EE/CSCI 451: Parallel and Distributed Computation

Lecture #15

10/8/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

Announcement

- PHW4 due on 10/9 (Friday)
- HW6 due on 10/9 (Friday)
- PHW5 and HW7 will be out tomorrow
  - PHW5 due on 10/22
  - HW7 due on 10/16
- Project Proposal due on 10/16
  - Remember to fill your group info here
- Midterm 2
  - Time: Oct. 23 (Friday) 3:30-5:30 PM
  - Covers material from Sept. 22 to Oct. 16
  - Logistics: same as Midterm 1

Course Project

- Project Proposal Template
  - Introduction
    - Background on the problem
    - Contributions/hypotheses of the project -- **parallelization, scalability** should be the focus
  - Description of the approach and implementation
    - Parallelization techniques if proposing a new technique
    - Selection criteria for benchmarks if comparing parallel algorithms
    - Description of the application if developing an application
    - Implementation platform and resource requirements
  - Evaluation methodology/Criteria for success
    - Datasets, evaluation metrics, and baselines for comparison
- We will evaluate the proposals and give feedback

Course Project

- Proposal Format
  - 12 pt font size
  - Single-column
  - Single space
  - Cite relevant papers
  - Length: 2-4 pages including references & appendix(if any)
  - A template proposal can be found Piazza under Resources-Course Project

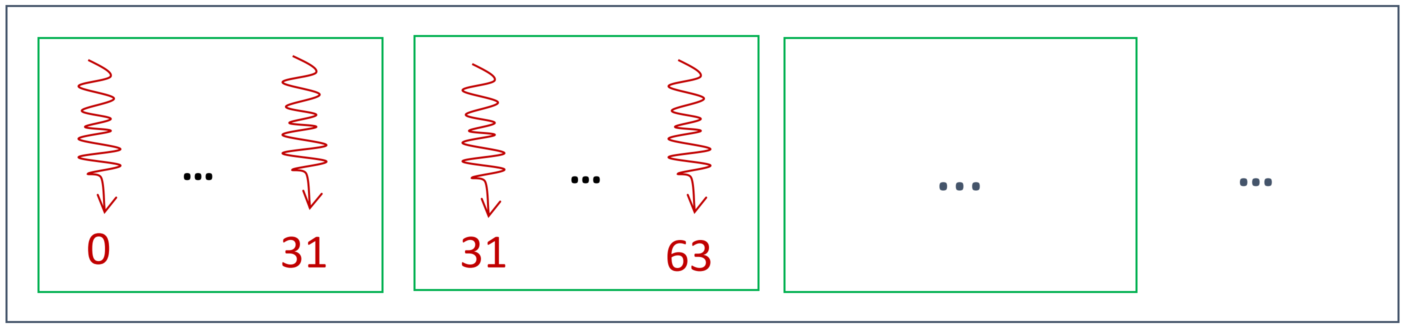
Outline

- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
  - CUDA Programming Model
  - GPU Architecture
  - Execution Model
  - Performance Model
  - Examples
  - CUDA, OpenCL, …
  - APU, Zynq…

Warp

- Basic unit of execution on GPU
- 32 threads form a Warp
    - share the same program counter (PC)
- Threads within a Warp execute the same instruction in each clock cycle (SIMT)

Thread block



Basic Organization

Typical organization

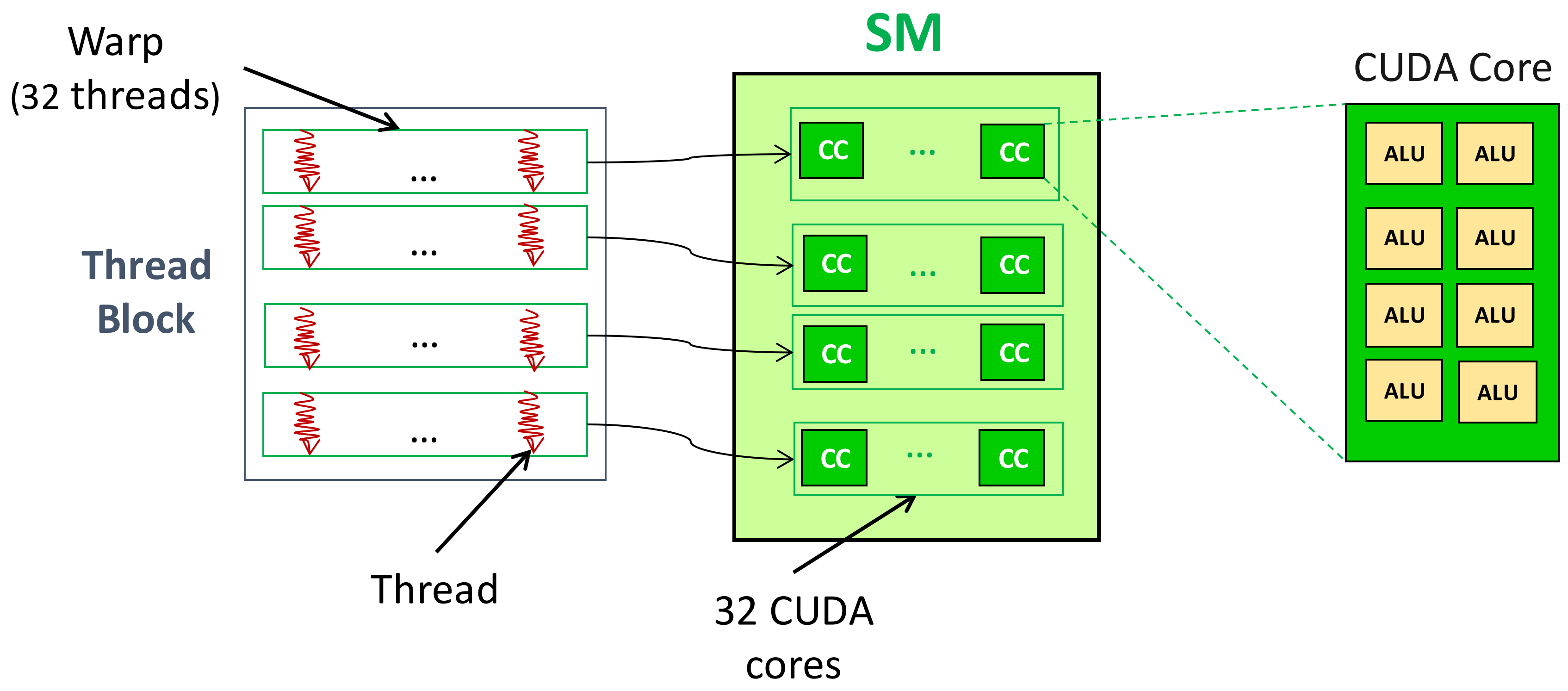
16 SM (Streaming Multiprocessors)

128 CC/SM → Up to 4 Warps can be concurrently executed on each SM

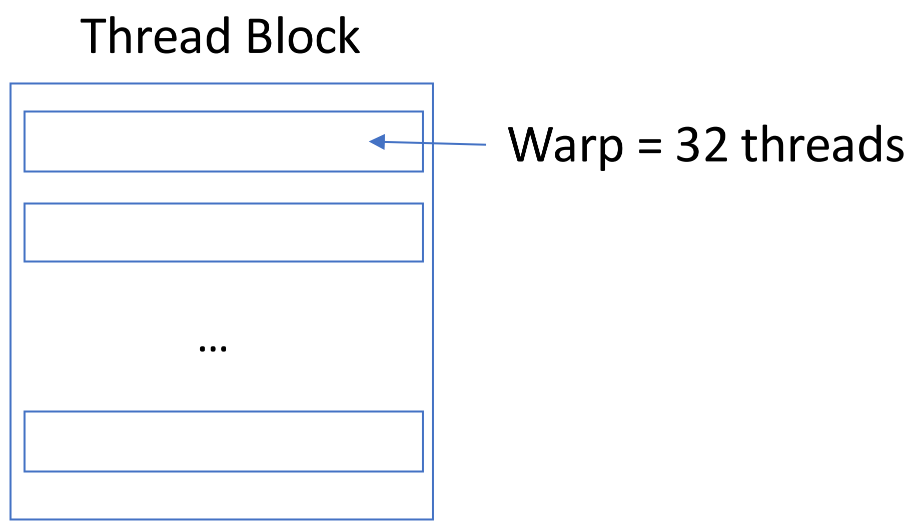
8 ALU/CC SIMT – (single instruction, multiple threads)

