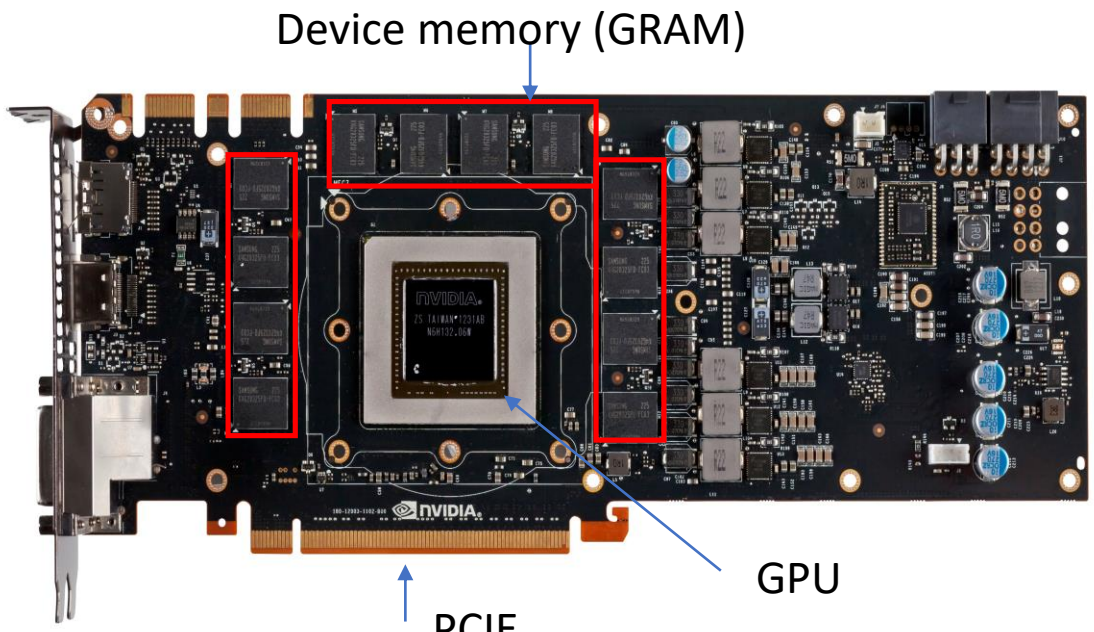
CUDA Programming Model (Part 1)

