When GPU Architecture Designs Meet Machine Learning

Tsung Tai Yeh
Computer Science Department of National Chiao Tung University

Acknowledgements and Disclaimer

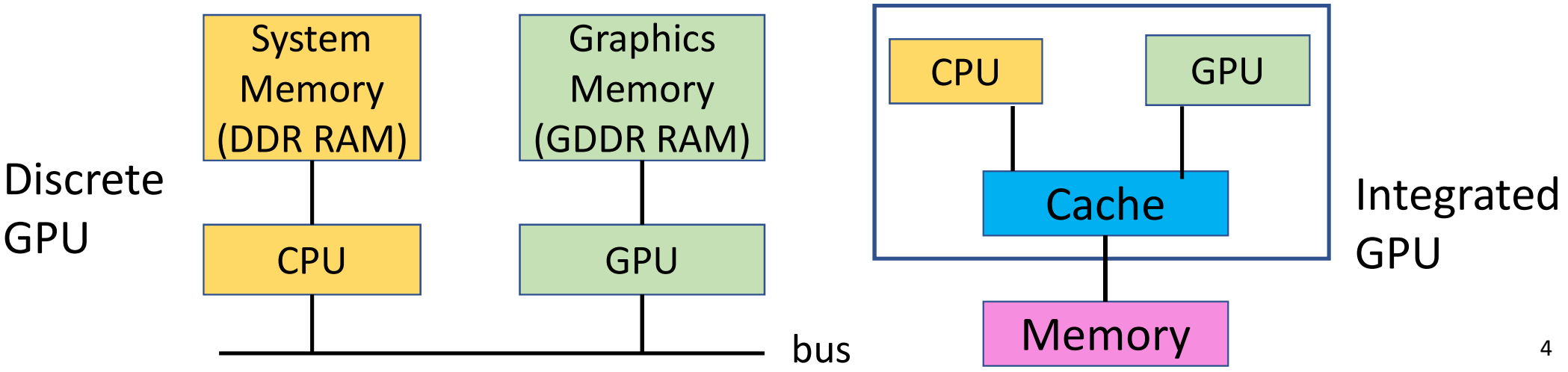
- Slides was developed in the reference with
  ECE 565, Computer Architecture, Purdue University, 2018
  GPGPU-Sim Tutorial , MICRO, 2012

Overview

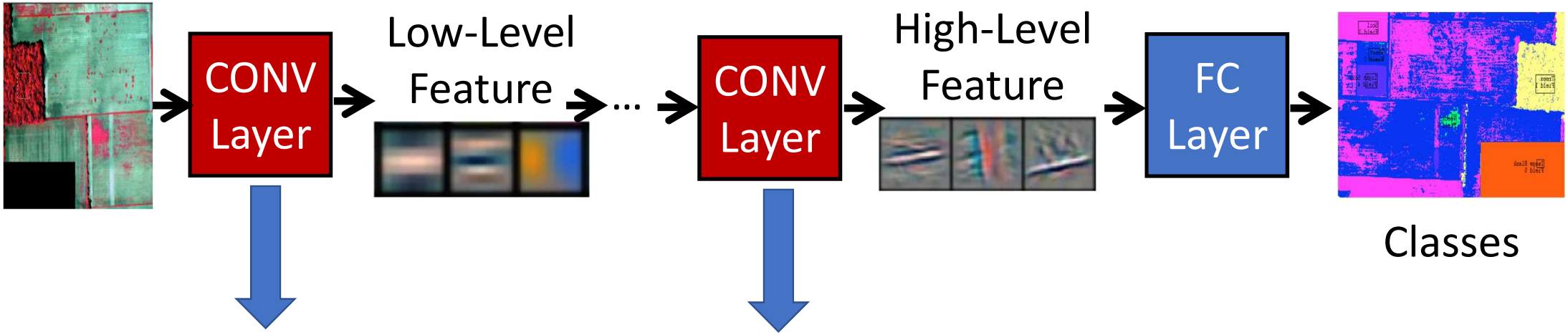
- Revisit GPGPU Programming & execution model
- GPU Micro-architecture
- GPU Tensor Core in ML applications

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



Convolutional (CONV) Layers

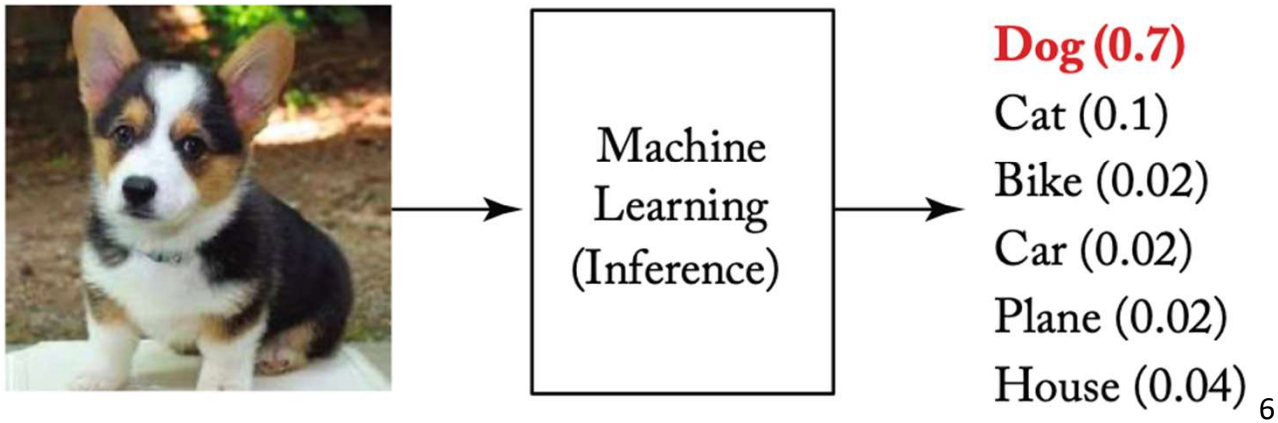


1. Convolutions mainly perform **vector-and-matrix multiplication**.
2. Convolutions takes more than **90%** of overall computation (critical path).
3. Optimization (software/hardware) for convolutions matters.

Training versus Inference

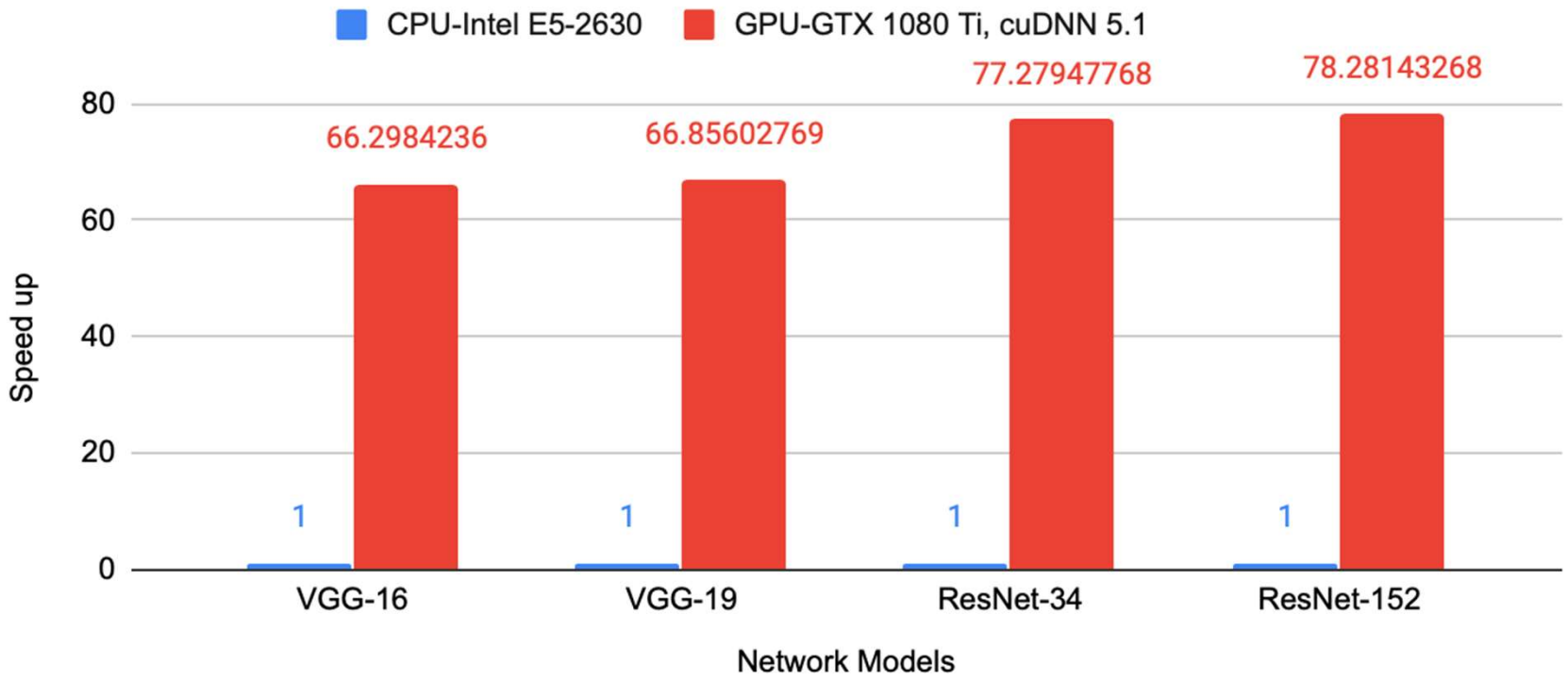
- **Training**: Determining the value of the **weights** in the network
  - Minimizing loss (L)
  - Loss (L): the gap between ideal correct probabilities and the probabilities computed by the DNN model
- **Inference**: Apply trained weights to determine output

  Include only forward



CPU vs GPU Training Time Comparison

- Normalized Training time on CPU and GPU (CPU has 16 cores, 32 threads)
- Why the model training on GPUs is much faster than on the CPU?



https://github.com/jcjohnson/cnn-benchmarks

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

6.7X

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

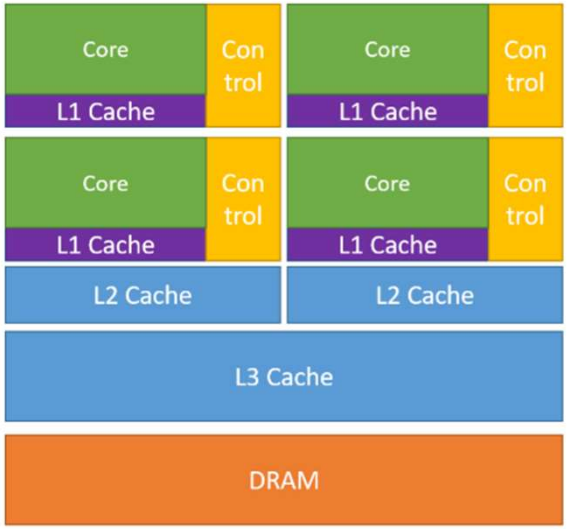
GPU: A **large** number of **simple** cores, the clock speed of each core is low, great for parallel tasks

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

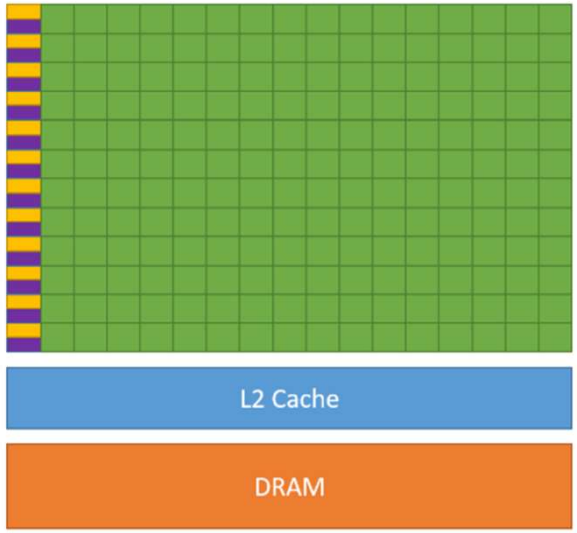
Why do we use GPU for computing ?

