GPU Computing

Lecture 5

Young-Ho Gong

광운대학교
KwangWoon University

Contents

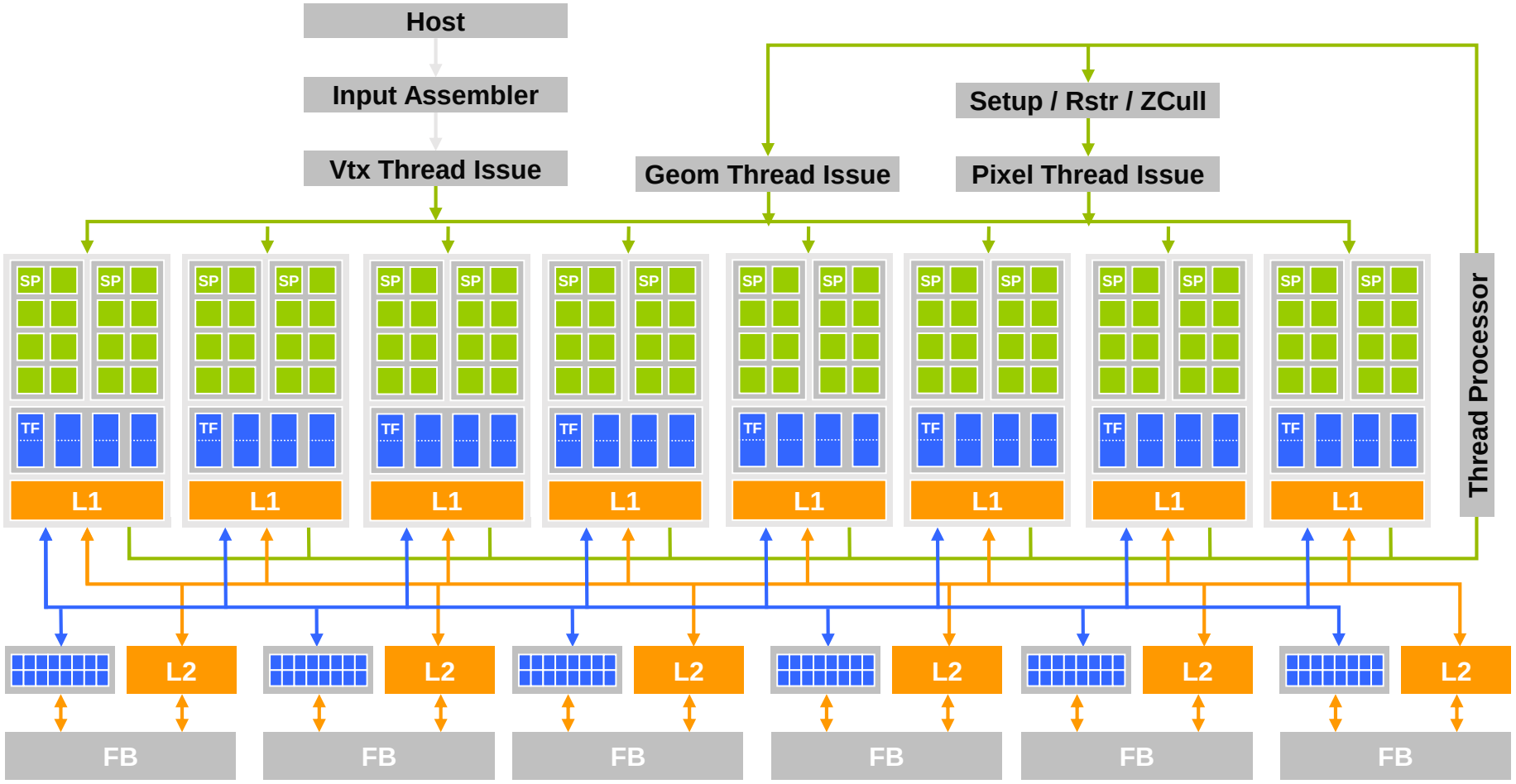
- CUDA Thread and GPU
- CUDA Hardware
  - G80 Architecture
  - Pascal Architecture
- Transparent Scalability
- Thread and Warp
- CUDA Thread Scheduling
  - SM Warp Scheduling
- Overall Execution Model

Underlying CUDA Hardware

- Nvidia GeForce 8 series (shortly, G80)
  - Firstly released in 2006
  - Includes GeForce 8800, 8600, 8400 and more
  - Unified shader architecture
    - So, it supports graphics and computing simultaneously
  - Some hardware features are still based on graphics features
  - Some are new for computing devices

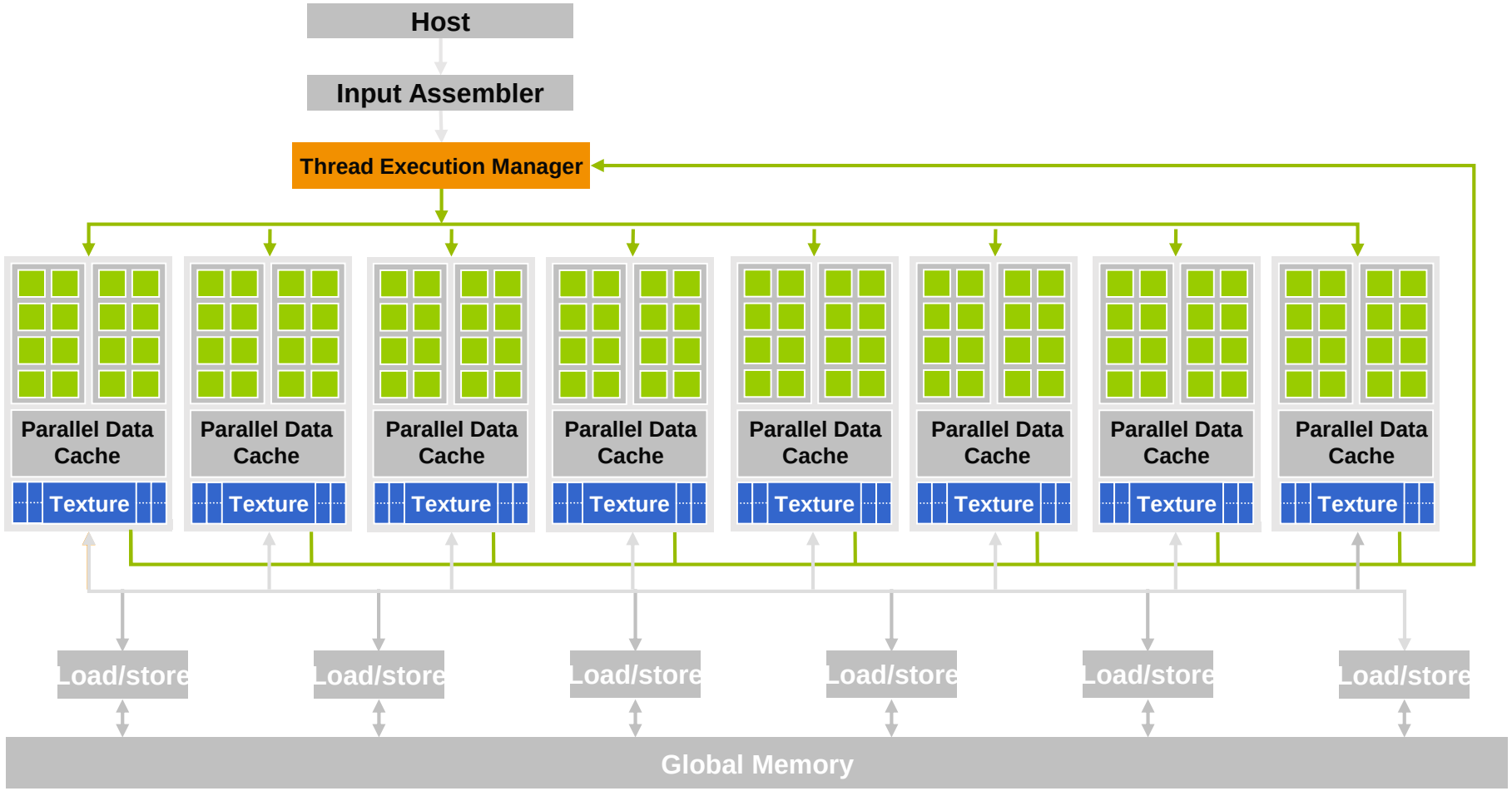
G80 Graphics Pipeline

- Graphics oriented → modified to support CUDA



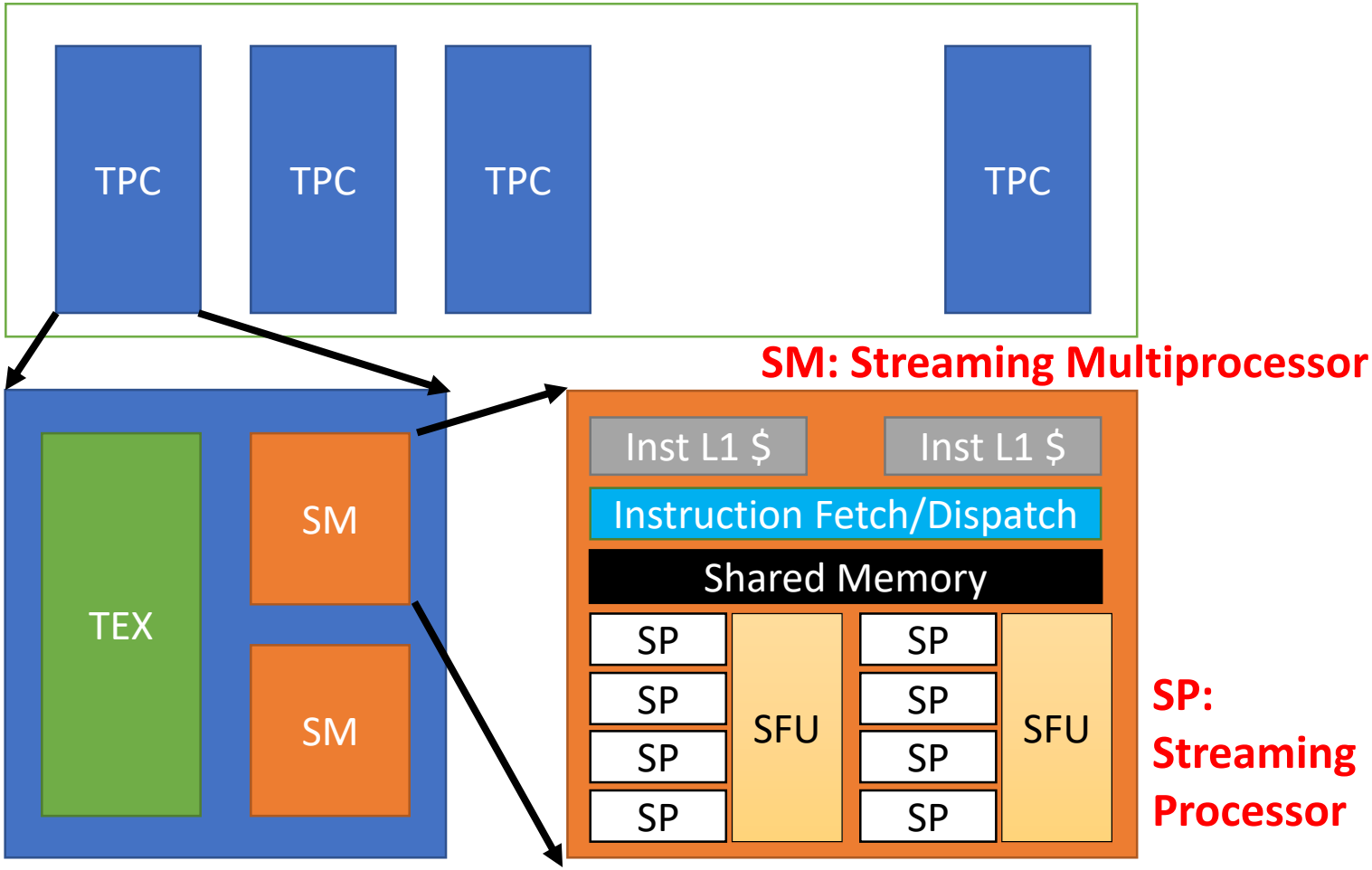
G80 Computing Pipeline

- Processors execute computing threads
- New operating mode/HW interface for computing



