Accelerator Architectures for Machine Learning (AAML)

Lecture 7: GPGPU

Tsung Tai Yeh
Department of Computer Science
National Yang-Ming Chiao Tung University

Acknowledgements and Disclaimer

- Slides was developed in the reference with
  Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019 tutorial
  Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
  Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC Berkeley, 2020
  CS231n Convolutional Neural Networks for Visual Recognition, Stanford University, 2020
  CS224W: Machine Learning with Graphs, Stanford University, 2021
  CS 15-779: Advanced Topics in Machine Learning Systems, CMU, 2025

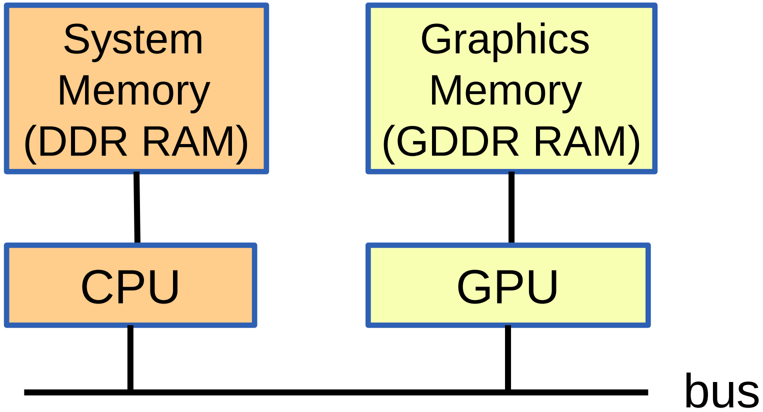
Outline

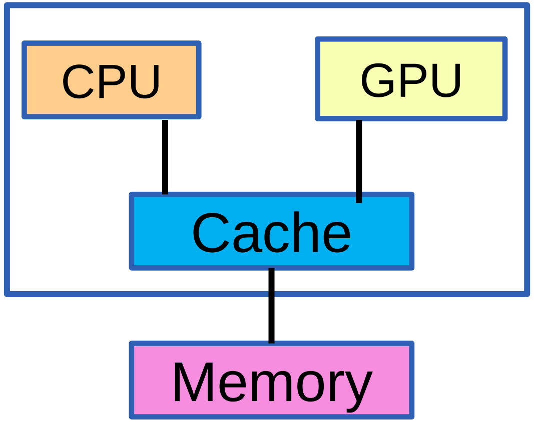
- GPU hardware basics
- Programming Model
- The SIMT Core
  - Warp Scheduling
  - Functional Unit
  - Operand collector

What is GPU?

- GPU = Graphics Processing Units
- Accelerate computer graphics rendering and rasterization
- Highly programmable (OpenGL, OpenCL, CUDA, HIP etc..)
- Why does GPU use GDDR memory?
    - DDR RAM -> low latency access, GDDR RAM -> high bandwidth

Discrete GPU

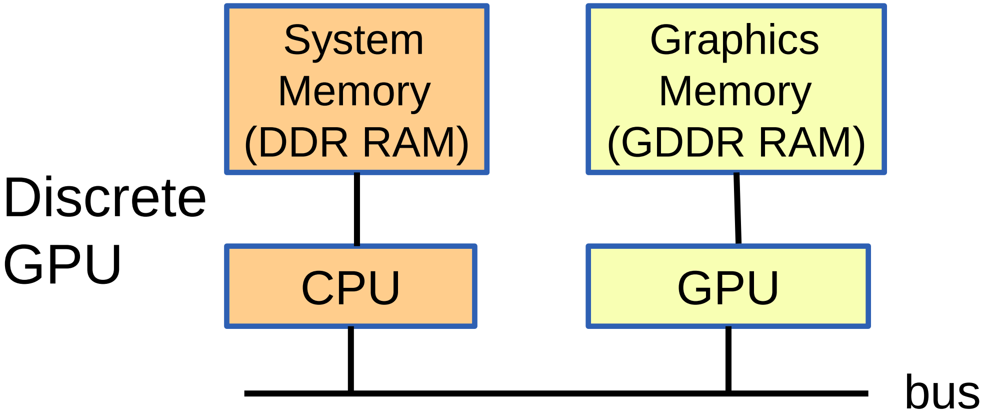




Integrated GPU

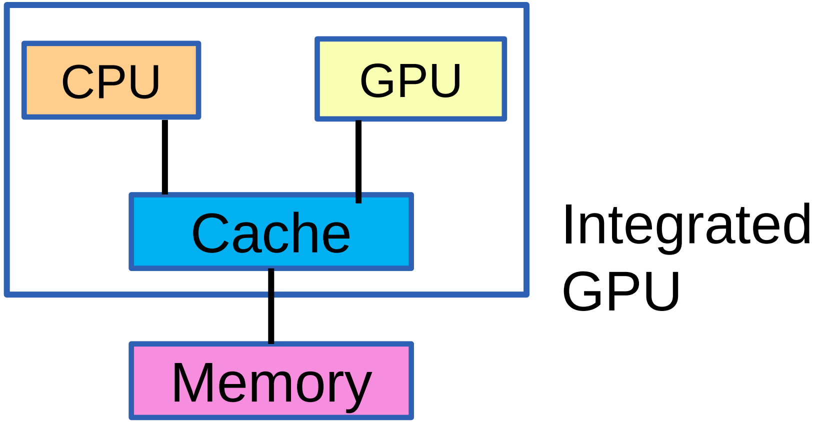
Discrete GPU

- A (PCIe) bus connecting the CPU and GPU
- Separate DRAM memory spaces
    - CPU (system memory) and the GPU (device memory)
- DDR for CPU vs. GDDR for GPU
    - CPU DRAM optimizes for low latency access
    - GPU DRAM is optimized for high throughput



Integrated GPU

- Have a single DRAM memory space
- Often found on low-power mobile devices
  - Ex. AMD APU
  - Private cache -> cache coherence



CPU vs GPU

	Cores	Clock Speed	Memory	Price	Speed
CPU (Intel Core i7-7700k)	4	4.2 GHz	DDR4 RAM	$385	~540 GFLOPs F32
GPU (Nvidia RTX 3090 Ti)	10496	1.7 GHz	DDR6 24 GB	$1499	36 TFLOPs F32

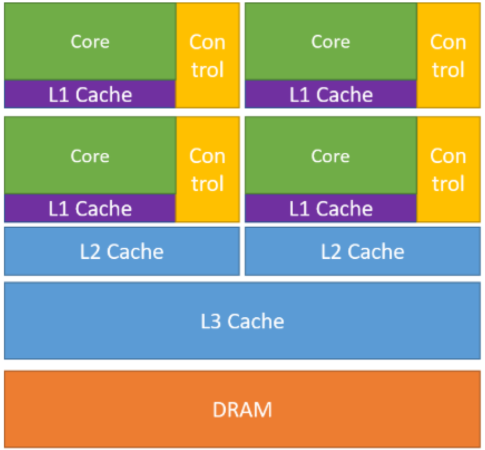
CPU: A **small** number of **complex** cores, the clock speed of each core is high, great for sequential tasks
GPU: A **large** number of **simple** cores, the clock speed of each core is low, great for parallel tasks

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htm

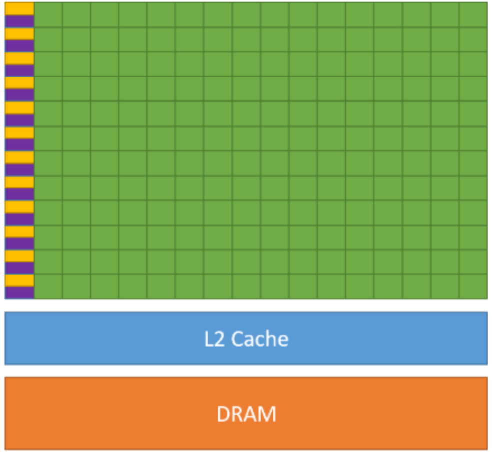
Why do we use GPU for computing ?

- What is difference between CPU and GPU?
  - GPU uses a large portion of silicon on the computation against CPU
  - GPU (2nJ/op) is more energy-efficient than CPU (200 pJ/op) at peak performance
  - Need to map applications on the GPU carefully (Programmers' duties)

CPU

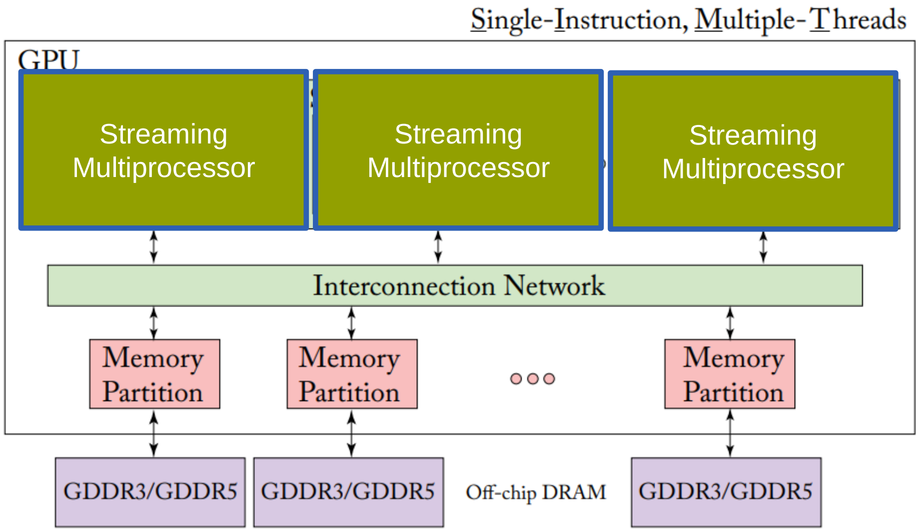


GPU



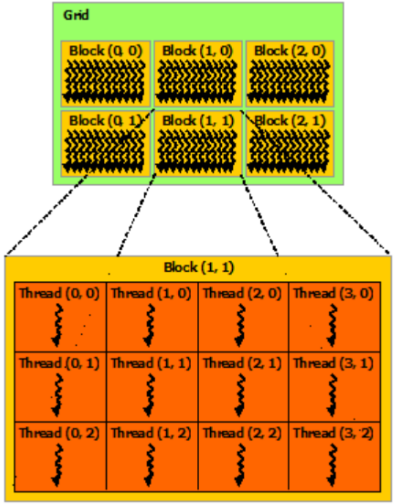
Modern GPU Architecture

- A modern GPU is composed of many cores
  - Streaming multiprocessors (SM) (Nvidia) or compute units (CU) (AMD)
- A GPU
  - Executes a single-instruction multiple-thread (SIMT) program (kernel)
- A streaming multiprocessor
  - Threads are interleaving on each SM
  - Has a local scratch memory and data cache



The GPU Thread

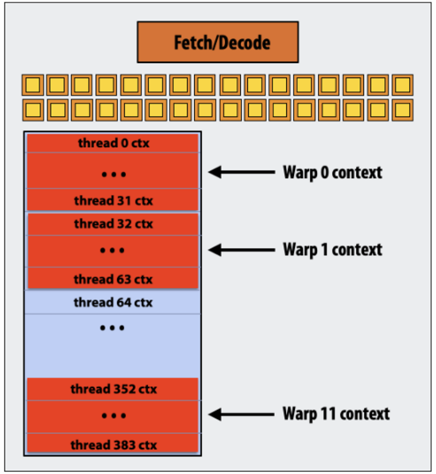
- **Thread**
  - The smallest unit of execution in CUDA
  - All threads execute the same kernel code (a function that runs on the GPU).
    - Threads can be identified using unique IDs, accessible through built-in variables like `**threadIdx**`.
    - Has its own set of registers and local memory.
  - The threads executing on a single core
    - Can communicate through a scratchpad memory
    - Synchronize using fast barrier operations

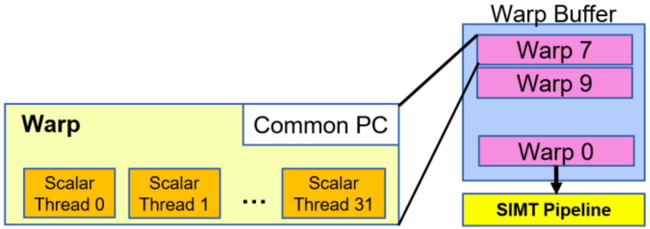


https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

The GPU Warp

- **Warp**
  - Group threads in a warp (32 threads)
  - The GPU executes instructions in each warp
  - Execute in lockstep
  - Warp buffer stores multiple warps
    - Interleaving warp execution to hide off-chip memory access latency and reduce the idle of GPU compute cores

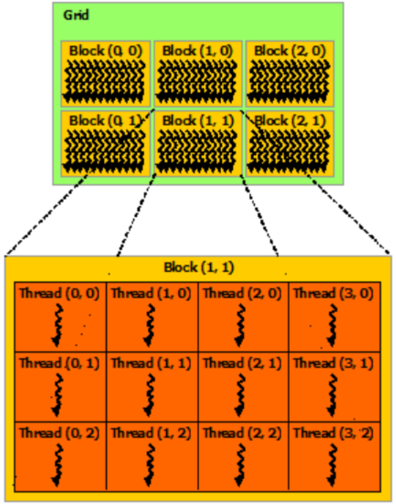




https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

The GPU Thread Block

- **Thread Block**
  - A block is a group of threads that execute together on the **same** Streaming Multiprocessor (SM)
  - A thread block contains multiple warps/threads
  - Specifically, block is a 3D array of threads
  - Threads within a block can communicate and synchronize with each other using shared memory and synchronization primitives like `__syncthreads()`
  - A thread block can have 1024 threads at most

