King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 26: Programming Tensor Cores

Markus Hadwiger, KAUST

Reading Assignment #14 (until Dec 8)

Read (required):

- Look at CUDA C++ Programming Guide 13.0
  - Chapter 10.24 Warp Matrix Functions
  - Chapter 10.29 Async. Data Copies using the Tensor Memory Accelerator (TMA)
  - Chapter 10.30 Encoding a Tensor Map on Device
- Look at PTX Instruction Set Architecture 9.0 Tensor Core Instructions
  - Chapter 5.5 *Tensors*
  - Chapter 9.7.14 *Warp Level Matrix Multiply-Accumulate Instructions*
  - Chapter 9.7.15 *Async. Warpgroup Level Matrix Multiply-Accumulate Instructions*
  - Chapter 9.7.16 *TensorCore 5th Generation Family Instructions*
- Look at NVIDIA GTC talks
  - GTC 2019: *Programming Tensor Cores,* Andrew Kerr et al.

    `https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9593-cutensor-high-performance-tensor-operations-in-cuda-v2.pdf`
  - GTC 2025: *Programming Blackwell Tensor Cores with CuTe and CUTLASS,* Cris Cecka, Mihir Awatramani

    `https://www.nvidia.com/en-us/on-demand/session/gtc25-s72720/`

Quiz #3: Dec 11

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Programming Tensor Cores

NVIDIA Architectures (since first CUDA GPU)

Tesla [CC 1.x]: 2007-2009

- G80, G9x: 2007 (Geforce 8800, ...)
  GT200: 2008/2009 (GTX 280, ...)

Fermi [CC 2.x]: 2010 (2011, 2012, 2013, …)

- GF100, ... (GTX 480, ...)
  GF104, ... (GTX 460, ...)
  GF110, ... (GTX 580, ...)

Kepler [CC 3.x]: 2012 (2013, 2014, 2016, …)

- GK104, ... (GTX 680, ...)
  GK110, ... (GTX 780, GTX Titan, ...)

Maxwell [CC 5.x]: 2015

- GM107, ... (GTX 750Ti, ...); [Nintendo Switch]
  GM204, ... (GTX 980, Titan X, ...)

Pascal [CC 6.x]: 2016 (2017, 2018, 2021, 2022, …)

- GP100 (Tesla P100, ...)
- GP10x: x=2,4,6,7,8, ...
  (GTX 1060, 1070, 1080, Titan X *Pascal*, Titan Xp, ...)

Volta [CC 7.0, 7.2]: 2017/2018

- GV100, ...
  (Tesla V100, Titan V, Quadro GV100, ...)

Turing [CC 7.5]: 2018/2019

- TU102, TU104, TU106, TU116, TU117, ...
  (Titan RTX, RTX 2070, 2080 (Ti), GTX 1650, 1660, ...)

Ampere [CC 8.0, 8.6, 8.7, 8.8]: 2020

- GA100, GA102, GA104, GA106, ...; [Nintendo Switch 2]
  (A100, RTX 3070, 3080, 3090 (Ti), RTX A6000, ...)

Hopper [CC 9.0], Ada Lovelace [CC 8.9]: 2022/23

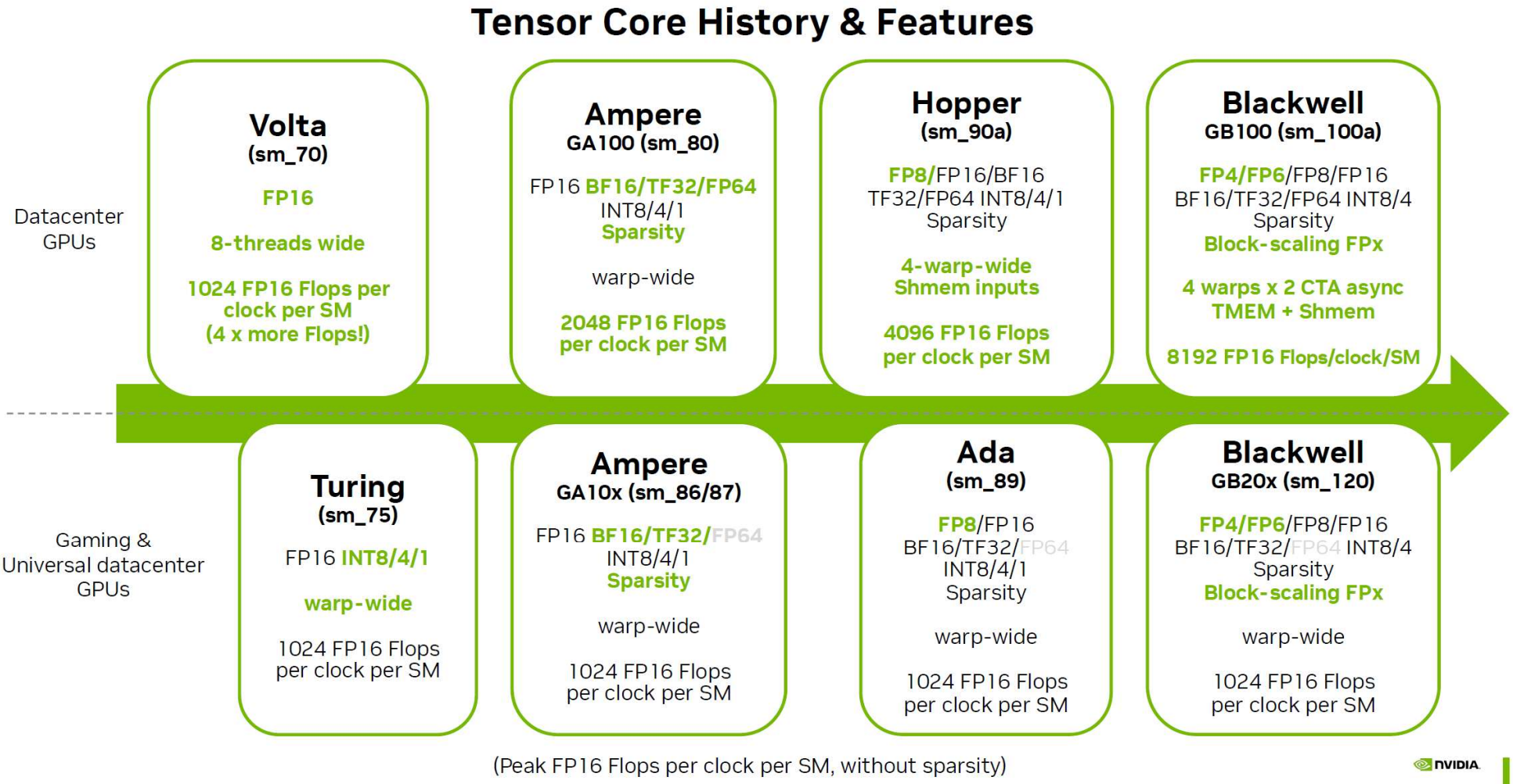
- GH100, AD102/103/104/106/107, ...
  (H100, H200, GH200, L20, L40, L40S, L2, L4,
  RTX 4080 (12/16 GB), RTX 4090, RTX 6000 (Ada), ...)

Blackwell [CC 10.0, 10.1(→11.0), 10.3, 12.0, 12.1]: 2024/2025

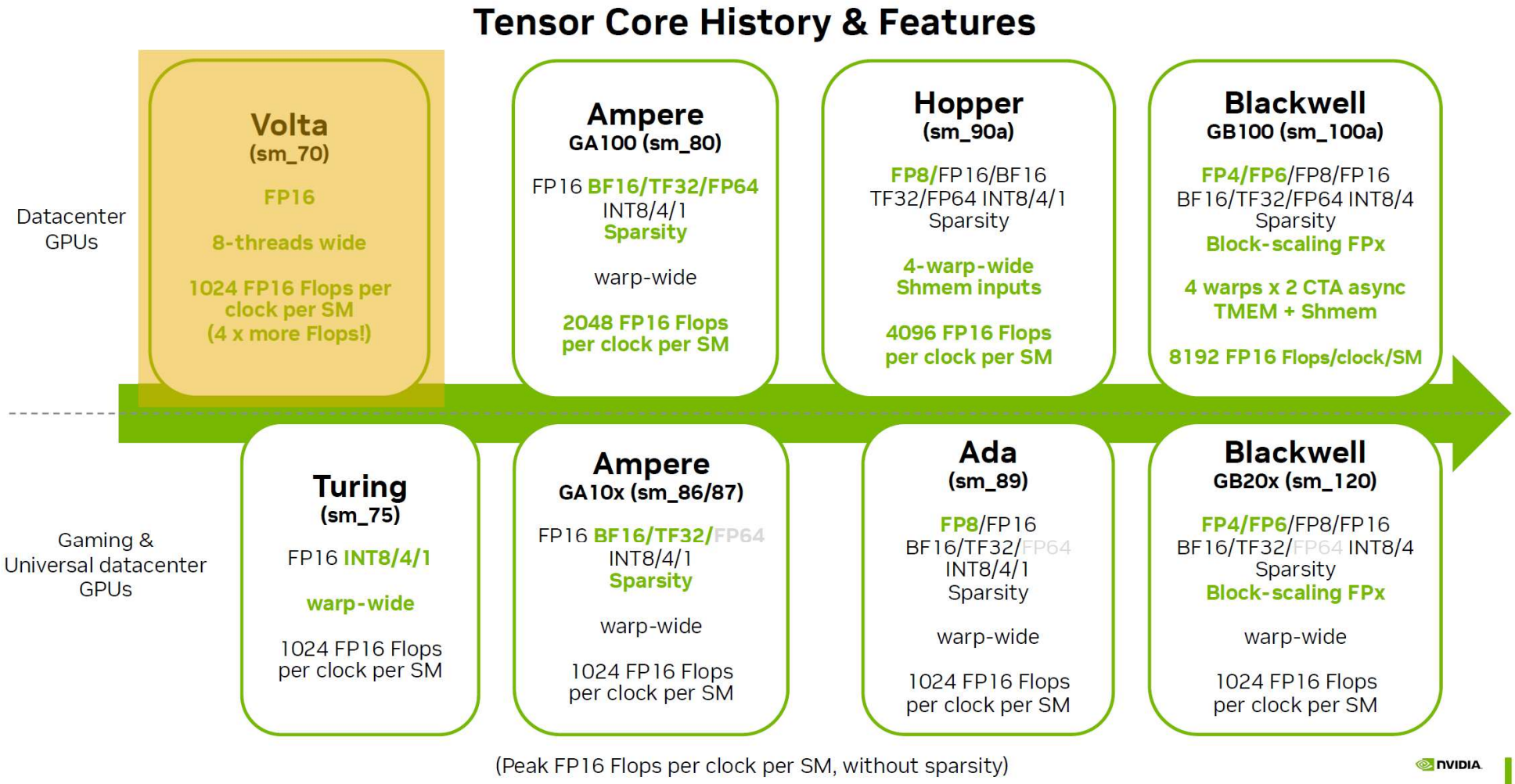
- GB100, GB200, GB202/203/205/206/207, G10, ...
  (RTX 5080/5090, HGX B200/B300, GB200/GB300 NVL72,
  RTX 4000/5000/6000 PRO Blackwell, B40, ...)

see `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units`
and `https://en.wikipedia.org/wiki/CUDA`

Tensor Cores (1st – 5th Generation)



Tensor Cores (1st – 5th Generation)



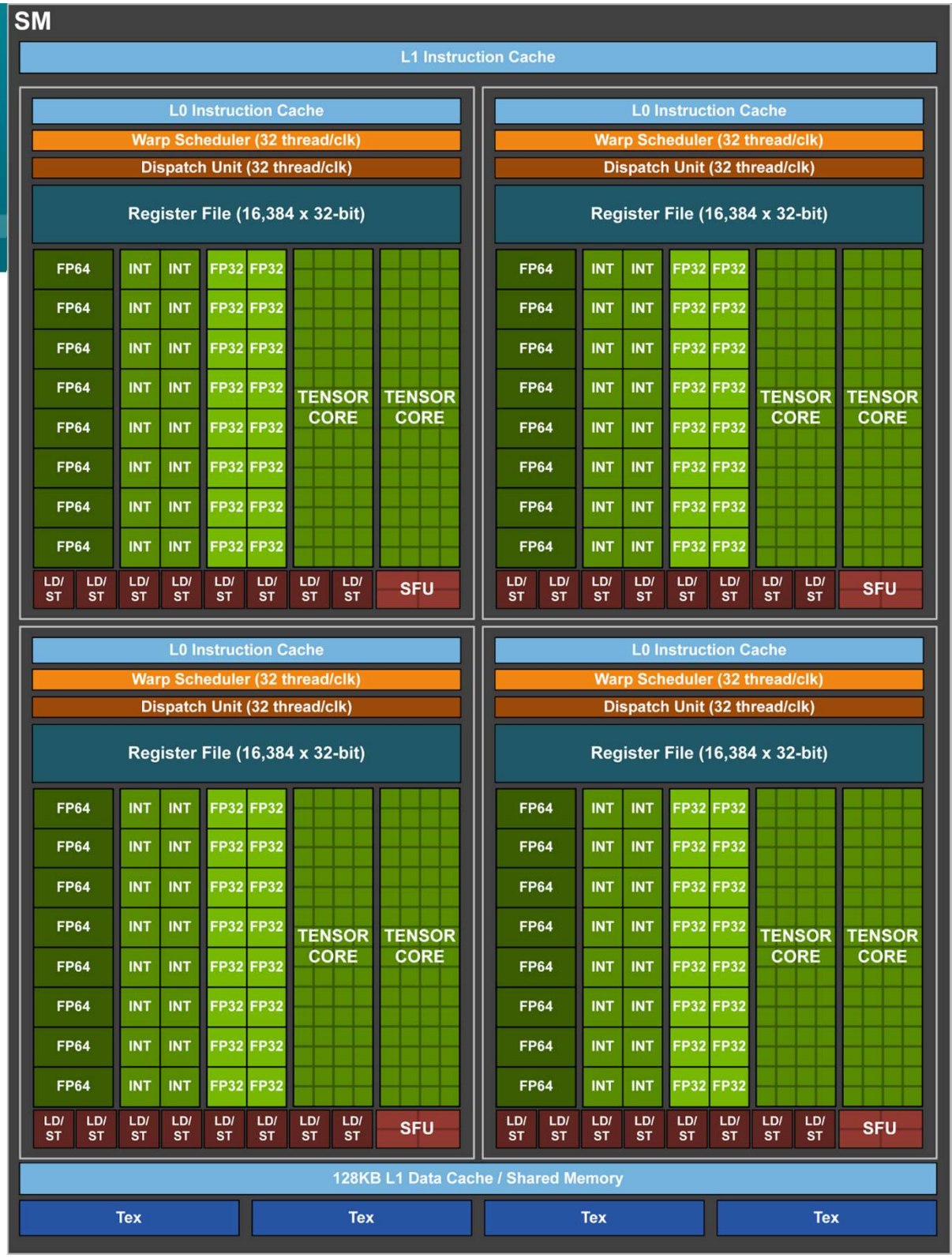
NVIDIA Volta SM

Multiprocessor: SM (CC 7.0)

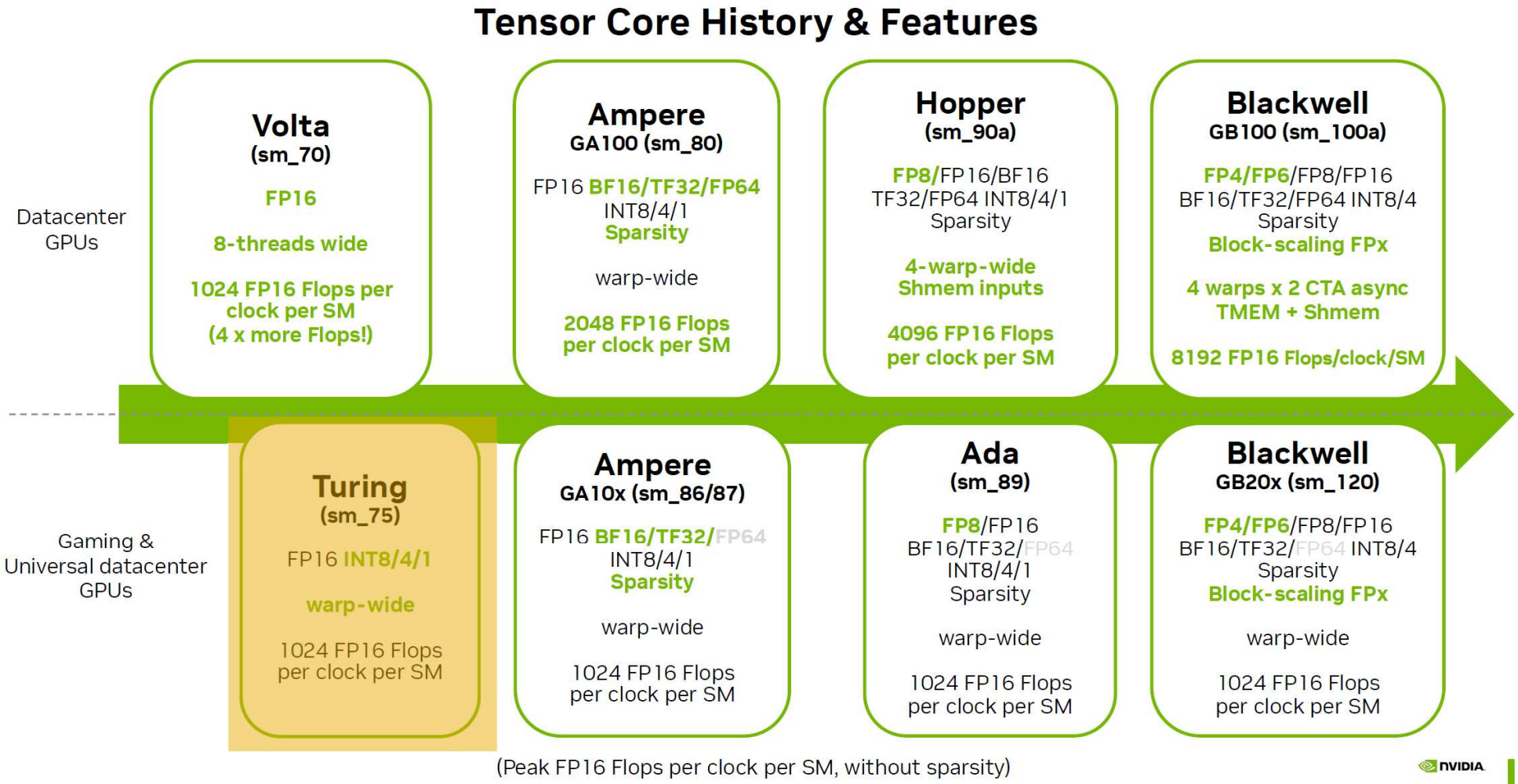
- 64 FP32 + 64 INT32 cores
- 32 FP64 cores
- 32 LD/ST units; 16 SFUs
- 8 tensor cores (FP16/FP32 mixed-precision)

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



Tensor Cores (1st – 5th Generation)



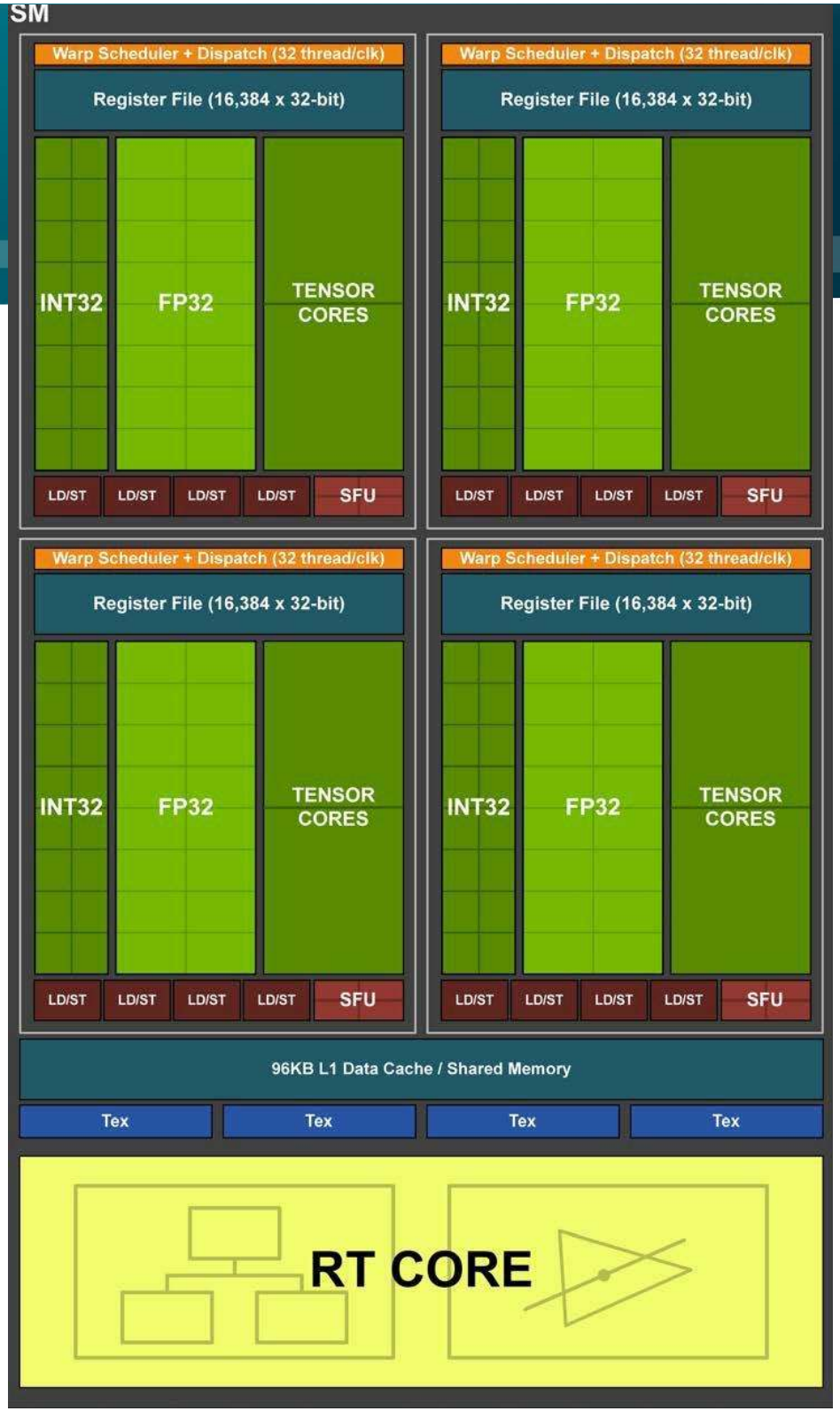
NVIDIA Turing SM

Multiprocessor: SM (CC 7.5)

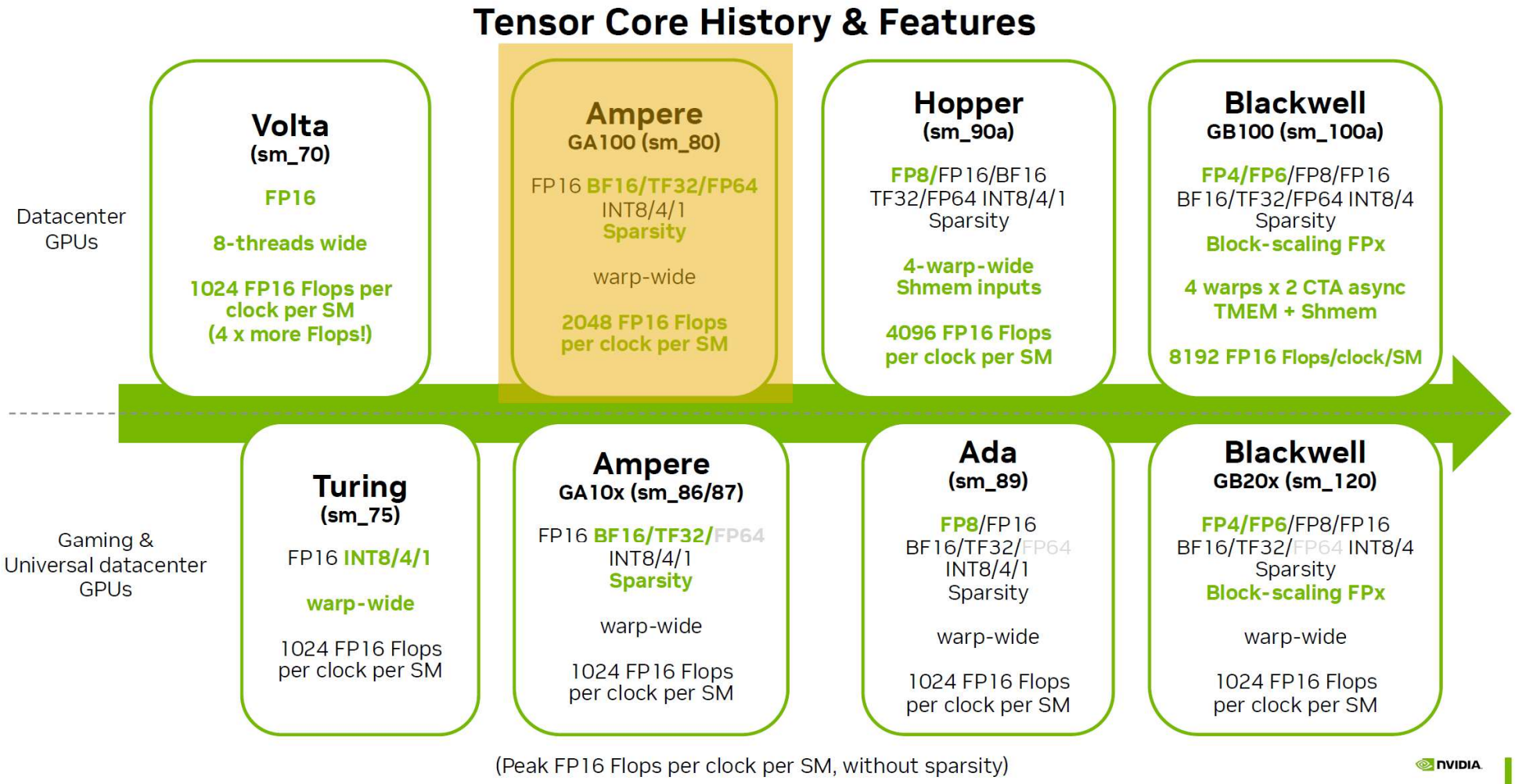
- 64 FP32 + INT32 cores
- 2 (!) FP64 cores
- 8 Turing tensor cores (FP16/32, INT4/8 mixed-precision)
- 1 RT (ray tracing) core

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



Tensor Cores (1st – 5th Generation)



NVIDIA GA100 SM

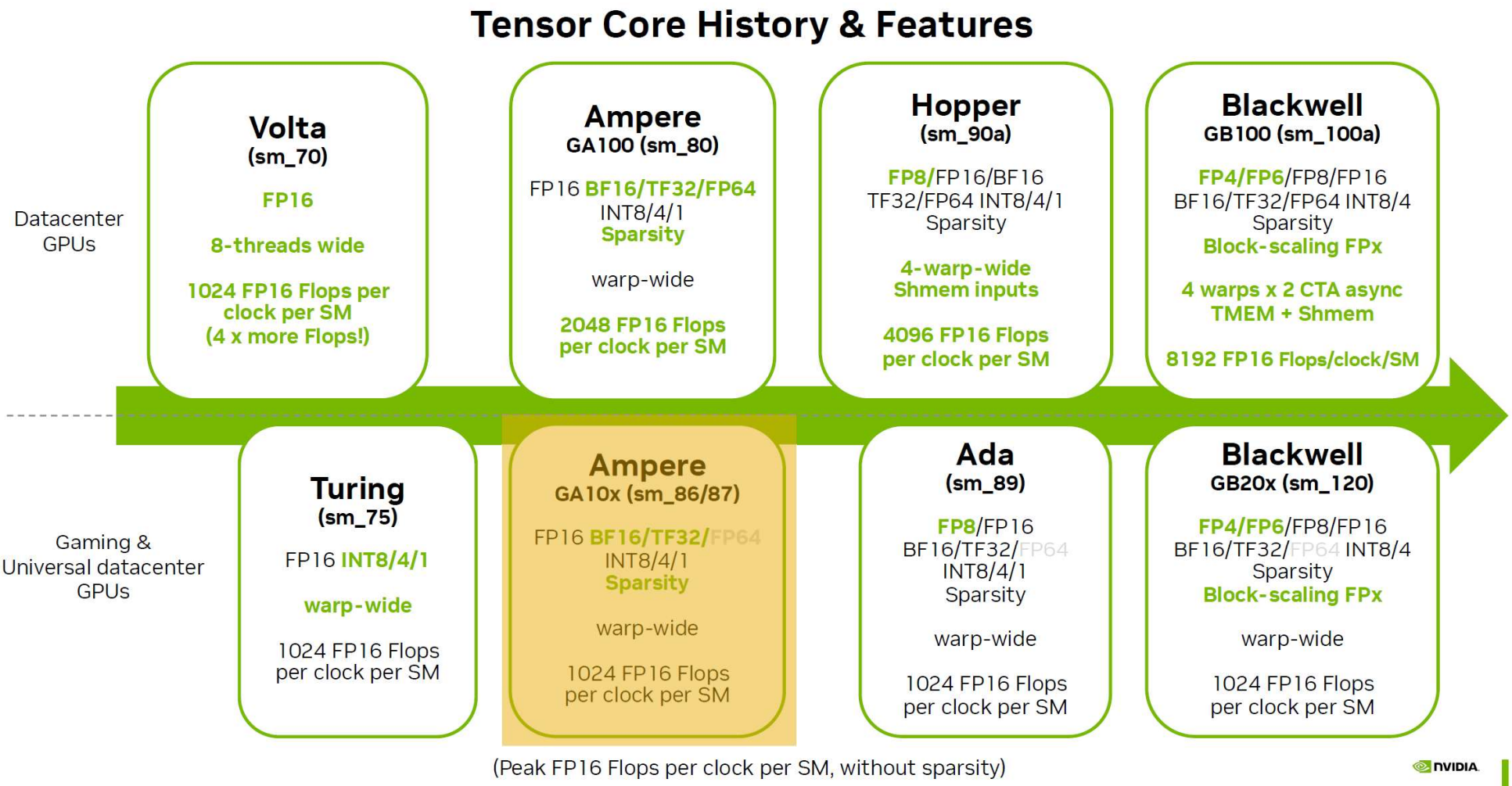
Multiprocessor: SM (CC 8.0)

- 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

Tensor Cores (1st – 5th Generation)



NVIDIA GA10x SM

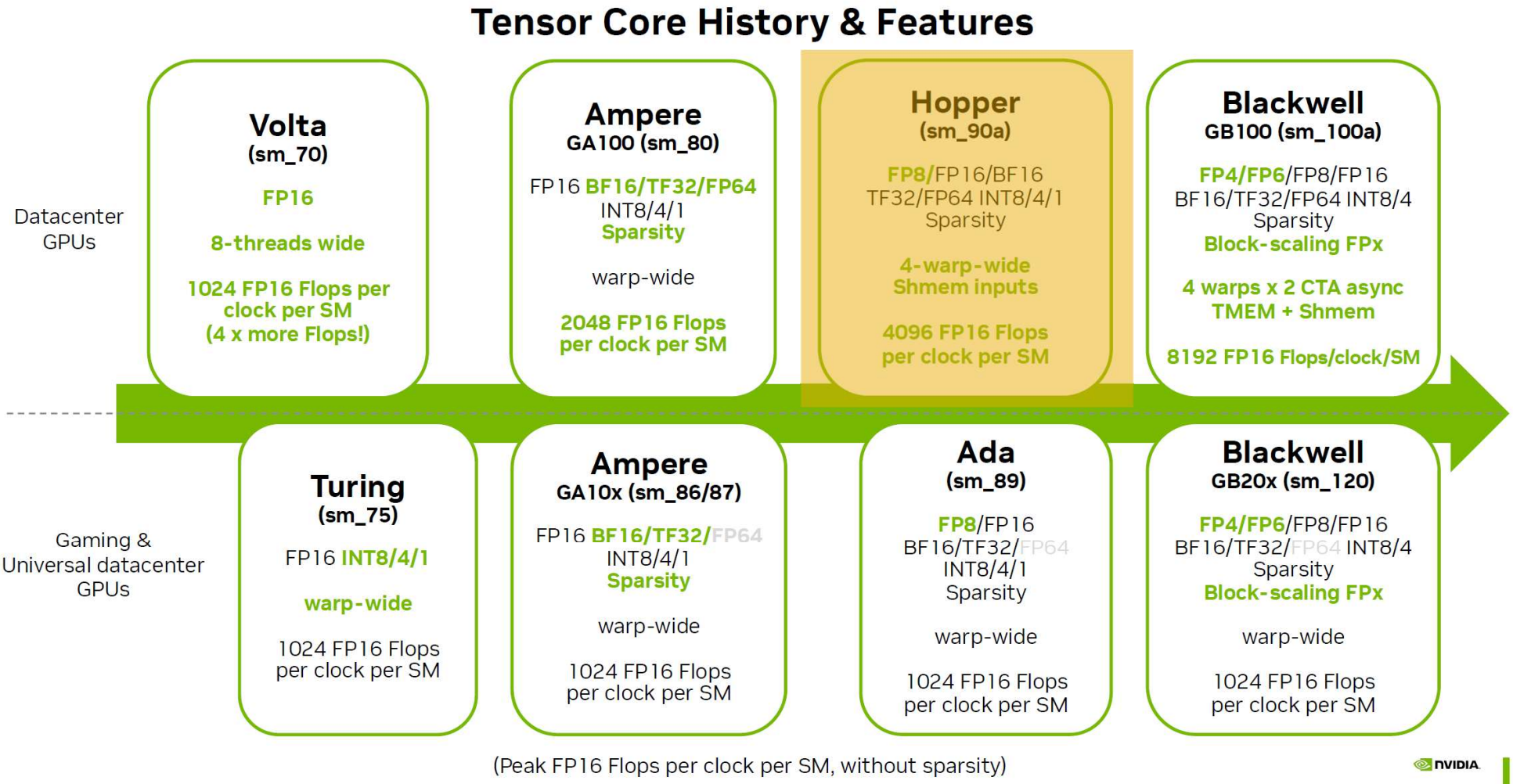
Multiprocessor: SM (CC 8.6)