GPU Internal Architecture

- Hierarchical Approach
  - SPA: Streaming Processor Array



CUDA Processor Terminology

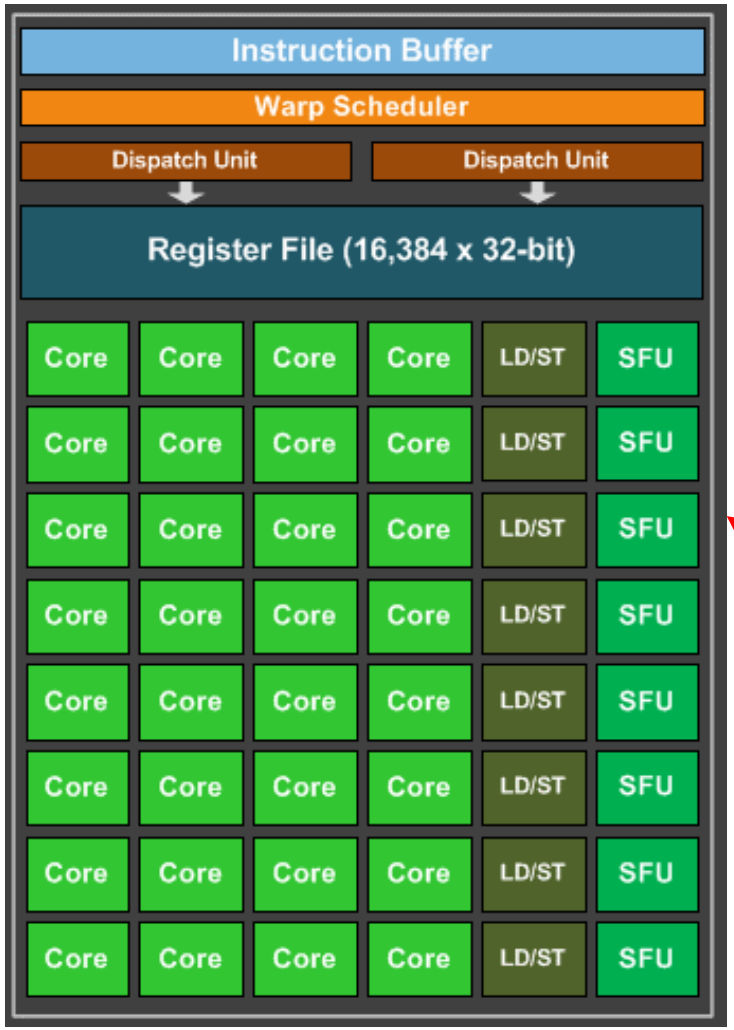
- SPA
  - Streaming Processor Array
  - For G80 series, it may have 1 to 8 TPCs
- TPC
  - Texture Processor Cluster (2 SMs + TEX)
  - TEX: Texture processor, for graphics purpose
- SM
  - Streaming Multiprocessor (8 SPs)
  - Multithreaded processor core
  - Fundamental processing unit for CUDA thread block
- SP (or CUDA Core)
  - Streaming Processor
  - Scalar ALU for a single CUDA thread

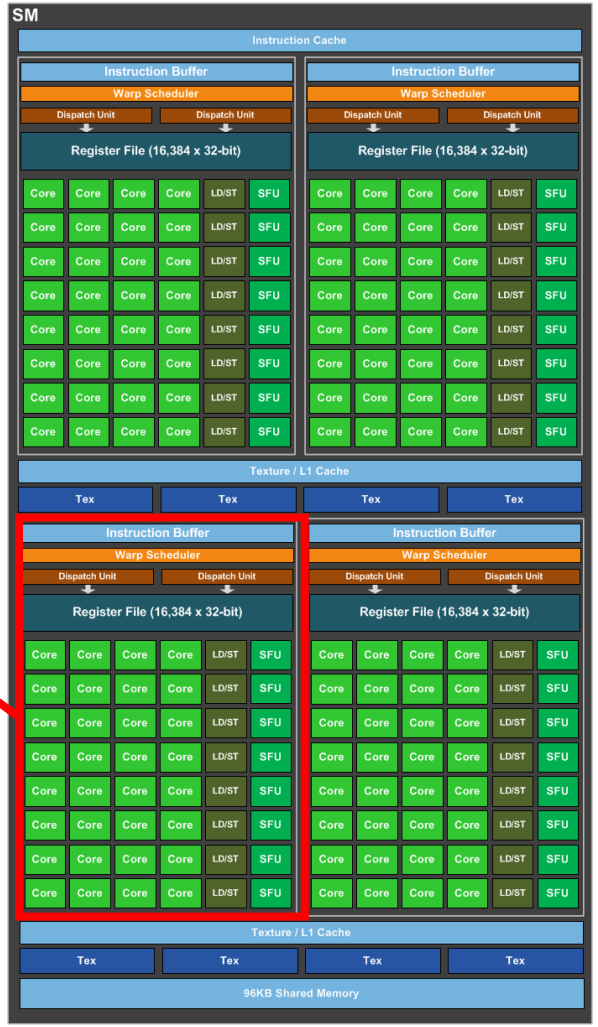
CUDA Architecture Improvements

- SFU
  - Special Functions Unit
    - For complex math functions: sin, cos, square root, …
  - Executes one special instruction per thread, per clock
- Recent CUDA architectures show:
  - TPC may have only one SM and only one TEX
    - For more speedup
  - Up to 32 SPs in a single SM
    - But still SM is the fundamental processing unit

Nvidia Pascal SM

- Big SM → 4 SMs in a single unit
  - Each SM = 32 cores + 8 SFU + 2 TEX





Transparent Scalability

- Scalability Issue:
  - SM is the fundamental processing unit.
  - But, a CUDA device may have various number of SMs
    - A single SM for low-tier tablets / smart phones
    - Up to 256 or more SMs for high-end desktops

- CUDA solution?
  - Block as a unit for SM processing

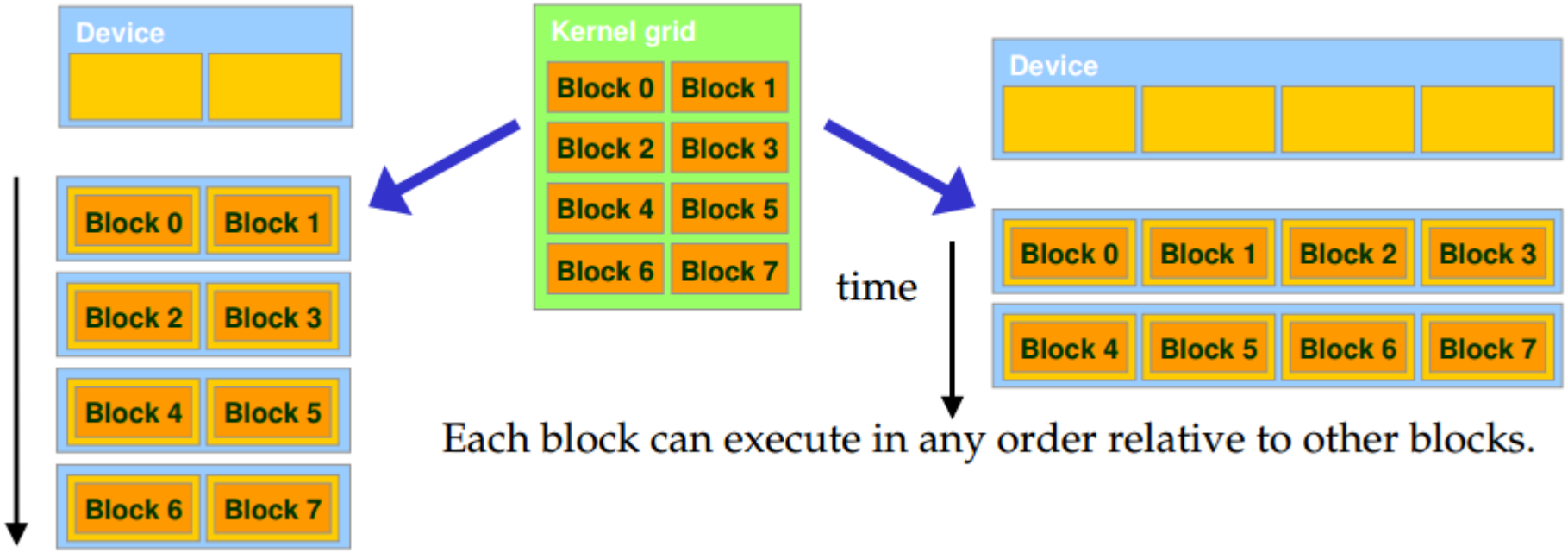
Transparent Scalability

- Scalability Issue:

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT, cuBLAS, cuRAND, cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL, SVM, OpenCurrent	PhysX, OptiX, iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java, Python, Wrappers	DirectCompute	Directives (e.g., OpenACC)	
CUDA-enabled NVIDIA GPUs						
Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series		
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series		
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series		
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series		
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series		
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER		