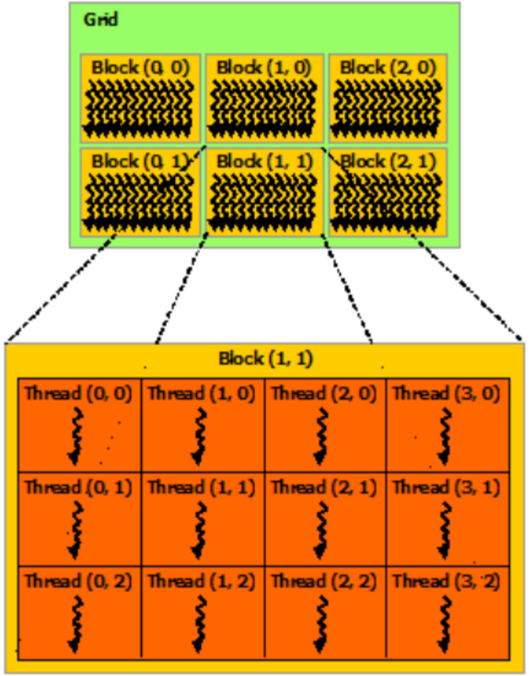


https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

The GPU Grid

- A **"grid"** can have multiple blocks
  - A grid is identified using `gridDim`: total number executing a given kernel.
- How to declare threads/blocks in GPU codes?
  - Blocks are organized as three dimensional grid of thread block

```
#define THREADS_PER_BLK 128
convolve<<<N/THREADS_PER_BLK, THREADS_PER_BLK>>>(N, input_array, output_array);
```



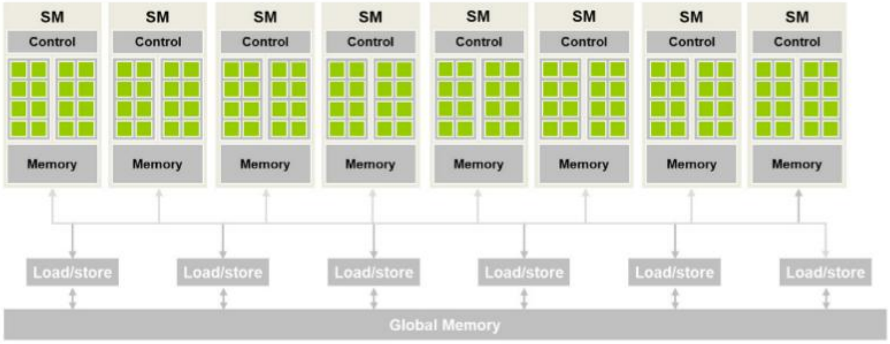
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Modern GPU Architecture

- Single Instruction Multiple Threading (SIMT)
    - **Multiple threads** execute the **same instruction** at the same time (like SIMD), but each thread can take its own path based on its data (like MIMD).
- Highly multithreaded GPU
    - Cover the long latency of memory loads and texture fetches from DRAM
    - Virtualize the physical processors as threads and thread blocks to provide transparent scalability
    - Support **fine-grained** parallel graphics shader programming models / computing programming models

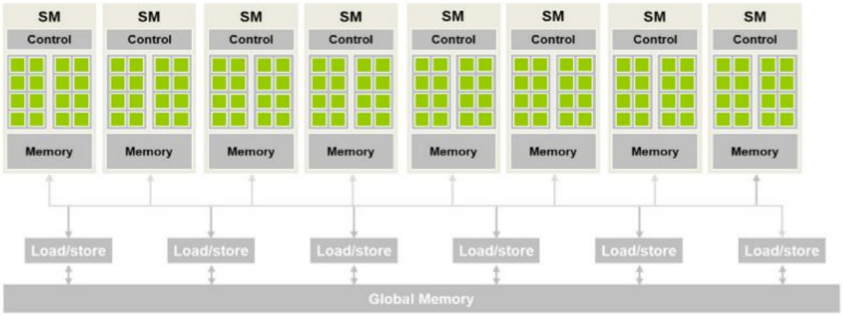
Modern GPU Architecture

- The compute core in each stream multiprocessor (SM)
  - In-order pipelined ALU
  - GPUs fill the pipeline effectively by **interleaving instructions** from different **warps** (32 threads) -> SIMT
  - Instruction dependency stalls the warp, because threads in a warp **execute in lockstep**



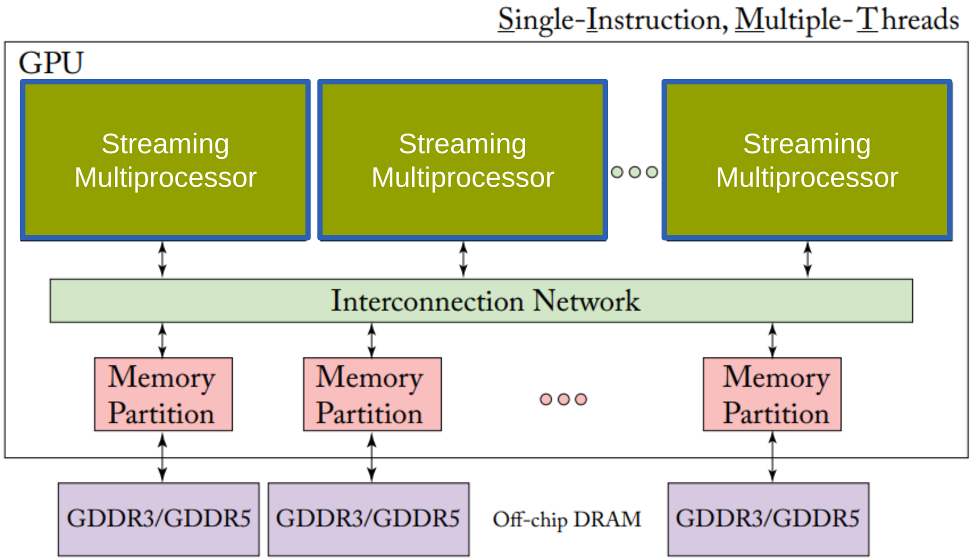
Modern GPU Architecture

- GPU Warp Execution
  - Instruction dependency stalls the warp, because threads in a warp **execute in lockstep**
  - GPU mitigates the impact of these stalls by **having multiple warps ready to execute**
  - When one warp stalls, the GPU switches to another warp, helping to fill the pipeline gaps caused by stalls and maintaining high utilization of the execution units.
  - Hide off-chip memory access latency -> interleaving warp execution



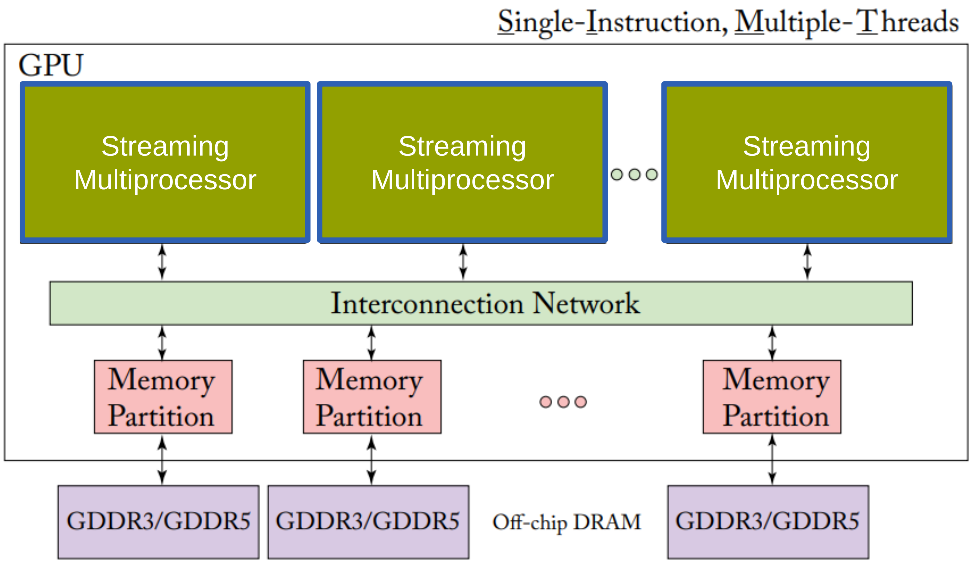
Modern GPU Architecture

- The GPU thread hierarchy
  - A warp (32 threads) -> a thread block (CTA) (< 32 warps) -> grid
- Each SM has a shared memory (16 – 64 KB) and a data cache
  - Threads within a CTA can communicate with each other via a per SM shared memory
  - The shared memory acts as a software controlled cache
  - Allocate shared memory using __shared__ in CUDA

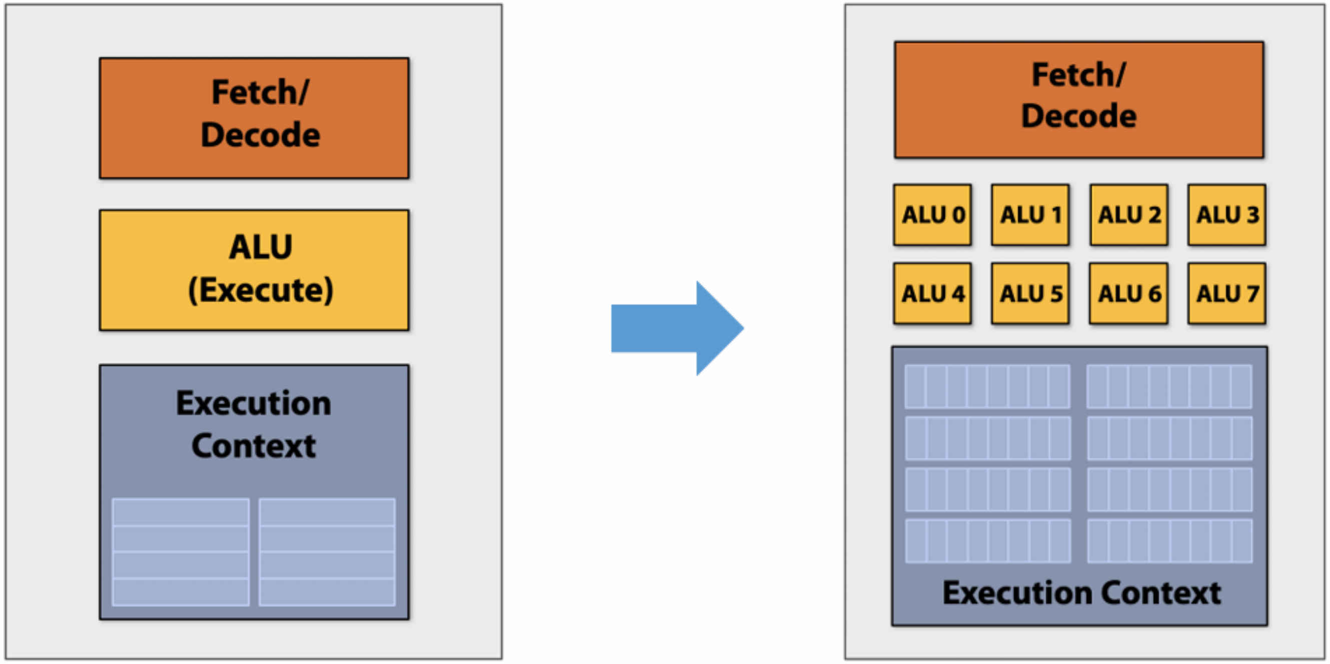


Modern GPU Architecture

- Synchronization
  - Threads within a CTA can synchronize using hardware-supported barrier instructions (__syncthreads())
  - Threads in different CTAs can communicate, but do so through a global address space that is accessible to all threads



SIMD vs. SIMT

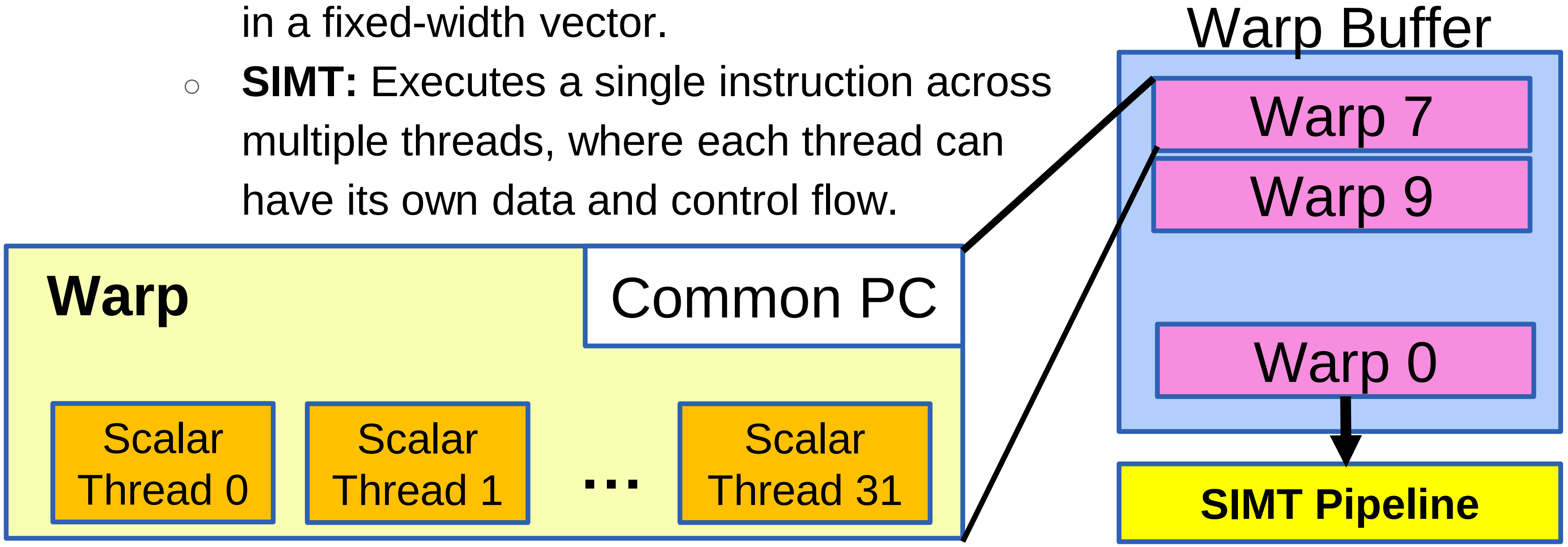


Conventional single instruction, single data processor

Modern single instruction, multiple data processor

SIMD vs. SIMT

- SIMT vs SIMD
  - **SIMD:** Executes a single instruction across multiple data elements in a fixed-width vector.
  - **SIMT:** Executes a single instruction across multiple threads, where each thread can have its own data and control flow.



