

King Abdullah University of Science and Technology

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

Markus Hadwiger, KAUST

Reading Assignment #15++



Further suggested reading:

- Raihan et al., arXiv, Feb 2019, Modeling Deep Learning Accelerator Enabled GPUs
 - https://arxiv.org/abs/1811.08309
 - See also GPGPU-SIM: http://www.gpgpu-sim.org/
- CUTLASS 2.8 template library (last update Nov 2021)
 - https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/
 - https://github.com/NVIDIA/cutlass
- Register Cache: Caching for Warp-Centric CUDA Programs
 - https://developer.nvidia.com/blog/register-cache-warp-cuda/
- cuSPARSE library description in the CUDA SDK
- CUSP library: http://cusplibrary.github.io/
- Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS, Maxim Naumov

- https://developer.download.nvidia.com/assets/cuda/files/psts_white_paper_final.pdf



Programming Tensor Cores

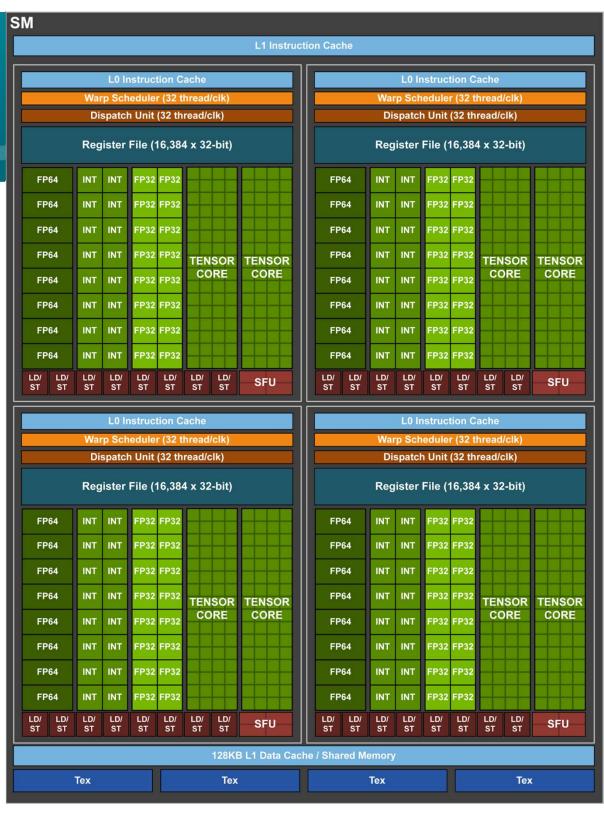
NVIDIA Volta SM

Multiprocessor: SM

- 64 FP32 + INT32 cores
- 32 FP64 cores
- 8 tensor cores (FP16/FP32 mixed-precision)

4 partitions inside SM

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



NVIDIA Turing SM

Multiprocessor: SM

- 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 each
 - 2 Turing tensor cores each
 - Each has: warp scheduler, dispatch unit, 16K register file



NVIDIA GA100 SM

Multiprocessor: SM

- 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 each
- 1 3rd gen tensor core each
- Each has: warp scheduler, dispatch unit, 16K register file



NVIDIA GA102 SM

Multiprocessor: SM

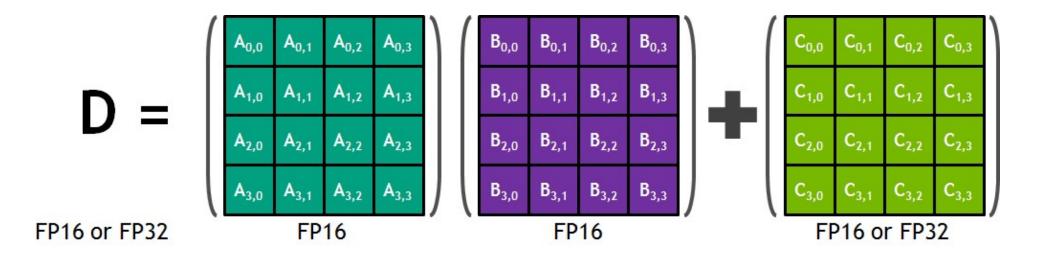
- 128 (64+64) FP32 + 64 INT32 cores
- 2 (!) FP64 cores
- 4 3rd gen tensor cores
- 1 2nd gen RT (ray tracing) core
- 4 partitions inside SM
 - 16+16 FP32 + 16 INT32 cores
 - 4 LD/ST units each
 - 1 3rd gen tensor core each
 - Each has: warp scheduler, dispatch unit, 16K register file