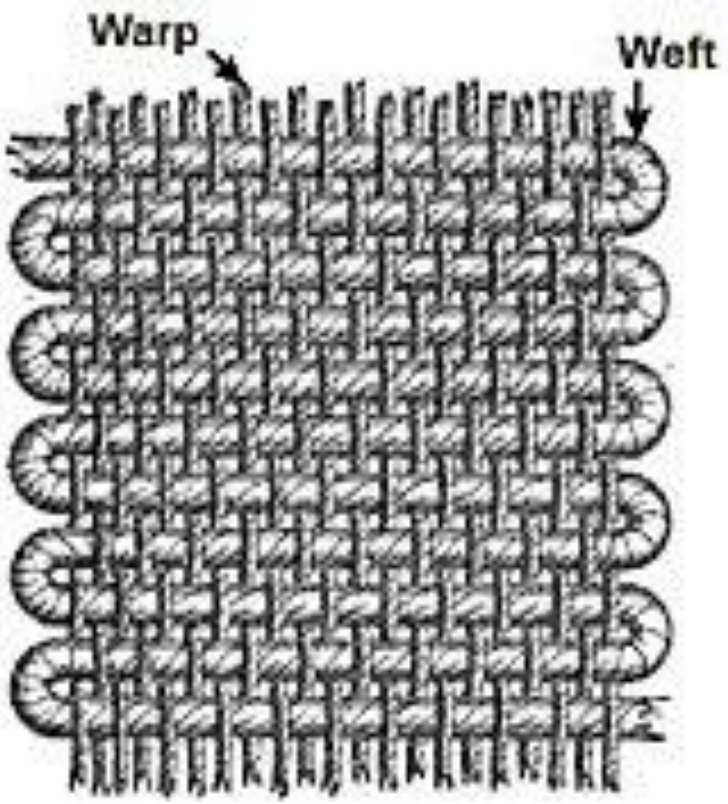
Transparent Scalability

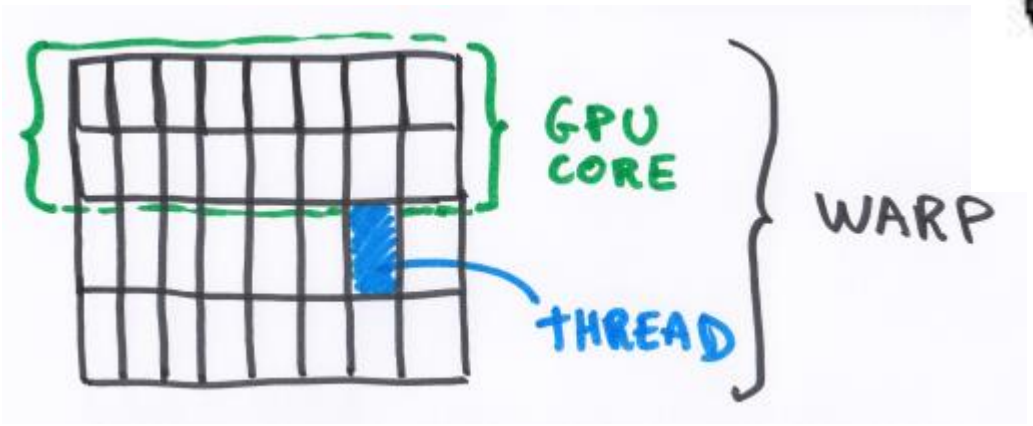
- Hardware is free to assign blocks to any processor at any time
  - A kernel scales across any number of parallel processors



Thread and Warp

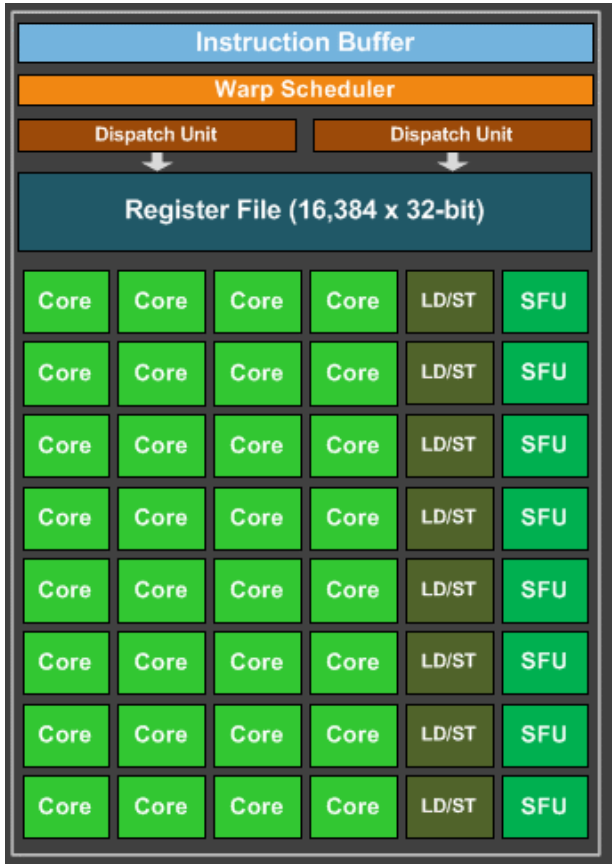
- Thread





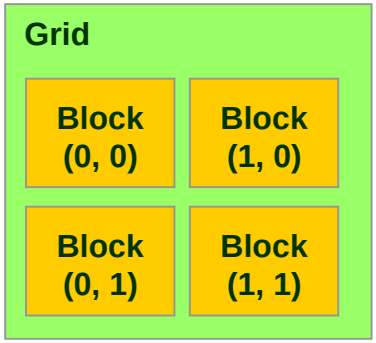
Streaming Multiprocessor (SM)

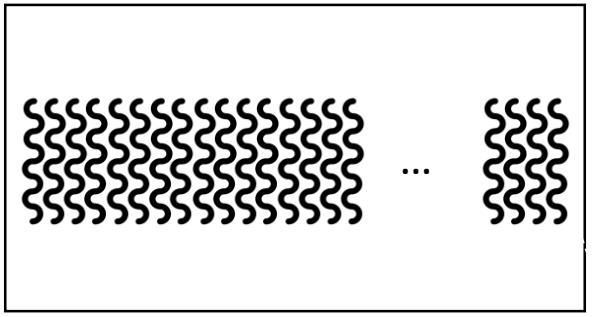
- Each SM = 32 cores + 8 SFU + 2 TEX
- Streaming Multiprocessor (SM)
  - 32 Streaming Processors (SP)
  - 8 Special Function Units (SFU)
- Multi-threaded instruction dispatch
  - 1 to 2048 threads active
  - Shared instruction fetch per 32 threads
  - Cover latency of texture/memory loads



CUDA Thread Block

- Programmer declares (Thread) Block:
  - Block size 1 to 1024 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Kernel Program
  - All threads execute **the same kernel program**
  - Threads have thread ID numbers within Block
  - Thread program uses thread ID to select work and address shared data

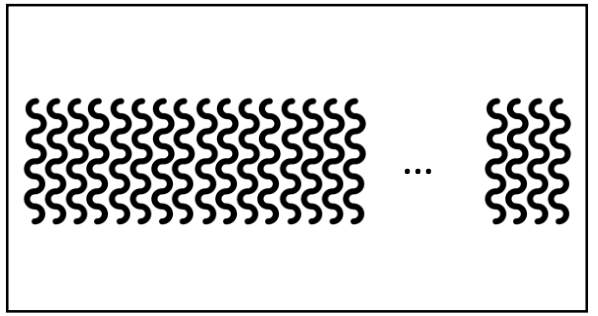




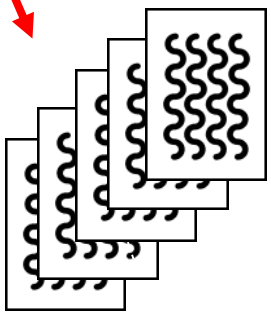
Thread block: 1 to 1024 threads

Thread Scheduling/Execution

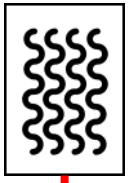
- Threads run concurrently
  - SM assigns/maintains thread ID #
  - **SM manages/schedules thread execution**
- Each thread block is divided in **32-thread Warps**
  - This is an implementation decision, not part of the CUDA programming model



