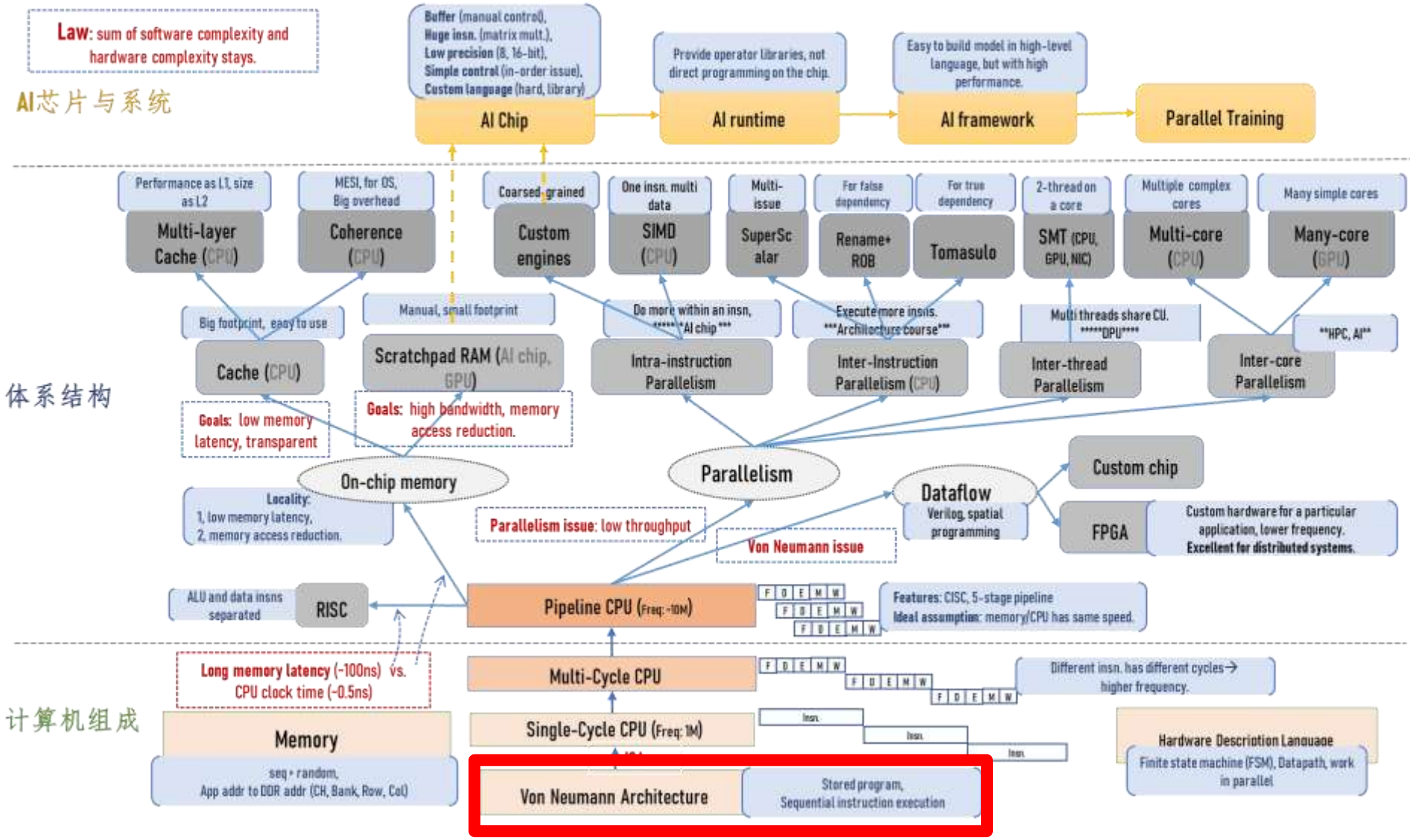
Graphics Processing Units

Prof. Zeke Wang

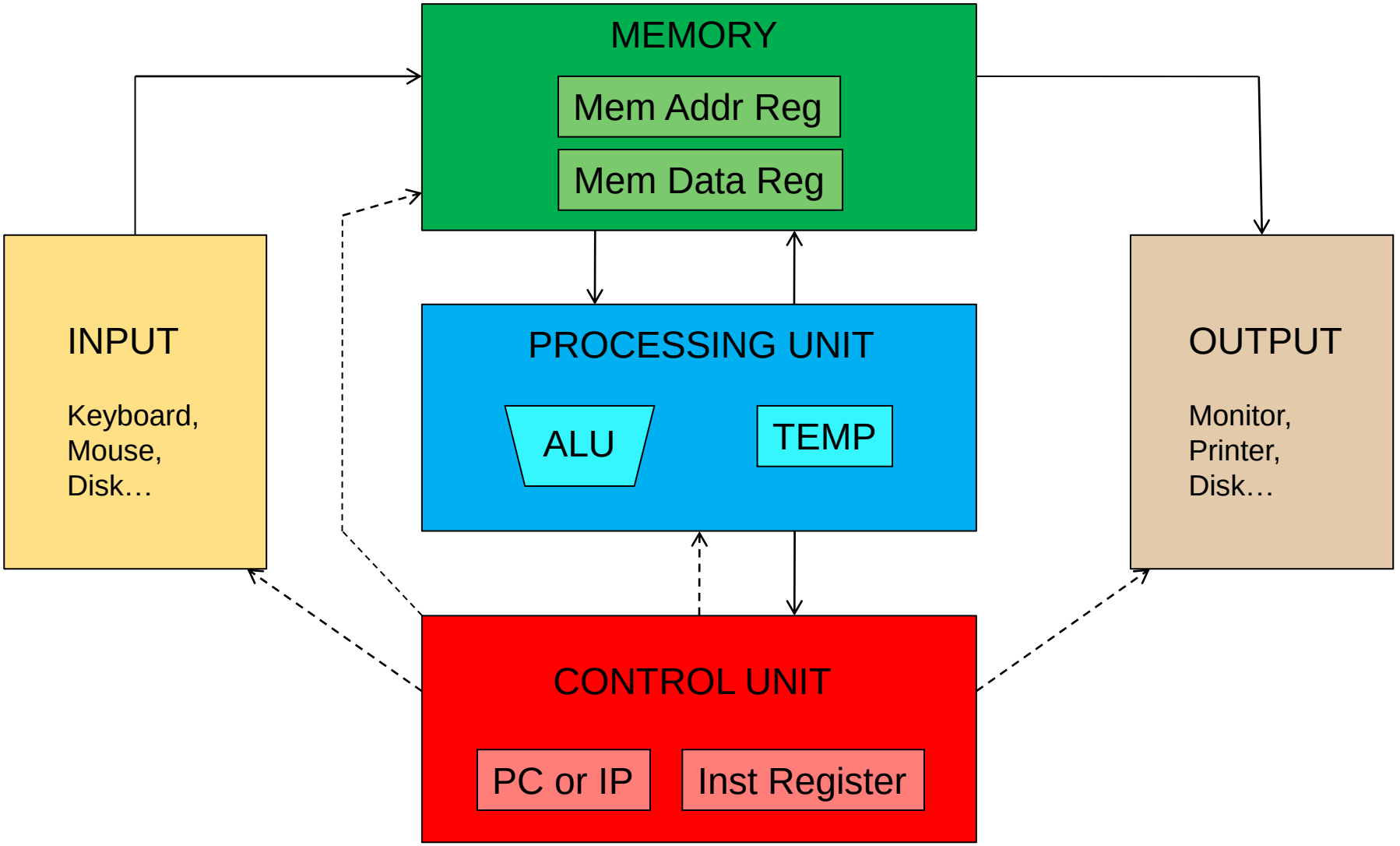
Zhejiang University

July 2024

Where Are We?


Law: sum of software complexity and hardware complexity stays.
AI芯片与系统
Buffer (manual control), Huge insn. (matrix mult.), Low precision (8, 16-bit), Simple control (in-order issue), Custom language (hard, library)
AI Chip
Provide operator libraries, not direct programming on the chip.
AI runtime
Easy to build model in high-level language, but with high performance.
AI framework
Parallel Training
Performance as L1, size as L2
Multi-layer Cache (CPU)
MESI, for OS, Big overhead
Coherence (CPU)
Coarsed-grained
Custom engines
One insn. multi data
SIMD (CPU)
Multi-issue
SuperScalar
For false dependency
For true dependency
Rename+ ROB
Tomasulo
2-thread on a core
SMT (CPU, GPU, NIC)
Multiple complex cores
Multi-core (CPU)
Many simple cores
Many-core (GPU)
体系结构
Big footprint, easy to use
Cache (CPU)
Goals: low memory latency, transparent
Manual, small footprint
Scratchpad RAM (AI chip, GPU)
Goals: high bandwidth, memory access reduction.
Do more within an insn, ***** *AI chip ***
Intra-instruction Parallelism
Execute more insns. ***Architecture course***
Inter-Instruction Parallelism (CPU)
Multi threads share CU. *****DPU*****
Inter-thread Parallelism
Inter-core Parallelism
HPC, AI
On-chip memory
Locality: 1, low memory latency, 2, memory access reduction.
Parallelism
Parallelism issue: low throughput
Von Neumann issue
Dataflow
Verilog, spatial programming
Custom chip
FPGA
Custom hardware for a particular application, lower frequency. Excellent for distributed systems.
ALU and data insns separated
RISC
Pipeline CPU (Freq: ~10M)
F D E M W
Features: CISC, 5-stage pipeline
Ideal assumption: memory/CPU has same speed.
计算机组成
Long memory latency (~100ns) vs. CPU clock time (~0.5ns)
Multi-Cycle CPU
Different insn. has different cycles→ higher frequency.
Memory
seq + random, App addr to DDR addr (CH, Bank, Row, Col)
Single-Cycle CPU (Freq: 1M)
insn.
Hardware Description Language
Finite state machine (FSM), Datapath, work in parallel
Von Neumann Architecture
Stored program, Sequential instruction execution

The von Neumann Model



von Neumann Model: Two Key Properties

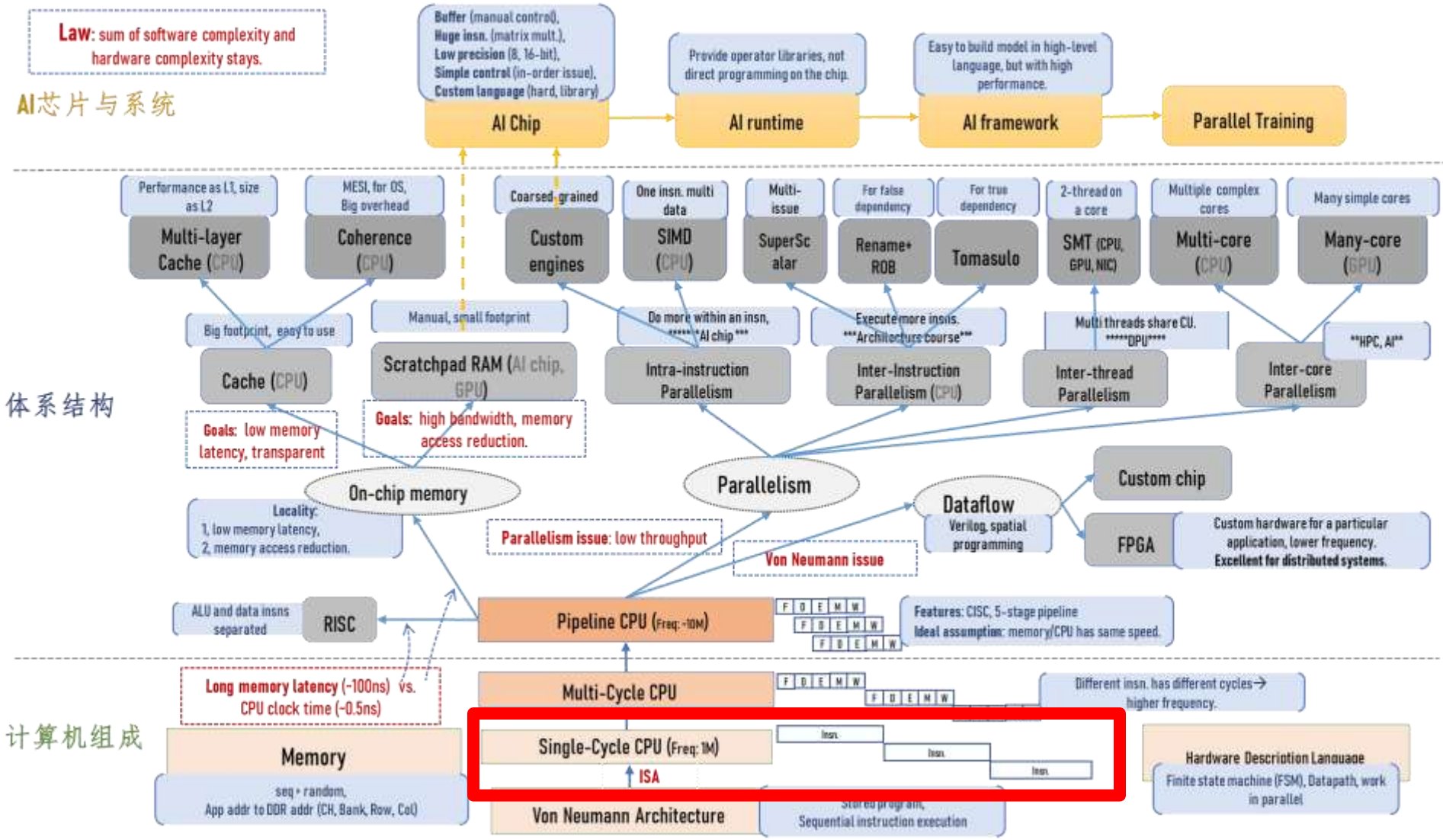
- Von Neumann model is also called *stored program computer* (instructions in memory).

- von Neumann Model has two key properties:

  - **1, Stored program**
    - Instructions stored in a linear memory array
    - Memory is unified between instructions and data
      - The interpretation of a stored value depends on the control signals

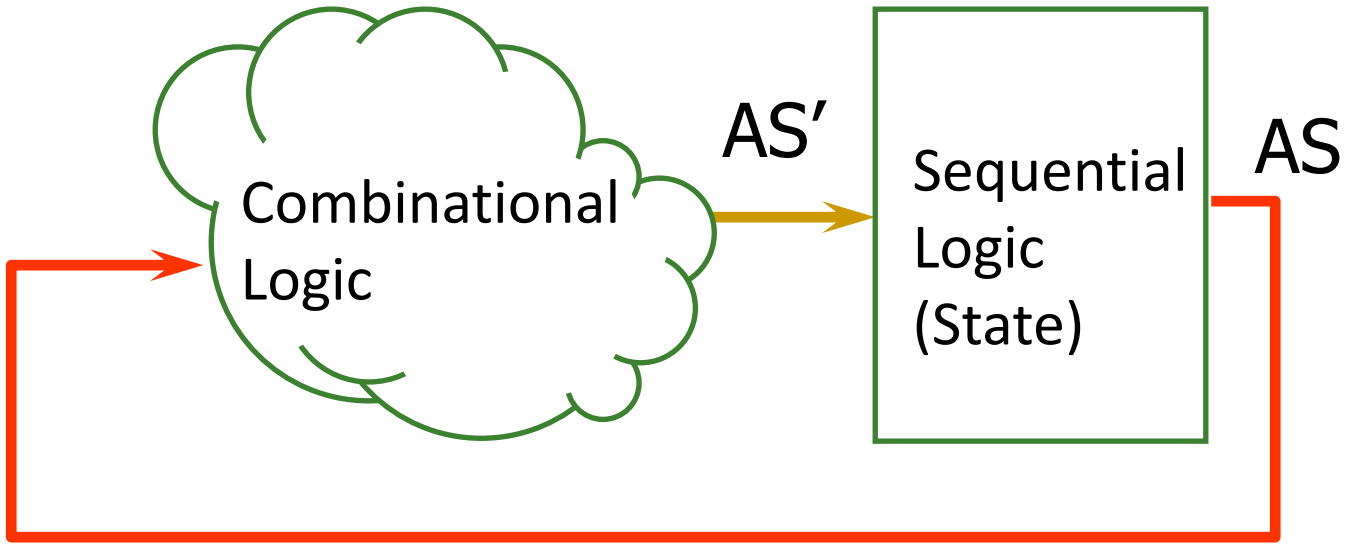
  - **2, Sequential instruction processing**
    - One instruction processed (fetched, executed, completed) at a time
    - Program counter (instruction pointer) identifies the current instruction
    - Program counter is advanced sequentially except for control transfer instructions

Where Are We?


Law: sum of software complexity and hardware complexity stays.
AI芯片与系统
Buffer (manual control), Huge insn. (matrix mult.), Low precision (8, 16-bit), Simple control (in-order issue), Custom language (hard, library)
AI Chip
Provide operator libraries, not direct programming on the chip.
AI runtime
Easy to build model in high-level language, but with high performance.
AI framework
Parallel Training
Performance as L3, size as L2
Multi-layer Cache (CPU)
MESI, for OS, Big overhead
Coherence (CPU)
Coarsed-grained
Custom engines
One insn. multi data
SIMD (CPU)
Multi-issue
SuperScalar
For false dependency
For true dependency
Rename+ ROB
Tomasulo
2-thread on a core
SMT (CPU, GPU, NIC)
Multiple complex cores
Multi-core (CPU)
Many simple cores
Many-core (GPU)
体系结构
Big footprint, easy to use
Cache (CPU)
Goals: low memory latency, transparent
Manual, small footprint
Scratchpad RAM (AI chip, GPU)
Goals: high bandwidth, memory access reduction.
Do more within an insn, ***** *AI chip ***
Intra-instruction Parallelism
Execute more insns. ***Architecture course***
Inter-Instruction Parallelism (CPU)
Multi threads share CU. *****DPU*****
Inter-thread Parallelism
HPC, AI
Inter-core Parallelism
On-chip memory
Locality: 1, low memory latency, 2, memory access reduction.
Parallelism
Parallelism issue: low throughput
Von Neumann issue
Dataflow
Verilog, spatial programming
Custom chip
FPGA
Custom hardware for a particular application, lower frequency. Excellent for distributed systems.
ALU and data insns separated
RISC
Pipeline CPU (Freq: ~10M)
F D E M W
Features: CISC, 5-stage pipeline
Ideal assumption: memory/CPU has same speed.
计算机组成
Long memory latency (~100ns) vs. CPU clock time (~0.5ns)
Multi-Cycle CPU
Different insn. has different cycles→ higher frequency.
Memory
seq + random, App addr to DDR addr (CH, Bank, Row, Col)
Single-Cycle CPU (Freq: 1M)
ISA
Insn.
Hardware Description Language
Finite state machine (FSM), Datapath, work in parallel
Von Neumann Architecture
Stored program, Sequential instruction execution

A Single-Cycle Microarchitecture
A Closer Look

Single-cycle Machine

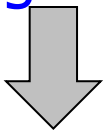


AS: Architectural State

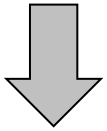
A Very Basic Instruction Processing Engine

- Each instruction takes a single clock cycle to execute.
- Only combinational logic is used to implement instruction execution.
  - *No intermediate, programmer-invisible state updates*

AS = Architectural (programmer visible) state
at the beginning of a clock cycle



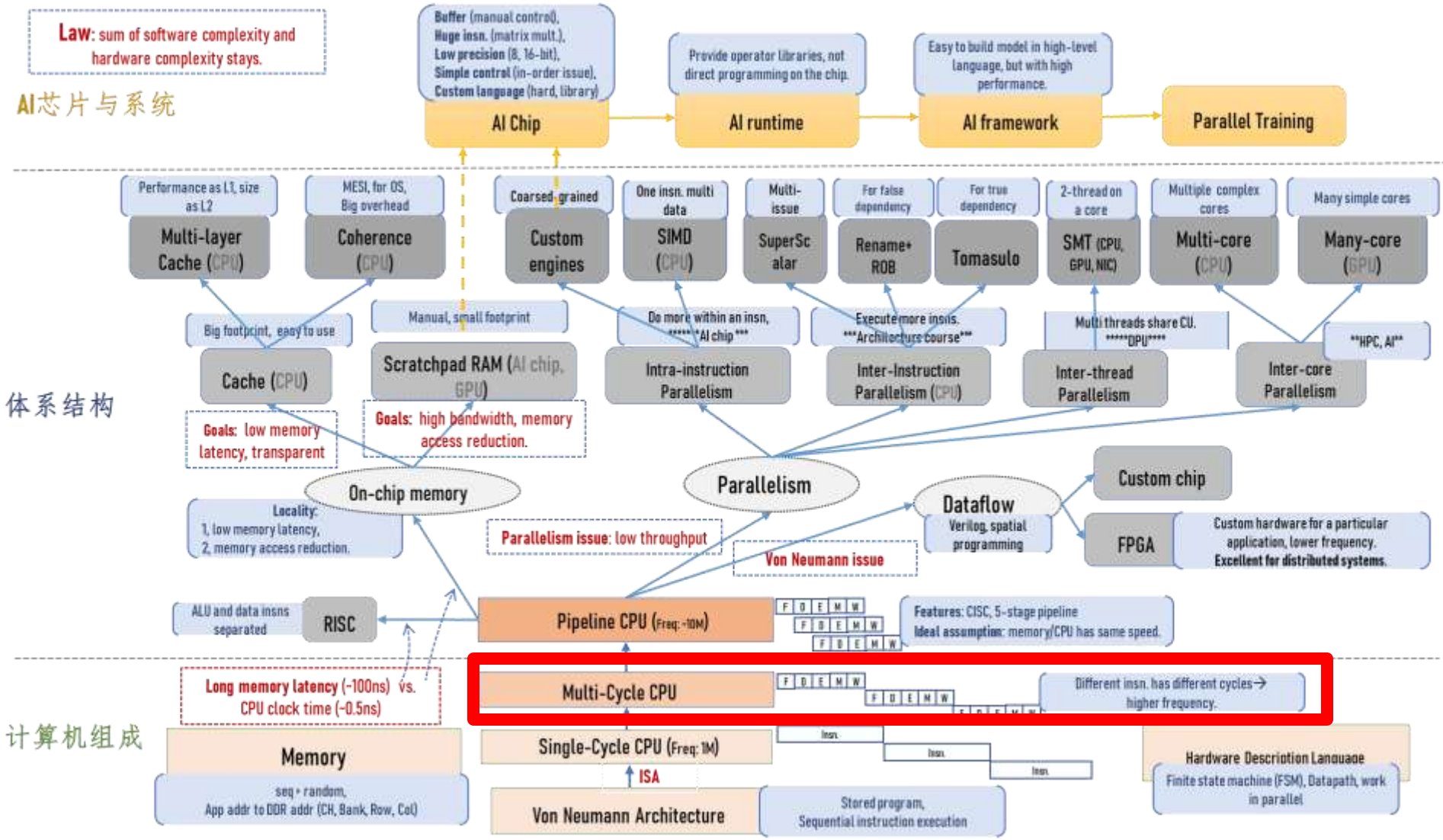
Process instruction in one clock cycle



AS' = Architectural (programmer visible) state
at the end of a clock cycle

Multi-Cycle Microarchitectures

Where Are We?


Law: sum of software complexity and hardware complexity stays.
AI芯片与系统
Buffer (manual control), Huge insn. (matrix mult.), Low precision (8, 16-bit), Simple control (in-order issue), Custom language (hard, library)
AI Chip
Provide operator libraries, not direct programming on the chip.
AI runtime
Easy to build model in high-level language, but with high performance.
AI framework
Parallel Training
Performance as L1, size as L2
Multi-layer Cache (CPU)
MESI, for OS, Big overhead
Coherence (CPU)
Coarsed-grained
Custom engines
One insn. multi data
SIMD (CPU)
Multi-issue
SuperScalar
For false dependency
Rename+ ROB
For true dependency
Tomasulo
2-thread on a core
SMT (CPU, GPU, NIC)
Multiple complex cores
Multi-core (CPU)
Many simple cores
Many-core (GPU)
体系结构
Big footprint, easy to use
Cache (CPU)
Goals: low memory latency, transparent
Manual, small footprint
Scratchpad RAM (AI chip, GPU)
Goals: high bandwidth, memory access reduction.
Do more within an insn, ***** *AI chip ***
Intra-instruction Parallelism
Execute more insns. ***Architecture course***
Inter-Instruction Parallelism (CPU)
Multi threads share CU. *****DPU*****
Inter-thread Parallelism
HPC, AI
Inter-core Parallelism
On-chip memory
Locality: 1, low memory latency, 2, memory access reduction.
Parallelism
Parallelism issue: low throughput
Von Neumann issue
Dataflow
Verilog, spatial programming
Custom chip
FPGA
Custom hardware for a particular application, lower frequency. Excellent for distributed systems.
ALU and data insns separated
RISC
Pipeline CPU (Freq: ~10M)
F D E M W
Features: CISC, 5-stage pipeline
Ideal assumption: memory/CPU has same speed.
Long memory latency (~100ns) vs. CPU clock time (~0.5ns)
Multi-Cycle CPU
Different insn. has different cycles→ higher frequency.
计算机组成
Memory
seq > random, App addr to DDR addr (CH, Bank, Row, Col)
Single-Cycle CPU (Freq: 1M)
insn.
ISA
Von Neumann Architecture
Stored program, Sequential instruction execution
Hardware Description Language
Finite state machine (FSM), Datapath, work in parallel

Multi-Cycle Microarchitectures

- Goal: Let each instruction take (close to) only as much time it really needs

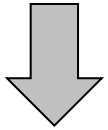
- Idea of multi-cycle CPU:
  - Decrease clock cycle time
  - Each instruction takes as many clock cycles as it needs to take
    - Multiple state transitions per instruction
    - The states followed by each instruction is different

The "Process Instruction" Step of Multi-Cycle CPU

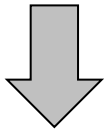
- ISA specifies abstractly what AS' should be, given an instruction and AS
  - It defines an abstract finite state machine where
    - State = programmer-visible state
    - Next-state logic = instruction execution specification
  - From ISA point of view, there are no "intermediate states" between AS and AS' during instruction execution
    - One state transition per instruction
- Microarchitecture implements how AS is transformed to AS'
  - We can have programmer-invisible state to optimize the speed of instruction execution: **multiple** state transitions per instruction
    - Single-cycle: AS → AS' (transform AS to AS' in a single clock cycle)
    - Multi-cycle: AS → AS+MS1 → AS+MS2 → AS+MS3 → AS' (take multiple clock cycles to transform AS to AS')

Multi-Cycle Microarchitecture

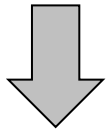
AS = Architectural (programmer visible) state
at the beginning of an instruction



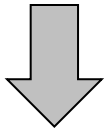
Step 1: Process part of instruction in one clock cycle



Step 2: Process part of instruction in the next clock cycle

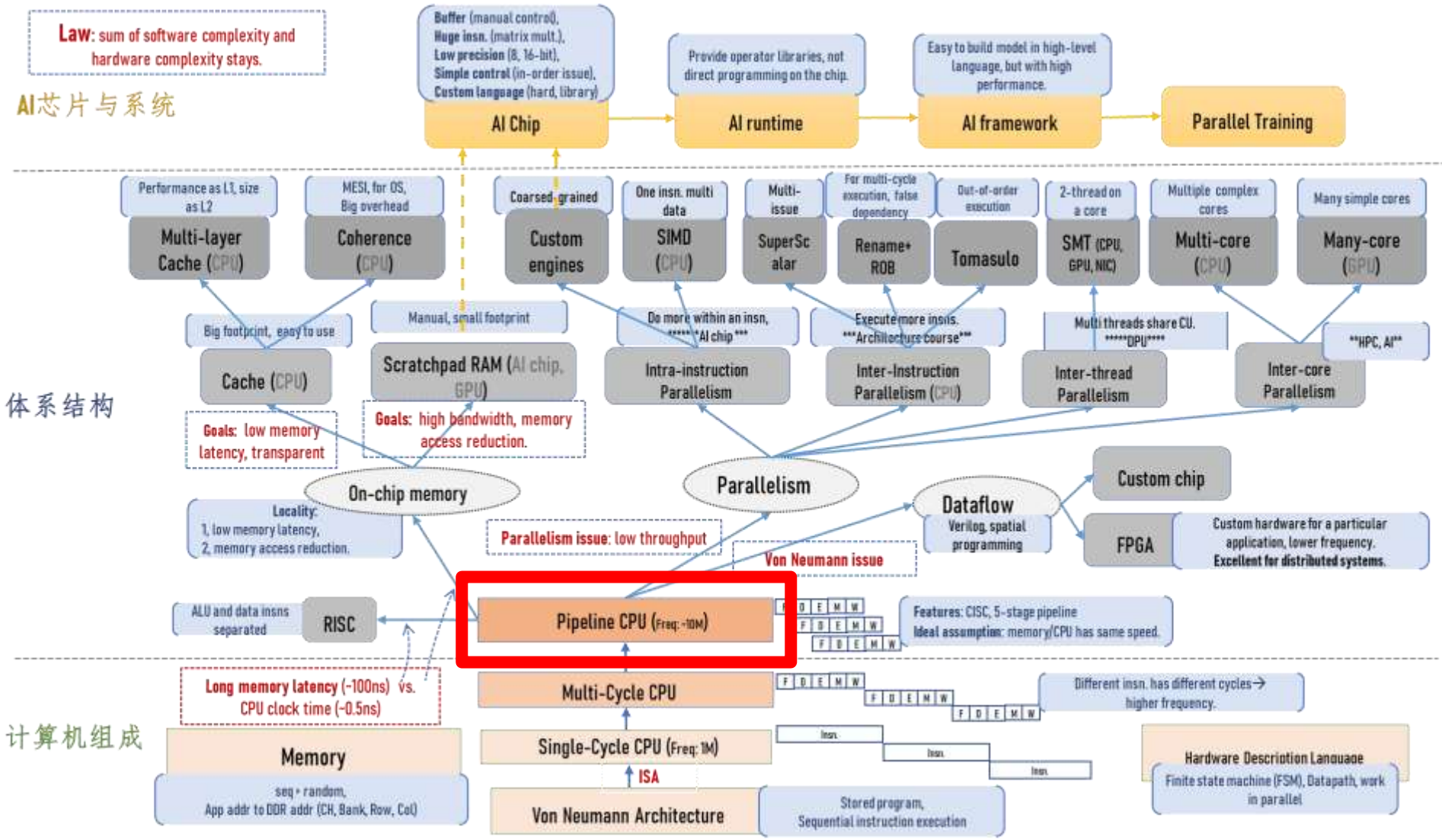


...



AS' = Architectural (programmer visible) state
at the end of a clock cycle

Where Are We?


Law: sum of software complexity and hardware complexity stays.
AI芯片与系统
Buffer (manual control), Huge insn. (matrix mult.), Low precision (8, 16-bit), Simple control (in-order issue), Custom language (hard, library)
AI Chip
Provide operator libraries, not direct programming on the chip.
AI runtime
Easy to build model in high-level language, but with high performance.
AI framework
Parallel Training
Performance as L1, size as L2
Multi-layer Cache (CPU)
MESI, for OS, Big overhead
Coherence (CPU)
Coarsed-grained
Custom engines
One insn. multi data
SIMD (CPU)
Multi-issue
SuperScalar
For multi-cycle execution, false dependency
Rename+ ROB
Out-of-order execution
Tomasulo
2-thread on a core
SMT (CPU, GPU, NIC)
Multiple complex cores
Multi-core (CPU)
Many simple cores
Many-core (GPU)
体系结构
Big footprint, easy to use
Cache (CPU)
Goals: low memory latency, transparent
Manual, small footprint
Scratchpad RAM (AI chip, GPU)
Goals: high bandwidth, memory access reduction.
On-chip memory
Locality:
1, low memory latency,
2, memory access reduction.
Do more within an insn, ******AI chip***
Intra-instruction Parallelism
Execute more insns. ***Architecture course***
Inter-Instruction Parallelism (CPU)
Multi threads share CU. ******DPU******
Inter-thread Parallelism
HPC, AI
Inter-core Parallelism
Parallelism
Parallelism issue: low throughput
Von Neumann issue
Dataflow
Verilog, spatial programming
Custom chip
FPGA
Custom hardware for a particular application, lower frequency.
Excellent for distributed systems.
ALU and data insns separated
RISC
Pipeline CPU (Freq: ~10M)
F D E M W
Features: CISC, 5-stage pipeline
Ideal assumption: memory/CPU has same speed.
计算机组成
Long memory latency (~100ns) vs. CPU clock time (~0.5ns)
Multi-Cycle CPU
Different insn. has different cycles→ higher frequency.
Memory
seq + random,
App addr to DDR addr (CH, Bank, Row, Col)
Single-Cycle CPU (Freq: 1M)
Insn.
ISA
Von Neumann Architecture
Stored program,
Sequential instruction execution
Hardware Description Language
Finite state machine (FSM), Datapath, work in parallel

Can We Use the Idle Hardware to Improve Concurrency?

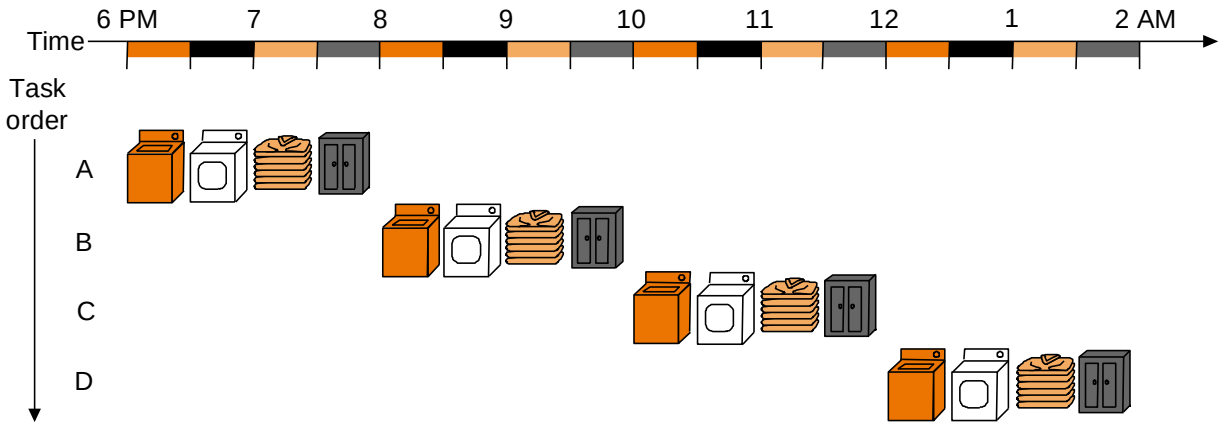
- **Goal**: More concurrency → Higher instruction throughput
  (i.e., more "work" completed in one cycle)

- **Key Idea**: When an instruction is using some resources in its processing phase, process other instructions on idle resources not needed by that instruction
  - E.g., when an instruction is being decoded, fetch the next instruction
  - E.g., when an instruction is being executed, decode another instruction
  - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
  - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

Pipelining: Basic Idea

- More systematically:
  - Pipeline the execution of multiple instructions
  - Analogy: "Assembly line processing" of instructions
- Idea of pipelining:
  - Divide the instruction processing cycle into distinct "stages" of processing
  - Ensure enough hardware resources to process one instruction in each stage
  - Process a **different** instruction in each stage
    - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)

The Laundry Analogy: Pipeline

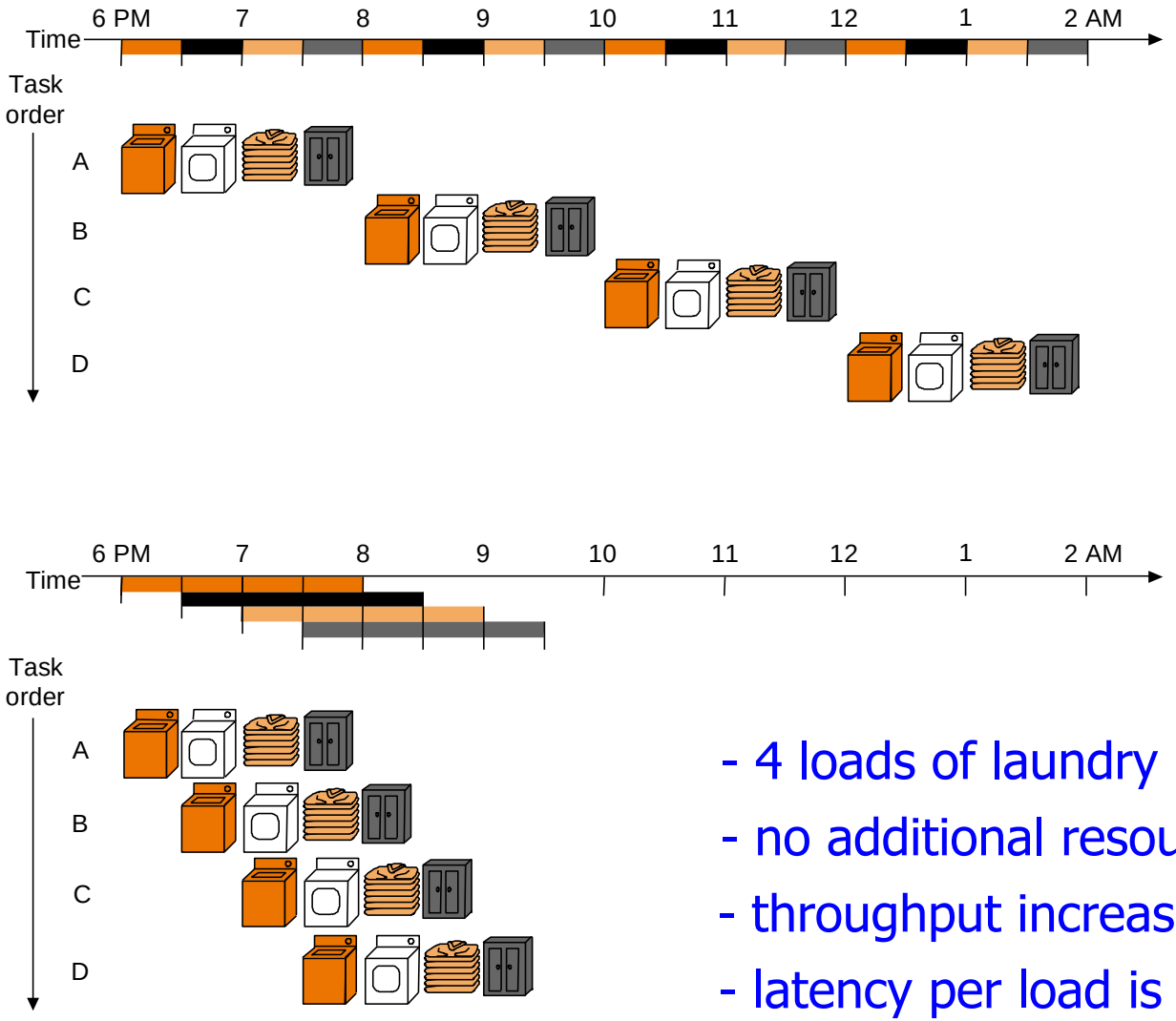


- "place one dirty load of clothes in the washer",
- "when the washer is finished, place the wet load in the dryer",
- "when the dryer is finished, take out the dry load and fold",
- "when folding is finished, put the clothes away".

Observations:

1, steps to do a load are sequentially dependent,
2, different steps do not share resources,
3, no dependence between different loads.

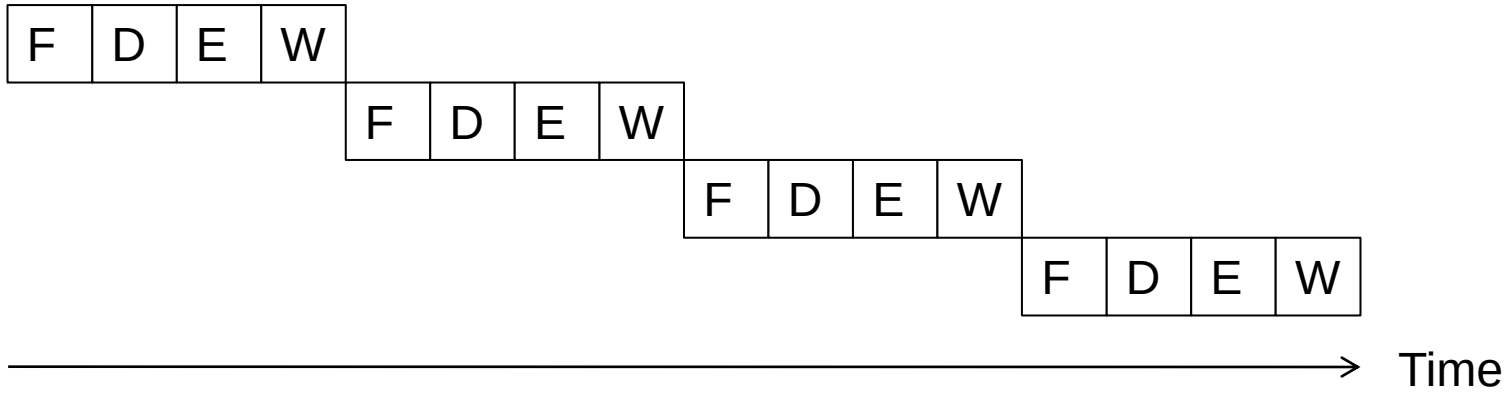
Pipelining Multiple Loads of Laundry



- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4X
- latency per load is the same

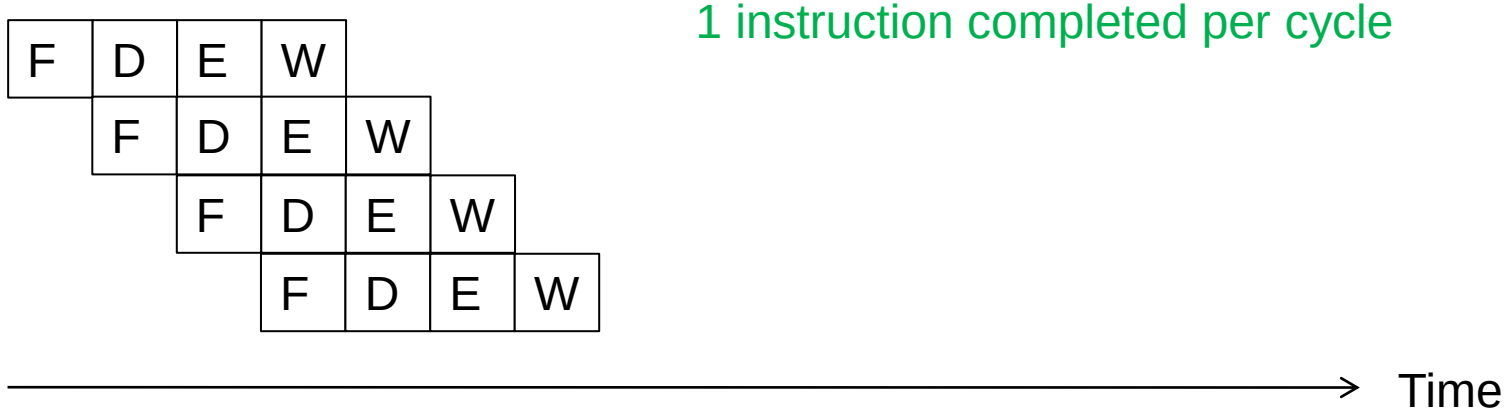
Example: Execution of Four Independent ADDs

- Multi-cycle: 4 cycles per instruction

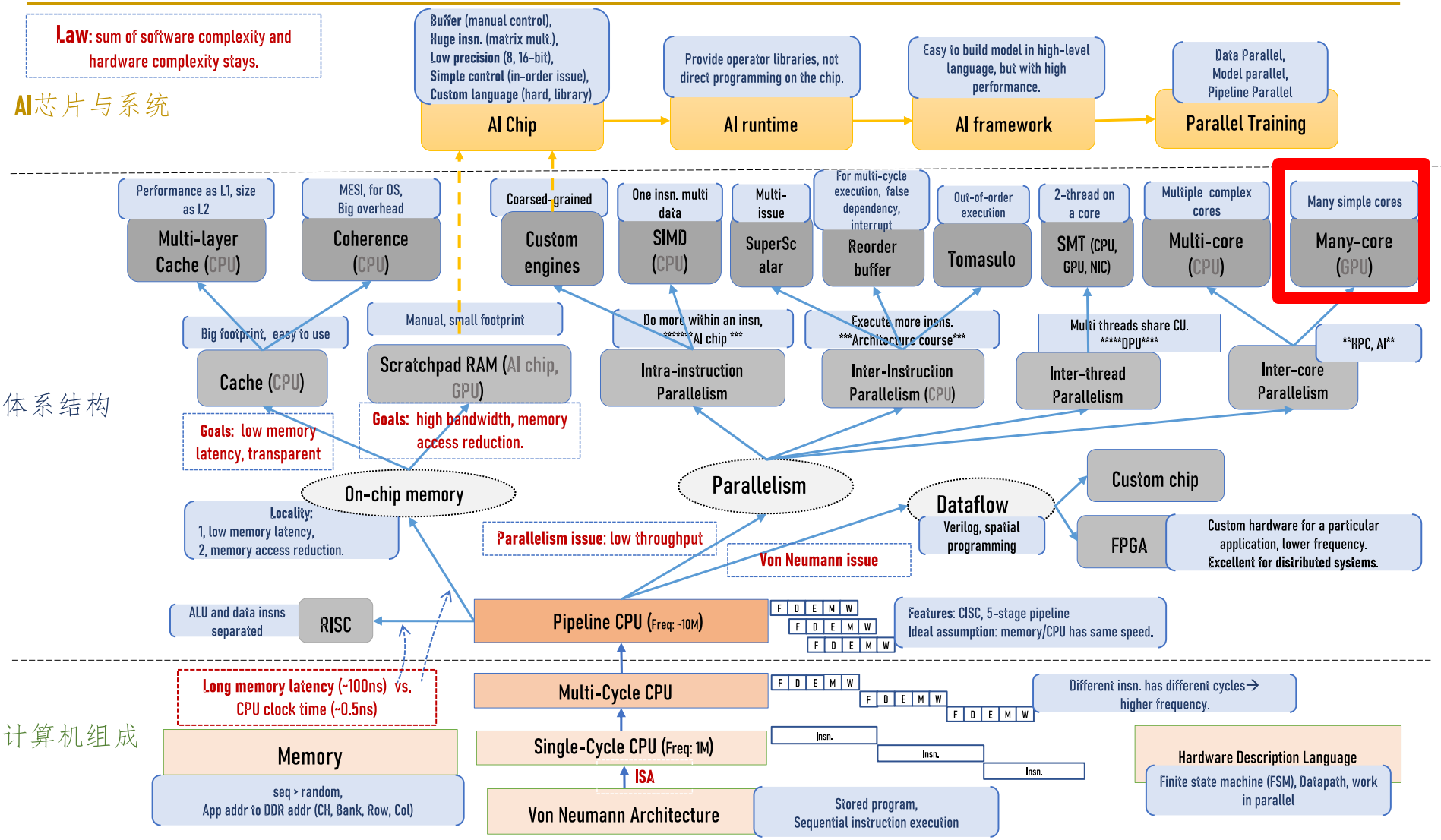


- Pipelined: 4 cycles per 4 instructions (steady state)

1 instruction completed per cycle



Where Are We?