Tensor Cores



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



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

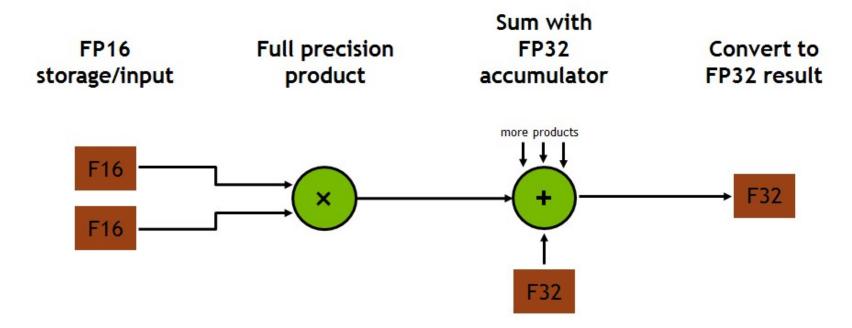
API currently only allows using larger shapes (16x16, ...) in warps (wmma)

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

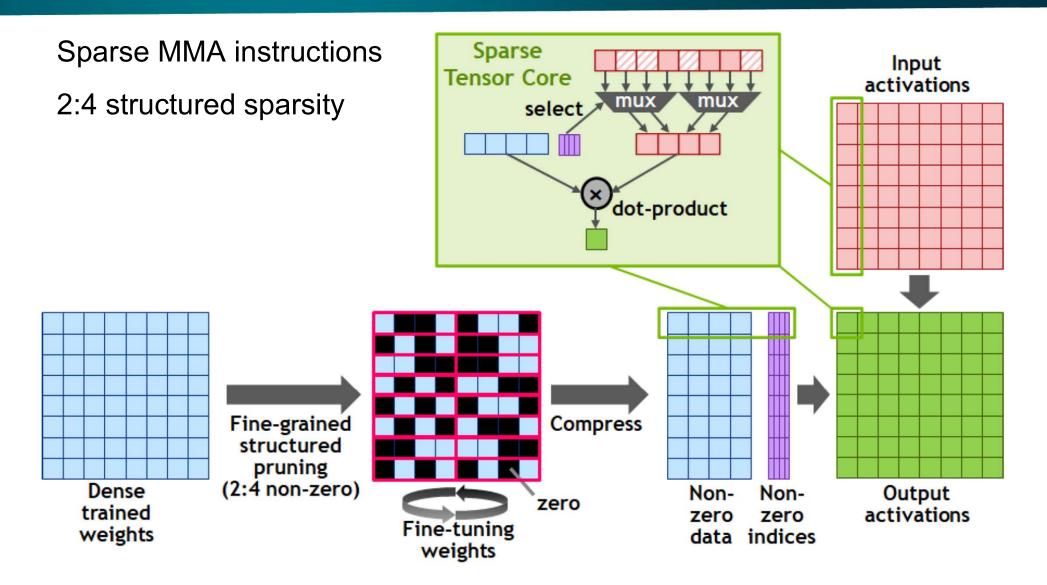


New in Ampere: TF32, BF16, FP64 **FP32 FP32** matrix matrix Precision Range sig exponent mantissa Format to TF32 e8 m23 and multiply **FP32** m10 e8 **TF32** FP32 accumulate e5 m10 **FP16** s e8 m7 FP32 **BF16** s Matrix

plus FP64 (new in Ampere; GA100 only)

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

Ampere Tensor Cores: Sparsity Support



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)

High-level options

- NVIDIA CUTLASS (template abstractions for hi-perf matrix-multiplies)
- NVIDIA cuBLAS
- NVIDIA cuDNN
- Integration into TensorFlow, ...