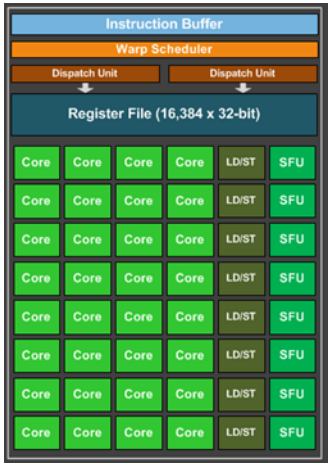
Thread block: 1 to 1024 threads



divided into warps



1 warp = 32 threads

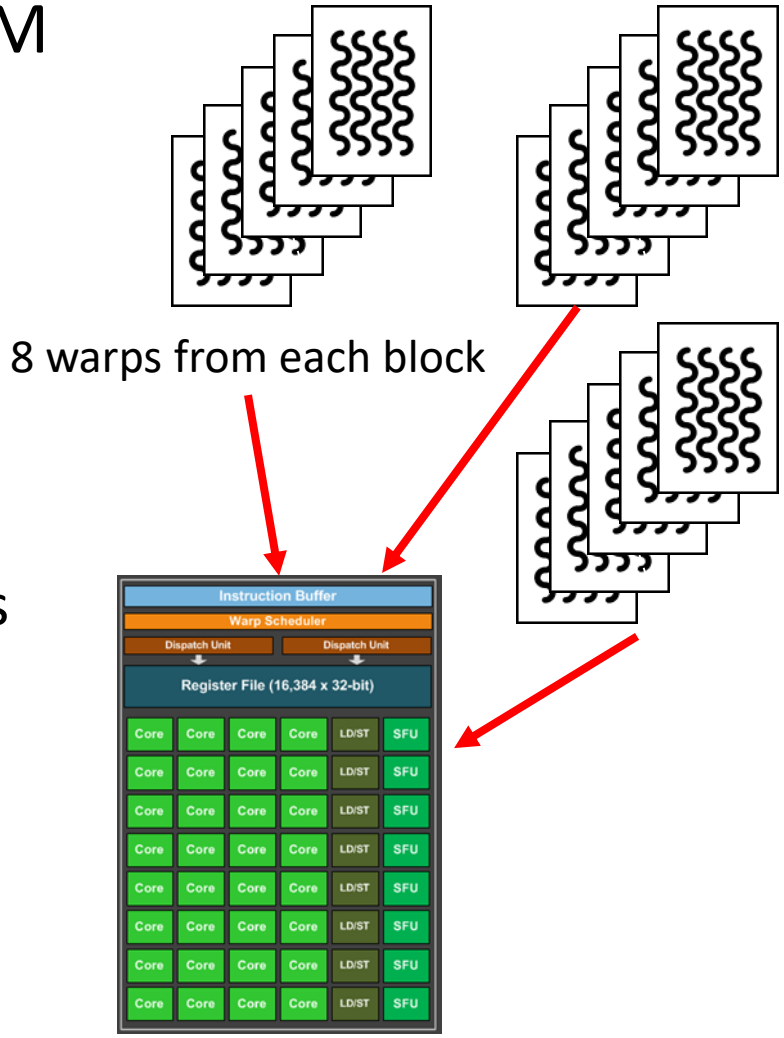


1 SM = 32 cores → Parallel Execution!

Thread Scheduling/Execution

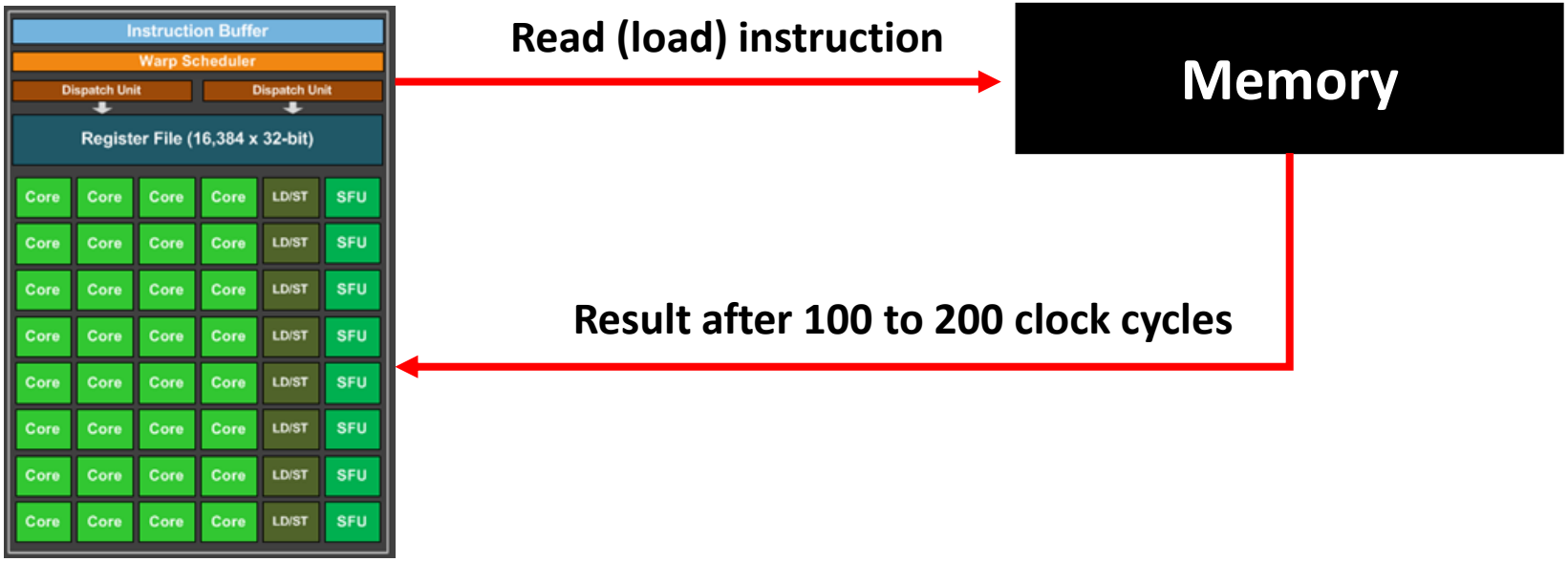
- Warps are scheduling units in SM

- A scenario
  - 3 blocks to an SM
  - Each block has 256 threads
- How many warps?
  - Each block has 256 / 32 = 8 warps
  - SM has 3 * 8 = 24 warps
  - At any point in time, only one of the 24 warps will be selected for instruction fetch and execution



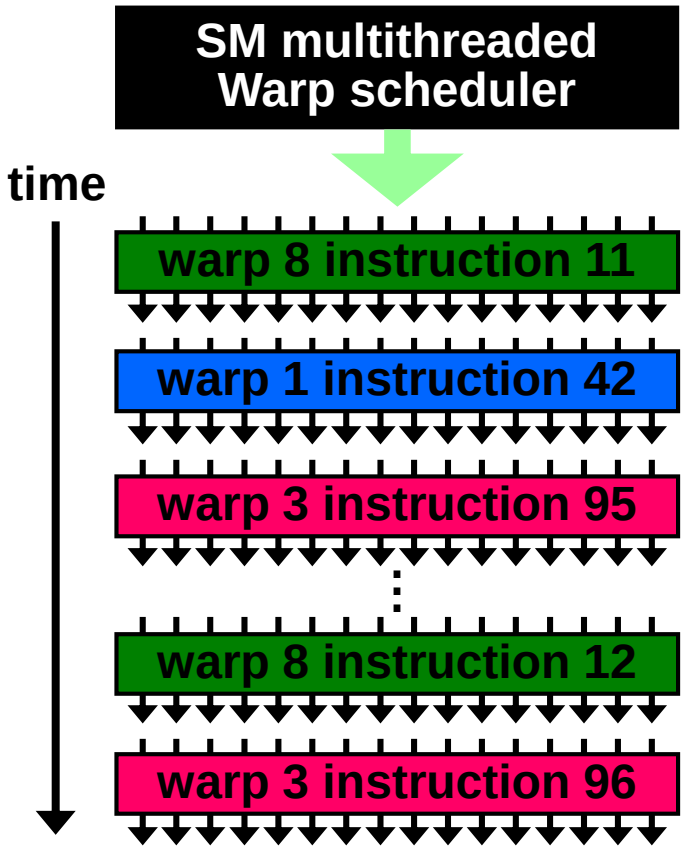
SM Warp Scheduling

- All threads in a Warp execute the same instruction when selected
  - Only one control logic for an SM
- Memory access → latency problem → scheduling required!



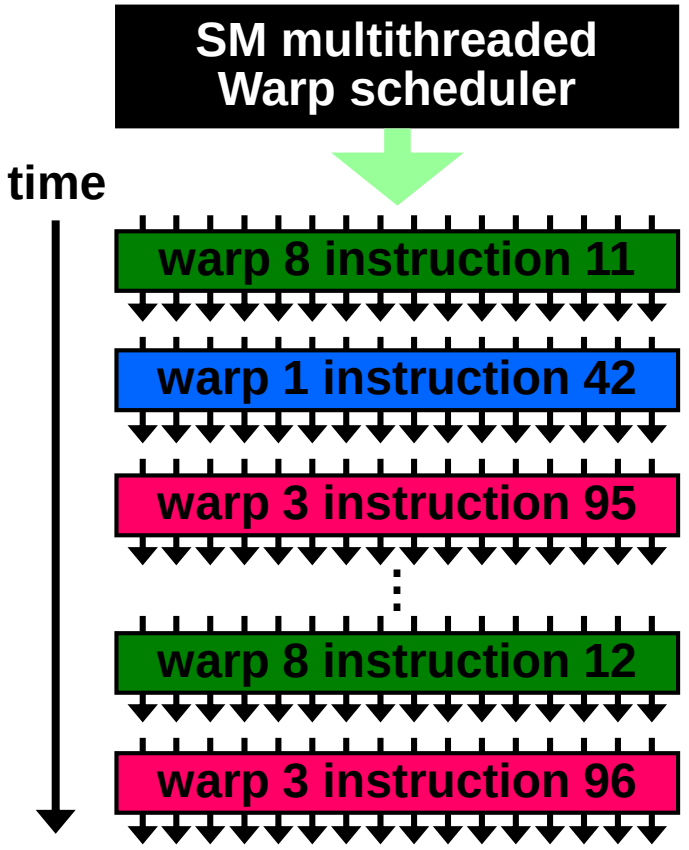
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy



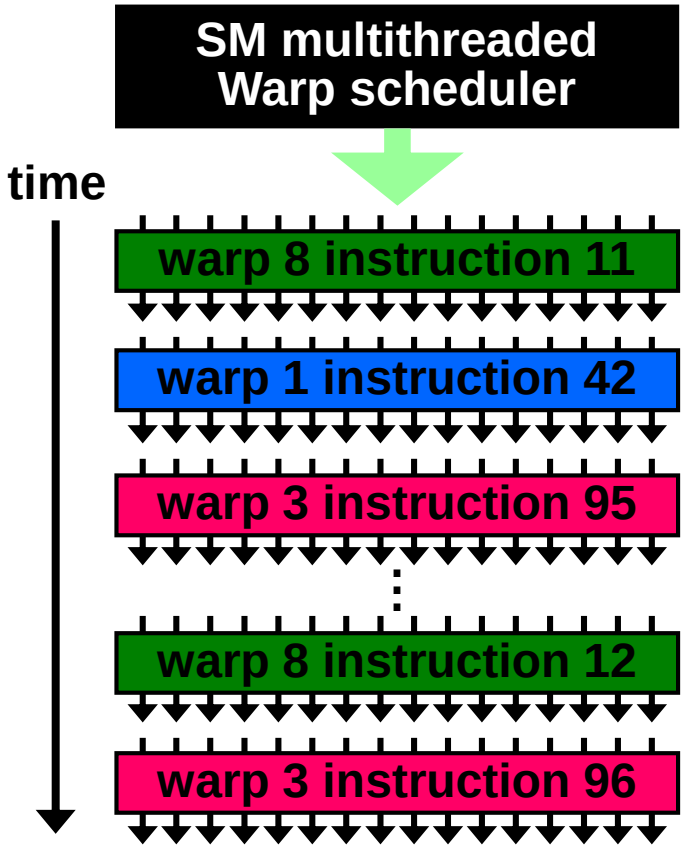
SM Warp Scheduling

- New Architecture:
- An SM = 32 SPs
  - They assign 1 threads per SP
- 1 clock cycle needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 25 Warps are needed to fully tolerate 100-cycle memory latency



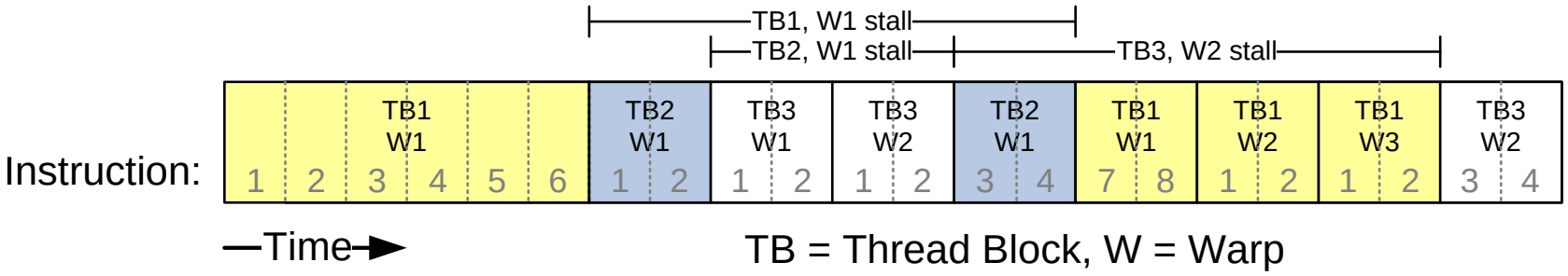
SM Warp Scheduling

- Old Architecture:
- An SM = 8 SPs
  - They assign 4 threads per SP
- 4 clock cycle needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency



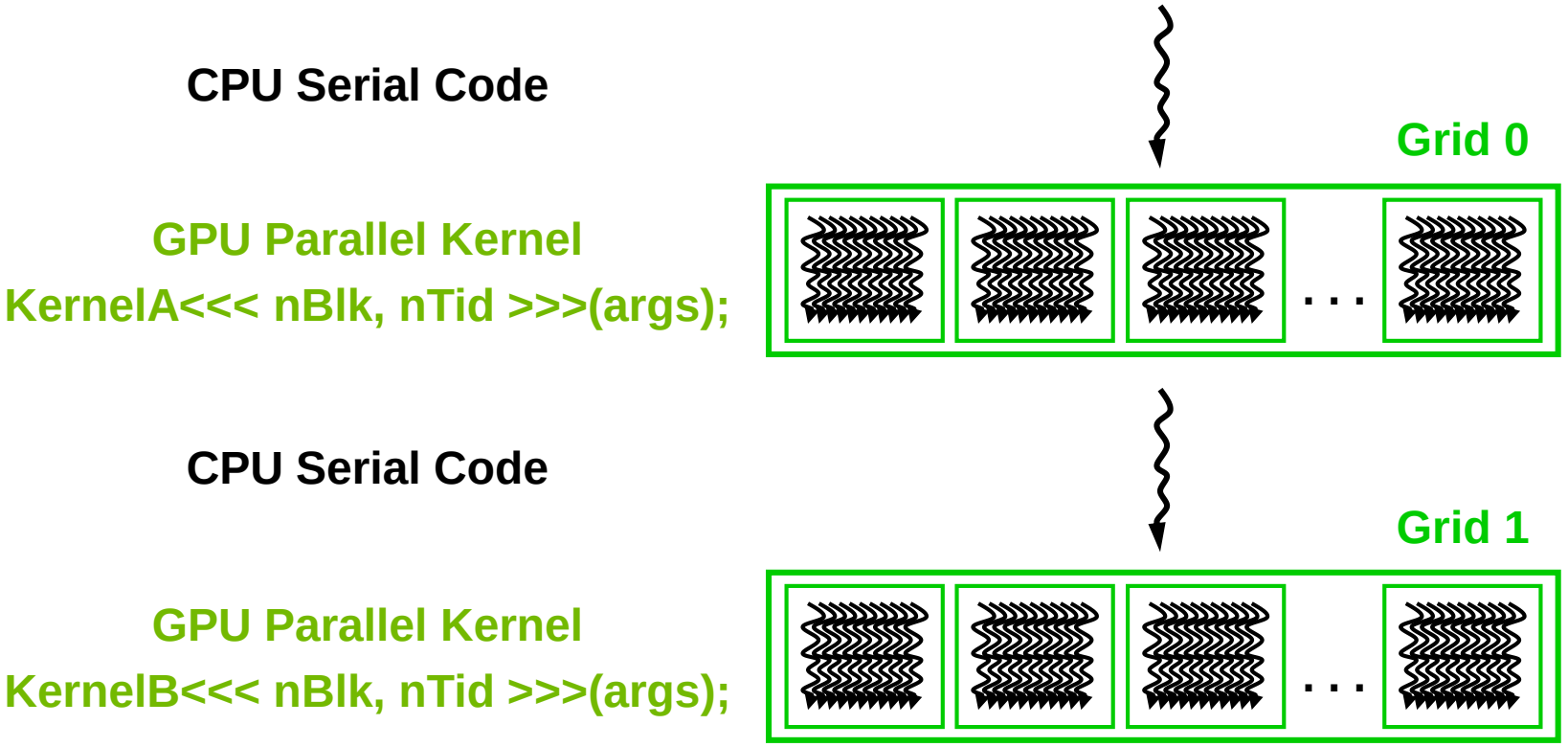
Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes ready after the needed values are deposited
  - Prevents hazards



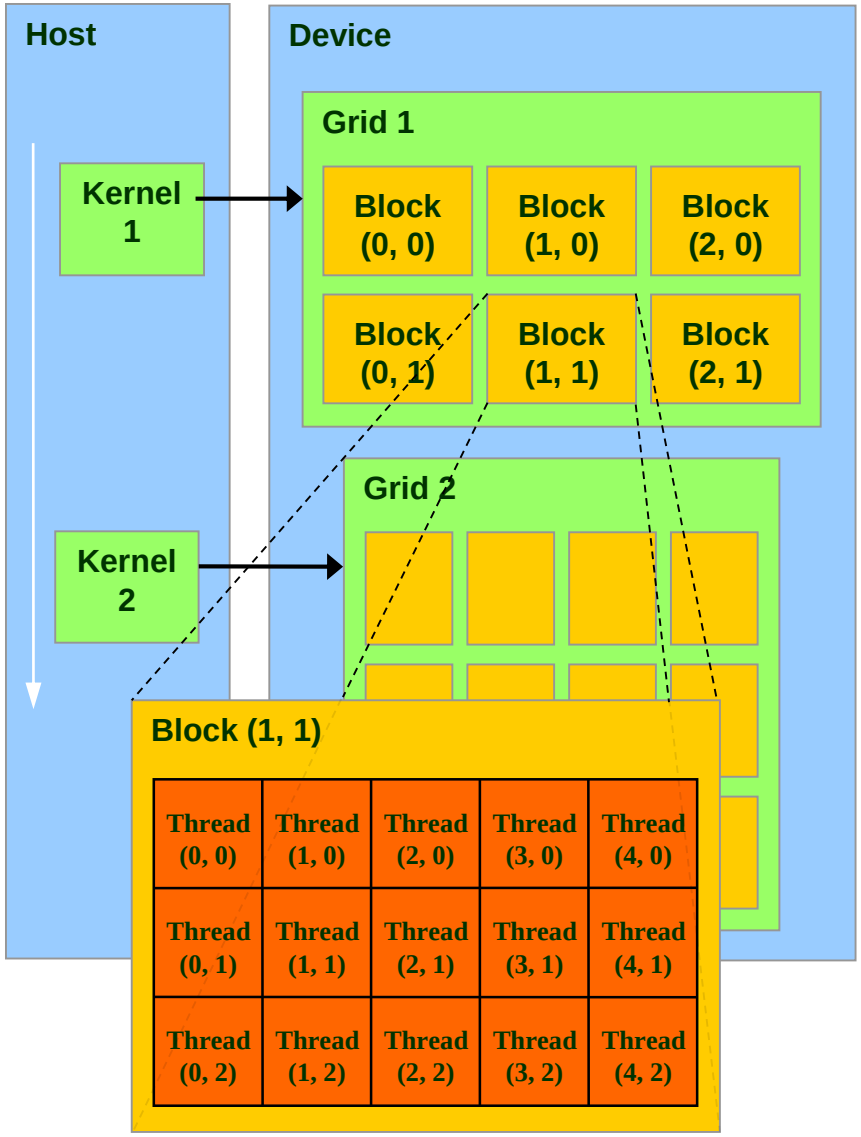
Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C Program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



Thread Life Cycle in HW

- Grid is launched on the SPA
- **Thread Blocks** are serially distributed to all the SM's
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
  - 2 levels of parallelism
- SM schedules and executes **Warps** that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can distribute more Thread Blocks



Thread Life Cycle in HW

- Threads are assigned to SMs in **Block granularity**
  - Up to 32 blocks to each SM as resource allows
  - SM in 1070 can take up to 2048 threads
  - Could be 256 (threads/block) * 8 blocks,
    or 512 (threads/block) * 4 blocks,
    or 1024 (threads/block)* 2 blocks, etc.

Intermediate Summary

- CUDA Thread and GPU

- CUDA Hardware
  - G80 architecture
  - Pascal architecture
- Transparent Scalability
- Thread and Warp
- CUDA thread scheduling
  - SM warp scheduling
- Overall execution model

- Next step?
  - Real example → matrix multiplication

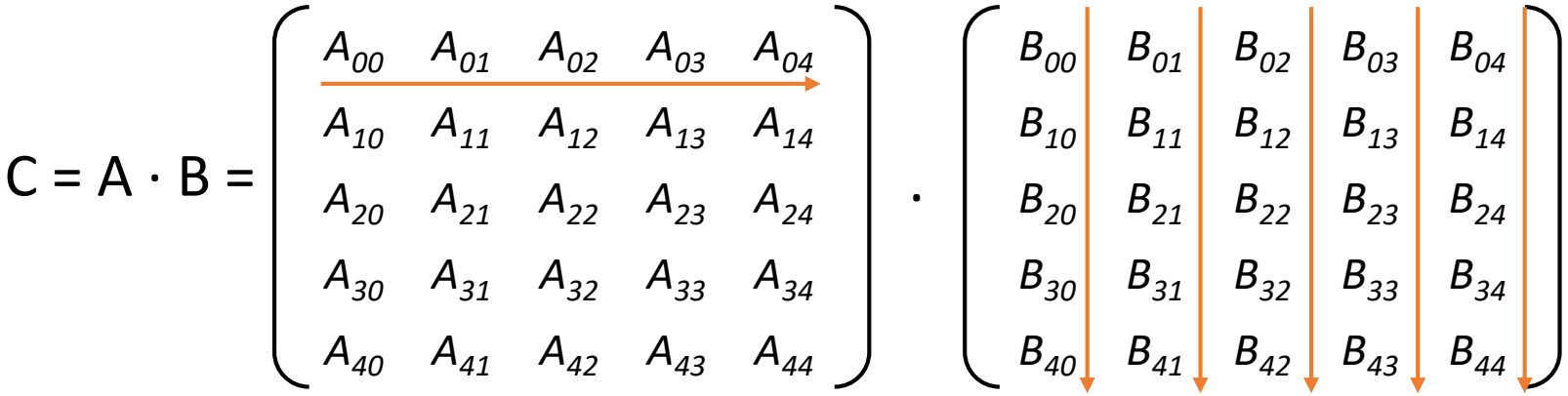
Contents

- Tiled Matrix Multiplication

- Review for the problem
- Thread Layout – single block
- Thread Layout – multiple blocks
- Index Values – local and global
- Kernel Function

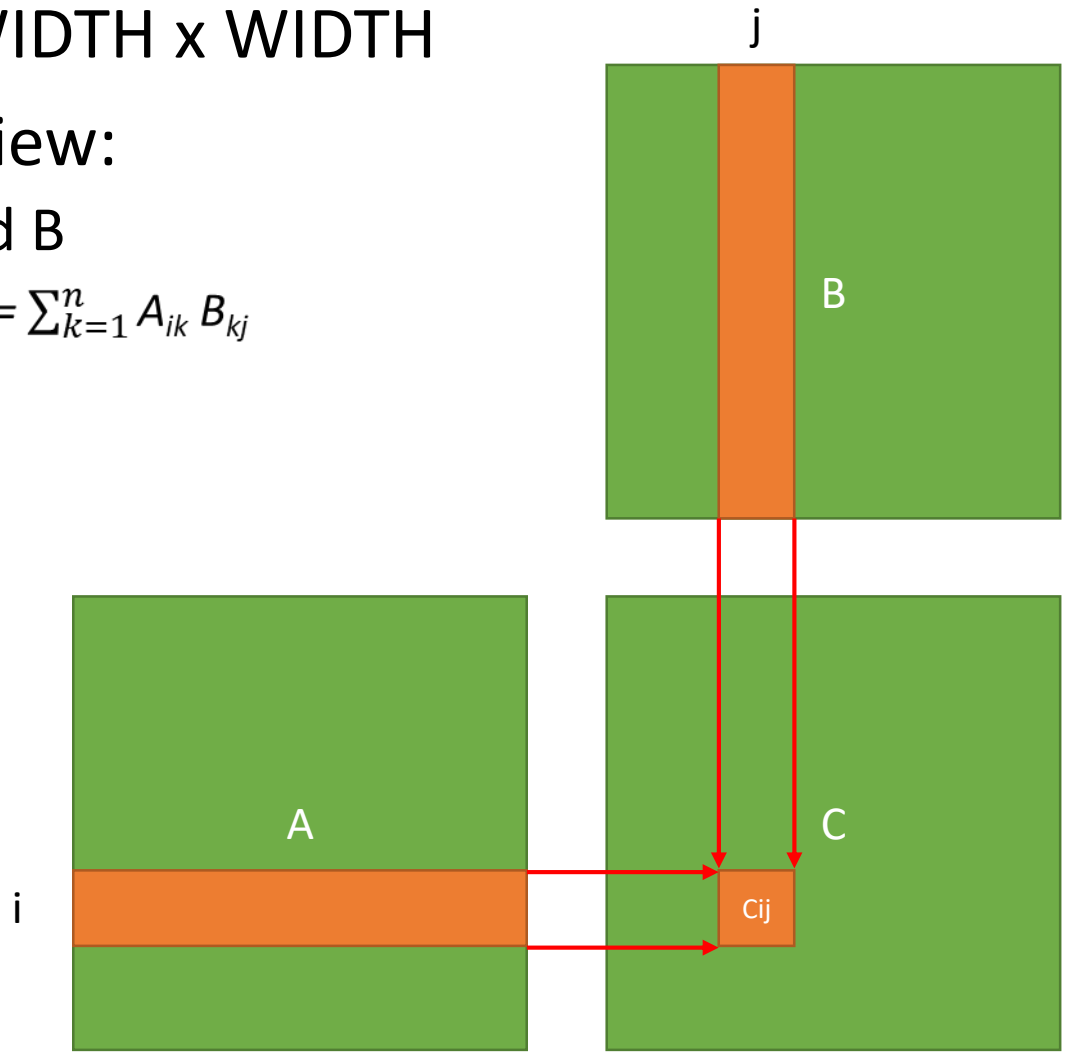
Example: Matrix Multiplication

- We want to multiply matrix A and matrix B
  - For simplicity, we assume square matrices



