



# CE/CZ 3001: Advanced Computer Architecture

## **Module 6: GPU Architecture and CUDA Programming**

- review

Asst Prof Liu Weichen  
School of Computer Science and Engineering  
Nanyang Technological University, Singapore

# Summary

- CUDA C programming
- NVIDIA GPU internals

# Device Code (kernel)

To indicate the code that is to run on the device

- use the CUDA C keyword `__global__` declaration specifier

```
1  __global__ void hello_GPU(void) {  
2      printf("Hello from GPU!");  
3  }
```

Device code are launched as **Kernel** in CUDA

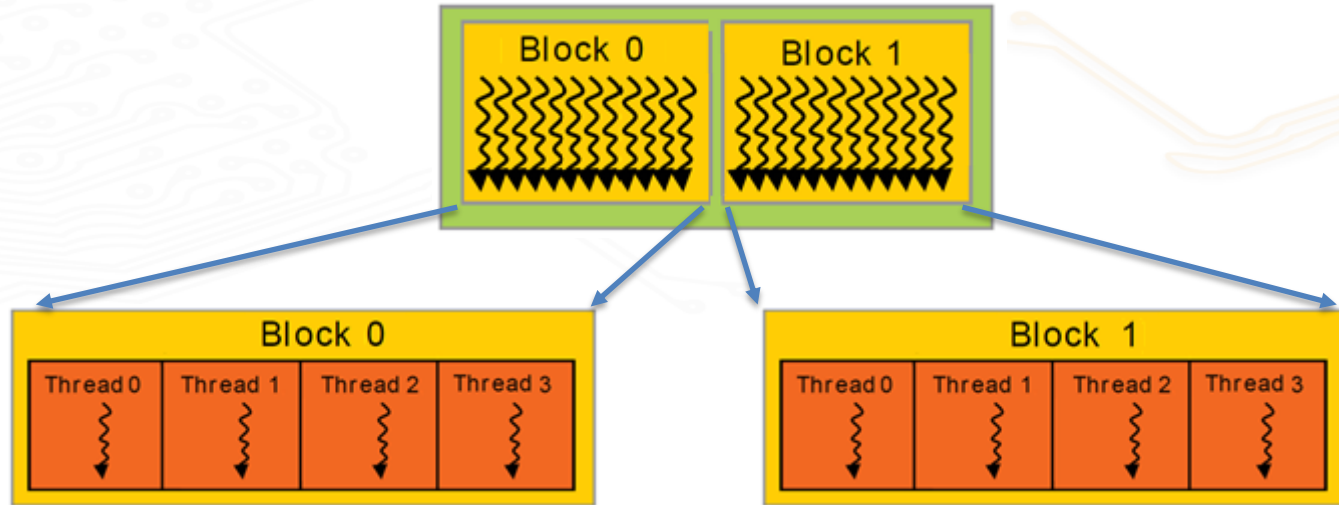
- which can be called from the host code as follows

```
4  int main(void) {  
5      hello_GPU<<<1,1>>>();  
6      printf("Hello from CPU!");  
7      return 0;  
8  }
```

# Threads and Thread Blocks

Example: `kernel<<<2, 4>>> ();`

- will launch 2 thread blocks, each with 4 threads  
→ total of 8 threads, executing 8 copies of the kernel



On latest (2019) NVIDIA GPUs

- a thread block may contain up to 1024 threads

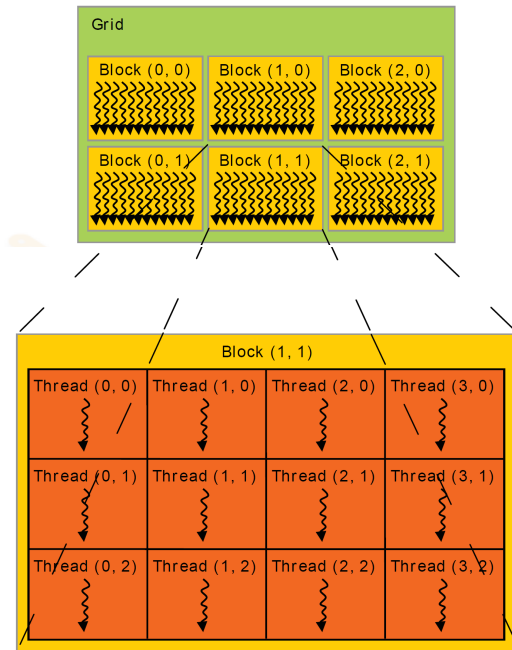
# Multidimension Threads and Blocks

Why are the thread and Block IDs extended with “.x”?

