



Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

GPU programming module

Week 3 : Introduction to CUDA

Lecture 5 - September 24th 2024

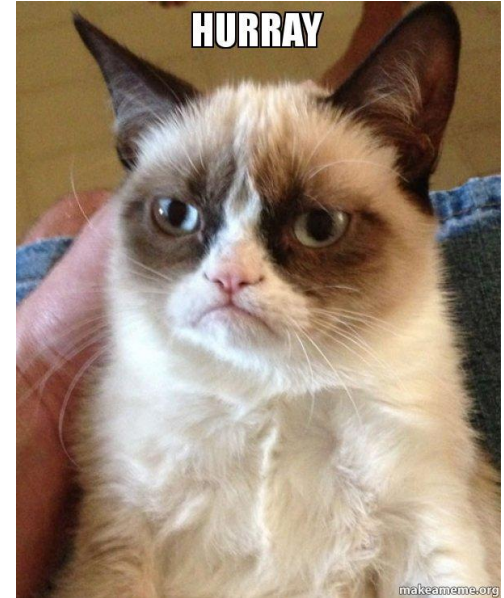
What we learnt in the previous lecture

- Brushed up on C++:
 - Core syntax
 - Variables & Operators
 - Control instructions & Functions
 - Compound data types
 - OOP
 - C++ compilation chain



Today

- Nvidia GPU architecture
- How to check if the system has a GPU & explore its characteristics
- Concept of parallelization
 - Threads / blocks / grid
- CUDA core syntax
- Our first “Hello world” CUDA kernel





NVidia GPU architecture

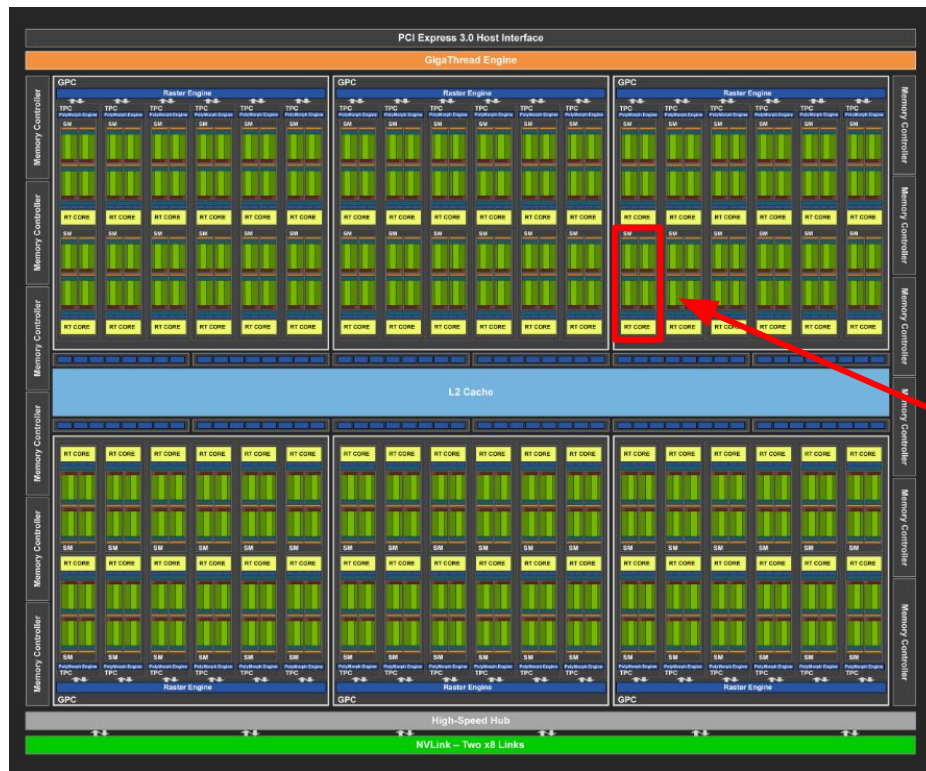


The NVidia GPU architecture



- The GPU architecture is built around a scalable array of **Streaming Multiprocessors (SM)**.
- Each SM in a GPU is designed to support concurrent execution of hundreds of threads

