King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 23: Prefix Sum Bank Conflicts; Programming Tensor Cores

Markus Hadwiger, KAUST

Reading Assignment #9 (until Nov 4)

Read (required):

- Programming Massively Parallel Processors book, 4th edition
  Chapter 11: Prefix Sum (Scan) – an introduction to work efficiency in parallel algorithms
- Warp Shuffle Functions
  - CUDA Programming Guide, Chapter 10.22 (pdf; 7.22 online)

Read (optional):

- Guy E. Blelloch: Prefix Sums and their Applications
  - `https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf/`
- CUDA Cooperative Groups
  - CUDA Programming Guide, Chapter 11 (pdf; 8 online)
  - `https://developer.nvidia.com/blog/cooperative-groups/`
- Warp Matrix Functions (==tensor core programming)
  - CUDA Programming Guide, Chapter 10.24 (pdf; 7.24 online)

Next Lectures

Lecture 24: Thu, Oct 31

Lecture 25: Mon, Nov 4

Lecture 26: Tue, Nov 5 (make-up lecture; 14:30 – 15:45)

Lecture 27: Thu, Nov 7: Vulkan tutorial #2

GPU Parallel Prefix Sum

- Basic parallel programming primitive;
  parallelize inherently sequential operations

Parallel Prefix Sum (Scan)

- **Definition:**

  The all-prefix-sums operation takes a binary associative operator ⊕ with identity *I*, and an array of n elements

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  and returns the ordered set

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- **Example:**
  if ⊕ is addition, then scan on the set

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

Work Efficiency

Guy E. Blelloch and Bruce M. Maggs:
Parallel Algorithms, 2004 (https://www.cs.cmu.edu/~guyb/papers/BM04.pdf)

In designing a parallel algorithm, it is more important to make it efficient than to make it asymptotically fast. The efficiency of an algorithm is determined by the total number of operations, or work that it performs. On a sequential machine, an algorithm's work is the same as its time. On a parallel machine, the work is simply the processor-time product. Hence, an algorithm that takes time t on a P-processor machine performs work W = Pt. In either case, the work roughly captures the actual cost to perform the computation, assuming that the cost of a parallel machine is proportional to the number of processors in the machine.

We call an algorithm work-efficient (or just efficient) if it performs the same amount of work, to within a constant factor, as the fastest known sequential algorithm.

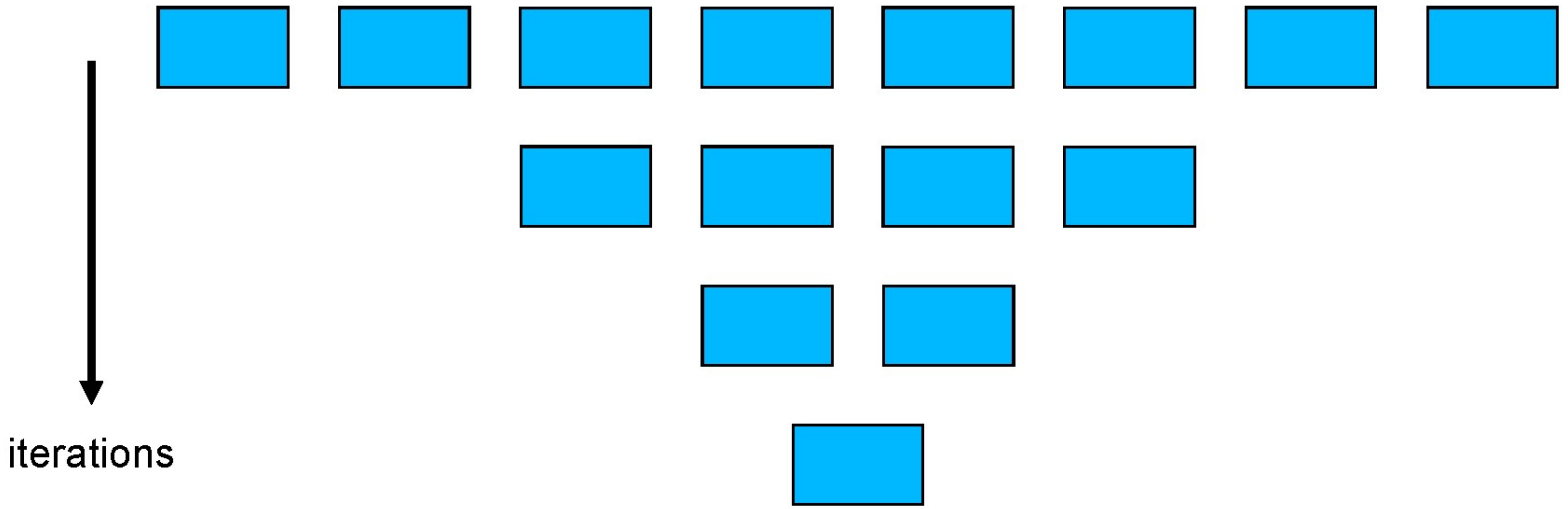
For example, a parallel algorithm that sorts n keys in O(sqrt(n) log(n)) time using sqrt(n) processors is efficient since the work, O(n log(n)), is as good as any (comparison-based) sequential algorithm.

However, a sorting algorithm that runs in O(log(n)) time using n^2 processors is not efficient.

The first algorithm is better than the second - even though it is slower - because its work, or cost, is smaller. Of course, given two parallel algorithms that perform the same amount of work, the faster one is generally better.

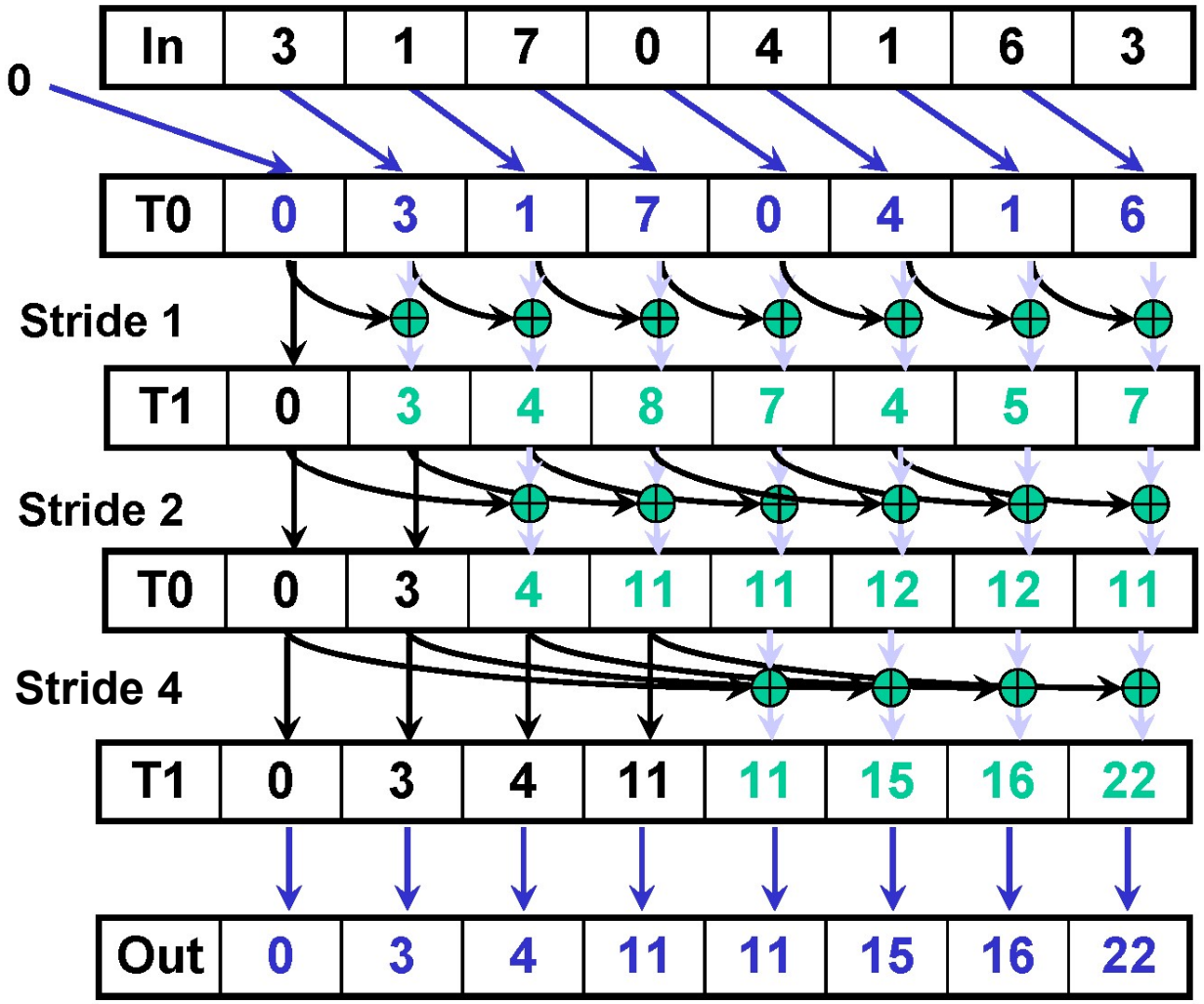
Typical Parallel Programming Pattern

- **log(n) steps**



Helpful fact for counting nodes of full binary trees:
If there are N leaf nodes, there will be N-1 non-leaf nodes

A First-Attempt Parallel Scan Algorithm



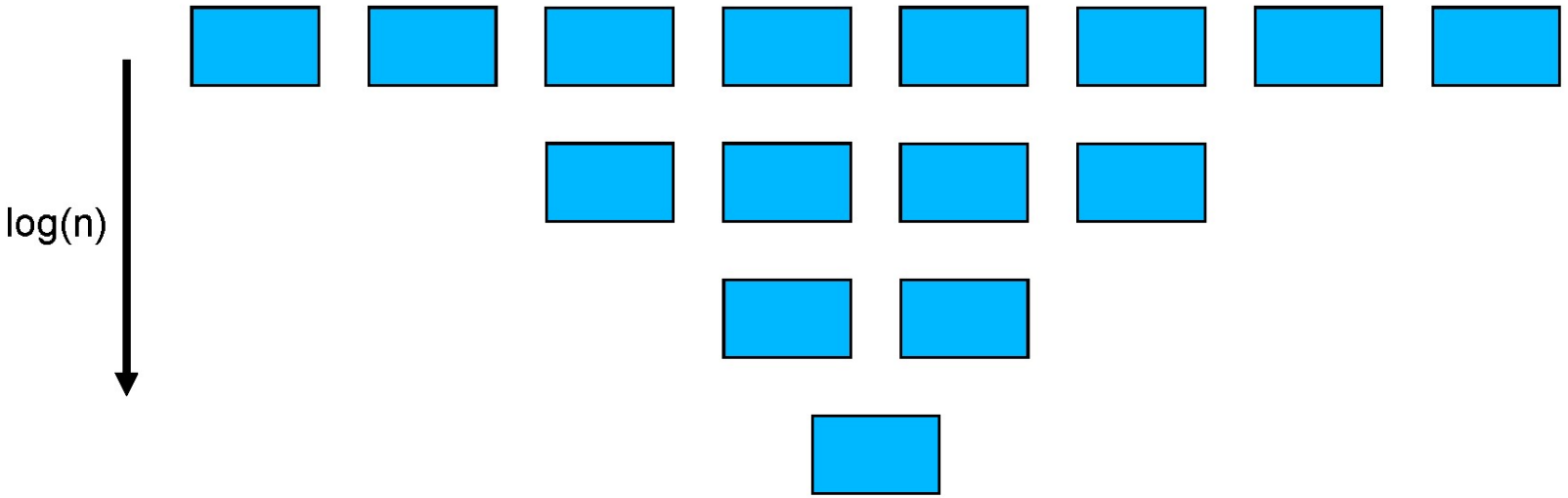
1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)
3. Write output to device memory.

Work Efficiency Considerations

- **The first-attempt Scan executes log(n) parallel iterations**
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **A parallel algorithm can be slow when execution resources are saturated due to low work efficiency**

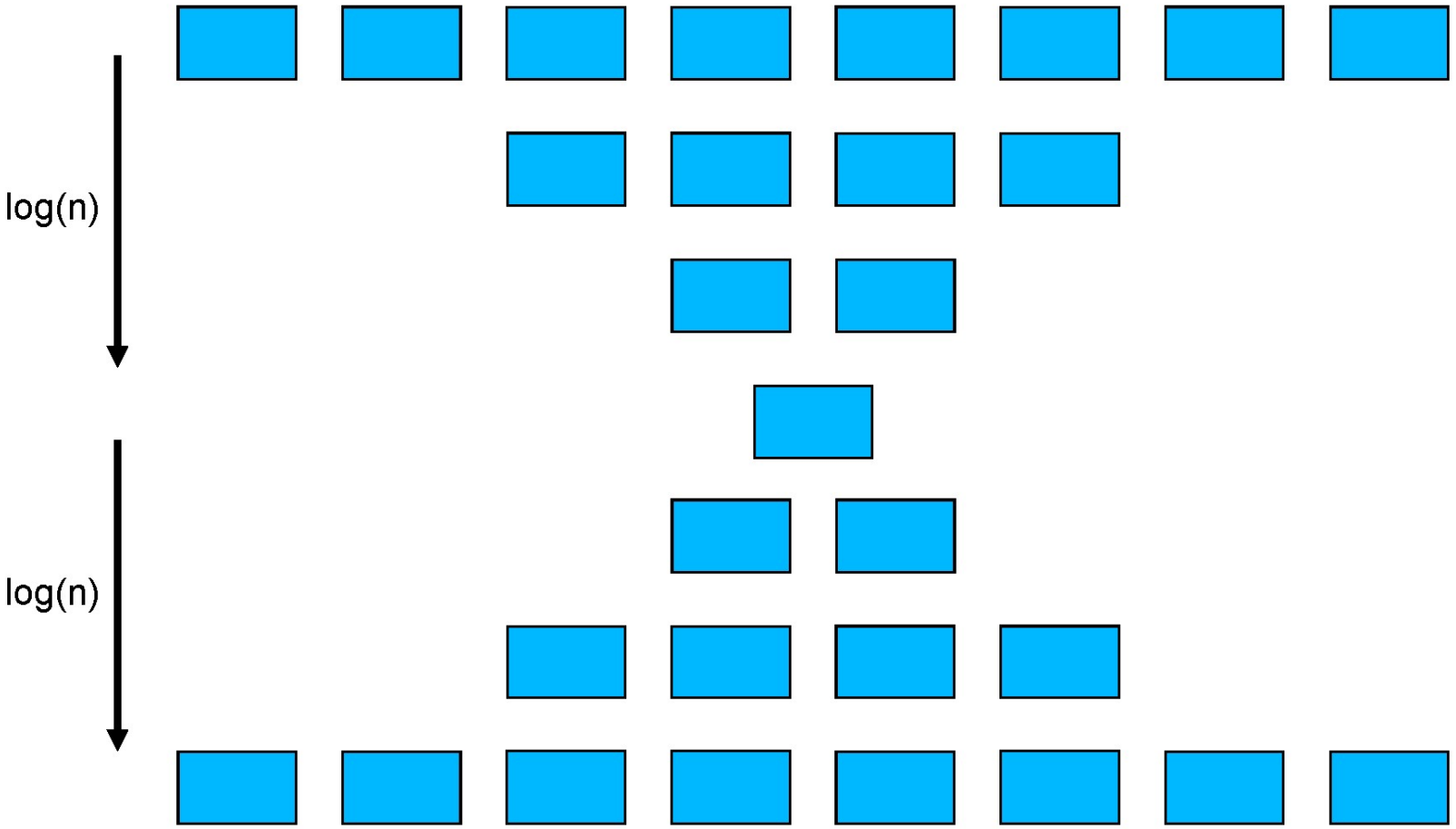
Typical Parallel Programming Pattern

- **2 log(n) steps**



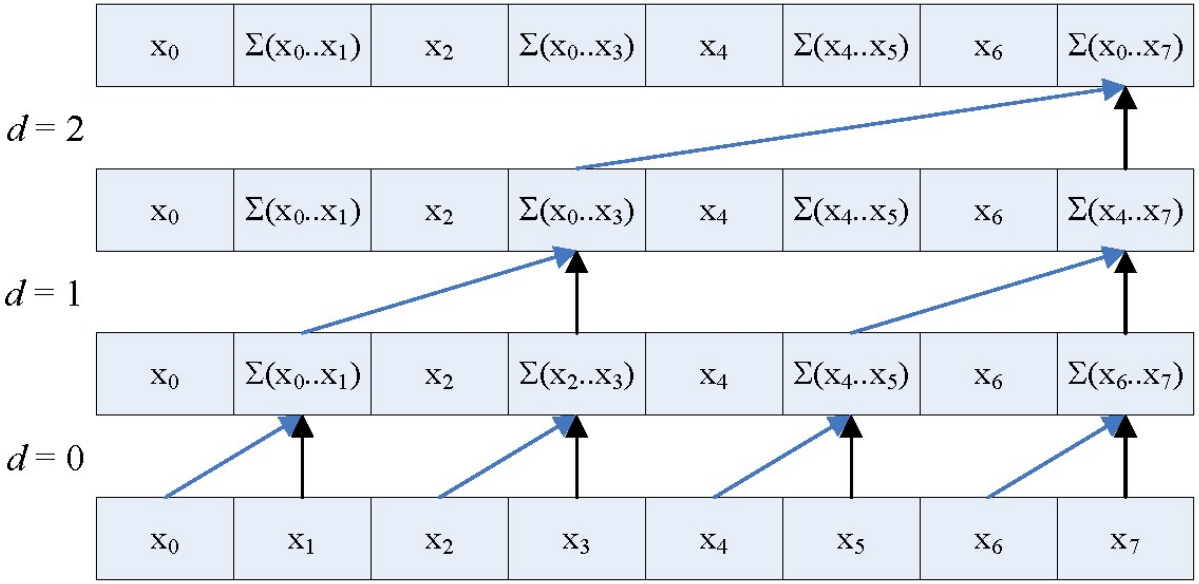
Typical Parallel Programming Pattern

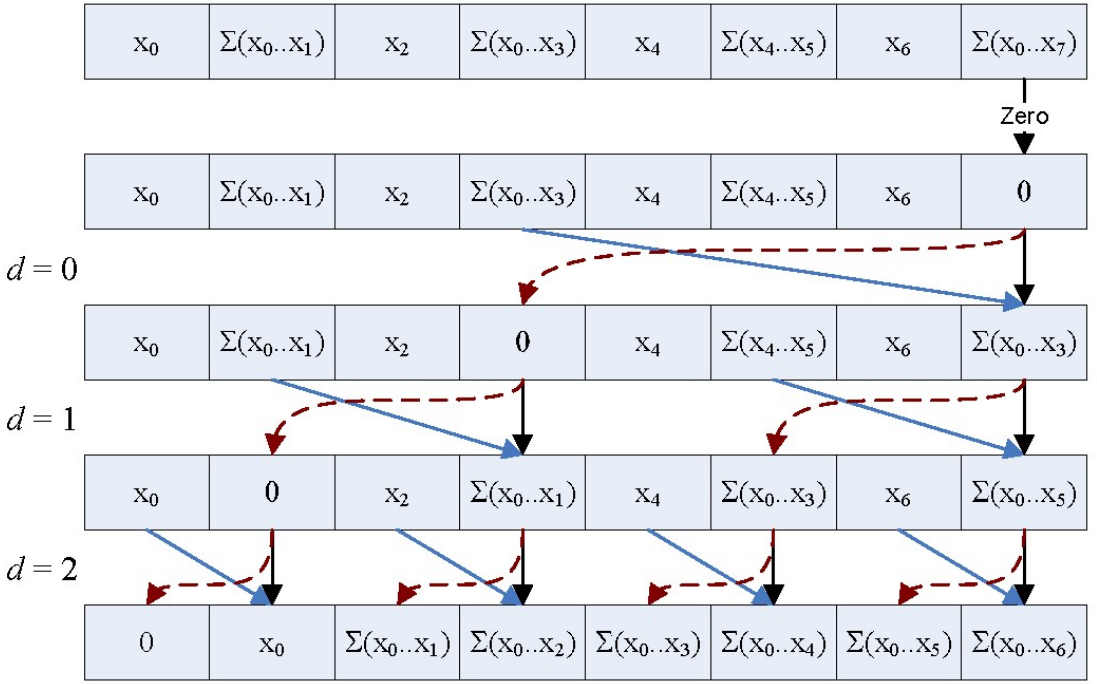
- **2 log(n) steps**



$O(n)$ Scan [Blelloch]

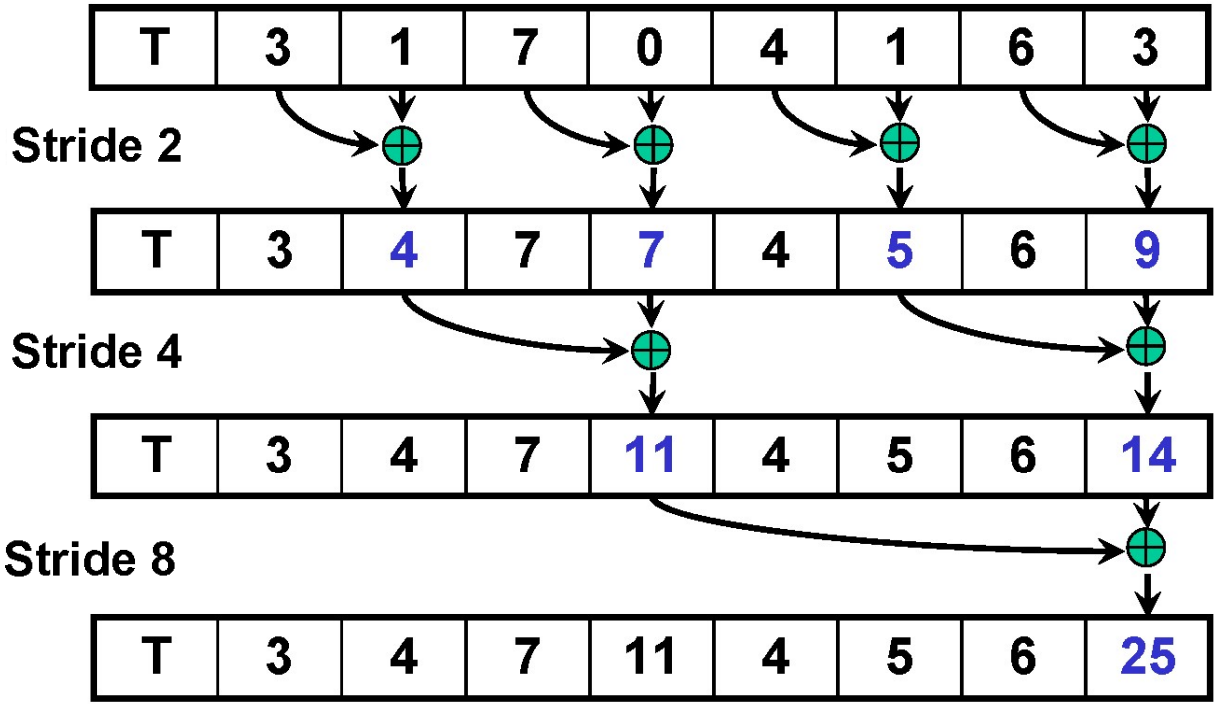
Courtesy John Owens





- Work efficient ($O(n)$ work)
- Bank conflicts, and lots of 'em

Build the Sum Tree



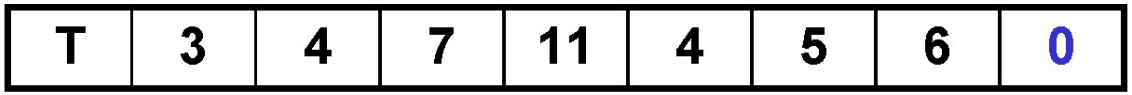
Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride / 2* elements away to its own value.

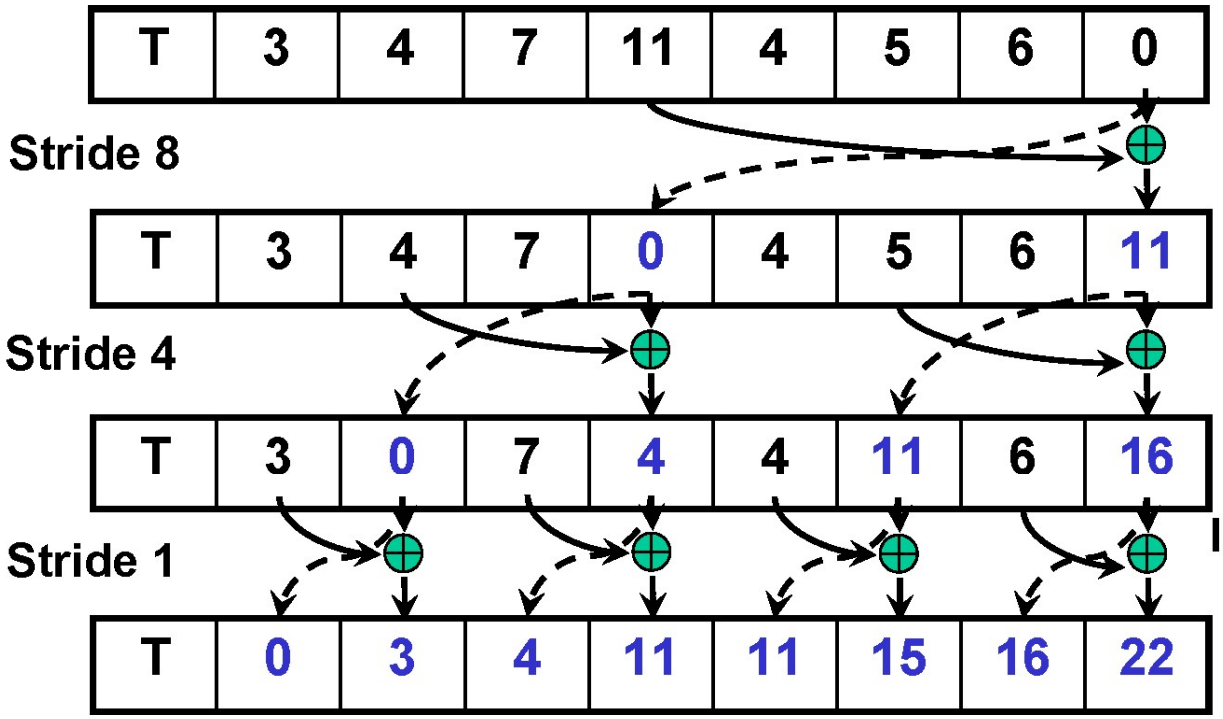
Note that this algorithm operates in-place: no need for double buffering

Down-Sweep Variant 1: Exclusive Scan



We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

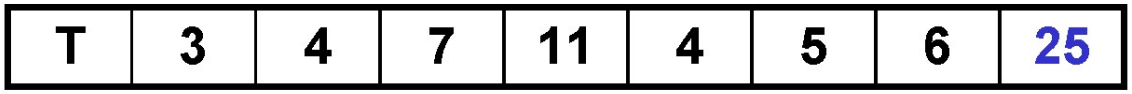
Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

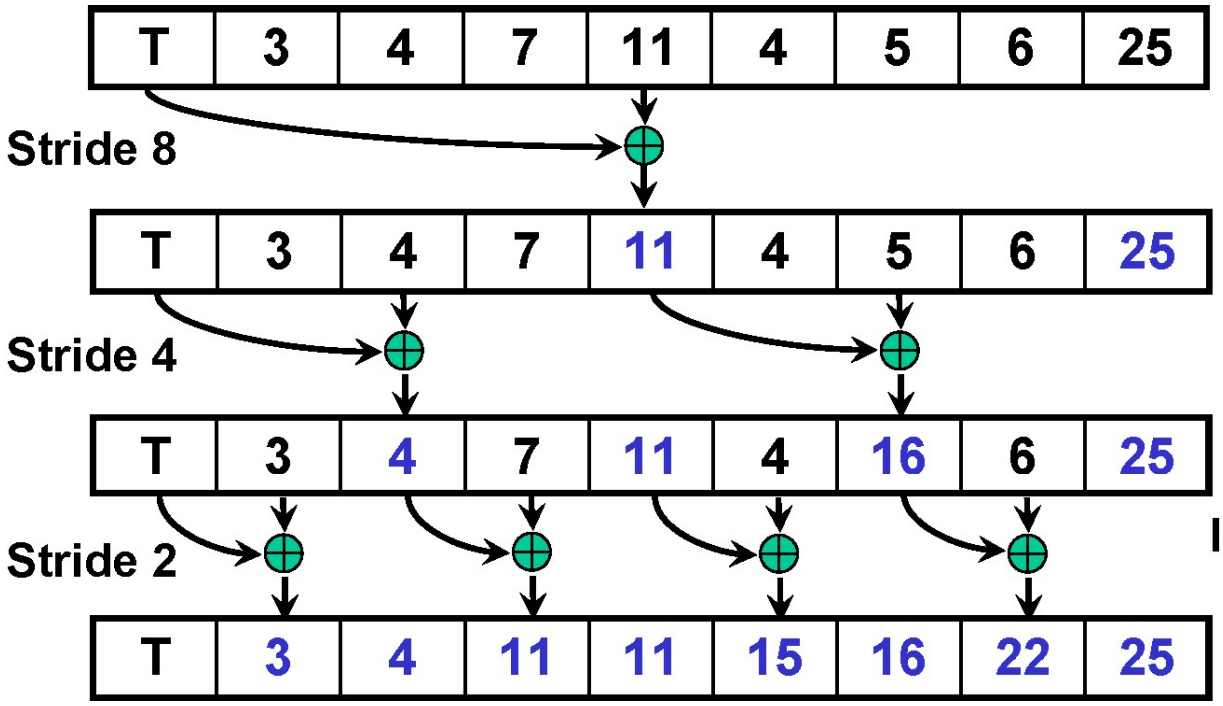
Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Down-Sweep Variant 2: Inlusive Scan



We now have an array of partial sums. Let's propagate the sums back.

Build Scan From Partial Sums



Iteration log(*n*)
***n*/2 threads**

Each ⊕ corresponds to a single thread.

Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: < 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Bank Conflicts in Scan
- Non-power-of-two -

Initial Bank Conflicts on Load

- **Each thread loads two shared mem data elements**
- **Tempting to interleave the loads**

```
temp[2*thid]   = g_idata[2*thid];
temp[2*thid+1] = g_idata[2*thid+1];
```

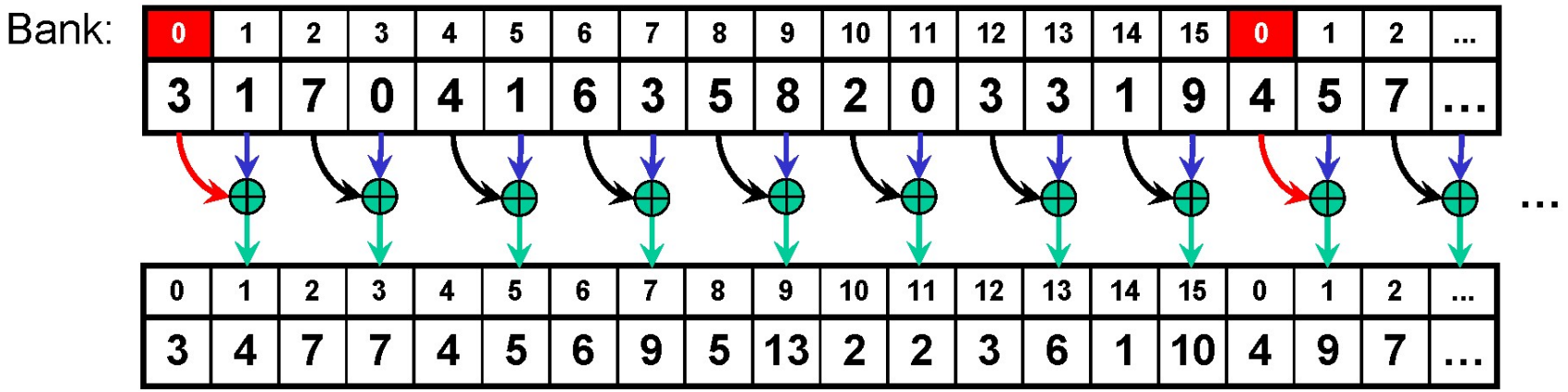
- **Threads:(0,1,2,…,8,9,10,…)→banks:(0,2,4,…,0,2,4,…)**

- **Better to load one element from each half of the array**

```
temp[thid]          = g_idata[thid];
temp[thid + (n/2)] = g_idata[thid + (n/2)];
```

Bank Conflicts in the tree algorithm

- **When we build the sums, each thread reads two shared memory locations and writes one:**
- **Th(0,8) access bank 0**



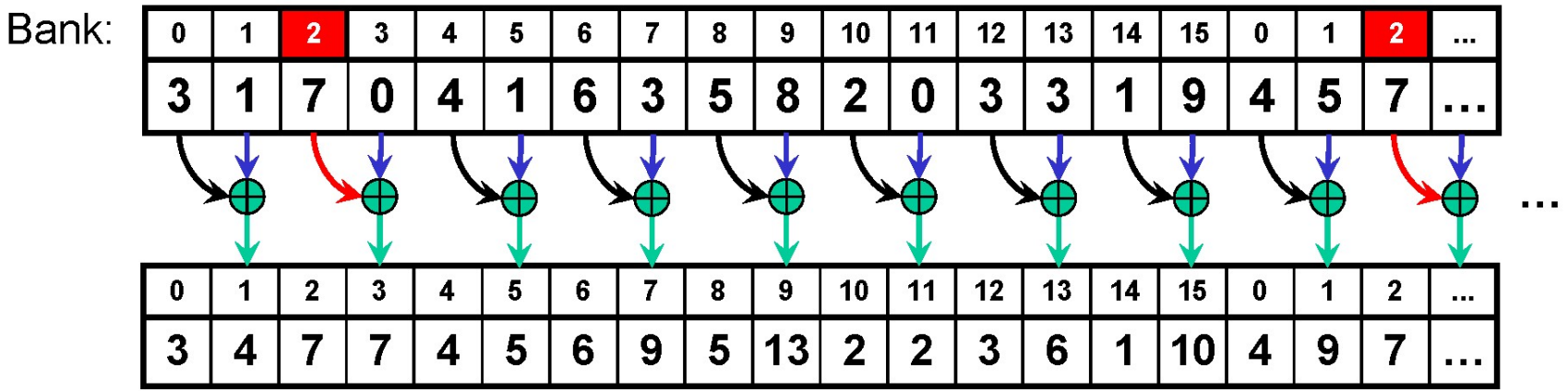
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **When we build the sums, each thread reads two shared memory locations and writes one:**
- **Th(1,9) access bank 2, etc.**



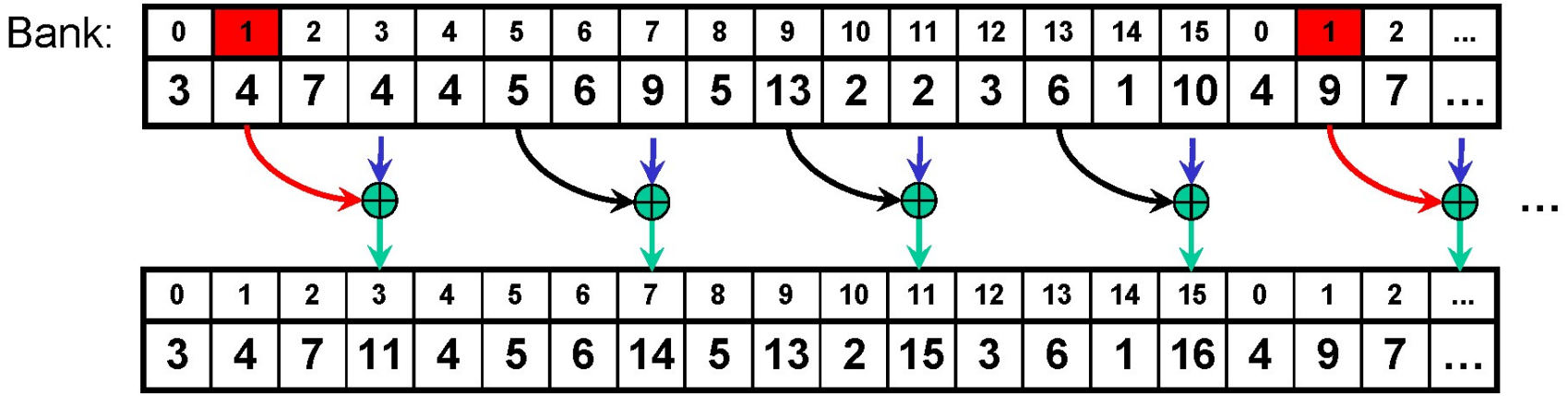
First iteration: 2 threads access each of 8 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Bank Conflicts in the tree algorithm

- **2nd iteration: even worse!**
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.



2nd iteration: 4 threads access each of 4 banks.

Each ⊕ corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses

Scan Bank Conflicts (1)

- **A full binary tree with 64 leaf nodes:**

Scale (s)	Thread addresses																															
1	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60																
4	0	8	16	24	32	40	48	56																								
8	0	16	32	48																												
16	0	32																														
32	0																															
Conflicts	Banks																															
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12																
4-way	0	8	0	8	0	8	0	8																								
4-way	0	0	0	0																												
2-way	0	0																														
None	0																															

- **Multiple 2-and 4-way bank conflicts**
- **Shared memory cost for whole tree**
  - 1 32-thread warp = 6 cycles per thread w/o conflicts
    - Counting 2 shared mem reads and one write (s[a] += s[b])
  - 6 * (2+4+4+4+2+1) = 102 cycles
  - 36 cycles if there were no bank conflicts (6 * 6)

Scan Bank Conflicts (2)

- **It's much worse with bigger trees!**
- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

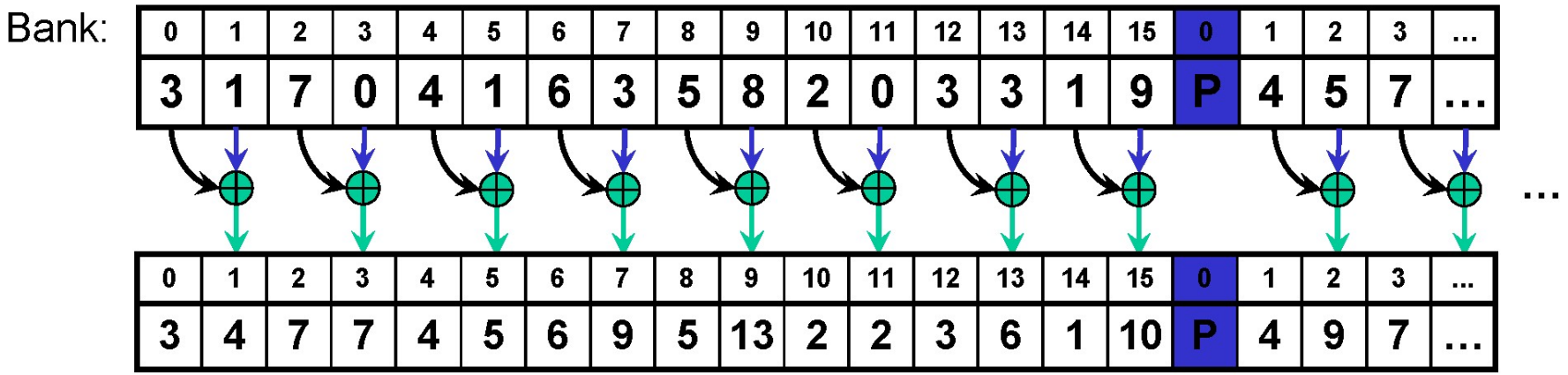
Scale (s)	Thread addresses																																
2	0	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100	104	108	112	116	120	122	
4	0	8	16	24	32	40	48	56	64	72	80	88	96	104	112	120																	
8	0	16	32	48	64	80	96	112																									
16	0	32	64	96																													
32	0	64																															
64	0																																
Conflicts	Banks																																
4-way	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	12	0	4	8	10	
8-way	0	8	0	8	0	8	0	8	0	8	0	8	0	8	0	8																	
8-way	0	0	0	0	0	0	0	0																									
4-way	0	0	0	0																													
2-way	0	0																															
None	0																																

- **Cost for whole tree:**
  - 12*2 + 6*(4+8+8+4+2+1) = 186 cycles
  - 48 cycles if there were no bank conflicts! 12*1 + (6*6)

Bank Conflicts in the tree algorithm

- **We can use padding to prevent bank conflicts**
  - Just add a word of padding every 16 words:
- **No more conflicts!**

32 for full warps!



Now, within a 16-thread half-warp, all threads access different banks.

32-thread full warp!

(Note that only arrows with the same color happen simultaneously.)

Use Padding to Reduce Conflicts

- **This is a simple modification to the last exercise**
- **After you compute a shared mem address like this:**

```
Address =     stride * thid;
```

- **Add padding like this:**

```
Address += (Address >> 4); // divide by NUM_BANKS
```

- **This removes most bank conflicts**
  - Not all, in the case of deep trees

Fixing Scan Bank Conflicts

- **Insert padding every NUM_BANKS elements**

```
const int LOG_NUM_BANKS = 4; // 16 banks
int tid = threadIdx.x;
int s = 1;
// Traversal from leaves up to root
for (d = n>>1; d > 0; d >>= 1)
{
    if (thid <= d)
    {
        int a = s*(2*tid); int b = s*(2*tid+1)
        a += (a >> LOG_NUM_BANKS); // insert pad word
        b += (b >> LOG_NUM_BANKS); // insert pad word
        shared[a] += shared[b];
    }
}
```

Fixing Scan Bank Conflicts

- **A full binary tree with 64 leaf nodes**

Leaf Nodes	Scale (s)	Thread addresses																													
64	1	0	2	4	6	8	10	12	14	17	19	21	23	25	27	29	31	34	36	38	40	42	44	46	48	51	53	55	57	59	61	63
	2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63															
	4	0	8	17	25	34	42	51	59																							
	8	0	17	34	51																											
	16	0	34											= Padding inserted																	
	32	0																													
	Conflicts	Banks																													
	None	0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15	2	4	6	8	10	12	14	0	3	5	7	9	11	13	15
	None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15															
	None	0	8	1	9	2	10	3	11																							
	None	0	1	2	3																											
	None	0	2																													
	None	0																													

- **No more bank conflicts!**
  - However, there are ~8 cycles overhead for addressing
    - For each s[a] += s[b] (8 cycles/iter. * 6 iter. = 48 extra cycles)
  - So just barely worth the overhead on a small tree
    - 84 cycles vs. 102 with conflicts vs. 36 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 128 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
2	0	4	8	12	17	21	25	29	34	38	42	46	51	55	59	63	68	72	76	80	85	89	93	97	102	106	110	114	119	123	127	131
4	0	8	17	25	34	42	51	59	68	76	85	93	102	110	119	127																
8	0	17	34	51	68	85	102	119																								
16	0	34	68	102																												
32	0	68											= Padding inserted																			
64	0																															
Conflicts	Banks																															
None	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15	4	8	12	0	5	9	13	1	6	10	14	2	7	11	15	3
None	0	8	1	9	2	10	3	11	4	12	5	13	6	14	7	15																
None	0	1	2	3	4	5	6	7																								
None	0	2	4	6																												
None	0	4																														
None	0																															

- **No more bank conflicts!**
  - Significant performance win:
    - 106 cycles vs. 186 with bank conflicts vs. 48 optimal

Fixing Scan Bank Conflicts

- **A full binary tree with 512 leaf nodes**
  - Only the last 6 iterations shown (root and 5 levels below)

Scale (s)	Thread addresses																															
8	0	17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340	357	374	391	408	425	442	459	476	493	510	527
16	0	34	68	102	136	170	204	238	272	306	340	374	408	442	476	510																
32	0	68	136	204	272	340	408	476																								
64	0	136	272	408																												
128	0	272												= Padding inserted																		
256	0																															
Conflicts	Banks																															
None	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2-way	0	2	4	6	8	10	12	14	0	2	4	6	8	10	12	14																
2-way	0	4	8	12	0	4	8	12																								
2-way	0	8	0	8																												
2-way	0	0																														
None	0																															

- **Wait, we still have bank conflicts**
  - Method is not foolproof, but still much improved
  - 304 cycles vs. 570 with bank conflicts vs. 120 optimal
- **But it does not pay of to optimize for the rest. Address calculations are getting too expensive**

Summary

- **Parallel Programming requires careful planning**
  - of the branching behavior
  - of the memory access patterns
  - of the work efficiency
- **Vector Reduction**
  - branch efficient
  - bank efficient
- **Scan Algorithm**
  - based in Balanced Tree principle:
    bottom up, top down traversal

Programming Tensor Cores

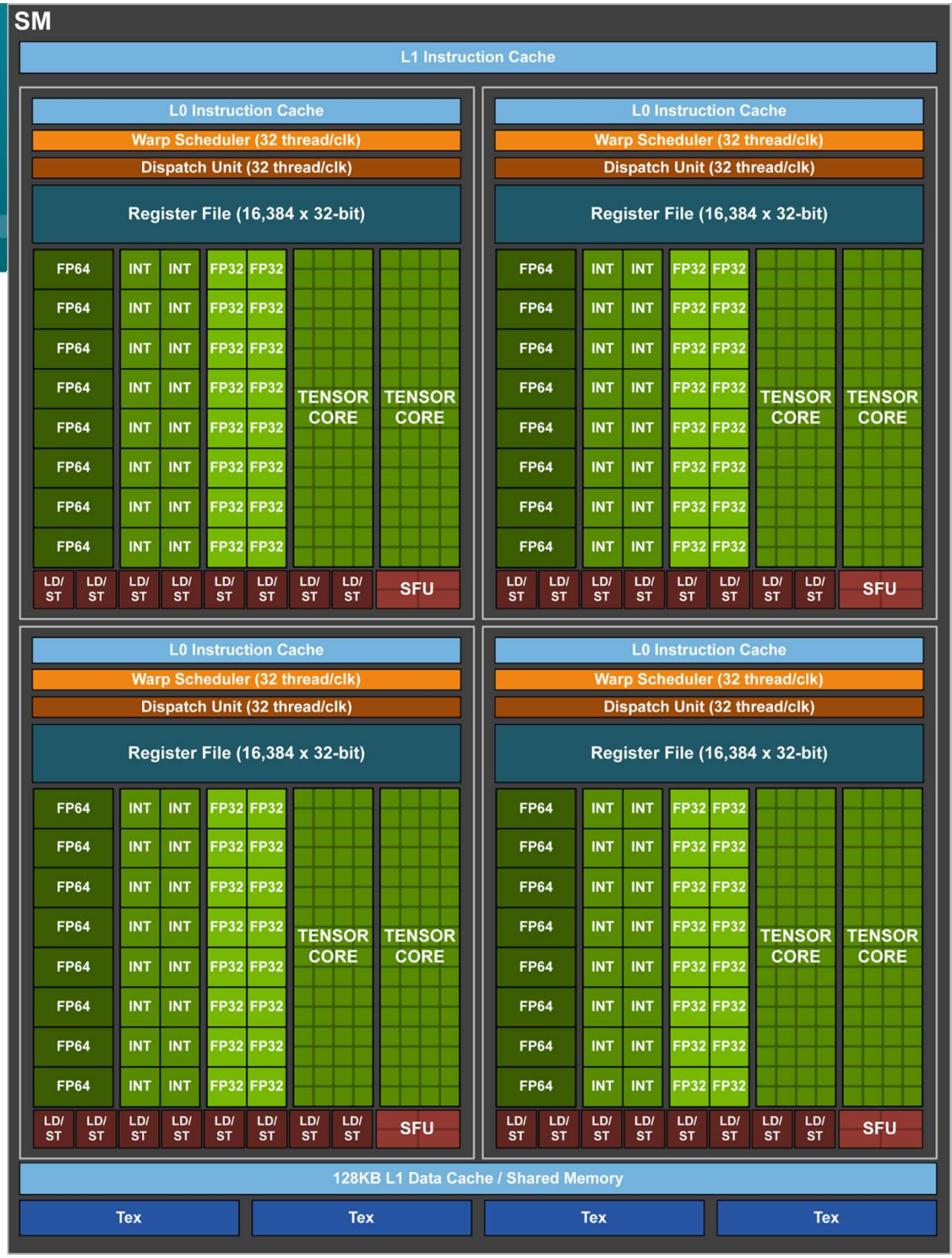
NVIDIA Volta SM

Multiprocessor: SM (CC 7.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 32 LD/ST units; 16 SFUs
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

- 16 FP32 + 16 INT32 cores each
- 8 FP64 cores each
- 8 LD/ST units; 4 SFUs each
- 2 tensor cores each
- Each has: warp scheduler, dispatch unit, register file



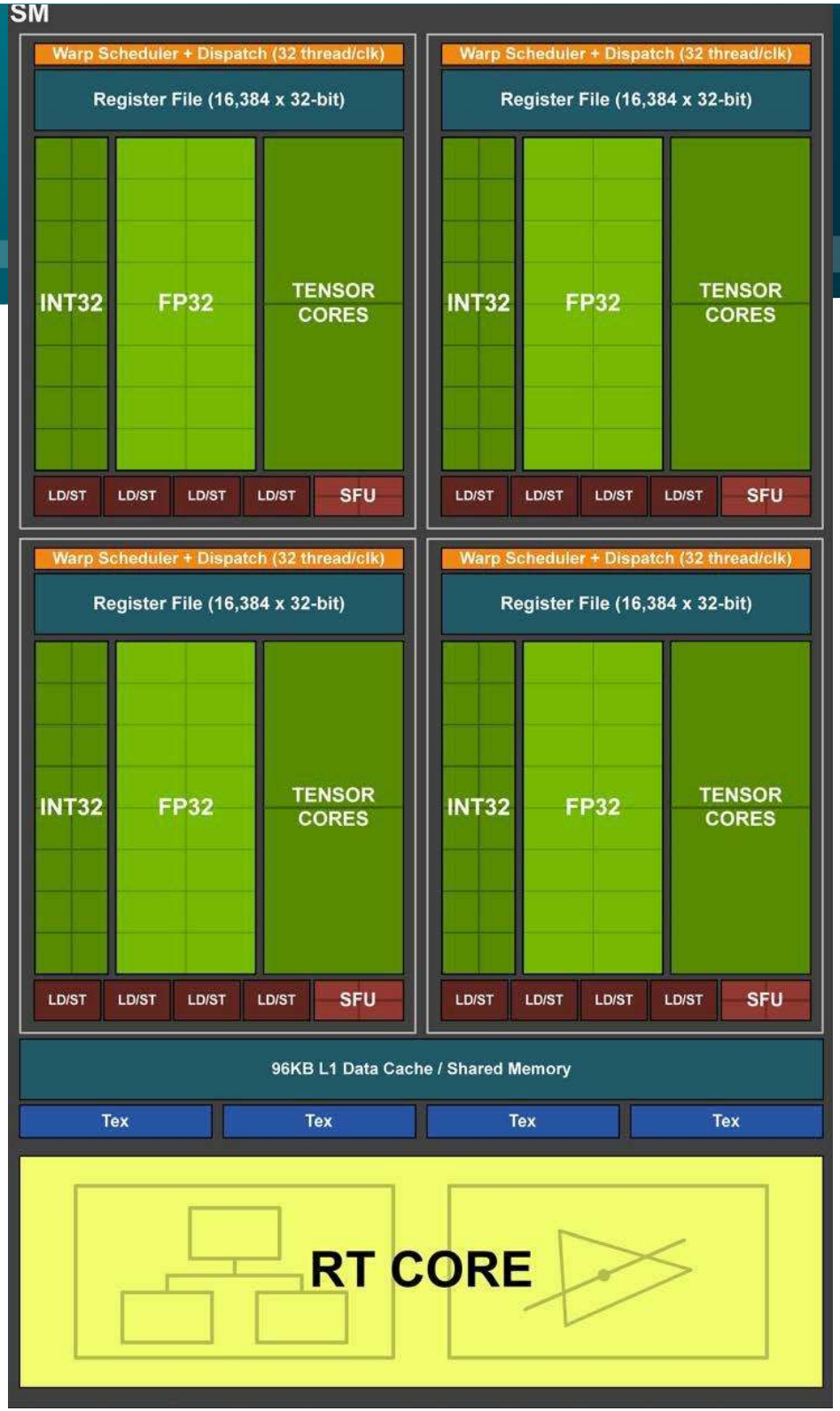
NVIDIA Turing SM

Multiprocessor: SM (CC 7.5)

- 64 FP32 + INT32 cores
- 2 (!) FP64 cores
- 8 Turing tensor cores (FP16/32, INT4/8 mixed-precision)
- 1 RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + INT32 cores each
- 4 LD/ST units; 4 SFUs each
- 2 Turing tensor cores each
- Each has: warp scheduler, dispatch unit, 16K register file



NVIDIA GA100 SM

Multiprocessor: SM (CC 8.0)

- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 16 FP32 + 16 INT32 cores
- 8 FP64 cores
- 8 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GA10x SM

Multiprocessor: SM (CC 8.6)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4 LD/ST units; 4 SFUs each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

NVIDIA GH100 SM

Multiprocessor: SM (CC 9.0)

- 128 FP32 + 64 INT32 cores
- 64 FP64 cores
- 4x 4th gen tensor cores
- ++ thread block clusters, DPX insts., FP8, TMA

4 partitions inside SM

- 32 FP32 + 16 INT32 cores
- 16 FP64 cores
- 8x LD/ST units; 4 SFUs each
- 1x 4th gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file

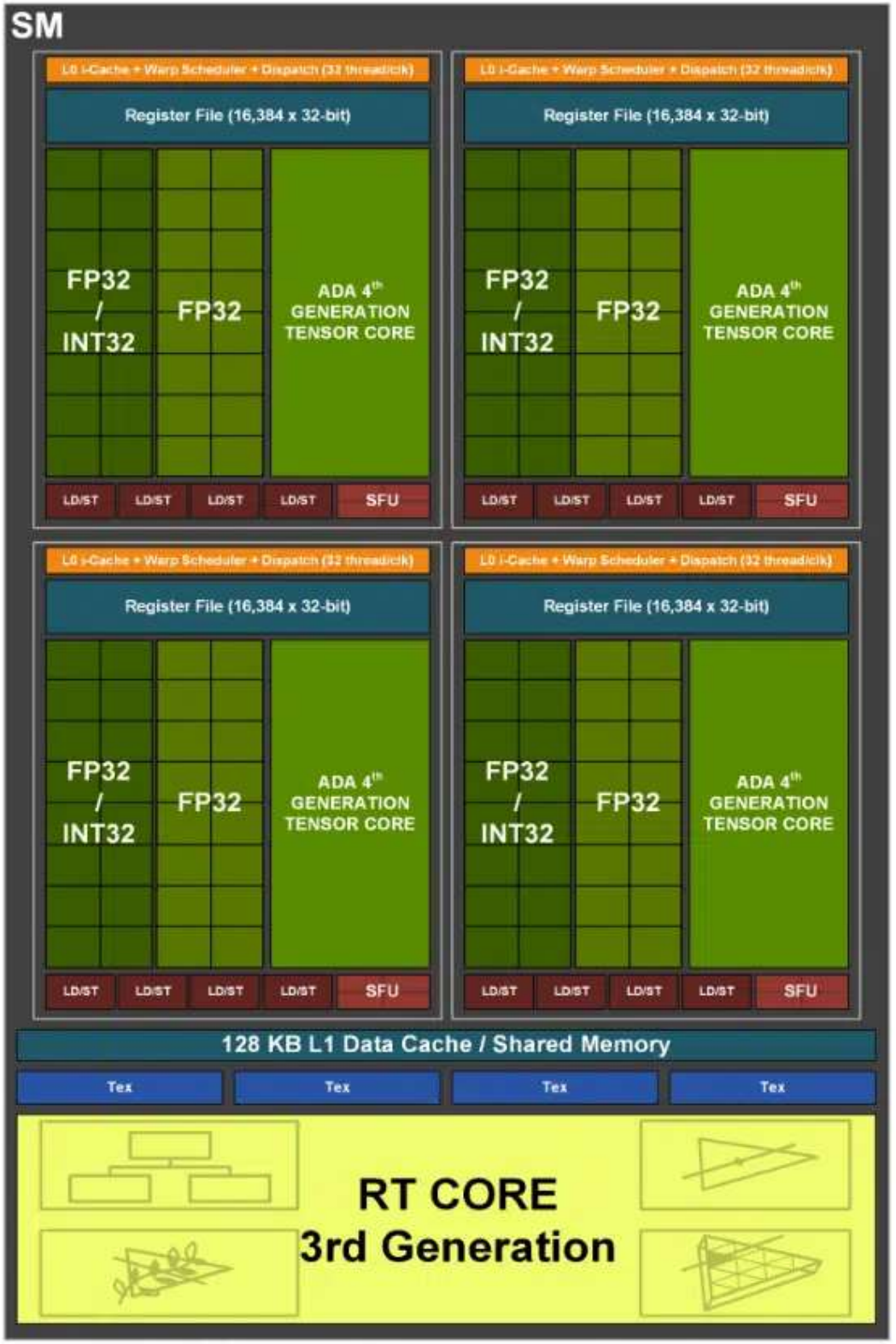
NVIDIA AD102 SM

Multiprocessor: SM (CC 8.9)

- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4x 4th gen tensor cores
- 1x 3rd gen RT (ray tracing) core
- ++ thread block clusters, FP8, … (?)

4 partitions inside SM

- 32 (16+16) FP32 + 16 INT32 cores
- 4x LD/ST units; 4 SFUs each
- 1x 4th gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file



Tensor Cores

Mixed-precision, fast matrix-matrix multiply and accumulate (mma)

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

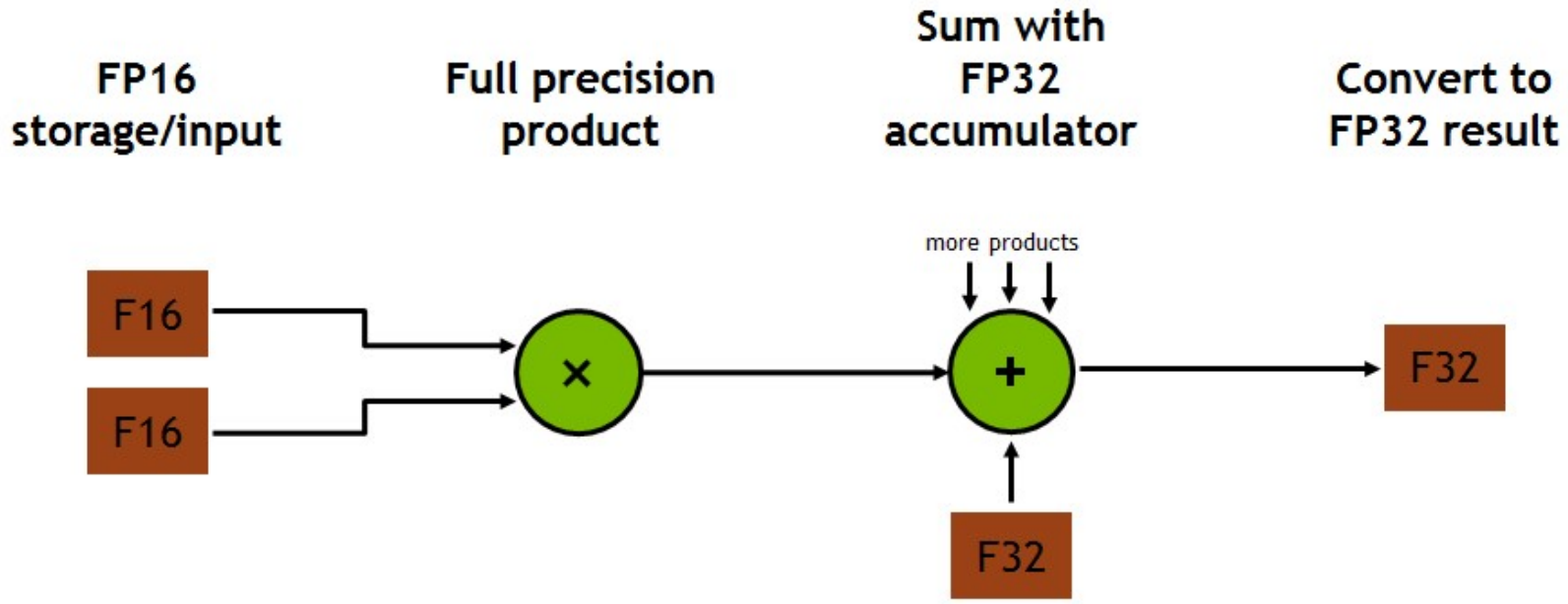
From this, build larger shapes (sizes), higher dimensionalities, ...

