go. This section aims to provide an insight on how to write efficient code in terms of instruction throughput. To conclude, we include a section mapping different CUDA concepts to SYCL/OpenCL ones.

CUDA Execution Model

The CUDA execution model exposes an abstract view of the NVIDIA GPU parallel architecture. Each generation of NVIDIA GPU device features some architectural differences, but from the CUDA programming point of view, all devices preserve the same fundamental concepts. In this section we will explain the main CUDA architectural features leveraged for CUDA programming.

CUDA SM

An NVIDIA GPU is built around a set of scalable streaming multiprocessors called an SM. An SM in a GPU is responsible for concurrent execution of groups of threads. When a group of threads is allocated to one SM, they remain there until the end of their life. Each SM is composed of a set of cores, shared memory (L1 cache), registers, load/store unit, and a scheduler unit.

The following image represents an abstract view of the NVIDIA GPU architecture.
CUDA thread hierarchy

The CUDA thread hierarchy is composed of a grid of thread blocks.

- **Thread block**: A thread block is a set of concurrently executing threads that reside on the same SM; share the resources of that SM, and cooperate among themselves using different hardware mechanisms. Each thread block has a block ID within its grid. A thread block can be one, two, or three dimensional.

- **Grid**: A grid is an array of thread blocks launched by a kernel, that read inputs from global memory; write results to global memory, and synchronize dependency among nested kernel calls. A grid will be described by a user and can be one, two, or three dimensional.

The following image represents an abstract view of the CUDA thread hierarchy.
In CUDA, each group of 32 consecutive threads is called a **warp**. A Warp is the primary unit of execution in an SM. Once a thread block is allocated to an SM, it will be further divided into a set of warps for execution. There is always a discrete number of warps per thread block. A thread will never be split between two warps. Each SM contains a warp scheduler that is responsible for scheduling the warps to the underlying cores on the SM. This hardware scheduler allocates more warps than the available cores on a GPU in order to minimize the effects of instruction latency. The following figure represents the relationship between the logical view and hardware view of a thread block, a warp and its mapping to an SM.
Synchronization

Barrier synchronization is used to synchronize the states of threads sharing the same resources. In CUDA, there are two synchronization levels:

- **System Level**: Blocks the calling user thread until all the work on the device is completed.

```c
cudaDeviceSynchronize()
```

- **Block Level**: Wait for all the threads in a thread block to reach the same synchronization point.

```c
__syncthreads()
```

CUDA parallel programming model

The CUDA Programming model design is based on the following assumptions:

- The CUDA device kernel code executes on a physically separate device to that running the host code.

- Both the host and the device maintain their own separate memory spaces in DRAM. The former is called host memory and the latter is called device memory. Therefore, device memory allocation/deallocation and data transfer between host and device memory are explicit.

The CUDA architecture applies the single-instruction multiple-thread (SIMT) parallelism model. By extending the C++ language, CUDA C++ allows the developer to define device kernel in the form of C++ functions. In CUDA, a kernel is annotated by adding the `__global__` declaration specifier in front of the kernel function. Following SIMT execution model, the generated kernel will be Executed $N$ times by $N$ different number of threads simultaneously on different regions of data. The total number of CUDA threads that will execute a kernel is defined as the number of threads per block and the number of blocks per grid. This is defined during the CUDA kernel dispatch using the triple chevron syntax: `<<<Grid, Thread Block>>>`. 

Threads executing the kernel in the CUDA device are distributed conceptually following the CUDA
thread hierarchy. Each thread will execute the kernel code based on its position in the thread block and the position of the thread block in the grid. The local position of a thread in the thread block is accessible via threadIdx, which is a vector of 3 components: x, y, z. The total number of threads in a thread block is represented by blockDim, which is a vector of 3 components: x, y, z. The local position of a thread block in a grid is represented by blockIdx, which is a vector of 3 components: x, y, z. The total number of thread blocks per grid is specified by gridDim which is a vector of 3 components: x, y, z.