Law: sum of software complexity and hardware complexity stays.
AI芯片与系统
Buffer (manual control), Huge insn. (matrix mult.), Low precision (8, 16-bit), Simple control (in-order issue), Custom language (hard, library)
AI Chip
Provide operator libraries, not direct programming on the chip.
AI runtime
Easy to build model in high-level language, but with high performance.
AI framework
Data Parallel, Model parallel, Pipeline Parallel
Parallel Training
Performance as L1, size as L2
Multi-layer Cache (CPU)
MESI, for OS, Big overhead
Coherence (CPU)
Coarsed-grained
Custom engines
One insn. multi data
SIMD (CPU)
Multi-issue
SuperScalar
For multi-cycle execution, false dependency, interrupt
Reorder buffer
Out-of-order execution
Tomasulo
2-thread on a core
SMT (CPU, GPU, NIC)
Multiple complex cores
Multi-core (CPU)
Many simple cores
Many-core (GPU)
体系结构
Big footprint, easy to use
Cache (CPU)
Goals: low memory latency, transparent
Manual, small footprint
Scratchpad RAM (AI chip, GPU)
Goals: high bandwidth, memory access reduction.
Do more within an insn, ******AI chip ***
Intra-instruction Parallelism
Execute more insns. ***Architecture course***
Inter-Instruction Parallelism (CPU)
Multi threads share CU. *****DPU****
Inter-thread Parallelism
HPC, AI
Inter-core Parallelism
On-chip memory
Locality: 1, low memory latency, 2, memory access reduction.
Parallelism
Parallelism issue: low throughput
Von Neumann issue
Dataflow
Verilog, spatial programming
Custom chip
FPGA
Custom hardware for a particular application, lower frequency. Excellent for distributed systems.
ALU and data insns separated
RISC
Pipeline CPU (Freq: ~10M)
F D E M W
Features: CISC, 5-stage pipeline
Ideal assumption: memory/CPU has same speed.
计算机组成
Long memory latency (~100ns) vs. CPU clock time (~0.5ns)
Multi-Cycle CPU
Different insn. has different cycles→ higher frequency.
Memory
seq > random, App addr to DDR addr (CH, Bank, Row, Col)
Single-Cycle CPU (Freq: 1M)
Insn.
ISA
Von Neumann Architecture
Stored program, Sequential instruction execution
Hardware Description Language
Finite state machine (FSM), Datapath, work in parallel

Agenda for Today

- Why GPU?

- Hardware Execution Model

- Programming Model
  - SISD vs. SIMD vs. SPMD
  - GPU Programming Example

- Advance
  - SIMT (Hardware) & Warp (Software)

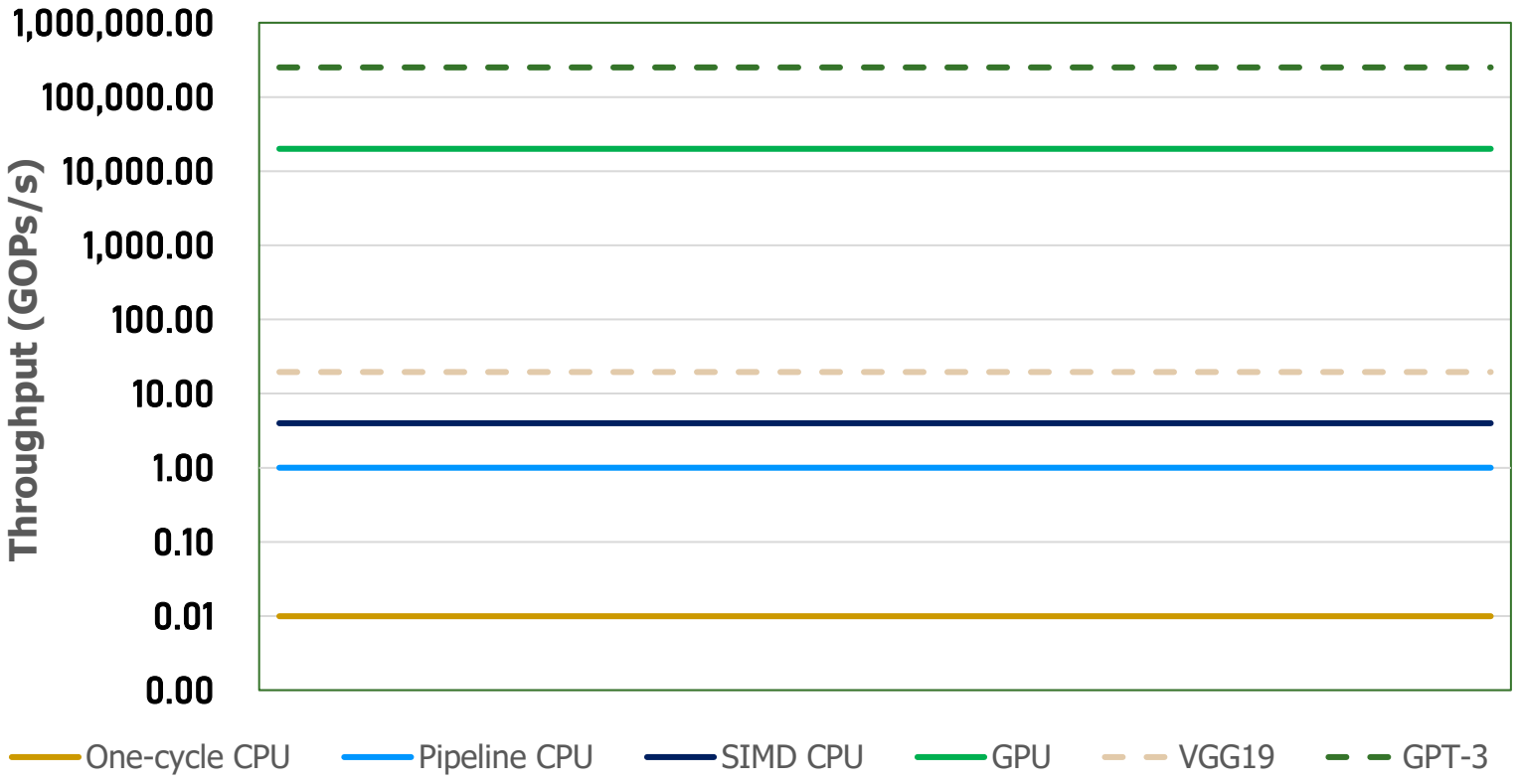
Why GPU?

Need More Computing Power.

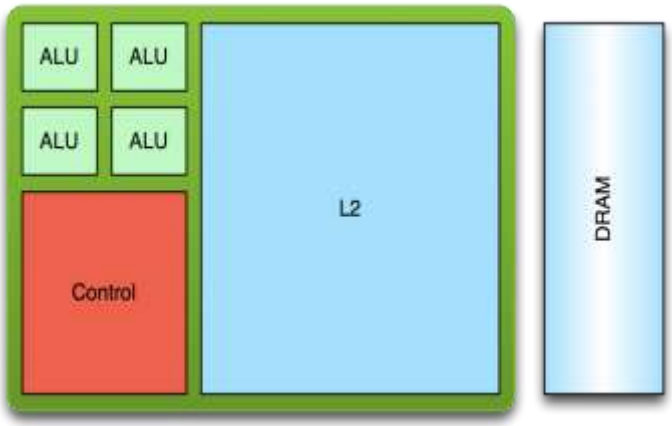
OpenAI: Compute Power Needed by NN Model

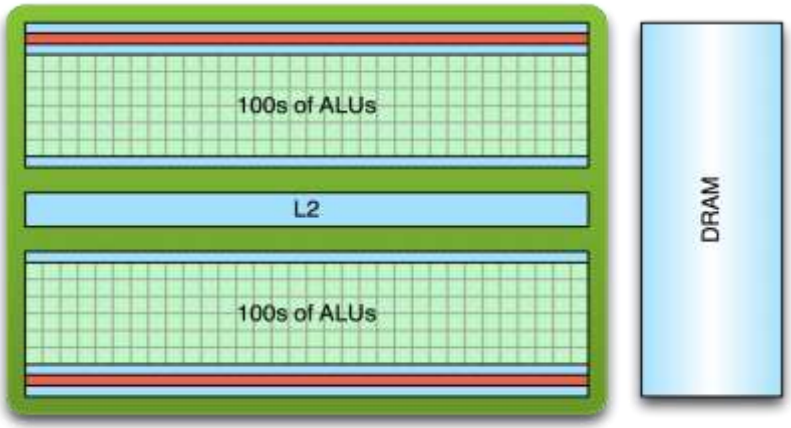
One Forward Pass of Model:

Model	Model Size	Compute/iteration (OPs)
VGG 19	114M	~19.6 B
"GPT-3"	175B	~250 T



CPU vs GPU: Compute Perspective





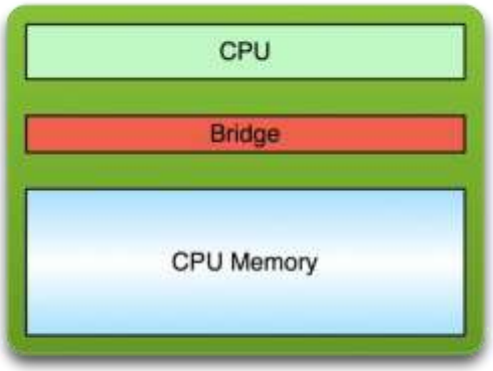
- **CPU**:
  - Few complex cores
  - Larger cache for low memory latency
  - Large and slow memory
- **GPU**:
  - Lots of simple cores
  - Small cache for low memory latency
  - Small and fast memory

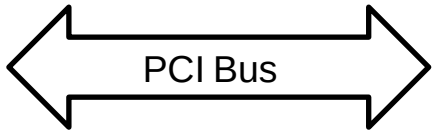
State-of-the-art CPU GPU and FPGA

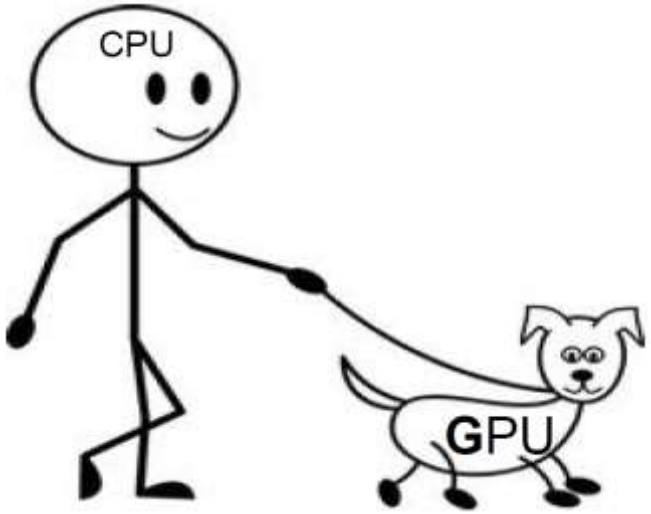
	Cores (Threads)	TFLOPS	Memory Size (Bandwidth)	PCIe	Network
CPU (AMD Threadripper 3995WX)	64 (128)	2.8 (FP32), 1.4 (FP64)	512GB (80GB/s)	32.0GB/s (PCIe 4.0 X16)	No
GPU (Nvidia H100)	18432 (128K)	67 (FP32), 34 (FP64), 989 (FP32, Tensor), 1979 (FP16, Tensor)	80GB (3350GB/s)	64.0GB/s (PCIe 5.0 X16)	No
FPGA (U280)	9,024 (25x18 MULs)	1.8 (FP32)	40GB (460GB/s)	16.0GB/s (PCIe 4.0 X8)	Yes

Relationship between CPU and GPU

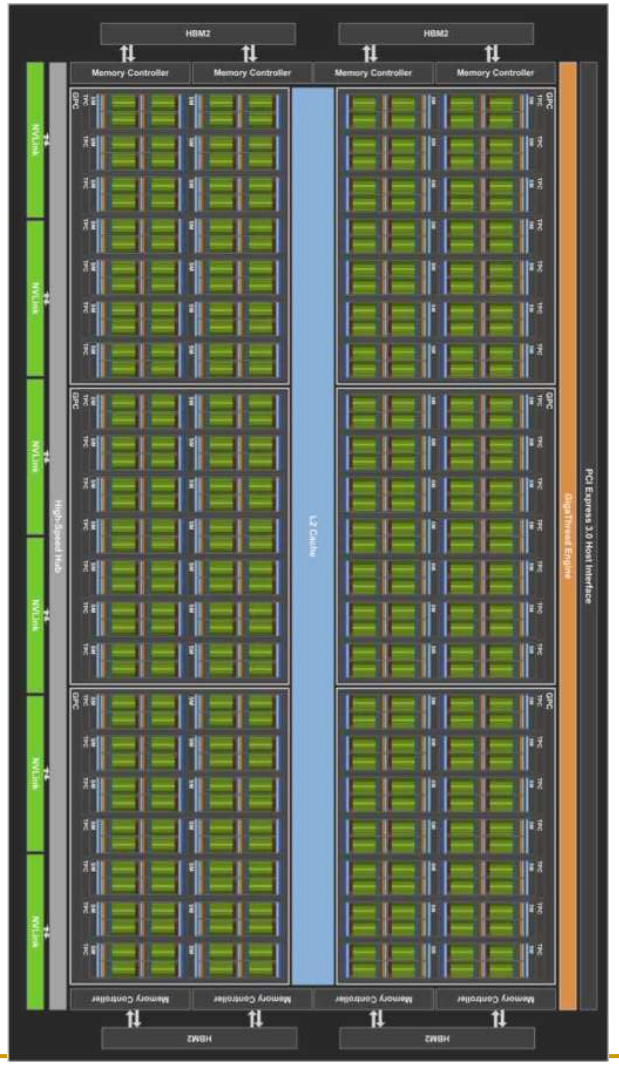
CPU







GPU

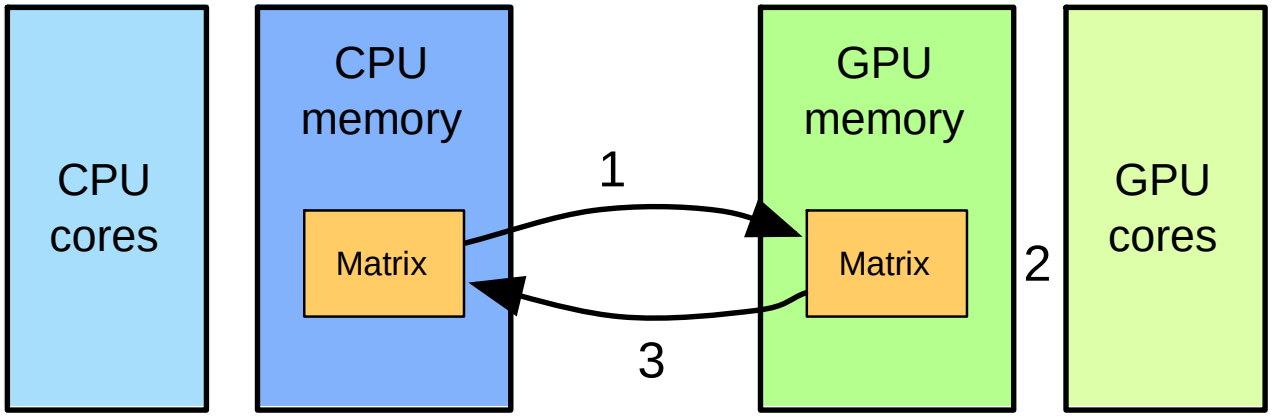


More cores → More trouble

Challenge: How to manipulate them?

GPU Computing

- **Key Idea**:
  - Computation is offloaded to the GPU
- **Three steps**:
  - CPU-GPU data transfer (1)
  - GPU kernel execution (2)
  - GPU-CPU data transfer (3)



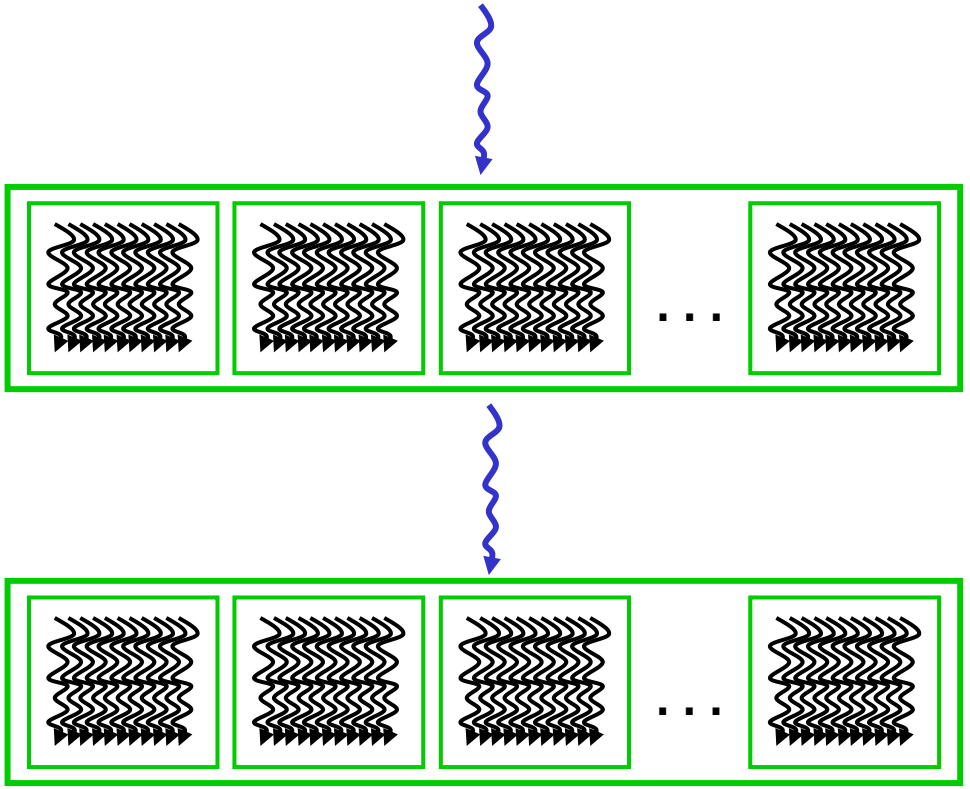
Programming Model: CPU and GPU

- CPU-GPU Co-processing:
  - **CPU**: Sequential or modestly parallel sections
  - **GPU**: Massively parallel sections

Serial Code (CPU):

Parallel Kernel (GPU):

```
KernelA<<<nBlk, nThr>>>(args);
```



Serial Code (CPU):

Parallel Kernel (GPU):

```
KernelB<<<nBlk, nThr>>>(args);
```

GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using threads, NOT SIMD instructions
- To understand this, let's go back to our parallelizable code example
- But, before that, let's distinguish between
  - Programming Model (Software)

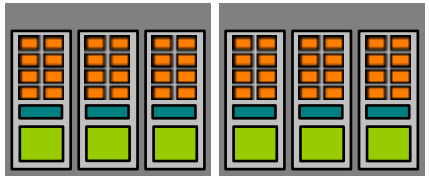
    vs.
  - Execution Model (Hardware)

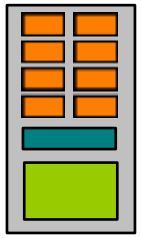
Programming Model vs. Hardware Execution Model

- Programming Model: how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Dataflow, Multi-threaded (MIMD, SPMD), …
- Hardware Execution Model: how the hardware executes the code underneath
  - E.g., Out-of-order execution, Vector processor, Array processor, Dataflow processor, Multiprocessor, Multithreaded processor, …
- **Discussion**: Execution Model can be very different from Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (a GPU)

GPU: Programming Model vs. Hardware Execution Model

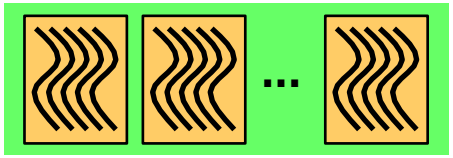
Hardware Execution Model

GPU 

Streaming Multi-processor 

CUDA core 

CUDA Programming Model

Grid 

Thread block 

Thread 

Agenda for Today

- Where is GPU? & Key Message
- Hardware Execution Model
- Programming Model
  - SISD vs. SIMD vs. SPMD
  - GPU Programming Example
- Advance
  - SIMT (Hardware) & Warp (Software)

A Many-core GPU (Hardware Execution Model)

NVIDIA GeForce GTX 285

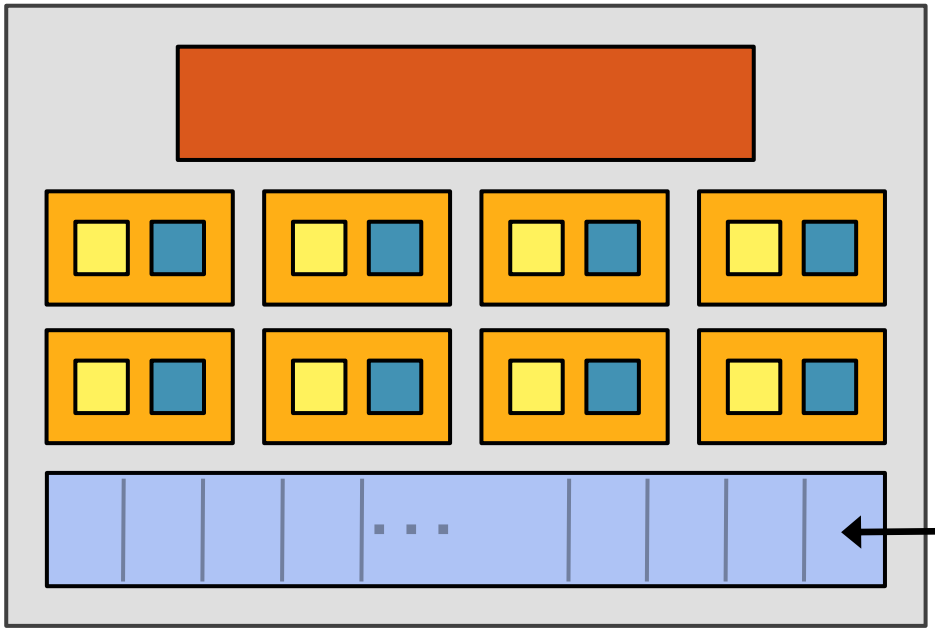
- **NVIDIA-speak**:
  - 240 stream processors (CUDA cores)
  - "SIMT execution"

- **Generic speak**:
  - 30 cores
  - 8 SIMD functional units per core

- NVIDIA, "NVIDIA GeForce GTX 200 GPU. Architectural Overview. White Paper," 2008.

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285 “core” (SM)



64 KB of storage for thread contexts (registers)

= SIMD functional unit, control shared across 8 units

= multiply-add

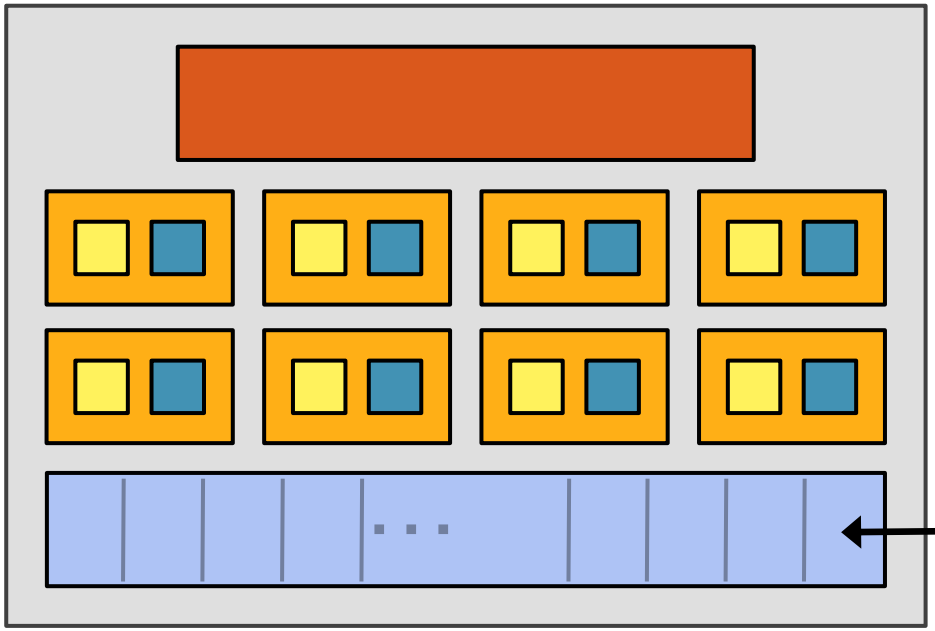
= multiply

= instruction stream decode

= execution context storage

Slide credit: Kayvon Fatahalian

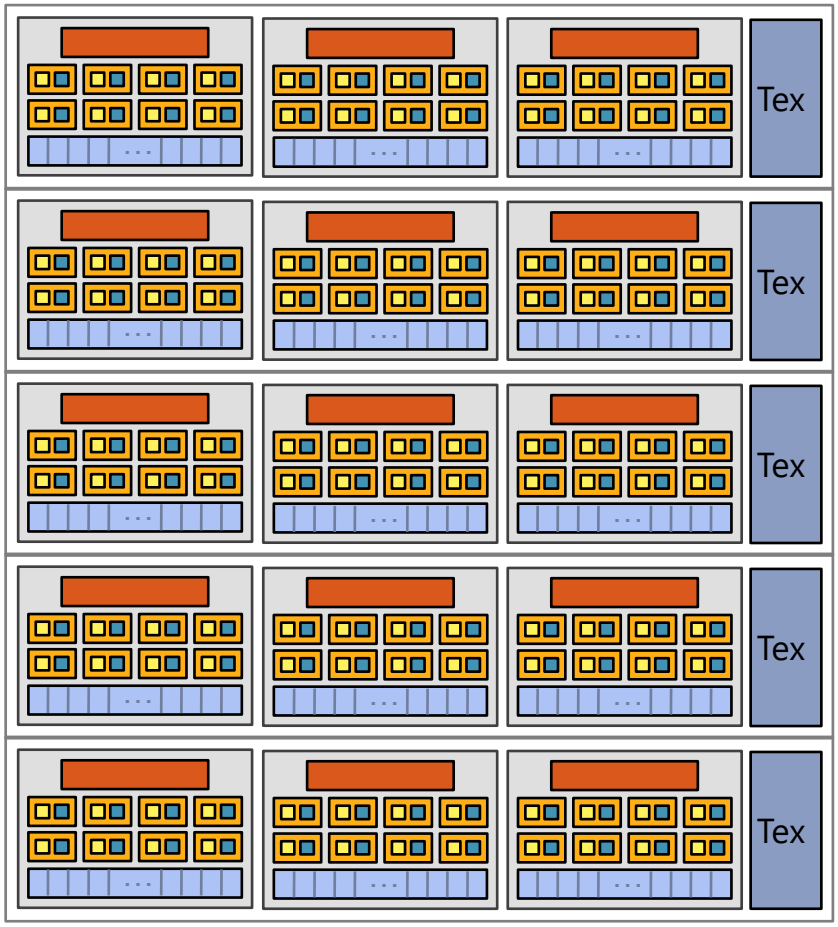
NVIDIA GeForce GTX 285 "core"

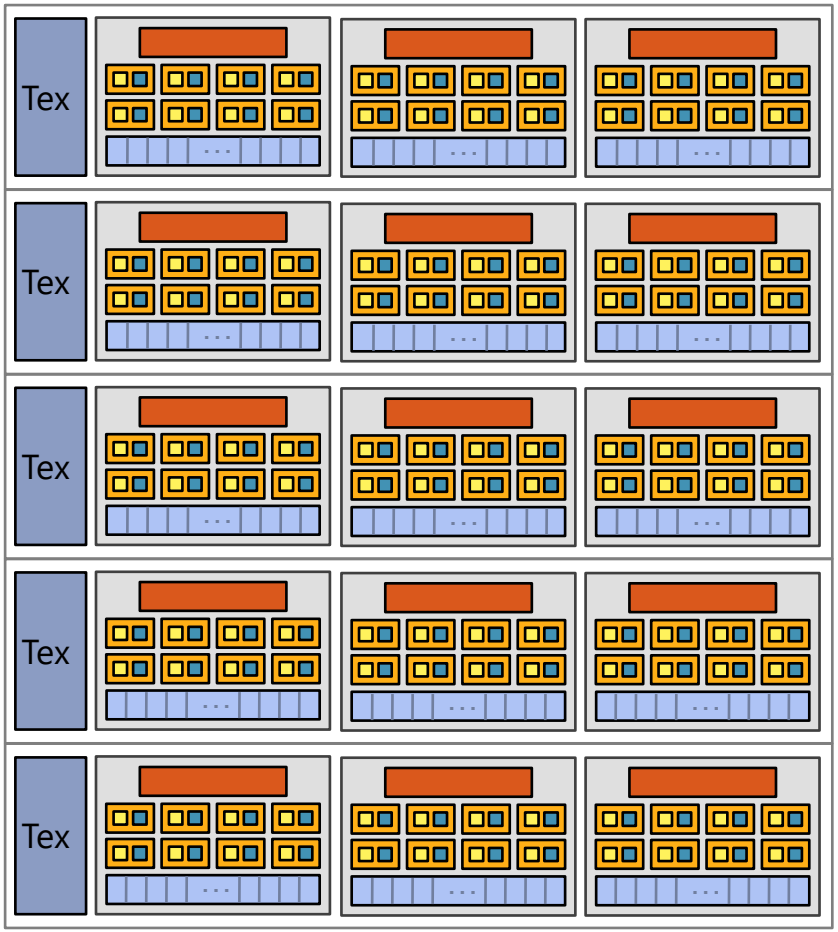


- Groups of 32 threads share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian

NVIDIA GeForce GTX 285

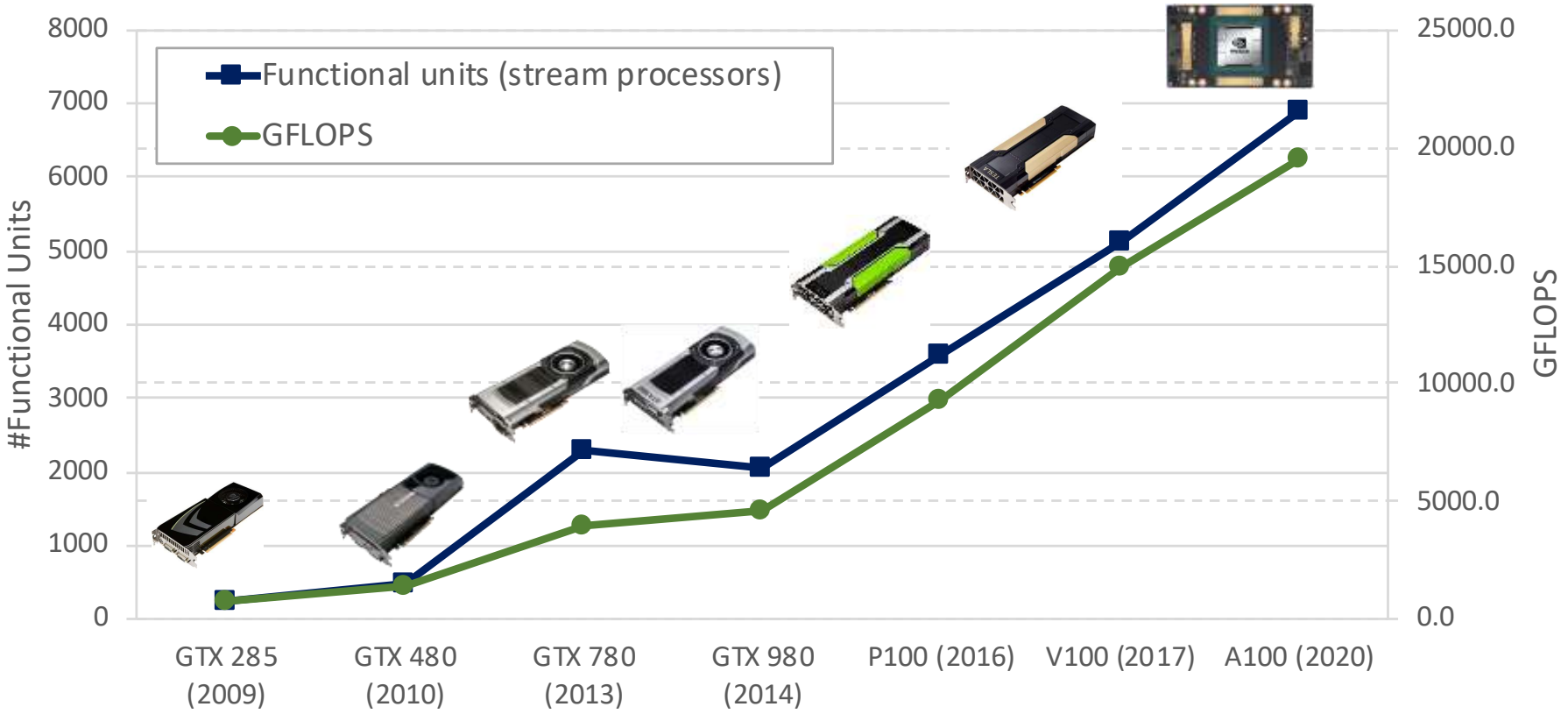




30 cores on the GTX 285: 30K threads

Slide credit: Kayvon Fatahalian

Evolution of NVIDIA GPUs: Compute



NVIDIA V100

- **NVIDIA-speak**:
  - 5120 stream processors (CUDA cores)
  - "SIMT execution"

- **Generic speak**:
  - 80 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning

- NVIDIA, "NVIDIA Tesla V100 GPU Architecture. White Paper," 2017.

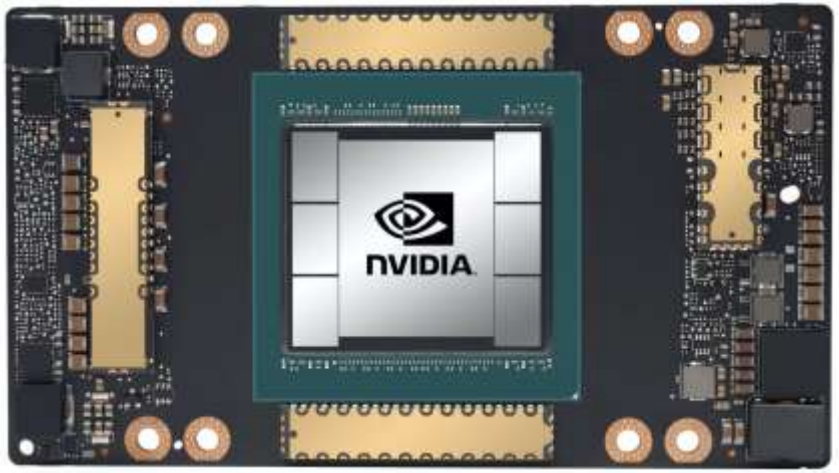
NVIDIA V100 Block Diagram

https://devblogs.nvidia.com/inside-volta/

80 cores on the V100

NVIDIA A100

- **NVIDIA-speak**:
  - 6912 stream processors (CUDA cores)
  - "SIMT execution"



- **Generic speak**:
  - 108 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning
    - Support for sparsity
    - New floating point data type (TF32)

- https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

NVIDIA A100 Block Diagram

https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

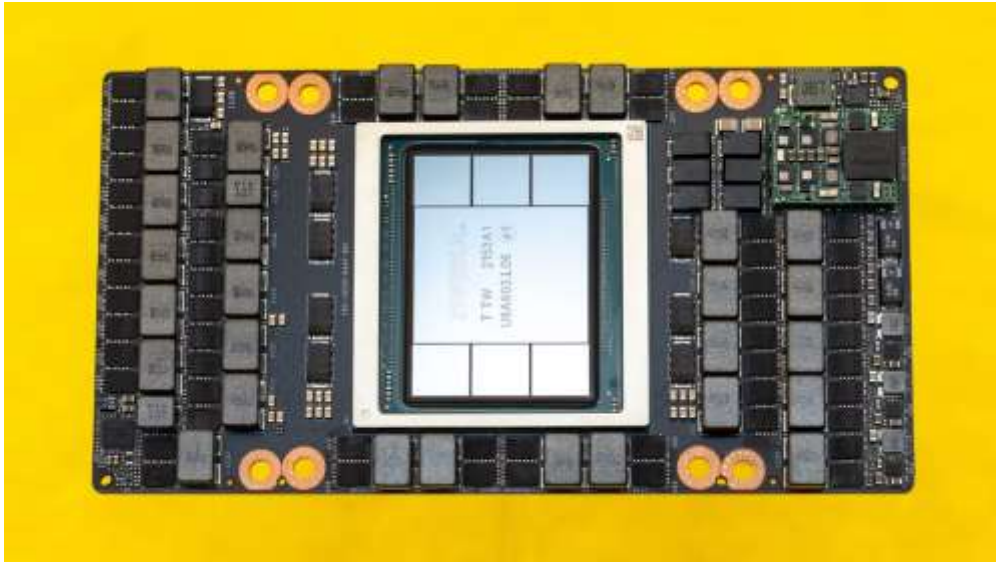
108 cores on the A100

(Up to 128 cores in the full-blown chip)

40MB L2 cache

NVIDIA H100

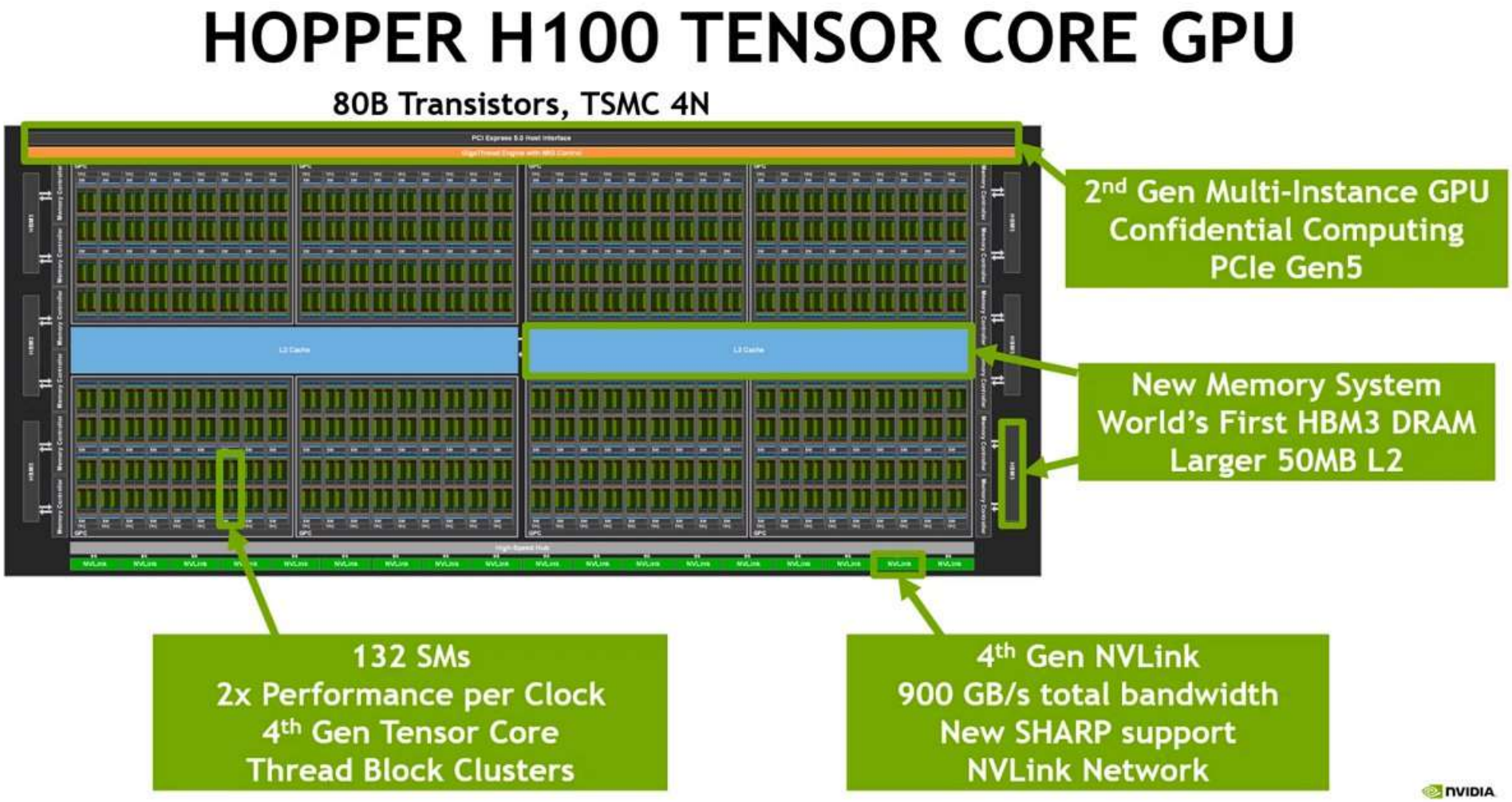
- **NVIDIA-speak**:
  - 8448 stream processors (CUDA cores)
  - "SIMT execution"



- **Generic speak**:
  - 132 cores
  - 64 SIMD functional units per core
  - Tensor cores for Machine Learning
    - Support for sparsity
    - Support for transformer

- https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

NVIDIA H100 Block Diagram



GPU Trend: H100 vs. A100

	FP8	FP16	FP32	FP64	Memory bandwidth	Memory capacity
H100	4000T	2000T	1000T	60T	3TB/s	80GB
A100	666T	666T	333T	20T	2TB/s	80GB

Compute power scales well.

GPU memory capacity does not scale well.

