Objectives

1. Develop simple CUDA programs using a different grid and block dimensions
2. Statically allocate arrays in global GPU memory
3. To write simple CUDA programs using CudaMallocHost() instead of malloc()

**CUDA Execution Model**

The following is taken from the CUDA tutorial [1] (<https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation> ):

‘*The NVIDIA GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity. The threads of a thread block execute concurrently on one SM, and multiple thread blocks can execute concurrently on one SM. As thread blocks terminate, new blocks are launched on the vacated SM.*

*An SM is designed to execute hundreds of threads concurrently. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores, they are issued in order and there is no branch prediction or speculative execution.*

*The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0.’*

The GPUs available in this module (SMB202) are **GTX 1080 Ti** - Pascal Architecture. **They have 28 Streaming Multiprocessors (SMs) and each SM has 128 cores (3584 cores in total)**.

**Memory Hierarchy**

**Global Memory:** Global memory is the GPU’s main memory. It has global scope. All threads can access it. Accessing this memory is slow and thus our implementation must consider this. Coalesced memory accesses reduce the number of global memory accesses.

**Local Memory**: Local memory is everything on the thread’s stack that can’t fit in registers. The scope of local memory is just the thread. **Local memory is part of the global memory (slow)**.

**Shared Memory:** It is a very fast memory located in each SM. Shared memory uses the same hardware as the L1 cache~5ns of latency). Its maximum size is of 48KB, but its size is user configurable (we can reduce the shared memory size and increase the L1 cache size and vice versa). The scope of shared memory is a block of threads. Shared memory consists of 32 banks of width 4 bytes. Element ‘i’ is in bank ‘i % 32’. A bank conflict occurs when 2 threads in a warp access different elements in the same bank. Bank conflicts cause serial memory accesses (poor performance).

**L1 cache**: Each SM has its own L1 cache. It uses the same hardware as the shared memory. Cache lines are of size 128bytes each, and they are aligned. Coalesced memory access improve performance.

**Registers**: This is the fastest memory possible, about 10x faster than shared memory. If we declare a variable inside the kernel, this variable will be stored in a register (if a free register exists). The GPUs we are using in this module have 64k 32bit registers, thus we can compute how many registers are available per thread. A ‘double’ variable uses two 32-bit registers.

**L2 cache**: It is shared among all the SMs. Normally its size is 1Mbyte.

**Constant memory:** It resides in global memory. It is used for constants only. Constants must be set from the host before running kernel. Constant memory is a faster global memory. Constant memory has a special cache.

**Texture memory**: It resides in global memory and has a special cache. It is complicated and will not be studied in this module.

**Instruction cache**: Each SM has an instruction cache but Nvidia keeps the hardware details secret.

**1st example – adding two 1d arrays**

In this example we will study six different CUDA implementations of the following program

*For (i=0; i<N; i++)*

*c[i]=a[i]+b[i];*

**Implementation #1 – Use many blocks of threads where every block contains just one thread (not good practice)**

**Task1**. Study 1d\_grid.cu program. In this implementation there is an 1d grid. Each block in the grid consists of just one thread. Thus, parallelism is achieved just among different blocks. This implementation is shown below. This implementation is not efficient as the amount of parallelism achieved is limited to 65535 threads. We cannot assign more blocks than 65535 in a single dimension (this number is device specific). BlockIdx.x is used as an index as the grid is 1d. Thus, if *VECTOR\_LENGTH is larger than the above value, this program does not work.* The rest of the program is similar to that we studied previous week, so if you have any queries read the previous week’s notes.

*\_\_global\_\_ void addWithBlocks(int\* a, int\* b, int\* c) {*

*/\* the index of the block in the 1D grid along the x-dimension is used to access to the elements of the array \*/*

*if (blockIdx.x < VECTOR\_LENGTH) {*

*c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];*

*}*

*}*

*int main(){*

*…*

dim3 dimGrid(MAX\_NUMBER\_OF\_BLOCKS, 1, 1);

dim3 dimBlock(1, 1, 1);

addWithBlocks << <dimGrid, dimBlock >> > (device\_a, device\_b, device\_c);

….

*}*

**Implementation #2 – Use many threads and just one block (not good practice)**

**Task2**. Study 1d\_block.cu program. In this implementation there is an 1d grid. The grid consists of just one block of threads. Again, the parallelism achieved is limited to the maximum number of threads can be assigned to a block (for this device this value is 1024). ThreadIdx.x is used as an index as each block is 1d.