SIMT Execution Model (1)

- Threads within the same thread block are mapped onto the same streaming multiprocessor
- Each Warp is executed by set of CUDA cores in SIMD fashion



SIMT Execution Model (2)



- SIMD within each Warp
- Each Warp is executed by collection of CC
- No guarantee all the Warps are executing the same instruction at the same time
- SIMT programming model does NOT guarantee SIMD execution across all the threads in the block

SIMT Execution Model (3)

- Single Instruction, Multiple Thread
- Thread block is partitioned into groups of 32 threads (Warp)
  - Always executing the **same** instruction
  - Shared instruction fetch/dispatch
  - Some threads become inactive when code path diverges
  - Hardware automatically handles divergence
- Warps are the primitive unit of scheduling
  - Largely invisible to the programmer
  - Must understand for performance [not correctness ?]

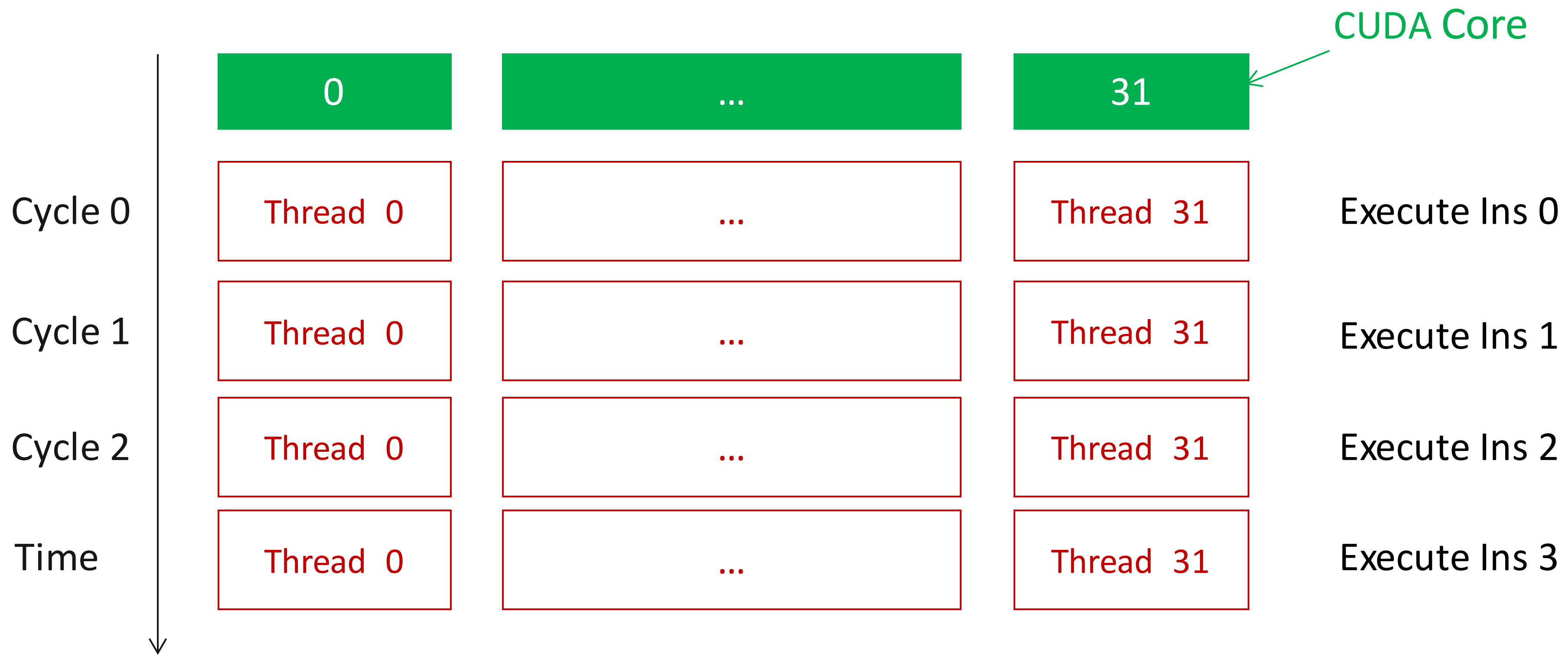
SIMT Execution Model (4)

- SIMT is a virtualized execution of SIMD
- # of threads can be > # of hardware units
  - For example:

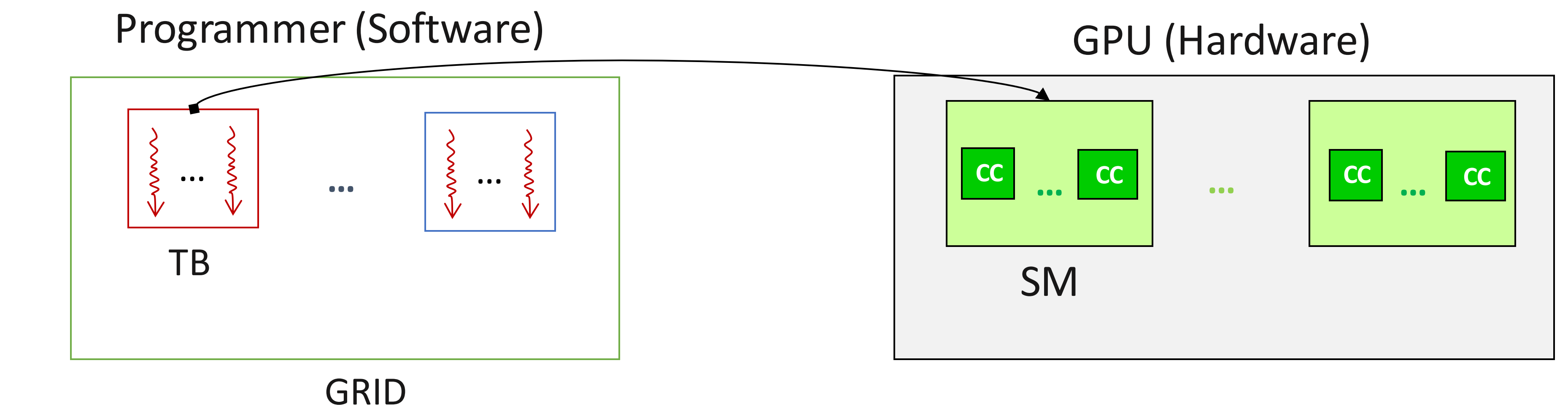
    32 threads in Warp and 16 CCs assigned to it
- Within Warp it is SIMD but no guarantee outside each Warp
  - Warp #0 of a thread block may be executing instruction i while Warp #1 of the same thread block may be executing instruction j, $i \neq j$

SIMT Execution Model (5)

- A Warp (32 threads) is executed by a set of CC (ex. 32 CC in 1 clock cycle or 16 CC in 2 clock cycles)



Parallelism (1)



Independent thread blocks
Parallel threads in thread block (TB)

SM: 1) Many CC
2) Many Warps in parallel in each SM
3) In each CC (thread)
ILP—Many functional units/thread