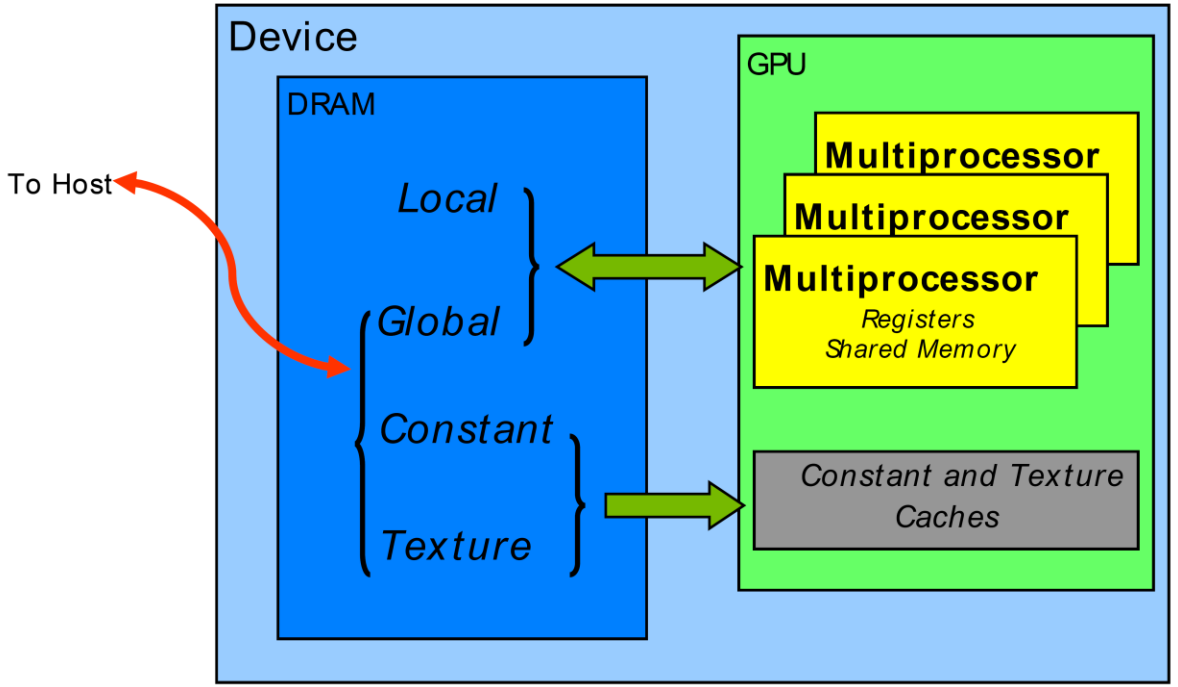
ECE 285

Cheolhong An

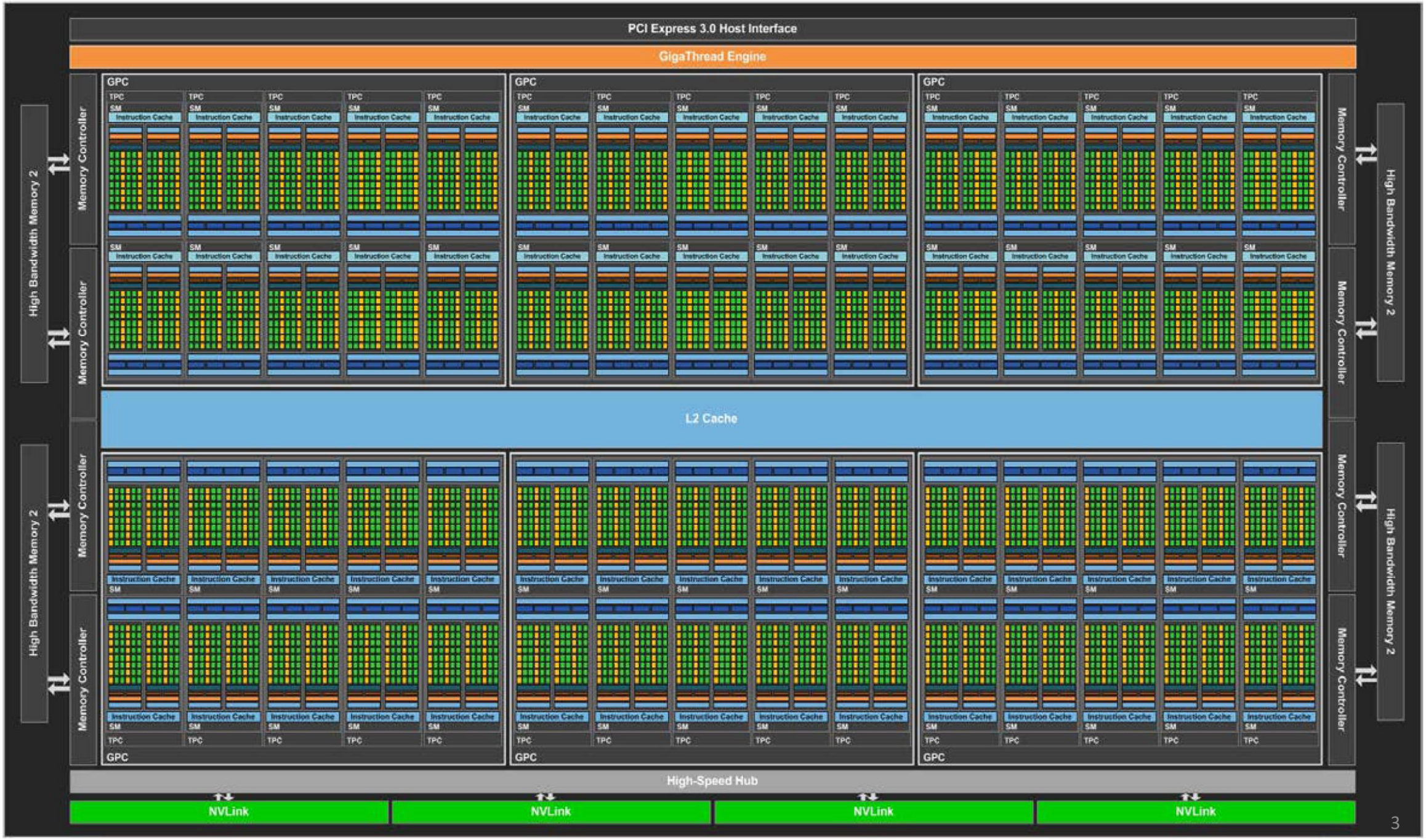
GPU Physical view vs Logical view



Physical view



Logical view

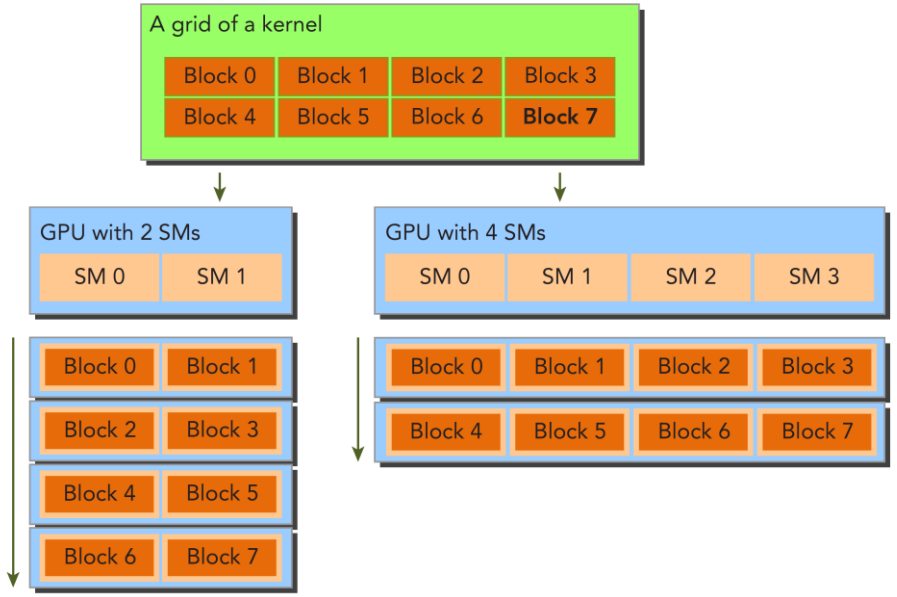

PCI Express 3.0 Host Interface
GigaThread Engine
GPC
TPC
SM
Instruction Cache
Memory Controller
High Bandwidth Memory 2
L2 Cache
High-Speed Hub
NVLink

Streaming Multiprocessor (SM)

- It is similar to CPU core HW
- The threads of a thread block execute concurrently on one multiprocessor.
- As thread blocks terminate, new blocks are launched on the vacated multiprocessors

