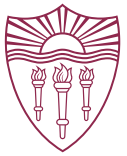


EE/CSCI 451: Parallel and Distributed Computation

Lecture #14

10/6/2020

Viktor Prasanna

prasanna@usc.edu

ceng.usc.edu/~prasanna

University of Southern California

USC University of Southern California

Announcement

- Midterm 1 grades are out on Blackboard
  - Maximum 66, Minimum 9
  - Average 45.6
  - Median 48.5
  - Cut-offs:
    - ≥47 A
    - ≥36 B
    - Rest of the students(4): B-/C+ (?)
    - If you like to talk, contact me (prasanna@usc.edu) to schedule a meeting
  - Note: The above is to give you feedback re. your performance. Letter grades are NOT assigned to midterm (or homeworks)

- Regrading requests:
  - Accepted until 11:59PM Friday (LA time)
  - Format: direct email to TA (ymeng643@usc.edu), copy instructor (prasanna@usc.edu) and grader (ziyaoyan@usc.edu).

Title: EE 451 Midterm 1 Regrading Request

Content: Problem number + Your reasoning

Note: Submit a request ONLY IF you are convinced that points are WRONGLY deducted.

2

Grading

- Weighted sum for a student $= \sum x_i * w_i$,

$$where\ x_i = \frac{Your\ total\ score\ in\ category\ i}{Max.\ total\ score\ of\ category\ i},$$

$$w_i = Weight\ of\ category\ i\ in\ percentage$$

Assignment	% of Grade
Homework	10
Programming HW	10
Course Project	15
Midterm 1 exam	20
Midterm 2 exam	20
Final exam	25
Total	**100**

- Final grades are given based on the total weighted sum
  - If (Weighted sum for a student >= final cutoff for A): student gets A
  - …

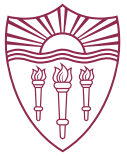
Announcement

- PHW4 due on 10/9 (Friday)
- HW6 due on 10/9 (Friday)
- PHW5 and HW7 will be out on 10/9
  - PHW5 due on 10/22
  - HW7 due on 10/16

Course Project

- Search for potential collaborator(s) online:
  - https://piazza.com/class/kdqbo664ofr30n?cid=5
- When you have identified your project and/or collaborating team members, pls. fill out this form. Every team should have one entry in the form.
  - https://docs.google.com/spreadsheets/d/1voizlvpu22w26D2Y3I4TOCjc5Kwe1SdAsmE605ADOs0/edit?usp=sharing
- Project timeline
  - Week 7-8: Identify team members and project topic, discuss with instructor or TA
  - End of Week 9: Project proposal due (Oct. 16)
  - Week 12-13: Presentation
  - End of Week 13: Project report due (Nov. 13)
- Grading breakdown for the course project
  - Proposal: 25%; Final presentation: 25%; Final report: 50%

Outline



Architecture, Programming and Execution Model of (GP) GPU

- GPU and data parallel programming
- CUDA Programming Model
- GPU Architecture

Lecture 1

- Execution Model
- Performance Model
- Examples
- CUDA, OpenCL, …
- APU, Zynq,…

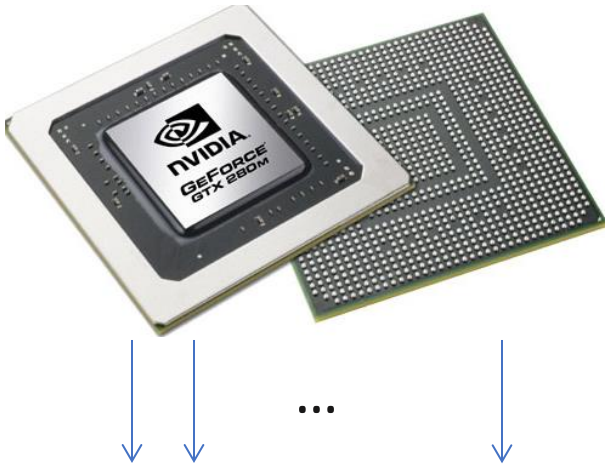
Lecture 2

Latency vs. Throughput Computing

- One thread (or few threads)
- Minimize latency

(cache, pipeline, superscalar, multiple functional units, branch prediction, ILP, … EE 457)

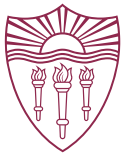
- Implicit parallelism



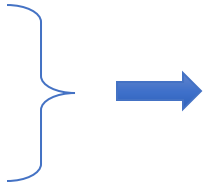
…

- Hundreds, thousands of threads
- Very simple processors
- Restrictive (parallel) programming model

GPU Design Considerations



- Very simple compute units
- Simple control



Large number of compute units per unit area

- Massive # of compute units
- Reduce impact of latency stalls by interleaving workgroups (threads)
- Explicit parallel programming (restrictive model)
- Evolving (games hardware, sometimes awkward from parallel programming perspective)
- General purpose computing?? Accelerator??

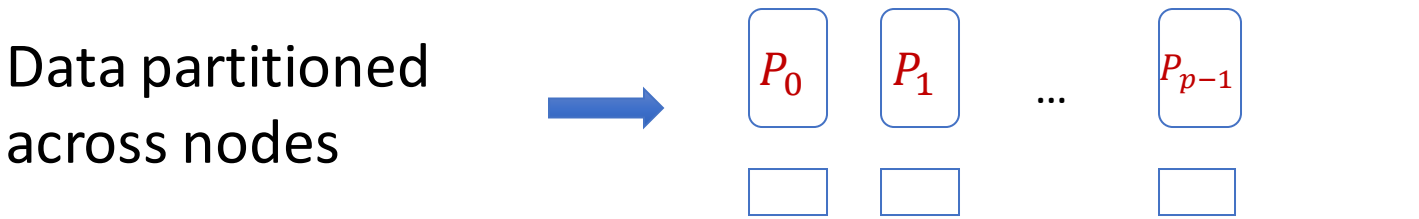
Thread

- An independent sequence of instructions in a parallel program
- Eg. Matrix multiplication

```
Each element of C = 0
for i = 1:n
    for j = 1:n
        for k = 1:n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
        end
    end
end
```

Thread

Data Parallel Programming

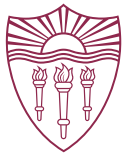


Program executes on data in parallel

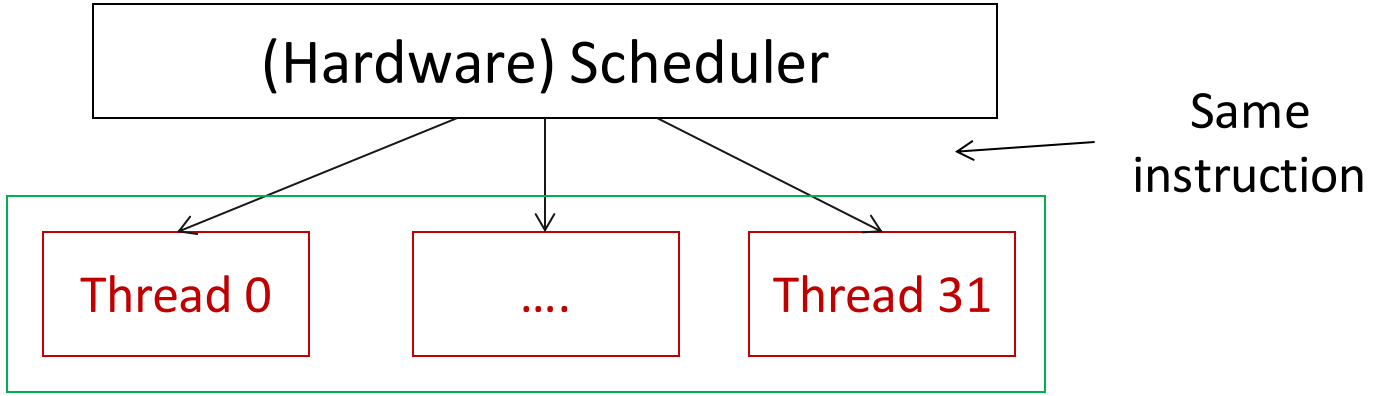
CUDA is a (restricted) Data Parallel programming model

CUDA = Compute Unified Device Architecture

Single Instruction Multiple Thread (SIMT)