- 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

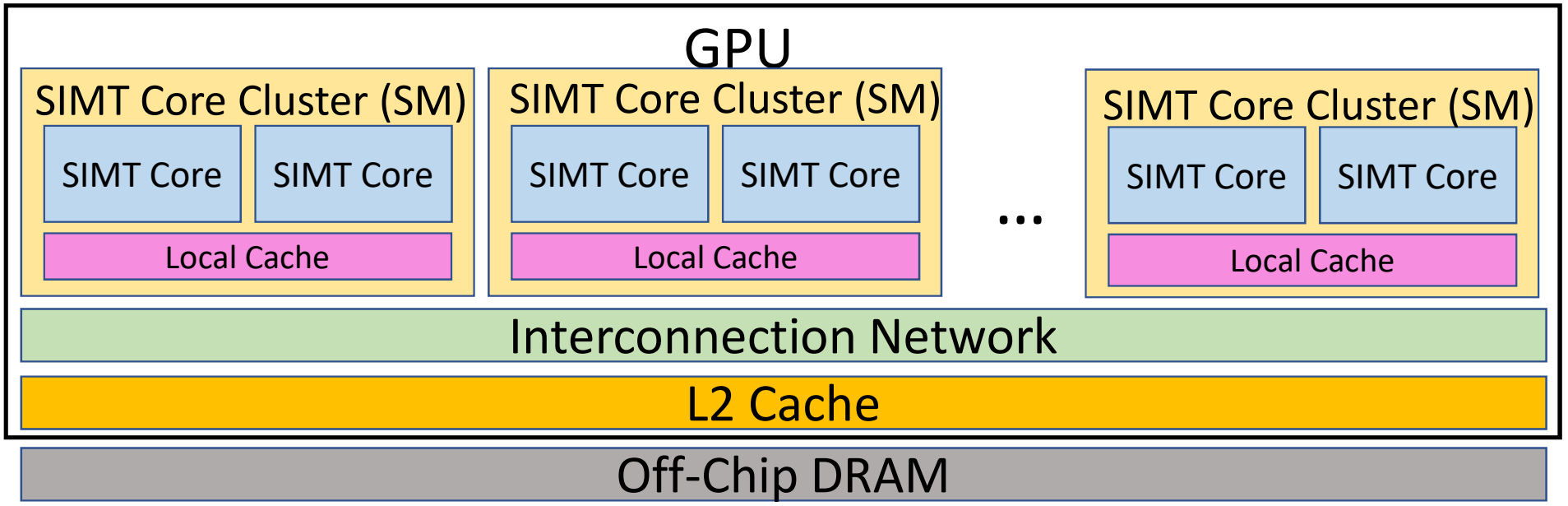


GPU

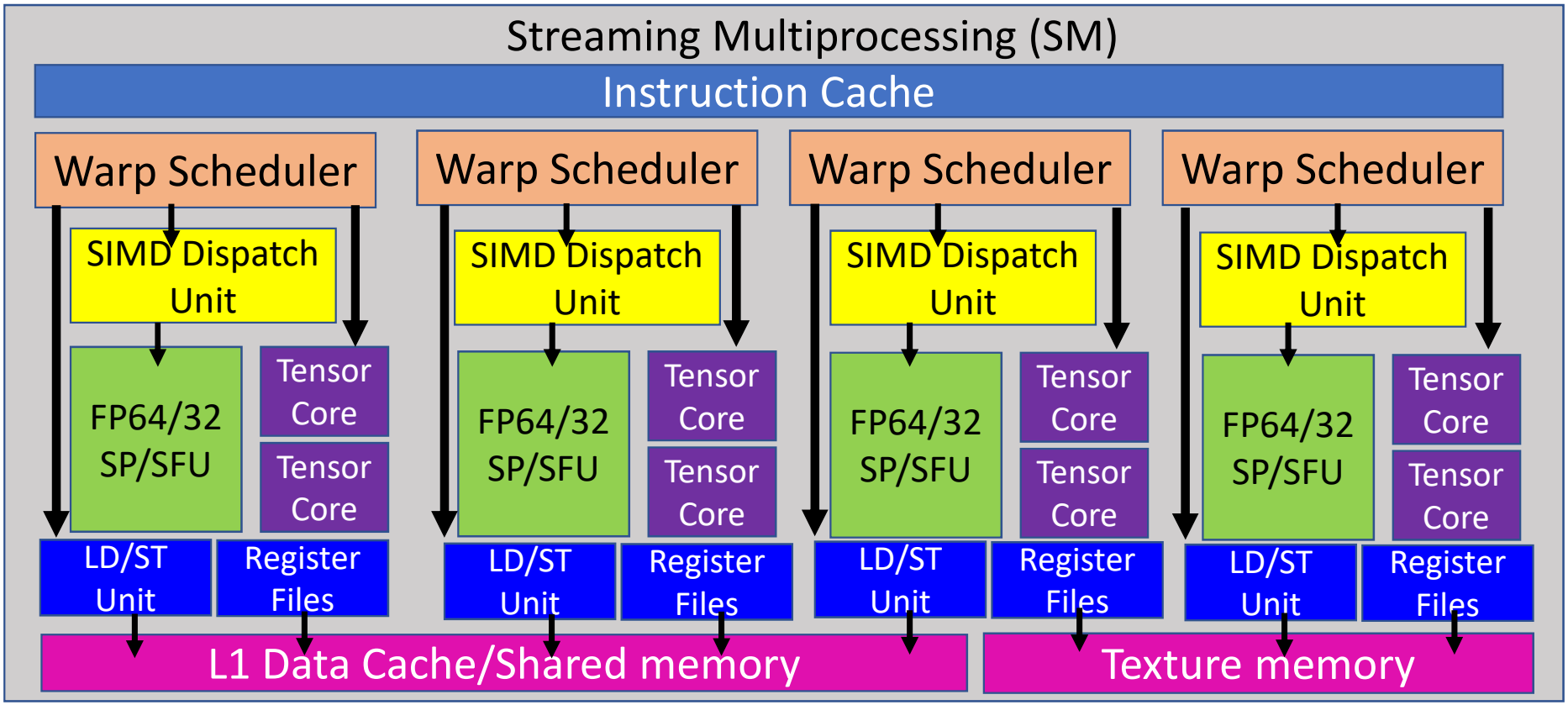


A Generic Modern GPU Architecture

- GPU **Single-Instruction, Multiple-Threads** (SIMT) operations
- A Streaming Multi-processor(SM) can serve multiple concurrent threads
- A SIMT core has its private local cache (L1/shared memory)



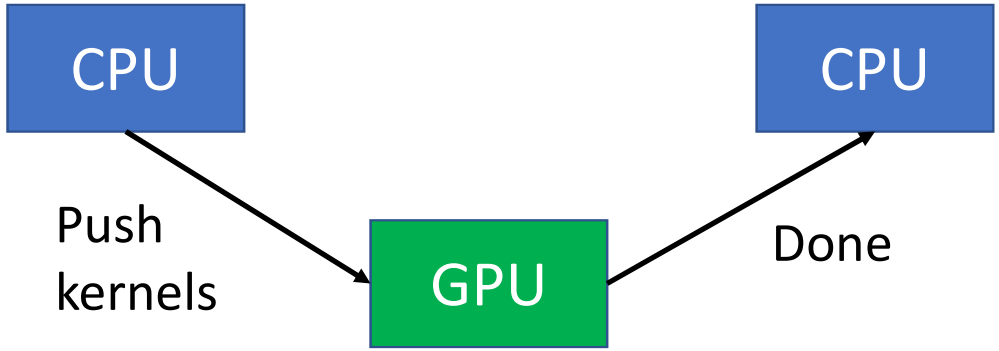
SIMT Core Micro-architecture



Zhu et.al., MICRO 2019

GPGPU Programming Model

- CPU offloads "**kernels**" consisting of multiple threads to GPU
- CPU transfer data to GPU memory (discrete GPU)
- Need to transfer result data back to CPU main memory
- CPU and GPU shares the same memory space (integrated GPU)



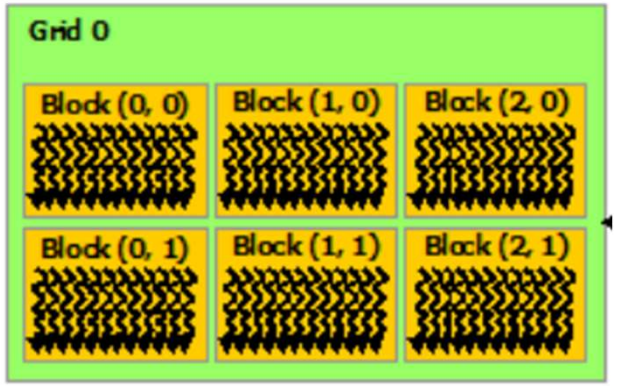
Could GPU spawn kernels within GPU? (Recursive calls)

Yes, CUDA dynamic parallelism

Could a GPU execute multiple kernels?

