Accelerator Architectures for Machine Learning (AAML)

Lecture 8: Tensor Core

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

Acknowledgements and Disclaimer

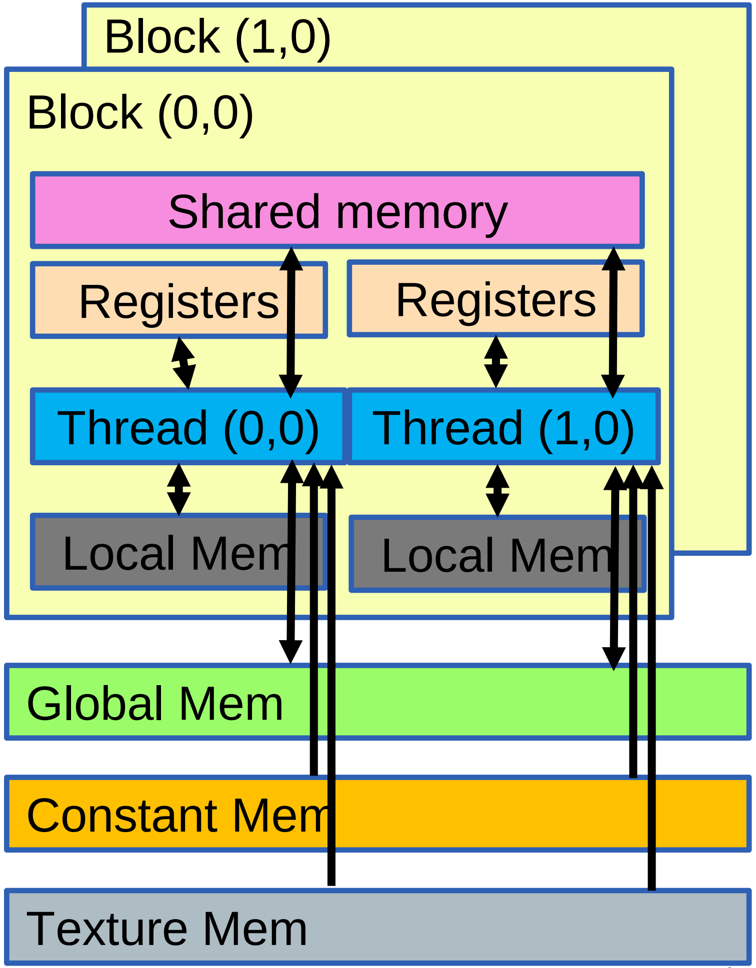
- Slides was developed in the reference with
  Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019 tutorial
  Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
  Yakun Sophia Shao, EE290-2: Hardware for Machine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

Outline

- GPU Memory System
- Tensor Core on the GPU
- Tensor Memory Accelerator

GPU Memory System

- **Global memory**
  - Device DRAM, shared across blocks
- **Local memory**
  - Reside in global memory
  - Store variable data consuming too many registers (register spilling)
- **Shared memory**
  - On-chip addressable memory
  - Direct mapped
- **Constant/Texture memory**
  - Read-only memory
- **Register File**
  - Each thread has its private register space



Global Memory

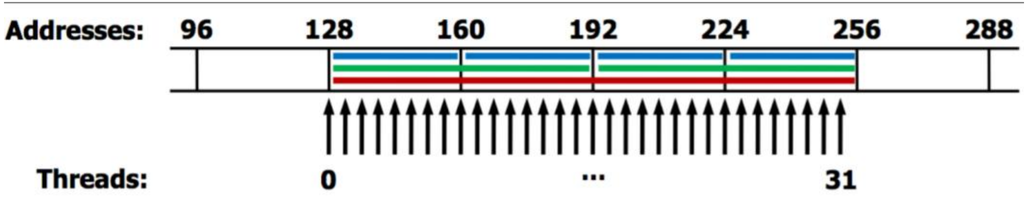
Built-in align variable:
__align__(int mem_byte)

- 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

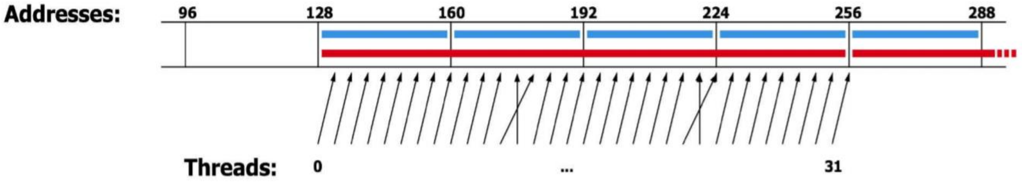
```
void kernel_copy(float *out, float *in, int offset)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x + offset;
    out[i] = in[i];
}
```

What’s wrong when offset > 1 ?

Coalesced/aligned memory access



Memory Divergent access



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

Memory Coalescing

- **Coalesced access**
  - If all threads in a warp access locations that fall within a single L1 data cache block and that block is not present in the cache
  - Only a single request needs to be sent to the lower level caches
- **Un-coalesced access**
  - If the threads within a warp access different cache blocks
  - Multiple memory accesses need to be generated

Memory Coalescing

- Combining memory access of threads in a warp into fewer transactions
  - E.g. Each thread in a warp accesses consecutive 4-byte memory
  - Send one 128-byte request to DRAM (Coalescing)
  - Instead of 32 4-byte requests
- Coalescing reduces the number of transactions between SIMT cores and DRAM
  - Less work for interconnect, memory partition, and DRAM

Memory Coalescing

- Supposed that a 3 x 4 matrix is shown : $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & a & b & c \end{bmatrix}$
- Which one is coalescing access pattern ?
  - Pattern B is coalescing access pattern

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

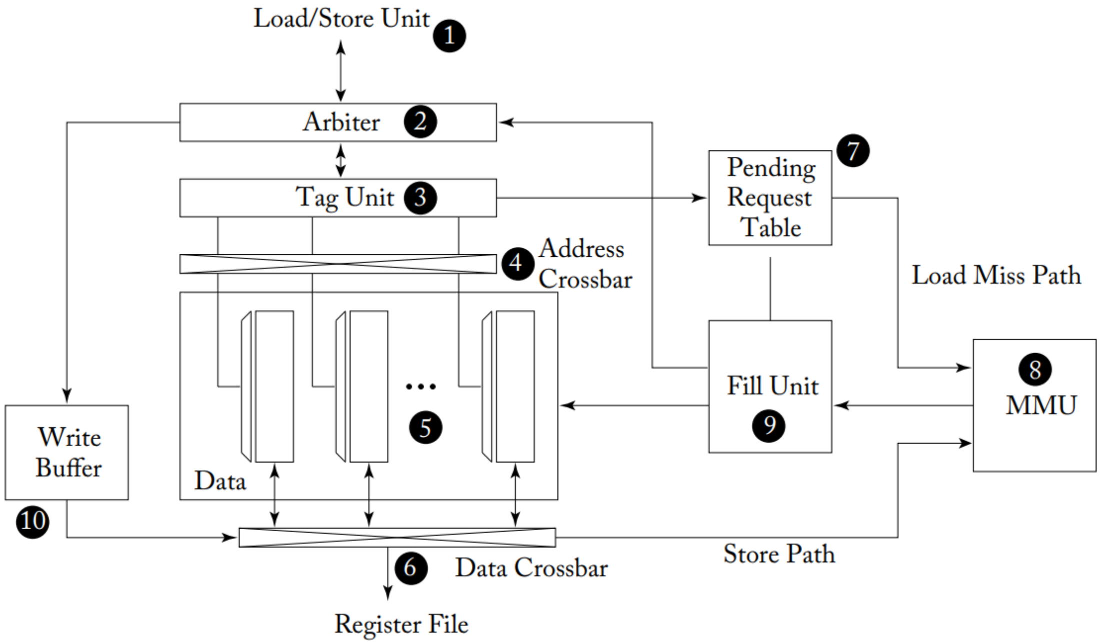
Time →

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 uses too many register than available (register spilling)

Data Cache & Shared Memory

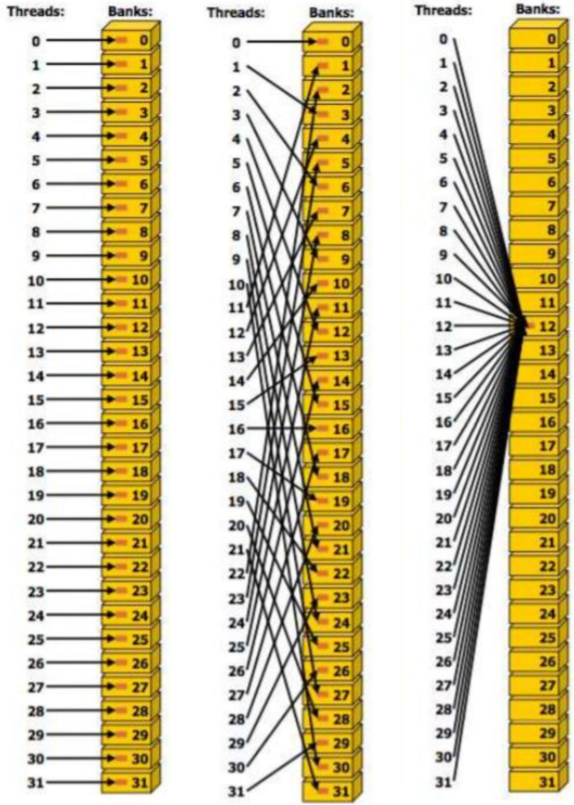
- A memory access request is first sent from the load/store unit inside the instruction pipeline to the L1 cache



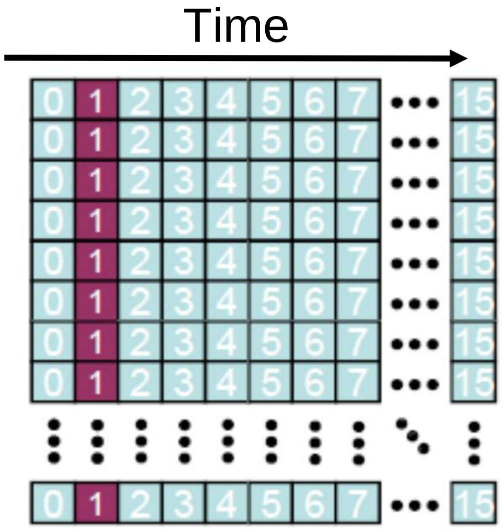
Shared Memory

Which one is bank conflict ?

- 32 banks organized as 32-bit successive words
- Threads share data in the same thread block
- Programmer-managed 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
  - double i_data = shared[base + tid]
  - char i_data = shared[base + tid]

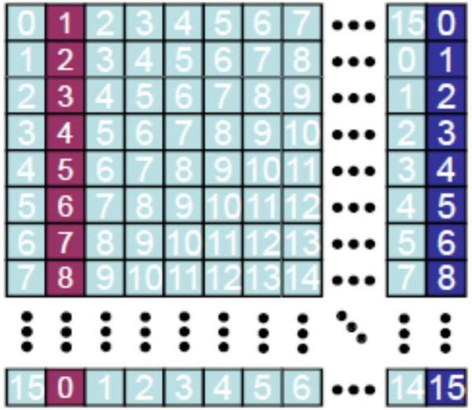


How to Resolve Bank Conflict ?



