

Advanced AI Computing and Systems

Lecture I: Processor Evolution — From CPU to GPU

Prof. Kai Chen

Syllabus and Readings

- Key Question
 - Why did AI computing move from CPU-centric systems to GPU-centric systems?
- Today's Path
 1. [Review] How code runs and completes computation?
 2. [Review] What a host machine looks like?
 3. [Core] CPU vs GPU microarchitecture.
 4. [Concept] Parallel programming models.
 5. [Core] Performance model for AI workloads.
- Readings
 - Hennessy & Patterson, [A New Golden Age for Computer Architecture](#), CACM 2019
 - Dally, Keckler & Kirk, [Evolution of the Graphics Processing Unit](#), IEEE Micro 2021
 - Williams, Waterman & Patterson, [Roofline: An Insightful Visual Performance Model for Multicore Architectures](#), CACM 2009
 - Dao et al., [FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness](#), NeurIPS 2022

Transformation Hierarchy of Computing

Workflow of Once Computation

- When we write
 - $Y = \text{Model}(X)$, what actually happens?
- Step 1: Design an algorithm *Model* to solve a problem, with input data *X*.
- Step 2: A program expresses the algorithm.
- Step 3: **System software** turns the program into an executable, parallel, and robust task.
- Step 4: **Operating system** schedules computing tasks and deploys them onto hardware for executing.
- Step 4: **Hardware** executes instructions and moves data to get results.
- Step 5: Computation is completed and result *Y* is returned to the application.

Transformation Hierarchy of Computing

Workflow of Once Computation

- When we write
 - $Y = \text{Model}(X)$, what actually happens?
- Step 1: Design an algorithm *Model* to solve a problem, with input data *X*.
- Step 2: A program expresses the algorithm.
- Step 3: **System software** turns the program into an executable, parallel, and robust task.
- Step 4: **Operating system** schedules computing tasks and deploys them onto hardware for executing.
- Step 4: **Hardware** executes instructions and moves data to get results.

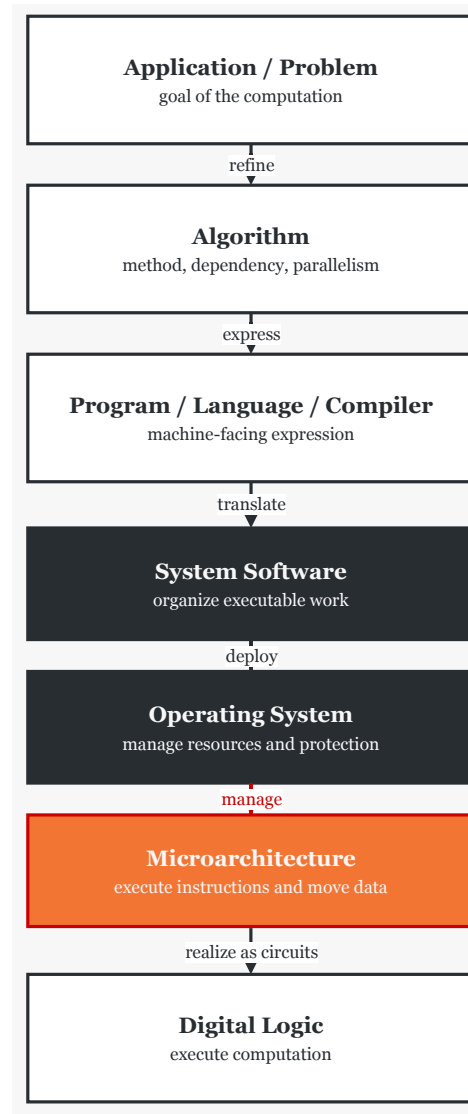
"Computing" is more than arithmetic, it is a full-stack process from problems to results.

In this course, we study how system software, operating systems, and hardware make computation **faster, more efficient, and more scalable.**

Transformation Hierarchy of Computing

Abstraction in System Design

- **Application / Problem**
 - AI inference, training, web search, recommendation
- **Algorithm**
 - GEMM, attention, optimization, sampling
- **Program / Language**
 - Python, C++, CUDA, Go
- **System Software**
 - Distributed frameworks, libraries, drivers, compilers, runtime

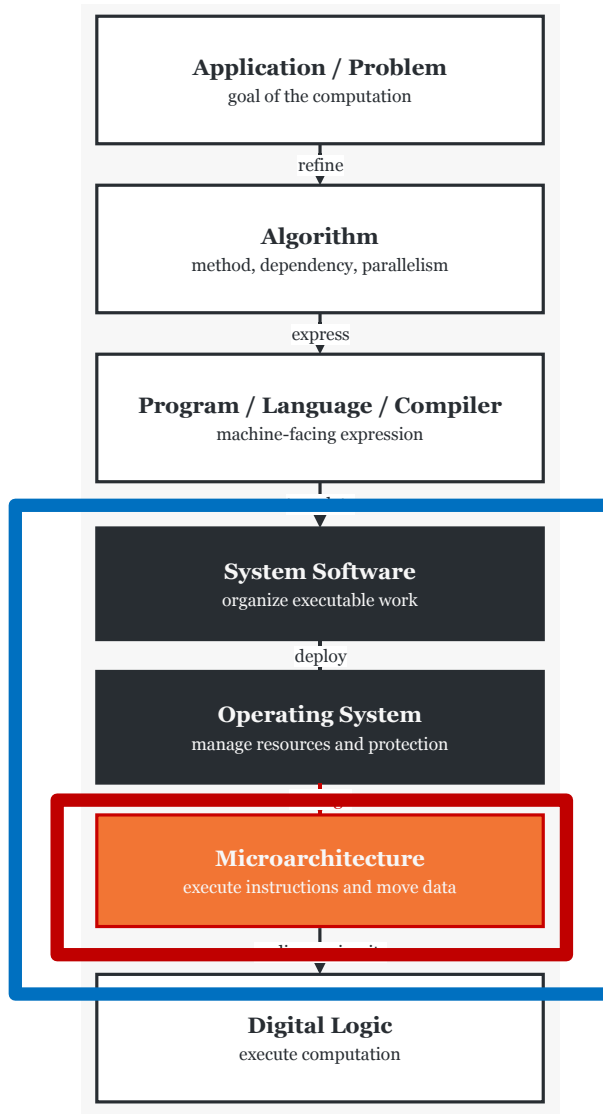


- **Operating System**
 - Process, thread, memory, scheduling, I/O
- **Microarchitecture**
 - Core, cache, RAM, pipeline, bus, controller
- **Digital Logic**
 - Gates, registers, wires, arithmetic units

Transformation Hierarchy of Computing

Abstraction in System Design

- **Application / Problem**
 - AI inference, training, web search, recommendation
- **Algorithm**
 - GEMM, attention, optimization, sampling
- **Program / Language**
 - Python, C++, CUDA, Go
- **System Software**
 - Distributed frameworks, libraries, drivers, compilers, runtime



- **Operating System**
 - Process, thread, memory, scheduling, I/O
- **Microarchitecture**
 - Core, cache, RAM, pipeline, bus, controller
- **Digital Logic**
 - Gates, registers, wires, arithmetic units

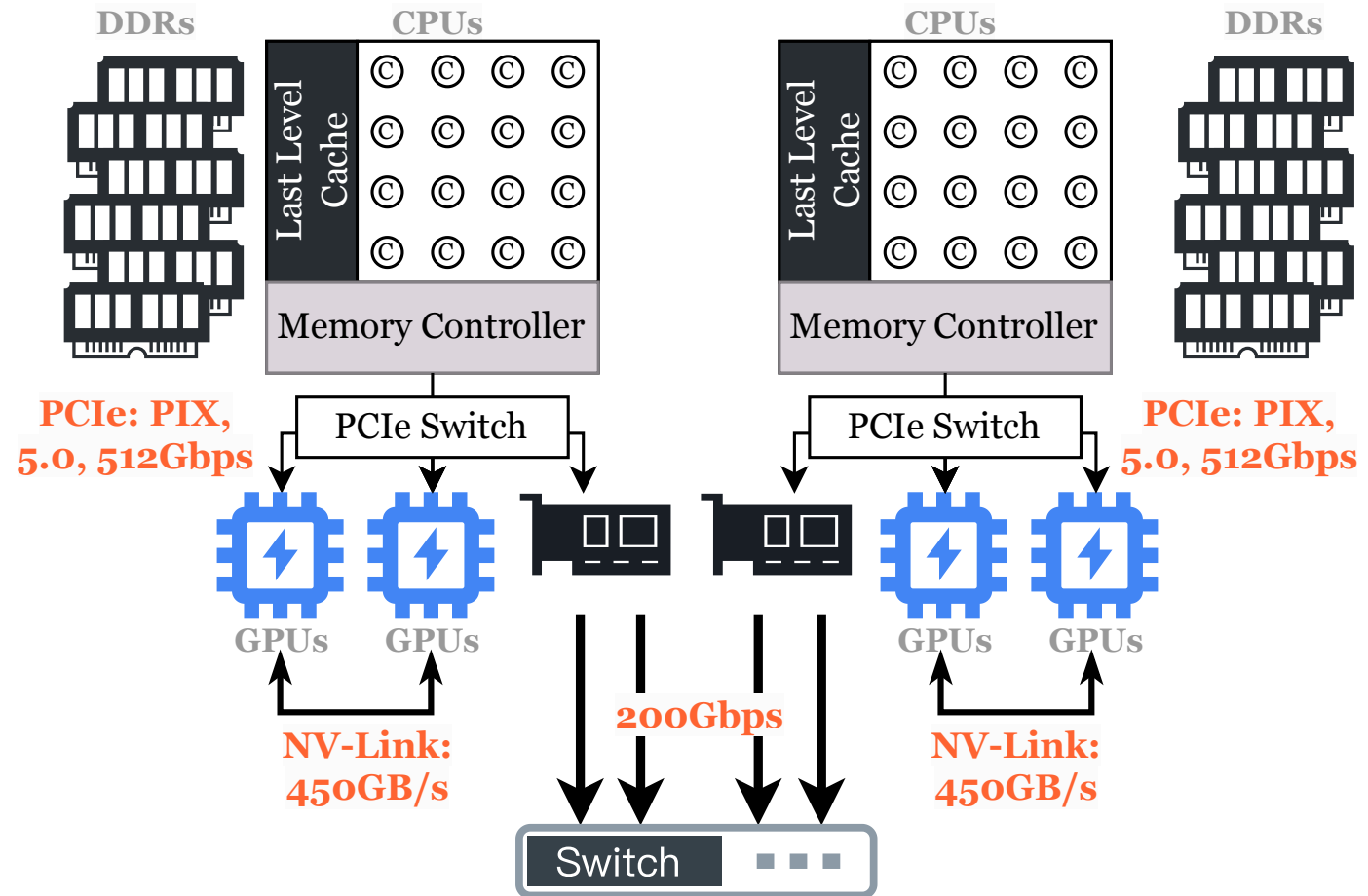
A Quick Review of Host Architecture

From System Design to Execution Platform

- A modern host system connects:

- CPU
- DDR / DRAM
- GPU / Accelerator
- SSD / Storage
- NIC / Network
- PCIe / Interconnect

- Key lesson: End-to-end performance depends on the interconnection.



A Quick Review of Host Architecture

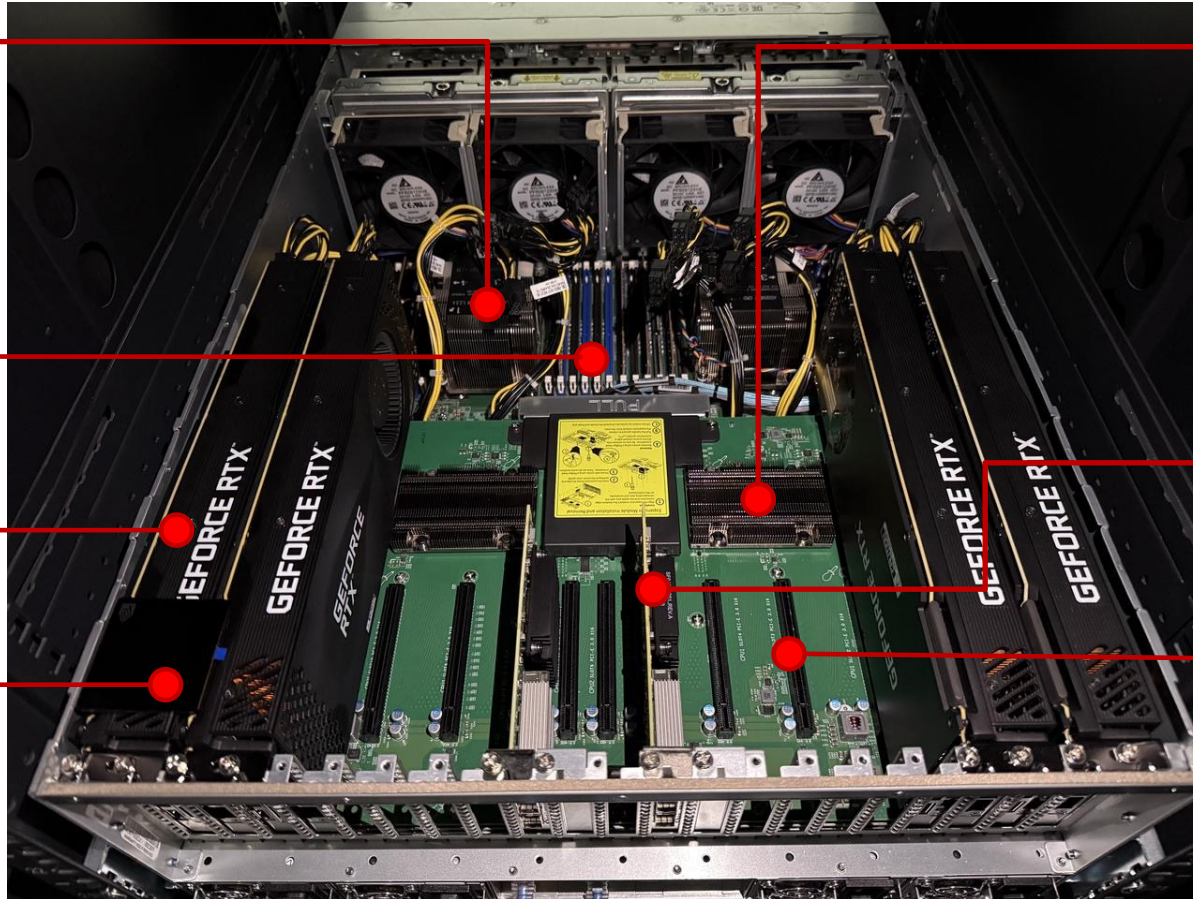
Building a Computer: Motherboard as the Integration Plane