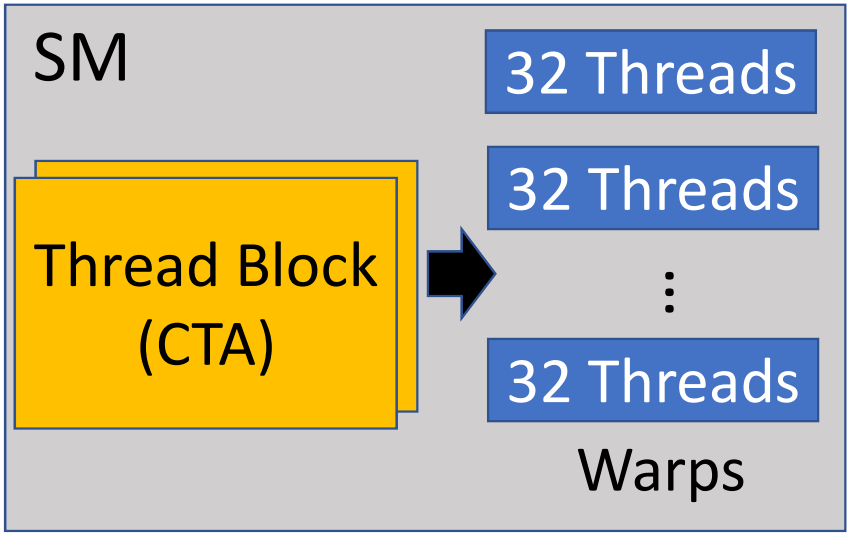
Yes, GPU supports "concurrent execution"

GPU Thread Hierarchy



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

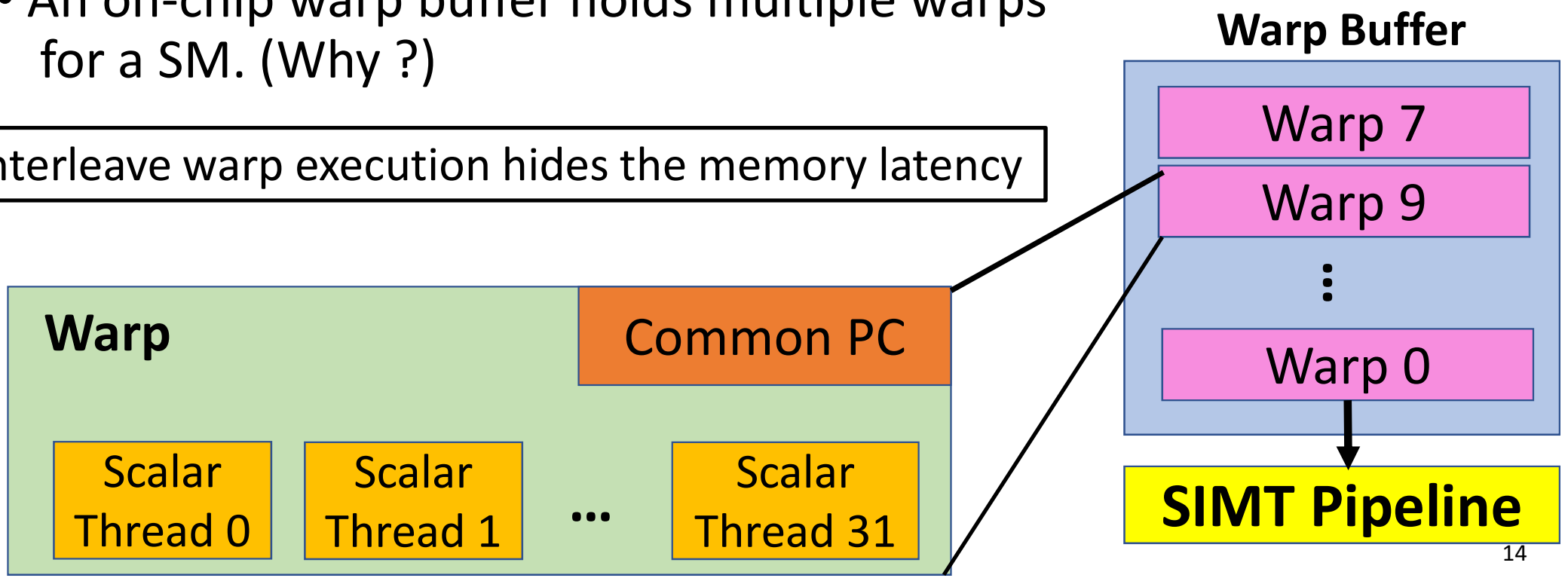
- Kernel = multiple threads grouped by grid , thread block or cooperative thread array (CTA) and warp (32 threads)
- A CTA includes up to **1024 threads**
- Each CTA is dispatched to a SIMT core as a unit of work
- All of warps in a CTA run in the core's pipeline until they are all done



SIMT Execution Model Revisited

- A thread within a warp is mapped to a ALU core in a SM
- A SM has multiple ALU core (64, 128 etc..)
- An on-chip warp buffer holds multiple warps for a SM. (Why ?)

Interleave warp execution hides the memory latency

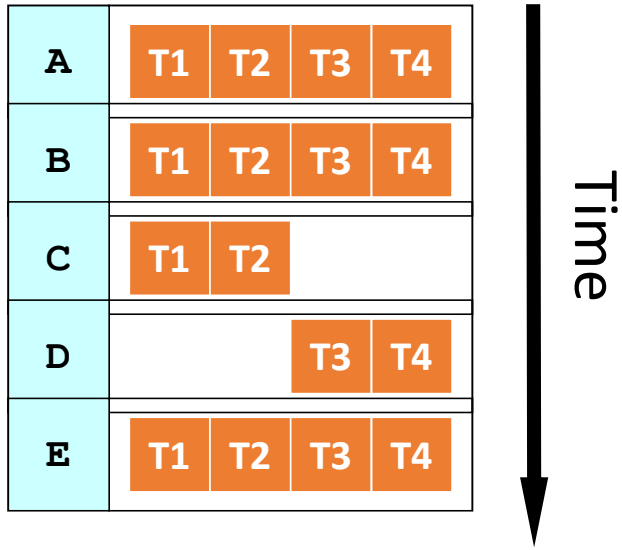


SIMT Execution Model

- 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



CUDA Programming Syntax

- Declaration Specifiers

	Execution on	**Callable from:**
__global__ void vadd(…)	Device	Host
__device__ void bar(…)	Device	Device
__host__ void func(…)	Host	Host

- Syntax for kernel launch
  - Foo<<<256, 128>>>(…); //256 thread blocks, 128 threads each
- Built in variables for thread identification
  - dim3 threadIdx.x, threadIdx.y, threadIdx.z;
  - dim3 blockIdx.x, blockIdx.y, blockIdx.z;
  - dim3 blockDim.x, blockDim.y, blockDim.z;

Example: SAXPY C Code

```
void saxpy_serial(int n, float a, float *x, float *y)
{
  for (int i = 0; i < n; ++i)
     y[i] = a*x[i] + y[i];
}

int main() {
  // omitted: allocate and initialize memory
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY
  kernel
  // omitted: using result
}
```

SAXPY CUDA Code

```
__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) y[i]=a*x[i]+y[i];
}

int main() {
  // omitted: allocate and initialize memory
  int n = 256
  int nblocks = n / 256;
  cudaMalloc((void**) &d_x, n);
  cudaMalloc((void**) &d_y, n);
  cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  // omitted: using result
}
```

CUDA Programming Revisited

- threadIdx.x [0 - 31], blockDim.x [32], blockIdx.x [0-15]

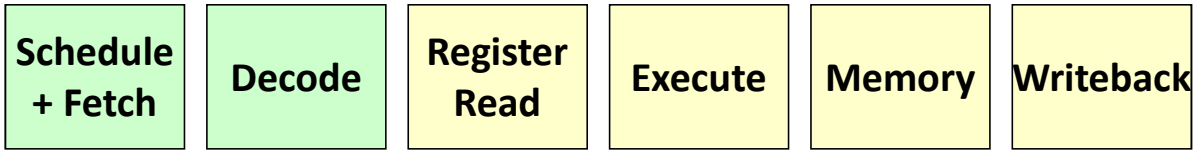
```
__global__ void MatAdd(float A[N], float B[N], float C[N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;

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

int main() {
  …
  dim3 numblocks(16, 1); // # of CTAs in one grid
  dim3 threads(32, 1); // # of threads in one CTA

  MatAdd<<<numblocks, threads >>>(A, B, C);
  …
}
```

CUDA Programming Revisited

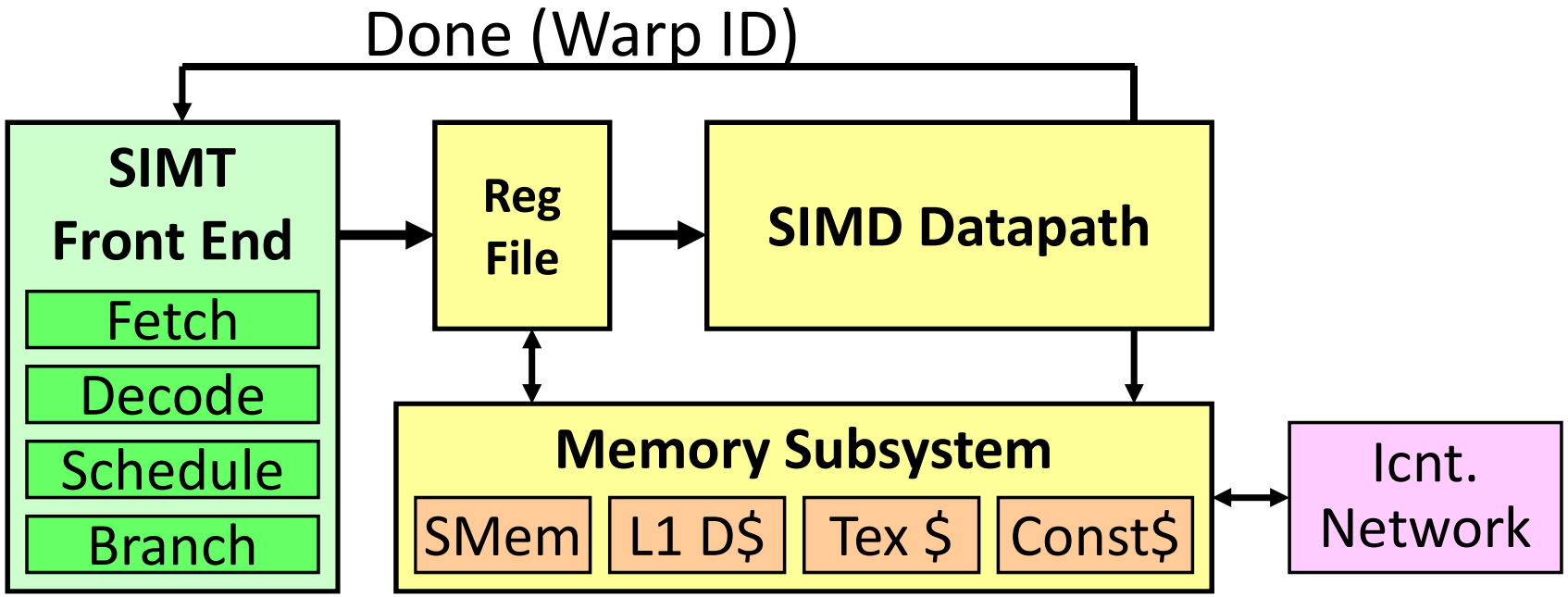
- What is performance problem shown in this implementation?
  - Each CTA has only “1” thread -> under-utilize SIMT lanes