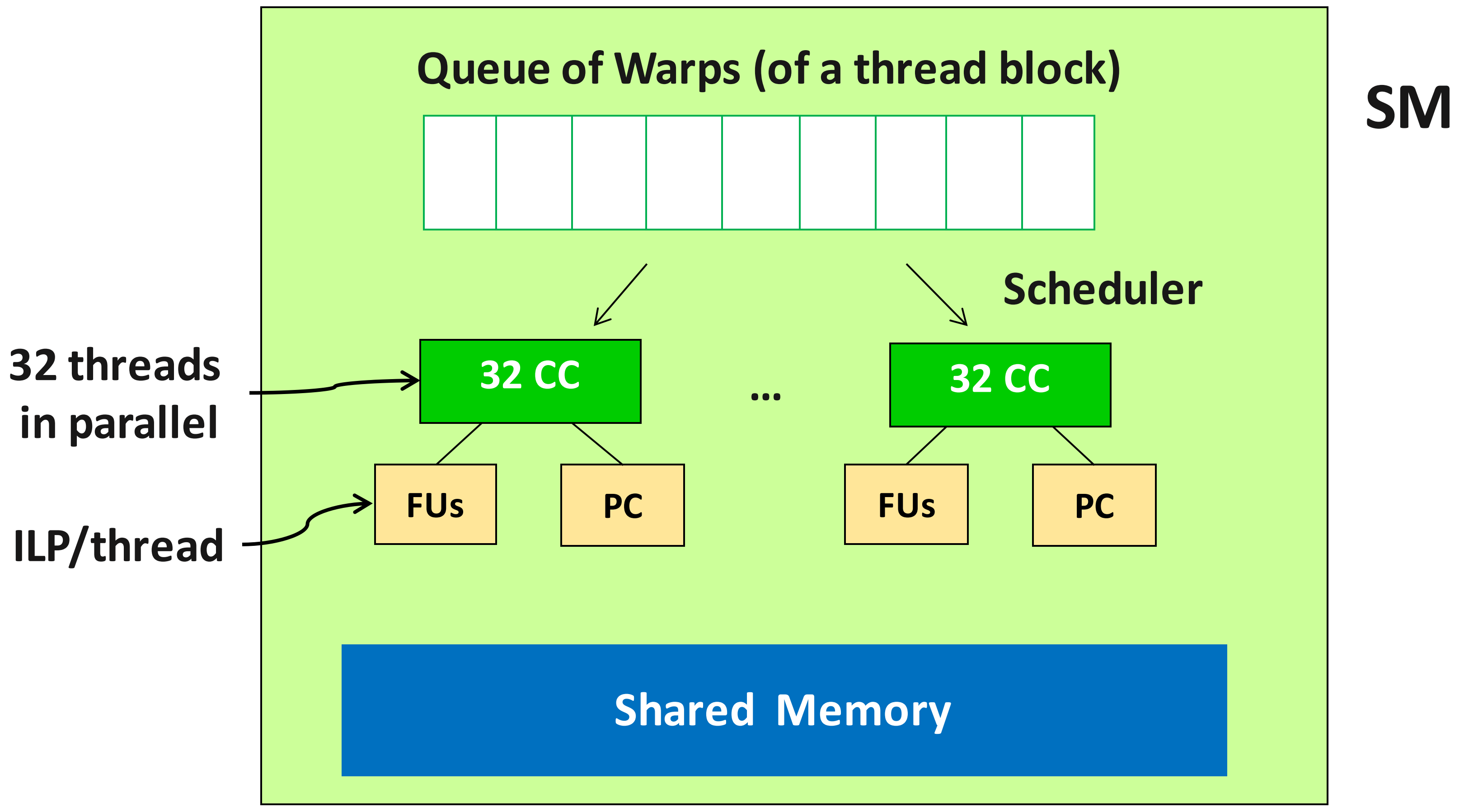
Parallelism (2)

How many program counters (PC) in each SM?

For every Warp, we need a PC

# of Warps can be $> \frac{\#\text{ of CC}}{32}$

Parallelism (3)



Parallelism (4)

Restriction

In each thread block – only one program!

Not **all** applications have such parallelism.

Synchronization (1)

- At run time, Warps in the same thread block may execute different instructions in the same clock cycle

Time ↓	Warp 0	Warp 1	Warp 2	Warp 3
Cycle 0	Execute Ins. 0	Execute Ins. 0	×	×
Cycle 1	Execute Ins. 1	×	×	×
Cycle 2	Execute Ins. 2	×	×	Execute Ins. 0
	Execute Ins. 3	×	×	Execute Ins. 1

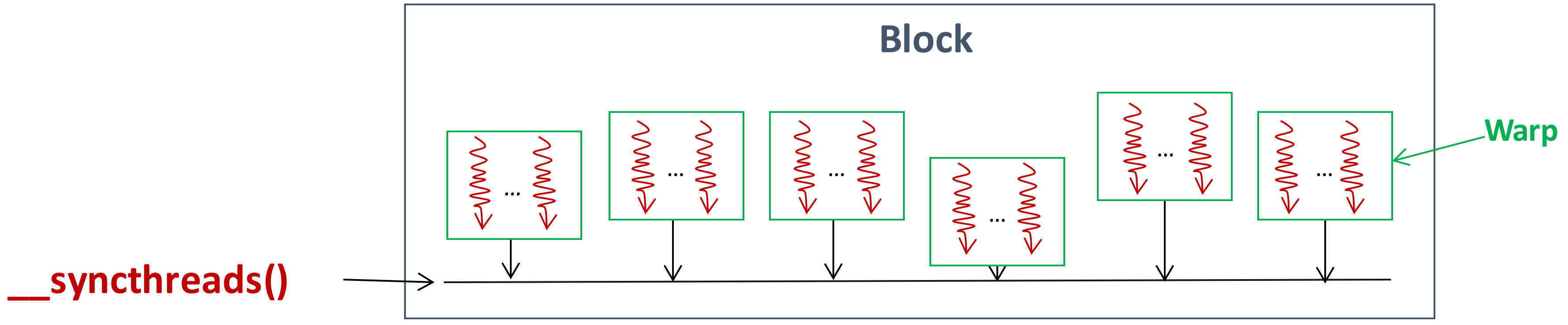
×: waiting to be scheduled, blocked due to I/O, global memory access, …

Synchronization (2)

- **Need for Synchronization:** When some threads within a block access the same addresses in shared or global memory, there are potential read-after-write, write-after-read, or write-after-write hazards

- **__syncthreads()** is a block level synchronization barrier;