■ CPU

■ DDR

■ GPU

■ NVLink



■ SSD / NVMe

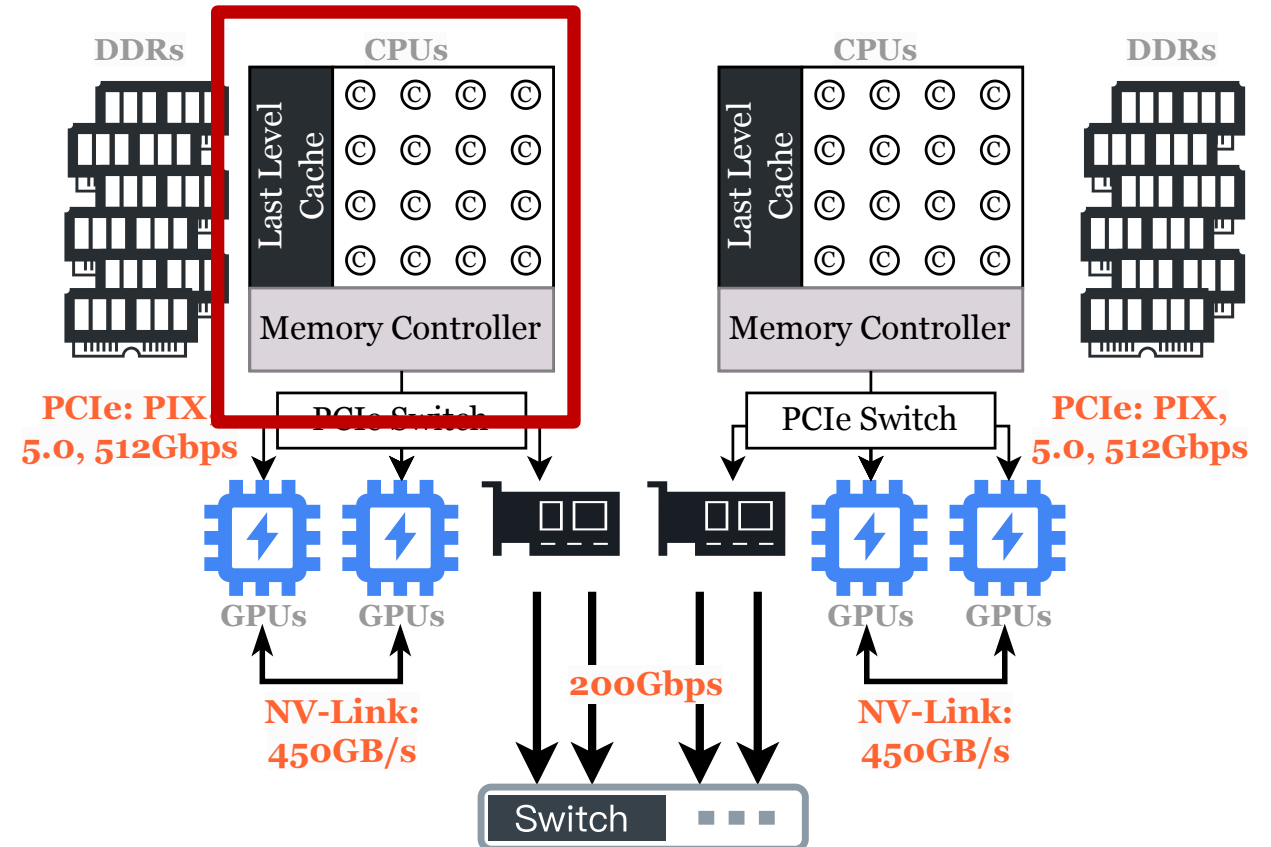
■ NIC

■ PCIe

A Quick Review of Host Architecture

Roles in AI Computing --- CPU Package: the Host Controller

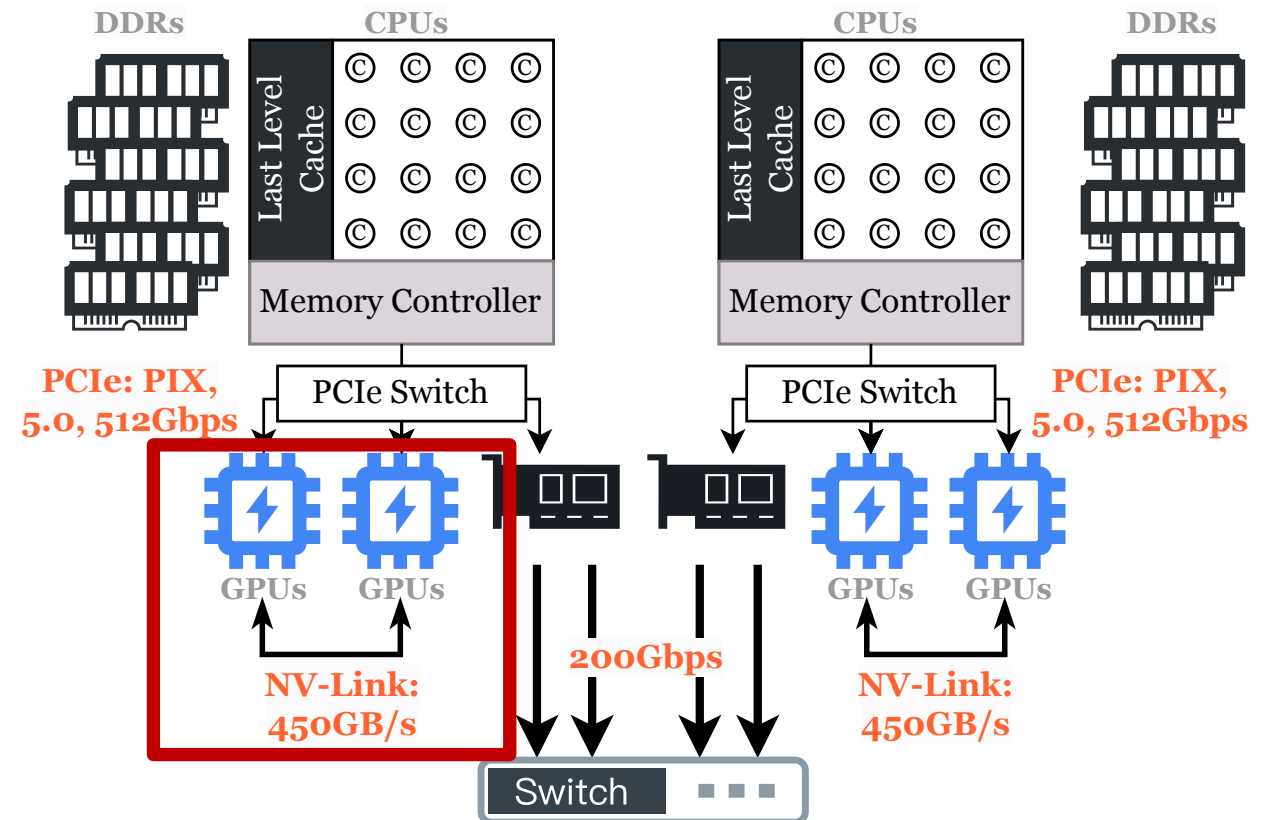
- Main Function: Run the Operating System and System Software
 - Install Debian / CentOS to manage all devices
 - E.g., Install file system, acts as the PCIe root, monitors power and temperature, facilitate data flow among devices, etc.
- Deploy system software and launch codes
 - Users launch python-based frameworks and calls inference / training tasks.
 - The CPU prepares input data from SSD/DRAM and calls GPU kernels to dispatch complex matrix computation to multiple GPUs.
 - Many services are deployed on CPUs such as chatbox, coding agents, webs, etc.



A Quick Review of Host Architecture

Roles in AI Computing --- GPU Package: the Main Computing Power

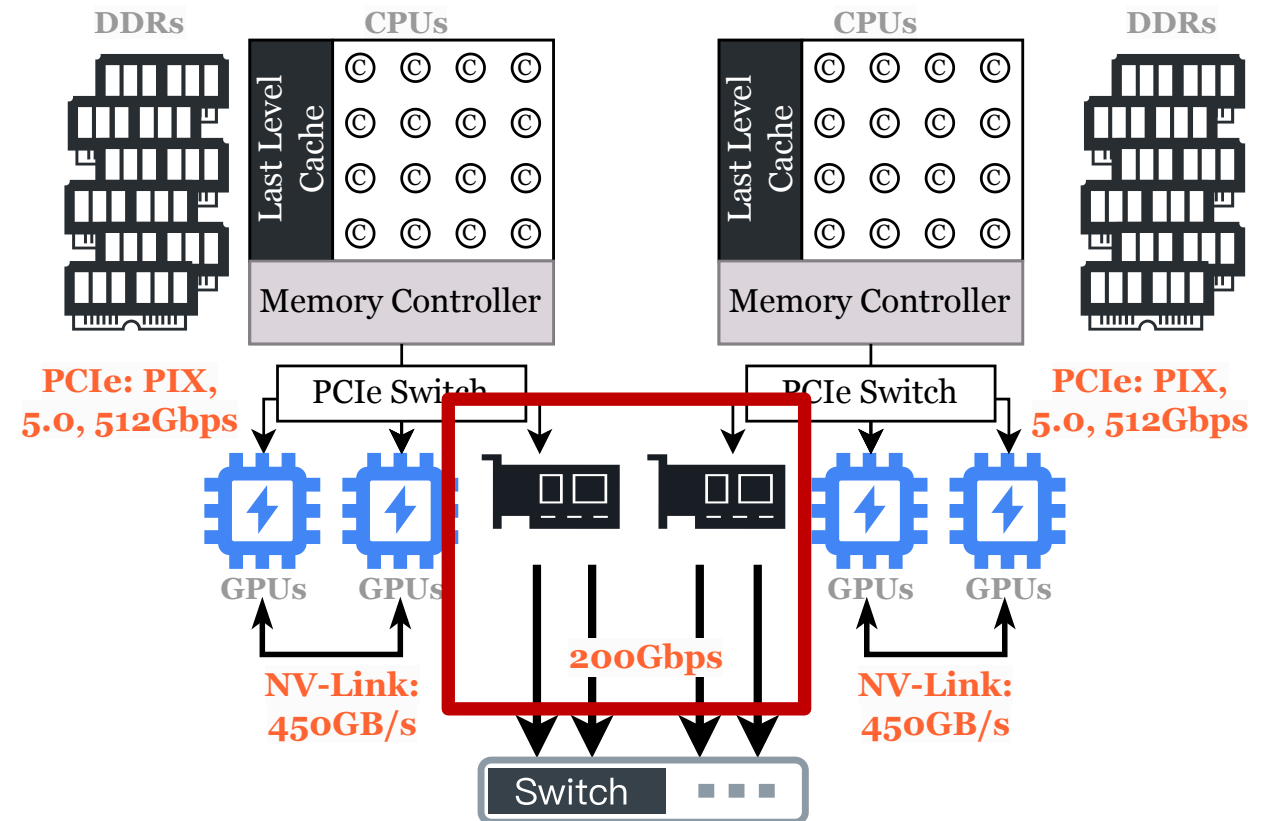
- Main Function: Provide High-Throughput Computation
 - Responsible for matrix and tensor operations and return results.
 - Package high-bandwidth memory (HBM) to maximum computation efficiency.
- Typically, a host server holds eight GPUs, limited by power supply and heat dissipation.



A Quick Review of Host Architecture

Roles in AI Computing --- NIC Package: the Host Scalable Input/Output

- Main Function: Scale Single Computing System for Distributed Computing
 - Connect many hosts to execute the same task.
 - E.g., 5D parallelism for AI training (Lecture 9).
 - A training task may involve thousands of servers, requiring a large network with high bandwidth (Lecture 5).
 - Package network stack to facilitate GPU-Direct-Networking, bypass CPU to avoid the bottleneck.



A Quick Review of Host Architecture

CUDA Host-Device Execution Model

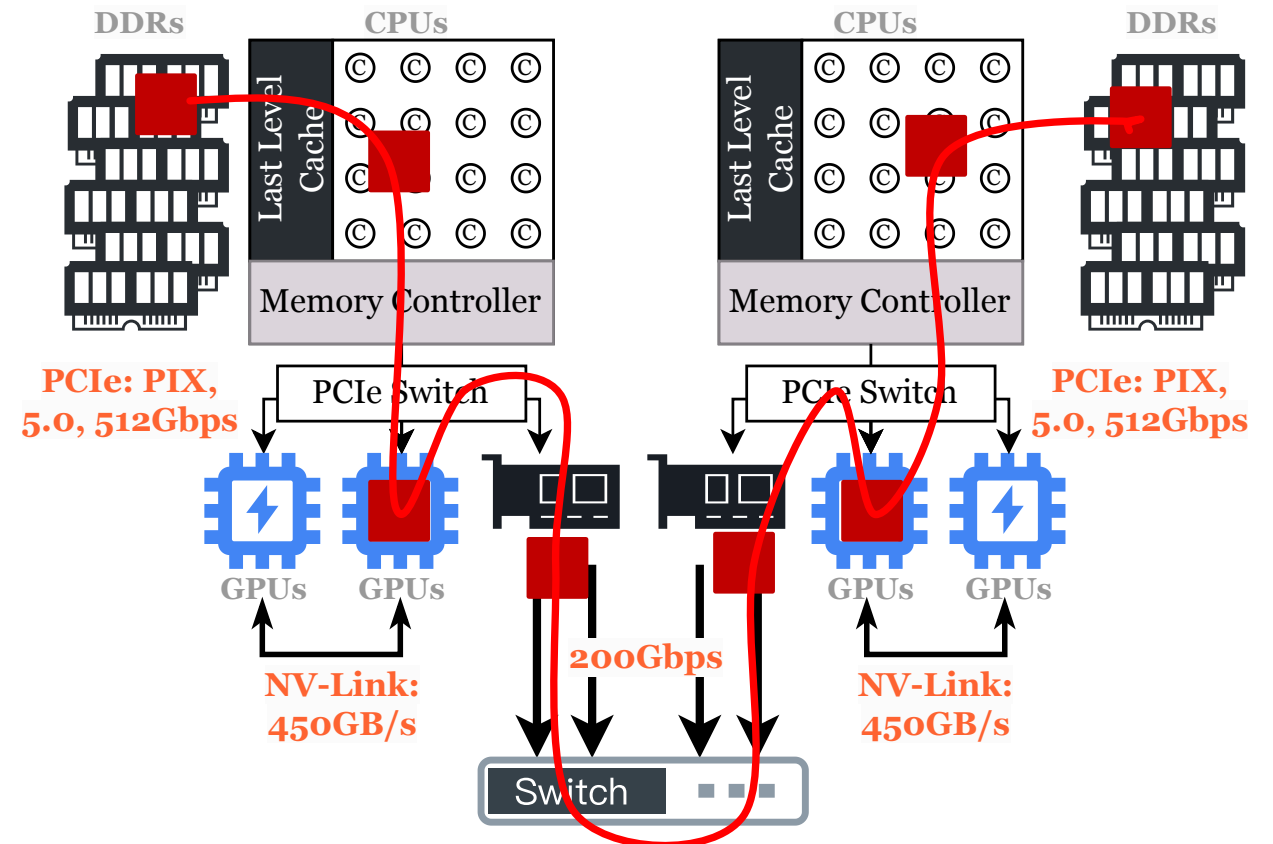
- Host -> Device -> Network -> Device -> Host
 - Host Side: CPU
 - Allocate memory, copy data, launch kernels, synchronization.
 - Device Side: GPU
 - Run many threads to execute kernels.
 - Network Side: NIC
 - Execute collective communication for remote memory access or copy.

```
// CPU: orchestrates
cudaMemcpyAsync(h_input -> d_buf);
compute_kernel<<<...>>>(d_buf);

// Network library: communicates GPU buffers
ncclAllReduce(d_buf, d_buf, ..., stream);

// GPU: continues computation after communication
compute_kernel<<<...>>>(d_buf);

cudaStreamSynchronize(stream);
```



CPU Microarchitecture

Lecture 1: Processor Evolution — From CPU to GPU

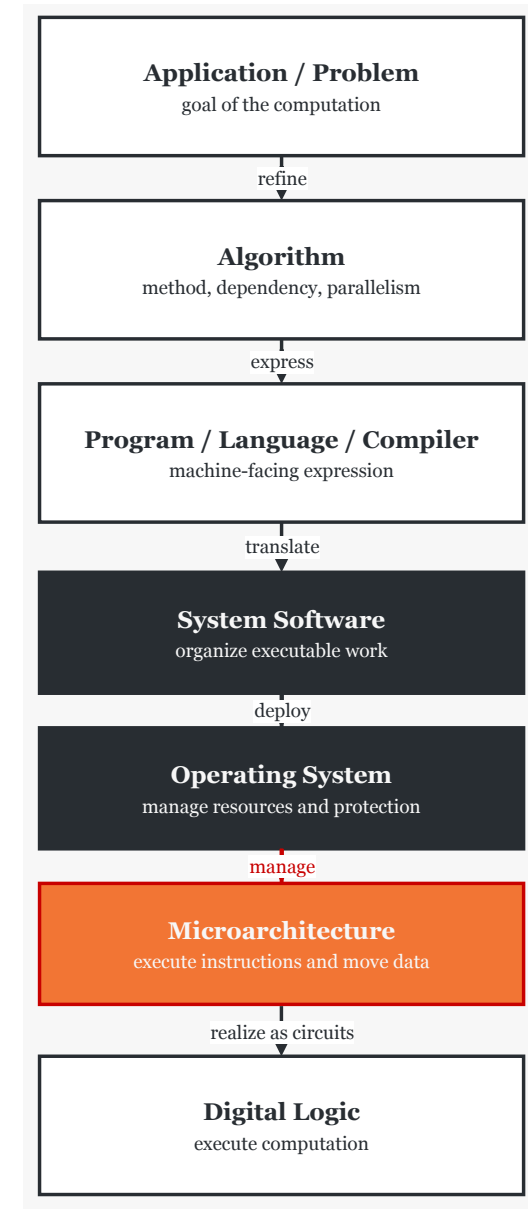


THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

The Bottom of Computing: Instruction Execution

- Instruction execution is the last step of computing.
 - Workflow
 - Load: from RAM to register file
 - Compute: add, mul among registers
 - Store: from register file to RAM
 - The **instruction efficiency** decides to maximum computing power and speed
- 
- Motivate two types of PU micro-architecture
 - CPU: optimize **latency** of a small number of **complex** threads.
 - GPU: optimize **throughput** of a large number of **simple** threads.



CPU: Optimize Latency of a Small Number of Complex Threads

Architecture Overview

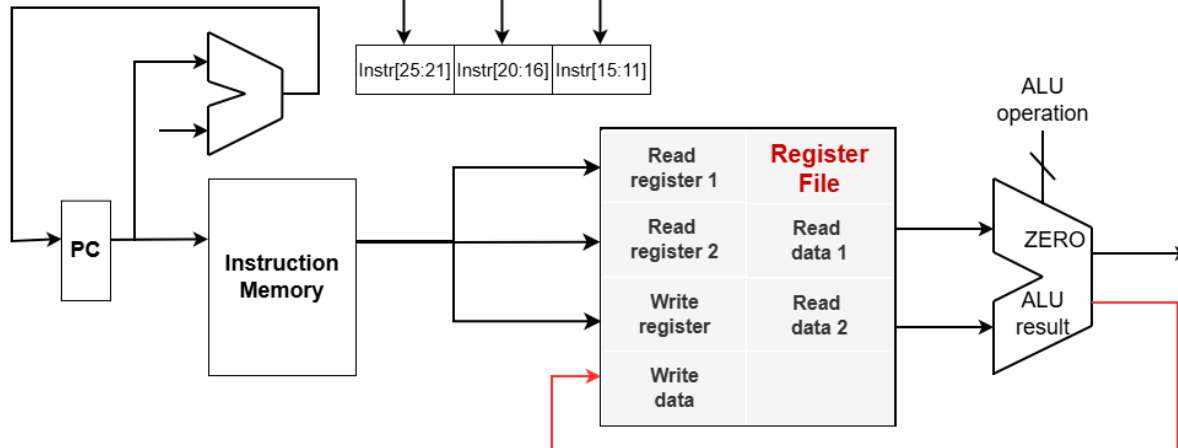
■ Three Components in a Computing Core:

- ALU for computing
- Register file for data load/store
- Instruction memory for pipelining

MIPS: add \$t0, \$s1, \$s2 (\$t0 = \$s1 + \$s2)

opcode	rs	rt	rd	shamt	funct
0	17 (\$s1)	18 (\$s2)	8 (\$t0)	0	32
00000	10001	10010	01000	0	100000

Register	Value
s1	100
s2	funct



CPU Specifications

Total Cores ?	128
Total Threads ?	256
Max Turbo Frequency ?	3.9 GHz
All Core Turbo Frequency ?	3.2 GHz
Processor Base Frequency ?	2 GHz
Cache ?	504 MB
Intel® UPI Speed	24 GT/s
Max # of UPI Links ?	6
TDP ?	500 W

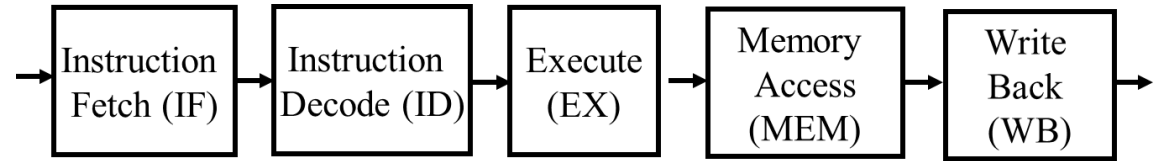
■ Specification of Intel Xeon 6980P

- 256 Threads --- decide parallelism
- 3.2GHz --- decide the speed of ALU/instruction execution
- 504MB Cache --- decide the speed of data load/store

CPU: Optimize Latency of a Small Number of Complex Threads

Five-Stage Instruction Pipeline

- Goal: Overlap time of data load/store
- Fetch (IF)
 - Get instructions from instruction memory/cache
- Decode (ID)
 - Resolve opcode and operands
- Execute (EX)
 - ALU, vector unit, load/store unit, tensor core
- Memory Access (MEM)
 - Load/store data through cache or memory hierarchy
- Write Back (WB)
 - Update architectural state



Time →

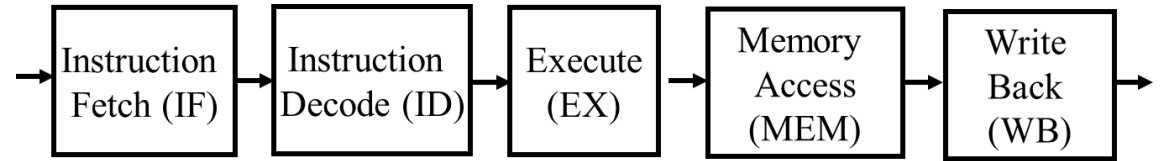
	1	2	3	4	5	6	7	8	9	10
Instruction 1	IF	ID	EX	MEM	WB					
Instruction 2		IF	ID	EX	MEM	WB				
Instruction 3			IF	ID	EX	MEM	WB			
Instruction 4				IF	ID	EX	MEM	WB		
Instruction 5					IF	ID	EX	MEM	WB	
Instruction 6						IF	ID	EX	MEM	WB

CPU: Optimize Latency of a Small Number of Complex Threads

Five-Stage Instruction Pipeline

■ Perfect Pipeline

- One instruction completes per cycle after filling the pipeline.
- No **bubbles** and maximum throughput.