In the following example, a device kernel called custom_kernel, is dispatched for 1 thread block and N threads. The size of A and B arrays is N. Independently, each element i of array A will be read by thread i, and its value will be written in B[i].

```cpp
// Kernel definition
__global__ void custom_kernel(float* A, float* B)
{
    // kernel thread Id within the thread block
    int i = threadIdx.x;
    A[i] = B[i];
}

int main()
{
    ...
    // Kernel dispatch for 1 thread block and N threads
    custom_kernel<<<1, N>>>(A, B);
    ...
}
```

Dynamic parallelism

In general, a GPU kernel is launched from a host thread. However, in CUDA it is possible to launch a kernel from within an existing kernel running on the GPU. This is called dynamic parallelism. The same syntax used to dispatch kernels on the host can be used to dispatch kernels from the GPU. The launched kernel is called child and the launcher kernel is called the parent thread. The parent kernel will not be completed until all its child kernels complete. When a parent kernel launches a child kernel, the execution of the child kernel is not guaranteed to begin until the parent kernel explicitly synchronizes on the child. Both parent and child kernel share the same global and constant memory, however each of them will have their own local and shared memory.

SYCL Execution Model

In SYCL, work is described as a command group (CG). A CG encapsulates a kernel with its dependencies, and it is processed as a single entity atomically by the SYCL runtime once submitted to a queue. A SYCL queue is an object associated with a single device on a given context. The SYCL scheduler can handle the execution of multiple command groups in different queues, while keeping the memory consistency by analyzing the dependencies of each command group and keeping track of which memory objects are being accessed.

Command groups are submitted to a SYCL queue via the submit function. When a command group is submitted to a queue, the SYCL runtime will then analyze the command group for data dependencies,
and then schedule its execution based on those dependencies. This will then result in a number of
commands including enqueuing kernel execution and enqueuing data copies in order to make the
data dependencies available in the memory region requested, and to execute the kernel function.

SYCL thread hierarchy

The SYCL thread hierarchy is composed of a 1, 2 or 3 dimensional grid of work items, called an ND-
range. These work items are composed of equally sized groups of threads of the same dimensionality
called work groups.

- **Work item**: A work item is a single thread within the thread hierarchy.

- **Work group**: A work group is a 1, 2 or 3 dimensional set of threads within the thread hierarchy.
  In SYCL, the only synchronization that is possible is across all work items within a work group
  using barriers.

- **Nd range**: An nd range is the representation of the thread hierarchy used by the SYCL runtime
  when executing work. An nd range is composed of three components:- the global range; the local
  range, and the number of work groups. The global range is the total number of work items within
  the thread hierarchy in each dimension; the local range is the number of work items in each work
  group in the thread hierarchy, and the number of work groups is the total number of work groups
  in the thread hierarchy, which can be derived from the global and local ranges.

Synchronization

SYCL provides synchronization points which can be called within a kernel function:

- The nd_item and group objects provide the mem_fence member function which inserts a
  memory fence on global memory access or local memory access across all work items within a
  work group.

- The nd_item and group objects provide the barrier member function which inserts a memory
  fence on global memory access or local memory access across all work items within a work
  group, and also blocks execution of each work item within a work group at that point until all
  work items in that work group have reached that point. SYCL does not provide any memory
  fences or barriers across the entire kernel, only across the work items within a work group.

SYCL parallel programming model

A kernel function is called from one of the following kernel function invoke API entries:

- **single_task**: The kernel function is executed exactly once, equivalent to an ND-range of \( \{0,0,0\} \).
  For example:

```c
h.single_task([]{ a[0] = 1.0f });
```
parallel_for: Invokes the kernel function ND-range times passing thread identification objects as parameters. Different identification objects have different properties. For example:

```
h.parallel_for(range, [](id<1> i) { a[i] = b[i] });
```

Note that the parallel_for API takes a C++ callable object by value (the lambda in the example above), in which the operator () is overloaded to take an id object as parameter. This enables the simple interface for using SYCL kernels, where no barriers or local memory is available. If local memory is required, or work group size is specified manually, then the nd_range object must be used as first parameter. This enables the programmer to use the nd_item overload inside the kernel. The nd_item object contains all the thread identification methods an information, alongside methods for barriers or work-group operations.