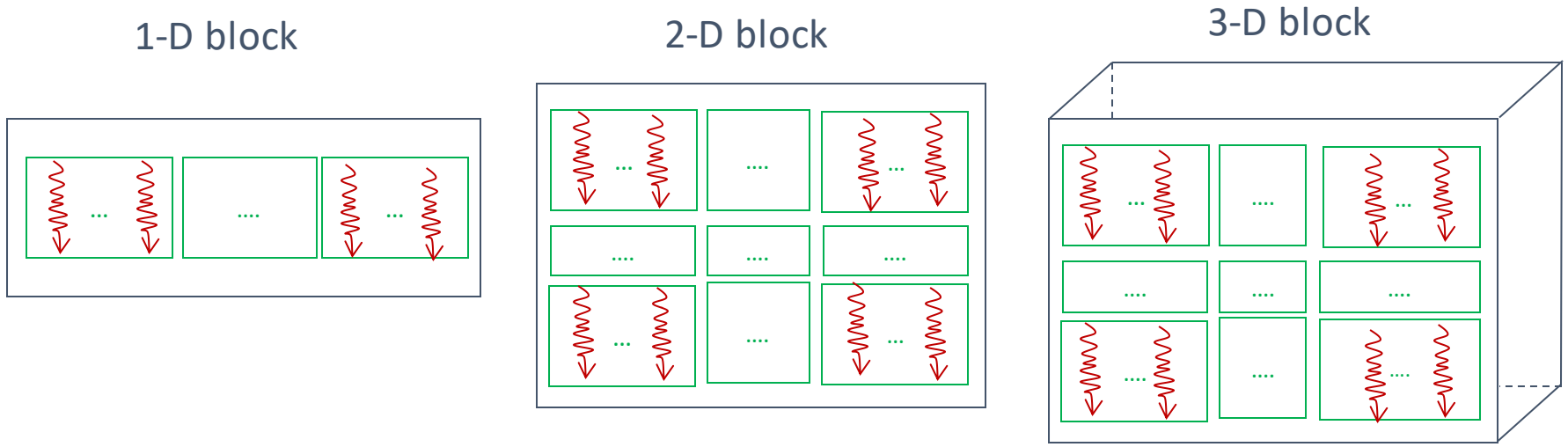
Data parallel programming model



- All threads execute the same set of instructions (on different data).
- SIMT is a virtualized execution of SIMD
- # of threads can be > # of hardware units

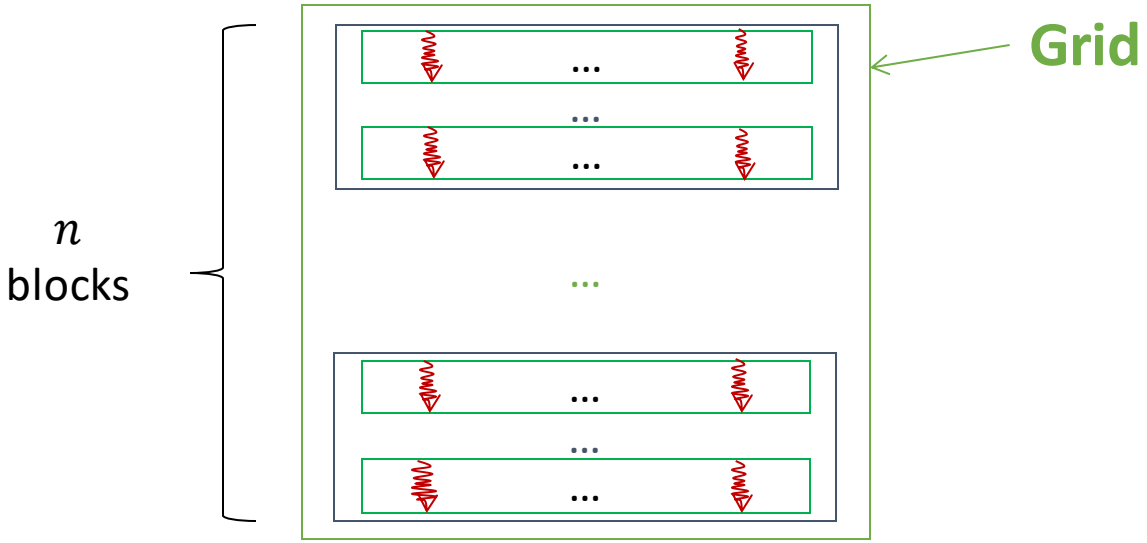
(Thread) Block

- Threads are grouped into blocks (<= 1K threads per block)
- Block can be one-, two-, or three-dimensional
- Thread Block is mapped to an SM (see later)
- Shared Memory Data Parallel Programming Model at Block level



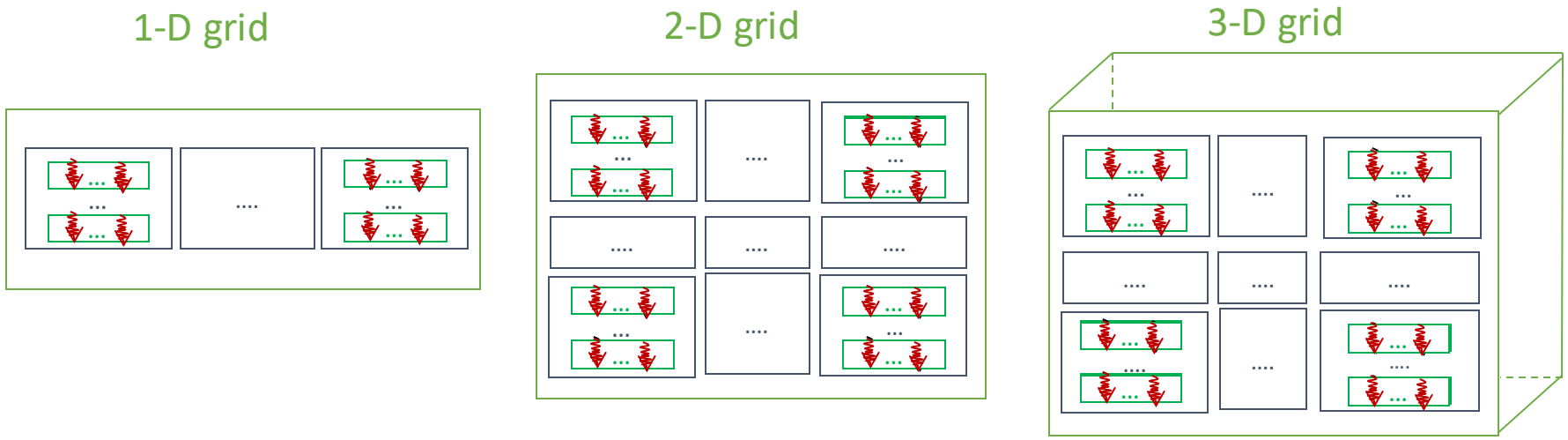
Grid (1)

- Blocks are grouped to form a **grid** of thread blocks

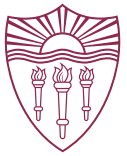


Grid (2)

- Grid can be one-, two-, or three-dimensional

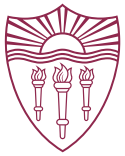


Thread Hierarchy



- Threads launched for a parallel section of the code (Kernel) are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
    - block level synchronization barrier
    - wait until all threads in the thread block have reached the point
  - Communicate with other threads in the same block
  - Thread block = Single Ins. Stream + (multiple data)