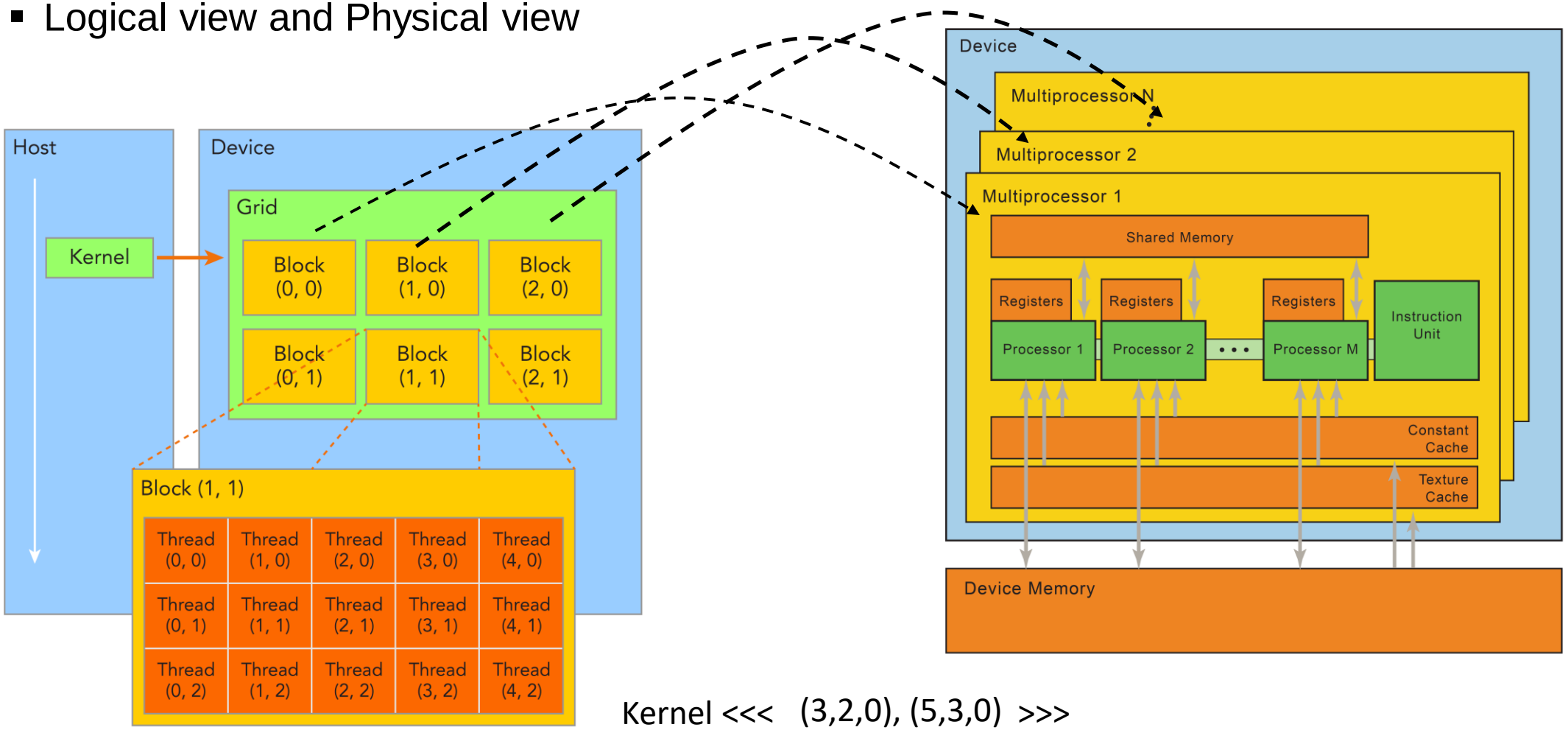
- Kernel <<< 8, 64 >>> ()