CUDA threads and blocks can be defined to be of 1-dimensional (1D, with x only), 2D (with x and y) or 3D (with x, y and z)

- which are suitable for addressing complex problem best described in multi-dimensions (e.g. Matrix).

```
int i = blockIdx.x * blockDim.x + threadIdx.x;  
int j = blockIdx.y * blockDim.y + threadIdx.y;  
C[i][j] = A[i][j] + B[i][j];
```



(Source: NVIDIA CUDA C Programming Guide)

# Sharing Data between Threads

Need to be able to access each other data

- use shared memory – declared as `__shared__`

```
1  __global__
2  void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3      __shared__ int tmp[3];
4      int i = threadIdx.x;
5      tmp[i] = d_a[i] * d_b[i];
6
7      if (i==0){
8          int sum = 0;
9          for (int j = 0; j < 3; j++)
10             sum = sum + tmp[j];
11         *d_c = sum;
12     }
13 }
```



# Synchronize between Threads

Need to synchronize

- Thread may not be ready to share its data
- use `__syncthreads()`

```
1  __global__
2  void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3      __shared__ int tmp[3];
4      int i = threadIdx.x;
5      tmp[i] = d_a[i] * d_b[i];
6      __syncthreads();
7      if (i==0){
8          int sum = 0;
9          for (int j = 0; j < 3; j++)
10             sum = sum + tmp[j];
11         *d_c = sum;
12     }
13 }
```

# Memory management between Host and Device

This vector addition is obviously most suitable to be executed by using the GPU

- by launching multiple (3) threads 
$$\begin{bmatrix} 7 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} 6 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ 6 \\ 8 \end{bmatrix}$$

But we need to first pass the data from the host to the device (GPU)

- which is done using the CUDA memory management functions:

`cudaMalloc()`, `cudaMemcpy()`, `cudaFree()`

- similar to the standard C equivalent functions:

`malloc()`, `memcpy()`, `free()`



# Synchronization between Host and Device

This program will most likely not work properly

- why?

When we need the host to wait for all the threads to complete the kernels execution

- use the CUDA function `cudaDeviceSynchronize()`

```
1  __global__ void hello_GPU(void) {  
2      printf("Hello from GPU!");  
3  }  
  
4  int main(void) {  
5      hello_GPU<<<1,4>>>();  
6      printf("Hello from CPU!");  
7      return 0;  
8  }
```

# SM Block Allocation

During runtime

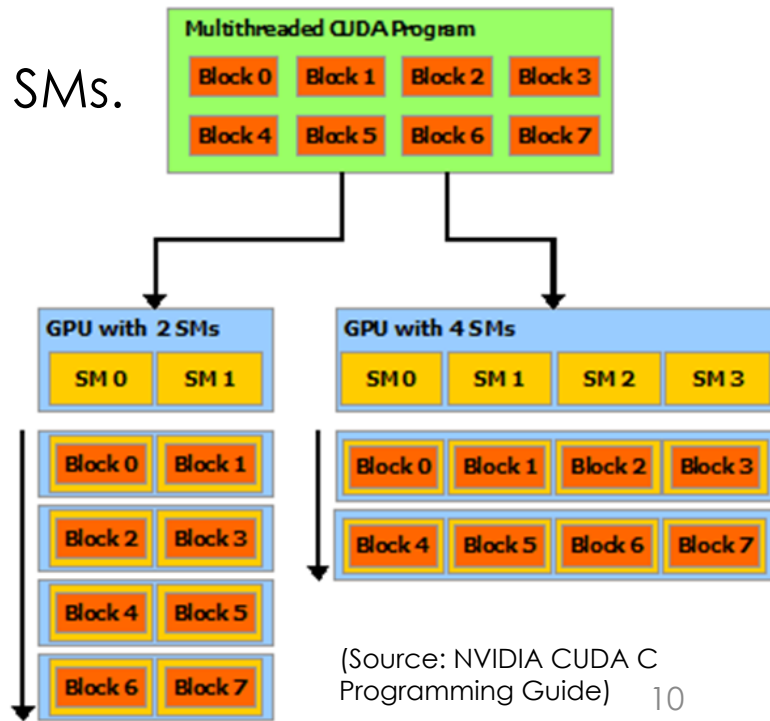
- each **block of threads** can be scheduled on any of the available SM within a GPU, in any order, concurrently or sequentially.
- but block **cannot split** among multiple SMs.

