



# 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



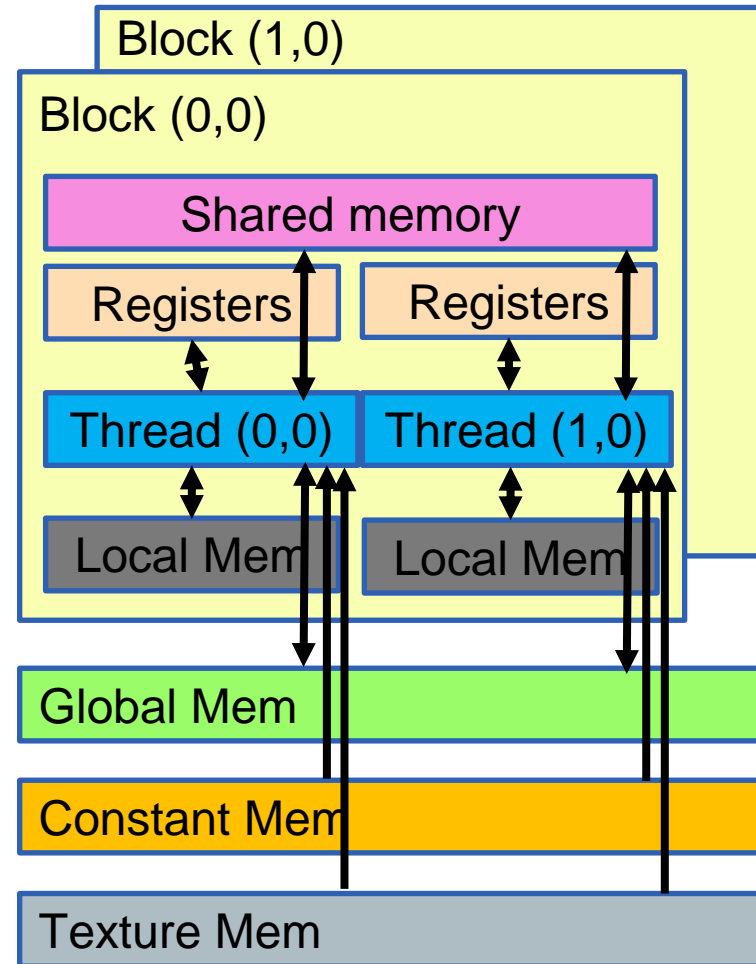
# Outline

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



# GPU Memory Spaces

- **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





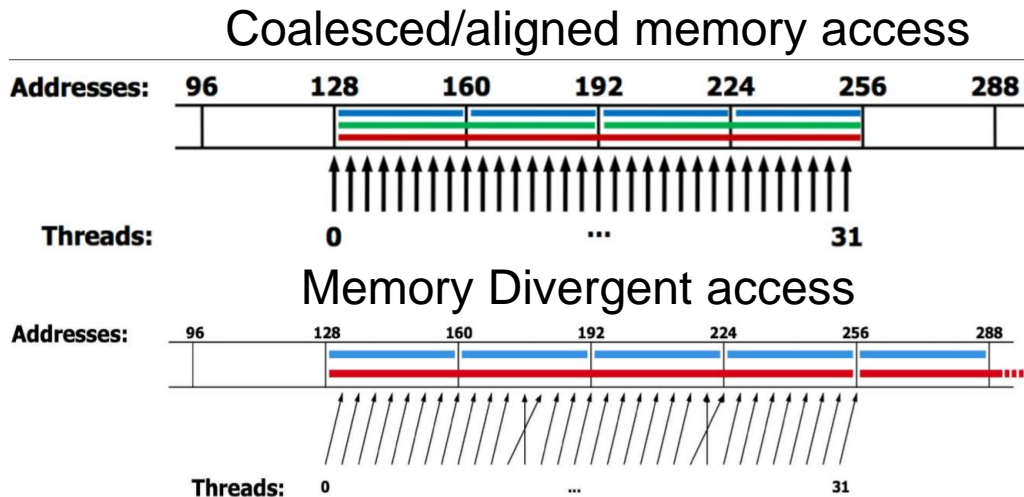
# Global Memory

- 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

Built-in align variable:  
`__align__(int mem_byte)`

```
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  $offset > 1$  ?





# 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 :
- Which one is coalescing access pattern ?
  - Pattern B is coalescing access pattern

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

Pattern A

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

Time →

Pattern B

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

Time →





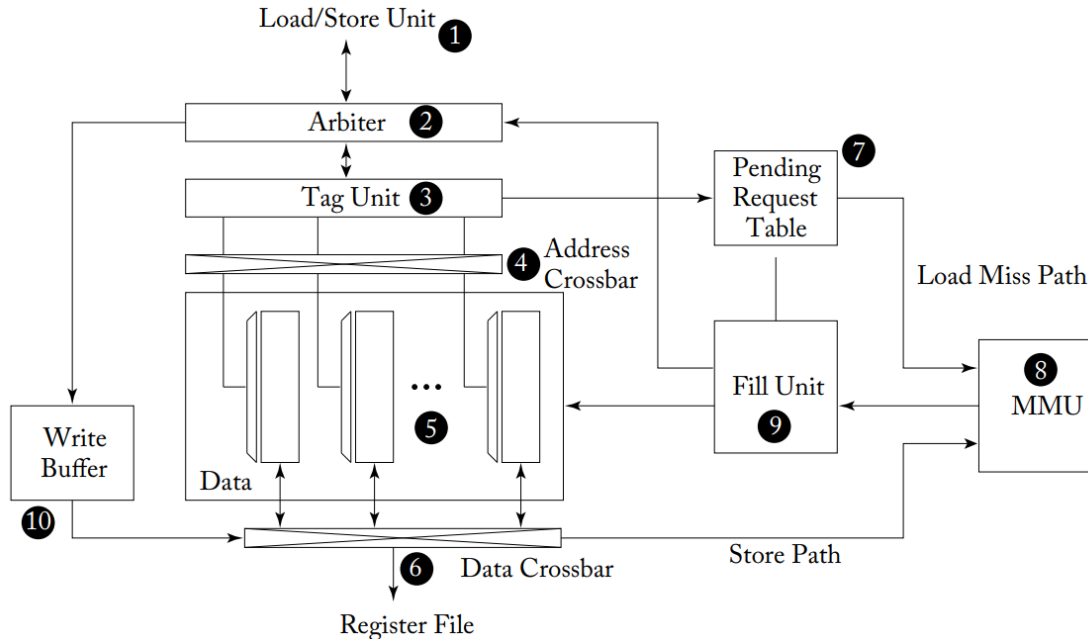
# Local Memory

- 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

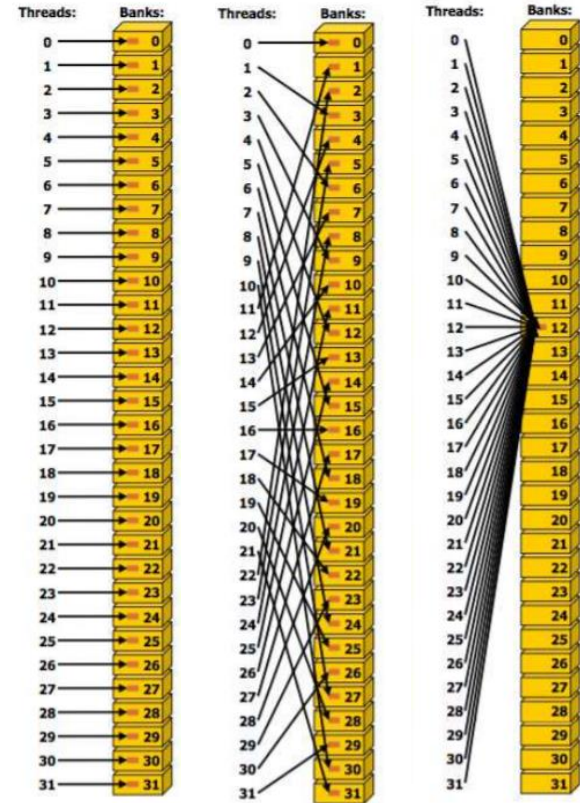




# Shared Memory

Which one is bank conflict ?

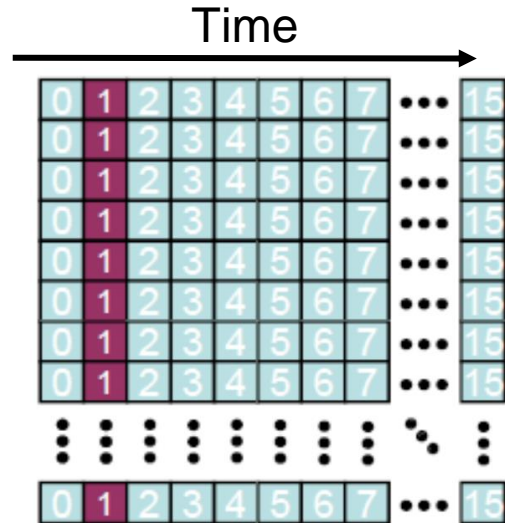
- 32 banks organized as 32-bit successive words
- Threads share data in the same thread block
- Programmer-managed on-chip cache
- Bank conflict
  - Two or more threads access words within the same bank
  - Serialized memory access (low memory bandwidth)