thread Block 0 ~ thread Block 7
Each block has 64 threads



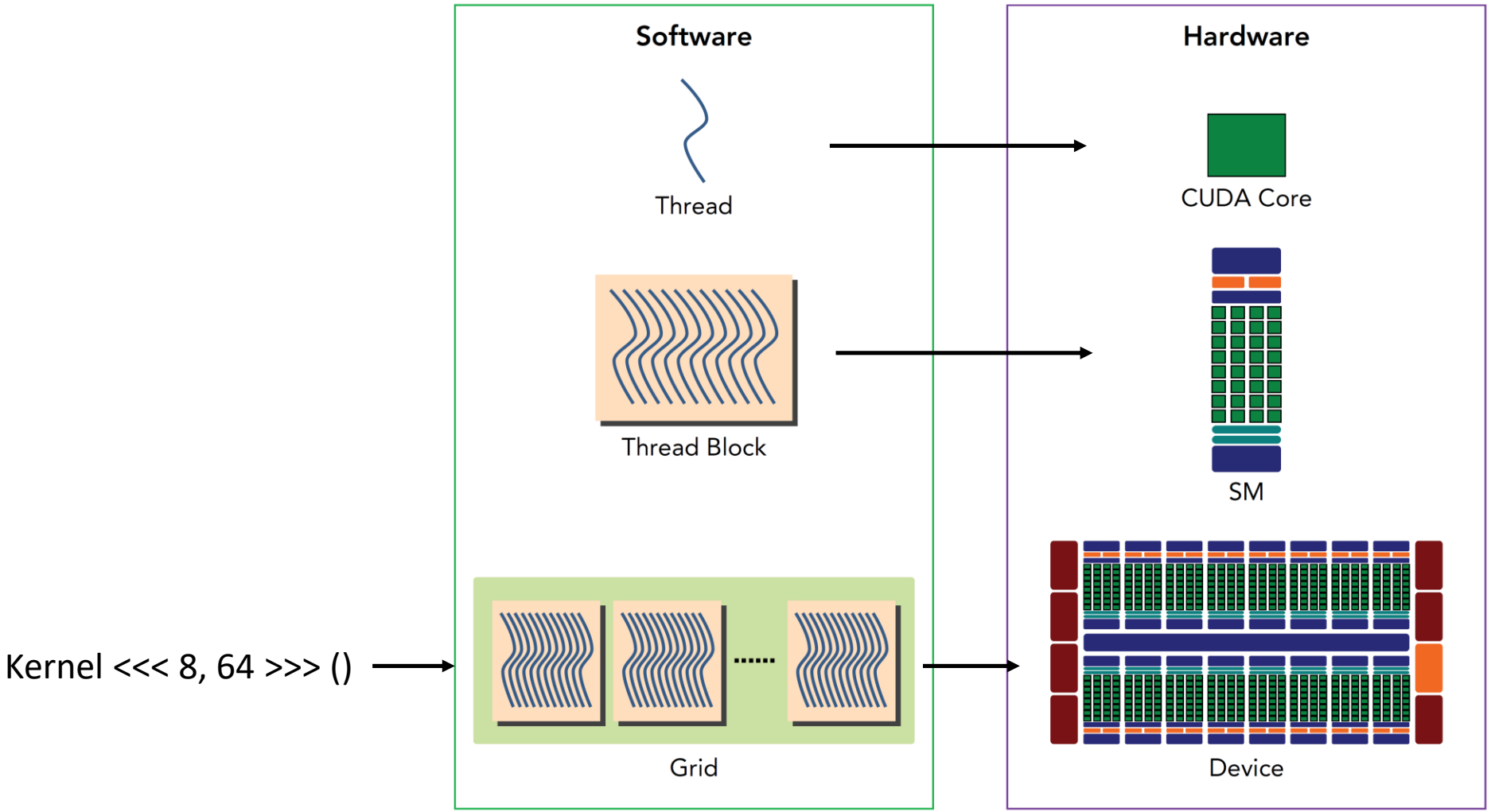
Parallel Threads and Threadblocks

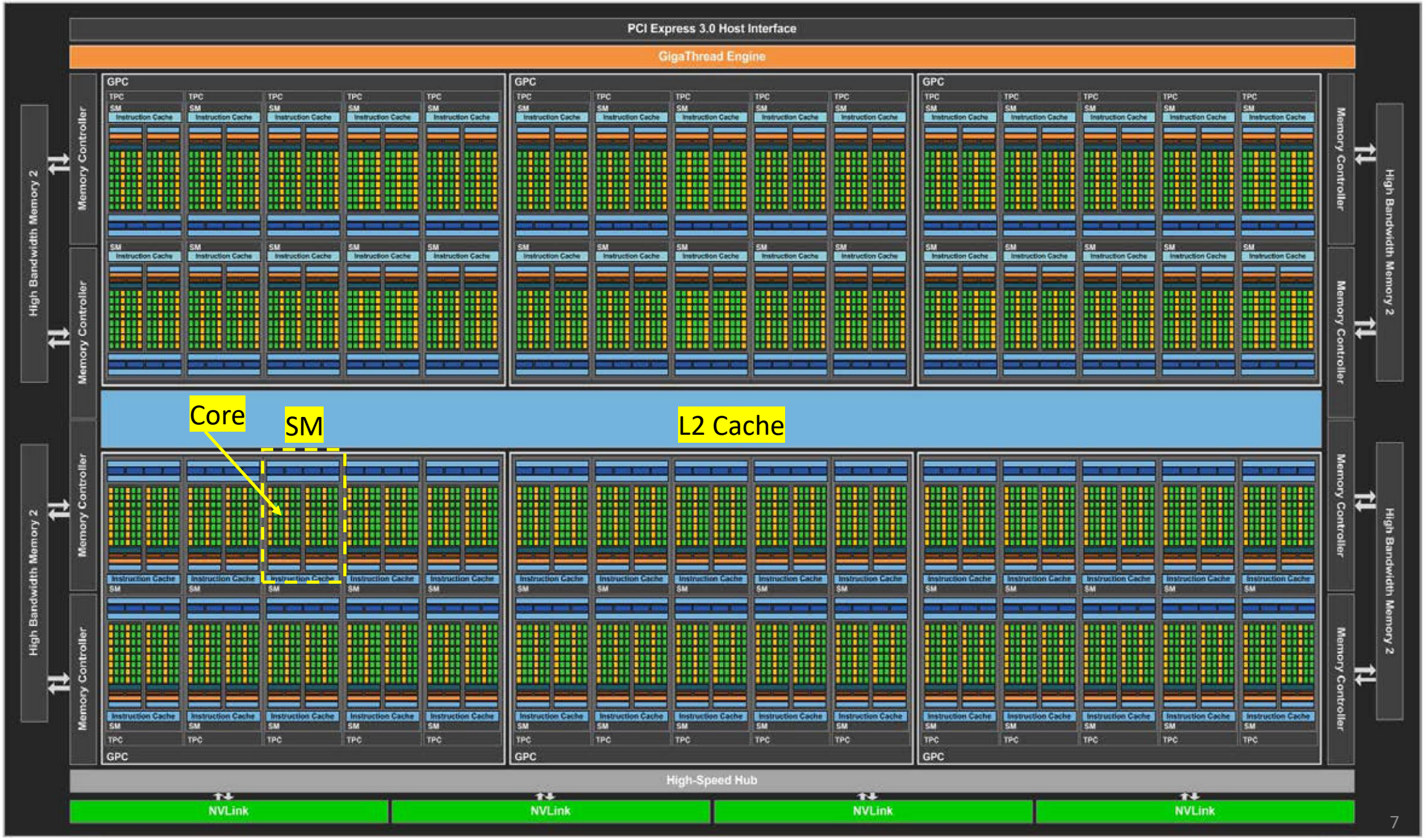
- Logical view and Physical view

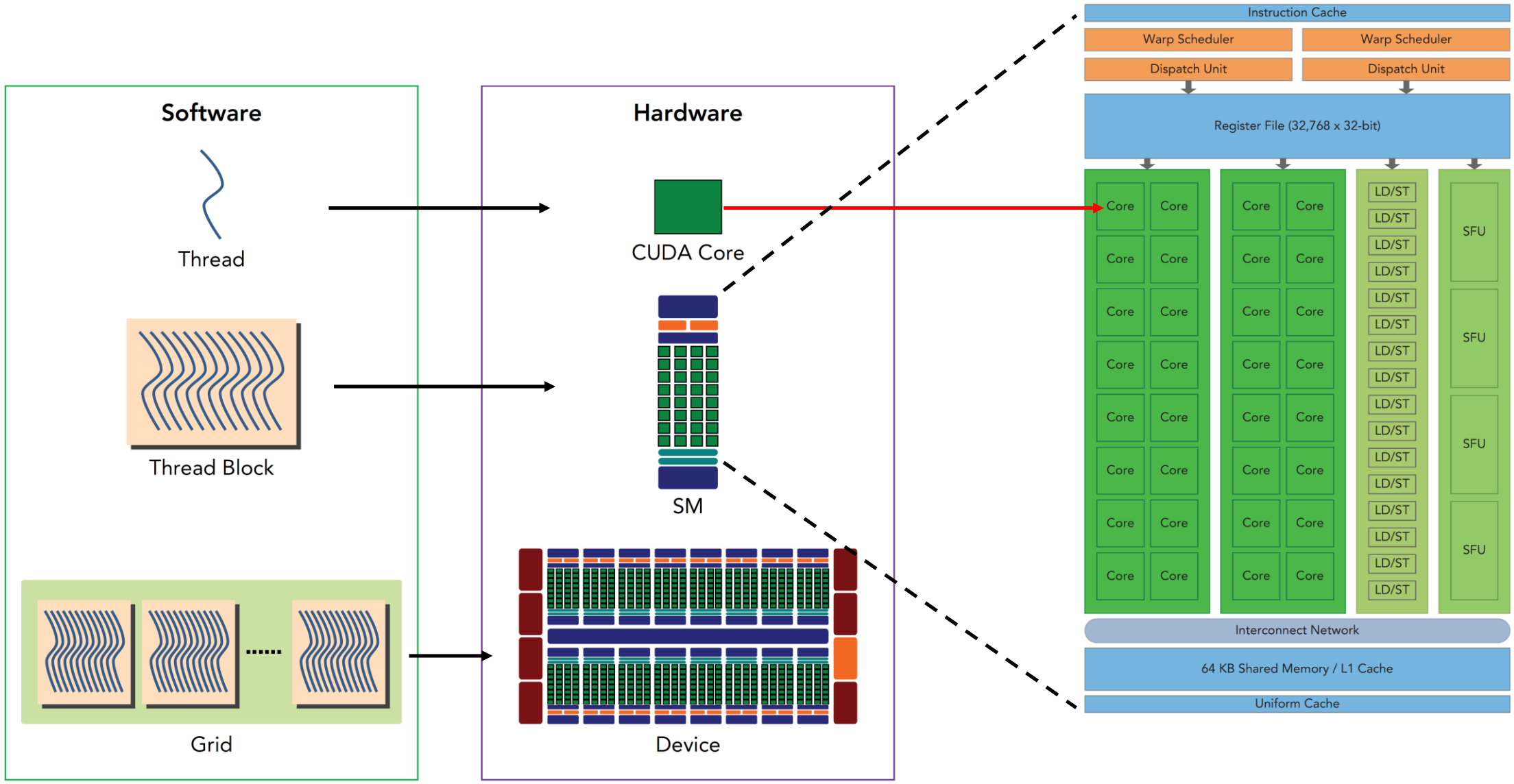


Kernel <<< (3,2,0), (5,3,0) >>>

Decomposition, Assignment => mapping




PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
High Bandwidth Memory 2
GPC
TPC
SM
Instruction Cache
Core
SM
L2 Cache
High-Speed Hub
NVLink


Software
Thread
Thread Block
Grid
Hardware
CUDA Core
SM
Device
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache

