Lab 3
GPU Parallel Programming in CUDA

CS3210 – 2023/24 Semester 1

Learning Outcomes

1. Learn how to compile and execute a CUDA program
2. Learn the different memory types available to a CUDA program
3. Learn how shared memory accesses are performed in CUDA
4. Learn how to vary the number of blocks, threads executed & investigate the effects of changing them

This lab aims to demonstrate the basic concepts in **CUDA** – a parallel computing platform for NVIDIA GPUs. Details on CUDA, GPGPU and architecture will be provided in the relevant sections of this Lab. This lab serves as a precursor to Assignment 2.

Using the SoC Compute Cluster for the GPU part of CS3210

Our `soctf` lab machines do not have discrete NVIDIA GPUs and thus cannot be used for CUDA programming. Instead, we will be using **NUS School of Computing's Compute Cluster** for this portion of the module (Lab 3 and Assignment 2). Note that the CS3210 module team does not directly control the SoC cluster's machines - they are a resource made available to the whole School and centrally managed by SoC IT.

The SoC Compute Cluster has over 45 nodes with GPUs that are shared across *all* SoC modules and research labs. For fairness, these are also managed with Slurm (there's no escaping Slurm!). Therefore, for labs and assignments involving GPU programming, we will **not** be using the CS3210 cluster (i.e., not the `soctf-pdc-*` machines), but instead using the reserved nodes in the SoC cluster via *SoC's own Slurm*.

Please now access our documentation on how to use the SoC Compute Cluster for GPUs: `https://nus-cs3210.github.io/student-guide/soc-gpus/`.

Once you are in the node, you may copy the code folder of this lab to your node with `scp`, or use `wget https://www.comp.nus.edu.sg/~srirams/cs3210/L3_code.zip`. Unlike the lab nodes, the `/home` folder is shared across all SoC Compute Cluster nodes, so you only have to do this once for all nodes to be synchronized.

Setup: Familiarization with SoC Slurm and GPU Nodes

Before we start writing GPGPU programs, let's make sure we can use SoC's GPU nodes properly.

Use the information in our documentation above to complete the following exercises. These exercises are less "hand-hold"-like to ensure that you read the documentation carefully. If the nodes specified in the exercises are not available at that time on SoC Slurm, please use any other node.

Exercise 1

Use `srun` to run the `hostname` command on any `xgpf` node. Did your program run immediately? Which node did you get?

Exercise 2

Use `srun` to run the `hostname` command on a node with a Titan V GPU. Which node did you get?

Exercise 3

Use `srun` to run the `nvidia-smi` command on a GPU with Compute Capability 8.0. What CUDA version is installed on this node?

Exercise 4

Use `srun` to get access to a single Multi-Instance GPU resource and run `nvidia-smi`. Confirm that you succeeded by looking at the MIG devices section of the output.

Part 1: Hello World in CUDA

This section will cover the basics on how to create, compile and execute a CUDA program. Let us do so by using the `hello.cu` program.

What is a CUDA program?

A CUDA program, with the extension `.cu`, is a modified C++ program with sections of CUDA code that are executed on the GPU, called kernels. In CUDA's terminology, the CPU is called the **host** and the GPU is called the **device**.

CPU vs GPU Code

In CUDA, the programmer has to explicitly separate the code that will run on the CPU from the code that runs on the GPU.

Function execution space specifiers

The separation of CPU and GPU code is done on the method/function level. That is to say, a C++ function can be marked as "GPU" or "CPU" code. This is done using **function execution space specifiers** that are placed before the return type, as follows

```
__global__ void hello(char *a, int len)
{
  int tid = threadIdx.x;
    if (tid >= len)
      return;
    a[tid] += 'A' - 'a';
}
```

Here are some function specifiers and their effects:

- `__device__` – will execute on the device. Can only be called from the device.
- `__global__` – will execute on the device. Callable from both the device and host. Usually serves as the entry point to the GPU kernel (CUDA kernel function).
- `__host__` – executes on the host. Can only be called from the host. Alternatively, a function with **no specifiers** is deemed to be `__host__`.

For more information on function specifiers, refer to the CUDA Programming Guide Section B.1

Organizing Threads in CUDA

This section explains how CUDA threads are laid out from the **programmer's perspective**.