This is **assigned by the runtime system**

- which knows about the internal detail of the underlining GPU
- enable automatic scalability

There is also a maximum number of blocks that each SM can handle (e.g. 8)

- but actual number will be constrained by the available resource.



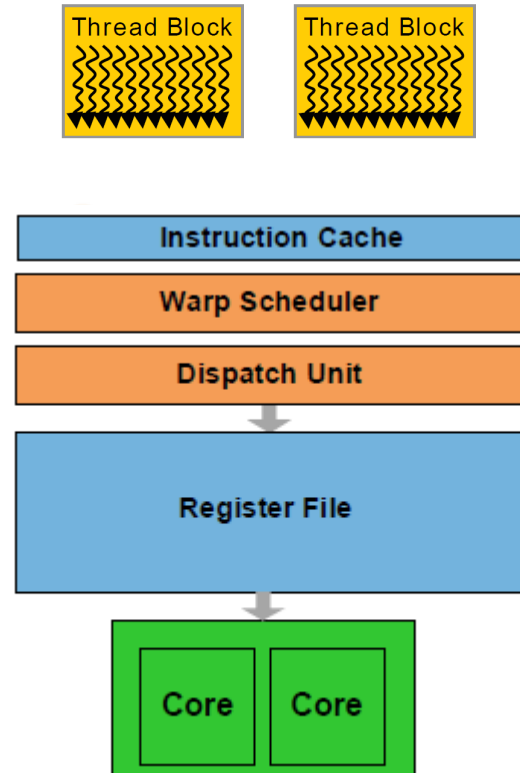
# Warp

In NVIDIA GPU, when a SM is given one or more thread blocks to execute

- it first partitions the threads into **group of 32** known as a **Warp**
  - **all threads** in a warp must execute the **same instruction** but with different data - i.e. SIMD

Each warp then gets scheduled by a **warp scheduler** for execution.

- i.e. threads are always scheduled in a warp group.



# Warp Execution

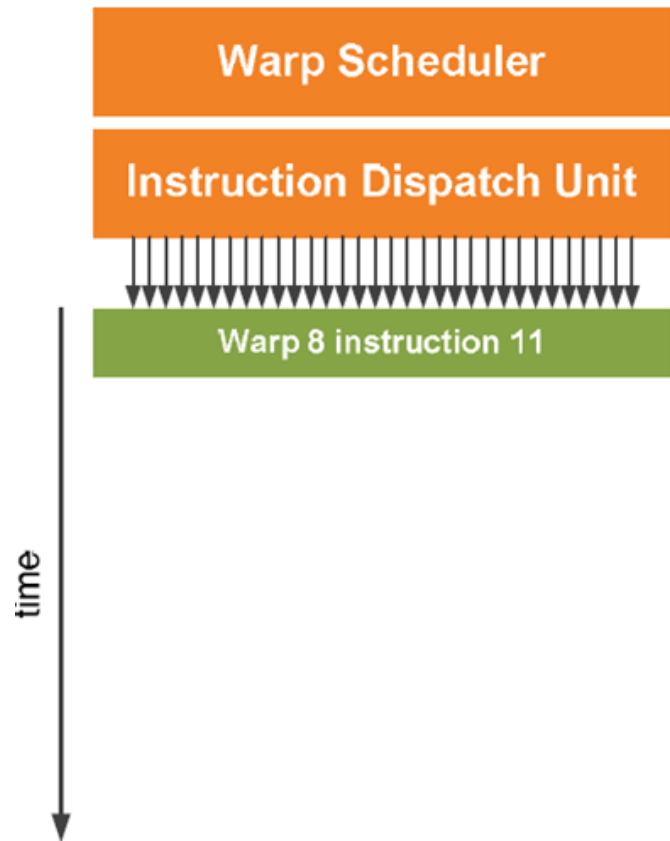
While a warp is waiting for the results of a previously executed instruction

- E.g. LD/ST data from memory
- a different warp that is ready is selected for execution.
- hence 'hiding' the latency

So it is more efficient if we have many threads and many blocks.