Warps and Thread Blocks

- Once a thread block is scheduled to an SM, threads in the thread block are further partitioned into warps.
- A warp consists of 32 consecutive threads
- all threads in a warp are executed in Single Instruction Multiple Thread (SIMT) fashion
- That is, all threads execute the same instruction, and each thread carries out that operation on its own private data
- A threadblock consists of up to 1024 threads
- However, It decreases current SM

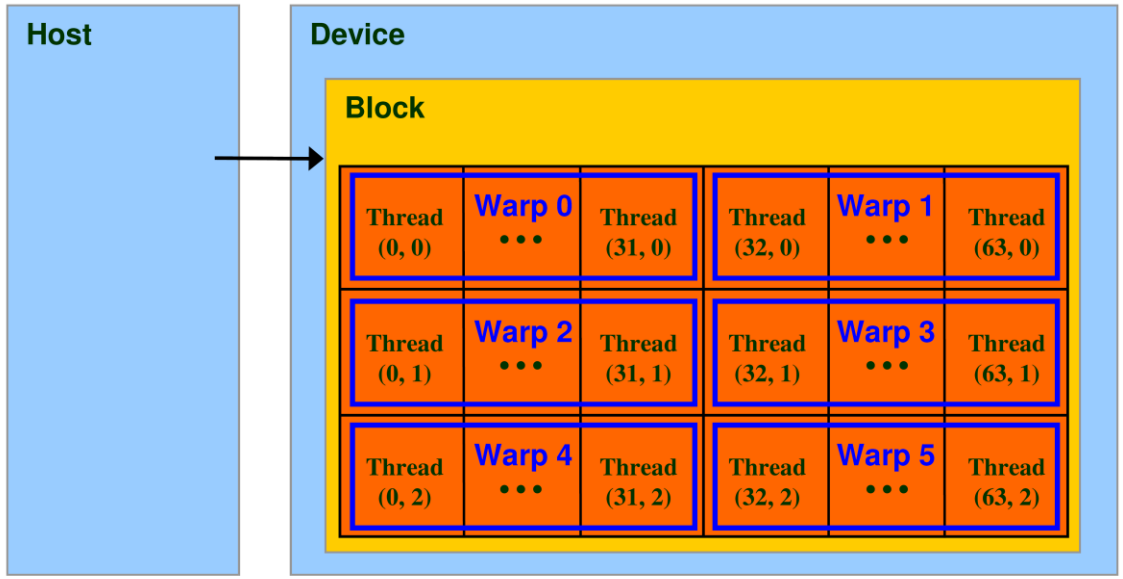
Kernel <<< 8, 1024>>> () ⟶ One SM can only run even if your GPU has 4 SMs