- Which one is bank conflict ?
  - `float i_data = 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]`



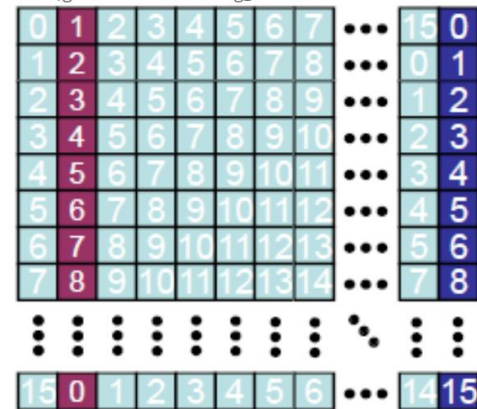


# How to Resolve Bank Conflict ?

- 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)





# 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]`



Memory padding

	Bank 0					Bank 3				
tid 0 →	0	1	2	3	4					
tid 1 →	0	1	2	3	4					
	0	1	2	3	4					
	0	1	2	3	4					
tid 4 →	0	1	2	3	4					

0	1	2	3	4
	0	1	2	3
4		0	1	2
3	4		0	1
2	3	4		0
1	2	3	4	



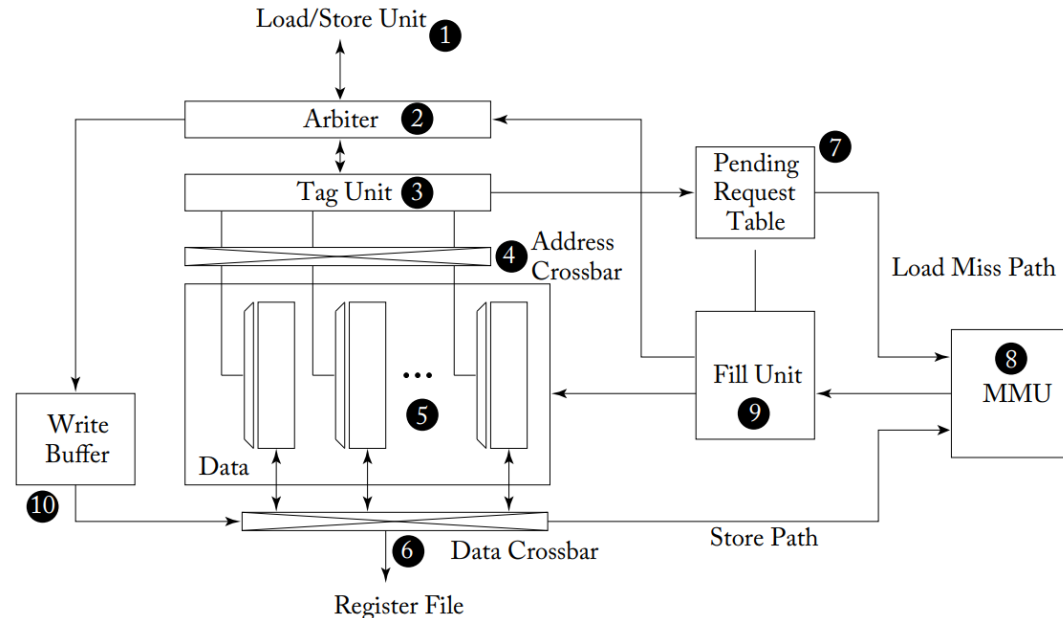
# Shared memory access

- **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**

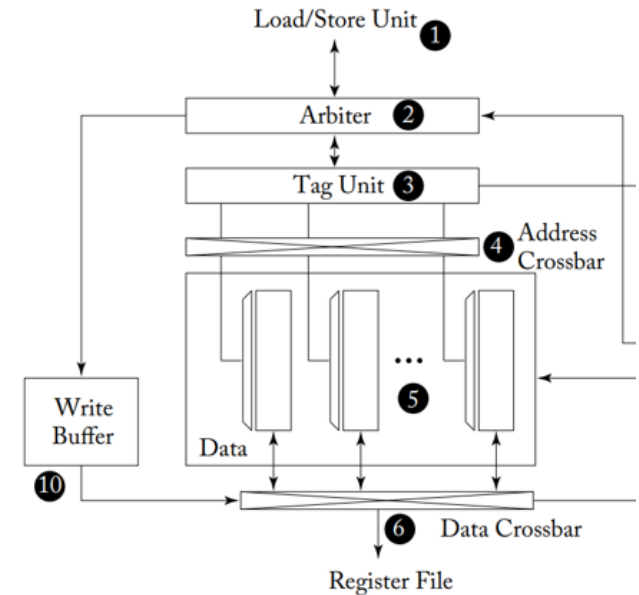
- Bypass tag lookup in the tag unit, since shared memory is direct mapped





# Shared memory access

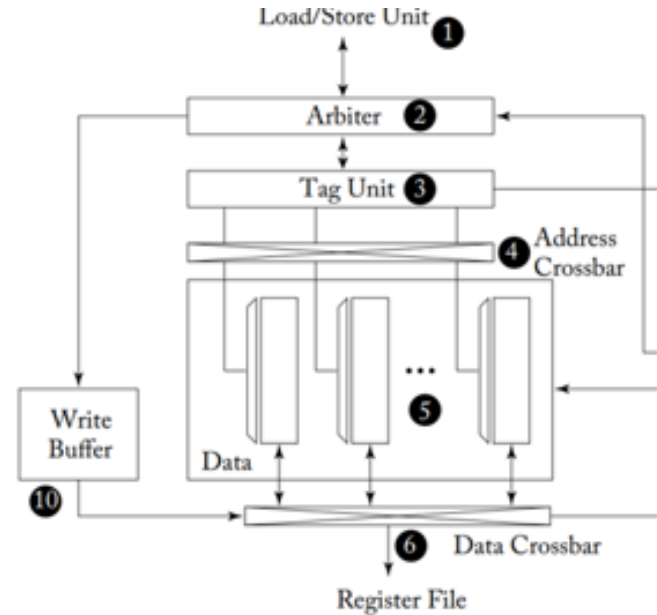
- **In the absence of bank conflict**
  - The latency of the direct mapped memory lookup is constant (single-cycle)
  - The tag unit determines which bank each thread's request maps to
  - The address cross bar distributes address to the individual banks within the data array
  - Each bank inside the 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

- Access to global memory is restricted to a single cache block per cycle -> help to reduce tag storage overhead
- The L1 cache block size is 128 bytes, is further divided into four 32-byte sectors