Threads, Blocks and Grid

An invocation of a CUDA kernel function (i.e. functions annotated with `__global__`) launches a new grid. The grid is a collection of **thread blocks**. A **thread block** contains a collection of **CUDA threads**. Threads in a block can be laid out in one, two or three dimensions. Similarly, blocks in a grid can be laid out in one, two or three dimensions. Figure 1 shows an example of thread organization.
The programmer has to explicitly specify the number of threads per block and the number of blocks per grid. All blocks within the same grid will have the same number of threads, laid out in the same fashion.

Specifying thread organization in CUDA

To specify the size of the block or grid, we can either:

- Specify an integer denoting the number of threads/blocks. This is only applicable if we want a 1 dimensional block/grid.
- Declare a variable of type `dim3`

`dim3` is a type defined by CUDA that stores the values that require dimensions. We can define a `dim3` variable as follows:

- `dim3 threadsPerBlock(4, 4, 4);` – creates a variable called `threadsPerBlock`. In our use case, the value is a grid/block that is three dimensional, with 4 blocks/threads on each dimension

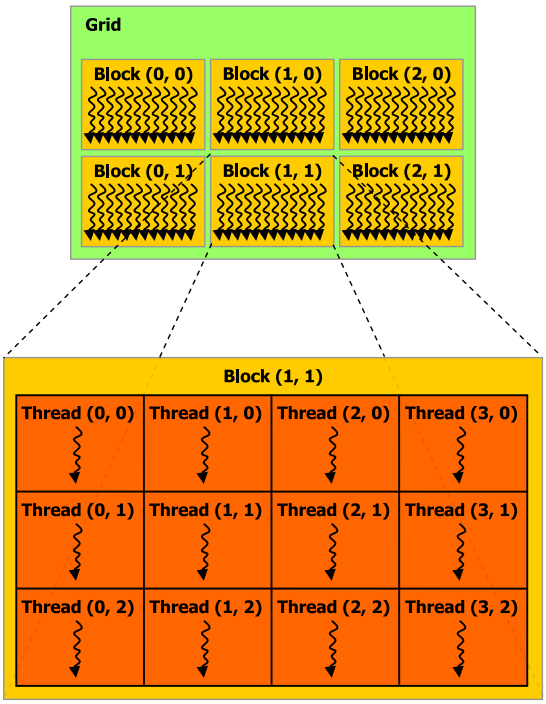

Figure 1: Organization of threads, blocks within grid

- `dim3 blocksPerGrid(8, 4);` – creates a variable `blocksPerGrid`. Two dimensional, with the x dimension having 8 elements, while the y dimension has 4 elements.

Such variables are then passed when the program makes the CUDA kernel function call, covered in the next section.

Invoking a CUDA kernel function

The block and grid dimensions have to be supplied when we call a kernel function. Recall that a grid is launched when a CUDA kernel function is invoked. Thus, this is when CUDA needs to know the grid and block sizes for that particular invocation. The kernel **has to be invoked from a host function** and this is done with a modified syntax as follows:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- `Dg` is of type `dim3` and specifies the dimensions and size of the grid
- `Db` is of type `dim3` and specifies the dimensions and size of each thread block
- `Ns` is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call and addition to statically allocated memory. (**Optional**, defaults to 0)
- `S` is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same thread block where the call is being made. (**Optional**, defaults to 0)

For instance, if we have a kernel function (i.e. annotated with the `__global__` specifier) called

`void convolve(int a, int b)` and we want to invoke it with the following thread dimensions:

- The grid has dimension $3 \times 3 \times 3$
- A block in the grid has dimension 4×3

Then, we write the following code in a **host** function at the point where we want to execute the kernel function:

```
dim3 gridDimensions(3, 3, 3);
dim3 blockDimensions(4, 3);
convolve<<<gridDimensions, blockDimensions>>>(5, 10);
```

Compiling and Executing

A CUDA program (application) is compiled using a proprietary compiler called NVIDIA CUDA Compiler (nvcc). For instance, if we want to compile `hello.cu` to an executable called `my-hello`, we can run the following command in a terminal (**Note that you can only run these commands via Slurm!**):

```
> nvcc -o my-hello hello.cu
```