API currently only allows using larger shapes (16x16, ...) in warps (wmma)

Tensor Cores

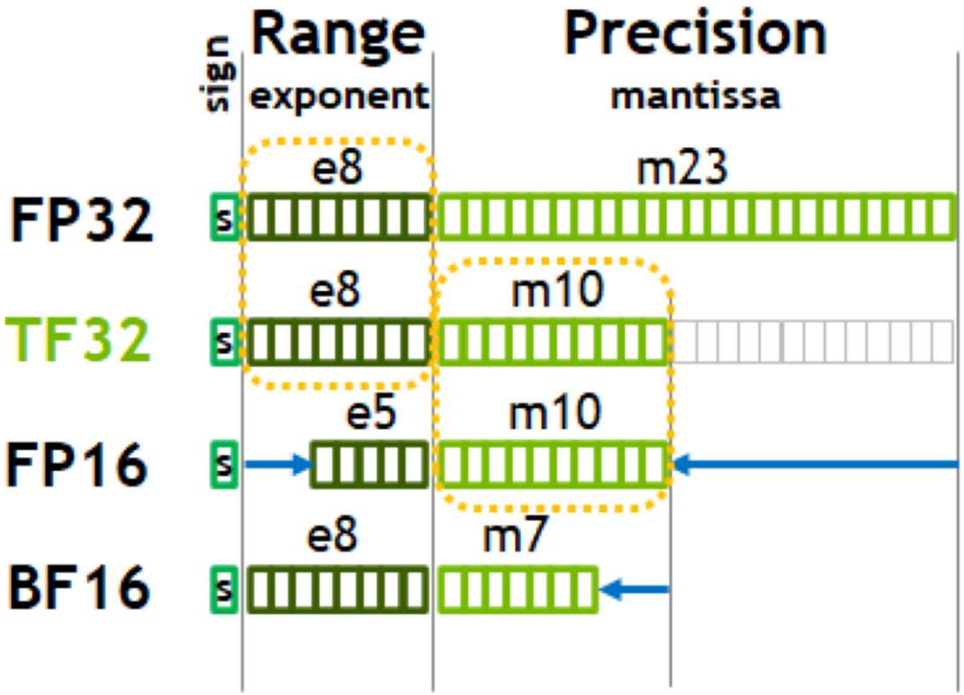
Fused matrix multiply and accumulate

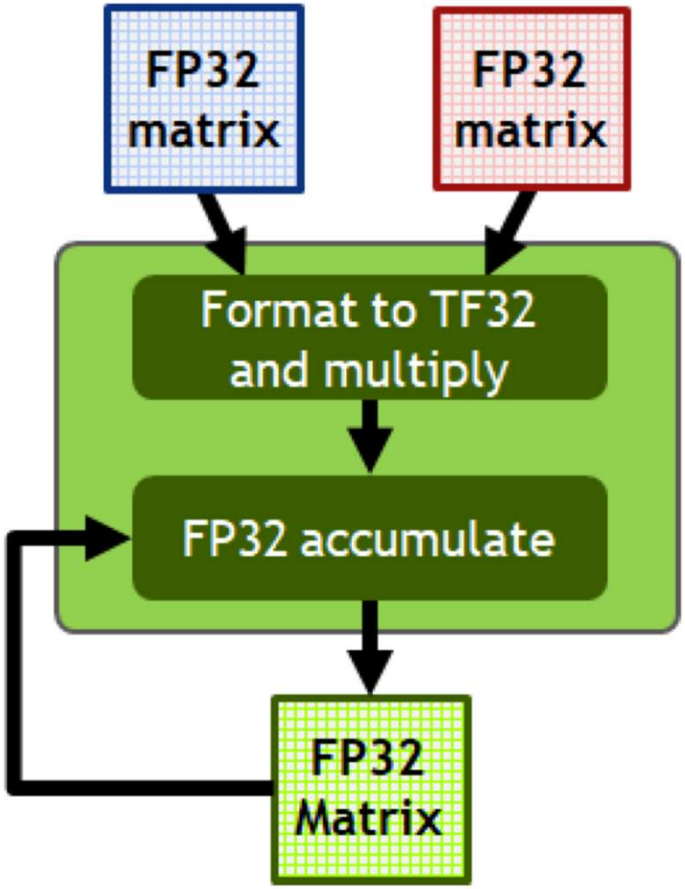
- Input matrices can be (at most) half-precision (FP16); (**Ampere has more!**)
- Accumulate can be FP16 or FP32; (**Ampere has more!**)



Ampere Tensor Cores: Mixed Precision

New in Ampere: TF32, BF16, FP64





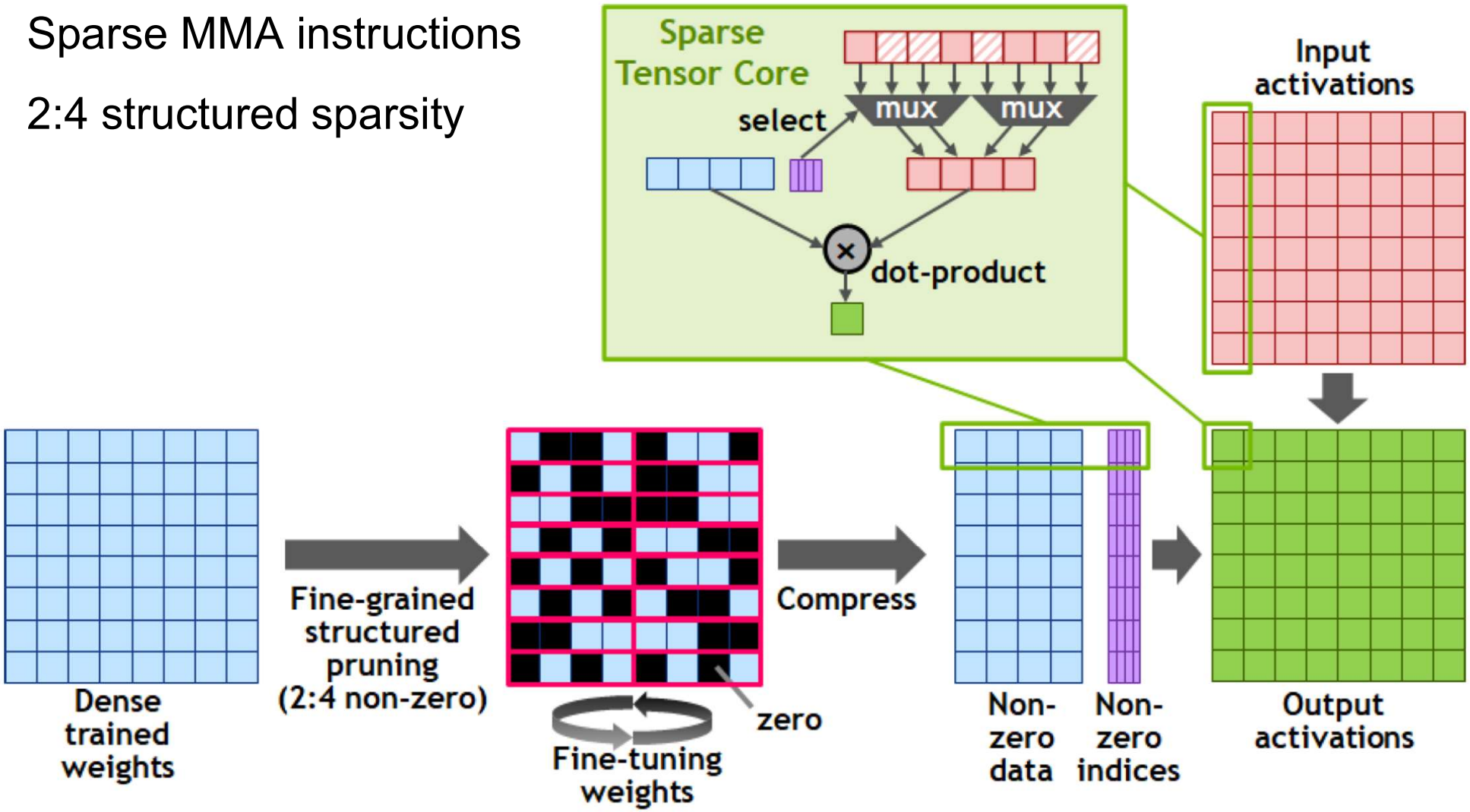
plus FP64 (new in Ampere; GA100 only)

plus INT4/INT8/binary data types (experimental; introduced in Turing)

Ampere Tensor Cores: Sparsity Support

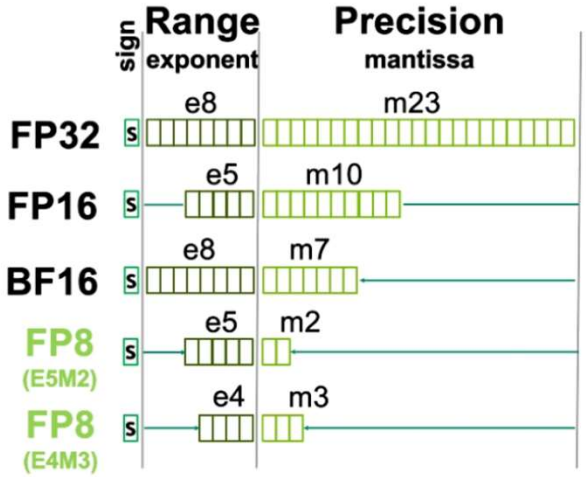
Sparse MMA instructions

2:4 structured sparsity

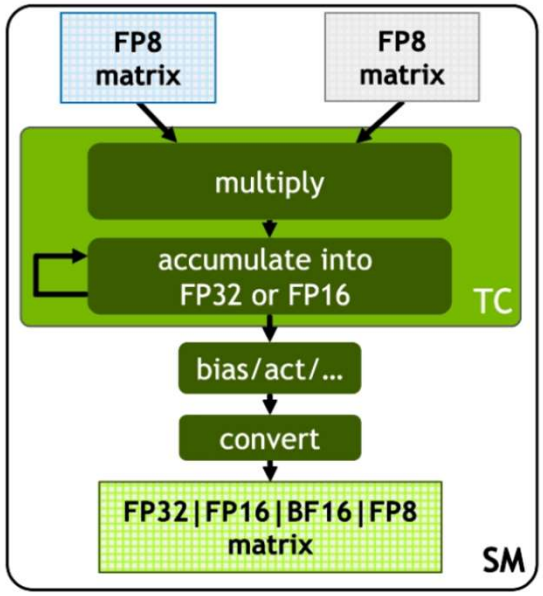


Tensor Cores: More Mixed Precision Options

New in Hopper: FP8



Allocate 1 bit to either range or precision



Support for multiple accumulator and output types

plus other data types from before (INT4/INT8/binary, …)

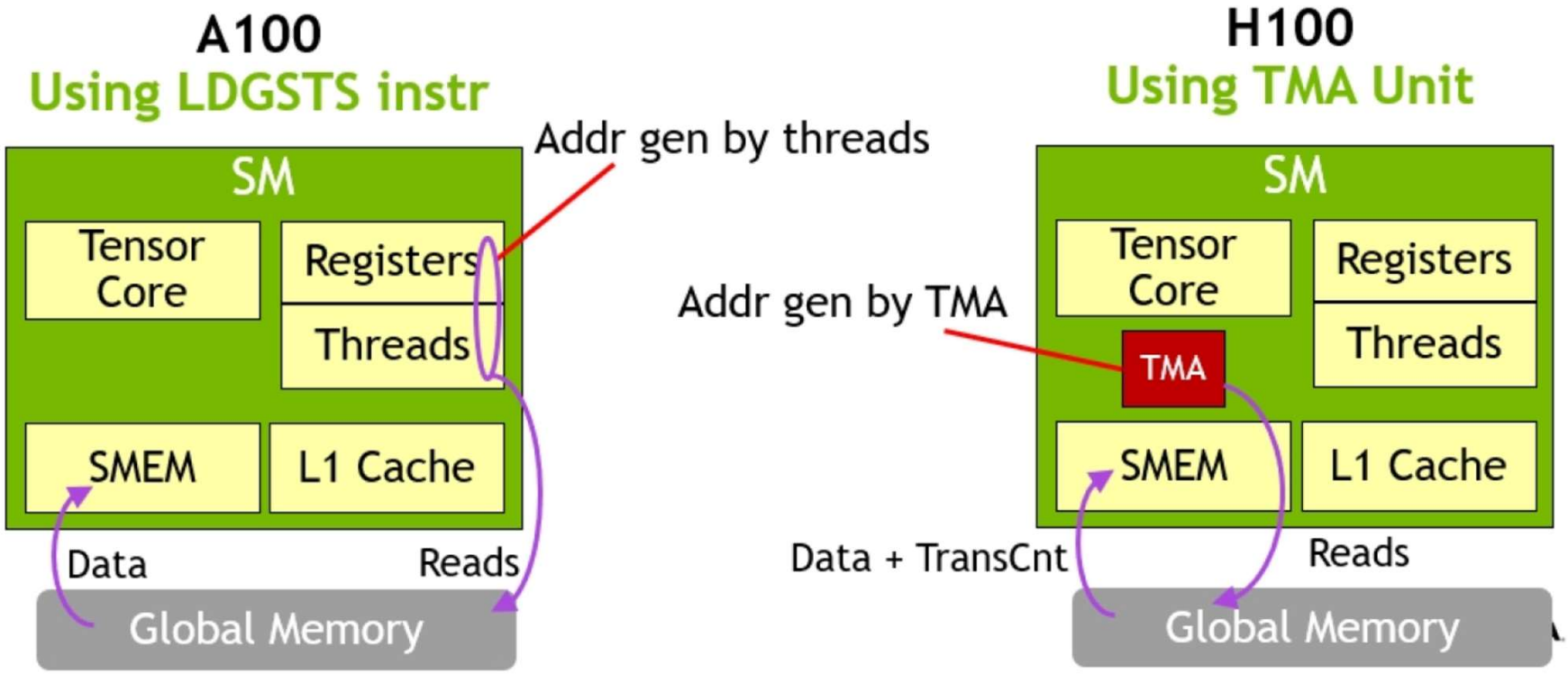
Tensor Cores: Hopper vs. Ampere

(preliminary)

	A100	A100 Sparse	H100 SXM5[1]	H100 SXM5[1] Sparse	H100 SXM5[1] Speedup vs A100
FP8 Tensor Core	NA	NA	2000 TFLOPS	4000 TFLOPS	6.4x vs A100 FP16
FP16	78 TFLOPS	NA	120 TFLOPS	NA	1.5x
FP16 Tensor Core	312 TFLOPS	624 TFLOPS	1000 TFLOPS	2000 TFLOPS	3.2x
BF16 Tensor Core	312 TFLOPS	624 TFLOPS	1000 TFLOPS	2000 TFLOPS	3.2x
FP32	19.5 TFLOPS	NA	60 TFLOPS	NA	3.1x
TF32 Tensor Core	156 TFLOPS	312 TFLOPS	500 TFLOPS	1000 TFLOPS	3.2x
FP64	9.7 TFLOPS	NA	30 TFLOPS	NA	3.1x
FP64 Tensor Core	19.5 TFLOPS	NA	60 TFLOPS	NA	3.1x
INT8 Tensor Core	624 TOPS	1248 TOPS	2000 TFLOPS	4000 TFLOPS	3.2x

Tensor Memory Accelerator (TMA)

Asynchronous transfers



Tensor Core APIs

Low-level options

- CUDA C WMMA (warp-level matrix multiply and accumulate)
- PTX wmma and mma (needed for some features) instructions
- SASS hmma instructions (not documented)

Intermediate-level options (template wrappers for low-level)

- NVIDIA CUTLASS (template abstractions for hi-perf matrix-multiplies)

CUTLASS 3.6.0 (October 2024)
`https://github.com/NVIDIA/cutlass`

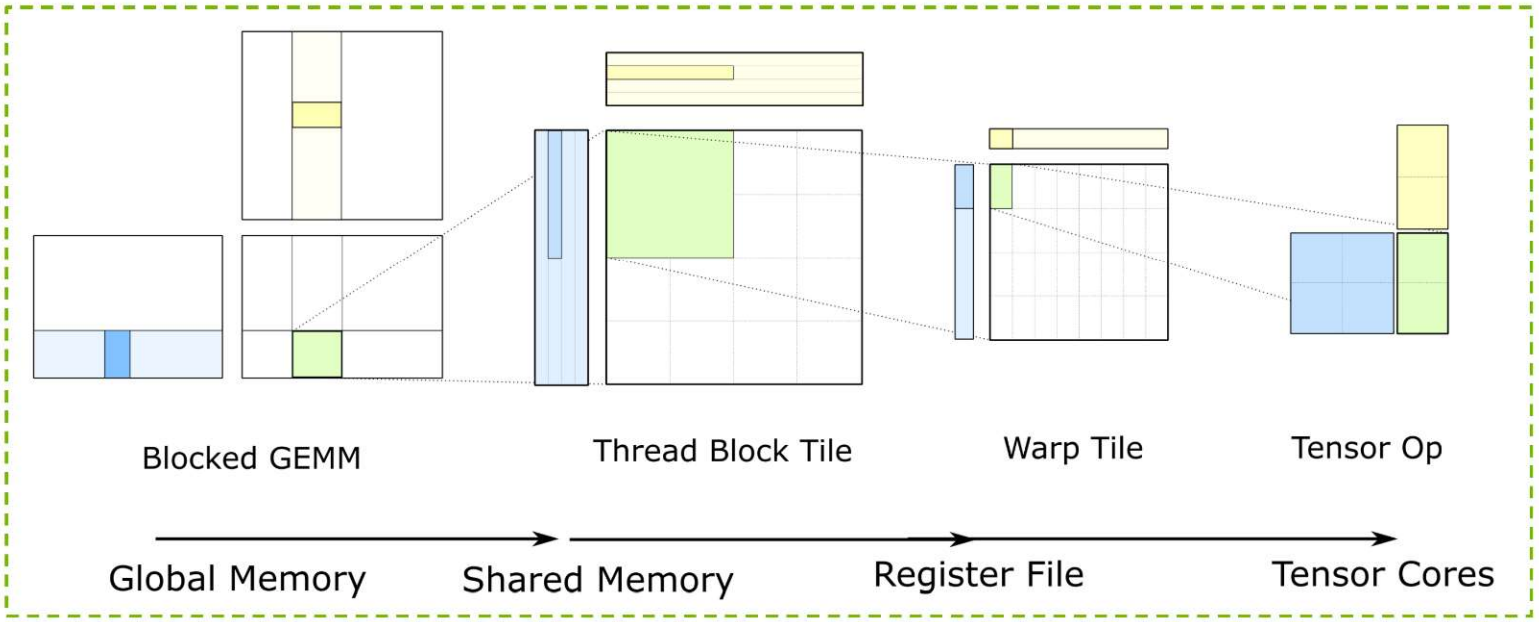
High-level options

- NVIDIA cuBLAS
- NVIDIA cuDNN
- Integration into TensorFlow, ...

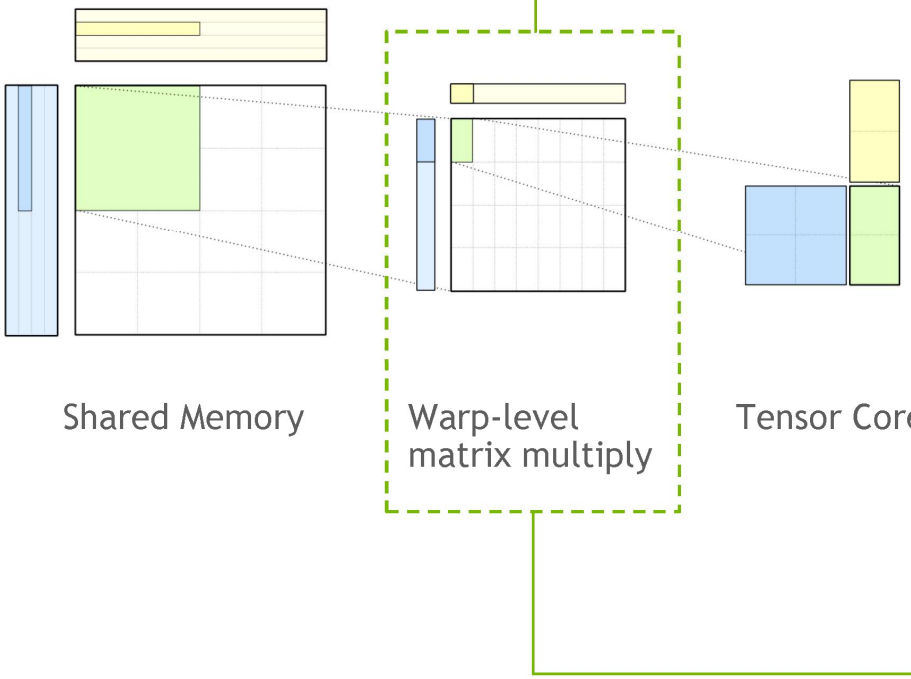
CUTLASS

CUDA C++ Templates as an Optimal Abstraction Layer for Tensor Cores

- Latency-tolerant pipeline from Global Memory
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES




```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
  GemmShape<64, 64, 16>,
  half_t, LayoutA,                  // GEMM A operand
  half_t, LayoutB,                  // GEMM B operand
  float, RowMajor                   // GEMM C operand
>;

__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];

// Construct iterators into SMEM tiles
Mma::IteratorA iter_A({smem_buffer_A, lda}, thread_id);
Mma::IteratorB iter_B({smem_buffer_B, ldb}, thread_id);

Mma::FragmentA frag_A;
Mma::FragmentB frag_B;
Mma::FragmentC accum;

Mma mma;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {


  iter_A.load(frag_A);  // Load fragments from A and B matrices
  iter_B.load(frag_B);

  ++iter_A; ++iter_B;   // Advance along GEMM K to next tile in A
                        //   and B matrices

                        // Compute matrix product
  mma(accum, frag_A, frag_B, accum);
}
```

CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES

Tile Iterator Constructors:

Initialize pointers into permuted Shared Memory buffers

Fragments:

Register-backed arrays holding each thread's data

Tile Iterator:

load() - Fetches data from permuted Shared Memory buffers

operator++() - advances to the next logical matrix in SMEM

Warp-level matrix multiply:

Decomposes a large matrix multiply into Tensor Core operations

```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
  GemmShape<64, 64, 16>,
  half_t, LayoutA,                  // GEMM A operand
  half_t, LayoutB,                  // GEMM B operand
  float, RowMajor                   // GEMM C operand
>;

__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];

// Construct iterators into SMEM tiles
Mma::IteratorA iter_A({smem_buffer_A, lda}, thread_id);
Mma::IteratorB iter_B({smem_buffer_B, ldb}, thread_id);

Mma::FragmentA frag_A;
Mma::FragmentB frag_B;
Mma::FragmentC accum;

Mma mma;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {


  iter_A.load(frag_A);  // Load fragments from A and B matrices
  iter_B.load(frag_B);

  ++iter_A; ++iter_B;   // Advance along GEMM K to next tile in A
                        //   and B matrices

                        // Compute matrix product
  mma(accum, frag_A, frag_B, accum);
}
```

CUDA C Warp Matrix Functions (WMMA)

Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C Programming Guide (11.8), Appendix B.24

namespace **`nvcuda::wmma`** (and **`nvcuda::wmma::experimental`**)

```
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);
```

Concept of a matrix *fragment* (section of a matrix split across threads in a warp)

Dimensions **`m,n,k`**: **`m x k matrix_a`**; **`k x n matrix_b`**; **`m x n accumulator`**

CUDA C Warp Matrix Functions (WMMA)

Data types (`T`)

Volta/Turing/Ampere/Hopper/Ada:

wmma API splits this into fragments

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16
signed char	signed char	int	8x32x16

CUDA C Warp Matrix Functions (WMMA)

Data types (`T`)

wmma API splits this into fragments

Alternate Floating Point support: **Ampere/Hopper/Ada only:**

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__nv_bfloat16	__nv_bfloat16	float	16x16x16
__nv_bfloat16	__nv_bfloat16	float	32x8x16
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Double Precision Support: **Ampere/Hopper only:**

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
double	double	double	8x8x4

Experimental support for sub-byte operations: **Turing/Ampere/Ada:**

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
precision::u4	precision::u4	int	8x8x32
precision::s4	precision::s4	int	8x8x32
precision::b1	precision::b1	int	8x8x128

CUDA C Warp Matrix Functions (WMMA)

Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C Programming Guide (12.6), Chapter 10.24 (online: 7.24)

```
#include <mma.h>

using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

PTX WMMA and MMA Instructions

PTX ISA 8.5, Section 9.7.13 (90 pages)

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
`wmma`	Dense	Floating-point - `.f16`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 6.0
`wmma`	Dense	Alternate floating-point format - `.bf16`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 7.0
`wmma`	Dense	Alternate floating-point format - `.tf32`	`.m16n16k8`	PTX ISA version 7.0
`wmma`	Dense	Integer - `.u8`/`.s8`	`.m16n16k16`, `.m8n32k16`, and `.m32n8k16`	PTX ISA version 6.3
`wmma`	Dense	Sub-byte integer - `.u4`/`.s4`	`.m8n8k32`	PTX ISA version 6.3 (preview feature)
`wmma`	Dense	Single-bit - `.b1`	`.m8n8k128`	PTX ISA version 6.3 (preview feature)

PTX WMMA and MMA Instructions

PTX ISA 8.5

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
mma	Dense	Floating-point - .f64	.m8n8k4	PTX ISA version 7.0
mma	Dense	Floating-point - .f16	.m8n8k4	PTX ISA version 6.4
			.m16n8k8	PTX ISA version 6.5
			.m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point format - .bf16	.m16n8k8 and .m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point format - .tf32	.m16n8k4 and .m16n8k8	PTX ISA version 7.0
mma	Dense	Integer - .u8/.s8	.m8n8k16	PTX ISA version 6.5
			.m16n8k16 and .m16n8k32	PTX ISA version 7.0
mma	Dense	Sub-byte integer - .u4/.s4	.m8n8k32	PTX ISA version 6.5
			.m16n8k32 and .m16n8k64	PTX ISA version 7.0
mma	Dense	Single-bit - .b1	.m8n8k128, .m16n8k128, and .m16n8k256	PTX ISA version 7.0
mma	Sparse	Floating-point - .f16	.m16n8k16 and .m16n8k32	PTX ISA version 7.1
mma	Sparse	Alternate floating-point format - .bf16	.m16n8k16 and .m16n8k32	PTX ISA version 7.1
mma	Sparse	Alternate floating-point format - .tf32	.m16n8k8 and .m16n8k16	PTX ISA version 7.1
mma	Sparse	Integer - .u8/.s8	.m16n8k32 and .m16n8k64	PTX ISA version 7.1
mma	Sparse	Sub-byte integer - .u4/.s4	.m16n8k64 and .m16n8k128	PTX ISA version 7.1

PTX WMMA and MMA Instructions

Load and store: wmma

wmma.load

Collectively load a matrix from memory for WMMA

Syntax

Floating point format `.f16` loads:

```
wmma.load.a.sync.aligned.layout.shape{.ss}.atype r, [p] {, stride};
wmma.load.b.sync.aligned.layout.shape{.ss}.btype r, [p] {, stride};
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride};

.layout = {.row, .col};
.shape  = {.m16n16k16, .m8n32k16, .m32n8k16};
.ss     = {.global, .shared};
.atype  = {.f16, .s8, .u8};
.btype  = {.f16, .s8, .u8};
.ctype  = {.f16, .f32, .s32};
```

Alternate floating point format `.bf16` loads:

```
wmma.load.a.sync.aligned.layout.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.layout.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m16n16k16, .m8n32k16, .m32n8k16};
.ss     = {.global, .shared};
.atype  = {.bf16 };
.btype  = {.bf16 };
.ctype  = {.f32 };
```

Alternate floating point format `.tf32` loads:

```
wmma.load.a.sync.aligned.layout.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.layout.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m16n16k8 };
.ss     = {.global, .shared};
.atype  = {.tf32 };
.btype  = {.tf32 };
.ctype  = {.f32 };
```

PTX WMMA and MMA Instructions

Load and store: wmma

wmma.load

Collectively load a matrix from memory for WMMA

Syntax

Double precision Floating point `.f64` loads:

```
wmma.load.a.sync.aligned.layout.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.layout.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m8n8k4 };
.ss     = {.global, .shared};
.atype  = {.f64 };
.btype  = {.f64 };
.ctype  = {.f64 };
```

Sub-byte loads:

```
wmma.load.a.sync.aligned.row.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.col.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m8n8k32};
.ss     = {.global, .shared};
.atype  = {.s4, .u4};
.btype  = {.s4, .u4};
.ctype  = {.s32};
```

Single-bit loads:

```
wmma.load.a.sync.aligned.row.shape{.ss}.atype r, [p] {, stride}
wmma.load.b.sync.aligned.col.shape{.ss}.btype r, [p] {, stride}
wmma.load.c.sync.aligned.layout.shape{.ss}.ctype r, [p] {, stride}
.layout = {.row, .col};
.shape  = {.m8n8k128};
.ss     = {.global, .shared};
.atype  = {.b1};
.btype  = {.b1};
.ctype  = {.s32};
```

PTX WMMA and MMA Instructions

wmma example

```
.global .align 32 .f16 A[256], B[256];
.global .align 32 .f32 C[256], D[256];
.reg .b32 a<8> b<8> c<8> d<8>;

wmma.load.a.sync.aligned.m16n16k16.global.row.f16
        {a0, a1, a2, a3, a4, a5, a6, a7}, [A];
wmma.load.b.sync.aligned.m16n16k16.global.col.f16
        {b0, b1, b2, b3, b4, b5, b6, b7}, [B];

wmma.load.c.sync.aligned.m16n16k16.global.row.f32
        {c0, c1, c2, c3, c4, c5, c6, c7}, [C];

wmma.mma.sync.aligned.m16n16k16.row.col.f32.f32
        {d0, d1, d2, d3, d4, d5, d6, d7},
        {a0, a1, a2, a3, a4, a5, a6, a7},
        {b0, b1, b2, b3, b4, b5, b6, b7},
        {c0, c1, c2, c3, c4, c5, c6, c7};