A GPU is a SIMD (SIMT) Machine

- Except it is **not** programmed using SIMD instructions

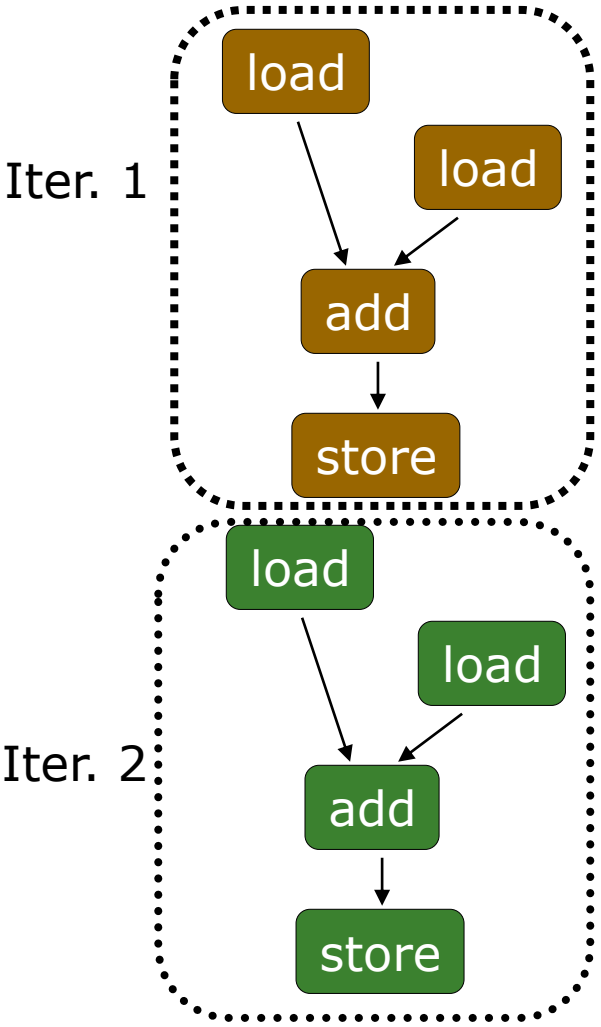
- It is programmed using threads (**SPMD programming model**)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

Agenda for Today

- Where is GPU? & Key Message
- Hardware Execution Model
- Programming Model
  - SISD vs. SIMD vs. SPMD
  - GPU Programming Example
- Advance
  - SIMT (Hardware) & Warp (Software)

How Can You Exploit Parallelism Here?

Scalar Sequential Code




```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Let's examine three programming options to exploit instruction-level parallelism present in this sequential code:

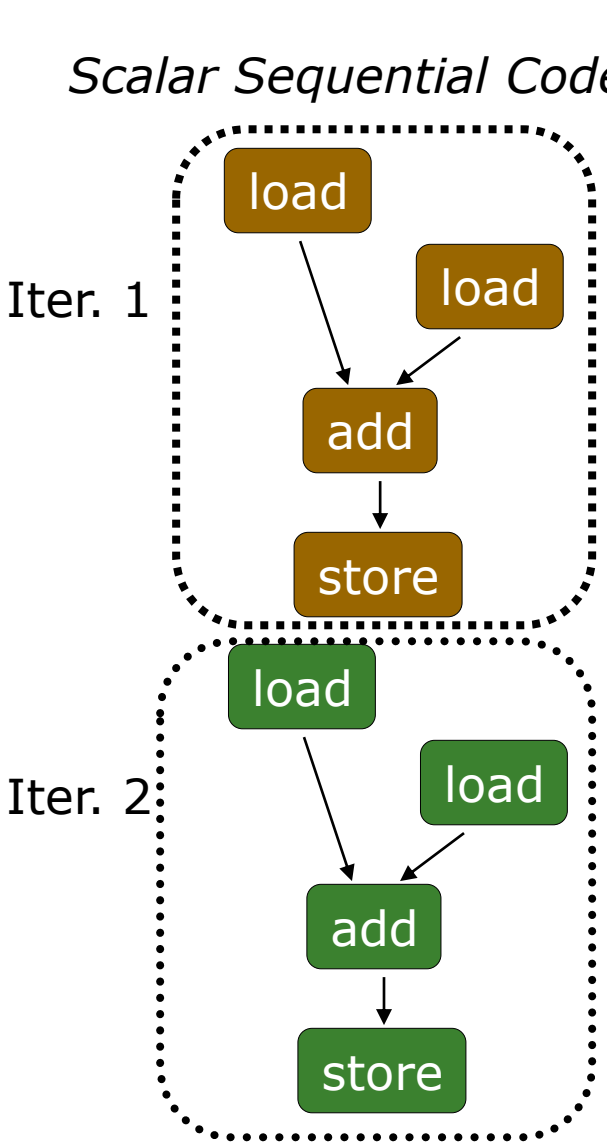
1. Sequential (SISD)

2. Data-Parallel (SIMD)

3. Multithreaded (SPMD)

Prog. Model 1: Sequential (SISD)

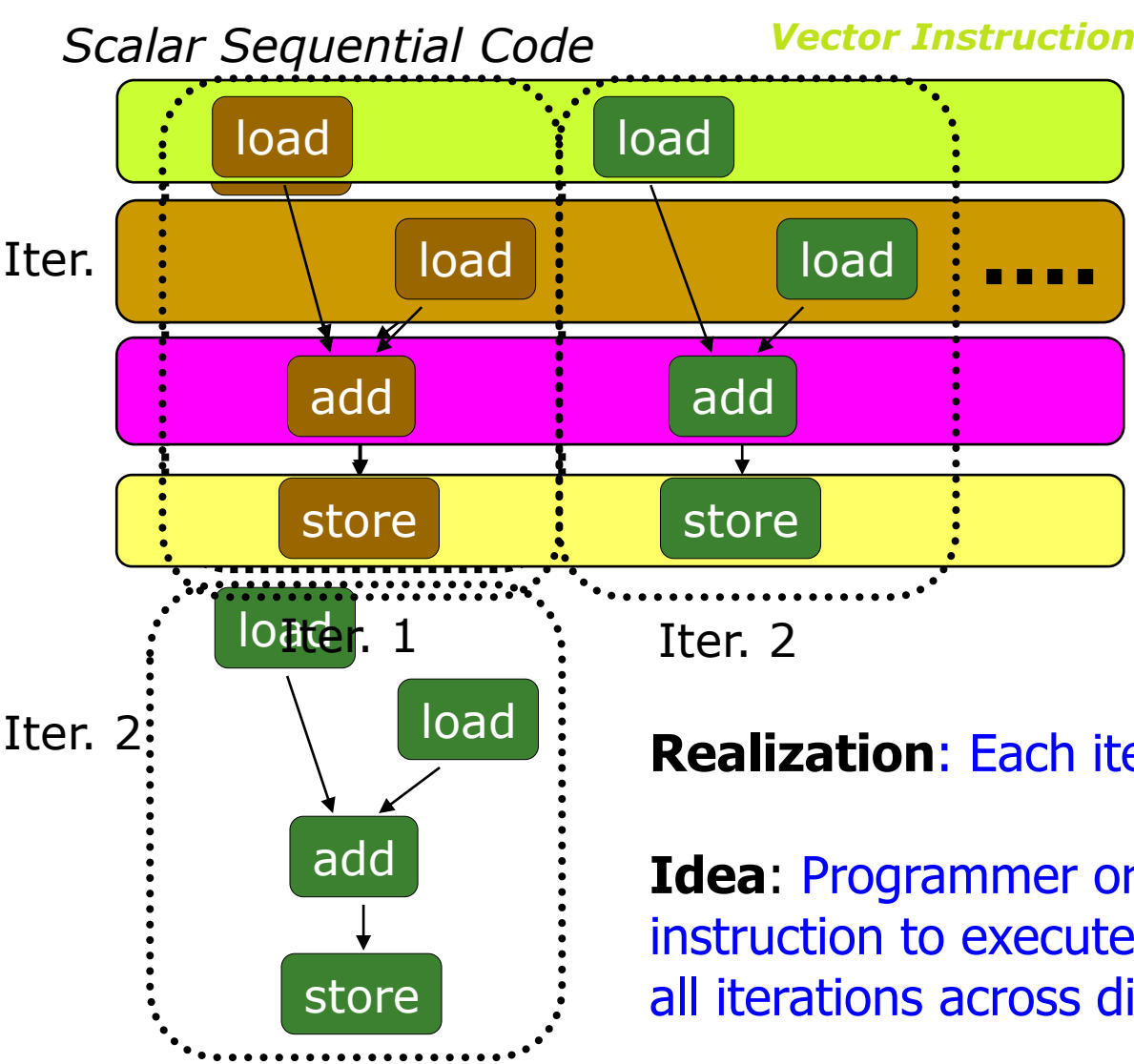
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



- Can be executed on thee processors:
- 1, Pipelined processor
- 2, Out-of-order execution processor
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- 3, Superscalar or VLIW processor
  - Can fetch and execute multiple instructions per cycle

Prog. Model 2: Data Parallel (SIMD)

```
for (i=0; i < N; i++)
  C[i] = A[i] + B[i];
```



Vectorized Code

VLD A → V1

VLD B → V2

VADD V1 + V2 → V3

VST V3 → C

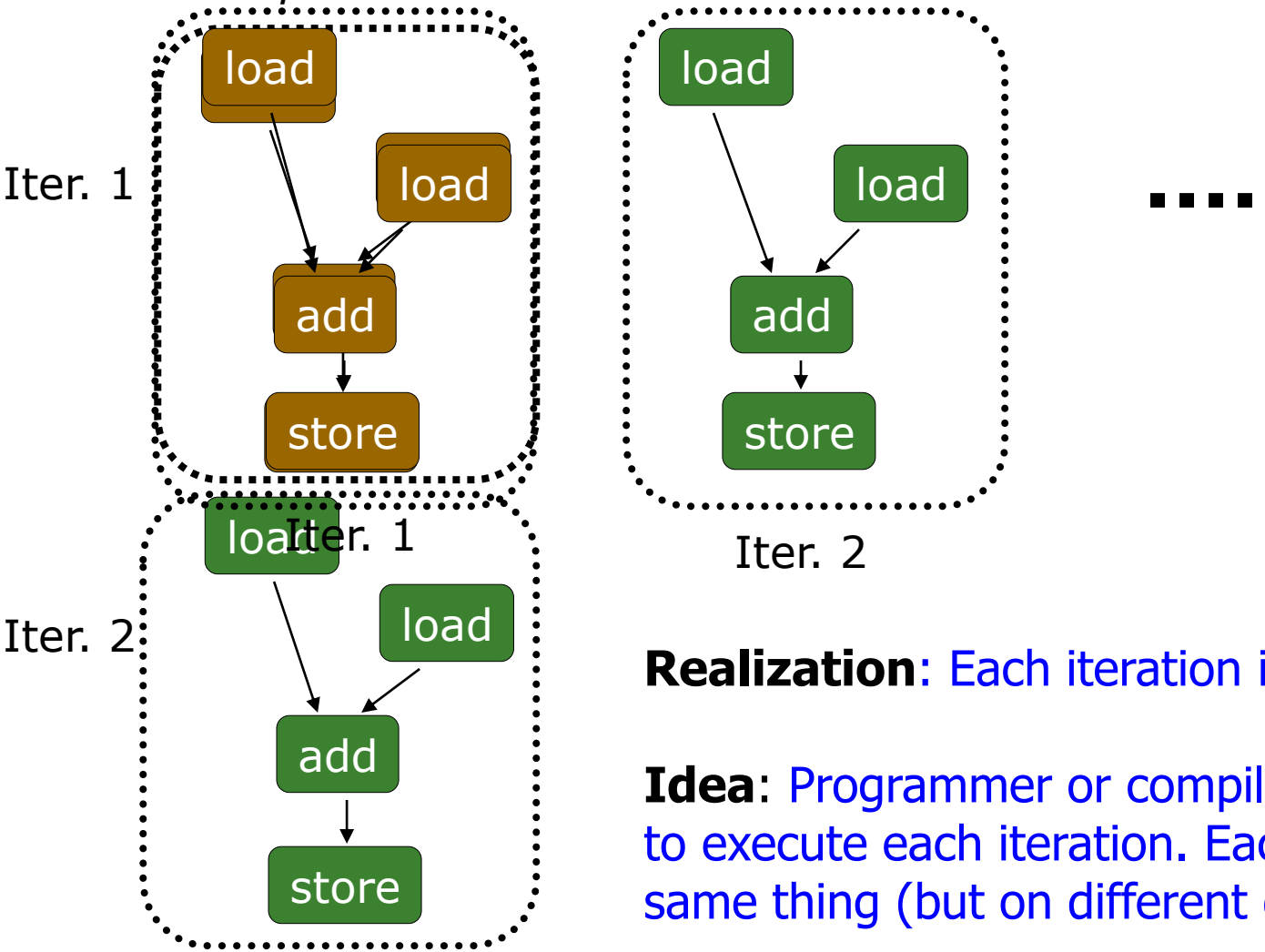
Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code

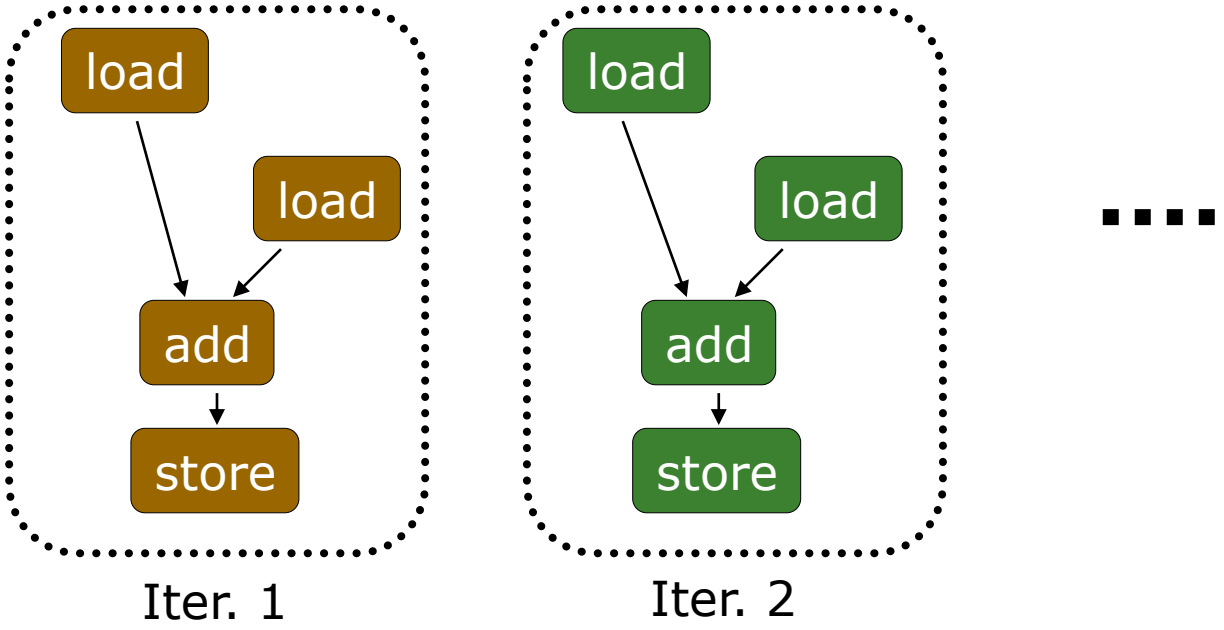


Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Realization: Each iteration is independent

This programming model (software) is called:

SPMD: Single Program Multiple Data

SPMD

- **SPMD**: Single procedure/program, multiple data
  - This is a **programming model** rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- **Key Idea of SPMD**: multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

Agenda for Today

- Where is GPU? & Key Message
- Hardware Execution Model
- Programming Model
  - SISD vs. SIMD vs. SPMD
  - GPU Programming Example
- Advance
  - SIMT (Hardware) & Warp (Software)

CUDA/OpenCL Programming Model

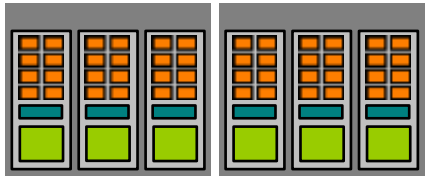
- Single Program Multiple Data (SPMD), e.g., CUDA
  - Bulk synchronous programming: Global (coarse-grain) synchronization between kernels

- The device (typically GPU) executes CUDA kernels
  - Grid
  - Thread Block
    - CUDA runtime schedules at granularity of thread block.
    - A thread block is a programming abstraction that represents a group of threads that can be executed in parallel.
    - Within a block, shared memory, and synchronization.
  - Thread
    - A thread corresponds to an iteration.

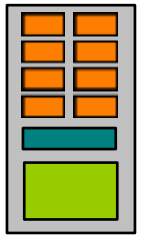
GPU: Programming Model vs. Hardware Execution Model

Hardware Execution Model

GPU



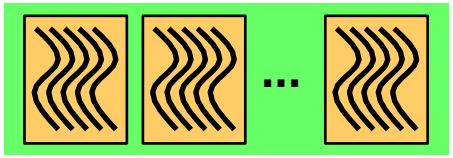
Streaming Multi-processor



CUDA core

CUDA Programming Model

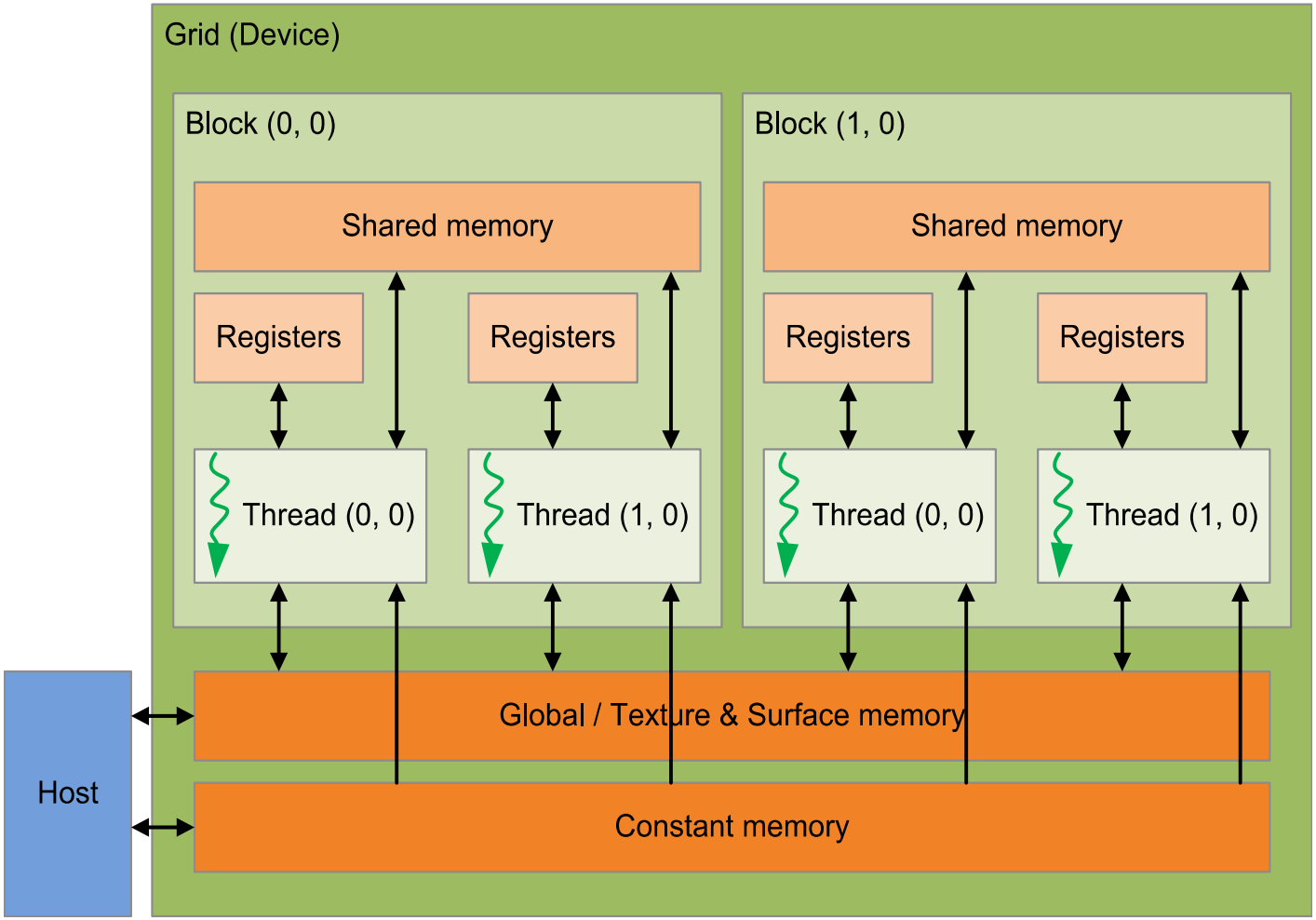
Grid



Thread block

Thread

CUDA: Memory Hierarchy



Traditional Program Structure in CUDA

- Function prototypes

```
float serialFunction(…);
__global__ void kernel(…);
```

- `main()`
  - 1) Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  - 2) Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  - 3) Execution configuration setup: #blocks and #threads
  - 4) Kernel call – `kernel<<<execution configuration>>>(args…);`
  - 5) Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`

- Kernel – `__global__ void kernel(type args,…)`
  - Automatic variables transparently assigned to registers
  - Shared memory: `__shared__`
  - Intra-block synchronization: `__syncthreads();`

Slide credit: Hwu & Kirk

CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

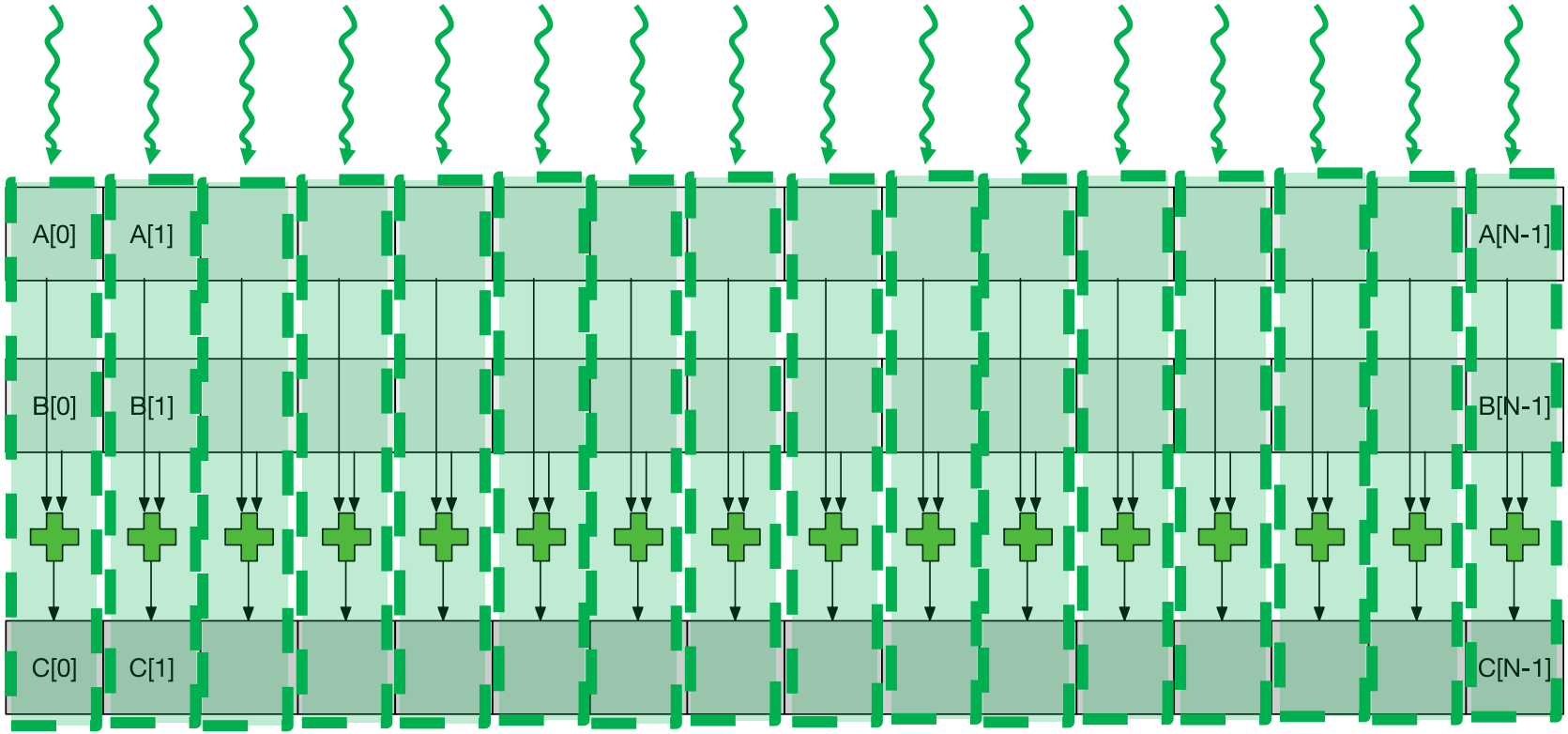
```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

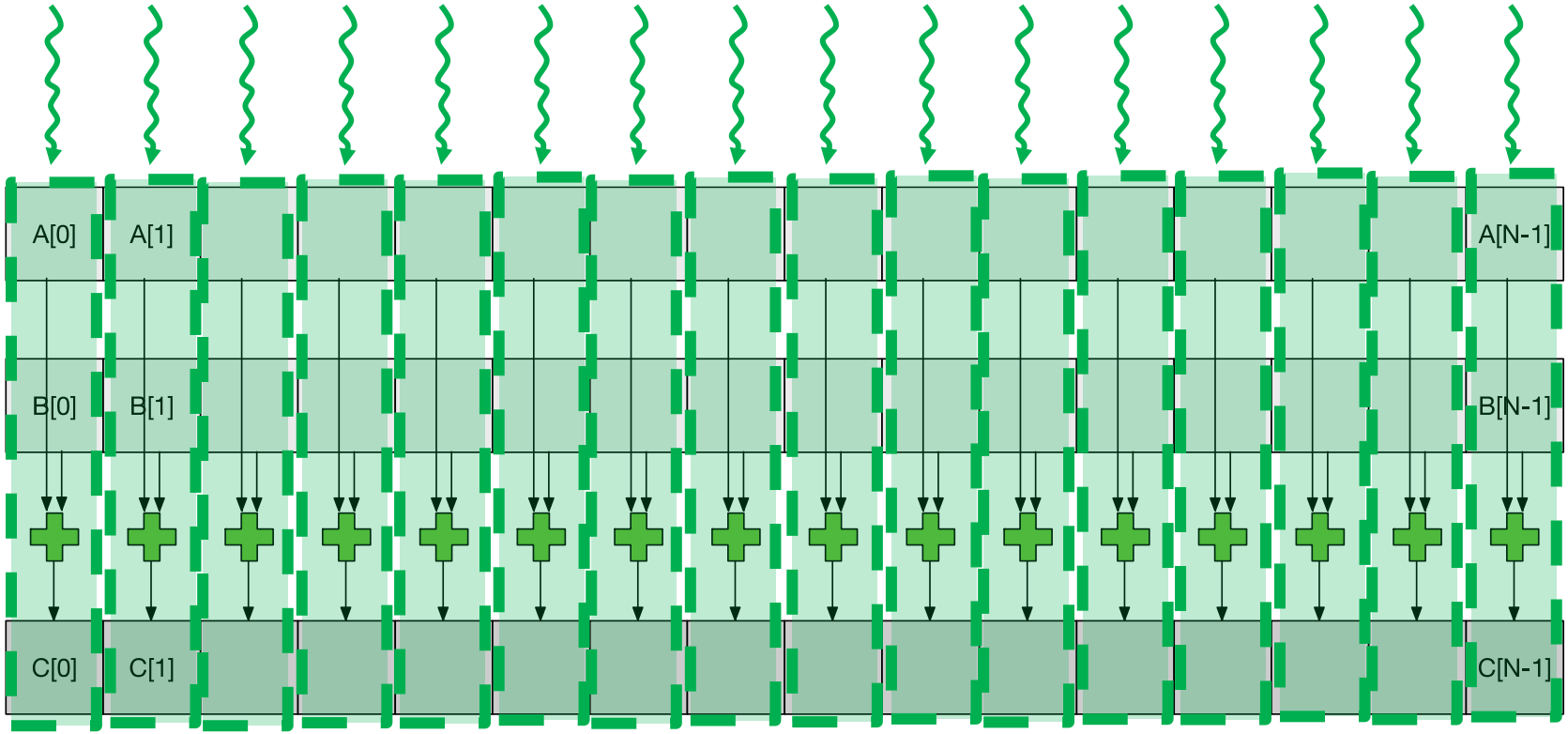
First GPU Example: Vector Addition (I)

- **Key Idea**: one GPU thread to each element-wise addition



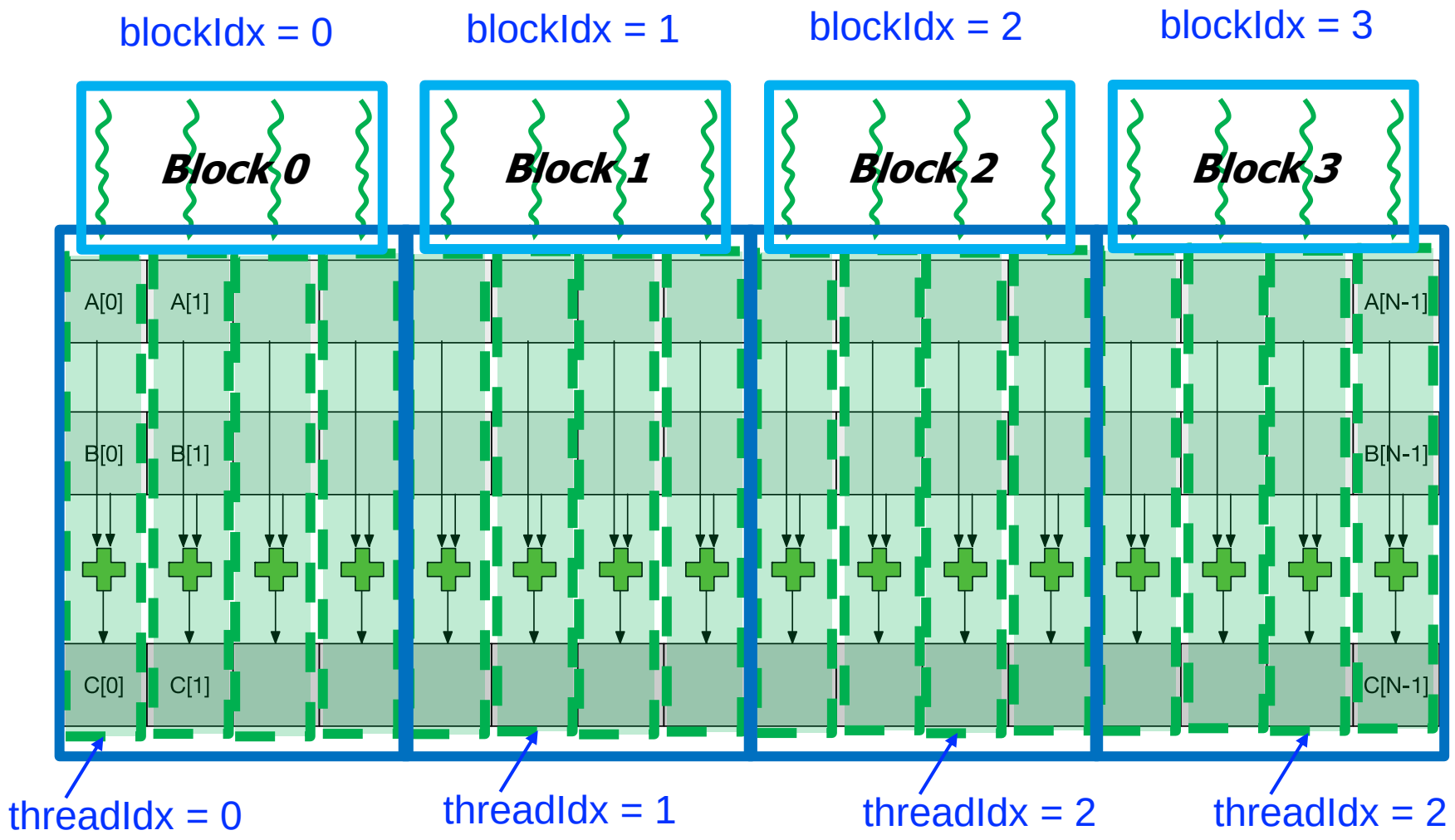
First GPU Example: Vector Addition (II)

- A grid: the whole set of threads
- We need a way to assign threads to GPU cores



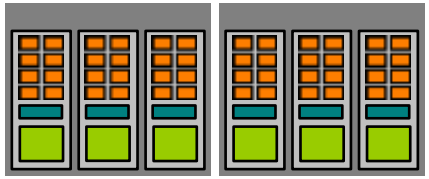
First GPU Example: Vector Addition (III)

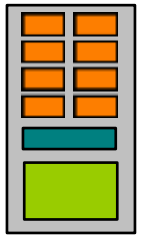
- We group threads into blocks



GPU: Programming Model vs. Hardware Execution Model

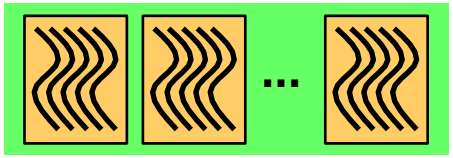
Hardware Execution Model

GPU 

Streaming Multi-processor 

CUDA core 

CUDA Programming Model

Grid 

Thread block 

Thread 

Host Code Example: Vector Addition

```
void vecadd(float* A, float* B, float* C, int N) {

    //1, Allocate GPU memory
    float *A_d, *B_d, *C_d;
    cudaMalloc((void**) &A_d, N*sizeof(float));
    cudaMalloc((void**) &B_d, N*sizeof(float));
    cudaMalloc((void**) &C_d, N*sizeof(float));

    //2, Copy data to GPU memory
    cudaMemcpy(A_d, A, N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, N*sizeof(float), cudaMemcpyHostToDevice);

    //3, Perform computation on GPU
    const unsigned int numThreadsPerBlock = 512;
    const unsigned int numBlocks = N/numThreadsPerBlock;
    vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);

    //4, Copy data from GPU memory
    cudaMemcpy(C, C_d, N*sizeof(float), cudaMemcpyDeviceToHost);

    //5, Deallocate GPU memory
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```

Slide credit: Izzat El Hajj

Kernel Code Example: Vector Addition

```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {
    int i = blockDim.x*blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

`blockDim`: block dimension

`blockIdx`: block index within a grid

`threadIdx`: thread index within a block

Slide credit: Izzat El Hajj

Boundary Conditions

- **Question**: What if the size of the input is not a multiple of the number of threads per block?
  - **Solution**: use the ceiling to launch extra threads then omit the threads after the boundary
- **Host code**:

```
const unsigned int numBlocks = (N +numThreadsPerBlock – 1)/numThreadsPerBlock;

vecadd_kernel<<<numBlocks, numThreadsPerBlock>>>(A_d, B_d, C_d, N);
```

- **Kernel code**:

```
__global__ void vecadd_kernel(float* A, float* B, float* C, int N) {

    int i = blockDim.x*blockIdx.x + threadIdx.x;

    if(i < N) {
        C[i] = A[i] + B[i];
    }
}
```

Sample GPU Program: Matrix Multiplication

CPU Program

```
void add matrix
 ( float *a, float* b, float *c, int N) {
   int index;
   for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
        index = i + j*N;
        c[index] = a[index] + b[index];
       }
 }

int main () {

  add matrix (a, b, c, N);
}
```