- 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

Tensor Cores (1st – 5th Generation)



NVIDIA GH100 SM

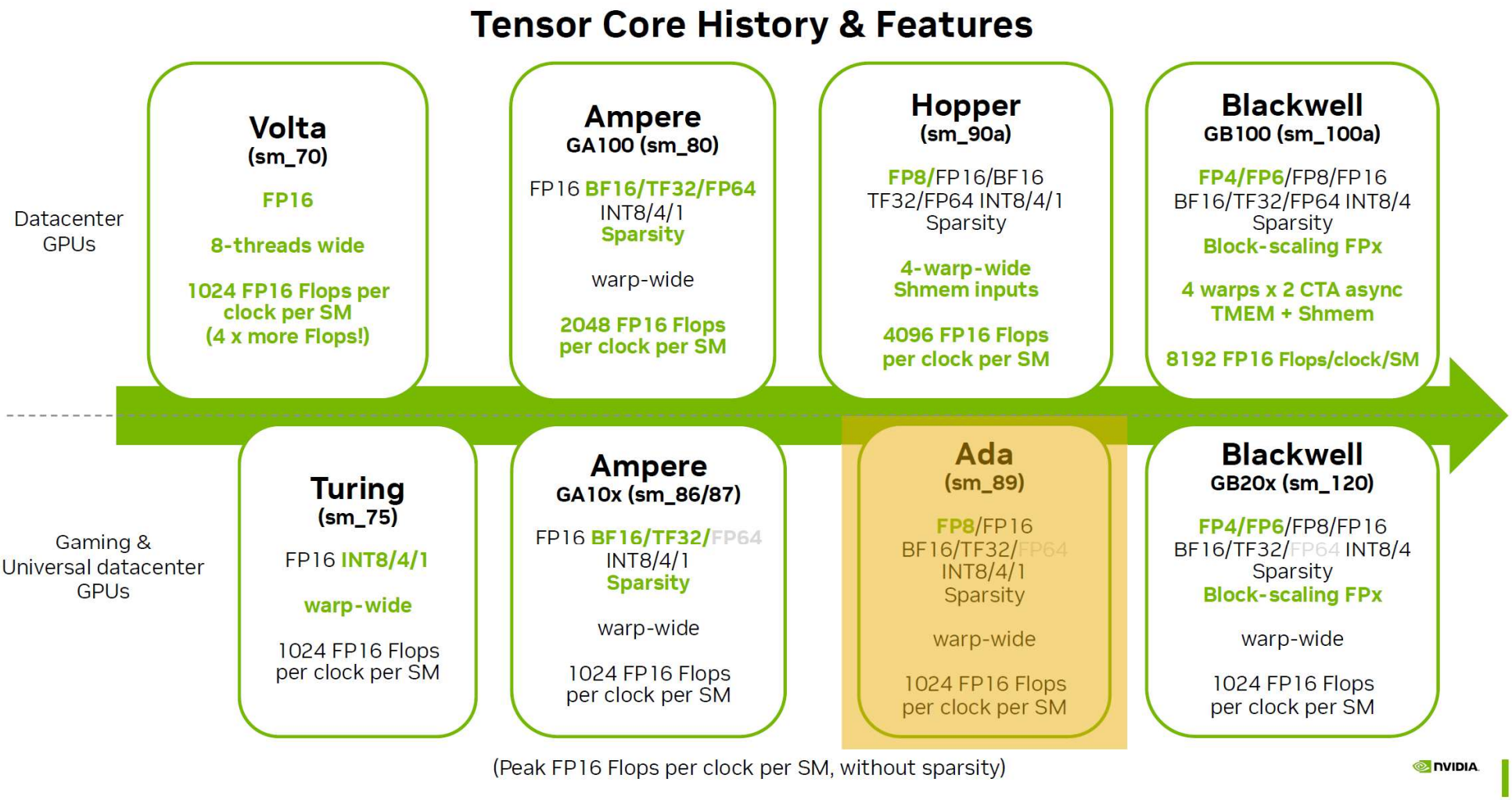
Multiprocessor: SM (CC 9.0)

- 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

Tensor Cores (1st – 5th Generation)



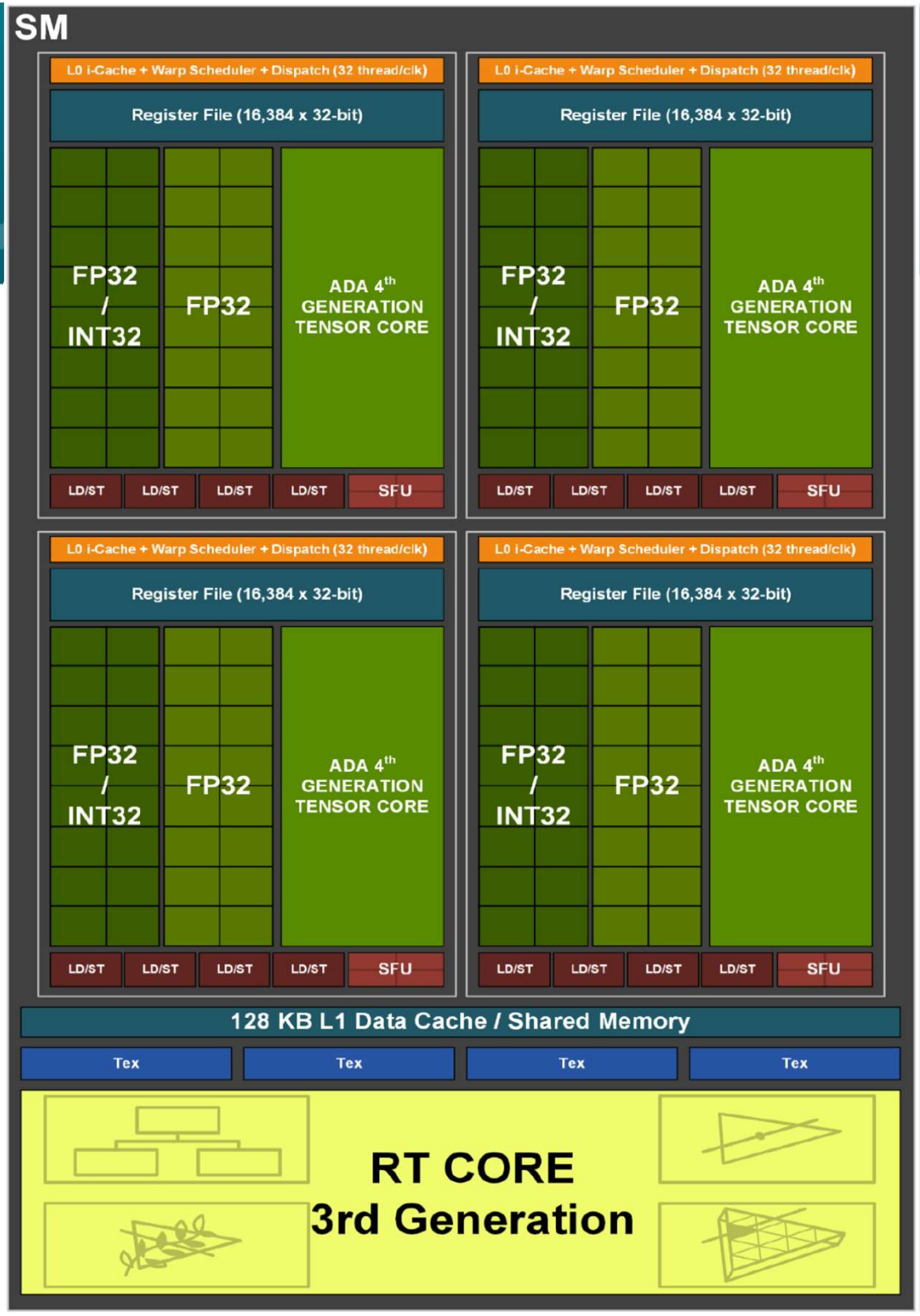
NVIDIA AD102 SM

Multiprocessor: SM (CC 8.9)

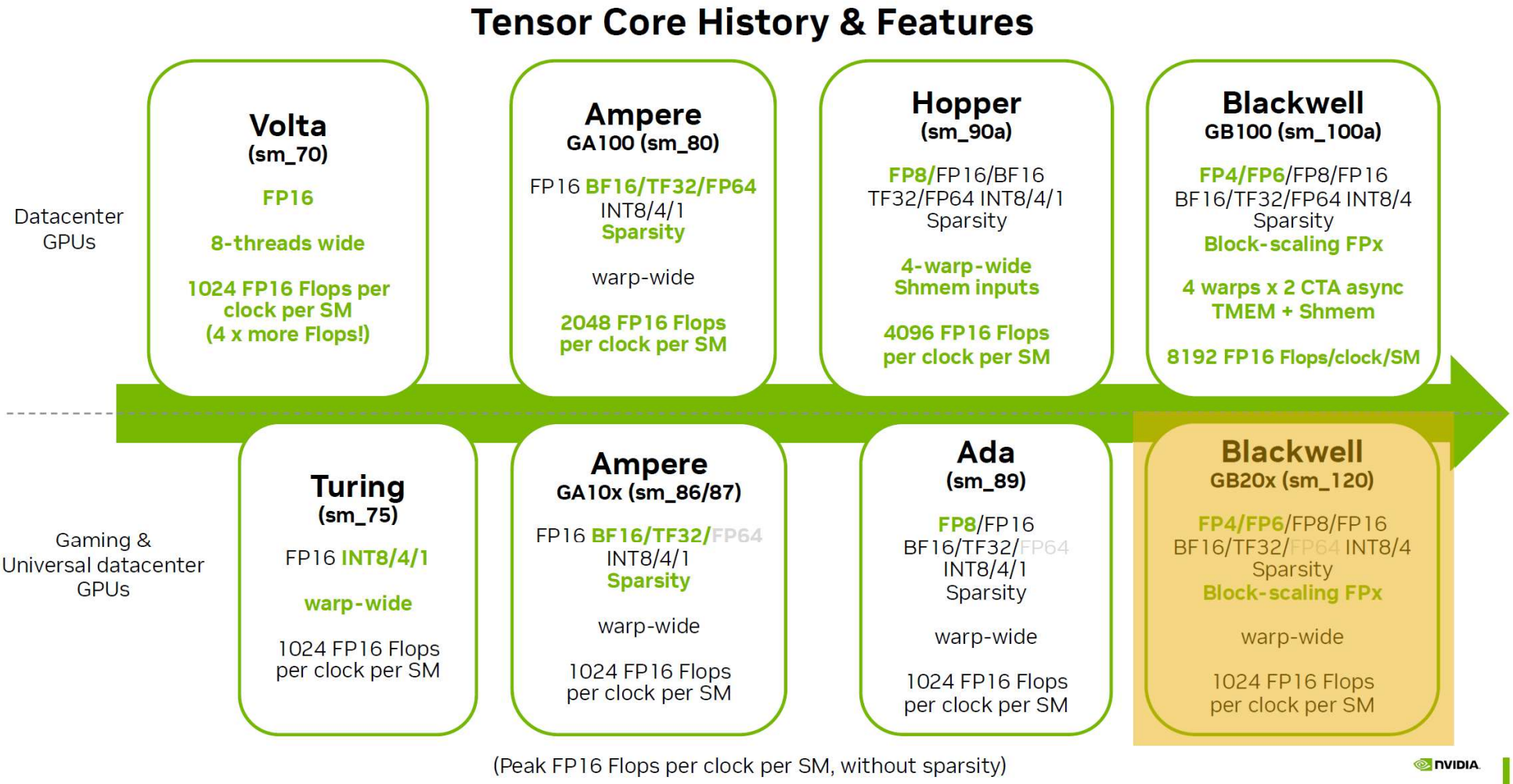
- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores (not in diagram)
- 4x 4th gen tensor cores
- 1x 3rd gen RT (ray tracing) core
- ++ thread block clusters, FP8, … (?)

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 (1st – 5th Generation)



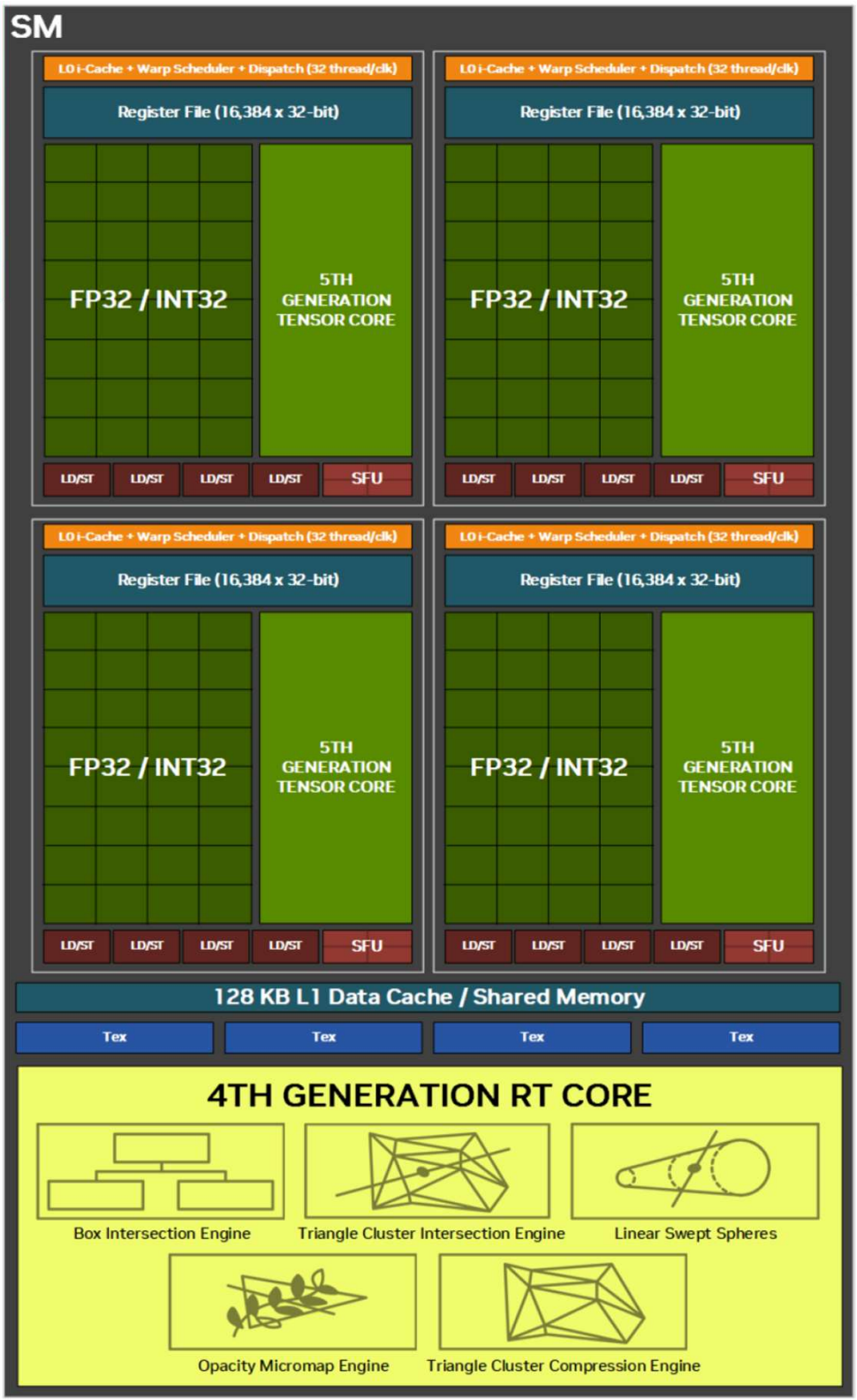
NVIDIA Blackwell SM

CC 12.0 SM (GB 202 Multiprocessor)

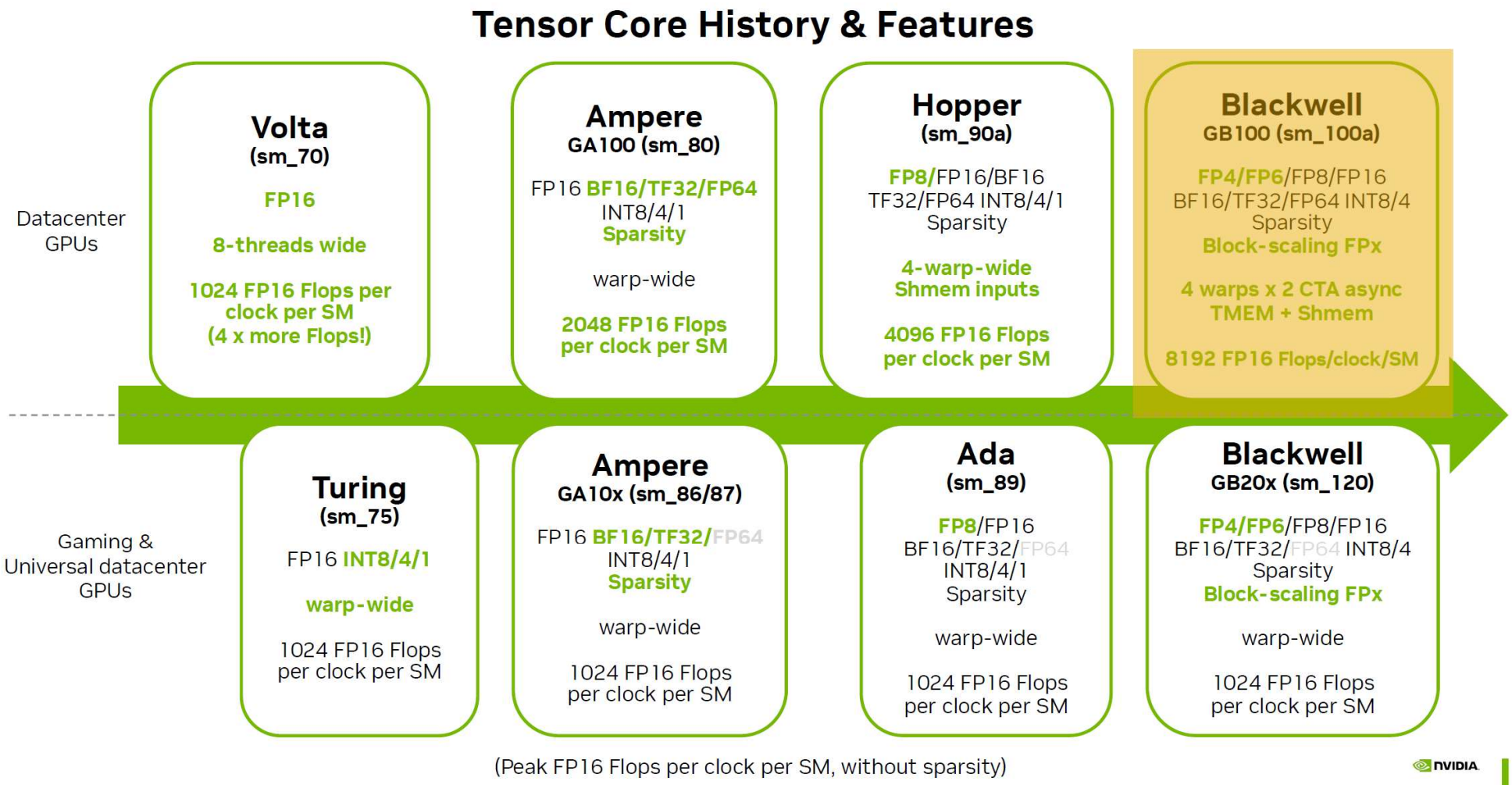
- 128 FP32/INT32 cores
- 2 FP64 cores
- 4x 5th gen tensor cores
- ++ thread block clusters, DPX insts., FP8, NVFP4, TMA

4 partitions inside SM

- 32 FP32/INT32 cores
- 4x LD/ST units each
- 1x 5th gen tensor core
- Each has: warp scheduler, dispatch unit, 16K register file



Tensor Cores (1st – 5th Generation)



NVIDIA Blackwell SM

CC 10.3 SM (GB300 Blackwell Ultra)

- 128 FP32/INT32 cores
- 64(?) FP64 cores
- 4x 5th gen tensor cores
- Tensor Memory Accelerator (TMA)
- ++ thread block clusters, DPX insts., FP8, NVFP4, 256 KB Tensor Memory (TMEM), needs 4 warps = warp group for full TMEM access (1 warp/partition)

4 partitions inside SM

- 32 FP32/INT32 cores
- 8x LD/ST units each
- 1x 5th gen tensor core
- 64 KB Tensor Memory (TMEM)
- Each has: warp scheduler, dispatch unit, 16K register file

Tensor Core APIs (1)

Low-level options

- WMMA (Warp-level matrix multiply and accumulate; 4 quad pairs per warp)
  - NVIDIA Volta – Ampere, Hopper
  - CUDA C WMMA (Chapter 10.24)
  - PTX wmma and mma (needed for some features; mma is PTX only) instructions (PTX ISA 9.0 Chapter 9.7.14, 140 pages)
  - SASS hmma instructions (not documented)
- WGMMA (Warpgroup-level matrix multiply and accumulate; 4 warps/warpgroup)
  - NVIDIA Hopper
  - PTX wgmma.mma_async (PTX ISA 9.0 Chapter 9.7.15, 53 pages); SASS
  - Tensor memory accelerator (TMA) [no actual memory; only memory transfer]
- TCGEN05.MMA (TensorCore 5th generation family instructions; single-thread issue!)
  - NVIDIA Blackwell (sm 10.0)
  - PTX tcgen05.mma (PTX ISA 9.0 Chapter 9.7.16, 116 pages); SASS
  - Tensor memory (TMEM) in the core (tcgen05.ld, tcgen05.st, tcgen05.cp); also use TMA!

Tensor Core APIs (2)

Intermediate-level options (template libraries wrapping low-level code)

- NVIDIA CUTLASS (and CuTe)
  Template abstractions for high-performance matrix-multiplies (GEMM)