- A single access of GDDR5 is 32-byte
- Each 128-byte cache block is composed of 32-bit entries at the same row in each of the 32 banks

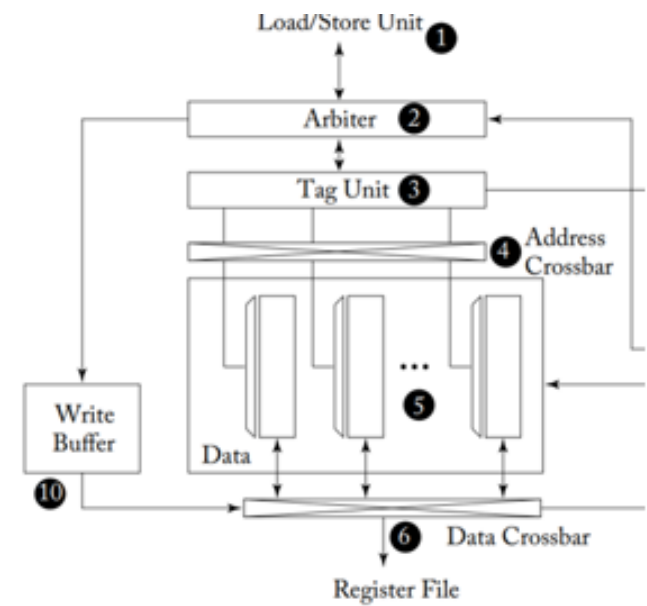






# L1 Data Cache Read

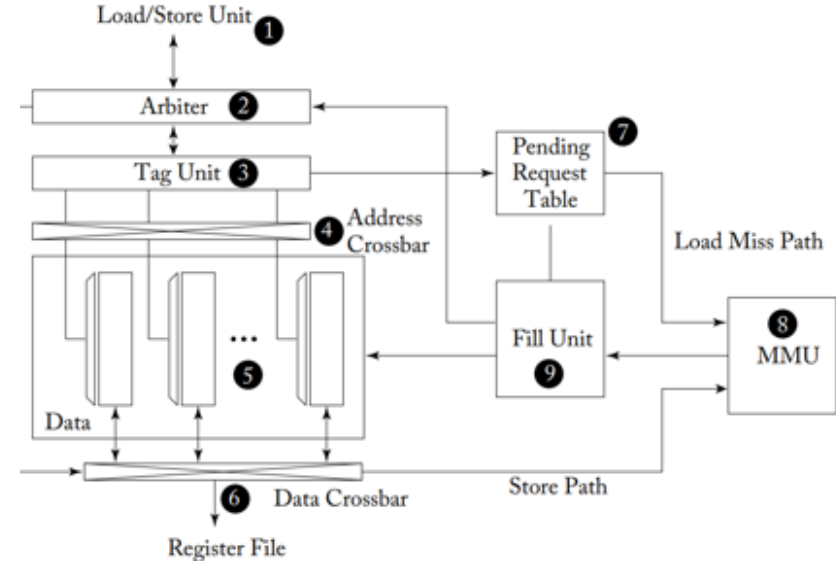
- 1) The LD/ST unit
  - Computes memory addresses
- 2) The arbiter
  - Requests the instruction pipeline schedule a writeback to the register file if enough resources are available
- 3) The tag unit
  - Check whether the access leads to a cache hit or a miss
- 4) Access the appropriate row of the data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



# L2 Cache Bank

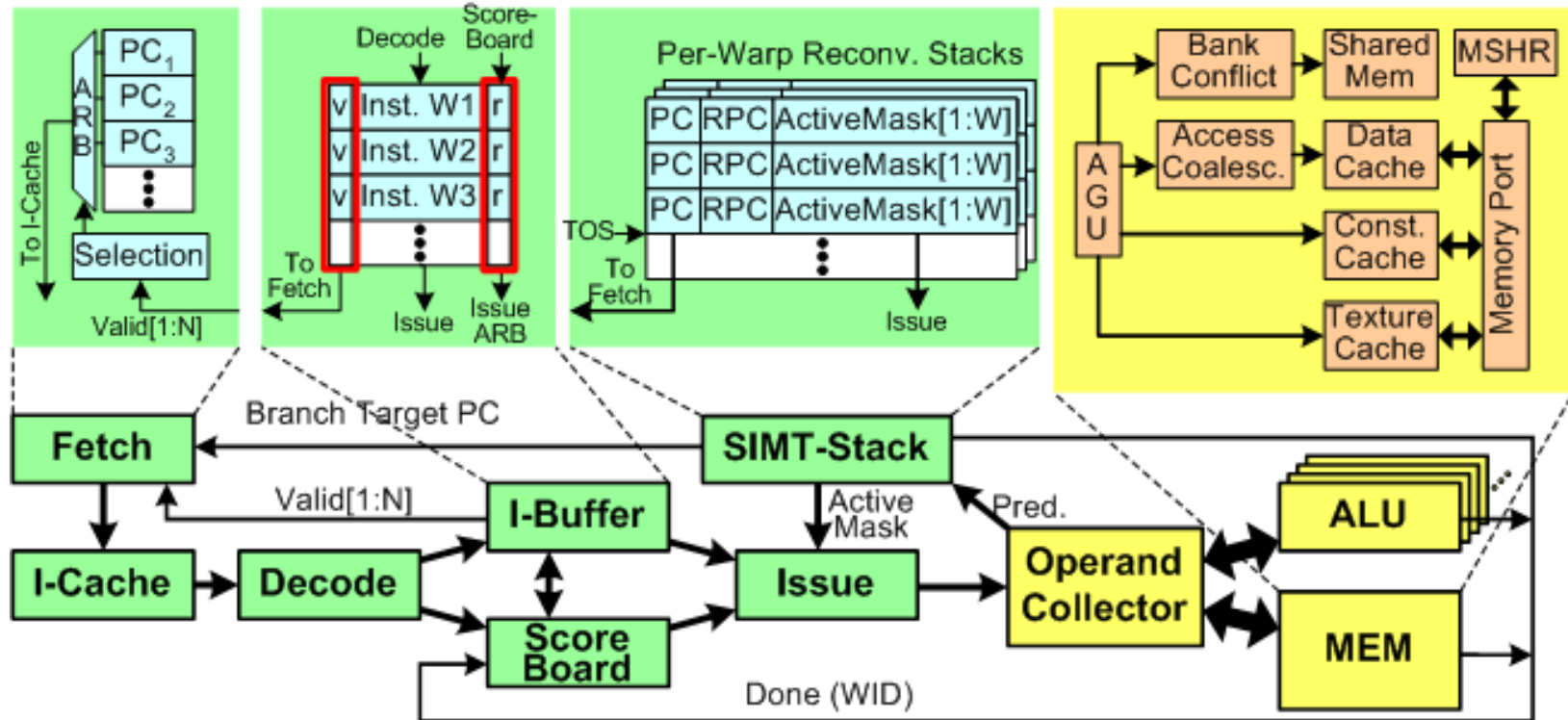
- A unified last level cache shared by all SIMT cores
- L1 cache request cannot span across two L2 cache lines

	<b>Local Memory</b>	<b>Global Memory</b>
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



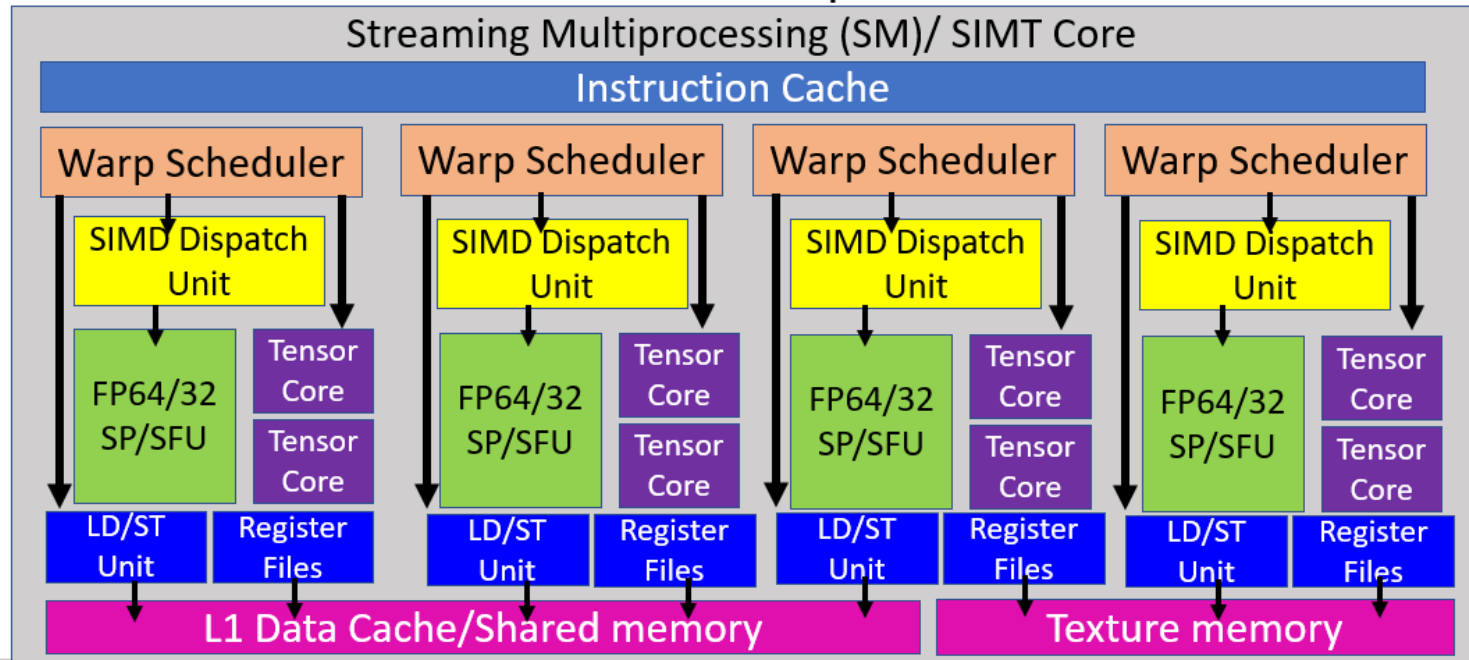
# GPU Micro-architecture





# Problems of DNNs on GPU

- DNNs require a large number of matrix computations
- Tensor core tailors for matrix computation on GPUs



Zhu et.al.,  
MICRO  
2019



# Inner Product

- **Inner product**

- Each inner product computes a single element of the product matrix C
- High memory transaction in  $B[k][n]$

- $B[0][j]$  and  $B[1][j]$  may stay in a cache line