Kernel <<< 8, 256>>> () ⟶ 4 SMs can run simultaneously

Generally golden rule, create only 2 warps ~ 8 warps per threadblock

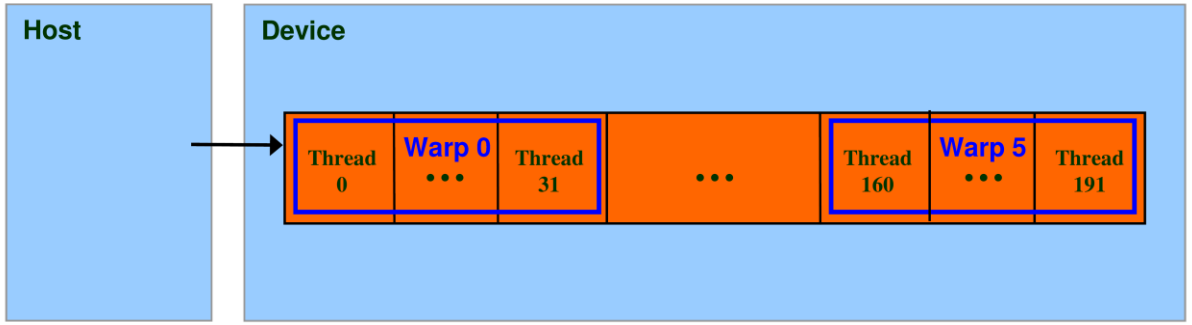
Threadblock

- A group of warps that is executed on a single SM
- A threadblock is a granularity to assigned into SMs
- Threadblocks execute concurrently on multiple SMs