- 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**]

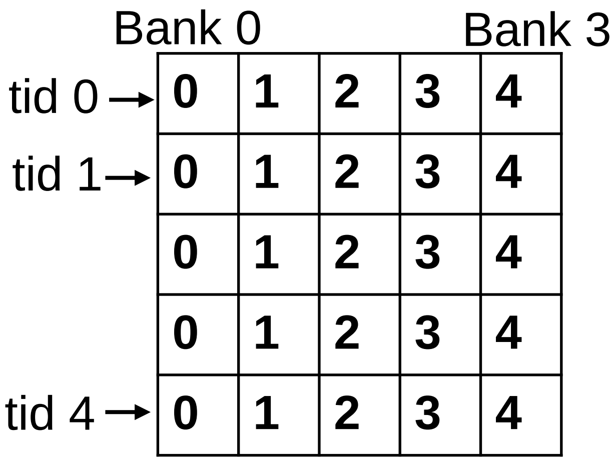
Memory padding (blue column)

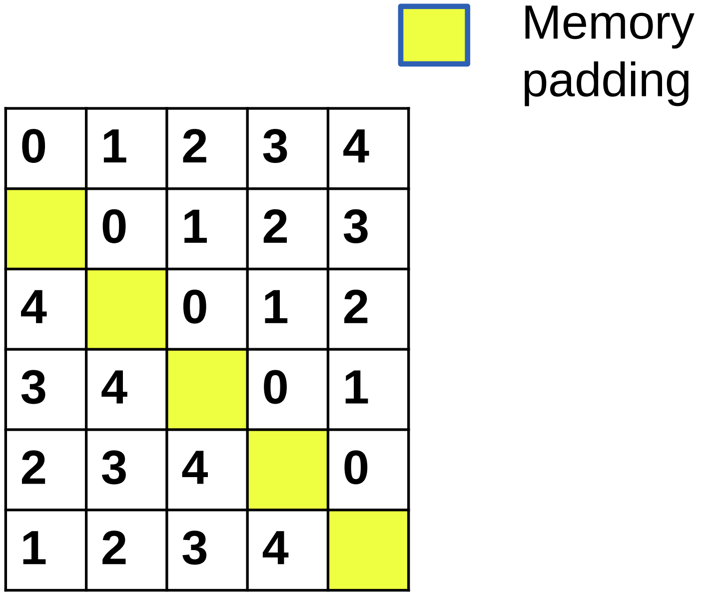


http://cuda-programming.blogspot.com/2013/02/bank-conflicts-in-shared-memory-in-cuda.html

How to Resolve Bank Conflict ?

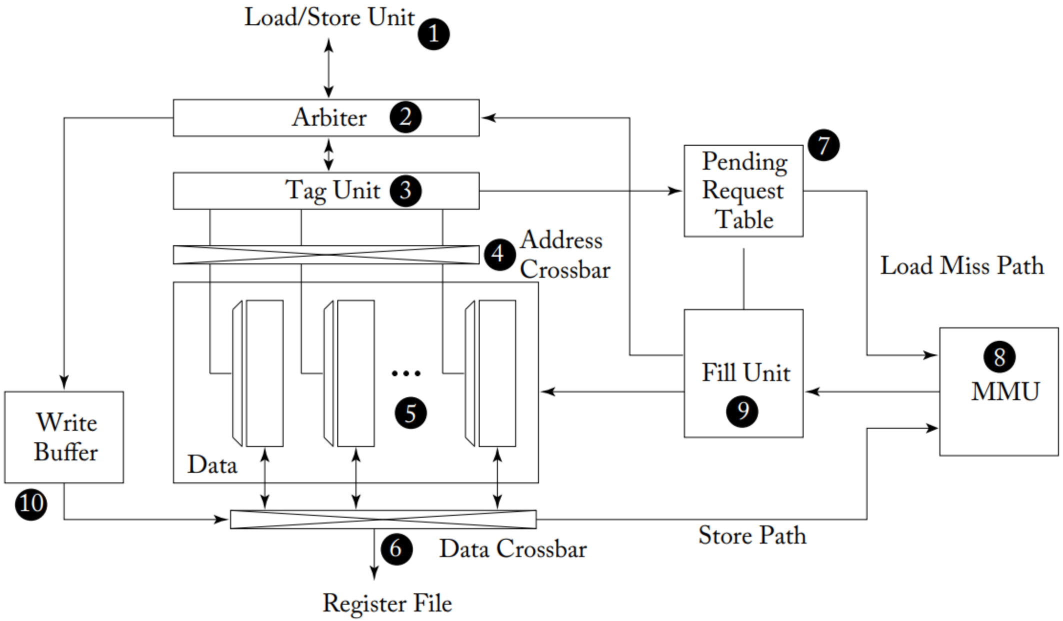
- Memory padding is one of solution to remove shared memory bank conflict
  - __shared__ a[32][32] -> __shared__ a[32][33]



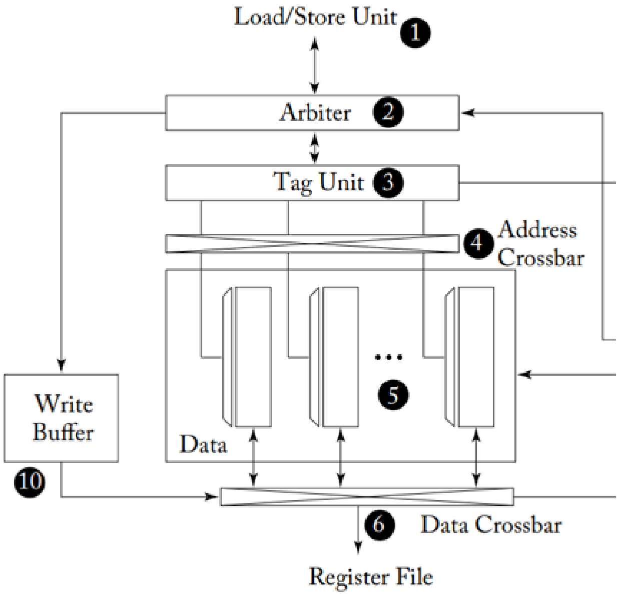


Shared memory access

- **Arbiter**
  - Determine whether the requested addresses within the warp will cause bank conflict
  - Split the request into two parts when the bank conflicts show
- **Accepted request**
  - Bypass tag lookup in the tag unit, since shared memory is direct mapped

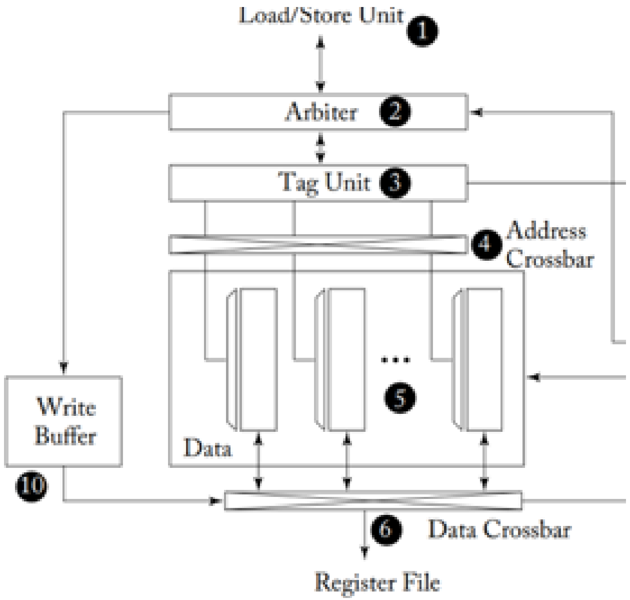


Shared memory access



- **In the absence of bank conflict**
  - The latency of the direct mapped memory lookup is constant (single-cycle)
  - The tag unit determines which bank each thread's request maps to
  - The address cross bar distributes address to the individual banks within the data array
  - Each bank inside the data array is 32-bits wide
  - Each bank has its own decoder allowing from independent access to different rows in each bank
  - The data is returned to the appropriate thread's lane for storage in the register file via the data crossbar

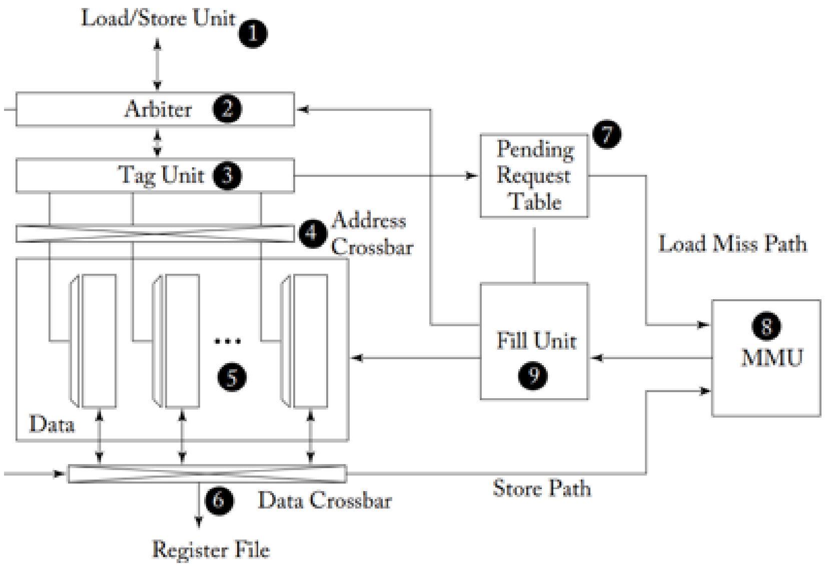
L1 Data Cache Read



- 1) The LD/ST unit
  - Computes memory addresses
- 2) The arbiter
  - Requests the instruction pipeline schedule a writeback to the register file if enough resources are available
- 3) The tag unit
  - Check whether the access leads to a cache hit or a miss
- 4) Access the appropriate row of the data array
  - In the event of a cache hit

L1 Data Cache Read

- 5) Pending request table (PRT)
  - The tag unit determines a cache miss
  - The arbiter informs the LD/ST unit to replay the request and sends request information
- 6) Memory Management Unit (MMU)
  - After an entry is allocated in the PRT
  - Virtual to physical address translation
- 7) Fill unit
  - Use the subid field in the memory request to lookup information about the request in the PRT



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 threads in a warp 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 GPU 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 ?
  - <u>Optimized for spatial locality</u> shown among threads in blocks
  - <u>Spatial locality implies threads of the same warp that read memory addresses are close together</u>
- 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

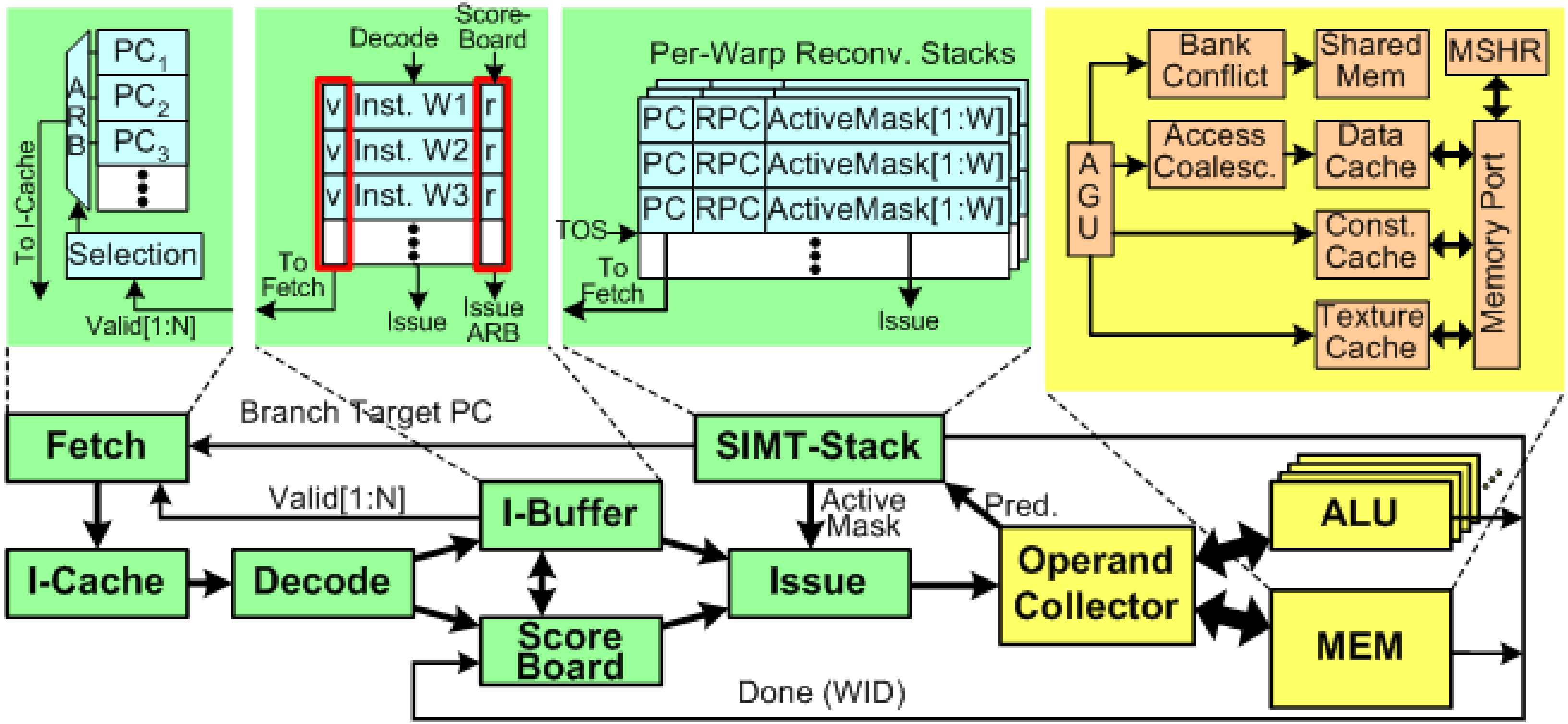
L2 Cache Bank

- A unified last level cache shared by all SIMT cores
- 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
  - in a "write miss" (the data is not in the cache), the data is written directly to main memory instead of loading the data block from memory into the cache first