Warp selection is based on certain prioritized scheduling policy

- E.g. Round Robin, Least Recently Fetched

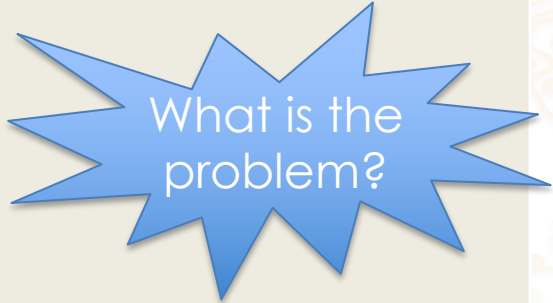


(Source: NVIDIA Fermi Compute Architecture Whitepaper)

# Thread Divergence On Performance

Consider the If-then-else statements in a kernel code as shown below.

```
1  __global__  
2  void kernel_x(int n, int x){  
3      int i;  
4      i = n * 20;  
5      if (i < 300)  
6          x = n;  
7      else  
8          x = n- 300;  
9      }  
10  
11  int main(void){  
12      :  
13      kernel_x<<<2, 64>>>(d_n, d_x);  
14      :
```



What is the problem?

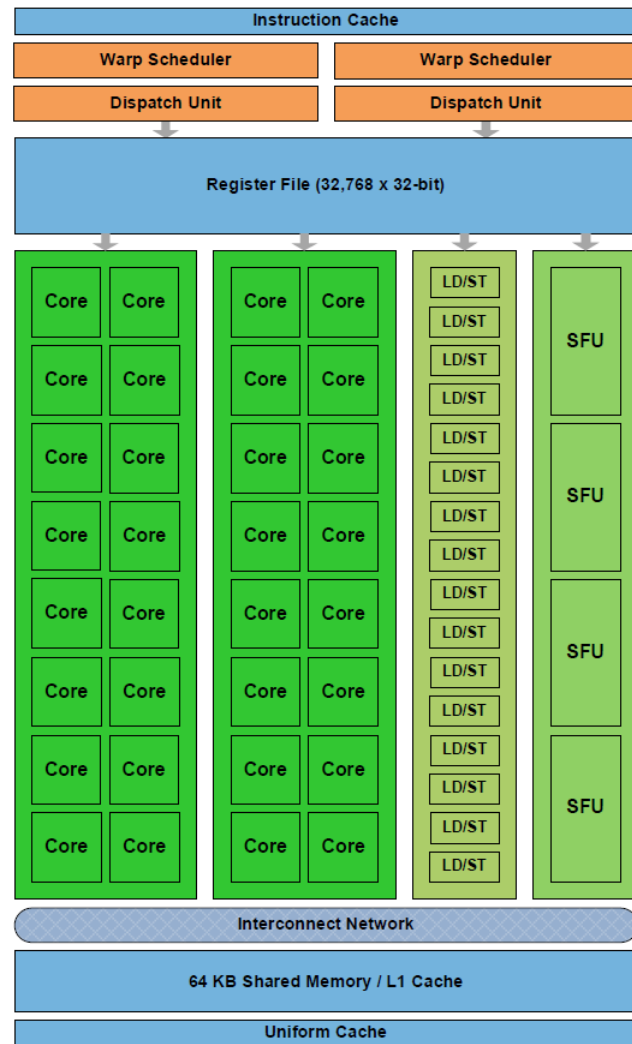
# Hardware Resource Allocation

On-chip resources are used to store the **execution context** for each warp processed by a SM