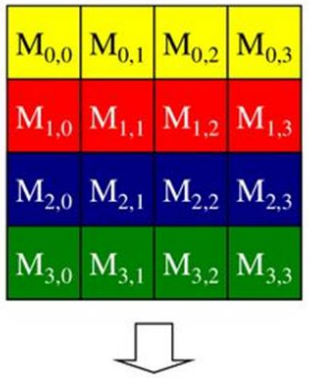
Example: Matrix Multiplication

- C = A*B of size WIDTH x WIDTH
- Memory usage view:
  - Read from A and B
  - To calculate $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$
- Using:
  - A[i*WIDTH + k]
  - B[k*WIDTH + j]



Row-major Matrix Layout in C/C++

- Logical layout:



- Physical layout: 1D array

  M = &(M[0][0]) M[y][x]

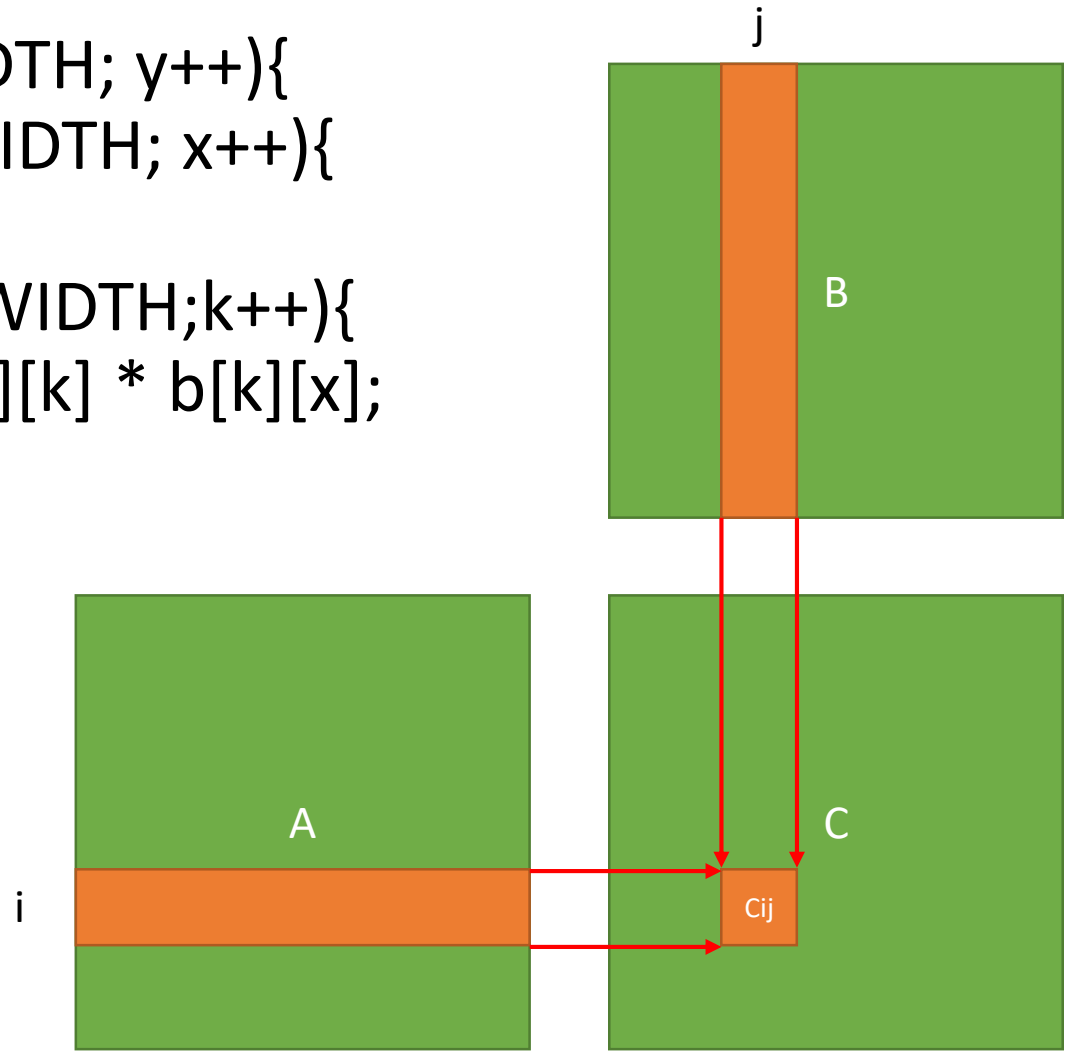
- Re-interpret

  M = cudaMalloc (…) M[y*WIDTH + x]

CPU version: triple for-loop

- Calculation code:

```
for (int y = 0; y < WIDTH; y++){
   for (int x = 0; x < WIDTH; x++){
         int sum=0;
         for (int k=0;k<WIDTH;k++){
               sum += a[y][k] * b[k][x];
          }
          c[y][x] = sum;
    }
}
```



CUDA version: kernel function

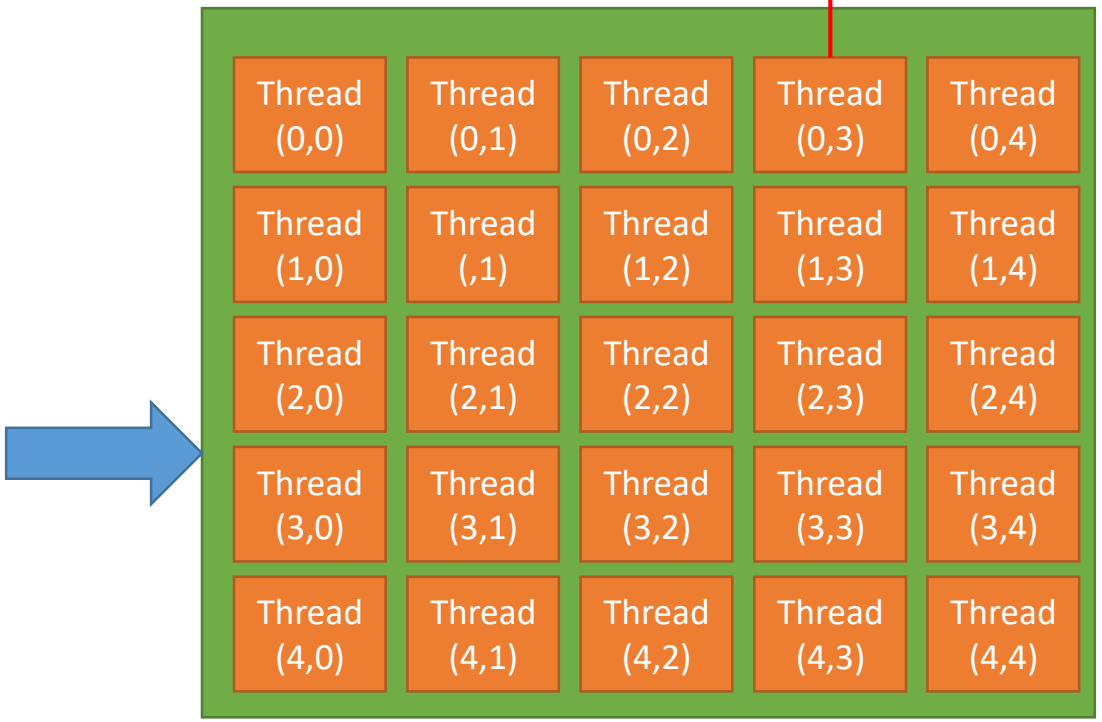
- Kernel code

```
__global__ void mulKernel( … ){
…
    for (int k=0;k<WIDTH;k++){
        sum += a[y*WIDTH+k] * b[k*WIDTH+x];
    }
    c[i] = sum;
}
```

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

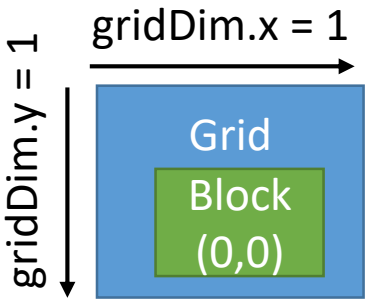
Each thread

$$\begin{pmatrix} C_{00} & C_{01} & C_{02} & C_{03} & C_{04} \\ C_{10} & C_{11} & C_{12} & C_{13} & C_{14} \\ C_{20} & C_{21} & C_{22} & C_{23} & C_{24} \\ C_{30} & C_{31} & C_{32} & C_{33} & C_{34} \\ C_{40} & C_{41} & C_{42} & C_{43} & C_{44} \end{pmatrix}$$

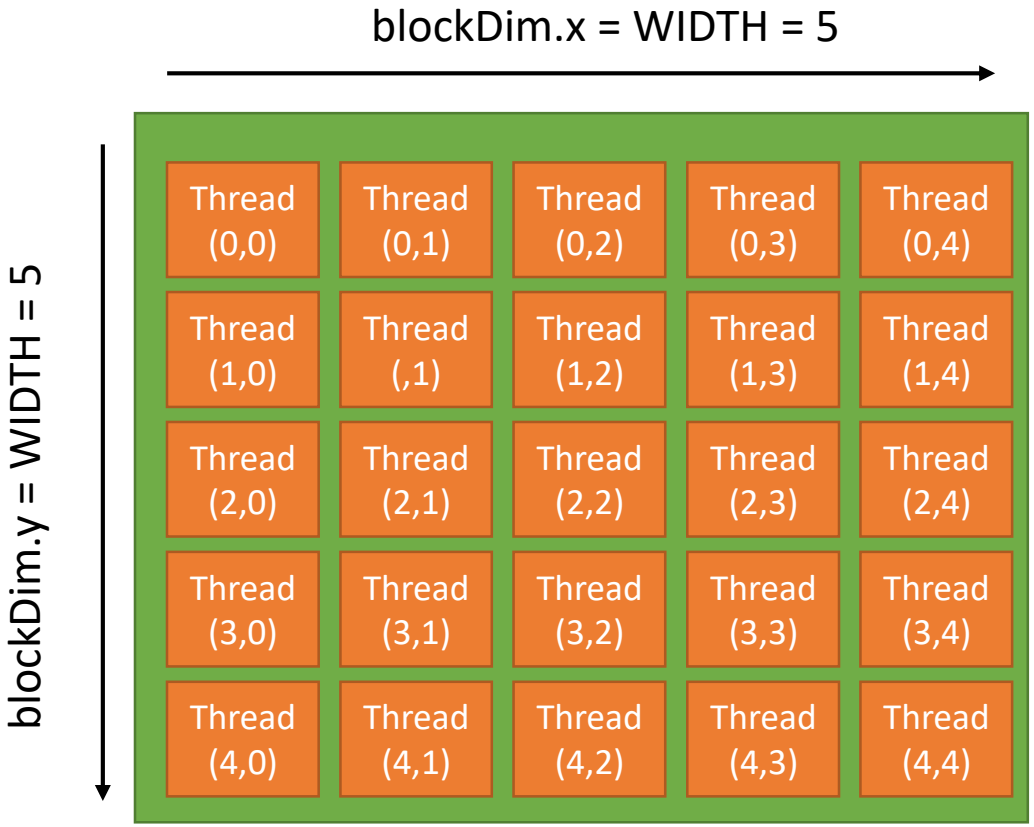


Thread Layout

- Matrix → 2D Layout
- Small size matrix → a single block!