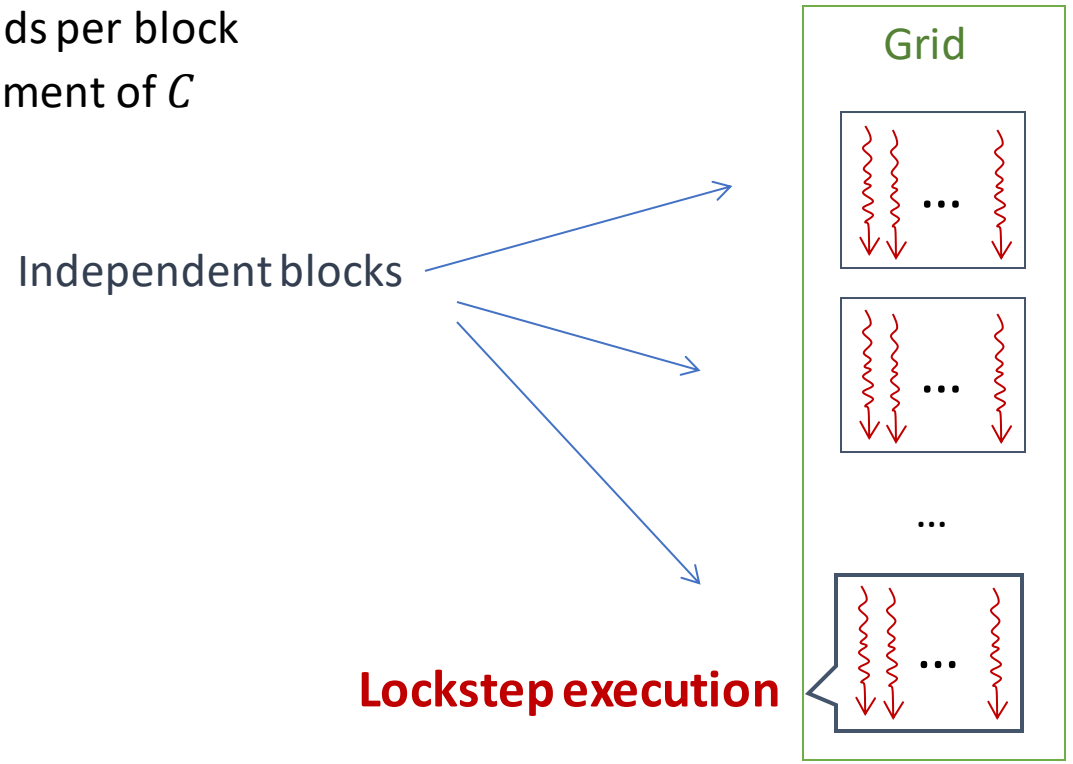
Grid and Thread Blocks



- Grid = Collection of thread blocks
- Blocks must be independent
- Thread blocks can run in any order
  - Concurrently or sequentially
  - Facilitates scaling of the same code across many devices
- Thread blocks can share data within the Thread block (only)
- Implicit barrier at the end of all blocks of a kernel
- All blocks in a kernel have the same number of threads

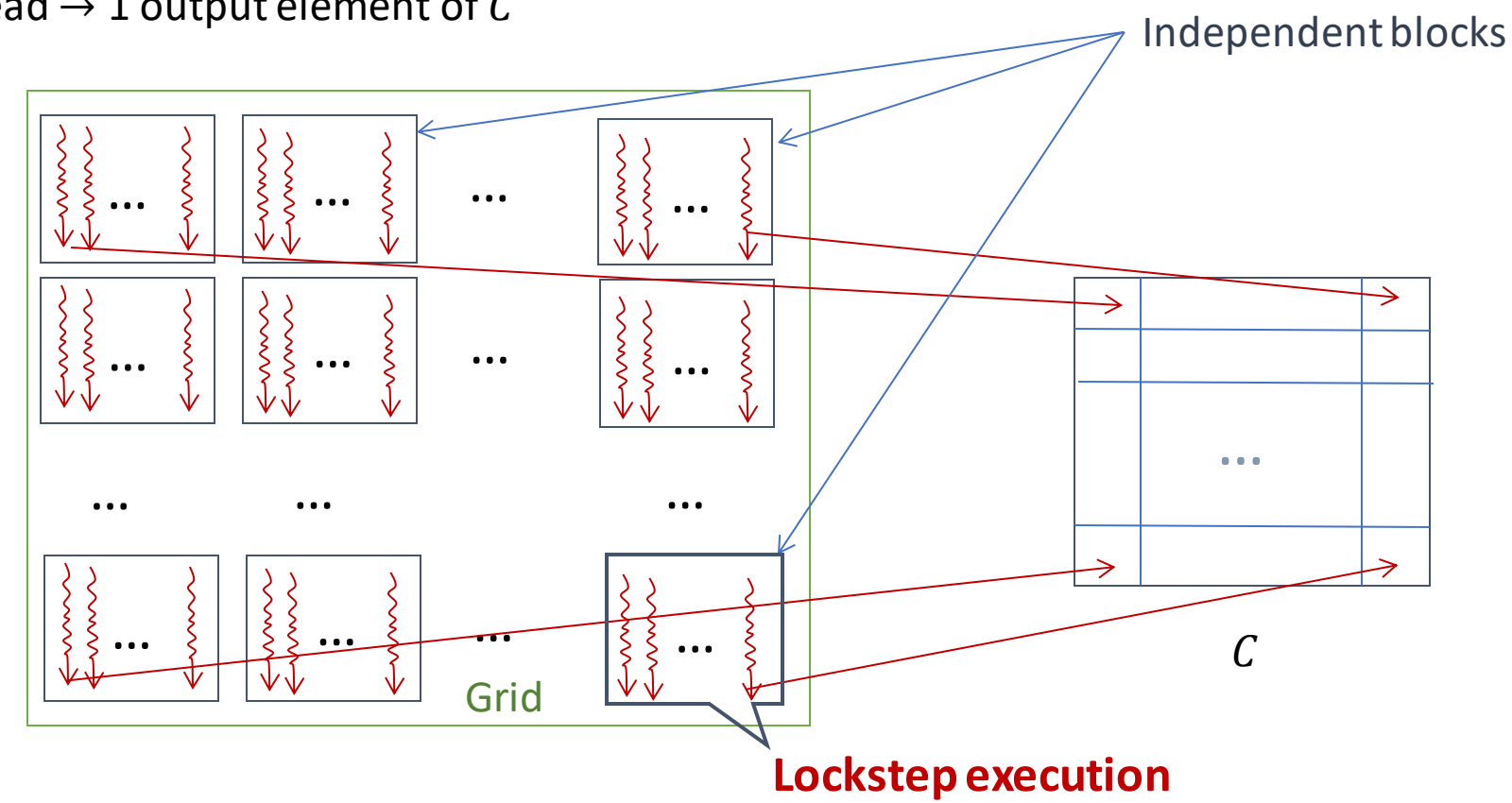
Example: Matrix Multiplication (1)

- 1024×1024 matrix multiplication: $C = A \times B$
- 1D Grid Configuration
  - 1024 blocks, 1024 threads per block
  - 1 thread → 1 output element of C

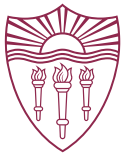


Example: Matrix Multiplication (2)

- 2D Grid Configuration
  - 32 × 32 blocks per grid, 32 × 32 threads per block
  - 1 thread → 1 output element of C



Kernel (1)



- A kernel is the unit of work that the main program running on the host computer offloads to the GPU for computation
- Specify three parameters when launching
  - The dimensions of grid
  - The dimension for each block
  - The kernel function (for each block) to run on GPU
- Once launched, a grid of blocks are created and queued to be run on the GPU's Streaming Multiprocessors (SM)
- Correctness of the (parallel) program should be independent of the scheduling of blocks (at run time, by hardware) – users' responsibility

Kernel (2)

- Eg. Matrix multiplication

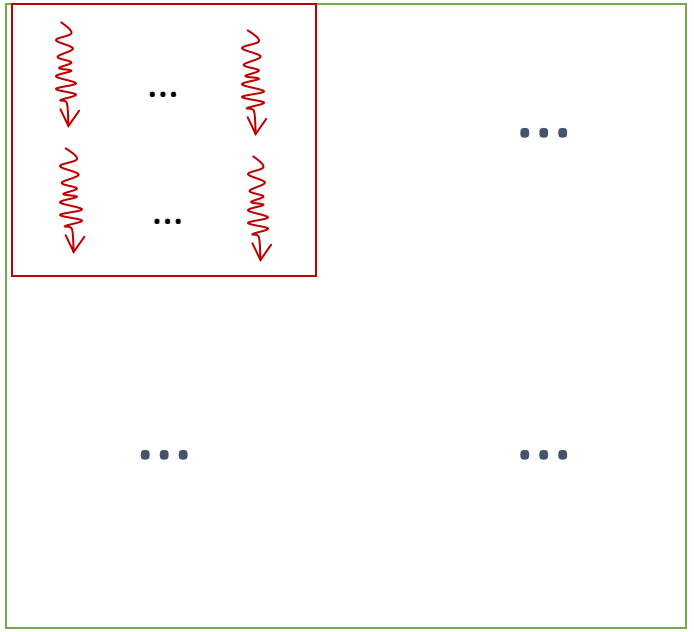
Each element of $C = 0$
for $i = 1{:}n$
  for $j = 1{:}n$
    for $k = 1{:}n$
      $C[i,j] = C[i,j] + A[i,k] * B[k,j]$
    end
  end
end