wmma.store.d.sync.aligned.m16n16k16.global.col.f32
        [D], {d0, d1, d2, d3, d4, d5, d6, d7};
```

PTX WMMA and MMA Instructions

mma: fixed assigments of matrix fragments to registers in each thread of warp

9.7.13.4.2. Matrix Fragments for mma.m8n8k4 with .f64 floating point type

A warp executing `mma.m8n8k4` with `.f64` floating point type will compute an MMA operation of shape `.m8n8k4`.

Elements of the matrix are distributed across the threads in a warp so each thread of the warp holds a fragment of the matrix.

- Multiplicand A:

.atype	Fragment	Elements (low to high)
`.f64`	A vector expression containing a single `.f64` register, containing single `.f64` element from the matrix A.	a0

Row\Col	0	1	2	3
0	T0:a0	T1:a0	T2:a0	T3:a0
1	T4:a0	T5:a0	T6:a0	T7:a0
2	→			
..	←			
7	T28:a0	T29:a0	T30:a0	T31:a0

%laneid:{fragments}

PTX WMMA and MMA Instructions

mma: fixed assigments of matrix fragments to registers in each thread of warp

9.7.13.4.1. Matrix Fragments for mma.m8n8k4 with .f16 floating point type

A warp executing `mma.m8n8k4` with .f16 floating point type will compute 4 MMA operations of shape `.m8n8k4`.

Elements of 4 matrices need to be distributed across the threads in a warp. The following table shows distribution of matrices for MMA operations.

MMA Computation	Threads participating in MMA computation
MMA computation 1	Threads with `%laneid` 0-3 (low group) and 16-19 (high group)
MMA computation 2	Threads with `%laneid` 4-7 (low group) and 20-23 (high group)
MMA computation 3	Threads with `%laneid` 8-11 (low group) and 24-27 (high group)
MMA computation 4	Threads with `%laneid` 12-15 (low group) and 28-31 (high group)

- Multiplicand A:

.atype	Fragment	Elements (low to high)
`.f16`	A vector expression containing two `.f16x2` registers, with each register containing two `.f16` elements from the matrix A.	a0, a1, a2, a3

- Row Major:

MMA computation 1

Row\Col	0	1	2	3
0	T0 : { a0, a1, a2, a3 }			
..	↓			
3	T3: { a0, a1, a2, a3 }			
4	T16: { a0, a1, a2, a3 }			
..	↓			
7	T19: { a0, a1, a2, a3 }			

MMA computation 3

Row\Col	0	1	2	3
0	T8 : { a0, a1, a2, a3}			
..	↓			
3	T11: { a0, a1, a2, a3 }			
4	T24: { a0, a1, a2, a3 }			
..	↓			
7	T27: { a0, a1, a2, a3 }			

MMA computation 2

Row\Col	0	1	2	3
0	T4 : { a0, a1, a2, a3}			
..	↓			
3	T7: { a0, a1, a2, a3 }			
4	T20: { a0, a1, a2, a3 }			
..	↓			
7	T23: { a0, a1, a2, a3 }			

MMA computation 4

Row\Col	0	1	2	3
0	T12 : { a0, a1, a2, a3}			
..	↓			
3	T15: { a0, a1, a2, a3 }			
4	T28: { a0, a1, a2, a3 }			
..	↓			
7	T31: { a0, a1, a2, a3 }			

%laneid:{fragments}

PTX WMMA and MMA Instructions

mma: fixed assignments of matrix fragments to registers in each thread of warp

9.7.13.4.1. Matrix Fragments for mma.m8n8k4 with .f16 floating point type

A warp executing `mma.m8n8k4` with .f16 floating point type will compute 4 MMA operations of shape `.m8n8k4`.

Elements of 4 matrices need to be distributed across the threads in a warp. The following table shows distribution of matrices for MMA operations.

MMA Computation	Threads participating in MMA computation
MMA computation 1	Threads with `%laneid` 0-3 (low group) and 16-19 (high group)
MMA computation 2	Threads with `%laneid` 4-7 (low group) and 20-23 (high group)
MMA computation 3	Threads with `%laneid` 8-11 (low group) and 24-27 (high group)
MMA computation 4	Threads with `%laneid` 12-15 (low group) and 28-31 (high group)

- Accumulators C (or D):

.ctype / .dtype	Fragment	Elements (low to high)
`.f16`	A vector expression containing four `.f16x2` registers, with each register containing two `.f16` elements from the matrix C (or D).	c0, c1, c2, c3, c4, c5, c6, c7
`.f32`	A vector expression of eight `.f32` registers.	

- `.ctype` is `.f16`

MMA computation 1

Row\Col	0	1	2	3	4	5	6	7
0	T0 : { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
3	T3: { c0, c1, c2, c3, c4, c5, c6, c7 }							
4	T16: { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
7	T19: { c0, c1, c2, c3, c4, c5, c6, c7 }							

MMA computation 3

Row\Col	0	1	2	3	4	5	6	7
0	T8 : { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
3	T11: { c0, c1, c2, c3, c4, c5, c6, c7 }							
4	T24: { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
7	T27: { c0, c1, c2, c3, c4, c5, c6, c7 }							

MMA computation 2

Row\Col	0	1	2	3	4	5	6	7
0	T4 : { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
3	T7: { c0, c1, c2, c3, c4, c5, c6, c7 }							
4	T20: { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
7	T23: { c0, c1, c2, c3, c4, c5, c6, c7 }							

MMA computation 4

Row\Col	0	1	2	3	4	5	6	7
0	T12 : { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
3	T15: { c0, c1, c2, c3, c4, c5, c6, c7 }							
4	T28: { c0, c1, c2, c3, c4, c5, c6, c7 }							
..								
7	T31: { c0, c1, c2, c3, c4, c5, c6, c7 }							

- `.ctype` is `.f32`

MMA computation 1

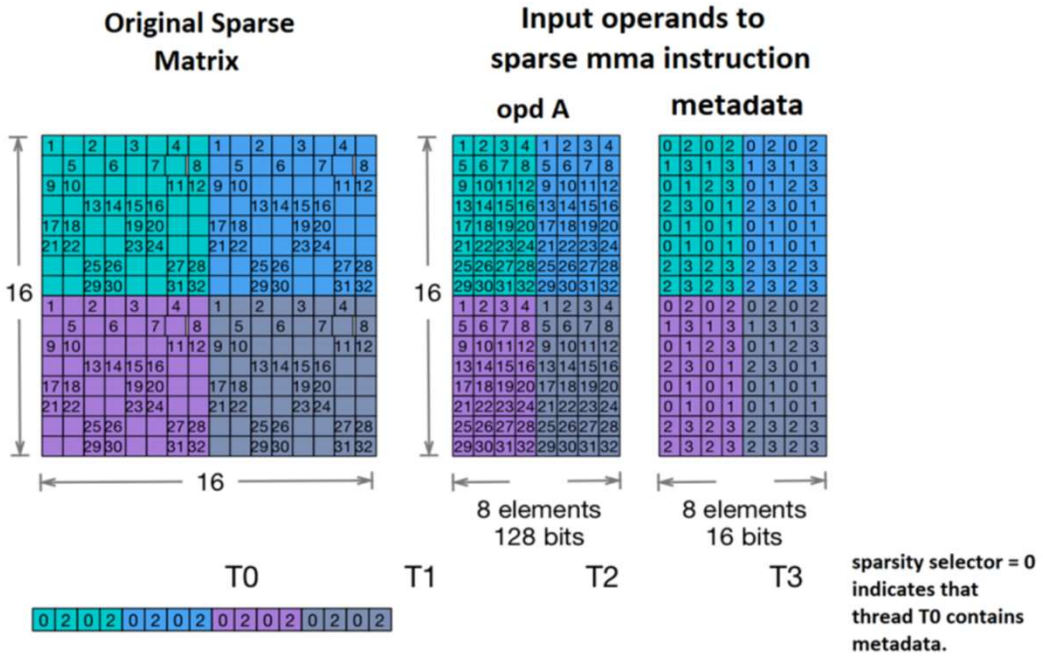
R\C	0	1	2	3	4	5	6	7
0	T0 : { c0, c1 }		T2 : { c0, c1 }		T0 : { c4, c5 }		T2 : { c4, c5 }	
1	T1 : { c0, c1 }		T3 : { c0 , c1 }		T1 : { c4, c5 }		T3 : { c4 , c5 }	
2	T0 : { c2, c3 }		T2 : { c2, c3 }		T0 : { c6, c7 }		T2 : { c6, c7 }	
3	T1 : { c2, c3 }		T3 : { c2 , c3 }		T1 : { c6, c7 }		T3 : { c6 , c7 }	
4	T16 : { c0, c1 }		T18 : { c0, c1 }		T16 : { c4, c5 }		T18 : { c4, c5 }	
5	T17 : { c0, c1 }		T19 : { c0 , c1 }		T17 : { c4, c5 }		T19 : { c4 , c5 }	
6	T16 : { c2, c3 }		T18 : { c2, c3 }		T16 : { c6, c7 }		T18 : { c6, c7 }	
7	T17 : { c2, c3 }		T19 : { c2 , c3 }		T17 : { c6, c7 }		T19 : { c6 , c7 }	

PTX WMMA and MMA Instructions

Sparse matrices: mma.sp

9.7.13.5. Matrix multiply-accumulate operation using `mma.sp` instruction with sparse matrix A

This section describes warp-level `mma.sp` instruction with sparse matrix A. This variant of the `mma` operation can be used when A is a structured sparse matrix with 50% zeros in each row distributed in a shape-specific granularity. For an `MxNxK` sparse `mma.sp` operation, the `MxK` matrix A is packed into `MxK/2` elements. For each K-wide row of matrix A, 50% elements are zeros and the remaining K/2 non-zero elements are packed in the operand representing matrix A. The mapping of these K/2 elements to the corresponding K-wide row is provided explicitly as metadata.



PTX WMMA and MMA Instructions

Load and store: mma ldmatrix

Warp-wide load matrix instruction

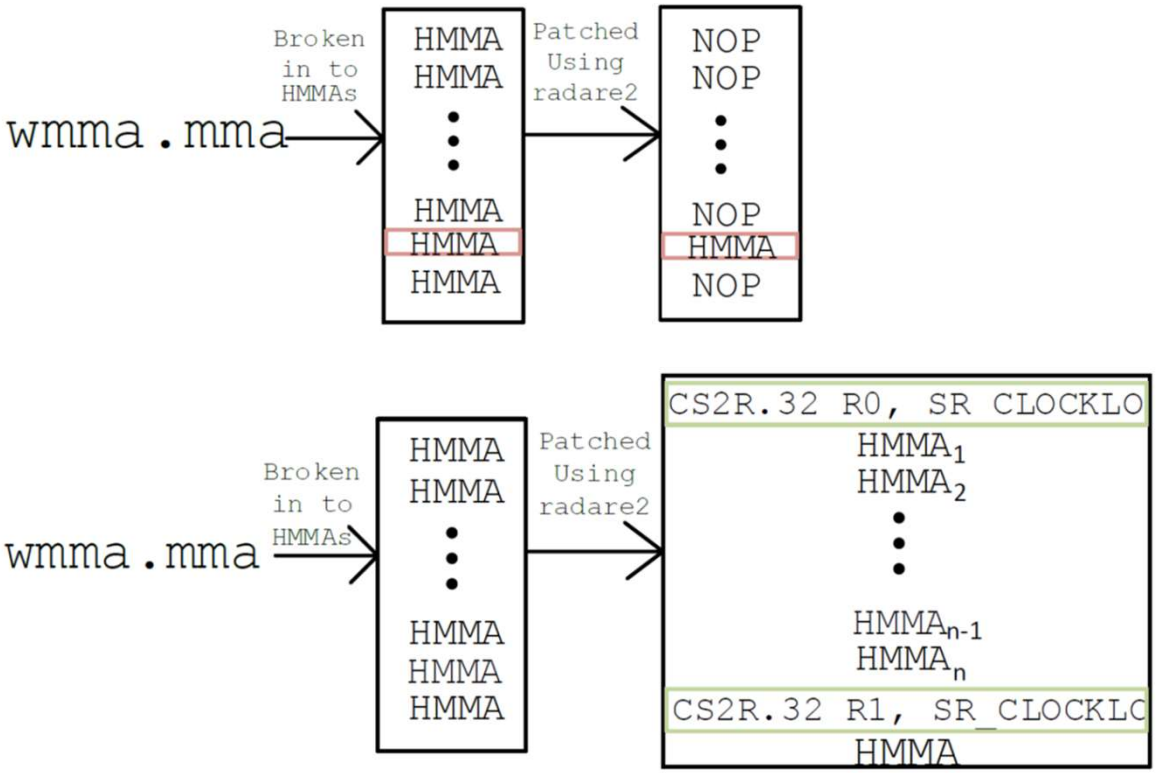
```
// Load a single 8x8 matrix using 64-bit addressing
.reg .b64 addr;
.reg .b32 d;
ldmatrix.sync.aligned.m8n8.x1.shared.b16 {d}, [addr];

// Load two 8x8 matrices in column-major format
.reg .b64 addr;
.reg .b32 d<2>;
ldmatrix.sync.aligned.m8n8.x2.trans.shared.b16 {d0, d1}, [addr];

// Load four 8x8 matrices
.reg .b64 addr;
.reg .b32 d<4>;
ldmatrix.sync.aligned.m8n8.x4.b16 {d0, d1, d2, d3}, [addr];
```

PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

Micro-benchmarking



PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