```
for(int m = 0; m < M; m++) {  
    for(int n = 0; n < N; n++) {  
        for(int k = 0; k < K; k++) {  
            C[m][n] += A[m][k]*B[k][n];  
        }  
    }  
}
```





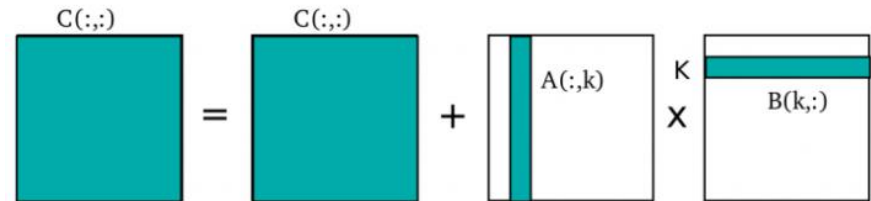


# Outer Product

- **Outer product**

- Raise k to the outer-most for loop
- Multiply (m, 1) and (1, n) matrix
- Accumulate k (m, n) matrix
- Good to do blocked matrix multiplication. How ?

```
for(int k = 0; k < K; k++) {  
    for(int m = 0; m < M; m++) {  
        for(int n = 0; n < N; n++) {  
            C[m][n] += A[m][k]*B[k][n];  
        }  
    }  
}
```





# Blocked Outer Product

```
% iterate through blocks
```

```
for k = 1: K/K0
```

```
  for l = 1: l/I0
```

```
    Ablock = &A(i*I0, k*K0)
```

```
    for j = 1: J/J0
```

```
      Cblock = &C(i*I0, j*J0)
```

```
      Bblock = &B(k*K0, j*J0)
```

```
      do_block(Ablock, Bblock, Cblock)
```

```
void do_block(Ablock, Bblock, Cblock){
```

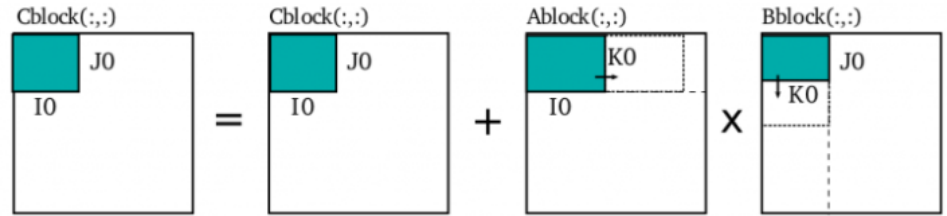
```
  for k0 = 1:K0
```

```
    for i0 = 1:I0
```

```
      for j0 = 1:J0
```

```
        Cblock(i0, j0) = Cblock(i0, j0)+ Ablock(i0, k0) *
```

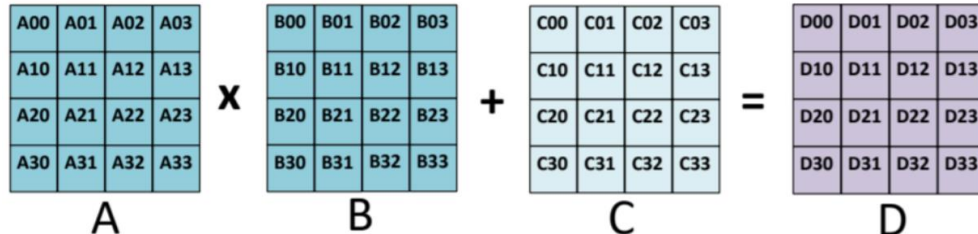
```
        Bblock(k0, j0) }
```





# Tensor Core

- Each tensor core is a programmable compute unit for matrix-multiply-and accumulation (MAC) – inner-product-based
- Each tensor can complete a single 4 x 4 MAC each clock cycle
  - Why does tensor core use 4 x 4 matrix ?
- The tensor core has two modes of operation:
  - **FP16 mode:** reads three 4 x 4 16-bit floating-point matrices as source operands
  - **Mixed-precision:** reads two 4 x 4 16-bit floating point matrices along with a third 4 x 4 32-bit floating-point accumulation matrix





# Warp Matrix Function (WMMA) API

- C++ API performs “warp-level matrix multiply and accumulate (WMMA)” on tensor cores
- CUDA 9.0 supports 16 x 16 x 16 tile size, while later versions have more flexibility
- Each tile is divided into fragments
  - A fragment is a set of tile elements that are mapped to registers of a thread
  - Input matrices are distributed across different threads
  - Each thread contains only a portion of a tile
- CUDA WMMA APIs
  - `Load_matrix_sync`, `store_matrix_sync`, `mma_sync`



# Tensor Core PTX instructions