  CUTLASS 4.3.1 (November 2025)
  `https://github.com/NVIDIA/cutlass`

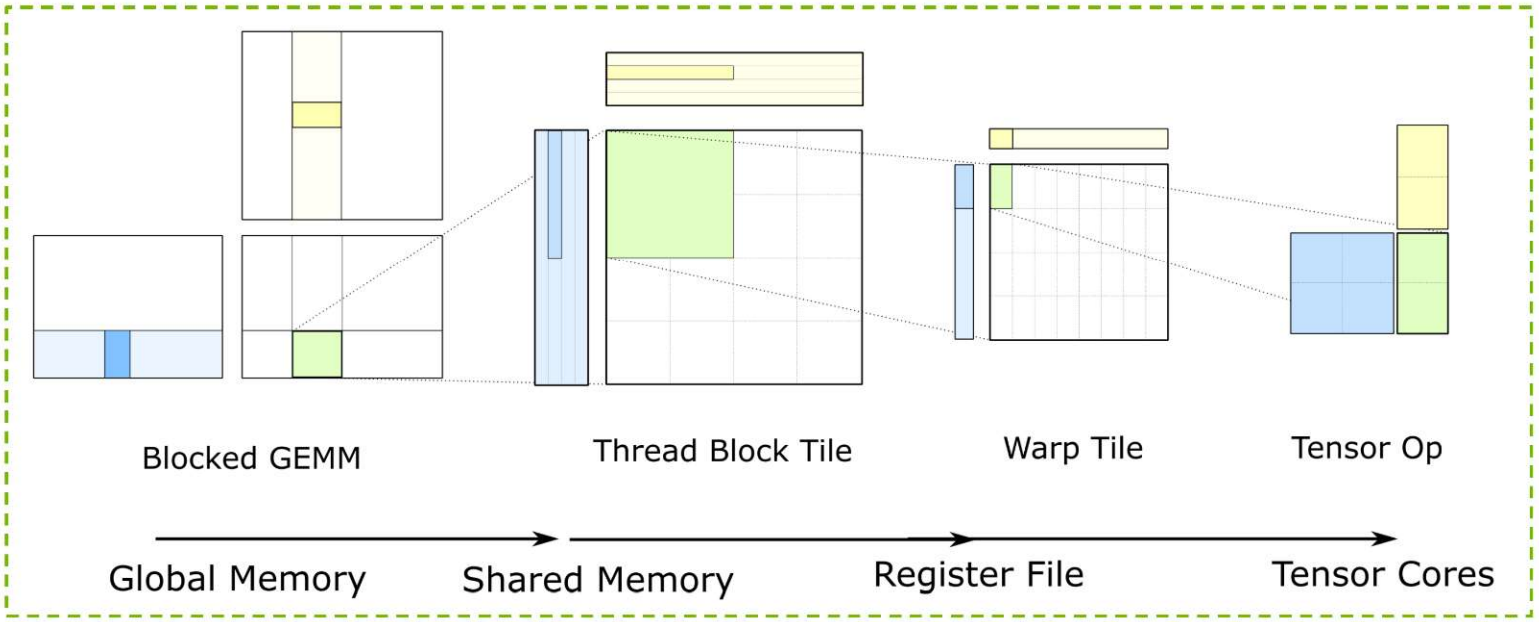
High-level options

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

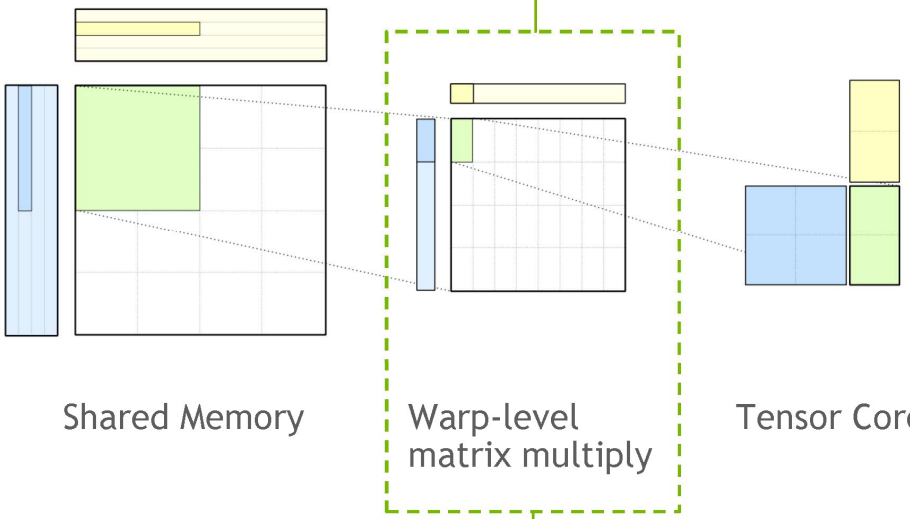
CUTLASS

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);
}
```

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

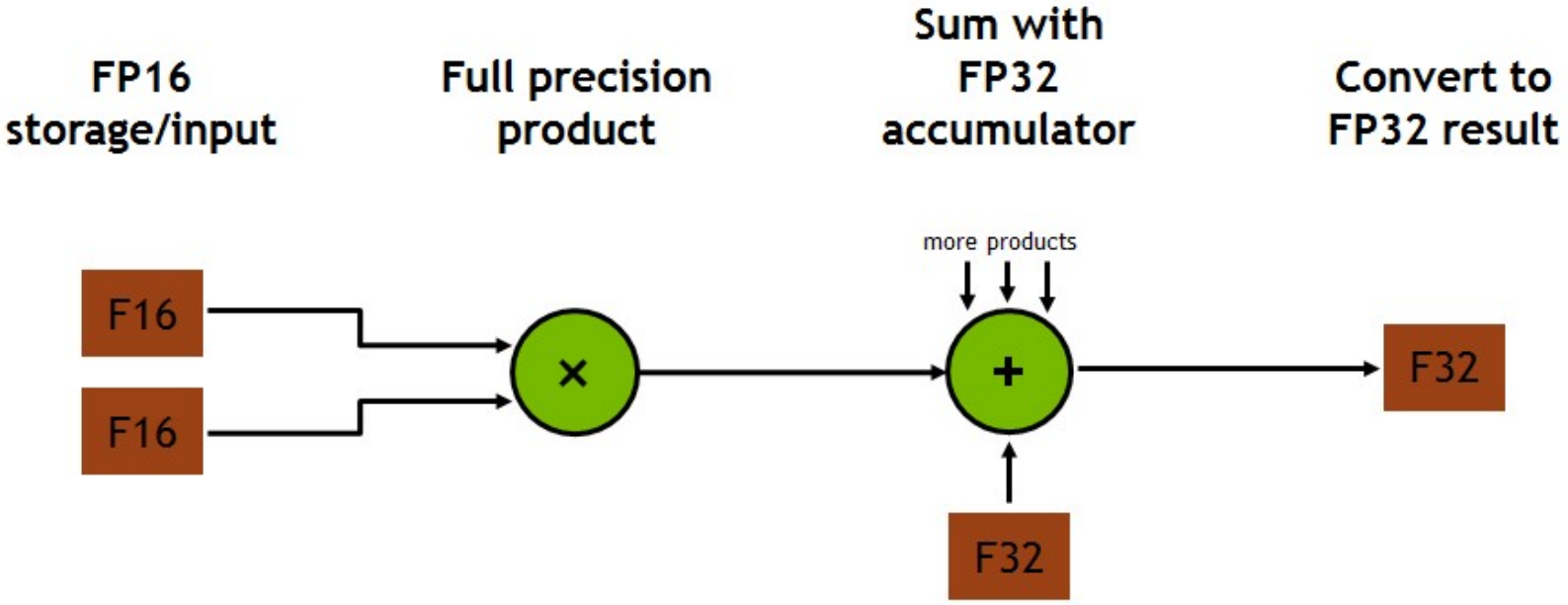
From this, build larger shapes (sizes), higher dimensionalities, ...

API currently only allows using larger shapes (16x16, ...)
in warps (wmma), warpgroups (wgmma),
warpgroups/CTA groups (tcgen05)

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!**)

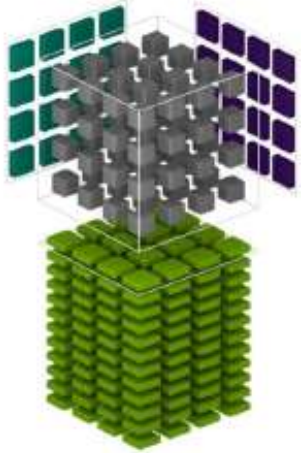


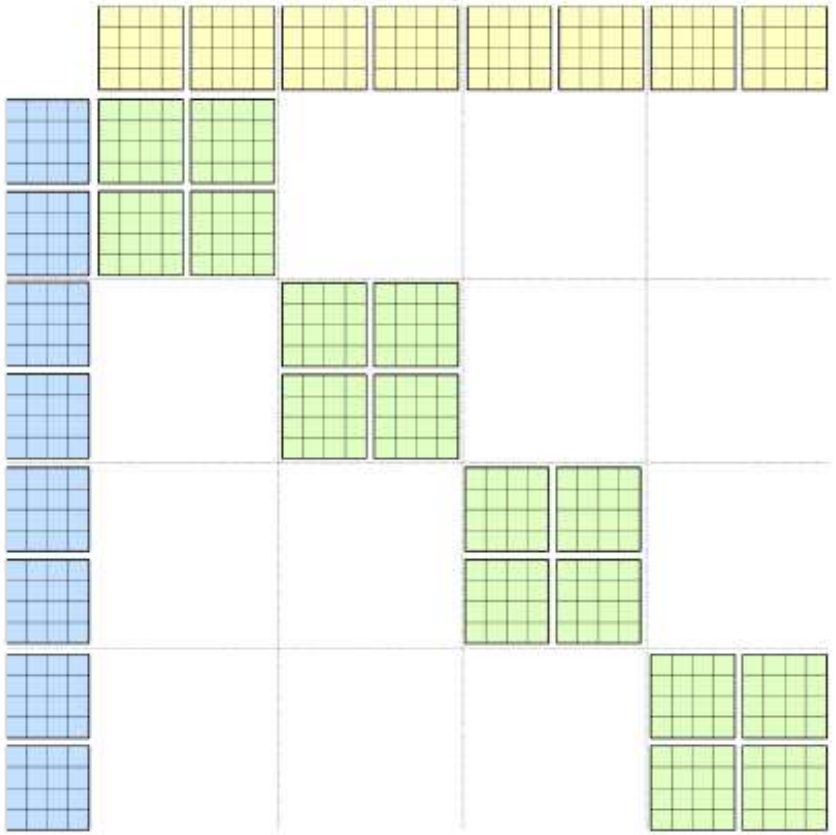
PROGRAMMING TENSOR CORES IN CUDA

mma.sync (new instruction in CUDA 10.1)

Feeding the Data Path

CUTLASS 1.3 - Native Volta Tensor Cores GEMM
(March 20, 2019)





Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

VOLTA MMA.SYNC

Warp-scoped matrix multiply instruction

mma.sync: new instruction in CUDA 10.1

- Directly targets Volta Tensor Cores

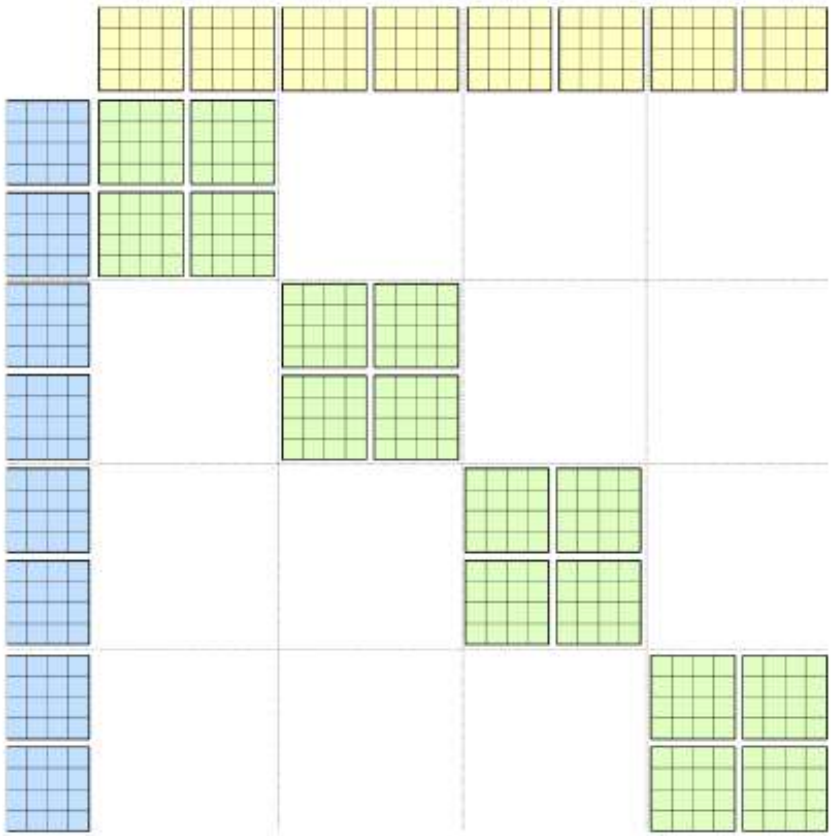
Matrix multiply-accumulate

$$D = A * B + C$$

- **A, B:** half
- **C, D:** float or half

Warp-synchronous:

- Four independent 8-by-8-by-4 matrix multiply-accumulate operations



Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

VOLTA MMA.SYNC

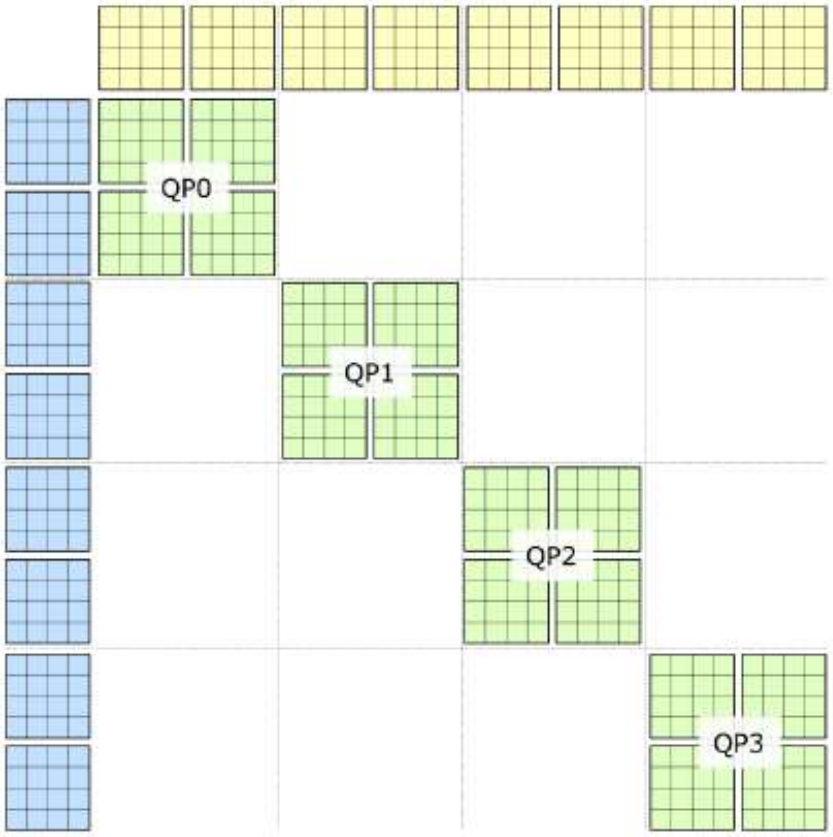
Warp-scoped matrix multiply instruction

Warp is partitioned into Quad Pairs

- QP0: `T0..T3` `T16..T19`
- QP1: `T4..T7` `T20..T23`
- QP2: `T8..T11` `T24..T27`
- QP3: `T12..T15` `T28..T31`

(eight threads each)

Each Quad Pair performs one 8-by-8-by-4 matrix multiply

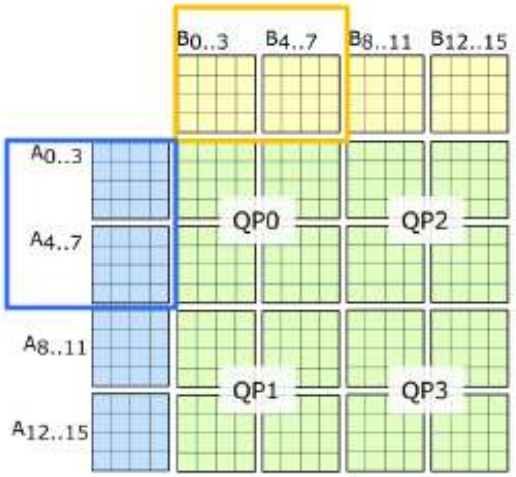


Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

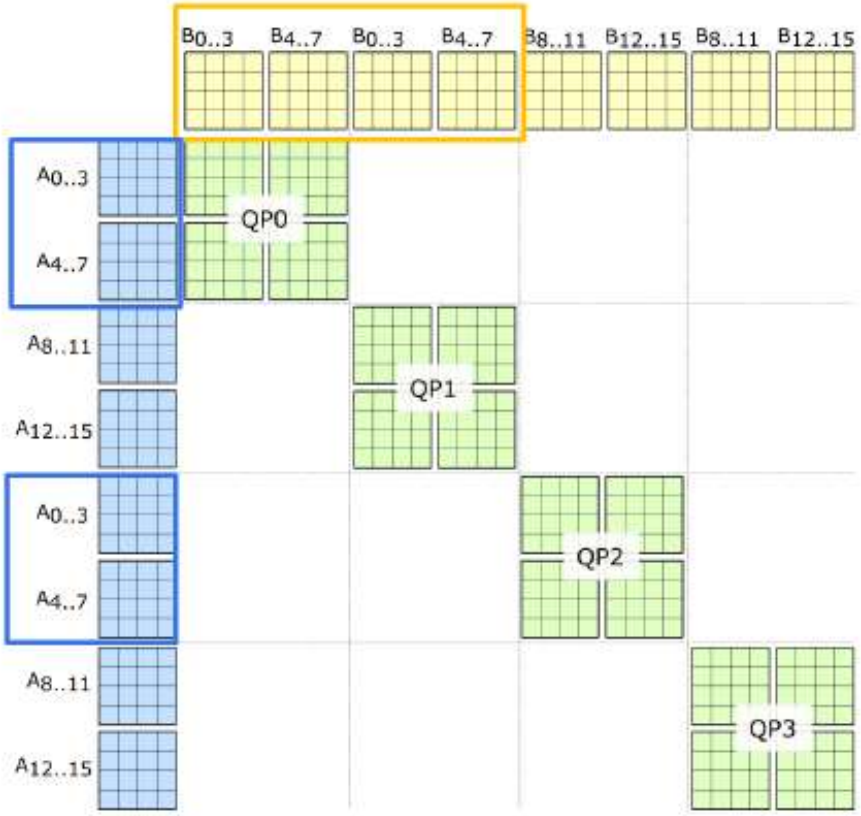
COMPOSING MATRIX MULTIPLIES

Replicate data to compute warp-wide 16-by-16-by-4 matrix product

- $A_{0..7}$: QP0,QP2 $A_{8..15}$: QP1, QP3
- $B_{0..7}$: QP0,QP1 $B_{8..15}$: QP2, QP3



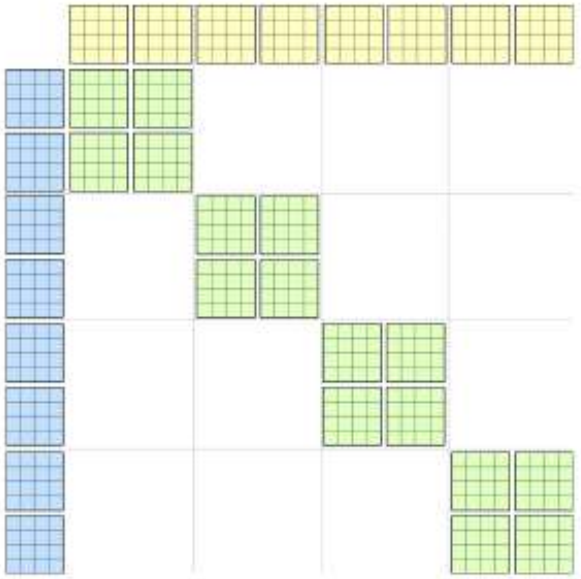
1 x mma.sync: 16-by-16-by-4



VOLTA MMA.SYNC D = A * B + C

PTX Syntax

```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype   d, a, b, c;

.alayout = {.row, .col};
.blayout = {.row, .col};
.ctype   = {.f16, .f32};
.dtype   = {.f16, .f32};
```



```
d: 8 x .dtype
a: 4 x .f16
b: 4 x .f16
c: 8 x .ctype
```

Note: .f16 elements must be packed into .f16x2

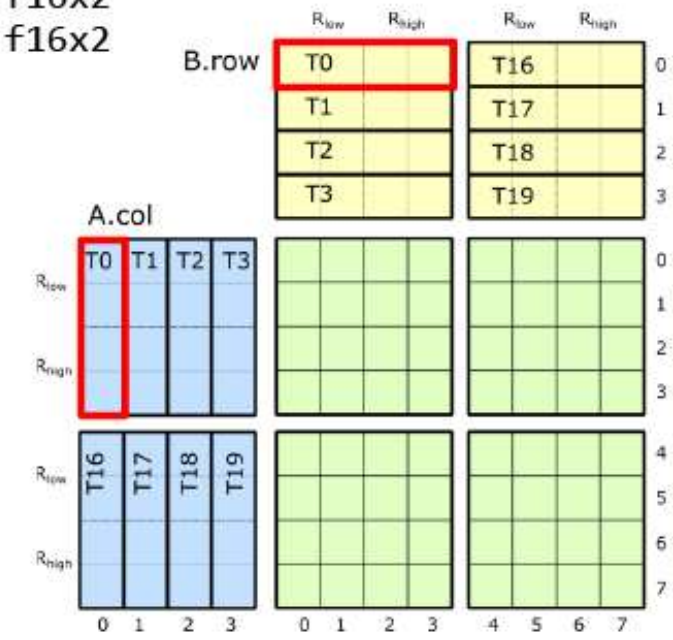
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma

THREAD-DATA MAPPING - F16 MULTIPLICANDS

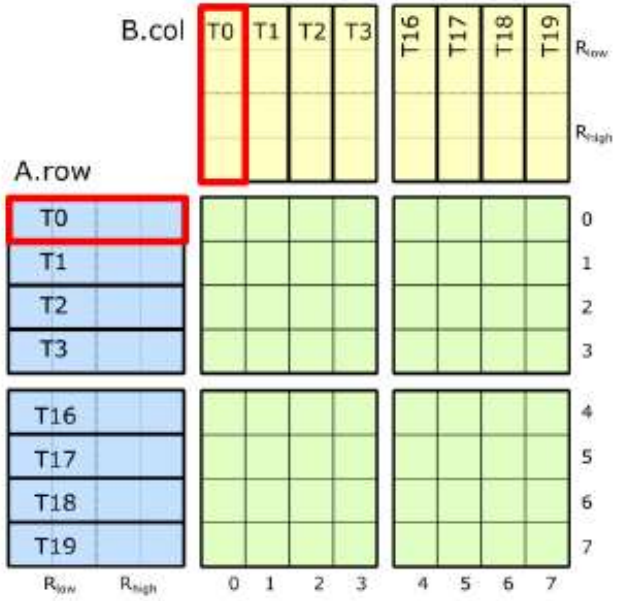
Distributed among threads in quad pair (QP0 shown)

```
mma.sync.aligned.m8n8k4.alayout.blayout.dtype.f16.f16.ctype   d, a, b, c;