Warp Level Matrix Multiply Accumulate (WMMA)

```
CUDA C Programming Guide (11.5), 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



Data types (T)

Volta, Turing, and Ampere:

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	ned char unsigned char int		32x8x16	
unsigned char	char unsigned char int		8x32x16	
signed char	nar signed char int		16x16x16	
signed char	signed char int		32x8x16	
signed char	signed char	int	8x32x16	



Data types (T)

Alternate Floating Point support:

Ampere only:

wmma API splits this into fragments

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:

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

Experimental support for sub-byte operations:

Turing and Ampere:

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



Warp Level Matrix Multiply Accumulate (WMMA)

CUDA C Programming Guide (11.5), Appendix B.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 ISA 7.5, Section 9.7.13

Instruction	Sparsity	Multiplicand Data-type	Shape	PTX ISA version
wmma	Dense	Floating-pointf16	.m16n16k16, .m8n32k16,and .m32n8k16	PTX ISA version 6.0
wmma	Dense	Alternate floating-point formatbf16	.m16n16k16, .m8n32k16,and .m32n8k16	PTX ISA version 7.0
wmma	Dense	Alternate floating-point formattf32	.m16n16k8	PTX ISA version 7.0
wmma	Dense	Integeru8/.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-bitb1	.m8n8k128	PTX ISA version 6.3 (preview feature)



PTX ISA 7.5

Instruction Sparsity		Multiplicand Data-type	Shape	PTX ISA version
mma	Dense	Floating-pointf64	.m8n8k4	PTX ISA version 7.0
mma	Dense	Floating-pointf16	.m8n8k4	PTX ISA version 6.4
			.m16n8k8	PTX ISA version 6.5
			.m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point formatbf16	.m16n8k8 and .m16n8k16	PTX ISA version 7.0
mma	Dense	Alternate floating-point formattf32	.m16n8k4 and .m16n8k8	PTX ISA version 7.0
mma	Dense	Integeru8/.s8	.m8n8k16	PTX ISA version 6.5
			.m16n8k16 and .m16n8k32	PTX ISA version 7.0
mma Dense		Sub-byte integer -	.m8n8k32	PTX ISA version 6.5
		.u4/.s4	.m16n8k32 and .m16n8k64	PTX ISA version 7.0
mma	Dense	Single-bitb1	.m8n8k128, .m16n8k128, and .m16n8k256	PTX ISA version 7.0
mma	Sparse	Floating-pointf16	.m16n8k16 and .m16n8k32	PTX ISA version 7.1
mma	Sparse	Alternate floating-point formatbf16	.m16n8k16 and .m16n8k32	PTX ISA version 7.1
mma	Sparse	Alternate floating-point formattf32	.m16n8k8 and .m16n8k16	PTX ISA version 7.1
mma	Sparse	Integeru8/.s8	.m16n8k32 and .m16n8k64	PTX ISA version 7.1
mma	Sparse	Sub-byte integer - .u4/.s4	.m16n8k64 and .m16n8k128	PTX ISA version 7.1



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 = {.ml6n16k16, .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 };
```