SIMD vs. SIMT

3 key features that SIMD doesn't have BUT SIMT HAVE:

- **Single instruction, multiple register sets**
  - Register Set per Thread: each thread executing under a single instruction has its own set of registers
  - This feature allows each thread to maintain its state and perform calculations on its own unique data set, leading to a more flexible and powerful execution model compared to traditional SIMD (work on a single, shared set of data)
- **Single instruction, multiple addresses**
  - Each thread in a warp can access different memory addresses
  - This is in contrast to traditional SIMD, where all elements usually perform operations on consecutive memory locations

SIMD vs. SIMT

3 key features that SIMD doesn't have BUT SIMT HAVE:

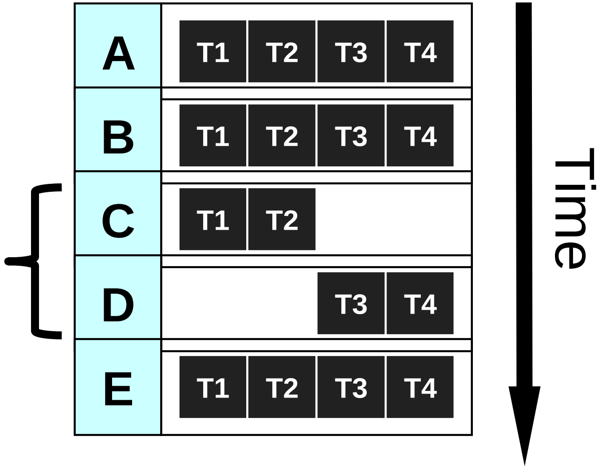
- **Single instruction, multiple flow paths**
  - Divergent Execution, if a warp encounters an ‘if-else’ statement, some threads might execute the ‘if’ block while others execute the ‘else’ block.
  - This feature provides a level of flexibility similar to MIMD architectures, allowing for individual threads to follow different execution paths based on their data.

SIMT Execution Model

- All threads in warps/wavefront execute the same instruction
- GPU runs warps/wavefront in lockstep on SIMT hardware
- Challenges: How to handle branch operations when different threads in a warp go to different path through program ?

```
w[] = {2, 4, 8, 10};
A: v = w[threadIdx.x];
B: if (v < 5)
C:       v = 1;
     else
D:       v = 20;
E:  w = bar[threadIdx.x] + v
```

Serialize operations in different paths



Control Divergence

- When threads within a warp take different control flow paths, SIMT will take multiple passes through these paths, one pass for each path.
  - For an if-else, some threads in a warp follow the if-path while others follow the else path, the hardware will **take two passes.**
  - During each pass, the threads that follow the other path are not allowed to take effect.

- When threads in the same warp follow different execution paths, we say that these threads exhibit **control divergence**

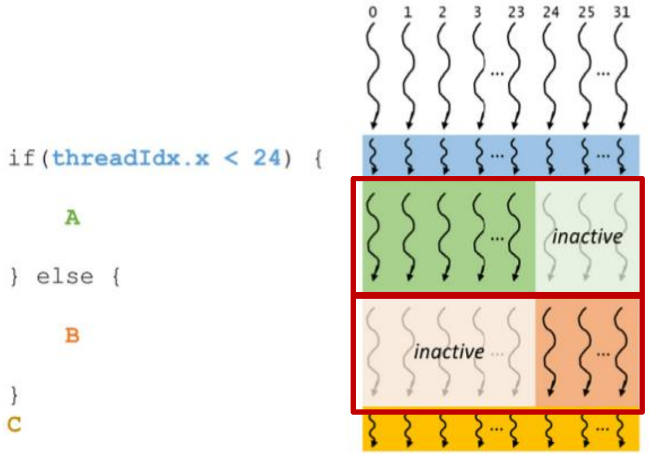


FIGURE 4.9

Example of a warp diverging at an if-else statement.

Control Divergence

- When threads in the same warp follow different execution paths, we say that these threads exhibit **control divergence**

- That is, SIMT unit must execute each paths serially, one after the other, to handle all possible outcomes.
  - This process is often referred to as "**masking**" where irrelevant data lanes are masked out during each pass.
- Cost
  - Execution resources that are consumed by the inactive threads in each pass, reducing overall efficiency.

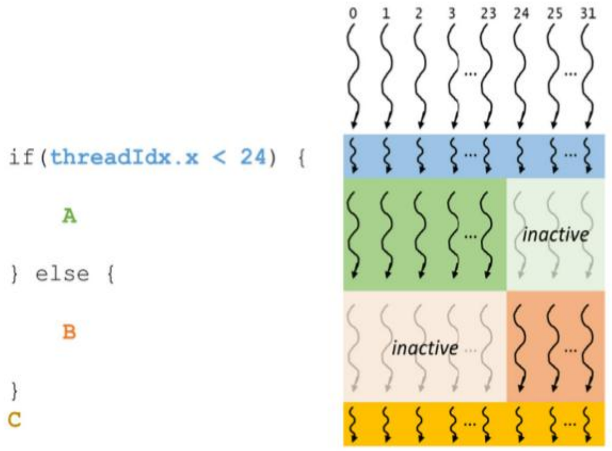
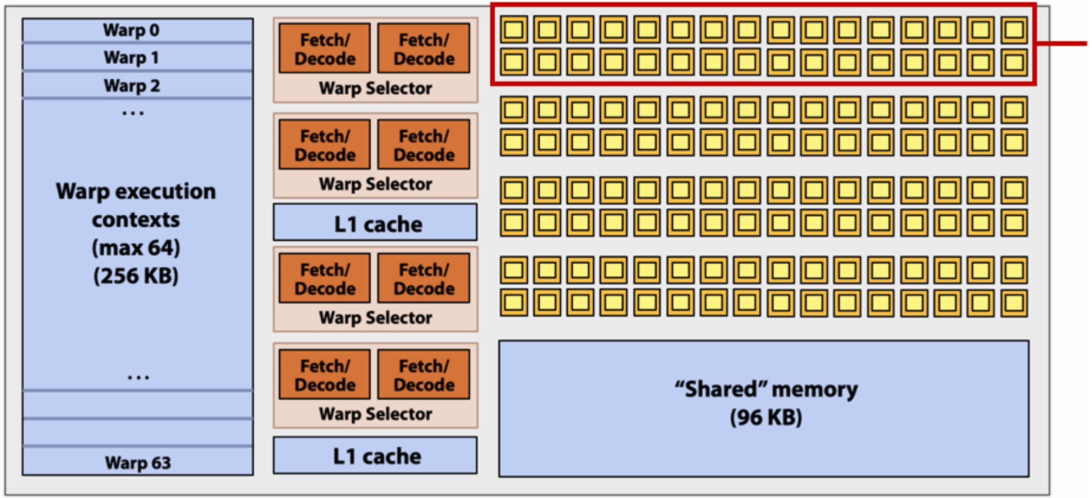


FIGURE 4.9

Example of a warp diverging at an if-else statement.

A GPU Core: Streaming Multiprocessor (SMM)

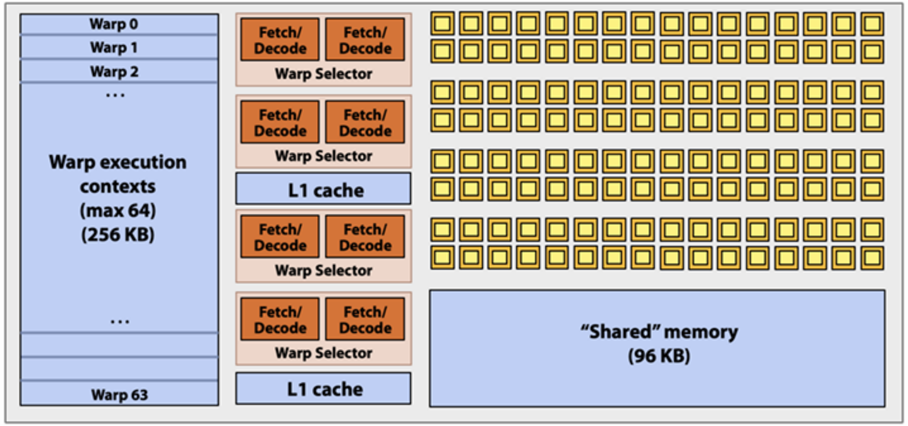
- In a SMM
  - Max warp execution contexts: 64 (64 x 32 = 2K total CUDA threads) and 96 KB of shared memory



SIMD functional unit

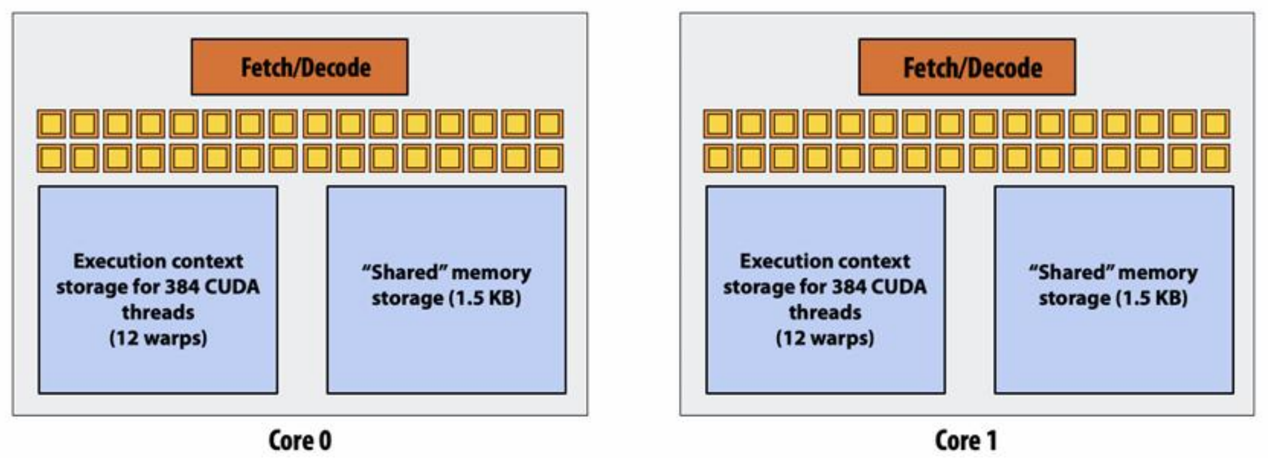
Running a Thread Block on an SMM

- **Warp**: a group of 32 CUDA threads shared an instruction stream
- SMM operation each clock
  - Select up to four runnable warps from 64 resident warps on an SMM (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)



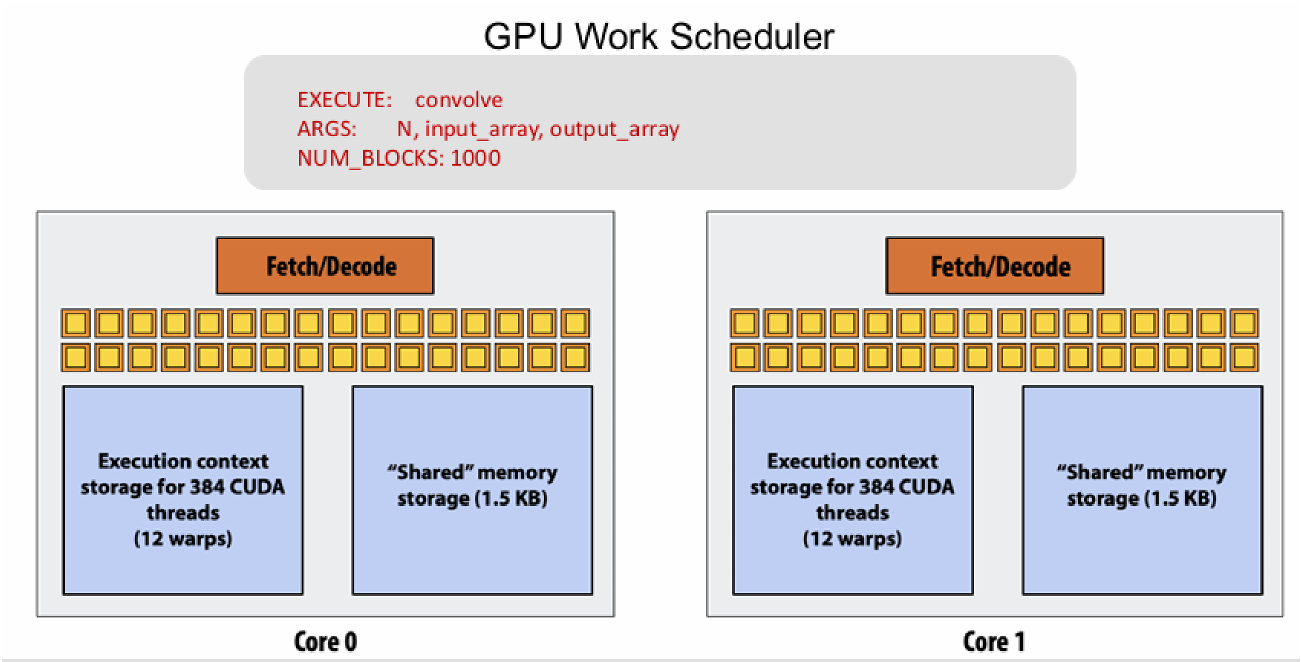
Running a CUDA Kernel

- Assume the host side launches 1000 thread blocks
  - Each thread blocks execute 128 CUDA threads
  - Each thread block allocate 130 x 4 = 512 bytes of shared memory and run the program on a two-SMM GPU