.alayout = {.row, .col};
.blayout = {.row, .col};
```

a: 2 x .f16x2
b: 2 x .f16x2



COL-ROW ("NT")

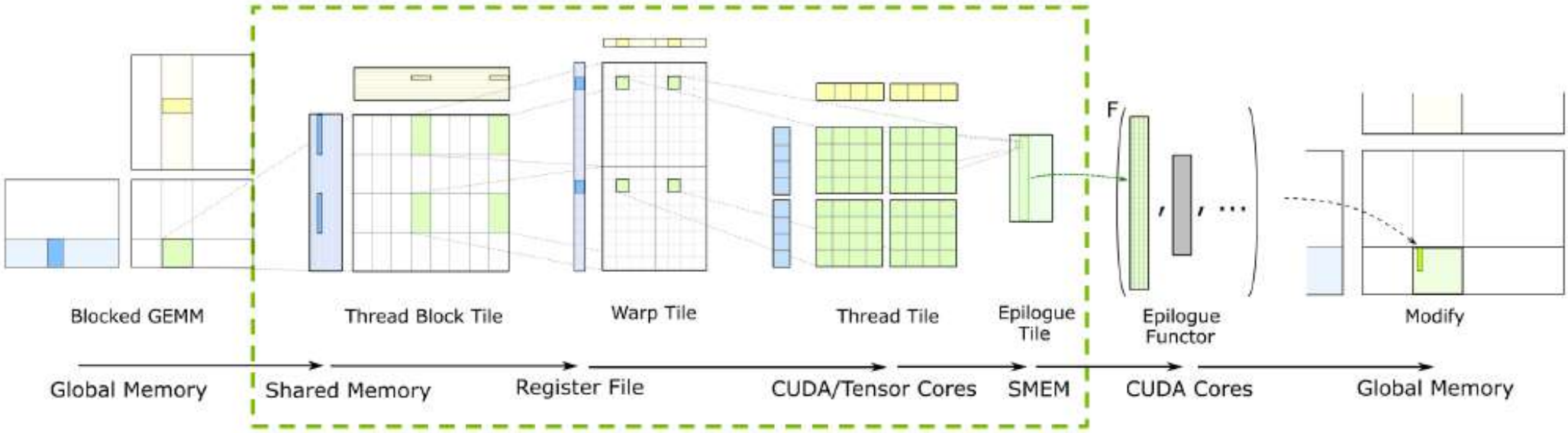


ROW-COL ("TN")

Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

FEEDING THE DATA PATH

Efficiently storing and loading through shared memory



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

CONFLICT-FREE ACCESS TO SHARED MEMORY

Efficiently storing and loading through shared memory

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: 128 bit access size

- Each phase processes 8 consecutive threads of a warp

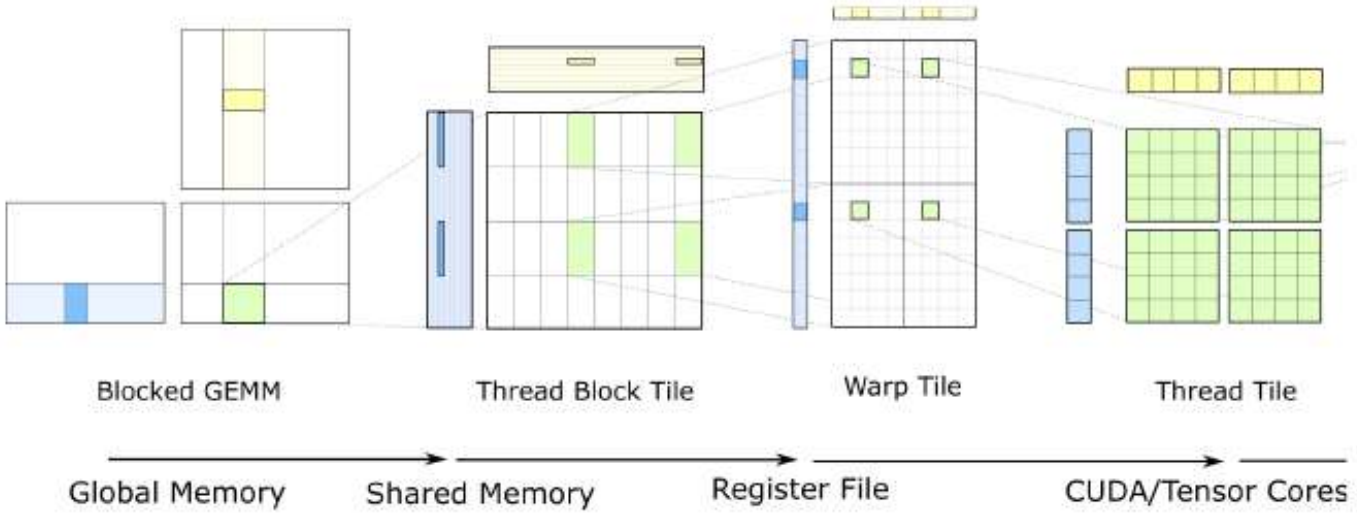
Slide borrowed from: Guillaume Thomas-Collignon and Paulius Micikevicius. "Volta Architecture and performance optimization." GTC 2018.
http://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf

FEEDING THE DATA PATH

Efficiently storing and loading through shared memory

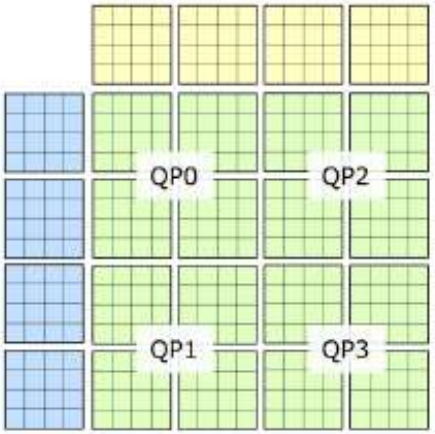
Must move data from shared memory to registers as efficiently as possible

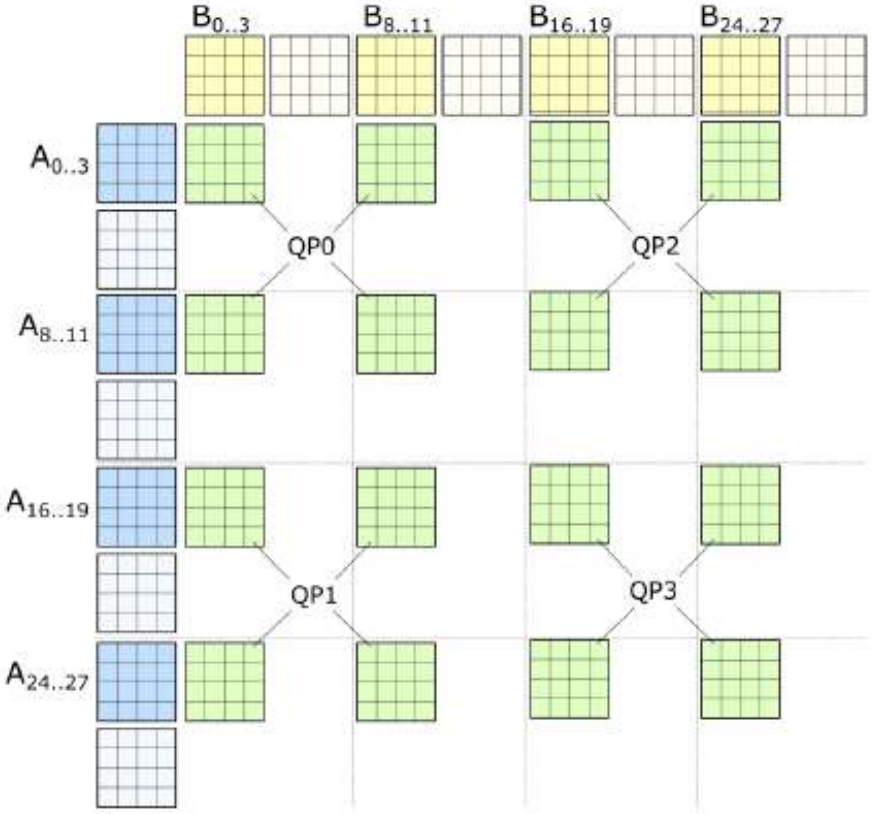
- 128 bit access size
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



MMA.SYNC GEMM: SPATIALLY INTERLEAVED

Accumulator tiles may not be contiguous



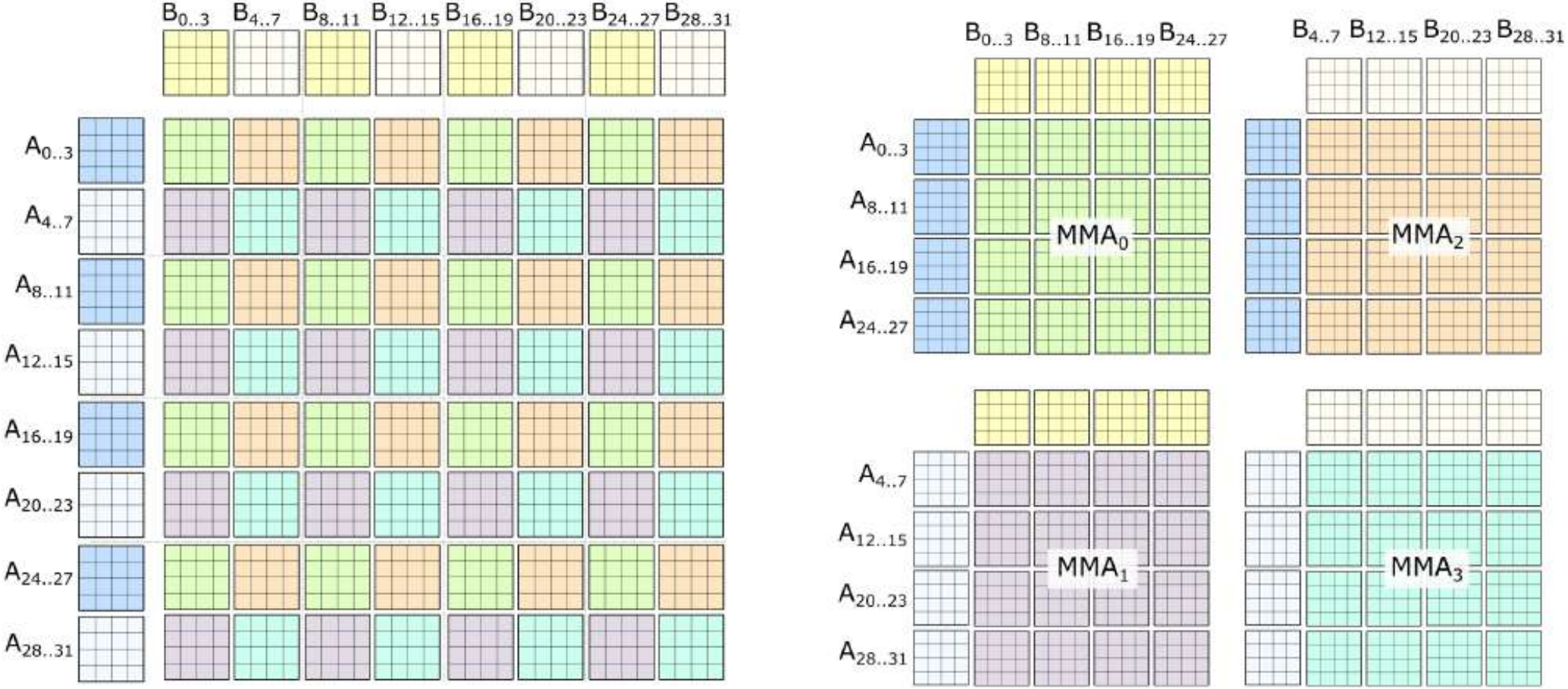


1 x mma.sync: 16-by-16-by-4

Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

PROGRAMMING TENSOR CORES:
NATIVE VOLTA TENSOR CORES WITH CUTLASS

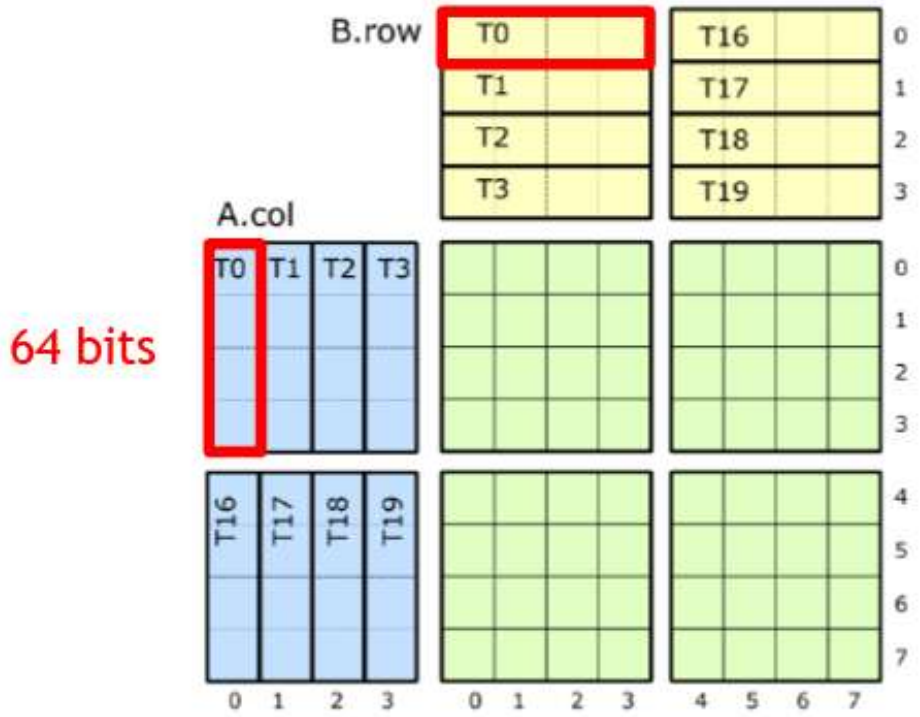
MMA.SYNC GEMM: SPATIALLY INTERLEAVED



4 x mma.sync: 32-by-32-by-4 (spatially interleaved)

Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

THREAD-DATA MAPPING - F16 MULTIPLICANDS

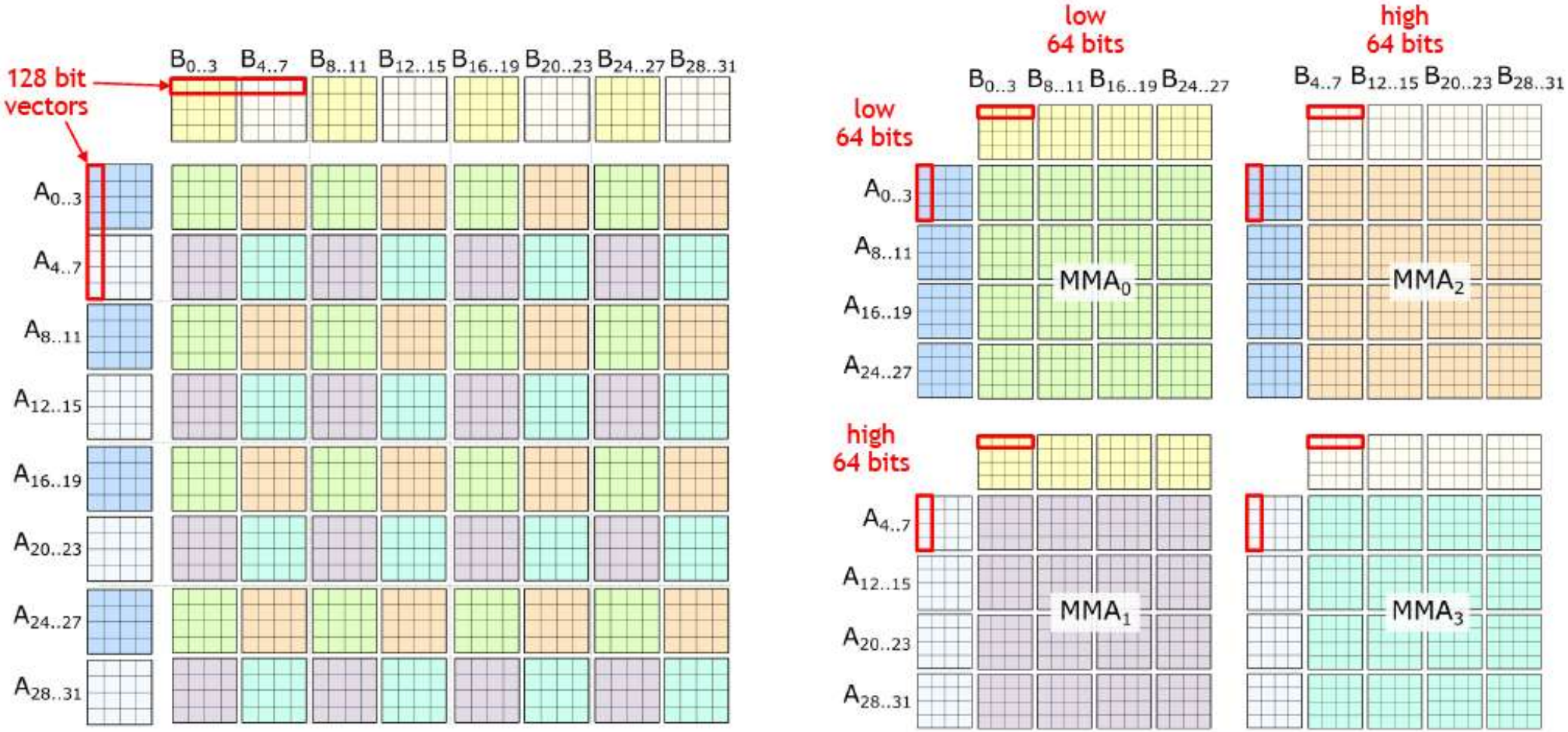


COL-ROW ("NT")

PROGRAMMING TENSOR CORES:

NATIVE VOLTA TENSOR CORES WITH CUTLASS

SPATIALLY INTERLEAVED: 128 BIT ACCESSES

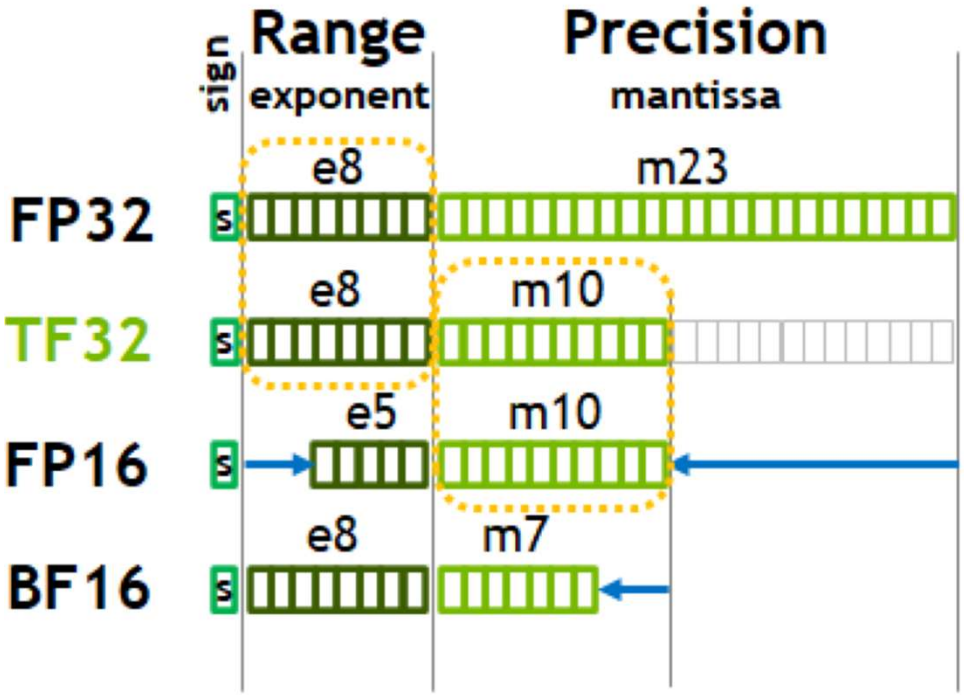


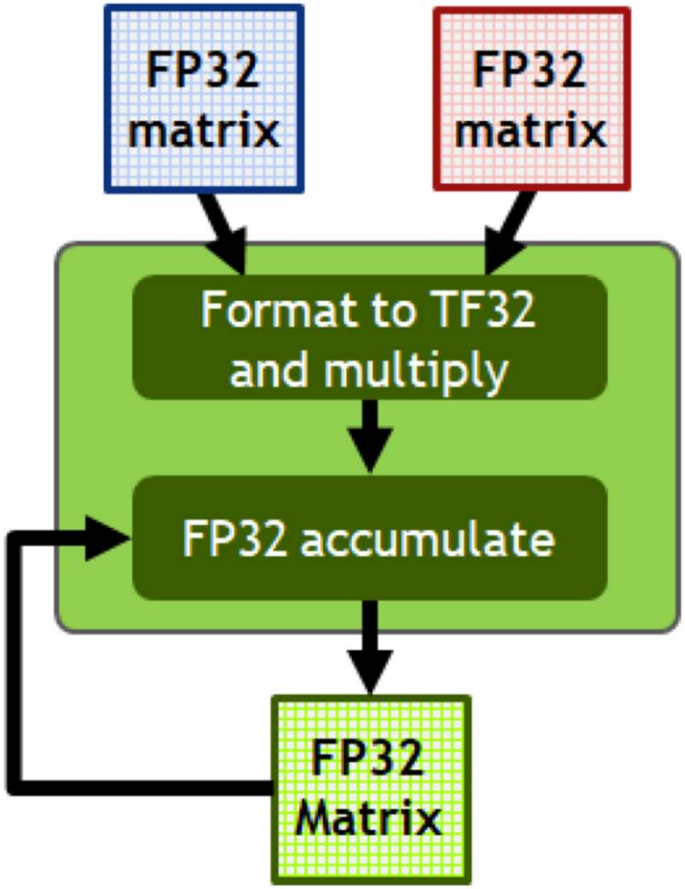
4 x mma.sync: 32-by-32-by-4 (spatially interleaved)

Andrew Kerr, Timmy Liu, Mostafa Hagog,
Julien Demouth, John Tran

Ampere Tensor Cores: Mixed Precision

New in Ampere: TF32, BF16, FP64





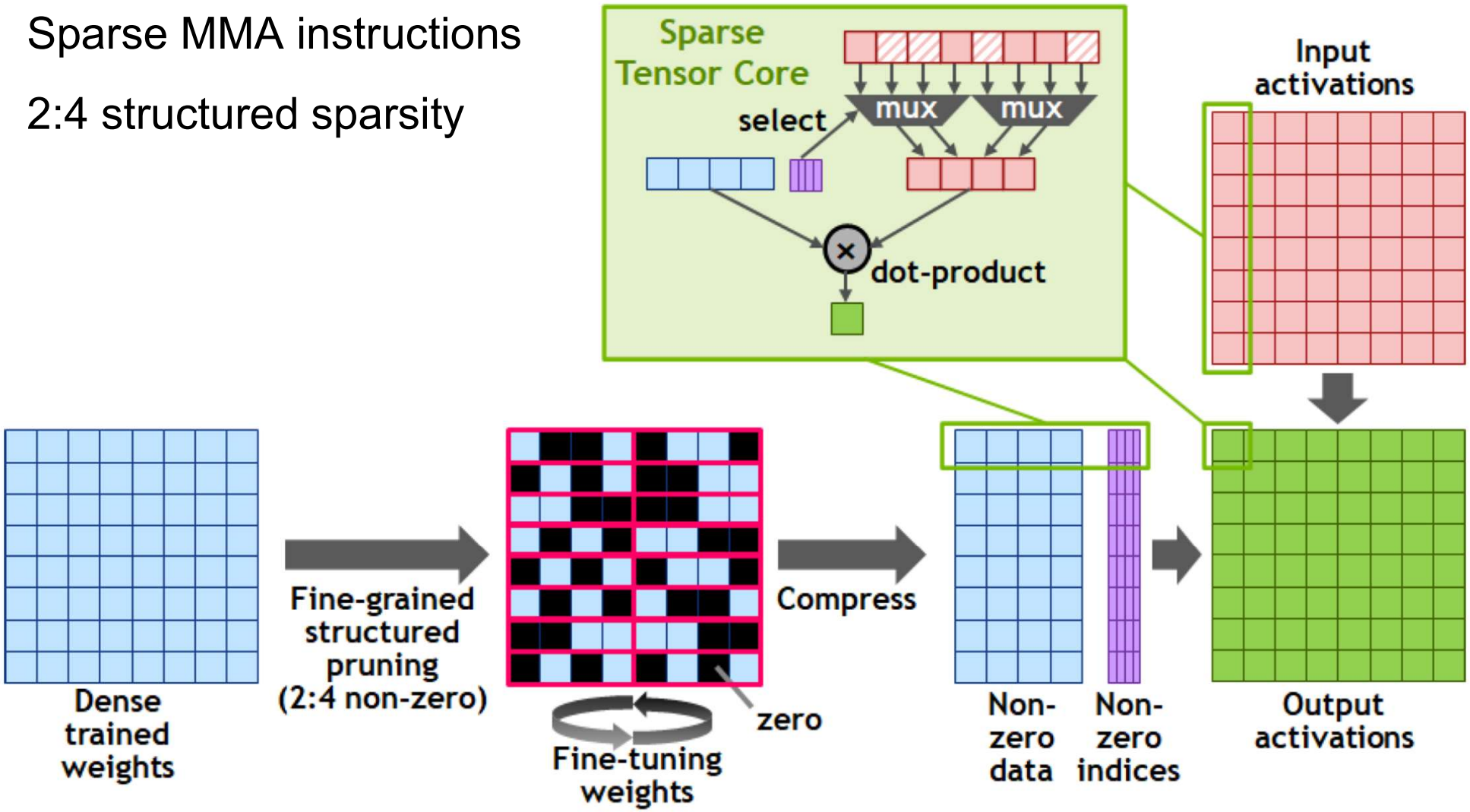
plus FP64 (new in Ampere; GA100 only)

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

Ampere Tensor Cores: Sparsity Support

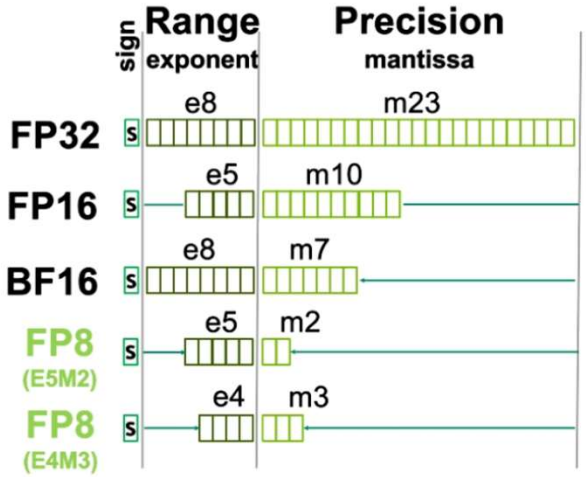
Sparse MMA instructions

2:4 structured sparsity

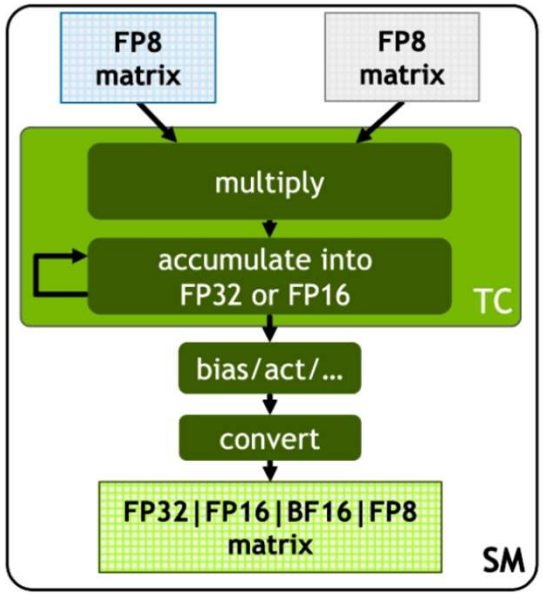


Tensor Cores: More Mixed Precision Options

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, …)

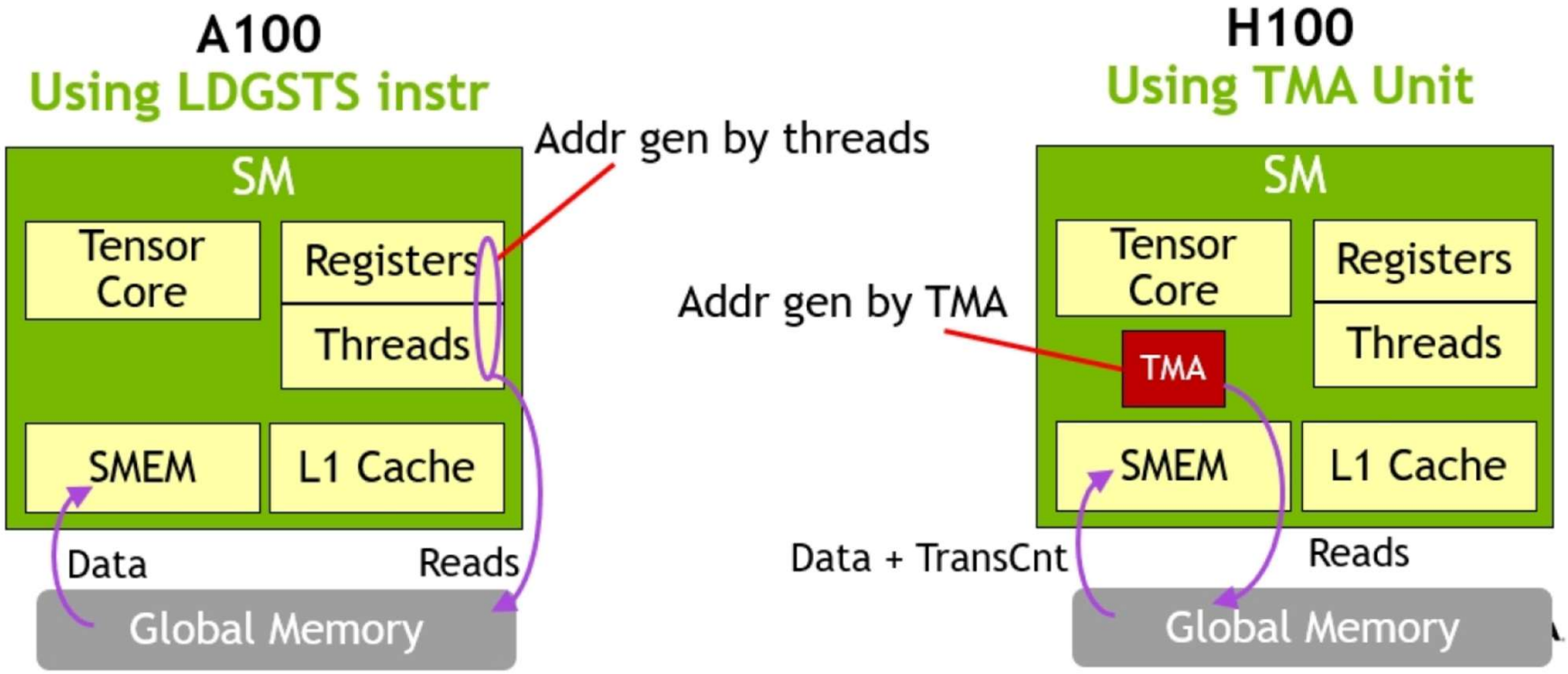
Tensor Cores: Hopper vs. Ampere

(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)

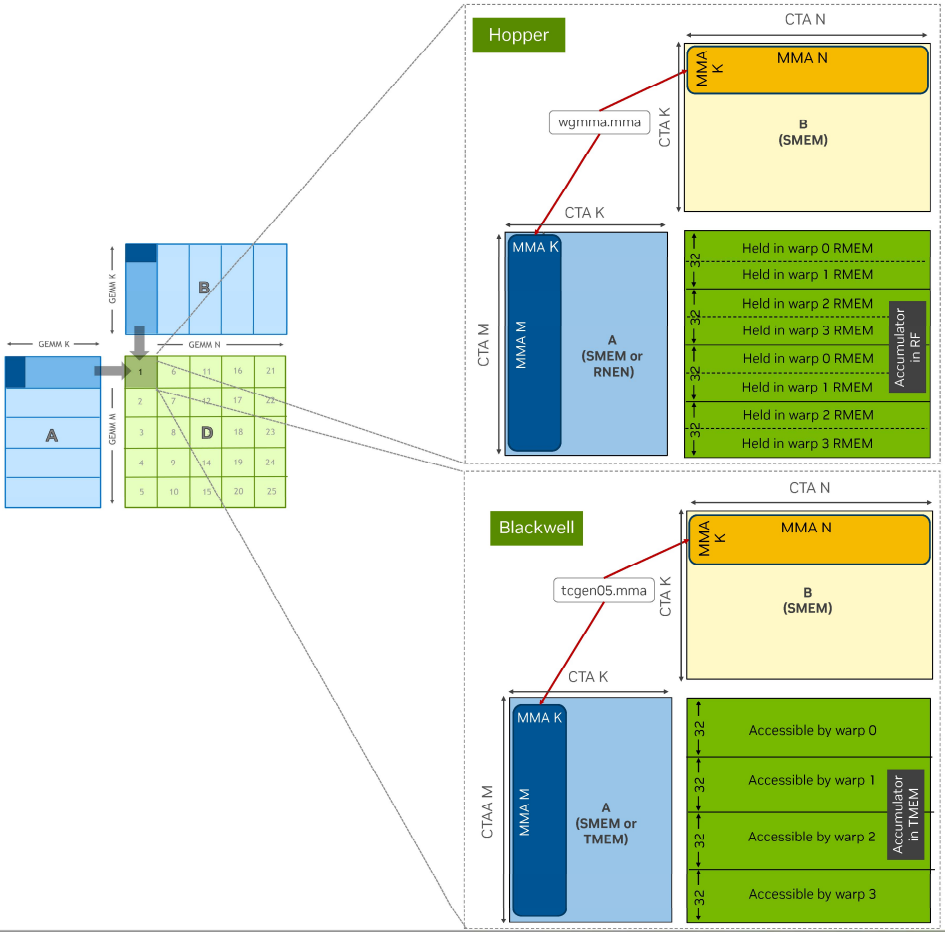
Asynchronous transfers



Programming Blackwell Tensor Cores with CuTe and CUTLASS

Hopper Tensor Cores → Blackwell Tensor Cores

$$D_{MxN} = \sum_k A_{MxK} * B_{KxN}$$



- 2x throughput vs Hopper: FP16, BF16, TF32, INT8, FP8
- Tensor Memory replaces Register Memory for Tensor Core input and output.
- Expands Async. MMA execution to Epilogues.

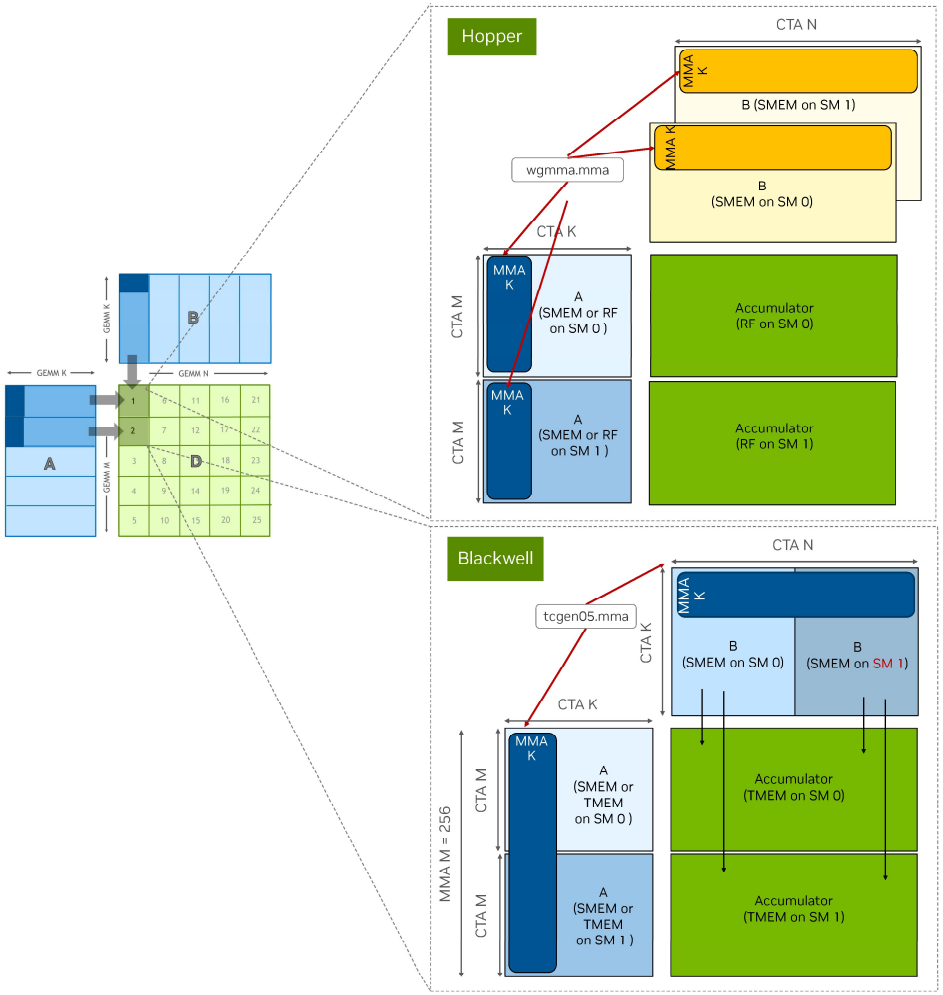
		Hopper	Blackwell
Operand sourcing	Input ‘A’	Registers / Shared Memory	Tensor Memory / Shared Memory
	Input ‘B’	Shared Memory	Shared Memory
	Accumulator ‘D’	Registers	Tensor Memory
Async. execution	Instruction issue	Warpgroup (4 warps)	1 thread
	Instruction completion	Warpgroup wide arrive/waits	SMEM mbarrier

Cris Cecka, Mihir Awatramani

Programming Blackwell Tensor Cores with CuTe and CUTLASS

Blackwell Tensor Core

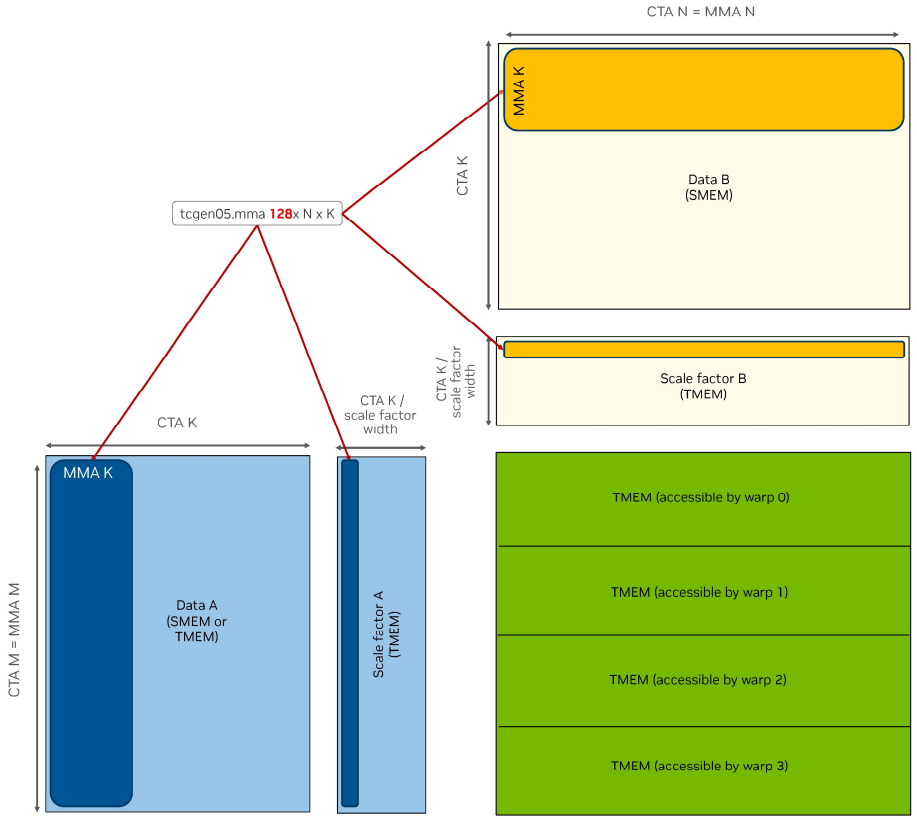
Expanding Tensor Core execution to 2 SMs



- Blackwell expands Tensor Core instruction to 2 SMs.
- Pair of CTAs in 2x1 cluster execute an MMA across 2 SMs
  - 2x1 cluster → 1 CTA pair
  - 2x2 cluster → 2 CTA pairs in 1x2 layout
  - 4x4 cluster → 8 CTA pairs in 2x4 layout
- CTA 0 in a 2x1 CTA pair is the “leader” CTA; elects 1 to thread issue the MMA for both SMs/CTAs.
- A, B and accumulator split evenly across 2 SMs; each SM write/reads its local SMEM/TMEM.

Blackwell Tensor Core

Support for New Block-scaled Formats



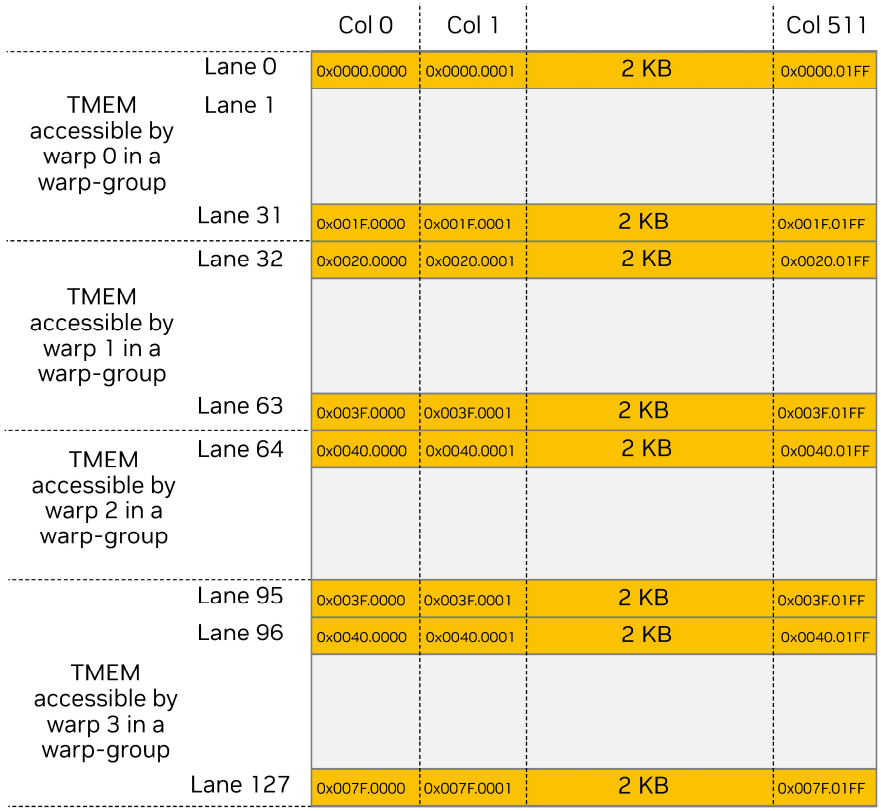
- Block-scaled type support in Tensor Core hardware: MXFP8, MXFP6, MXFP4, NVFP4
- MXFP8/FP6/FP4 support mixed AB input.
- 2x throughput vs Hopper FP8 for MXFP8/FP6/FP4.
- 4x throughput vs Hopper FP8 for NVFP4 and MXFP4 (non-mixed).
- A and B scaling factor matrices need to be sourced from TMEM..

Format name	Data format	Scale format	Can be mixed with
MXFP8	E4M3, E5M2	E8 or None	MXFP6, MXFP4
MXFP6	E2M3, E3M2	E8 or None	MXFP8, MXFP4
MXFP4	E2M1	E8 or None	MXFP8, MXFP6
NVFP4	E2M1	E8, E4M3	-

Cris Cecka, Mihir Awatramani

Blackwell Tensor Memory (TMEM)

New memory on each SM for Tensor Core Inputs and Outputs



- New memory on each SM; same size as the Register File: 256 KB.
- TMEM shape: 128 lanes x 2 KB each (512 columns, 4 B per column).
  - Warps can only access TMEM within its sub-partition. Need a warpgroup to access all of TMEM
- Alloc/dealloc not automatically managed like SMEM.
  - Kernels explicitly allocate and deallocate on the SM using tcgen05.alloc and tcgen05.dealloc instructions.
- Used for Tensor Core (TC) ops. SIMT operations not supported on TMEM.
- Warps explicitly move data in/out of TMEM before/after TC ops.
  - RMEM ←→ TMEM; tcgen05.load, tcgen05.store PTX instructions
  - TMEM ← SMEM; tcgen05.cp PTX instruction.
  - Non-linear addressing; data is moved in blocks of pre-defined layouts.

CUDA C Warp Matrix Functions (WMMA)

Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C++ Programming Guide (13.0), Chapter 10.24

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 13.0, Chapter 10.24

```
#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 9.0, Chapter 9.7.14

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 9.0

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

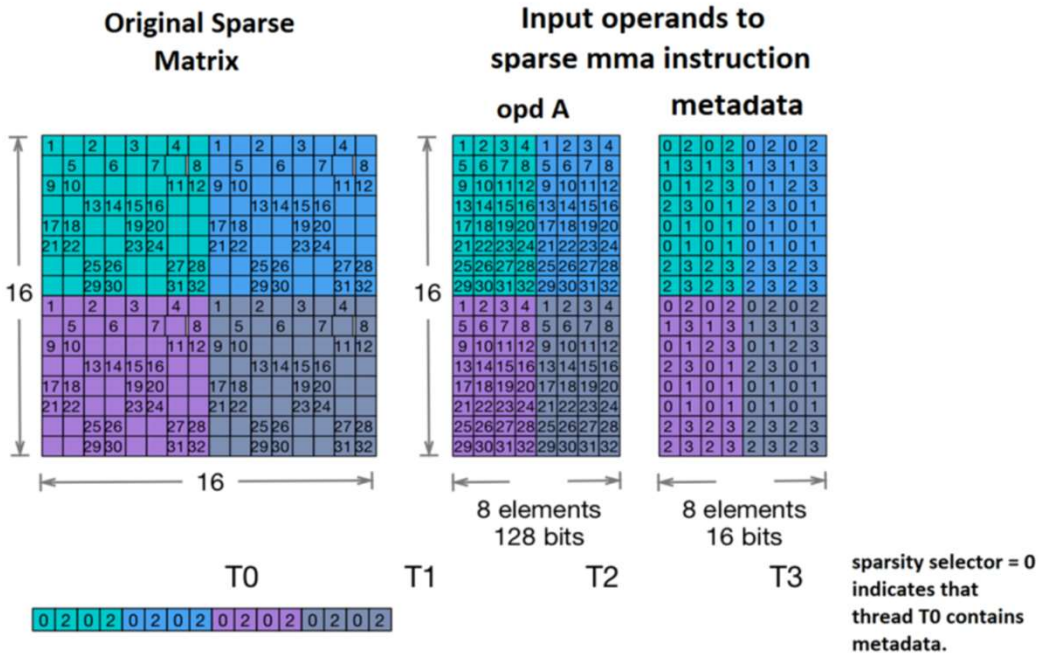
R\C	0	1	2	3	4	5	6	7
0	T0 : { c0, c1 }		T2 : { c0, c1 }		T0 : { c4, c5 }		T2 : { c4, c5 }	
1	T1 : { c0, c1 }		T3 : { c0 , c1 }		T1 : { c4, c5 }		T3 : { c4 , c5 }	
2	T0 : { c2, c3 }		T2 : { c2, c3 }		T0 : { c6, c7 }		T2 : { c6, c7 }	
3	T1 : { c2, c3 }		T3 : { c2 , c3 }		T1 : { c6, c7 }		T3 : { c6 , c7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