■ For N instructions and M stages:

- Total Cycles = $M + N - 1$
- Average throughput

$$= N / (M + N - 1) \text{ instructions per cycle}$$
- Instruction throughput per second

$$\text{Frequency} * N / (M + N - 1)$$

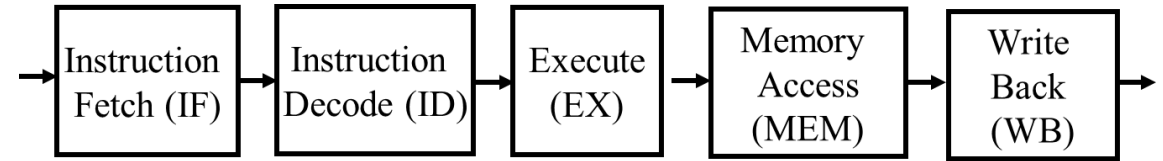
Time →

	1	2	3	4	5	6	7	8	9	10
Instruction 1	IF	ID	EX	MEM	WB					
Instruction 2		IF	ID	EX	MEM	WB				
Instruction 3			IF	ID	EX	MEM	WB			
Instruction 4				IF	ID	EX	MEM	WB		
Instruction 5					IF	ID	EX	MEM	WB	
Instruction 6						IF	ID	EX	MEM	WB

CPU: Optimize Latency of a Small Number of Complex Threads

Why Pipeline Stalls: Hazards

- Any condition that causes the pipeline to stall is called a **hazard**.
 - Result: Much bubbles and degraded efficiency
- Various types of hazards
 - Data Hazard**: Stall due to dependencies of some data
 - Read-after-write (RAW)
 - Write-after-write (WAW)
 - Write-after-read (WAR)
 - Control Hazard**: Stall due to branch and synchronization
 - Structural hazard: Stall due to contention on hardware such as memory channel, cache slot, queue, etc.



Time →

	1	2	3	4	5	6	7	8	9	10
Instruction 1	IF	ID	EX	MEM	WB					
Instruction 2		IF	ID	EX	MEM	WB				
Instruction 3			IF	ID	EX	MEM	WB			
Instruction 4				IF	ID	EX	MEM	WB		
Instruction 5					IF	ID	EX	MEM	WB	
Instruction 6						IF	ID	EX	MEM	WB

CPU: Optimize Latency of a Small Number of Complex Threads

Data Hazards

RAW

Read After Write: true dependency

One instruction relies on the output from the previous instruction.

Example: AND waits for R0 produced by ADD.

cannot be renamed away

```
ADD R0, R2, R1 ; R0 ← R2 + R1
AND R1, R0, #2 ; R1 ← R0 & 2
```

WAR

Write After Read: false dependency

A later write may overwrite a register before an earlier instruction finishes reading it.

Example: AND must wait until ADD reads R1.

renaming can remove it

```
ADD R0, R2, R1 ; R0 ← R2 + R1
AND R1, R3, #2 ; R1 ← R3 & 2
```

WAW

Write After Write: false dependency

Two nearby instructions write to the same architectural register.

Example: SUB must wait until ADD finishes writing R0.

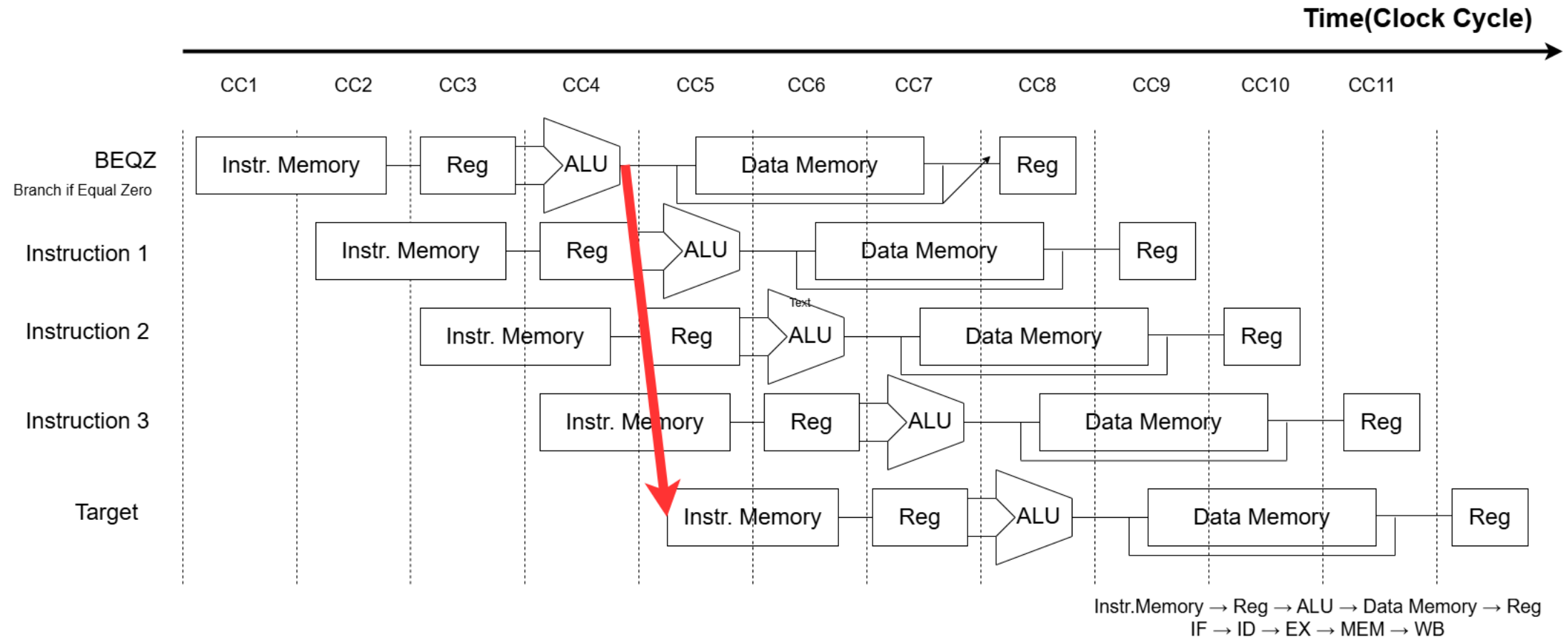
renaming can remove it

```
ADD R0, R2, R1 ; R0 ← R2 + R1
SUB R0, R3, #1 ; R0 ← R3 - 1
```

CPU: Optimize Latency of a Small Number of Complex Threads

Control Hazards

- Case #1: Deeper pipeline causes worse penalty



CPU: Optimize Latency of a Small Number of Complex Threads

Control Hazards

- Case #2: Branches create uncertainty

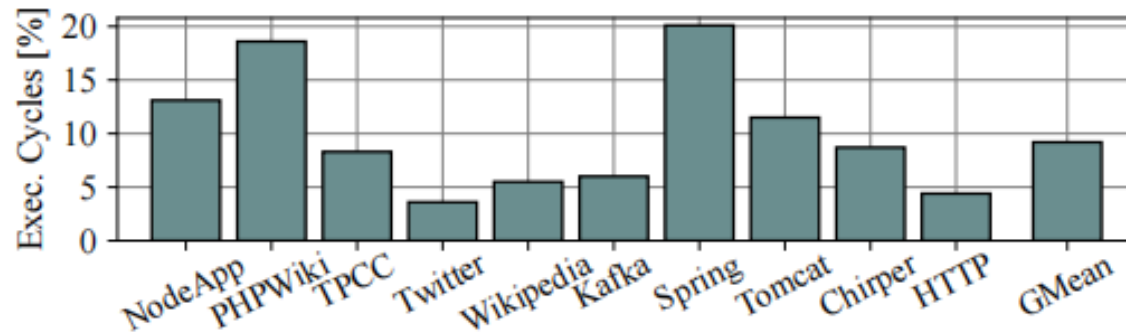


Fig. 1: Execution cycles wasted due to conditional branch mispredictions.

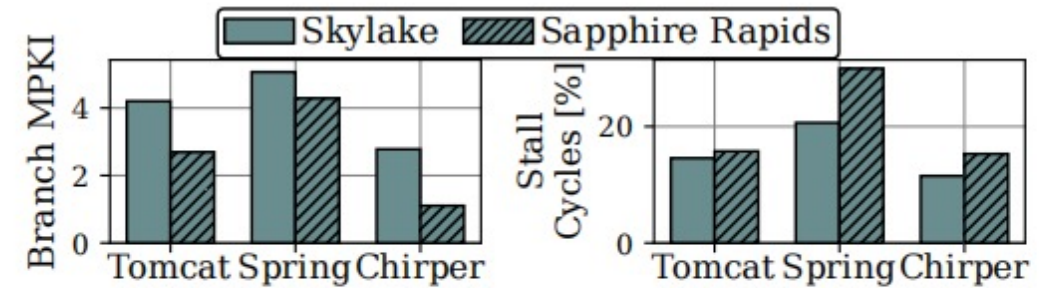


Fig. 1: Comparison of stall cycles due to conditional branch mispredictions and branch MPKI on Intel Skylake (solid) and Intel Sapphire Rapids (striped)

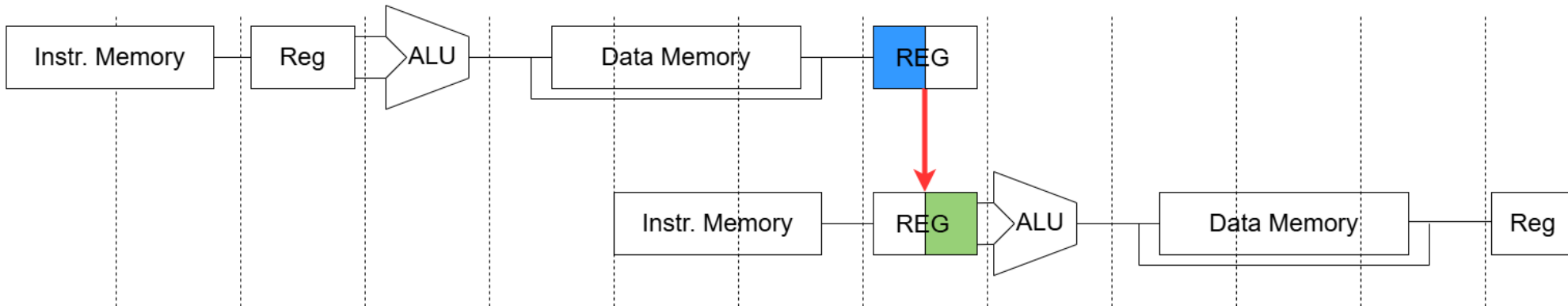
Source: D. Schall, M. Ďuračková, and B. Grot, "The Last-Level Branch Predictor Revisited," HPCA '26, Table I. /
Source: D. Schall, A. Sandberg, and B. Grot, "The Last-Level Branch Predictor," MICRO '24, Fig. 1.

CPU: Optimize Latency of a Small Number of Complex Threads

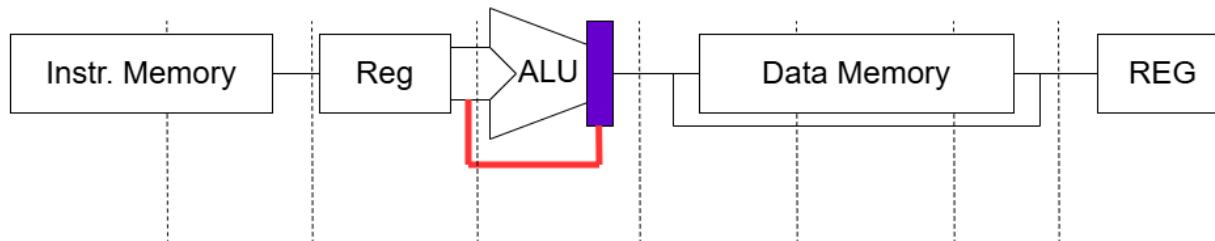
Optimization #1: Data Forwarding

- Enable Register-to-Register or Register-to-ALU Forwarding
 - E.g., result is passed directly from pipeline register to ALU input.
 - The **consumer can execute earlier** —> Fewer stalls and higher pipeline efficiency

Register-to-Register



Register-to-ALU



CPU: Optimize Latency of a Small Number of Complex Threads

Optimization #2: Register Renaming

WAR**Write After Read**

Before renaming

```
ADD R4, R2, R1 ; R4 ← R2 + R1
AND R1, R0, #2 ; R1 ← R0 & 2
```

Rename R1 to R3

After renaming

```
ADD R4, R2, R1 ; R4 ← R2 + R1
AND R3, R0, #2 ; R3 ← R0 & 2
```

WAW**Write After Write**

Before renaming

```
ADD R0, R2, R1 ; R0 ← R2 + R1
SUB R0, R3, #1 ; R0 ← R3 - 1
```

Rename R0 to R4

After renaming

```
ADD R0, R2, R1 ; R0 ← R2 + R1
SUB R4, R3, #1 ; R4 ← R3 - 1
```

- Use more registers to **decouple dependencies**.

CPU: Optimize Latency of a Small Number of Complex Threads

Examples for Data Forwarding and Register Renaming

Example:

```
lw  R1, 4(R2)
add R4, R1, R5
sub R4, R2, R3
or  R3, R1, R2
```

With data forwarding

Clock	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	M	WB				
I2		IF	S	ID	EX	M	WB		
I3				IF	ID	EX	M	WB	
I4					IF	ID	EX	M	WB

Use register renaming and instruction reordering

```
lw  R1, 4(R2)
sub R6, R2, R3
add R4, R1, R5
or  R3, R1, R2
```

Clock	1	2	3	4	5	6	7	8
I1	IF	ID	EX	M	WB			
I2		IF	ID	EX	M	WB		
I3			IF	ID	EX	M	WB	
I4				IF	ID	EX	M	WB

CPU: Optimize Latency of a Small Number of Complex Threads

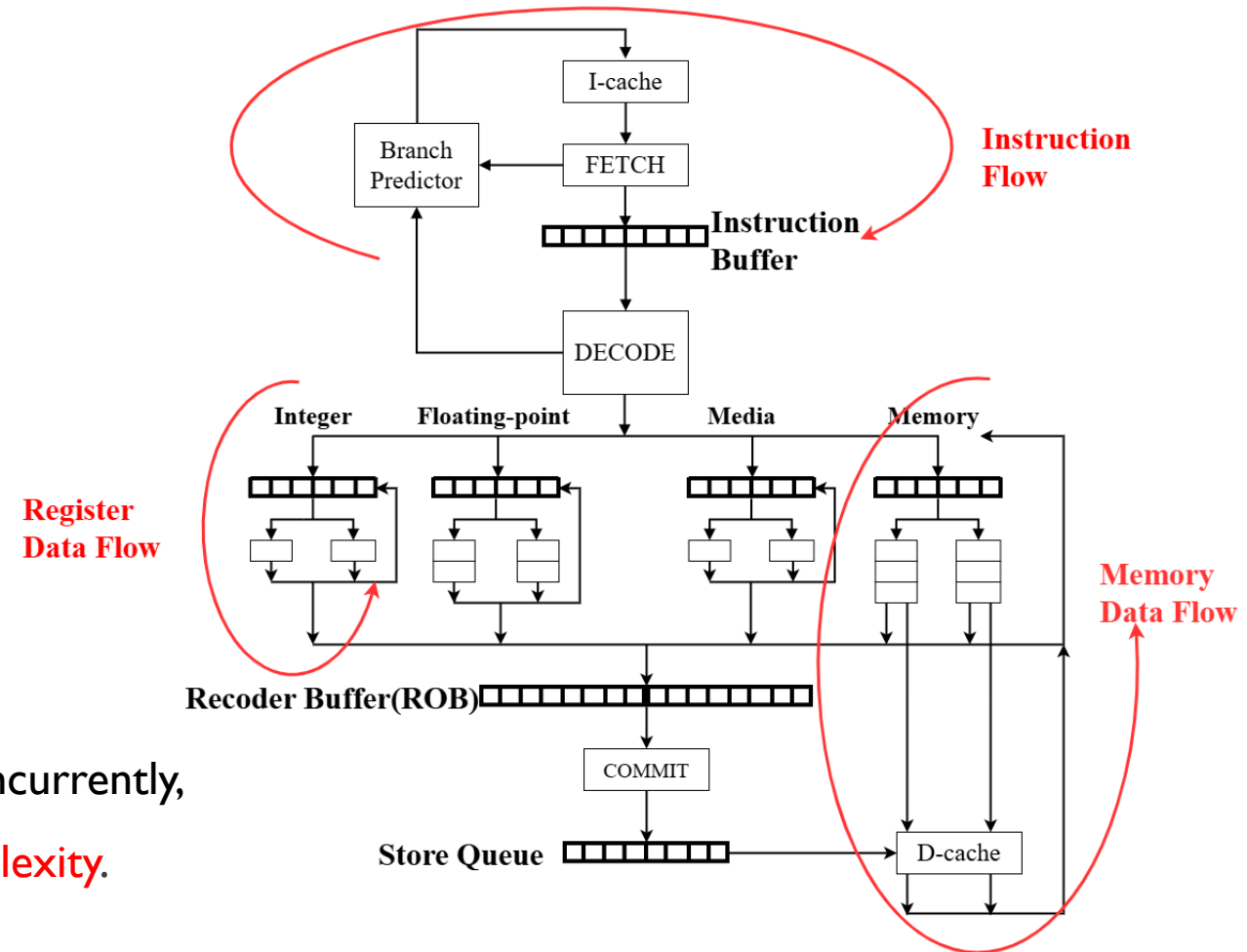
Optimization #3: Out-of-Order (OoO) Execution

- Out-of-order execution separates:
 - **Program order:** order written by the programmer
 - **Execution order:** order chosen by hardware
 - **Commit order:** order that updates architectural state
- Typically, the compiler would find independent instruction sets to enable OoO execution, reducing the stall time of the instruction pipeline.
- At runtime, CPU involves a controller to find independent instructions while some instructions are waiting.
 - Preempt current pipeline
 - Issue independent instruction
 - Restore preempted instruction when it finishes waiting

CPU: Optimize Latency of a Small Number of Complex Threads

Optimization #4: Superscalar

- Superscalar CPU: tries to **execute multiple instructions** every cycle.
- Need a "wide" pipeline
 - Wide fetch
 - Wide decode
 - Multiple execution units
 - Multiple load/store paths
 - Wide writeback
 - Multi-ported register file
- Challenge:
 - If instruction pipeline can issue N instructions concurrently,
 - The CPU needs resolve **$O(N^2)$ dependence complexity**.