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    if(i< N && j < N) C[i][j]=A[i][j]+B[i][j];
}

int main() {
  …
  dim3 numblocks(N, N); // total CTAs in one kernel
  dim3 threadsPerBlock(1, 1); // N x N threads in one CTA

  MatAdd<<<numblocks, threadsPerBlock>>>(A, B, C);
  …
}
```

SIMT Pipeline



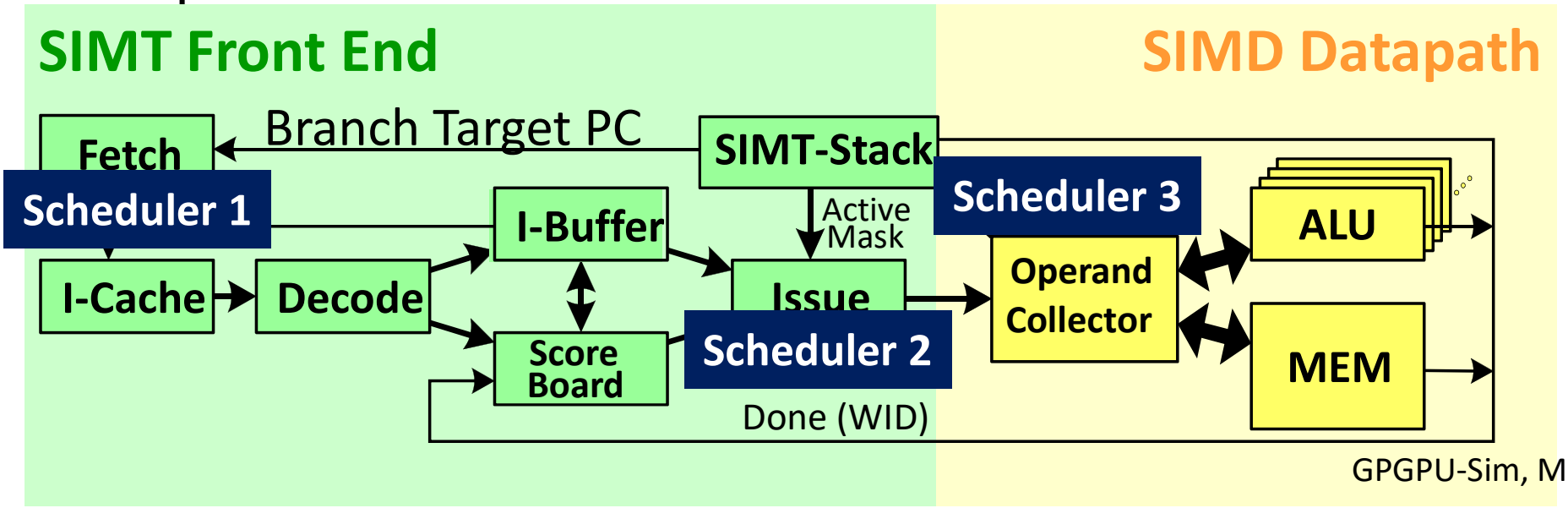
- 5 stage In-Order SIMT pipeline
- Register values of all threads stays in core



GPGPU-Sim, MICRO

Inside a SIMT Core

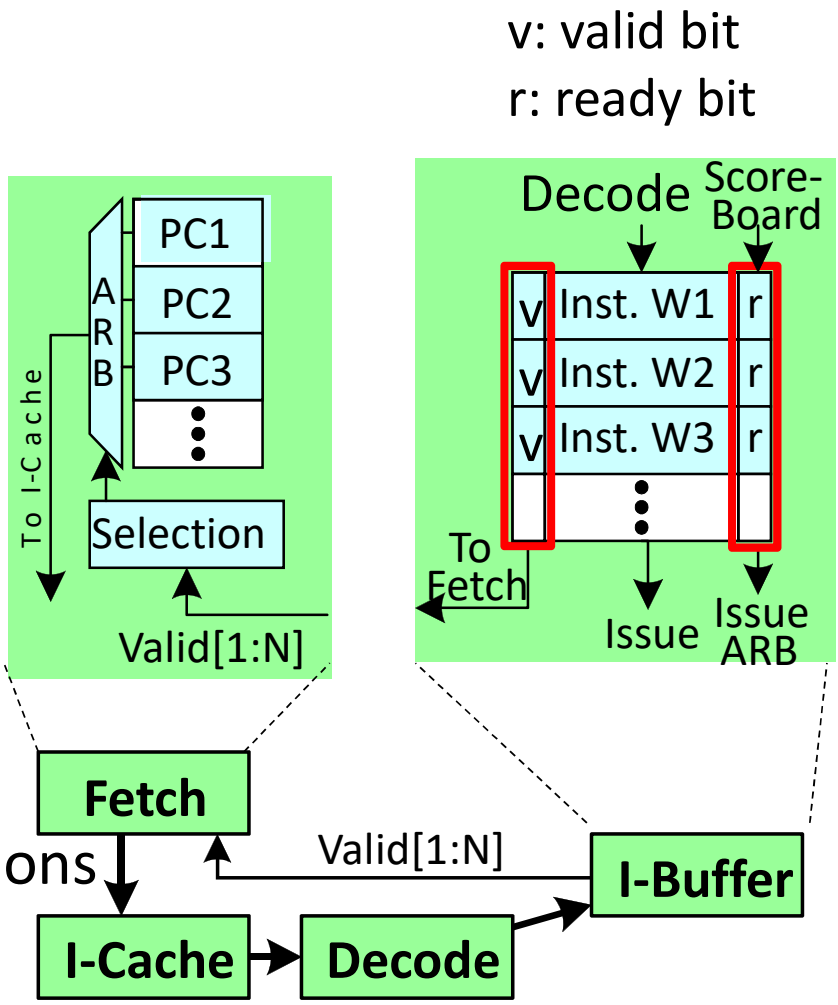
- Fetch, Warp Issue, and Operand Schedulers
- Scoreboard ->data hazard and SIMT stack->control flow
- Large register file
- Multiple SIMD functional units



GPGPU-Sim, MICRO

Fetch + Decode

- **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



Scoreboard

- Scoreboard keeps track of dependencies between instructions that have already issued
- **Out-of-order execution divides ID stage**
  - **Issue:** decode instructions, check for structural hazards
  - **Read operands**: wait until no data hazard, then read operands
- Instructions execute whenever no dependent on previous instructions and no hazards
- In order issue, out-of-order execution, commit (completion)
  - No register renaming

SIMT Stack

- Handle Branch Divergence
  - Top-of-stack entry is popped when a warp reaches to reconvergence point
  - Active mask indicates the diverging path of threads

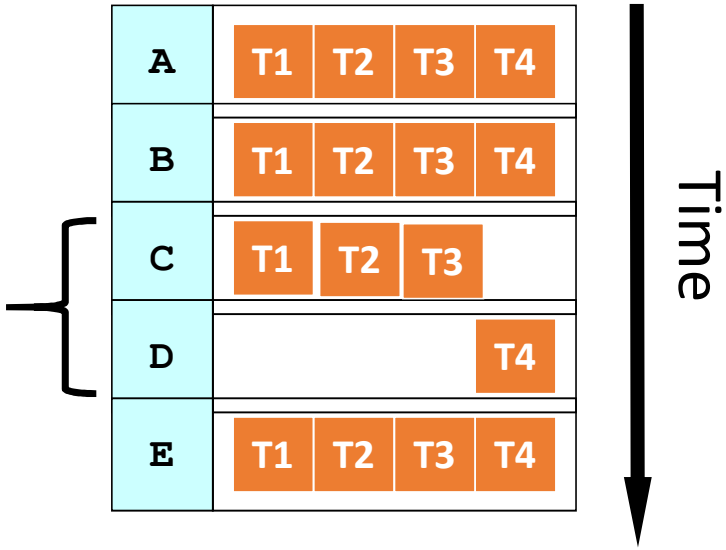
One stack per warp

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



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

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	R0: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	W1:i2:r1	W0:i2:r5	W1:i2:r5
2	W3:i1:r6		W3:i1:r2			
3						

Register Bank Conflict

- 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 ?

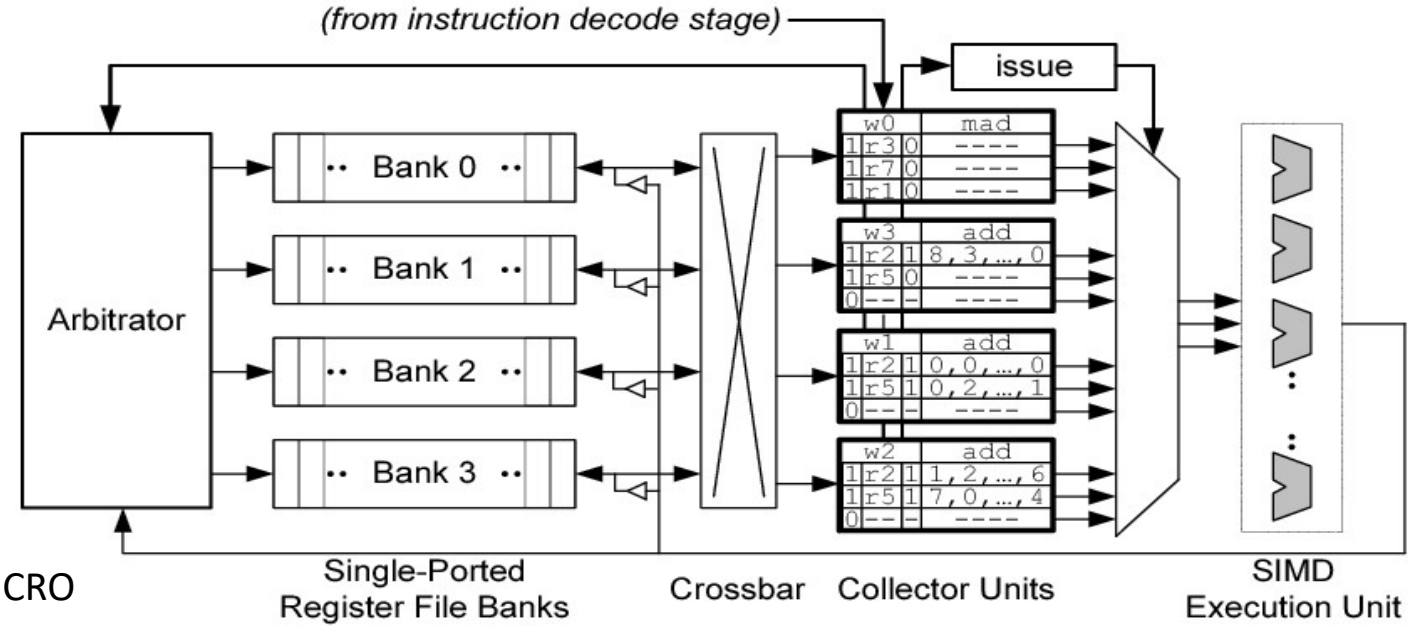
Bank 0	**Bank 1**	**Bank 2**	**Bank 3**
…	…	…	…
W1:r7	W1:r4	W1:r5	W1:r6
W1:r3	W1:r0	W1:r1	W1:r2
W0:r4	W0:r5	W0:r6	W0:r7
W0:r0	W0:r1	R0:r2	W0:r3

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

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

Operand Collector

- A valid bit, a register identifier, a ready bit, and operand data