GPU Program

```
__global__  add_matrix
  ( float *a, float *b, float *c, int N) {
int i = blockIdx.x *  blockDim.x + threadIdx.x;
Int j = blockIdx.y * blockDim.y  + threadIdx.y;
int index = i + j*N;
 if (i < N && j < N)
   c[index] = a[index]+b[index];
}

Int main() {
  dim3 dimBlock( blocksize, blocksize) ;
  dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```

Slide credit: Hyesoon Kim

Indexing and Memory Access

- Images are 2D data structures
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

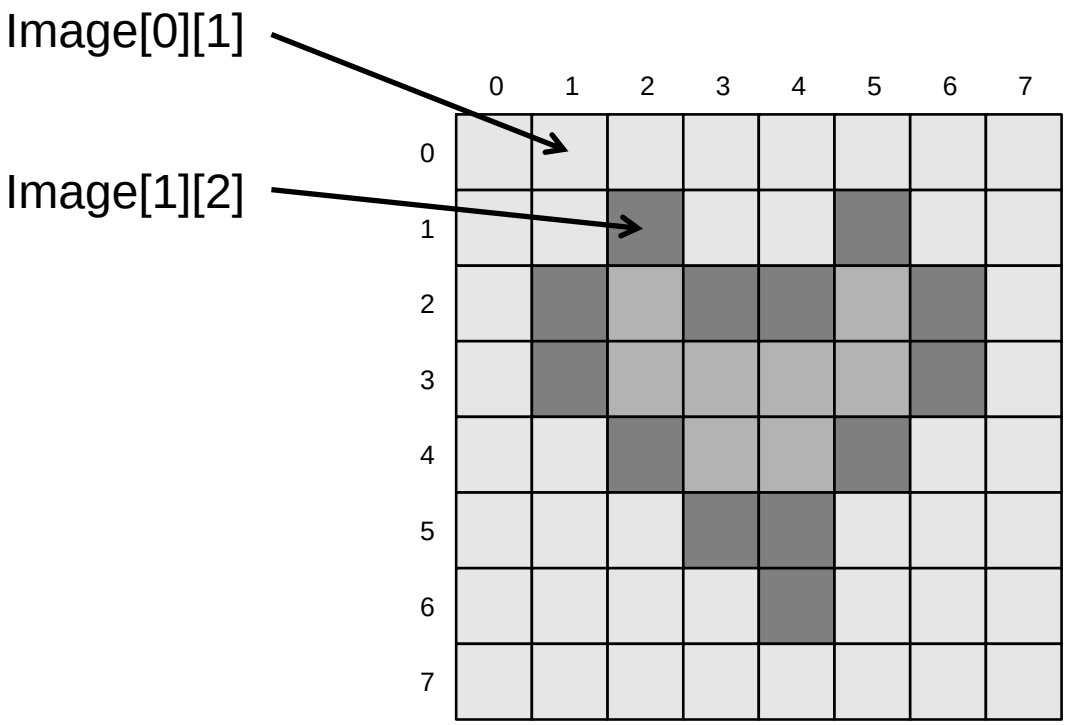
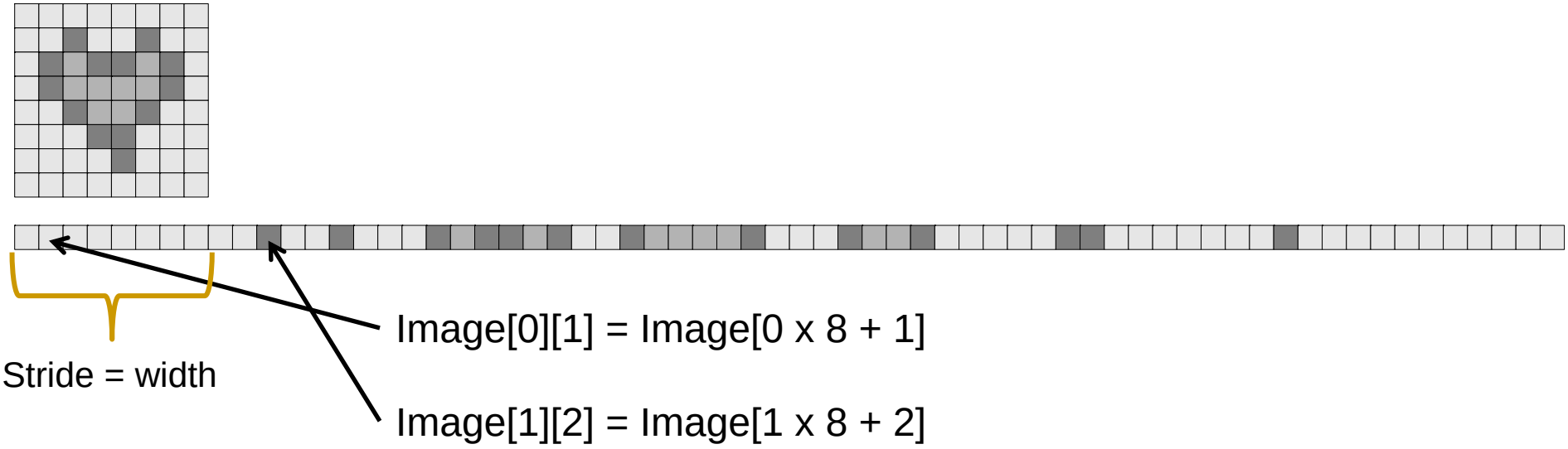


Image Layout in Memory

- Row-major layout
- Image[j][i] = Image[j x width + i]

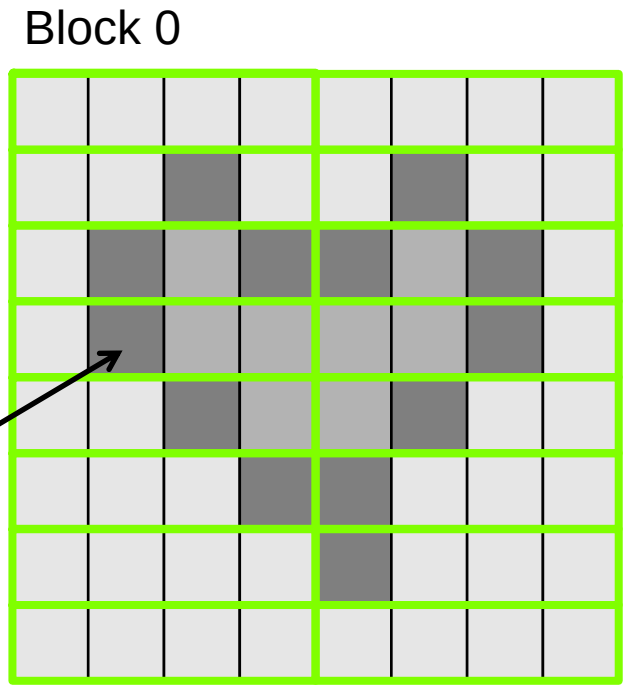


Indexing and Memory Access: 1D Grid

- One GPU thread per pixel
- Grid of Blocks of Threads
  - `gridDim.x, blockDim.x`
  - `blockIdx.x, threadIdx.x`

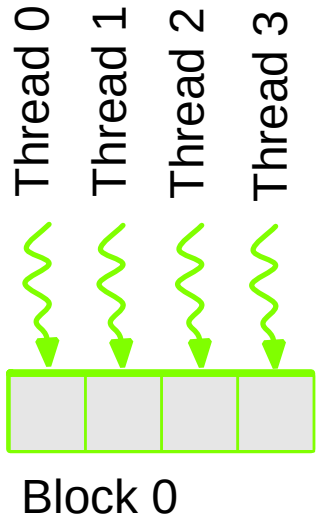
`blockIdx.x`

`threadIdx.x`



6 * 4 + 1 = 25

`blockIdx.x * blockDim.x + threadIdx.x`



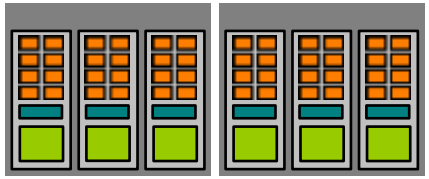
Agenda for Today

- Where is GPU? & Key Message
- Hardware Execution Model
- Programming Model
  - SISD vs. SIMD vs. SPMD
  - GPU Programming Example
- Advance
  - SIMT (Hardware) & Warp (Software)

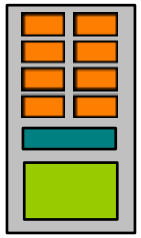
GPU: Programming Model vs. Hardware Execution Model

Hardware Execution Model

GPU



Streaming Multi-processor

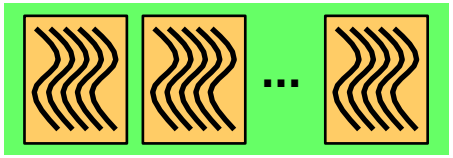


SIMT

CUDA core

CUDA Programming Model

Grid



Thread block

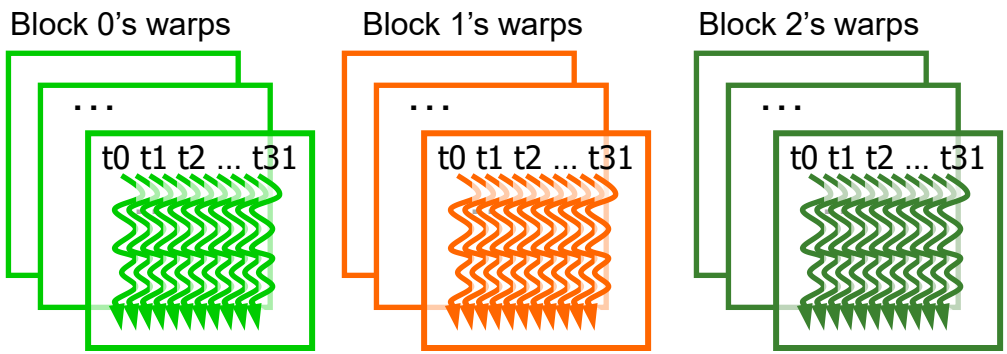
Wrap

Thread

SIMT (Hardware) & Warp (Software)

- **SIMT**: **S**ingle **I**nstruction **M**ultiple **T**hread
  - More precisely, SIMD (Single Instruction Multiple Data)
  - Key Feature: 16 CUDA cores in a SM are executed in a lock step.

- **Warp:**
  - A warp, a basic execution unit, consists of 32 consecutive threads
  - A thread block is divided into warps for SIMT execution.

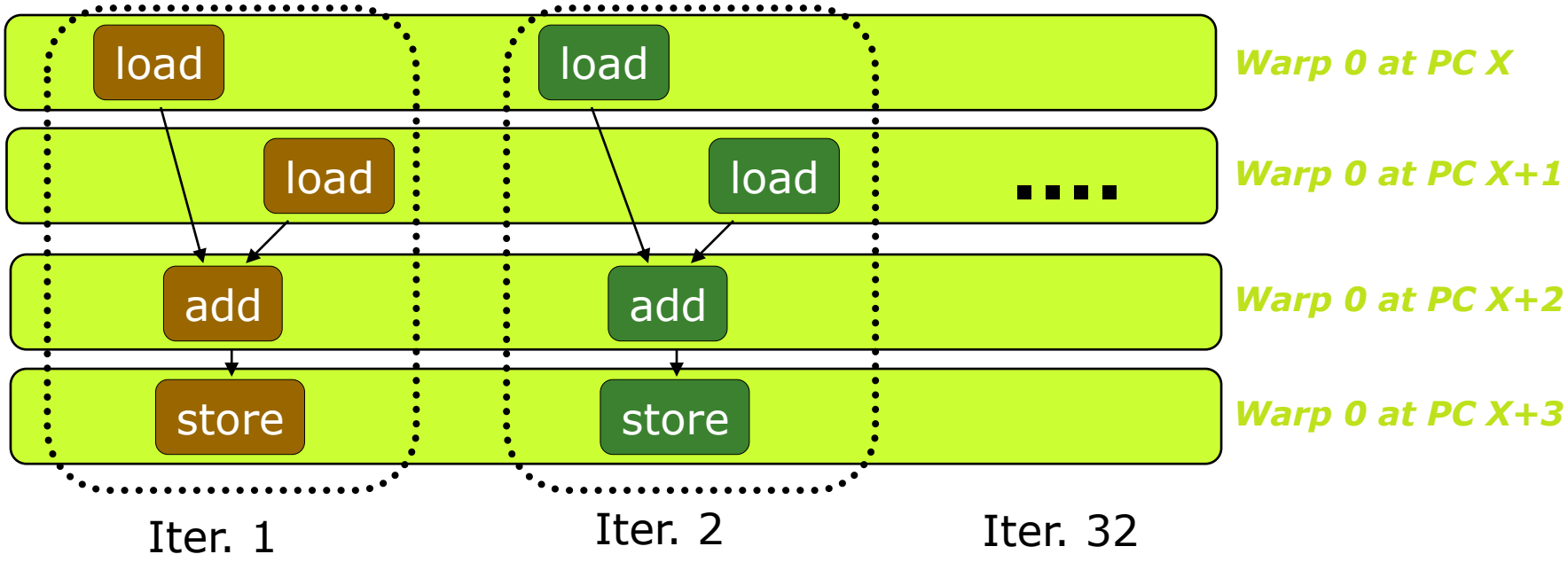


Why SIMT and Warp?

Reduce GPU scheduling overhead

How to Form Warps?

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

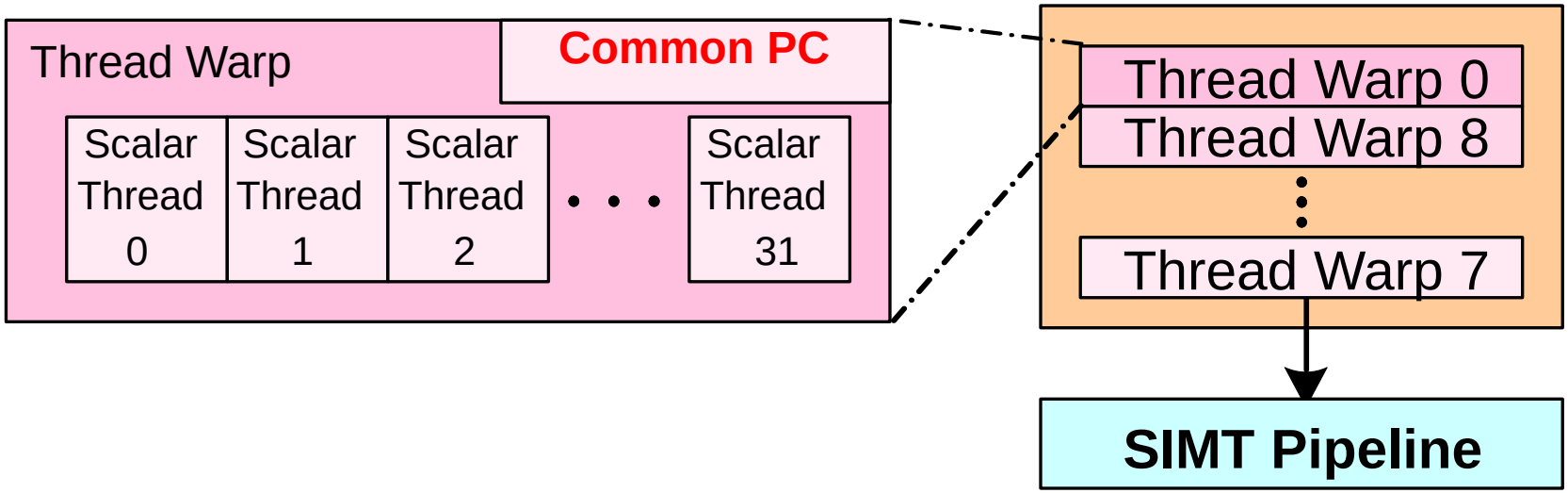


Warp: A set of threads that execute the same instruction (i.e., at the same PC)

Mapping Warps on a SIMT Hardware

- **Warp**:
  - A thread block is divided into warps.
  - A warp executes the same instruction on different data elements

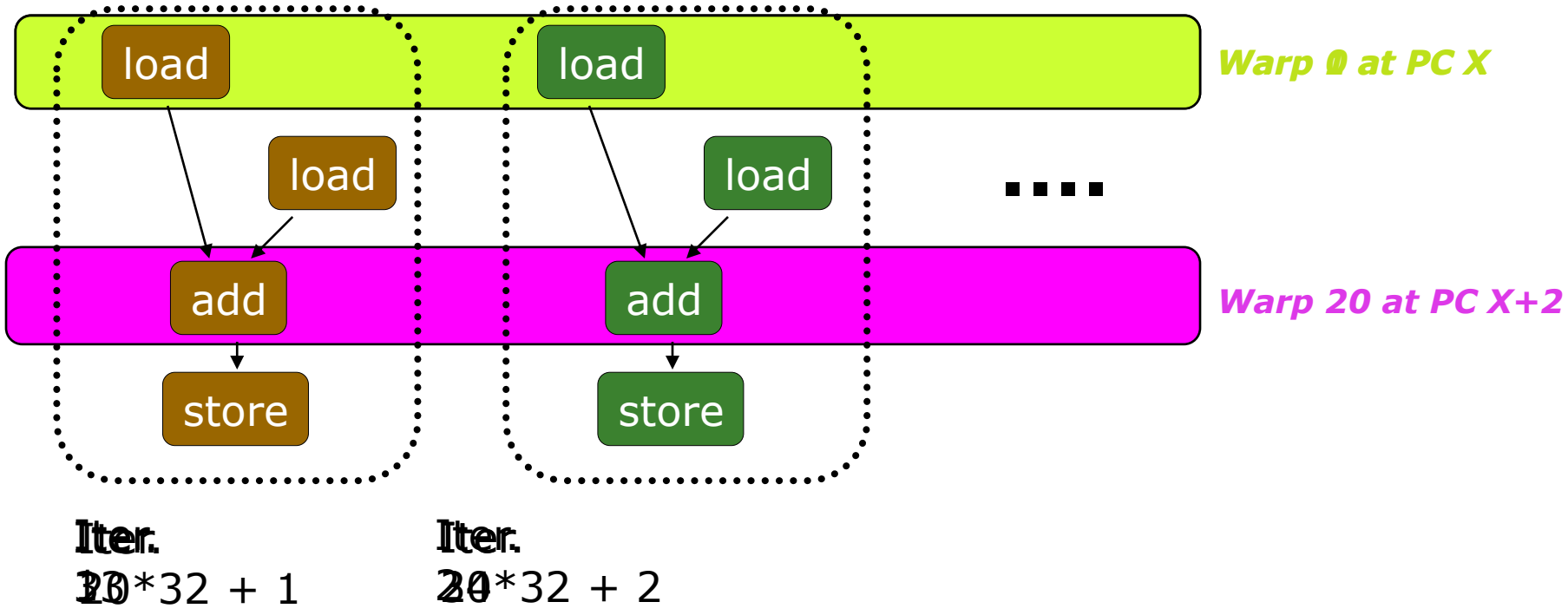
- **SIMT Pipeline**:
  - 16 CUDA cores are executed in a lock step to serve each warp.



GPU Execution with Warps

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

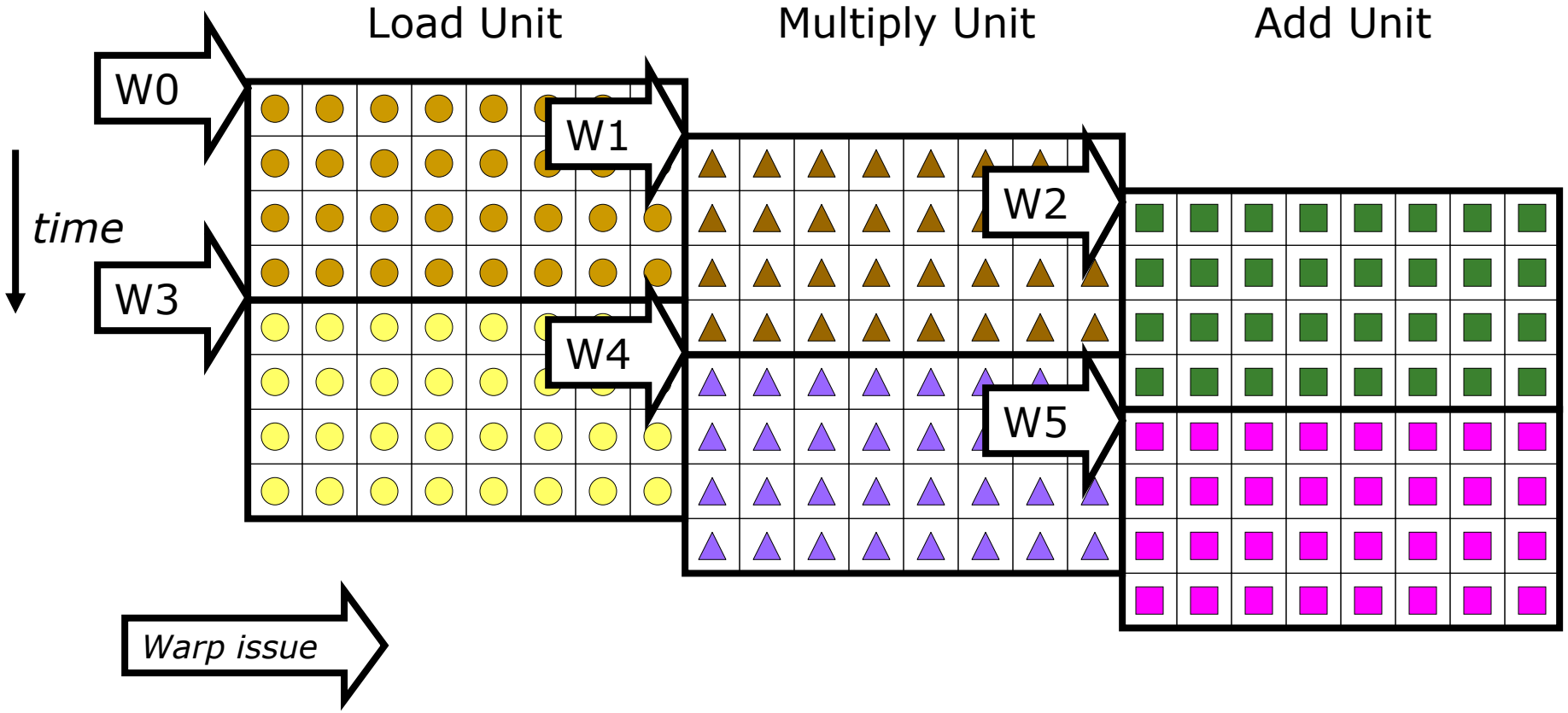
- Assume: a warp consists of 32 threads
- If you have 32K iterations, and 1 iteration/thread → 1K warps
- Warps can be interleaved on the same pipeline → Fine grained multithreading of warps.



Warp Instruction Level Parallelism

Can overlap execution of multiple instructions

- Example machine has 32 threads per warp and 8 lanes
- Completes 24 operations/cycle while issuing 1 warp/cycle



Slide credit: Krste Asanovic

SIMT is not SIMD!

SIMD vs. SIMT Execution Model

- **SIMD**: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

SIMT Code vs. SIMD Code

CPU scalar code

```
for (ii = 0; ii < 100000; ++ii) {
C[ii] = A[ii] + B[ii];
}
```

```
// there are 100000 threads
__global__ void KernelFunction(…) {
  int tid = blockDim.x * blockIdx.x + threadIdx.x;
  int varA = aa[tid];
  int varB = bb[tid];
  C[tid] = varA + varB;
}
```

CUDA code

```
// there are 25000 loops with SIMD=4
…
v_A = vec_load (A);
v_B = vec_load (B);
 v_C = vec_add(v_A, v_B);
Vec_store(v_C, C)
…
}
```

CPU vector code

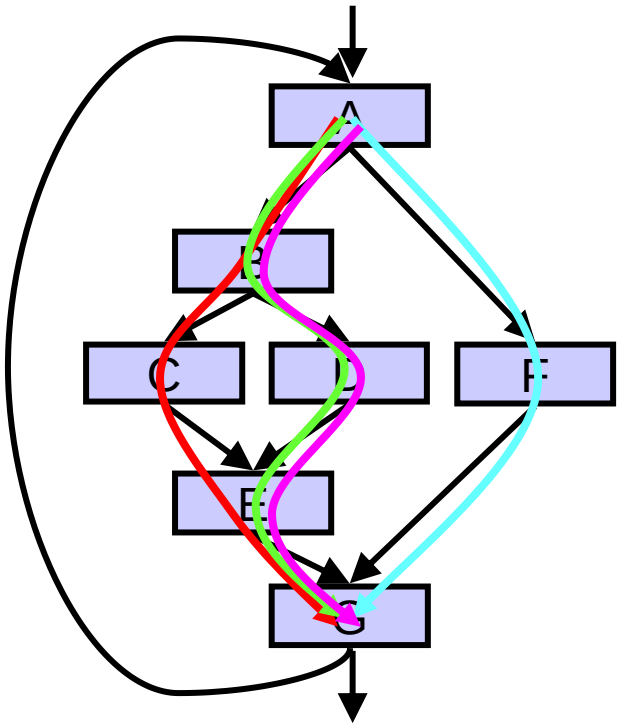
Warp-based SIMD vs. Traditional SIMD

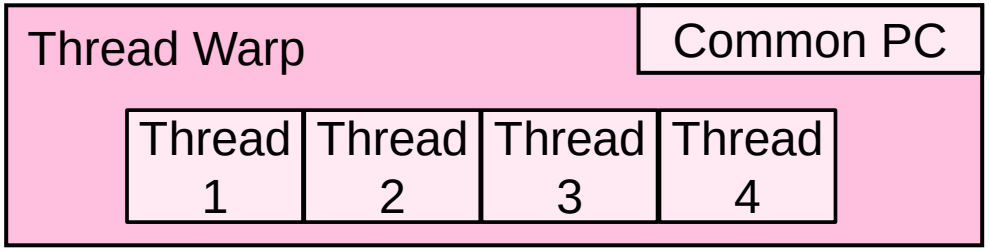
- Traditional SIMD contains a single thread
  - Sequential instruction execution; lock-step operations in a SIMD instruction
  - Programming model is SIMD (no extra threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables multithreading and flexible dynamic grouping of threads
  - ISA is scalar → SIMD operations can be formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware

Threads Can Take Different Paths in Warp-based SIMD

- Each thread can have conditional control flow instructions
- Threads can execute different control flow paths



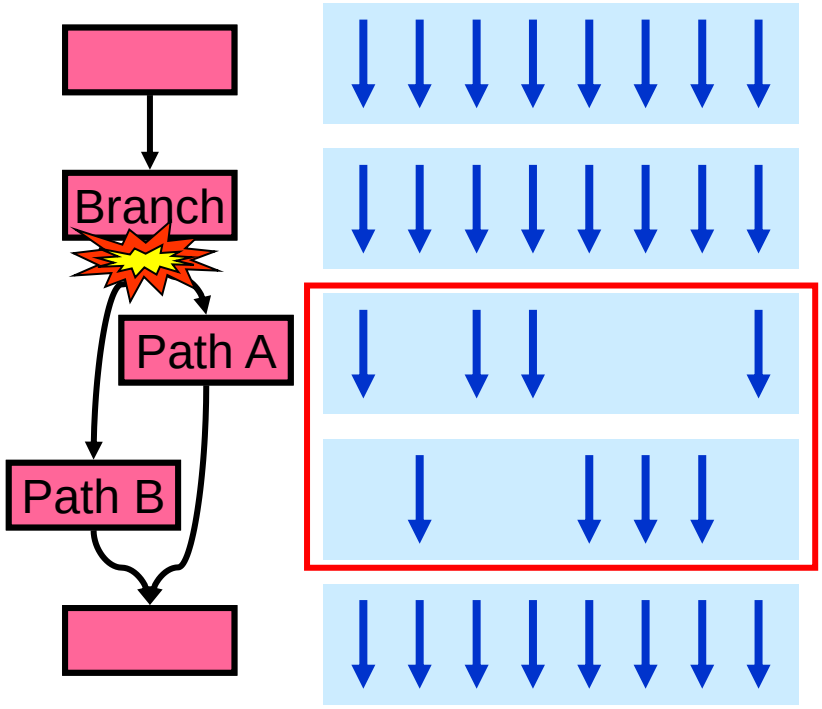


Slide credit: Tor Aamodt

Control Flow Problem in GPUs/SIMT

- A GPU uses a SIMD pipeline to save area on control logic
  - Groups scalar threads into warps

- **Branch divergence** occurs when threads inside warps branch to different execution paths

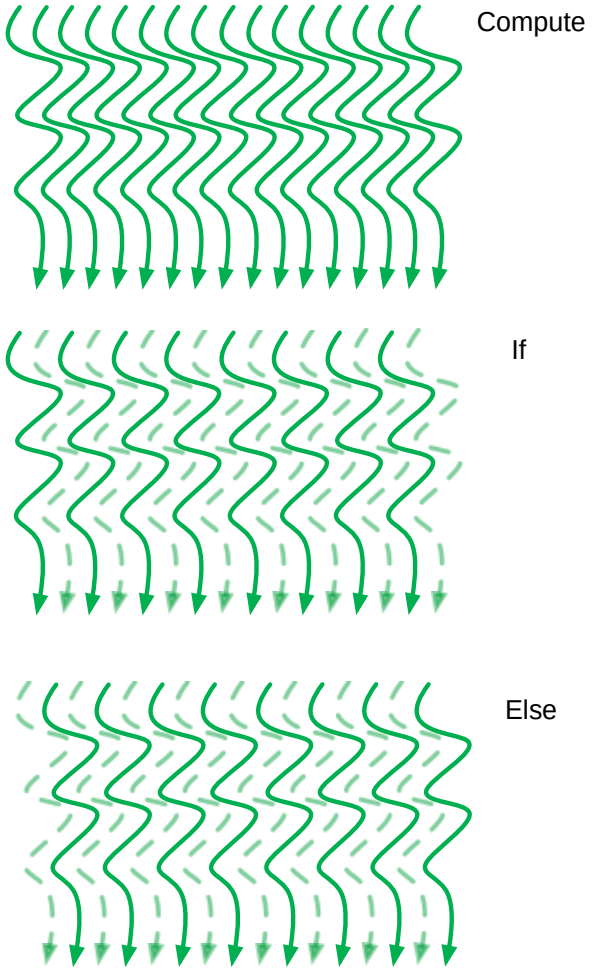


Slide credit: Tor Aamodt

SIMD Utilization

- Intra-warp divergence

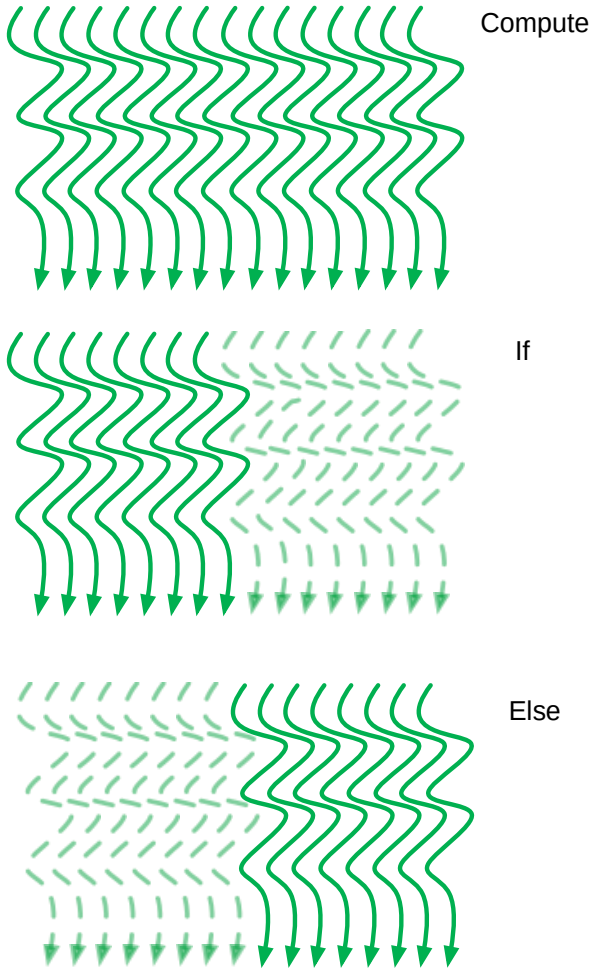
```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0){
  Do_this(threadIdx.x);
}
else{
  Do_that(threadIdx.x);
}
```



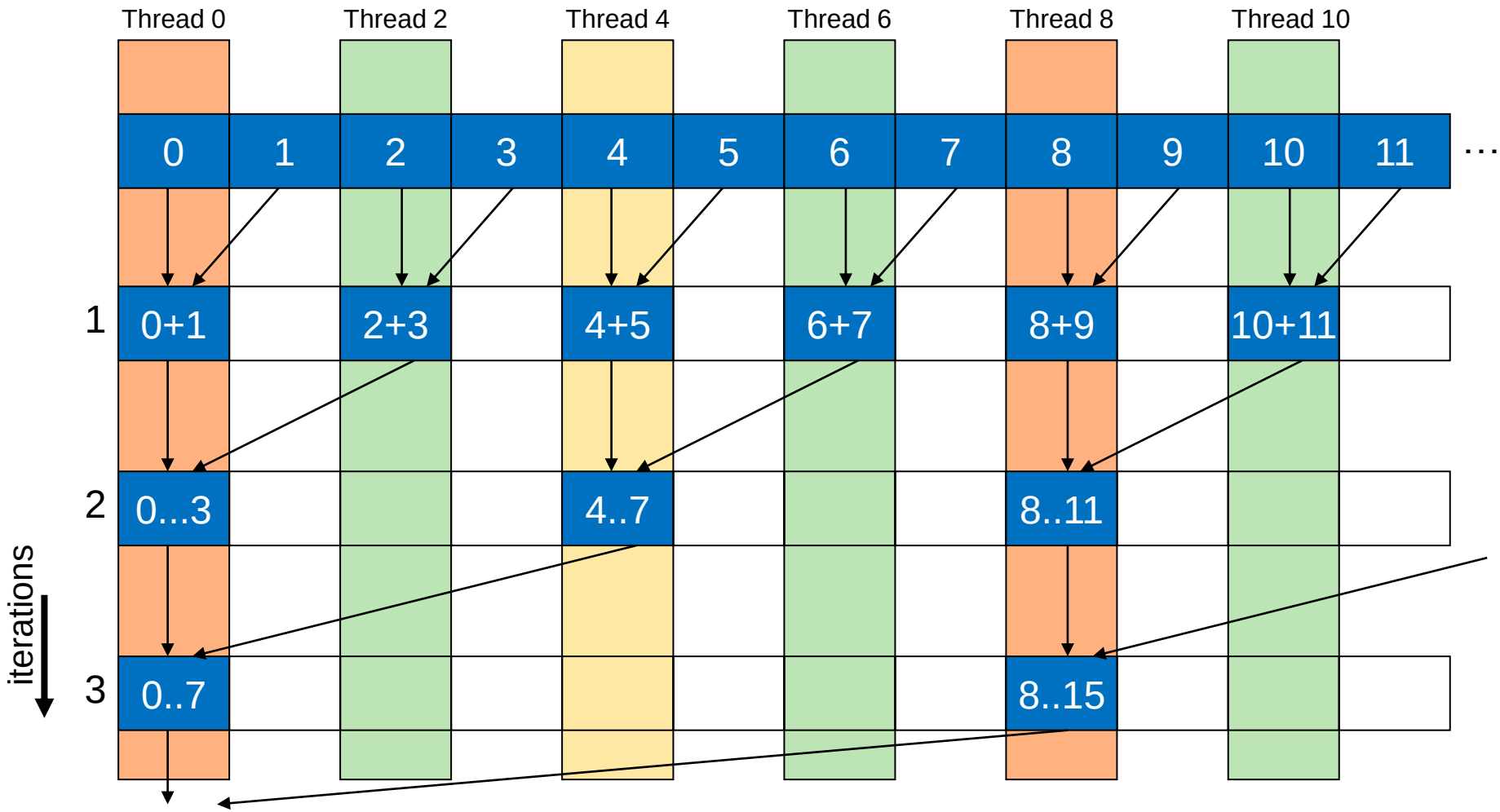
Increasing SIMD Utilization

- Divergence-free execution

```
Compute(threadIdx.x);
if (threadIdx.x < 32){
  Do_this(threadIdx.x * 2);
}
else{
  Do_that((threadIdx.x%32)*2+1);
}
```



Vector Reduction: Naïve Mapping (I)



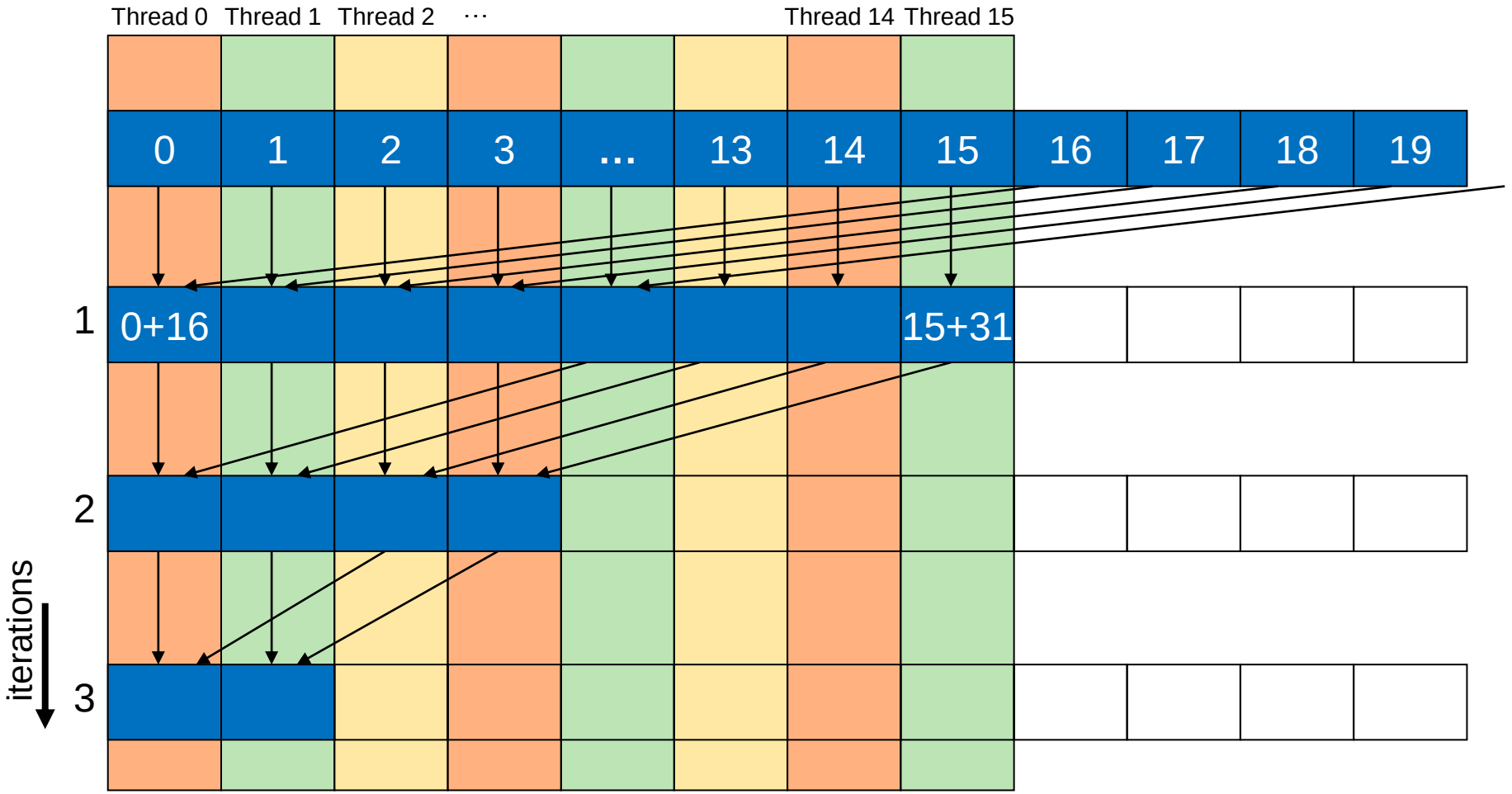
Vector Reduction: Naïve Mapping (II)

- Program with low SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {

  __syncthreads();

  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t + stride];

}
```

Divergence-Free Mapping (I)

- All active threads belong to the same warp



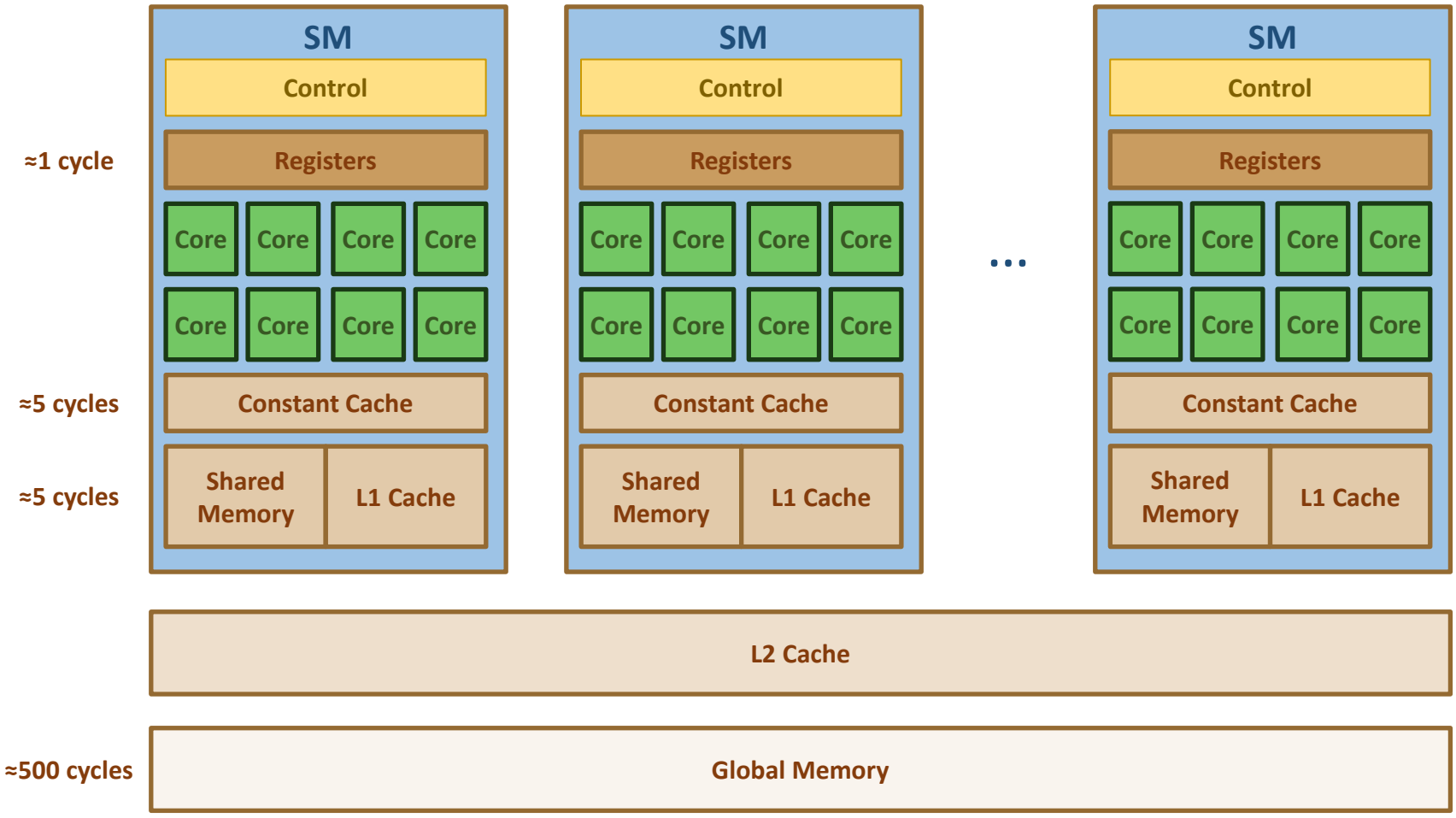
Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 0;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```

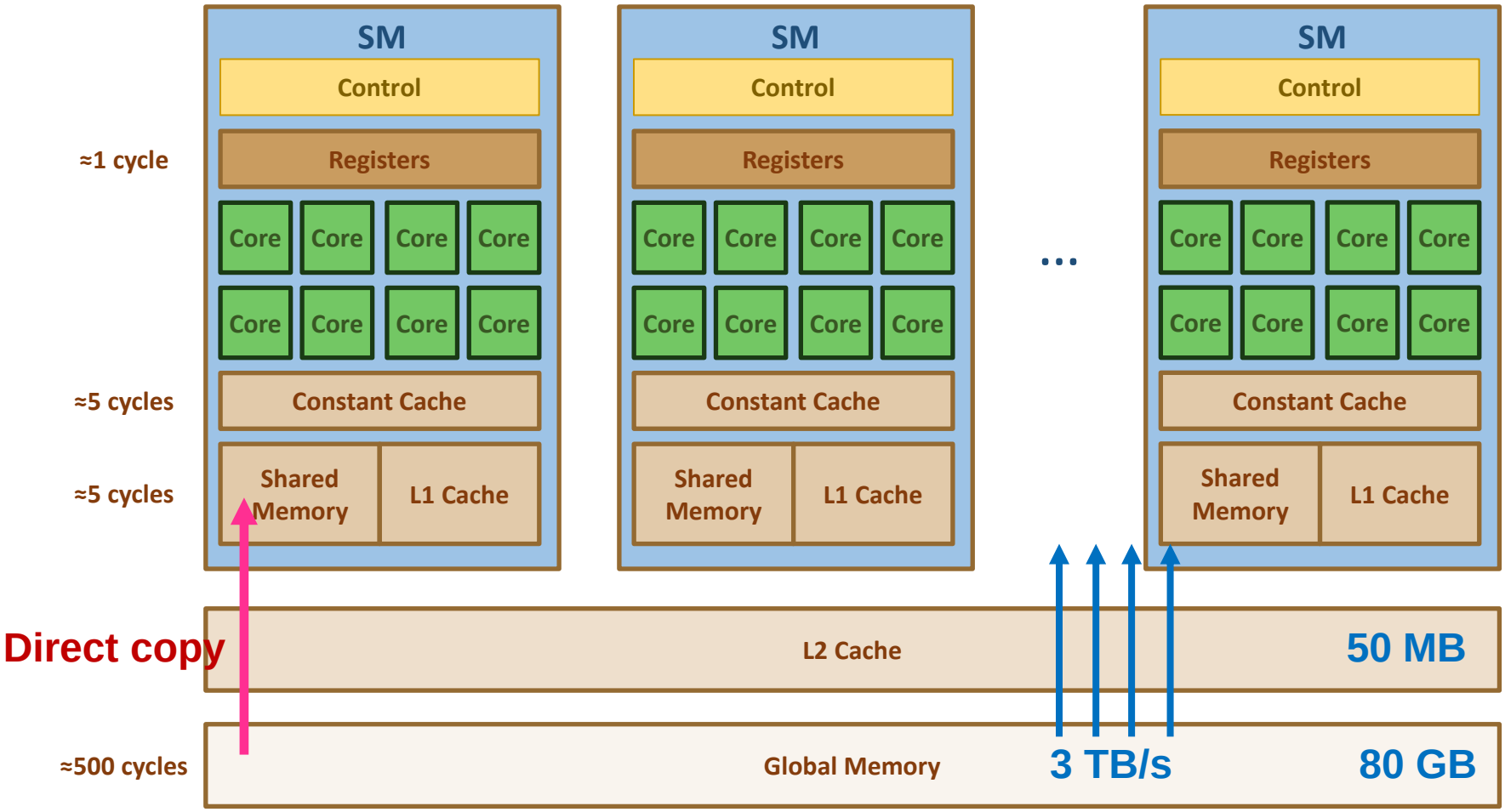
GPU Memories

Memory in the GPU Architecture



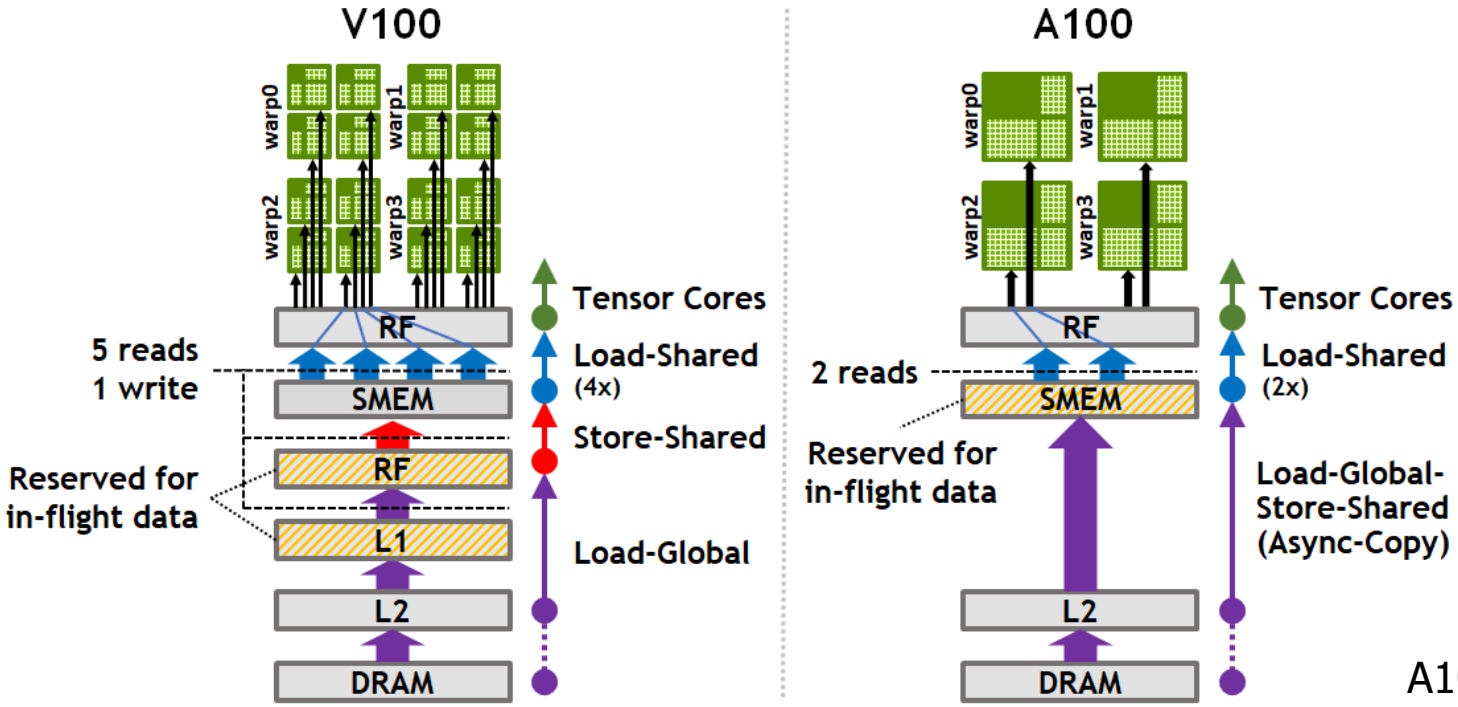
Slide credit: Izzat El Hajj

Memory in the GPU Architecture



NVIDIA V100 & A100 Memory Hierarchy

- Example of data movement between GPU global memory (DRAM) and GPU cores.



A100 improves SM bandwidth efficiency with a new load-global-store-shared asynchronous copy instruction that bypasses L1 cache and register file (RF). Additionally, A100's more efficient Tensor Cores reduce shared memory (SMEM) loads.