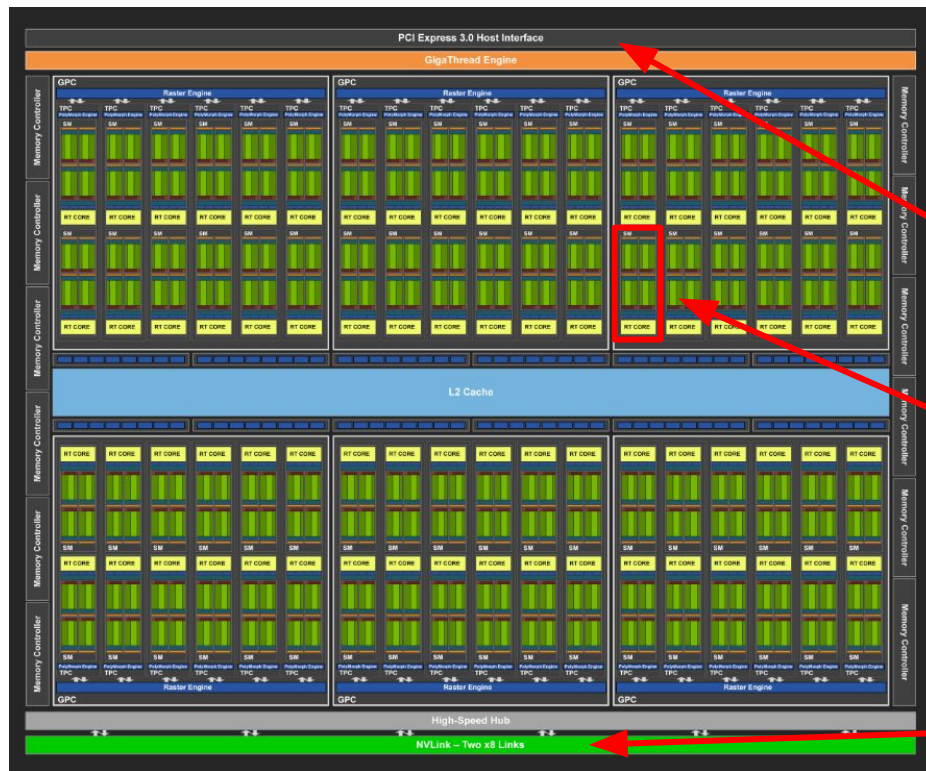
The NVidia GPU architecture



- The GPU architecture is built around a scalable array of **Streaming Multiprocessors (SM)**.
- Each SM in a GPU is designed to support concurrent execution of hundreds of threads

SM

The NVidia GPU architecture



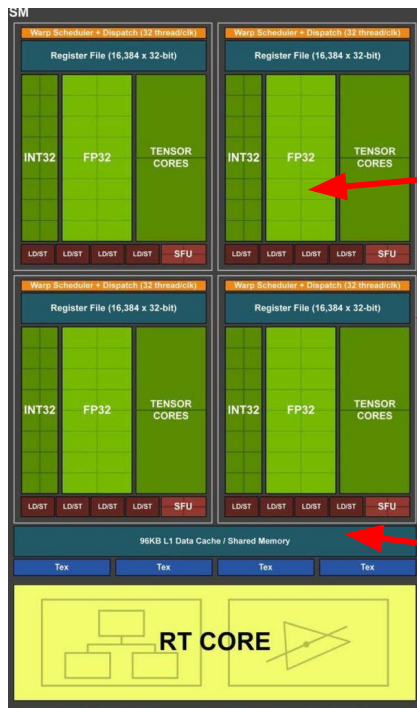
- The GPU architecture is built around a scalable array of **Streaming Multiprocessors (SM)**.
- Each SM in a GPU is designed to support concurrent execution of hundreds of threads

PCIe interconnect: Can be used for connecting GPU to host CPU

SM

NVlink : Can be used to connect to additional GPUs

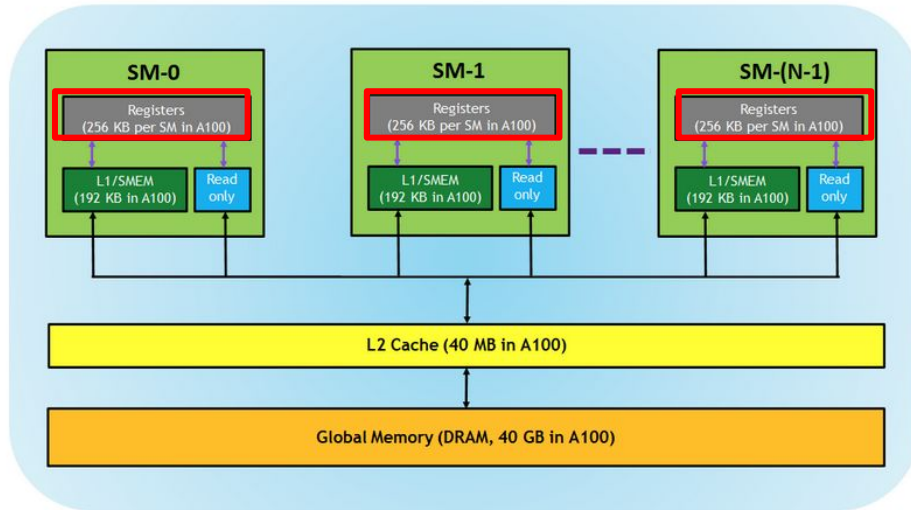
The Streaming Multiprocessor



The SM consists of :