		Cumulative Clock Cycles
SET1	HMMA.884.F32.F32.STEP0 R8, R24.reuse.COL, R22.reuse.ROW, R8;	10
	HMMA.884.F32.F32.STEP1 R10, R24.reuse.COL, R22.reuse.ROW, R10;	12
	HMMA.884.F32.F32.STEP2 R4, R24.reuse.COL, R22.reuse.ROW, R4;	14
	HMMA.884.F32.F32.STEP3 R6, R24.COL, R22.ROW, R6;	18
SET2	HMMA.884.F32.F32.STEP0 R8, R20.reuse.COL, R18.reuse.ROW, R8;	20
	HMMA.884.F32.F32.STEP1 R10, R20.reuse.COL, R18.reuse.ROW, R10;	22
	HMMA.884.F32.F32.STEP2 R4, R20.reuse.COL, R18.reuse.ROW, R4;	24
	HMMA.884.F32.F32.STEP3 R6, R20.COL, R18.ROW, R6;	28
SET3	HMMA.884.F32.F32.STEP0 R8, R14.reuse.COL, R12.reuse.ROW, R8;	30
	HMMA.884.F32.F32.STEP1 R10, R14.reuse.COL, R12.reuse.ROW, R10;	32
	HMMA.884.F32.F32.STEP2 R4, R14.reuse.COL, R12.reuse.ROW, R4;	34
	HMMA.884.F32.F32.STEP3 R6, R14.COL, R12.ROW, R6;	38
SET4	HMMA.884.F32.F32.STEP0 R8, R16.reuse.COL, R2.reuse.ROW, R8;	40
	HMMA.884.F32.F32.STEP1 R10, R16.reuse.COL, R2.reuse.ROW, R10;	42
	HMMA.884.F32.F32.STEP2 R4, R16.reuse.COL, R2.reuse.ROW, R4;	44
	HMMA.884.F32.F32.STEP3 R6, R16.COL, R2.ROW, R6;	54

(a) Disassembled SASS instructions for Mixed precision mode

PTX WMMA to SASS

Raihan et al., 2019

Get SASS code from cuobjdump disassembly

		Cumulative Clock Cycles
SET1	HMMA.884.F16.F16.STEP0 R4, R22.reuse.T, R12.reuse.T, R4;	12
	HMMA.884.F16.F16.STEP1 R6, R22.T, R12.T, R6;	21
SET2	HMMA.884.F16.F16.STEP0 R4, R16.reuse.T, R14.reuse.T, R4;	25
	HMMA.884.F16.F16.STEP1 R6, R16.T, R14.T, R6;	34
SET3	HMMA.884.F16.F16.STEP0 R4, R18.reuse.T, R8.reuse.T, R4;	38
	HMMA.884.F16.F16.STEP1 R6, R18.T, R8.T, R6;	47
SET4	HMMA.884.F16.F16.STEP0 R4, R2.reuse.T, R10.reuse.T, R4;	51
	HMMA.884.F16.F16.STEP1 R6, R2.T, R10.T, R6;	64

(b) Disassembled SASS instructions for FP16 mode

PTX WMMA to SASS

Raihan et al., 2019, reverse-engineered matrix fragment assignment

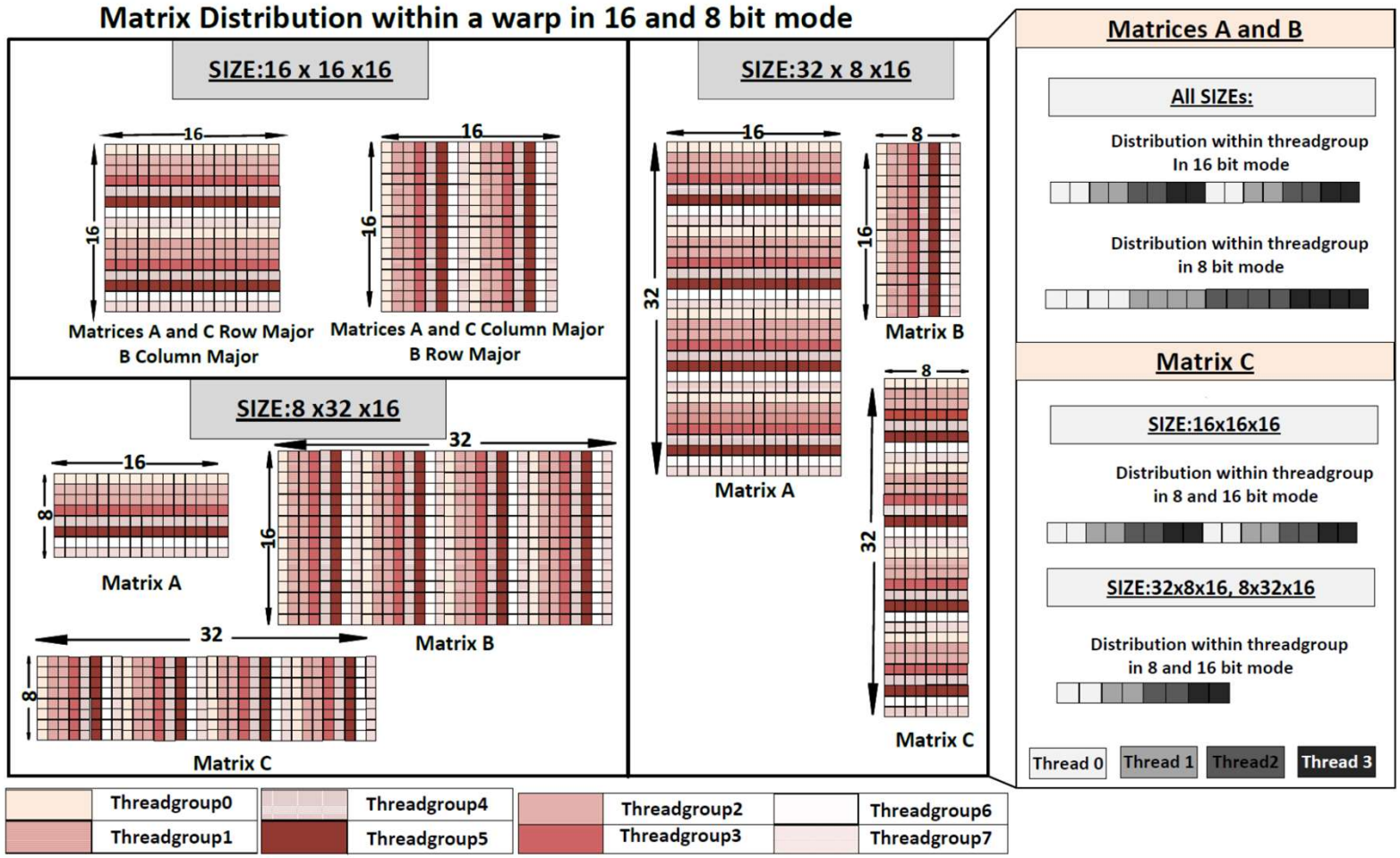


Figure 8: Distribution of operand matrix elements to threads for tensor cores in the RTX 2080 (Turing).

PTX WMMA to SASS

Raihan et al., 2019, reverse-engineered Tensor core microarchitecture

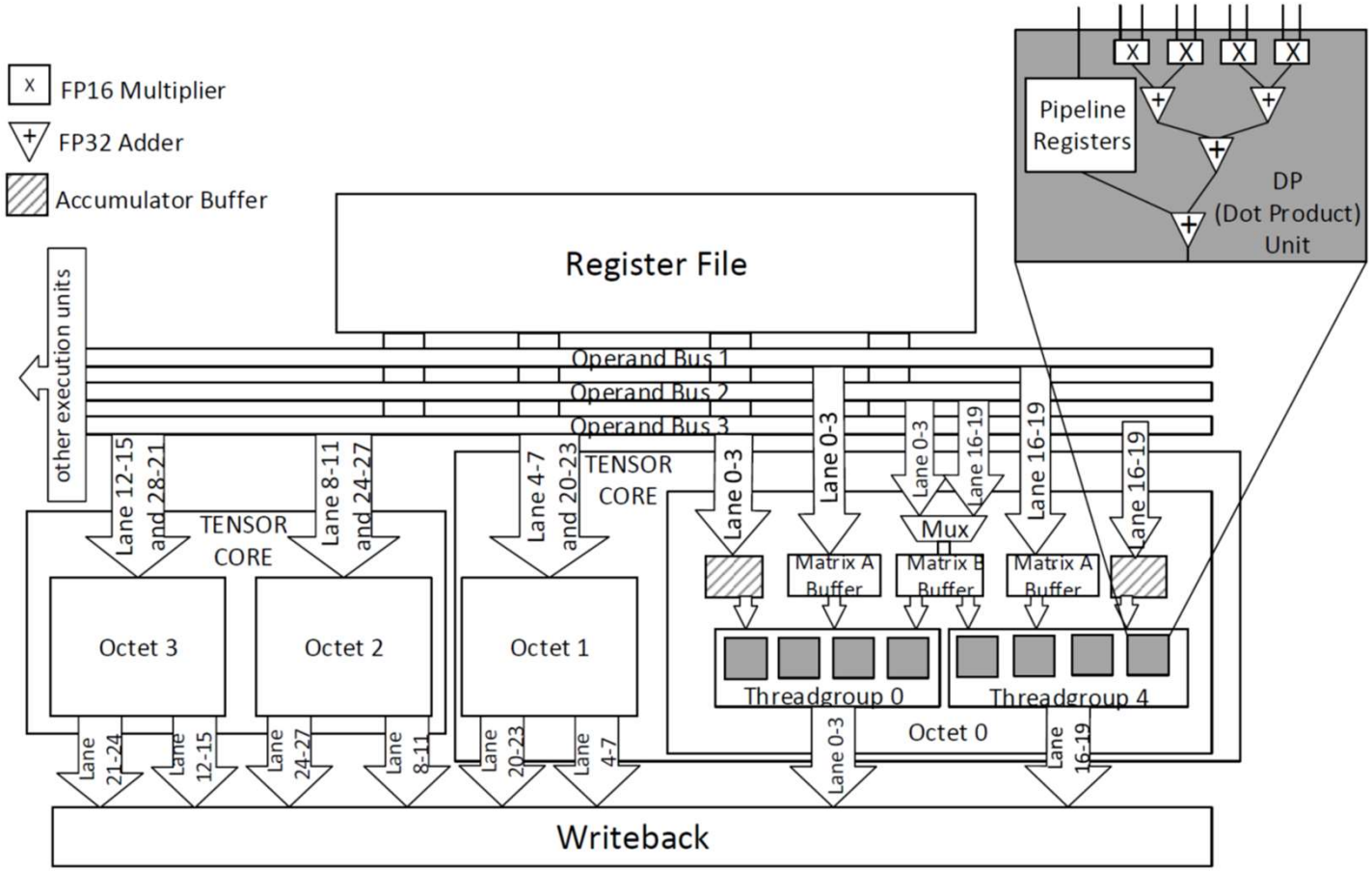
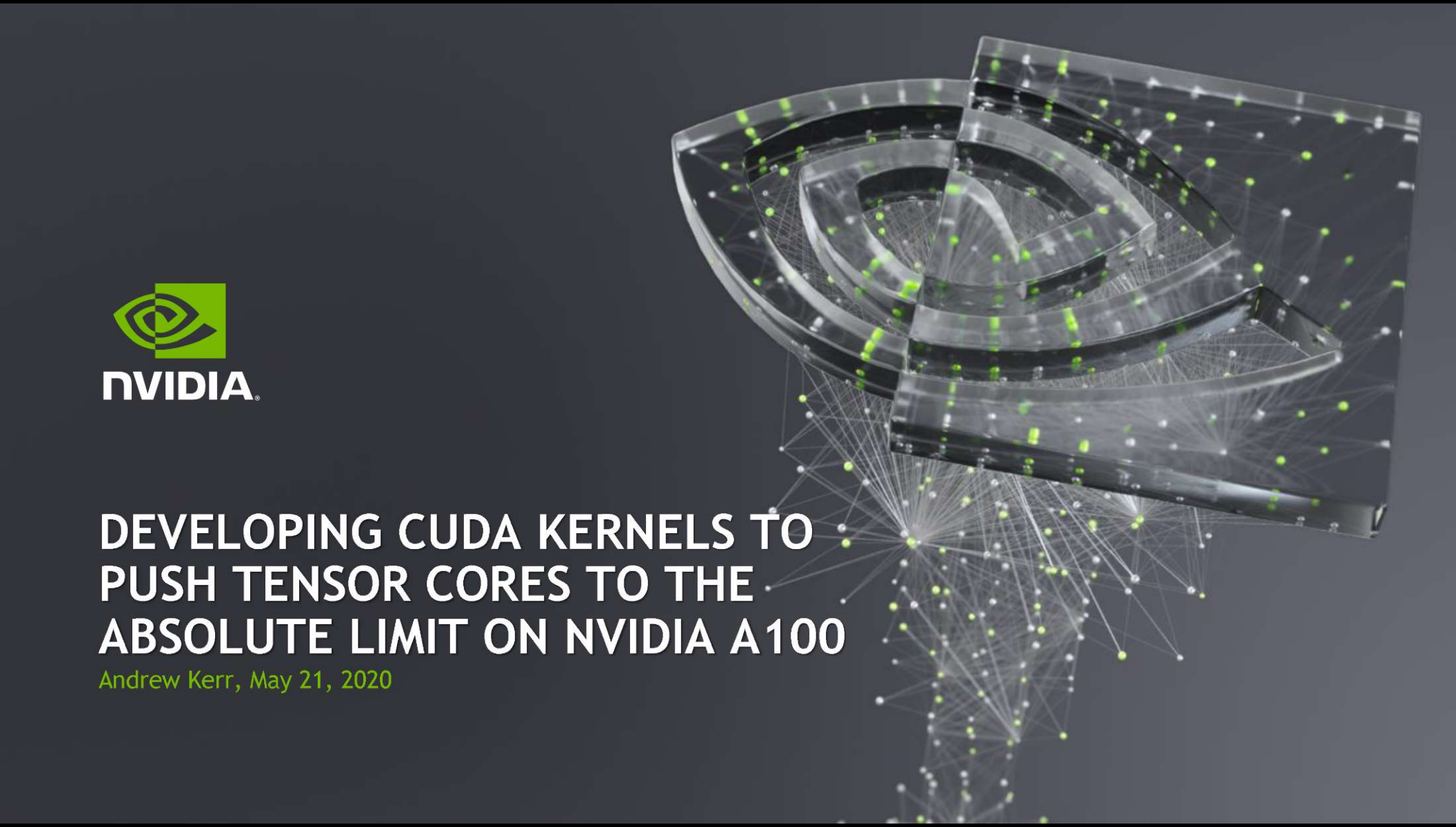


Figure 13: Proposed Tensor Core Microarchitecture



DEVELOPING CUDA KERNELS TO PUSH TENSOR CORES TO THE ABSOLUTE LIMIT ON NVIDIA A100

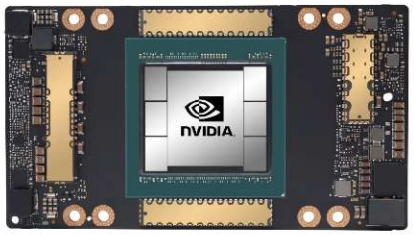
Andrew Kerr, May 21, 2020

https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21745-developing-cuda-kernels-to-push-tensor-cores-to-the-absolute-limit-on-nvidia-a100.pdf

NVIDIA AMPERE ARCHITECTURE

NVIDIA A100

New and Faster Tensor Core Operations



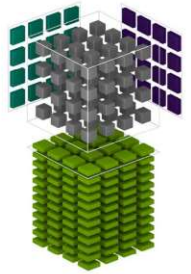
- Floating-point Tensor Core operations **8x** and **16x** faster than F32 CUDA Cores
- Integer Tensor Core operations **32x** and **64x** faster than F32 CUDA Cores
- New IEEE double-precision Tensor Cores **2x** faster than F64 CUDA Cores

Additional Data Types and Mode

- Bfloat16, double, Tensor Float 32

Asynchronous copy

- Copy directly into shared memory – deep software pipelines



Many additional new features – see “Inside NVIDIA Ampere Architecture”

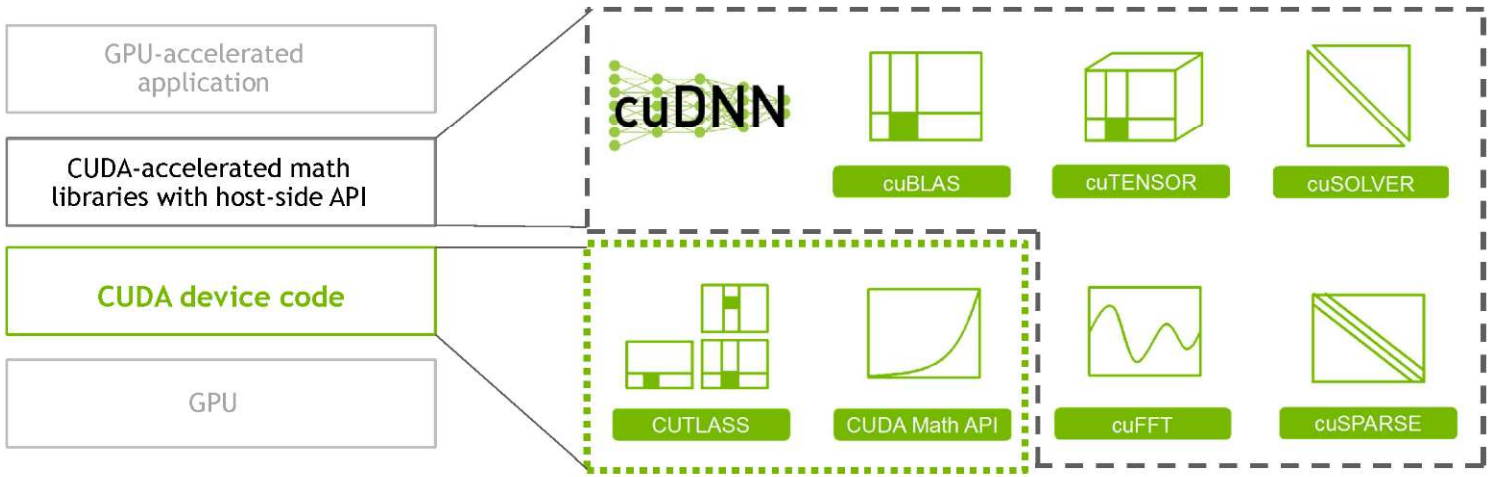
PROGRAMMING NVIDIA AMPERE ARCHITECTURE

Deep Learning and Math Libraries using Tensor Cores (with CUDA kernels under the hood)

- cuDNN, cuBLAS, cuTENSOR, cuSOLVER, cuFFT, cuSPARSE
- "CUDNN V8: New Advances in Deep Learning Acceleration" (GTC 2020 - S21685)
- "How CUDA Math Libraries Can Help you Unleash the Power of the New NVIDIA A100 GPU" (GTC 2020 - S21681)
- "Inside the Compilers, Libraries and Tools for Accelerated Computing" (GTC 2020 - S21766)

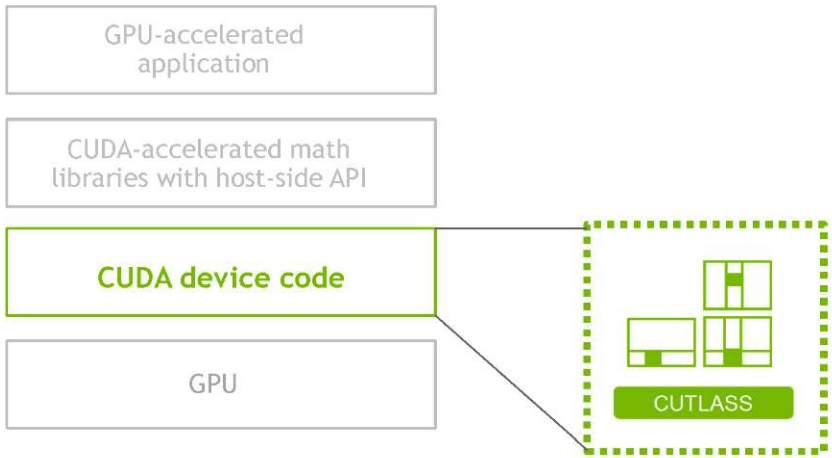
CUDA C++ Device Code

- CUTLASS, CUDA Math API, CUB, Thrust, libcu++



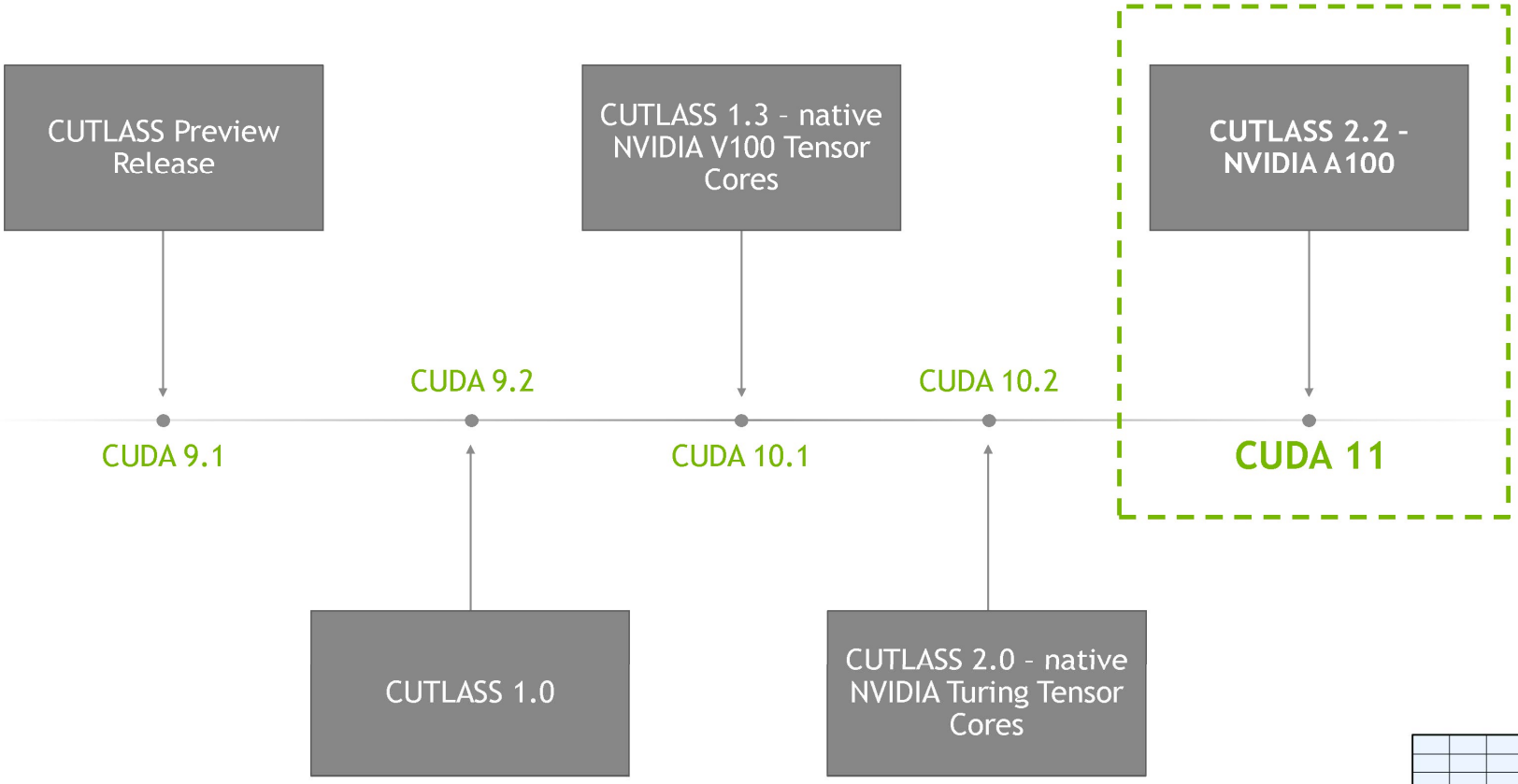
PROGRAMMING NVIDIA AMPERE ARCHITECTURE with CUDA C++

This is a talk for CUDA programmers

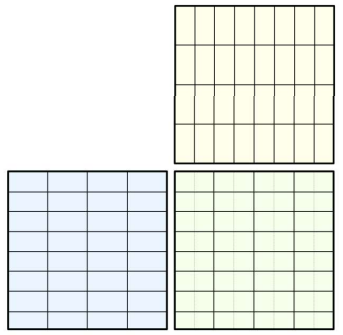


CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



https://github.com/NVIDIA/cutlass



CUTLASS

What's new?

CUTLASS 2.2: optimal performance on NVIDIA Ampere Architecture

- Higher throughput Tensor Cores: more than 2x speedup for all data types
- New floating-point types: bfloat16, Tensor Float 32, double
- Deep software pipelines with cp.async: efficient and latency tolerant

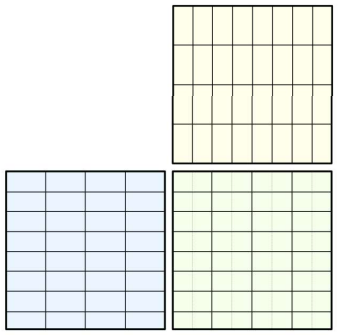
CUTLASS 2.1

- Planar complex: complex-valued GEMMs with batching options targeting Volta and Turing Tensor Cores
- BLAS-style host side API

CUTLASS 2.0: significant refactoring using modern C++11 programming

- Efficient: particularly for Turing Tensor Cores
- Tensor Core programming model: reusable components for linear algebra kernels in CUDA
- Documentation, profiling tools, reference implementations, SDK examples, more..

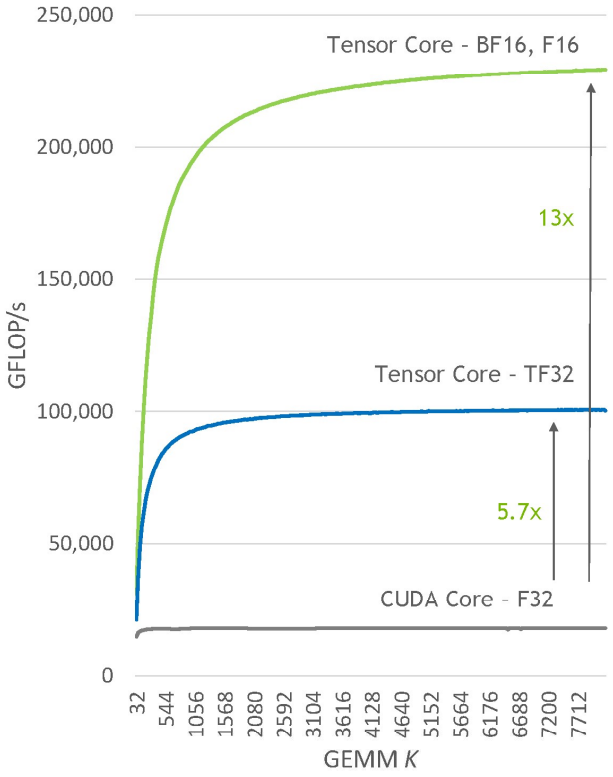
https://github.com/NVIDIA/cutlass



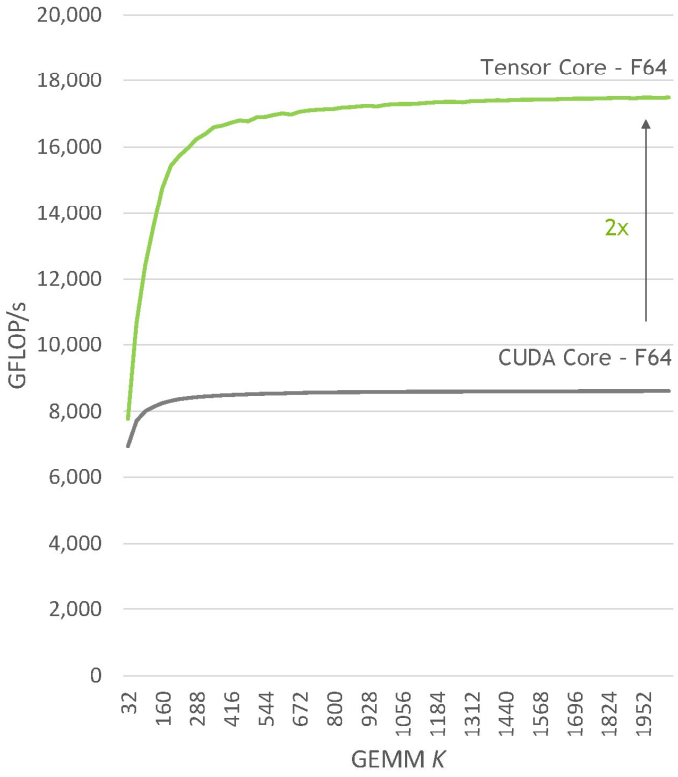
CUTLASS PERFORMANCE ON NVIDIA AMPERE ARCHITECTURE

CUTLASS 2.2 - CUDA 11 Toolkit – NVIDIA A100

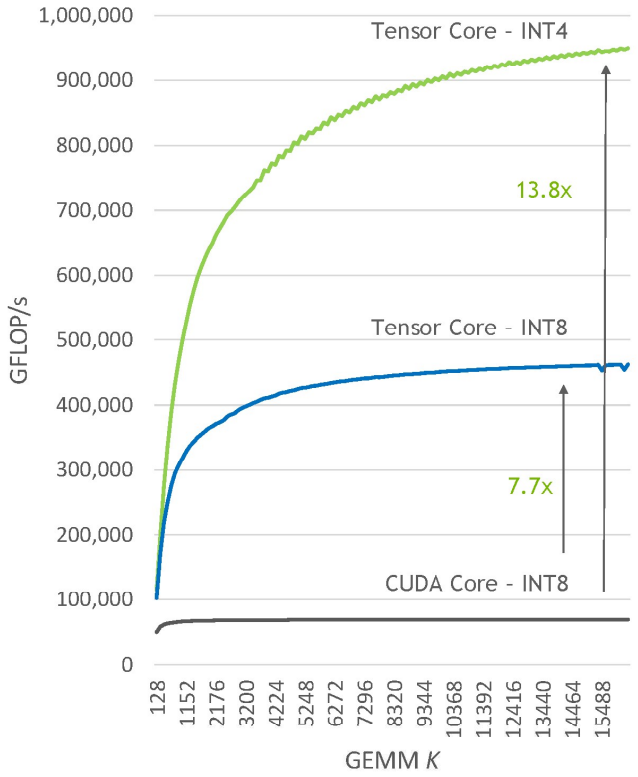
Mixed Precision Floating Point



Double Precision Floating Point

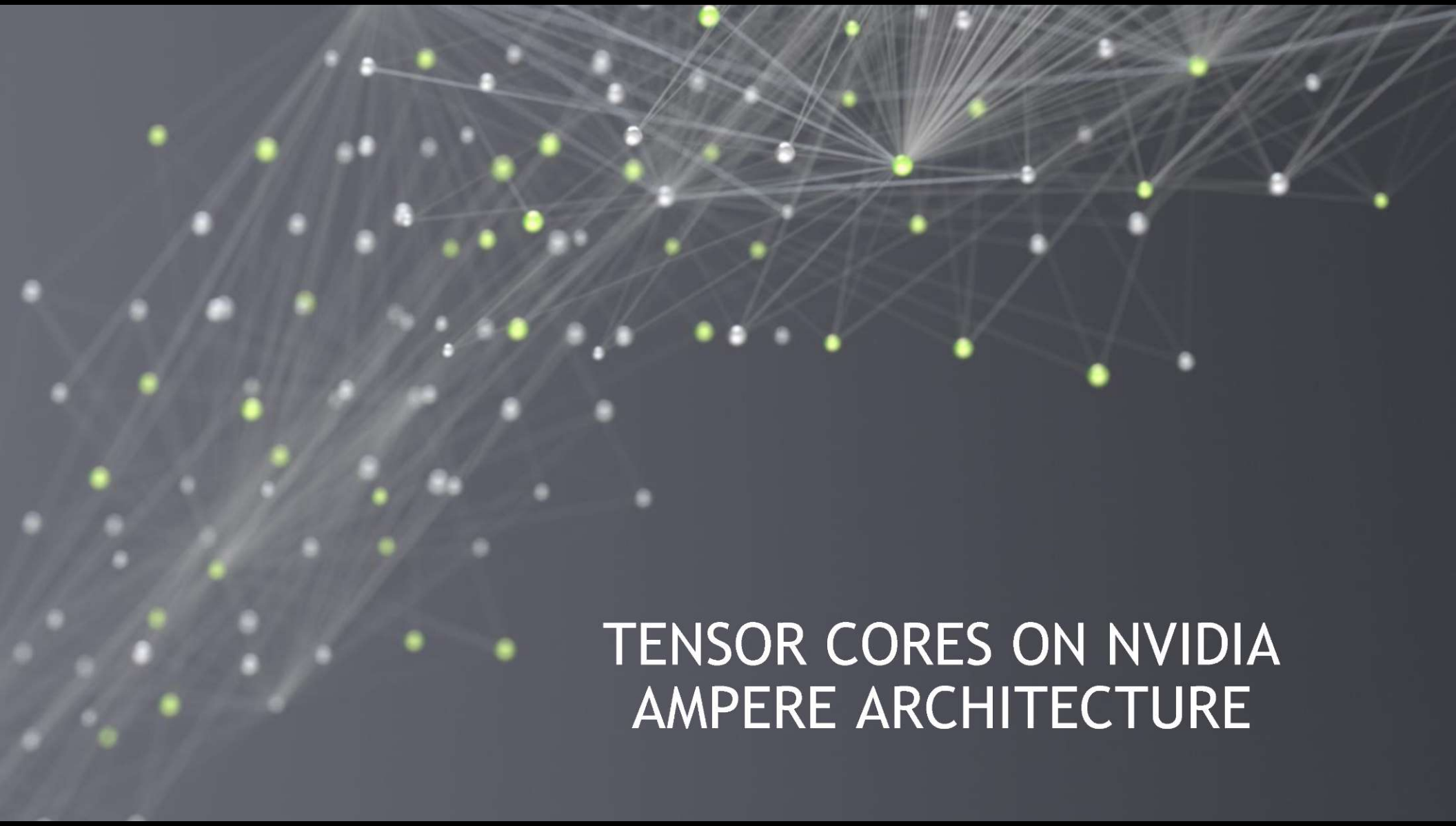


Mixed Precision Integer



m=3456, n=4096

NVIDIA


TENSOR CORES ON NVIDIA
AMPERE ARCHITECTURE

WHAT ARE TENSOR CORES?

Matrix operations: D = op(A, B) + C

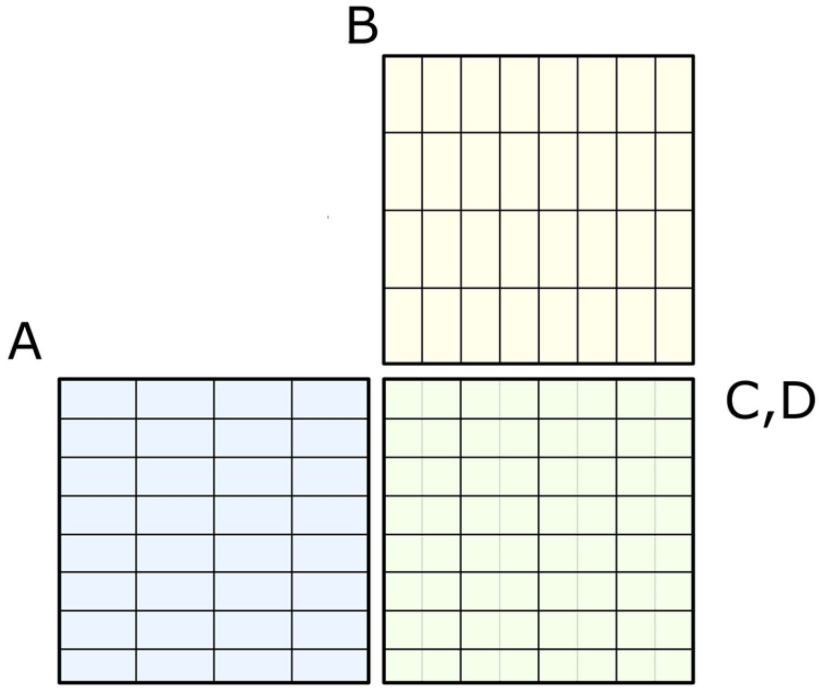
- Matrix multiply-add
- XOR-POPC

Input Data types: A, B

- half, bfloat16, Tensor Float 32, double, int8, int4, bin1

Accumulation Data Types: C, D

- half, float, int32_t, double



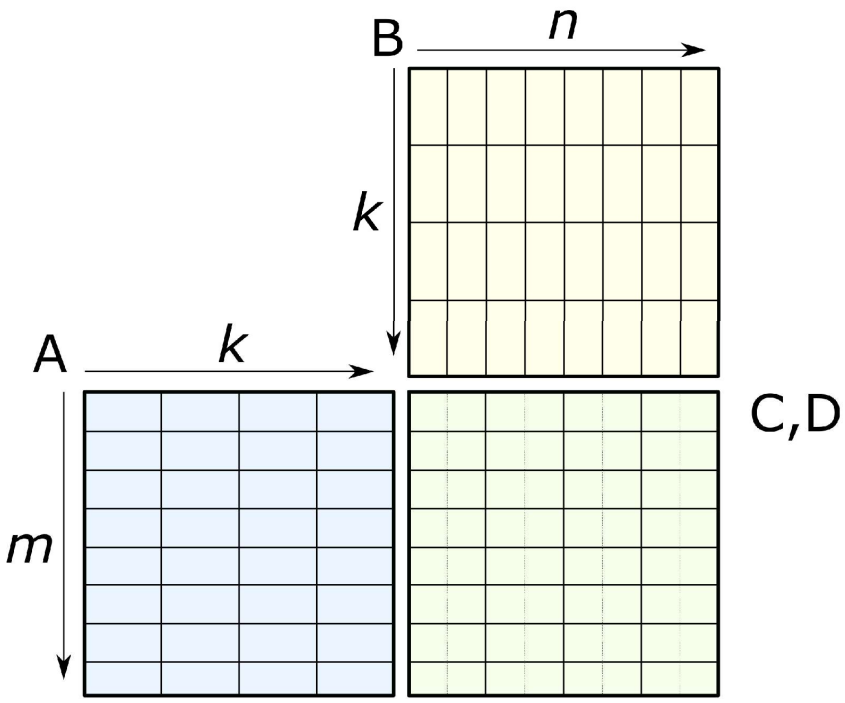
WHAT ARE TENSOR CORES?

Matrix operations: D = op(A, B) + C

- Matrix multiply-add
- XOR-POPC

M-by-*N*-by-*K* matrix operation

- Warp-synchronous, collective operation
- 32 threads within warp collectively hold A, B, C, and D operands



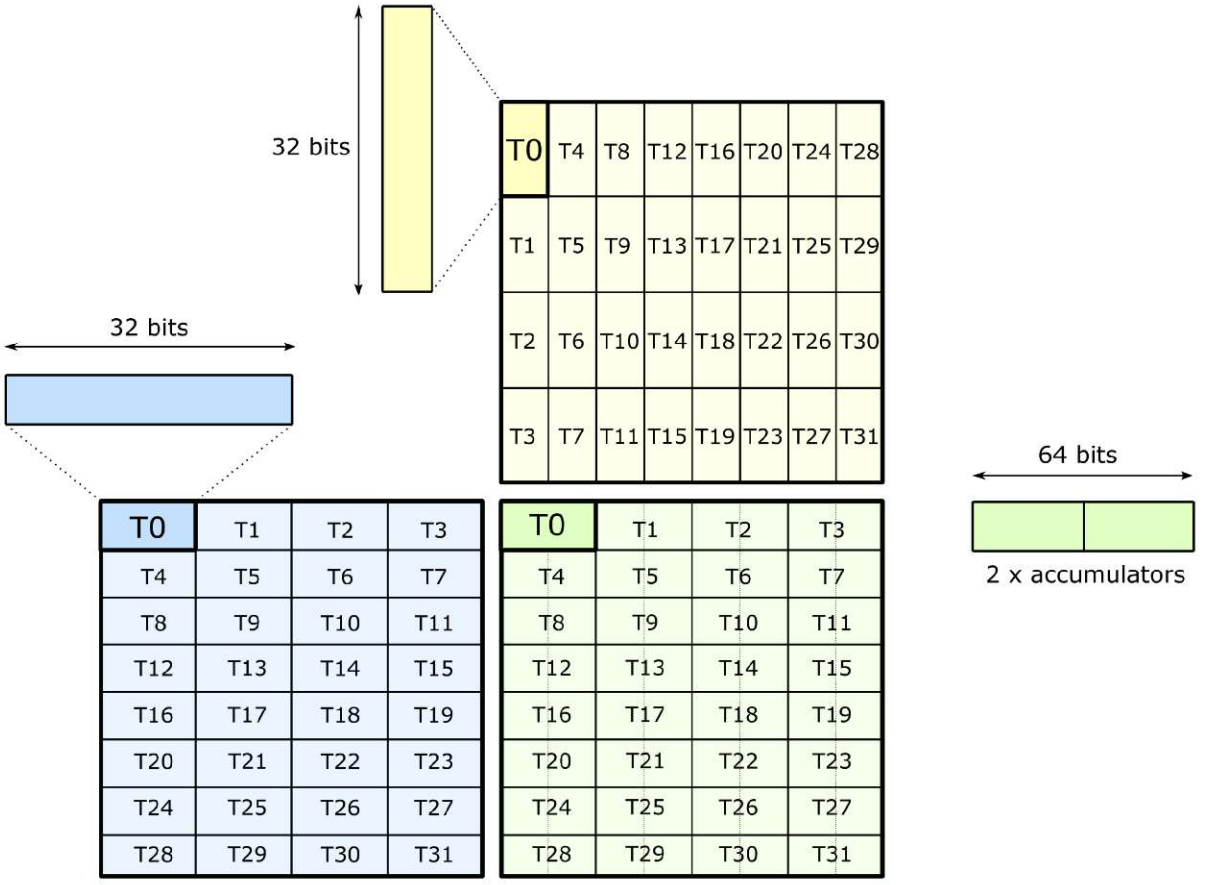
NVIDIA AMPERE ARCHITECTURE - TENSOR CORE OPERATIONS

PTX	Data Types (A * B + C)	Shape	Speedup on NVIDIA A100 (vs F32 CUDA cores)	Speedup on Turing* (vs F32 Cores)	Speedup on Volta* (vs F32 Cores)
mma.sync.m16n8k16 mma.sync.m16n8k8	F16 * F16 + F16 F16 * F16 + F32 BF16 * BF16 + F32	16-by-8-by-16 16-by-8-by-8	16x	8x	8x
mma.sync.m16n8k8	TF32 * TF32 + F32	16-by-8-by-8	8x	N/A	N/A
mma.sync.m8n8k4	F64 * F64 + F64	8-by-8-by-4	2x	N/A	N/A
mma.sync.m16n8k32 mma.sync.m8n8k16	S8 * S8 + S32	16-by-8-by-32 8-by-8-by-16	32x	16x	N/A
mma.sync.m16n8k64	S4 * S4 + S32	16-by-8-by-64	64x	32x	N/A
mma.sync.m16n8k256	B1 ^ B1 + S32	16-by-8-by-256	256x	128x	N/A

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

* Instructions with equivalent functionality for Turing and Volta differ in shape from the NVIDIA Ampere Architecture in several cases.

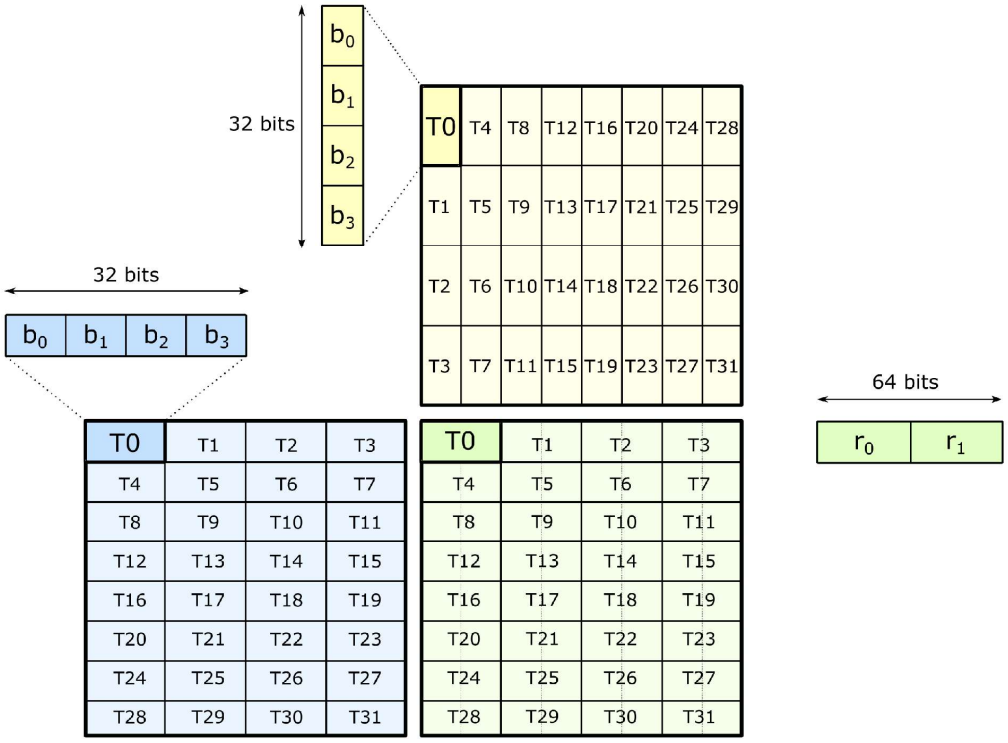
TENSOR CORE OPERATION: FUNDAMENTAL SHAPE



Warp-wide Tensor Core operation: 8-by-8-by-128b

S8 * S8 + S32

8-by-8-by-16

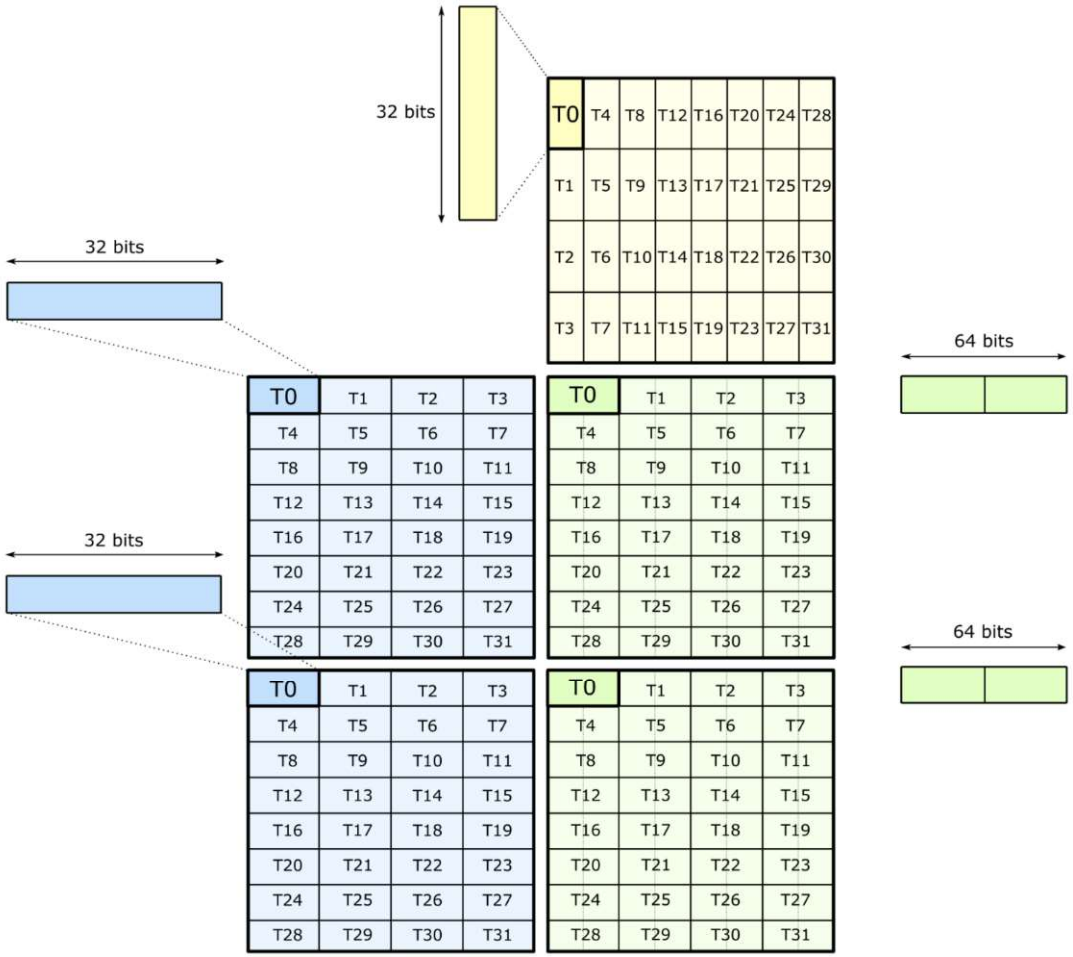


mma.sync.aligned
(via inline PTX)