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};
```



wmma example

```
.global .align 32 .f16 A[256], B[256];
.global .align 32 .f32 C[256], D[256];
.req .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};
```



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

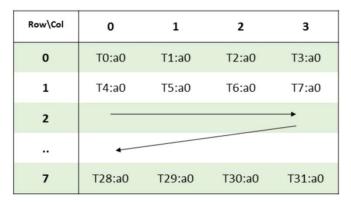
9.7.13.4.2. Matrix Fragments for mma.m8n8k4 with .f64 floating point type

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.	а0



%laneid:{fragments}

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 <code>%laneid</code> O-3 (low group) and 16-19 (high group)
MMA computation 2	Threads with <code>%laneid 4-7</code> (low group) and 20-23 (high group)
MMA computation 3	Threads with <code>%laneid 8-11</code> (low group) and 24-27 (high group)
MMA computation 4	Threads with <code>%laneid 12-15</code> (low group) and 28-31 (high group)

Row Major:

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

4	T24: { a0, a1, a2, a3 }
-	ł
7	T27: { a0, a1, a2, a3 }

Row\Co

0

3

MMA computation 2

Row\Col	0	1	2	3			
0		T4:{ a0,	a1, a2, a3}				
-		Ļ					
3	T7: { a0, a1, a2, a3 }						
4	T20: { a0, a1, a2, a3 }						
-	Ļ						
7		T23: { a0,	a1, a2, a3 }				
				1230			

MMA computation 4

MMA computation 3

2

T8:{ a0, a1, a2, a3}

T11: { a0, a1, a2, a3 } T24: { a0, a1, a2, a3 }

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

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

%laneid:{fragments}

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

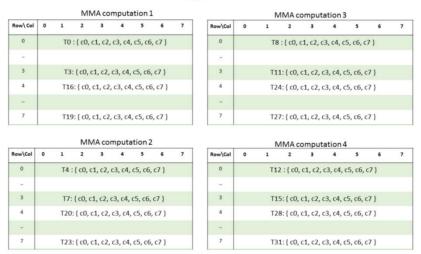
9.7.13.4.1. Matrix Fragments for mma.m8n8k4 with .f16 floating point type

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 <code>%laneid 0-3</code> (low group) and 16-19 (high group)
MMA computation 2	Threads with <code>%laneid 4-7</code> (low group) and 20-23 (high group)
MMA computation 3	Threads with <code>%laneid 8-11</code> (low group) and 24-27 (high group)