Kernel (3)

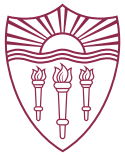
- Kernel = grid of thread blocks
- Thread blocks are independent
- Implicit barrier at the end of grid

Note: The instruction stream in different thread blocks in a grid can be different

Ex: Matrix Multiplication



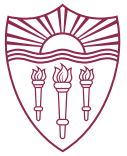
Outline



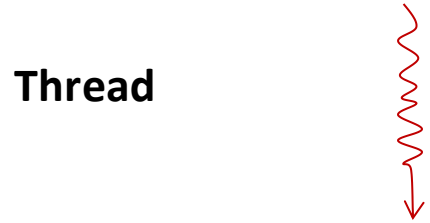
Architecture, Programming and Execution Model of (GP) GPU

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

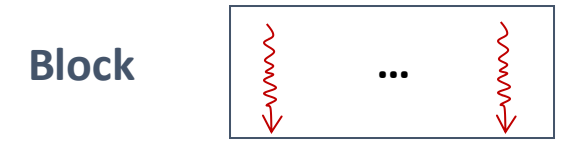
CUDA Multithread Programming Model (1)



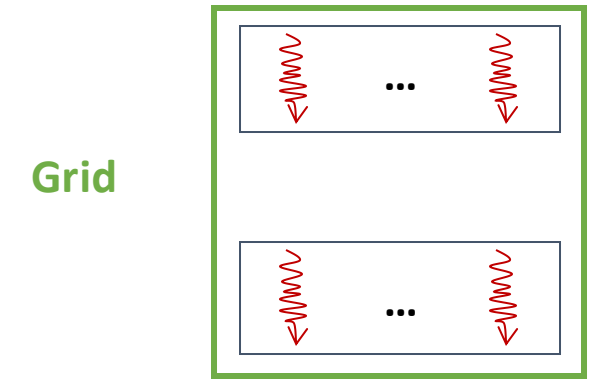
- CUDA program = Host + Kernel



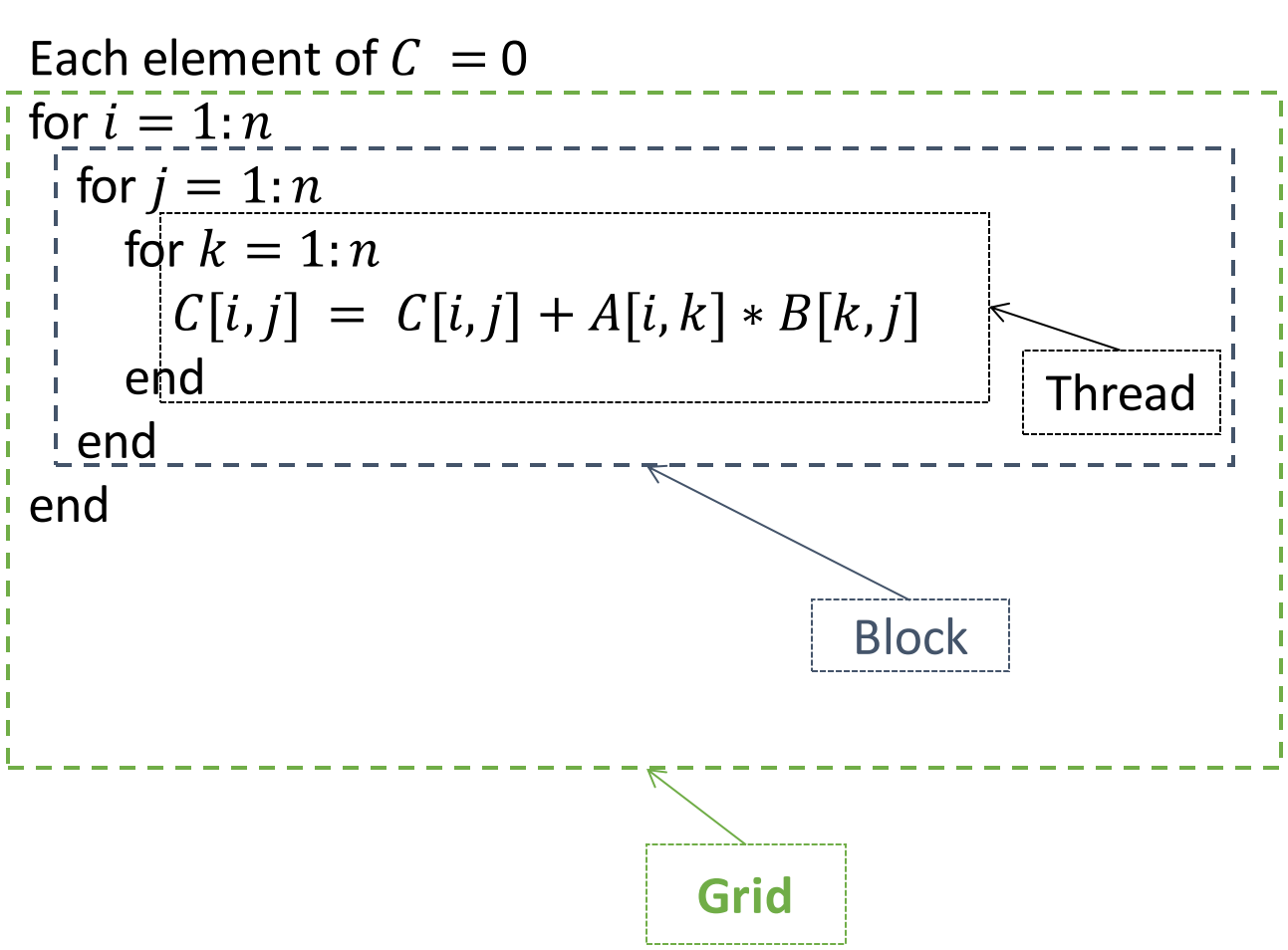
- Host code runs on CPU and kernel code runs on GPU

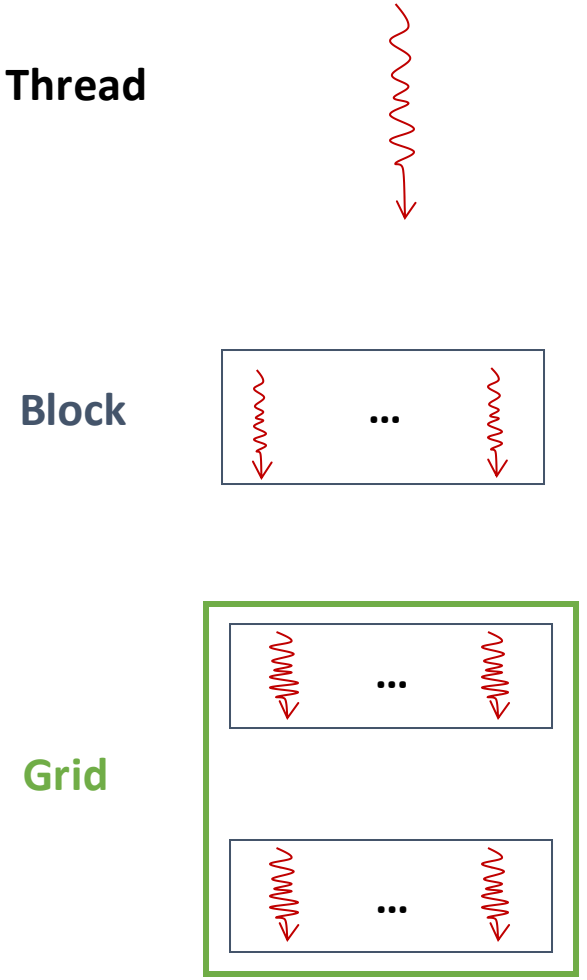


- Kernel code specifies
  - the number of threads per block
  - the number of blocks over the entire grid
  - the instruction stream for each block and the data to be used