The following example is a SYCL implementation of the custom kernel example represented in the section CUDA parallel programming model. Similar to the CUDA custom kernel example, the kernel code reads the buffer B and writes it to buffer A.

```cpp
int main()
{
    ...
    // create SYCL queue
    cl::sycl::queue q;
    // construct BuffA and BuffB from A and B pointer
    cl::sycl::buffer<float, 1> buffA(A, cl::sycl::range<1>{N});
    cl::sycl::buffer<float, 1> buffB(B, cl::sycl::range<1>{N});
    q.submit([&](cl::sycl::handler& cgh){
        auto accA= buffA.get_access<cl::sycl::access::mode::write>(cgh);
        auto accB = buffB.get_access<cl::sycl::access::mode::read>(cgh);
        // Kernel dispatch for N threads
        cgh.parallel_for<class custom_kernel>(cl::sycl::range{N},
            [](cl::sycl::item<1> item_id){
                int i= item_id.get(0);
                accA[i]= accB[i];
            });
    });
    ...
}
```

The example above uses a lambda expression to define the command group. It is submitted to a SYCL queue (q) via the submit method. This command group contains the definition of accessor objects that define a series of input and output dependencies (accA and accB). It executes a kernel API function parallel_for to perform the device dispatch of a kernel named custom_kernel on the device. The kernel functor is the lambda expression passed as the parallel_for. Also, the total number of threads is passed as , cl::sycl::range<1>{N}, a parameter of parallel_for.

Similarities and Differences

Whilst CUDA has been designed for NVIDIA GPUs, and exposes low-level hardware details to the user,
SYCL (and OpenCL) have been designed to be independent of the underlying architectures. This causes some differences in how the programming model exposes different aspects of the hardware. In this section, we describe the similarities and differences between the CUDA and SYCL programming models.

Device Discovery and platform configuration

SYCL is designed to support multiple platforms, contexts and devices. SYCL applications need to perform device discovery at runtime if they want to be generic. On CUDA, device discovery is not necessary since the platform configuration is already known.

For simplicity, it is possible to rely on the default SYCL implementation behavior by creating a SYCL queue with the default parameters, which will automatically select the device it considers the best for running the application.

In addition, it is worth noting that SYCL always offers a host device, where command groups (and kernels) are executed natively on the host without interacting with any device or OpenCL implementation.

Custom platform selection is enabled via the device selector class. For example, a device selector could look for a device of a particular vendor supporting double floating point precision. The device selector can be passed to a queue, a context, a device or a platform.

Explicit vs Implicit copy operations

In CUDA, developers are responsible of making data available for kernels to run on the device. This can be achieved either by using explicit copy operations before enqueuing, or by using memory allocation mechanisms that allow direct access from the GPU (by, for example, triggering a page fault directly from the device). Regardless of how this happens, the user is responsible to issue the instructions in the correct order to guarantee the correct execution order of the different kernels. This is particularly important when using multiple streams in different devices.

In SYCL, developers describe the dependencies for each kernel in the command group, and it is the SYCL runtime scheduler that determines the execution order based on the dependency information. The SYCL runtime is responsible for issuing the memory copy or update operations in the correct order. This is called dataflow execution, since the execution order of the kernels is led by the flow of the input data.

Nomenclature Mapping

Thread block vs Work-group

In CUDA, a group of threads is named a thread block or simply a block. This is equivalent to the OpenCL/SYCL concept of work-group. Both block and work-group can access the same level of the hierarchy, and expose similar synchronization operations.

Note that in CUDA, some operations enable the communication across multiple threads in the same group simply using registers. These operations are not available in SYCL 1.2.1, since it relies on the underlying hardware model of OpenCL 1.2.
A CUDA Stream Multiprocessor (SM) represents hardware processors composed of multiple execution cores. The SMs are equivalent to OpenCL/SYCL Compute Units, and each core on the SM is equivalent to a Processing Element (PE). However, the processing elements in OpenCL can represent other hardware on non-GPU architectures. This is implementation-specific, but allows for a conceptual equivalence between the two.