- Arbiter selects operand that don't conflict on a given cycle
- Writeback ? (read + write port)

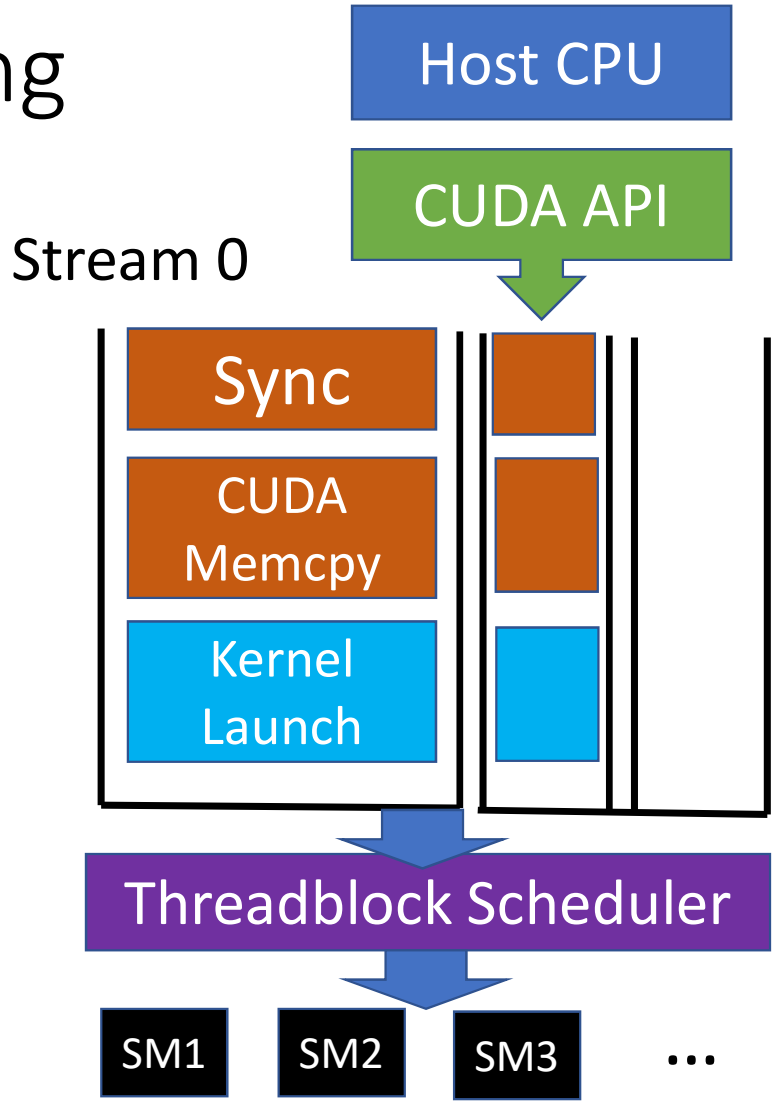


ALU Pipelines

- **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
- **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

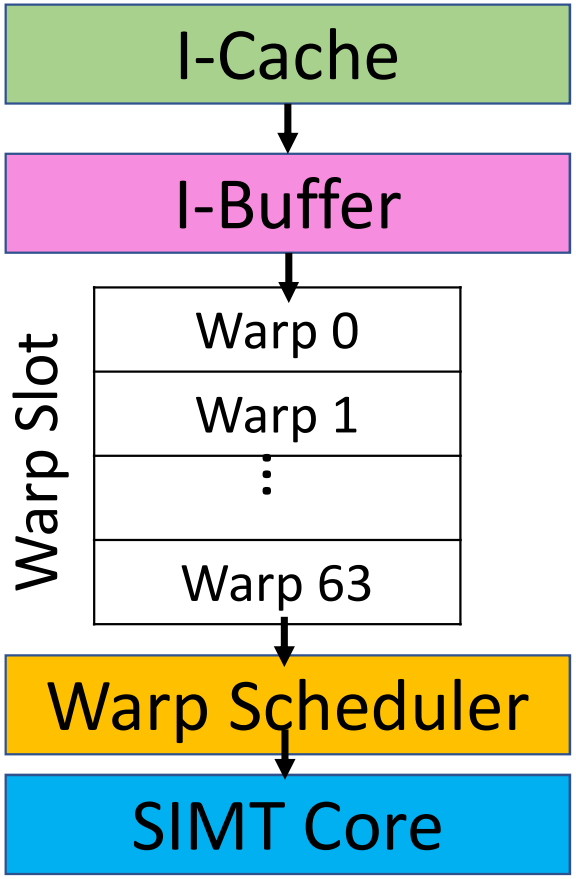
Thread Block (CTA) Scheduling

- CTA scheduler dispatches CTAs across each SM
- Scans through SMs to issue a CTA to a SM with available resources at the **round-robin manner**
- Multiple concurrent kernels
  - Different kernels can be executed across SMs



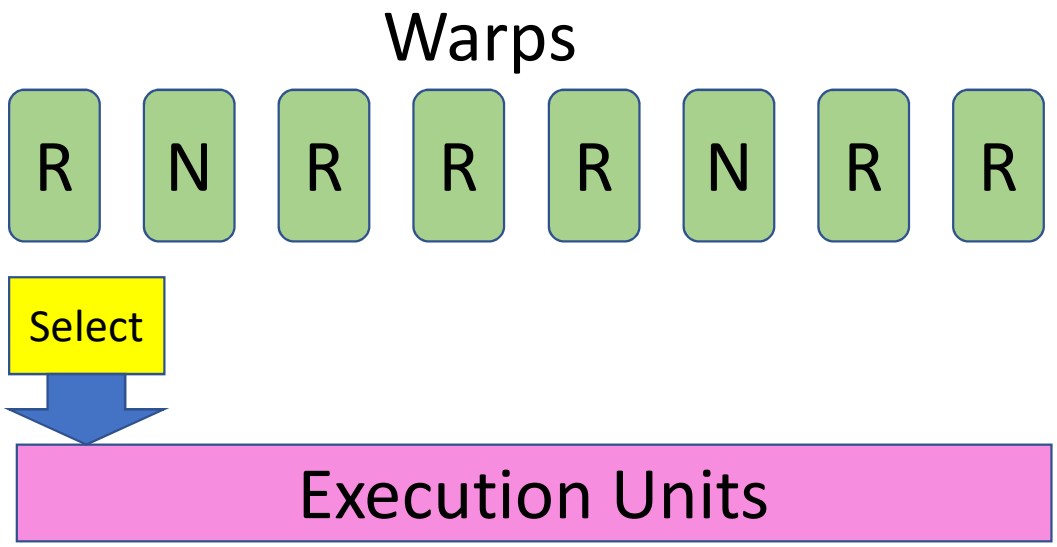
Warp Scheduling

- 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



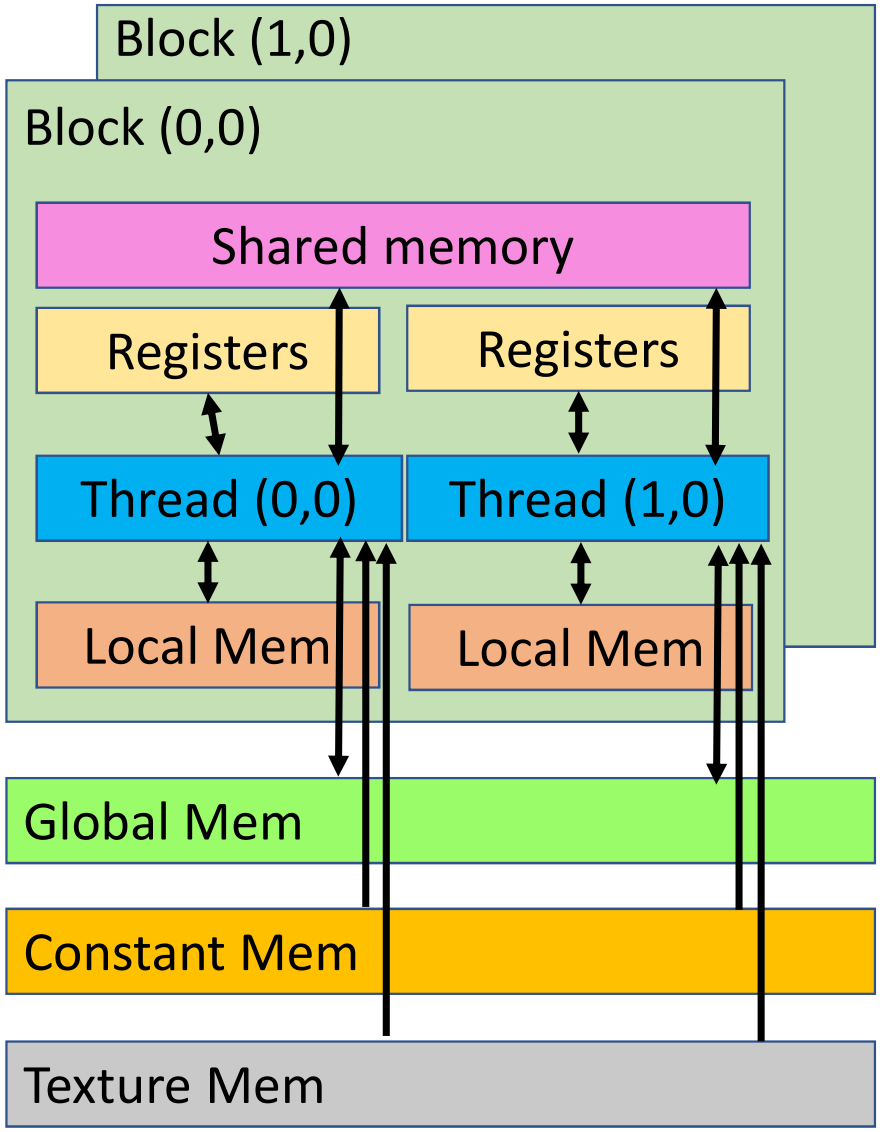
Greedy-Then-Oldest Scheduling

- 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



Memory Spaces

- **Global memory**
  - Device DRAM, shared across blocks
- **Local memory**
  - Reside in global memory
  - Each thread has private local mem space
  - Store variable data consuming too many registers
- **Shared memory**
  - On-chip memory
- **Constant/Texture memory**
  - Read-only memory
- **Register**
  - SRAM, each thread has its private register space



Global Memory

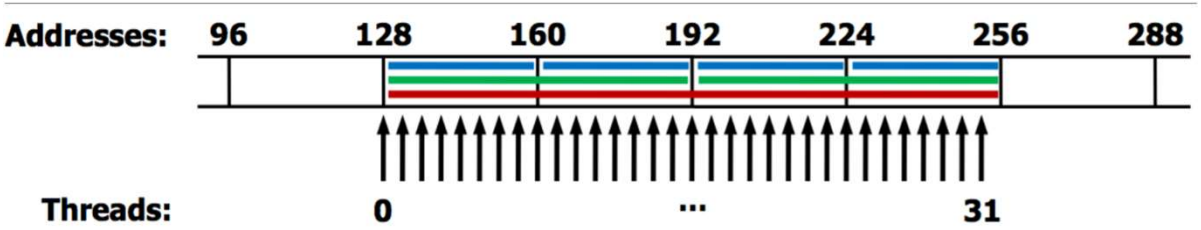
Built-in align variable:
__align__(int mem_byte)

- Global memory resides in off-chip DRAM
- Global memory is accessed via 32, 64, 128 byte memory transaction
- Misaligned/uncoalescing memory increases # of memory transaction

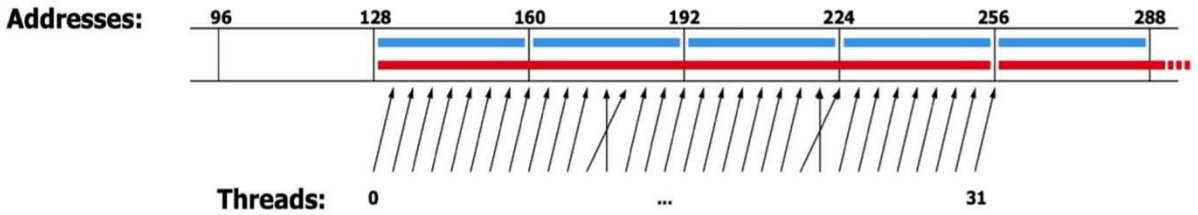
```
void kernel_copy(float *out, float *in,
int offset)
{
   int i = blockIdx.x * blockDim.x +
threadIdx.x + offset;
   out[i] = in[i];
}
```

What's wrong when offset > 0 ?

Coalesced/aligned memory access



Memory Divergent access



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

Coalescing

- Combining memory access of threads in a warp into fewer transactions
  - E.g. Each thread in a warp accesses **consecutive 4-byte memory**
  - **Coalesced:** all threads in a warp access locations that fall in a single L1 data cache block (128 bytes)
  - **Uncoalesced:** threads within a warp access different cache blocks then multiple memory accesses need to be generated
- Coalescing reduces the number of transactions between SIMT cores and DRAM