MMA computation 4	Threads with <code>%laneid</code> 12-15 (low group) and 28-31 (high group)

.ctype is .f16



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 .f32

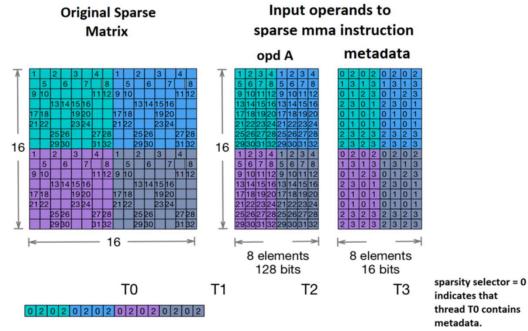
	MMA computation 1								
R\C	0	1	2	3	4	5	6	7	
0	T0:{0	:0, c1 }	T2:{	:0, c1 }	T0:{0	c4, c5 }	T2:{0	4, c5 }	
1	T1:{0	:0, c1 }	T3:{0	0, c1 }	T1:{0	c4, c5 }	T3:{c	4 , c5 }	
2	T0:{0	2, c3 }	T2:{	2, c3 }	T0:{0	c6, c7 }	T2:{0	6, c7 }	
3	T1:{0	2, c3 }	T3:{0	2, c3 }	T1:{0	c6, c7 }	T3:{c	6, c7 }	
4	T16:{	c0, 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)	



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.



25



Load and store: mma Idmatrix

Warp-wide load matrix instruction

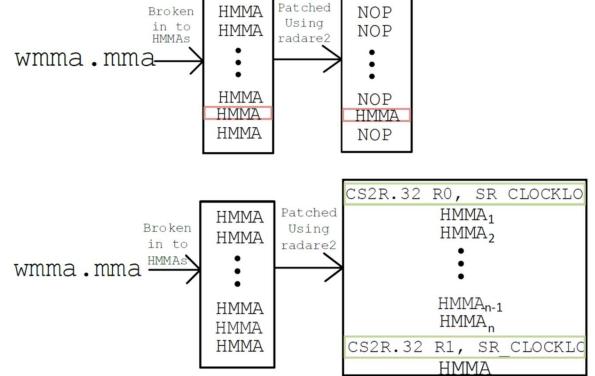
```
// Load a single 8x8 matrix using 64-bit addressing
.reg .b64 addr;
.reg .b32 d;
ldmatrix.sync.aligned.m8n8.x1.shared.b16 {d}, [addr];
// Load two 8x8 matrices in column-major format
.reg .b64 addr;
.reg .b32 d<2>;
ldmatrix.sync.aligned.m8n8.x2.trans.shared.b16 {d0, d1}, [addr];
// Load four 8x8 matrices
.reg .b64 addr;
.reg .b32 d<4>;
ldmatrix.sync.aligned.m8n8.x4.b16 {d0, d1, d2, d3}, [addr];
```



Raihan et al., 2019

Get SASS code from cuobjdump disassembly

Micro-benchmarking

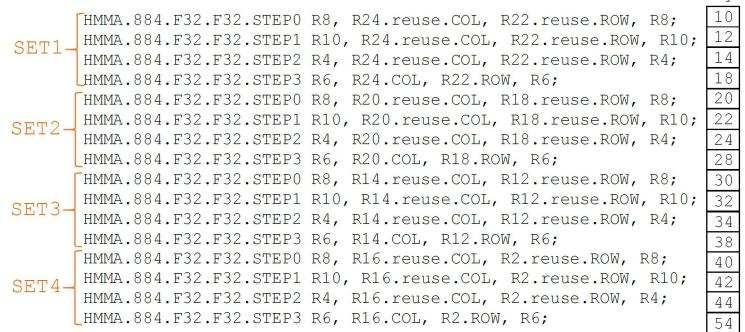




Raihan et al., 2019

Get SASS code from cuobjdump disassembly

Cumulative ClockCycles

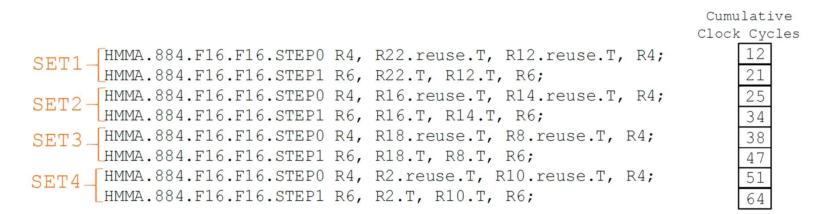


(a) Disassembled SASS instructions for Mixed precision mode



Raihan et al., 2019

Get SASS code from cuobjdump disassembly



(b) Disassembled SASS instructions for FP16 mode



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

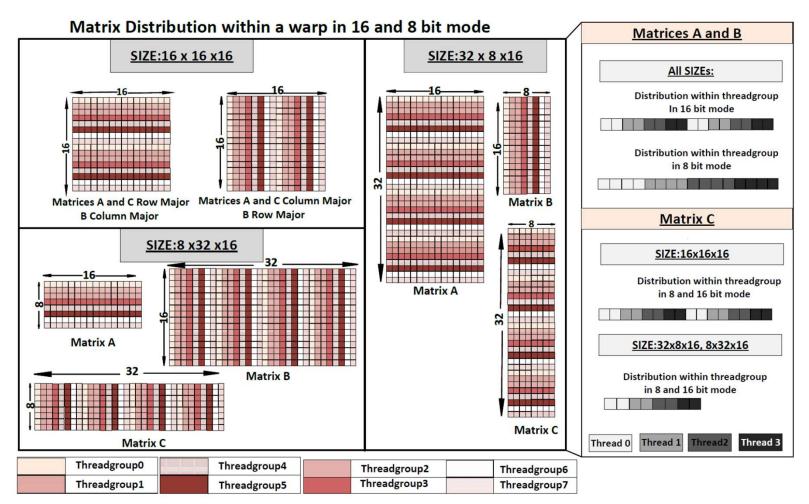


Figure 8: Distribution of operand matrix elements to threads for tensor cores in the RTX 2080 (Turing).



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

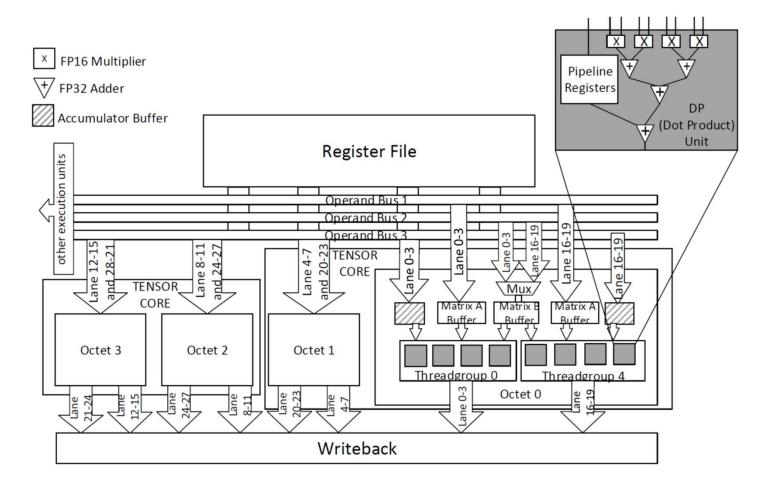