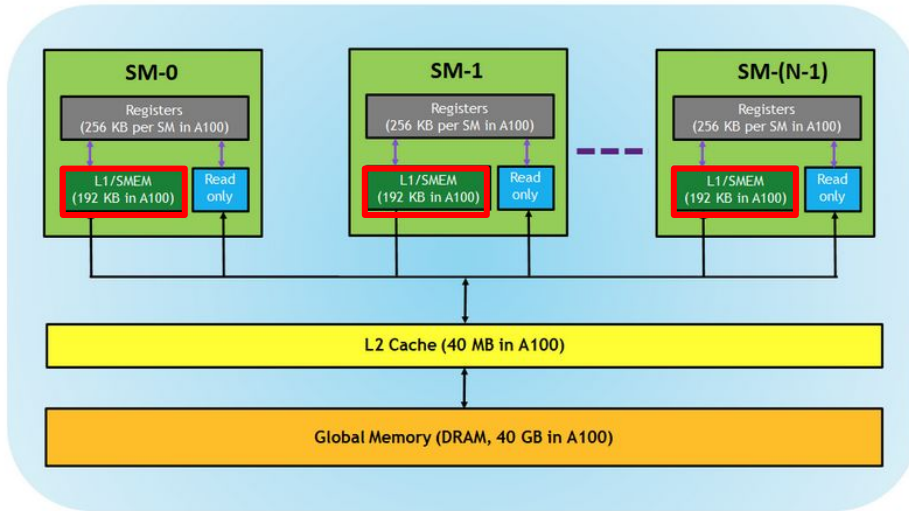
- **Execution cores**
 - e.g. single precision floating-point, special function units etc.
- **Schedulers for warps**
 - These are used for issuing instructions to warps based on a particular scheduling policies.
- **Registers**
 - fast on-chip memory used to store operands for the operations executed by the GPU cores
- **Caches**
 - Intermediate high-speed storage resources between the processor and memory
 - L1/constant/texture cache, Shared memory

GPU memory hierarchy



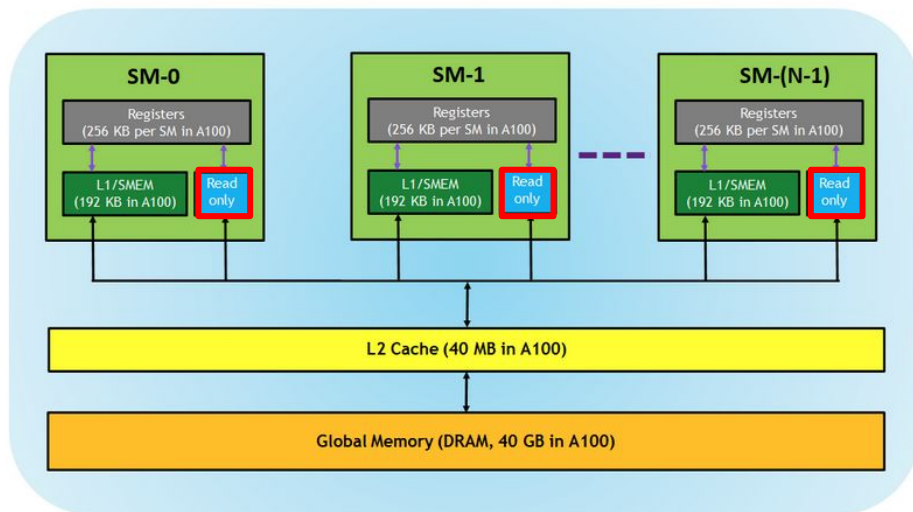
- **Registers**
 - Memory private to each thread
 - Fastest form of memory