A100 feature: Direct copy from L2 to scratchpad, bypassing L1 and register file.

https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

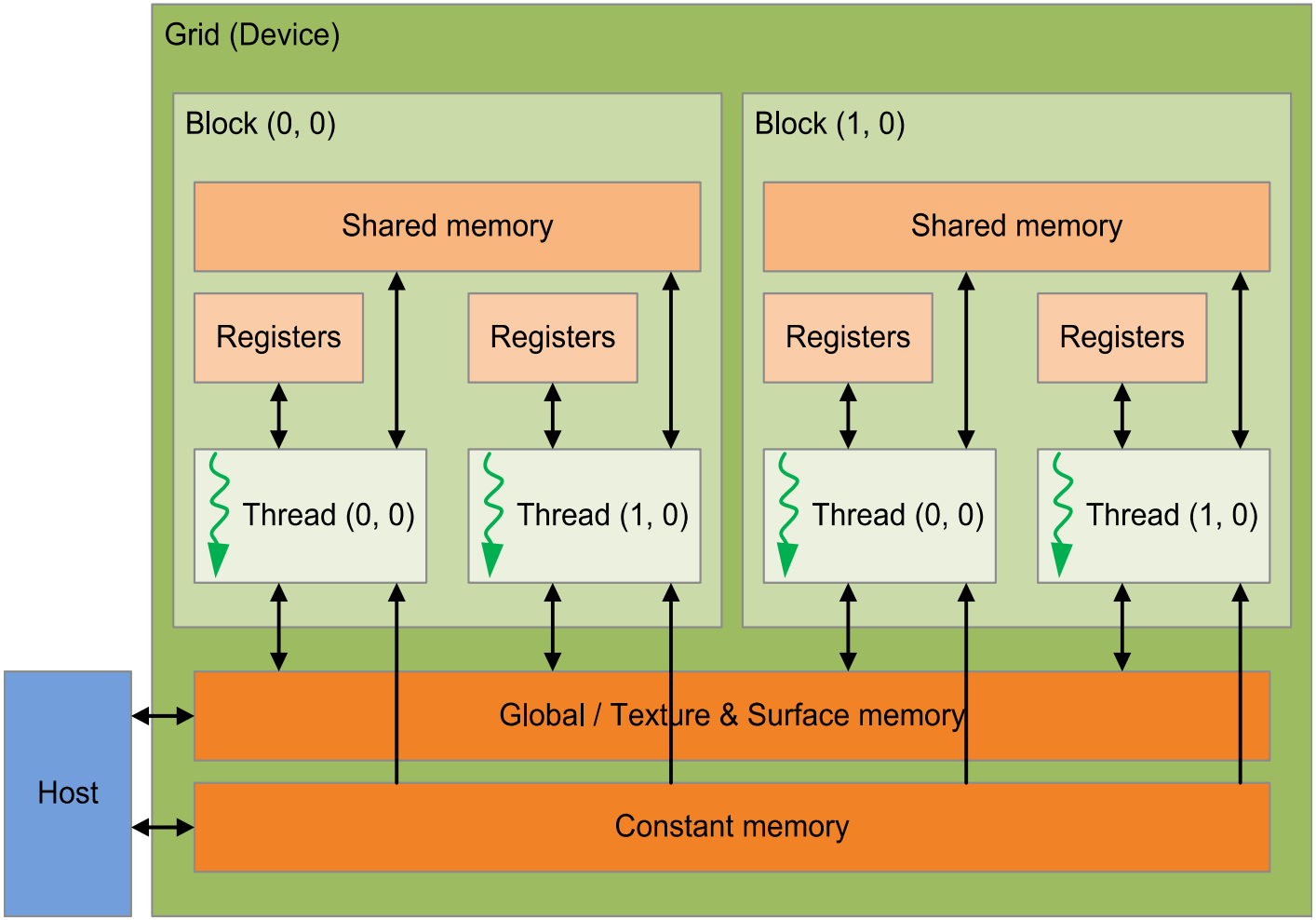
CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`int localArr[N];`	global	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`

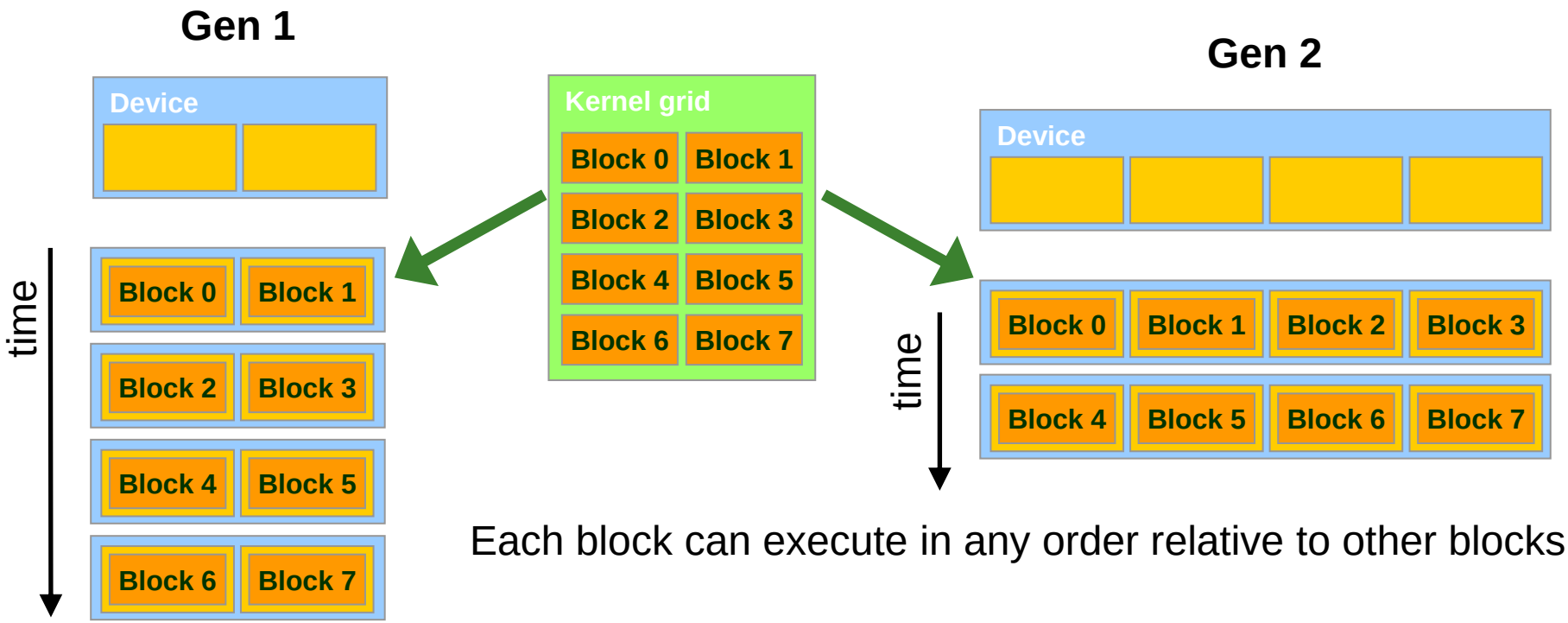
- Recall `cudaMalloc(…)` allocates memory from the host
  - Constant memory can also be allocated and initialized from the host
- Automatic variables without any qualifier reside in a register
  - Except arrays that reside in global memory

Memory Hierarchy in CUDA Programs



Nvidia's Success: Transparent Scalability

- Hardware is free to schedule thread blocks



Each block can execute in any order relative to other blocks.

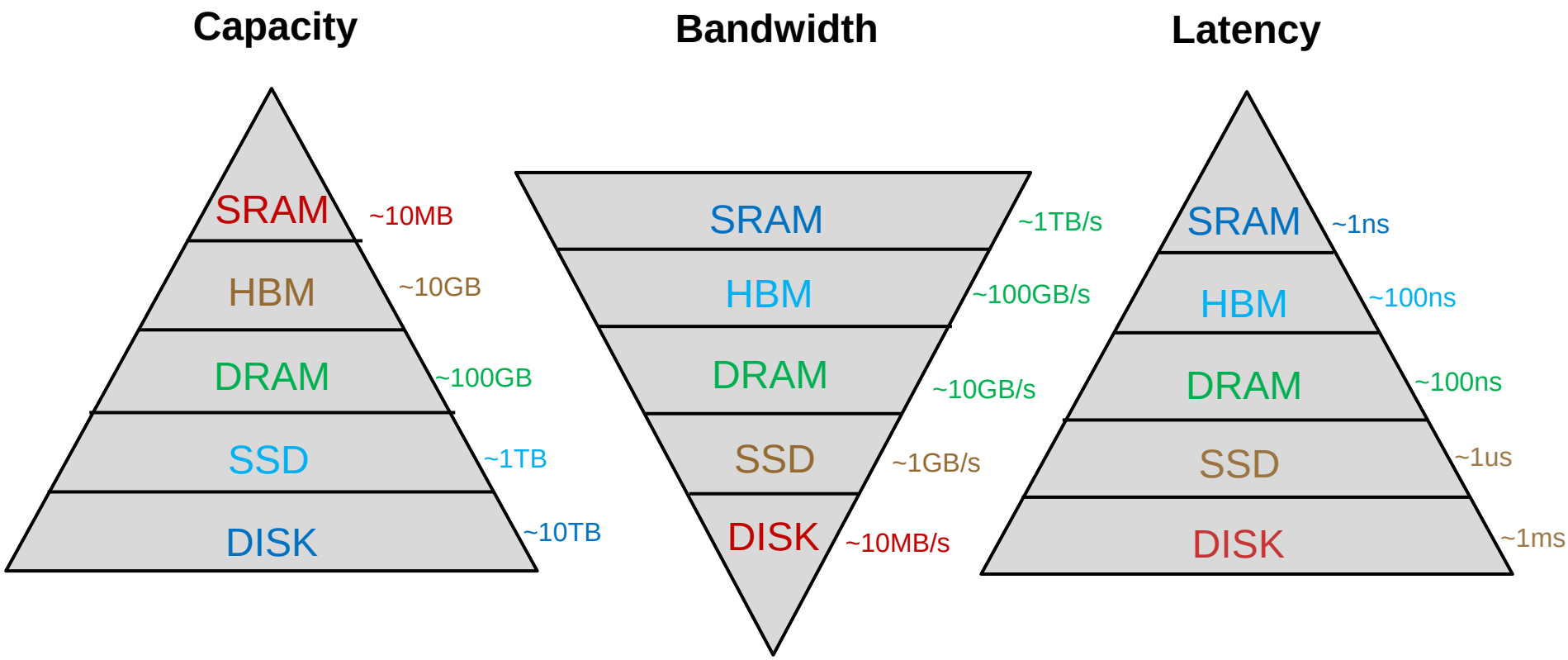
The CUDA code stays the same and enjoys performance improvement while GPU hardware evolves.

Slide credit: Hwu & Kirk

Key Messages:

- Programming model is the key success of Nvidia, rather than the GPU itself.
- GPU has an order of magnitude higher memory bandwidth and compute power than CPU.
- Offloading a task to GPU pays off only when the task has enough compute intensity.
- AI task needs compute-intensive accelerators, e.g., GPU and AI processor.

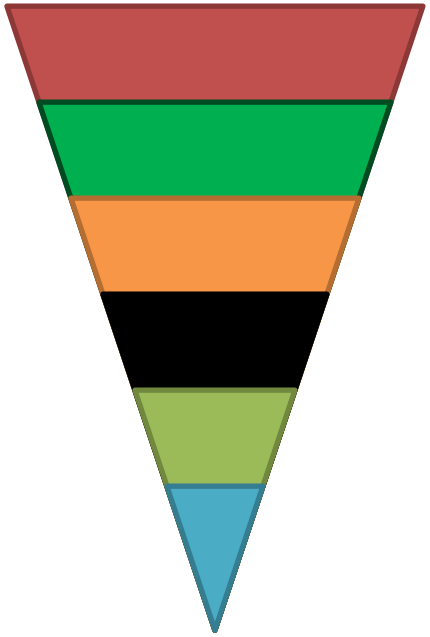
Recall: Comparison of Memories



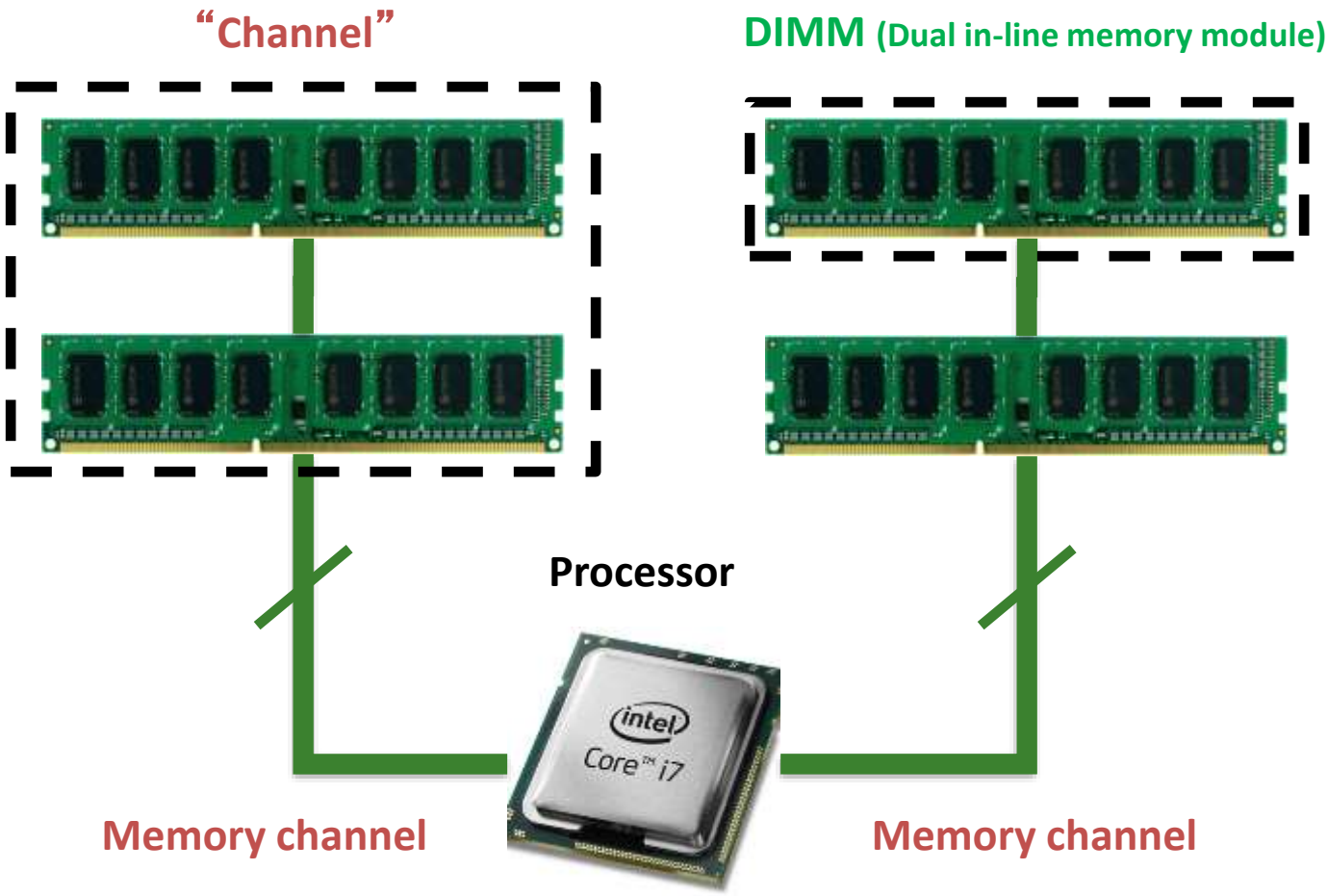
The DRAM Subsystem
The Top-Down View

DRAM Subsystem Organization

- Channel
- DIMM
- Rank
- Chip
- Bank
- Row/Column



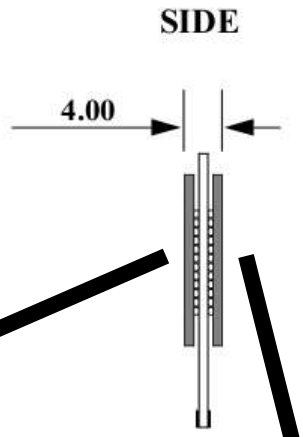
The DRAM Subsystem



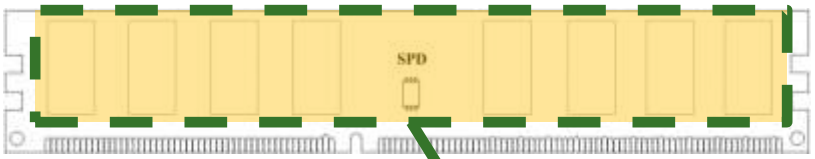
Breaking down a DIMM (module)

DIMM (Dual in-line memory module)

Side view

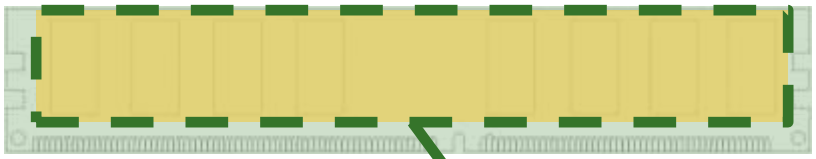


Front of DIMM



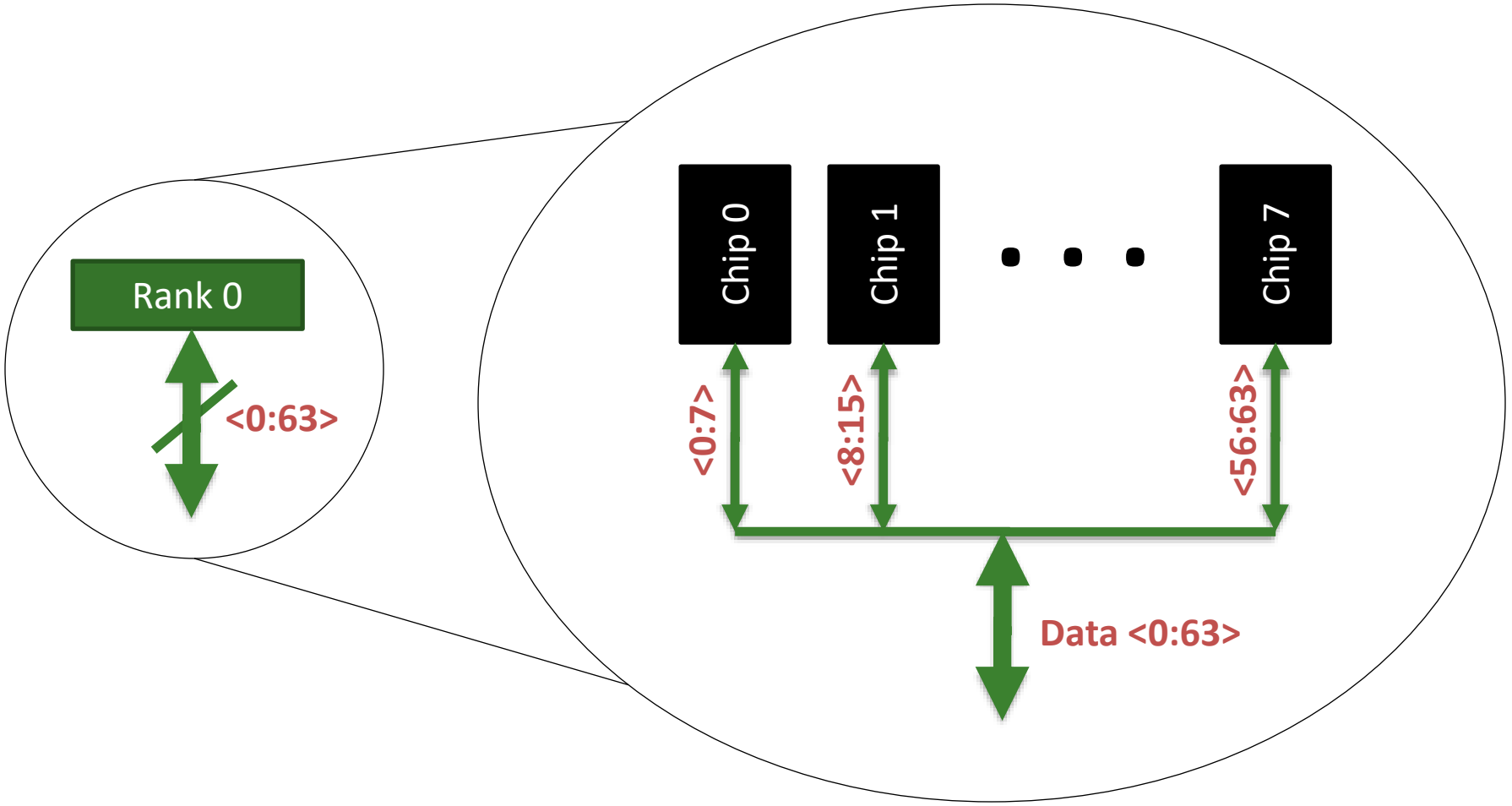
Rank 0: collection of 8 chips

Back of DIMM

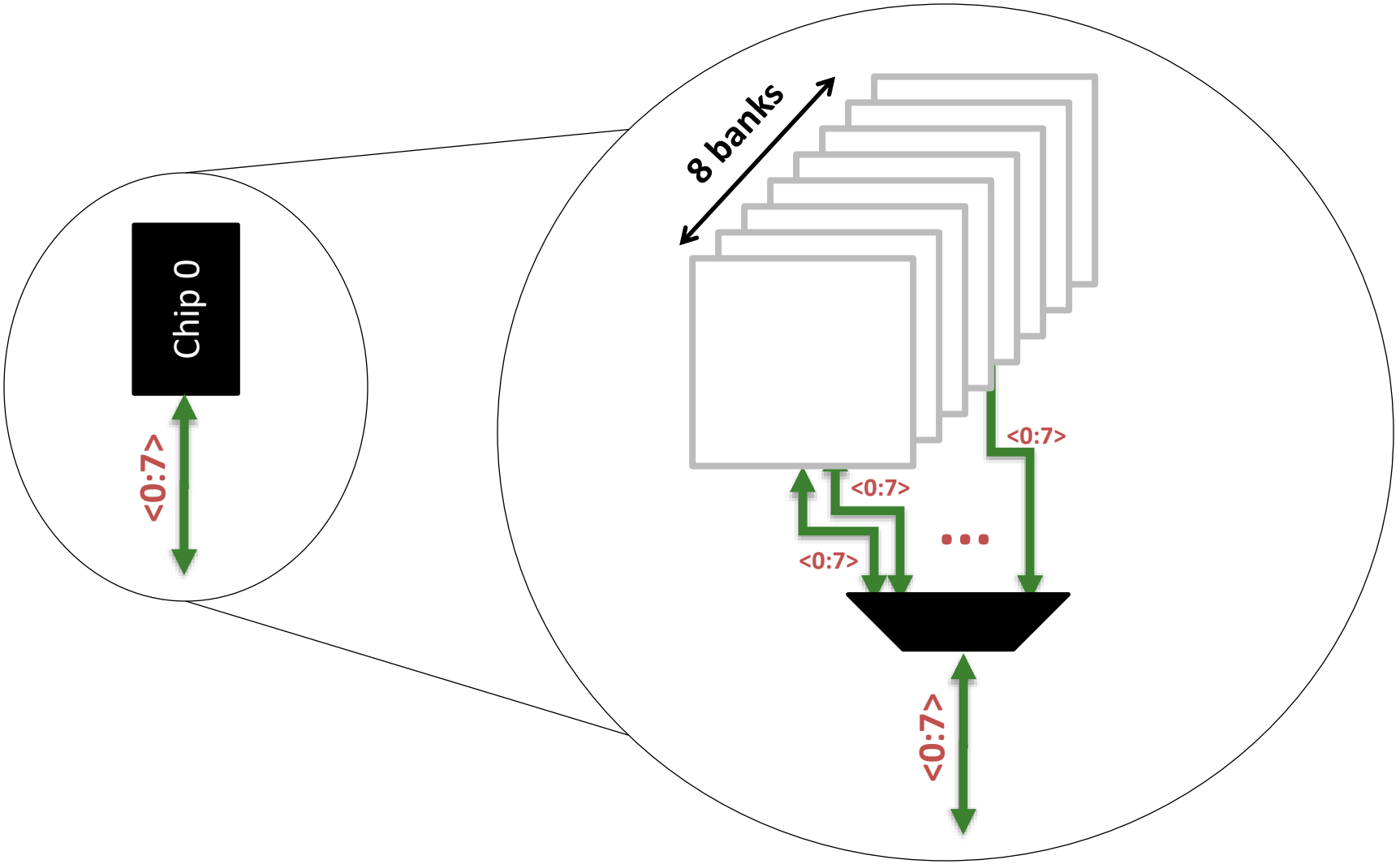


Rank 1

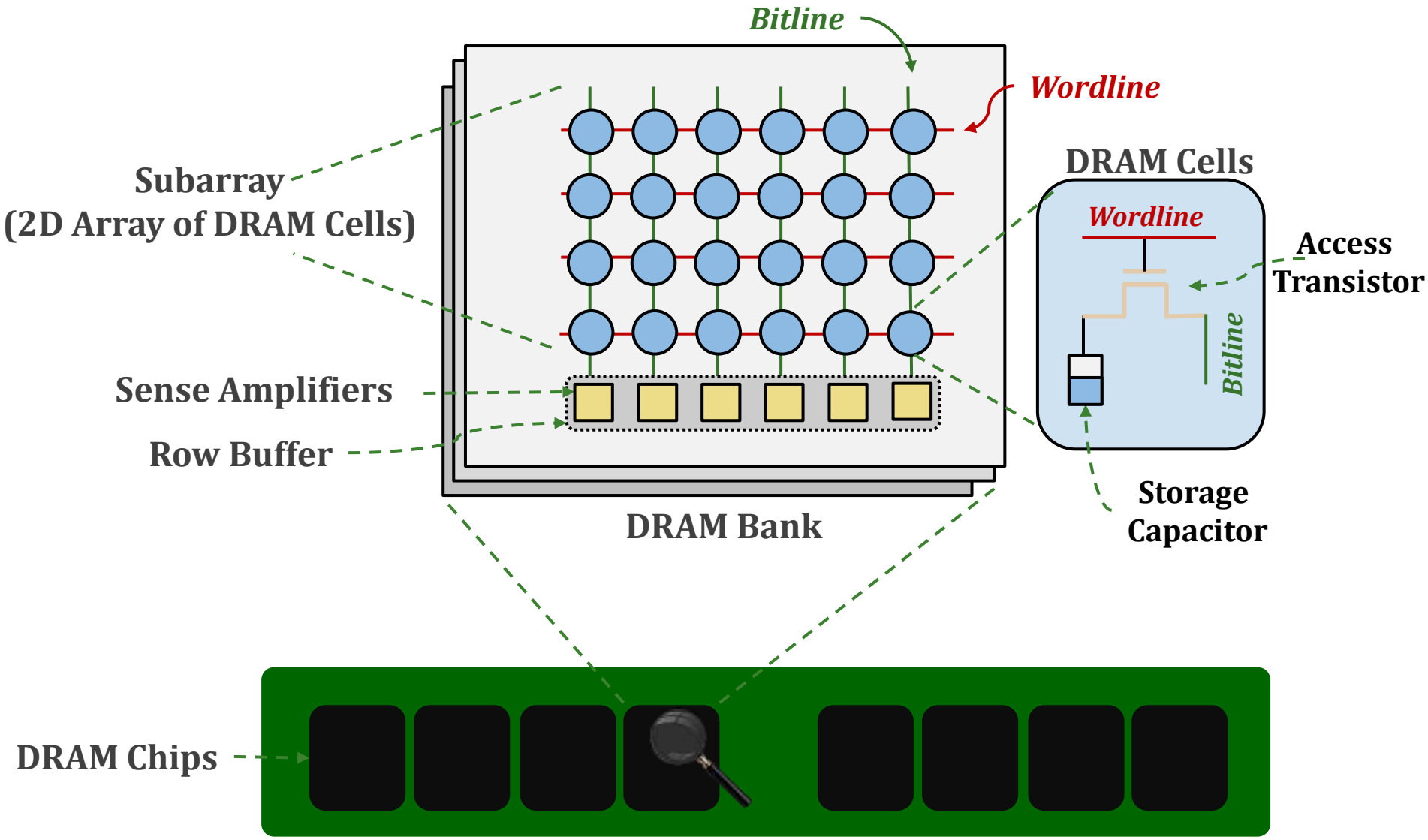
Breaking down a Rank



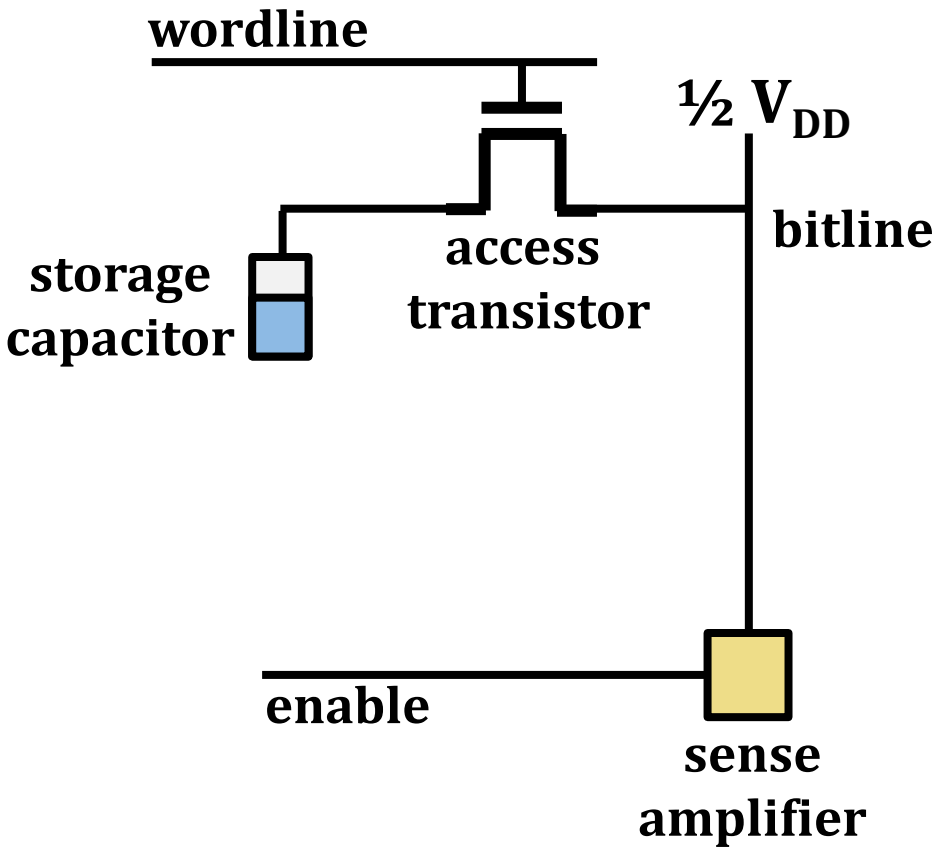
Breaking down a Chip



Inside a DRAM Chip



DRAM Cell Operation

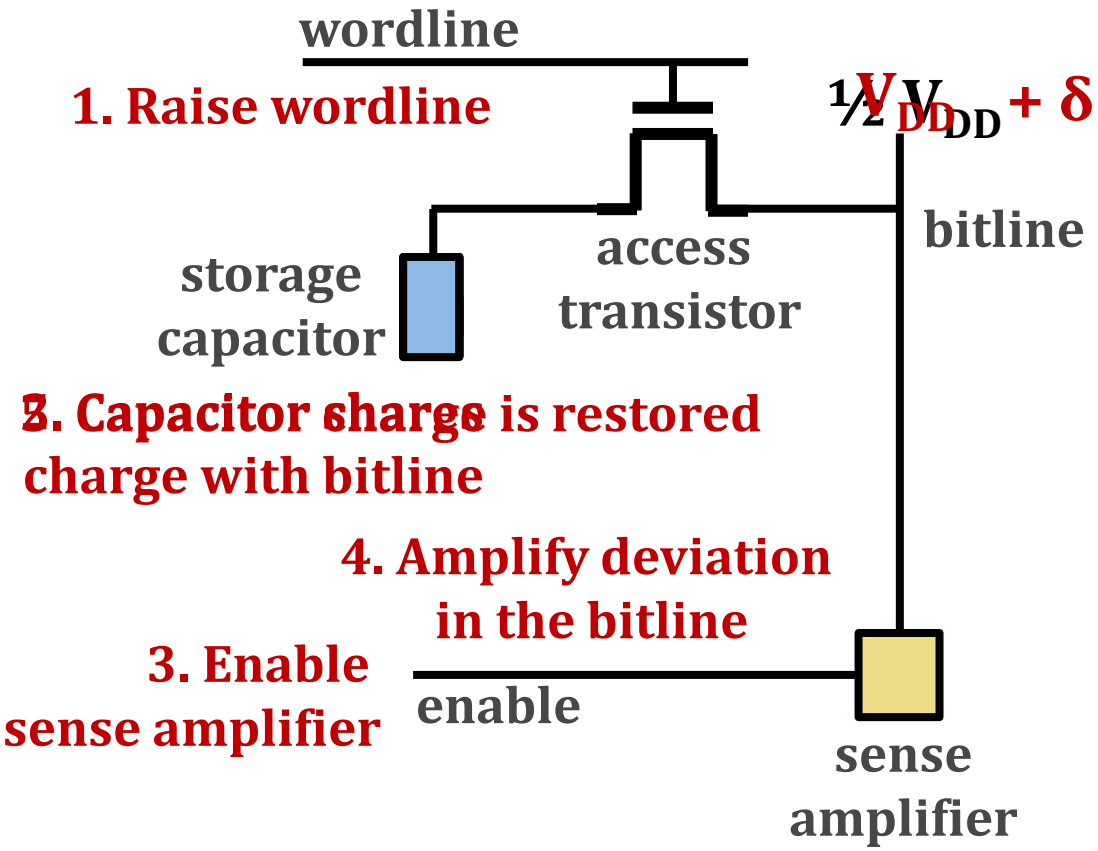


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation - ACTIVATE

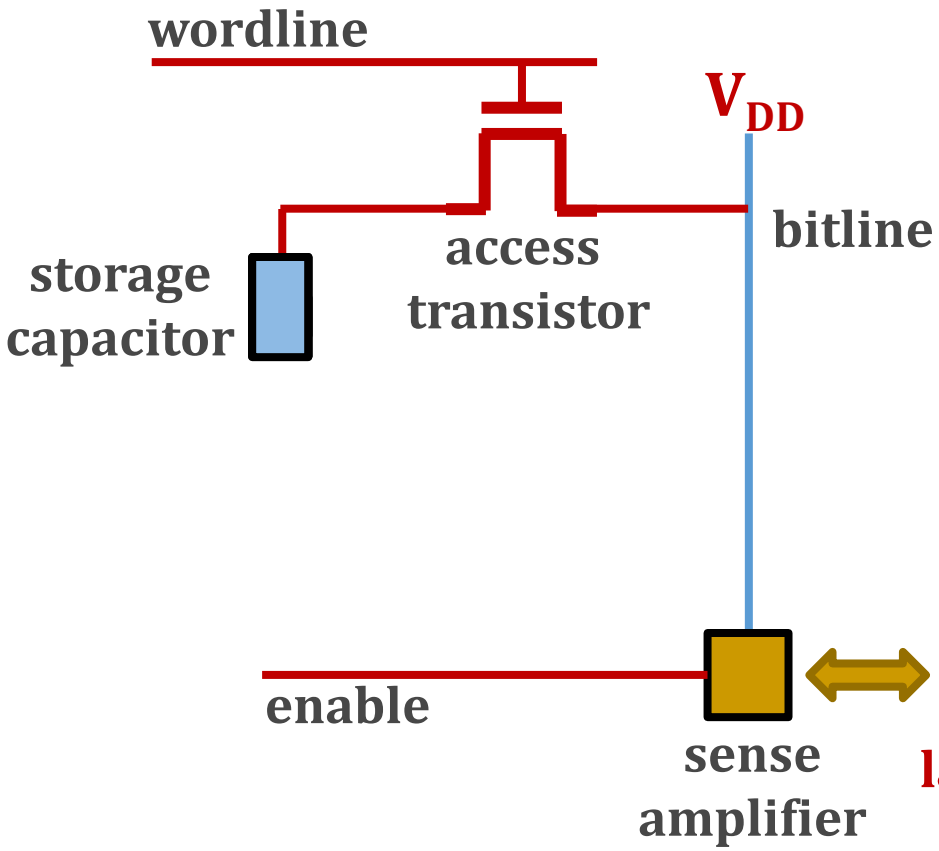


1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Cell Operation – READ/WRITE



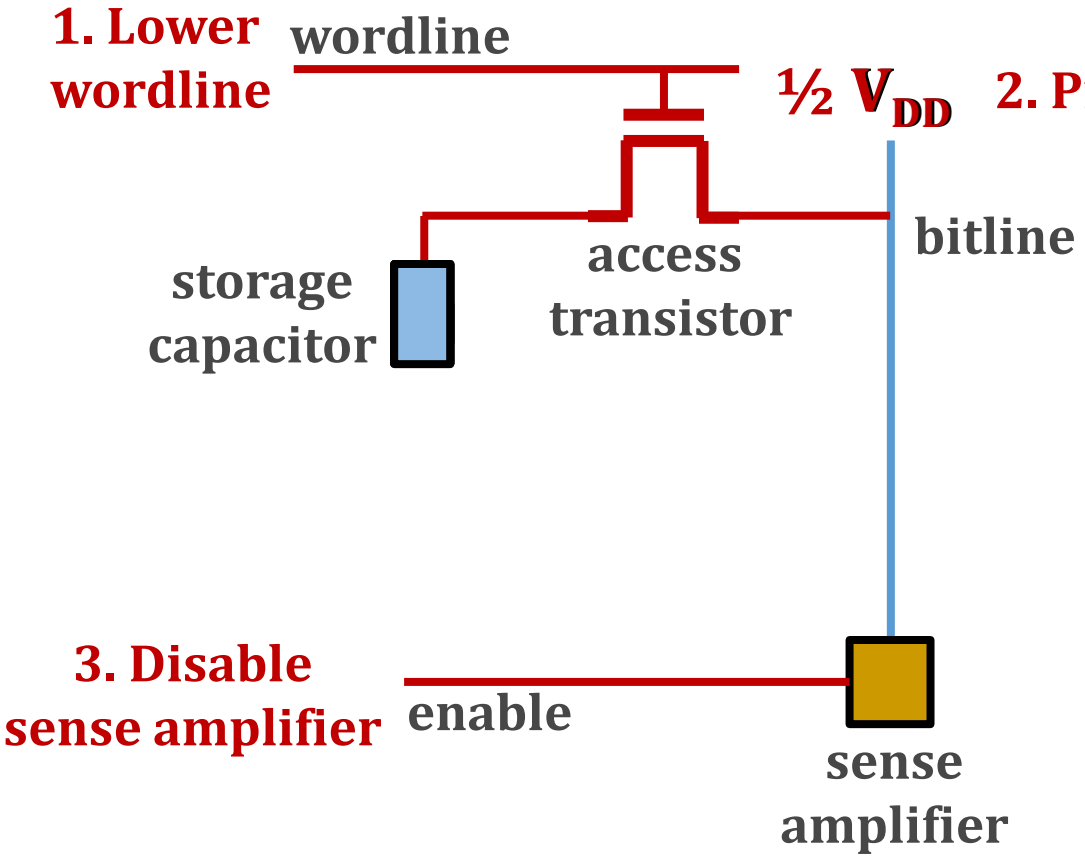
1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

Read/Write the value latched in sense amplifier

DRAM Cell Operation - PRECHARGE



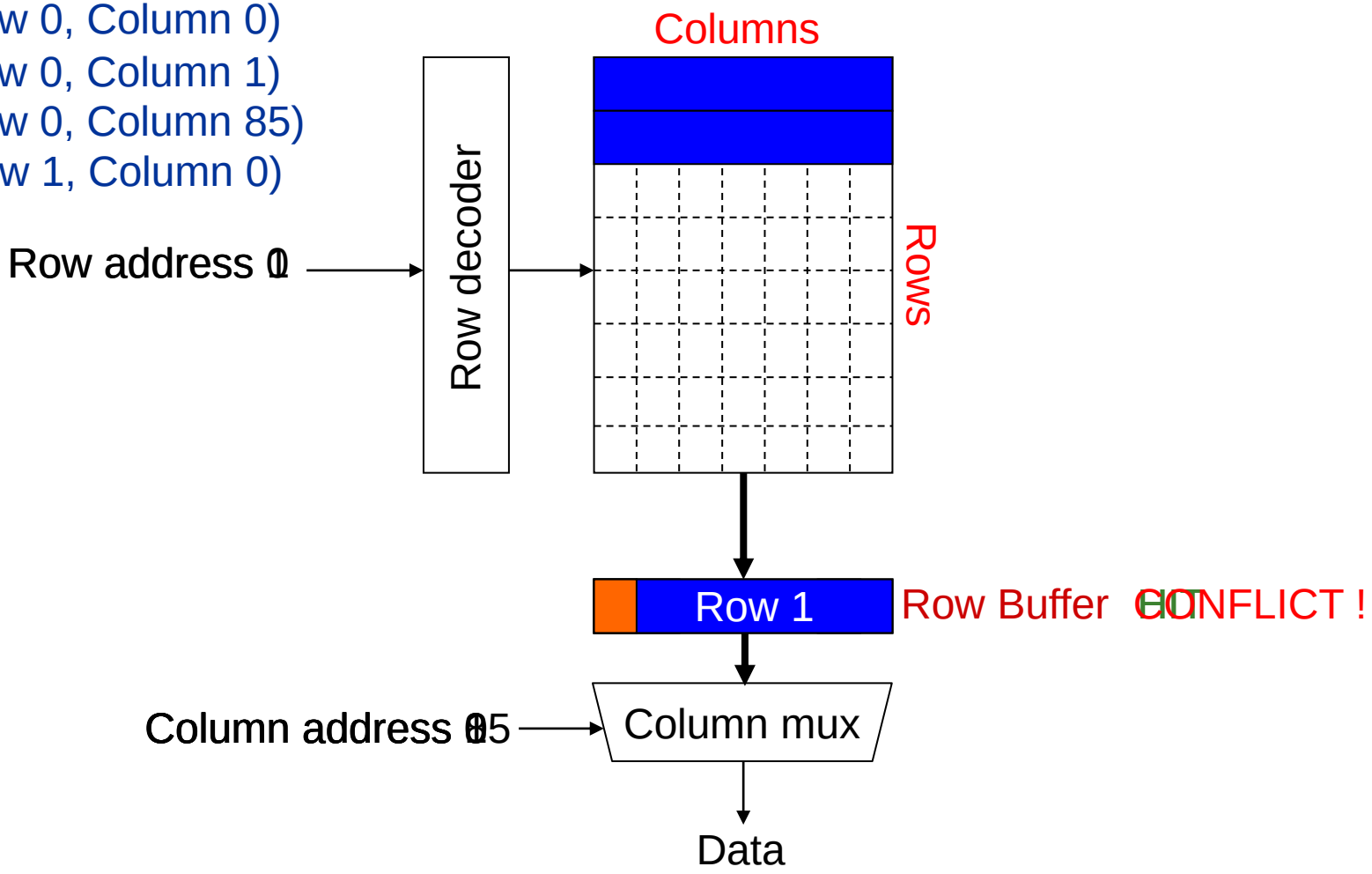
1. ACTIVATE (ACT)

2. READ/WRITE

3. PRECHARGE (PRE)

DRAM Bank Operation

Access Address:
(Row 0, Column 0)
(Row 0, Column 1)
(Row 0, Column 85)
(Row 1, Column 0)



Long Global Memory Access Latency

How to optimize global memory access?