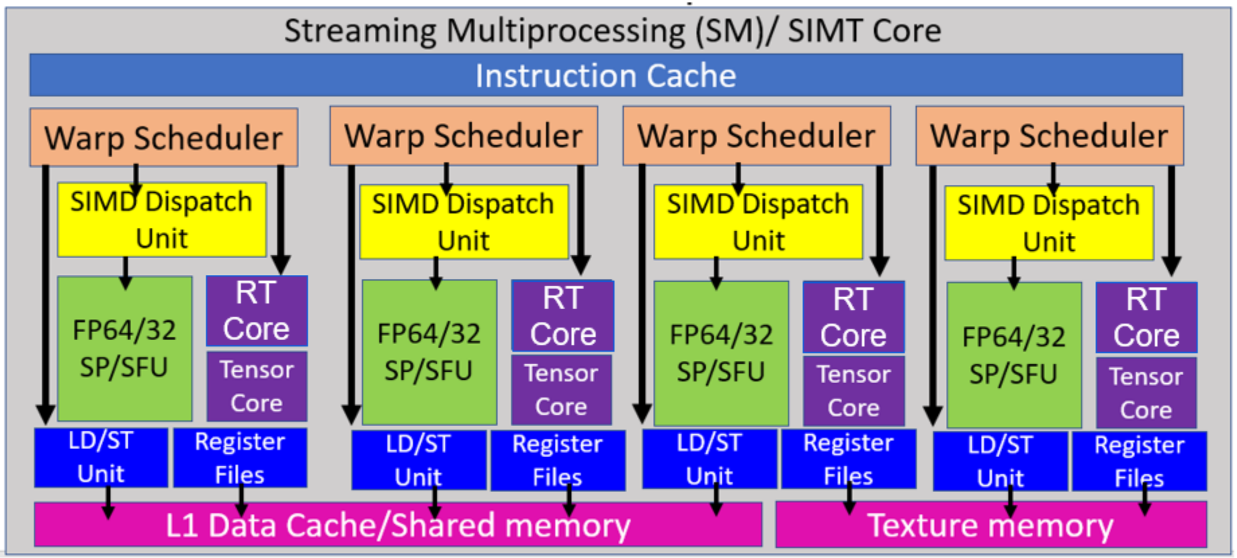
GPU Micro-architecture

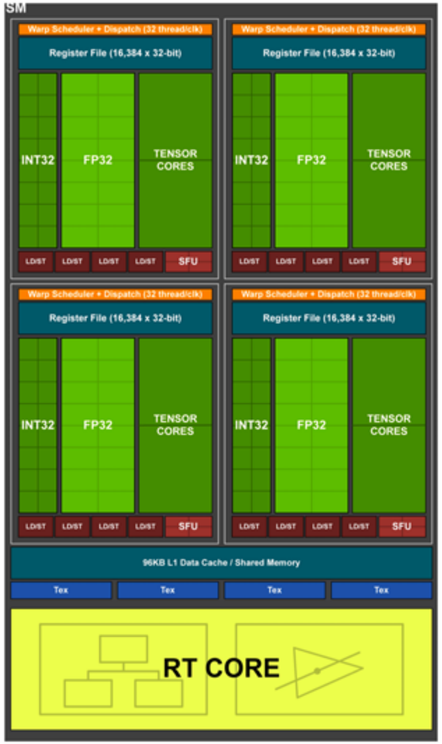


http://gpgpu-sim.org/manual/index.php/Main_Page

GPU Architecture Overview

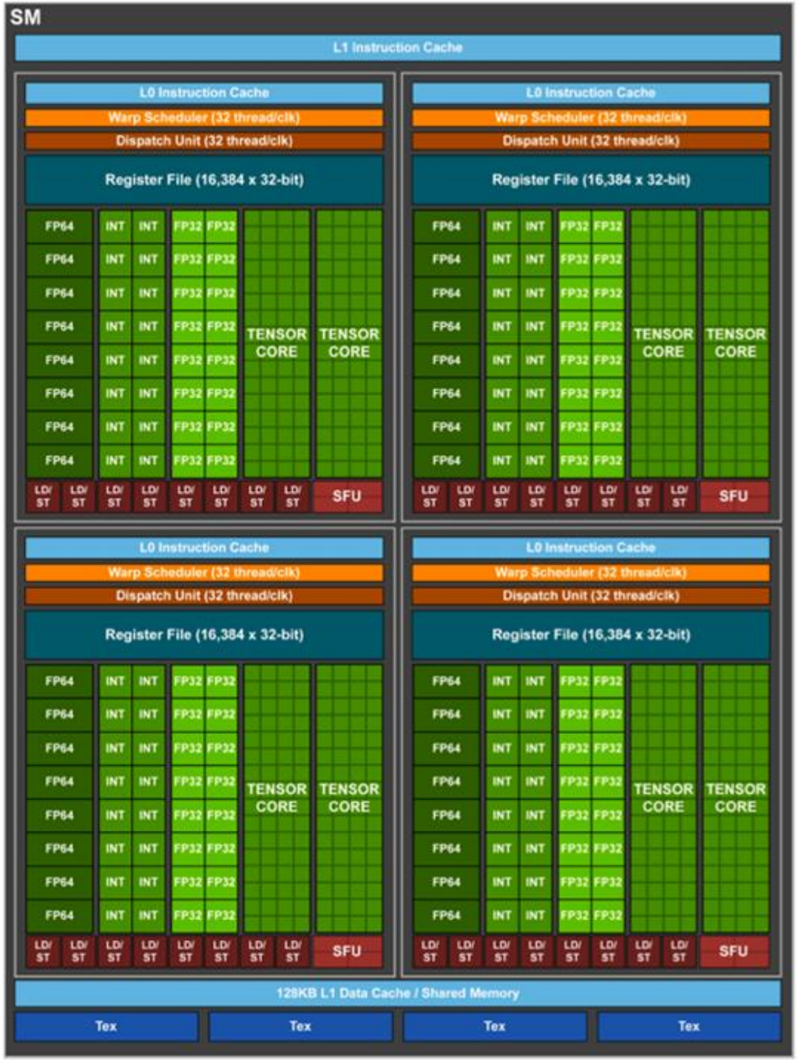
- GPU includes FP, SFU (Special Functional Unit), Ray Tracing (RT) Core, and Tensor Core





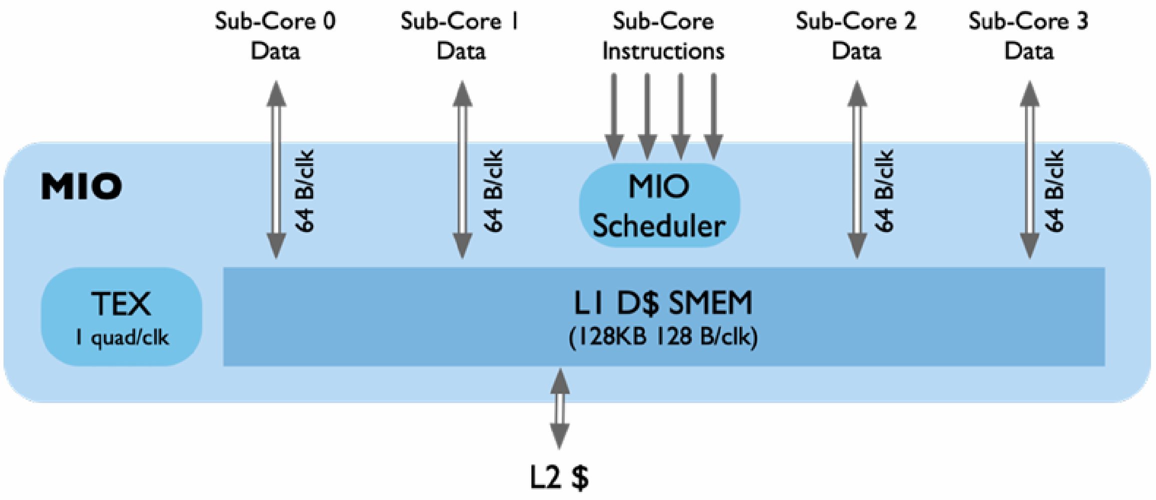
GPU Architecture (Volta)

- GPU includes
  - FP
  - SFU (Special Functional Unit)
  - Ray Tracing (RT) Core
  - Tensor Core



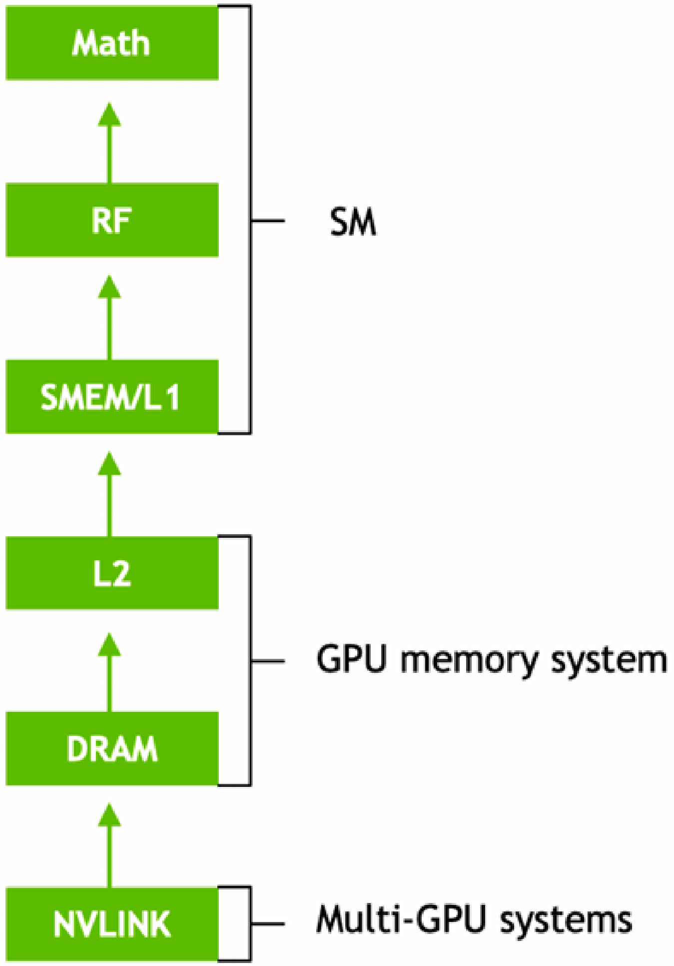
GPU Architecture (Volta)

- Shared MIO (TEX/L1$/SMEM)
  - Unified storage with L1$



Ampere GPU

- **Global Memory** access
  - ~380 cycles
- **L2** memory access
  - ~200 cycles
- **Shared memory/L1** cache access
  - ~34 cycles
- **Fused multiply-and-add (FFMA)**
  - 4 cycles (a*b+c)
- **Tensor core matrix multiply**
  - 1 cycles



$A \times B = C, \ where \ A \in R^{32 \times 32}, B \in R^{32 \times 32}$

Matrix Multiply on GPU w/o Tensor Core

- Assume a GPU has 8 SMs and each SM has 8 warps
- One SM conducts the matrix multiplication of
  - Matrix A’ (4 x 4) and Matrix B’ (4 x 32)
- There are 8 warps in one SM
  - Each warp handles Matrix (4 x 4) multiplies Matrix (4 x 4)
  - Each thread conducts Matrix (1 x 4) multiples Matrix (1 x 4)
  - How many threads in a warp are in active?
    - 16 = (4 x 4) * (4 x 4)/(1 x 4)*(1 x 4)