  - Less work for interconnect, memory partition, and DRAM

Quiz I

- Supposed that a 3 x 4 matrix is shown :
- Which one is coalescing access pattern ?
  - Patten B is coalescing access pattern

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & a & b & c \end{bmatrix}$$

Pattern A

Thread 0:	1, 2, 3
Thread 1:	4, 5, 6
Thread 2:	7, 8, 9
Thread 3:	a, b, c

Time

Pattern B

Thread 0:	1, 5, 9
Thread 1:	2, 6, a
Thread 2:	3, 7, b
Thread 3:	4, 8, c

Time

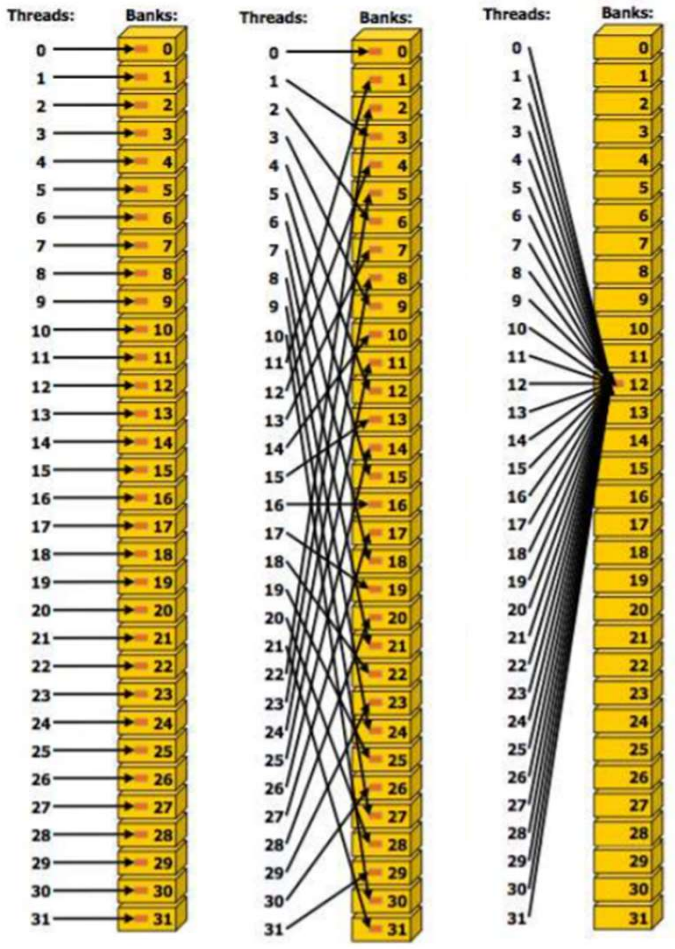
Local Memory

- Off-chip memory
- High latency and low bandwidth as the global memory
- When will use the local memory ?
  - Large structure or array that use too much register space
  - A kernel use too many register than available (**register spilling**)

Shared Memory

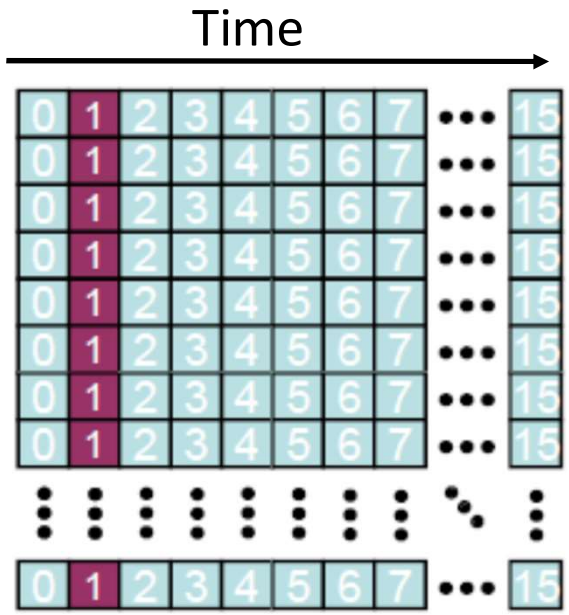
- **32 banks** organized as **32-bit successive words**
- Data shared to threads within the same CTA
- **Programmable on-chip cache**
- **Bank conflict**
  - Two or more threads access words within the same bank
  - Serialized memory access (low memory bandwidth)
- Which one is bank conflict ?
  - float i_data = shared[base + S * tid]; S = 3
  - float i_data = shared[base + S * tid]; S = 2

Which one is bank conflict ?

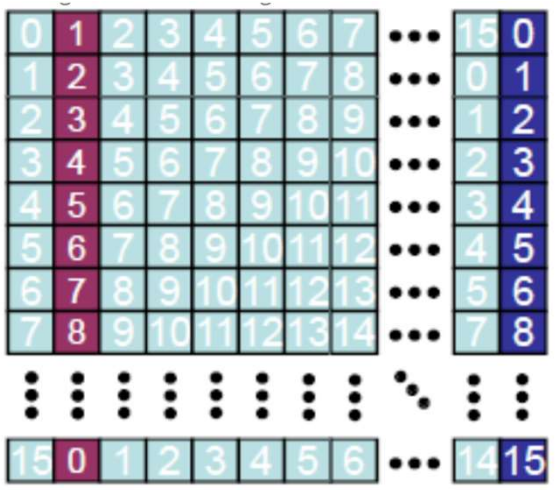


How to Resolve Bank Conflict ?

- Shared memory size is 16 x 16
- Each thread takes charge of each row operation
- Threads in one block access the same location (each column) -> 16-way bank conflict
- Solution ?
  - memory padding
  - Add one float at the end of each row
  - Changing access pattern
  - __shared__ sData[TILE_SIZE][TILE_SIZE **+ 1**]



Memory padding (blue column)



http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html

Constant Memory

- **What is the constant memory ?**
  - Optimized when **warp of threads read the same location**
  - 4 bytes per cycle through broadcasting to threads in a warp
  - Serialized when warp of threads read in different locations
  - Very slow when constant cache miss (read data from global mem.)
- Where is the constant memory (64KB) ?
  - Data is stored in the device global memory
  - Read data through SM constant cache (8KB)
- Declaration of constant memory