CPU: Optimize Latency of a Small Number of Complex Threads

Optimization #5: Speculative Execution

- Prediction for next instructions
 - Issue risky instructions on the chance of considerable benefits.
- Benefit
 - Quick completion --- low latency
 - Keep deep pipeline busy --- high throughput
- Cost
 - **Wrong-path** work wastes energy and cycles.
 - **Recovery penalty** grows with pipeline depth and speculation window.

CPU: Optimize Latency of a Small Number of Complex Threads

Speculative Execution: Branch Prediction

- Naïve speculative execution: Static Prediction
 - The compiler resolves user's codes and find the branches that would not be taken.
 - Remove corresponding instructions to avoid control hazards.
- Dynamic Branch Prediction
 - Learn branch behavior autonomously, no compiler analysis or profiling
 - Adapt to changing branch behavior

Example: Why branch is predictable

Source-Level Loop

```
for (i = 0; i < 100; i++) {  
    ...  
}
```

Assembly: Loop Control

```
addi r10, r0, 100  
addi r1, r0, r0  
  
L1:  
    ...  
    ...  
addi r1, r1, 1  
bne r1, r10, L1  
    ...
```

loop back-edge branch

Source-Level Conditions

```
if (aa == 2) aa = 0;  
if (bb == 2) bb = 0;  
if (aa != bb) ...
```

Assembly: Conditional Branches

```
addi r2, r0, 2  
bne r10, r2, L_bb  
xor r10, r10, r10  
L_bb:  
bne r11, r2, L_xx  
xor r11, r11, r11  
L_xx:  
beq r10, r11, L_exit  
    ...  
L_exit:
```

three conditional branches

CPU: Optimize Latency of a Small Number of Complex Threads

Optimization #6: OoO Superscalar with Hyper-Threading

- Problem: Even OoO superscalar cores have idle slot issues.
- Reasons:
 - Cache miss, branch misprediction, dependency chain.....
- Hyper-threading introduces simultaneous multithreading (SMT)
 - Multiple hardware threads share one physical core.
 - Each logical thread has its own architectural state.
 - Execution units, caches, queues, and predictors are shared.



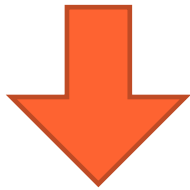
- The closest architecture to modern CPUs.

CPU: Optimize Latency of a Small Number of Complex Threads

Summary

- A modern CPU core spends many transistors on

- Data Forwarding
- Register Renaming
- OoO Execution
- Superscalar
- Speculative Execution
-



- Answer: How can one or a few complex threads make progress quickly?



- Ideal Scenarios

- Control-heavy programs
- Irregular data structures
- OS and runtime services
- Low-latency request handling
-



- Much complex micro-architecture and difficult to scale!

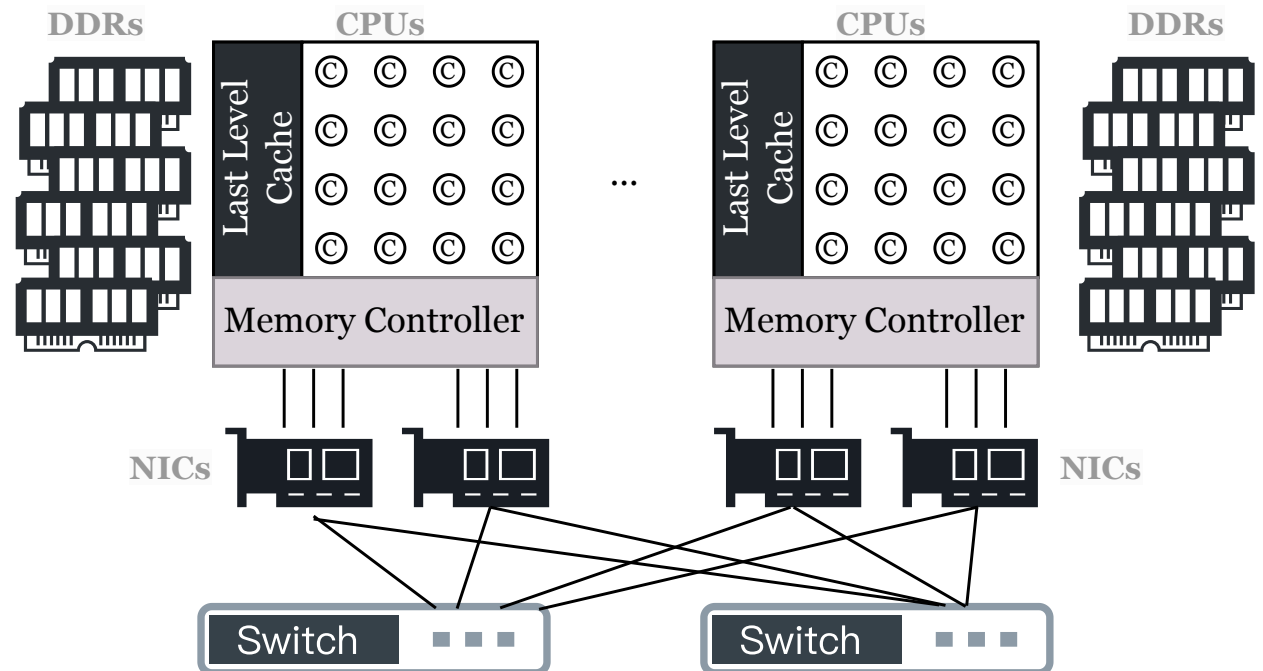
GPU Microarchitecture

Lecture 1: Processor Evolution — From CPU to GPU

From CPU to GPU: Parallelism for Huge Computing Power!

Multiprocessors

- **Problem:** How do we scale the throughput of a single core?
 - With many optimizations, a superscalar core achieves its ceiling --- **High Frequency; Deep Pipeline; Minimal Hazards.**
 - Answer: Scale the number of superscalar cores with **memory hierarchy**, scale the number of servers with **networking**, and design **parallelism**.
- **Memory Hierarchy (Lecture 3)**
 - Multiple cores share global memory (Caches and DRAM)
 - **Tightly Coupled Multiprocessors**
- **Networking (Lecture 4&5)**
 - No shared global memory
 - Data exchange via networking
 - **Loosely Coupled Multiprocessors**



From CPU to GPU: Parallelism for Huge Computing Power!

Multiprocessors

- **Parallelism:** Do multiple jobs concurrently.
 - Job: instructions, operations, tasks
- Data Parallelism
 - Different pieces of data can be operated in parallel
 - Vector processing and array processing
- Task Parallelism
 - Different tasks can be operated in parallel
 - Multithreading

From CPU to GPU: Parallelism for Huge Computing Power!

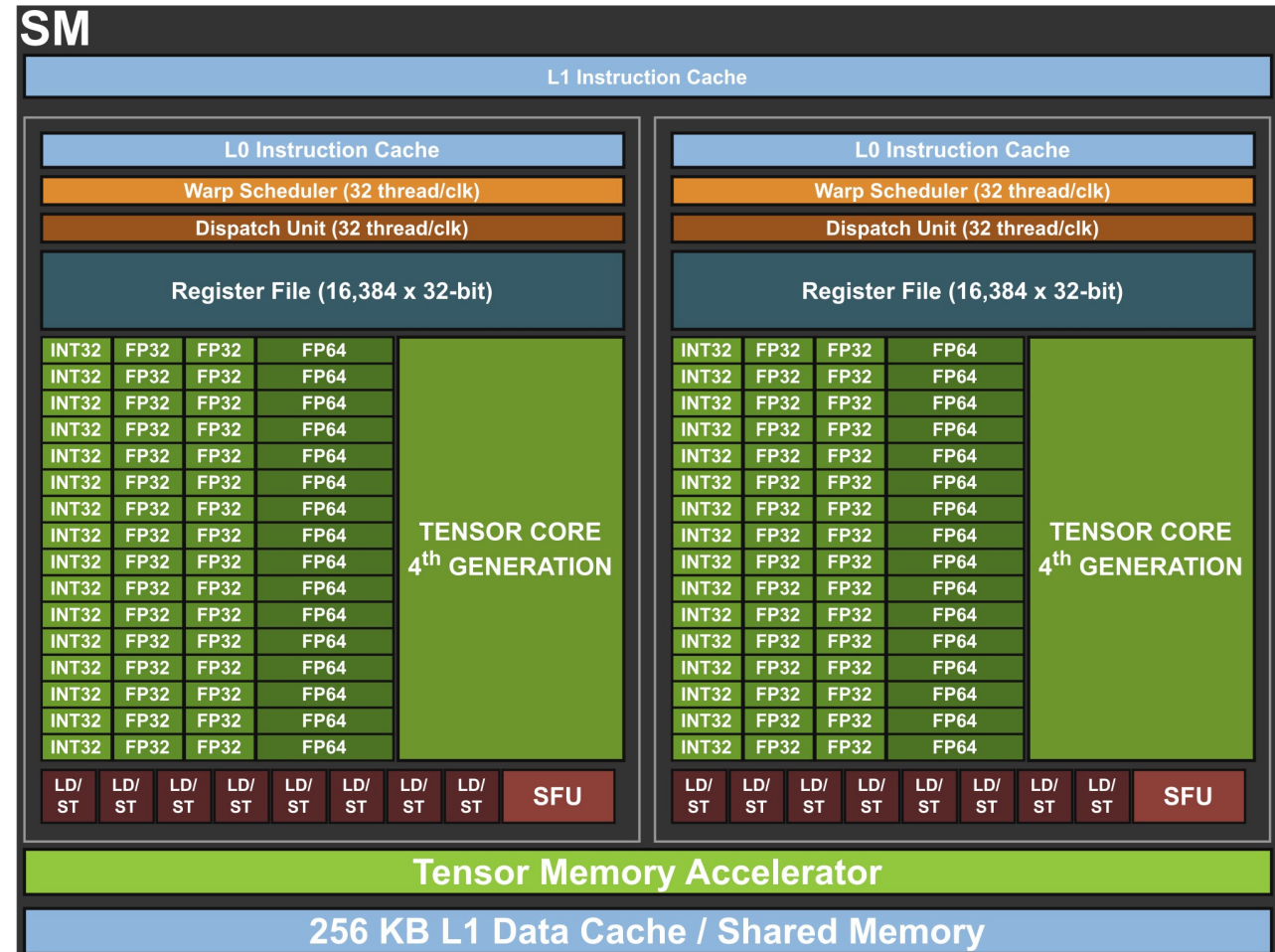
Difference of Scaling Cores: Control vs. Compute

- **CPU's Goal:** Be general to complex tasks.
 - Hazards are easy to occur, forcing CPU microarchitecture to implement much optimizations.
 - "Control logic" such as branches and dependencies in complex tasks incur many synchronization among cores.
 - Result: Scale to hundreds of heavy cores.
 - **GPU's Goal:** Be specific to tasks that are easy to parallel.
 - Control logic is less, data parallelism is easy.
 - Example: vector processing.
 - Result: Scale to 1,000s to 10,000s of light cores.
- 
- 10x to 100x of computing power than CPU!

GPU: Optimize Throughput of a Large Number of Simple Threads

Chip Organization

- Hundreds of **Streaming Multiprocessors (SM)s**
- Each SM consists of:
 - Processing units, e.g., INT32, FP32, FP64, etc.
 - Simplified pipelines compared to CPU cores.
 - Each unit executes specific instructions that are designed to specific computing precision.
 - Warp Scheduler
 - When deploy multiple threads, GPU packages a branch of threads into a warp.
 - The scheduler decides which warp would run at one time slot, and these threads shard a register file.
 - When one warp waits, schedule another ready warp.
 - High-speed shared memory for intra-SM data exchange.
 - Tensor core: accelerator for matrix operations.

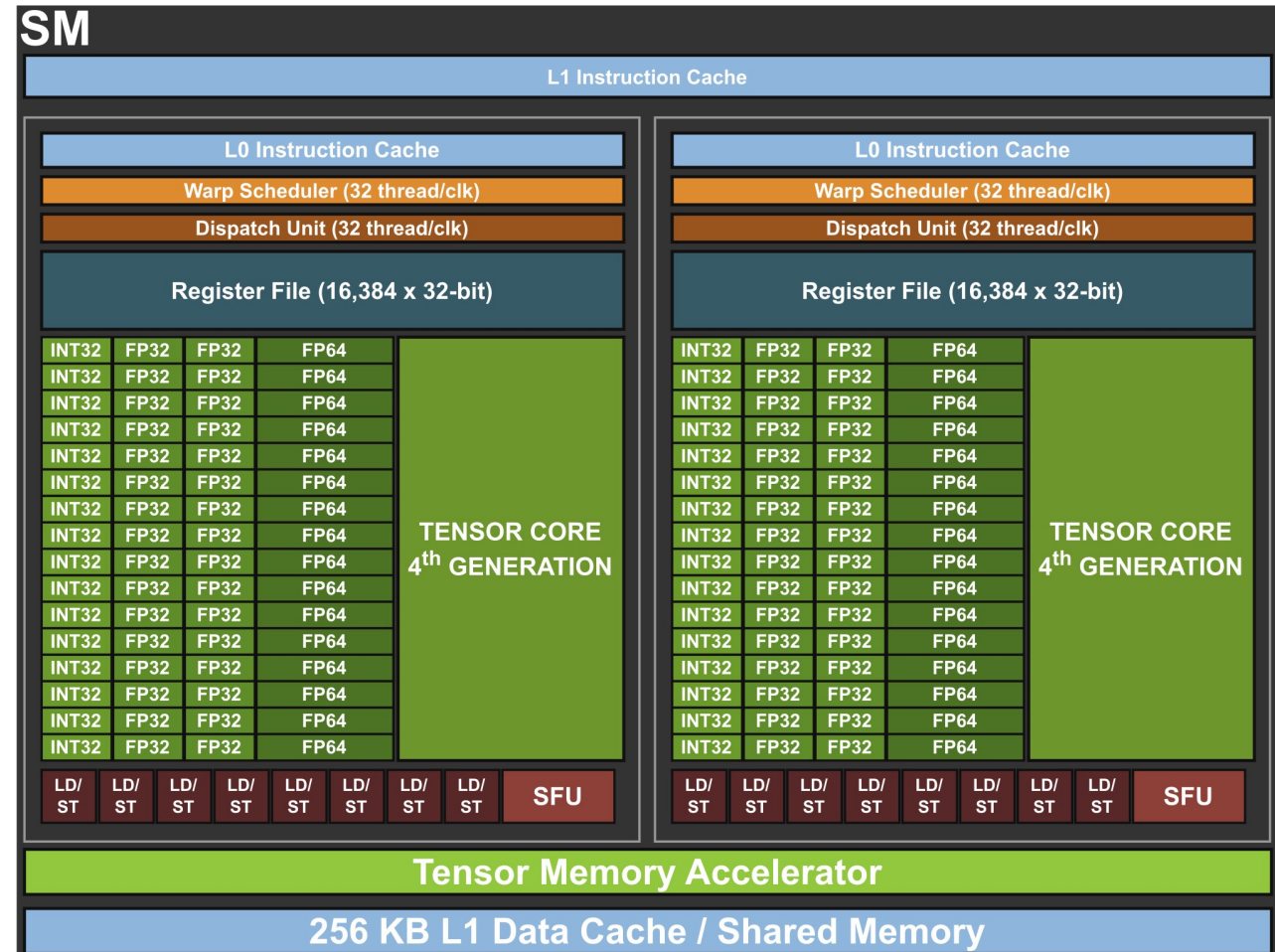
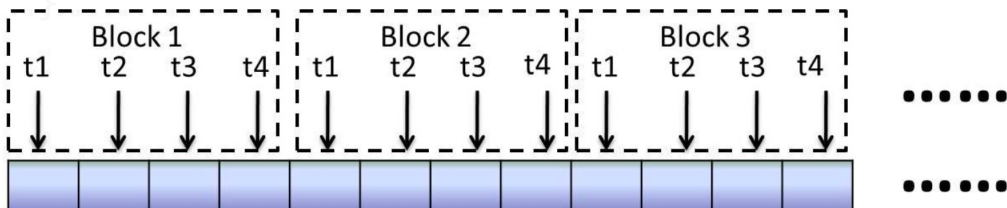


GPU: Optimize Throughput of a Large Number of Simple Threads

CUDA Programming and Execution Model

- Compute Unified Device Architecture (CUDA)
 - A parallel computing framework developed by NVIDIA.
- Kernel --- GPU-Level Jobs
- Thread Block --- SM-Level Jobs
- Warp of Threads --- Minimal Executing Units
- Thread --- Operator

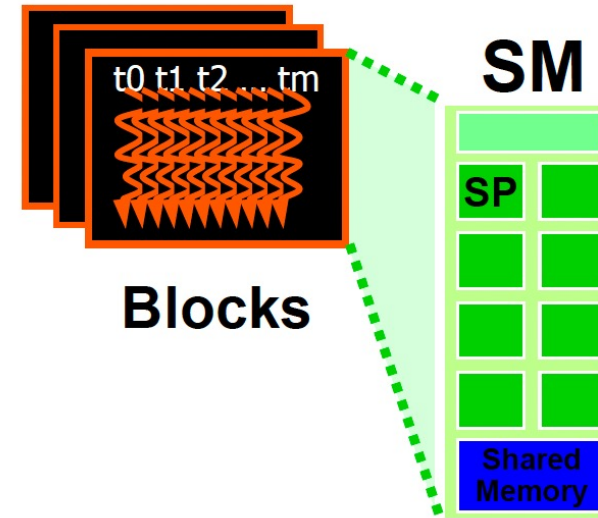
```
// Device code
__global__ void VecAdd(int* A, int* B,
int* C)
{
    int i = blockDim.x * blockIdx.x +
threadIdx.x;
```



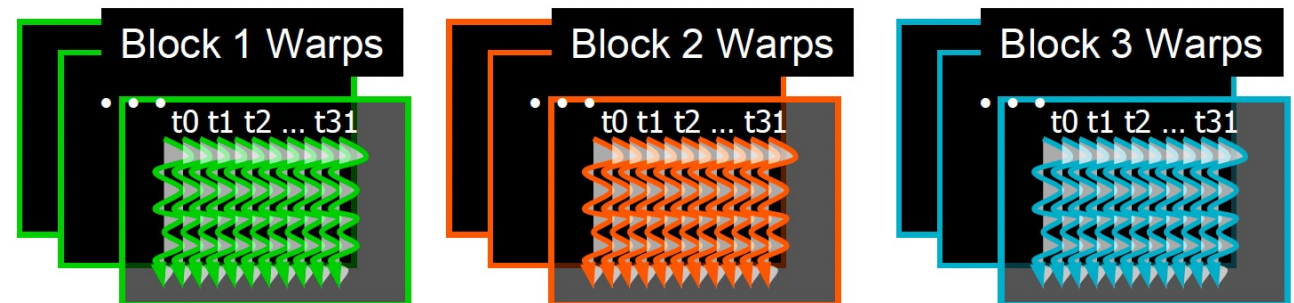
GPU: Optimize Throughput of a Large Number of Simple Threads

Example: Executing Thread Blocks

- A thread block is assigned to one SM
 - Up to 8 blocks to each SM
 - Fermi SM can take up to 1536 threads
 - Could be $256 * 6$ blocks
 - Or $512 * 3$ blocks, etc.