$A \times B = C, \ where \ A \in R^{32 \times 32}, B \in R^{32 \times 32}$

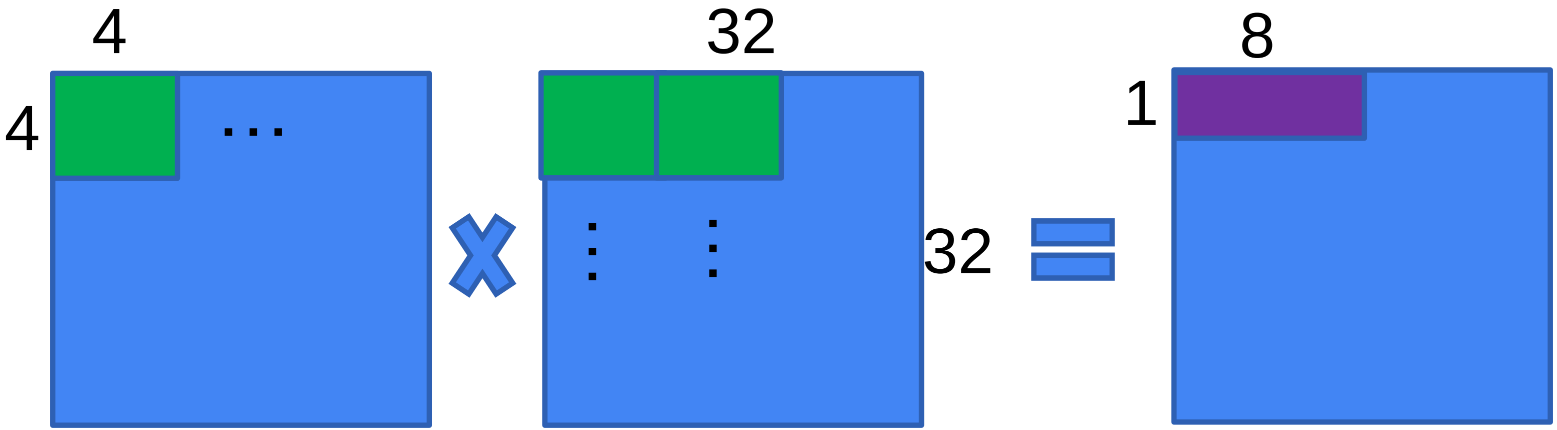
Matrix Multiply on GPU w/o Tensor Core

- Assume a GPU has 8 SMs and each SM has 8 warps
- Load matrix A and matrix B from DRAM to shared MEM
- How much data we need to load when matrix size is 32 x 32?
  - 4 bytes x 32 x 32 x 2
  - Loading two 32 x 32 floats into a shared memory tile can happen in parallel by using 2 x 32 warps
  - Therefore, we only need to do a single sequential load from global to shared memory, which takes 200 cycles

$A \times B = C,\ where\ A \in R^{32\times32}, B \in R^{32\times32}$

Matrix Multiply on GPU w/o Tensor Core

- Each SM does 8X dot products to compute 8 outputs of C
  - 16 threads in a warp handle a 4 x 4 tile, 32 threads in a warp tackles two 4 x 4 tiles
  - Each SM accumulates 8 partial results

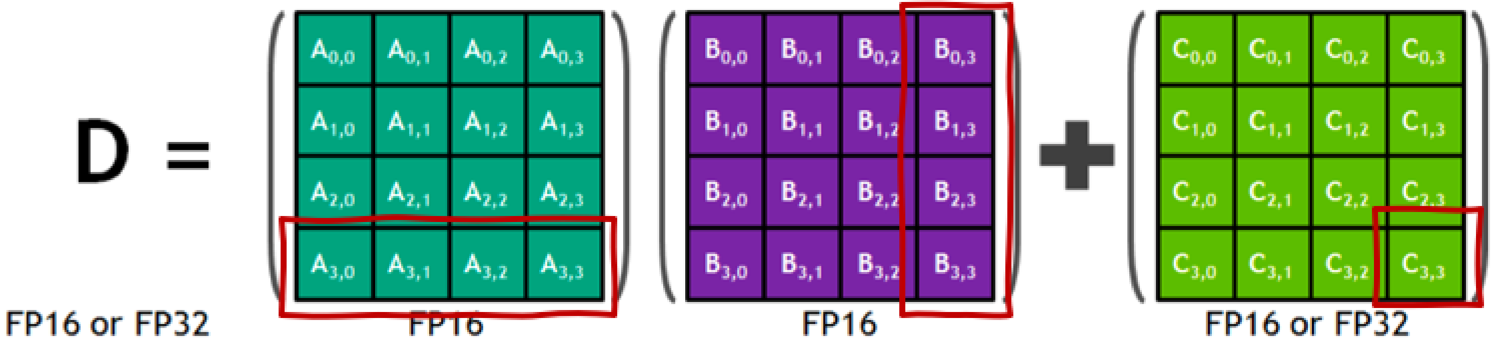


$$A \times B = C, \ where \ A \in R^{32\times32}, B \in R^{32\times32}$$

Matrix Multiply on GPU w/o Tensor Core

- How many shared memory access do we need during such a GEMM?
  - 8 times shared memory access
  - Execute 8 times FFMA
  - What is the total cycles spend in the 32 x 32 matrix multiply?
    - 200 cycles (Global Memory) + 8 x 34 cycles (shared memory) + 8 x 4 cycles (FFMA) = 504 cycles

Tensor Core



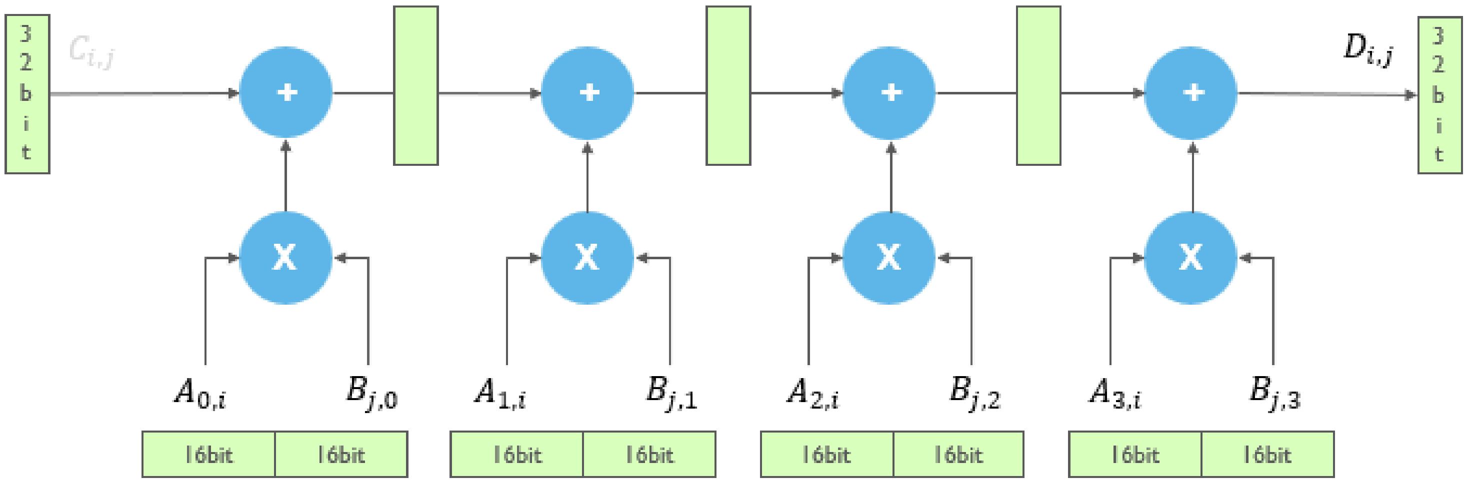
$$D_{0,0} = A_{0,0} * B_{0,0} + A_{0,1} * B_{1,0} + A_{0,2} * B_{2,0} + A_{0,3} * B_{3,0} + C_{0,0}$$

$$D_{1,1} = A_{1,0} * B_{0,1} + A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1} + A_{1,3} * B_{3,1} + C_{1,1}$$

...

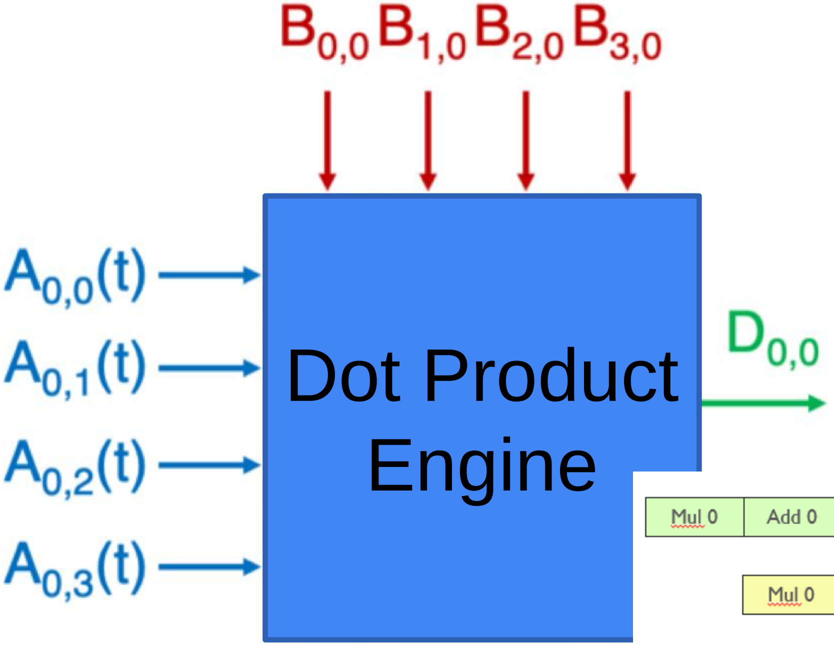
$$D_{3,3} = A_{3,0} * B_{0,3} + A_{3,1} * B_{1,3} + A_{3,2} * B_{2,3} + A_{3,3} * B_{3,3} + C_{3,3}$$