$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$$

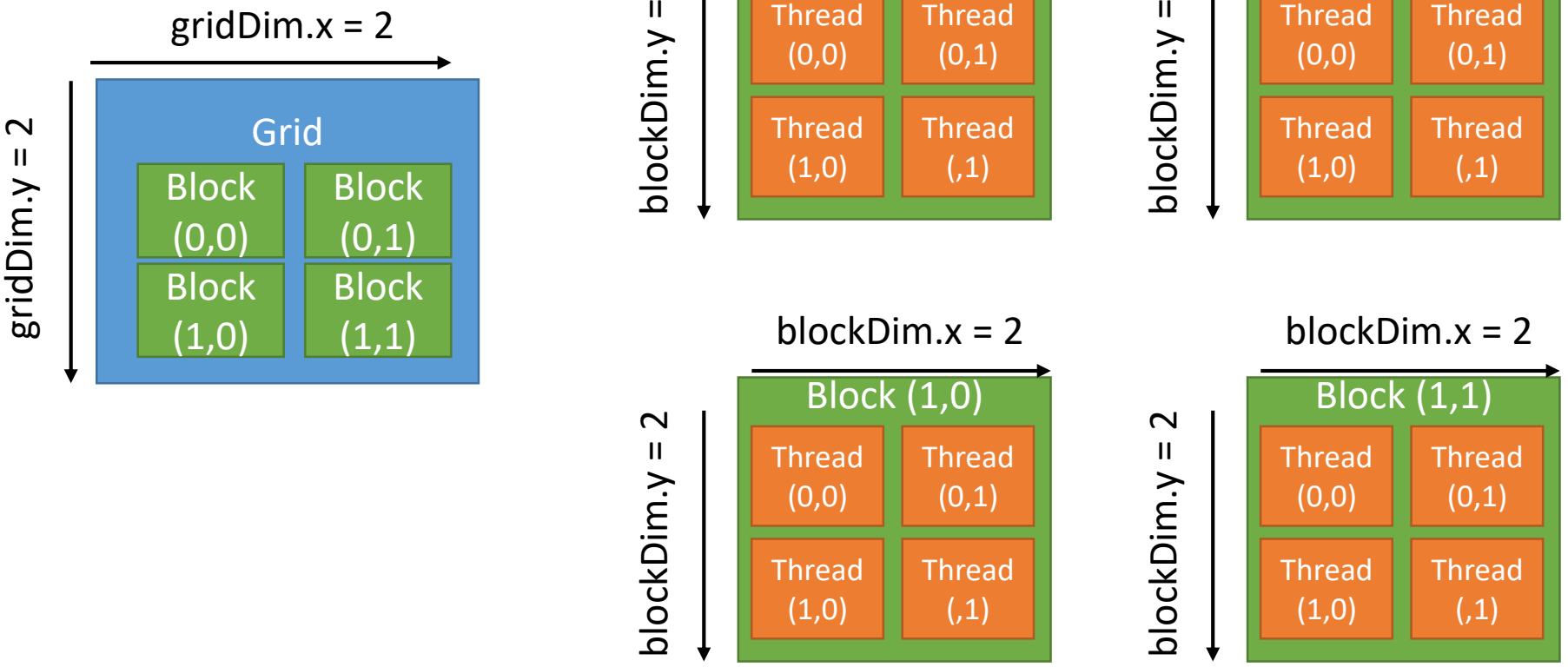


Any Problem?

- We used only one thread block..
- Each thread block can execute at most 1024 threads
  - Some old architecture can execute only 256 or 512 threads
  - In the future, it can be enlarged to 4096 and more?
- So maximum matrix size is …
  - 32 x 32 = 1024
  - With a single thread block
- Solution?
  - Use multiple thread blocks!

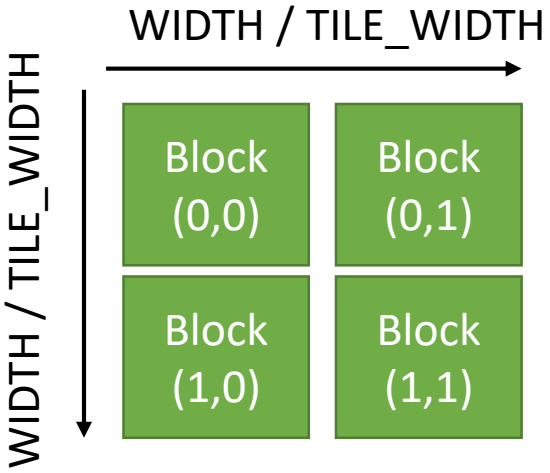
Thread Layout

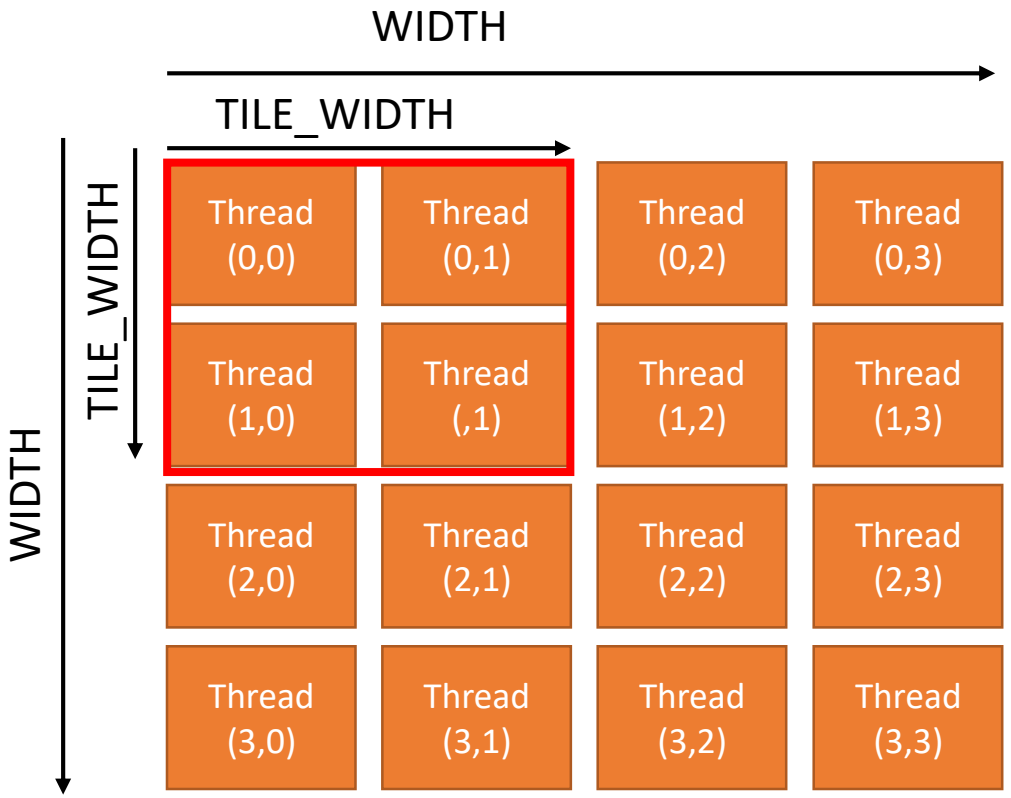
- Matrix → 2D Layout
- Tiled approach: use multiple blocks



Simplified Thread Layout

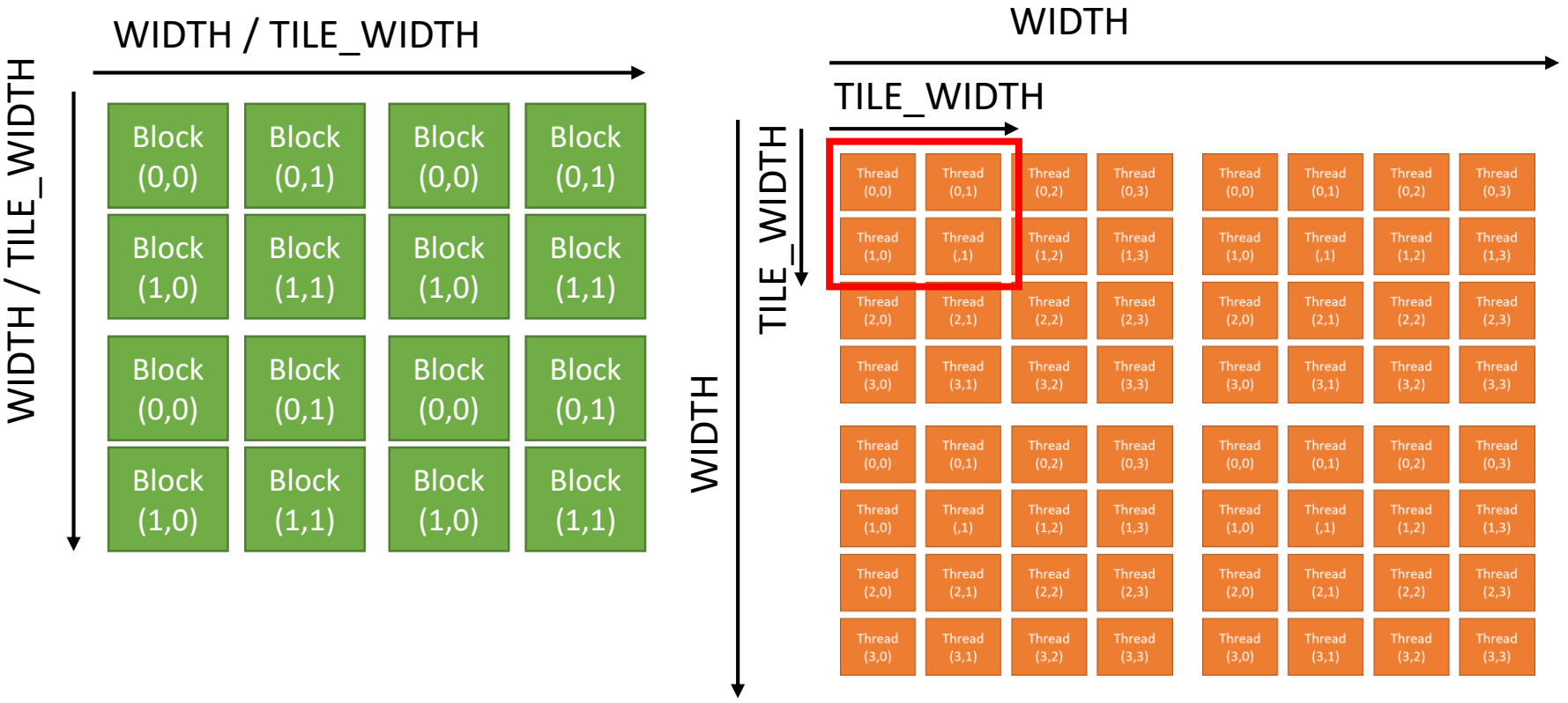
- Assumption: Square matrices
- Global size: WIDTH x WIDTH
- Tile → a block: TILE_WIDTH x TILE_WIDTH
- Grid: (WIDTH / TILE_WIDTH) x (WIDTH / TILE_WIDTH)