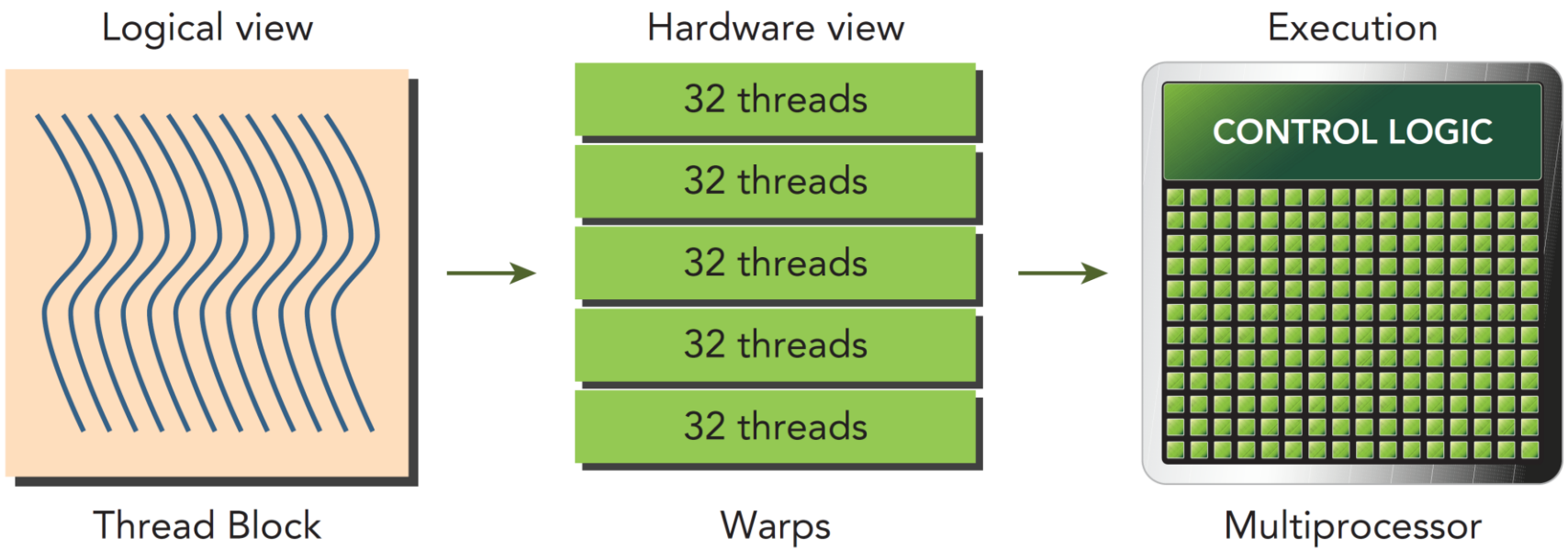


Warp

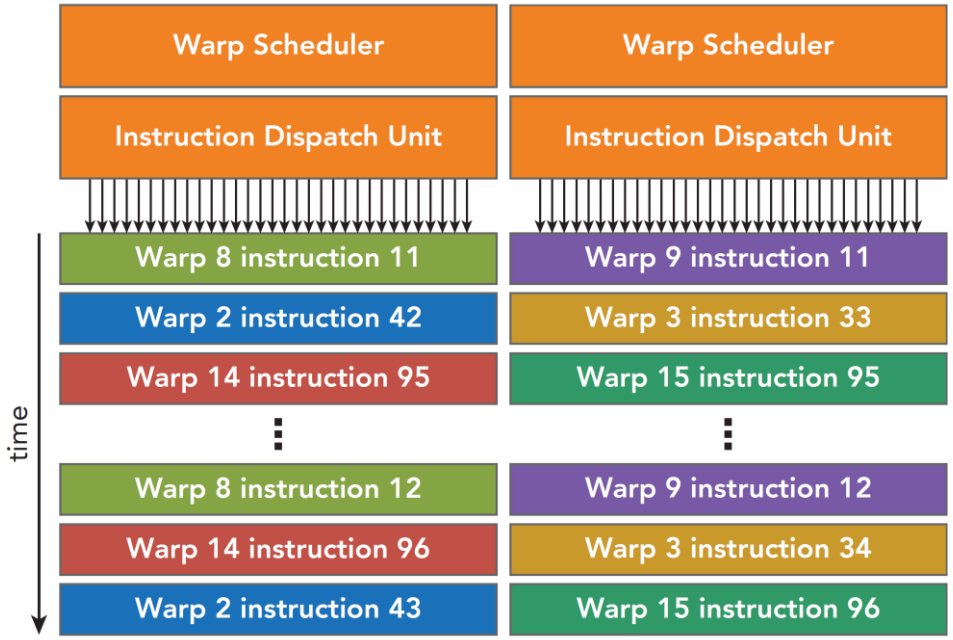
- A group of threads executed physically in parallel (SIMD)
- **The warp size is 32 threads**
- Every consecutive 32 threads in a threadblock is assigned into a warp
  - ex) Kernel0 <<< 5, 40 >>>
  - kernel0: total 200 threads (5*40) , 10 warps (not 7 wraps)
  - 40 threads: 2 warps
  - 0-31 -> warp0, 32-39 -> warp1



```
Warp 0: thread  0, thread  1, thread  2, ... thread 31
Warp 1: thread 32, thread 33, thread 34, ... thread 63
Warp 3: thread 64, thread 65, thread 66, ... thread 95
Warp 4: thread 96, thread 97, thread 98, ... thread 127
```



Warp mapping

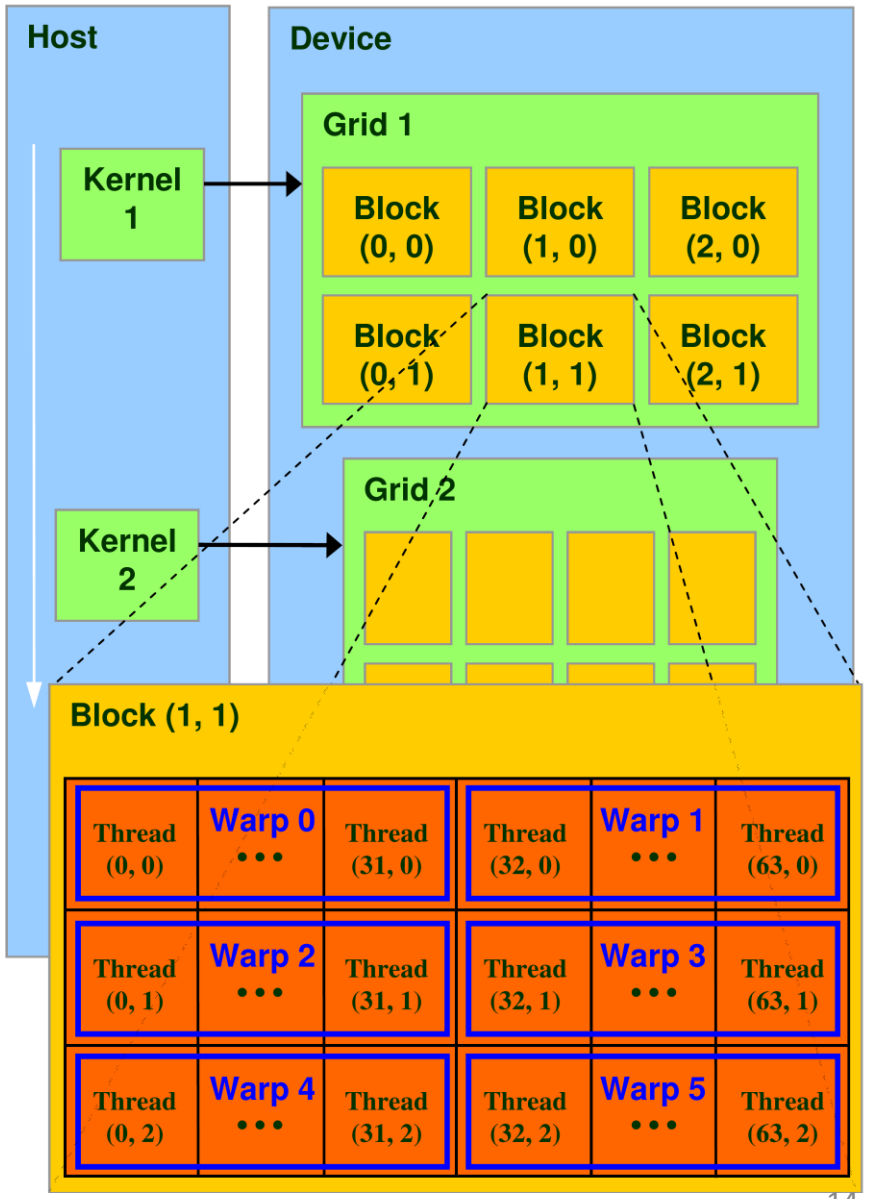


Grid and Kernel

- Grid is a group of threadblocks which executes a single kernel
- Multiple Grids are created if you call multiple kernels

ex)

```
main() {
        Kernel1<<<    , >>> ()
        Kernel2<<<    , >>> ()
}
```



Predefined kernel variables (1/3)

- If you call a kernel function,
  predefined kernel variables are assigned
  blockIdx, threadIdx, blockDim, gridDim, etc.