This will create an executable called `my-hello` in the directory where `nvcc` was executed. To run `my-hello`, we run it as though it is a normal executable as follows:

```
> ./my-hello
```

Specifying Compute Capability

By default, the `nvcc` compiler may not target the latest Compute Capability (CC) for your GPU. To target the right CC, you should set the `-arch` flag as follows:

```
> nvcc -arch native -o my-hello hello.cu
```

Alternatively, you can specify the Compute Capability exactly (e.g., 7.0) as such:

```
> nvcc -arch sm_70 -o my-hello hello.cu
```

More information on nvcc can be found in the CUDA Compiler Driver NVCC section of the CUDA Toolkit Documentation

Exercise 5

Compile and execute `hello.cu` on one of the GPU nodes – we have provided you a **gpu_job.sh** Slurm script that can compile and run your code. You can run it with `sbatch gpu_job.sh`. Inspect `hello.cu` and answer the following questions:

- How many threads are there per block? How many blocks are in the grid?
- How many threads are there in total?
- Draw out a diagram as per Figure 1.

Determining the thread's location in grid/block

Once invoked, the code within the kernel runs on the device. Often computation done by the kernel is based on the thread location within the block and grid. CUDA provides the developer with **built-in variables** that can be called in device code to find the following:

- `gridDim` – the dimensions of the grid as `dim3` (i.e. dimensions in X direction can be found in `gridDim.x`, Y in `gridDim.y` and Z in `gridDim.z`
- `blockIdx` – the block index the thread is running in

  (`blockIdx.x`, `blockIdx.y` & `blockIdx.z`)
- `blockDim` – the dimensions of the block

  (`blockDim.x`, `blockDim.y` & `blockDim.z`)
- `threadIdx` – the index of the thread **within its block**

  (`threadIdx.x`, `threadIdx.y` & `threadIdx.z`)

Exercise 6

Modify `hello.cu` such that it runs on a 3D grid of dimension $2 \times 2 \times 2$, where each block has a dimension of 2×4.

Hint: You will need to change how the `tid` is calculated (in the kernel function `hello`), so that there's no overlapping work done.

Executing CUDA threads – hardware's perspective

Grouping threads into blocks (and blocks into a grid) is a useful abstraction for the programmer. This section explains how these threads, which are organized by the programmer, is executed by the hardware.

Architecture of the GPU

To better understand the execution model, let us go through the architecture of the GPU. In particular, we will be using the VoltaGV100 GPU as an example.

The Volta GV100 GPU is used by the NVIDIA Tesla V100 accelerator. This GPU is composed of six GPU Processing Clusters (GPCs), each with 14 Streaming Multiprocessors (SMs) and memory controllers. With 84 SMs, a full GV100 GPU has a total of 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores, and 336 texture units.

Figure 2 shows the full Volta GV100, while Figure 3 shows the diagram of an SM.

Figure 2: Volta GV100 full GPU with 84 SM units.