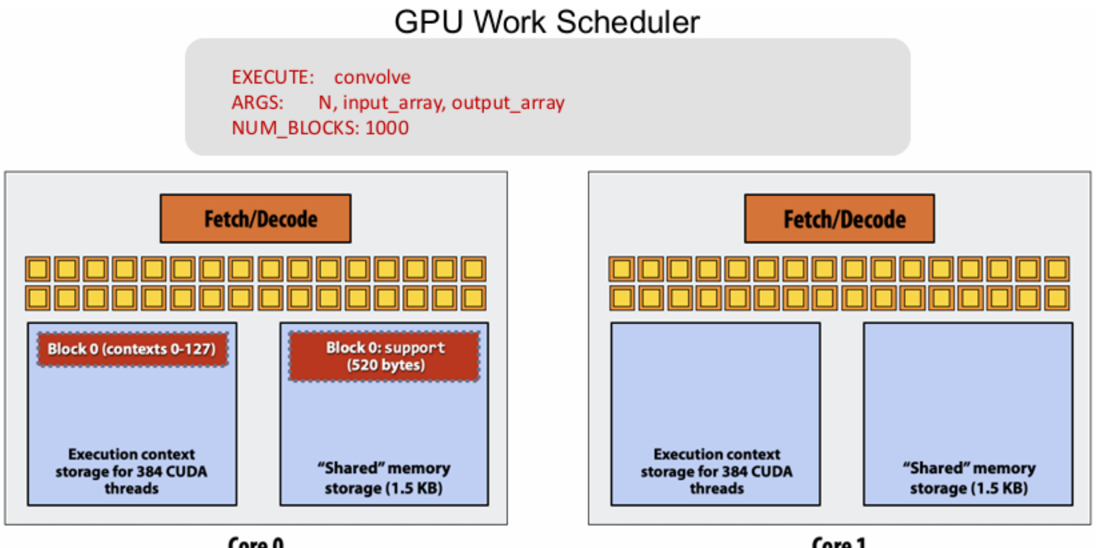
Running a CUDA Kernel

- Step 1: Host sends CUDA kernel to device



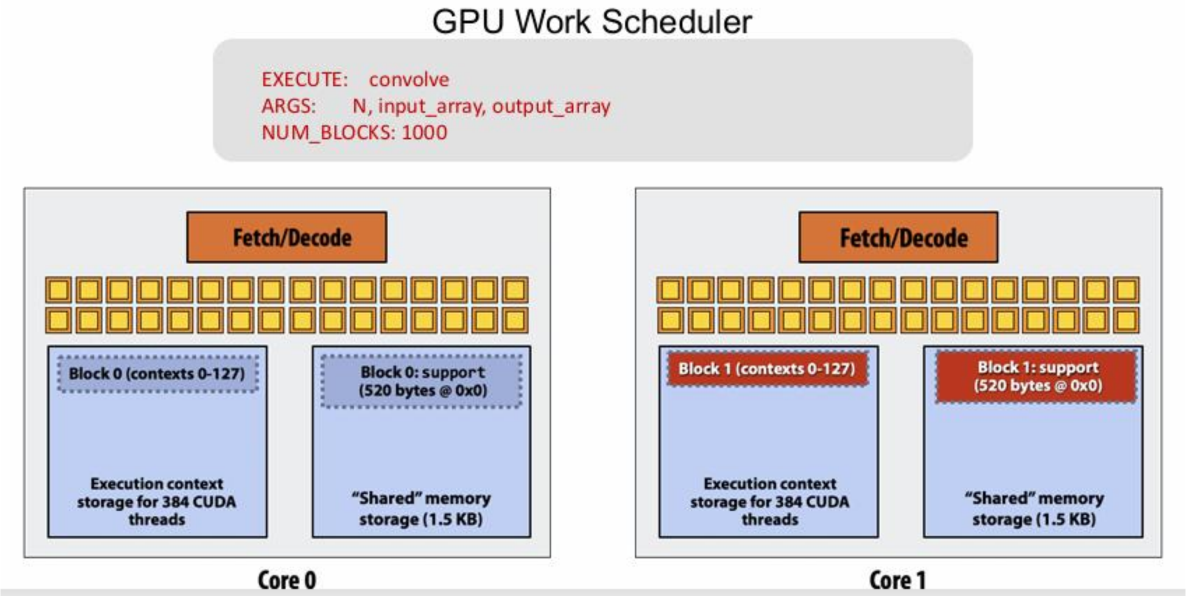
Running a CUDA Kernel

- Step 2: Scheduler maps block 0 to core 0
  - reserves execution contexts for 128 threads and 520 bytes of shared memory