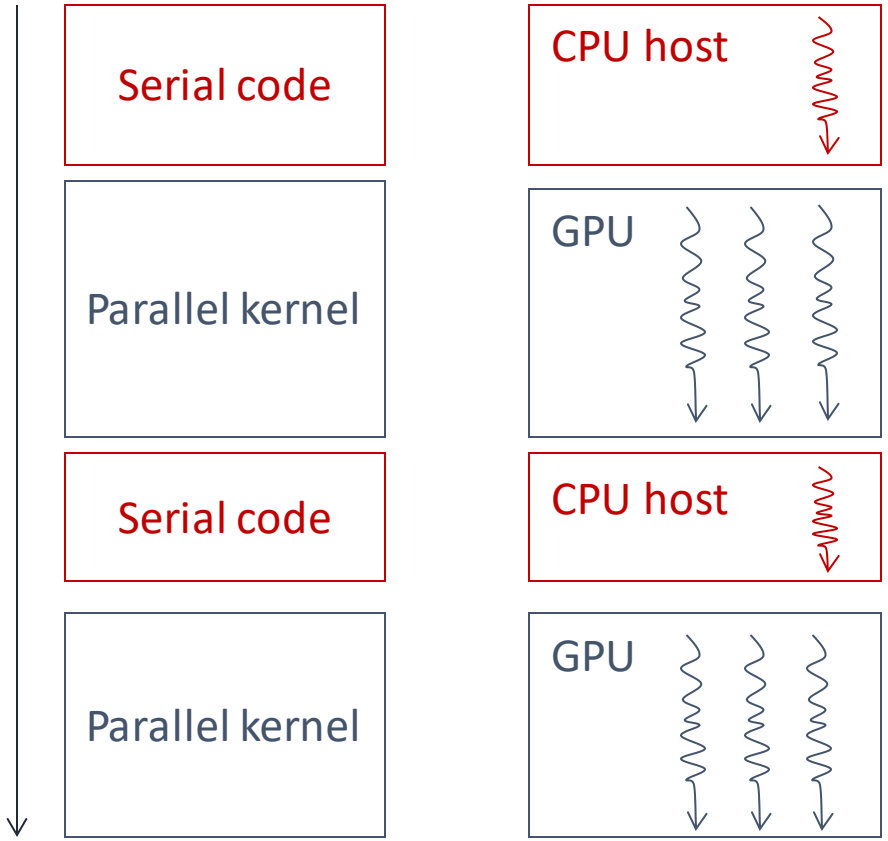


CUDA Multithread Programming Model (2)





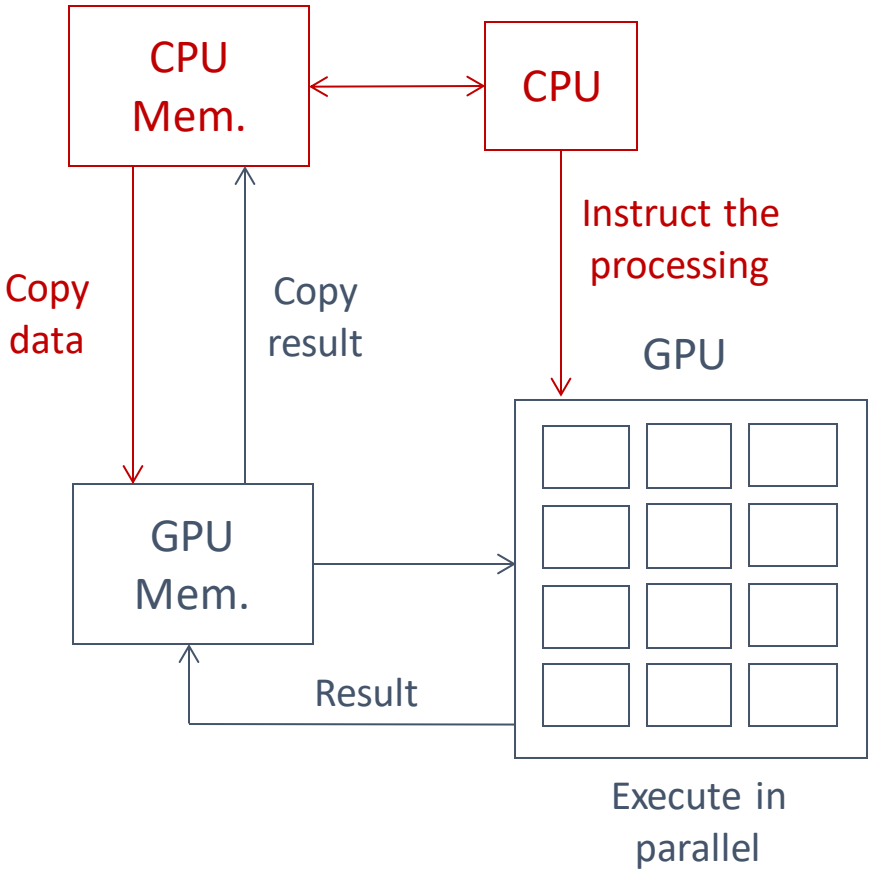
Program Execution (1)



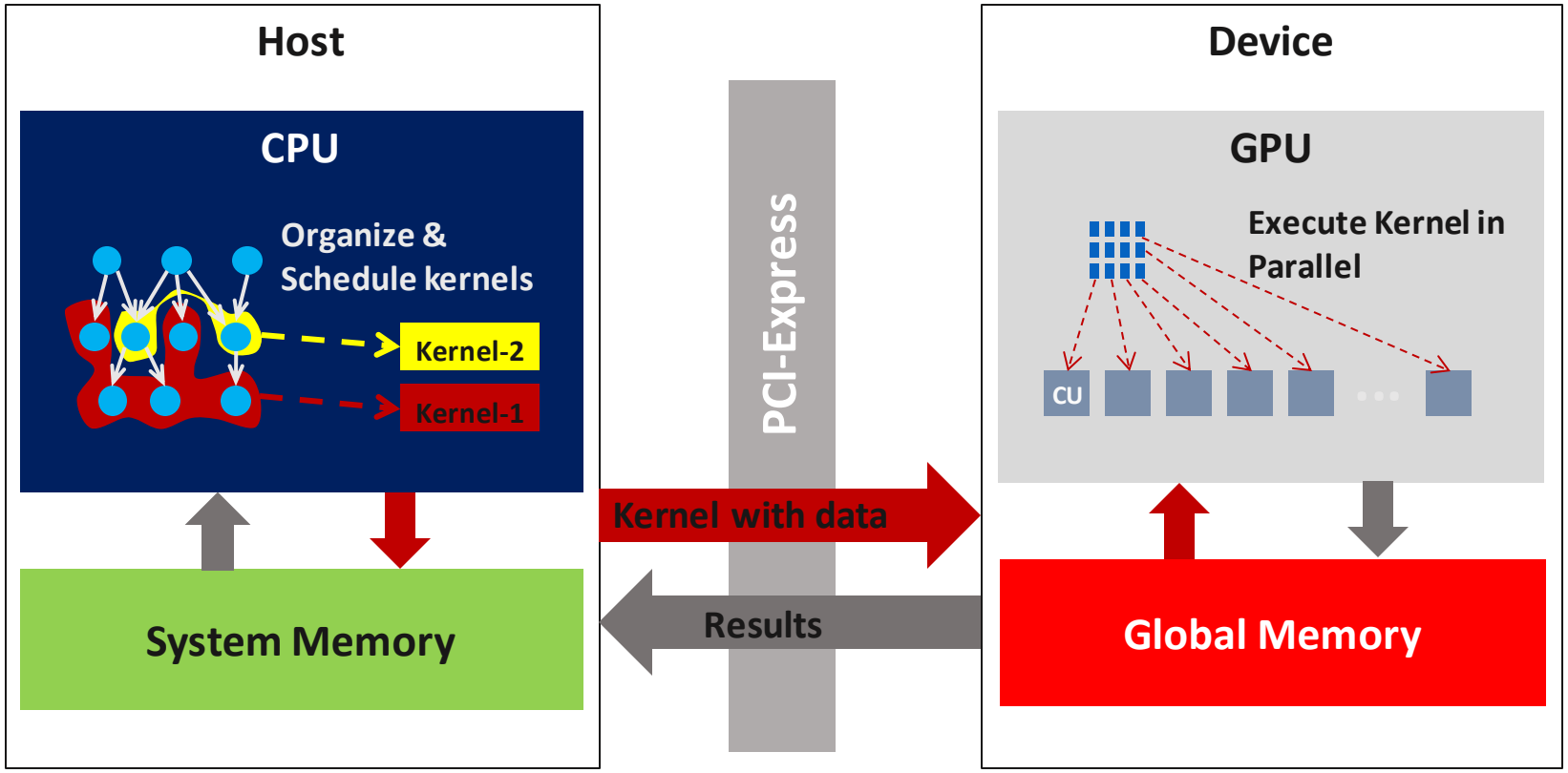
Note: GPU and CPU computations and communications can overlap