Figure 13: Proposed Tensor Core Microarchitecture



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

Andrew Kerr, May 21, 2020

NVIDIA AMPERE ARCHITECTURE

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

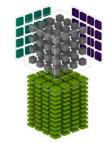
Bfloat16, double, Tensor Float 32

Asynchronous copy

Copy directly into shared memory - deep software pipelines

Many additional new features - see "Inside NVIDIA Ampere Architecture"





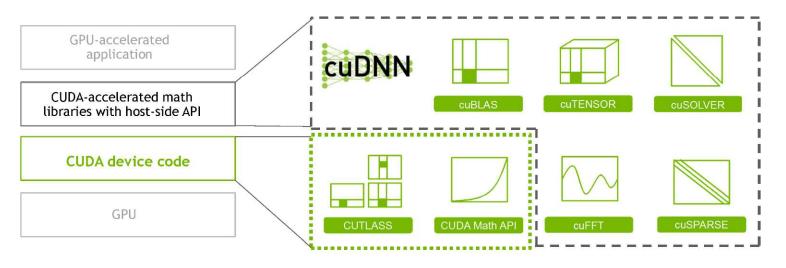
PROGRAMMING NVIDIA AMPERE ARCHITECTURE

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

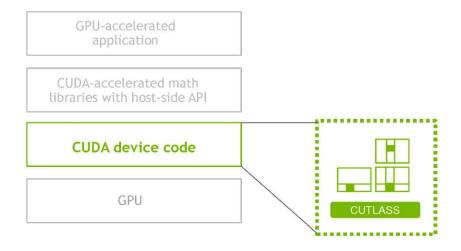
• CUTLASS, CUDA Math API, CUB, Thrust, libcu++



PROGRAMMING NVIDIA AMPERE ARCHITECTURE with CUDA C++



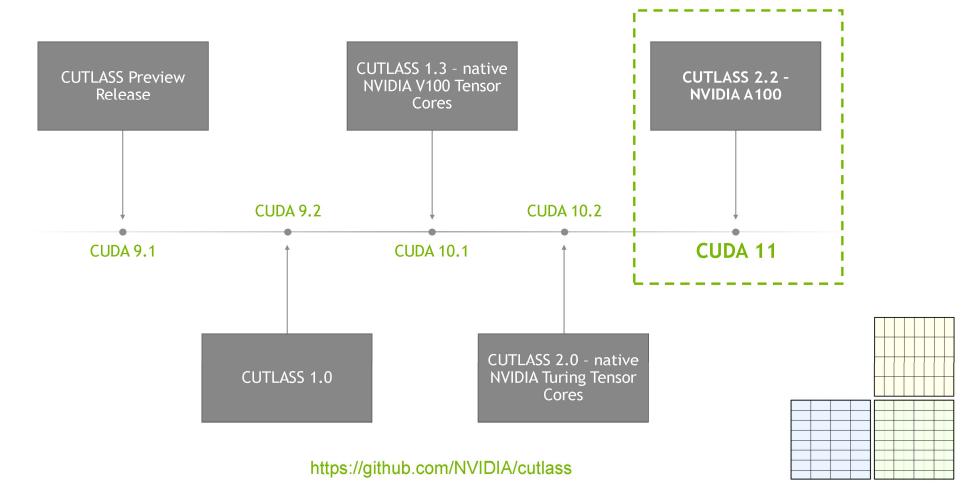
This is a talk for CUDA programmers





CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



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

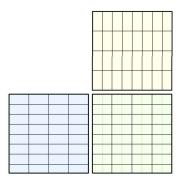
CUTLASS 2.1

- 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 250,000 20,000 1,000,000 Tensor Core - INT4 Tensor Core - BF16, F16 Tensor Core - F64 18,000 900,000 200,000 16,000 800,000 13.8x 14,000 13x 700,000 2x 150,000 12,000 600,000 GFLOP/s 10'000 GFLOP/s GFLOP/s Tensor Core - INT8 500,000 CUDA Core - F64 Tensor Core - TF32 100,000 8,000 400,000 6,000 300,000 7.7x 5.7x 50,000 4,000 200,000 CUDA Core - INT8 CUDA Core - F32 100,000 2,000 0 0 0 128 1152 2176 3200 4224 5248 5248 6272 7296 8320 10368 11392 12416 13440 32 544 1056 1568 2080 2592 3104 3104 3104 4128 4640 32 160 288 416 544 9344 14464 15488 672 800 6176 6688 7200 1056 1184 1312 1440 1568 1696 1824 1952 5152 5664 928 7712 GEMM K GEMM K GEMM K m=3456, n=4096 10

TENSOR CORES ON NVIDIA AMPERE ARCHITECTURE

WHAT ARE TENSOR CORES?

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

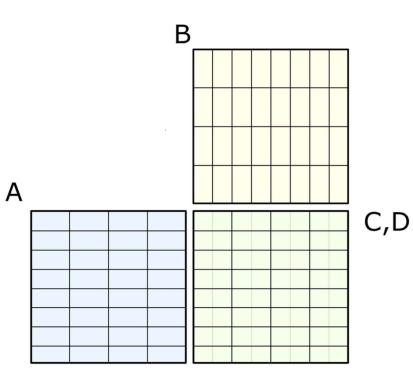
- Matrix multiply-add
- XOR-POPC