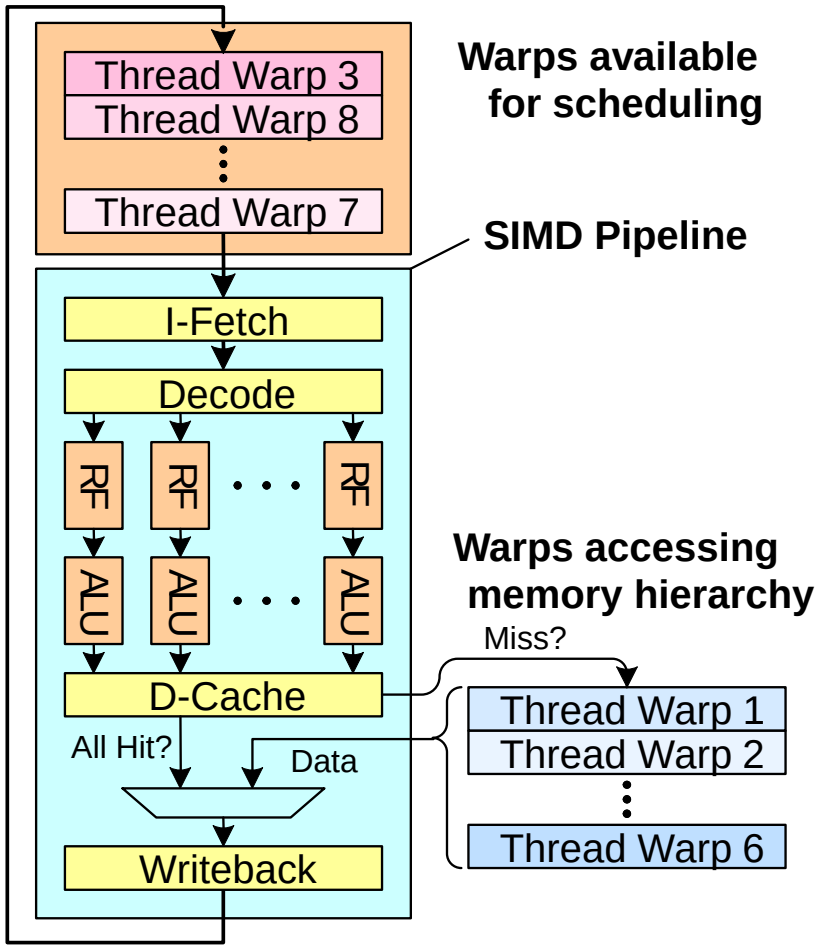
Multithreading

Memory Coalescing

Shared Memory

Latency Hiding via Warp-Level FGMT

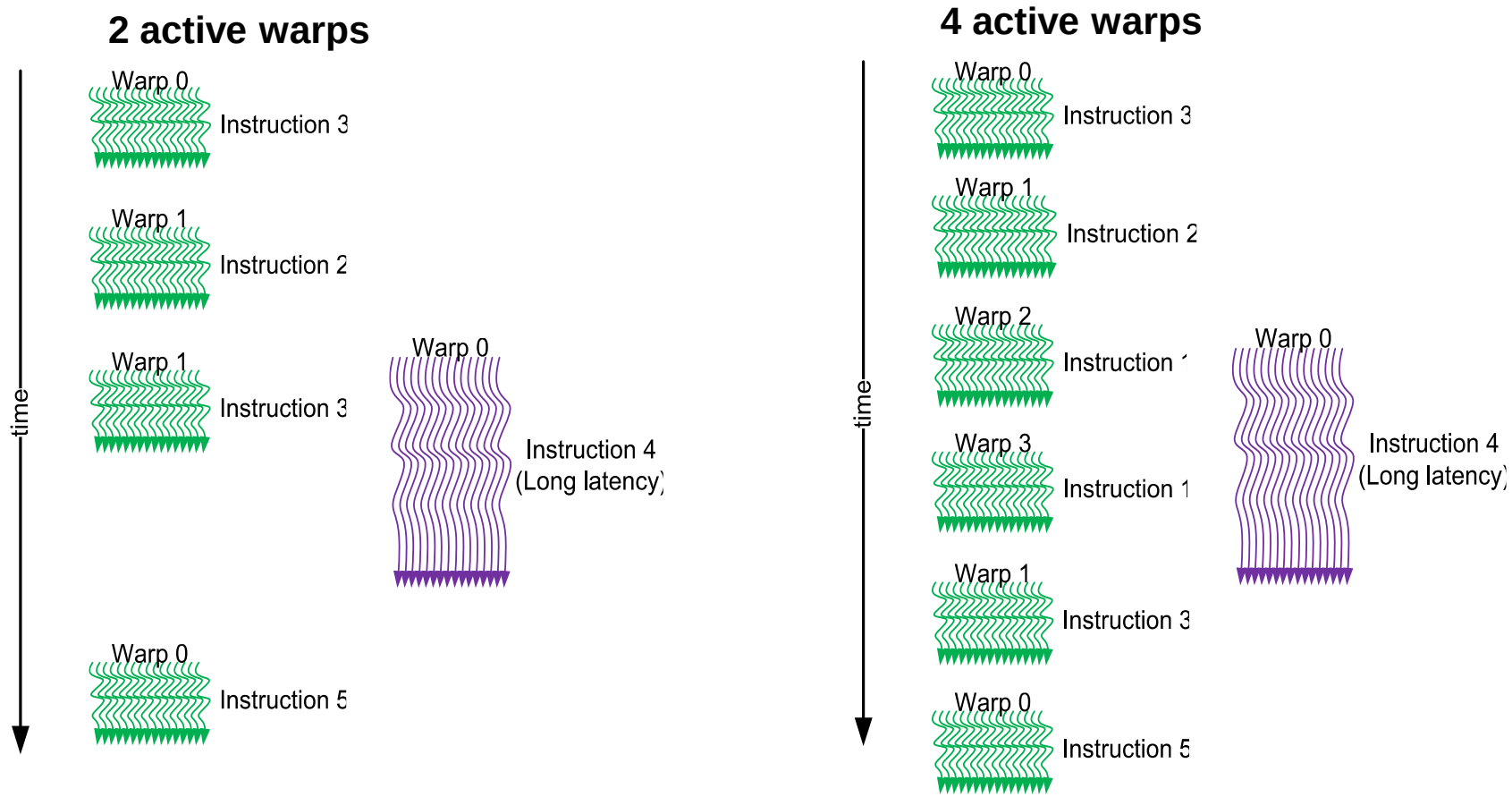
- **Warp**: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels



Slide credit: Tor Aamodt

Latency Hiding and Occupancy

- FGMT can hide long latency operations (e.g., memory accesses)
- Occupancy: ratio of active warps to the maximum number of warps per GPU core



Memory Coalescing (I)

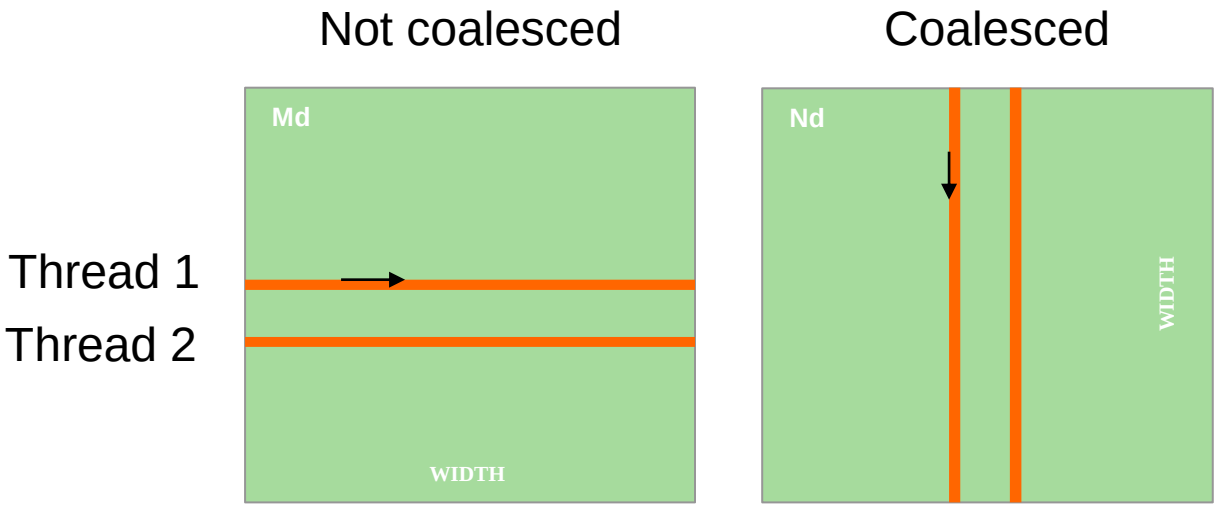
- Memory Coalescing:
  - When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called memory divergence

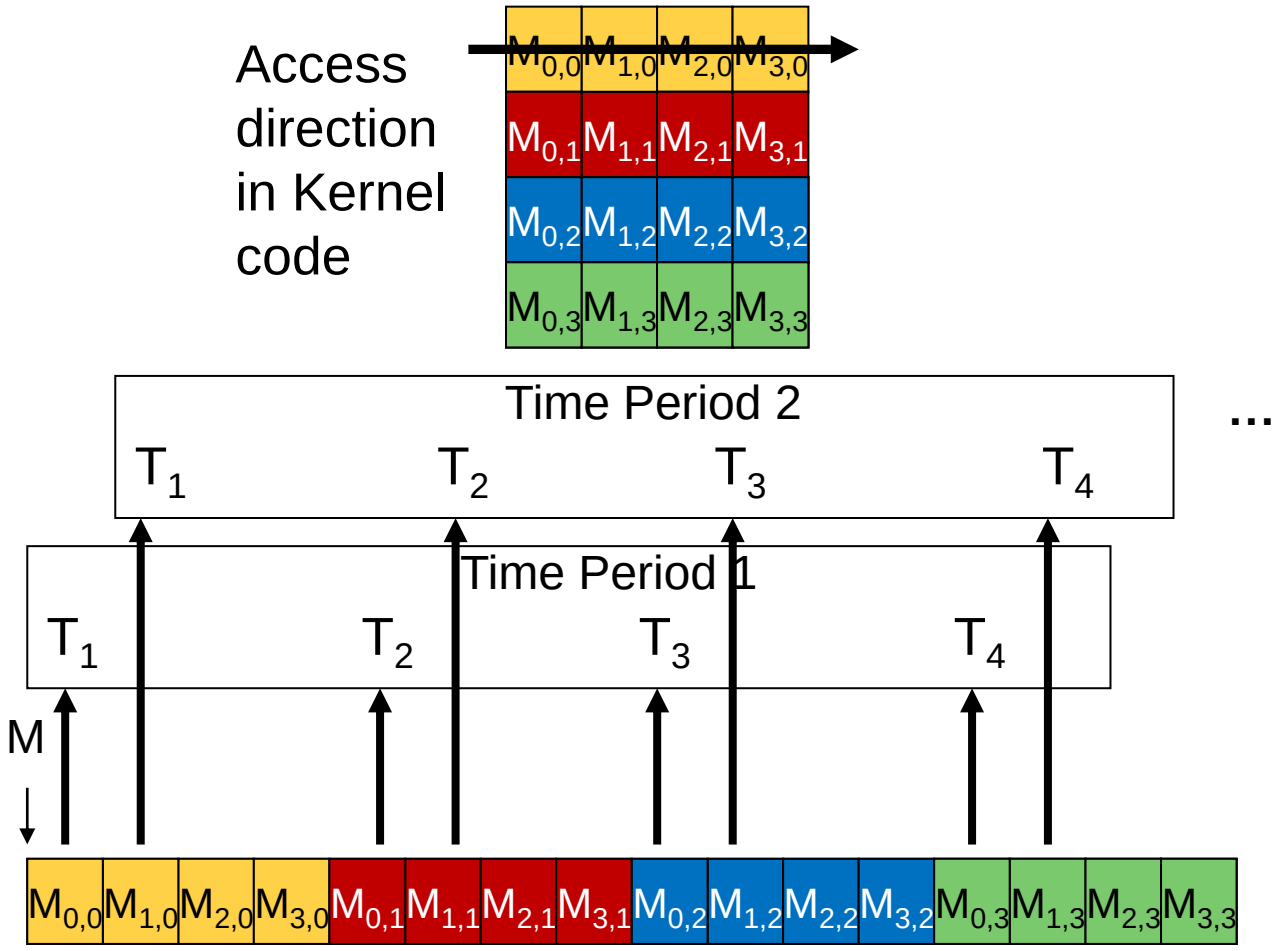
Slide credit: Izzat El Hajj

Memory Coalescing (II)

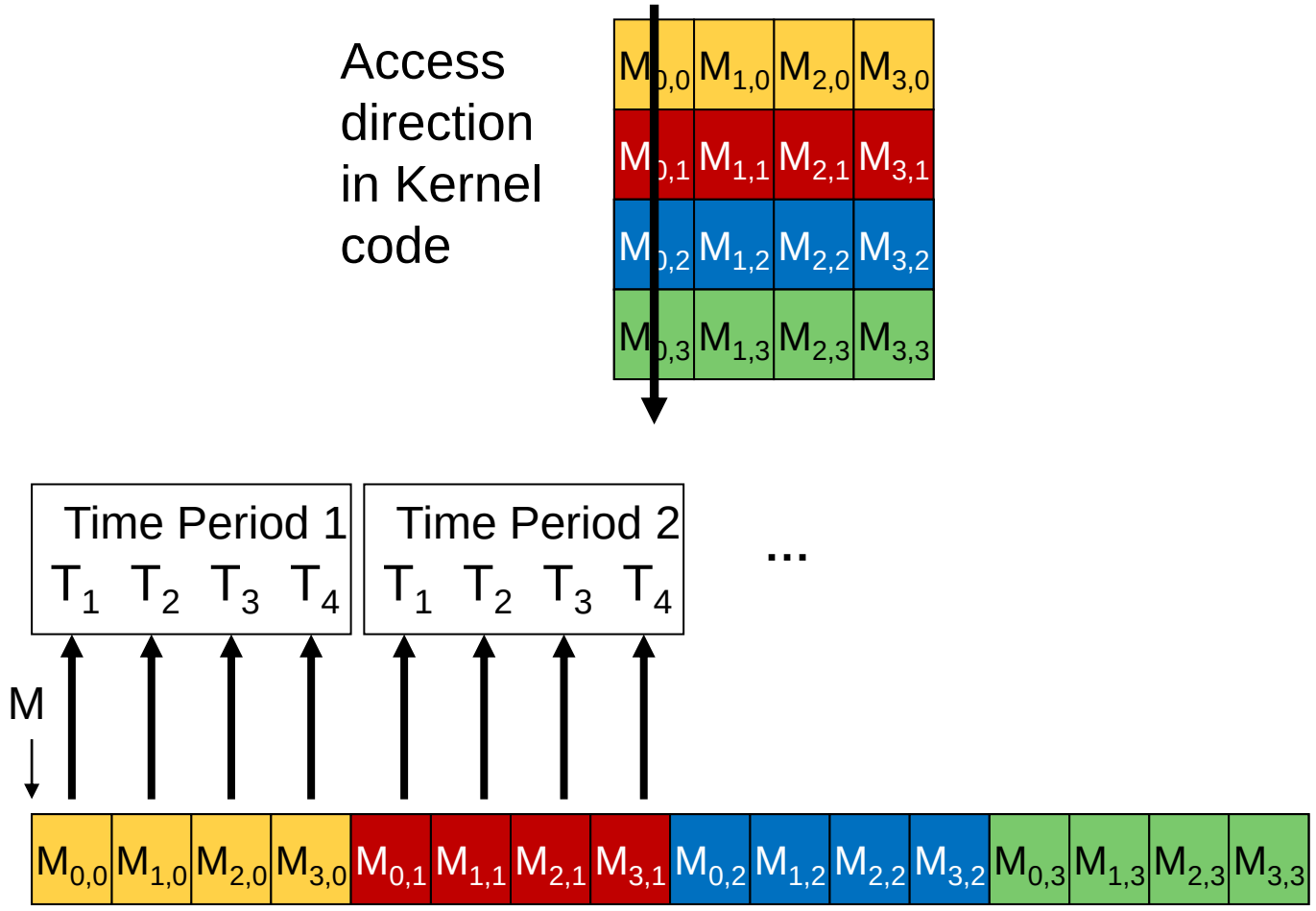
- When accessing global memory, we want to make sure that concurrent threads access nearby memory locations
- Peak bandwidth utilization occurs when all threads in a warp access one cache line (or several consecutive cache lines)



Uncoalesced Memory Accesses



Coalesced Memory Accesses



Shared Memory

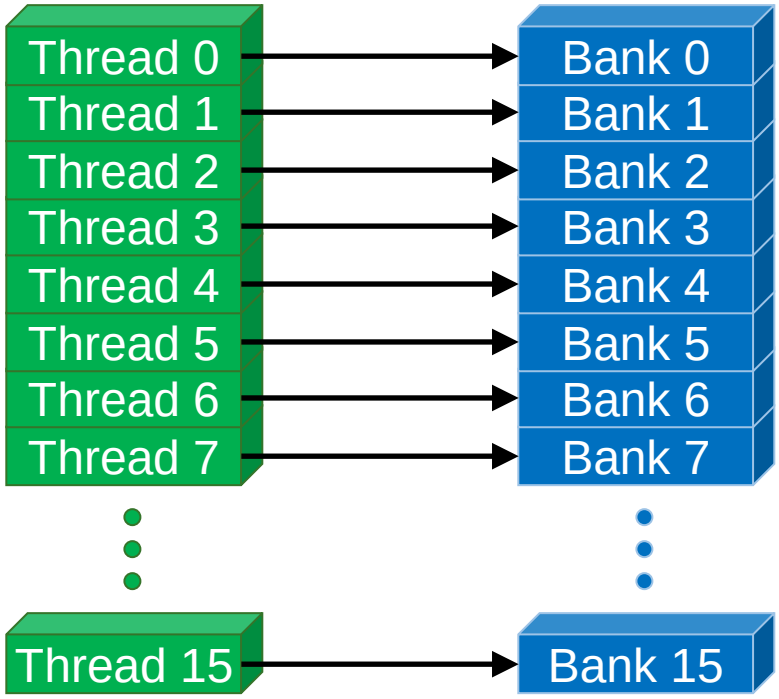
- Shared memory is an interleaved (banked) memory
  - Each bank can service one address per cycle

- Typically, 32 banks in NVIDIA GPUs
  - Successive 32-bit words are assigned to successive banks
    - Bank = Address % 32

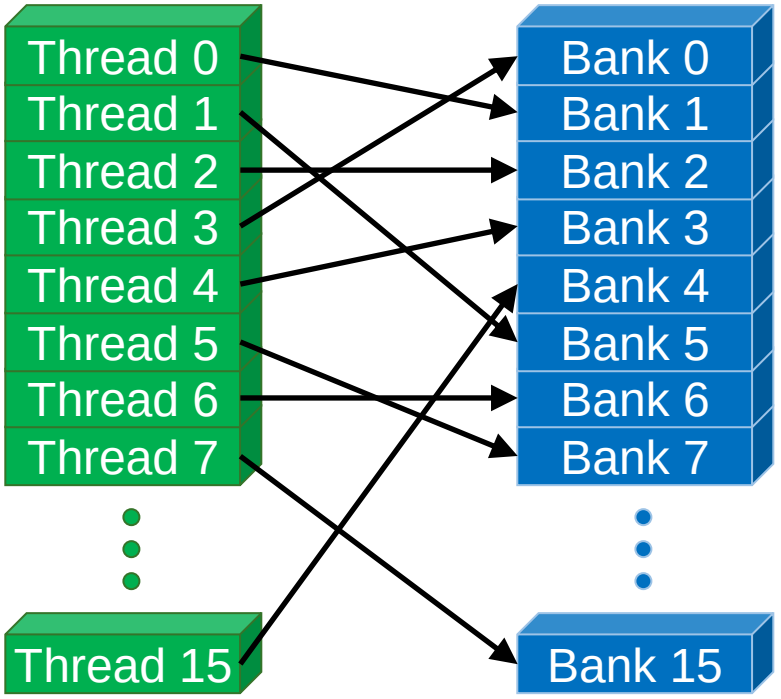
- Bank conflicts are only **possible within a warp**
  - No bank conflicts between different warps

Shared Memory Bank Conflicts (I)

- Bank conflict free



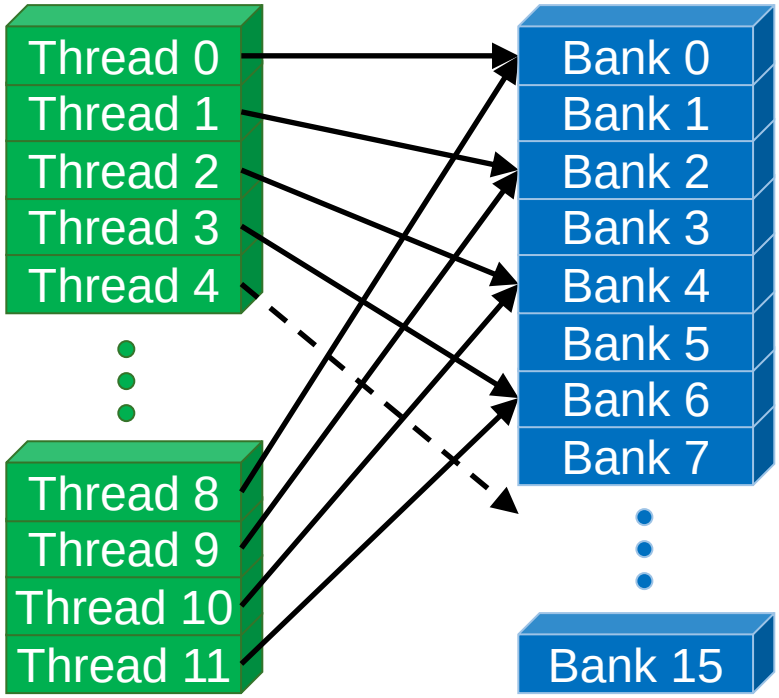
Linear addressing: stride = 1



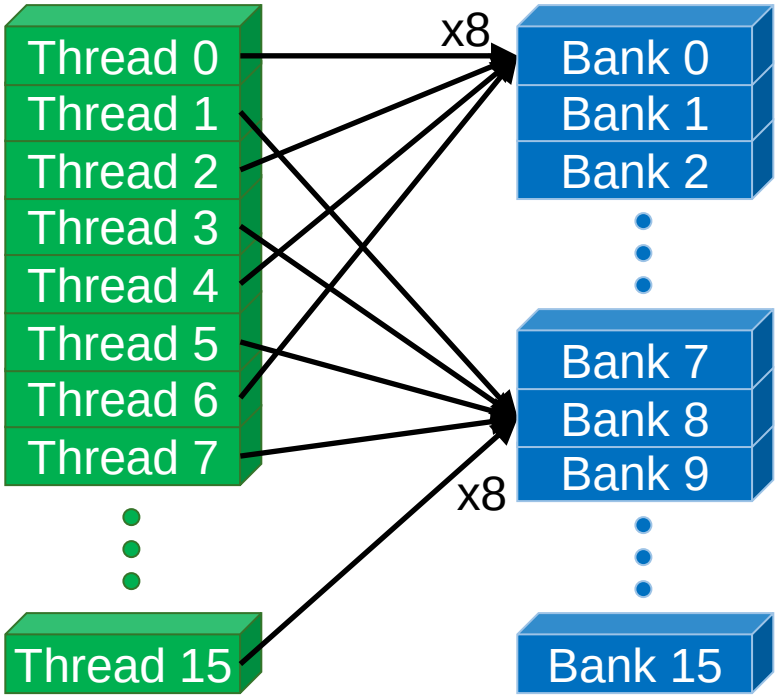
Random addressing 1:1

Shared Memory Bank Conflicts (II)

- N-way bank conflicts

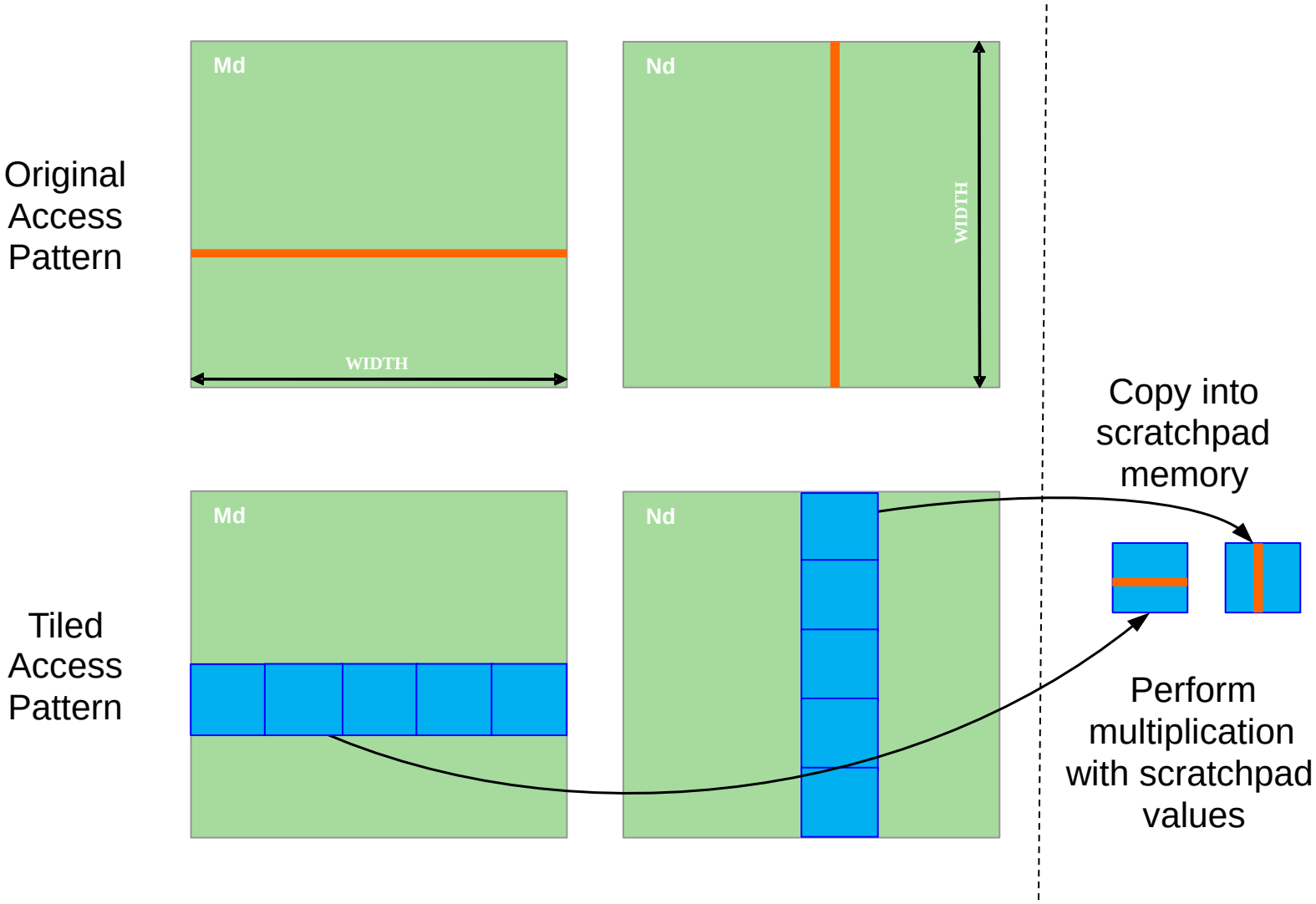


2-way bank conflict: stride = 2



8-way bank conflict: stride = 8

Use Shared Memory to Improve Coalescing



Slide credit: Hwu & Kirk

Reducing Shared Memory Bank Conflicts

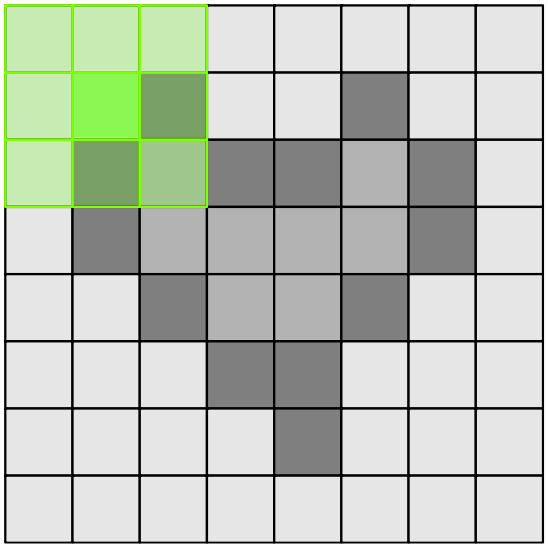
- Bank conflicts are only possible within a warp
  - No bank conflicts between different warps

- If strided accesses are needed, some optimization techniques can help
  - Padding
  - Randomized mapping
    - Rau, "Pseudo-randomly interleaved memory," ISCA 1991
  - Hash functions
    - V.d.Braak+, "Configurable XOR Hash Functions for Banked Scratchpad Memories in GPUs," IEEE TC, 2016

Data Reuse

- Data reuse:
  - Same memory locations accessed by neighboring threads

9 elements per thread

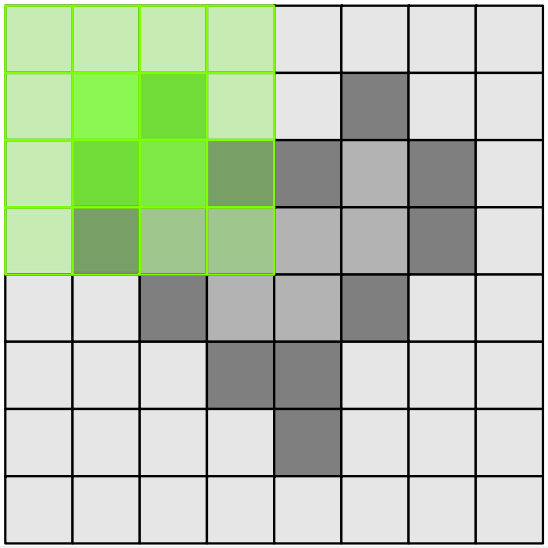


```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Data Reuse: Tiling

- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory

$(L\_SIZE+2)^2/L\_SIZE^2$ elements per thread



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory l_data
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Synchronization Function

- `void __syncthreads();`
- Synchronizes all threads in a block

- Once all threads in a block have reached this point, execution resumes normally

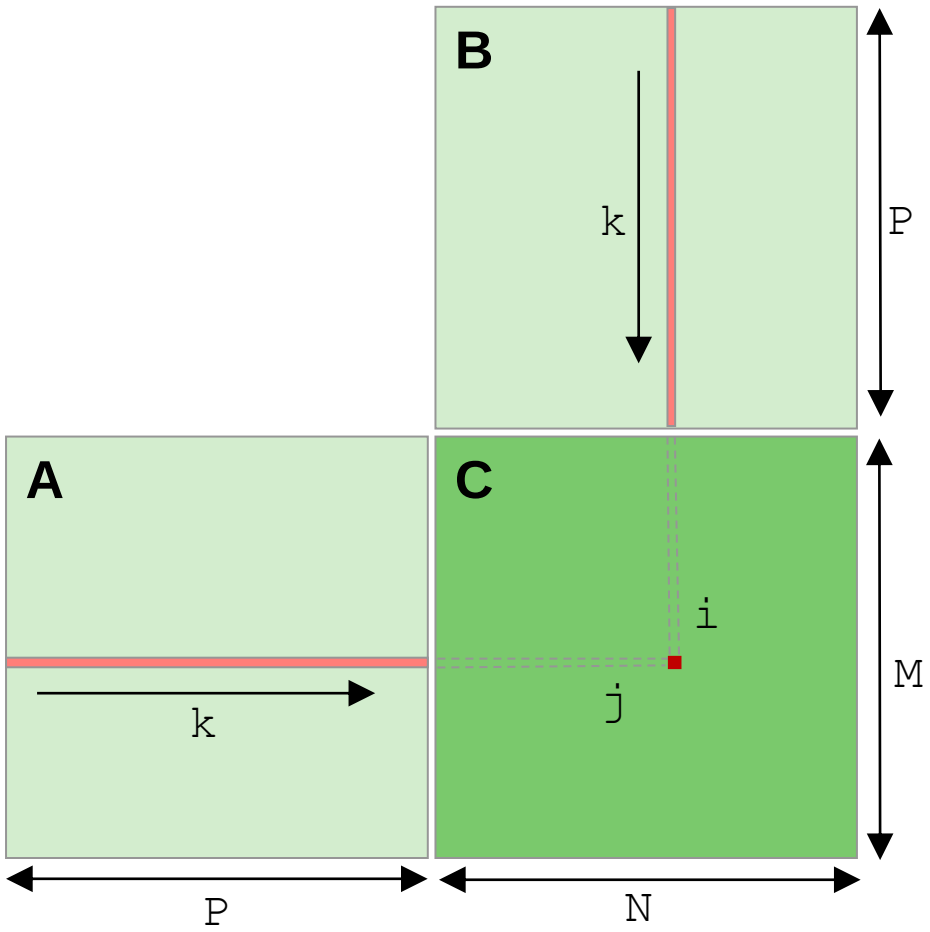
- Used to avoid RAW / WAR / WAW hazards when accessing shared or global memory

Tiling/Blocking in On-chip Memories

- Tiling or Blocking
  - Divide loops operating on arrays into computation chunks so that each chunk can hold its data in the on-chip RAM (or other on-chip memory, e.g., scratchpad)
  - Avoids on-chip RAM conflicts between different chunks of computation
  - Essentially: Divide the working set so that each piece fits in the on-chip RAMs
  - Let's first see an example for CPUs

Naïve Matrix Multiplication (I)

- Matrix multiplication: C = A x B
- Consider two input matrices A and B in row-major layout
  - A size is M x P
  - B size is P x N
  - C size is M x N

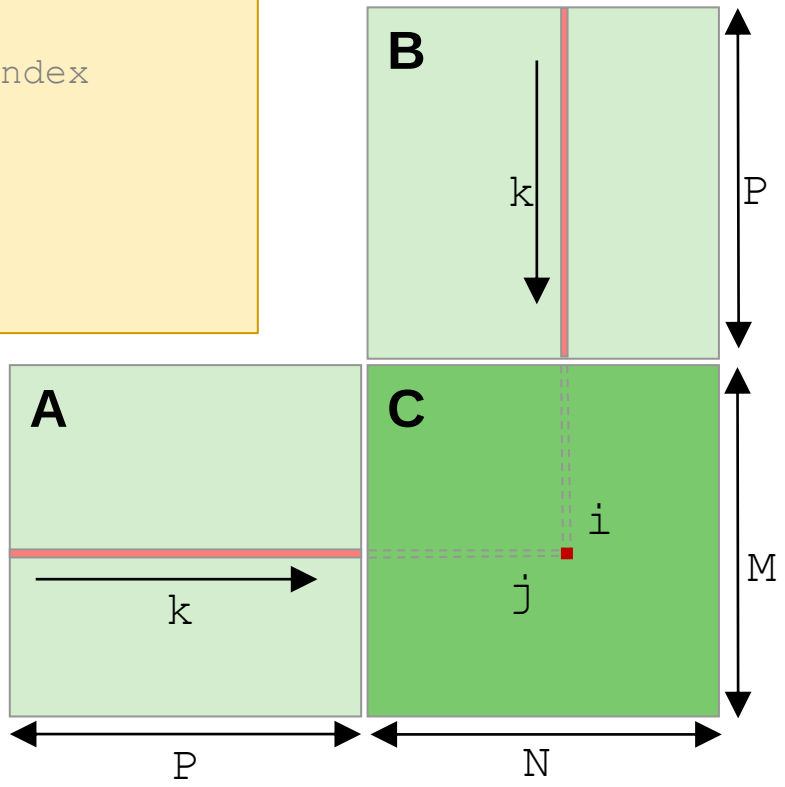


Naïve Matrix Multiplication (II)

- Naïve implementation of matrix multiplication
  - Poor access locality

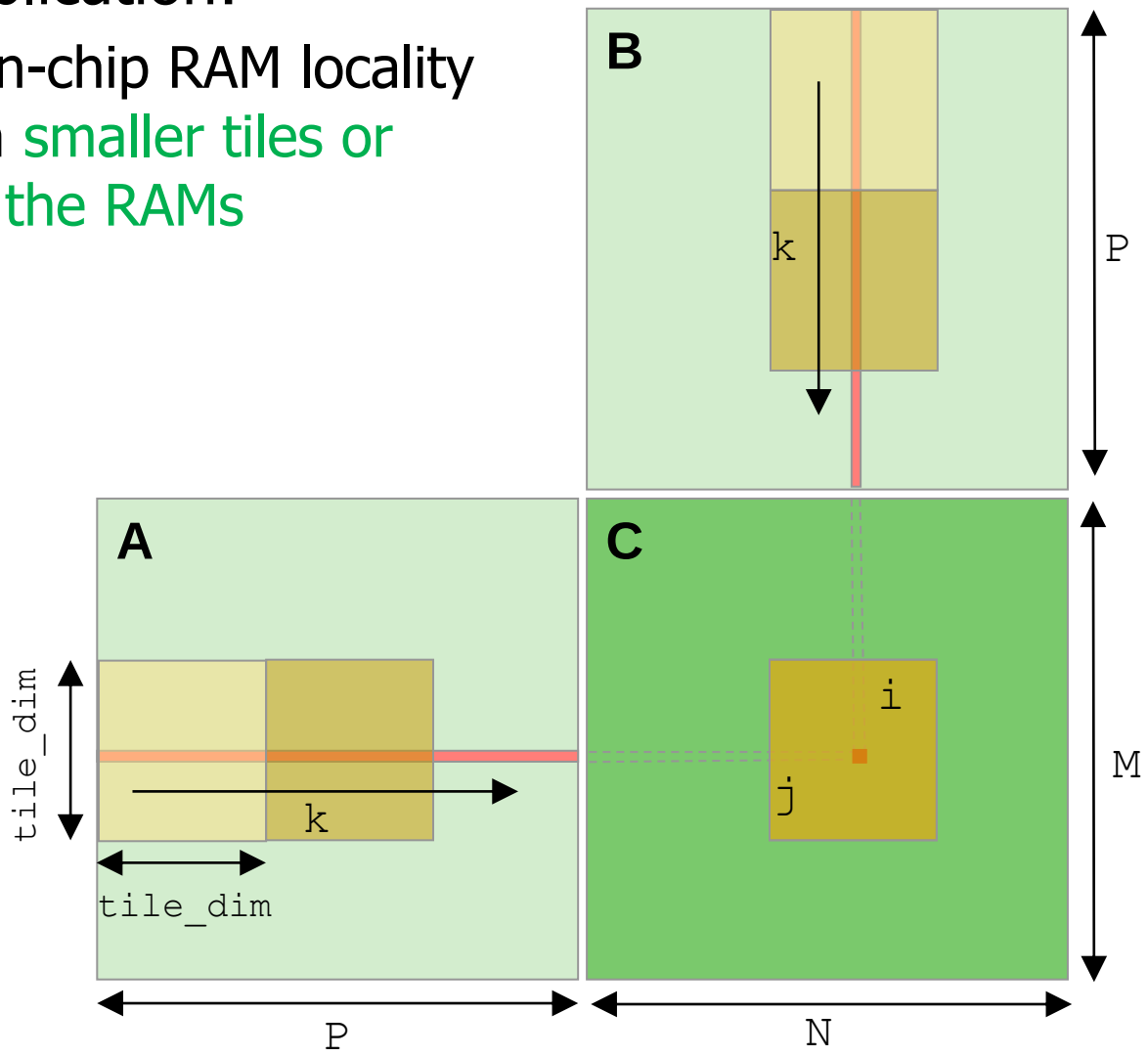
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (i = 0; i < M; i++){ // i = row index
    for (j = 0; j < N; j++){ // j = column index
        C(i, j) = 0; // Set to zero
        for (k = 0; k < P; k++) // Row x Col
            C(i, j) += A(i, k) * B(k, j);
    }
}
```

Consecutive accesses to B are far from each other, in different memory lines.
Every access to B is likely to cause a row buffer miss



Tiled Matrix Multiplication (I)

- Tiled Matrix Multiplication:
  - Achieve better on-chip RAM locality by computing on smaller tiles or blocks that fit in the RAMs



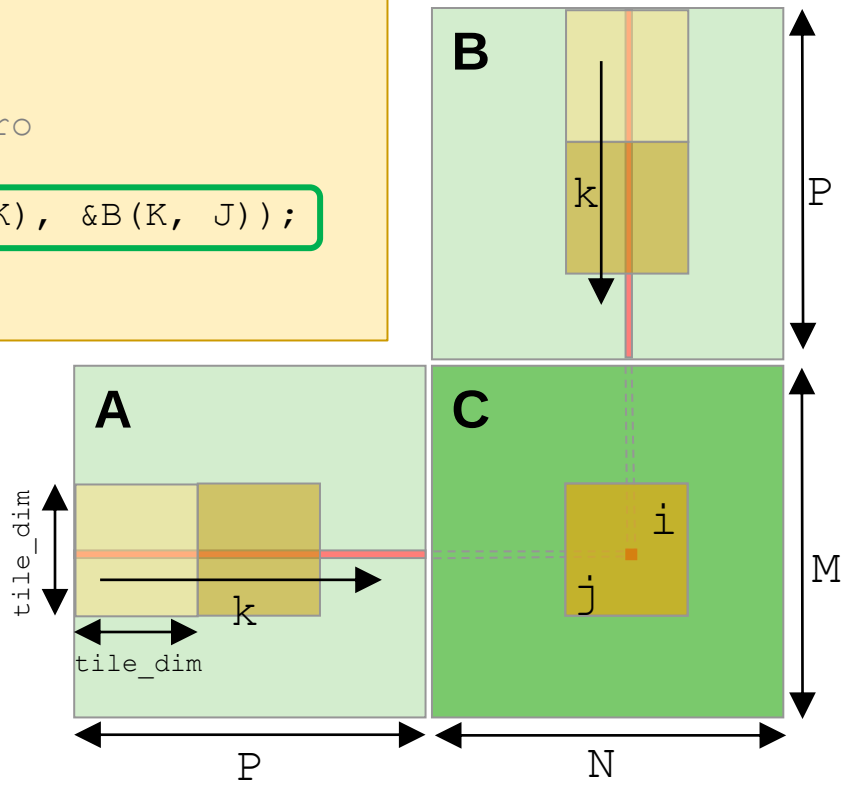
Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

Tiled Matrix Multiplication (II)

- Tiled implementation operates on submatrices (tiles or blocks) that fit fast RAMs (cache, scratchpad, RF)

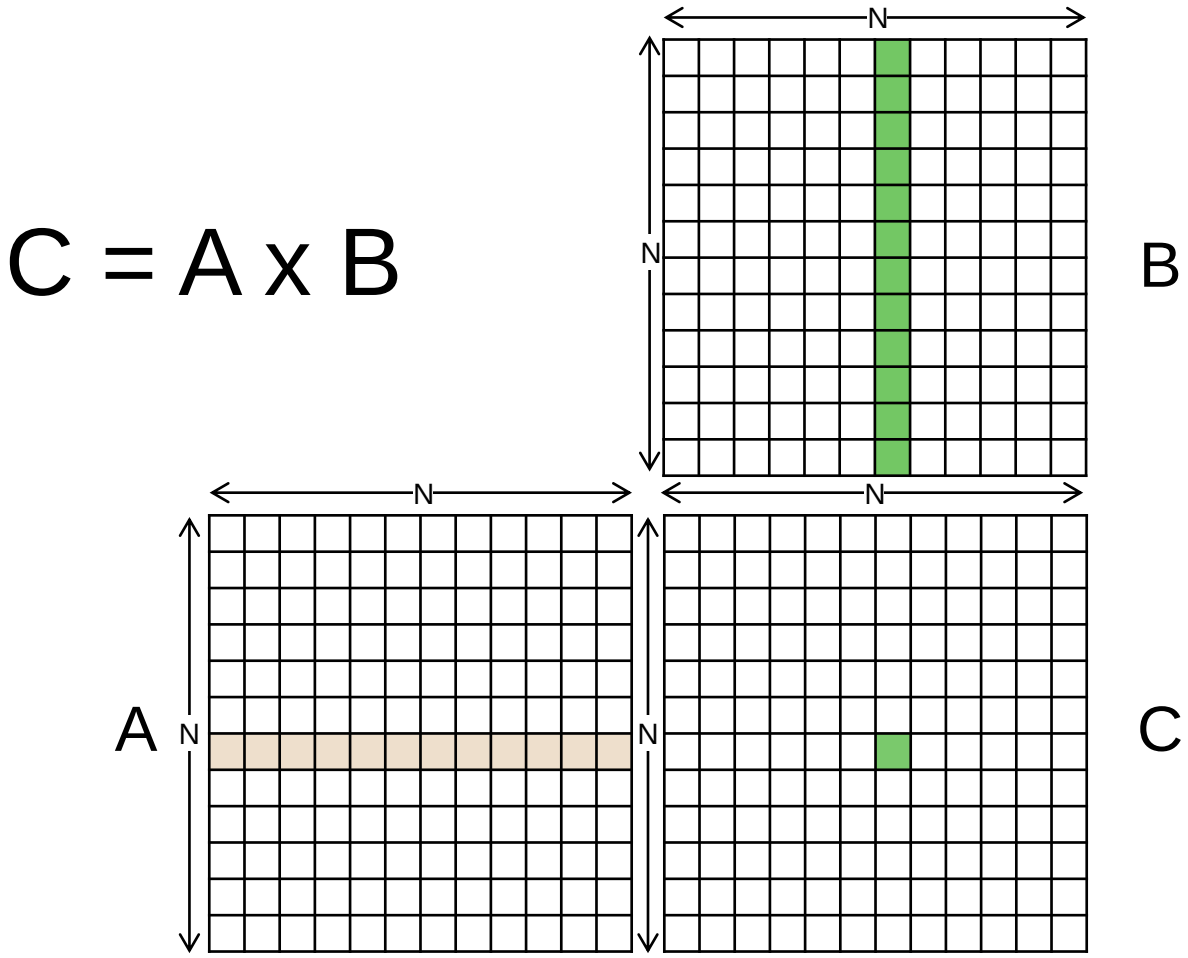
```
#define A(i,j) matrix_A[i * P + j]
#define B(i,j) matrix_B[i * N + j]
#define C(i,j) matrix_C[i * N + j]

for (I = 0; I < M; I += tile_dim){
    for (J = 0; J < N; J += tile_dim){
        Set_to_zero(&C(I, J)); // Set to zero
        for (K = 0; K < P; K += tile_dim)
            Multiply_tiles(&C(I, J), &A(I, K), &B(K, J));
    }
}
```

Multiply small submatrices (tiles or blocks) of size `tile_dim x tile_dim`



Lam+, "The cache performance and optimizations of blocked algorithms," ASPLOS 1991. https://doi.org/10.1145/106972.106981
Bansal+, "Chapter 15 - Fast Matrix Computations on Heterogeneous Streams," in "High Performance Parallelism Pearls", 2015. https://doi.org/10.1016/B978-0-12-803819-2.00011-2
Kirk & Hwu, "Chapter 5 - Performance considerations," in "Programming Massively Parallel Processors (Third Edition)", 2017. https://doi.org/10.1016/B978-0-12-811986-0.00005-4