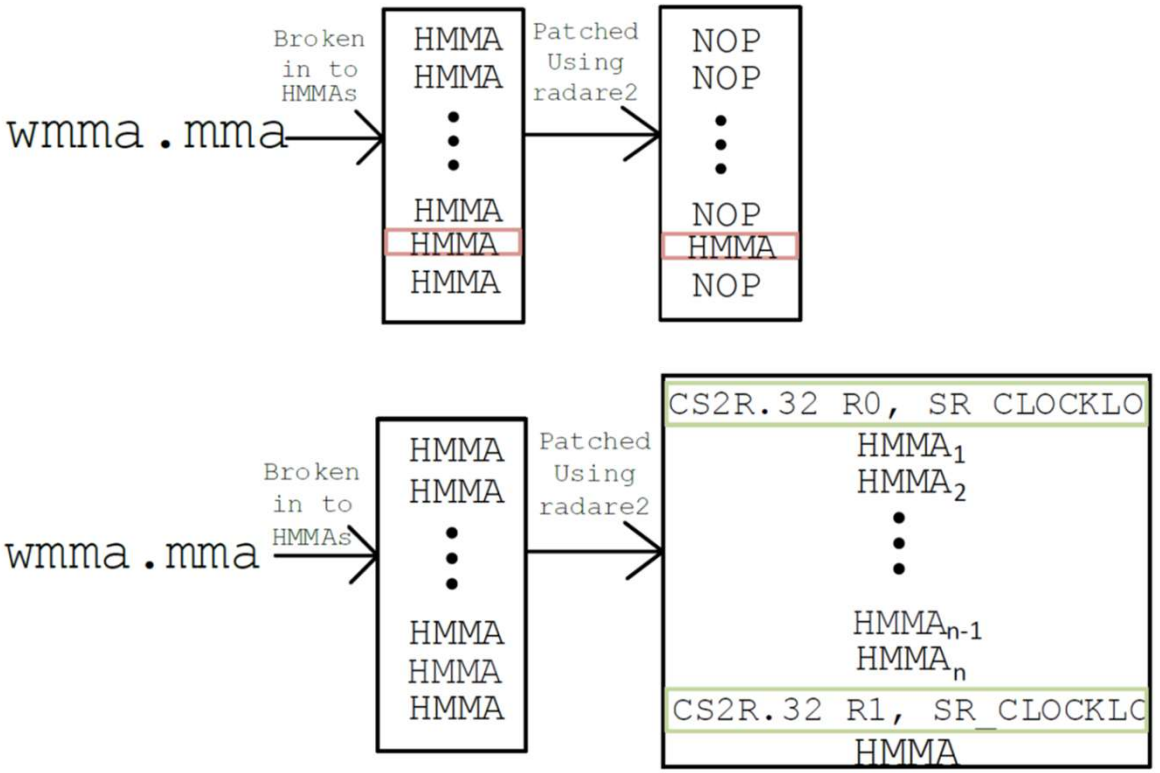
```
// Load a single 8x8 matrix using 64-bit addressing
.reg .b64 addr;
.reg .b32 d;
ldmatrix.sync.aligned.m8n8.x1.shared.b16 {d}, [addr];

// 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

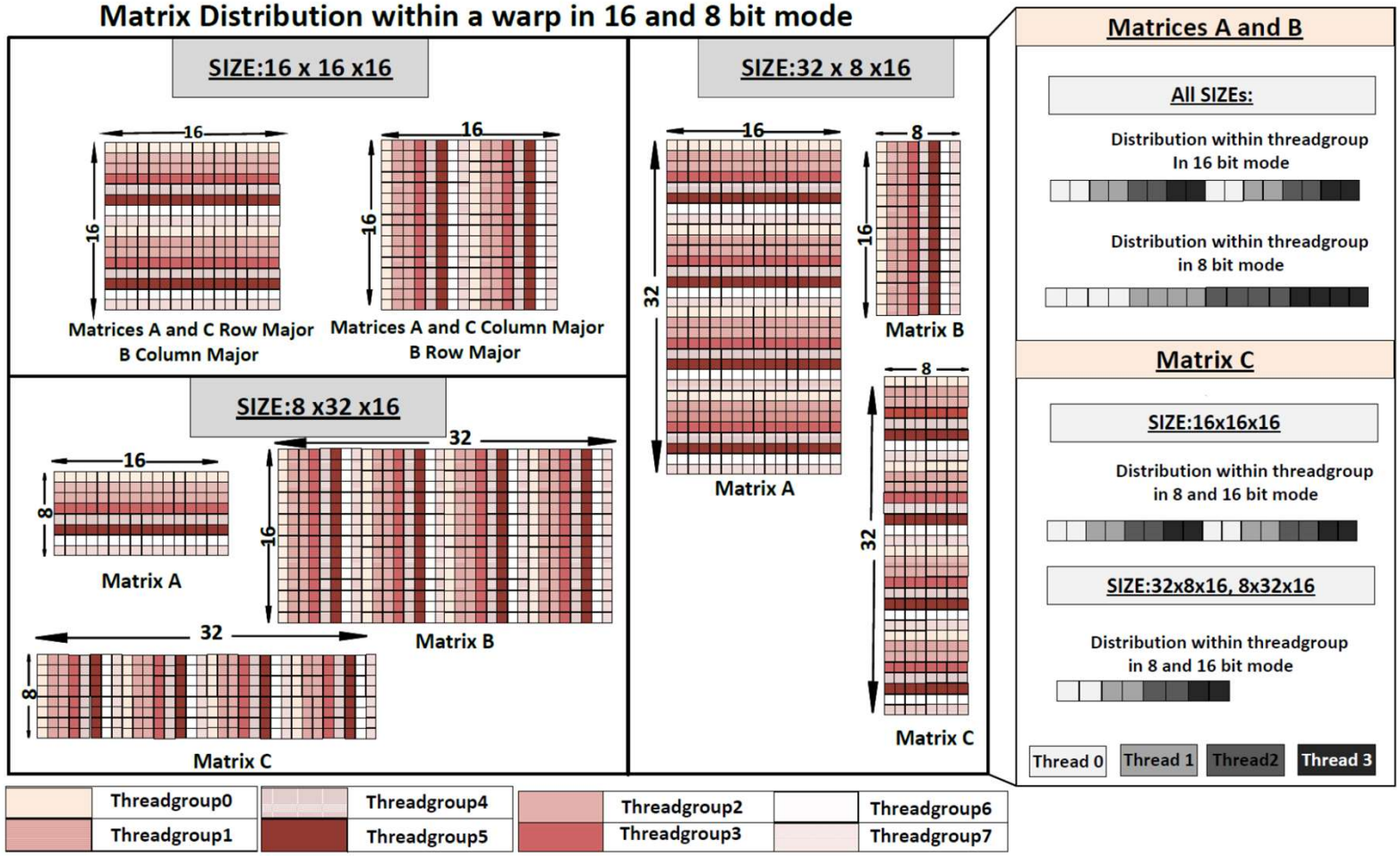


Figure 8: Distribution of operand matrix elements to threads for tensor cores in the RTX 2080 (Turing).

PTX WMMA to SASS

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

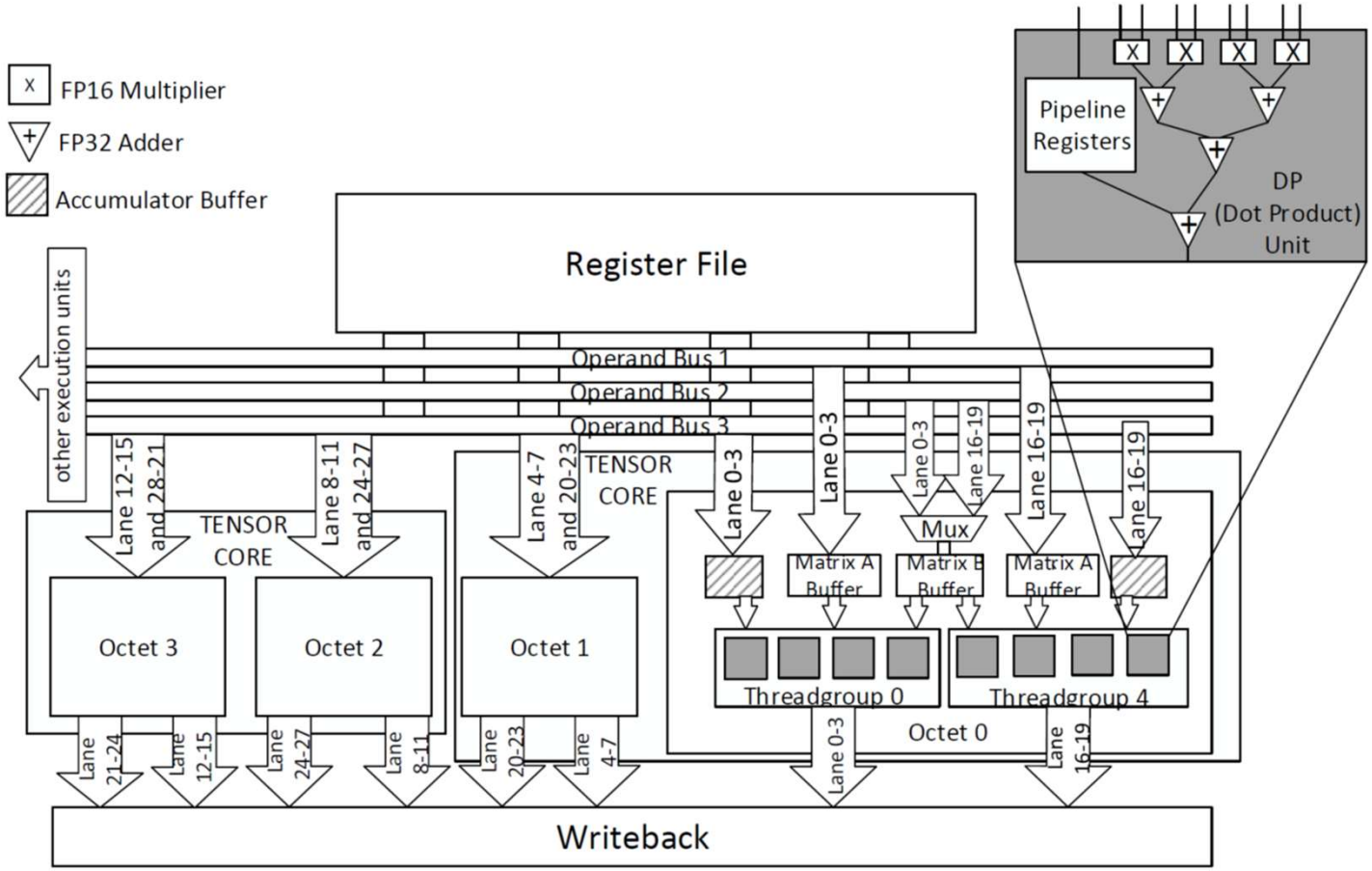
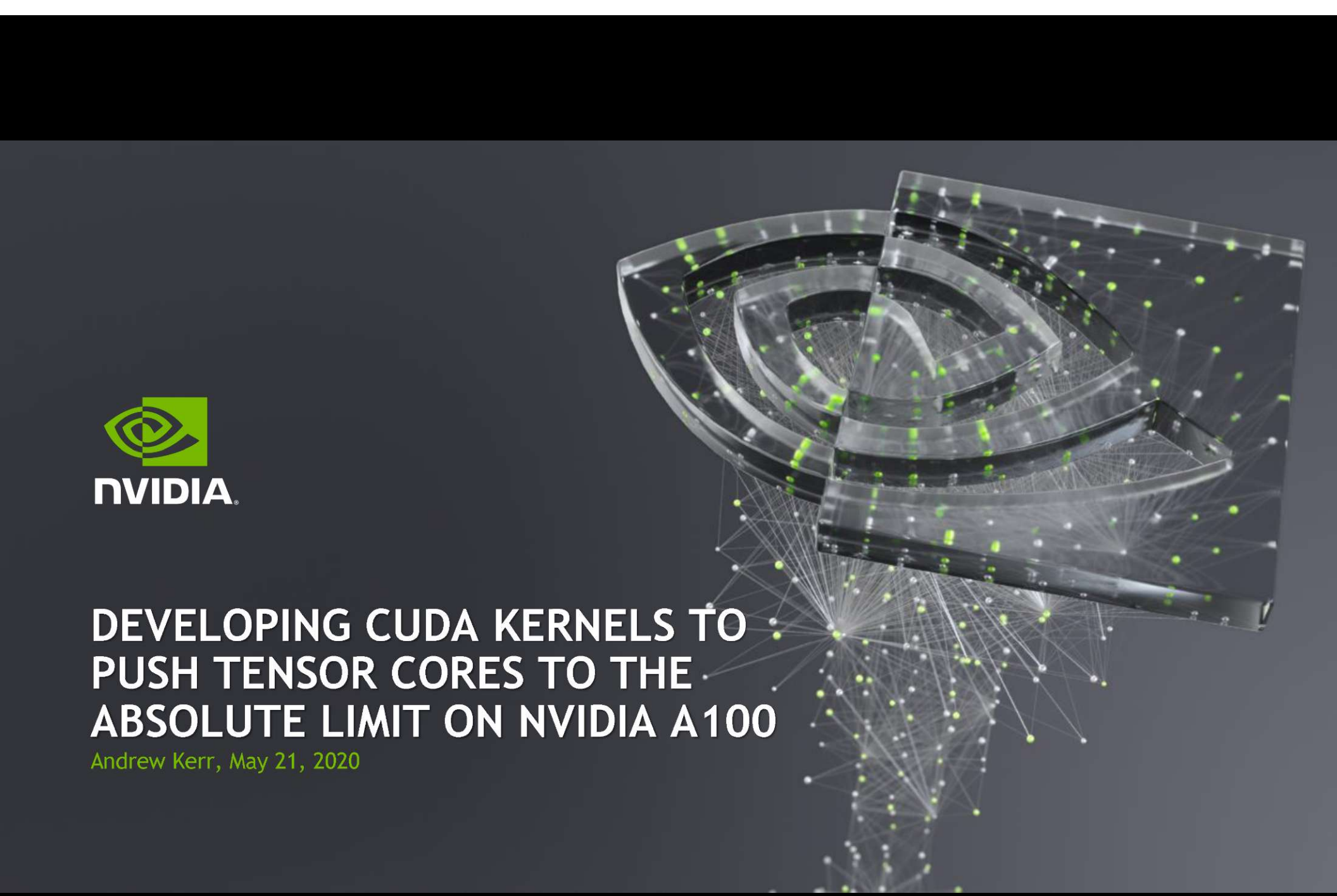


Figure 13: Proposed Tensor Core Microarchitecture



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

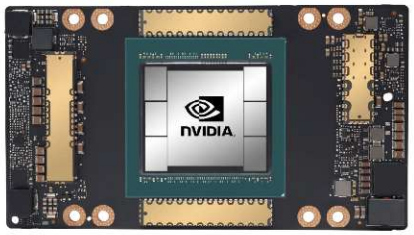
Andrew Kerr, May 21, 2020

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



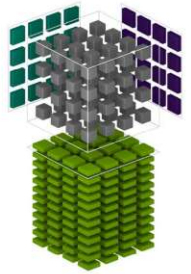
- Floating-point Tensor Core operations **8x** and **16x** faster than F32 CUDA Cores
- Integer Tensor Core operations **32x** and **64x** faster than F32 CUDA Cores
- New IEEE double-precision Tensor Cores **2x** faster than F64 CUDA Cores

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”

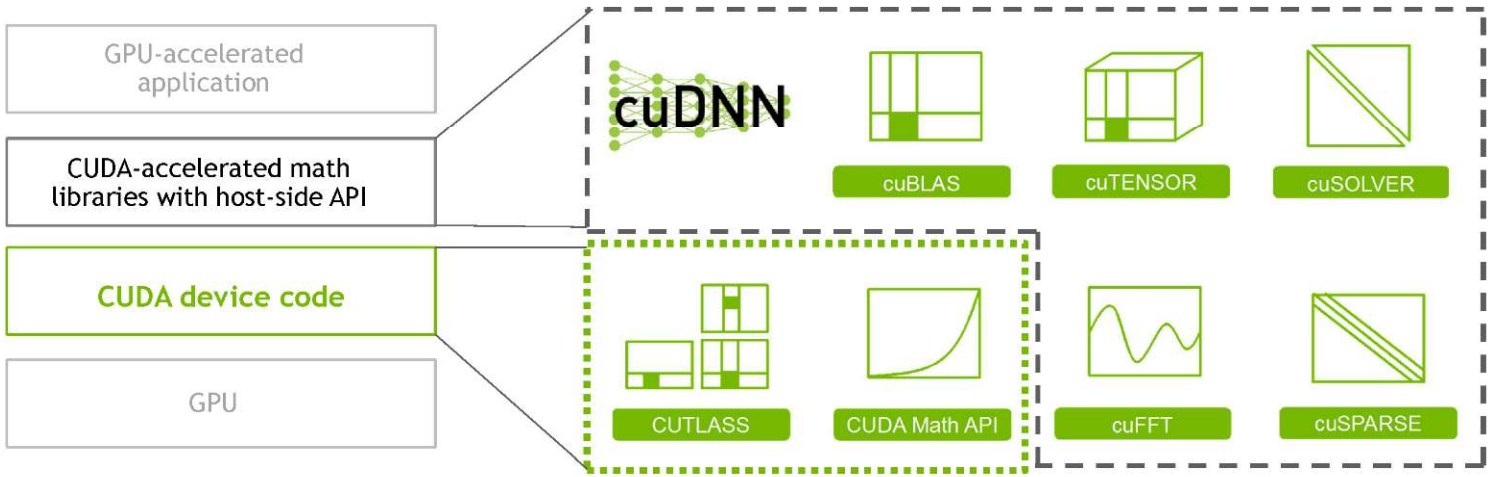
PROGRAMMING NVIDIA AMPERE ARCHITECTURE

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)

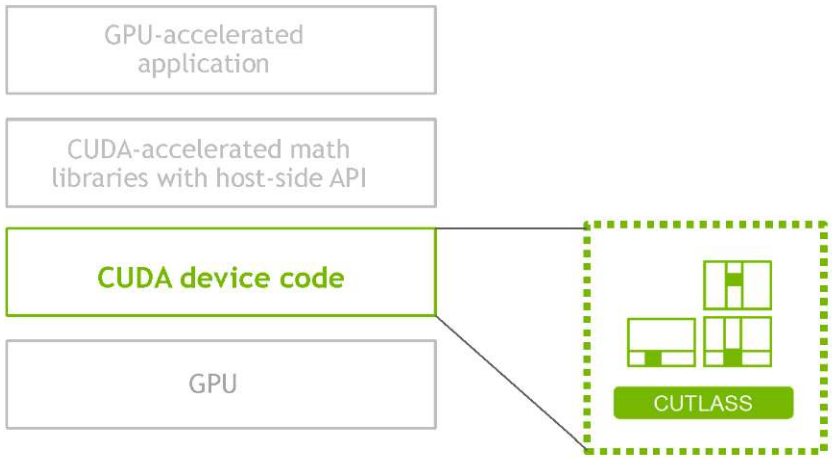
CUDA C++ Device Code

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



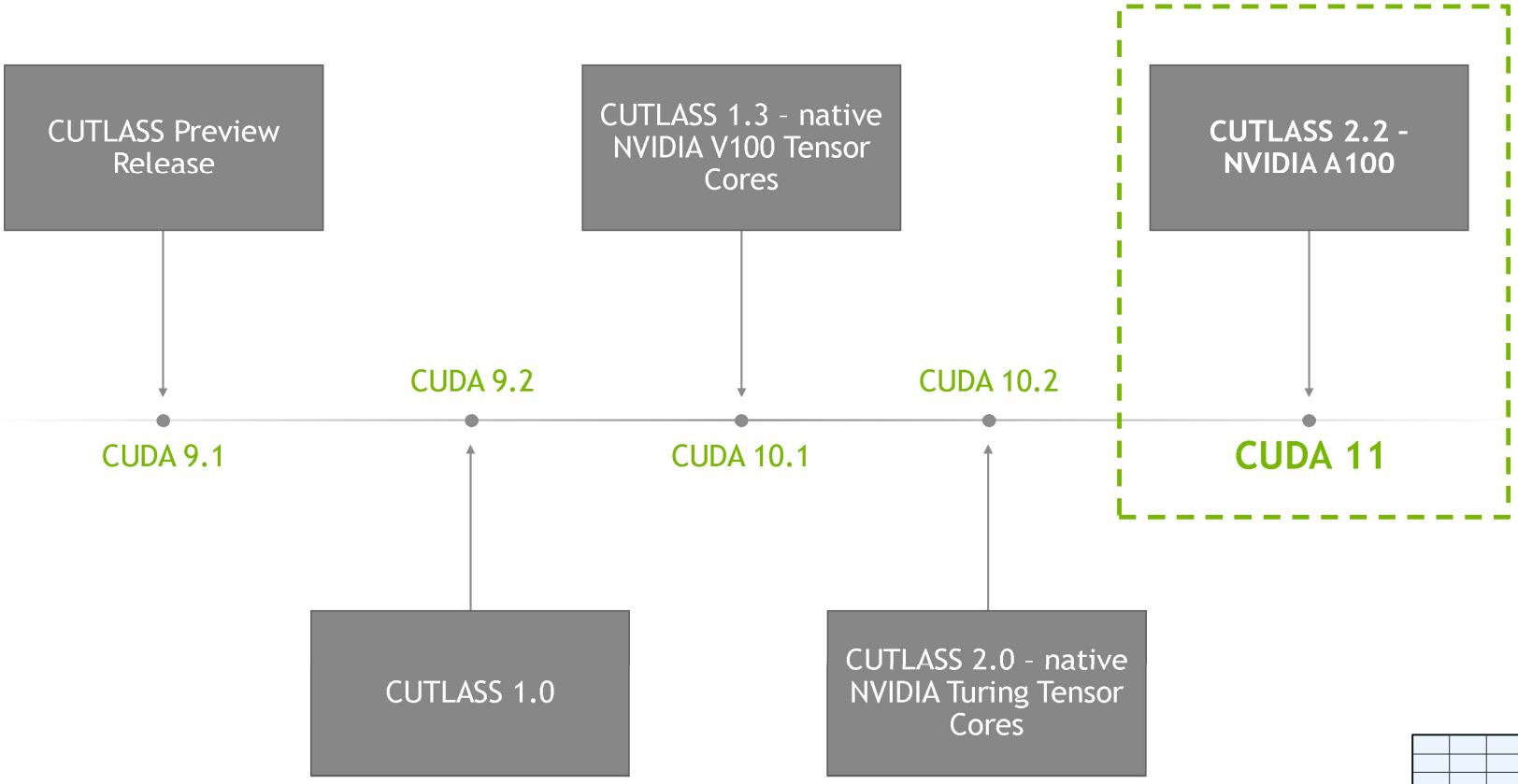
PROGRAMMING NVIDIA AMPERE ARCHITECTURE with CUDA C++

This is a talk for CUDA programmers

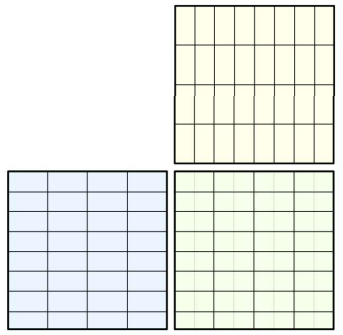


CUTLASS

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

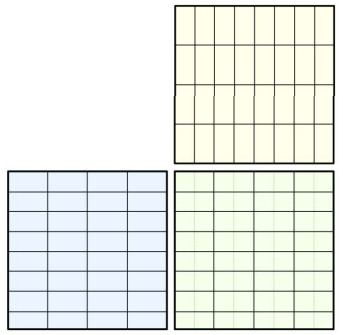
CUTLASS 2.1

- 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..

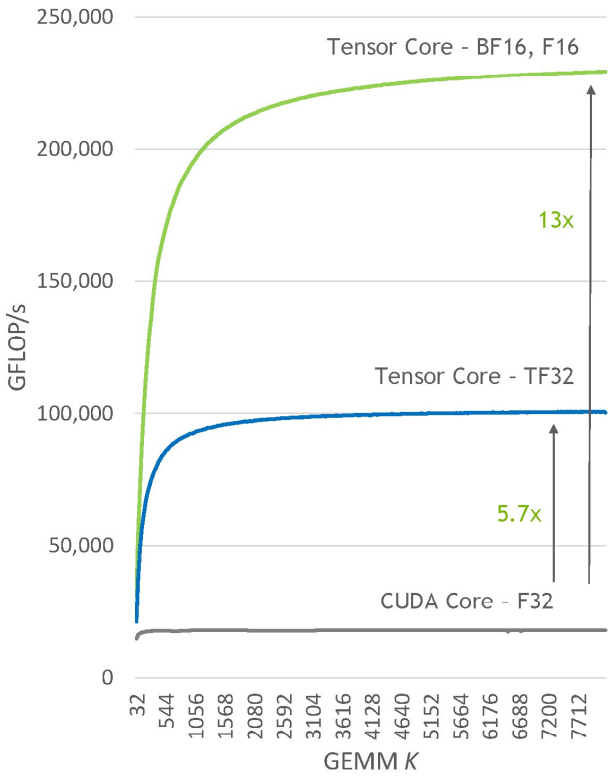
https://github.com/NVIDIA/cutlass



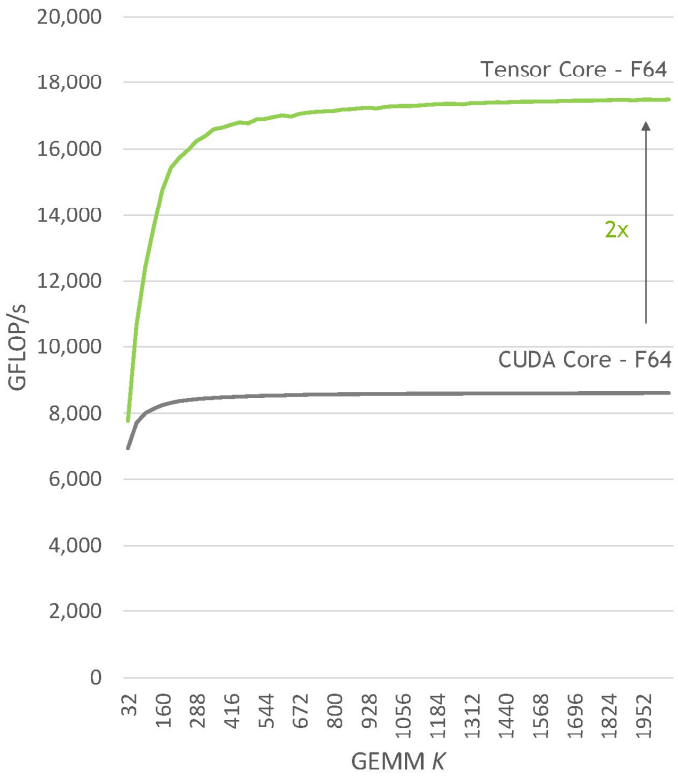
CUTLASS PERFORMANCE ON NVIDIA AMPERE ARCHITECTURE

CUTLASS 2.2 - CUDA 11 Toolkit – NVIDIA A100

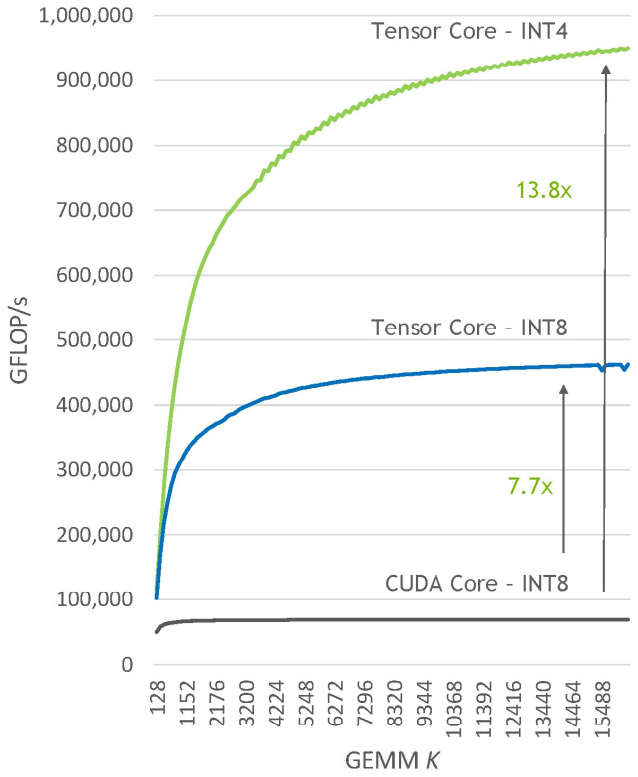
Mixed Precision Floating Point



Double Precision Floating Point

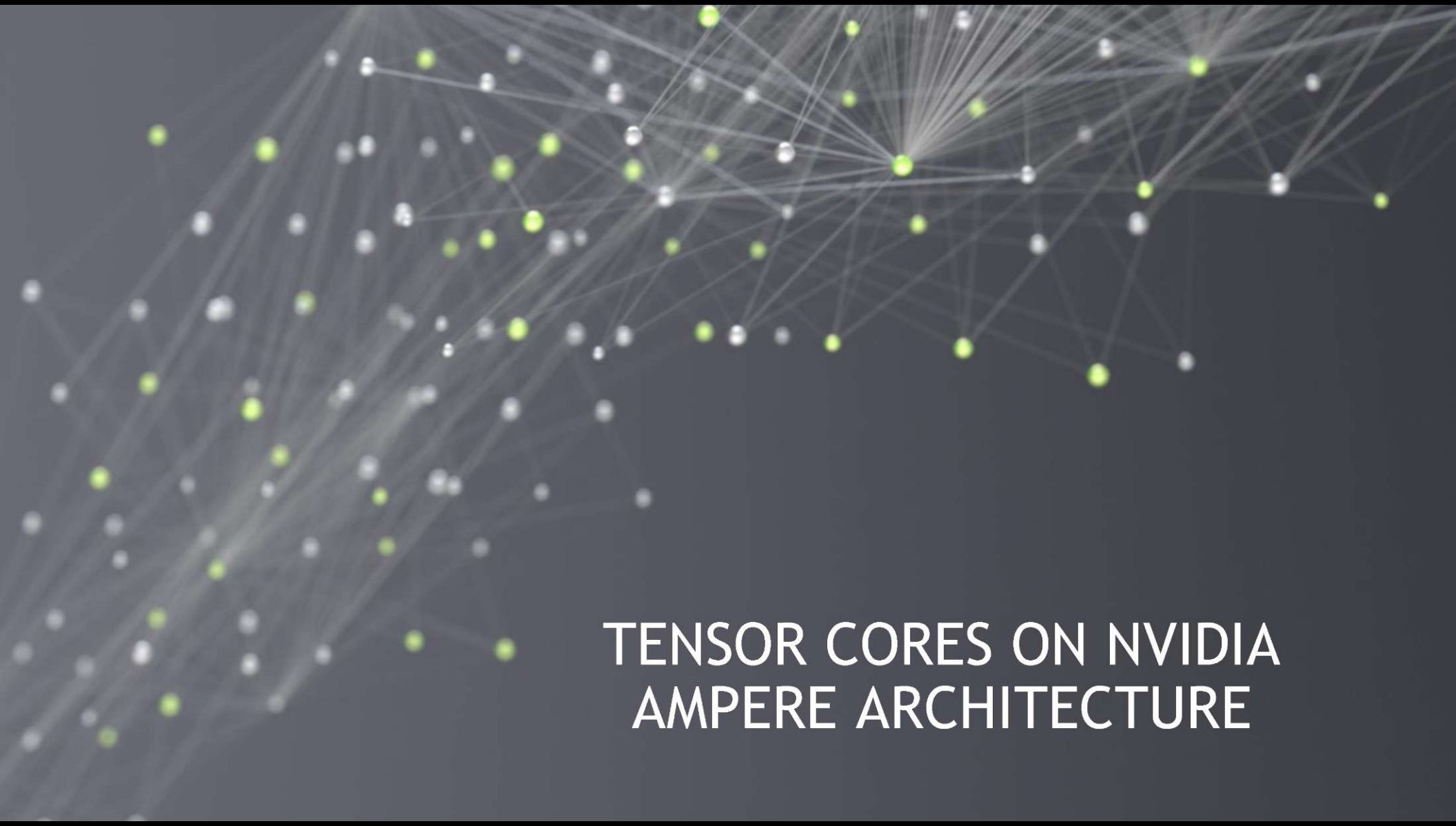


Mixed Precision Integer



m=3456, n=4096

NVIDIA


TENSOR CORES ON NVIDIA
AMPERE ARCHITECTURE

WHAT ARE TENSOR CORES?

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

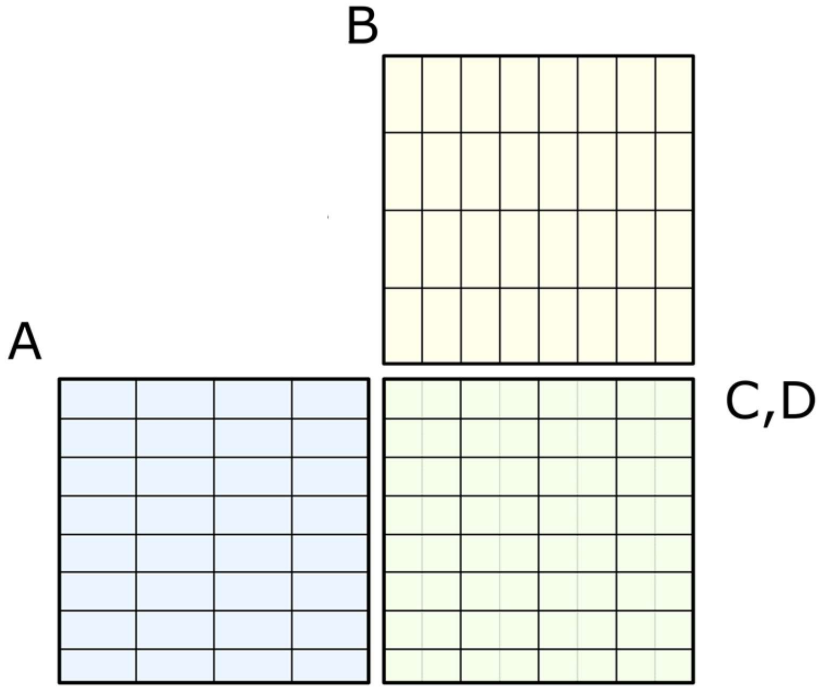
- Matrix multiply-add
- XOR-POPC

Input Data types: A, B

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

Accumulation Data Types: C, D

- half, float, int32_t, double



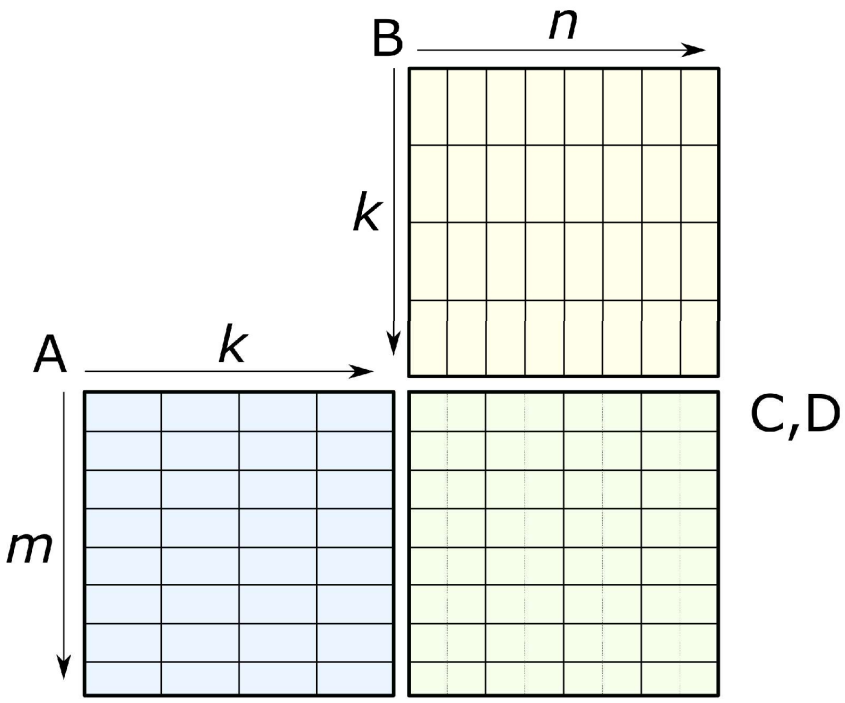
WHAT ARE TENSOR CORES?

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



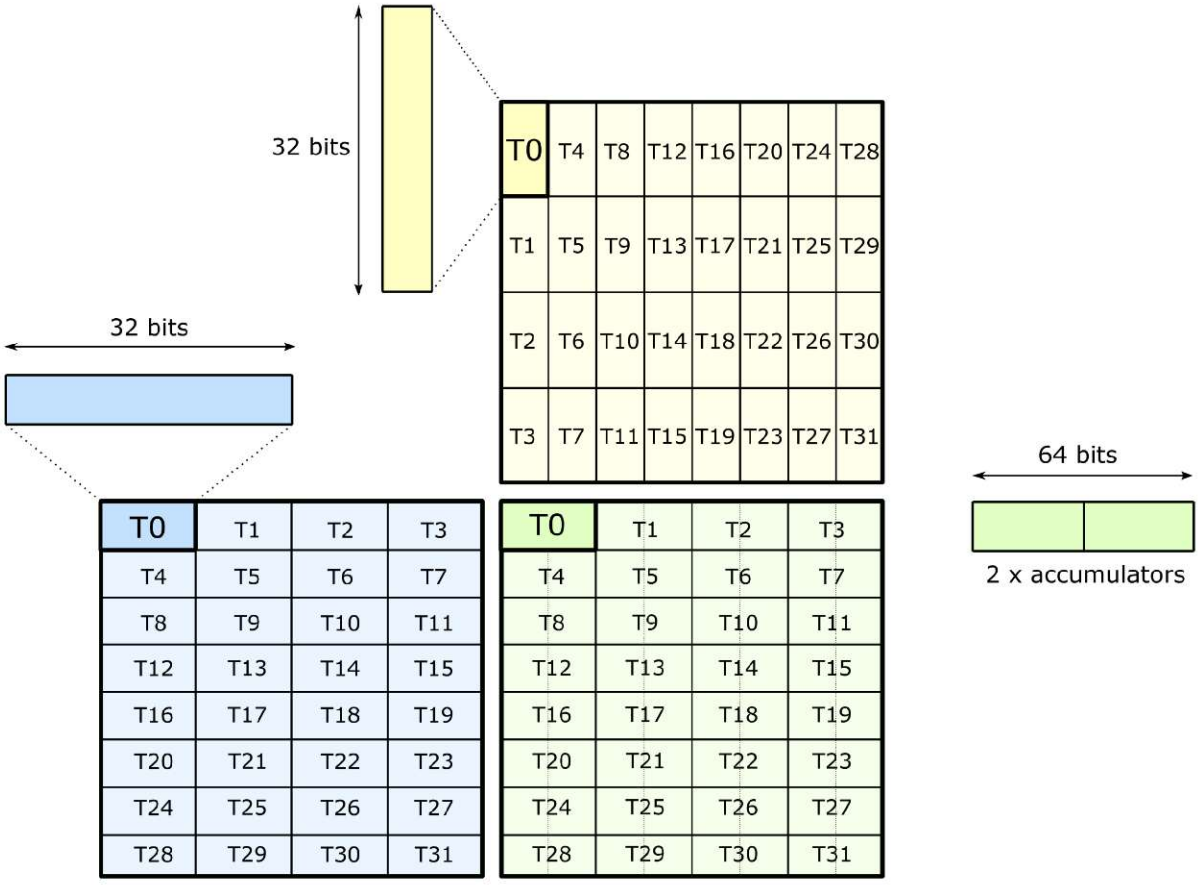
NVIDIA AMPERE ARCHITECTURE - TENSOR CORE OPERATIONS

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.

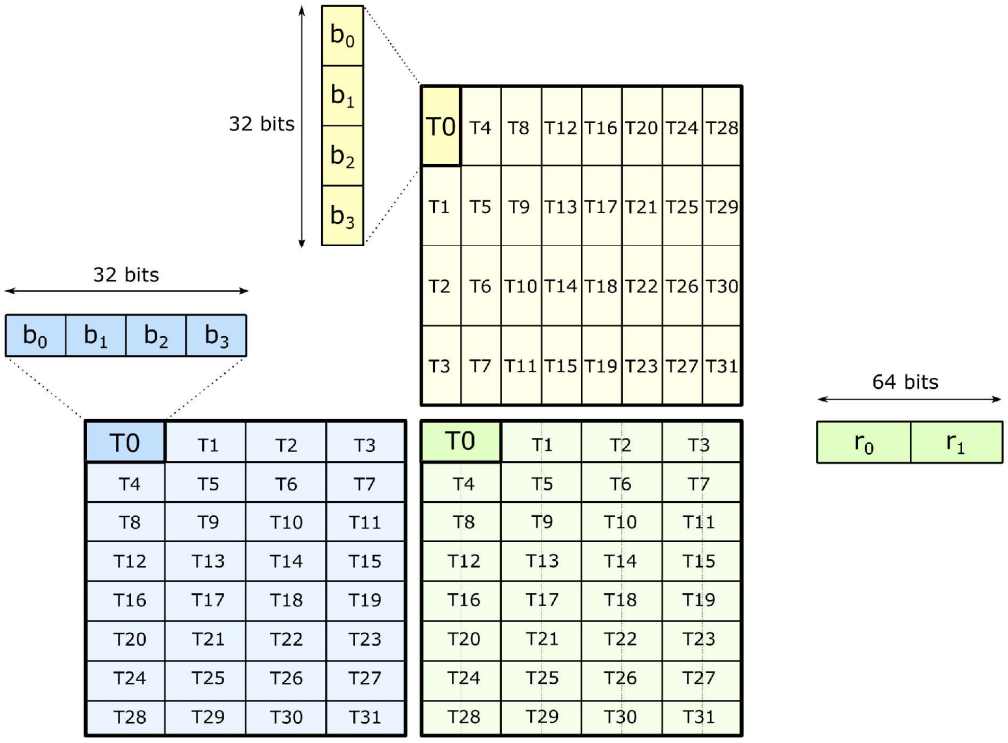
TENSOR CORE OPERATION: FUNDAMENTAL SHAPE



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

S8 * S8 + S32

8-by-8-by-16

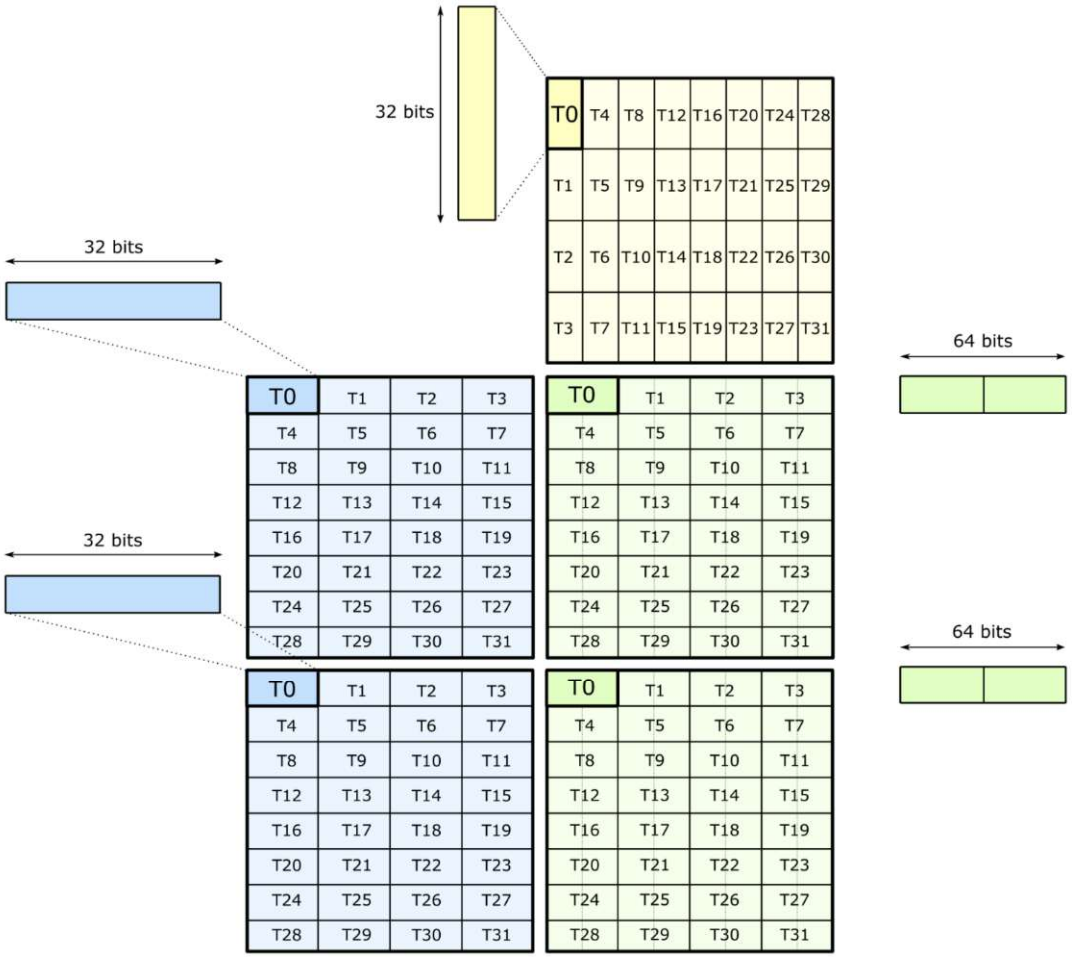


mma.sync.aligned
(via inline PTX)