```
int32_t        D[2];
uint32_t const A;
uint32_t const B;
int32_t const  C[2];


// Example targets 8-by-8-by-16 Tensor Core operation
asm(
  "mma.sync.aligned.m8n8k16.row.col.s32.s8.s8.s32 "
  "  { %0, %1 }, "
  "    %2,       "
  "    %3,       "
  "  { %4, %5 }; "
  :
    "=r"(D[0]), "=r"(D[1])
  :
    "r"(A),     "r"(B),
    "r"(C[0]), "r"(C[1])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

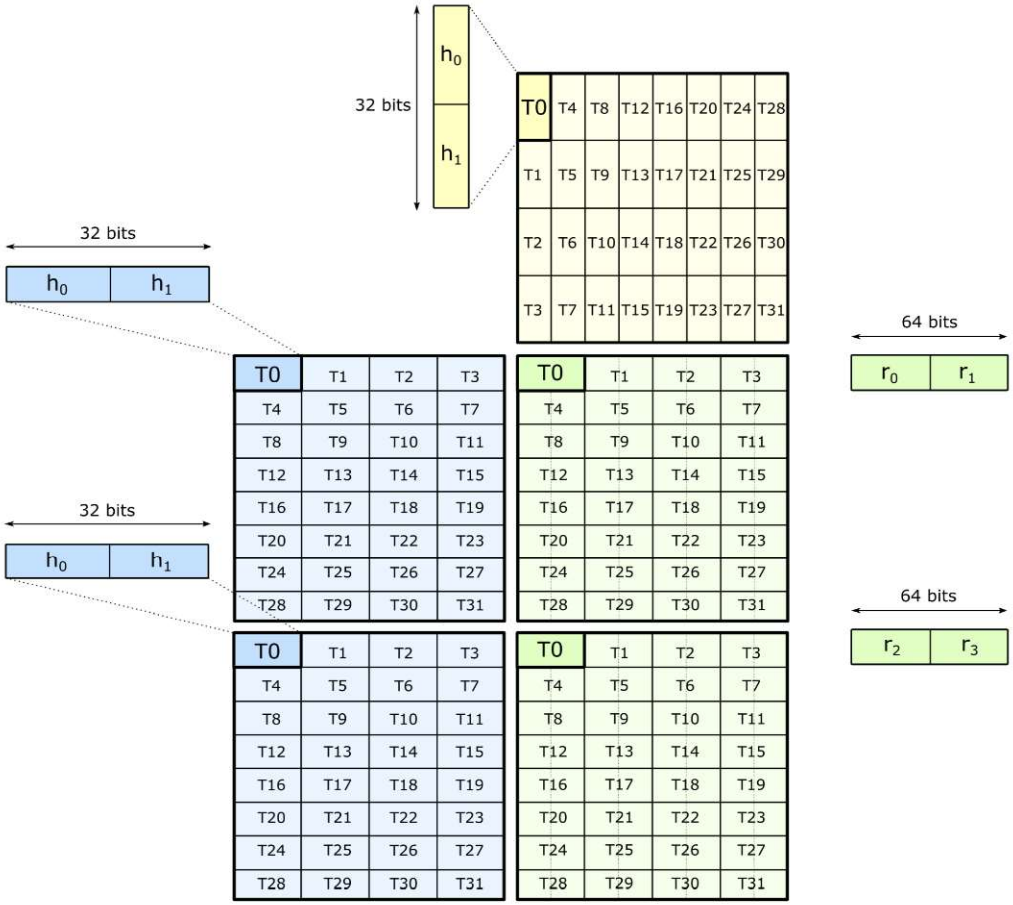
EXPANDING THE *M* DIMENSION



Warp-wide Tensor Core operation: 16-by-8-by-128b

F16 * F16 + F32

16-by-8-by-8

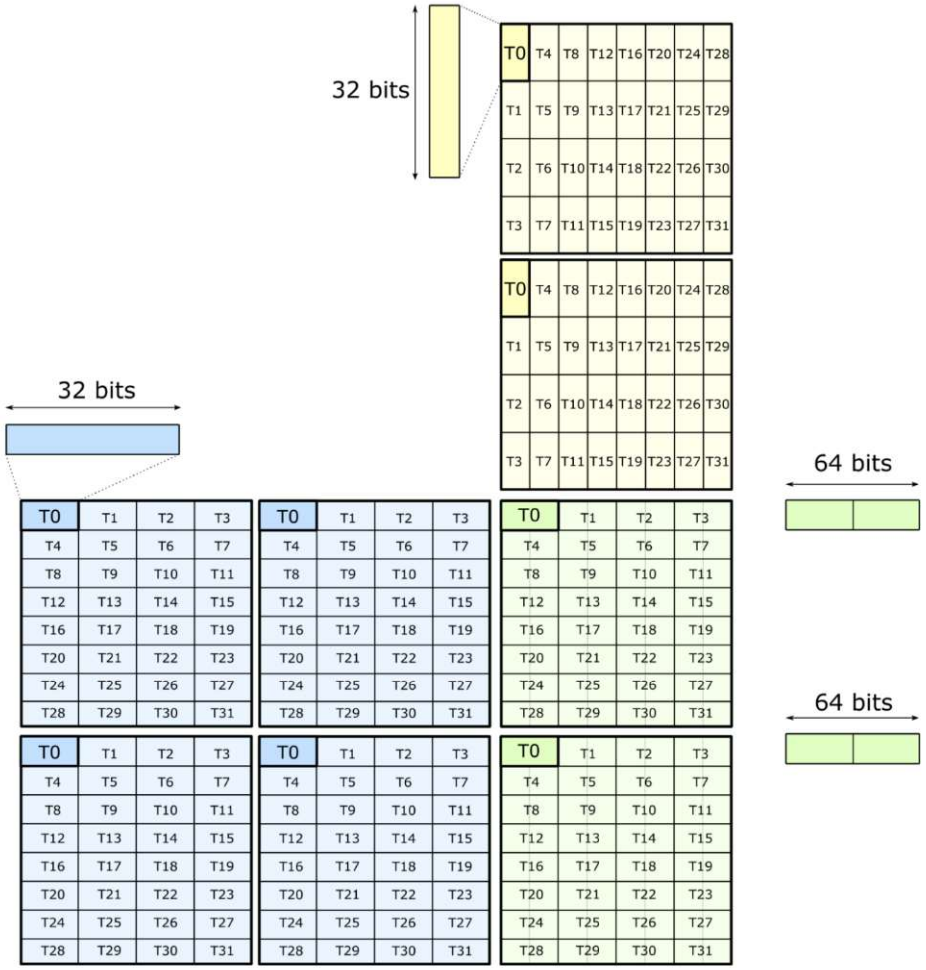


mma.sync.aligned
(via inline PTX)

```
float          D[4];
uint32_t const A[2];
uint32_t const B;
float    const C[4];


// Example targets 16-by-8-by-8 Tensor Core operation
asm(
  "mma.sync.aligned.m16n8k8.row.col.f32.f16.f16.f32 "
  "  { %0, %1, %2, %3 }, "
  "  { %4, %5},          "
  "    %6,               "
  "  { %7, %8, %9, %10 };"
  :
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
  :
    "r"(A[0]), "r"(A[1]),
    "r"(B),
    "f"(C[0]), "f"(C[1])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

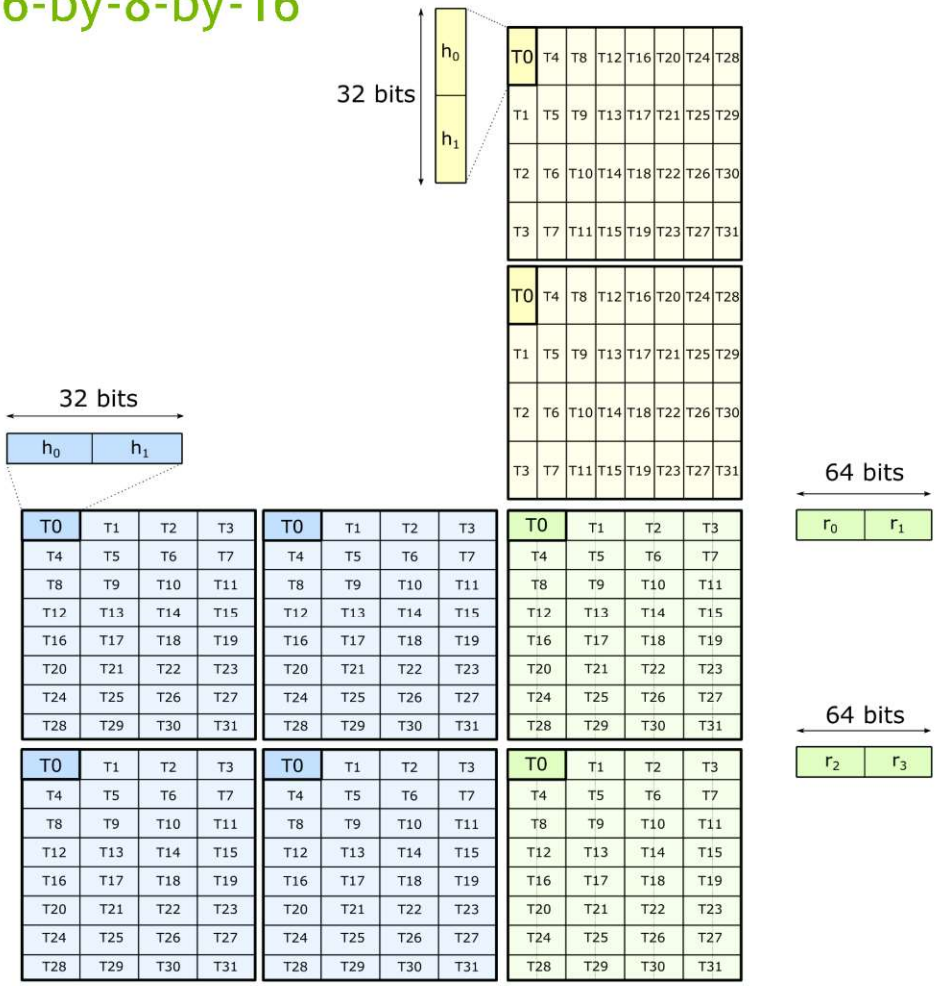
EXPANDING THE *K* DIMENSION



Warp-wide Tensor Core operation: 16-by-8-by-256b

F16 * F16 + F32

16-by-8-by-16



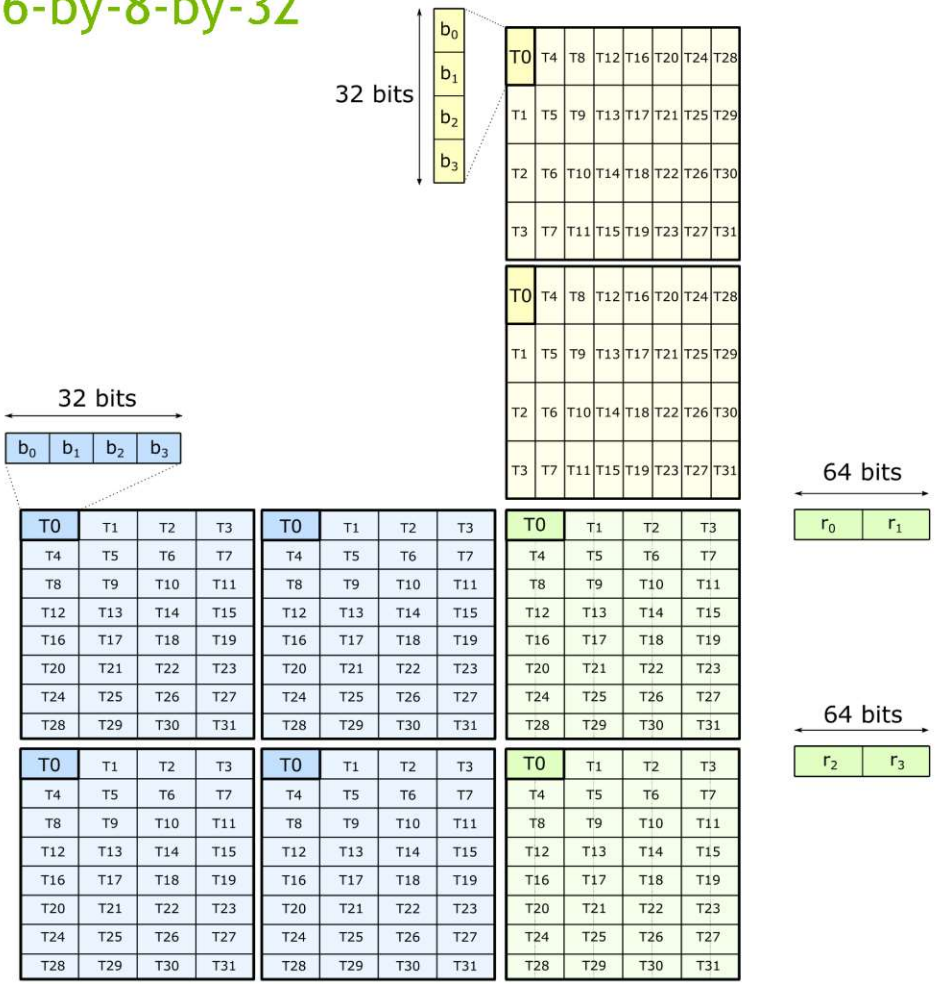
mma.sync.aligned
(via inline PTX)

```
float          D[4];
uint32_t const A[4];
uint32_t const B[2];
float    const C[4];


// Example targets 16-by-8-by-32 Tensor Core operation
asm(
  "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
  "  { %0, %1, %2, %3 },     "
  "  { %4, %5, %6, %7 },     "
  "  { %8, %9 },             "
  "  { %10, %11, %12, %13 };"
  :
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
  :
    "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
    "r"(B[0]), "r"(B[1]),
    "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

S8 * S8 + S32

16-by-8-by-32



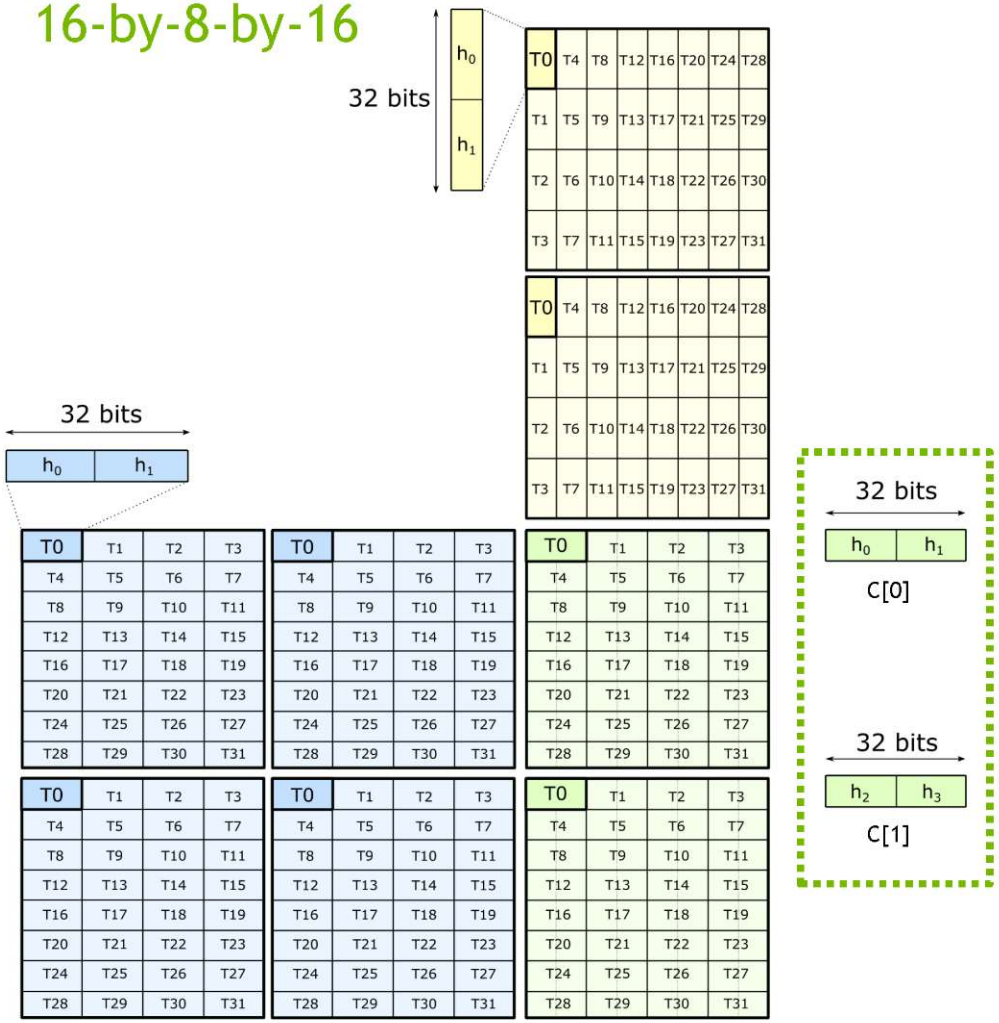
mma.sync.aligned
(via inline PTX)

```
int32_t        D[4];
uint32_t const A[4];
uint32_t const B[2];
int32_t const  C[4];


// Example targets 16-by-8-by-32 Tensor Core operation
asm(
  "mma.sync.aligned.m16n8k32.row.col.s32.s8.s8.s32 "
  "  { %0, %1, %2, %3 },     "
  "  { %4, %5, %6, %7 },     "
  "  { %8, %9 },             "
  "  { %10, %11, %12, %13 };"
  :
    "=r"(D[0]), "=r"(D[1]), "=r"(D[2]), "=r"(D[3])
  :
    "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
    "r"(B[0]), "r"(B[1]),
    "r"(C[0]), "r"(C[1]), "r"(C[2]), "r"(C[3])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

HALF-PRECISION : F16 * F16 + F16

16-by-8-by-16



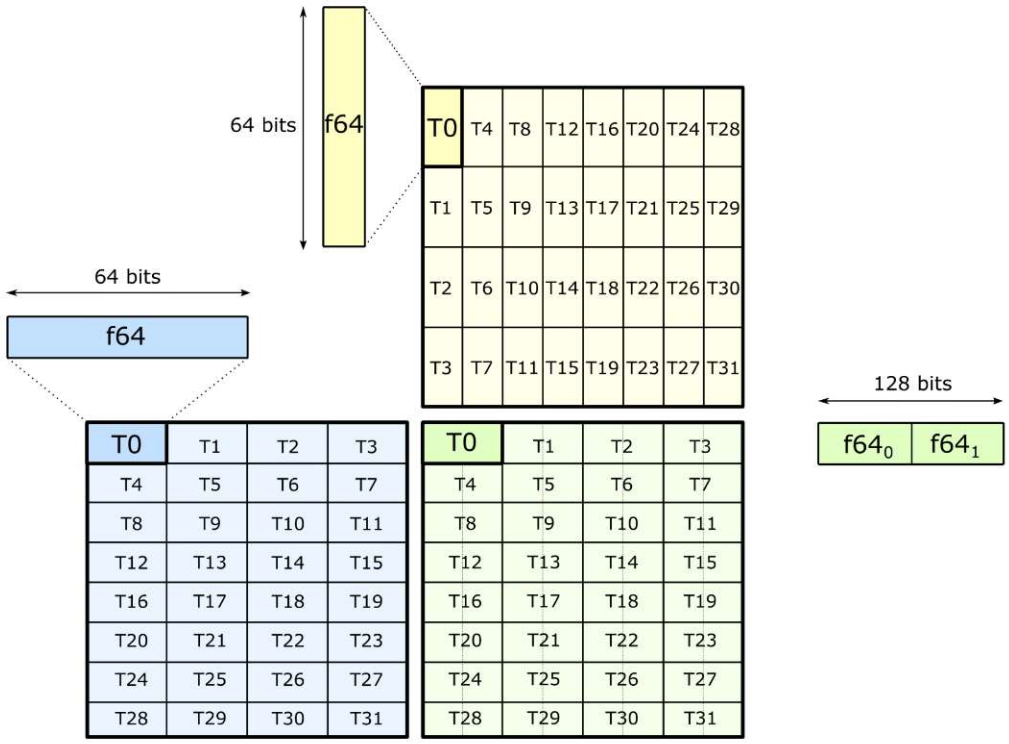
mma.sync.aligned
(via inline PTX)

```
uint32_t       D[2];  // two registers needed (vs. four)
uint32_t const A[4];
uint32_t const B[2];
uint32_t const C[2];  // two registers needed (vs. four)


// Example targets 16-by-8-by-16 Tensor Core operation
asm(
  "mma.sync.aligned.m16n8k16.row.col.f16.f16.f16.f16 "
  "  { %0, %1},              "
  "  { %2, %3, %4, %5 },     "
  "  { %6, %7 },             "
  "  { %8, %9 };             "
  :
    "=r"(D[0]), "=r"(D[1])
  :
    "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
    "r"(B[0]), "r"(B[1]),
    "r"(C[0]), "r"(C[1])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

DOUBLE-PRECISION: F64 * F64 + F64

8-by-8-by-4



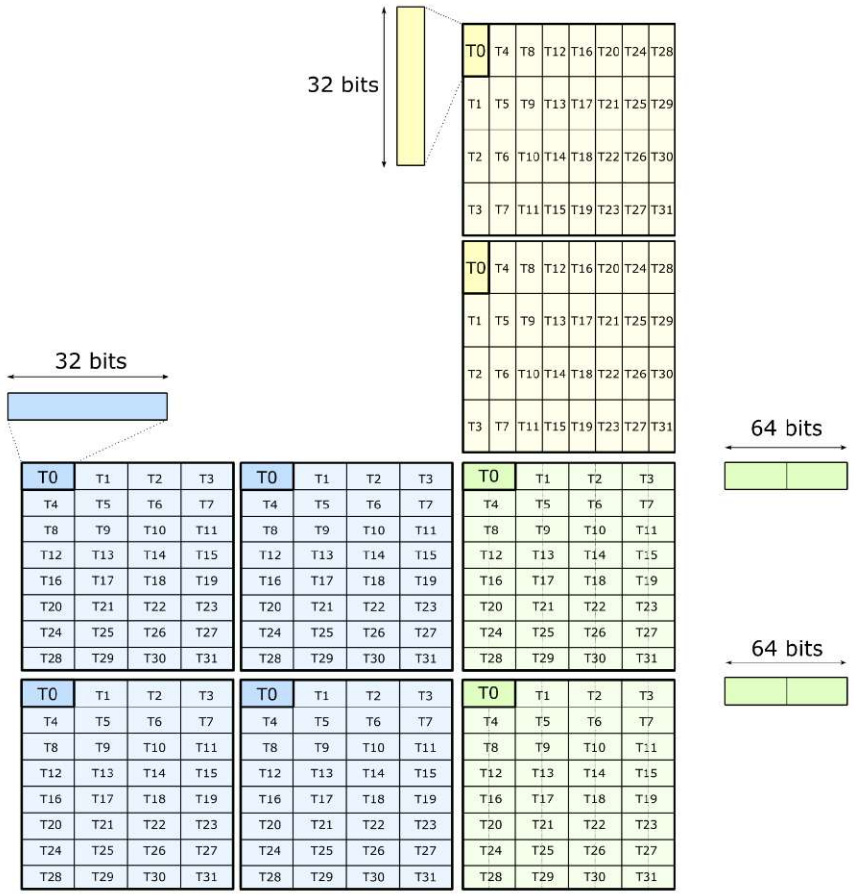
mma.sync.aligned
(via inline PTX)

```
uint64_t       D[2];   // two 64-bit accumulators
uint64_t const A;      // one 64-bit element for A operand
uint64_t const B;      // one 64-bit element for B operand
uint64_t const C[2];   // two 64-bit accumulators


// Example targets 8-by-8-by-4 Tensor Core operation
asm(
  "mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64 "
  "  { %0, %1},    "
  "    %2,         "
  "    %3,         "
  "  { %4, %5 };   "
  :
    "=l"(D[0]), "=l"(D[1])
  :
    "l"(A),
    "l"(B),
    "l"(C[0]), "l"(C[1])
);
```

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

CUTLASS: wraps PTX in template

m-by-*n*-by-*k*



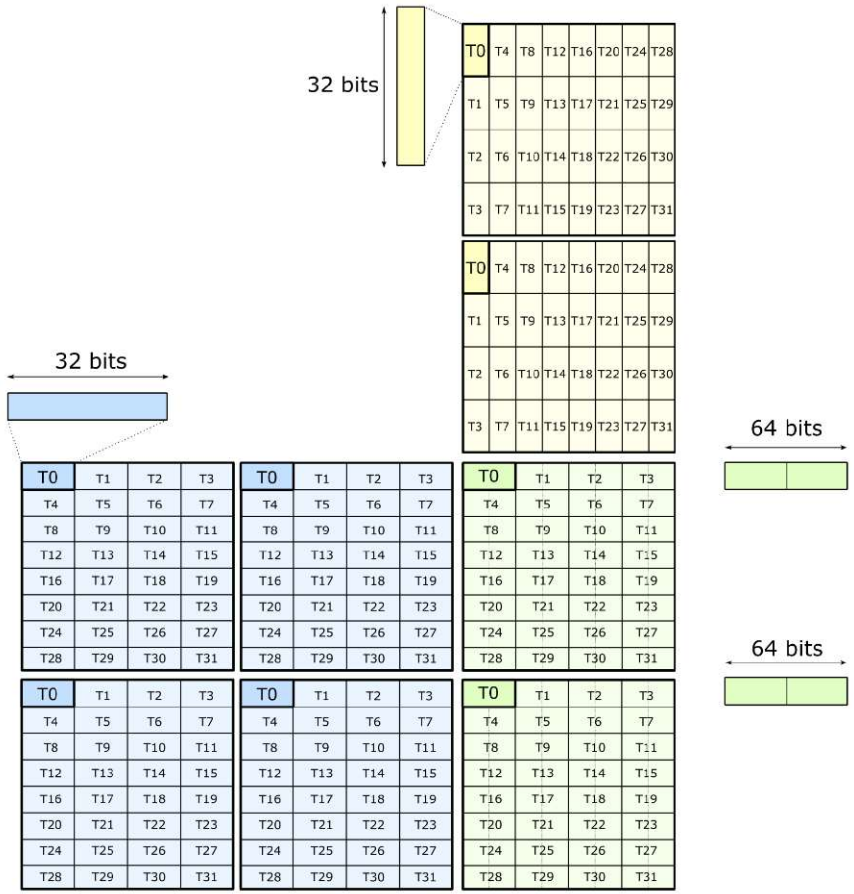
cutlass::arch::Mma

```
/// Matrix multiply-add operation
template <
  /// Size of the matrix product (concept: GemmShape)
  typename Shape,
  /// Number of threads participating
  int kThreads,
  /// Data type of A elements
  typename ElementA,
  /// Layout of A matrix (concept: MatrixLayout)
  typename LayoutA,
  /// Data type of B elements
  typename ElementB,
  /// Layout of B matrix (concept: MatrixLayout)
  typename LayoutB,
  /// Element type of C matrix
  typename ElementC,
  /// Layout of C matrix (concept: MatrixLayout)
  typename LayoutC,
  /// Inner product operator
  typename Operator
>
struct Mma;
```

https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/arch/mma_sm80.h

CUTLASS: wraps PTX in template

16-by-8-by-16



cutlass::arch::Mma

```
__global__ void kernel() {

  // arrays containing logical elements
  Array<half_t, 8> A;
  Array<half_t, 4> B;
  Array< float, 4> C;

  // define the appropriate matrix operation
  arch::Mma< GemmShape<16, 8, 16>, 32, ... > mma;

  // in-place matrix multiply-accumulate
  mma(C, A, B, C);

  ...
}
```

https://github.com/NVIDIA/cutlass/blob/master/include/cutlass/arch/mma_sm80.h


EFFICIENT DATA MOVEMENT
FOR TENSOR CORES

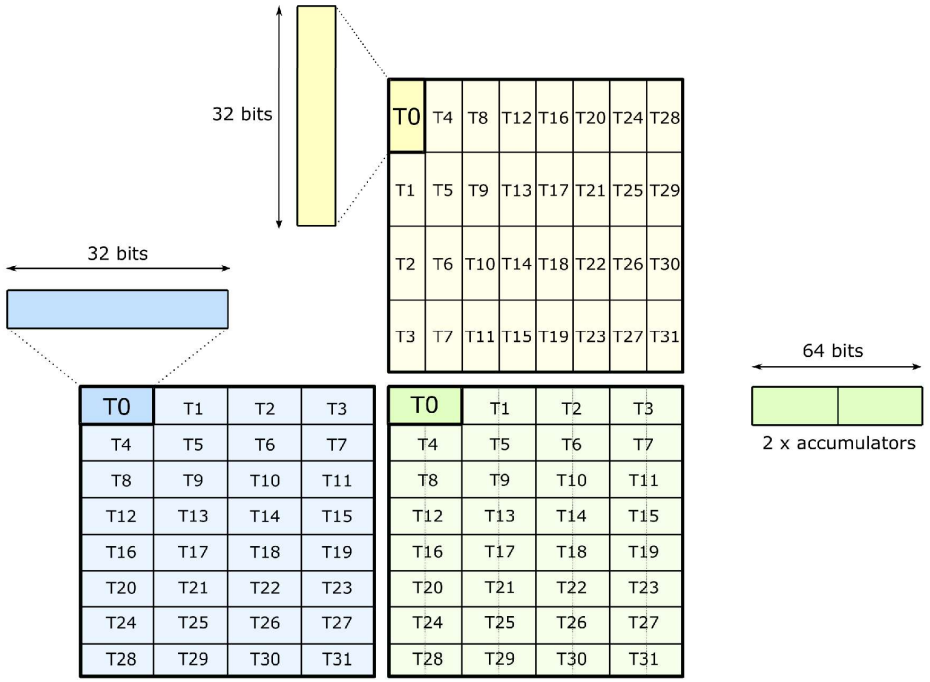
HELLO WORLD: TENSOR CORES

Map each thread to coordinates of the matrix operation

Load inputs from memory

Perform the matrix operation

Store the result to memory



CUDA example

```
__global__ void tensor_core_example_8x8x16(
  int32_t        *D,
  uint32_t const *A,
  uint32_t const *B,
  int32_t const  *C) {

  // Compute the coordinates of accesses to A and B matrices

  int outer = threadIdx.x / 4;     // m or n dimension
  int inner = threadIdx.x % 4;     // k dimension

  // Compute the coordinates for the accumulator matrices
  int c_row = threadIdx.x / 4;
  int c_col = 2 * (threadIdx.x % 4);

  // Compute linear offsets into each matrix
  int ab_idx = outer * 4 + inner;
  int cd_idx = c_row * 8 + c_col;

  // Issue Tensor Core operation
  asm(
    "mma.sync.aligned.m8n8k16.row.col.s32.s8.s8.s32 "
    "  { %0, %1 }, "
    "    %2,       "
    "    %3,       "
    "  { %4, %5 }; "
    :
      "=r"(D[cd_idx]), "=r"(D[cd_idx + 1])
    :
      "r"(A[ab_idx]),
      "r"(B[ab_idx]),
      "r"(C[cd_idx]),  "r"(C[cd_idx + 1])
  );
}
```

PERFORMANCE IMPLICATIONS

Load A and B inputs from memory: 2 x 4B per thread

Perform one Tensor Core operation: 2048 flops per warp

2048 flops require 256 B of loaded data

➔ 8 flops/byte

NVIDIA A100 Specifications:

- 624 TFLOP/s (INT8)
- 1.6 TB/s (HBM2)

➔ 400 flops/byte

8 flops/byte * 1.6 TB/s ➔ 12 TFLOP/s

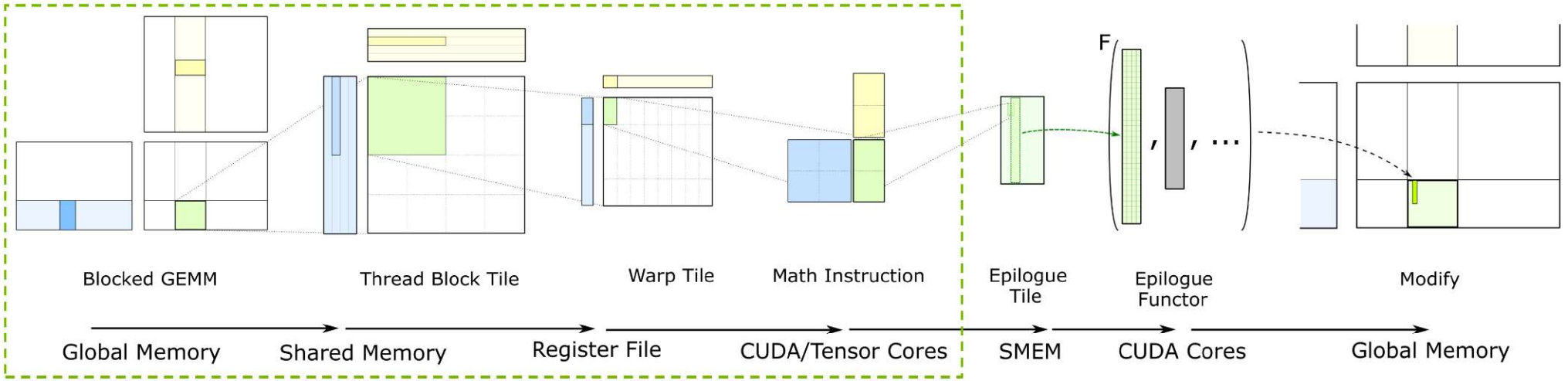
This kernel is global memory bandwidth limited.

CUDA example

```
__global__ void tensor_core_example_8x8x16(
  int32_t        *D,
  uint32_t const *A,
  uint32_t const *B,
  int32_t const  *C) {

  // Compute the coordinates of accesses to A and B matrices

  int outer = threadIdx.x / 4;     // m or n dimension
  int inner = threadIdx.x % 4;     // k dimension

  // Compute the coordinates for the accumulator matrices
  int c_row = threadIdx.x / 4;
  int c_col = 2 * (threadIdx.x % 4);

  // Compute linear offsets into each matrix
  int ab_idx = outer * 4 + inner;
  int cd_idx = c_row * 8 + c_col;

  // Issue Tensor Core operation
  asm(
    "mma.sync.aligned.m8n8k16.row.col.s32.s8.s8.s32 "
    "  { %0, %1 }, "
    "    %2,       "
    "    %3,       "
    "  { %4, %5 }; "
    :
      "=r"(D[cd_idx]), "=r"(D[cd_idx + 1])
    :
      "r"(A[ab_idx]),
      "r"(B[ab_idx]),
      "r"(C[cd_idx]),  "r"(C[cd_idx + 1])
  );
}
```

FEEDING THE DATA PATH

Efficient storing and loading through Shared Memory



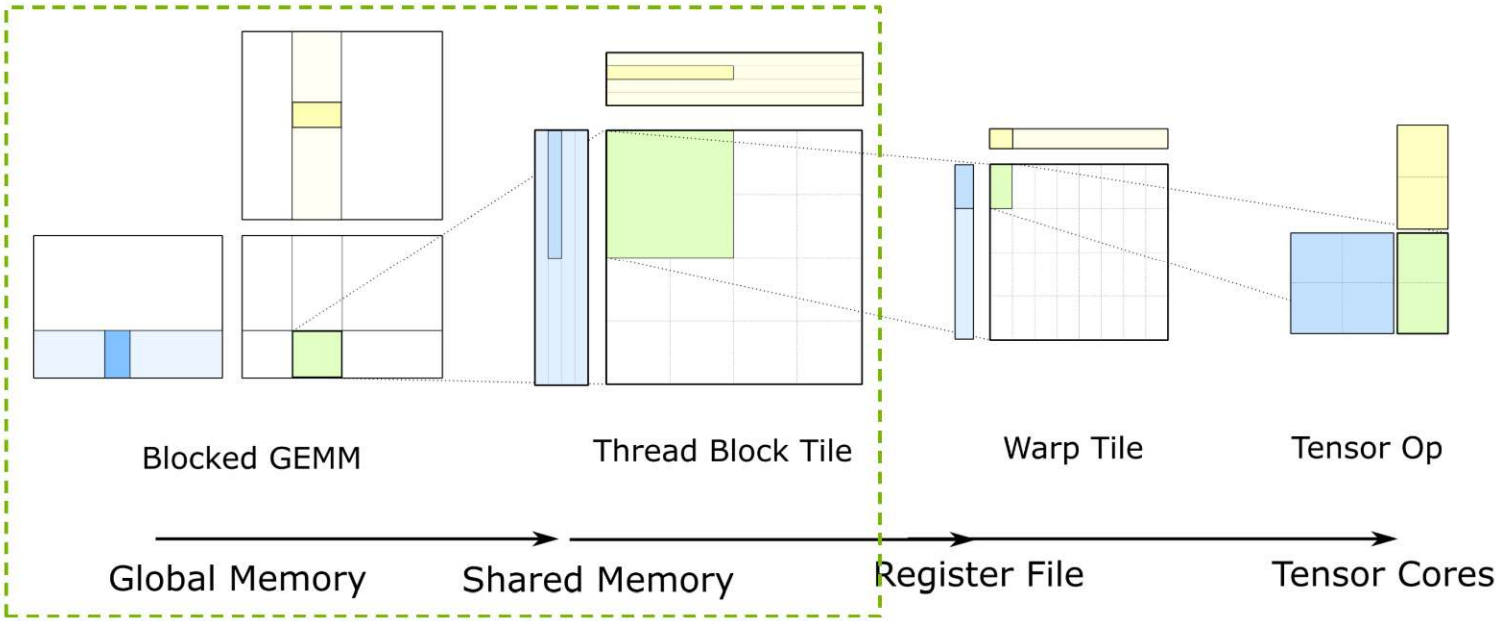
Tiled, hierarchical model: reuse data in Shared Memory and in Registers

See CUTLASS GTC 2018 talk for more details about this model.