  - __constant__ float c_mem[size];
  - cudaMemcpyToSymbol() // copy host data to constant memory

Texture Memory

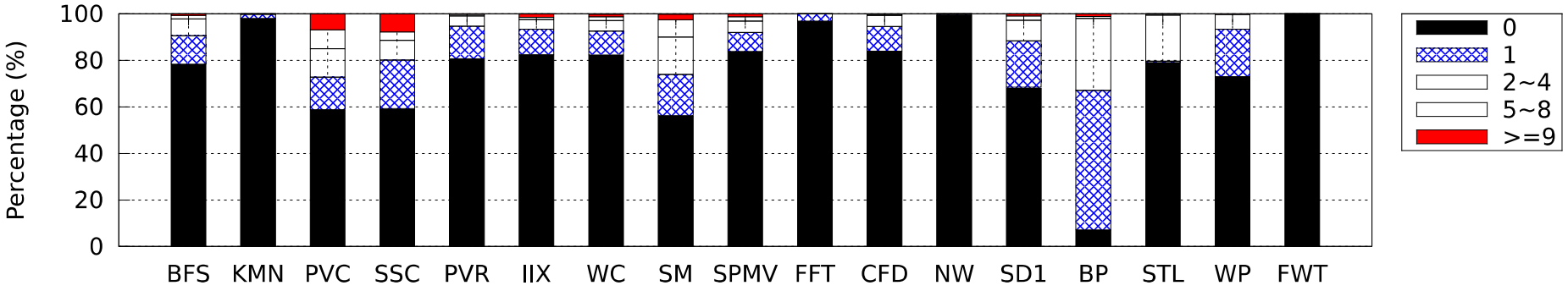
- **What is the texture memory ?**
  - Optimized for spatial locality shown among threads in blocks
  - Spatial locality implies threads of the same warp that read memory address that are close together
- Where is the texture memory ?
  - 28 – 128 KB texture cache per SM (Nvidia GPU arch. 8.6)
- Declaration of texture memory
  - text1D(texObj, x) // fetch from region of memory with texture object and coordinate x
  - text2D(texObj, x, y) // 2 D texture object with coordinate x and y

L1 Data Cache

- The first level cache per SM
- Non-coherent
- Single ported (128-Byte wide)
  - Take multiple cycles to service non-coalesced accesses

GPU Data Cache Problem

- GPU is cache inefficiency
  - Why L1 cache miss rate on GPU is high ?
  - Massive number of parallel threads increase the cache contention
  - Large cache can reduce the cache contention and improve the GPU performance (done ?)



L1 Cache Reuse Count Distribution for a GPU with 32KB L1 caches

Nearly **80%** inserted cache lines are never reused before eviction

Chen et al., MICRO 2014

Miss Status Holding Registers (MSHR)

- Tracking the status of misses in progress
- A fully-associative array
- Service a fixed number of miss requests for a single cache line
  - MSHRs are limited (configurable)
  - Memory unit stalls if cache runs out of MSHRs

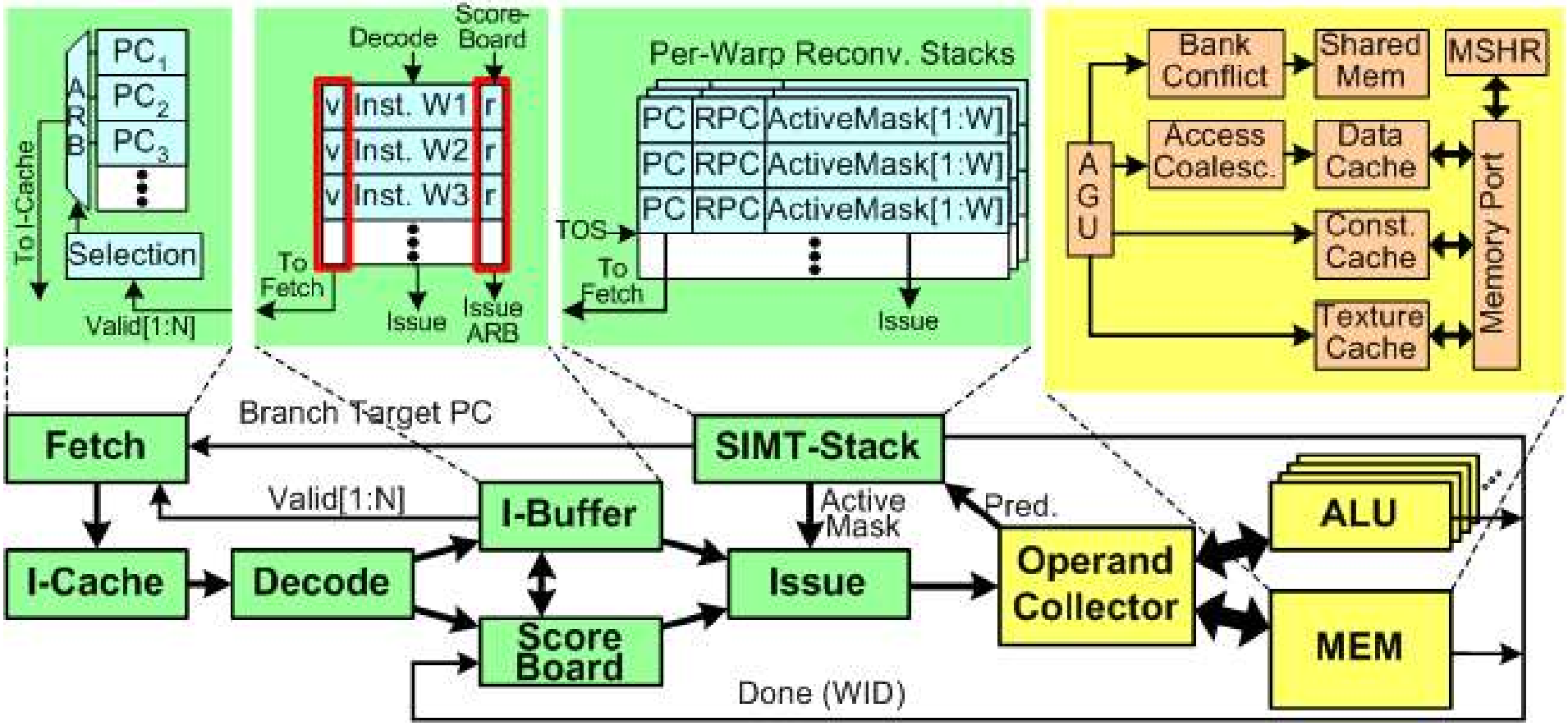
L2 Cache Bank

- A unified last level cache shared by all SM
- L1 cache request cannot span across two L2 cache lines

	Local Memory	**Global Memory**
Write Hit	Write-back	Write-back
Write Miss	Write-no-allocate	Write-no-allocate

- What are advantages of write-back policy ?
  - Fast data write speed
- Write-no-allocate
  - The cache doesn’t allocate a cache line on a write miss

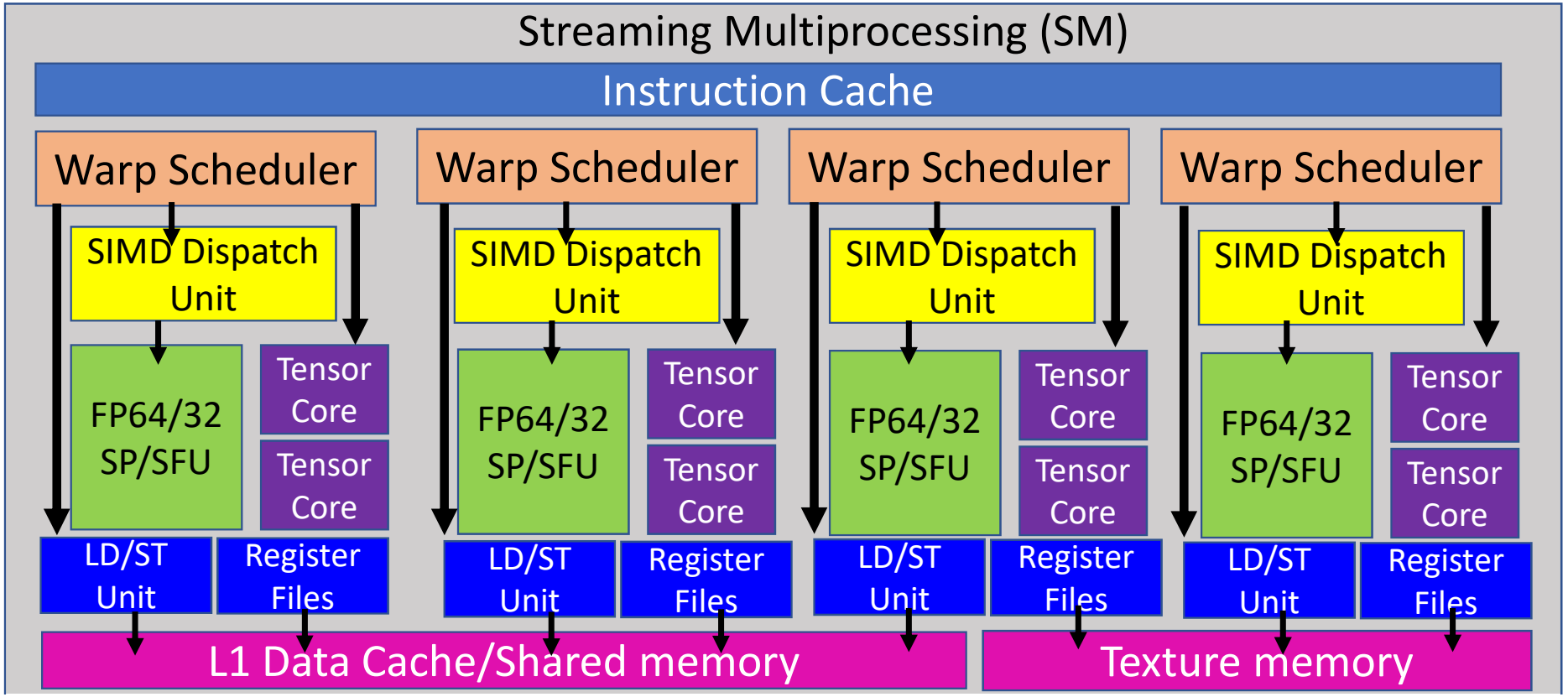
GPU Micro-architecture Summary



http://gpgpu-sim.org/manual/index.php/Main_Page

What is Tensor Core ?

- Execute 4 x 4 x 4 matrix multiplication and addition in one cycle (D = A x B + C)



Zhu et.al., MICRO 2019

Why do we need Tensor Core on GPUs ?

- Higher throughput for GEMM ?