Program Execution (2)

- Allocate memory for the computation on the GPU
- Copy data from CPU memory to GPU memory
- GPU runs the kernel code specified by CPU in parallel
- Copy the result from GPU memory to CPU memory

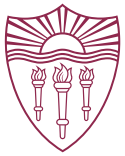


Program Execution (3)



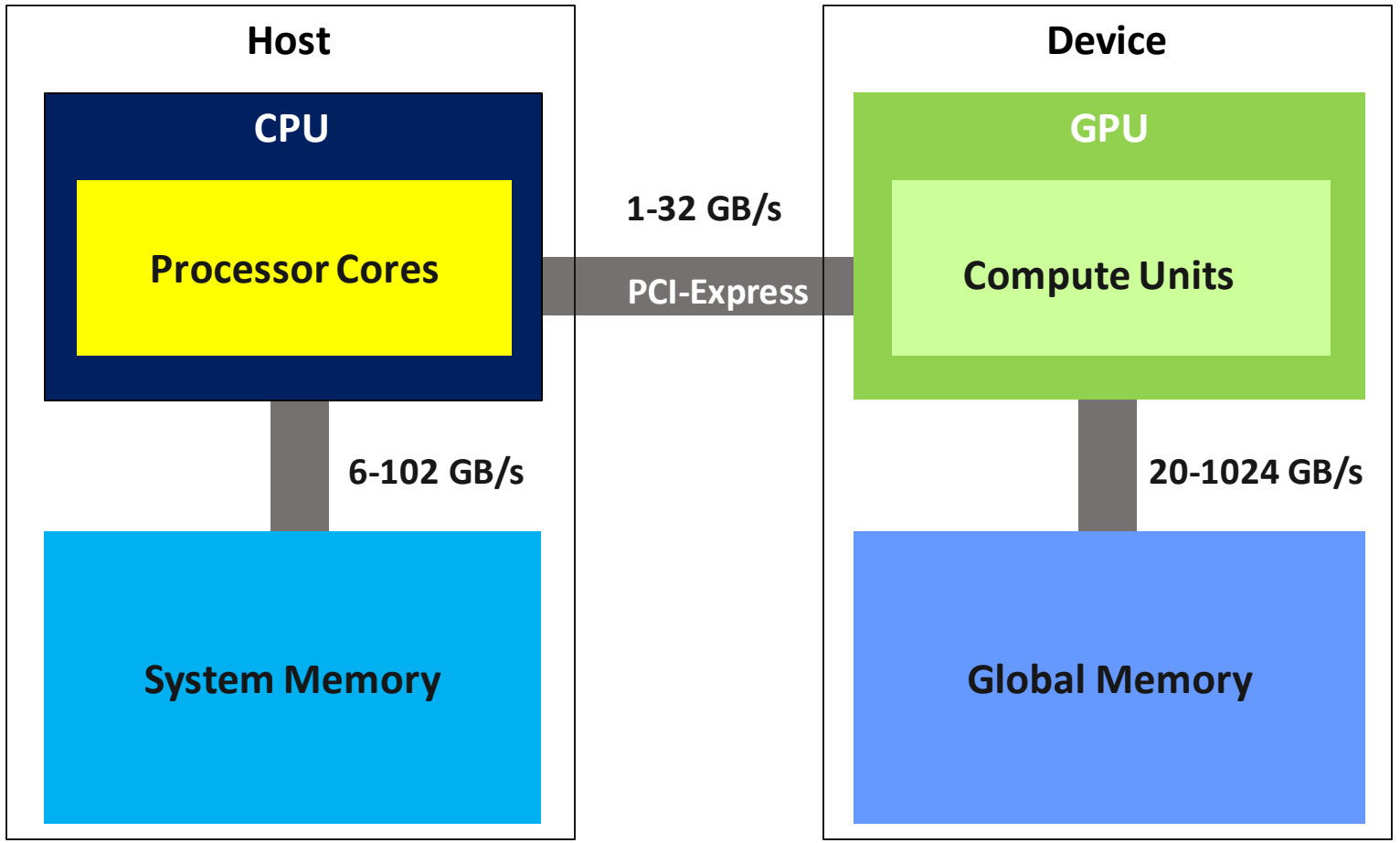
CUDA (Compute Unified Device Architecture)

Outline

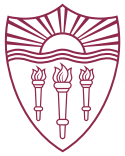


- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
  - Programming Model
  - GPU Architecture
  - Execution Model
  - CUDA, OpenCL, …

System Organization



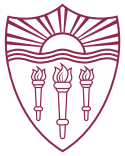
PCI Express



- Peripheral Component Interconnect Express
  - Notation: PCI-E or PCIe
  - For attaching hardware devices in a computer
  - Full-duplex communication
  - Latest standard: PCIe 5.0
    - PCIe 6.0 expected in year 2021

- Lane
  - ×1, ×2, ×4, ×8, **×16** (common use), ×32
  - More lanes → link width ↑
  - Automatically configured