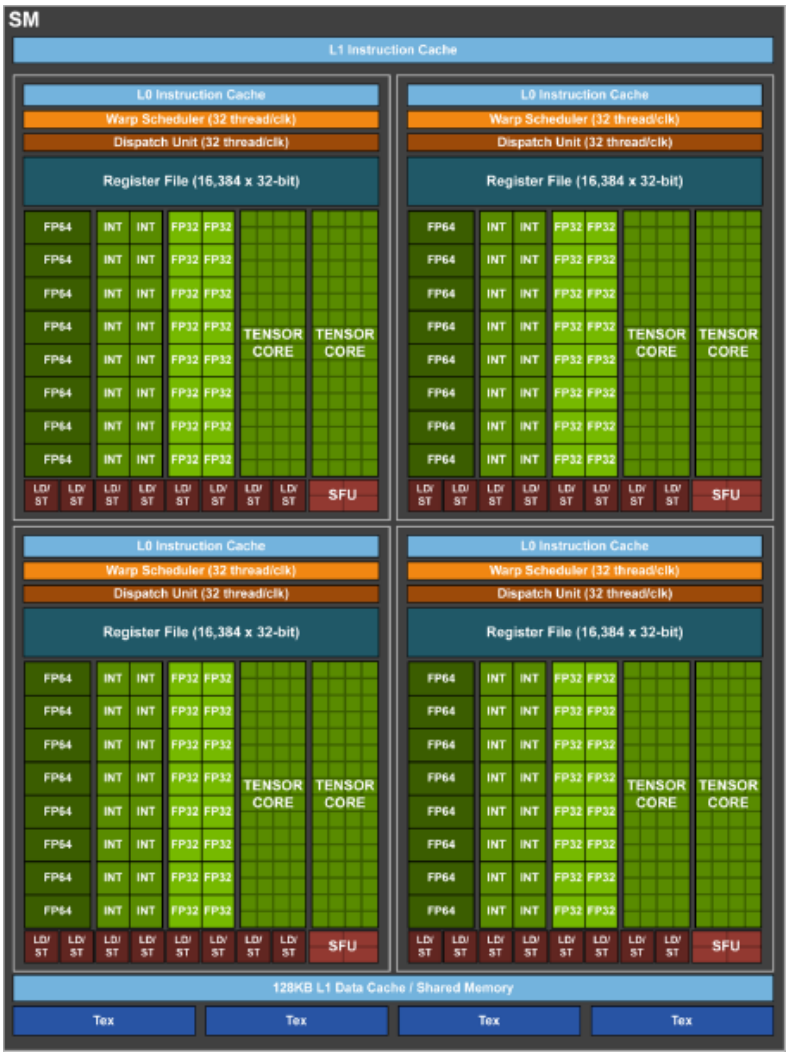

Figure 3: Streaming Mutiprocessor (SM) for Volta GV100.

Each **thread block** is assigned a **Streaming Multiprocessor** by the GPU. Once it is assigned, it does not migrate (i.e. change SMs). The SM will break down the thread block into **groups of 32 threads each** called **warps**. The warps execute in a Single Instruction Multiple Threads (SIMT) fashion, which is similar to an SIMD execution (i.e. lockstep execution).

If the thread block does not divide by 32 evenly, then a warp might be executing with less than 32 threads. However, the warp will be executed **as though there are 32 threads**.

Exercise 7

Compile and execute `slow.cu`. Note that you will have to modify `gpu_job.sh` if you are using it (the lines after the `echo` calls for "Compiling.." and "Running..."). Notice the extremely slow runtime when each block contains 1 thread. Calculate the number of warps launched when running 1024 blocks of 1 thread each, versus 1 block of 1024 threads.

Exercise 8

Compile and execute `printing.cu`. Notice that there is a trend in how the thread IDs are printed out. Notice any correlation with our knowledge on how the GPU executes a CUDA program?

Part 2: CUDA Memory Model

This section presents the CUDA memory model and its usage in a CUDA program.

CUDA Memory Model

CUDA offers the programmer access to various types of memory, that either resides on the host or device. These can be further subdivided based on whether the host can access the memory in question. GPUs can be in a system either as discrete (e.g. most consumer graphics cards) or integrated. Figure 4 shows the memory organization of integrated and discrete GPUs.

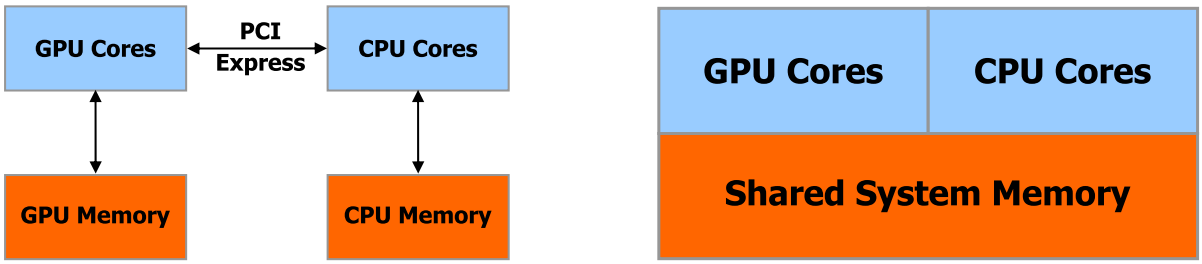

Figure 4: Memory organization in discrete (left) and integrated (right) GPUs

These different types of memory in a GPU is akin to the memory hierarchy of computers and can be visualized in Figure 5.