```
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

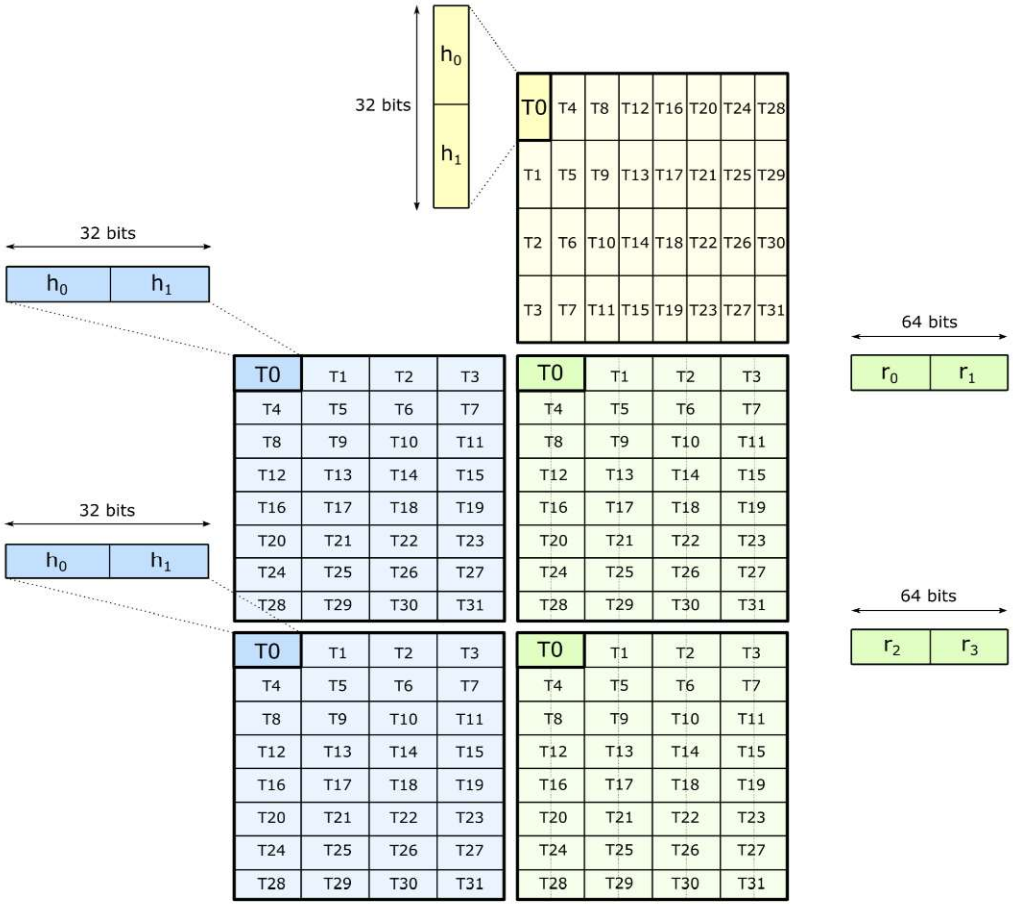
EXPANDING THE *M* DIMENSION



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

F16 * F16 + F32

16-by-8-by-8

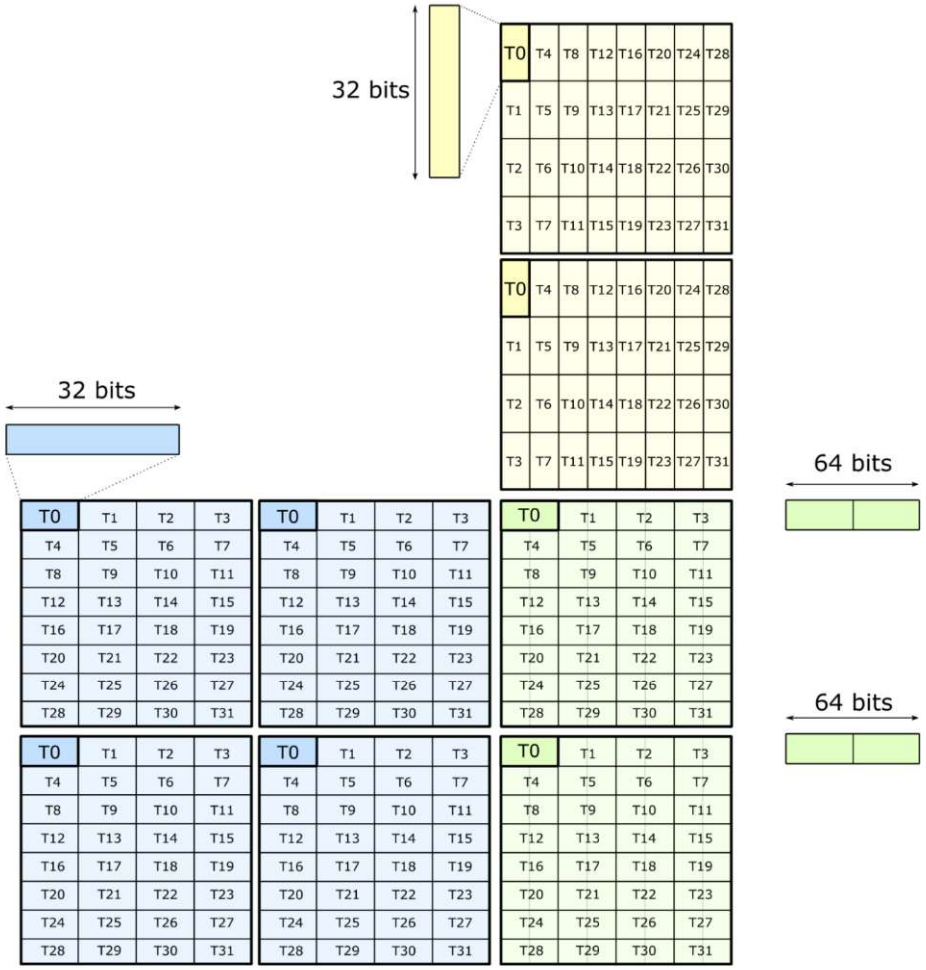


mma.sync.aligned
(via inline PTX)

```
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

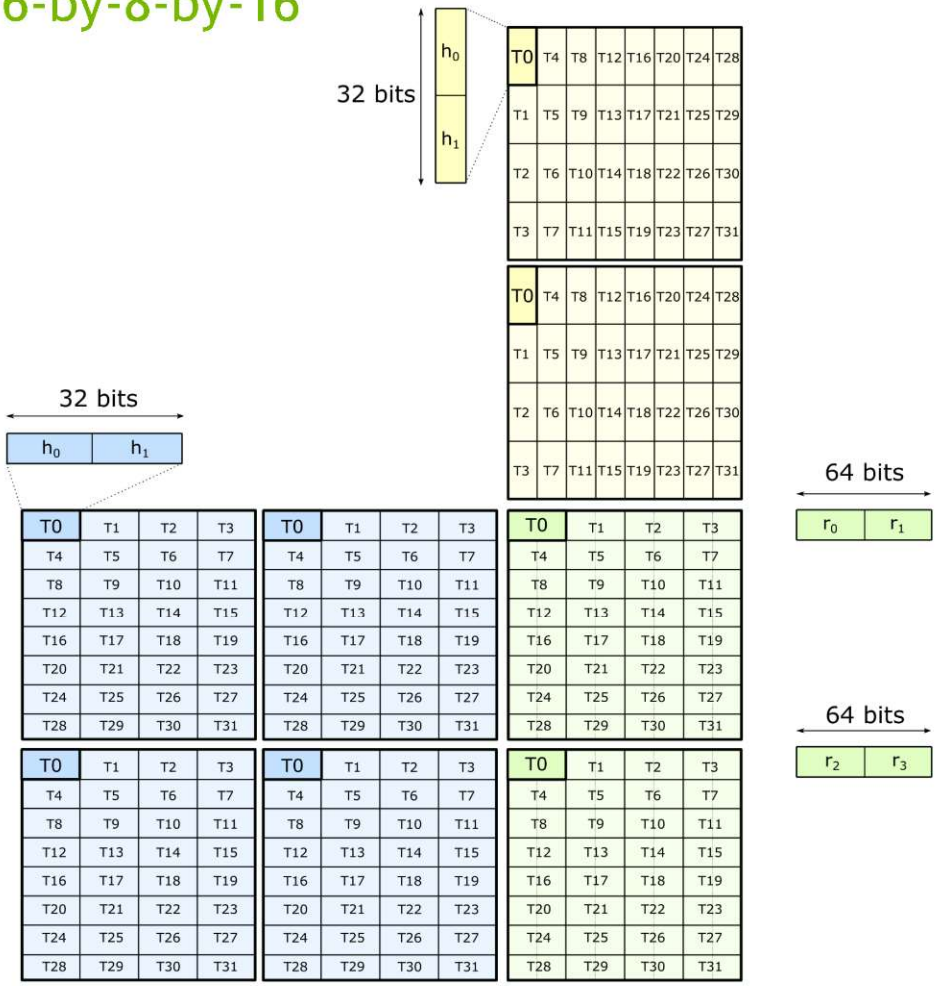
EXPANDING THE *K* DIMENSION



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

F16 * F16 + F32

16-by-8-by-16



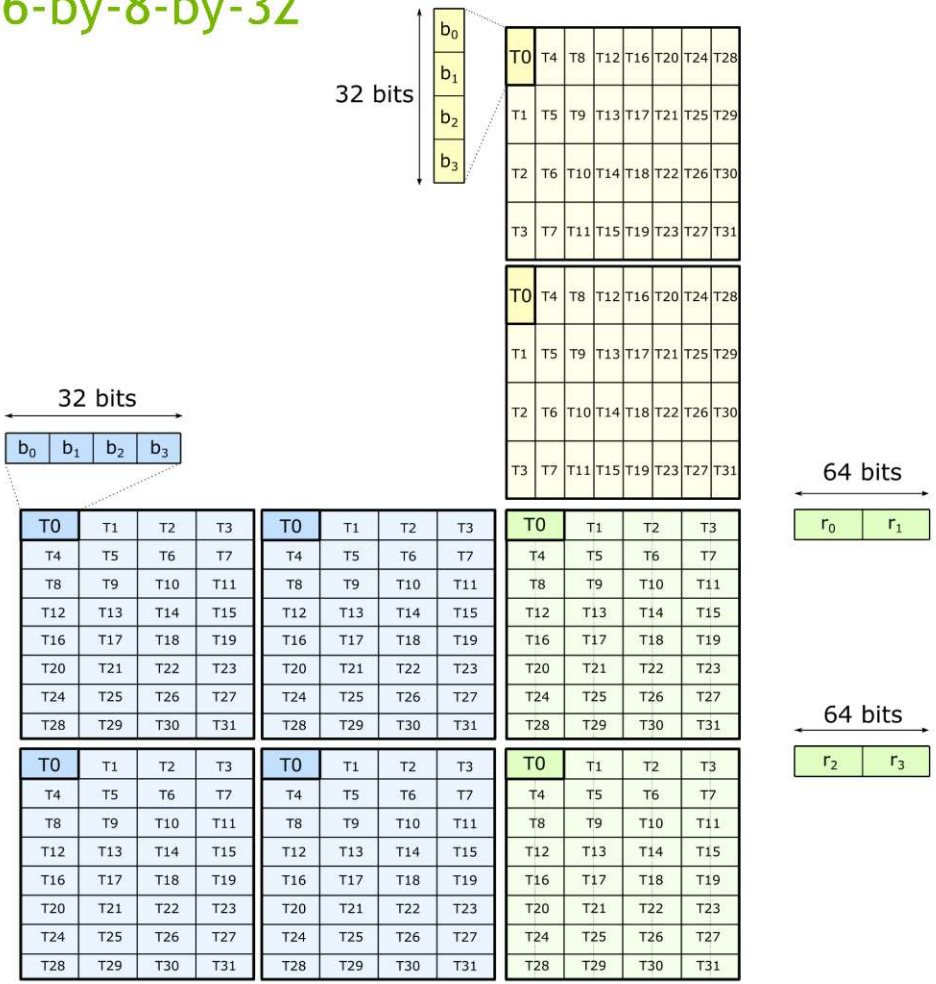
mma.sync.aligned
(via inline PTX)

```
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



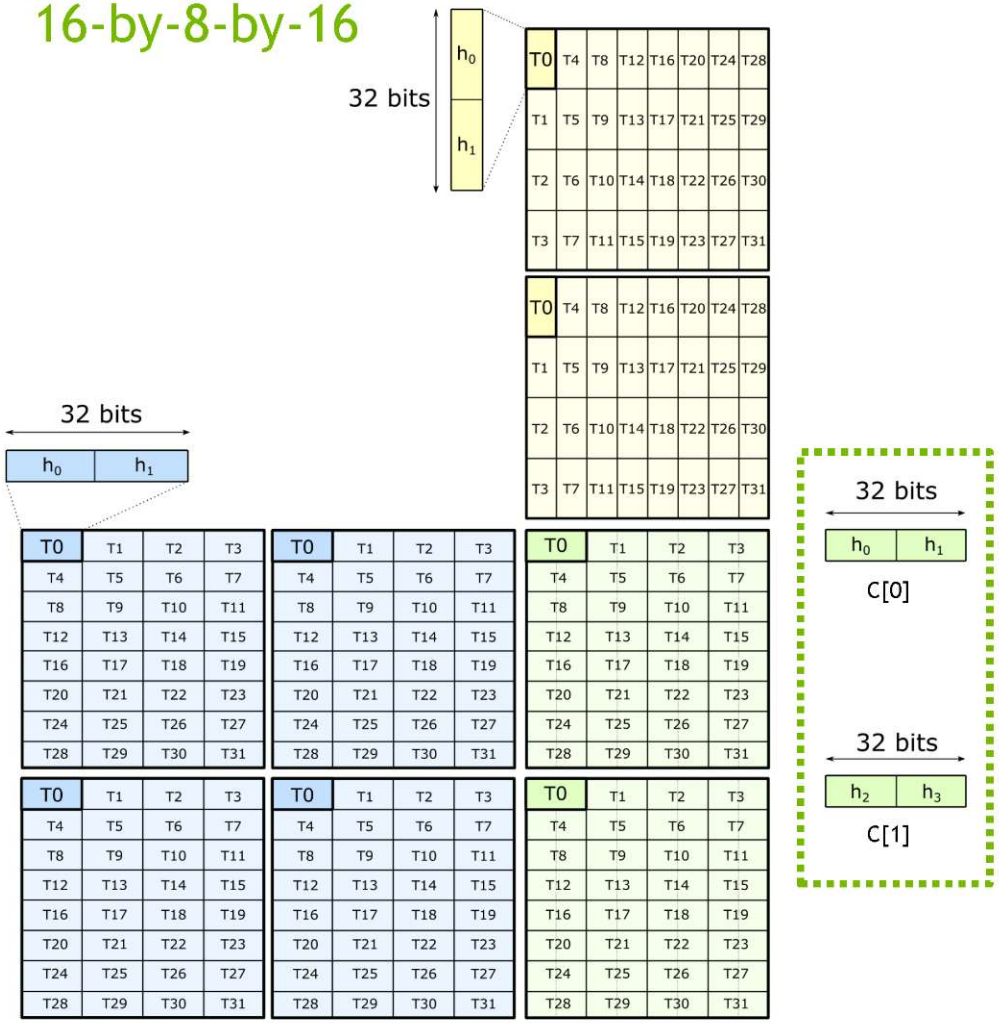
mma.sync.aligned
(via inline PTX)

```
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



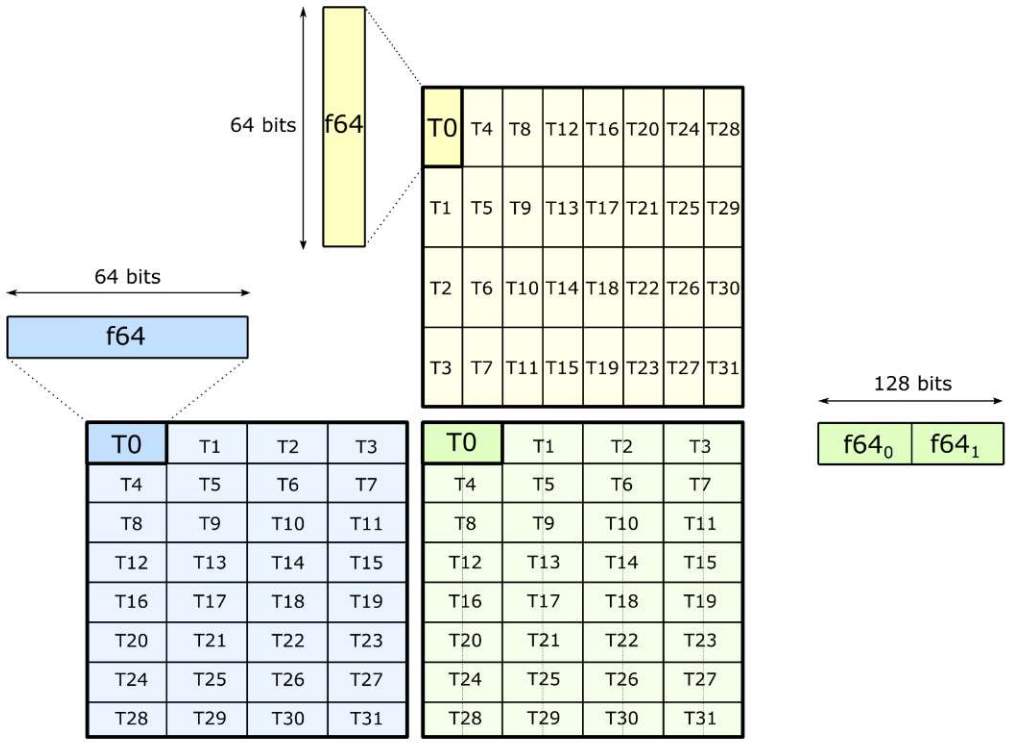
mma.sync.aligned
(via inline PTX)

```
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



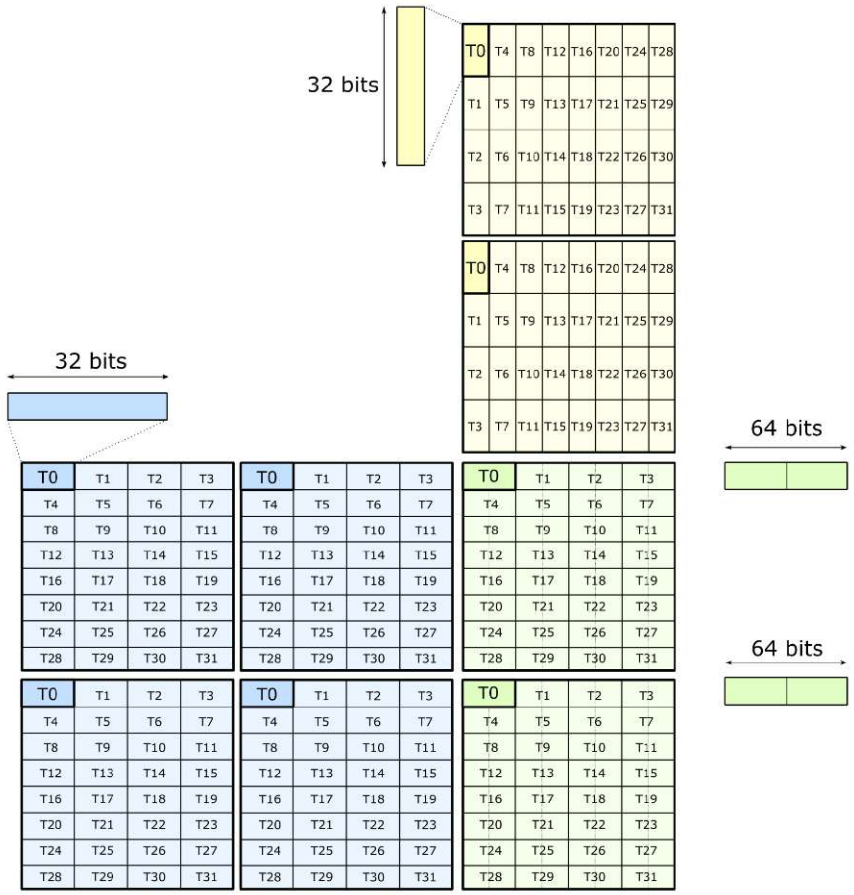
mma.sync.aligned
(via inline PTX)

```
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*



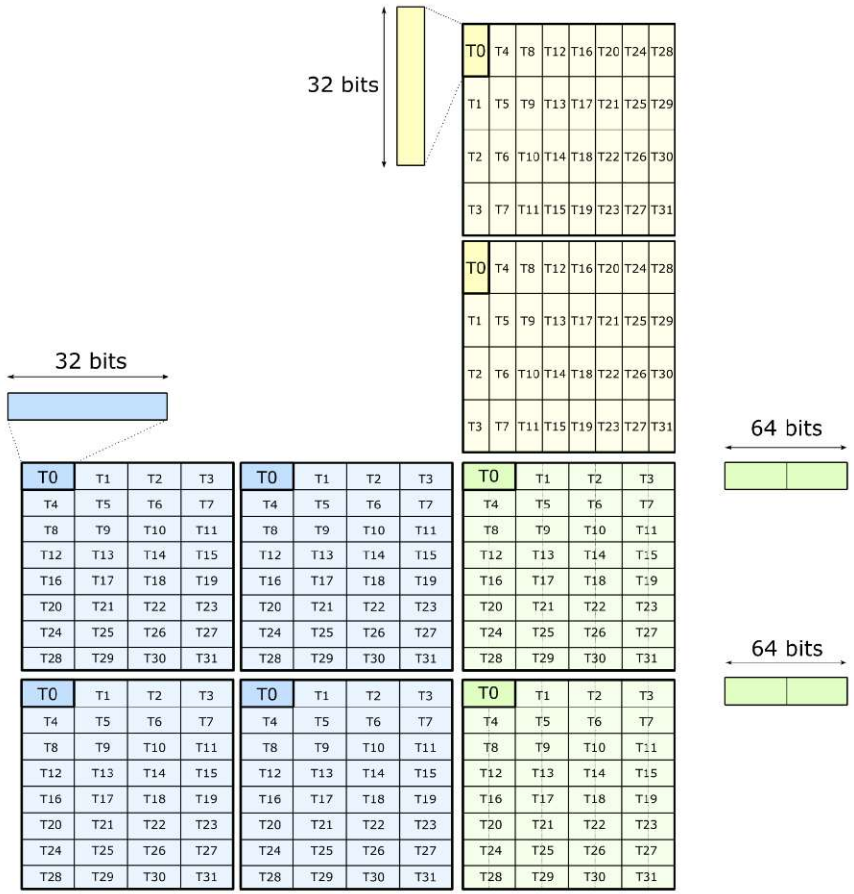
cutlass::arch::Mma

```
/// 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

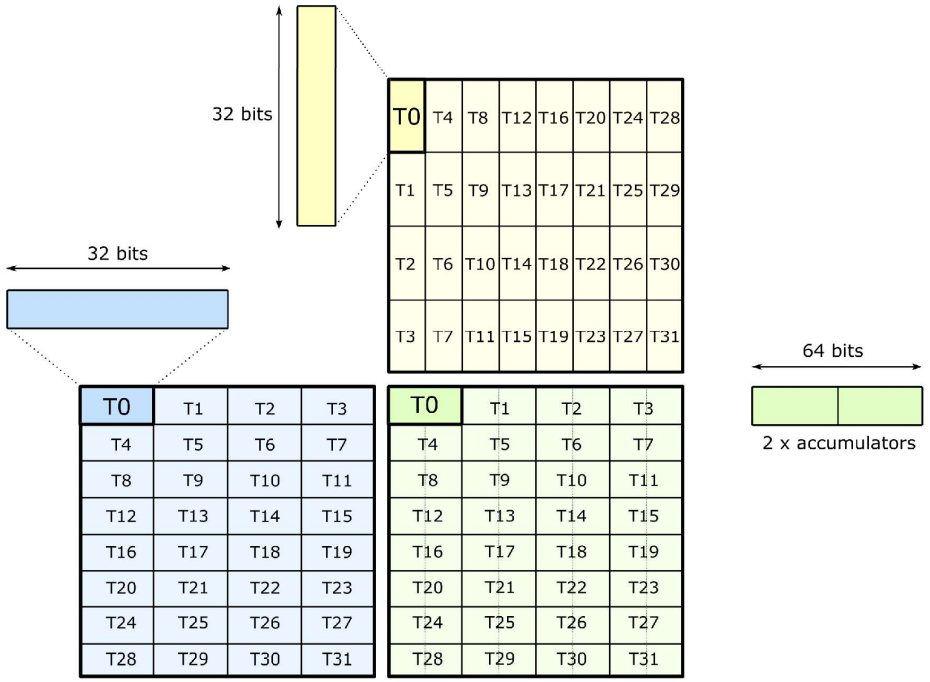
HELLO WORLD: TENSOR CORES

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

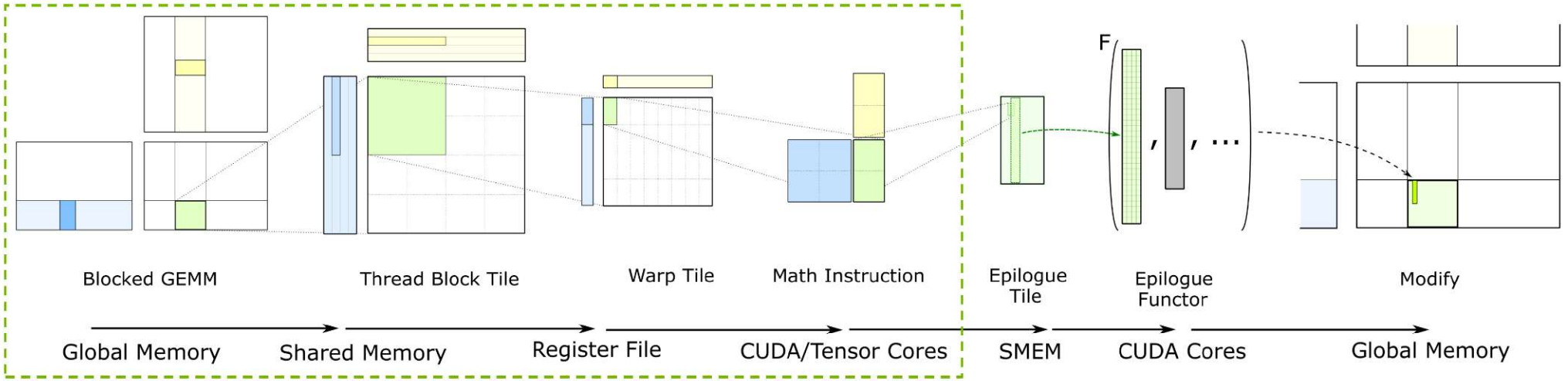
This kernel is global memory bandwidth limited.