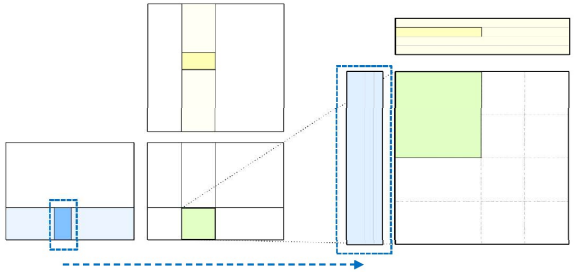
FEEDING THE DATA PATH

Move data from Global Memory to Tensor Cores as efficiently as possible

- **Latency-tolerant pipeline from Global Memory**
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



ASYNCHRONOUS COPY: EFFICIENT PIPELINES

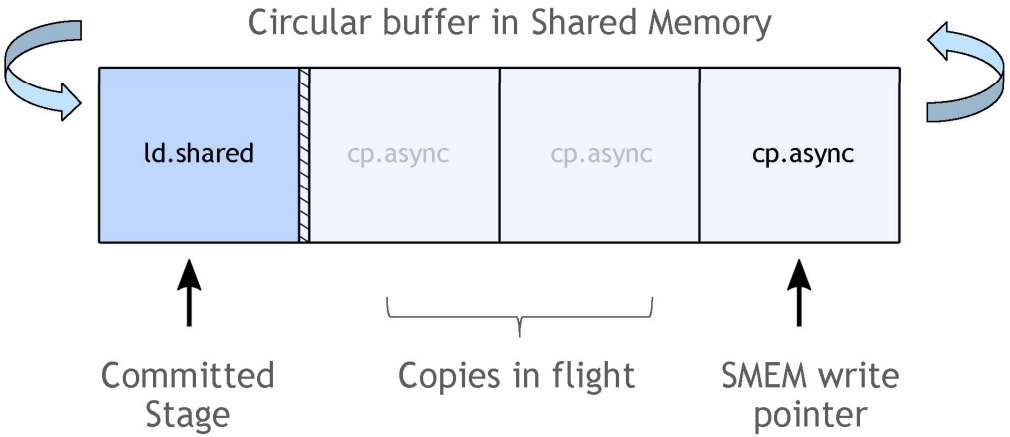


New NVIDIA Ampere Architecture feature: cp.async

- Asynchronous copy directly from Global to Shared Memory
- See *"Inside the NVIDIA Ampere Architecture"* for more details (GTC 2020 – S21730)

Enables efficient software pipelines

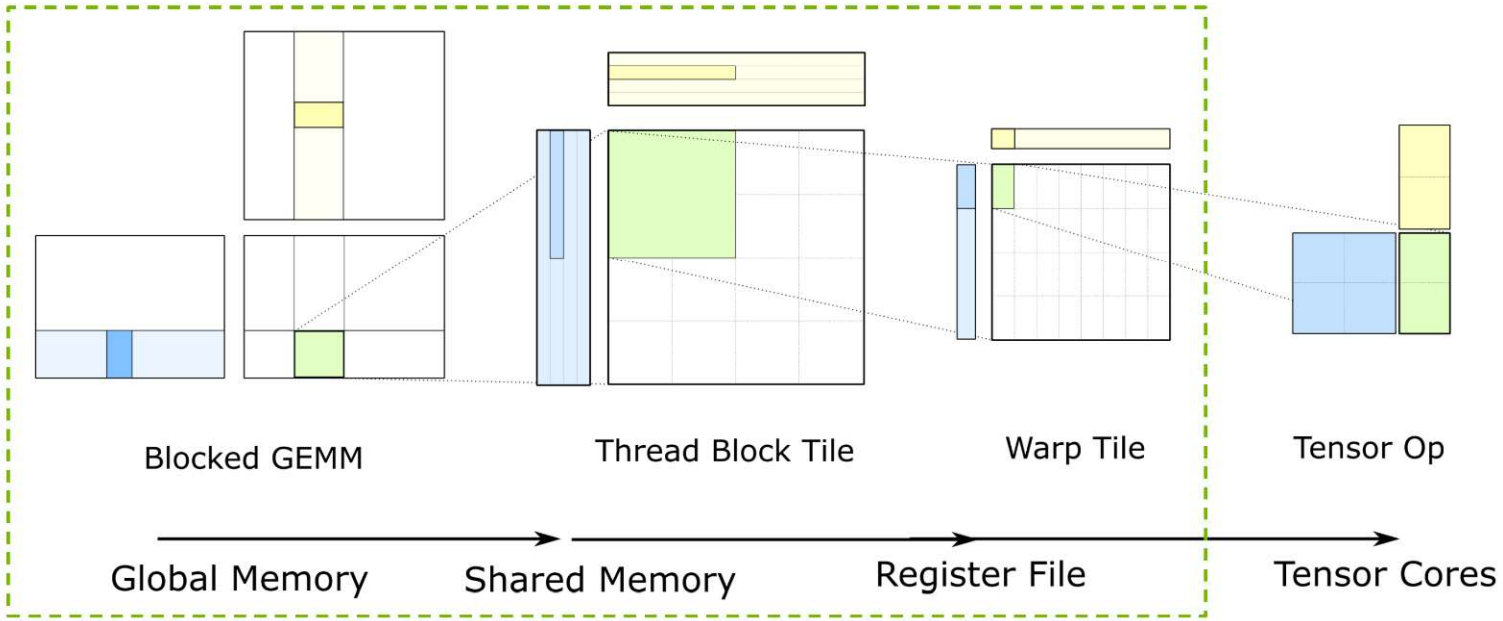
- Minimizes data movement: L2 → L1 → RF → SMEM becomes L2 → SMEM
- Saves registers: RF no longer needed to hold the results of long-latency load instructions
- Indirection: fetch several stages in advance for greater latency tolerance



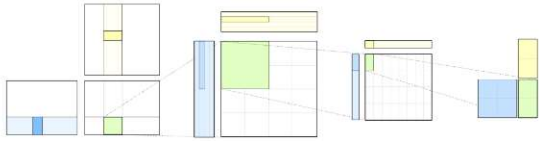
FEEDING THE DATA PATH

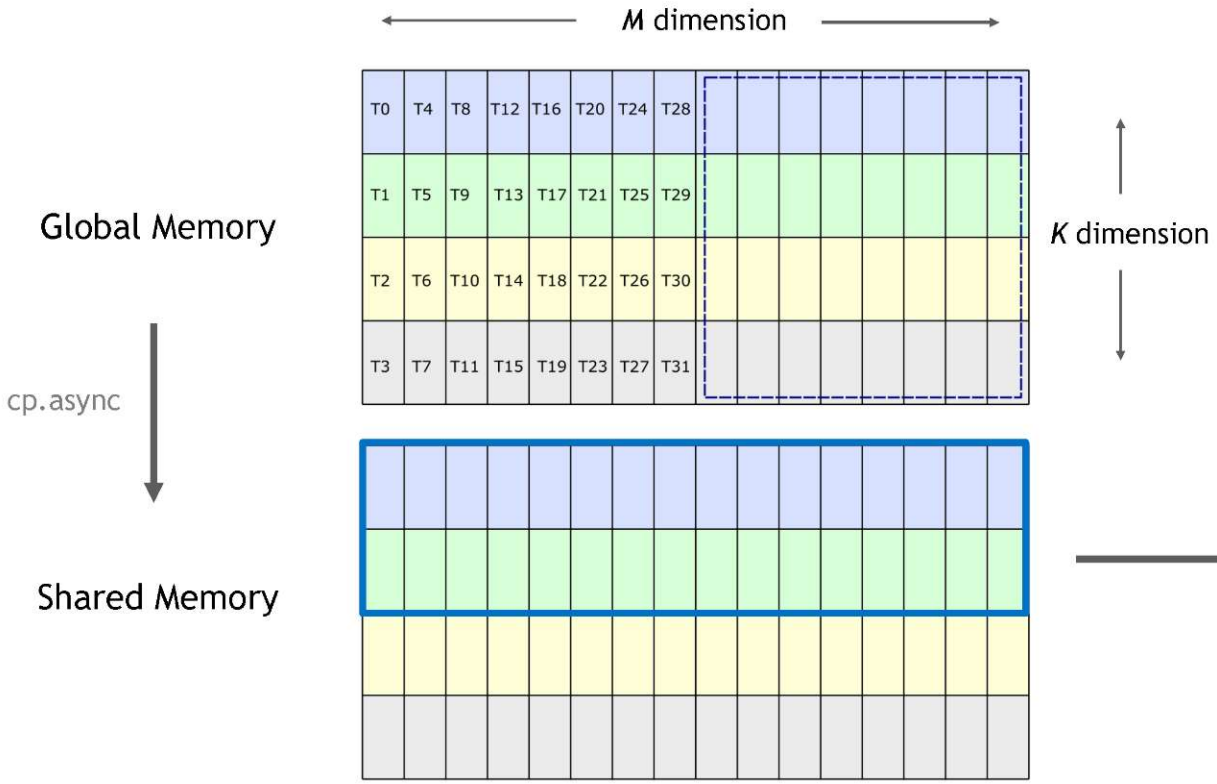
Move data from Global Memory to Tensor Cores as efficiently as possible

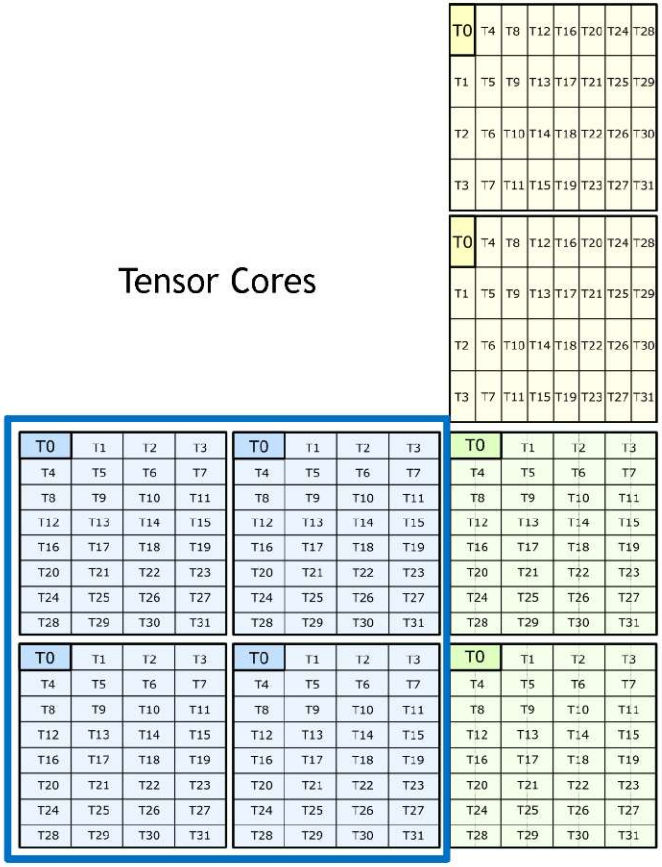
- Latency-tolerant pipeline from Global Memory
- **Conflict-free Shared Memory stores**
- **Conflict-free Shared Memory loads**



GLOBAL MEMORY TO TENSOR CORES







LDMATRIX: FETCH TENSOR CORE OPERANDS

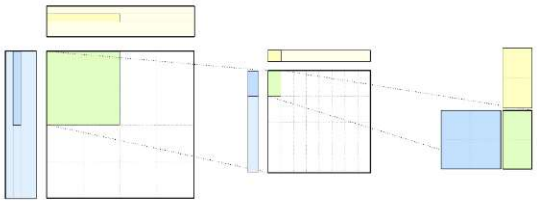
PTX instruction to load a matrix from Shared Memory

Each thread supplies a pointer to 128b row of data in Shared Memory

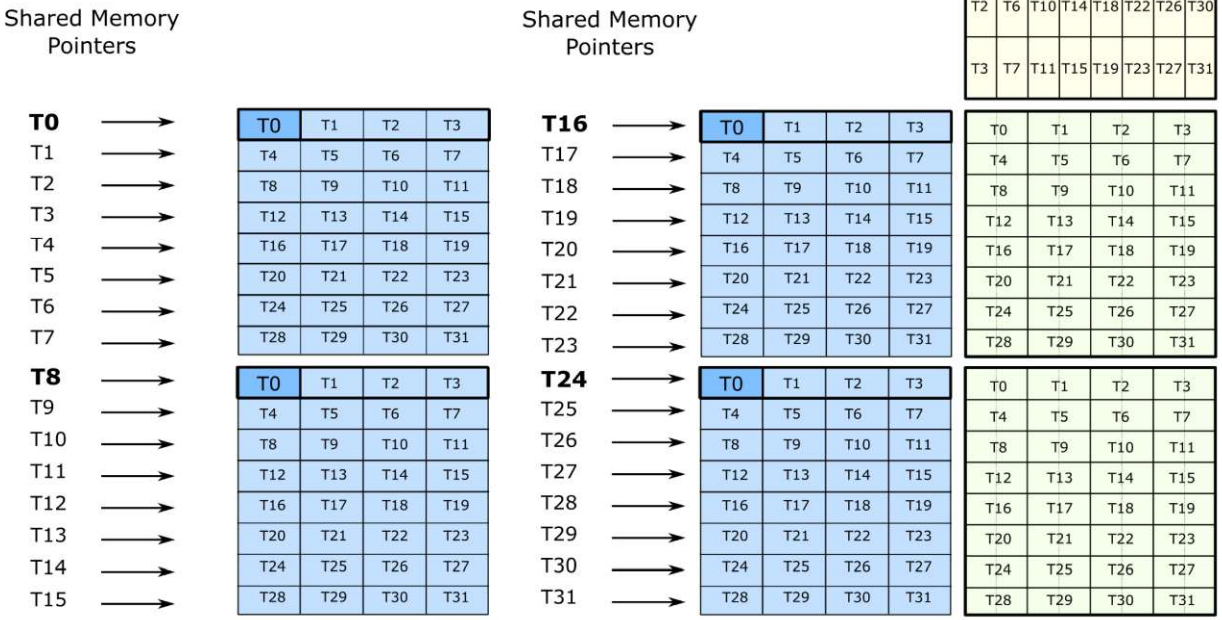
Each 128b row is broadcast to groups of four threads

(potentially different threads than the one supplying the pointer)

Data matches arrangement of inputs to Tensor Core operations



Shared Memory Tensor Cores



LDMATRIX: PTX INSTRUCTION

PTX instruction to load a matrix from SMEM

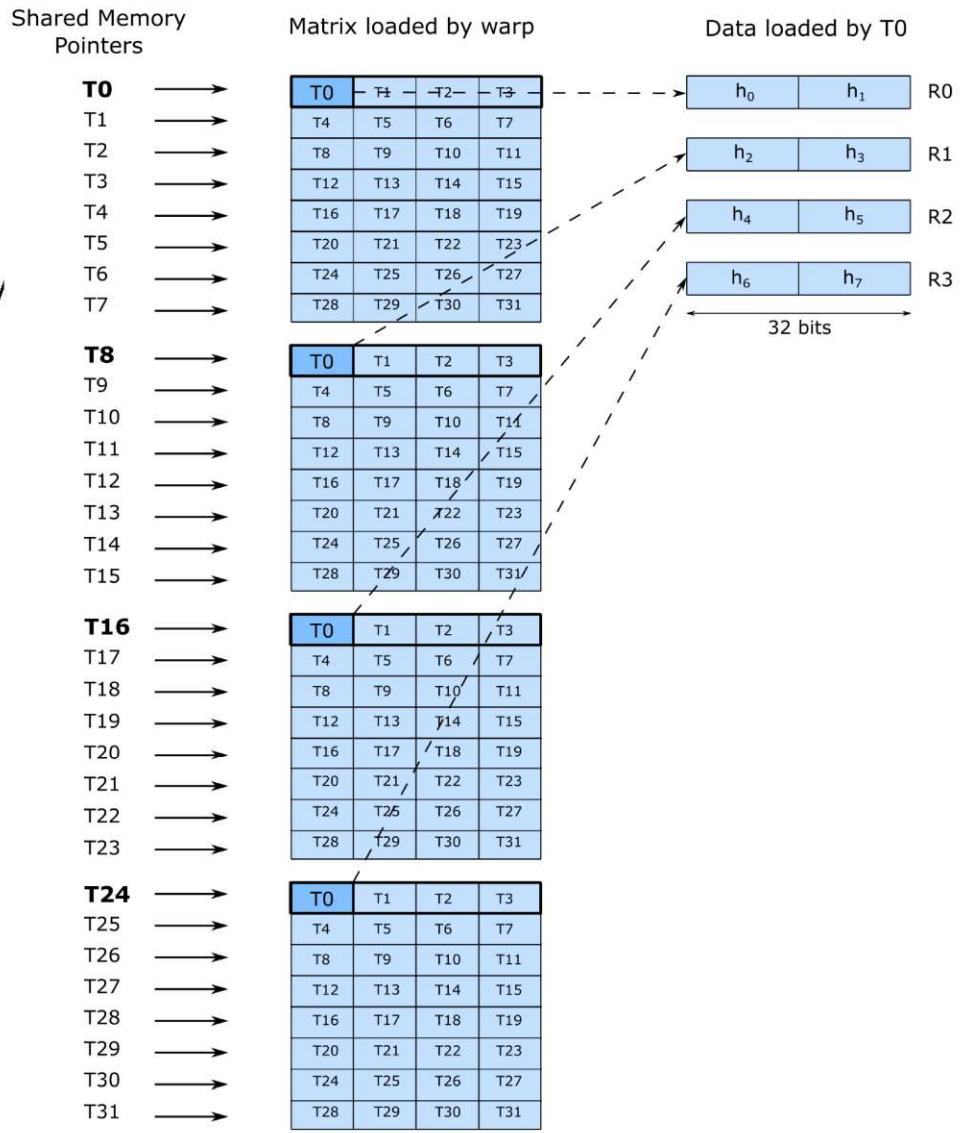
Each thread supplies a pointer to 128b row of data in Shared Memory

Each 128b row is broadcast to groups of four threads

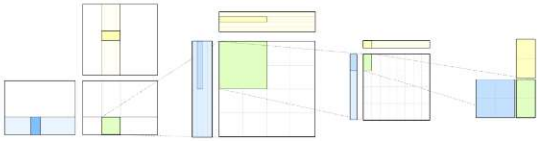
(potentially different threads than the one supplying the pointer)

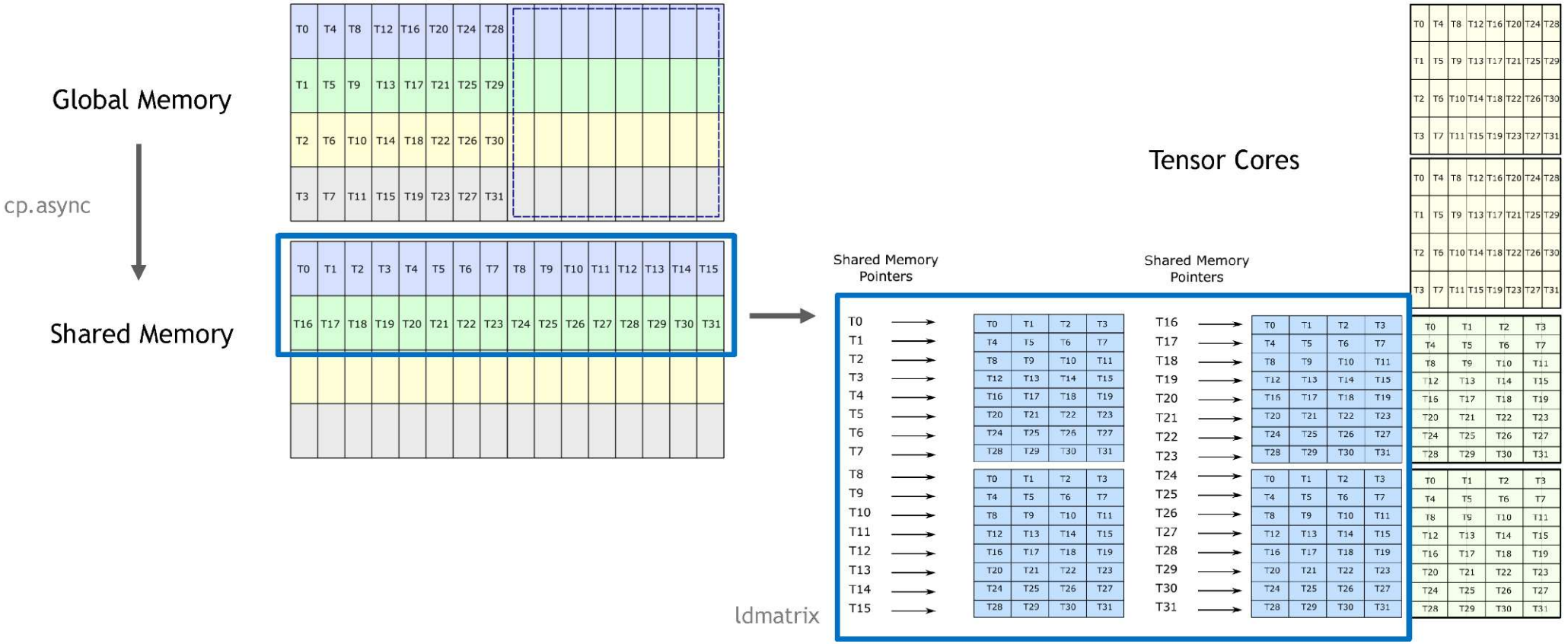
Data matches arrangement of inputs to Tensor Core operations

```
// Inline PTX assembly for ldmatrix

uint32_t  R[4];
uint32_t  smem_ptr;

asm volatile (
  "ldmatrix.sync.aligned.x4.m8n8.shared.b16 "
  "{%0, %1, %2, %3}, [%4];                  "
 :
  "=r"(R[0]), "=r"(R[1]), "=r"(R[2]), "=r"(R[3])
 :
  "r"(smem_ptr)
);
```



GLOBAL MEMORY TO TENSOR CORES





NVIDIA AMPERE ARCHITECTURE – SHARED MEMORY BANK TIMING

Bank conflicts between threads in the same phase

4B words are accessed in 1 phase

8B words are accessed in 2 phases:

- Process addresses of the **first 16** threads in a warp
- Process addresses of the **second 16** threads in a warp

16B words are accessed in 4 phases:

- Each phase processes **8** consecutive threads of a warp

Phase 0: T0 .. T7

Phase 1: T8 .. T15

Phase 2: T16 .. T23

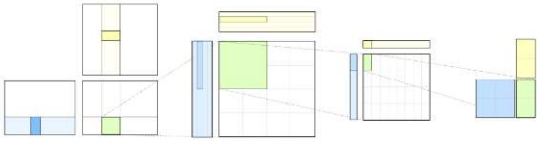
Phase 3: T24 .. T31

128 bit access size

Slide borrowed from: Guillaume Thomas-Collignon and Paulius Micikevicius. "Volta Architecture and performance optimization." GTC 2018.

http://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

GLOBAL MEMORY TO TENSOR CORES



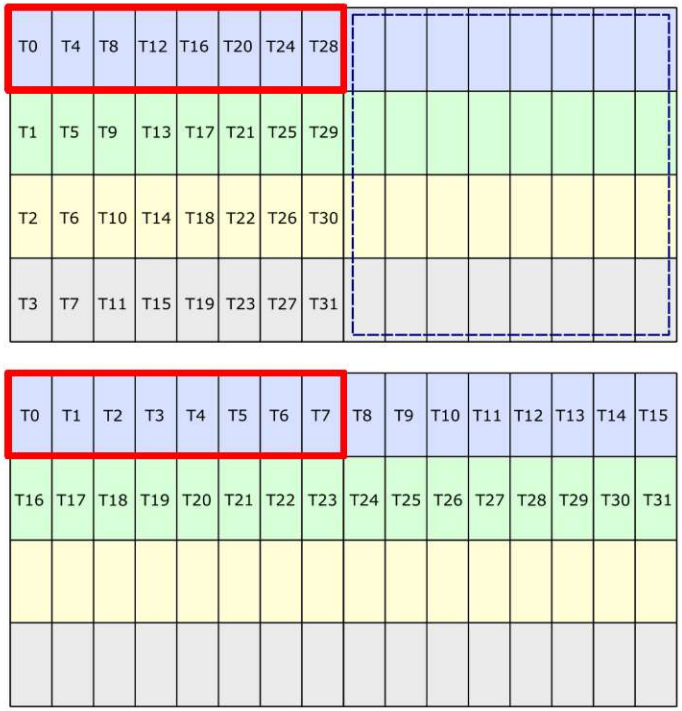
Global Memory

cp.async

Shared Memory

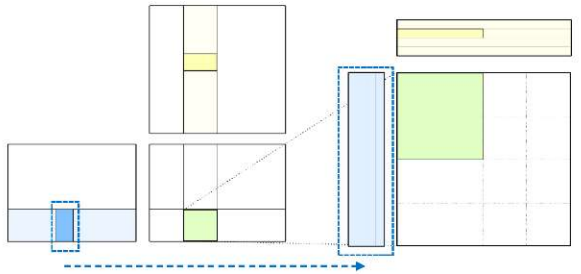
ldmatrix

Registers



Bank conflict on either store or load from Shared Memory

GLOBAL TO SHARED MEMORY



Load from Global Memory

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

Load
(128 bits per thread)

Store
(128 bits per thread)

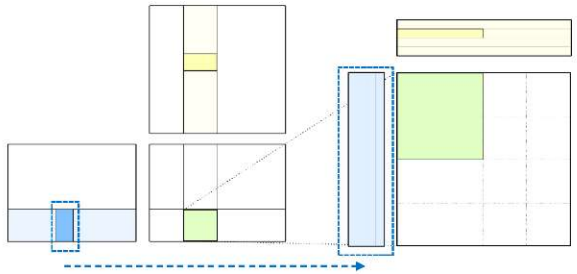
Permuted Shared Memory layout

XOR function maps thread index to Shared Memory location

Store to Shared Memory

T0	T1	T2	T3	T4	T5	T6	T7
T9	T8	T11	T10	T13	T12	T15	T14
T18	T19	T16	T17	T22	T23	T20	T21
T27	T26	T25	T24	T31	T30	T29	T28

GLOBAL TO SHARED MEMORY



Load from Global Memory

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

Load
(128 bits per thread)

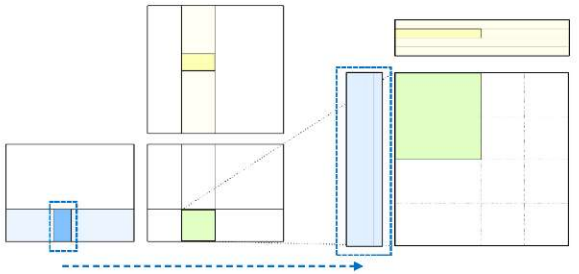
Store
(128 bits per thread)

Store to Shared Memory

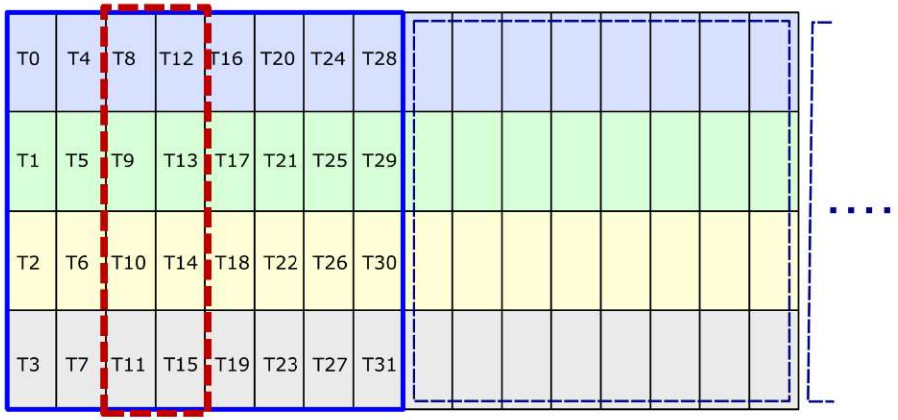
T0	T1	T2	T3	T4	T5	T6	T7
T9	T8	T11	T10	T13	T12	T15	T14
T18	T19	T16	T17	T22	T23	T20	T21
T27	T26	T25	T24	T31	T30	T29	T28

Phase 0: T0 .. T7
Phase 1: T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31

GLOBAL TO SHARED MEMORY



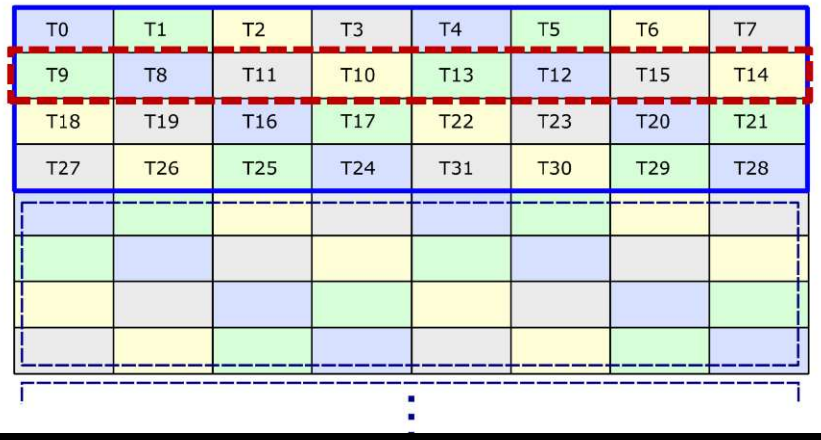
Load from Global Memory



Load
(128 bits per thread)

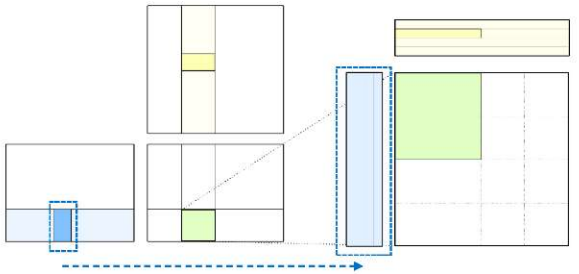
Store
(128 bits per thread)

Store to Shared Memory

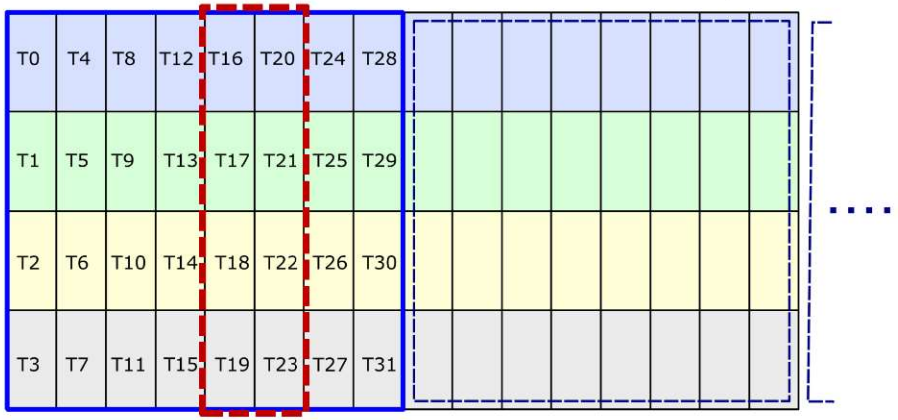



```
Phase 0:  T0 .. T7
Phase 1:  T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
```

GLOBAL TO SHARED MEMORY



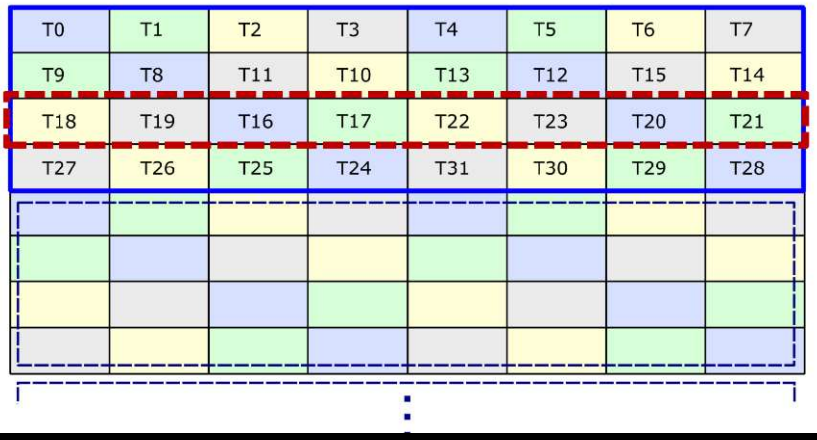
Load from Global Memory



Load
(128 bits per thread)

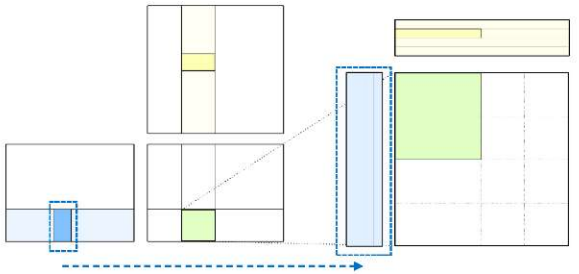
Store
(128 bits per thread)

Store to Shared Memory




```
Phase 0:  T0 .. T7
Phase 1:  T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
```

GLOBAL TO SHARED MEMORY



Load from Global Memory

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

Load
(128 bits per thread)

Store
(128 bits per thread)