GPU memory hierarchy



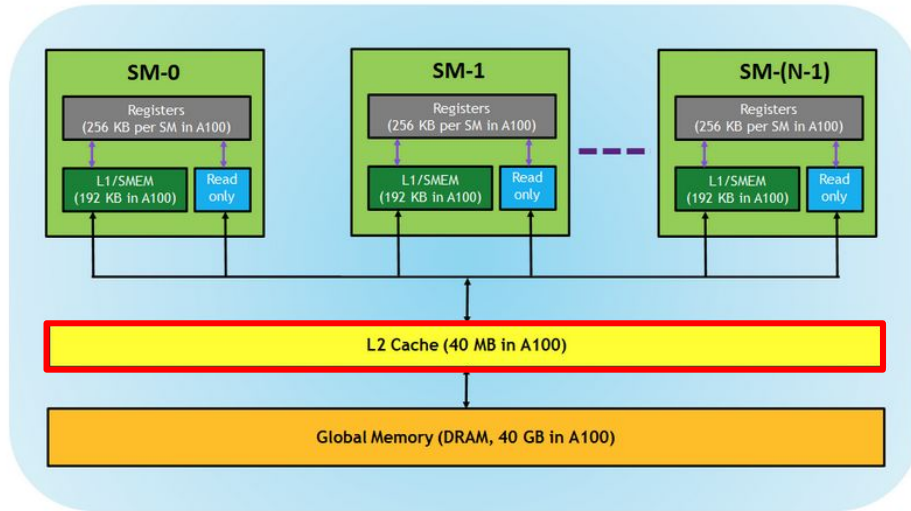
- **Registers**
 - Memory private to each thread
 - Fastest form of memory
- **L1 cache/Shared memory**
 - Fast accessible memory that can be accessed by threads in the same block and threads of different blocks in the same SM

GPU memory hierarchy



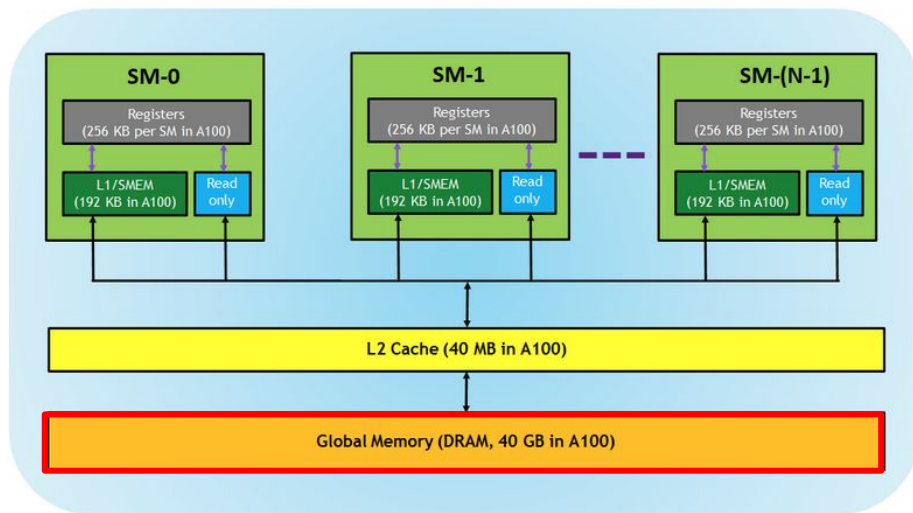
- **Registers**
 - Memory private to each thread
 - Fastest form of memory
- **L1 cache/Shared memory**
 - Fast accessible memory that can be accessed by threads in the same block and threads of different blocks in the same SM
- **Read-only**
 - Each SM has a constant/texture cache memory which is read-only to kernel code. Fast but limited in size

GPU memory hierarchy



- **Registers**
 - Memory private to each thread
 - Fastest form of memory
- **L1 cache/Shared memory**
 - Fast accessible memory that can be accessed by threads in the same block and threads of different blocks in the same SM
- **Read-only**
 - Each SM has a constant/texture cache memory which is read-only to kernel code. Fast but limited in size
- **L2 Cache**
 - Memory that all threads in all blocks can access. Fast but limited in size.

GPU memory hierarchy



- **Registers**
 - Memory private to each thread
 - Fastest form of memory
- **L1 cache/Shared memory**
 - Fast accessible memory that can be accessed by threads in the same block and threads of different blocks in the same SM
- **Read-only**
 - Each SM has a constant/textures cache memory which is read-only to kernel code. Fast but limited in size
- **L2 Cache**
 - Memory that all threads in all blocks can access. Fast but limited in size.
- **Global memory**
 - GPU's DRAM memory
 - Slow but large



Exploring the GPU



nvidia-smi

nvidia-smi: NVIDIA System Management Interface program

- Command line utility
- Aids in the management and monitoring of NVIDIA GPU devices

Lets try this out!

```
ssh <username>@login.hep.wisc.edu
```

```
ssh g38nXX # You can choose XX = 01-16
```

```
nvidia-smi
```