  wait until all threads in the thread block have reached this point

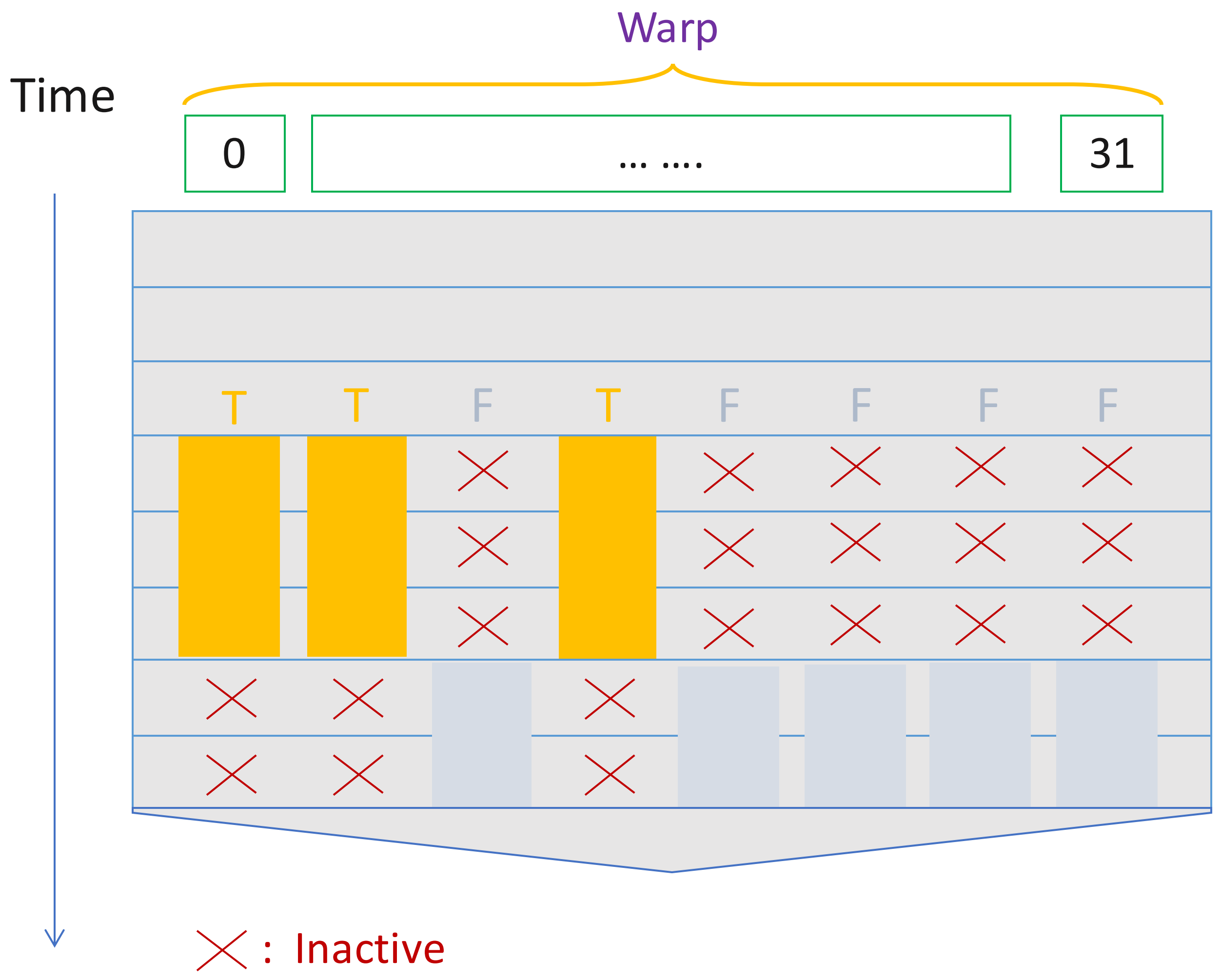


Branching (1)

- Branching (e.g., if-else, for, while, do, switch, etc.) causes a **divergence** in the flow of execution
- GPU (SM) simply executes one path of the branch and then the other (i.e., serial execution)
- Those threads that take the branch are executed and those that do not are marked as inactive
- Once the taken branch is resolved, the other side of the branch is executed, until the threads converge once more

```
if (some_condition) then {
    Action_a();
}
else {
    Action_b();
}
```

Branching (2)



```
<unconditional code>

if (x > 0) {
    Action_a()
} else {
    Action_b()
}

<unconditional code>
```

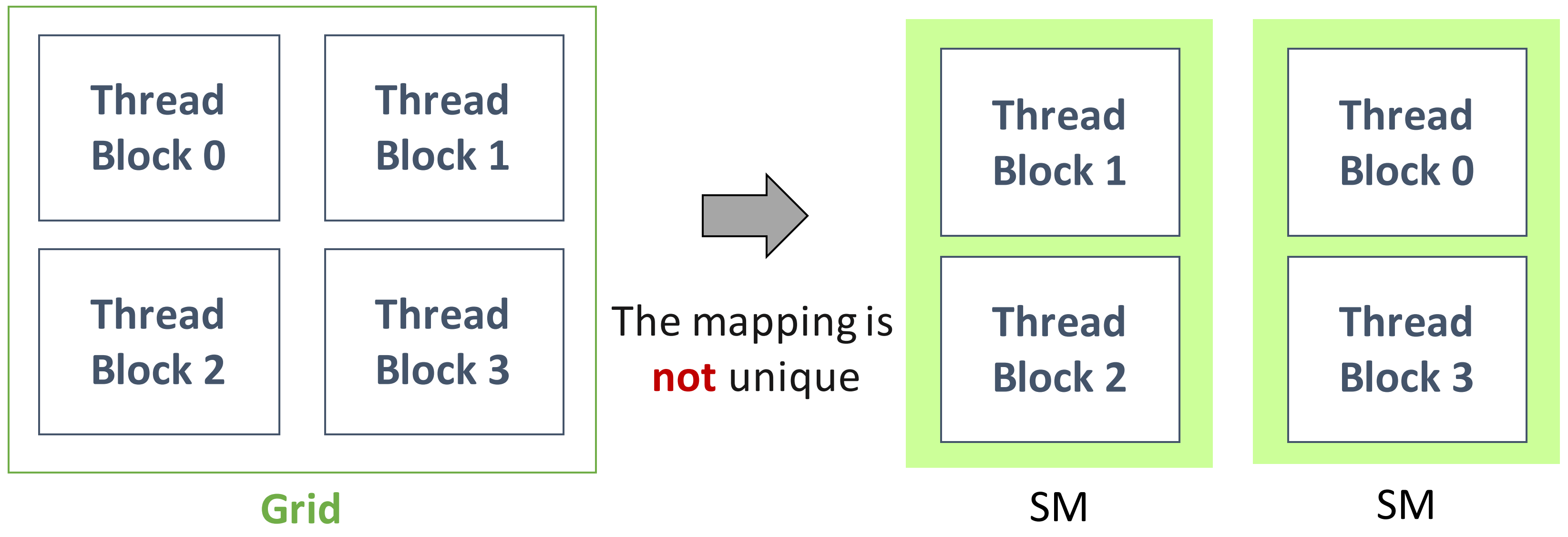
Impact of Divergence (branching)

- Serial execution of the two codes (then, else portions)

- Severe performance degradation

Global Execution Model

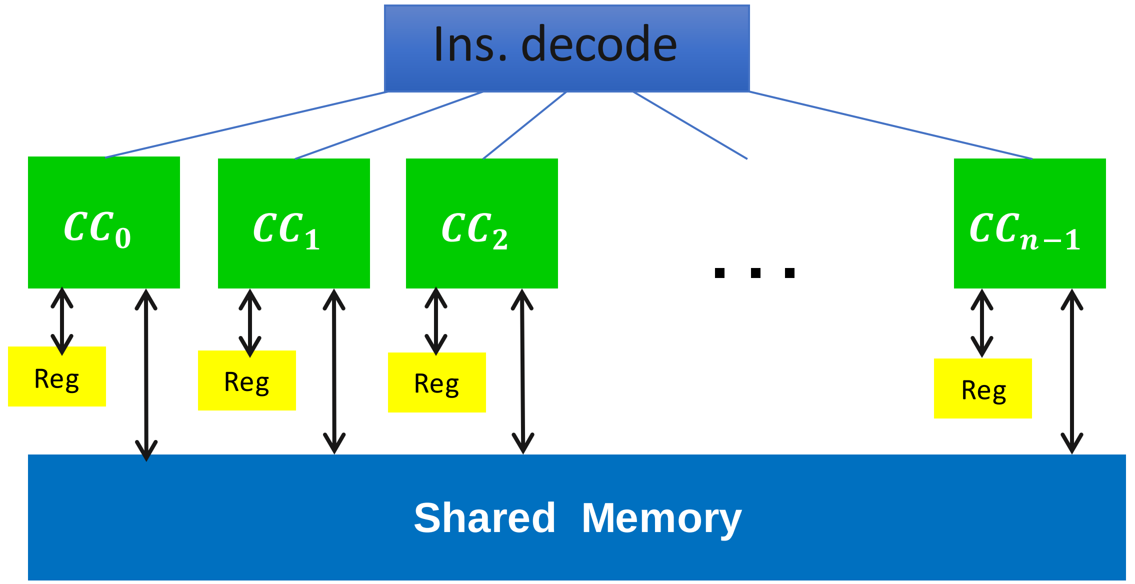
- Kernel = Collection of thread blocks
- Concurrent blocks processed using SM