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



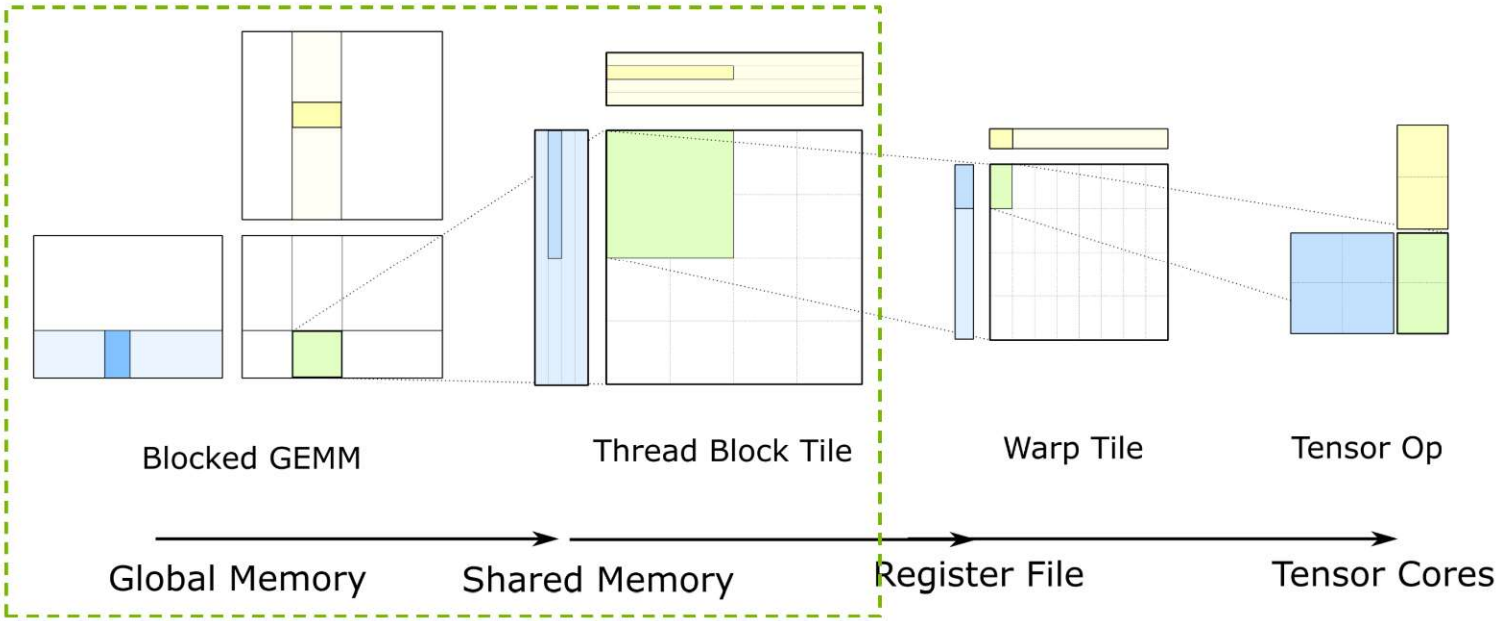
Tiled, hierarchical model: reuse data in Shared Memory and in Registers

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

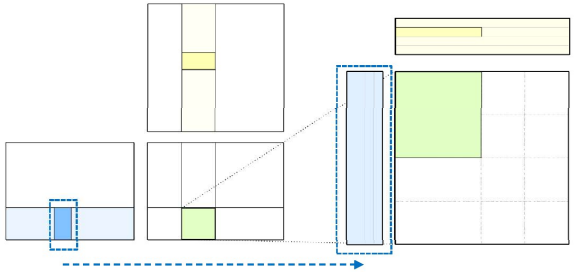
FEEDING THE DATA PATH

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

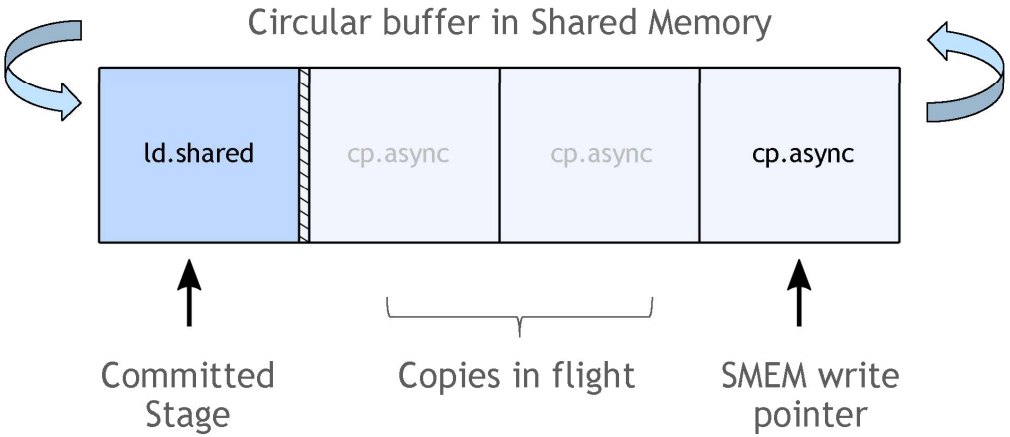


New NVIDIA Ampere Architecture feature: cp.async

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

Enables efficient software pipelines

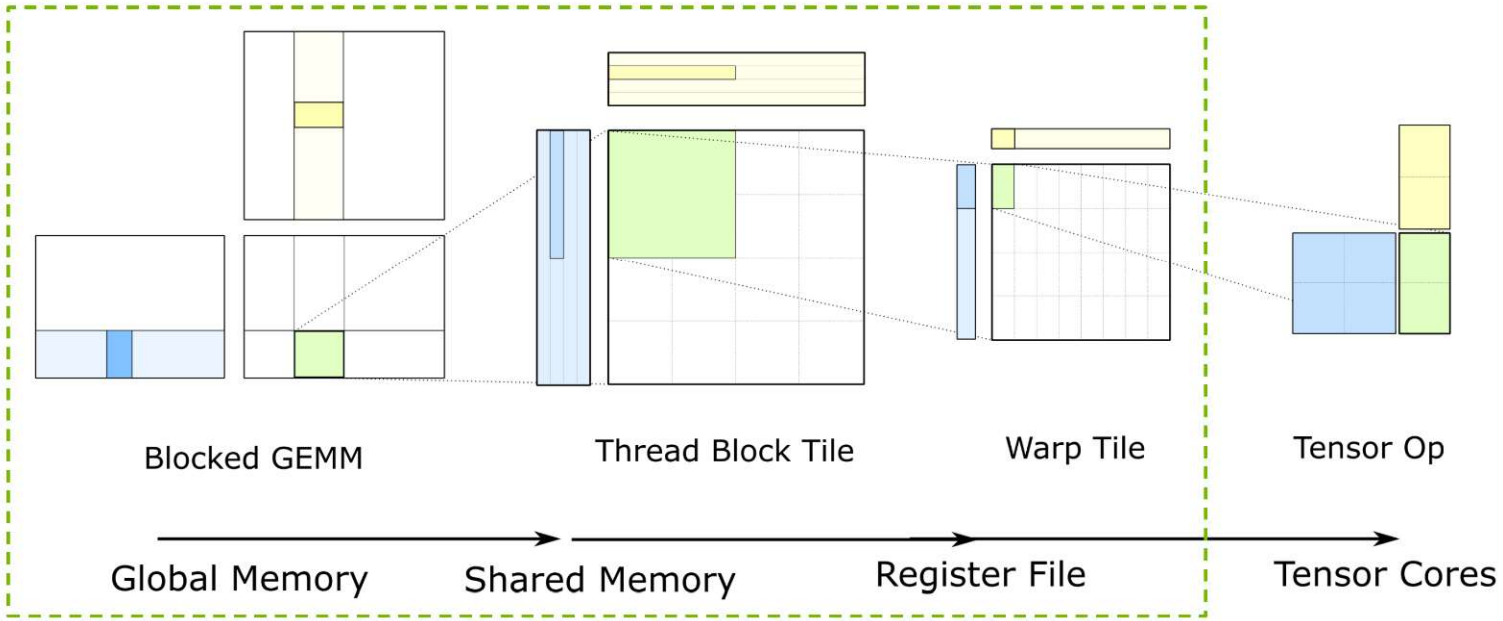
- Minimizes data movement: L2 → L1 → RF → SMEM becomes L2 → SMEM
- Saves registers: RF no longer needed to hold the results of long-latency load instructions
- Indirection: fetch several stages in advance for greater latency tolerance



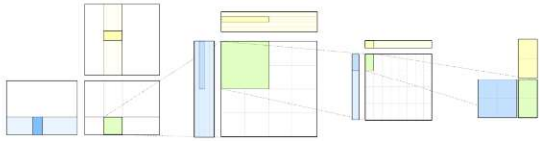
FEEDING THE DATA PATH

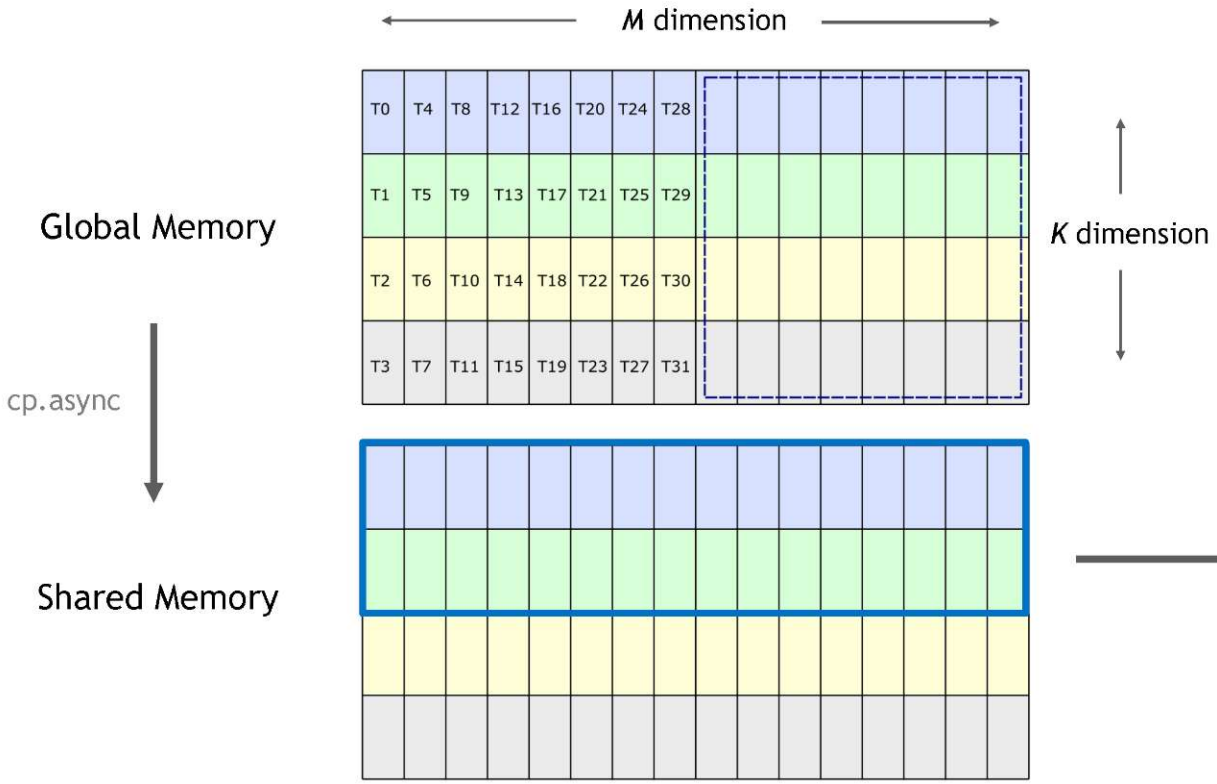
Move data from Global Memory to Tensor Cores as efficiently as possible

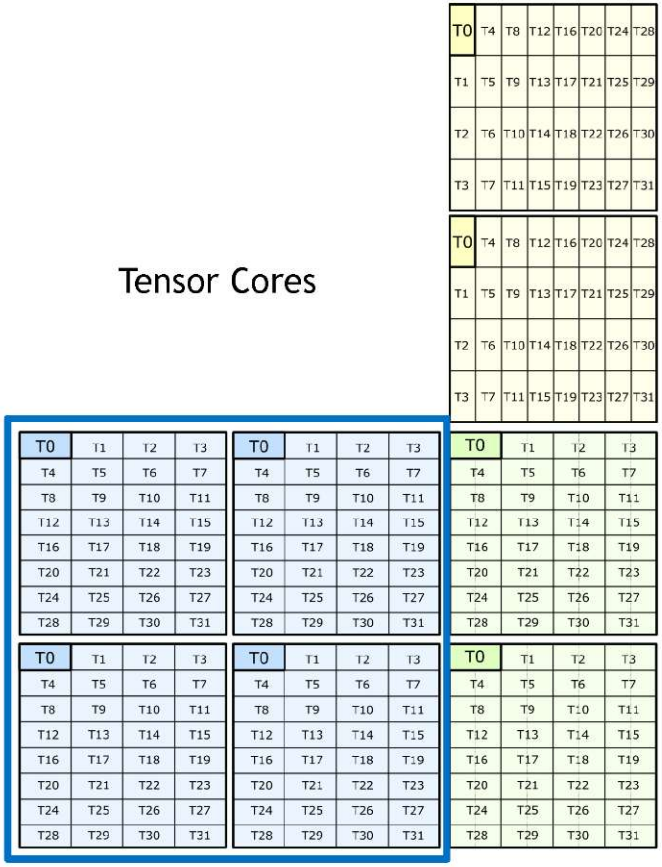
- Latency-tolerant pipeline from Global Memory
- **Conflict-free Shared Memory stores**
- **Conflict-free Shared Memory loads**



GLOBAL MEMORY TO TENSOR CORES







LDMATRIX: FETCH TENSOR CORE OPERANDS

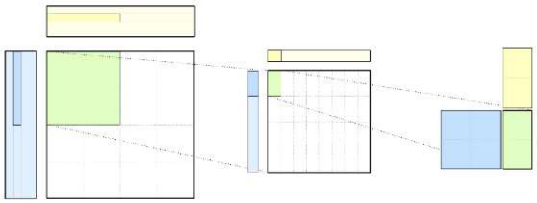
PTX instruction to load a matrix from Shared Memory

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

Shared Memory Pointers

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

Shared Memory Pointers

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

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

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

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

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

LDMATRIX: PTX INSTRUCTION

PTX instruction to load a matrix from SMEM

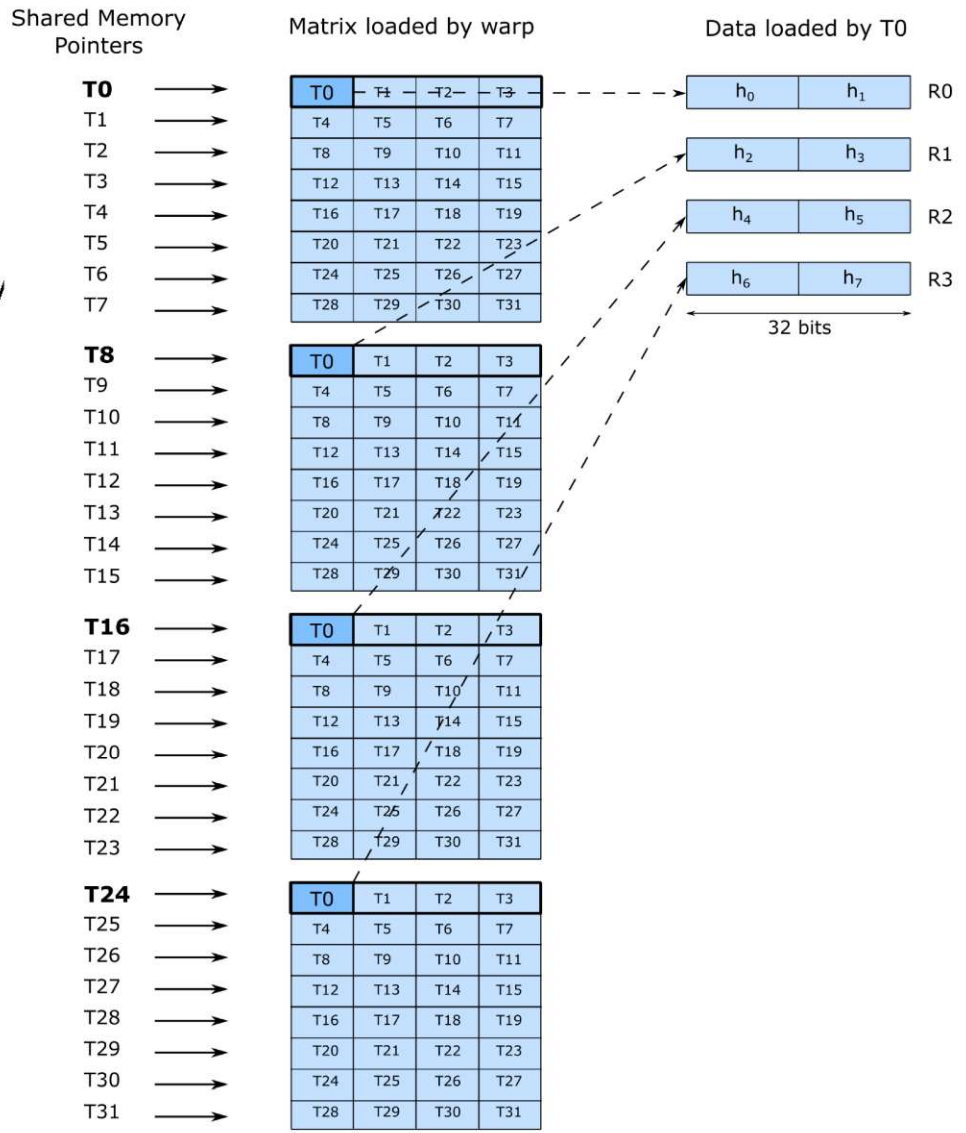
Each thread supplies a pointer to 128b row of data in Shared Memory

Each 128b row is broadcast to groups of four threads

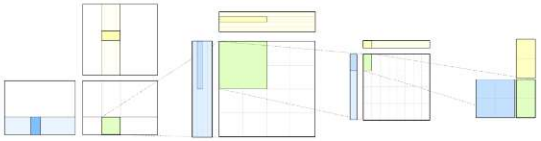
(potentially different threads than the one supplying the pointer)

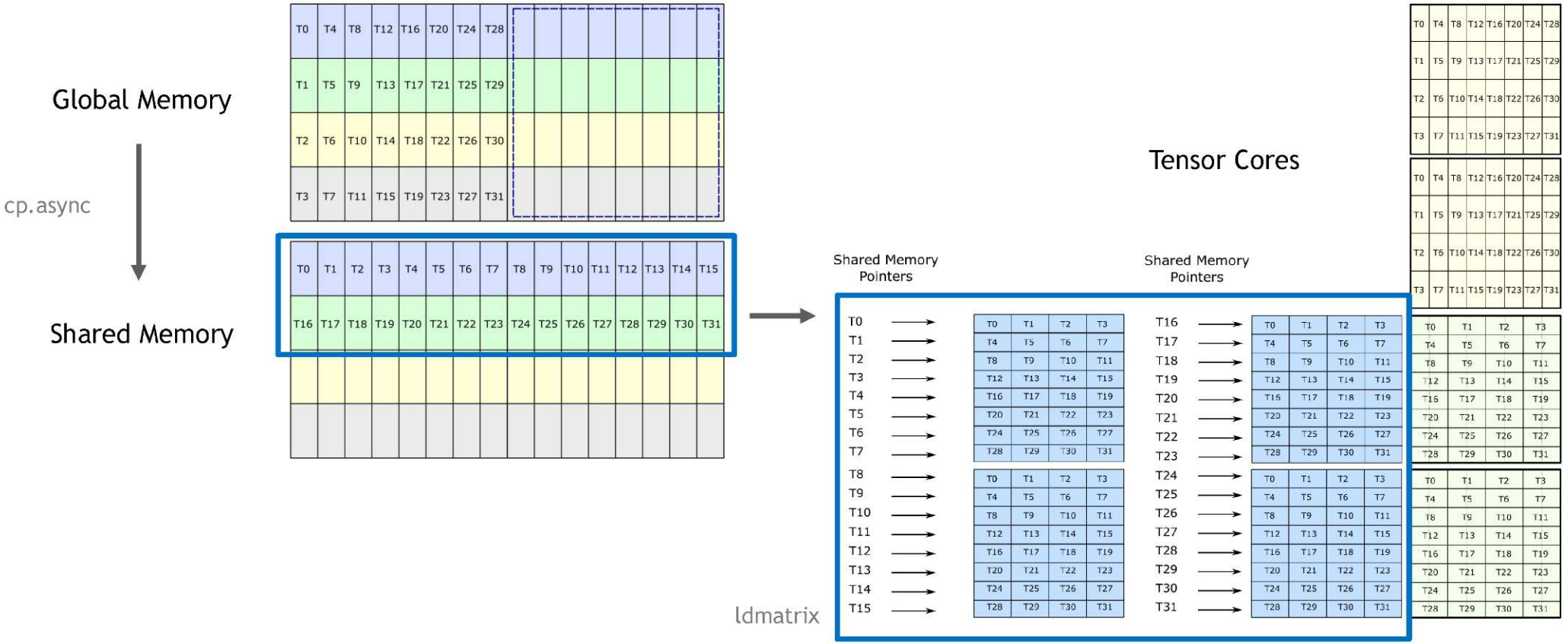
Data matches arrangement of inputs to Tensor Core operations

```
// 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

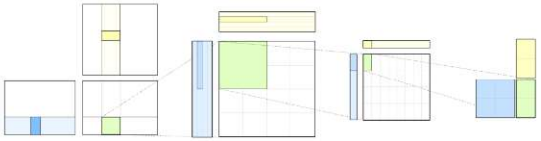
Phase 3: T24 .. T31

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



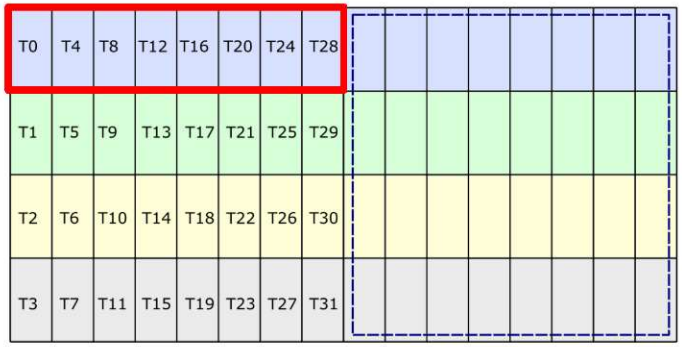
Global Memory

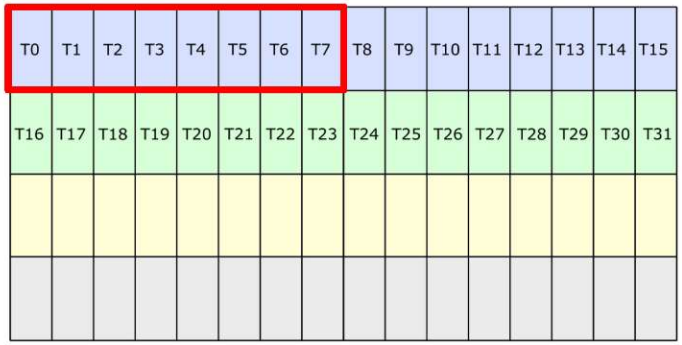
cp.async

Shared Memory

ldmatrix

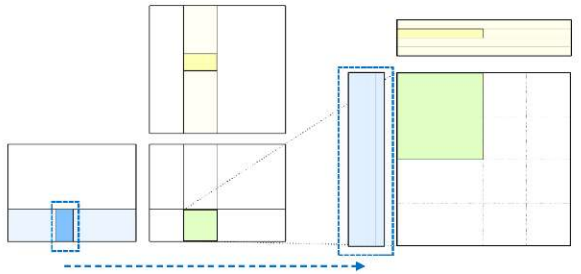
Registers





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)

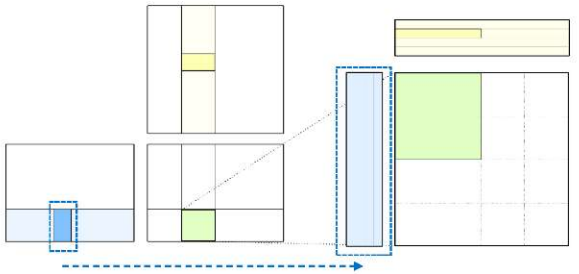
Permuted Shared Memory layout

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)

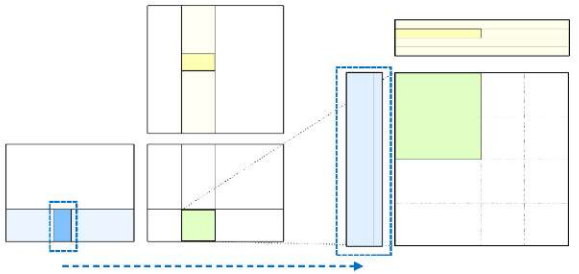
Store
(128 bits per thread)

Store to Shared Memory

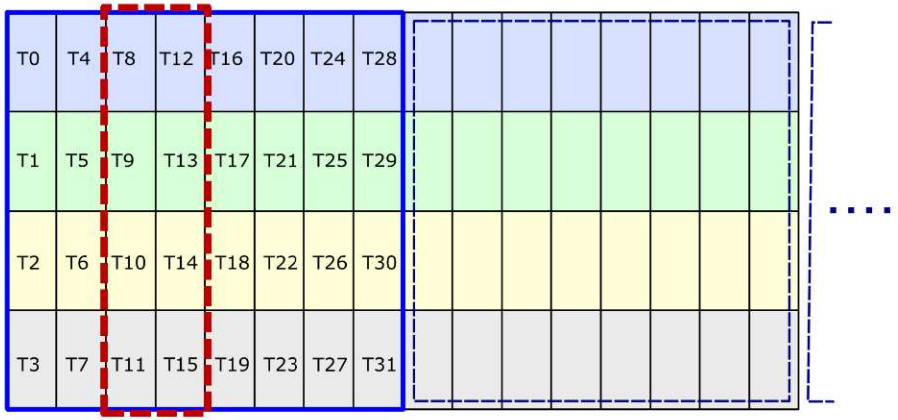
T0	T1	T2	T3	T4	T5	T6	T7
T9	T8	T11	T10	T13	T12	T15	T14
T18	T19	T16	T17	T22	T23	T20	T21
T27	T26	T25	T24	T31	T30	T29	T28

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

GLOBAL TO SHARED MEMORY



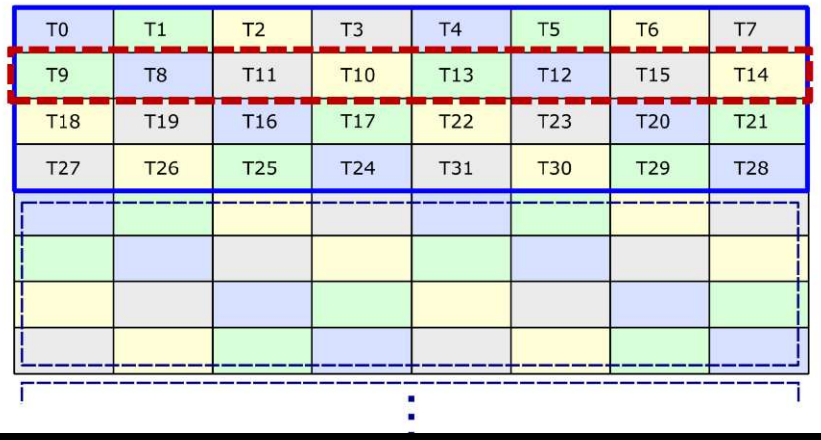
Load from Global Memory



Load
(128 bits per thread)

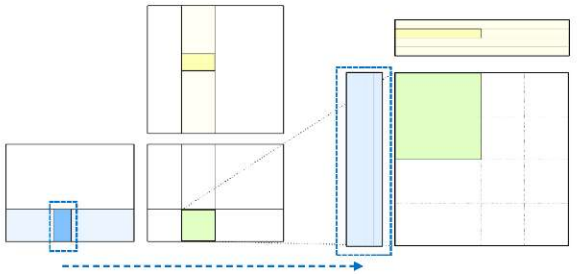
Store
(128 bits per thread)

Store to Shared Memory

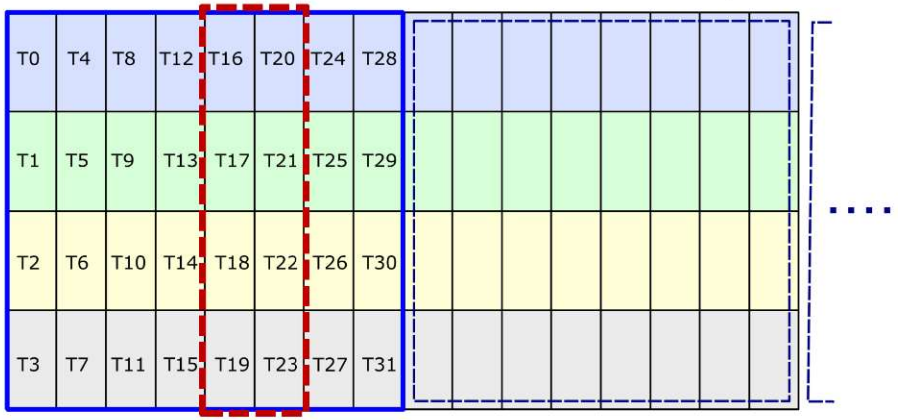