```
wmma.load.a.sync.layout.shape.type          ra, [pa] {stride};  
wmma.load.b.sync.layout.shape.type          rb, [pb] {stride};  
wmma.load.c.sync.layout.shape.type          rc,  
[pc] {stride};  
wmma.mma.sync.alayout.blayout.shape.dtype.ctype  rd, ra, rb, rc;  
wmma.store.d.sync.layout.shape.type          rd, [pd] {stride};
```

- Matrices A, B, and C are stored in registers ra, rb, and rc
- The “layout” specifies the operand matrix stored in memory with a row-major or column-major layout
- The “shape” represents the fragment size of operand matrices
- The type indicates the precision of operand matrices
- The “stride” operand indicates the beginning of each row



# WMMA Operations on Tensor Core

- Given A, B, C, and D are 16 x 16 matrices
- A warp computes a matrix multiply and accumulate  
 $D = A \times B + C$
- 32 threads in a warp are divided into **“8” threadgroups**
- Each threadgroup consists of 4 threads in a warp

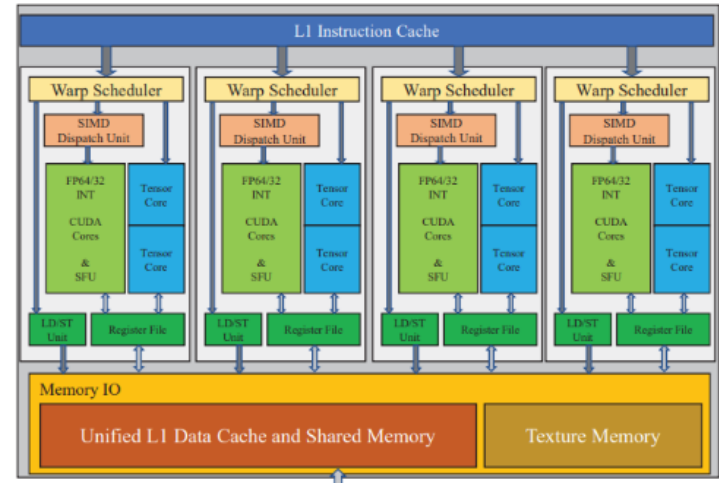


# GPU Tensor Core

- **GPU tensor core**

- Specialized hardware for the MAC operation
- Multiple warps work together to complete the WMMA operation (e.g.  $16 \times 16 \times 16$ )

$$A_{16 \times 16} \times B_{16 \times 16} + C_{16 \times 16}$$





# GPU Tensor Core

- GPU tensor core**

- A WMMA operation breaks into 4 sets of machine-level HMMA (Half-precision MMA) instructions at the compile time (why?)
- Each set of HMMA instructions compute the product of a 4 x 4 tile of A and a 4 x 8 tile of B

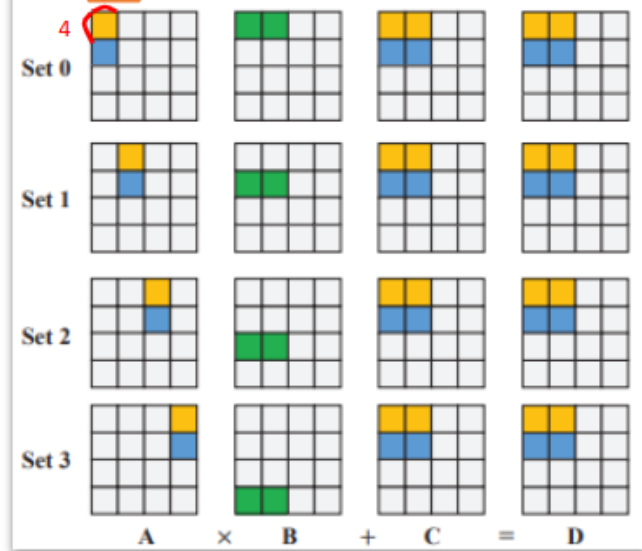