Note: All threads in a block execute the same code

Local Execution Model

- SIMT (Single Ins. Multiple Threads) (in SM)
- Parallel execution across CUDA Cores (CC)



n=16,32

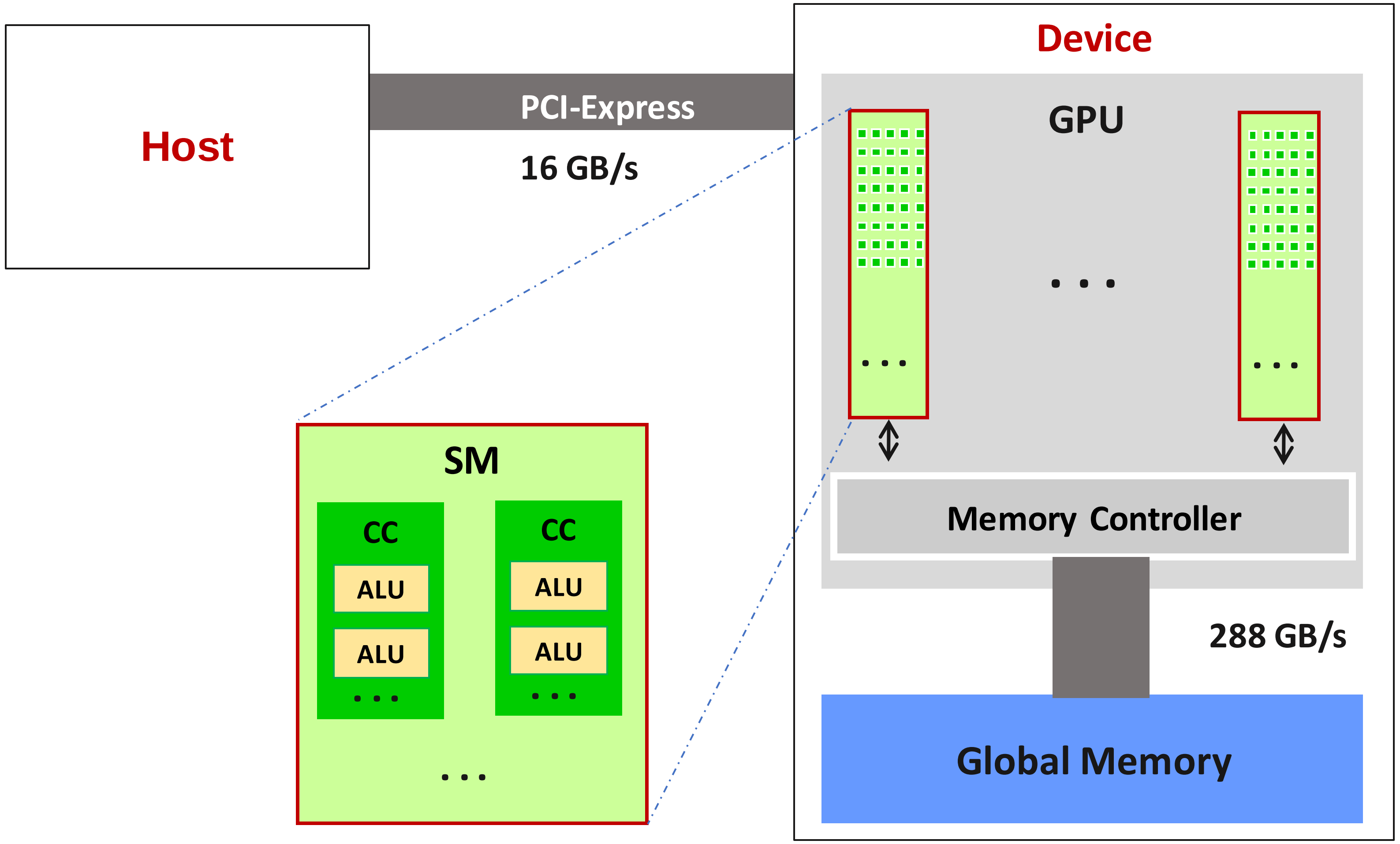
Outline

- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
  - Programming Model
  - GPU Architecture
  - Execution Model
  - Performance Model
  - Examples
  - CUDA, OpenCL, …
  - APU, Zynq…

A Simple Performance Model (1)

- ***N*** No. of threads in a grid
- ***D*** GB of input data used for a kernel
- ***M*** GB of data accessed from global memory to the device for each thread
- ***C*** Tflops executed in each thread
- Example Device Configuration
  - 16 SM in a Device, 128 CC/SM, 8 ALU/CC, 2 flops/ALU/cycle, 1 GHz
  - Peak performance
    - 16*128*8*2*1 G = 32 TFLOPS
  - Bandwidth
    - PCI-e 3.0 (x16) : 16 GB/s
    - GDDR5 memory controller : 288 GB/s

A Simple Performance Model (2)



A Simple Performance Model (3)

- Assume:
  - no diverging conditional branch
  - No memory bank conflicts

- Total Execution time for a kernel

T(DataTransferCPU←→Device) + (# Threads/ # CC) x T(Thread)

$= 2 \times (D/16) + N/(16\times 128) \times$ [T(MemOpsglobal) + T(Computation Ops)]

$= 2 \times (D/16) + N/(16\times 128) \times [M/288 + C/32$ TFLOPS]

Outline

- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
  - Programming Model
  - GPU Architecture
  - Execution Model
  - Examples
  - CUDA, OpenCL, …
  - APU, Zynq…

Matrix Multiplication (1)

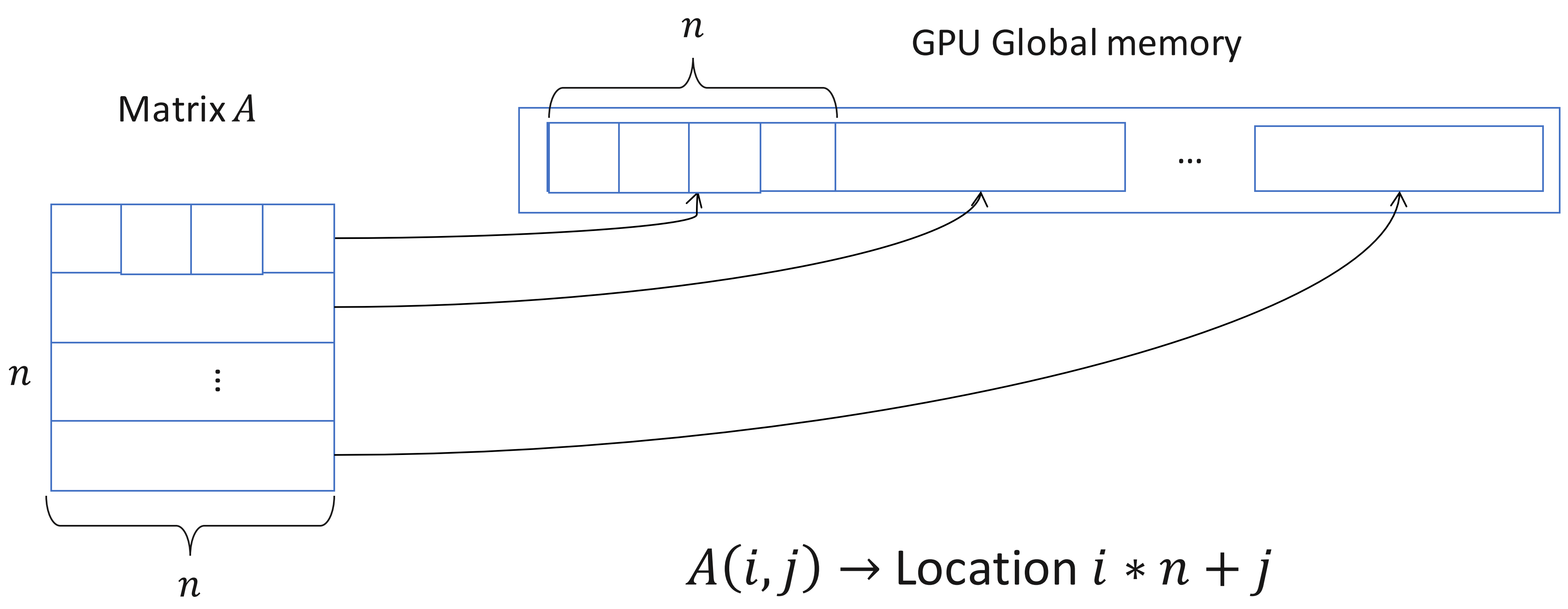
Example: $C = A \times B$

Matrix size $= 1024 \times 1024$

- Allocate device memory for A, B, C
- Load A and B to GPU memory
- Invoke kernel function
- Copy result from device to host