- Each block is divided into 32-thread warps
 - Scheduling by the warp scheduler within a SM.
- Single Instruction, Multiple Threads (SIMT) abstraction
 - Threads in a warp should execute same codes/instructions.
 - However, thread is allowed to execute different branches.

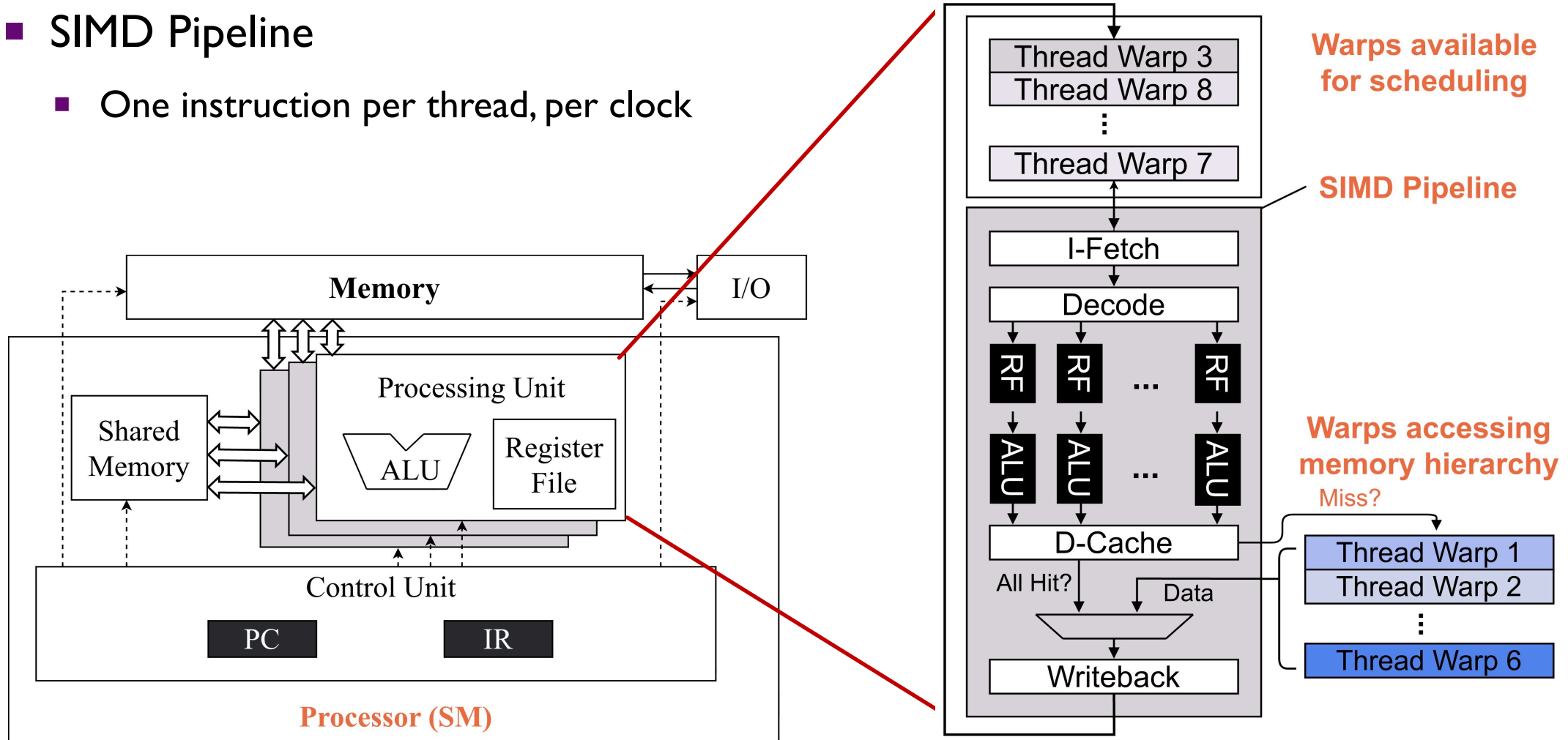


GPU: Optimize Throughput of a Large Number of Simple Threads

Microarchitecture: SMs are SIMD Units

■ SIMD Pipeline

- One instruction per thread, per clock

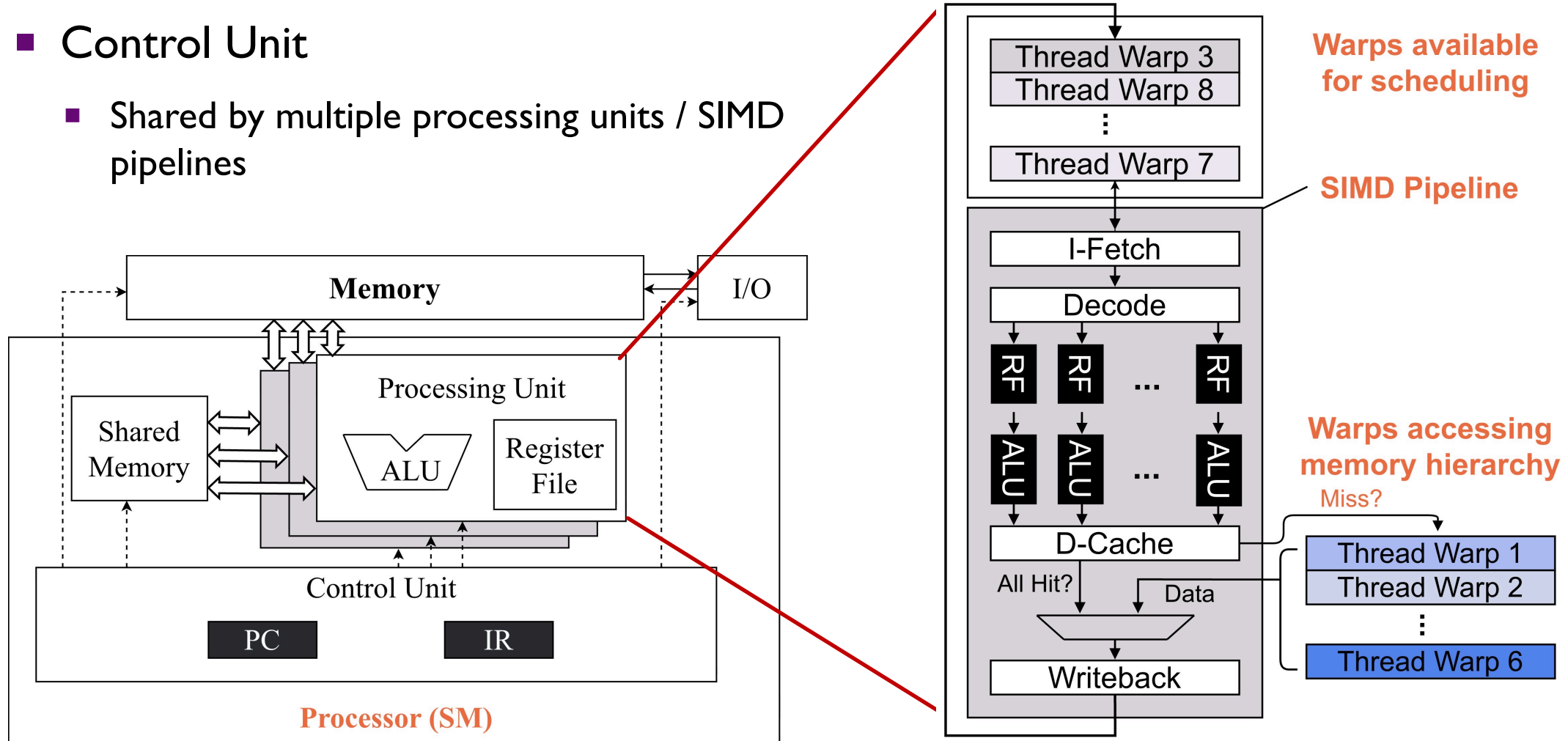


GPU: Optimize Throughput of a Large Number of Simple Threads

Microarchitecture: SMs are SIMD Units

■ Control Unit

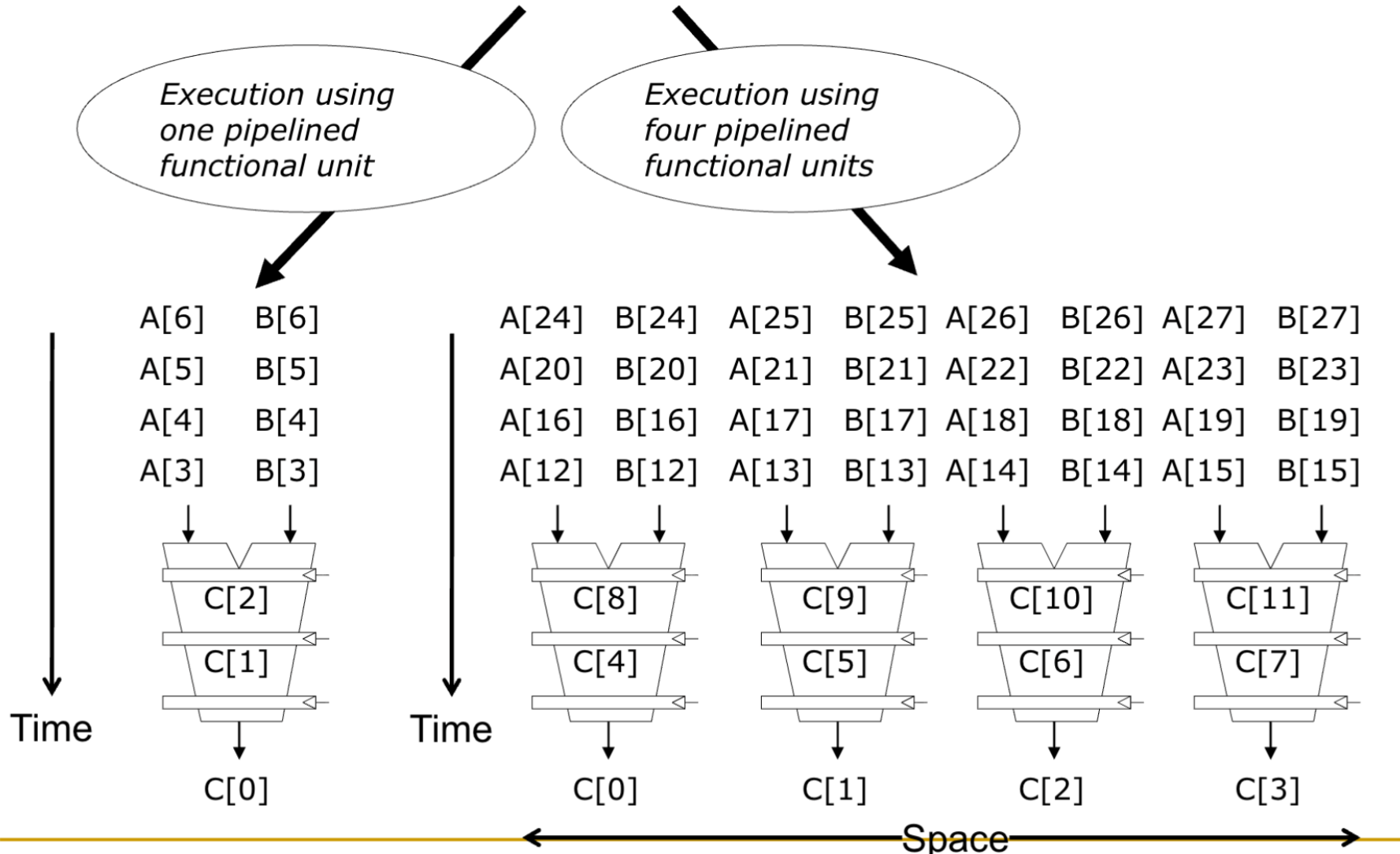
- Shared by multiple processing units / SIMD pipelines



GPU: Optimize Throughput of a Large Number of Simple Threads

Warp Execution

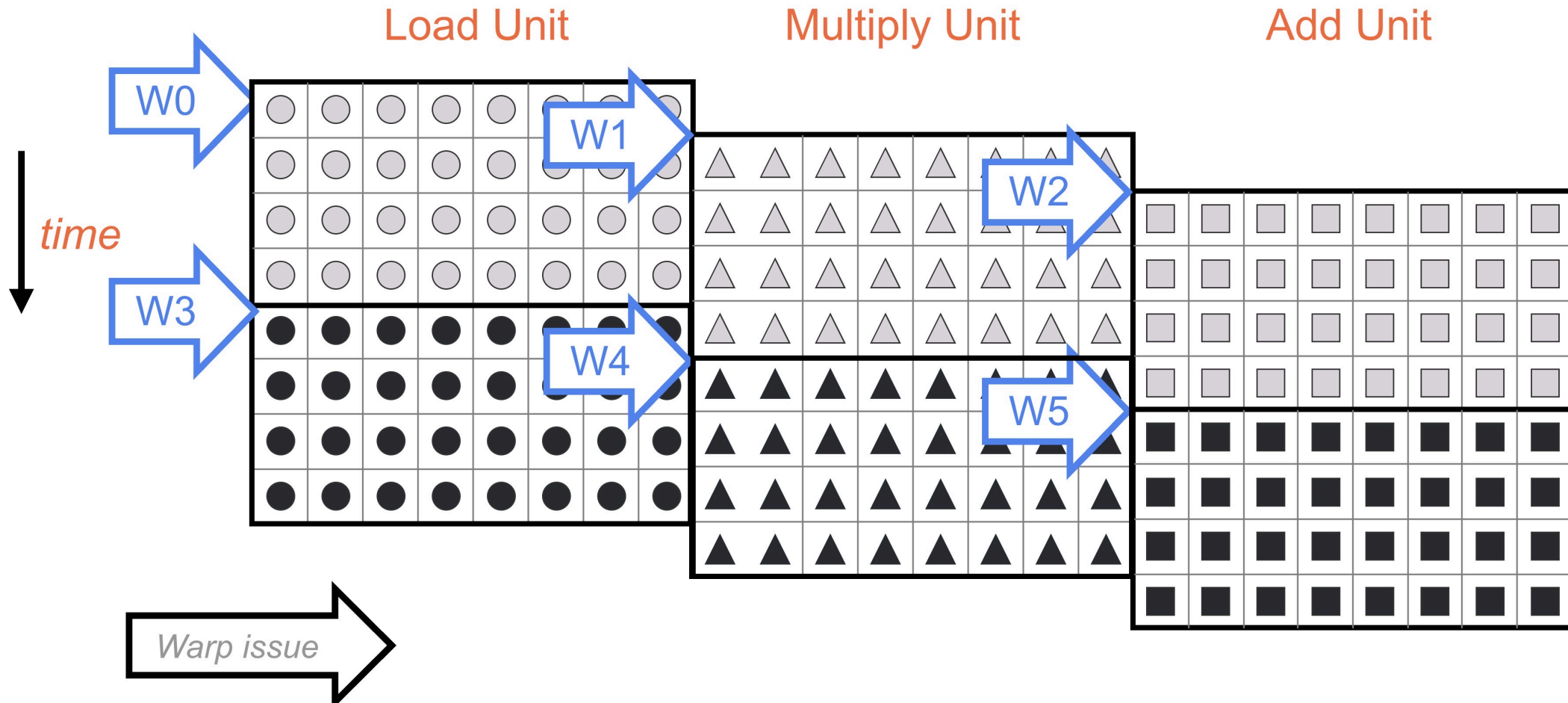
32-thread warp executing $\text{ADD } A[\text{tid}], B[\text{tid}] \rightarrow C[\text{tid}]$