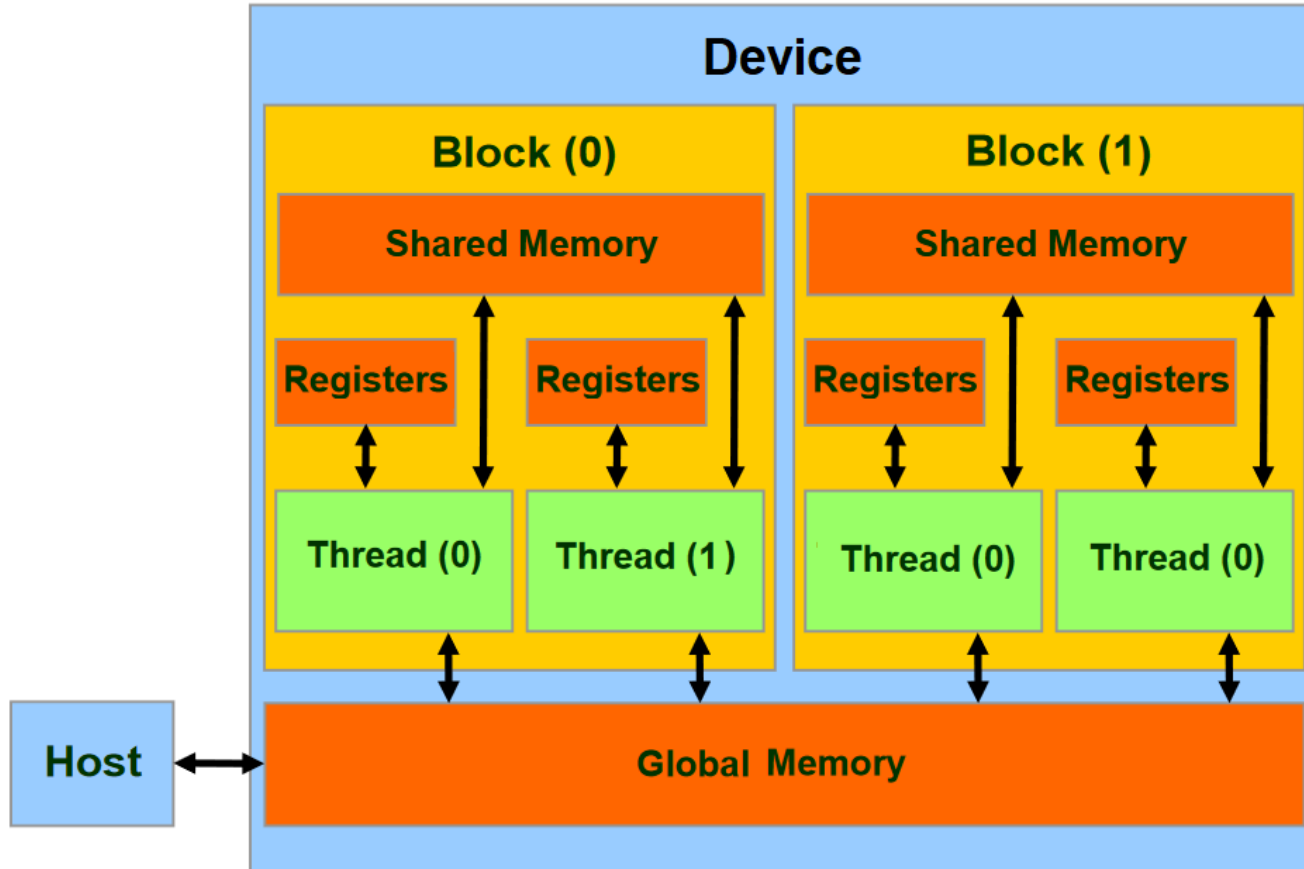
- E.g. program counters, registers, shared memory, etc.

The context is **maintained on-chip** during the entire lifetime of the warp

- allow fast switching between warps
- without the need of the conventional CPU's context switching.



# Memory Resources



# Hardware Resource

Once a thread block is launched on a SM

- all its warps are resident until their executions finish.

Thus a new block is not launched on an SM

- until there is sufficient number of free registers for all warps of the new block,
- and until there is enough free shared memory for the new block.