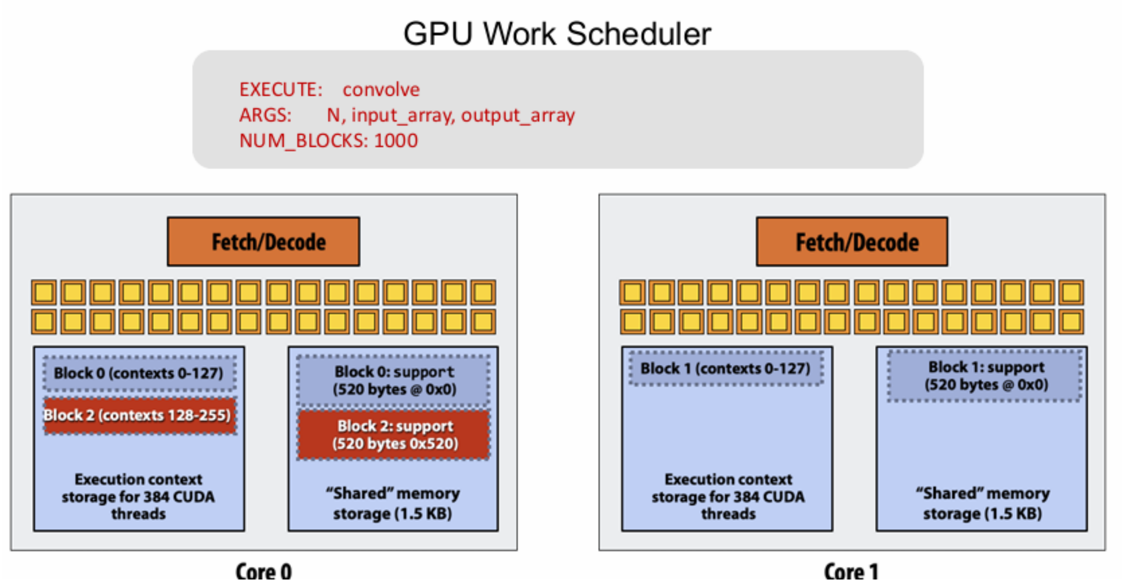
Running a CUDA Kernel

- Step 3: Scheduler continues to map blocks to available execution contexts



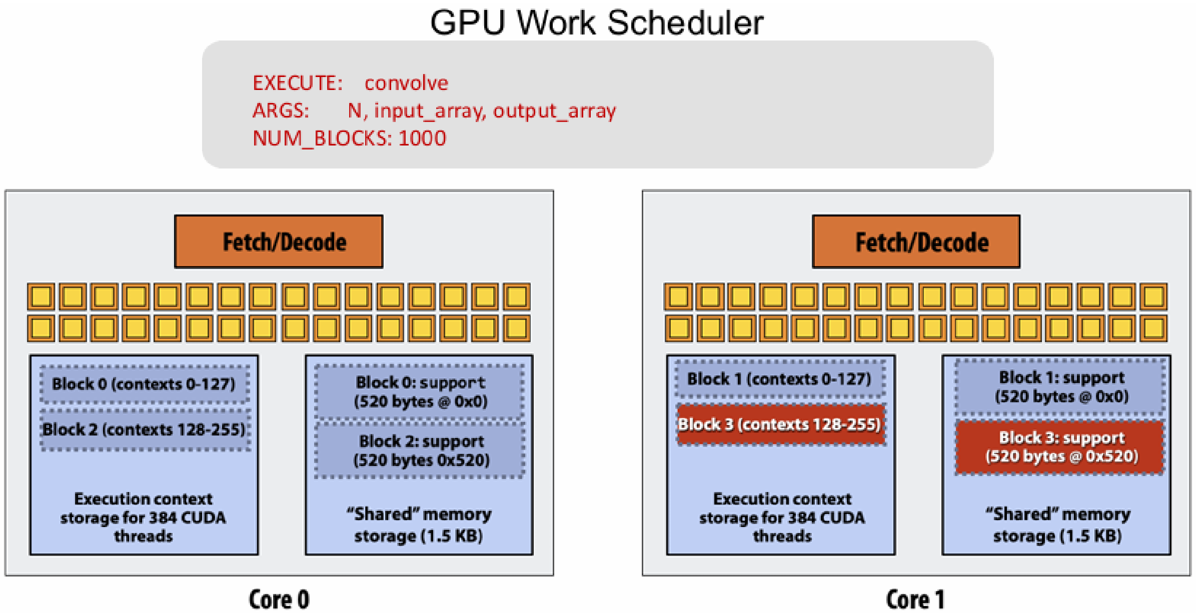
Running a CUDA Kernel

- Step 3: Scheduler continues to map blocks to available execution contexts



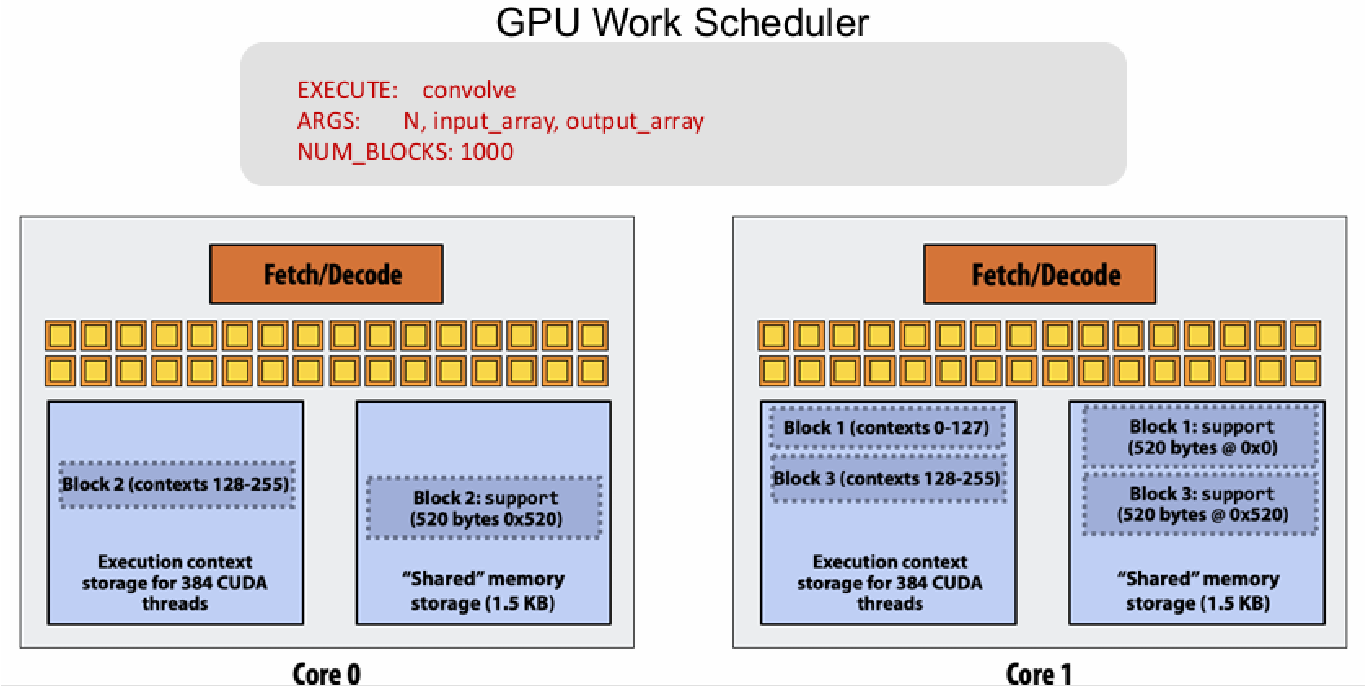
Running a CUDA Kernel

- Third block won't fit due to insufficient shared storage
  3 x 520B > 1.5 KB



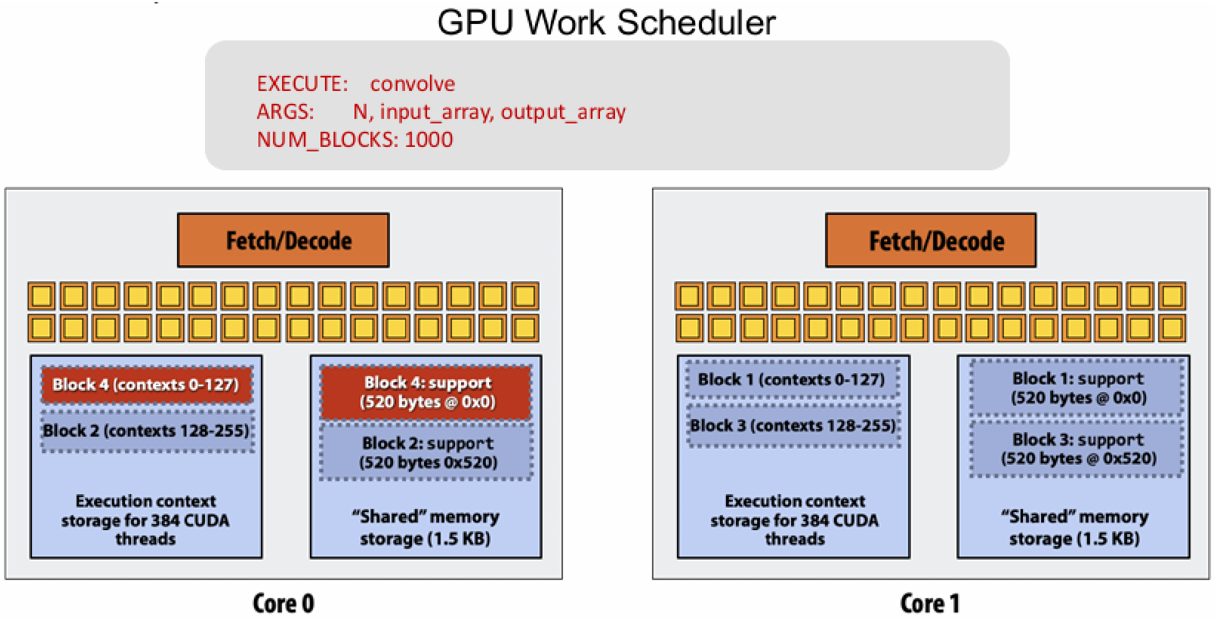
Running a CUDA Kernel

- Step 4: thread block 0 completes on core 0



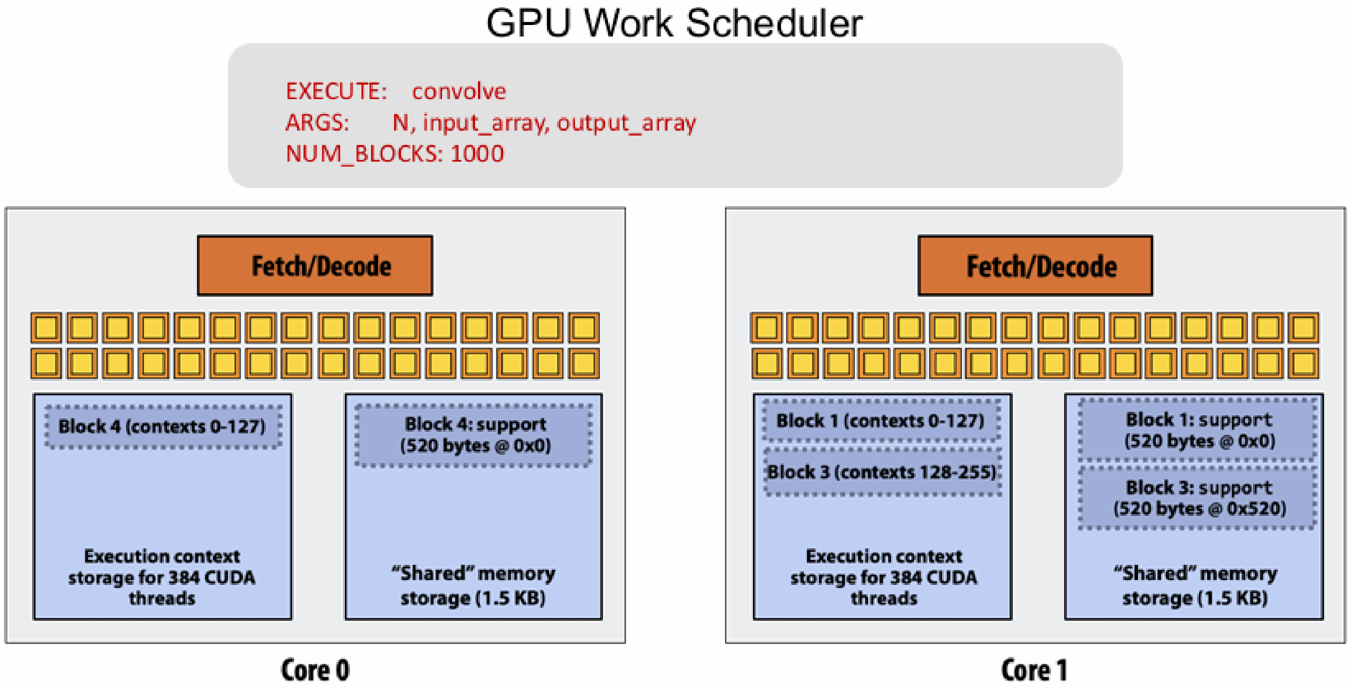
Running a CUDA Kernel

- Step 5: thread block 4 is scheduled on core 0
  - Mapped to execution contexts 0-127



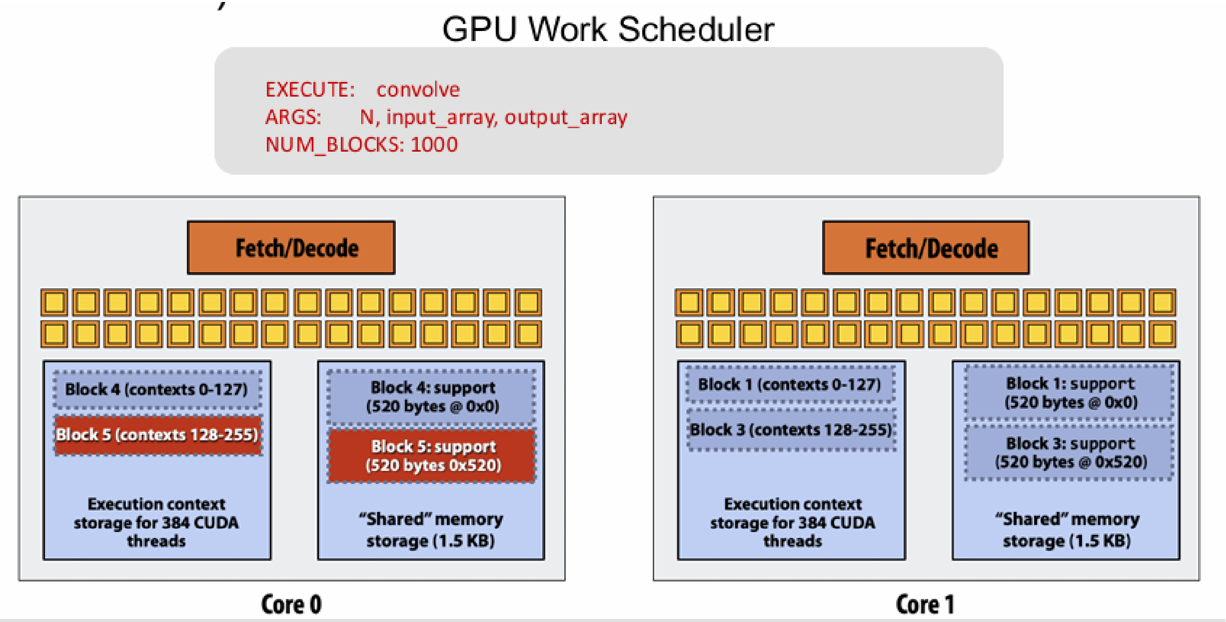
Running a CUDA Kernel

- Step 6: thread block 2 complete on core 0

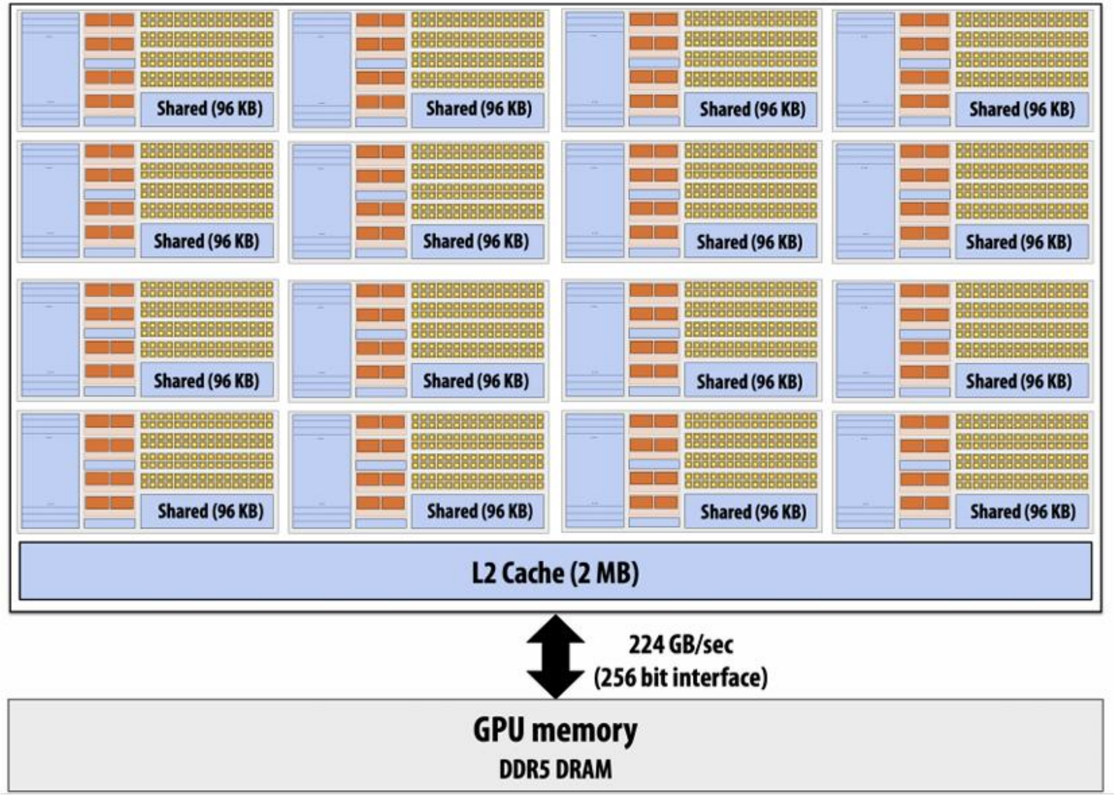


Running a CUDA Kernel

- Step 7: thread block 5 is scheduled on core 0
  - Mapped to execution contexts 128-255



NVIDIA GTX 980 Contains 16 SMMs



1.1 GHz clock

16 SMM cores per chip

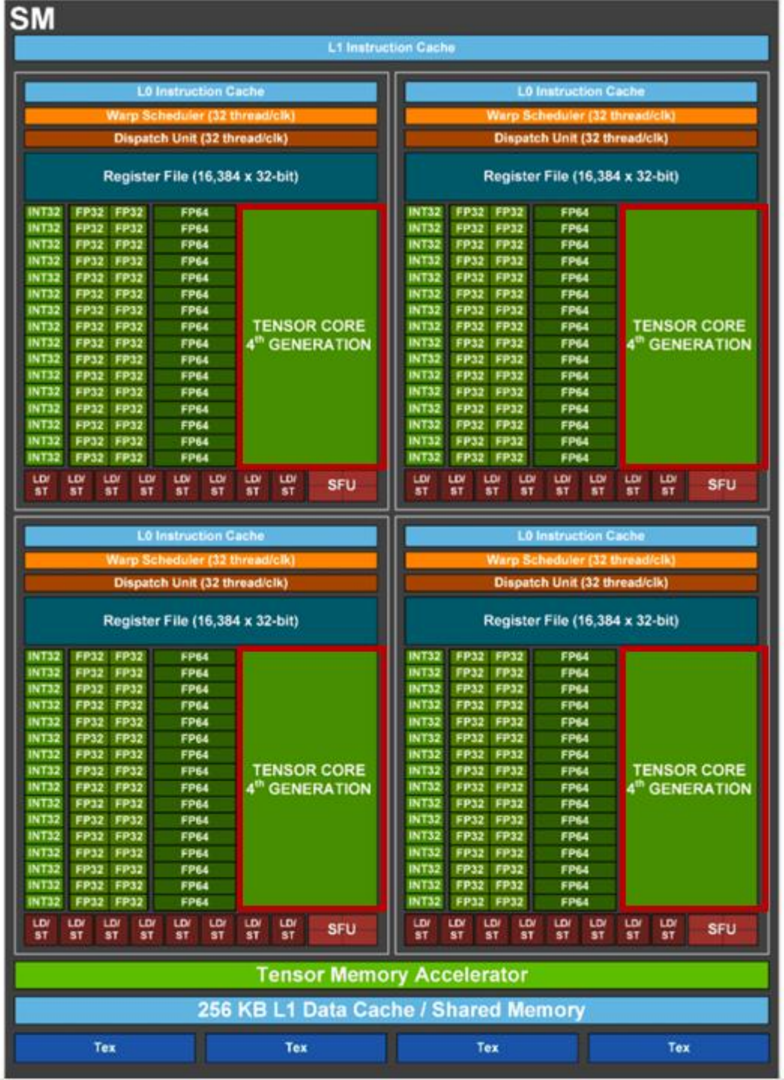
16 x 4 warps x 32 threads / warp
= 2048 SIMD mul-add ALUs
= 4.6 TFLOPs