In CUDA, as described in the section CUDA parallel programming model, kernels are scheduled as a grid of thread blocks that execute serially or in parallel. Each thread block is typically mapped into a number of warps, and some developers make assumptions on the size of a thread block to ensure a predefined number of warps.

This may be beneficial for GPU-based architectures, where grouping threads that are executing the same set of instructions concurrently enables efficient SIMD execution. In addition, scheduling warps in a given order may improve locality and allow it to hide architectural latencies.

On non-GPU architectures usually there is no concept of a warp, and execution is not necessarily SIMT. Programs optimized for a given warp size will not execute optimally on non-GPU hardware. In particular, algorithms written around the CUDA assumption of a 32-bit warp size may have a performance impact when using non CUDA architectures.

In SYCL, a kernel is scheduled as an nd-range that contains a number of work-groups. Each work-group contains a number of work-items, determined by the size of the work-group. When using a SYCL implementation capable of running on NVIDIA GPU-hardware, defining the size of the work-group to be a multiple of the warp size will have a similar effect to that of setting the size of the thread-block.

Other GPU vendors may have different architectural features or sizes, such as AMD GPUs using the concept of wave-front instead of warp, with different properties. On non-GPU hardware, the size of the work-group can be set to a value that reflects the number of concurrent streams of instructions that can run simultaneously on a given processing element.

In SYCL, work-group size can be left empty, which the underlying OpenCL implementation can then set to the optimal value according to an internal heuristic. Additionally, SYCL does provide a mechanism to retrieve a multiple of the work group size that is preferred [@munshi2009opencl]. The value is a suggestion from the SYCL runtime as to the optimal work group size multiple for a particular kernel executing on a particular device. SYCL kernel code cannot make any assumptions about how individual work items are executed based on the this value.

When developers are writing optimized code for a given GPU architecture, they can consult the SYCL or OpenCL implementation notes to see how the warp (or wave-front) concept is mapped on work group sizes. Typically, choosing a work group size that is a multiple of the preferred one will be enough. Note any assumption of warp execution is not performance portable (although developers can select different kernels depending on the underlying architecture). The following code illustrates how to get the preferred work-group size by querying a SYCL kernel.

```cpp
auto preferredWorkGroupSizeMultiple = kernel.get_work_group_info<
    cl::sycl::info::kernel_work_group::preferred_work_group_size_multiple>();
queue.submit([&](cl::sycl::handler &cgh){
    cgh.parallel_for(kernel,
    cl::sycl::range<1>(preferredWorkGroupSizeMultiple),
    [=](cl::sycl::id<1> idx){
        /* kernel code */
    });
```
Performance portability in SYCL can be achieved by either adjustment of work group sizes (equivalent to adjusting the thread block size in CUDA) or algorithmic selection at compile time or runtime.

It is important to note that many algorithms in the literature are focused on optimization of GPU pipelines, and in other architectures they may not be efficient.

Compute Unit Indexing

In CUDA, the identification of each thread within a grid is possible using different built-in variables, such as threadIdx or blockDim. These built-in variables do not exist with SYCL. Different kernel types are strongly typed, and each of them has its own object defining the different methods. The following code snippet illustrates the usage of the threadIdx in CUDA:

```c
// Kernel definition
__global__ void custom_kernel(float* A, float* B)
{
    // kernel thread Id within the thread block
    int i = threadIdx.x;
    A[i] = B[i];
}
```

In SYCL, to enable type-safe dispatch of different kernel types, different classes are used to dispatch the different types of kernels. In particular, the item, nd_item, h_item and id classes represent objects that contain the index space information for the different types of kernels.

The following code represents the usage of the cl::sycl::item class in SYCL:

```c
cgh.parallel_for<class custom_kernel>(cl::sycl::range{N},
    [=](cl::sycl::item<1> item_id){
        // the global thread id
        int i= item_id.get(0);
        accA[i]= accB[i];
    });
```

Upon encountering any of the above classes, the device compiler can perform a series of frontend optimizations to replace the objects with the underlying OpenCL-equivalent built-in function.

For details on the equivalence between the CUDA built-in functions and the SYCL objects, see the Quick Reference chapter.