Input Data types: A, B

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

Accumulation Data Types: C, D

half, float, int32_t, double



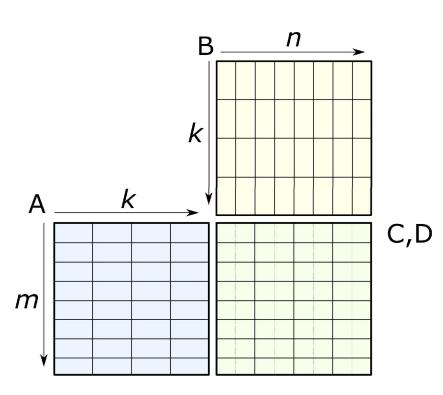
WHAT ARE TENSOR CORES?

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



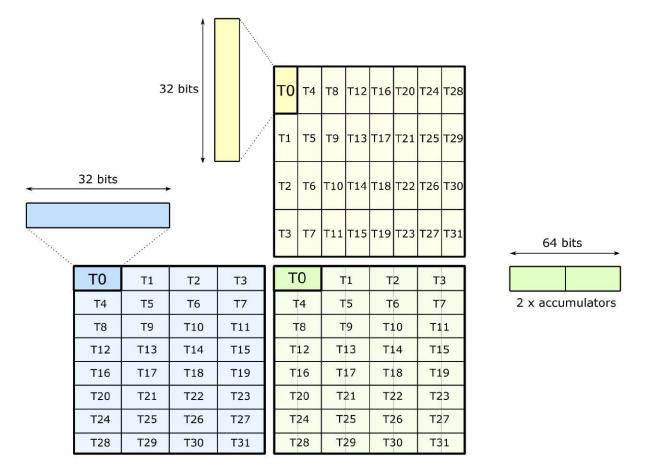
NVIDIA AMPERE ARCHITECTURE - TENSOR CORE OPERATIONS

ΡΤΧ	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

		32	2 bits	b ₀ b ₁ b ₂ b ₃	ТО т1	T4 T5				т20 т21		
<u>جــــــــــــــــــــــــــــــــــــ</u>	32 bits				т2	т6	Т10	T14	T18	T22	T26	т30
b ₀	b ₁ b ₂	2 b ₃			т3	Т7	T 11	Т15	Т19	T23	Т27	т31
	Т0	T1	Т2	Т3	Т	0	Т	1	Ţ	2	T:	3
	T4	Т5	Т6	T7	٦	4	т	5	Т	6	T	7
	Т8	Т9	T10	T11	٦	8	т	9	Τſ	0	Т1	1
	T12	T13	T14	T15	Т	12	T:	13	Τſ	.4	Τ1	5
	T16	T17	T18	Т19	т	16	T:	17	T	.8	Τ1	9
	T20	T21	T22	T23	Т	20	T:	21	T2	22	Т2	3
	T24	T25	T26	T27	Т	24	T2	25	T2	26	Т2	7
	T28	T29	Т30	T31	Т	28	T2	29	T3	30	Т3	1

mma.sync.aligned (via inline PTX)

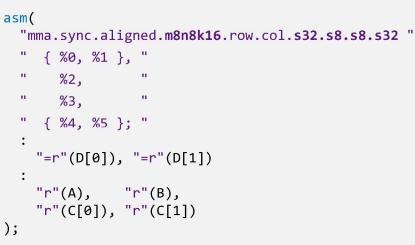
int32_t	D[2];
<pre>uint32_t const</pre>	A;
<pre>uint32_t const</pre>	В;
<pre>int32_t const</pre>	C[2];

64 bits

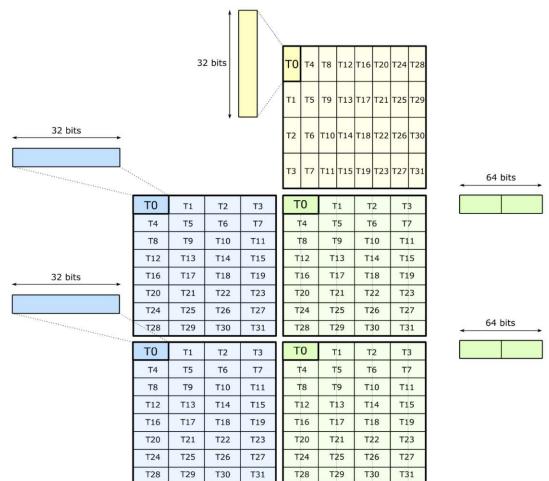
 r_1

r_o

// Example targets 8-by-8-by-16 Tensor Core operation



EXPANDING THE M DIMENSION



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

F16 * F16 + F32

16-by-8-by-8

32 bits		3.	2 bits	h _o		T0 T1 T2 T3	Т5 Т6	т9 т10	т13 т14	T17 T18	T21 T22	T24 T25 T26 T27	T29 T30
	Т0	T1	Т2	Т3		Т	0	т	1	т	2	T.	3
	T4	T5	Т6	T7		Т	4	Т	5	т	6	T	7
	Т8	Т9	T10	T11		Т	8	Т	9	T	.0	Т1	1
	T12	T13	T14	T15		т	12	T:	13	T	4	T1	5
32 bits	T16	T17	T18	T19		т	16	T	17	T18	T19	9	
b b	T20	T21	T22	T23		T.	20	T21		T22	Т2	3	
h ₀ h ₁	T24	T25	T26	T27		T.	24	T:	25	T2	26	T2	7
and the second	T28	T29	T30	T31	L	Т.	28	T	29	T3	30	Т3	1
	Т0	T1	T2	Т3	٢	Т	0	Т	1	Т	2	T.	3
	T4	T5	Т6	T7	Γ	т	4	т	5	т	6	Т,	7
	Т8	Т9	T10	T11		т	8	т	9	т	.0	T1	1
	T12	T13	T14	T15		т	12	T	13	τı	.4	T1	5
	T16	T17	T18	T19	T	т	16	T	17	T	.8	T1	9