The tiles processed by each set of dense HMMA instructions in Worktuple 0

MMA dimension

$$A_{8 \times 4} \times B_{4 \times 8} + C_{8 \times 8}$$

Each Set of dense HMMA:

HMMA 884, F32.F32.STEP0 RD, RA, RB, RC;  
 HMMA 884, F32.F32.STEP1 RD, RA, RB, RC;  
 HMMA 884, F32.F32.STEP2 RD, RA, RB, RC;  
 HMMA 884, F32.F32.STEP3 RD, RA, RB, RC;



Threadgroup 0 Threadgroup 4 Shared  
in Worktuple 0





# GPU Tensor Core

- GPU tensor core

- 2 octets in a tensor core
- Inside an octet
  - 8 DPs (Dot Product units)
  - Each DP can compute 4-dim vector dot product per cycle
  - Operand buffers A, B, C

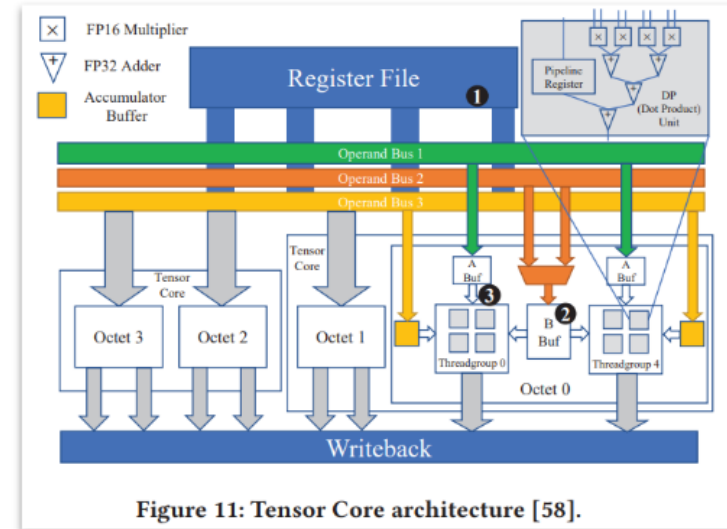
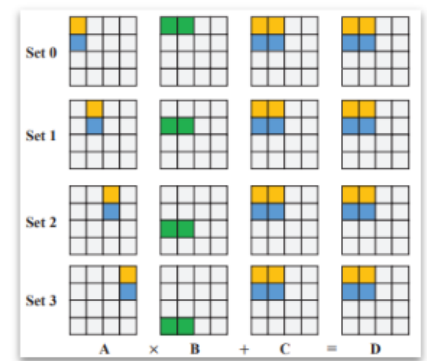


Figure 11: Tensor Core architecture [58].

Tensor Core architecture



# GPU Tensor Core

- GPU tensor core

- Mapping

- A worktuple (2 threadgroups) is mapped to one octet
    - Each threadgroup takes 4 DPs

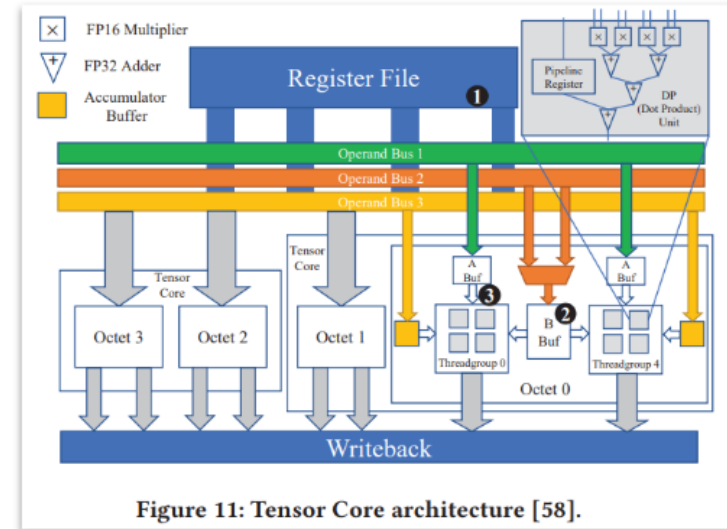
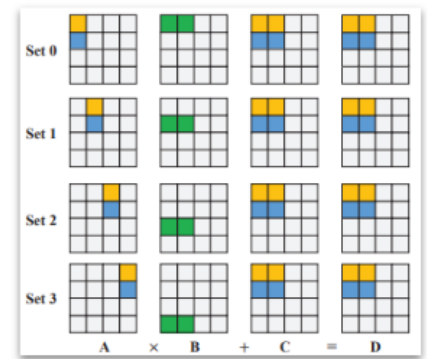


Figure 11: Tensor Core architecture [58].

Tensor Core architecture

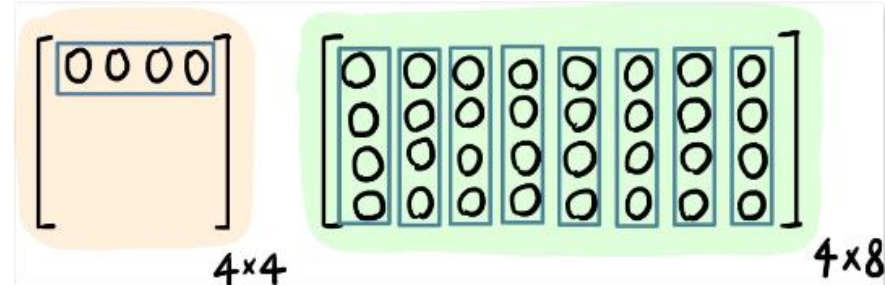


# GPU Tensor Core

- **GPU tensor core**

- Cycle calculation

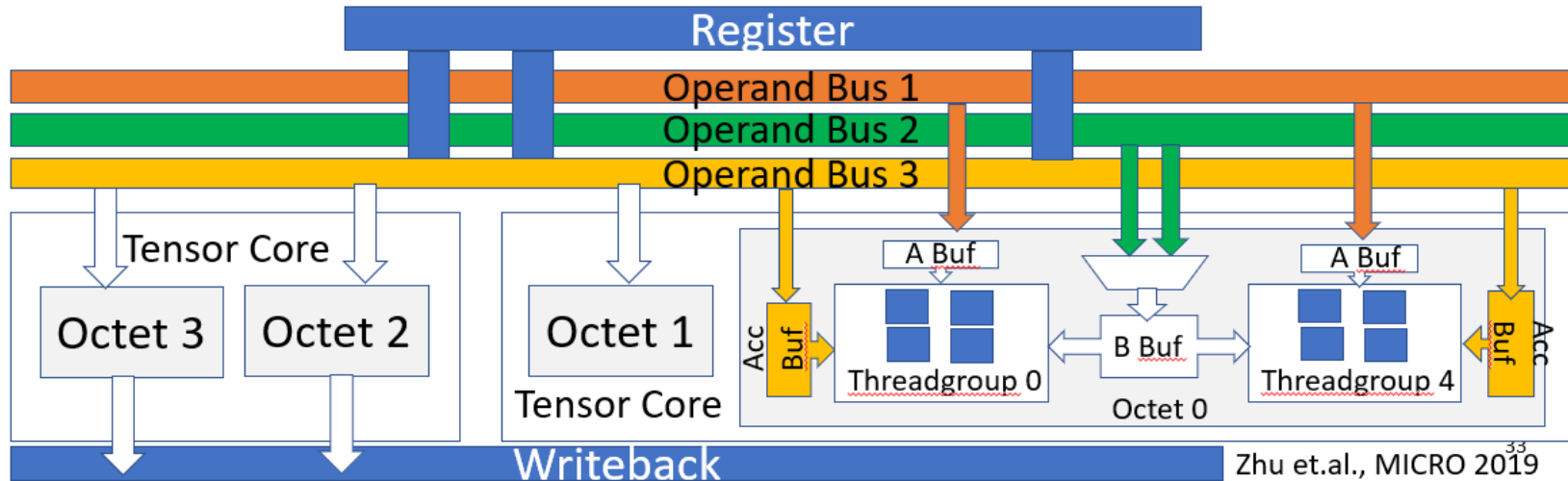
- One threadgroup computes  $4 \times 8 = 32$  dot products in one set of HMMA instruction
    - 4 DP units compute four 4-dim dot products per cycle
    - At least  $32/4 = 8$  cycles to finish a  $4 \times 8 \times 4$  matrix multiplication





# Tensor Core Microarchitecture

- Each tensor core performs 16 four-element dot products each cycle