Up to 16 x 64 = 1024 interleaved warps per chip
(32,768 CUDA threads / chip)

NVIDIA H100 GPU

- SMMs remain the same
  - Clock speed: 1064 MHz -> 1110 MHz
  - Map warps per SMM: 64 -> 64
  - Threads per warp: 32 -> 32
  - Shared memory per SMM: 96 KB -> 168 KB (A100) -> 256 KB (H100)
  - SMM: 16 -> 132
  - Peak performance: 4.6 TFLOPs -> 1000 TFLOPs (mainly because of tensor cores)

NVIDIA H100 GPU with Tensor Core

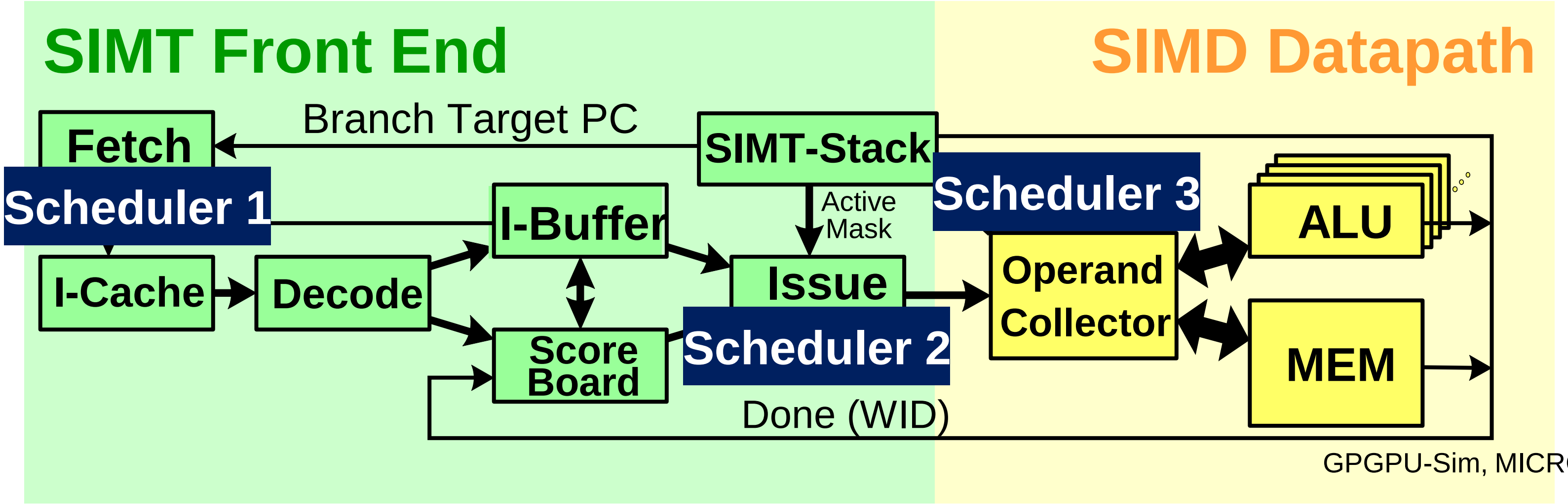


Takeaway Questions

- What are features of the GPU?
  - (A) Large L1 cache
  - (B) Needs the memory with high memory bandwidth
  - (C) The frequency of SIMT core is high
- How does GPU hide off-chip memory access latency?
  - (A) Increase the number of compute units
  - (B) Using large L1 data cache
  - (C) Interleaving warp execution

The SIMT Core

- SIMT front end
  - The instruction fetch: fetch, I-cache, Decode, and I-buffer
  - The instruction issue: I-buffer, Scoreboard, Issue, SIMT stack
- SIMD data path
  - Operand collector, ALU, Memory

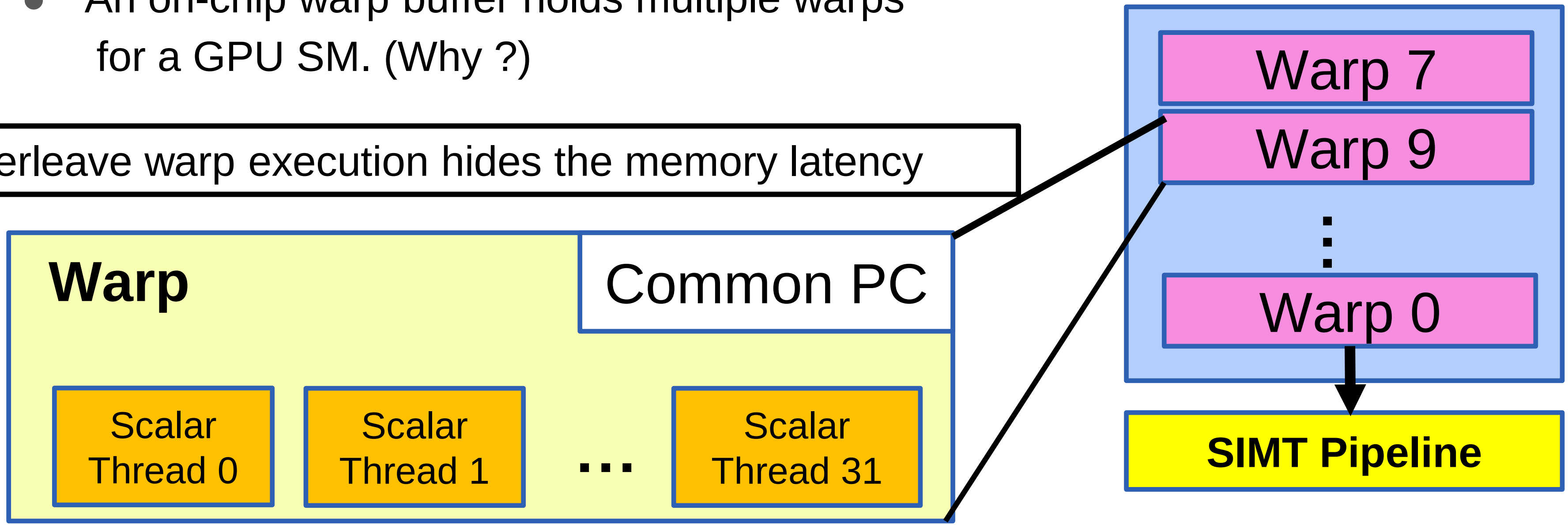


GPGPU-Sim, MICRO

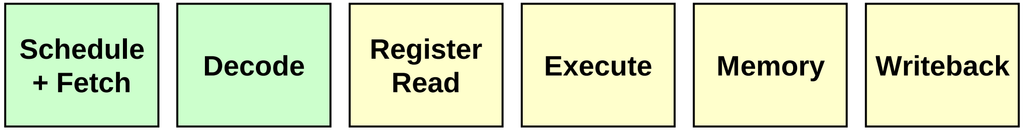
The Execution in the SIMT core

- In each cycle, the hardware selects a warp for scheduling
  - The warp’s program counter is used to access an instruction memory to find the next instruction to execute for the warp
- An on-chip warp buffer holds multiple warps for a GPU SM. (Why ?)

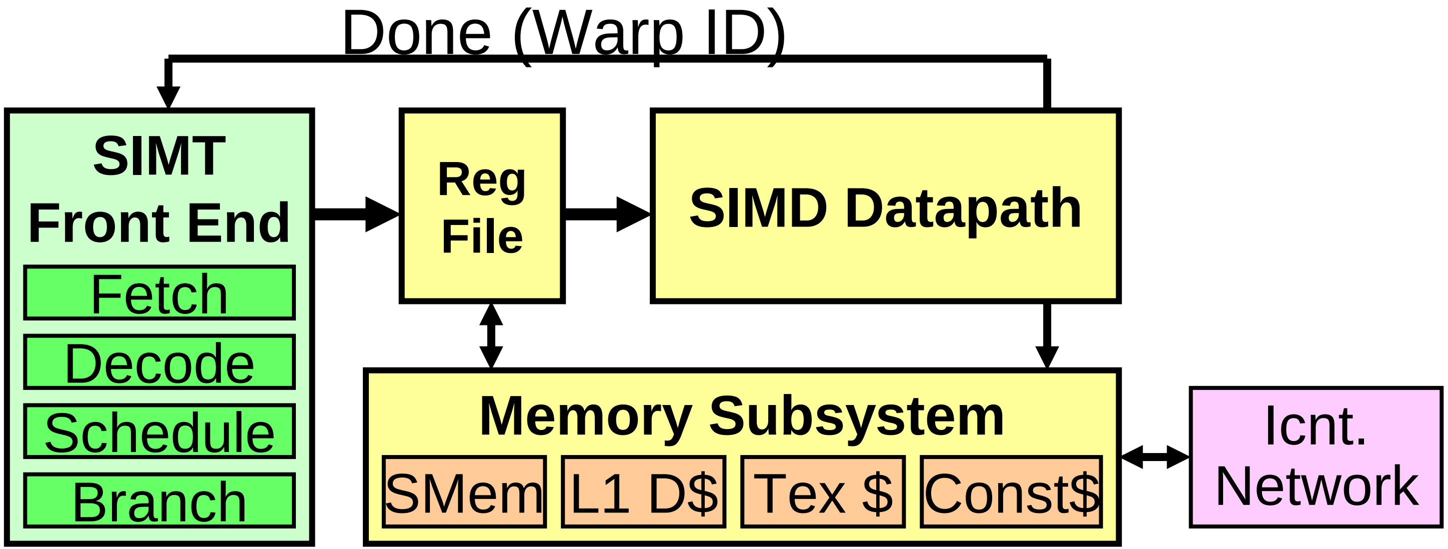
Interleave warp execution hides the memory latency



SIMT Pipeline



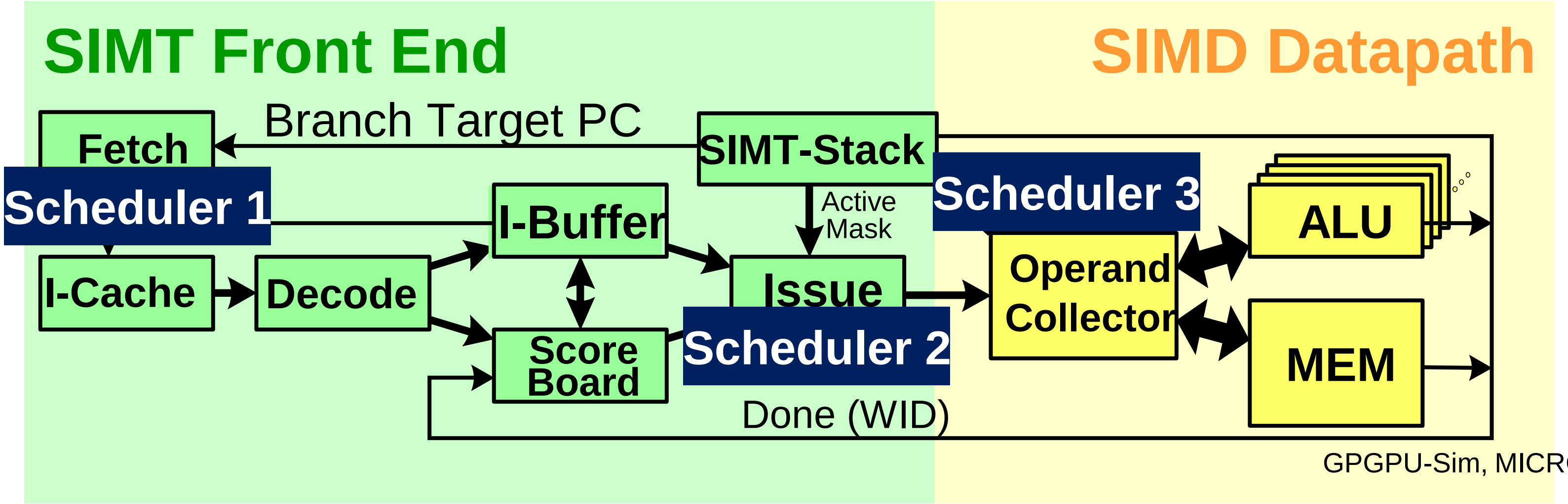
- 5 stage In-Order SIMT pipeline
- Register values of all threads stays in core



GPGPU-Sim, MICRO

Inside a SIMT Core

- Fetch, Warp Issue, and Operand Schedulers
- Scoreboard ->data hazard and SIMT stack->control flow
- Large register file
- Multiple SIMD functional units