```
Phase 0:  T0 .. T7
Phase 1:  T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
```

GLOBAL TO SHARED MEMORY



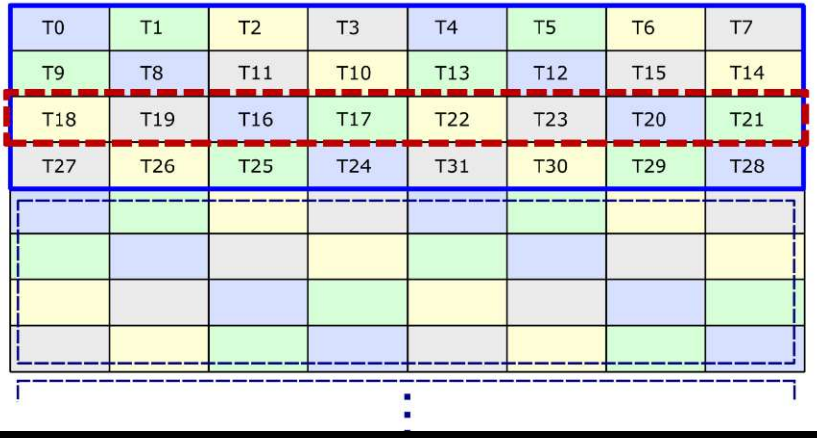
Load from Global Memory



Load
(128 bits per thread)

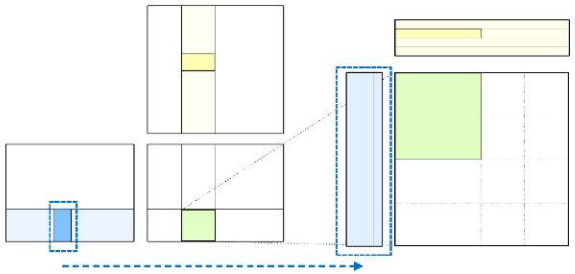
Store
(128 bits per thread)

Store to Shared Memory

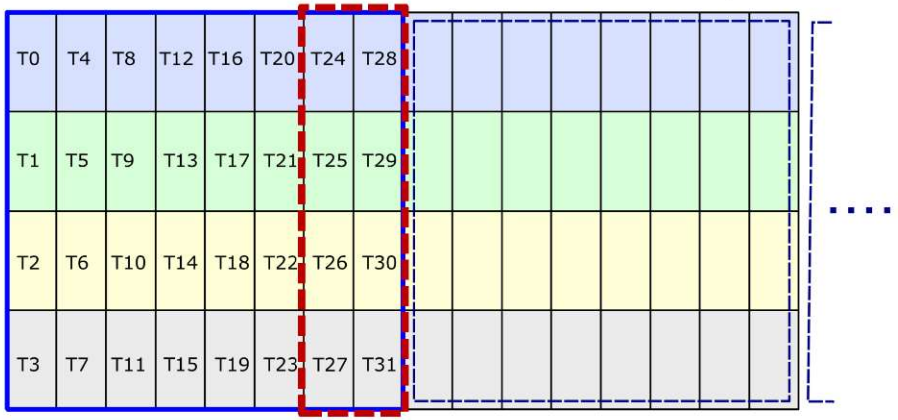



```
Phase 0:  T0 .. T7
Phase 1:  T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
```

GLOBAL TO SHARED MEMORY



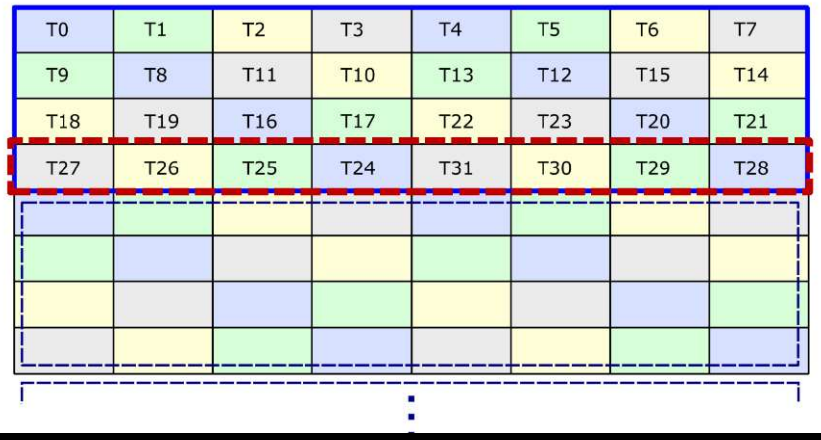
Load from Global Memory



Load
(128 bits per thread)

Store
(128 bits per thread)

Store to Shared Memory

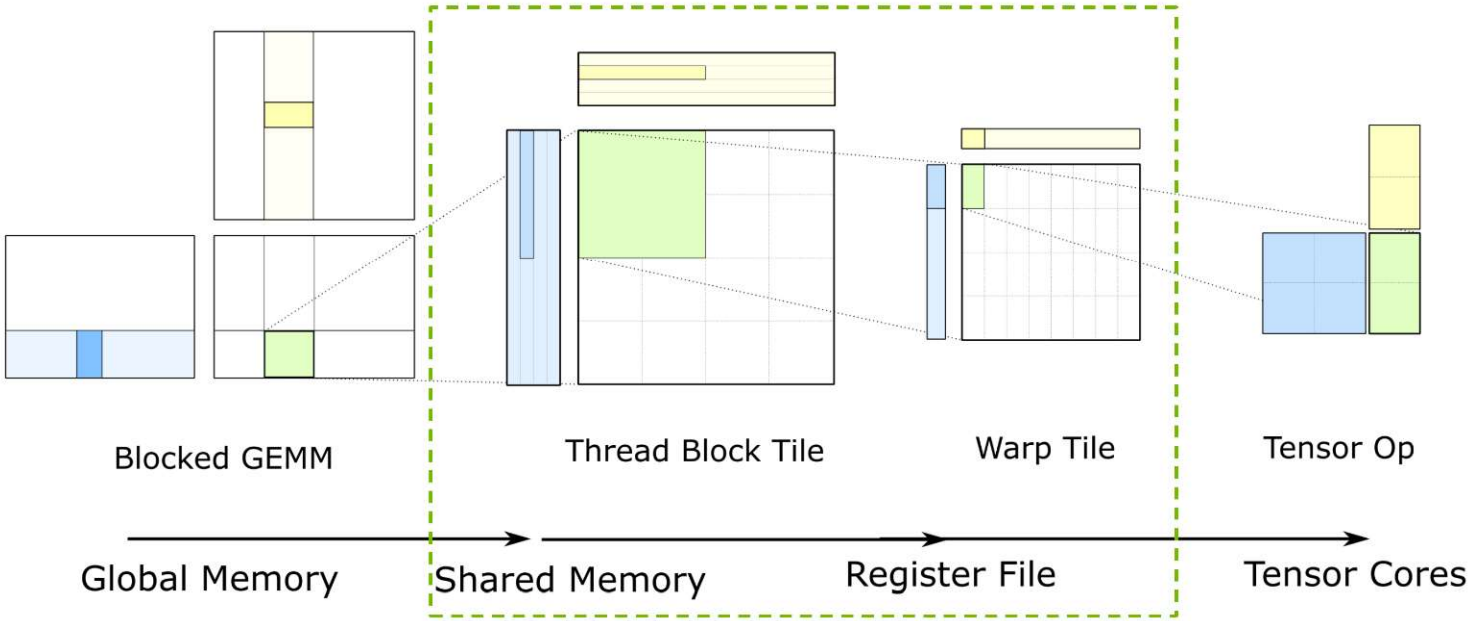



```
Phase 0:  T0 .. T7
Phase 1:  T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
```

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**



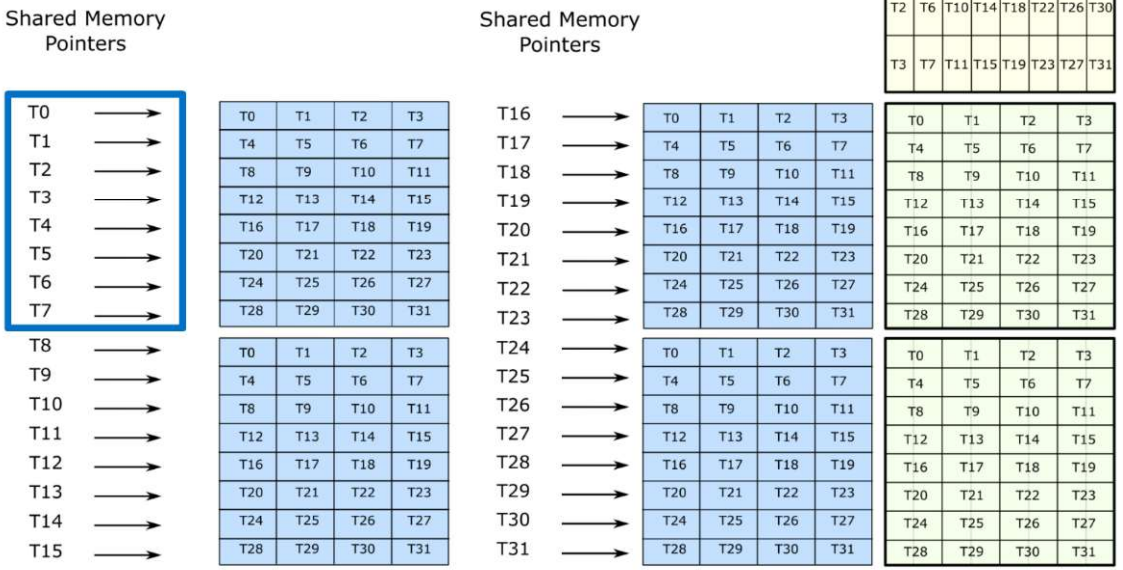
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

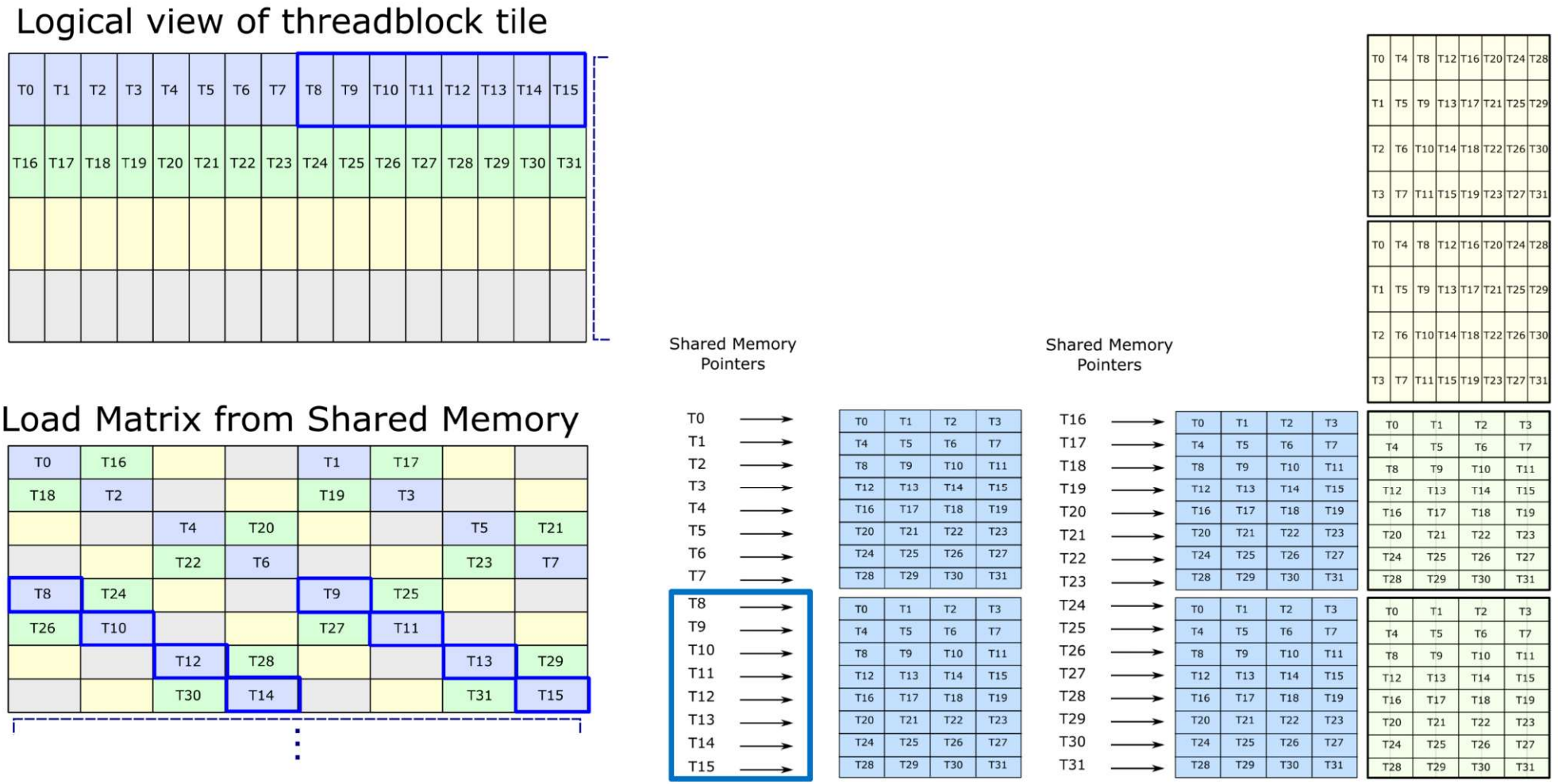
T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

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

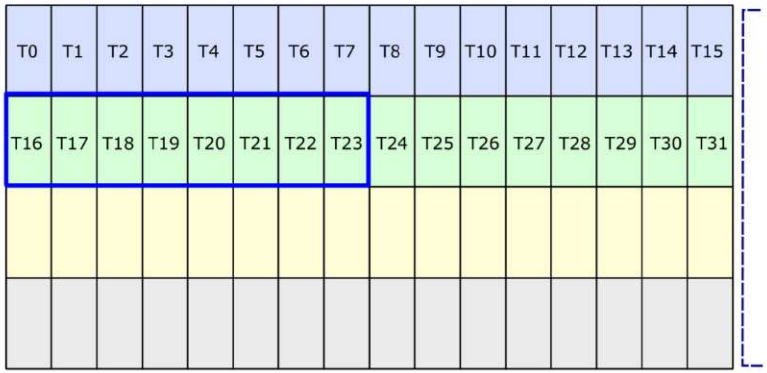


LOADING FROM SHARED MEMORY TO REGISTERS

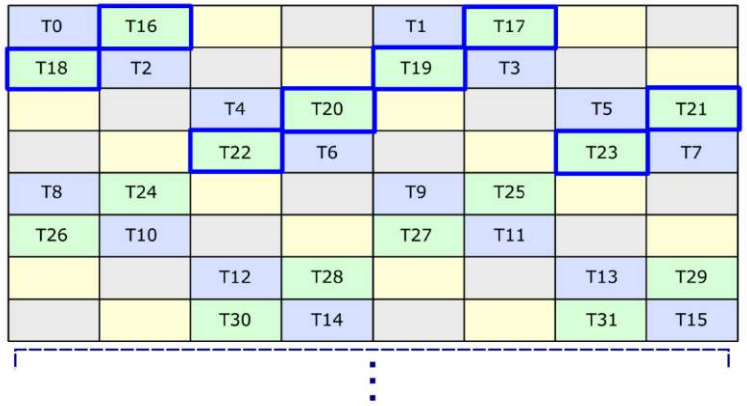


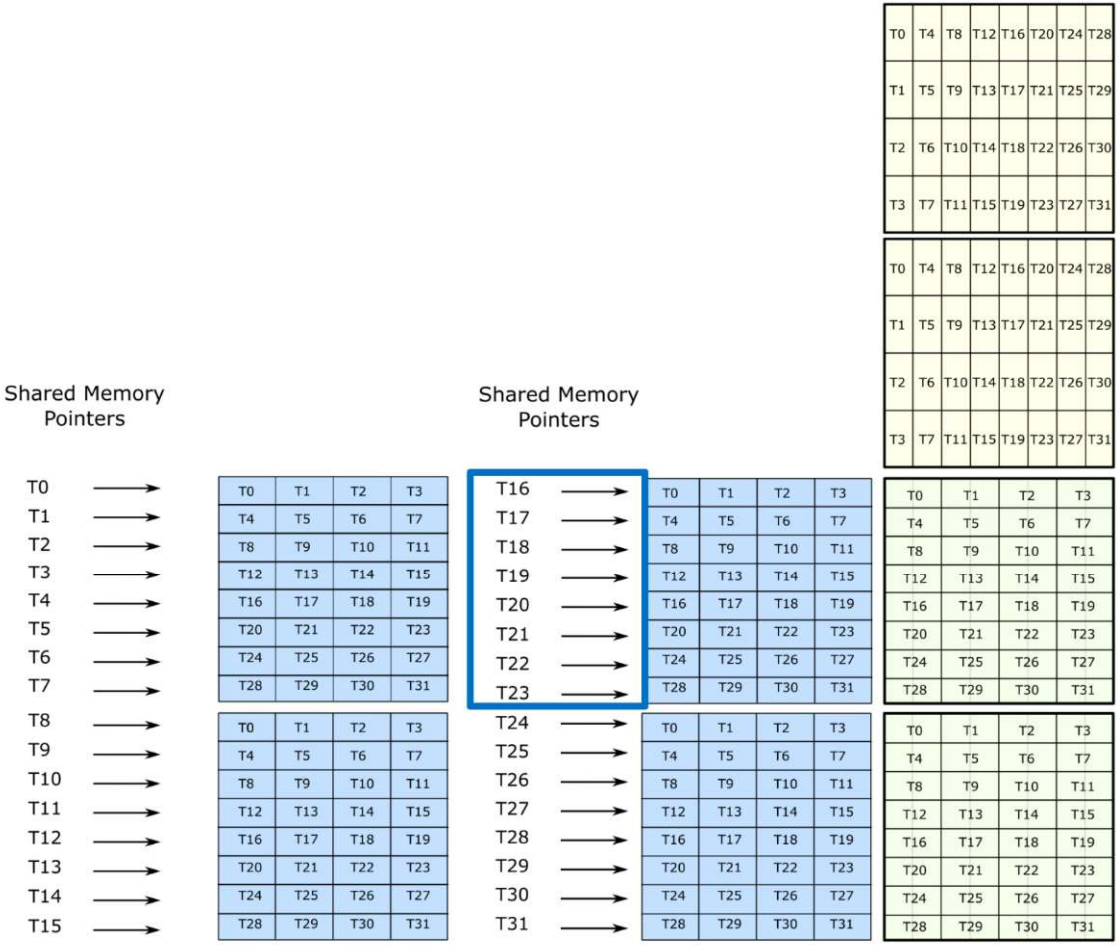
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile



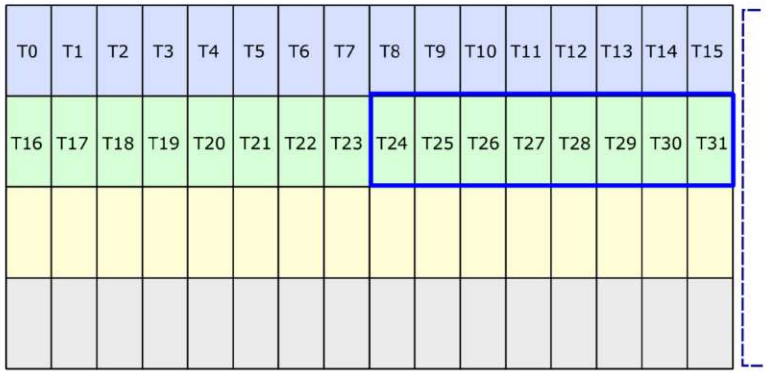
Load Matrix from Shared Memory



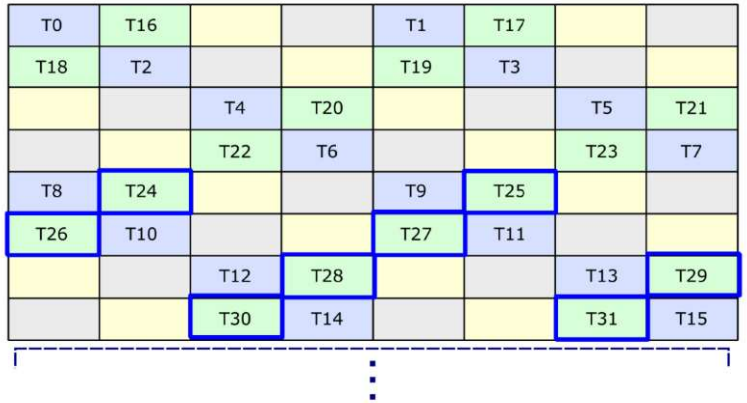


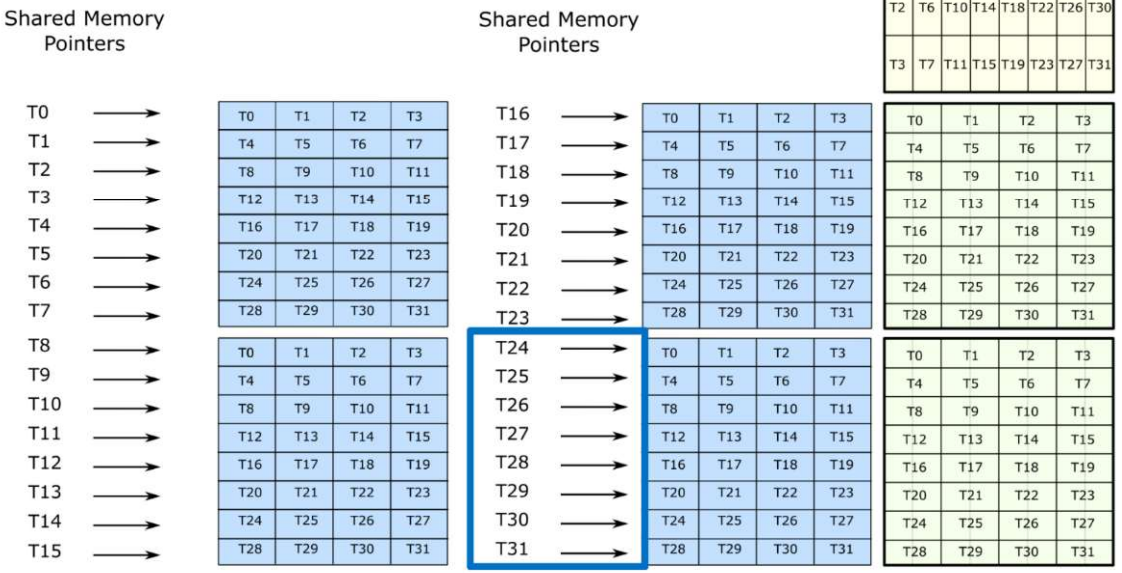
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

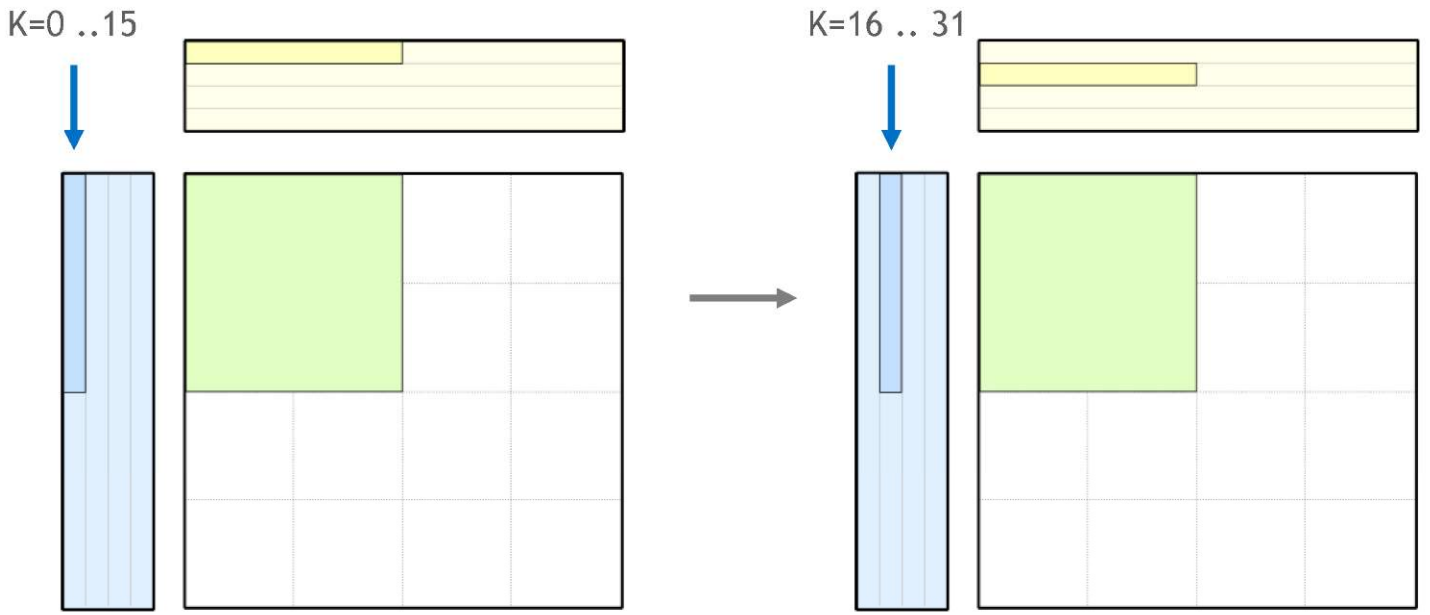


Load Matrix from Shared Memory

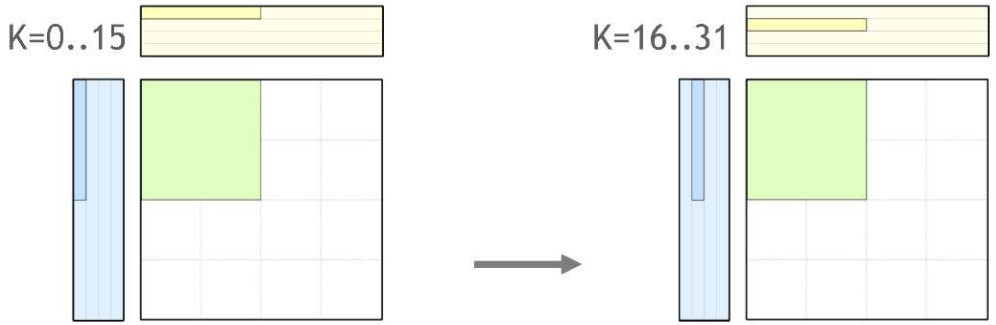




ADVANCING TO NEXT K GROUP



ADVANCING TO NEXT K GROUP



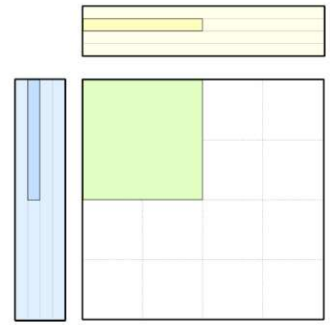
T0	T16			T1	T17		
T18	T2			T19	T3		
		T4	T20			T5	T21
		T22	T6			T23	T7
T8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		T30	T14			T31	T15

		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;

smem_ptr = smem_ptr ^ 2;

LOADING FROM SHARED MEMORY TO REGISTERS



Logical view of threadblock tile

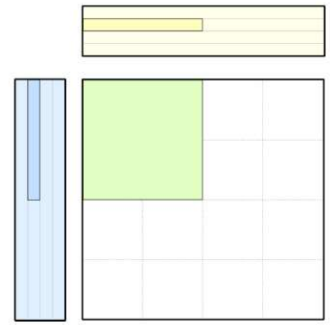
Phase 0

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

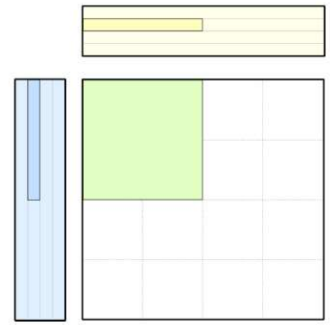
Logical view of threadblock tile

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

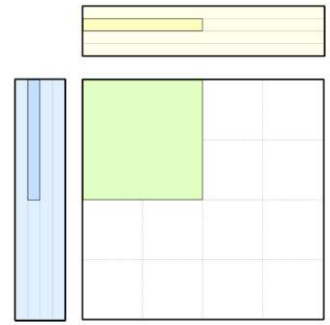
Phase 2

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

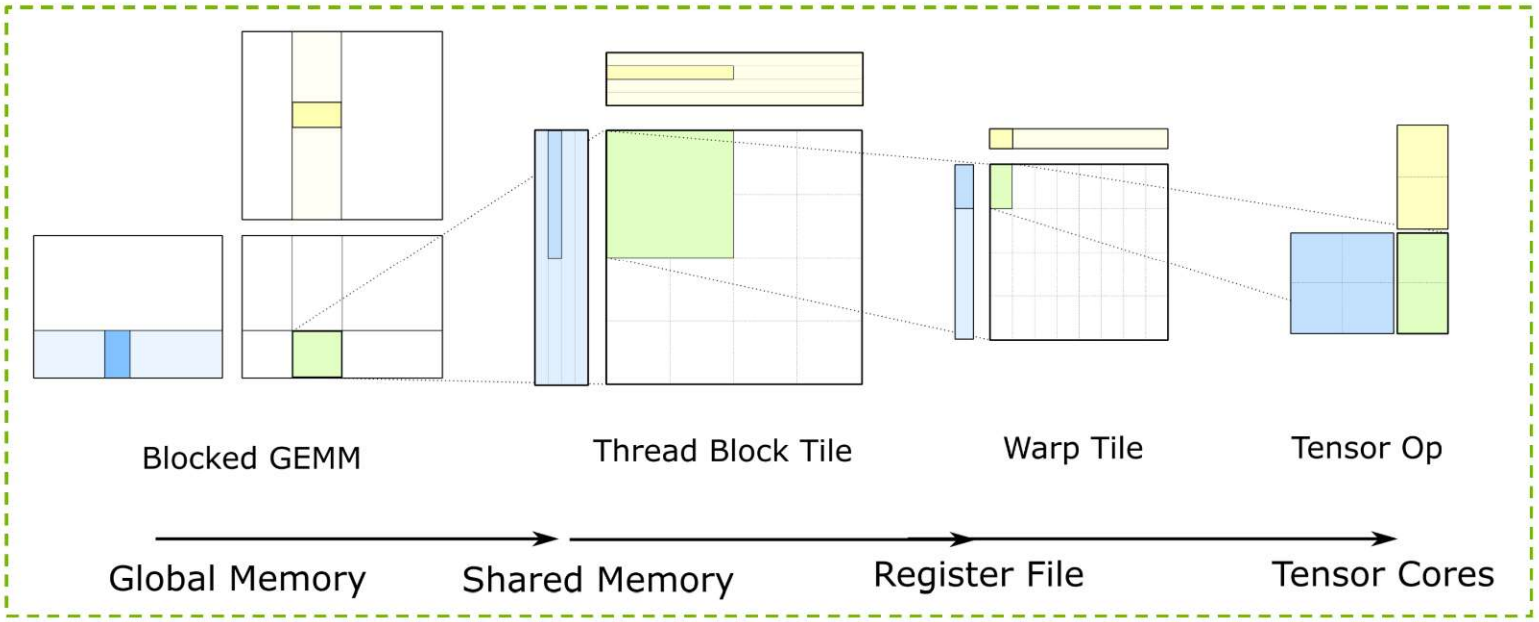
Load Matrix from Shared Memory

		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		

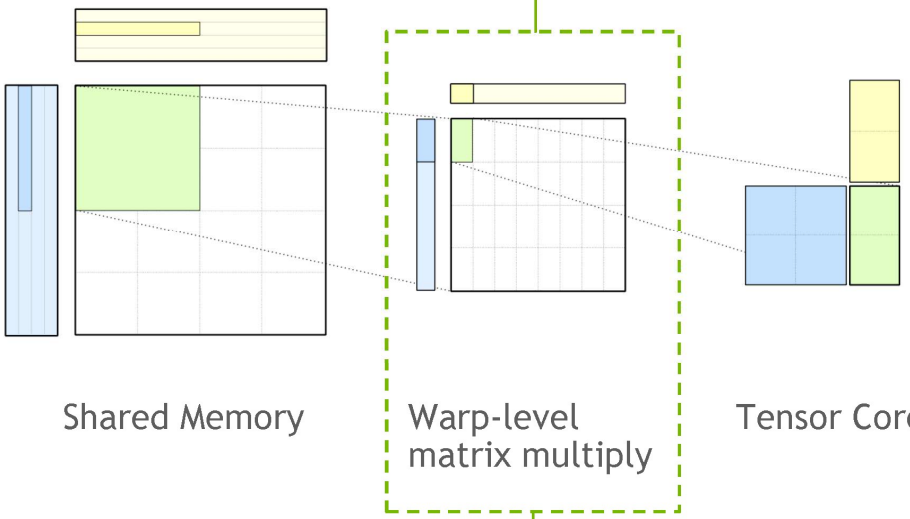
CUTLASS

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.