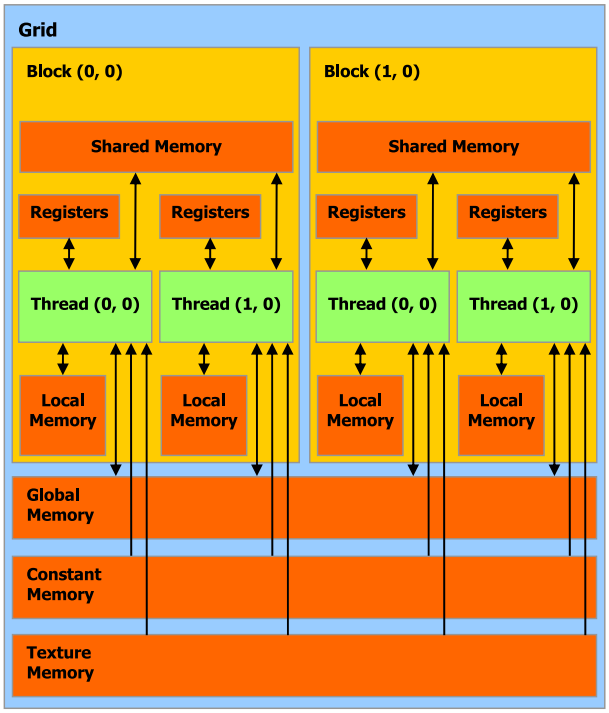

Figure 5: Memory Hierarchy in CUDA (or GPUs in general)

Table 1 provides a summary of the different memory types that are available in CUDA.

Type	Scope	Access type	Speed	CUDA declaration syntax	Explicit sync
Register	thread	RW	fastest	Compiler decides (from local vars)	no
Local	thread	RW	depends*	`float x;`	no
Shared	block	RW	fast	`__shared__ float x;`	yes
Global	program	RW	slow	`__device__ float x;`	yes
Constant	program	R	slow	`__constant__ float x;`	yes
Texture	program	R	slow	`__texture__ float x;`	yes

Table 1: Different memory types in the CUDA/GPU memory model

Note that the declaration syntax provided in Table 1 can only be used with **fixed size types** (i.e. primitives or structs).

*Local memory is actually an abstraction of global memory that is private to the thread, which means its access speed is slower than shared memory.

Arguments to a kernel invocation (e.g. variables `ad` and `len` in `hello<<<1, N>>>(ad, len)` in `hello.cu`) are passed to the device using **constant memory** and are limited to 4KB [1]

For more details on Variable Memory Space Specifiers, refer to the CUDA Programming Guide Section B.2

Next, we will discuss details about selected CUDA memory types and their usage in an application.

Global Memory – Static & Dynamically Allocated

The previous section showed that we can declare a variable in global memory using the `__device__` specifier. From Table 1, we know that the variable has program scope, which means that it can be accessed by both the host and device.

Exercise 9

Compile and run `global_comm.cu`. Observe how the variable `result` can be accessed by both the device and host (albeit indirectly). What's the reason behind this indirect access?

However, there is a limitation when using `__device__` variables. From the example in `global_comm.cu`, we can see that arrays need to have their sizes explicitly specified during compile time. As such, if we only know the size of an array during run-time, the `__device__` variable is not a very efficient and scalable solution. Thus, we need a way to define and allocate memory dynamically.

`cudaMalloc` – `malloc` for CUDA

One of the ways to allocate memory dynamically is through the use of what CUDA calls **linear arrays**. You can think of it like `malloc` but on CUDA. One can declare a **linear array** by invoking `cudaMalloc` in either host or device code, as follows:

Method Signature for `cudaMalloc`

```
cudaError_t cudaMalloc (void** devPtr, size_t size)
```

- `devPtr` – is a pointer to a pointer variable that will receive the memory address of the allocated linear array
- `size` – requested allocation size in **bytes**

Exercise 10

Compile and run `cudaMalloc_comm.cu`. Observe the following:

- The global memory "declaration"
- The host can only access the linear array through `cudaMemcpy`
- The device access to global memory
- Like in normal `malloc`, it is good practice to `free` your dynamically allocated memory. This is done using `cudaFree`.