- Each warp uses two tensor cores, two octets in a warp access each tensor core
- Matrix A and C, each threadgroup fetches operands to its separate buffer
- Threadgroups fetch matrix B operands to a shared buffer





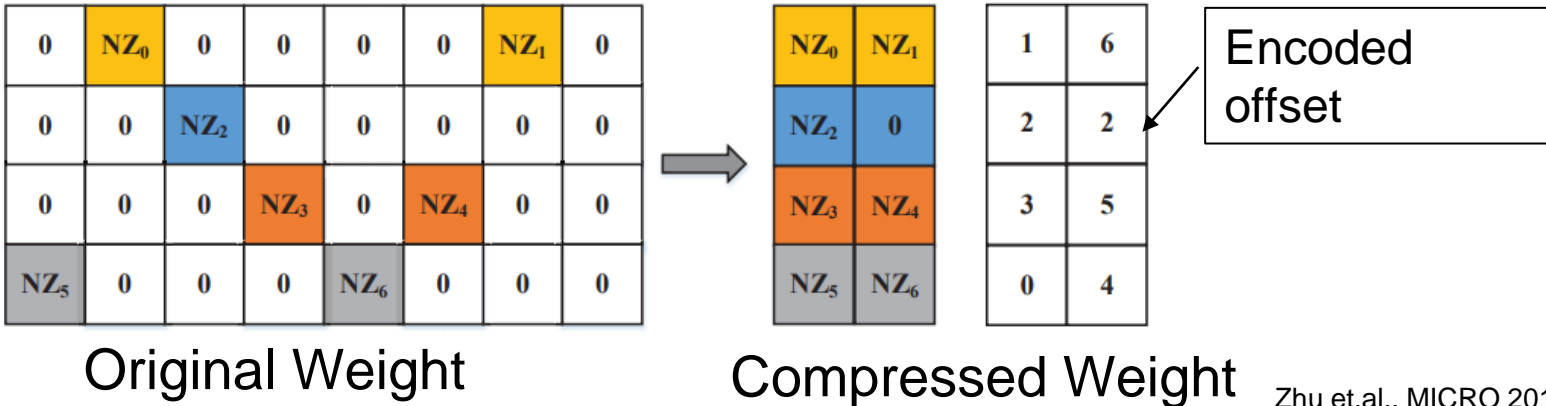
# What should we learn from Tensor Core ?

- Parallelism
  - Thread-level Parallelism (TLP) for MMA execution
  - Special functional units for DP calculation
- Data reuse
  - Increase the tiling block reuse through local memory buffer
- ISA Support
  - Need the supports from special ISA (WMMA) in the compiler
- What else ?



# 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 ?

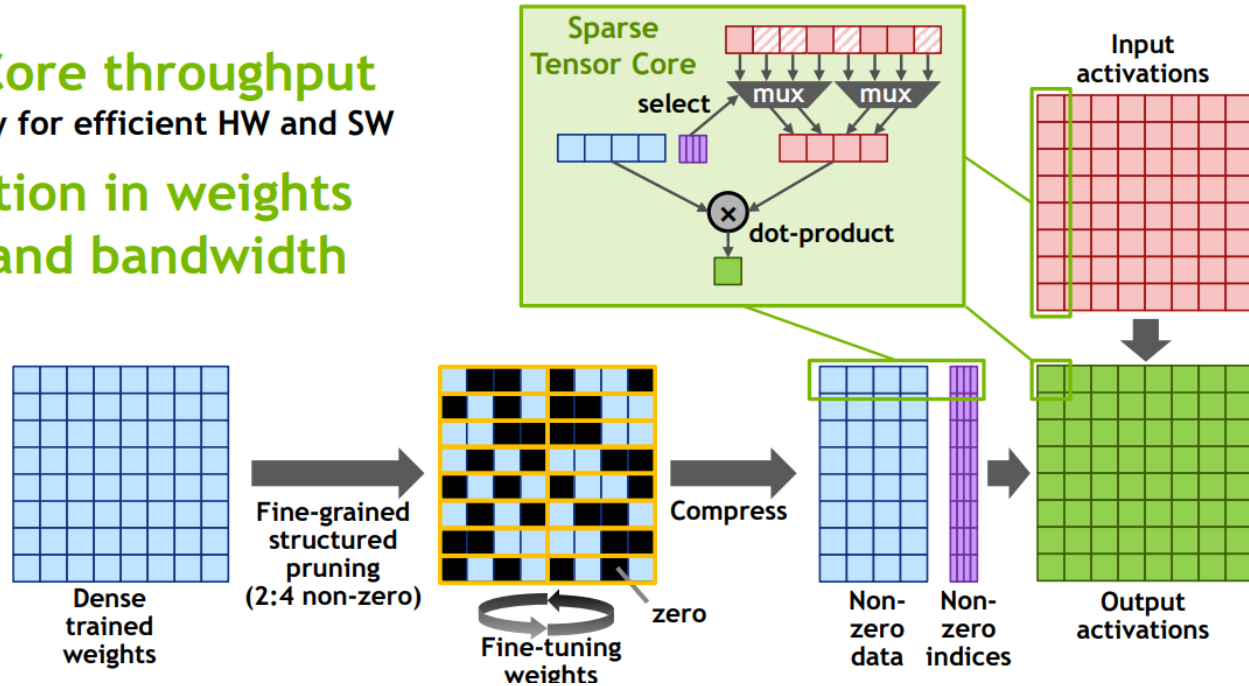


Zhu et.al., MICRO 2019



# Sparse Tensor Core in Nvidia A100 GPU

**2x Tensor Core throughput**  
Structured-sparsity for efficient HW and SW  
**~2x reduction in weights footprint and bandwidth**





# Dual-side sparse tensor core

- **Activation sparsity**

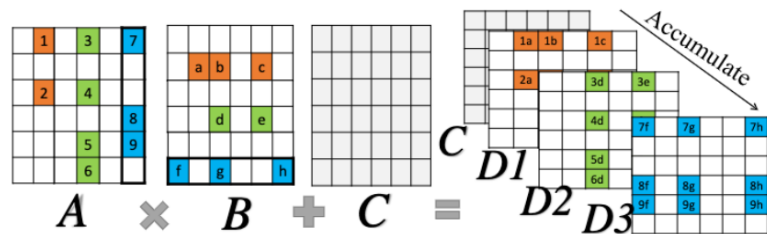
- Dynamic sparsity – the zero value was created during the runtime
- Hard to predict, data dependent

- **Dual-side sparse tensor core**

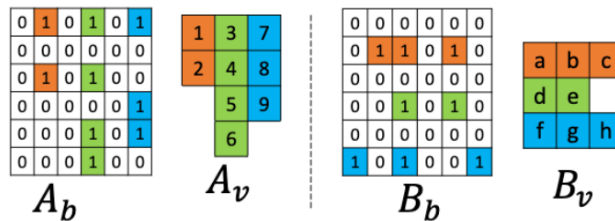
- Support SpCONV and SpGEMM
- Outer-product-based tensor core

- **How to encode dynamic sparsity ?**

- Bitmap encoding
- Each matrix has a b(bitmap) and a v(value) matrix



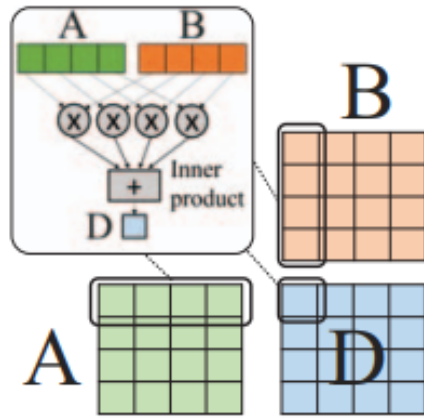
(a) Dense outer product.



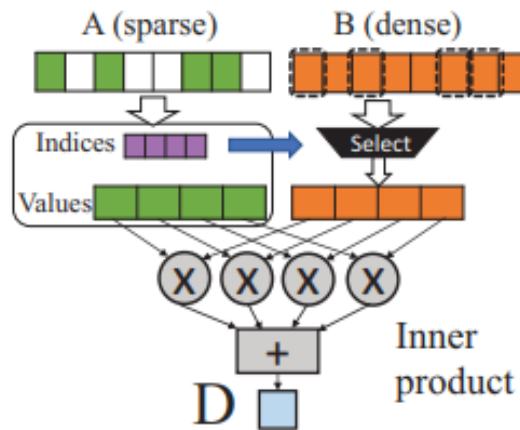




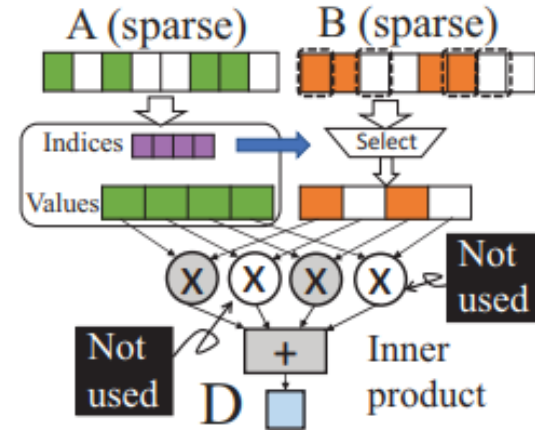
# Tensor Core Comparison



4 x 4 x 4 matrix multiplication



Sparse inner-product unit