Then let's check out some additional GPU information :

```
touch deviceInfo.cu # Copy the code snippet from the next slide
```

```
export LD_LIBRARY_PATH=/usr/local/cuda/lib
```

```
export PATH=$PATH:/usr/local/cuda/bin
```

```
nvcc deviceInfo.cu -o deviceInfo
```

```
./deviceInfo
```



```

#include <stdio.h>

int main()
{
    int nDevices;
    cudaGetDeviceCount(&nDevices);
    printf("-----\n");
    for (int i = 0; i < nDevices; i++)
    {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        printf("Device Number: %d\n", i);
        printf("  Device name: %s\n", prop.name);
        printf("  Memory Clock Rate (KHz): %d\n", prop.memoryClockRate);
        printf("  Memory Bus Width (bits): %d\n", prop.memoryBusWidth);
        printf("  Compute capability: %d.%d\n", prop.major, prop.minor);
        printf("  Peak Memory Bandwidth (GB/s): %f\n\n", 2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
        printf("  Number of SMs: %d\n", prop.multiProcessorCount);
        printf("  Maximum grid dimensions: %d,%d,%d\n", prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2]);
        printf("  Warp size %d\n", prop.warpSize);
        printf("  Max # of threads / block: %d\n", prop.maxThreadsPerBlock);
        printf("  Max size of a block blockDim.x : %d, .y : %d, .z : %d\n", prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
    }
    printf("-----\n");
}

```

```

#include <stdio.h>

int main()
{
    int nDevices;
    cudaGetDeviceCount(&nDevices);
    printf("-----\n");
    for (int i = 0; i < nDevices; i++)
    {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        printf("Device Number: %d\n", i);
        printf("  Device name: %s\n", prop.name);
        printf("  Memory Clock Rate (KHz): %d\n", prop.memoryClockRate);
        printf("  Memory Bus Width (bits): %d\n", prop.memoryBusWidth);
        printf("  Compute capability: %d.%d\n", prop.major, prop.minor);
        printf("  Peak Memory Bandwidth (GB/s): %f\n\n", 2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
        printf("  Number of SMs: %d\n", prop.multiProcessorCount);
        printf("  Maximum grid dimensions: %d,%d,%d\n", prop.maxGridSize[0], prop.maxGridSize[1], prop.maxGridSize[2]);
        printf("  Warp size %d\n", prop.warpSize);
        printf("  Max # of threads / block: %d\n", prop.maxThreadsPerBlock);
        printf("  Max size of a block blockDim.x : %d, .y : %d, .z : %d\n", prop.maxThreadsDim[0], prop.maxThreadsDim[1], prop.maxThreadsDim[2]);
    }
    printf("-----\n");
}

```

Questions :

- How many devices are found?
- What type of GPUs are they?
- How many SMs per device?
- What is the warp size?
- How many threads are allowed per block?

CUDA programming model

The CUDA programming model

CUDA → **C**ompute **U**nified **D**evice **A**rchitecture.

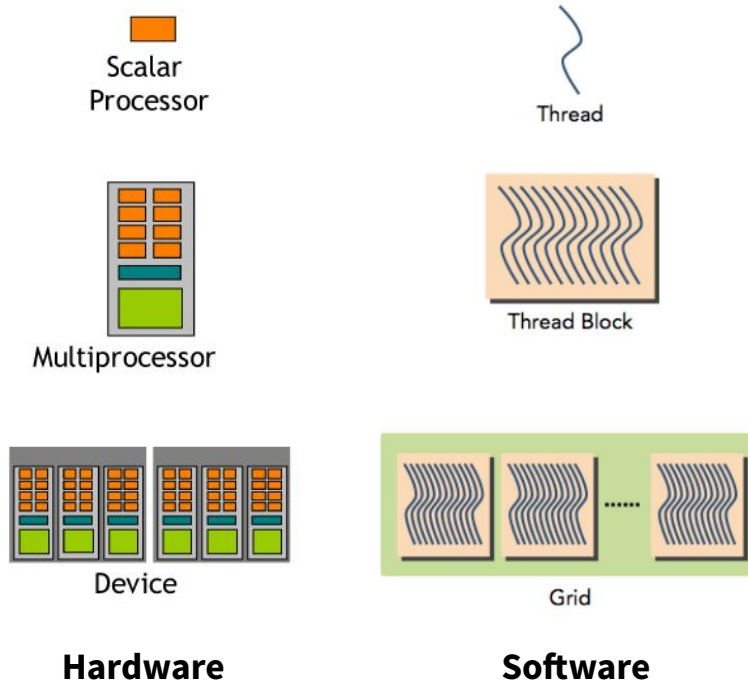
- It is an extension of C/C++ programming
- Developed by Nvidia and is used to develop applications executed on NVidia GPUs

To execute any CUDA program, there are three main steps:

- Copy the input data from CPU or host memory to the device memory
- Execute the CUDA program
- Copy the results from device memory to host memory