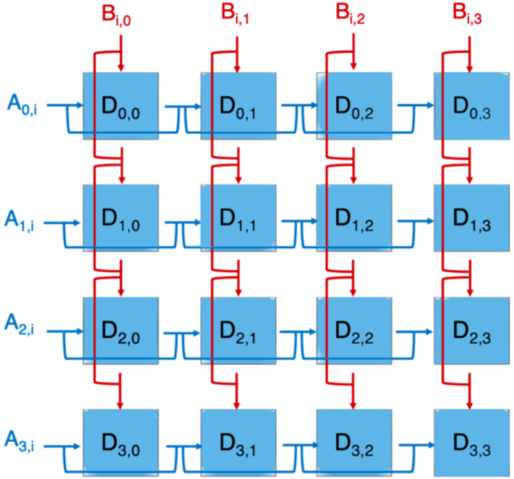
Tensor Core Instruction Pipelining

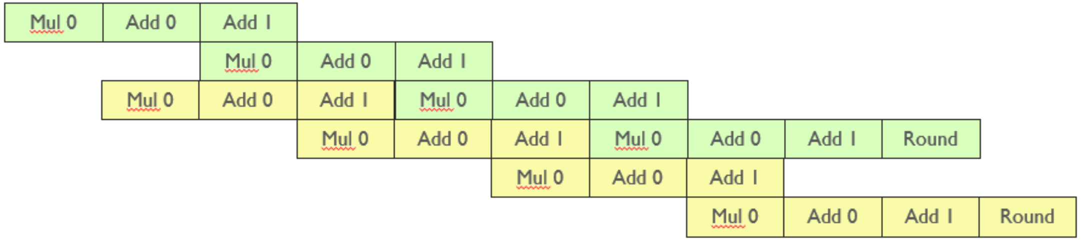


$$D_{0,0} = A_{0,0} * B_{0,0} + A_{0,1} * B_{1,0} + A_{0,2} * B_{2,0} + A_{0,3} * B_{3,0} + C_{0,0}$$

Tensor Core Instruction Pipelining

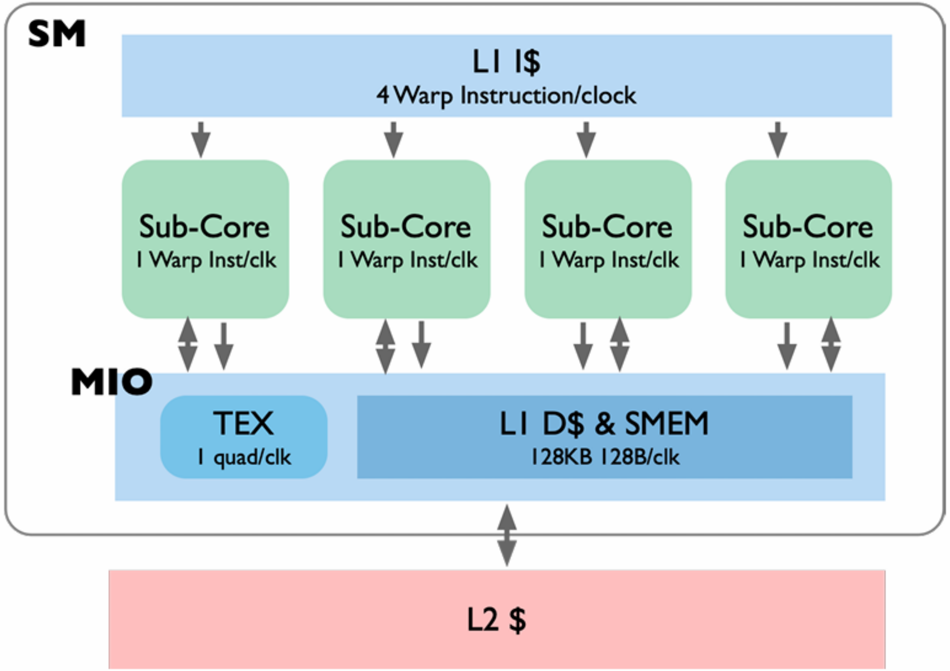






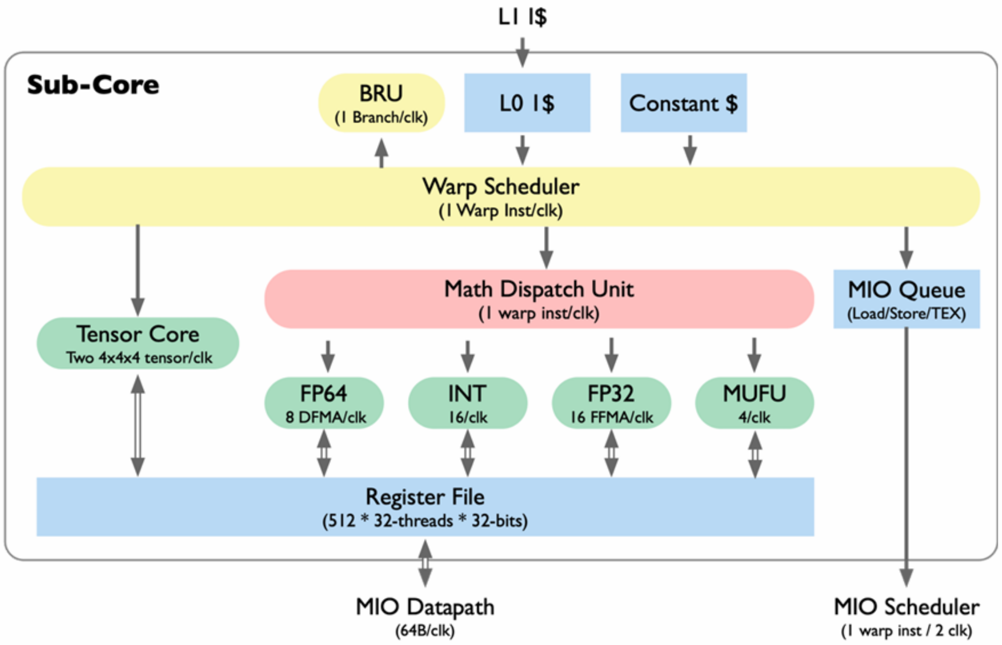
Tensor Core (Volta)

- Sub-Core
  - Include Tensor Core + FP64 + FP32 + INT8 + SPU
- Tensor Core
  - Each SM Sub-Core has two 4 x 4 x 4 Tensor Core
  - Warp scheduler issues GEMM instruction to Tensor Core



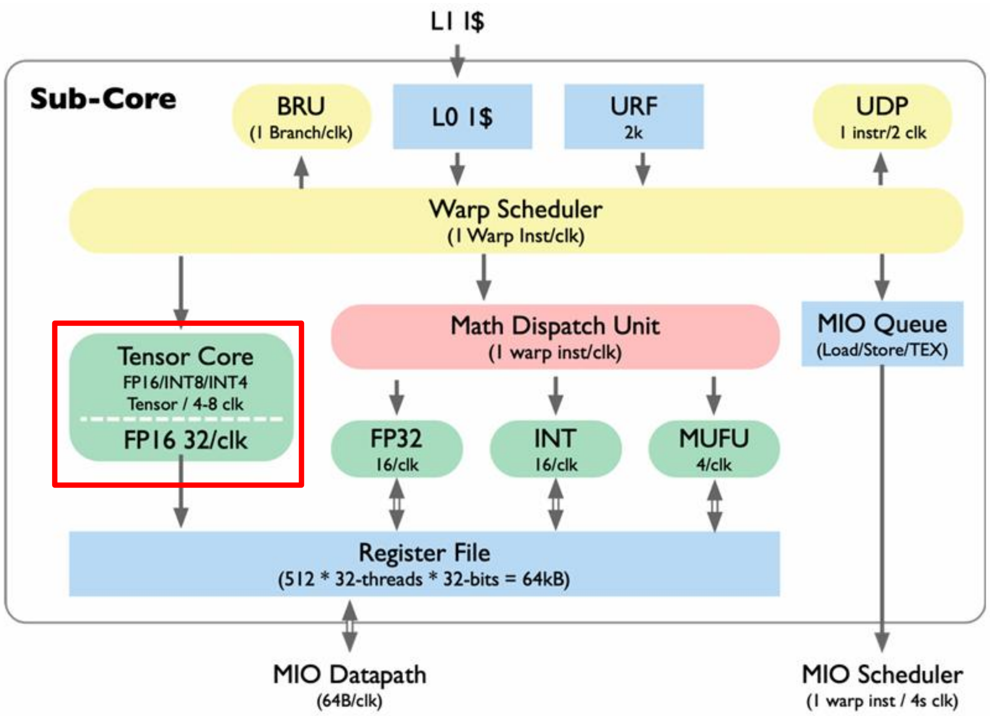
Tensor Core (Volta)

- Tensor Core
  - Executes 4 x 4 x 4 GEMM multiple times
  - Write results back to register files
  - Two 4 x 4 x 4 tensor cores in a sub-core
- Math Dispatch Unit
  - Keeps 2 + Datapaths Busy



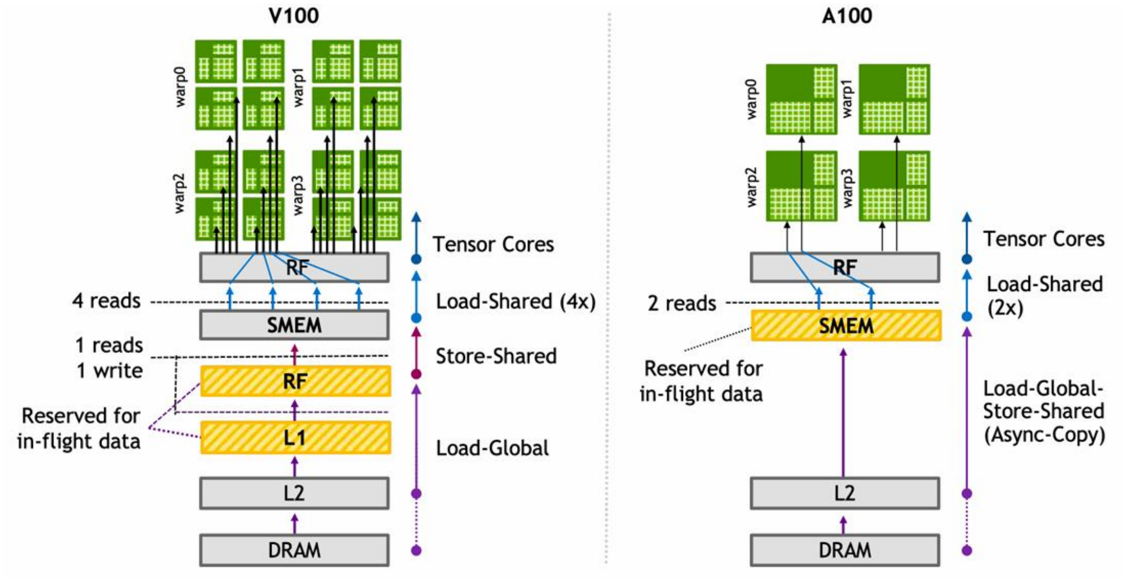
Tensor Core (Turing)

- Tensor Core
  - Add INT8/4 support
  - FP16 fast path
  - Complete one multi-threading GEMM in 4-8 cycles



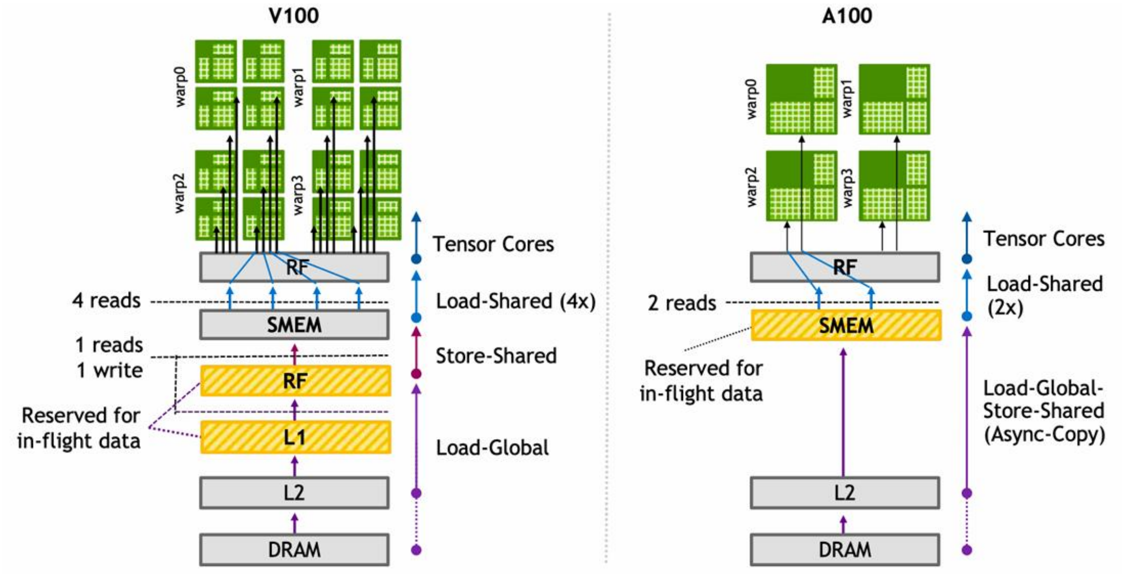
Tensor Core (Ampere)

- Before Ampere GPU, if we want to use shared MEM
  - GPU needs to first pass data from the global memory to REGs and then write data from REGs to shared memory



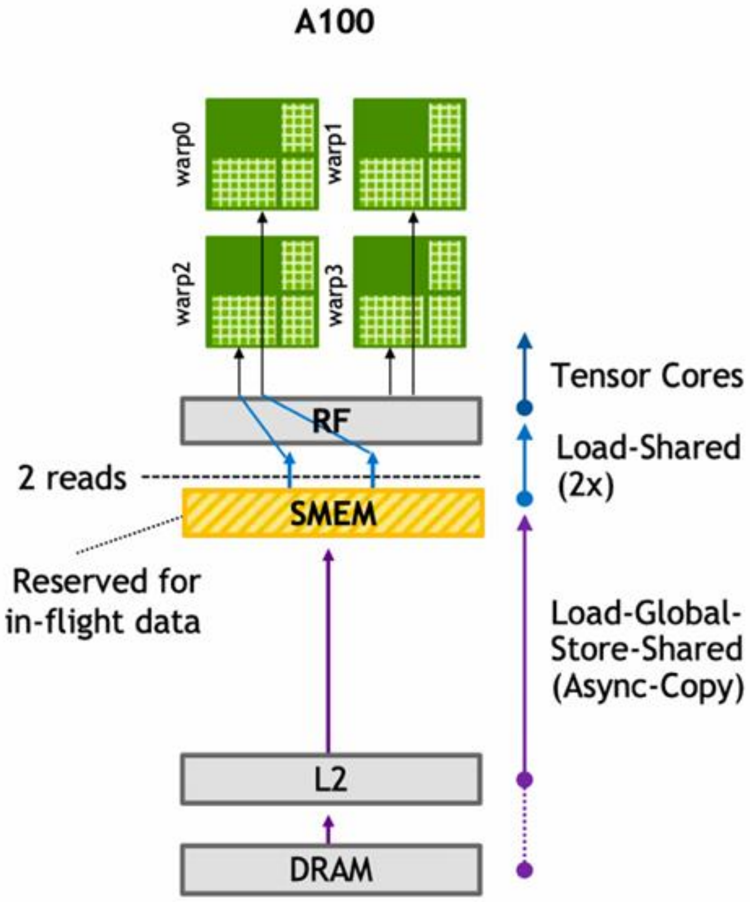
Tensor Core (Ampere)