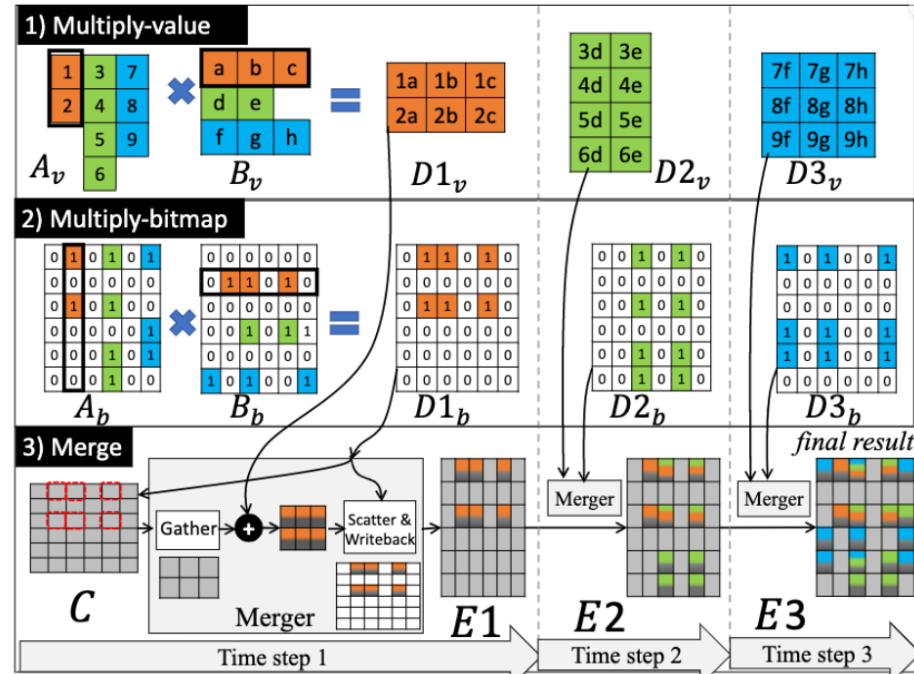
Dual-side sparsity unit



# Bitmap-encoding outer product

## ● Outer-product SpGEMM

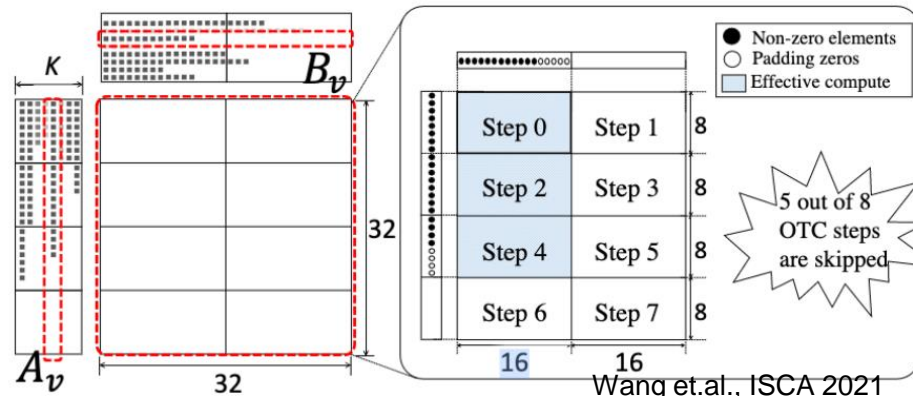
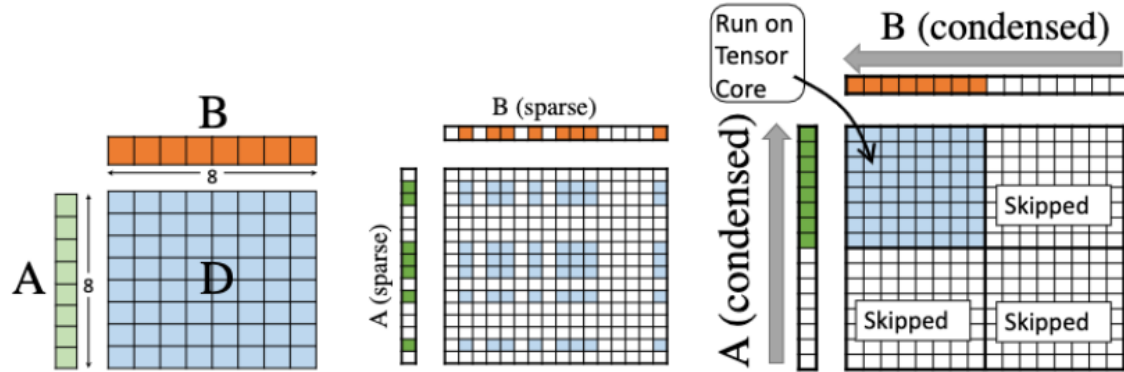
- Multiply matrix  $v$
- Multiply matrix  $b$
- Merger
  - Fetch updated values from matrix  $b$
  - Accumulate values in matrix  $v$





# Outer product tensor core

- **Outer product tensor core (OTC)**
  - The size of matrix in OTC is  $8 \times 8$
  - The size of  $A$  and  $B$  is  $(32, k)$  and  $(k, 32)$
  - Two tensor cores do  $8 \times 16$  matrix comp.
  - The data sparsity decides the rate of acceleration

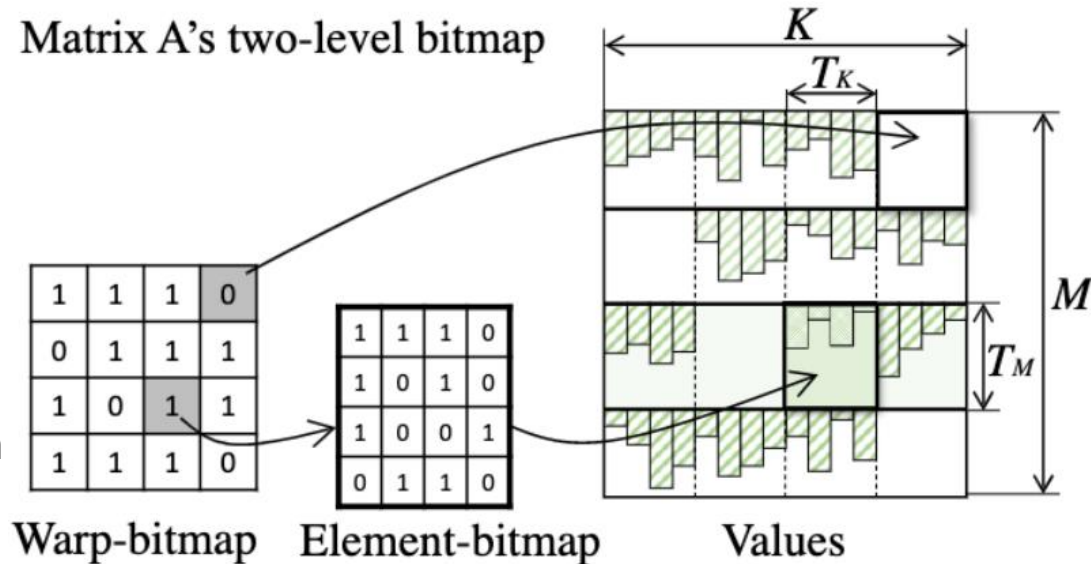




# Two-level Bitmap Encoding

- **Two-level bitmap encoding**

- When the size of matrix is too large
- Bitmap matrix is large too
- Warp bitmap
  - Represent if a tile has value
- Element bitmap
  - Represent the location of non-zero in a tile





# Outer-product friendly im2col

## • The im2col work

- Rearranges input feature maps as an input of GEMM
- Improperly designed
  - Harm input data reuse
- Sliding a 1 x 4 window
- Zig-zag way to scan over the feature map



(a) Inner product friendly im2col.





# Takeaway Questions

- How does tensor core accelerate the matrix computation ?
  - (A) Increase the on-chip buffer size
  - (B) Increase the frequency of tensor cores
  - (C) Reduce the data movement
- How to increase the utilization of the tensor core ?
  - (A) Use image to column (Im2col)
  - (B) Lower the data precision (using int8)
  - (C) Increase the number of registers