GPU: Optimize Throughput of a Large Number of Simple Threads

Warp Execution

- Interleave execution among "warps" to hide latencies



GPU: Optimize Throughput of a Large Number of Simple Threads

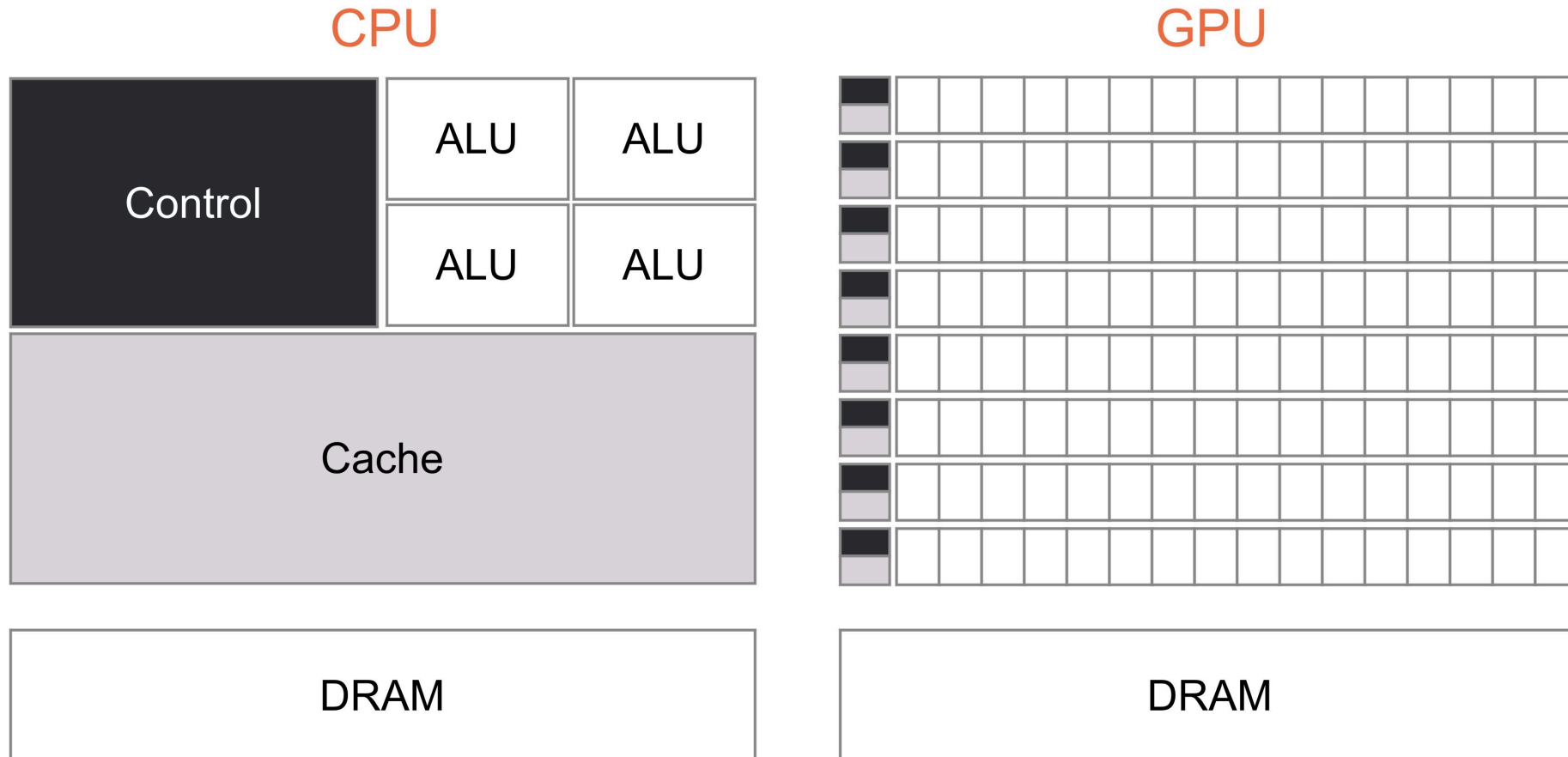
A Summary of GPU Terms

Generic Term	NVIDIA Term	AMD Term	Comments
Vector length	Warp size	Wavefront size	Number of threads that run in parallel on a SIMD unit
Pipelined unit / Scalar pipeline	CUDA core	-	Functional unit that executes instructions of one GPU thread
SIMD unit / SIMD pipeline	Group of N CUDA cores (e.g., N=16 in Fermi)	Vector ALU	SIMD unit that executes instructions for an entire warp
GPU core	SM	Compute Unit	Contains one or more control units (e.g., warp scheduler) and one or several SIMD pipelines

GPU: Optimize Throughput of a Large Number of Simple Threads

CPU vs GPU

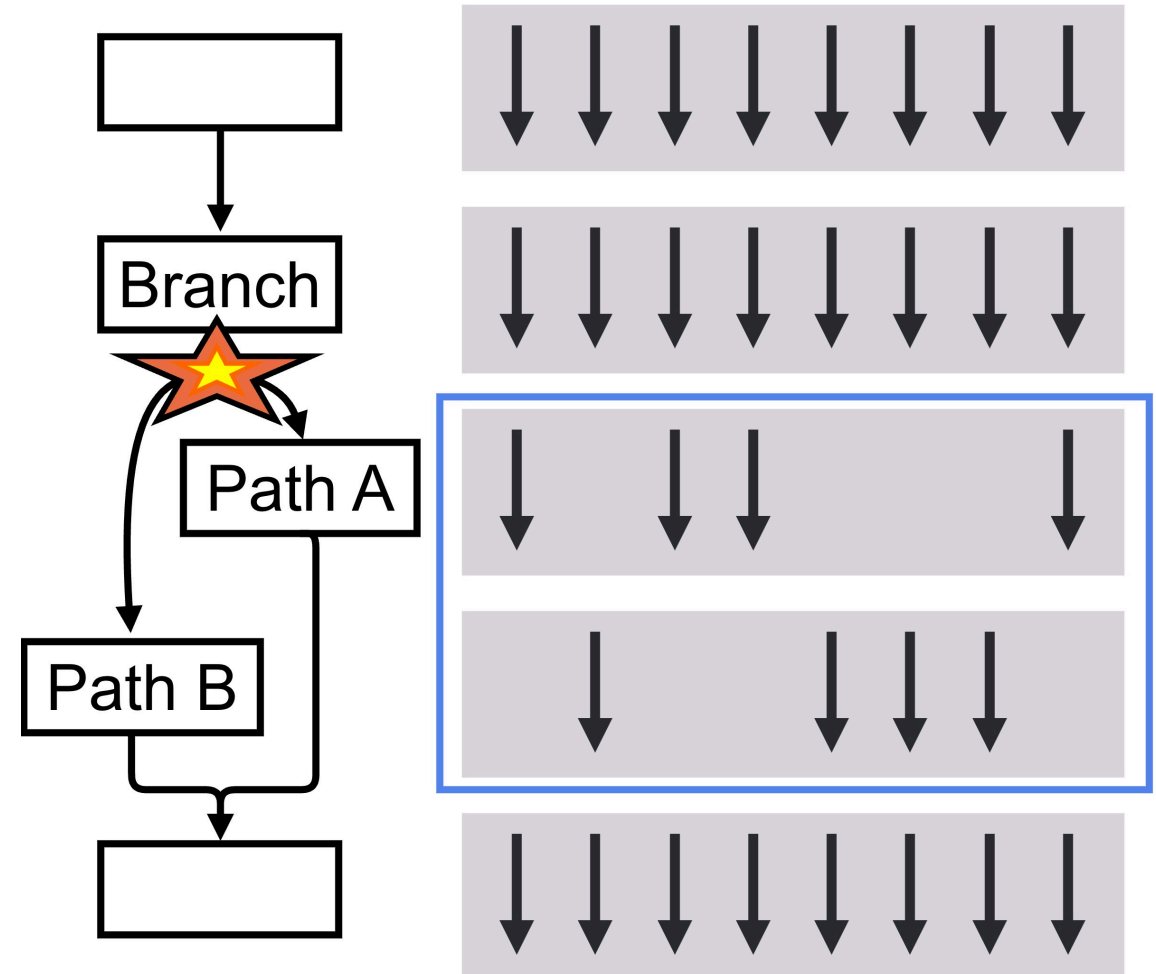
- **CPU:** A few out-of-order, superscalar, hyper-threading cores --- heavy cores
- **GPU:** Many in-order, SIMD-pipeline cores --- light cores
 - Enable 1,000 to 10,000 threads for parallel computing



GPU: Optimize Throughput of a Large Number of Simple Threads

Issue of SIMD Utilization: Control Divergence

- 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 path
- Result: Only a subset of threads execute, waste SIMD pipeline utilization

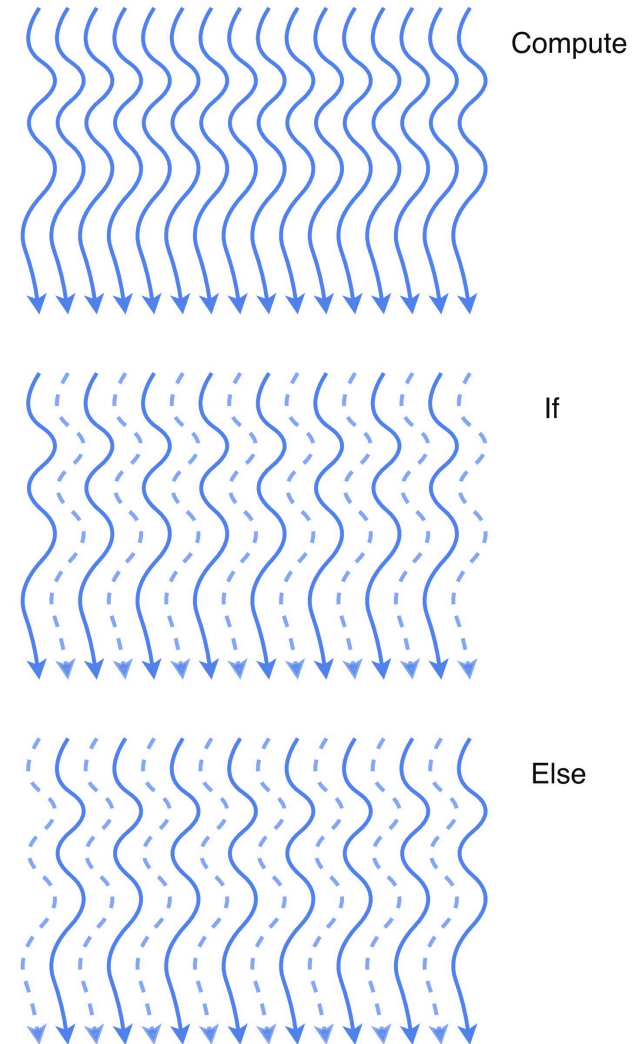


GPU: Optimize Throughput of a Large Number of Simple Threads

Issue of SIMD Utilization: Control Divergence

- Example: Intra-warp divergence

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

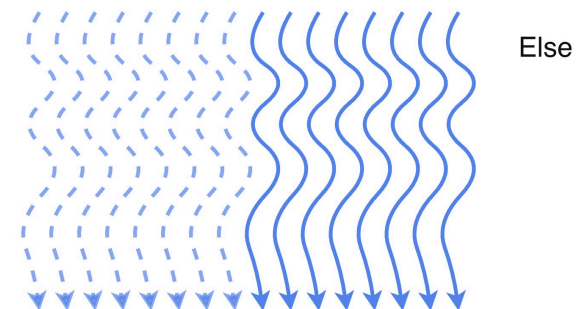
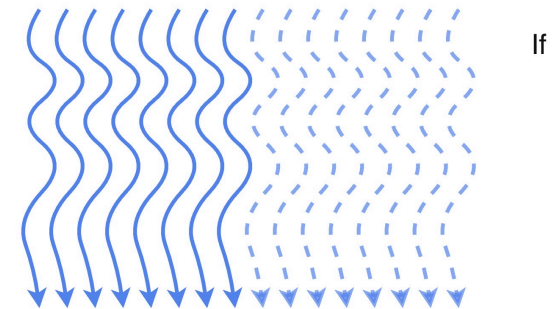
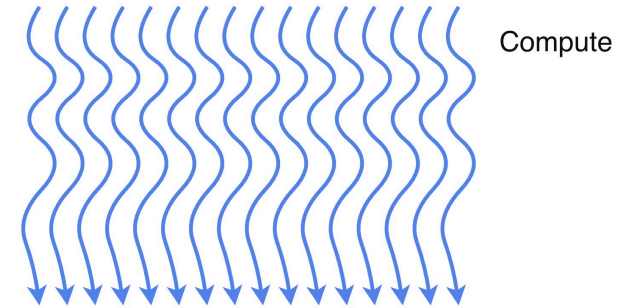


GPU: Optimize Throughput of a Large Number of Simple Threads

Issue of SIMD Utilization: Control Divergence

- Increasing SIMD utilization with **divergence-free** programs.

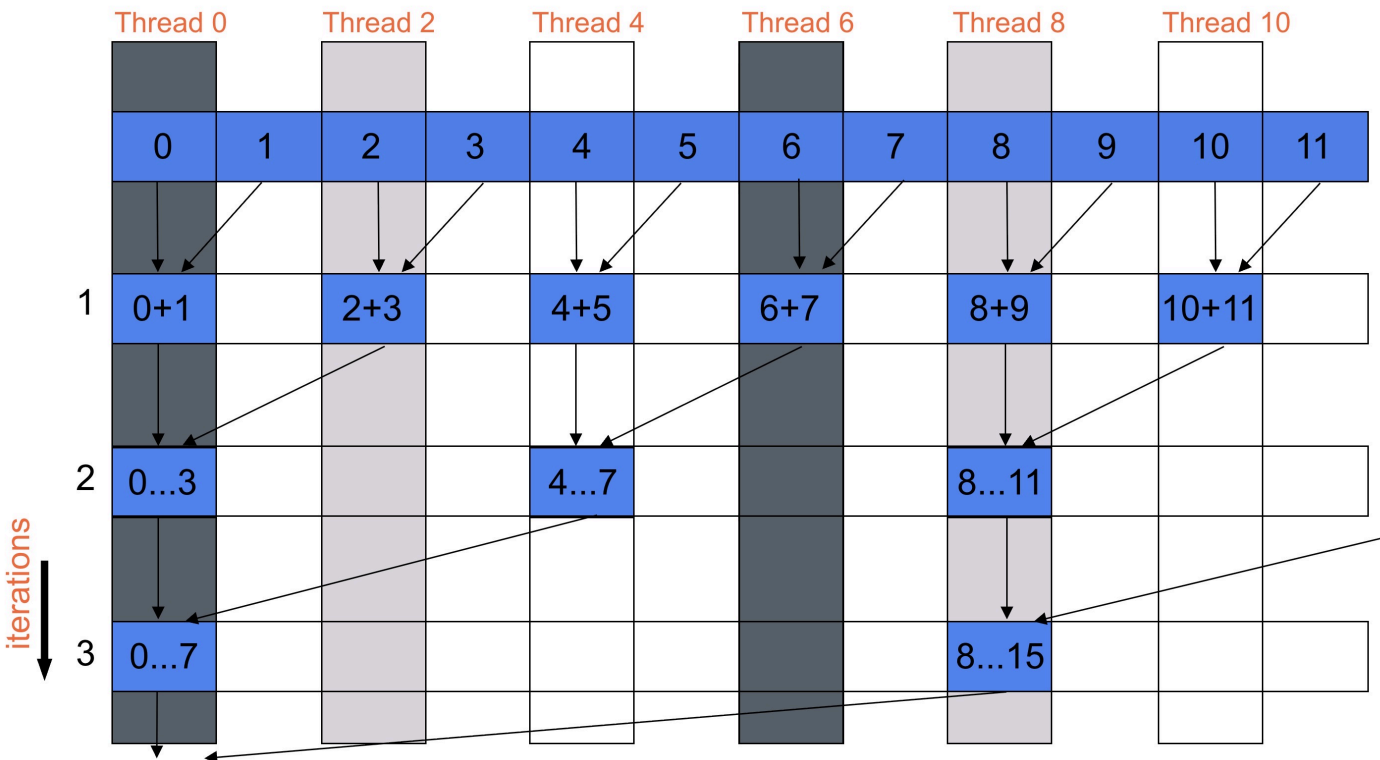
```
Compute(threadIdx.x);  
if (threadIdx.x < 32){  
    Do_this(threadIdx.x * 2);  
}  
else{  
    Do_that((threadIdx.x%32)*2+1);  
}
```