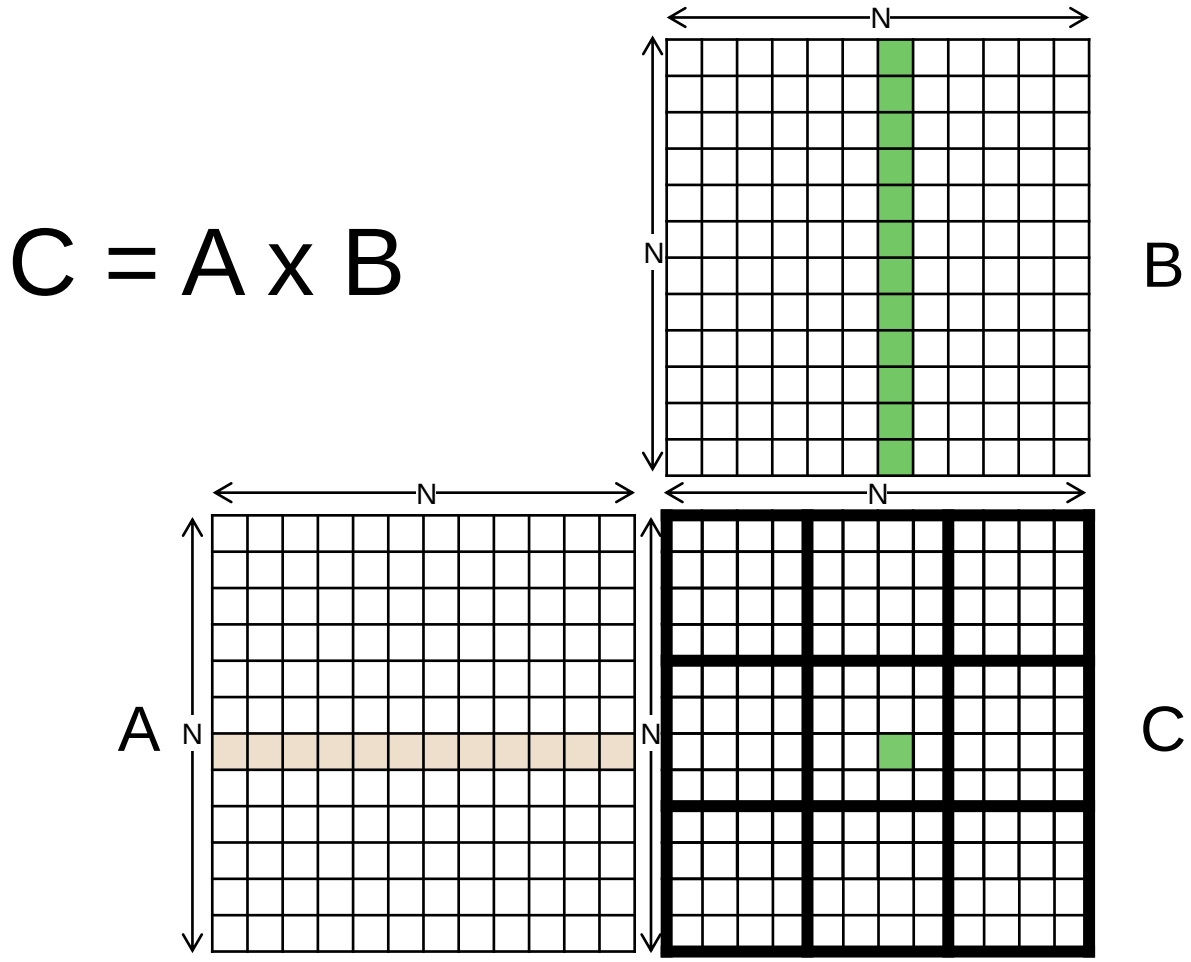
Example: Matrix-Matrix Multiplication (I)



Slide credit: Izzat El Hajj

Example: Matrix-Matrix Multiplication (II)

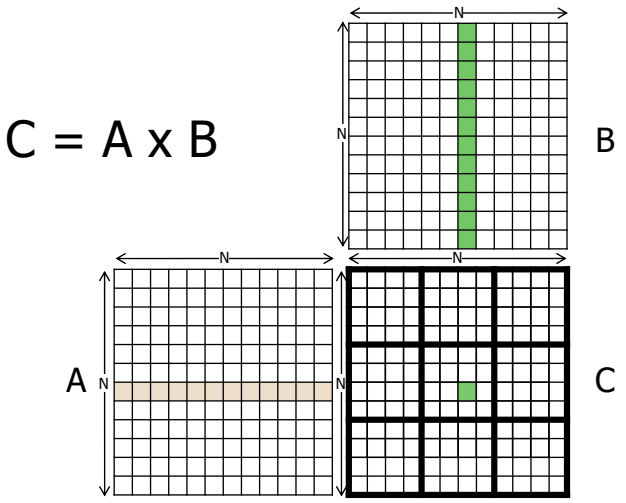
Parallelization approach: assign one thread to each element in the output matrix (C)



Slide credit: Izzat El Hajj

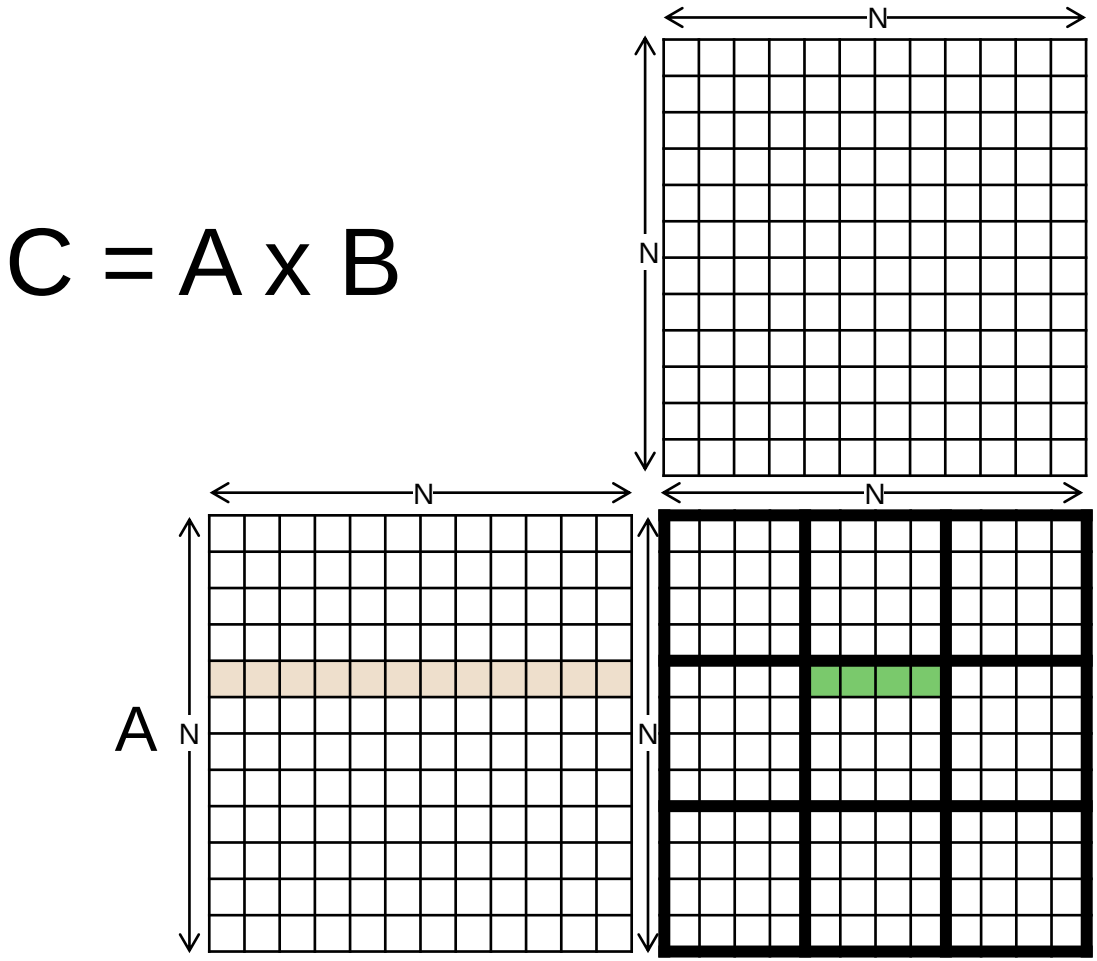
Example: Matrix-Matrix Multiplication (III)

```
__global__ void mm_kernel(float* A, float* B, float* C, unsigned int N) {

    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

    float sum = 0.0f;
    for(unsigned int i = 0; i < N; ++i) {
        sum += A[row*N + i]*B[i*N + col];
    }
    C[row*N + col] = sum;

}
```



Slide credit: Izzat El Hajj

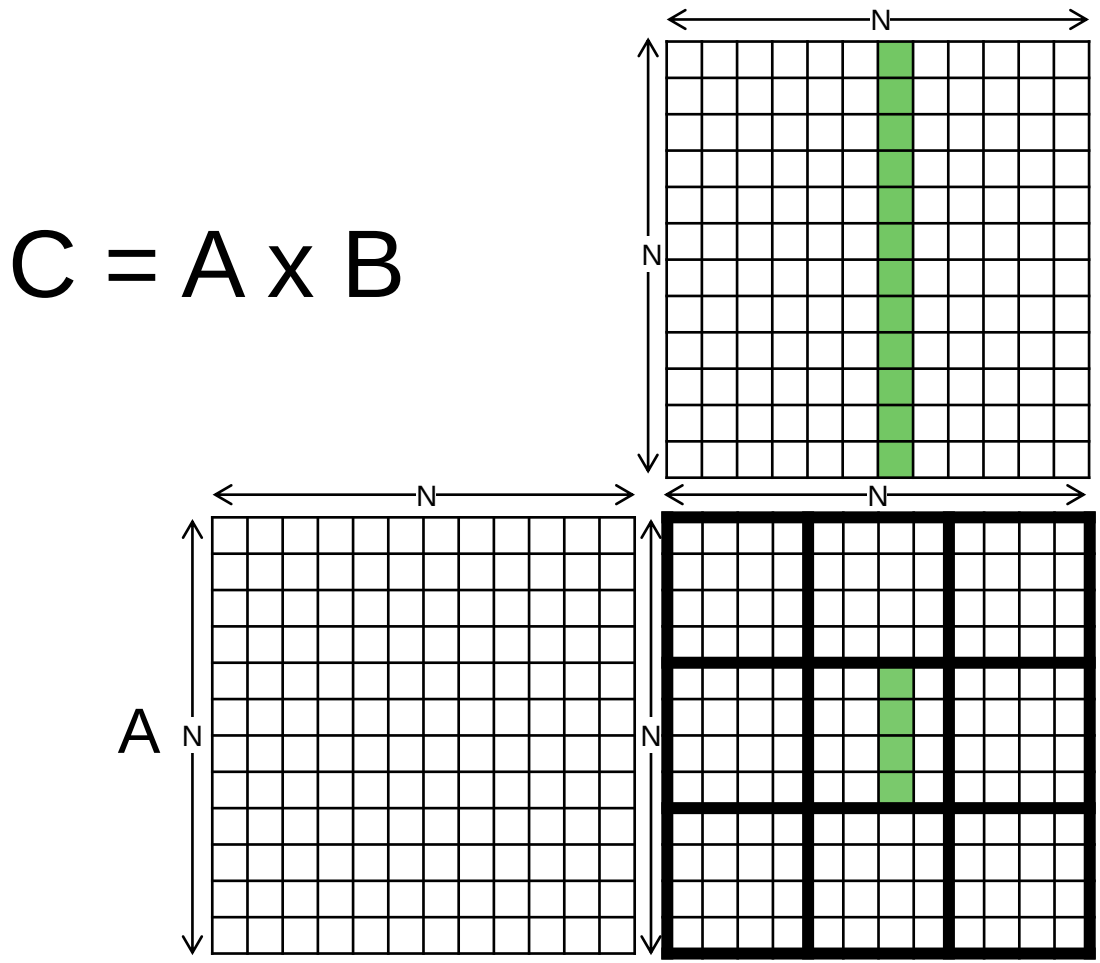
Reuse in Matrix-Matrix Multiplication (I)



Some of the threads in the same thread block use the same input data

Slide credit: Izzat El Hajj

Reuse in Matrix-Matrix Multiplication (II)



Some of the threads in the same thread block use the same input data

Slide credit: Izzat El Hajj

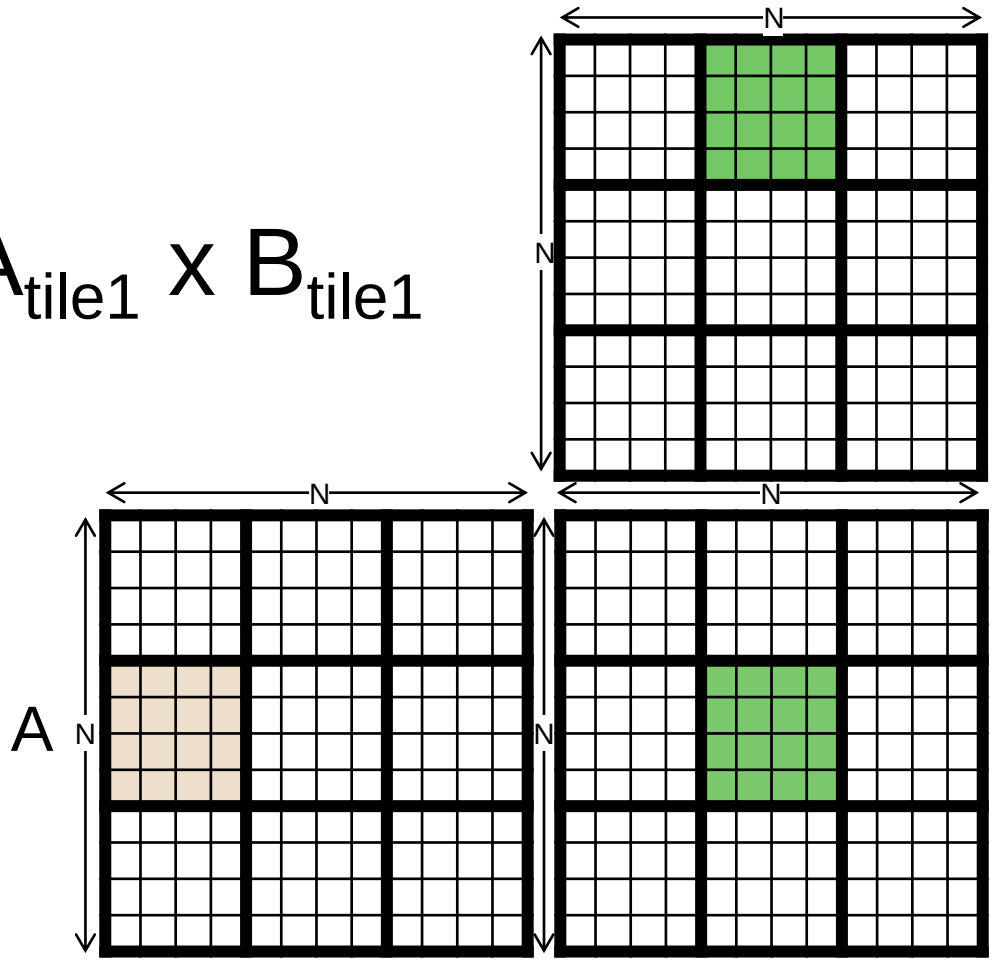
Reuse in Matrix-Matrix Multiplication (III)

- Sometimes, we are lucky:
  - The thread finds the data in the L1 cache because it was recently loaded by another thread
- Sometimes, we are not lucky:
  - The data gets evicted from the L1 cache before another thread tries to load it
- Solution:
  - Let the threads work together to load part of the data and ensure that all threads that need it use it before loading more data
  - Use shared memory to ensure data stays close
  - Optimizing called tiling because divides input to tiles

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (I)

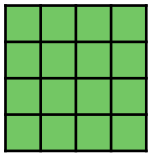
$C_{tile} = A_{tile1} \times B_{tile1}$



Step 1: Load the first tile of each input matrix to shared memory (each thread loads one element)

Slide credit: Izzat El Hajj

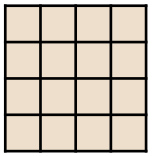
Tiled Matrix-Matrix Multiplication (II)

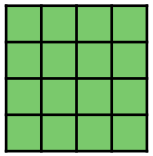


B_{tile1}

$C_{tile} \mathrel{+}= A_{tile1} \times B_{tile1}$

Step 2: Each thread computes its partial sum from the tiles in shared memory (threads wait for each other to finish)

A_{tile1} 

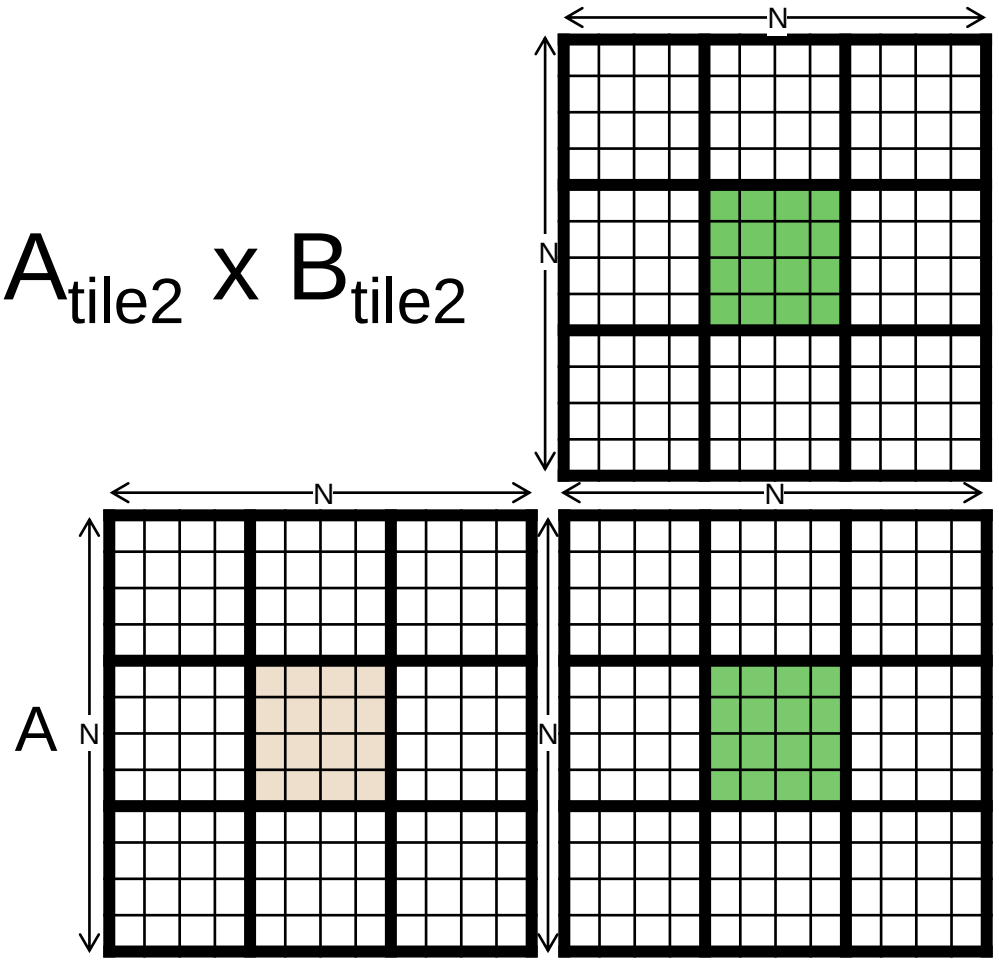


C_{tile}

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (III)

$C_{tile} \mathrel{+}= A_{tile2} \times B_{tile2}$

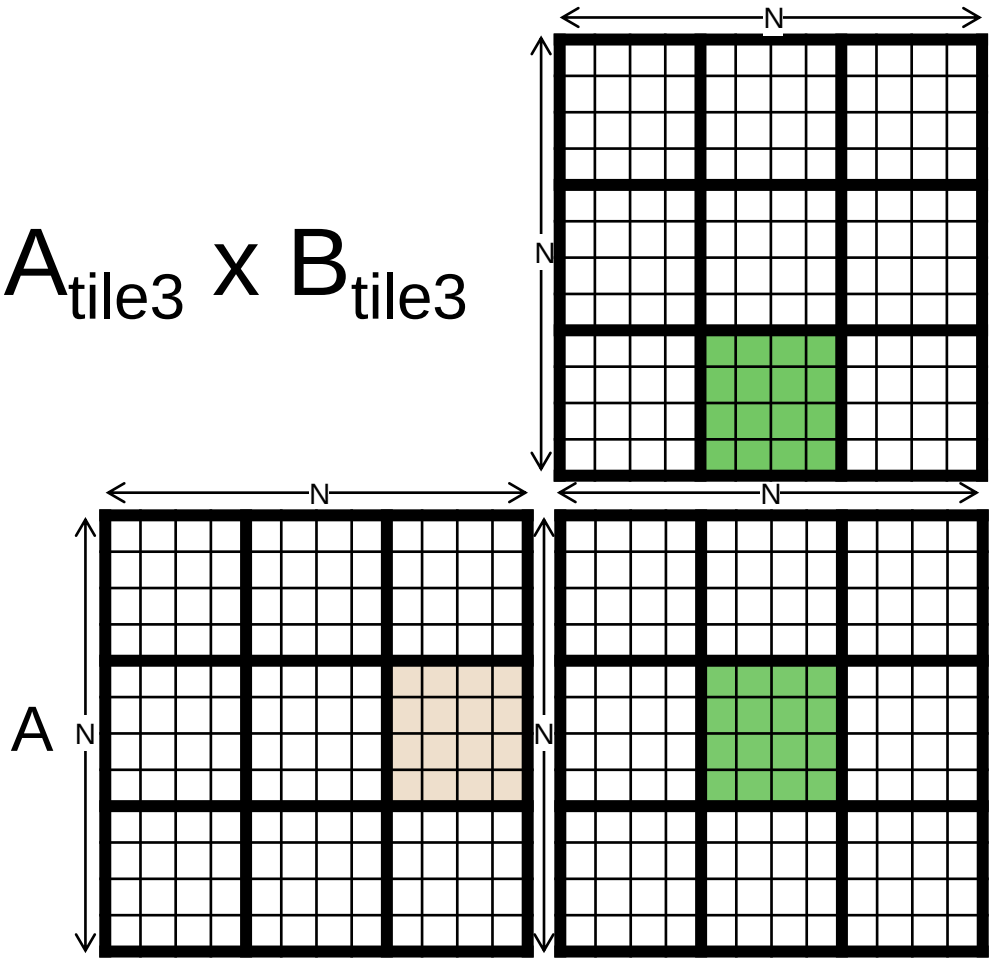


…accumulate the second tile

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (IV)

$C_{tile} \mathrel{+}= A_{tile3} \times B_{tile3}$



…and accumulate the third tile

Slide credit: Izzat El Hajj

Tiled Matrix-Matrix Multiplication (V)

```
__shared__ float A_s[TILE_DIM][TILE_DIM];
__shared__ float B_s[TILE_DIM][TILE_DIM];

unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;

float sum = 0.0f;

for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {

    // Load tile to shared memory
    A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];
    __syncthreads();

    // Compute with tile
    for(unsigned int i = 0; i < TILE_DIM; ++i) {
        sum += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
    }
    __syncthreads();

}

C[row*N + col] = sum;
```

Declare arrays in shared memory

Threads wait for each other to finish loading before computing

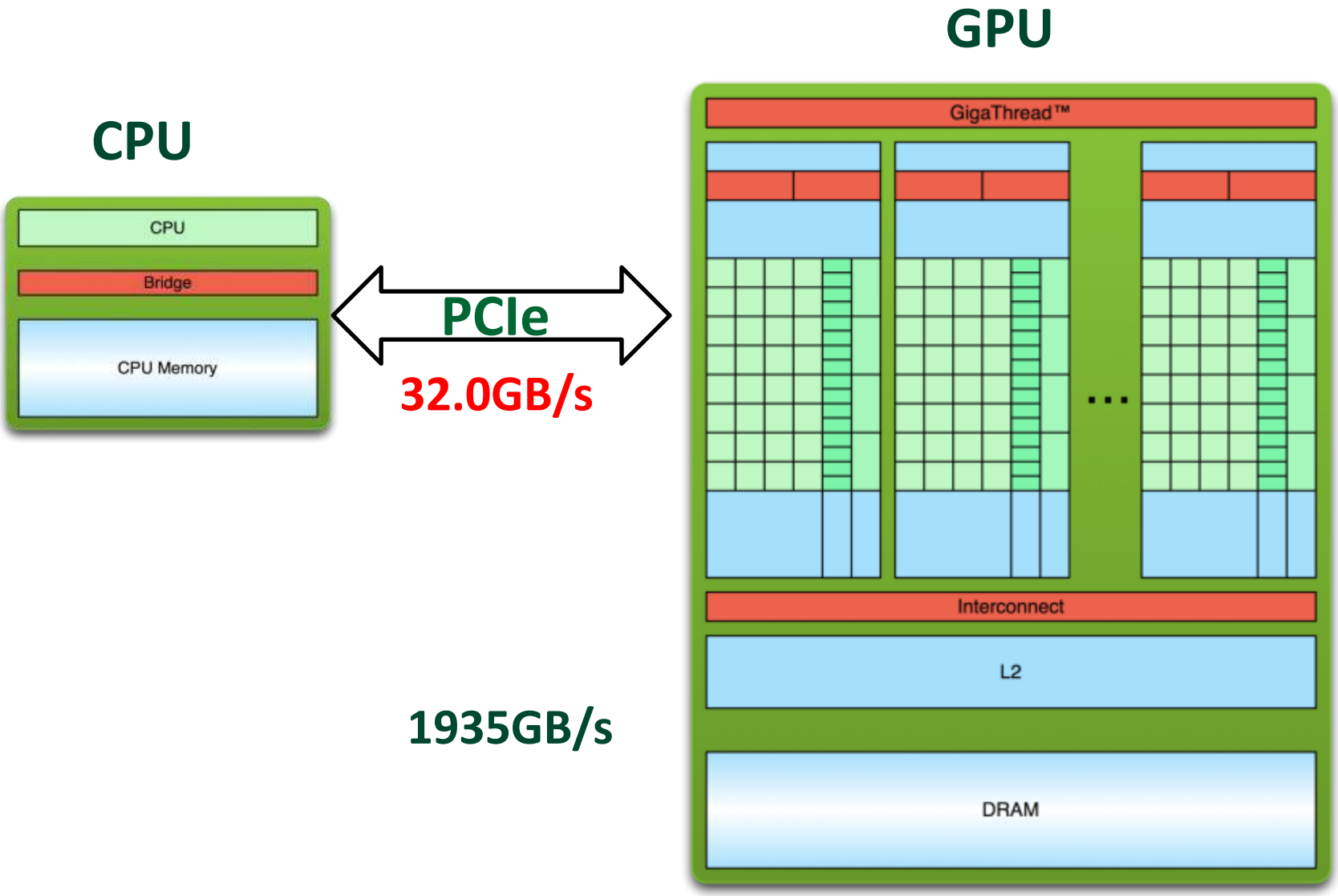
Threads wait for each other to finish computing before loading

Slide credit: Izzat El Hajj

State-of-the-art CPU GPU and FPGA

	Cores (Threads)	TFLOPS	Memory Size (Bandwidth)	PCIe	Network
CPU (AMD Threadripper 3995WX)	64 (128)	2.8 (FP32), 1.4 (FP64)	512GB (80GB/s)	32.0GB/s (PCIe 4.0 X16)	No
GPU (Nvidia A100)	8192 (128K)	19.5 (FP32), 9.7 (FP64), 156 (FP32, Tensor), 312 (FP16, Tensor)	40/80GB (1935GB/s)	32.0GB/s (PCIe 4.0 X16)	No
FPGA (U280)	9,024 (25x18 MULs)	1.8 (FP32)	40GB (460GB/s)	16.0GB/s (PCIe 4.0 X8)	Yes

Limitation of GPU



Limitation of GPU

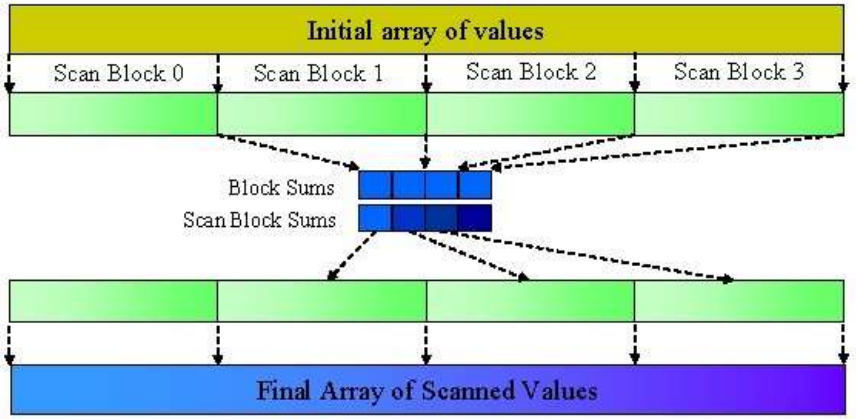
Serial Code of Prefix sum:

```
// Fills prefix sum array
void fillPrefixSum(int arr[], int n, int
prefixSum[])
{ prefixSum[0] = arr[0];
  // Adding present element
  for (int i = 1; i < n; i++)
   prefixSum[i] = prefixSum[i-1] + arr[i]; }
```

GPU Code of Prefix sum:

Multi-pass (ISSUE)

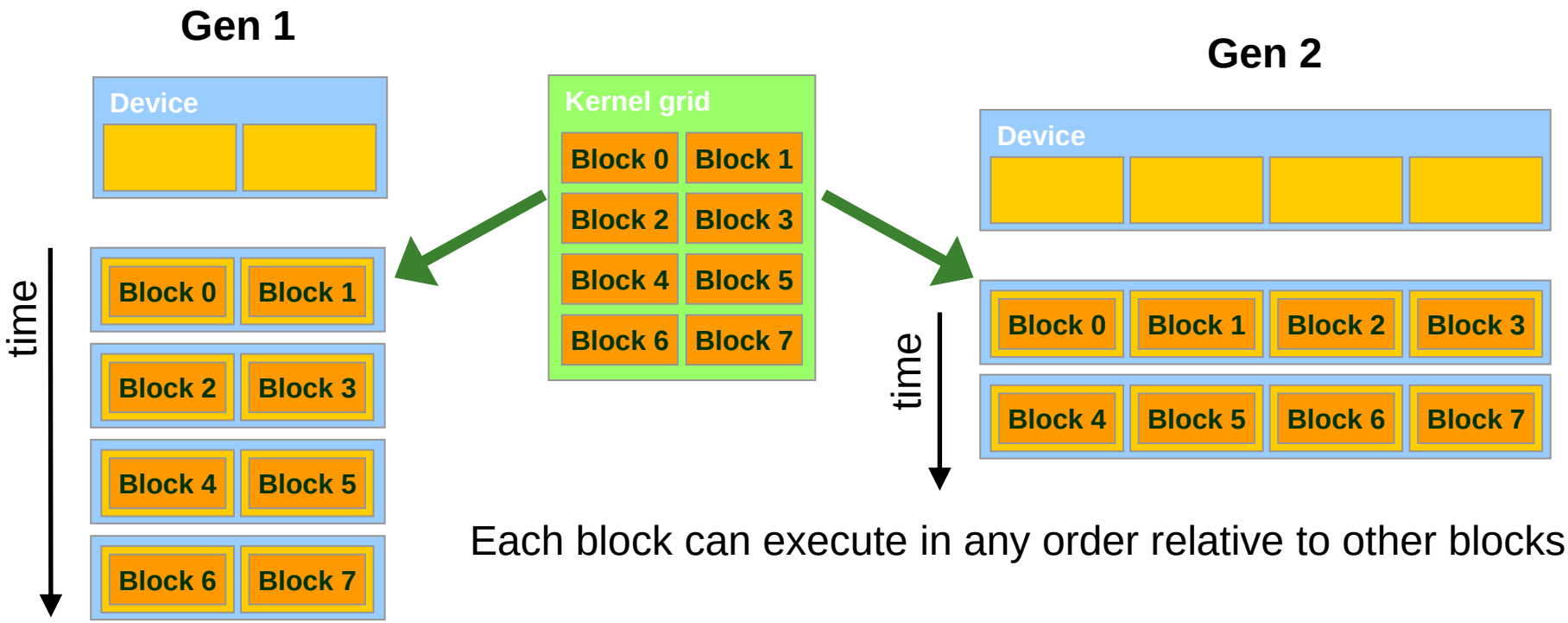
Alterations for Arbitrary Sized Arrays



- Divide the large array into blocks that can be scanned by a single thread block
- Scan each block and write the total sums of each block to another array of blocks
- Scan the block sums, generating an array of block increments
- The result is added to each of the element of their respective block

Nvidia's Success: Transparent Scalability

- Hardware is free to schedule thread blocks



Each block can execute in any order relative to other blocks.

The CUDA code stays the same and enjoys performance improvement while GPU hardware evolves.

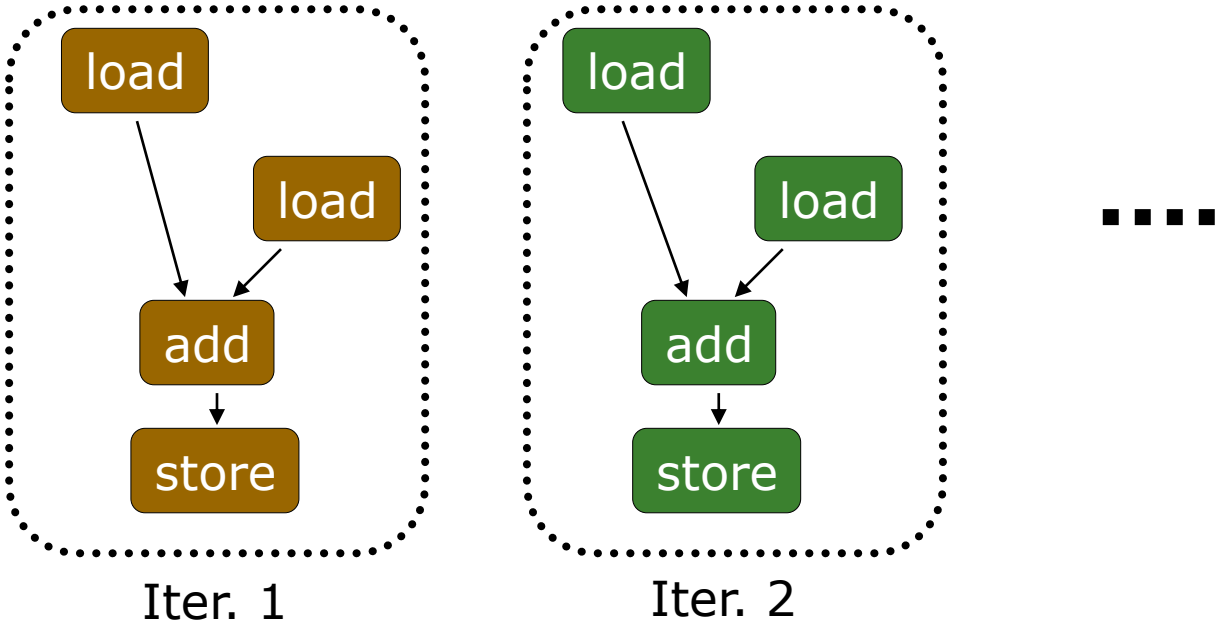
Slide credit: Hwu & Kirk

Key Messages:

- Programming model is the key success of Nvidia, rather than the GPU itself.
- GPU has an order of magnitude higher memory bandwidth and compute power than CPU.
- Offloading a task to GPU pays off only when the task has enough compute intensity.
- AI task needs compute-intensive accelerators, e.g., GPU and AI processor.

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Realization: Each iteration is independent

This programming model (software) is called:
SPMD: Single Program Multiple Data
Executed on a SIMT machine (hardware)
Single Instruction Multiple Thread

A GPU is a SIMD (SIMT) Machine

- Except it is **not** programmed using SIMD instructions

- It is programmed using threads (SPMD programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into a **warp (wavefront)** by the hardware
  - A warp is essentially a SIMD operation formed by hardware!

SIMD vs. SIMT Execution Model

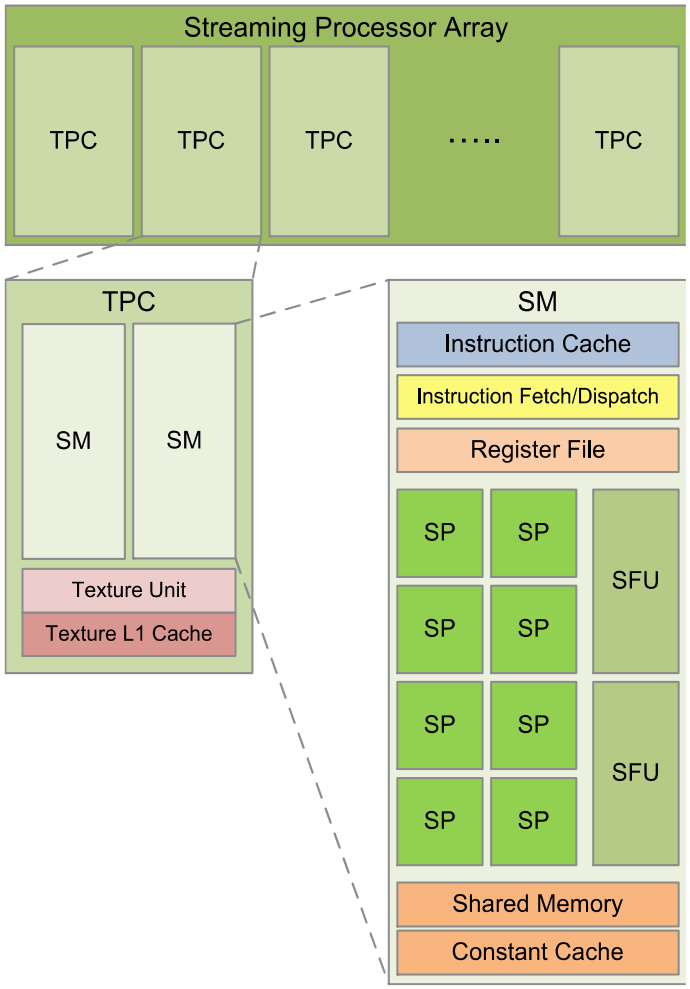
- **SIMD**: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently on any type of scalar pipeline
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

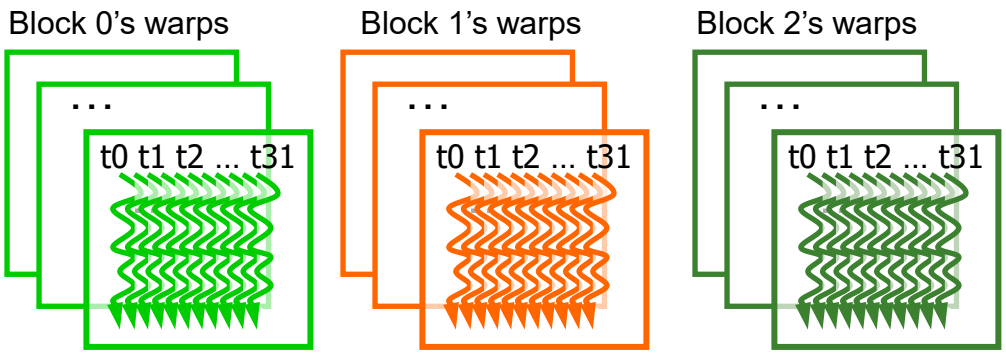
Brief Review of GPU Architecture (I)

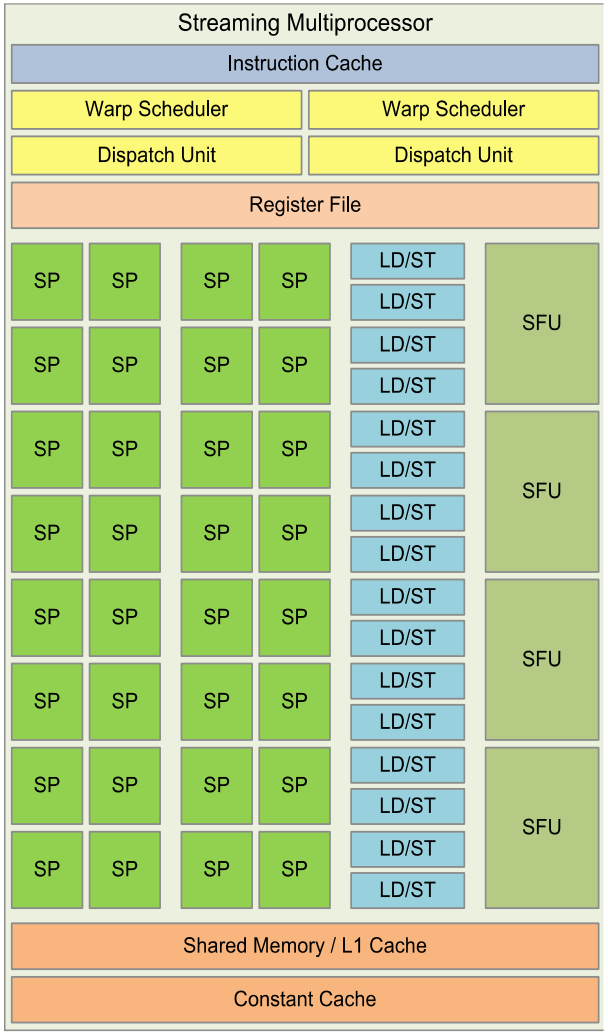
- Streaming Processor Array
  - Tesla architecture (G80/GT200)



Brief Review of GPU Architecture (II)

- Streaming Multiprocessors (SM)
  - Streaming Processors (SP)
- Blocks are divided into warps
  - SIMD unit (32 threads)





NVIDIA Fermi architecture

Brief Review of GPU Architecture (III)

- Streaming Multiprocessors (SM) or Compute Units (CU)
  - SIMD pipelines
- Streaming Processors (SP) or CUDA ”cores”
  - Vector lanes
- Number of SMs x SPs across generations
  - Tesla (2007): 30 x 8
  - Fermi (2010): 16 x 32
  - Kepler (2012): 15 x 192
  - Maxwell (2014): 24 x 128
  - Pascal (2016): 56 x 64
  - Volta (2017): 80 x 64

Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)

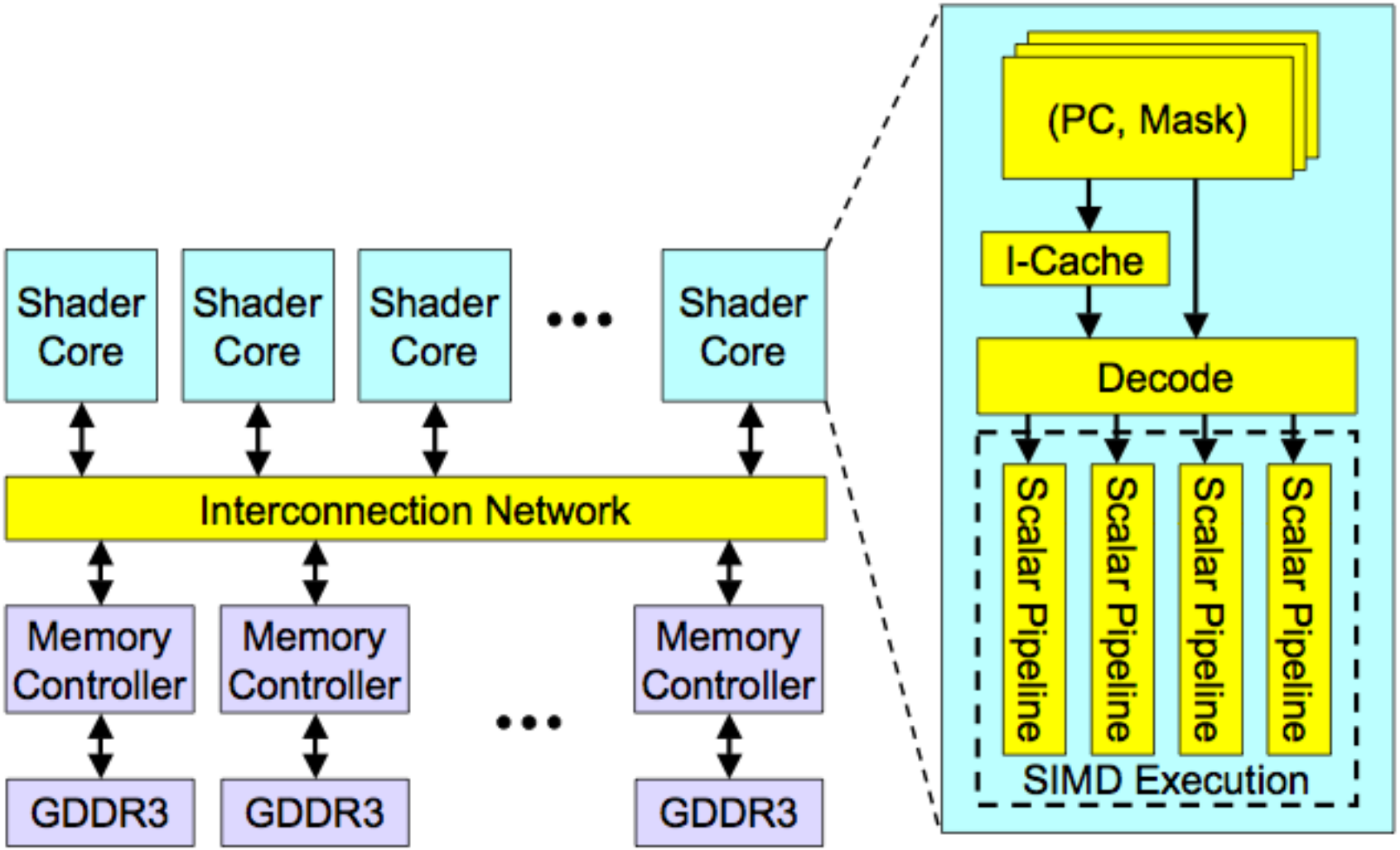
SIMD vs. SIMT Execution Model

- **SIMD**: A single sequential instruction stream of SIMD instructions → each instruction specifies multiple data inputs
  - [VLD, VLD, VADD, VST], VLEN