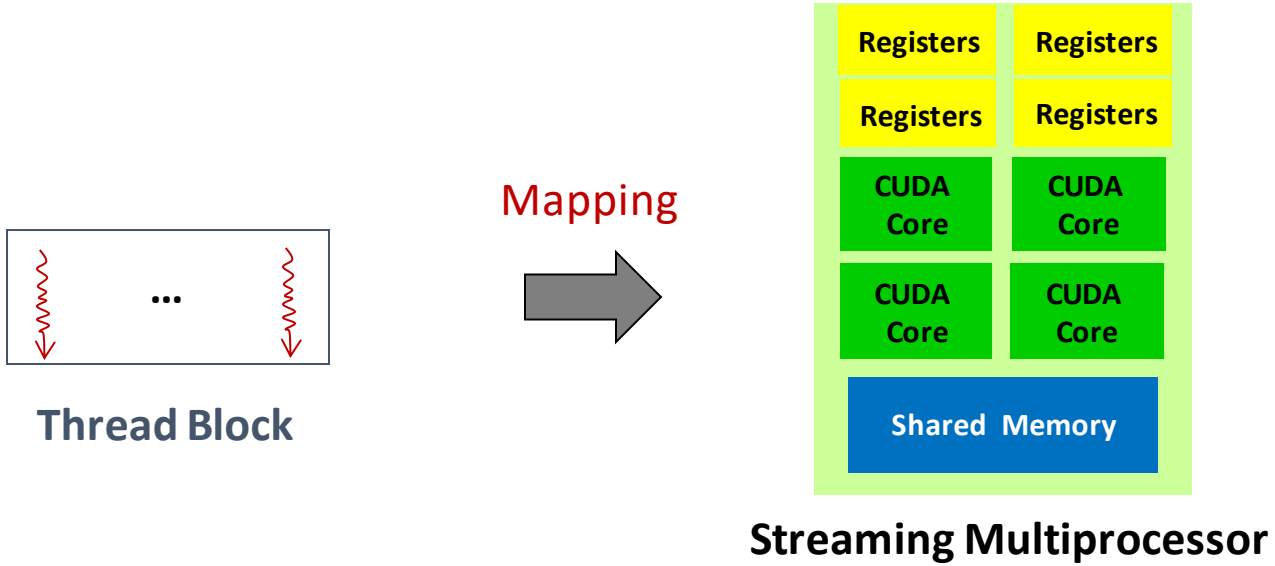
PCI Express Bandwidth



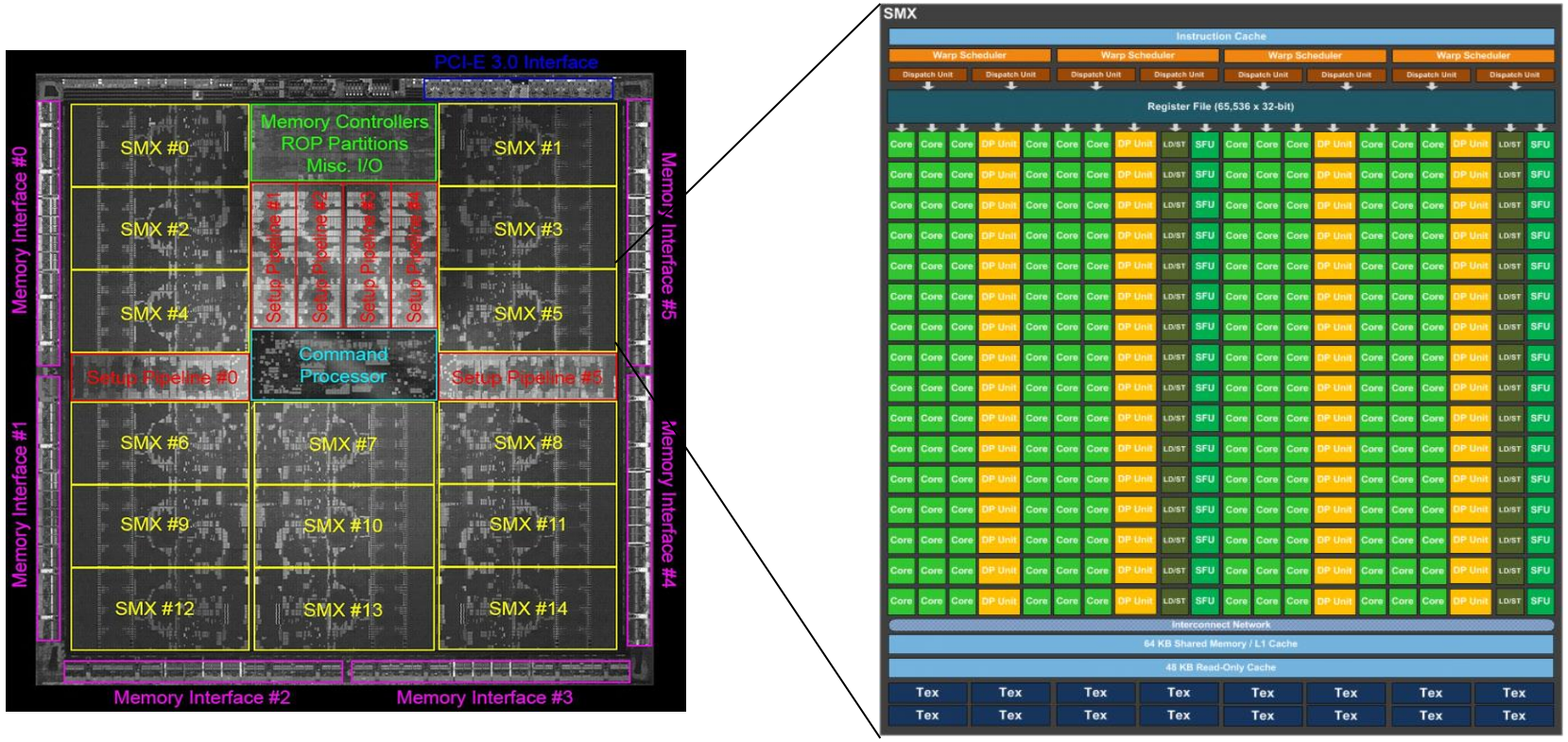
For a 16-lane bus, in each direction			
PCIe Generation	Clock rate	Total Bandwidth (x16)	Year
PCIe 1.X	2.5 GHz	4 GB/s	2005
PCIe 2.X	5 GHz	8 GB/s	2007
PCIe 3.0	8 GHz	16 GB/s	2010
PCIe 4.0	16 GHz	32 GB/s	2018
PCIe 5.0	32 GHz	63 GB/s	2019

Streaming Multiprocessor (SM) (1)

- NVIDIA GPU architecture:
  - Scalable array of multithreaded Streaming Multiprocessors (SM)
- A SM is a compute unit, has multiple **CUDA Cores** (CC)

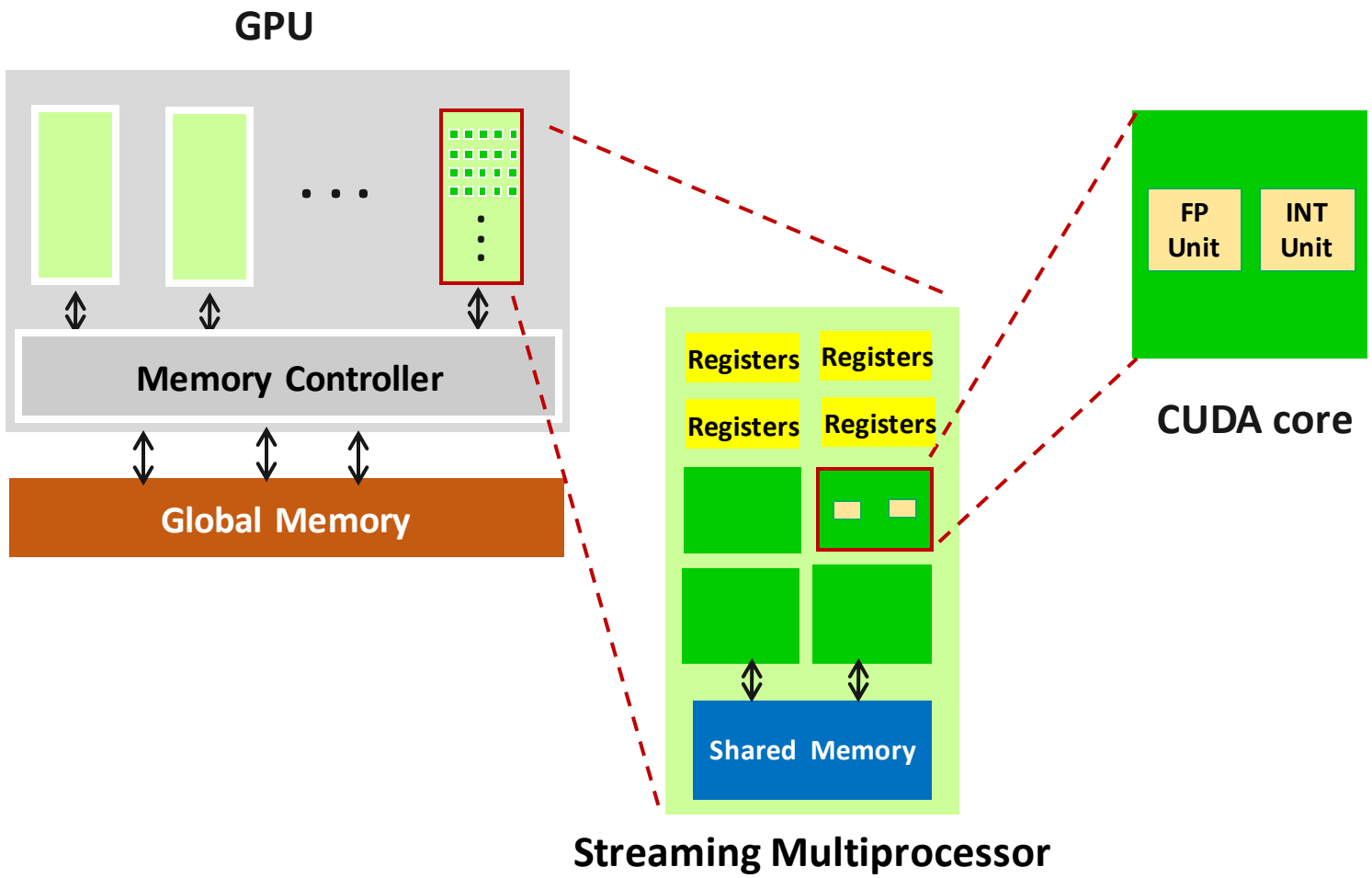


Streaming Multiprocessor (SM) (2)