Matrix Multiplication (2)

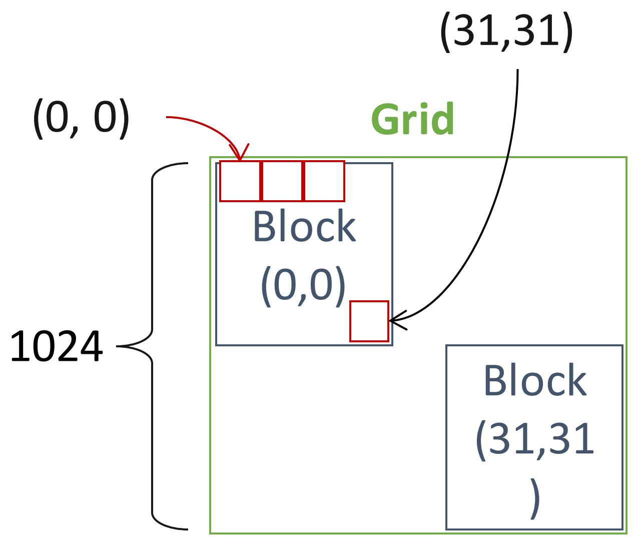
- Example Data layout: All matrices are stored in **row major** order



$A(i,j) \rightarrow$ Location $i * n + j$

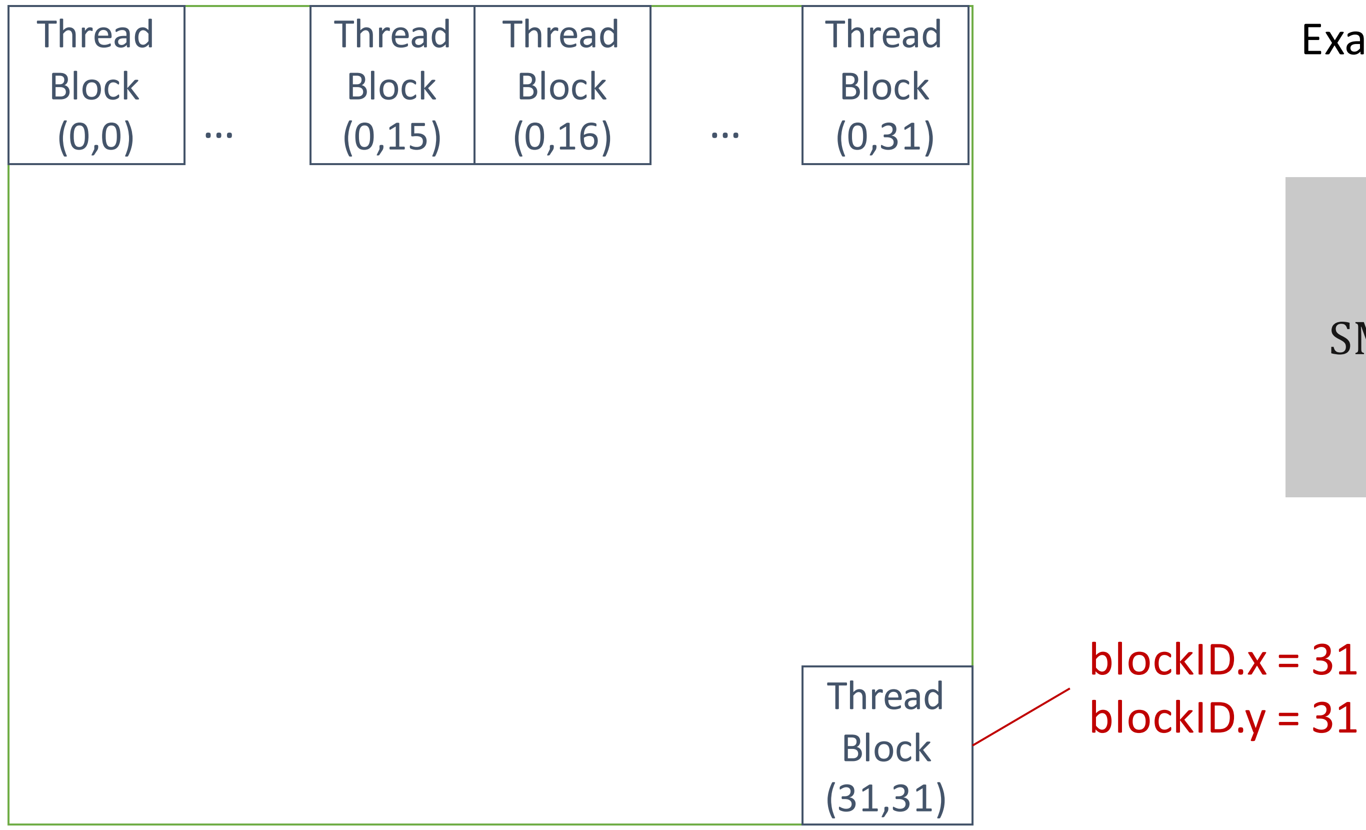
Matrix Multiplication (3)

- Total # of threads = 1K $\times$ 1K
  - 1 thread for each element of C
- Example grid, thread block, thread definition
  - 2-D grid: 32 $\times$ 32 – Thread blocks
  - 2-D block: 32 $\times$ 32 – Threads/Block
  - Each thread computes one element of C

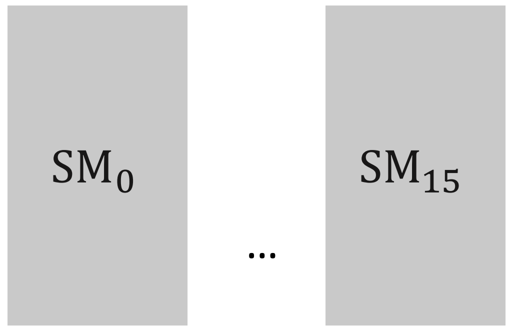


Matrix Multiplication (4)