- **SIMT**: Multiple instruction streams of scalar instructions → threads grouped dynamically into warps
  - [LD, LD, ADD, ST], NumThreads

- Two Major SIMT Advantages:
  - Can treat each thread separately → i.e., can execute each thread independently (on any type of scalar pipeline) → MIMD processing
  - Can group threads into warps flexibly → i.e., can group threads that are supposed to *truly* execute the same instruction → dynamically obtain and maximize benefits of SIMD processing

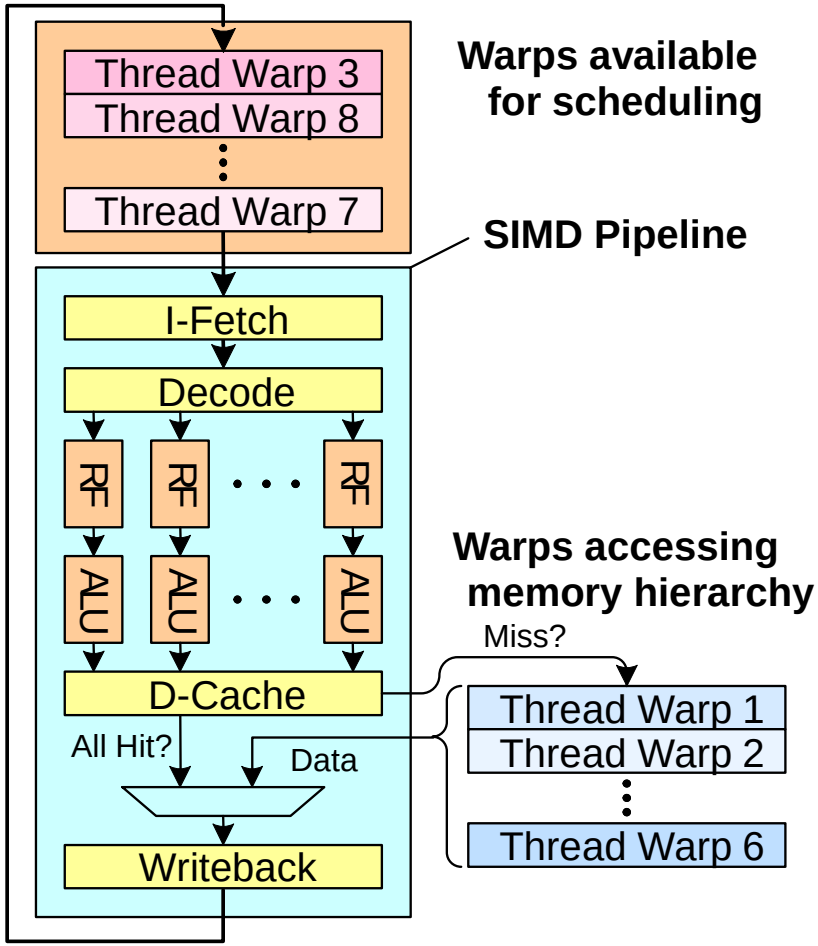
High-Level View of a GPU



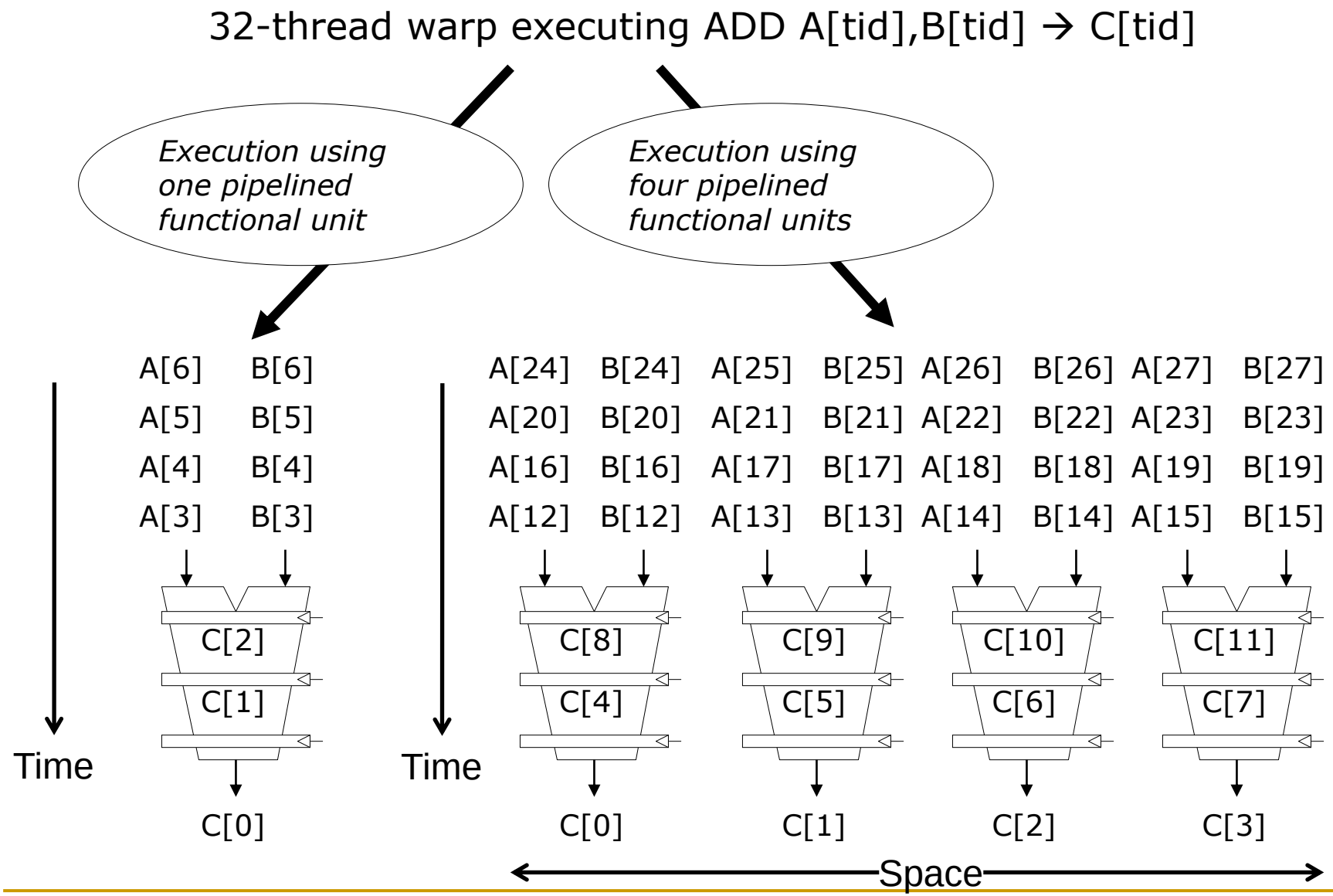
Lindholm et al., "NVIDIA Tesla: A Unified Graphics and Computing Architecture," IEEE Micro 2008.

Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - No interlocking: One instruction per thread in pipeline at a time.
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels

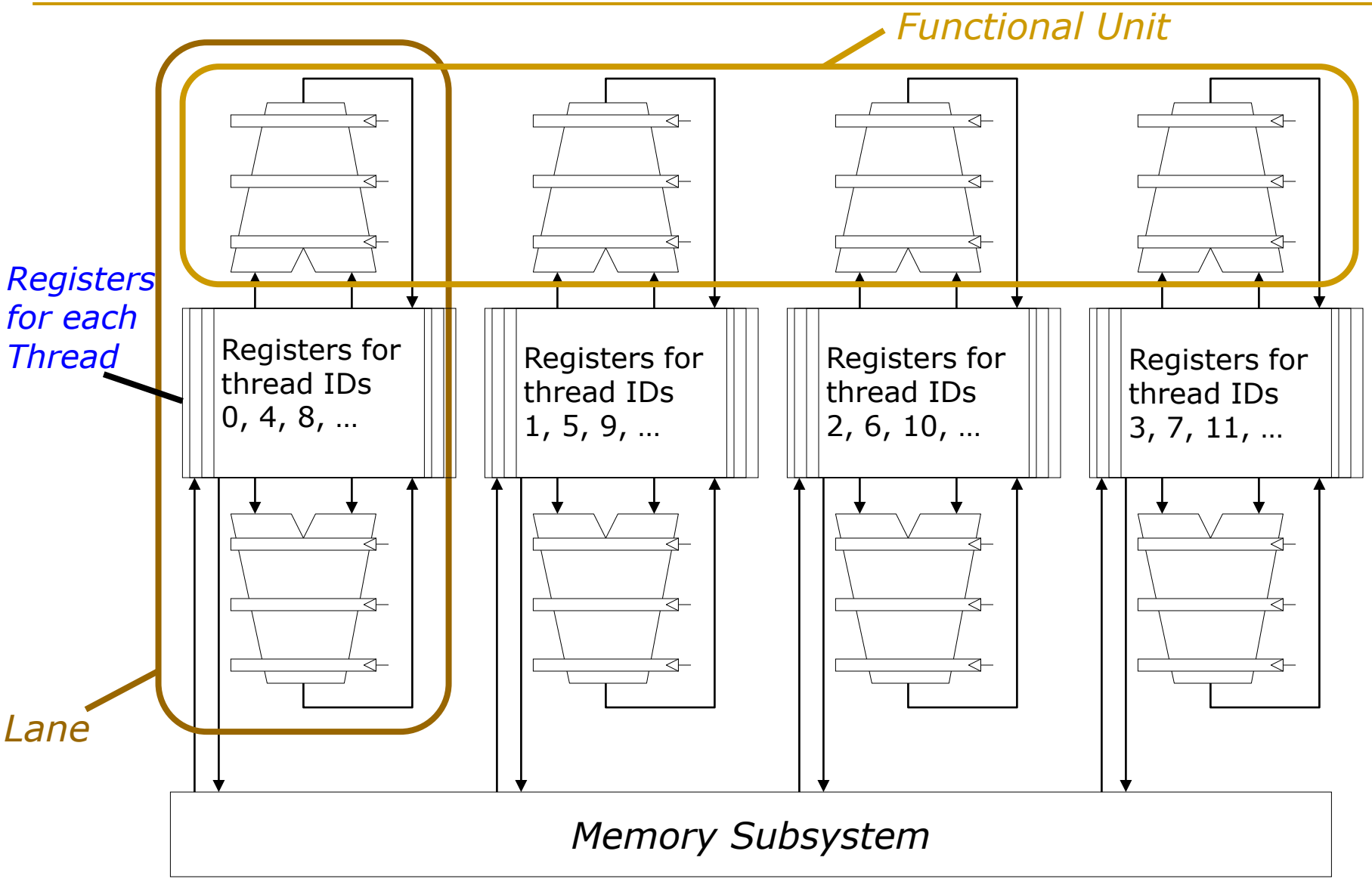


Warp Execution (Recall the Slide)



Slide credit: Krste Asanovic

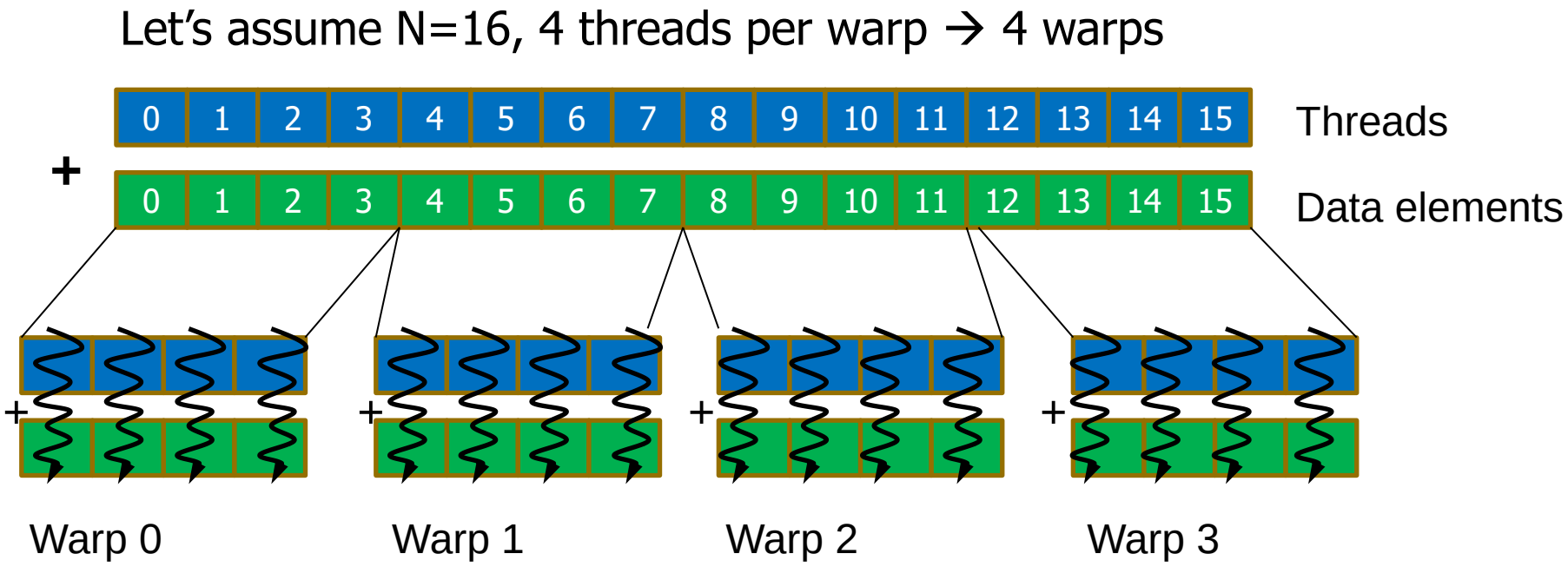
SIMD Execution Unit Structure



Slide credit: Krste Asanovic

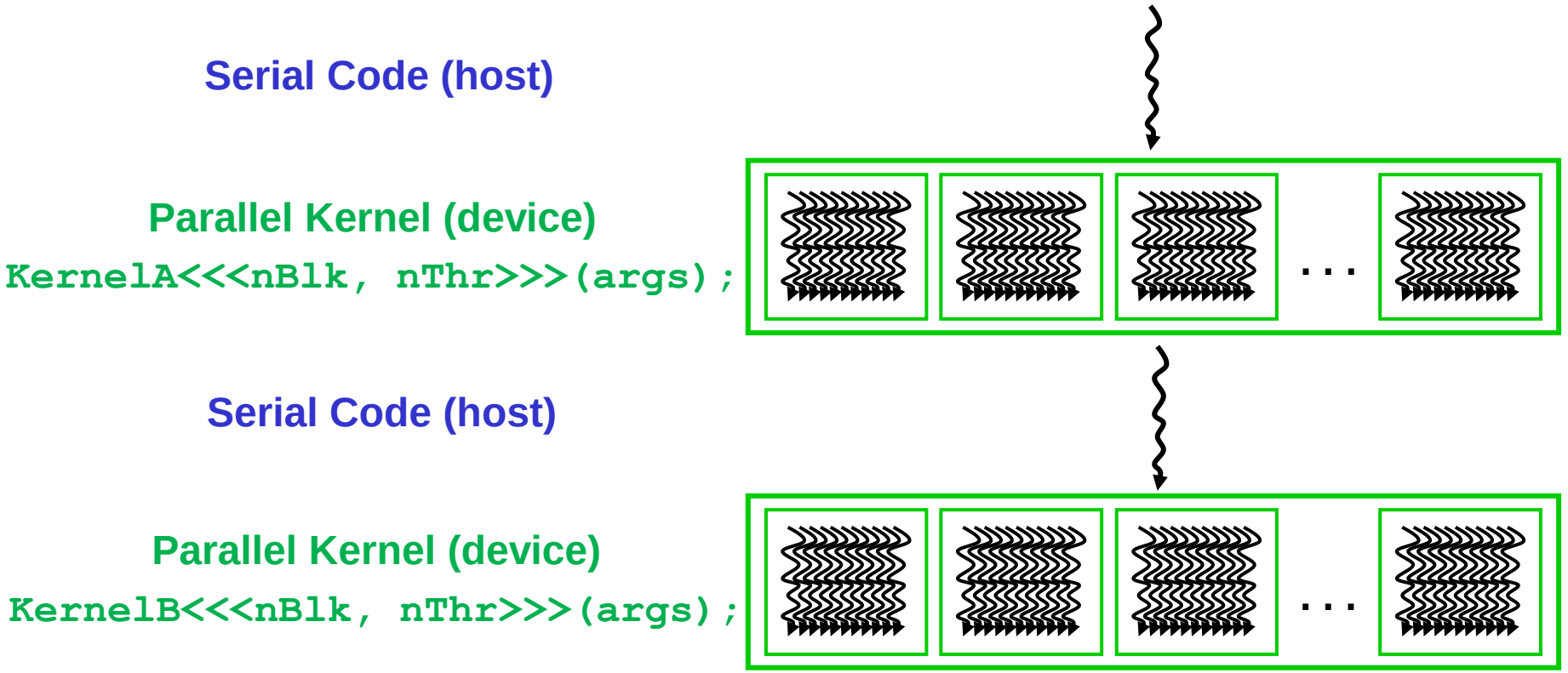
SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements



Warps *not* Exposed to GPU Programmers

- CPU threads and GPU kernels
  - Sequential or modestly parallel sections on CPU
  - Massively parallel sections on GPU: Blocks of threads

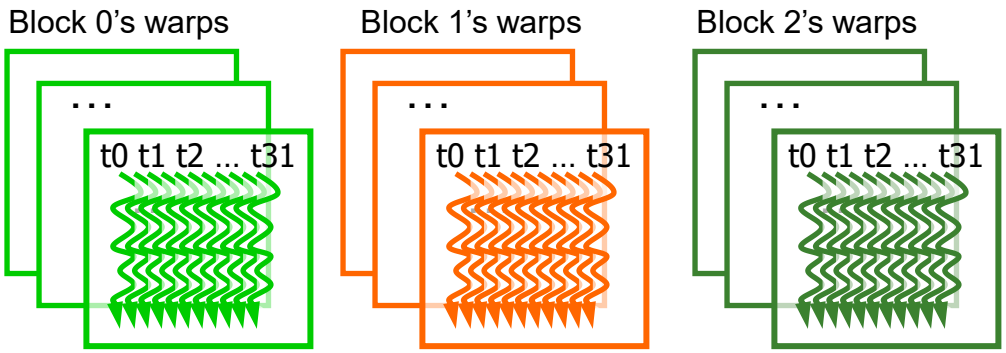


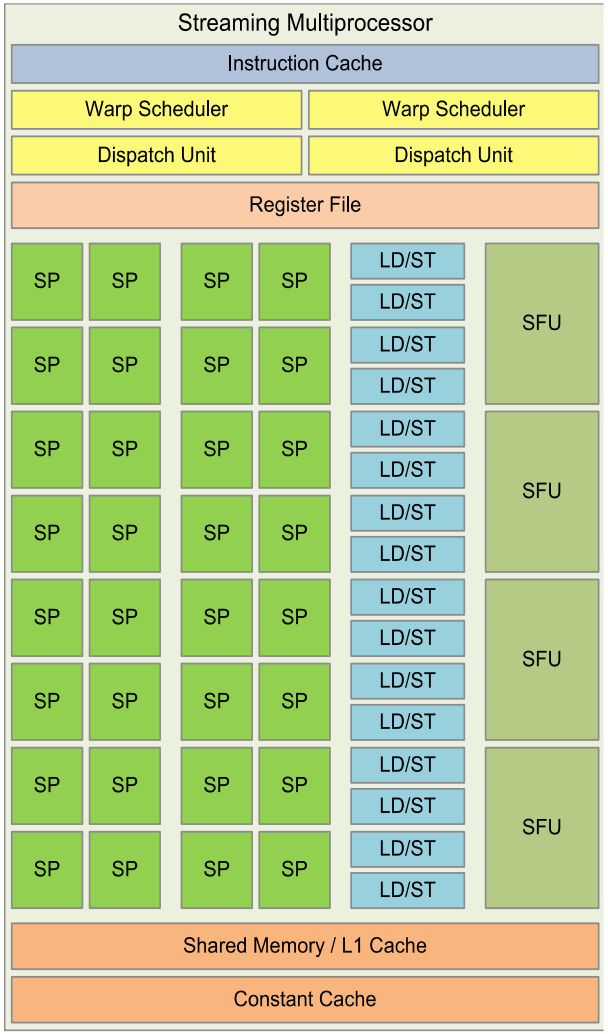
Slide credit: Hwu & Kirk

From Blocks to Warps

- GPU cores: SIMD pipelines
  - Streaming Multiprocessors (SM)
  - Streaming Processors (SP)

- Blocks are divided into warps
  - SIMD unit (32 threads)





NVIDIA Fermi architecture

SPMD

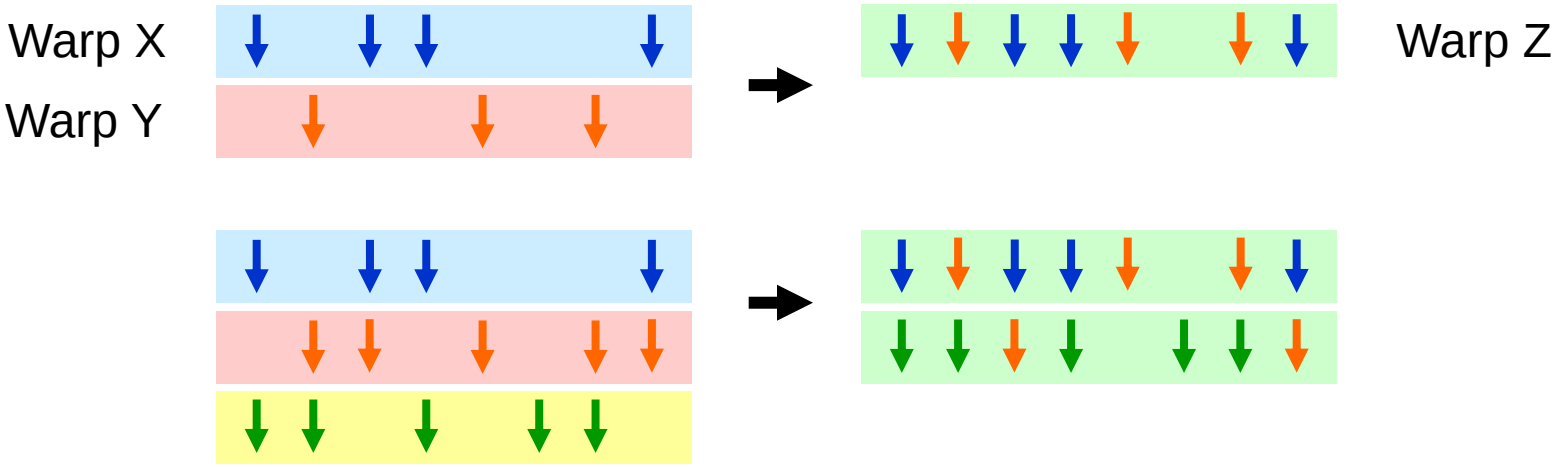
- Single procedure/program, multiple data
  - This is a **programming model** rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure 1) works on different data, 2) can execute a different control-flow path, at run-time
  - Many scientific applications are programmed this way and run on MIMD hardware (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD hardware

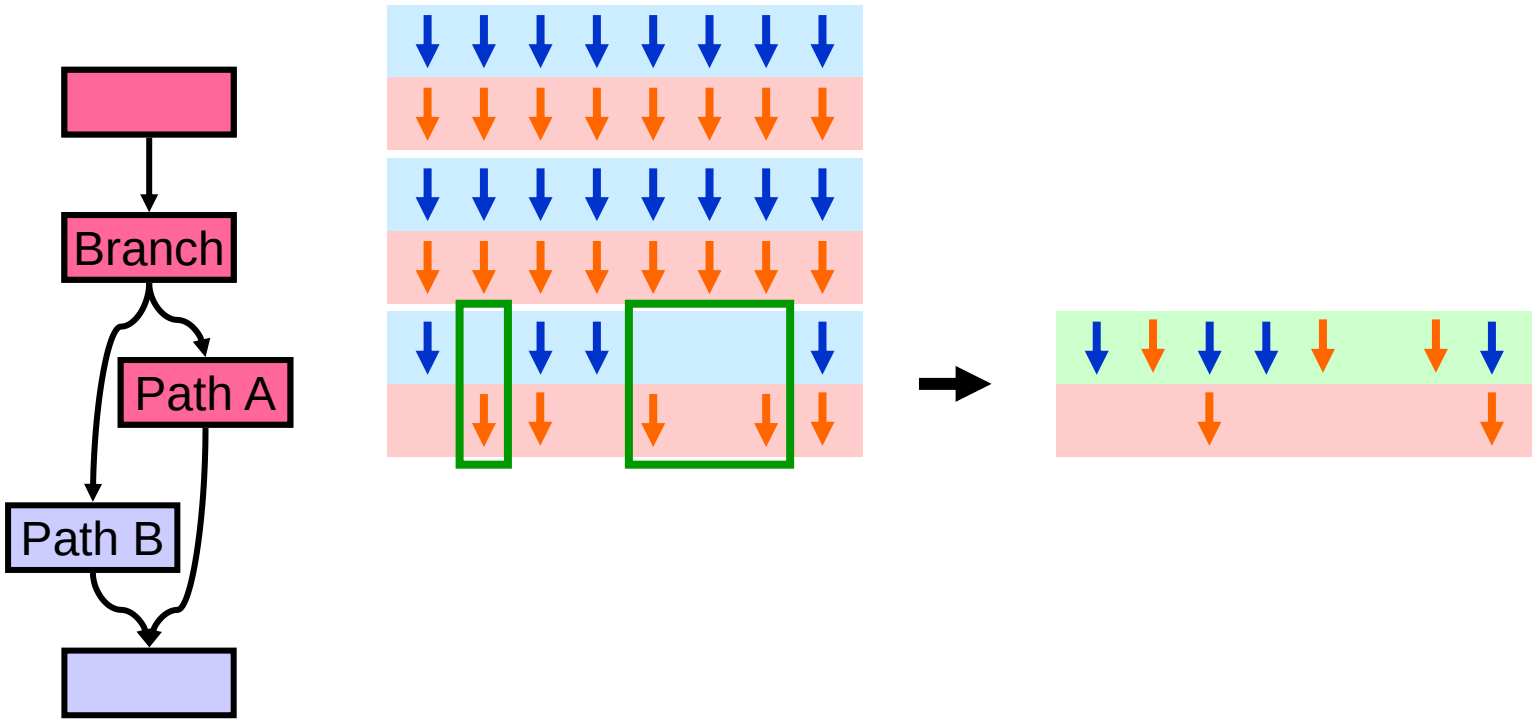
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)
- Form new warps from warps that are waiting
  - Enough threads branching to each path enables the creation of full new warps



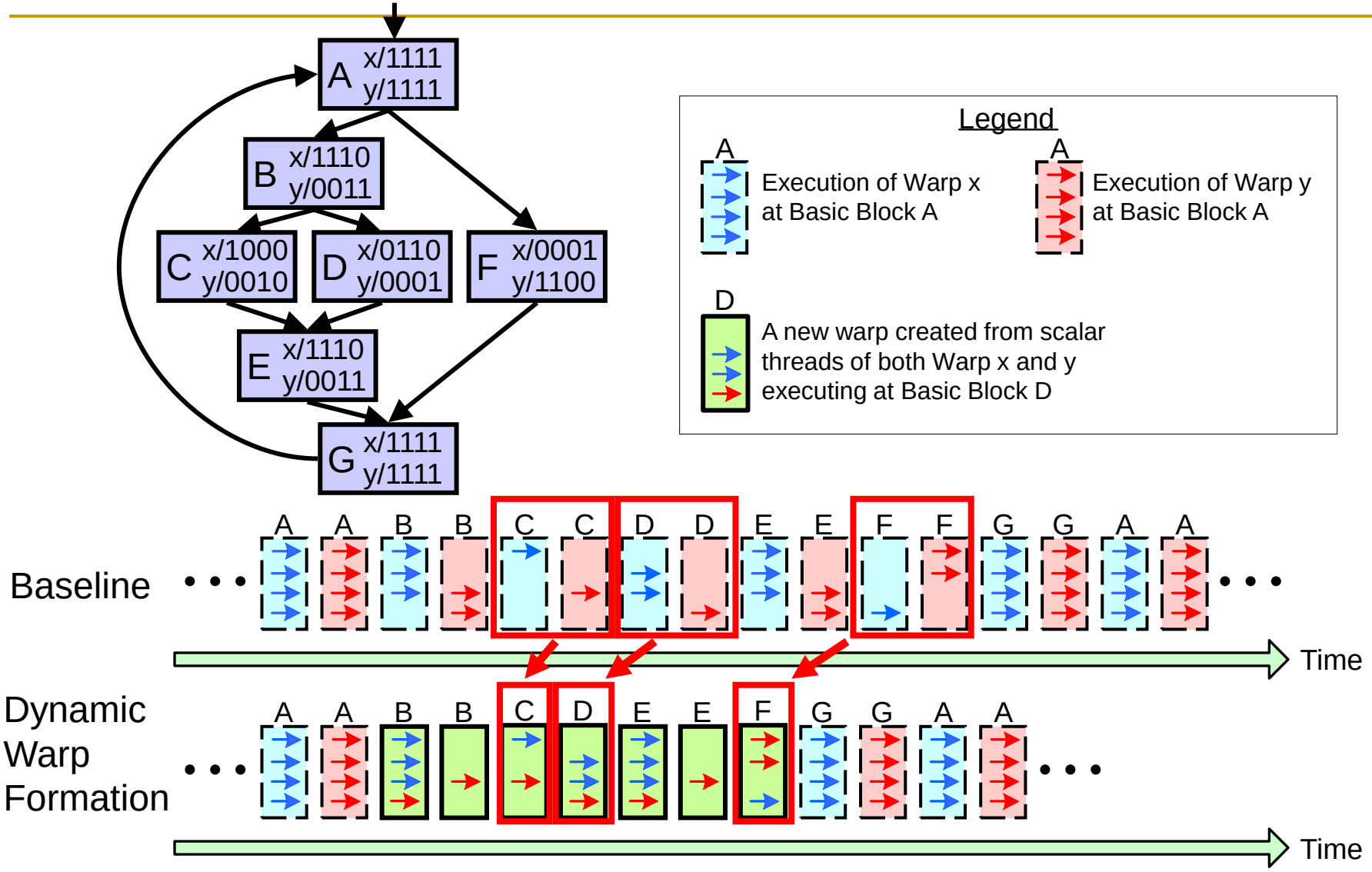
Dynamic Warp Formation/Merging

- Idea: Dynamically merge threads executing the same instruction (after branch divergence)

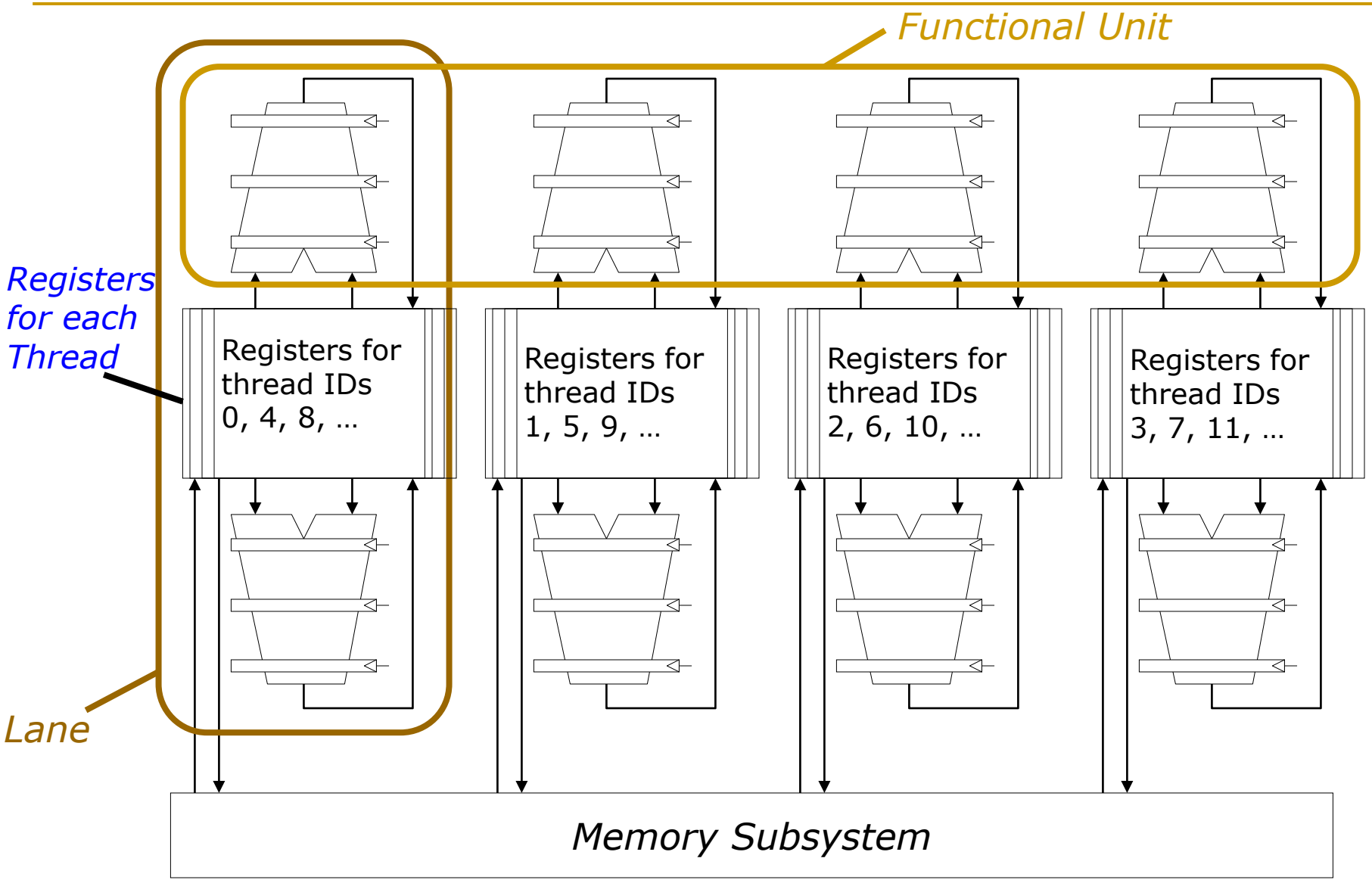


- Fung et al., "Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow," MICRO 2007.

Dynamic Warp Formation Example



Hardware Constraints Limit Flexibility of Warp Grouping



Slide credit: Krste Asanovic

Clarification of Some GPU Terms

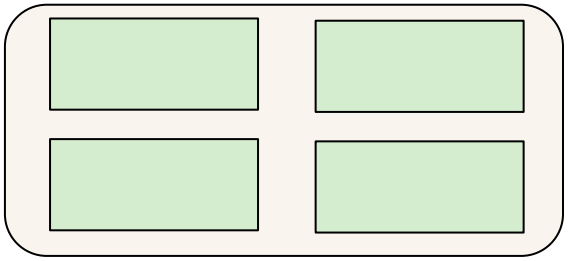
Generic Term	NVIDIA Term	AMD Term	Comments
Vector length	Warp size	Wavefront size	Number of threads that run in parallel (lock-step) on a SIMD functional unit
Pipelined functional unit / Scalar pipeline	Streaming processor / CUDA core	-	Functional unit that executes instructions for one GPU thread
SIMD functional unit / SIMD pipeline	Group of N streaming processors (e.g., N=8 in GTX 285, N=16 in Fermi)	Vector ALU	SIMD functional unit that executes instructions for an entire warp
GPU core	Streaming multiprocessor	Compute unit	It contains one or more warp schedulers and one or several SIMD pipelines

Programming Model vs. Hardware Execution Model

Hardware Programming Model

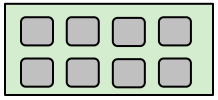
Programming Model

GPU



Thread blocks

Streaming Multi-processor



Thread block (s)

Wrap

CUDA core:

Thread

NVIDIA H100 Block Diagram

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

144 cores on the full GH100

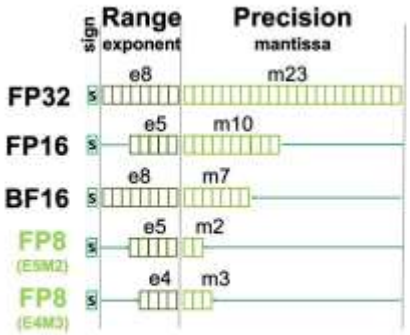
60MB L2 cache

NVIDIA H100 Core

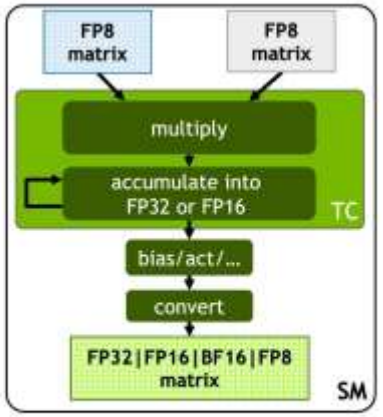
48 TFLOPS Single Precision*

24 TFLOPS Double Precision*

800 TFLOPS (FP16, Tensor Cores)*



Allocate 1 bit to either range or precision



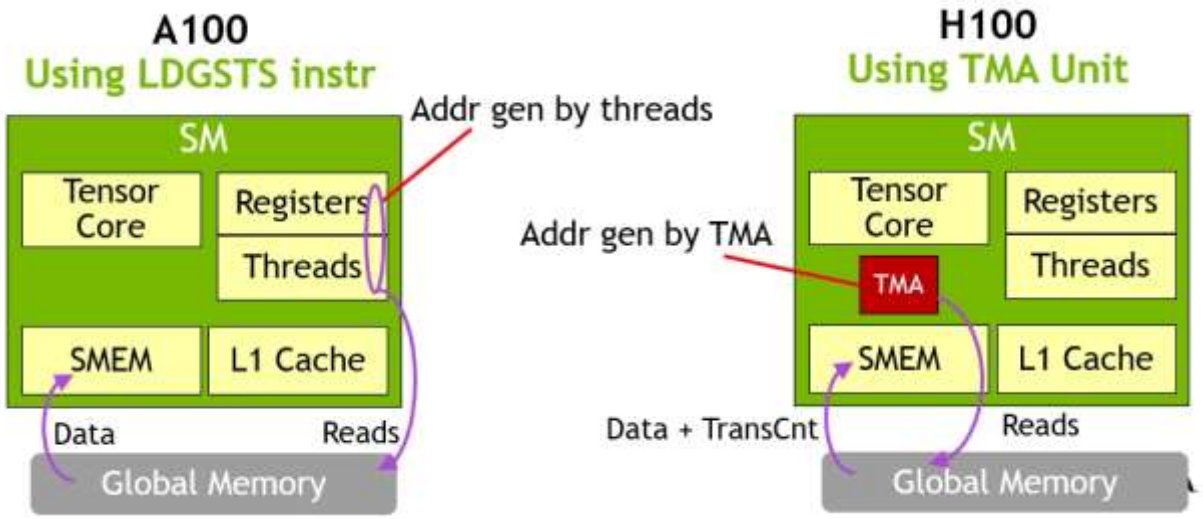
Support for multiple accumulator and output types

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

* Preliminary performance estimates

NVIDIA H100 Tensor Memory Accelerator

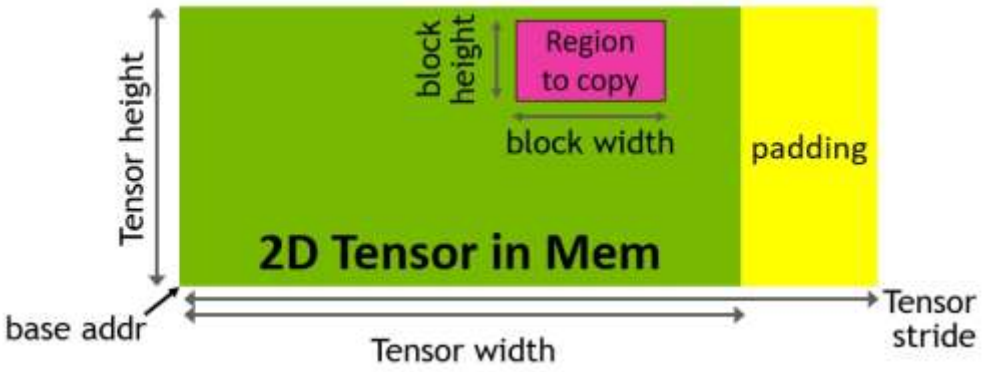
- Asynchronous memory copy with LDGSTS instruction vs. TMA



TMA unit reduces addressing overhead

A single thread per warp issues the TMA operation

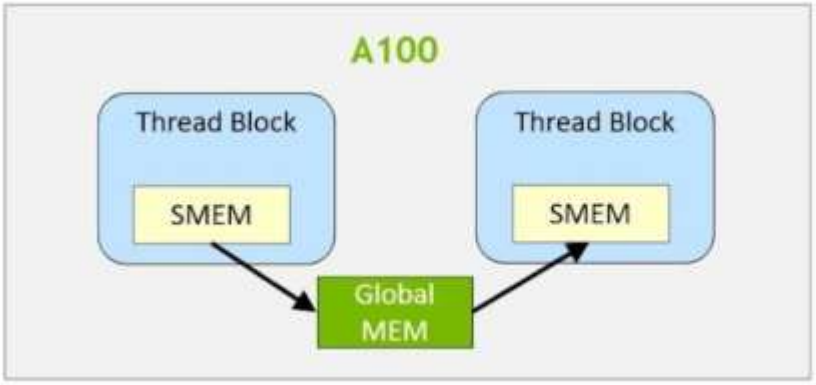
Support for different tensor layouts (1D-5D)

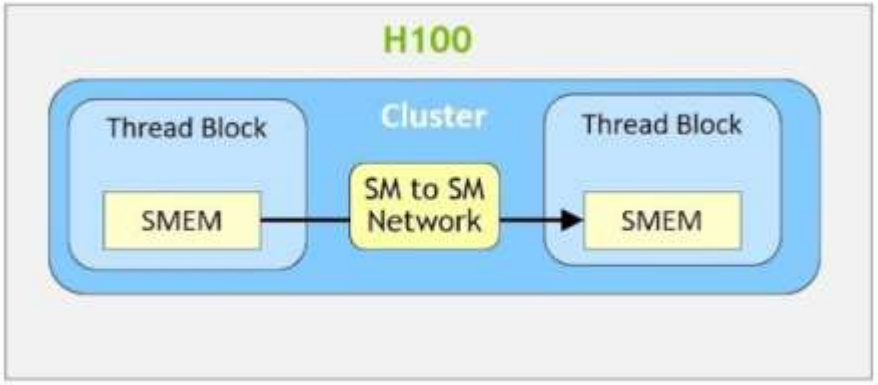


https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

NVIDIA H100 Distributed Shared Memory

- Shared memory virtual address space distributed across the blocks of a cluster
- Load, store, and atomic operations to other SM's shared memory





Thread block clusters and distributed shared memory (DSMEM) are leveraged via `cooperative_groups` API

TMA unit supports copies across thread blocks in a cluster

Asynchronous transaction barriers

https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/