- blockIdx
  block index within a grid

- threadIdx
  thread index within a threadblock

ex) kernel<<< 2, 4>>> ()

blockIdx.x=0, blockIdx.x=1 (2 threadblocks)

threadIdx.x = 0~3 (4 threads/threadblock)

Predefined kernel variables (2/3)

- gridDim
  the number of threadblocks in a gird
- blockDim
  the number of threads in a threadblock

kernel<<< 2, 4>>> ()

blockDim.x = 4, blockDim.y = 1, blockDim.z = 1

gridDim.x = 2, gridDim.y=1, gridDim.z = 1

e.g) class_lab (c1_checkDimension)

Predefined kernel variables (3/3)

- Instead of directly assigning dimensions,
  kernel<<< 2, 4>>> ()

we can use “dim3” type to assign dimensions.

dim3 type is defined in CUDA and unused fields are initialized to 1.

e.g)

```
dim3 nthreads;
ntreads.x = 4; nthreads.y = 1; nthreads.z = 1;
dim3 nblocks;
nblocks.x = 2; nblocks.y = 1; nblocks.z = 1;
kernel<<<nblocks, nthreads>>>()
```

How to decide blockDim and GridDim ?

- kernel<<< #threadblock, #threads/threadblock>>> ()
- Golden rule
  1) First, **decide the blockDim which is the number of threads in a threadblock**

$$64 \times n, \text{ where } n \in \{1, 2, 3, 4\}$$

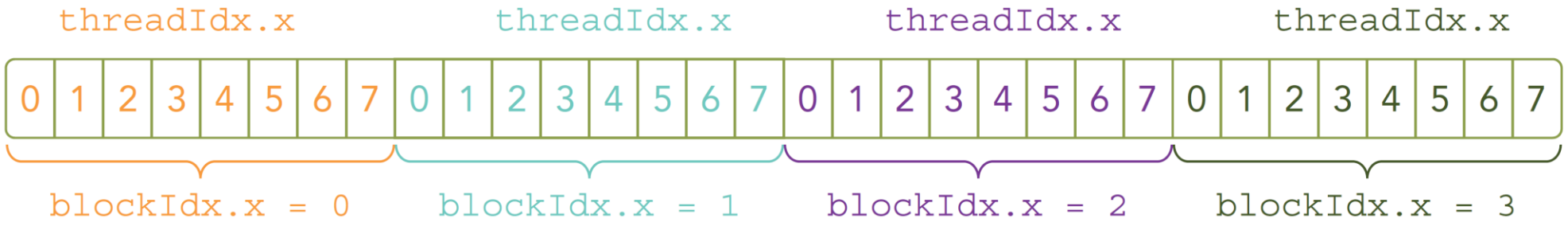
  2) derive the gridDim from your problem size
  e.g) image size (WxH)
  The total number of pixels is WxH
  dim3 nthreads(64);
  dim3 nthblocks((WxH + nthread.x-1)/nthreads.x);

↑ Division should be done with the ceiling operation.
Can you answer the reason ?

- Class lab: c1_defineGridBlock

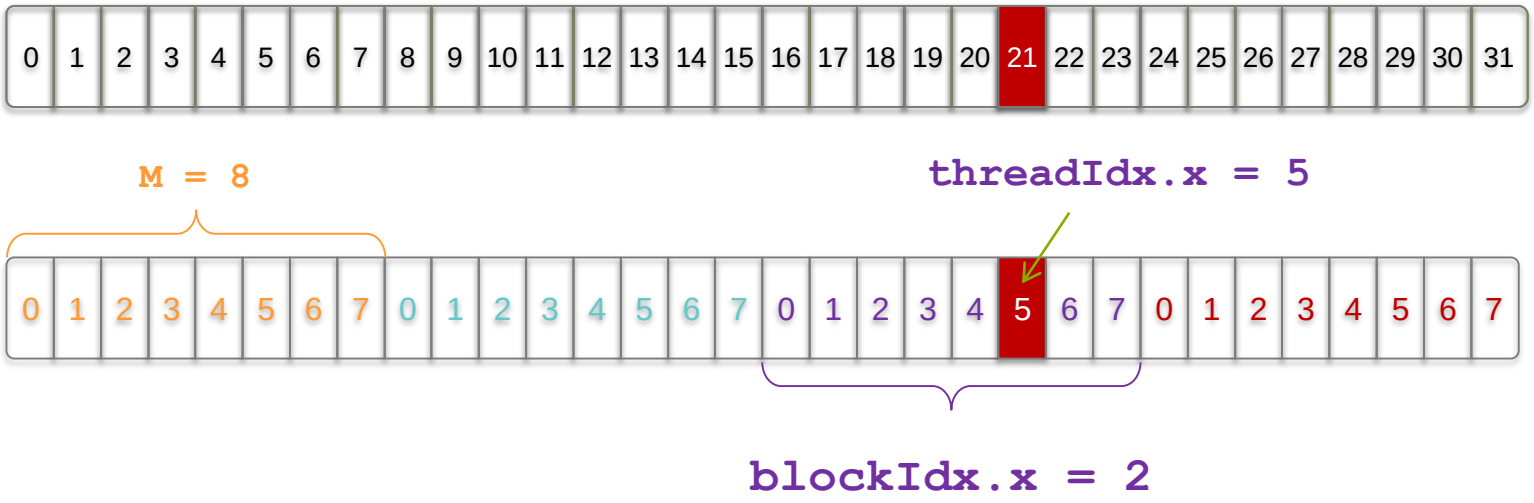
Example: ThreadIdx and BlockIdx

- Kernel<<< 4, 8 >>>()



Example: Linear index

- Kernel<<< 4, 8 >>>()
- Which thread will operate on the red element?




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