- Add LDGST SASS (Load Global Store Shared) Inst.
    - Don’t require to first pass data from global memory to register before using the shared memory

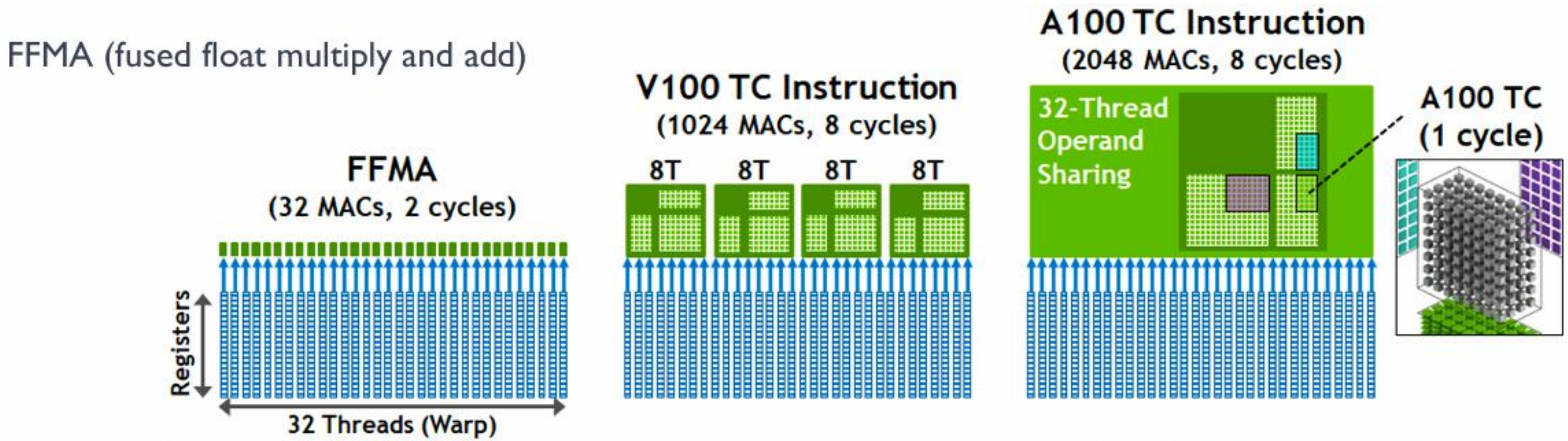


Tensor Core (Ampere)

- In Volta GPU
  - A tensor core only has 8 threads
- Operand Sharing
  - Data shares across 32 threads in a warp
  - <u>Reduce the data movement across threads in a warp</u>



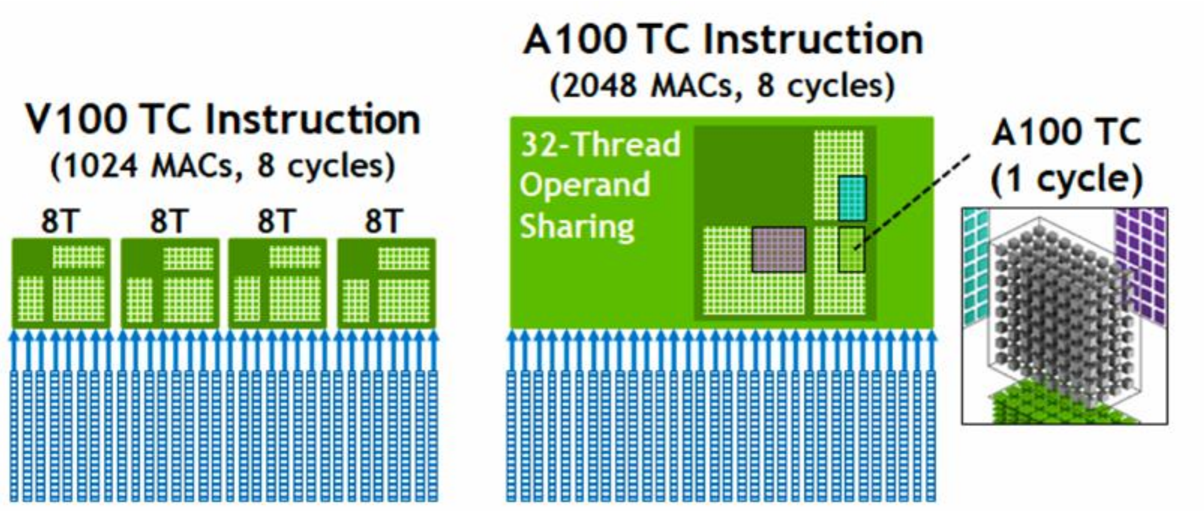
Tensor Core (Ampere)



16x16x16 matrix multiply	FFMA	V100 TC	A100 TC	A100 vs. V100 (improvement)	A100 vs. FFMA (improvement)
Thread sharing	1	8	32	4x	32x
Hardware instructions	128	16	2	8x	64x
Register reads+writes (warp)	512	80	28	2.9x	18x
Cycles	256	32	16	2x	16x

Tensor Core (Ampere)

- Ampere GPU enhances 16 x 8 x 16 WMMA Inst.
  - Reduce the times of register accesses from 80 to 28
  - Reduce the number of FFMA instruction issue from 16 to 2



Tensor Core

- Each tensor core is a programmable compute unit for matrix-multiply-and accumulation (MAC) – inner-product-based
- In Nvidia Volta GPU Tensor Core
  - A tensor core executes 64 MACs with the FP16 format
    - One cycle completes 4 x 4 x 4 matrix multiplication
- In Nvidia A100 GPU Tensor Core
  - A tensor core executes 256 MACs with the FP16 format
    - One cycle completes 8 x 4 x 8 matrix multiplication

$$A \times B = C,\ where\ A \in R^{32\times32}, B \in R^{32\times32}$$

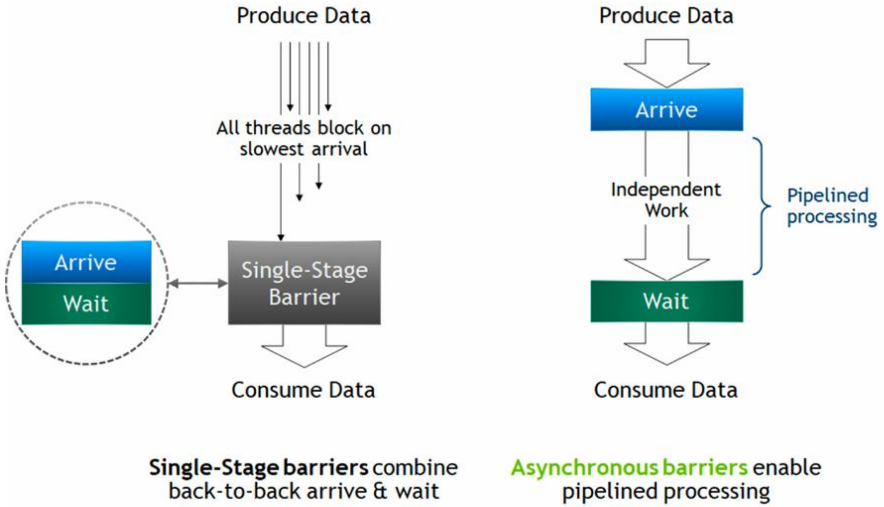
Matrix Multiply on GPU w/ Tensor Core

- How many A100 GPU tensor cores do we need to complete a 32 x 32 matrix multiplication in one cycle?
  - With tensor core, we can perform a 4 x 4 matrix multiplication in one cycle
  - To do a 32 x 32 matrix multiply, we need to do 8 x 8 = 64 Tensor Core operations
- Assume a GPU has 8 SMs, each SM has 8 tensor cores
    - 200 cycles (Global Memory) + 1 x 34 cycles (shared memory) + 1 x 1 cycle (tensor core) = 235 cycles
    - w/o tensor core: 504 cycles

Tensor Core (Hopper)

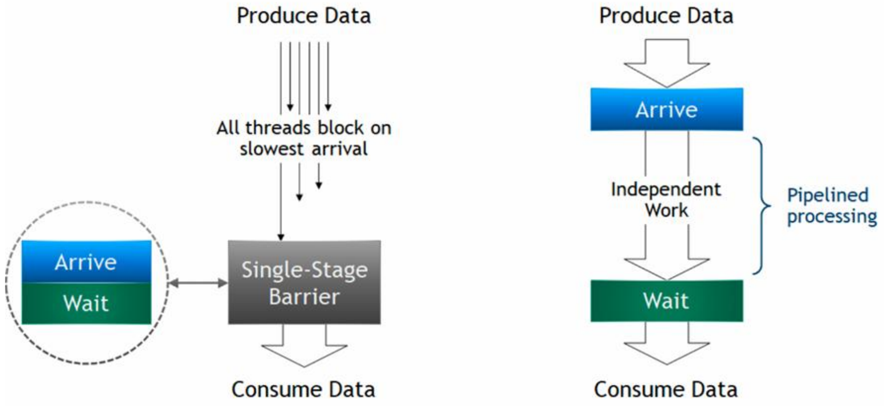
Tensor Core (Hopper)

- Before Hopper GPU Tensor Core
  - Load data from global memory to registers
  - The warp scheduler activates Tensor Core to do GEMM
  - Write back outcomes of Tensor Core to registers



Tensor Core (Hopper)

- Ampere GPU Tensor Core
    - Each thread has individual matrix tile address
    - Pipelining the data movement among global memory <-> shared memory <-> registers

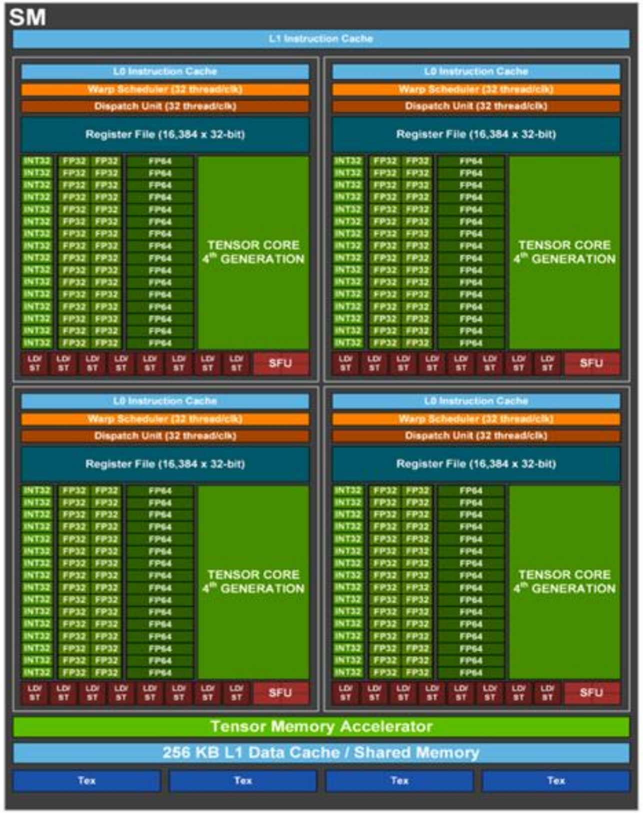


Single-Stage barriers combine back-to-back arrive & wait

Asynchronous barriers enable pipelined processing

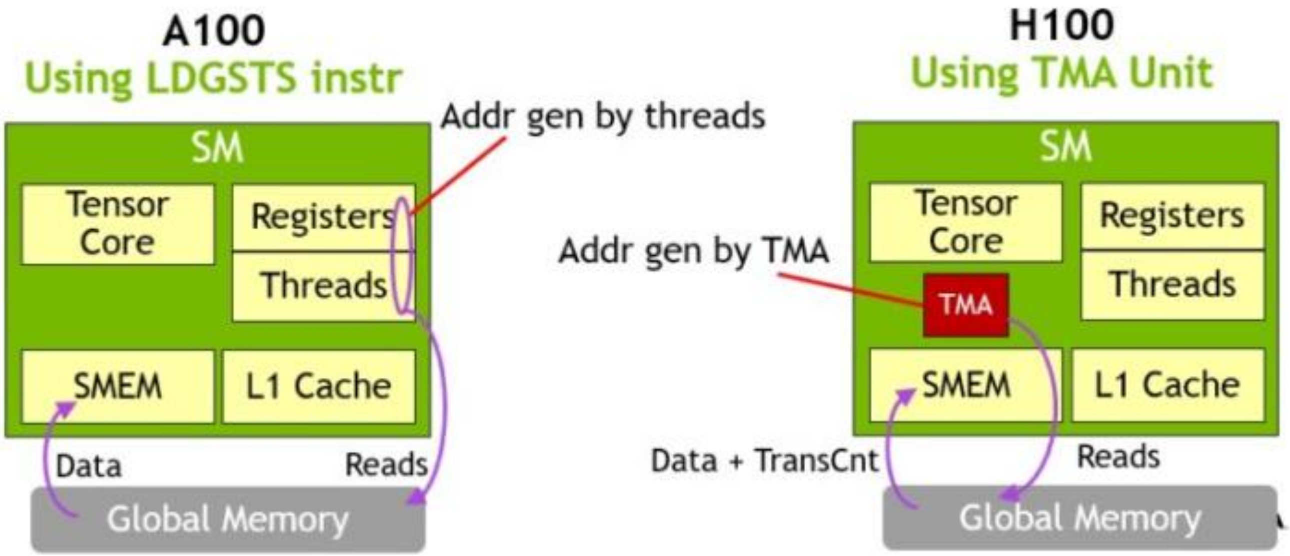
Tensor Core (Hopper)