  - A CUDA (SIMT) core offers 1 single precision multiply-and-accumulate operation per GPU cycle
  - Tensor core can multiply two 4 x 4 F16 matrices and add the multiplication product fp32 matrix per GPU cycle
  - Tensor core can achieve 125 Tflops/s vs 15.7 Tflops/s for the single precision operation
  - Domain-specific Accelerator within the GPU

PTX ISAs for Tensor Core

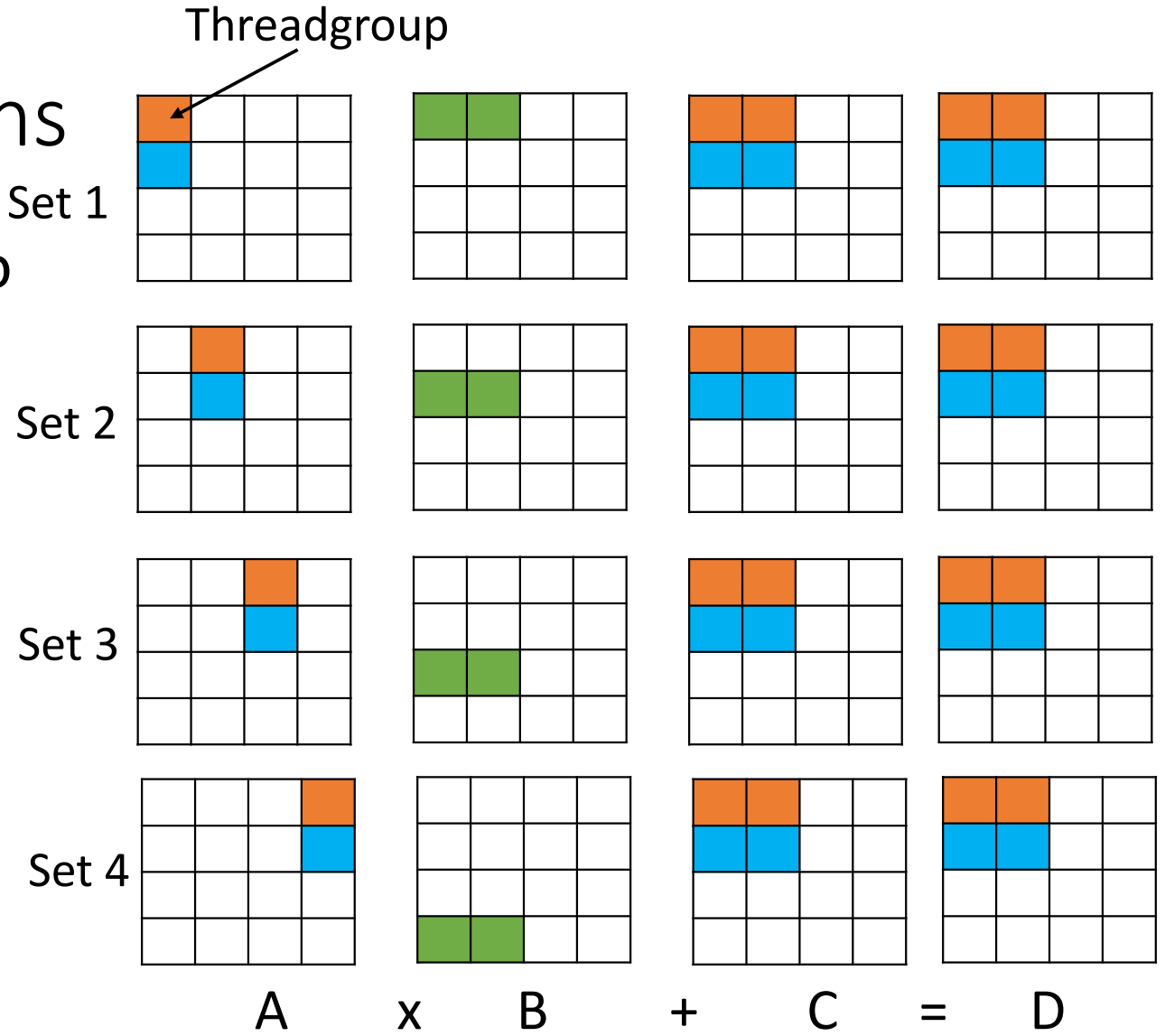
- Two execution modes
  - FP 16 mode: All matrices are FP 16
  - Mixed precision mode: FP32 accumulator to write results back to FP32
- Warp Matrix Multiply Accumulate (wmma) instruction
  - **wmma.load** // Collectively load a matrix from memory
  - **wmma.store** // Collectively store a matrix from memory
  - **wmma.mma** // Perform a single matrix multiply-and-accumulate operation across a warp
  - **Load_matrix_sync, store_matrix_sync and mma_sync** // warp-wide barrier sync.
  - Can access wwma ISAs through cuBLAS, cuDNN and CUTLASS

WMMA Operations on Tensor Core

- Given A, B, C, and D are 16 x 16 matrices
- A warp computes a matrix multiply and accumulate
  D= A x B + C
- 32 threads in a warp are divided into “**8**” **threadgroups**
- Worktuple: 2 threadgroups

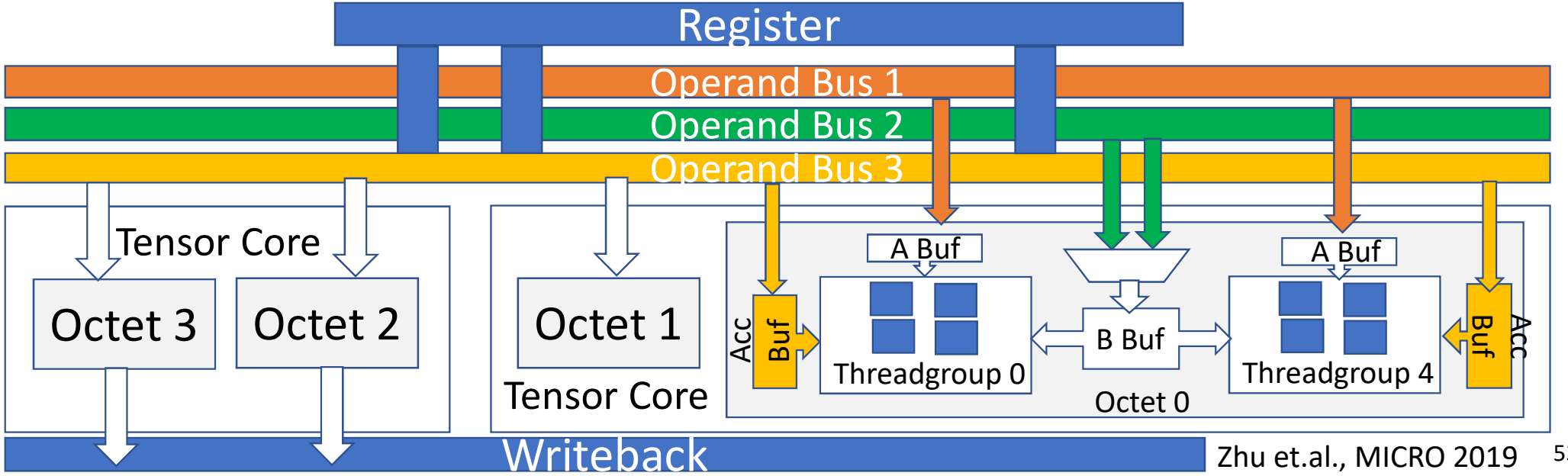
WMMA Operations

- One WMMA breaks into 4 set of HMMA (SASS)
- Each set of HMMA instruction computes a product **4 x 4** tile of A and **4 x 8** tile of B
- Two threadgroups of worktuple share 4 x 8 tile of B
- 4 x 4 tile of A is private to each theadgroup



Tensor Core Details

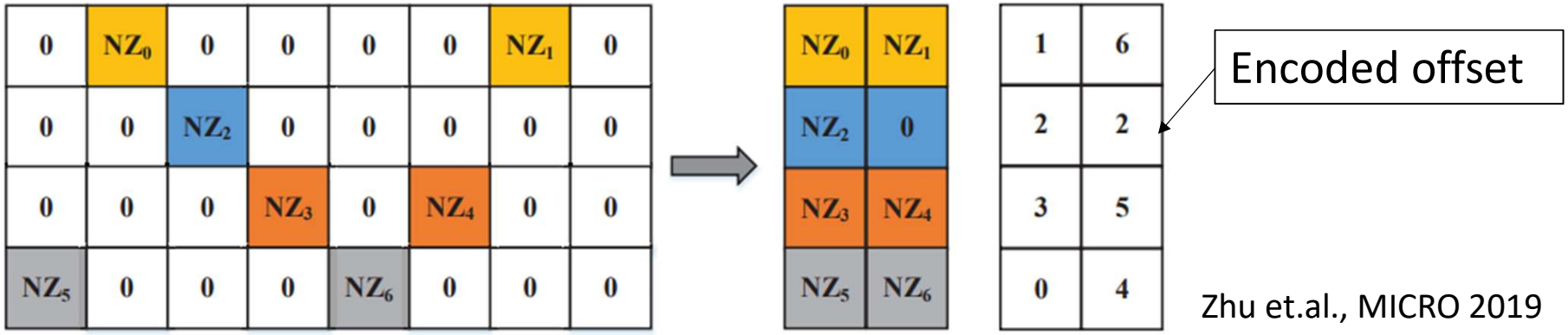
- Each Octet has 8 dot product (DP) units
- A DP unit can compute 4-dim vector DP per cycle
- Operand Buffer A can hold a 4 x 4 tile, buffer B holds a 4 x 8 tile
- 4 DP units compute four 4-dim DP/cycle, 8 cycles for 4 x 8 x 4 matrix mul.



Zhu et.al., MICRO 2019

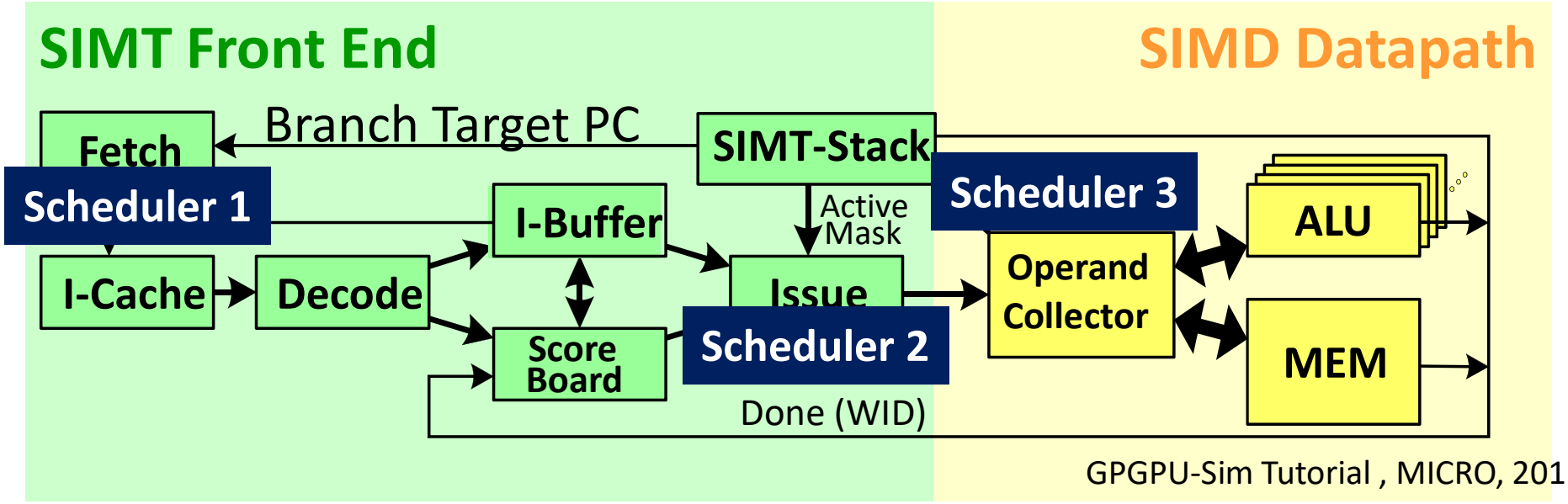
Sparse Tensor Core

- Improve tensor core utilization in sparse MMA
- Sparse MMA is used in the compressed model
- Data encoding + tensor core mapping



Conclusion

- **Programmable GPU** accelerates apps with massive parallelism
- GPU follows SIMT execution model
- GPU Tensor Core increases the throughput of ML apps



GPGPU-Sim Tutorial , MICRO, 2012