Grid and Blocks and mapping

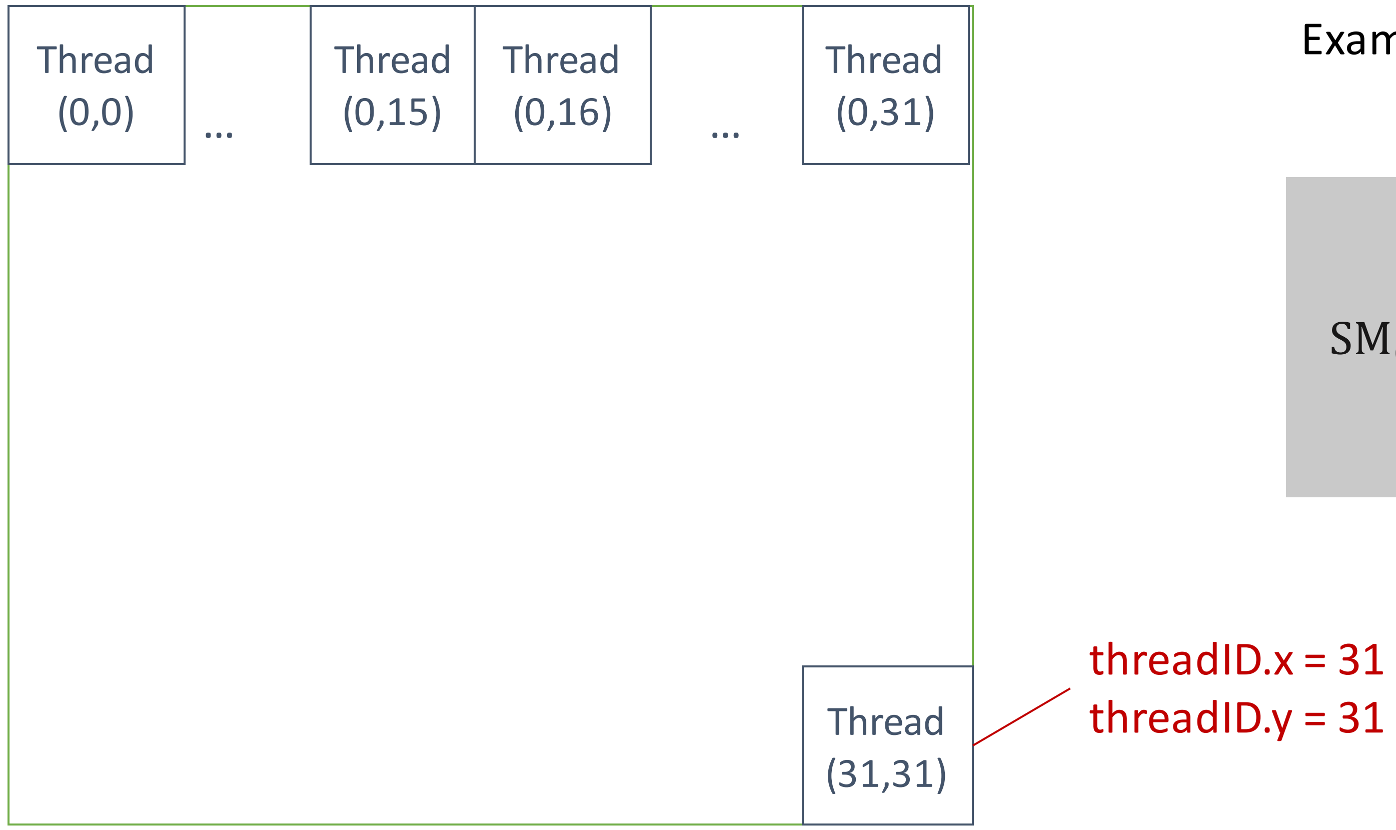


Example GPU organization

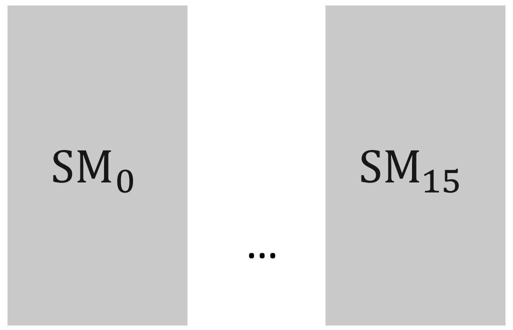


Matrix Multiplication (4)

Thread Block



Example GPU organization



Matrix Multiplication (5)

Algorithm 1 (straight forward inner product, no optimizations)

Grid configuration:
1-D or 2-D

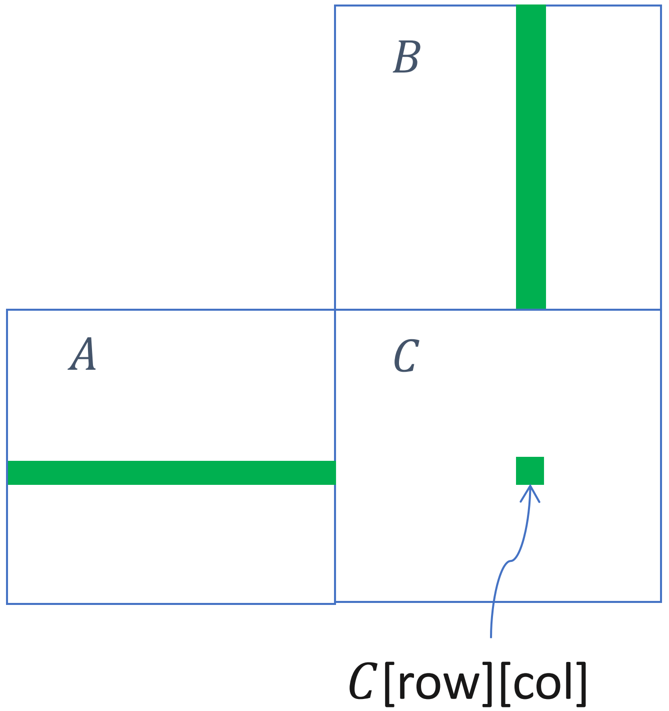
Block configuration:
1-D or 2-D or 3-D

- Kernel function <<dim_grid,dim_block>>(A, B, C){

```
    Local_C = 0;
    row  = row index of the thread in the matrix
    col  = column index of the thread in the matrix
    for ($k$=0; $k$<1024; $k$++)
        Local_C += $A$[row][$k$] * $B$[$k$][col];
    End for
    $C$[row][col] = Local_C
}
```

Local register

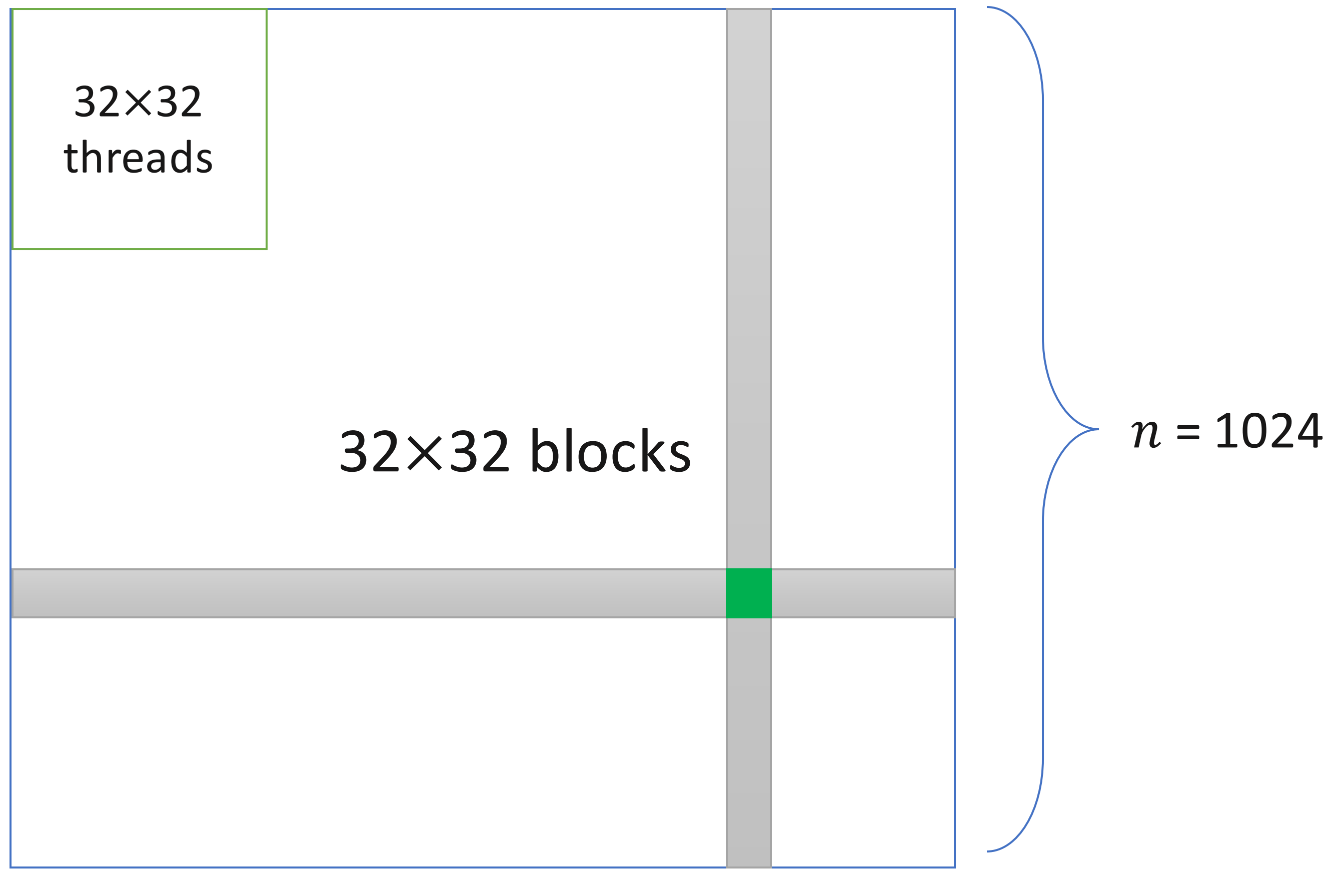
Global memory (data in row major order)