GPU: Optimize Throughput of a Large Number of Simple Threads

Control Divergence: Vector Reduction

- Program with low SIMD utilization
- High divergence.



```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

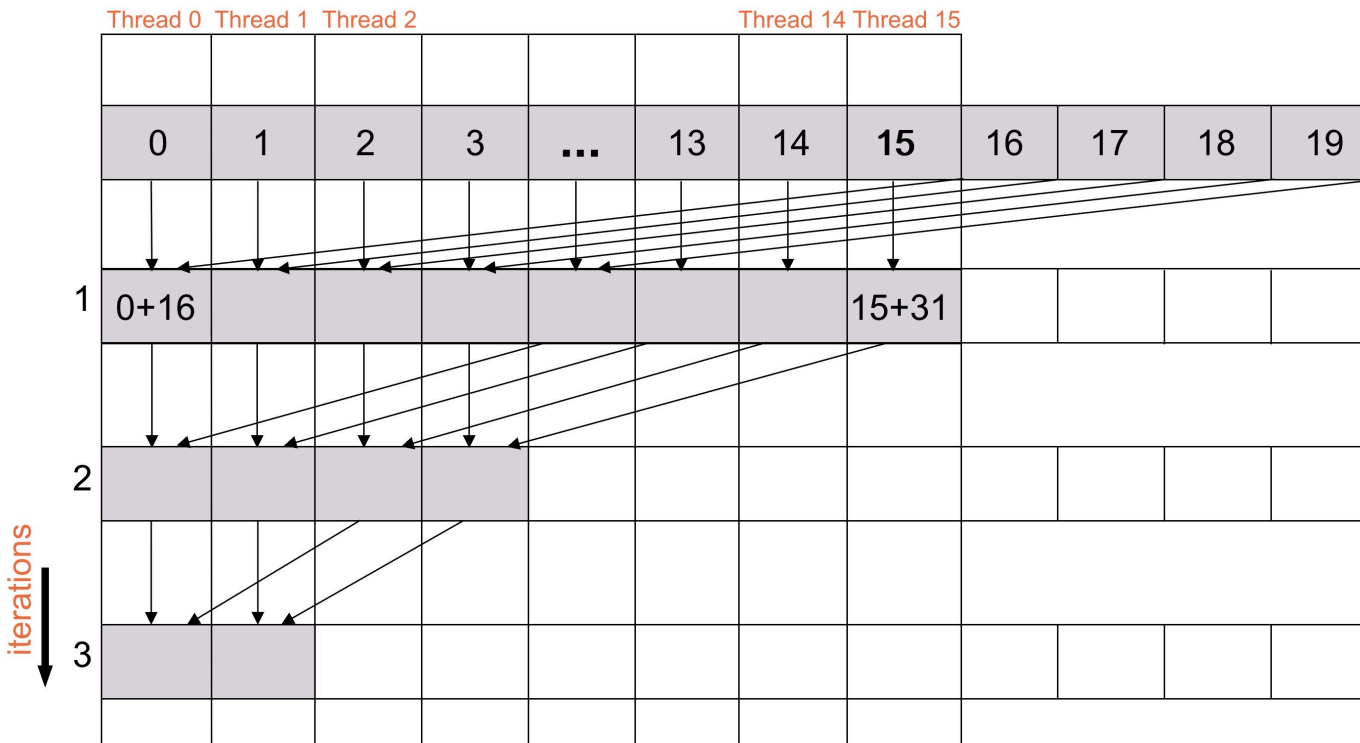
for (int stride = 1; stride < blockDim.x;
     stride *= 2) {

    __syncthreads();

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

GPU: Optimize Throughput of a Large Number of Simple Threads

Control Divergence: Vector Reduction



■ Optimized program

- All active threads belong to the same

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

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

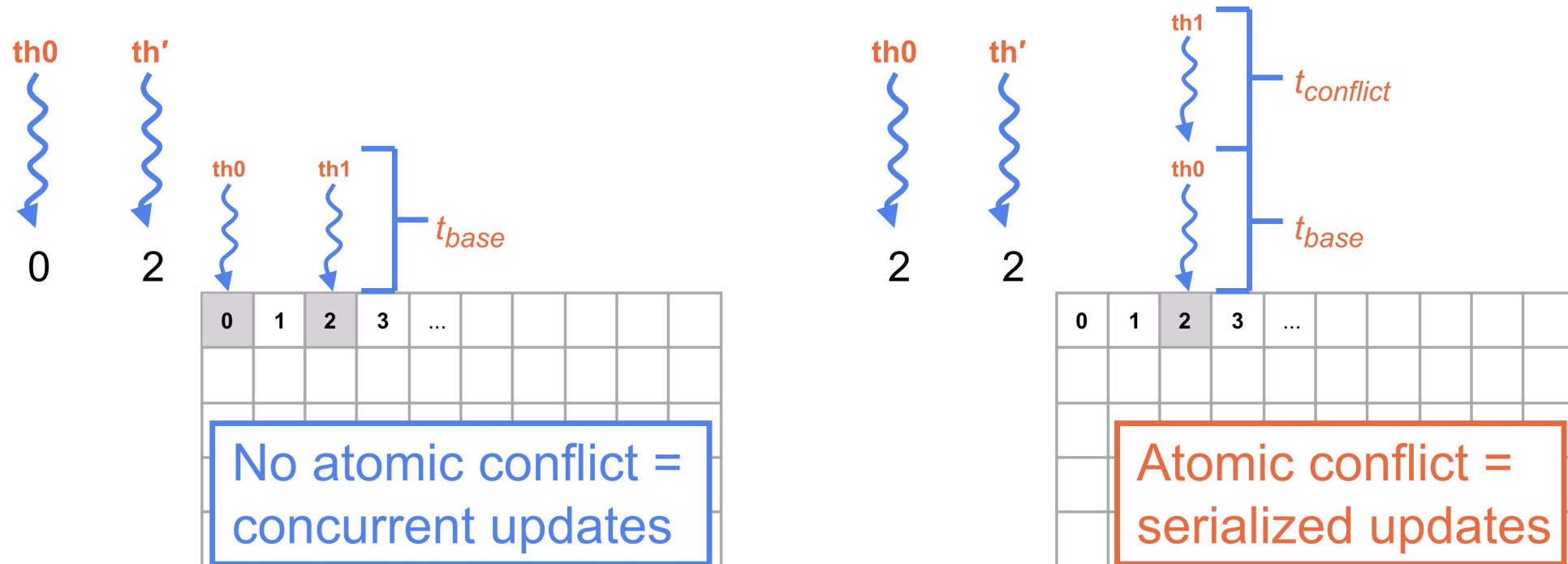
    __syncthreads();

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

GPU: Optimize Throughput of a Large Number of Simple Threads

Issue of SIMD Utilization: Atomic Operations

- Atomic operations are required when threads update the value of the same memory location
 - CUDA: `int atomicAdd (int*, int);`
- Atomic conflicts: the number of threads in a warp that update the same memory position



GPU: Optimize Throughput of a Large Number of Simple Threads

Atomic Operations: Histogram Calculation

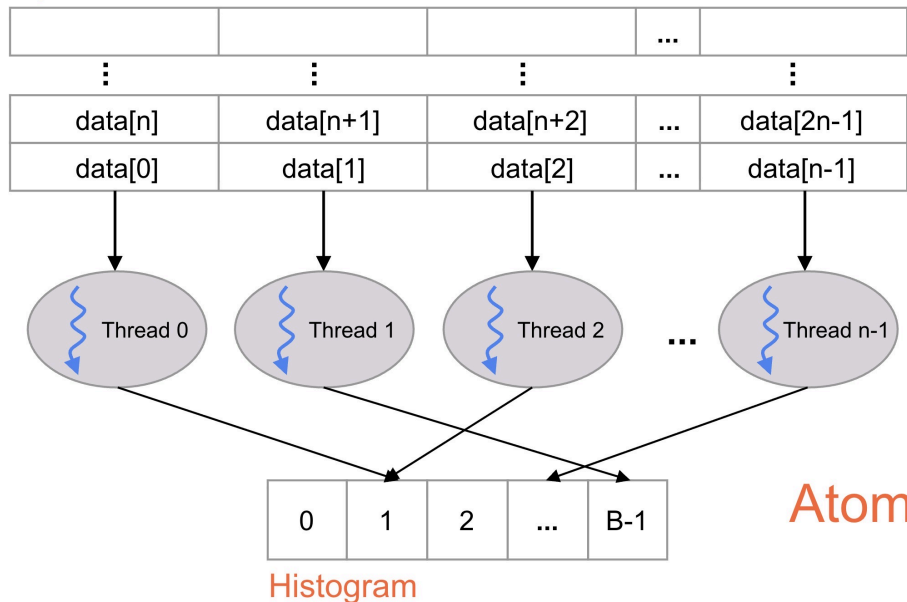
- Histograms count the number of data instances in disjoint categories (bins)

```

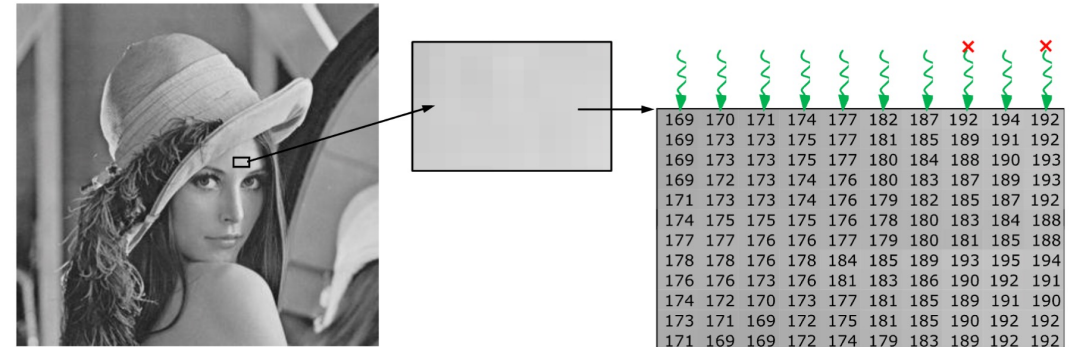
for (each pixel i in image I) {
    Pixel = I[i]                // Read pixel
    Pixel' = Computation(Pixel) // Optional computation
    Histogram[Pixel']++         // Vote in histogram bin
}

```

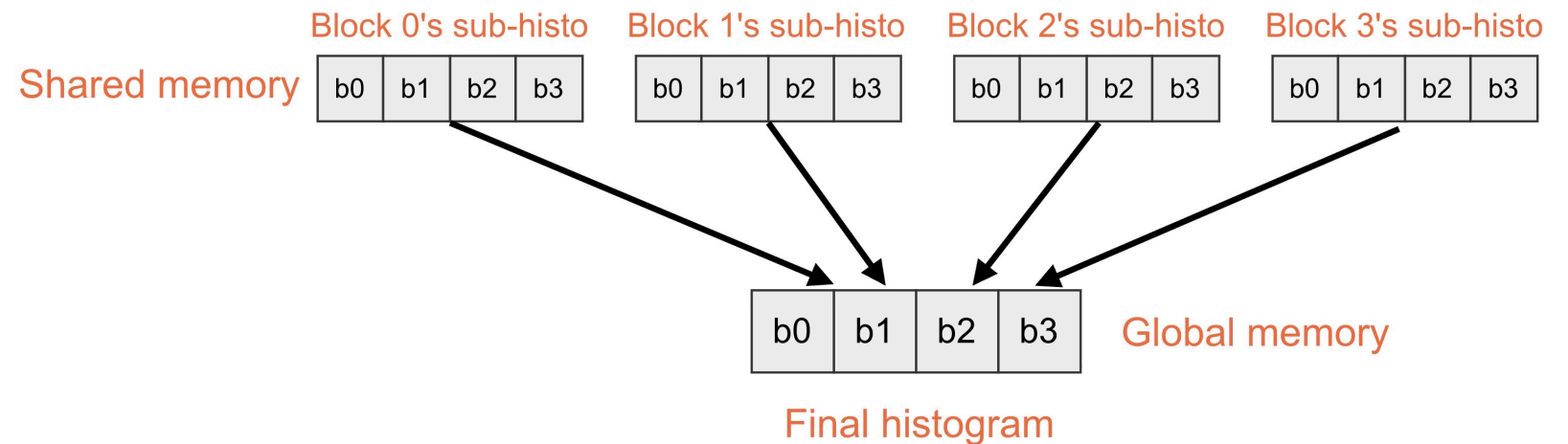
Input data



- Frequent conflicts in a natural images



- | | | | | | | |
|--|---------|--|--|---------|--|--|
| | Block 0 | | | Block 1 | | |
| | Block 2 | | | Block 3 | | |
| | Block 0 | | | Block 1 | | |
| | Block 2 | | | Block 3 | | |
| | Block 0 | | | Block 1 | | |
| | Block 2 | | | Block 3 | | |
| | Block 0 | | | Block 1 | | |
| | Block 2 | | | Block 3 | | |
| | Block 0 | | | Block 1 | | |
| | Block 2 | | | Block 3 | | |



GPU: Optimize Throughput of a Large Number of Simple Threads

Why AI Moved Toward GPU-Centric Computing?

- Scaling laws made compute the bottleneck
- Transformers made the computation massively data-parallel

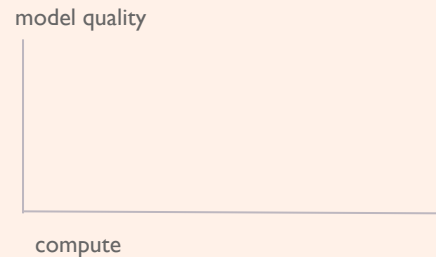
LINE 1: DEMAND

Scaling laws turn AI into sustained FLOP demand

Empirical loss improves predictably as compute, parameters, and data increase.

Scales model size and token count together.

The hardware target becomes high-throughput training, not low-latency single-thread execution.



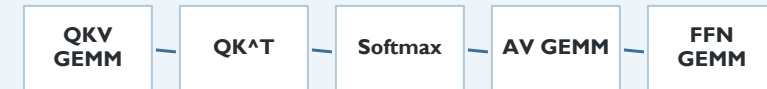
LINE 2: STRUCTURE

Transformers expose SIMD/SIMT-friendly work

Most expensive layers are dense tensor kernels:
e.g., QKV projection, QK^T attention scores, AV product, FFN GEMM.

Parallel axes: batch, token, head, hidden dimension.

Control flow is small; arithmetic intensity is high when tiled well.



GPU-centric execution

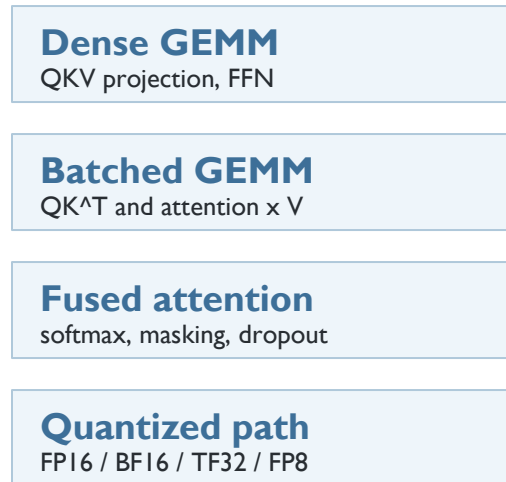
many simple threads + high bandwidth + tensor units

CPU remains the host controller; GPU becomes the dominant floating-point compute engine.

GPU: Optimize Throughput of a Large Number of Simple Threads

Why NVIDIA SM Outperforms Other GPUs for Transformer Kernels?

TRANSFORMER KERNELS



Streaming Multiprocessor (SM)

Warp schedulers

select ready warps; hide latency

Registers

per-thread state for many warps

Shared memory / L1

explicit tiling and data reuse

Tensor Cores

MMA path for GEMM / attention

FLASHATTENTION AS EXAMPLE

IO-aware attention

standard

materialize $S \times S$ scores in HBM

FlashAttention

tile Q/K/V through SRAM

result

fuse softmax; reduce HBM traffic

Parallel Programming Models: pThread and CUDA

Lecture 1: Processor Evolution — From CPU to GPU



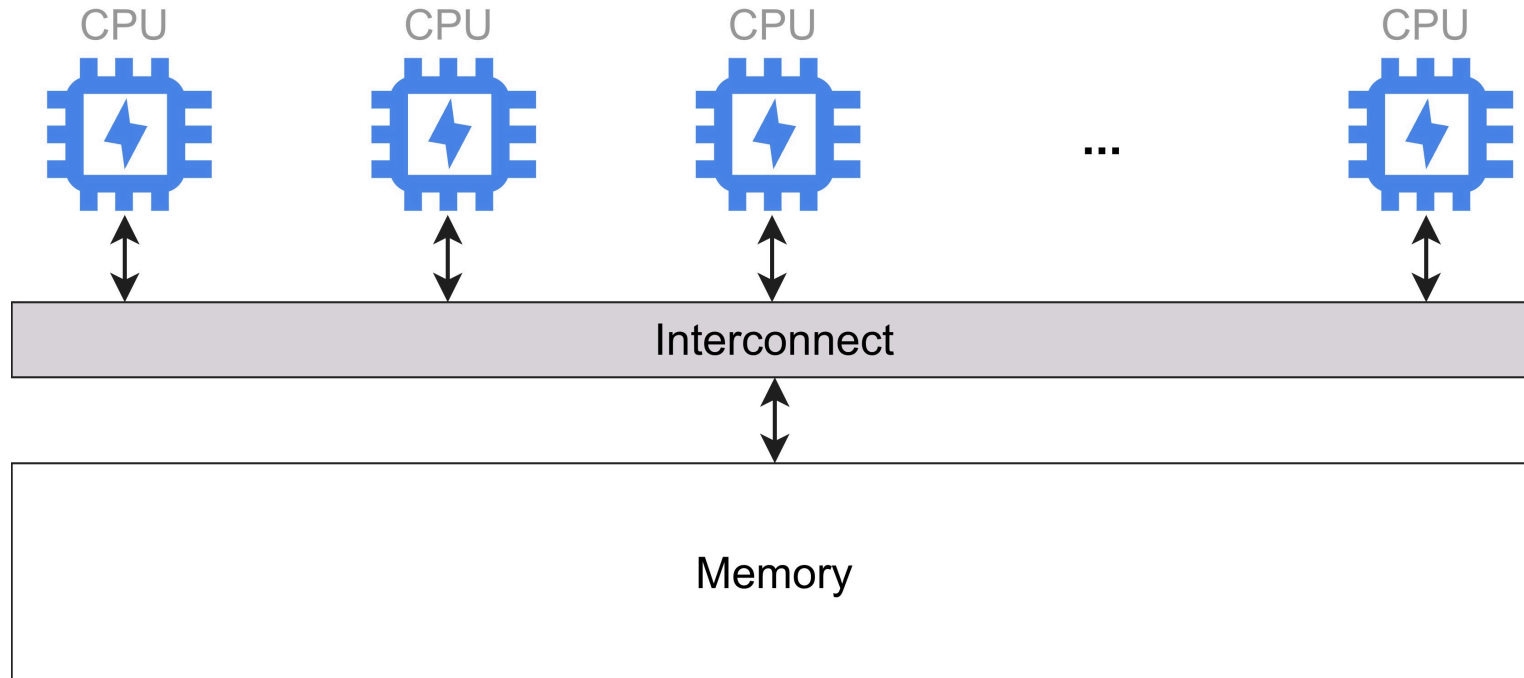
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

CPU Multicore Programming: pThread

Thread Parallelism with Shared Memory Programming Model

- Break down a complex task into multiple lightweight sub-tasks.
 - One process contains multiple threads.
 - Threads share the same virtual address space, global variables and heap objects are shared.
 - Synchronization is the programmer's responsibility.



CPU Multicore Programming: pThread

Programming with POSIX threads

- POSIX threads, also known as pThread library.
 - A standard for Unix-like operating systems.
 - Linked with C/C++ programs.
- Example: parallel reduction with a fork-join pattern.
 - Partition input into per-thread chunks.
 - Compute local partial sums independently.
 - Join workers and reduce partial results.