Store to Shared Memory

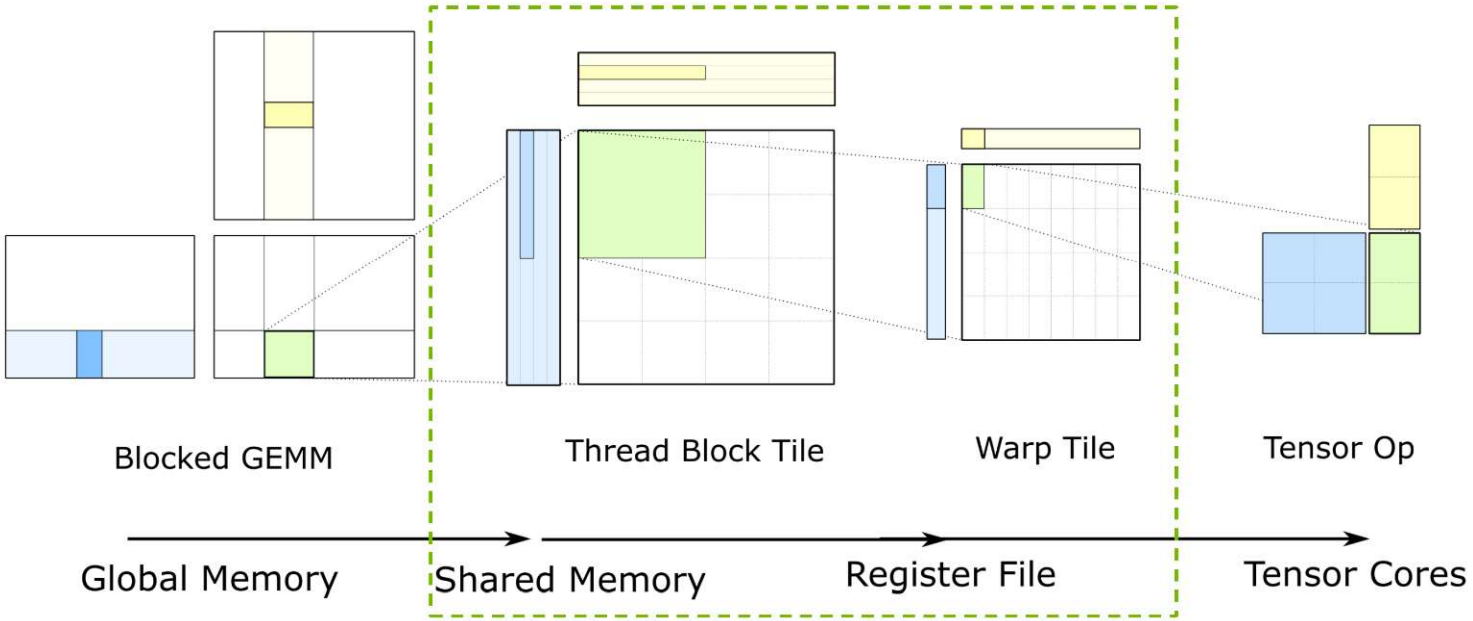
T0	T1	T2	T3	T4	T5	T6	T7
T9	T8	T11	T10	T13	T12	T15	T14
T18	T19	T16	T17	T22	T23	T20	T21
T27	T26	T25	T24	T31	T30	T29	T28

Phase 0: T0 .. T7
Phase 1: T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31

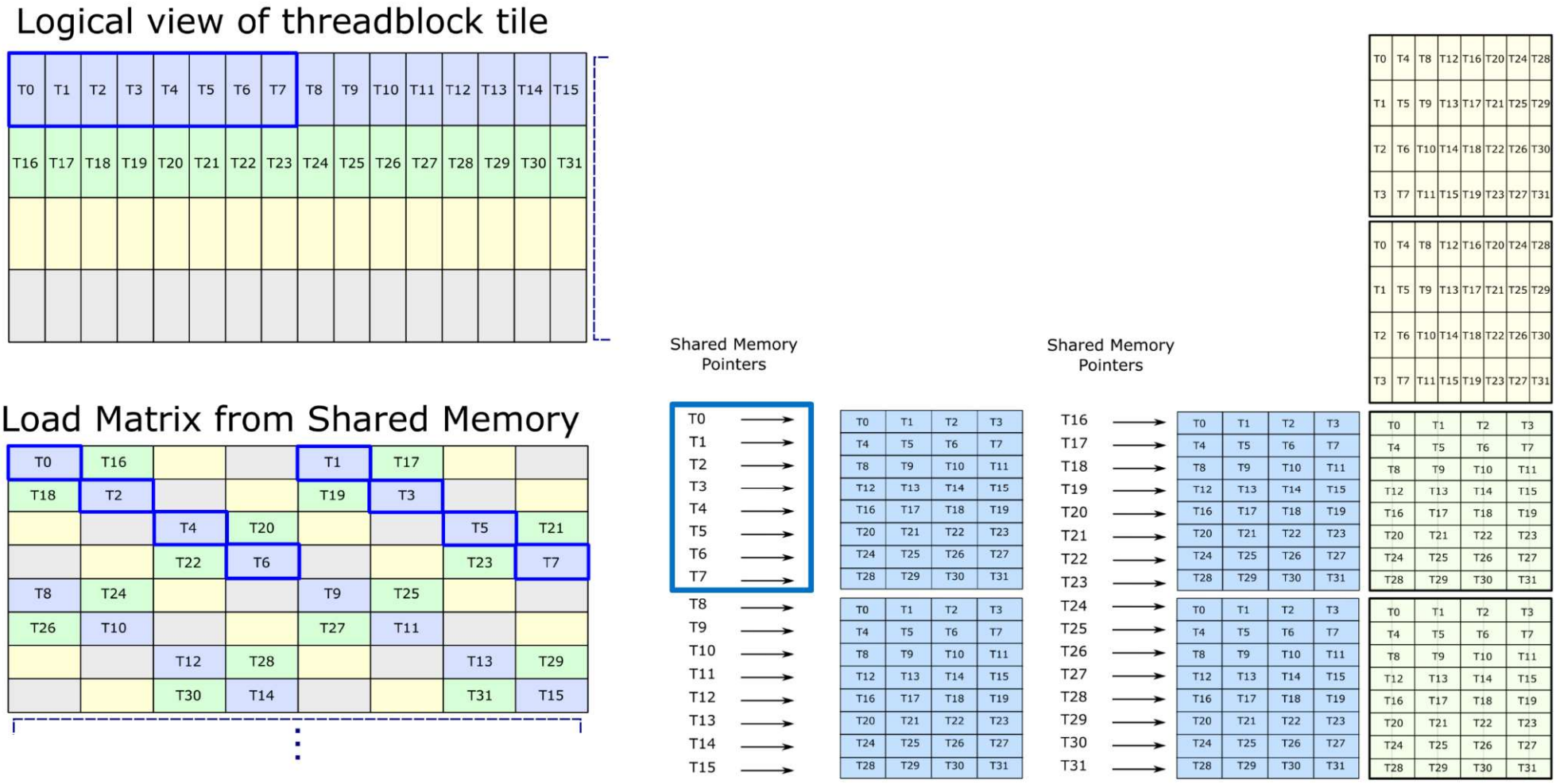
FEEDING THE DATA PATH

Move data from Global Memory to Tensor Cores as efficiently as possible

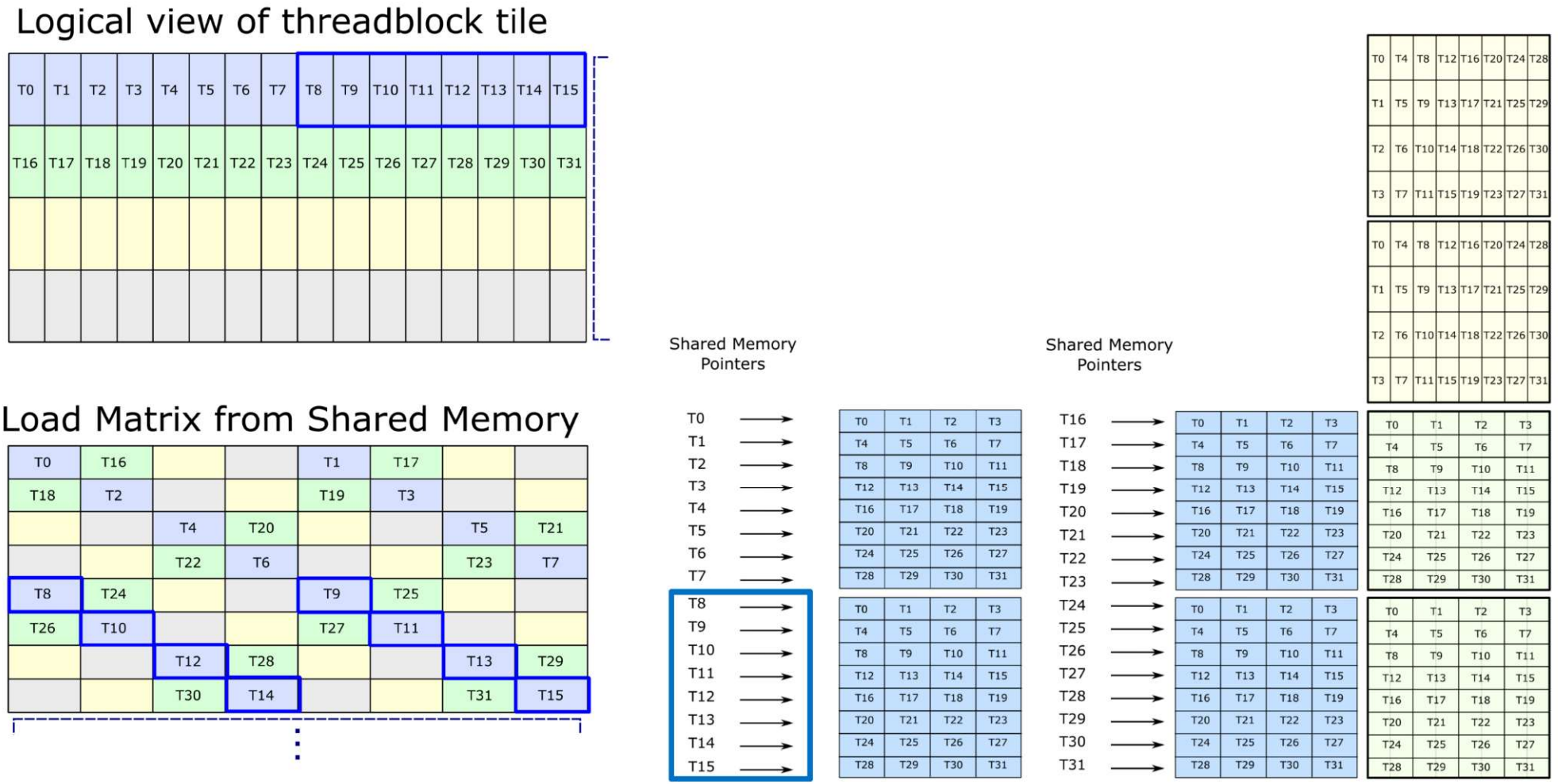
- Latency-tolerant pipeline from Global Memory
- Conflict-free Shared Memory stores
- **Conflict-free Shared Memory loads**



LOADING FROM SHARED MEMORY TO REGISTERS

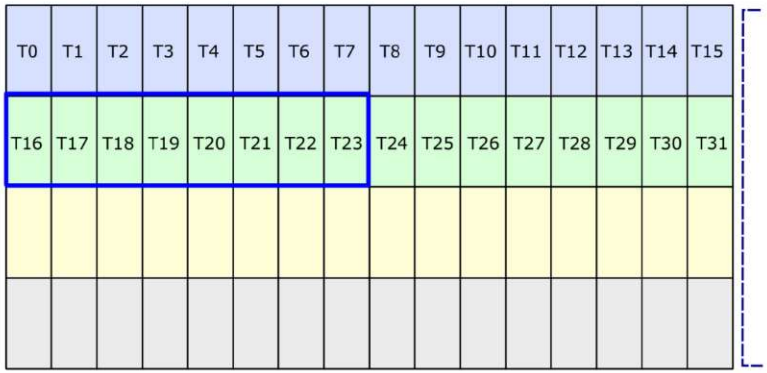


LOADING FROM SHARED MEMORY TO REGISTERS

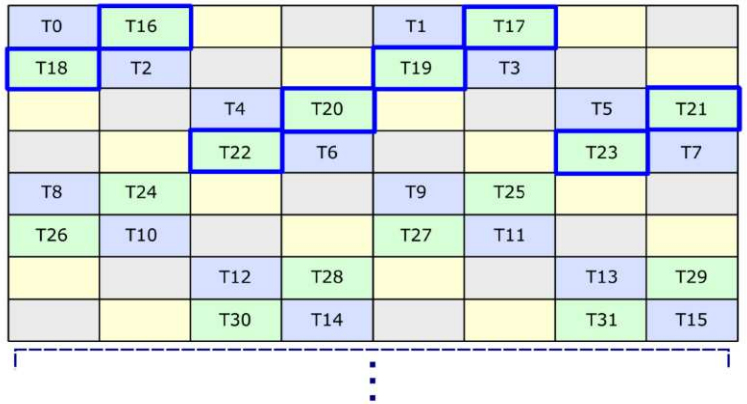


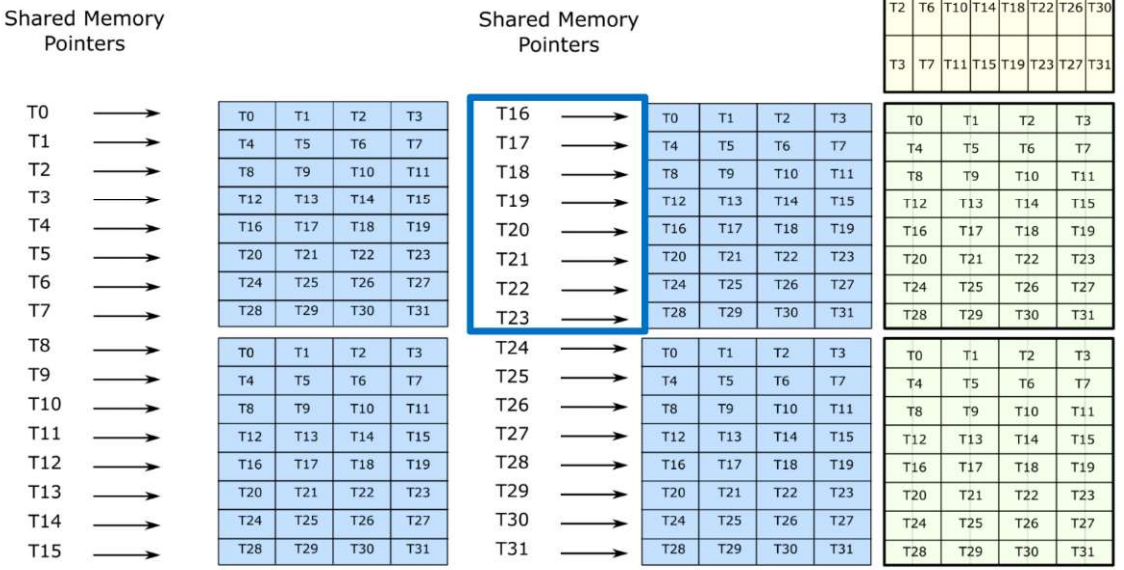
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile



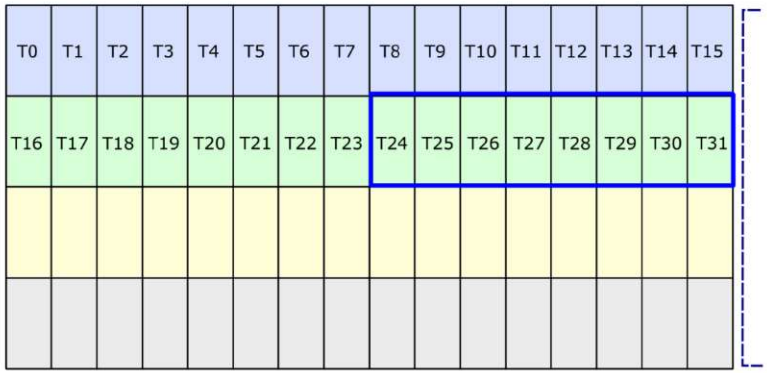
Load Matrix from Shared Memory



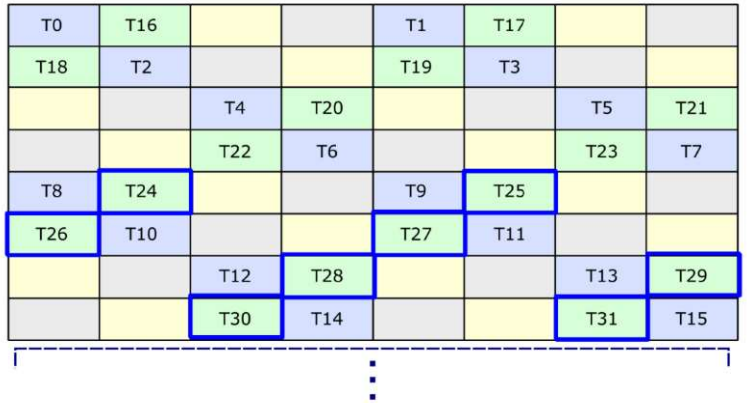


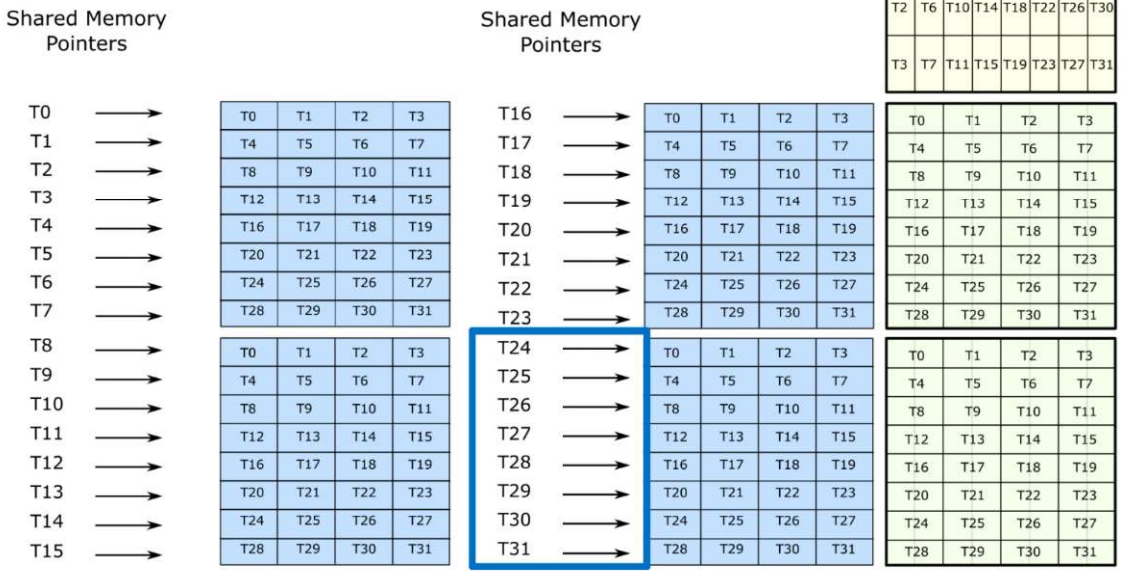
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

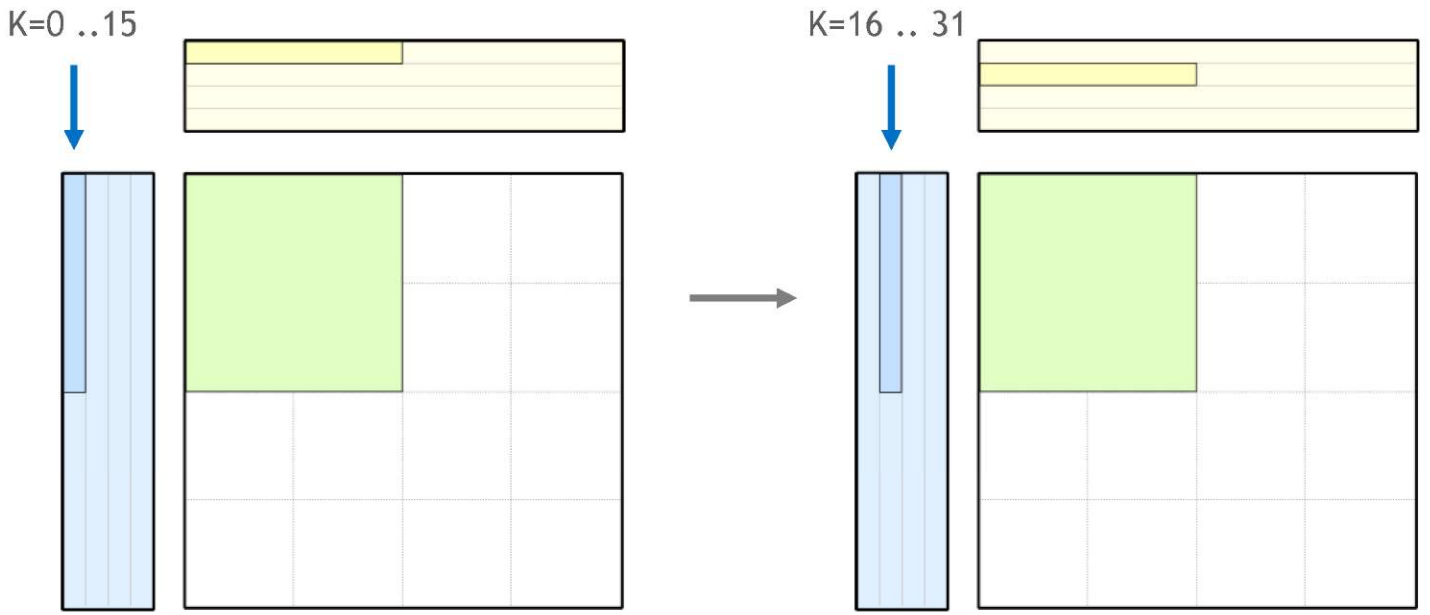


Load Matrix from Shared Memory

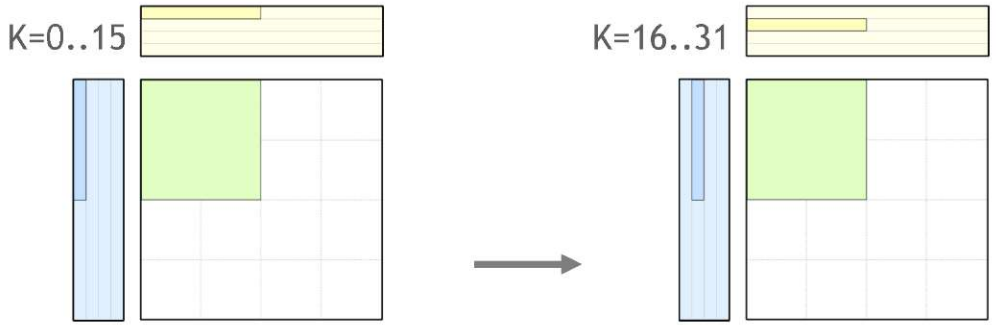




ADVANCING TO NEXT K GROUP



ADVANCING TO NEXT K GROUP



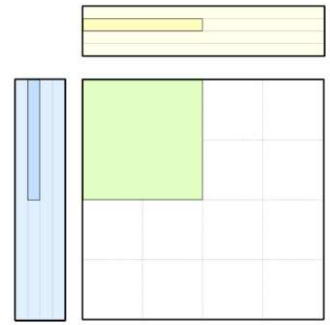
T0	T16			T1	T17		
T18	T2			T19	T3		
		T4	T20			T5	T21
		T22	T6			T23	T7
T8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		T30	T14			T31	T15

smem_ptr = row_idx * 8 + column_idx;

		T0	T16			T1	T17
		T18	T2			T19	T3
T4	T20			T5	T21		
T22	T6			T23	T7		
		T8	T24			T9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

smem_ptr = smem_ptr ^ 2;

LOADING FROM SHARED MEMORY TO REGISTERS



Logical view of threadblock tile

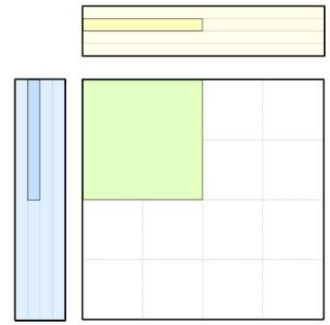
Phase 0

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

Load Matrix from Shared Memory

		T0	T16			T1	T17
		T18	T2			T19	T3
T4	T20			T5	T21		
T22	T6			T23	T7		
		T8	T24			T9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

LOADING FROM SHARED MEMORY TO REGISTERS



Phase 1

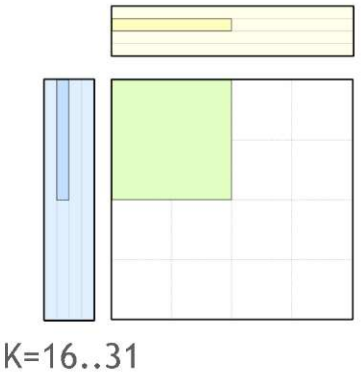
Logical view of threadblock tile

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

Load Matrix from Shared Memory

		T0	T16			T1	T17
		T18	T2			T19	T3
T4	T20			T5	T21		
T22	T6			T23	T7		
		T8	T24			T9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

LOADING FROM SHARED MEMORY TO REGISTERS



Logical view of threadblock tile

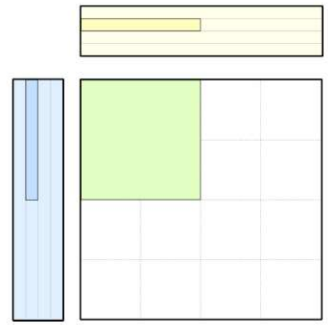
Phase 2

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

Load Matrix from Shared Memory

		T0	T16			T1	T17
		T18	T2			T19	T3
T4	T20			T5	T21		
T22	T6			T23	T7		
		T8	T24			T9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

LOADING FROM SHARED MEMORY TO REGISTERS



Logical view of threadblock tile

Phase 3

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

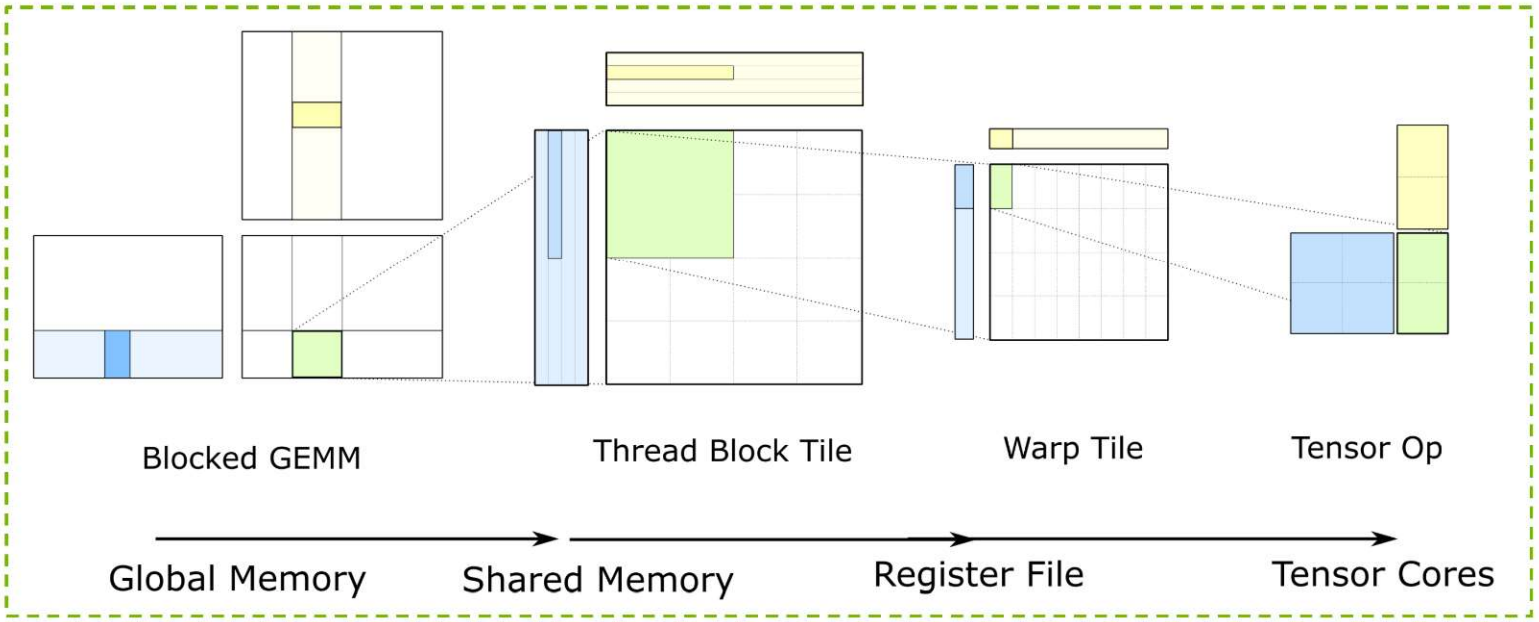
Load Matrix from Shared Memory

		T0	T16			T1	T17
		T18	T2			T19	T3
T4	T20			T5	T21		
T22	T6			T23	T7		
		T8	T24			T9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

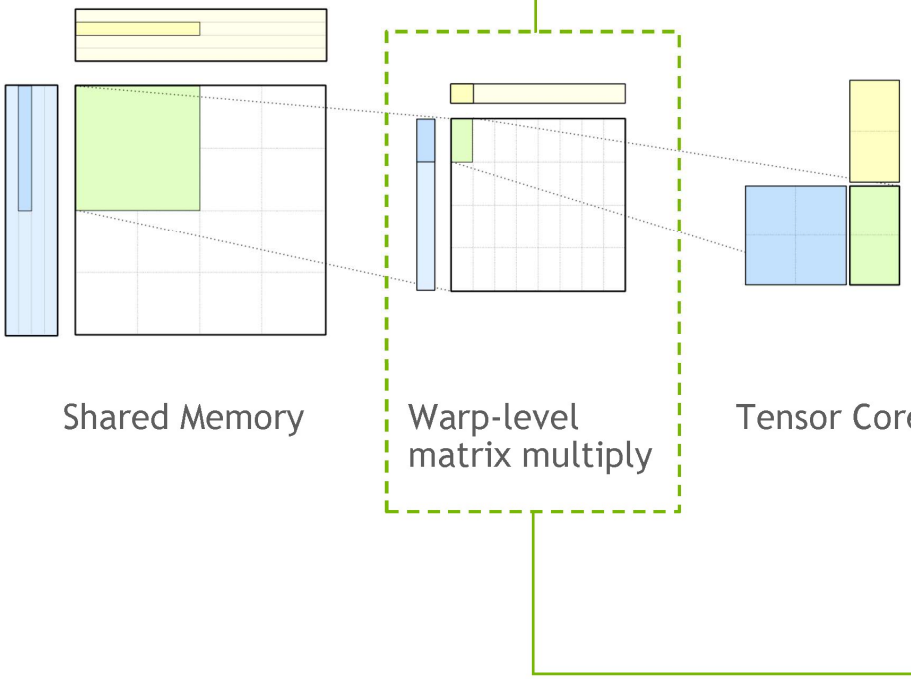
CUTLASS

CUDA C++ Templates as an Optimal Abstraction Layer for Tensor Cores

- Latency-tolerant pipeline from Global Memory
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES




```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
  GemmShape<64, 64, 16>,
  half_t, LayoutA,                  // GEMM A operand
  half_t, LayoutB,                  // GEMM B operand
  float, RowMajor                   // GEMM C operand
>;

__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];

// Construct iterators into SMEM tiles
Mma::IteratorA iter_A({smem_buffer_A, lda}, thread_id);
Mma::IteratorB iter_B({smem_buffer_B, ldb}, thread_id);

Mma::FragmentA frag_A;
Mma::FragmentB frag_B;
Mma::FragmentC accum;

Mma mma;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {


  iter_A.load(frag_A);  // Load fragments from A and B matrices
  iter_B.load(frag_B);

  ++iter_A; ++iter_B;   // Advance along GEMM K to next tile in A
                        //   and B matrices

                        // Compute matrix product
  mma(accum, frag_A, frag_B, accum);
}
```

CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES

Tile Iterator Constructors:

Initialize pointers into permuted Shared Memory buffers

Fragments:

Register-backed arrays holding each thread's data

Tile Iterator:

load() - Fetches data from permuted Shared Memory buffers

operator++() - advances to the next logical matrix in SMEM

Warp-level matrix multiply:

Decomposes a large matrix multiply into Tensor Core operations

```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
  GemmShape<64, 64, 16>,
  half_t, LayoutA,                  // GEMM A operand
  half_t, LayoutB,                  // GEMM B operand
  float, RowMajor                   // GEMM C operand
>;

__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];

// Construct iterators into SMEM tiles
Mma::IteratorA iter_A({smem_buffer_A, lda}, thread_id);
Mma::IteratorB iter_B({smem_buffer_B, ldb}, thread_id);

Mma::FragmentA frag_A;
Mma::FragmentB frag_B;
Mma::FragmentC accum;

Mma mma;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {


  iter_A.load(frag_A);  // Load fragments from A and B matrices
  iter_B.load(frag_B);

  ++iter_A; ++iter_B;   // Advance along GEMM K to next tile in A
                        //   and B matrices

                        // Compute matrix product
  mma(accum, frag_A, frag_B, accum);
}
```

Thank you.