*\_\_global\_\_ void addWithThreads(int\* a, int\* b, int\* c) {*

*/\* the index of the thread within a block is used to access to the elements of the array \*/*

*if (threadIdx.x < VECTOR\_LENGTH) {*

*c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];*

*}*

*}*

*int main(){*

*...*

*dim3 dimGrid(1, 1, 1);*

*dim3 dimBlock(1024, 1, 1);*

*addWithThreads << <dimGrid, dimBlock >> > (device\_a, device\_b, device\_c);*

*...*

*}*

**Implementation #3 – 1D grid and 1D block**

**Task3**. Study 1d\_grid\_1d\_block.cu program. This is a proper implementation using both multiple blocks and multiple threads per block. This way, we can use a very large number of threads. BlockIdx.x is used to index the blocks and ThreadIdx.x is used to index the thread in a block (Fig.1). This implementation uses 1024 threads per block; this is the maximum number of threads per block in this GPU device. Given that each thread applies just one array addition, we need as many threads as the number of the array elements (alternative implementations exist where each thread executes more than one additions). Thus, the number of blocks is given by Eq.1

*Num.blocks = ceil( VECTOR\_LENGTH / MAX\_NUMBER\_OF\_THREADS ) (1)*

To understand why the ceiling operator is needed, let me give you an example. Consider that *VECTOR\_LENGTH=1025 and MAX\_NUMBER\_OF\_THREADS=1024.* In this case, 2 blocks are needed and not one, as each block consists of 1024 threads. In C programming language, the following operation (*VECTOR\_LENGTH / MAX\_NUMBER\_OF\_THREADS ) will give ‘1’ and not ‘2’, as by default this is an integer operation and the result is always given by ‘floor*(*VECTOR\_LENGTH / MAX\_NUMBER\_OF\_THREADS )’.* So, the ceil() operation is necessary. Given that ceil() operation is computationally expensive, we use Eq.3 instead. Eq.3 comes from Eq.2 and Eq.1. Eq.2 holds for unsigned integer values.

*ceil ( a / b) = (a + b –1) / b (2)*

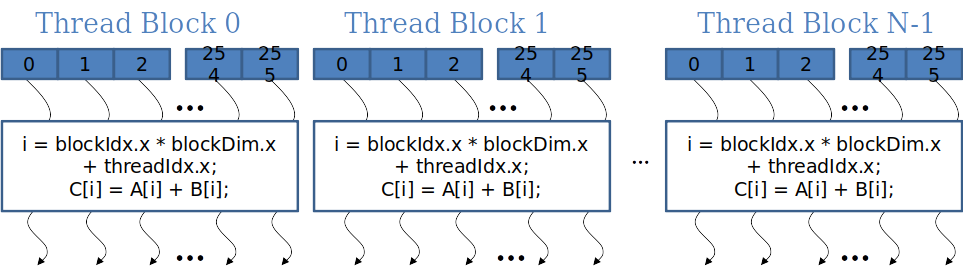


Fig.1. 1d Grid of 1d thread blocks

*\_\_global\_\_ void addWithThreadsAndBlocks(int\* a, int\* b, int\* c) {*

*/\* Global ID of a thread in a 1D grid of 1D block \*/*

*int index = threadIdx.x + blockIdx.x \* blockDim.x;*

*if (index < VECTOR\_LENGTH) {*

*c[index] = a[index] + b[index];*

*}*

*}*

*int main(){*

*…*

*int number\_of\_blocks = (VECTOR\_LENGTH + MAX\_NUMBER\_OF\_THREADS - 1) / MAX\_NUMBER\_OF\_THREADS;*

*/\*the above is equivalent to : number\_of\_blocks = ceil(VECTOR\_LENGTH / MAX\_NUMBER\_OF\_THREADS);*

dim3 dimGrid(number\_of\_blocks, 1, 1);

dim3 dimBlock(MAX\_NUMBER\_OF\_THREADS, 1, 1);