```

main thread
  |
  | partition input into T chunks
  v
pthread_create worker threads
  |
  v
worker 0      worker 1      ...      worker T-1
  |            |            |            |
  | local sum  | local sum  |            | local sum
  |            |            |            |
  v            v            v            v
main thread waits using pthread_join()
  |
  v
reduce partial sums and return total
  
```

```

size_t chunk = (n + T - 1) / T;

for (int tid = 0; tid < T; ++tid) {
    tasks[tid] = (Task){
        x,
        tid * chunk,
        min(n, (tid + 1) * chunk),
        0.0
    };

    pthread_create(&threads[tid], NULL,
        partial_sum, &tasks[tid]);
}

double total = 0.0;
for (int tid = 0; tid < T; ++tid) {
    pthread_join(threads[tid], NULL);
    total += tasks[tid].partial;
}
  
```

```

void *partial_sum(void *arg) {
    Task *t = (Task *)arg;
    double sum = 0.0;

    for (size_t i = t->begin; i < t->end; ++i)
        sum += t->x[i];

    t->partial = sum;
    return NULL;
}
  
```

CPU Multicore Programming: pThread

Race Condition and Critical Section

■ Example

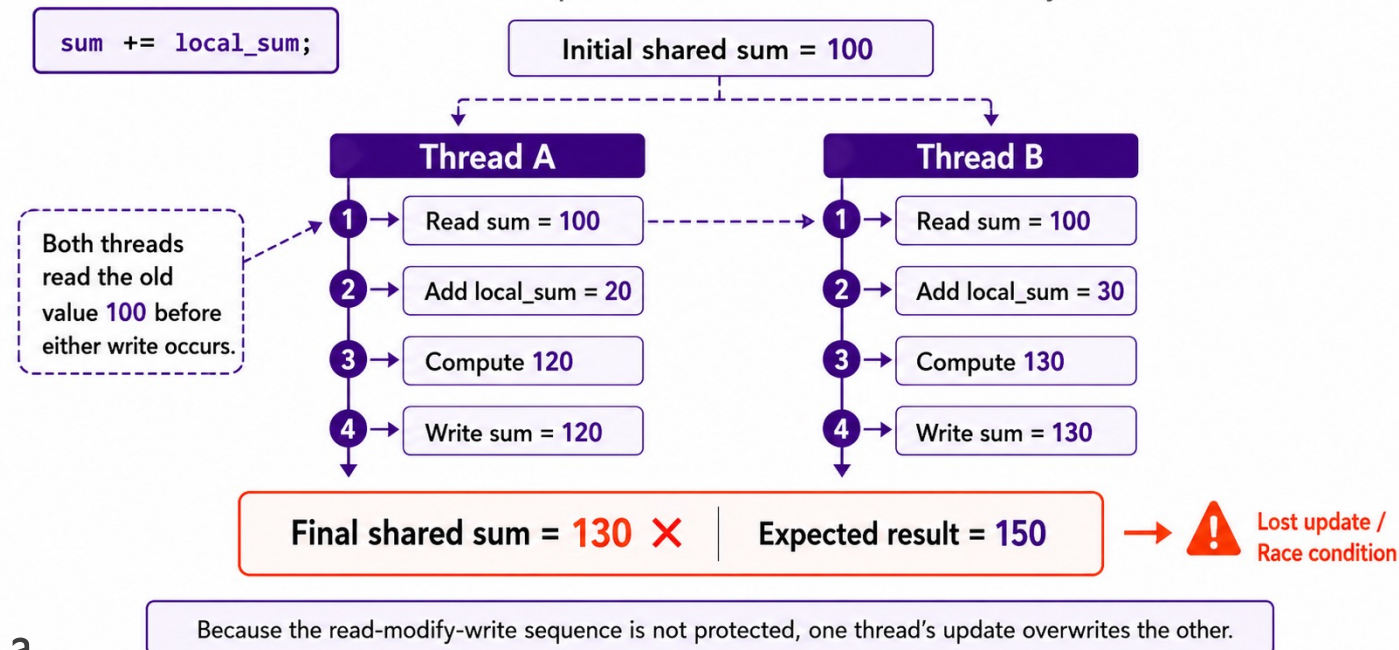
- `sum += local_sum;`
- For each thread
 - Load `sum`
 - Add `local_sum`
 - Store `sum`
- If two threads execute concurrently
 - both may read the old value;
 - both compute new values;
 - one write overwrites the other.



- **Race Conditions:** multiple threads access a shared resource and final result depends on timing.
- A **critical section** is code that must be executed by only one thread at a time.

Race Condition Example

Two threads update a shared variable concurrently



CPU Multicore Programming: pThread

Mutex: Protect Shared State

- Mutex pattern:

```
pthread_mutex_lock(&lock);
sum += local_sum;
pthread_mutex_unlock(&lock);
```

- The Mutex provides:

- Serialized access to critical section
- Race avoidance for protected shared state

- Cost: Lock contention

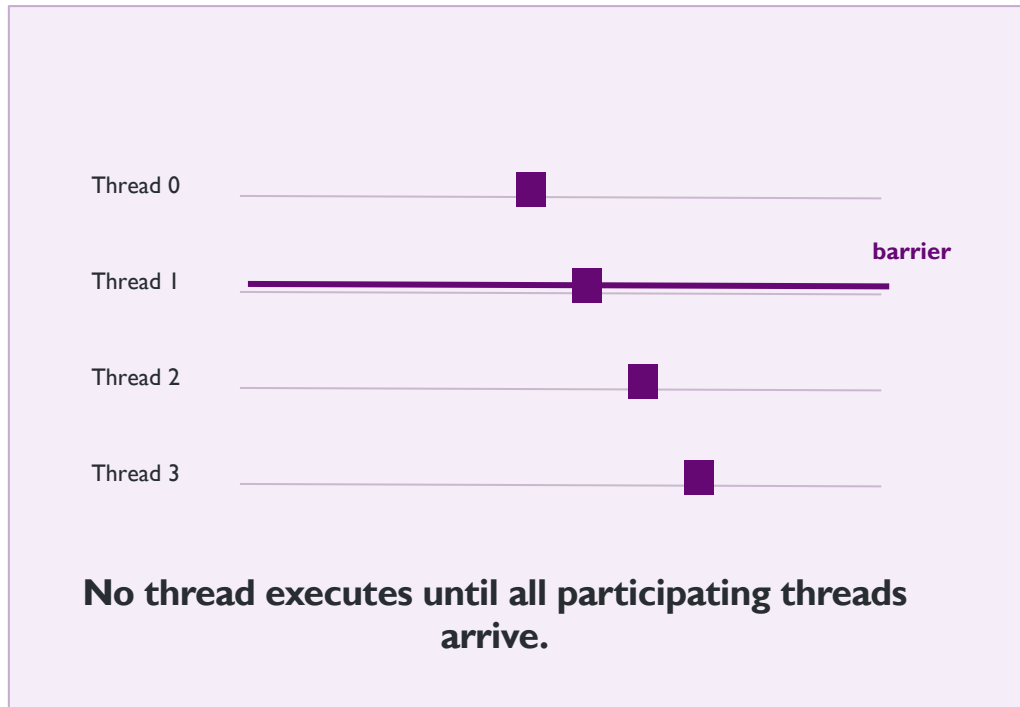
Time	flag	Thread				
		0	1	2	3	4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
:	:			:	:	:
?	2	—	—	crit sect	susp	busy wait

Only one thread holds the mutex; others wait and the protected region is serialized

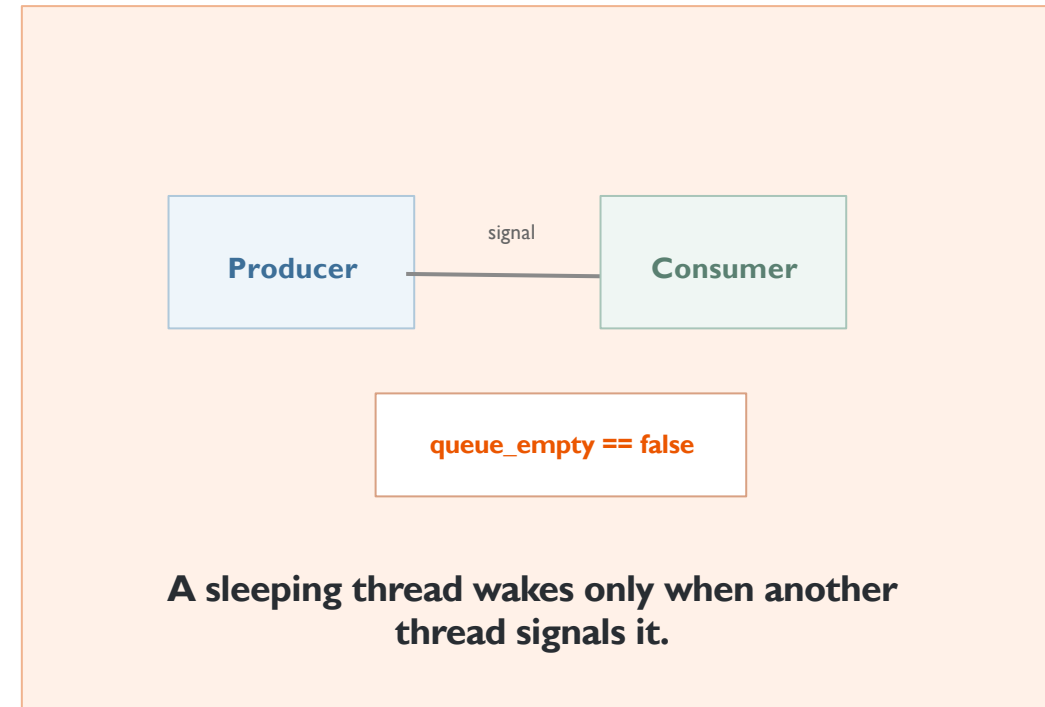
CPU Multicore Programming: pThread

Barriers and Condition Variables

- **Barrier:** All threads wait until everyone reaches the same point.
 - Use cases: phase-based algorithms, debugging, iterative solvers, etc.



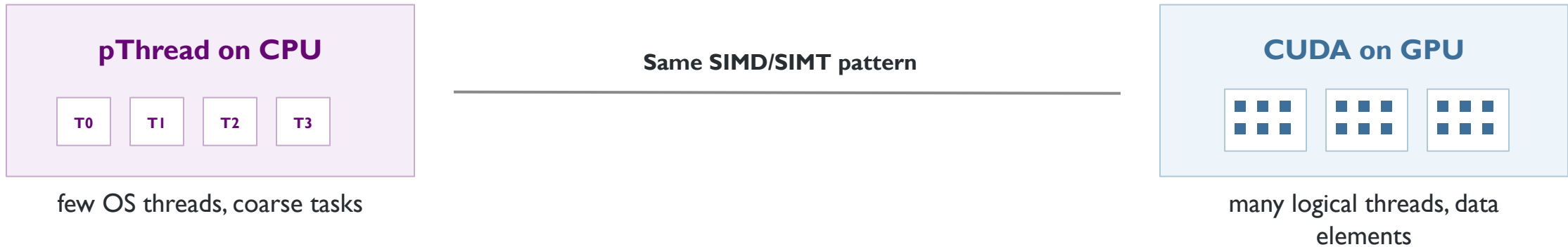
- **Condition variable:** A thread sleeps until a condition becomes true.
 - Use cases: worker pools, event-driven scheduling, etc.



CPU Multicore Programming: pThread

From pThread to CUDA: Same Idea, Different Granularity

- Both divide work across parallel workers, but CUDA makes the worker **much smaller and more numerous**.

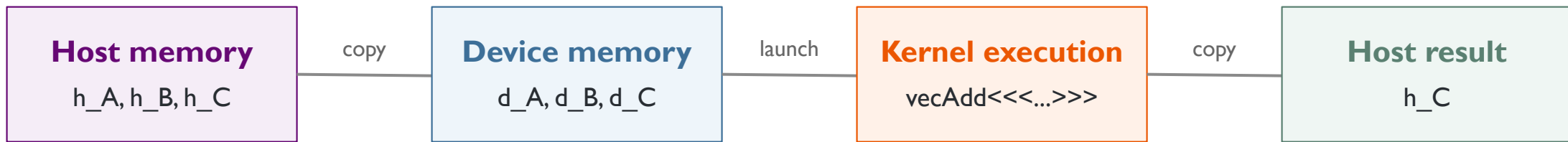


Question	pThread	CUDA
Work unit	OS thread / task	CUDA thread / data element
Launch API	pthread_create + pthread_join	kernel<<<grid, block>>>()
Memory model	shared virtual address space	host memory + device memory
Sync style	mutex, barrier, condition variable	block barrier + kernel boundary

GPU Programming: CUDA

CUDA Program Structure

- CUDA starts on the CPU: host code manages device memory, data movement, and kernel launch.



```
cudaMalloc(&d_A, bytes);
cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
vecAdd<<<numBlocks, blockSize>>>(d_A, d_B, d_C, n);
cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
cudaFree(d_A);
```

Important detail

Kernel launch is asynchronous.

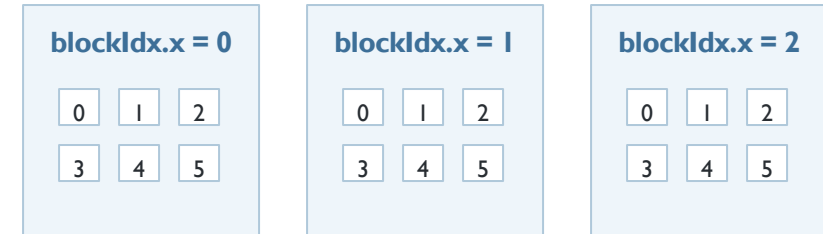
Use `cudaDeviceSynchronize()` or a blocking copy/event when host code needs the result.

GPU Programming: CUDA

CUDA Kernel Definition and Thread Indexing

Kernel Definition

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



`blockDim.x` is the number of threads per block.
`threadIdx.x` is the local index inside that block.

- Every CUDA thread runs the same kernel code; built-in IDs select the data element.

$$i = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}$$

GPU Programming: CUDA

CUDA Synchronization

- `__syncthreads()`
 - Scope: One CUDA block
 - Use: Tile coordination in the shared memory.
- Kernel boundary
 - Scope: Whole threads
 - Use: Global phase between kernels
 - Achieved by launching separate kernels

```
__shared__ float tile[256];  
tile[threadIdx.x] = A[i];  
__syncthreads();  
use(tile);
```



- Load data from global memory to shared memory.
- Wait until all threads in the block complete the shared-memory load.

Performance Models

Lecture 1: Processor Evolution — From CPU to GPU

Performance Models

Performance Is Not Peak FLOPs

- Upper Bound: Hardware Throughput
 - Real Performance: Limited by **Load Imbalance**, **Pipeline Efficiency**, and **Memory Access**
- Speedup of a parallel program
 - P = The number of cores
 - $T(I)$ = Serial running time
 - $T(P)$ = Parallel running time
 - $\text{Speedup} = T(I) / T(P)$
- Efficiency = Speedup / P



	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

- Speedup and efficiency usually vary with problem size and the number of involved cores

Performance Models

Amdahl's Law: The Serial Fraction Wins

- When problem size is fixed, even a small serial fraction caps total speedup

$$S(p) = 1 / ((1 - f) + f / p)$$

f: parallelizable fraction

1 - f: serial fraction

p: number of processors / threads

NUMERICAL INTUITION

f = 0.90

max speedup -> 10x

f = 0.99

max speedup -> 100x

FIXED PROBLEM SIZE

serial

parallelizable work

More processors squeeze the blue part, but the purple serial part remains.

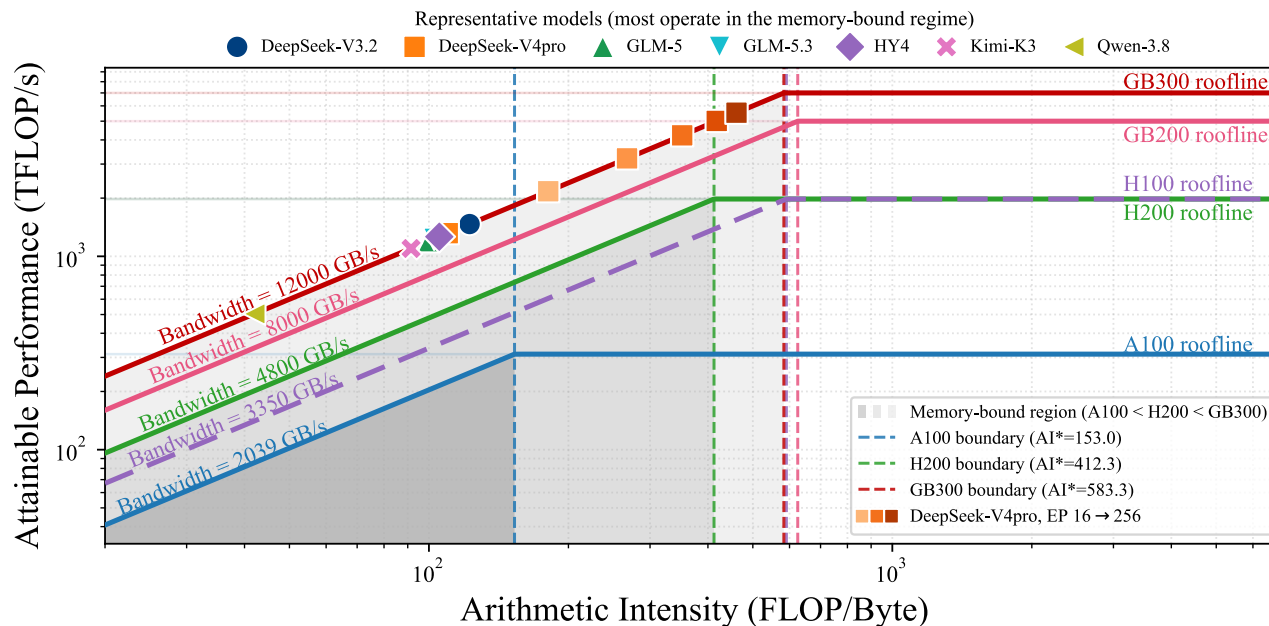
Performance Models

Roofline Model

- Memory wall: how much computation it performs for each byte moved.

$$\text{Arithmetic Intensity} = \text{FLOPs} / \text{Bytes moved from memory}$$

- Roofline Model: Compute-Bound or Memory-Bound?



Why peak is hard

- memory-bound: limited by bandwidth x intensity
- compute-bound: still needs vectorization, tensor cores, occupancy
- real ceilings fall with poor locality or pipeline utilization