Another Example

- WIDTH = 8
- TILE_WIDTH = 2
- Block dimension = 4 x 4



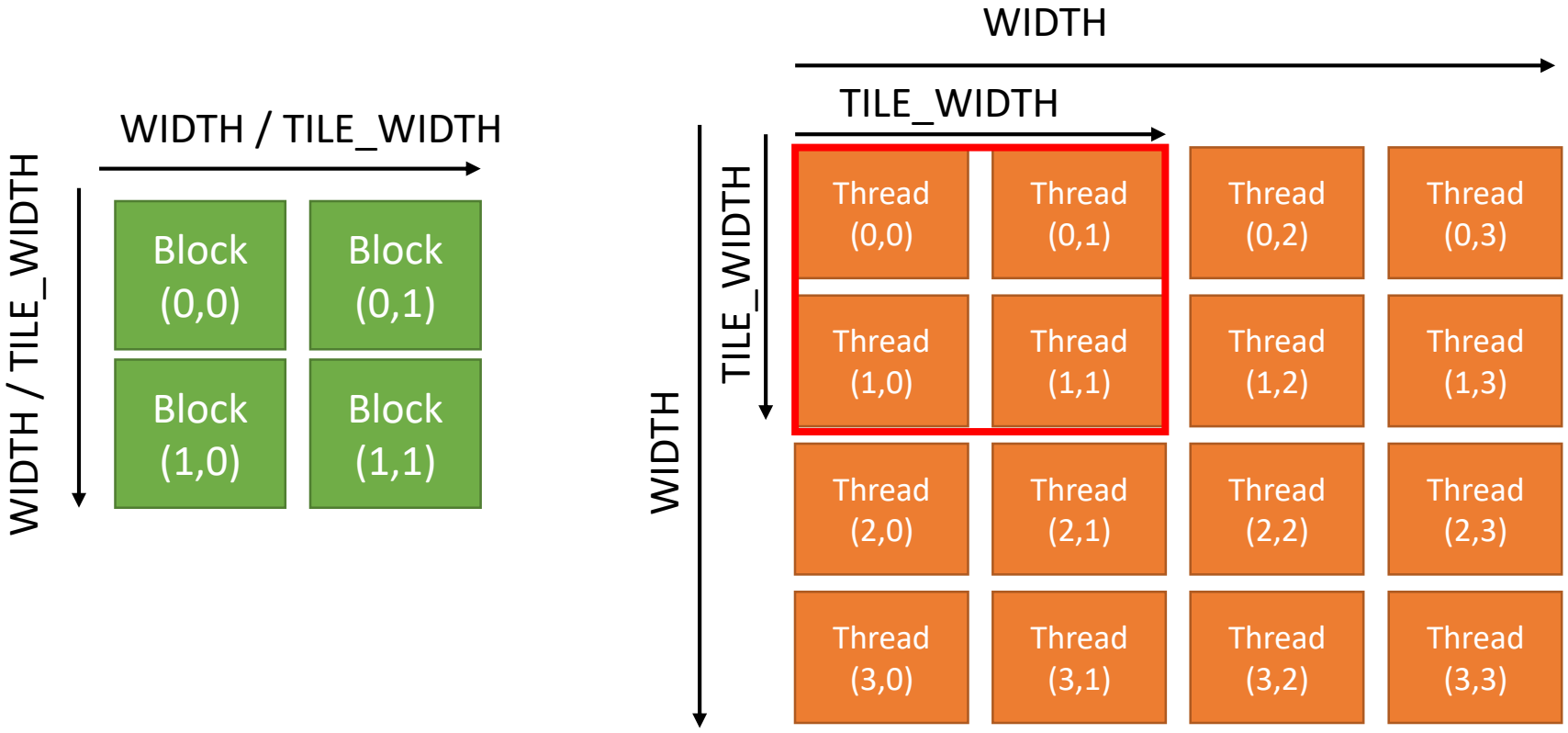
Kernel Launch

```
const int WIDTH = 8;

const int TILE_WIDTH = 2;


// Setup the execution configuration

dim3 dimGrid (WIDTH/TILE_WIDTH, WIDTH/TILE_WIDTH, 1);

dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);


// Launch the device computation threads
MatrixMulKernel <<<dimGrid, dimBlock>>> (dev_c, dev_a, dev_b, WIDTH);
```

- How to get the index values?
  - Threadidx
  - Blockidx

Simplified Thread Layout Again

- Assumption: Square matrices
- Global size: WIDTH x WIDTH
- Tile → a block: TILE_WIDTH x TILE_WIDTH
- Grid: (WIDTH / TILE_WIDTH) x (WIDTH / TILE_WIDTH)



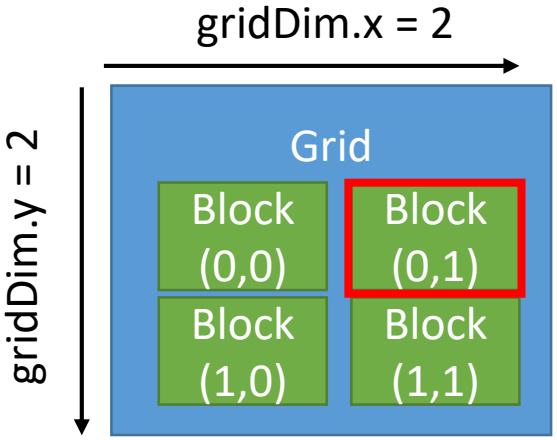
Local Index

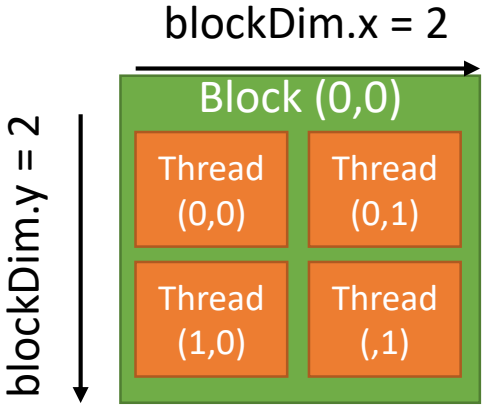
- For block (0,1)
  - blockIdx.y = 0
  - blockIdx.x = 1
- Local index
  - threadIdx.y = 1
  - threadIdx.x = 0

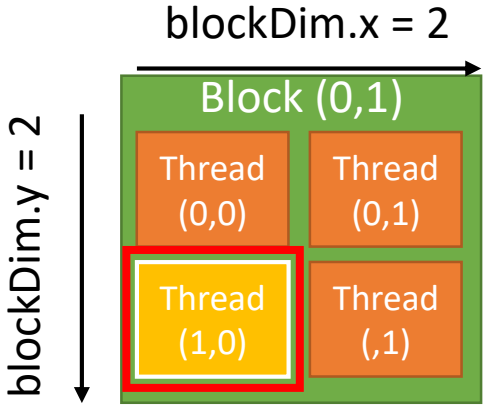
- Global index

y = blockIdx.y * blockDim.y + threadIdx.y
0 * 2 + 1 → 1

x = blockIdx.x * blockDim.x + threadIdx.x
1 * 2 + 0 → 2



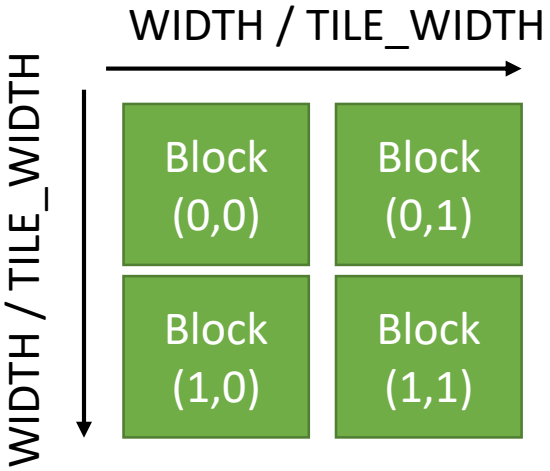


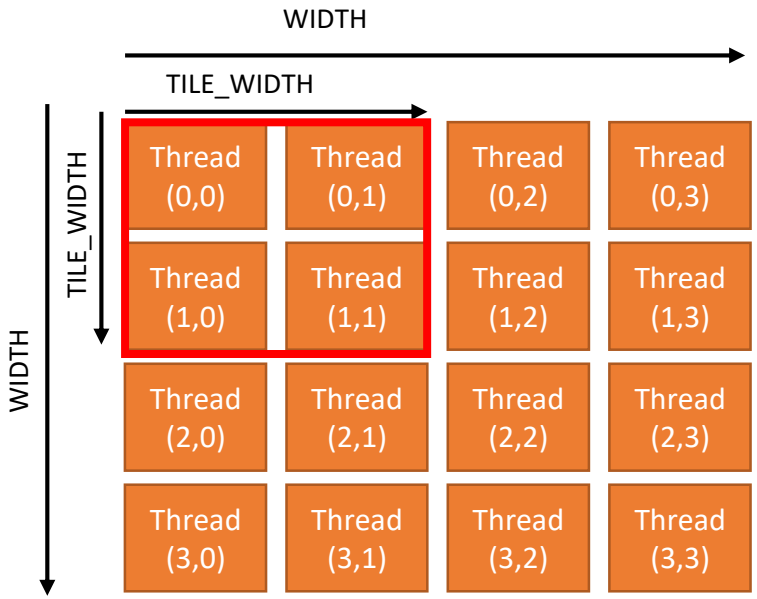


Indices for Block 0,0

- Global index
  y = blockIdx.y * blockDim.y + threadIdx.y
  0 * 2 + 1 → 1
  x = blockIdx.x * blockDim.x + threadIdx.x
  0 * 2 + 0 → 2

- In the block 0,0
  - y = 0 * 2 = threadIdx.y
  - x = 0 * 2 + threadIdx.x





Assignment

- Tile approach-based Matrix multiplication
  - 16x16 Matrix multiplication
  - Use TILE_WIDTH (but, 1<TILE_WIDTH<16).
    - For simplicity, it would be better to select TILE_WIDTH among 2,4, and 8.

1) Draw the structure of grid/block/thread reflecting TILE_WIDTH

2) Write your CUDA code using WIDTH and TILE_WIDTH

3) Print the results

Thank you

Any questions?

E-mail: yhgong@kw.ac.kr
Lab: https://sites.google.com/view/yhgong/

광운대학교
KwangWoon University