- TMA (Tensor Memory Accelerator)
  - Asynchronous pass data from global memory to shared MEM
  - Single thread schedule model
    - <u>Not all threads in a warp get involve in data loading from global MEM to shared MEM</u>



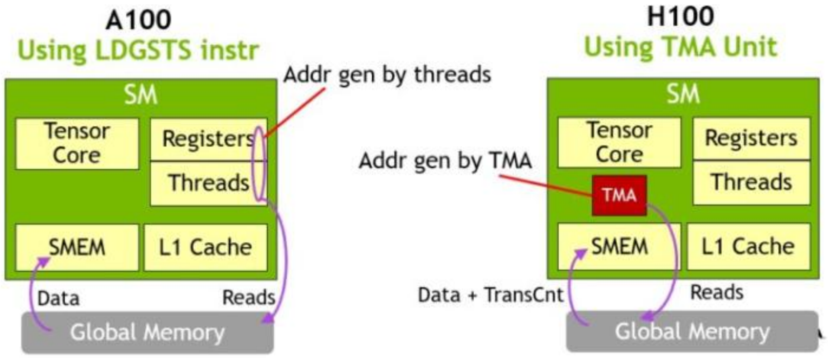
Tensor Memory Accelerator (TMA)

- **Asynchronous barrier in Nvidia A100 GPU**
  - A set of threads are producing data that they all consume after a barrier



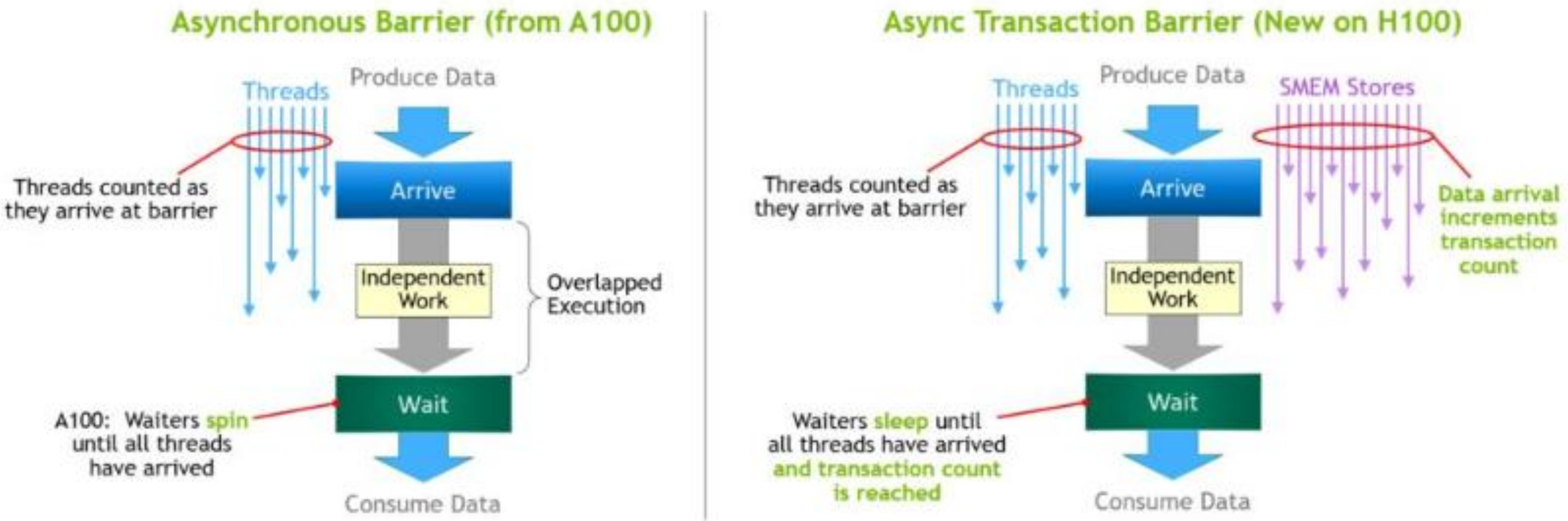
Tensor Memory Accelerator (TMA)

- **Asynchronous barrier in Nvidia A100 GPU**
  - First, threads signal Arrive when they are done producing their portion of the shared data
  - Finally, the threads need the data produced by all the other threads
    - They do a Wait, which blocks them until every thread has signaled Arrive



Tensor Memory Accelerator (TMA)

- **Asynchronous transaction barrier in Nvidia H100 GPU**
- **Hopper (TMA):** A single thread per warp issues TMA operation



Tensor Core Programming

- C++ API performs "warp-level matrix multiply and accumulate (WMMA)"
  - Perform 16 x 16 x 16 matrix multiply on the Tensor Core
- CUDA WMMA APIs

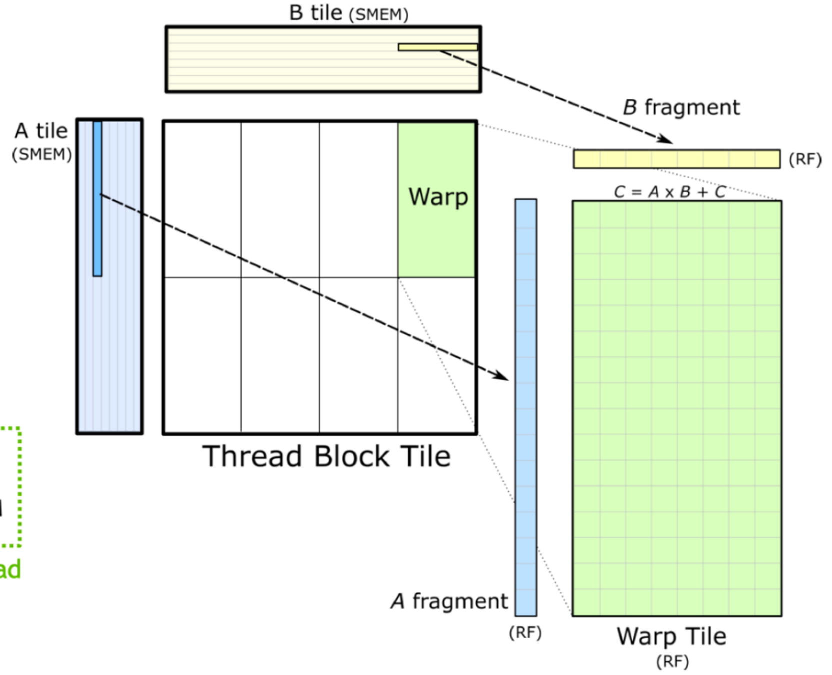
```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const fragment<...> &c, bool satf=false);
```

Warp-level Tensor Core Programming

- Load fragment matrix A and B from shared MEM (SMEM) to registers
- The data in the SMEM is stored in k dimension

```
for (int k = 0; k < Ktile; k += warp_k)
{
    .. load A tile from SMEM into registers
    .. load B tile from SMEM into registers

    for (int tm = 0; tm < warp_m; tm += thread_m)
        for (int tn = 0; tn < warp_n; tn += thread_n)

            for (int tk = 0; tk < warp_k; tk += thread_k)

                .. compute thread_m by thread_n by thread_k GEMM

}
```