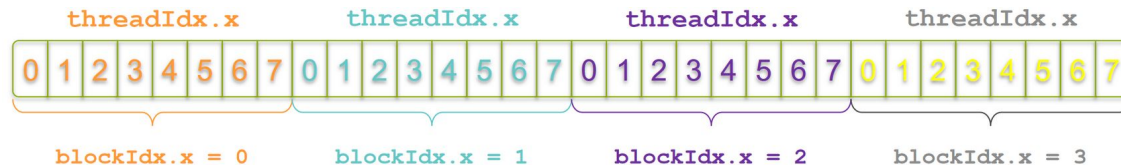
Hardware to software mapping



- A scalar processor or CUDA core is equivalent to a software thread
 - Scalar processors are grouped into a SM
- Each execution of a GPU function is done concurrently on a number of threads referred to as a **thread block**
- Each thread block is executed by one SM and cannot be migrated to other SMs in GPU
- The set of thread blocks executing the GPU function is called a grid.
- In CUDA terminology the GPU is referred to as the device

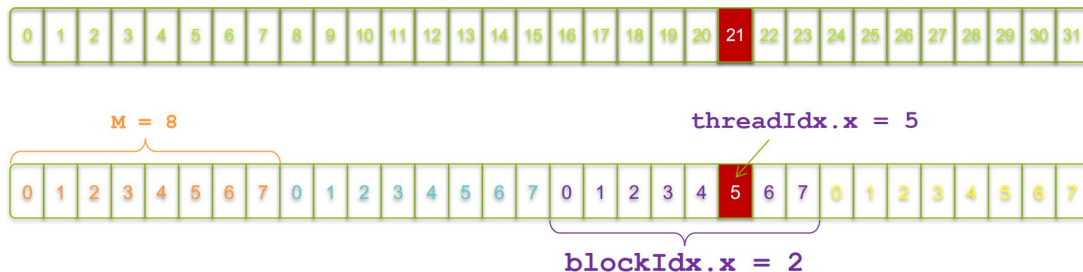
Threads & blocks

- In CUDA, built-in variables are available in order to express threads and blocks :
 - `threadIdx` & `blockIdx`
- The variables have 3-dimensional indexing & provide a natural way to express elements in vectors and matrices :
 - `threadIdx.x` , `threadIdx.y` `threadIdx.z`



Indexing using blockIdx and threadIdx

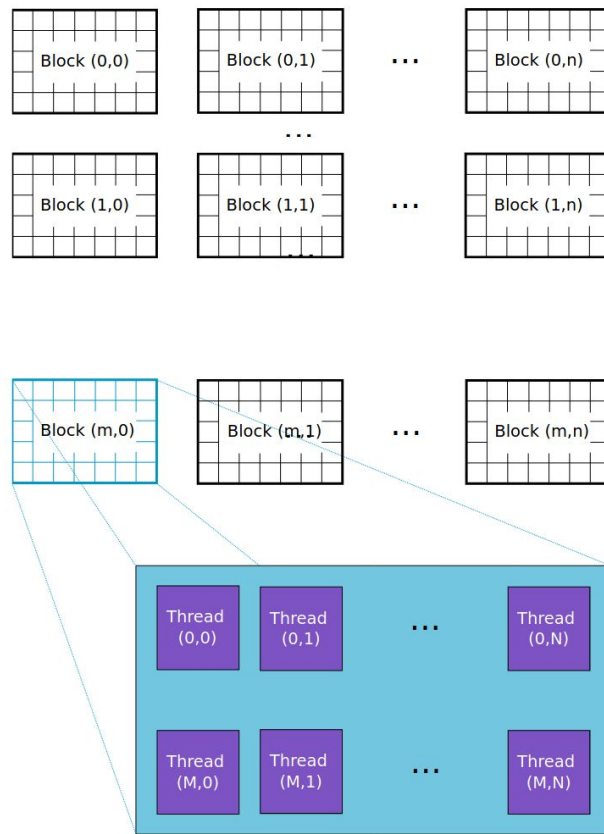
- The `threadIdx` & `blockIdx` variables can be used to express the unique index of an element in an array/matrix etc.
- Assuming that each block consists of a number of `M` threads :
 - `index = threadIdx.x + blockIdx.x * M;`



```
int index = threadIdx.x + blockIdx.x * M;  
          =      5      +      2      * 8;  
          = 21;
```