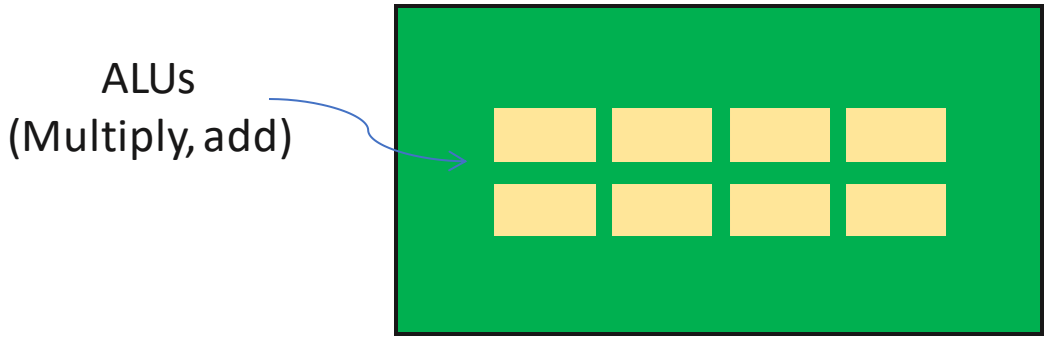
- Eg. Tesla V100
  - 80 Volta SM (SM: a new type of Streaming Multiprocessor)
  - 5376 CUDA cores

Inside GPU


GPU
Memory Controller
Global Memory
Registers
Registers
Registers
Registers
Shared Memory
Streaming Multiprocessor
FP Unit
INT Unit
CUDA core

Basic Organization (1)

CUDA Core (CC)

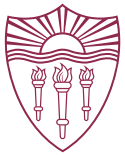


Ex. 8 ALUs (Multiply, add) = 8 functional units

Each functional unit can execute 2 flops per clock cycle
Each processing core runs at 1 GHz

8×2×1=16 GFlops/sec (peak performance)

Basic Organization (2)



Typical organization

16 SM (Streaming Multiprocessors)

128 CC/SM

8 ALU/CC (SIMD Functional units)

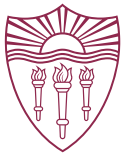
SIMT – (single instruction, multiple threads)

16×128×8=16 K (total # of ALUs)

Suppose 2 flops/cycle in each ALU

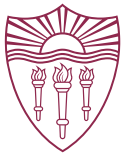
At 1 GHz 16×128×8×2= 32 TFlops/sec (peak performance)

Memory Model (1)



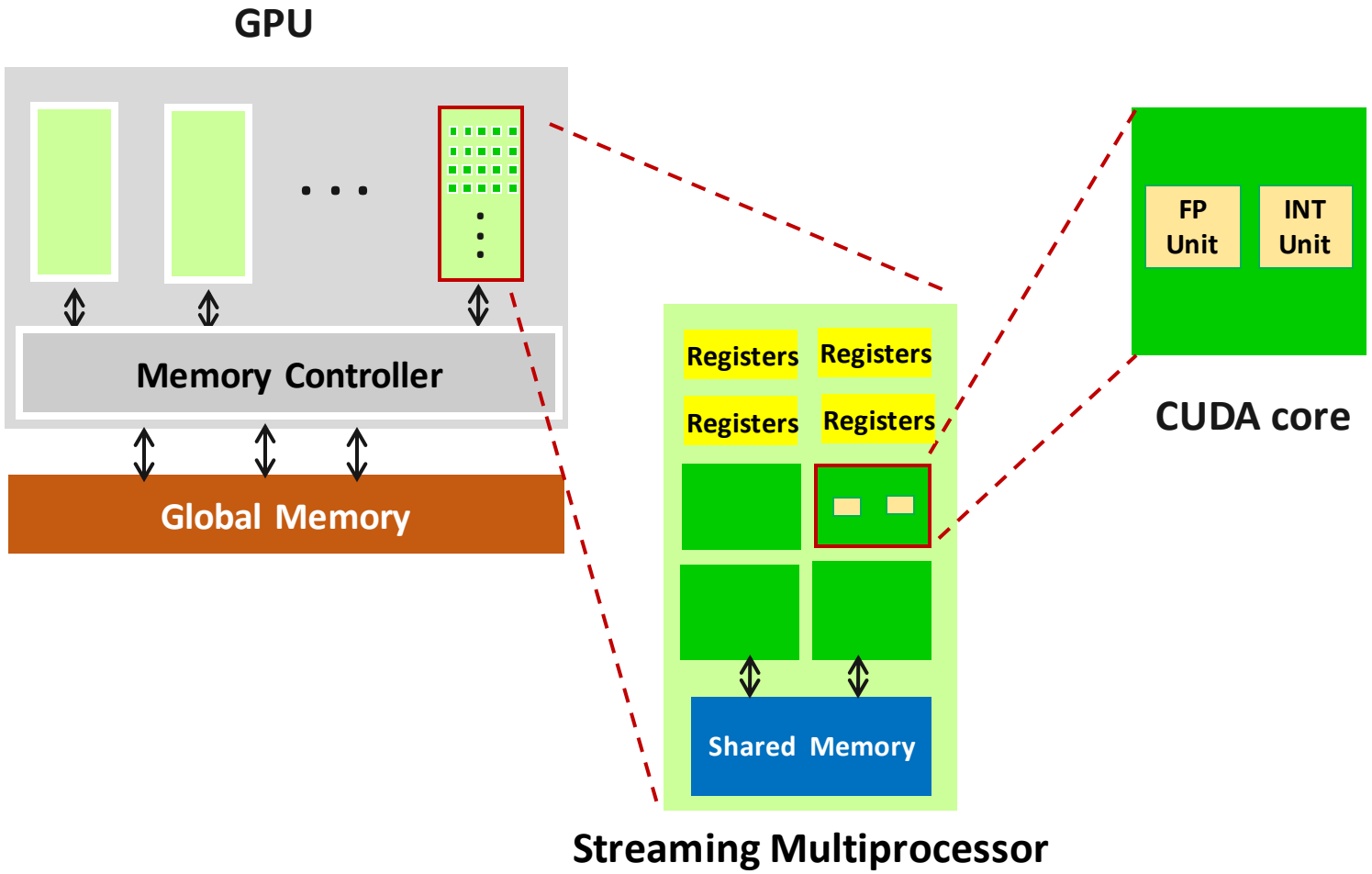
- Registers
  - Per thread
  - Fastest memory
  - Data lifetime = thread lifetime
- Shared memory
  - Each thread block has own shared memory
    - Accessible only by threads within that block
  - On-chip
  - Data lifetime = block lifetime
- Global (device) memory
  - Accessible by all threads as well as host (CPU)
  - Off-chip from the multiprocessors
  - Data lifetime = from allocation to deallocation
- Host (CPU) memory
  - Not directly accessible by CUDA threads

Memory Model (2)

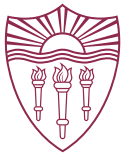


	Memory Type		
	Registers	**Shared Memory**	**Global Memory**
Bandwidth	8 TB/s	1.5 TB/s	180 GB/s
Latency	1 cycle	1-32 cycles	400-600 cycles

Memory Model (3)



Registers

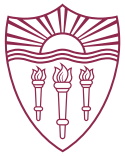


- Compiler will place variables declared in kernel in registers when possible
- Limited number of registers
- Eg. Nvidia Fermi has 8 K 32-bit registers

```
kernel() {
        int x, y;
        …
}
```

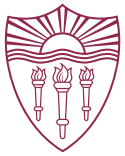
In registers

Shared Memory



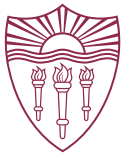
- Shared memory is a **user-controlled** L1 cache
- Shared memory is organized as a bank-switched architecture
  - Organized as 32 banks
  - Each bank can serve only a **single** operation per cycle
    - If >1 threads access the same bank at the same clock cycle→ **Bank Conflict**

Global Memory



- Writable from both the GPU and the CPU
- Can be accessed from any device on the PCI-E bus
- Off-chip from the multiprocessors
  - Long access latency
- Global memory is also partitioned into banks

Summary



- Architecture, Programming and Execution Model of (GP) GPU
  - GPU and data parallel programming
    - SIMT It means Shared Memory Lock Instructions
    - Thread, thread block, grid
    - Thread hierarchy
    - Kernel
  - CUDA Programming Model
  - GPU Architecture
    - Streaming multiprocessor
    - Memory model