by each CUDA thread

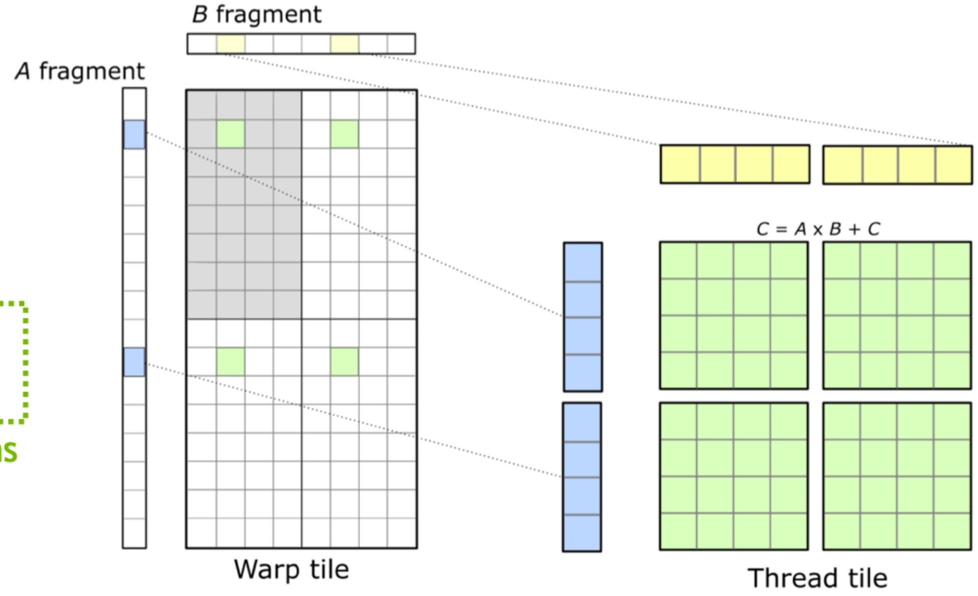


Warp-level Tensor Core Programming

- Matrix A, B, C is stored in REGs

```
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)
        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

Fused multiply-accumulate instructions

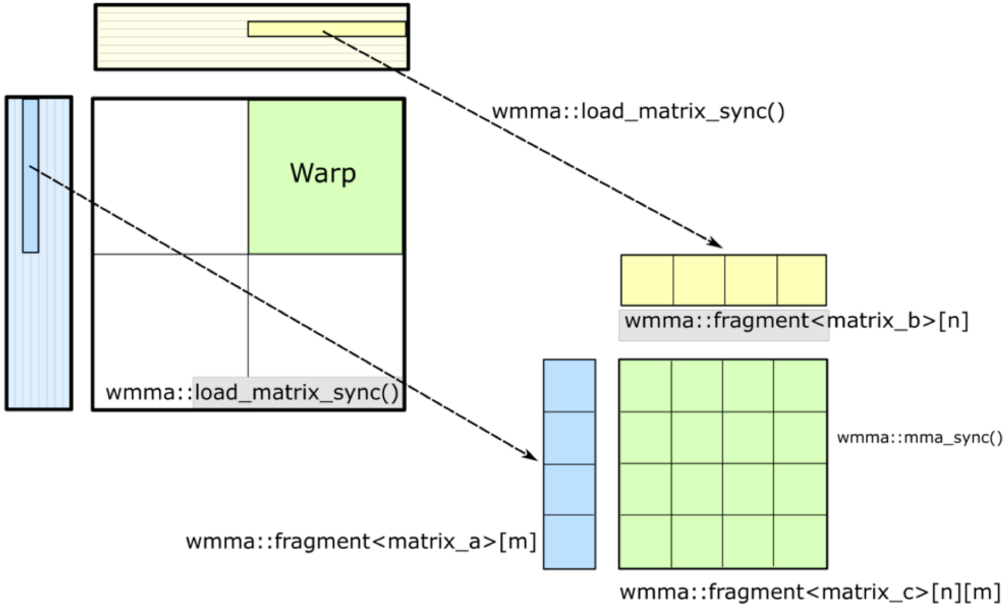


Warp-level Tensor Core Programming

- Matrix A, B, C is stored in REGs

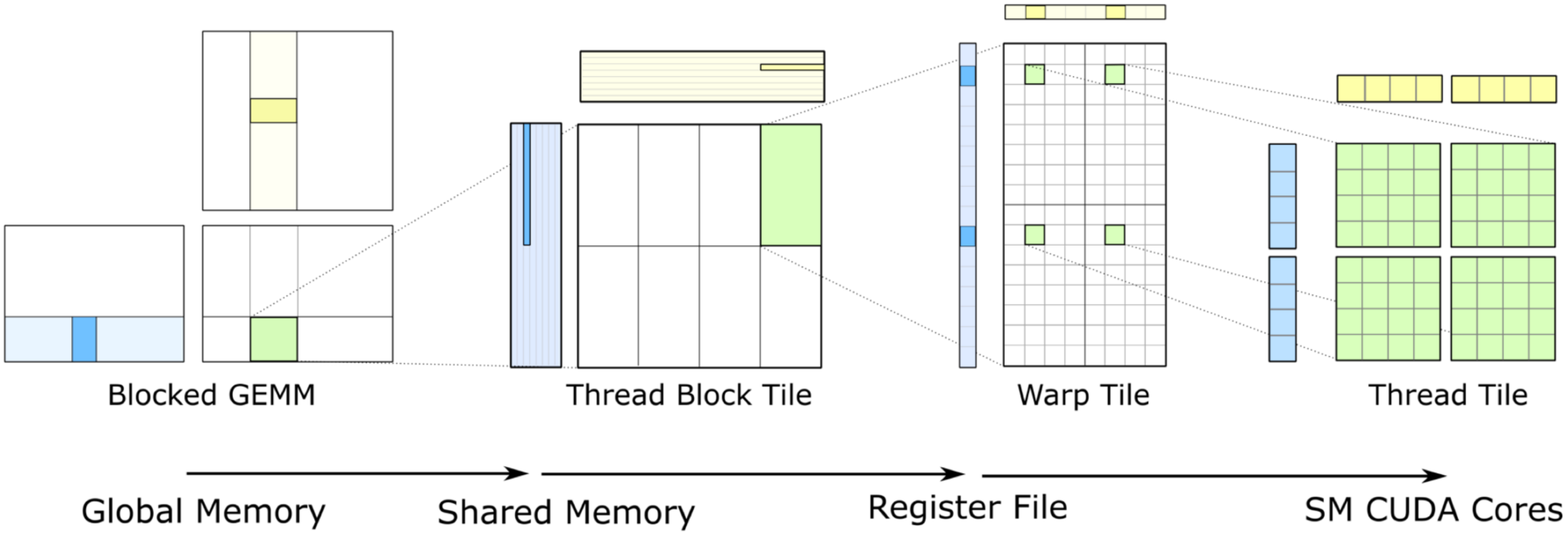
```
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)

        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

Fused multiply-accumulate instructions



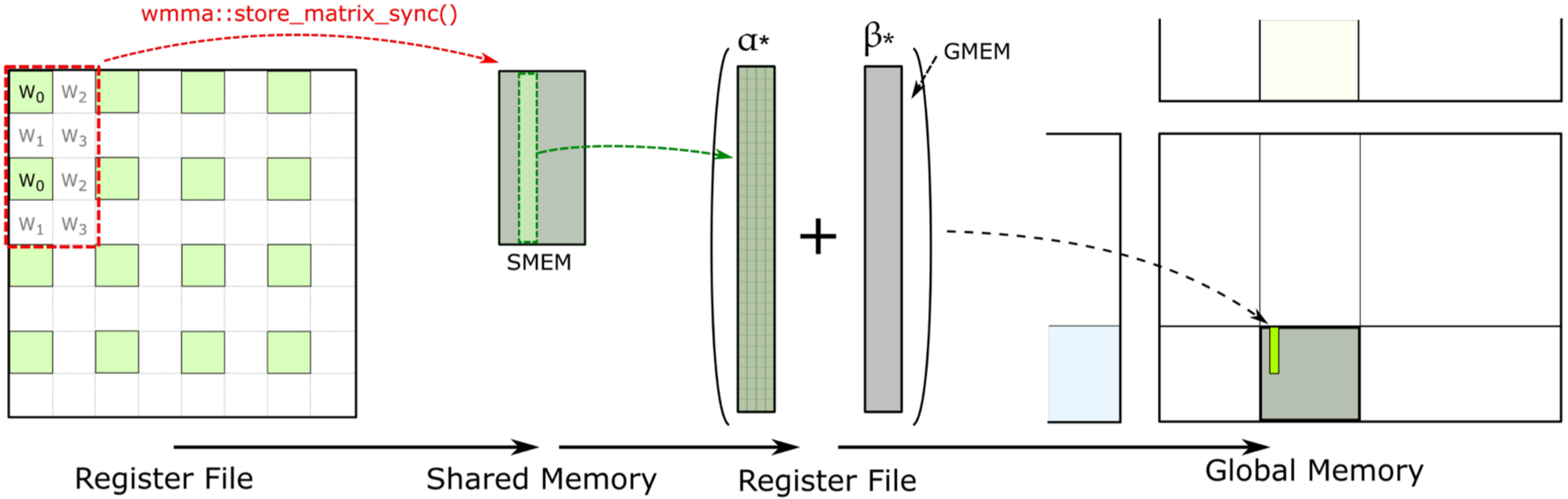
Warp-level Tensor Core Programming

- Data reuse in each type of memory



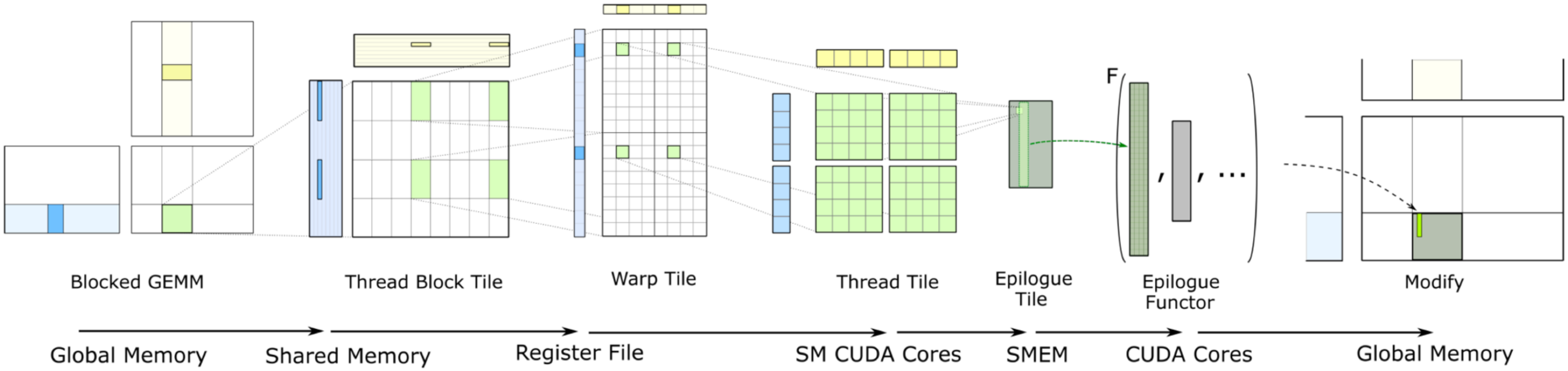
Warp-level Tensor Core Programming

- Write back WMMA Results
  - GEMM: C = α AB + β C



Warp-level Tensor Core Programming

- GPU data flow in matrix multiplication



Sparse Tensor Core

- Improve tensor core utilization in sparse MMA
- Sparse MMA is shown on model compression
- Data encoding + tensor core mapping
- Does this work on graph workloads with dynamic sparsity ?

0	NZ_0	0	0	0	0	NZ_1	0
0	0	NZ_2	0	0	0	0	0
0	0	0	NZ_3	0	NZ_4	0	0
NZ_5	0	0	0	NZ_6	0	0	0

Original Weight

NZ_0	NZ_1
NZ_2	0
NZ_3	NZ_4
NZ_5	NZ_6

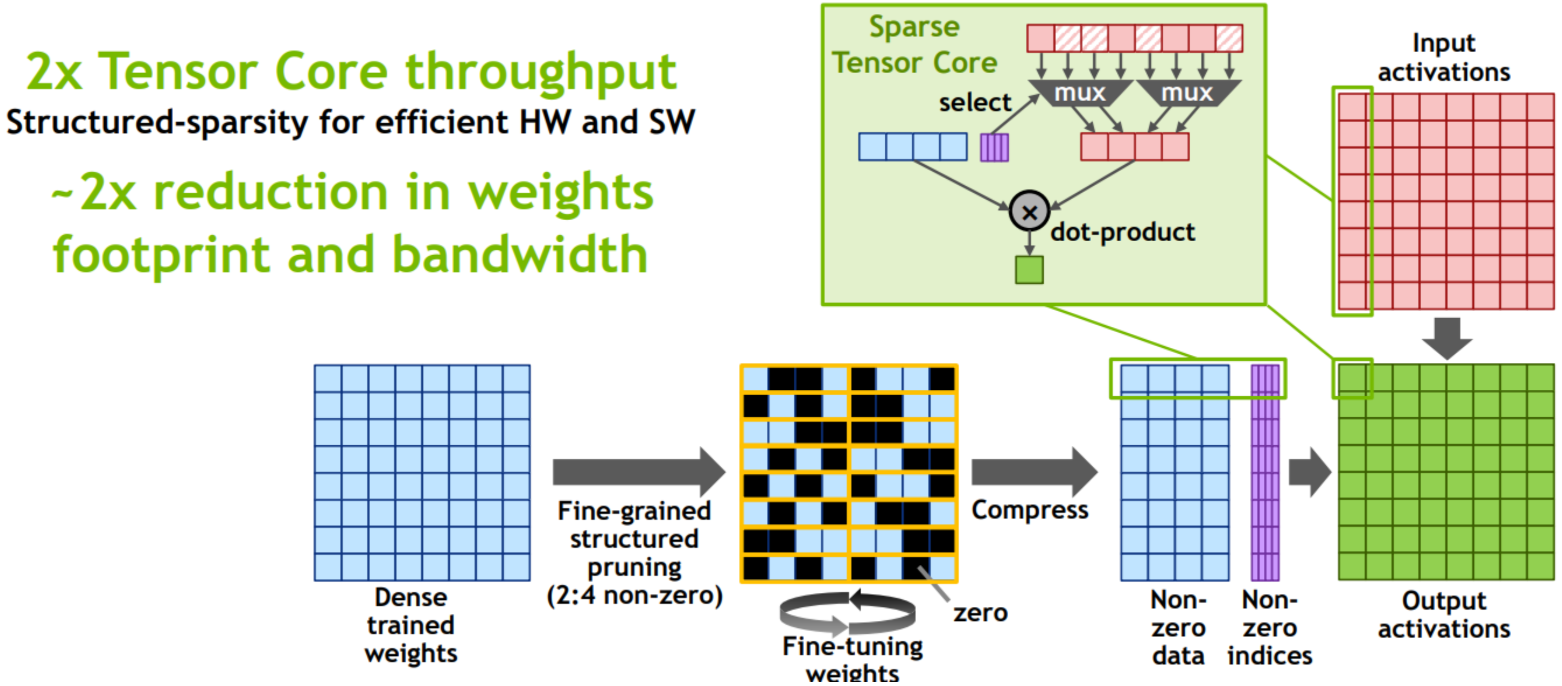
1	6
2	2
3	5
0	4

Encoded offset

Compressed Weight

Zhu et.al., MICRO 2019

Sparse Tensor Core in Nvidia A100 GPU



https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21730-inside-the-nvidia-ampere-architecture.pdf

Takeaway Questions

- How does tensor core accelerate the matrix computation ?
  - (A) Specialized dot-product engine
  - (B) Increase the frequency of tensor cores
  - (C) Reduce the data movement
- How to reduce global memory transactions of tensor core ?
  - (A) Use image to column (Im2col)
  - (B) Lower the data precision (using int8)
  - (C) Increase the number of registers