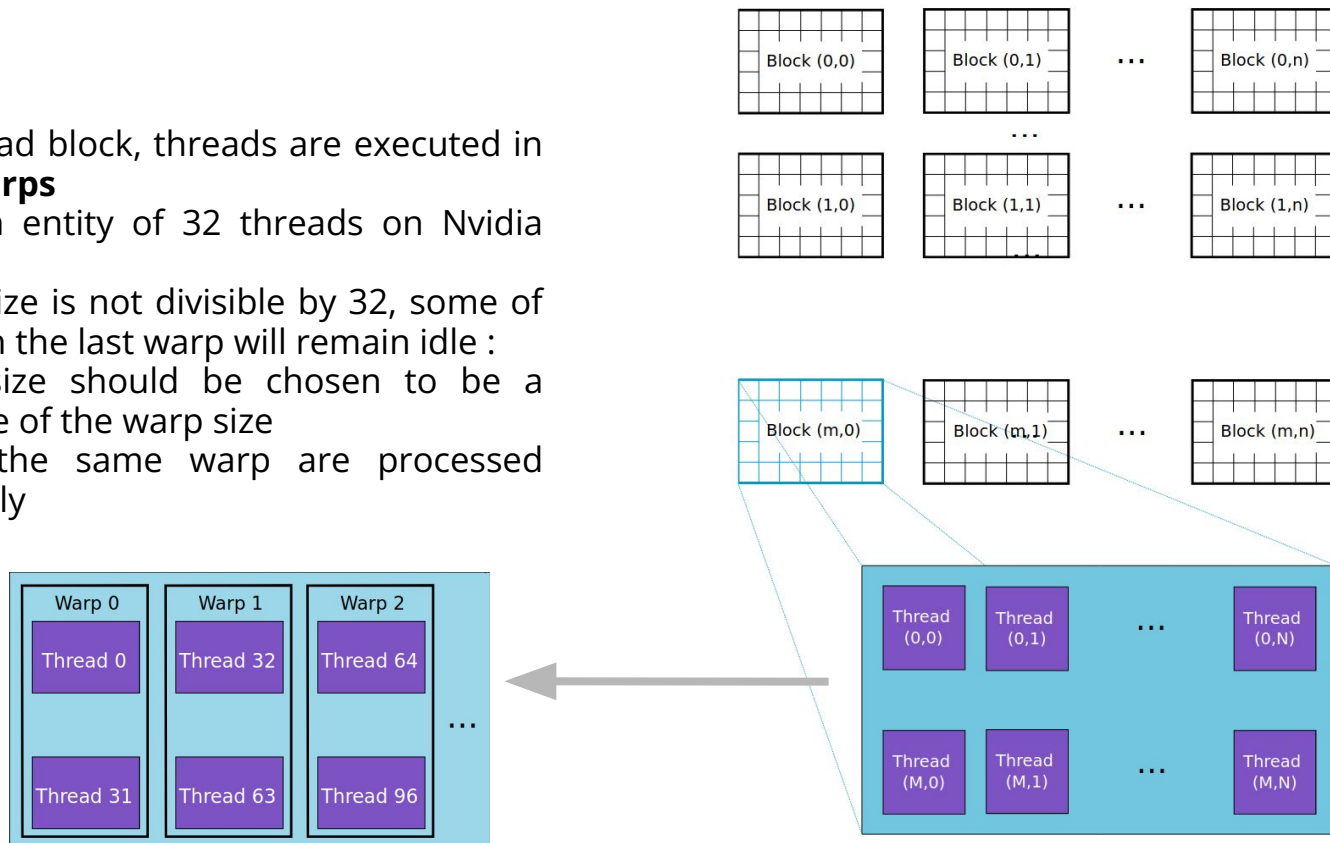

Arranging threads in blocks

- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).
- The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable :
 - blockDim.x, blockDim.y, blockDim.z
- CUDA thread blocks are grouped into a grid
- The dimension of the grid is accessible within the kernel through the built-in **gridDim** variable:
 - gridDim.x, gridDim.y, gridDim.z



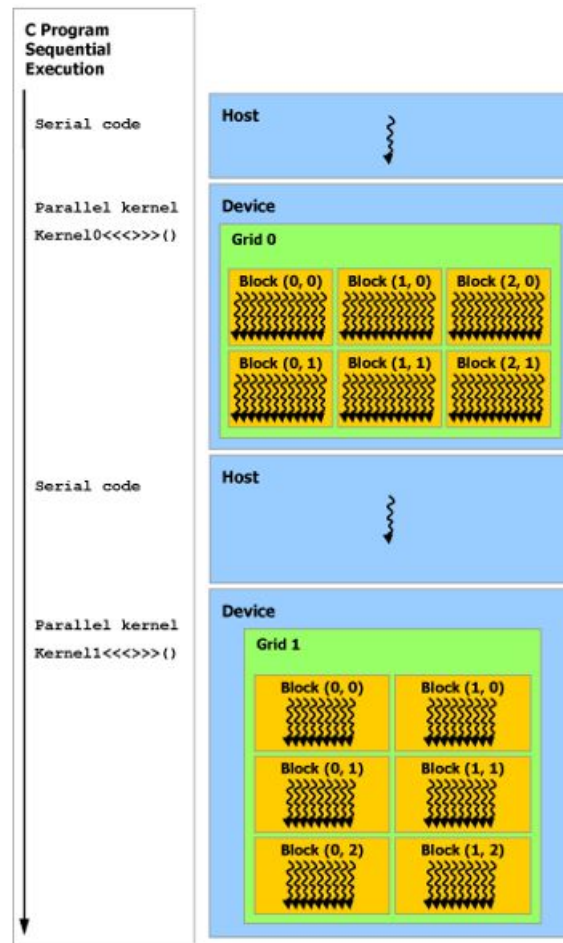
Warps

- Within a thread block, threads are executed in groups → **Warps**
- A warp is an entity of 32 threads on Nvidia GPUs
- If the block size is not divisible by 32, some of the threads in the last warp will remain idle :
 - block size should be chosen to be a multiple of the warp size
- Threads in the same warp are processed simultaneously