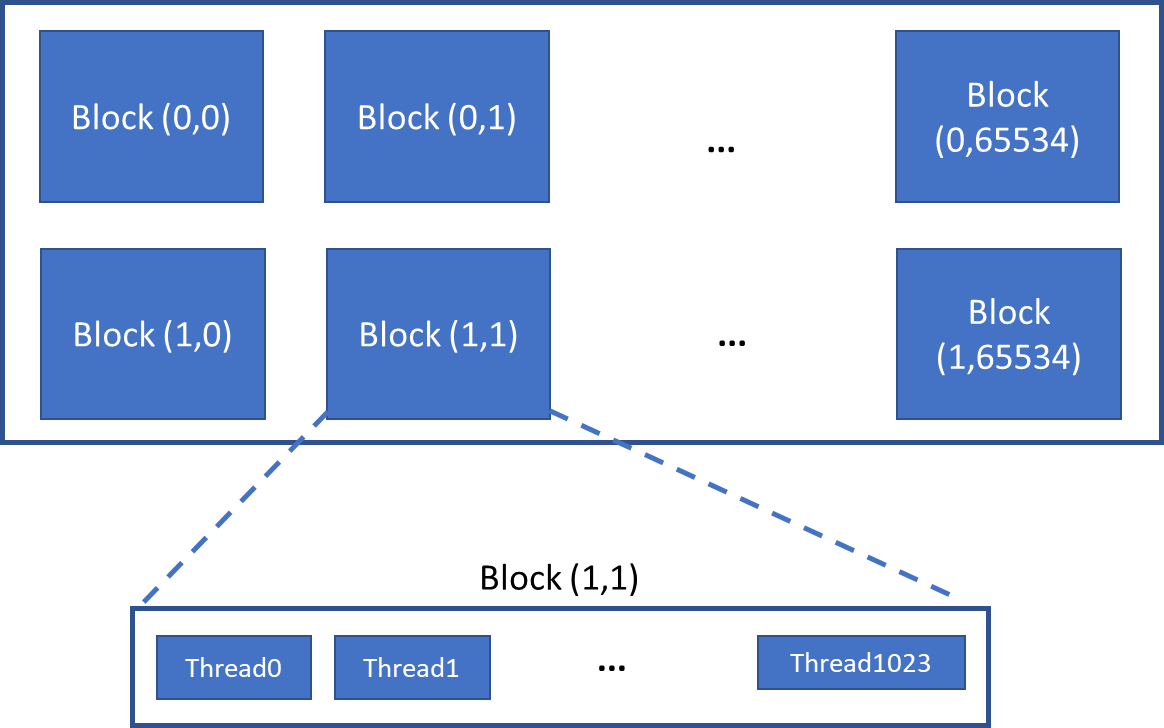
addWithThreadsAndBlocks << <dimGrid, dimBlock >> > (device\_a, device\_b, device\_c);

*...*

*}*

**Implementation #4 – 2D grid and 1D block**

**Task4**. Study 2d\_grid\_1d\_block.cu program. This implementation uses a 2d grid, where each block is 1d (Fig.2). Pay attention to the ‘*block\_id*’ and ‘*thread\_id*' values below.



*Fig.2. 2d grid of 1d blocks*

*\_\_global\_\_ void addWith2DGrid1DBlock(int\* a, int\* b, int\* c) {*

*/\* Global ID of a block within the 2D grid of blocks computed based on the row-major convention. Here the row-major convention is used to pass from the two-dimensional to the one-dimensional representation of the grid \*/*

*int block\_id = blockIdx.x + gridDim.x \* blockIdx.y;*

*/\* Global ID of a thread within the 2D grid of 1D blocks. This id is then used to access to the elements of the vectors \*/*

*int thread\_id = blockDim.x \* block\_id + threadIdx.x;*

*if (thread\_id < VECTOR\_LENGTH) {*

*c[thread\_id] = a[thread\_id] + b[thread\_id];*

*}*

*}*

*int main(){*

*…*

*int number\_of\_blocks = (VECTOR\_LENGTH + MAX\_NUMBER\_OF\_THREADS - 1) / MAX\_NUMBER\_OF\_THREADS; //this is equivalent to the following: number\_of\_blocks = ceil(VECTOR\_LENGTH / MAX\_NUMBER\_OF\_THREADS);*

*int num\_blocks\_y = (number\_of\_blocks + MAX\_NUMBER\_OF\_BLOCKS\_PER\_DIM - 1) / MAX\_NUMBER\_OF\_BLOCKS\_PER\_DIM;//this is equivalent to the following: number\_of\_blocks\_y = ceil(number\_of\_blocks / MAX\_NUMBER\_OF\_BLOCKS\_PER\_DIM);*

*int num\_blocks\_x = (number\_of\_blocks + num\_blocks\_y - 1) / (num\_blocks\_y);//this is equivalent to the following: number\_of\_blocks\_x = ceil(number\_of\_blocks / num\_blocks\_y);*

*dim3 dimBlock(MAX\_NUMBER\_OF\_THREADS, 1, 1);*

*dim3 dimGrid(num\_blocks\_x, num\_blocks\_y, 1);*

*addWith2DGrid1DBlock << <dimGrid, dimBlock >> > (device\_a, device\_b, device\_c);*

*…*

*}*

The number of threads is fixed to 1024 (maximum number). The number of blocks is given by Eq.3. The number of blocks in x and y dimension is given by Eq.4 and Eq.5, respectively. 65535 is the maximum number of blocks in one dimension; this number is hardware specific.