Unified Memory

We observes in the previous section that global memory, whether static or dynamically allocated, cannot be accessed by the host directly. However, in CUDA SDK 6.0, there is a new component introduced to CUDA called **Unified Memory**. This defines a managed memory space that has a **common memory addressing space**, allowing both CPU and GPU to access them as though it is part of their memory space (especially for CPU).

The only difference between **unified memory** and the global memory declarations described previously, is how we allocate the memory in code. We can do the following to replace our global memory declarations to use **unified memory**:

- Replacing `__device__` with `__managed__`
- Replacing `cudaMalloc` with `cudaMallocManaged`

Exercise 11

Compile and run both `global_comm_unified.cu` and `cudaMalloc_comm_unified.cu` (Note that you may need to tell `nvcc` to target architecture `sm_70` or below). Compare the code with their non-unified counterparts. What difference(s) do you observe?

Shared Memory

Another important memory type in CUDA is **shared memory**. Since this resides only in the device, the memory is accessible to the device faster than global memory. As such, it allows us to store intermediate values during computation. Unlike **local memory**, as the name suggests, **shared memory** is **shared** within the **same thread block**. Thus, a thread residing in a different **thread block** will not see the same values in the same shared memory location.

Exercise 12

Compile and run `shared_mem.cu`. Observe/Ponder on the following:

- Are there any differences between shared and global memory?
- Do the results printed out differ between runs?

Part 3: Synchronization in CUDA

This final section presents synchronization constructs in CUDA.

Atomic Memory Accesses

As with any parallel program, there are times where we want CUDA threads to use a shared memory location, be it shared (within block) or global (within device) memory. However, this may lead to race conditions. Hence, there needs to be a way to ensure **mutual exclusion** when accessing these shared locations.

CUDA provides the programmer with a set of **atomic functions**, each performing a different operation on either a 32-bit or 64-bit memory location. These functions are atomic and this guarantees mutual exclusion.

For details on the CUDA atomic functions available, refer to CUDA Programming Guide Section B.12.

Exercise 13

Compile and run `atomic.cu`. Observe on the following:

- What are the values that are printed out? Are they consistent across different runs?
- How does the code resolve global memory access race condition?

Synchronization Constructs

Another synchronization construct that you may be familiar with is **barriers**. These prevent a group of processes/threads from proceeding past the barrier till all participating processes/threads reached the barrier. CUDA provides a simple barrier construct with the `__syncthreads()` function. Threads in the same block wait on `syncthreads` until all threads of the block have reached `syncthreads`. Furthermore, it guarantees that accesses to global and shared memory made up to this point are visible to all threads. Please note that there is no synchronization among threads from different blocks when you use `syncthreads`.

Exercise 14

Compile and run `synchronise.cu`. Observe/Ponder on the following:

- What is the significance of counter values printed out with/without `syncthreads`
- Why does the values vary when `__syncthreads` is used in a kernel launch containing multiple blocks?
- Why do you think the is_done variable is marked as volatile?

For details on the CUDA synchronization constructs available, refer to CUDA Programming Guide Section B.6.

Bringing it all together

Exercise 15

Challenge - bringing it all together.

In this exercise, we are going to use all of our knowledge so far to optimize a GPU kernel that sums a list of parallel numbers.

Compile and run `sum.cu`.

`sum` uses both the CPU and GPU to sum a large array of N numbers together. In our example, each number in the array is set to 42 for repeatability. Notice that the expected result is 420000000 (42 * N), and how long the CPU and GPU "Atomic Global" routines took to compute this value. Also, notice that the "Atomic Shared" and "Reduce Shared" kernels are currently failing to produce the right output.

Now, open `sum.cu` in your editor, and study its contents. We have implemented the "Atomic Global" kernel for you, but we have also suggested two optimizations that you can implement as `reduceAtomicShared` (for "Atomic Shared") and `reduceShared` (for "Reduce Shared").

Your task: implement `reduceAtomicShared` and `reduceShared` based on the suggestions in the comments and observe the speedup over the previous methods. Understanding these tricks may be useful for the next assignment.

(Note: you do not need to submit anything for this lab)