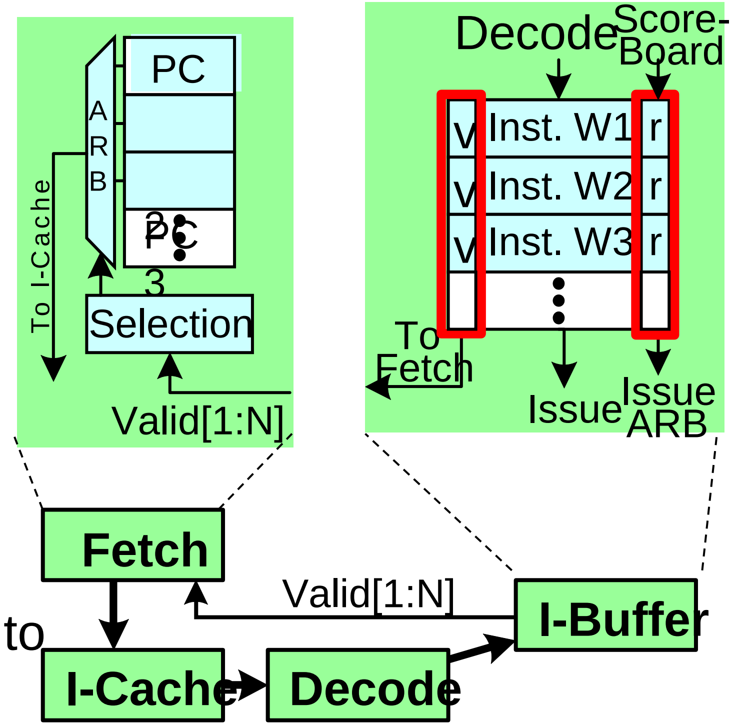


GPGPU-Sim, MICRO

Fetch + Decode

v: valid bit
r: ready bit

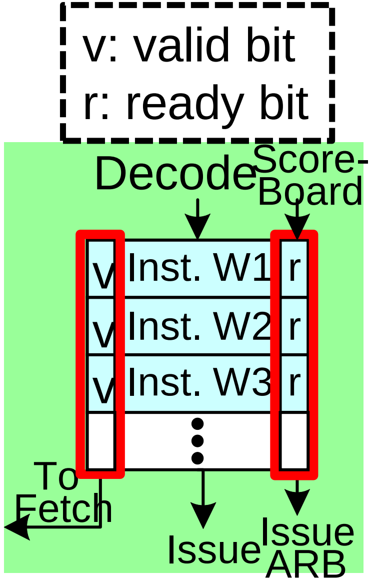
- **I-Cache**
  - Fetch instructions of warps in a round robin manner
  - Read-only, set associative
  - FIFO or LRU replacement
- **I-Buffer**
  - Store instructions fetched from I-cache
  - Each warp has two I-buffer entries
  - Valid bit indicates non-issued decode instructions
  - Ready bit indicates instructions are ready to be issued to the execution pipeline



GPGPU-Sim, MICRO

Instruction Issue

- A round-robin arbiter
  - Choose instructions of a warp from I-Buffer to issue to the rest of the pipeline
  - Allow dual issue
- Instruction issue
  - Memory instructions are issued to memory pipeline
  - SP and SFU pipeline
- Issue stage
  - Barrier operations are executed
  - SIMT stack is updated
  - Register dependency is tracking (Scoreboard)
  - Warps wait for barrier (__synthreads()) at issue stage



GPGPU-Sim, MICRO

The Execution in the SIMT core

- After fetching an instruction
  - The instruction is decoded
  - Source operand registers are fetched from the register file
  - Determine SIMT execution mask values
- SIMD execution
  - Execution proceeds in a single-instruction, multiple-data manner
  - Each thread executes on the function unit associated with a lane provided the SIMT execution set is set
- Function unit
  - Special function unit (SFU), load/store unit, floating-point, integer function unit, Tensor core

ALU Pipelines

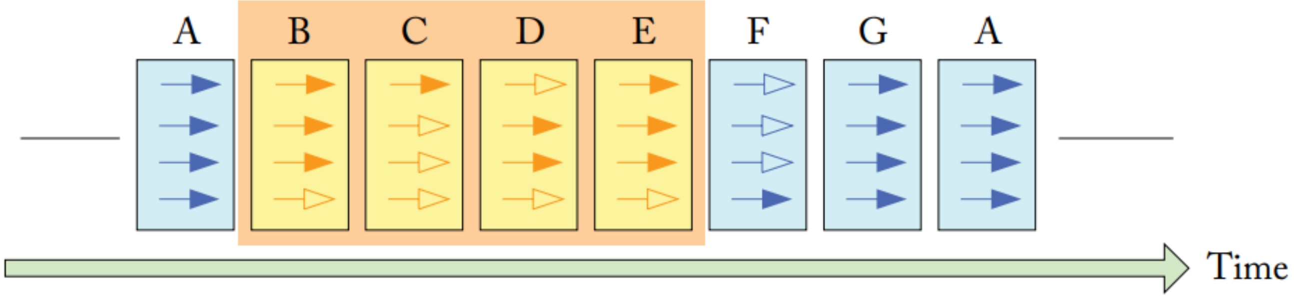
- SIMD execution unit
  - SP units executes ALU instructions except some special ones
  - SFU units executes special functional instructions (sine, log …)
  - Different types of instructions takes varying execution cycles
  - A SIMT core has one SP and SFU unit
  - Each unit has an independent issue port from the operand collector.
- Writeback
  - Each pipeline has a result bus for writeback
  - Except SP and SFU shares a result bus
  - Time slots on the shared bus is pre-allocated

Scoreboard

- Dynamically scheduling instructions so that they can execute out of order when there are no conflicts and the hardware is available
- Solutions for WAR:
  - Stall writeback until registers have been read
  - Read registers only during Read Operands stage
- Solution for WAW:
  - Detect hazard and stall issue of new instruction until other instruction completes
- Instructions with hazards -> not ready flag in I-Buffer

SIMT Execution Masking

- **SIMT Execution masking**
  - **Tackle the nested control flow**
  - **Skipping computation entirely** while all threads in a warp avoid a control flow path
  - Serialize execution of threads following different paths within a given warp
  - An arrow with a hollow head indicates the thread is masked off



(b) Re-convergence at Immediate Post-Dominator of B

SIMT Stack

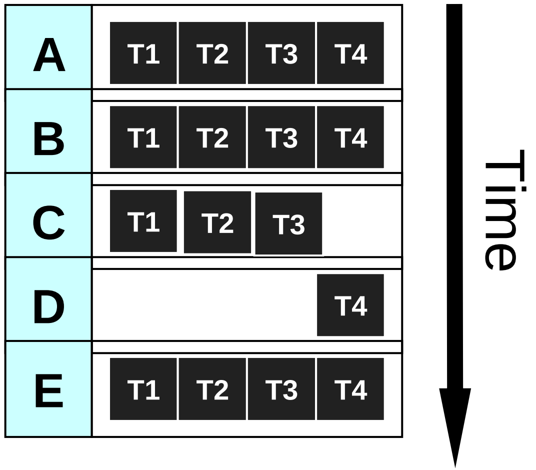
- SIMT stack includes
  - A reconvergence program counter (RPC)
  - The address of the next instruction to execute (Next PC)
  - An active mask

One stack per warp
SIMT Stack

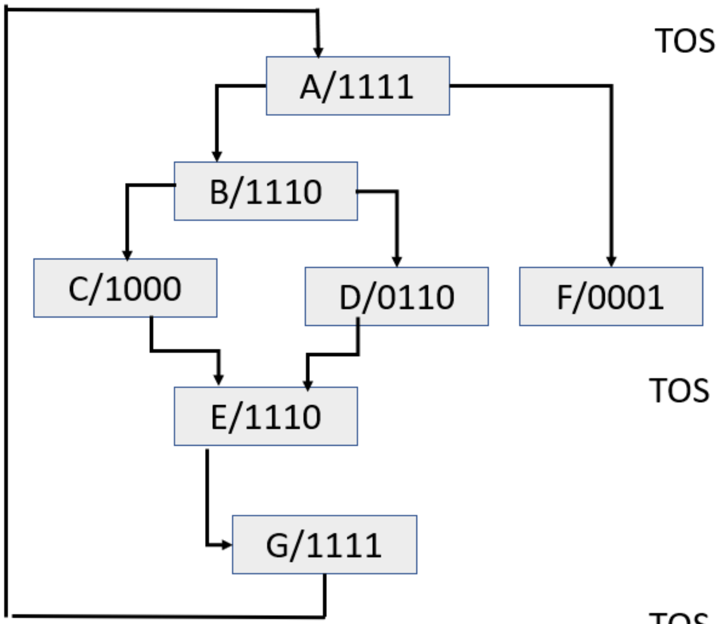
PC	RPC	Active Mask
E	-	1111
D	E	0001
C	E	1110

```
w[] = {2, 4, 8, 10};
A: v = w[threadIdx.x];
B: if (v < 9)
C:      v = 1;
     else
D:      v = 20;
E:  w = bar[threadIdx.x] + v
```

Serialize operations in different paths



SIMT Stack



Top-of-Stack (TOS)

Initial State

Re-converge PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	B	1110

TOS →

After Divergent Branch

Re-converge PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110
E	D	0110
E	C	1000

TOS →

After Reconvergence

Re-converge PC	Next PC	Active Mask
-	G	1111
G	F	0001
G	E	1110

TOS →

SIMT Stack

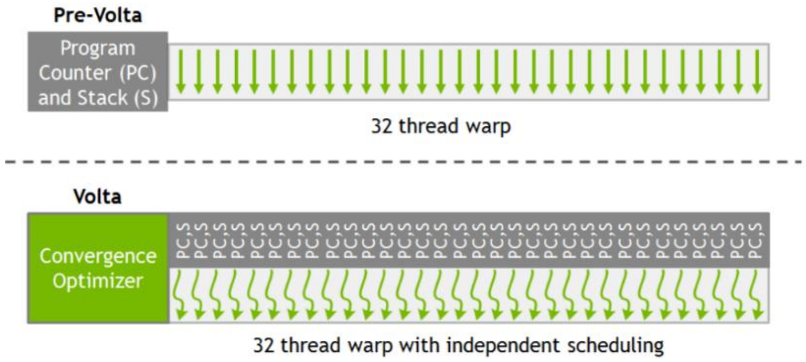
- Predicate register
  - A **predicate register** is part of the scalar register file that is shared by all threads in a warp. These registers are used as predication registers to control the activity of each thread within a warp.
  - **Predicate masks <-> active mask**
  - Specifically, the compiler utilizes this scalar register file to emulate a SIMT stack in software when it encounters potentially divergent branches in the compute kernel.

One stack per warp SIMT Stack

PC	RPC	**Active Mask**
E	-	1111
D	E	0001
C	E	1110

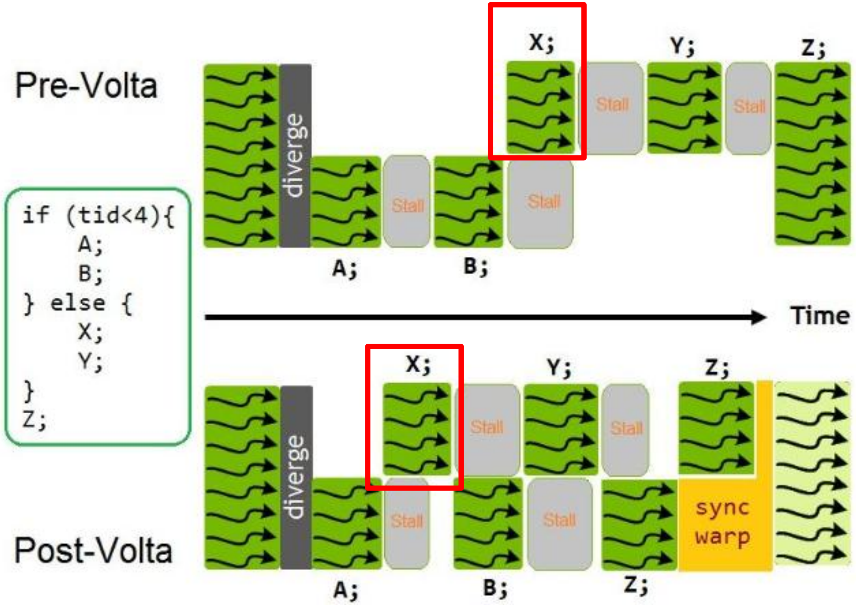
Independent Thread Scheduling

- Pre-Volta GPU
  - The passes in the presence of “if-else” are executed sequentially
- From the Volta GPU onwards,
  - **The “if-else” passes may be executed concurrently**, meaning that the execution of one pass may be interleaved with the execution of another pass.
  - This feature is referred to as independent thread scheduling.
  - <u>Allocate per-thread scheduling resources such as program counter (PC) and call stack (S)</u>



Independent Thread Scheduling

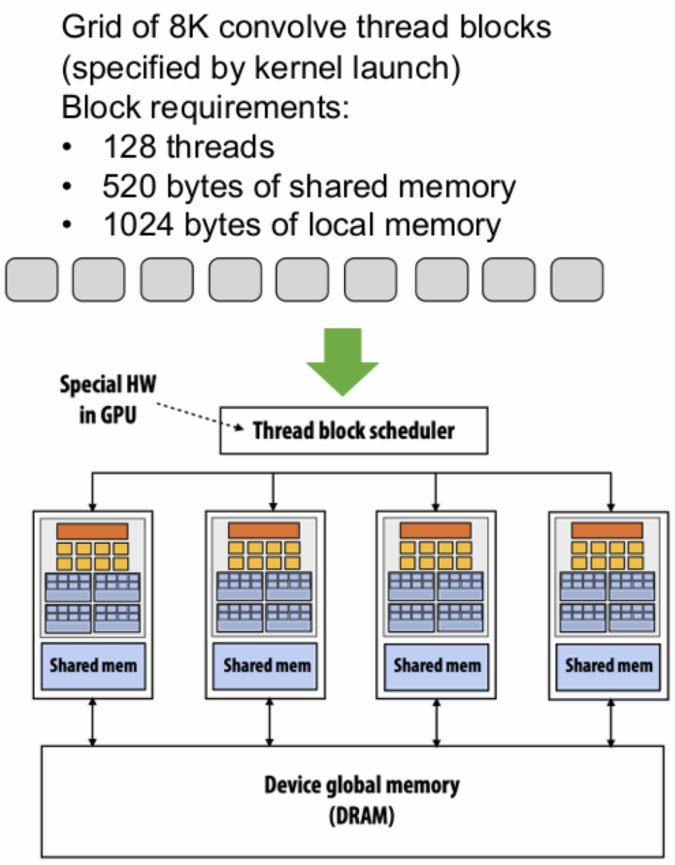
- Replace the stack with per warp convergence barriers
- Scheduler optimizer
  - Determines how to group active threads from the same warp together into SIMT units