CUDA kernel

- CUDA kernel is a function that gets executed on the GPU
- The kernel expresses the portion of the application that is parallelizable
 - It will be executed multiple times in parallel by different CUDA threads



CUDA function declarations

Declaration	Callable from:	Executed on:
<code>__global__</code>	host	device
<code>__device__</code>	device	device
<code>__host__</code>	host	host

- **`__global__`** keyword defines a kernel function:
 - Is launched by host and executed on the device
 - Must return void
- **`__device__`** and **`__host__`** can be used together
- **`__host__`** declaration, if used alone, can be omitted

Launching a CUDA kernel

- Let's assume we have the following kernel :

```
__global__ void mykernel() {  
    ...Do something...  
}
```

This is the grid dimension i.e. the number of blocks that will be launched
CUDA

This is the block dimension i.e. the number of threads within a block

- How do we launch it?

`myKernel<<<nBlocks,nThreads>>>();`

- The above command will launch the kernel with **nBlocks**, each of which has **nThreads**.
- The kernel is executed multiple times concurrently by different threads
- The total number of invocations of the kernel body is now **nBlocks** * **nThreads**.

Our first Hello World kernel

```
#include <stdio.h>
```

```
__global__ void cuda_hello() {  
    printf("Hello World from GPU");  
}
```

__global__ function declaration since the kernel is called from the host & executed on the device.

```
int main() {  
    int gridDim = 1;  
    int blockDim = 1;  
    cuda_hello<<<gridDim,blockDim>>>();  
    return 0;  
}
```

Launched with `gridDim*blockDim` number of threads.

Called from main which is executed on the host

```
touch cuda_hello.cu  
nvcc cuda_hello.cu -o cuda_hello  
./cuda_hello
```

Lets try this out!


What do you observe? Or rather don't observe?

Our first Hello World kernel

```
#include <stdio.h>
```

```
__global__ void cuda_hello() {  
    printf("Hello World from GPU");  
}
```

```
int main() {  
    int gridDim = 1;  
    int blockDim = 1;  
    cuda_hello<<<gridDim,blockDim>>>();  
    return 0;  
}
```



- **Why is nothing printed out on the screen?**
 - Lets try and change the number of threads/block
 - Does this have any impact?

Our first Hello World kernel

```
#include <stdio.h>

__global__ void cuda_hello() {
    printf("Hello World from GPU");
}

int main() {
    int gridDim = 1;
    int blockDim = 1;
    cuda_hello<<<gridDim,blockDim>>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

- Kernel launches are asynchronous
- Control returns to the CPU immediately!
- CPU needs to synchronize before consuming the results

Blocks the CPU until all preceding CUDA calls have complete

Our first Hello World kernel

```
#include <stdio.h>
```

```
__global__ void cuda_hello() {  
    printf("Hello World from GPU. Running thread %d \n", threadIdx.x);  
}
```

```
int main() {  
    int gridDim = 1;  
    int blockDim = 1;  
    cuda_hello<<<gridDim,blockDim>>>();  
    cudaDeviceSynchronize();  
    return 0;  
}
```

Lets try and access some of the device built-in variables !

Try this out :

- Can you also print out the `blockIdx.x` from the `cuda_hello` kernel?
- Let's play around with the number of threads and blocks. What do you observe?



Wrapping-up



Overview of today's lecture

- Learnt about the Nvidia GPU architecture
- Checked if our system has a GPU
- Explored the GPU characteristics
- Got introduced to the concept of parallelization :
 - Threads / blocks / grid
- Learnt about CUDA kernels and the CUDA core syntax
- Wrote our first “Hello world” CUDA kernel

In the next lecture

- We will get deeper into the CUDA programming model :
 - Basic memory management
 - More on synchronization
 - Error handling





Back-up



Resources

1. NVIDIA Deep Learning Institute material [link](#)
2. 10th Thematic CERN School of Computing material [link](#)
3. Nvidia turing architecture white paper [link](#)
4. CUDA programming guide [link](#)
5. CUDA runtime API documentation [link](#)
6. CUDA profiler user's guide [link](#)