Matrix Multiplication (6)

row = 32×blockID.x + threadID.x
col = 32×blockID.y + threadID.y

Layout:



Matrix Multiplication (7)

1K $\times$ 1K MM

- Each thread loads 2×1024 elements from global memory
  - — 1024 elements from A
  - — 1024 elements from B
- Note: threads may idle for hundreds of clock cycles for each access
- Total data communicated from global memory to device = $n^2[2n]$
- Algorithm does not exploit data sharing (data reuse) among the threads

Block Matrix Multiplication

- **Optimization**: load portions of A and B into shared memory
  - — 64 KB shared memory can not hold two 1K×1K matrices
  - — Decompose matrices A and B into non-overlapping sub-matrices, eg. 32×32 sub-matrix

(See Discussion Session)

Outline

- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
  - Programming Model
  - GPU Architecture
  - Execution Model
  - Examples
  - CUDA, OpenCL, …
  - APU, Zynq…

CUDA, OpenCL, …

- OpenCL
  - Cross-vendor standard for heterogeneous computing
  - Maintained by Khronos Group
  - Provides C/C++ extension and API
- CUDA
  - NVIDIA’s parallel computing architecture using GPU
  - Provides CUDA C (C extension) and API
  - Supports OpenCL
- OpenACC
  - Designed to simplify parallel programming of heterogeneous CPU/GPU system

CUDA vs. OpenCL Terms

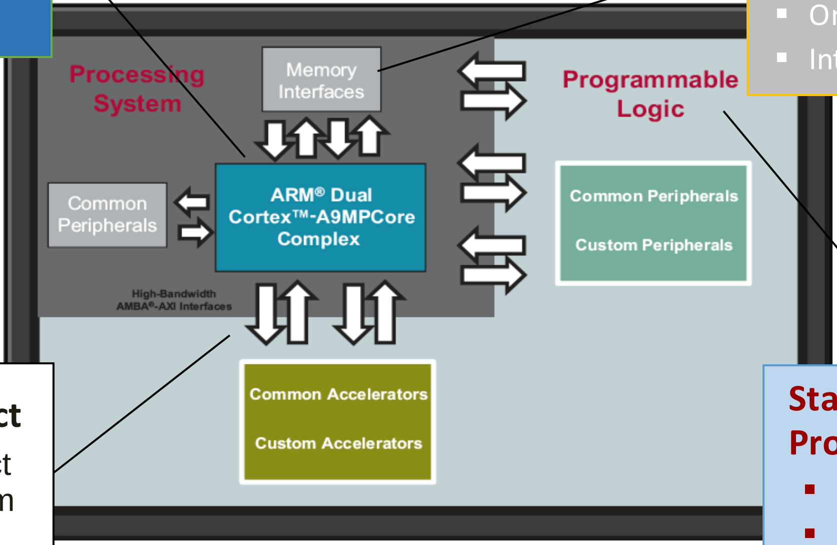
CUDA	OpenCL
Global Memory	Global Memory
Per-Block Shared Memory	Local Memory
Local Memory	Private Memory
Kernel	Program
Block	Work-group
Thread	Work-item
NVIDIA	NVIDIA, AMD, Intel, ARM …

Zynq

- Zynq-7000 Programmable SoC family
  - Efficient ARM + FPGA

Processor Core Complex
- Dual ARM Cortex-A9 MPCore
- Single / Double Precision Floating Point support
- Up to 1 GHz operation

High BW Memory
- Internal
  - L1 Cache – 32KB/32KB (per Core)
  - L2 Cache – 512KB Unified
- On-Chip Memory of 256KB
- Integrated Memory Controllers



AMBA Standard Interconnect
- High bandwidth interconnect between Processing System and Programmable Logic

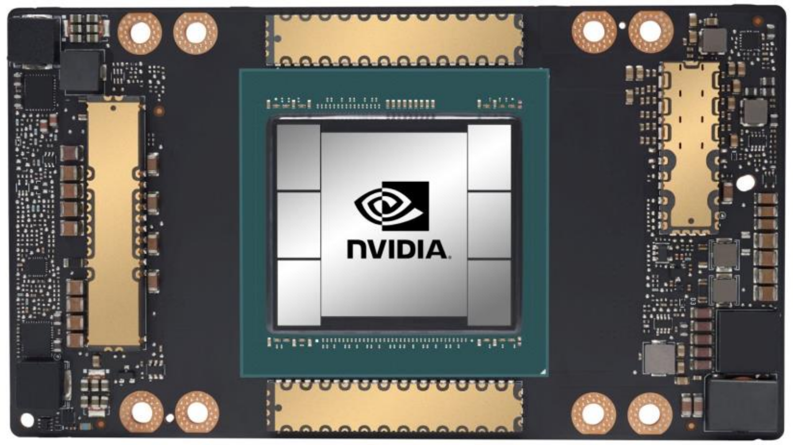
State-of-the-art 7 Series Programmable Logic
- Artix-7 & Kintex-7 FPGA Fabric
- 28K-350K logic cells

State-of-the-art GPU Example

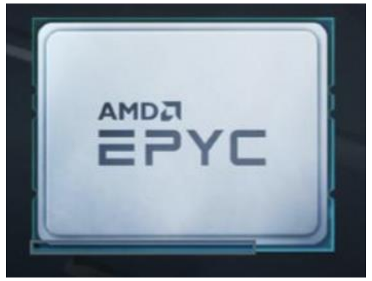
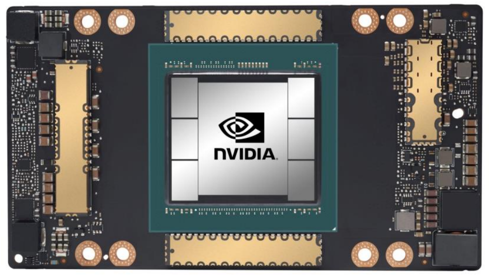
- GPU family – Nvidia

— A100 Tensor Core GPU

  - 108 Streaming Multiprocessors
  - 40 MB L2 cache
  - 40 GB high bandwidth memory
  - 1.6 TB/s peak bandwidth
  - 600 GB/s inter-GPU communication bandwidth
  - 54.2 billion transistors



Comparison

AMD	Nvidia	Xilinx MPSoC
		
16-64 heavy cores	~7 K light cores	1-8 M fine grained cores
Up to 40 B transistors	54 B transistors	35 B transistors
3.4 GHz	1200 MHz	Up to 600 MHz
225 W	400 W	30 W

Summary

- GPU and data parallel programming
- CUDA model
- SIMT
- Kernel → grid, thread blocks, threads
- Explicit parallelism (data parallelism/Block, task parallelism across Blocks)