B depends on A, Y depends on X

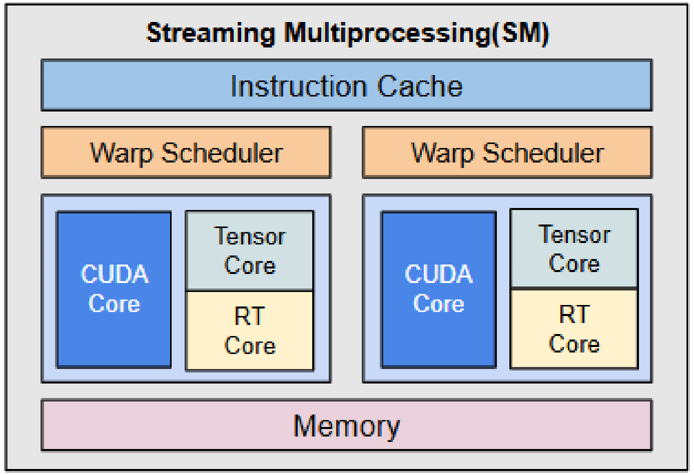
Thread Block (CTA) Scheduling

- A CTA is issued to one SIMT core at a time
- Scans through SIMT cores to issue a CTA to a SIMT core with available resources at round-robin manner
  - Threads (available warp buffer)
  - The shared memory space
  - The register file
- Multiple concurrent kernels
  - Different kernels can be executed across SIMT cores



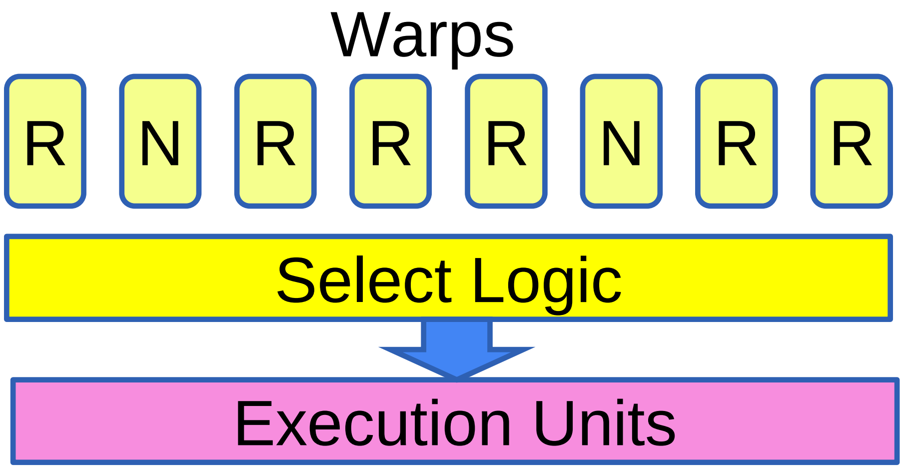
Warp Scheduling

- The warp scheduler is responsible for <u>arranging the execution order of warps on an SM</u>
- Warp scheduler selects an instruction of a warp that is ready to execute
- Instruction-level parallelism (ILP)
  - Pick instructions of the same warp
- Thread-level parallelism (TLP)
  - Choose instructions across different warps
- Multiple Warp schedulers on a SIMT Core
- Impact on the SIMT Core utilization



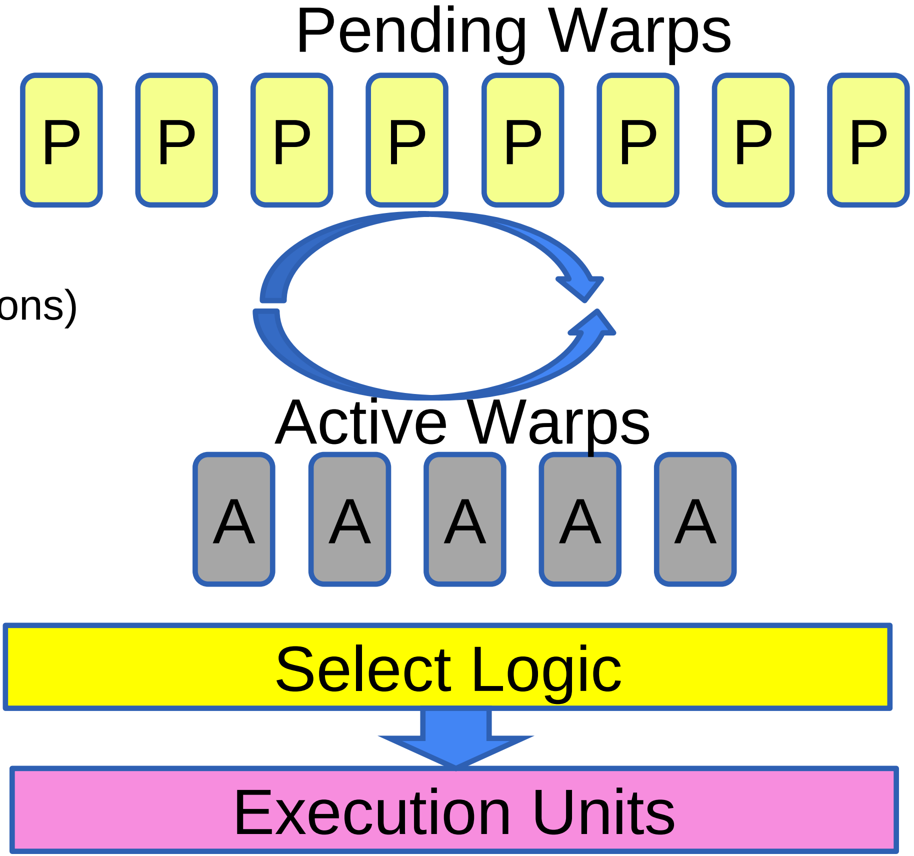
Loose Round Robin (LRR) Scheduling

- Scan through warps and select the one ready warp (R)
- If warp is not ready (N), skip that one and go to the next one
- Warp all runs on the same chance
- Problems
  - Potentially all warps reach memory access phase together and get stall



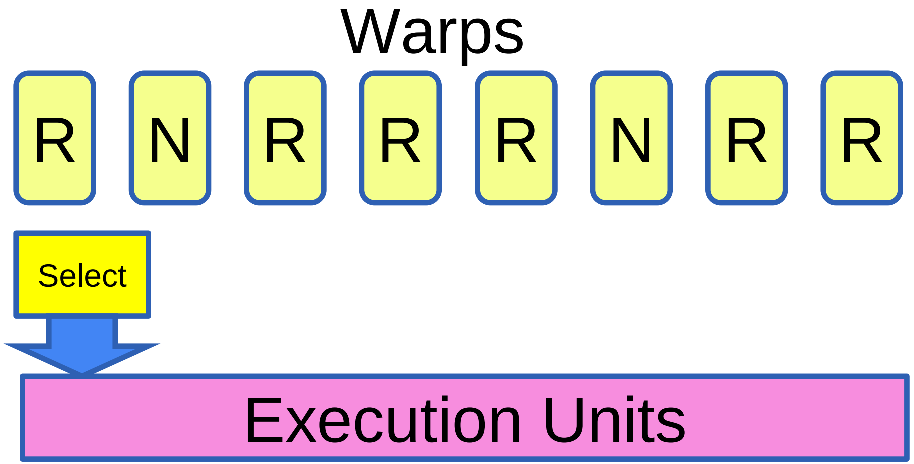
Two-Level (TL) Scheduling

- Warps are divided into two groups
  - Pending warps (potentially waiting for long latency instructions)
  - Active warps (ready to execute)
  - Warps move between pending and active warps
  - Active warps are issued in LRR
- Overlap warps with memory access and ALU instructions



Greedy-Then-Oldest Scheduling

- Select instructions of a single warp until it stalls
- Then pick the oldest warp to the next
- Improve the cache locality of the greedy warp

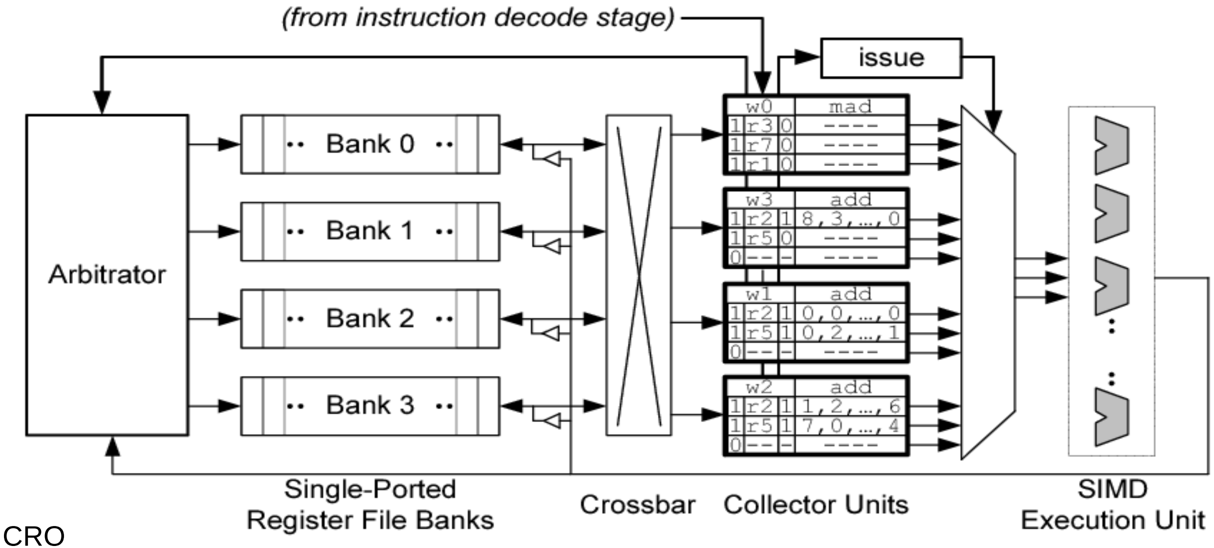


Register File

- 256 KB register files on a SIMT core
- How many registers can be used by one thread ?
  - Maximum number of warps per SIMT core is 64
  - 32 threads per warp
  - 256 KB / 64 / 32 / 32-bit = 32
- Need “4 ports” (e.g. FMA) -> increase area greatly
- What is the solution ?
  - Banked single ported register file

Operand Collector

- Operand collector aims to increase register file bandwidth
- A valid bit, a register identifier, a ready bit, and operand data
- Arbiter selects operand that don’t conflict on a given cycle



GPGPU-Sim, MICRO

Register Bank Conflict

- On cycle 4, issue instruction i2 after a delay due to bank conflict
- Low utilization of register banks
- Solutions ?

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
W1:r4	W1:r5	W1:r6	W1:r7
W1:r0	W1:r1	W1:r2	W1:r3
W0:r4	W0:r5	W0:r6	W0:r7
W0:r0	W0:r1	W0:r2	W0:r3

Cycle	Warp	Instruction
0	W3	i1: mad r2, r5, r4, r6
1	W0	i2: add r5, r5, r1
4	W1	i2: add r5, r5, r1

Cycle

Bank	1	2	3	4	5	6
0	W3:i1:r4					
1	W3:i1:r5	W0:i2:r1	W0:i2:r5	W0:i2:r5	W1:i2:r1	W1:i2:r5
2	W3:i1:r6		W3:i1:r2			
3						

Register Bank Conflict

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
W1:r7	W1:r4	W1:r5	W1:r6
W1:r3	W1:r0	W1:r1	W1:r2
W0:r4	W0:r5	W0:r6	W0:r7
W0:r0	W0:r1	W0:r2	W0:r3

- Swizzle banked register layout
- W0:r0 -> bank 0, W1:r0 -> bank 1, W2:r0 -> bank 2, W3:r0 -> bank 3
- Save 1 cycle against the naïve bank layout. Could we do better ?

Cycle	Warp	Instruction
0	W3	i1: mad r2, r5, r4, r6
1	W0	i2: add r5, r5, r1
4	W1	i2: add r5, r5, r1

Cycle

Bank	1	2	3	4	5	6
0						
1	W3:i1:r5	W0:i2:r1	W0:i2:r5	W3:i1:r2	W1:i2:r1	
2	W3:i1:r6			W0:i2:r5	W1:i2:r5	
3	W3:i1:r4					

Takeaway Questions

- How does GPU hide the instruction fetch latency?
  - (A) Use SIMT stack
  - (B) Use multiple instruction fetcher
  - (C) Use instruction buffer
- What is the purpose of the SIMT stack?
  - (A) Record the register location
  - (B) Handle the branch divergence
  - (C) Increase the speed of SIMT execution

Takeaway Questions

- What are correct descriptions of SIMT execution model?
  - (A) Every thread in a warp tackles the same instruction
  - (B) Threads within a warp can walk different control paths concurrently
  - (C) Every thread in a warp accesses the shared data