*number\_of\_blocks = ceil (VECTOR\_LENGTH / 1024) (3)*

number\_of\_blocks\_y = ceil (number\_of\_blocks / 65535) (4)

number\_of\_blocks\_x = ceil (number\_of\_blocks / num\_blocks\_y) (5)

**Implementation #5 – Implementation #4 using static memory allocation**

**Task5**. Study ‘*2d\_grid\_1d\_block\_STATIC\_ALLOCATION.cu*’ program. This example uses the above implementation but instead of allocating the device arrays dynamically, they are allocated statically. An array can be allocated in the GPU’s global memory (DDR) by using the following command before main function.

*\_\_device\_\_ int device\_a [VECTOR\_LENGTH];*

Now, instead of using cudaMemcpy(), *cudaMemcpyToSymbol()* and *cudaMemcpyFromSymbol()* are used. Their syntax is very similar.

**Implementation #6 – Implementation #5 using cudaMallocHost() instead of malloc()**

**Task5**. Study ‘*2d\_grid\_1d\_block\_pinned\_memory.cu*’ program. This is the implementation #5 above, but instead of using malloc(), cudaMallocHost() is used. CudaMallocHost() performs much faster than malloc() and thus it is recommended. The memory that has been allocated using CudaMallocHost() must be deallocated using cudaFreeHost().

Why cudaMallocHost() is faster? When we define an array, its memory address is virtual not physical. The operating system is responsible for translating the virtual memory address to physical. The virtual address space is larger than the physical and therefore physical memory might not be enough to run all programs; as a consequence, physical pages in main memory can be swapped to the hard disk to make space for other pages.

cudaMallocHost() allocates memory that is ‘page-locked’ and accessible to the device at all times; the operating system will not allow these pages to be swapped to the disk. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as malloc(). Allocating excessive amounts of memory with cudaMallocHost() may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

**2nd example – adding two 2d arrays**

**Task6**. Try to implement the following program on your own. A solution can be found in ‘matAdd.cu’ file.

*for (i=0; i<N; i++)*

*for (j=0; j<N; j++)*

*C[i][j]=A[i][j] + B[i][j];*

Unlike to the previous examples, this example has two loop indexes, i and j. Given that all the iterations can be executed in parallel (no dependencies exist), we can parallelize both loops. Therefore, we will be needing two global indexes not one. Let us consider the simple case where N=32. We could map the problem to a single 2d block of threads (32x32). The code for that follows

*\_\_global\_\_ void matAdd() {*

*int i = threadIdx.x;*

*int j = threadIdx.y;*

*if (i < 32 && j < 32) {*

*device\_c[i][j] = device\_a[i][j] + device\_b[i][j];*

*}*

*}*

Visualize the 32x32 matrix, where j spans in the x dimension and i spans in the y dimension.

To deal with large input sizes, we need multiple blocks too. To achieve enough parallelism, both i and j must use both multiple blocks and multiple threads. The implementation below uses 1d Grid and 1d blocks. The grid is created with enough blocks to have one thread per matrix element.

*\_\_global\_\_ void matAdd() {*

*int i = blockIdx.x \* blockDim.x + threadIdx.x;*

*int j = blockIdx.y \* blockDim.y + threadIdx.y;*

*if (i < N && j < N) { //this is not needed – for better performance remove it*

*device\_c[i][j] = device\_a[i][j] + device\_b[i][j];*

*}*

*}*

*int main(){*

*…*

*dim3 dimBlock(32, 32, 1);*

*dim3 dimGrid((N + 32 - 1) / 32, (N + 32 - 1) / 32, 1);*

*matAdd << <dimGrid, dimBlock >> > ();*

*…*

*}*

Further reading

1. CUDA Documentation, available at <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model>
2. CUDA Programming guide, available at <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications>
3. Jonathan Hui blog, CUDA Tutorial, available at <https://jhui.github.io/2017/03/06/CUDA/>