	T20	T21	T22	T23		Т.	20	T:	21	T2	22	Т2	3
	T24	T25	T26	T27	T	т	24	T:	25	T2	26	Т2	7
	T28	T29	T30	T31	t	T	28	T	29	T3	30	T3	1

mma.sync.aligned (via inline PTX)

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

64 bits

64 bits

r₂

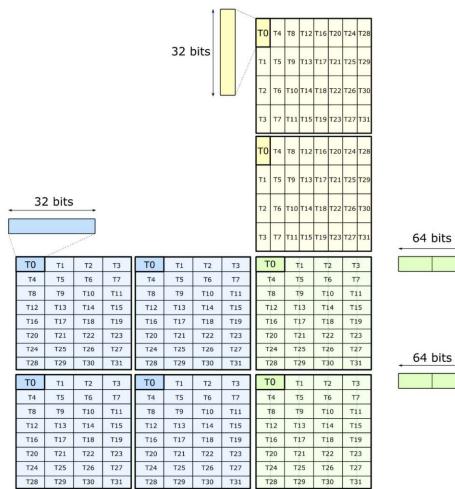
r₃

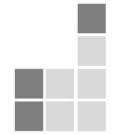
 r_1

ro

// Example targets 16-by-8-by-8 Tensor Core operation

EXPANDING THE K DIMENSION





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

F16 * F16 + F32

16-by-8-by-16																	
								ho	то	Т4	т8	T12	т16	т20	Т24	т28	
						32	oits	h ₁	т1	Т5	т9	т13	т17	Т21	Т25	т29	
							Ļ		т2	Т6	T10	т14	т18	т22	T26	т30	
									тз	77	т11	T15	т19	т23	T27	Т31	
									то	т4	т8	T12	т16	т20	T24	т28	
									т1	Т5	т9	т13	т17	Т21	Т25	т29	
+	32	2 bits	_						т2	т6	т10	т14	т18	т22	т26	т30	
	h _o	h	0 <u>1</u>						тз	т7	т11	т15	т19	Т23	т27	т31	
	Т0	T1	T2	Т3	TO	T1	T2	Т3	Т	0	т	1	T	2	T,	3	Γ
	T4	Т5	Т6	Т7				-	_								
					T4	T5	Т6	T7	٦	4	т	5	т	6	т	7	
	T8	Т9	T10	T11	T4 T8	T5 T9	T6 T10	T7 T11		4 8	ia [5 9	-		-	100	
	T8 T12	T9 T13	T10 T14		75255	COSTAN MARKAN	ULCOSES.		1	139 200	т	88 181 - 3	т	.0	т	1	
				T11	т8	Т9	T10	T11	T T	8	T T:	9	TI T1	.0	T T1	1	
	T12	T13	T14	T11 T15	T8 T12	T9 T13	T10 T14	T11 T15	T T	'8 12	T T:	'9 13 17	T1 T1 T1	.0 .4 .8	т т1 т1	1 5 9	
	T12 T16	T13 T17	T14 T18	T11 T15 T19	T8 T12 T16	T9 T13 T17	T10 T14 T18	T11 T15 T19	T T T	8 12 16	т т: т: т:	'9 13 17	T1 T1 T1 T1	.0 .4 .8 .2	T1 T1 T1 T1	1 5 9 3	
	T12 T16 T20	T13 T17 T21	T14 T18 T22	T11 T15 T19 T23	T8 T12 T16 T20	T9 T13 T17 T21	T10 T14 T18 T22	T11 T15 T19 T23	T T T T	8 12 16 20	т т: т: т:	9 13 17 21 25	T1 T1 T1 T1 T2	.0 4 .8 2 2.6	T1 T1 T1 T1 T1 T2	1 5 9 3 7	
	T12 T16 T20 T24	T13 T17 T21 T25	T14 T18 T22 T26	T11 T15 T19 T23 T27	T8 T12 T16 T20 T24	T9 T13 T17 T21 T25	T10 T14 T18 T22 T26	T11 T15 T19 T23 T27	T T T T	8 12 16 20 24 28	Т Т: Т: Т: Т:	9 13 17 21 25	T1 T1 T1 T1 T2 T2	.0 4 .8 2 2 6 0	T1 T1 T1 T1 T2 T2	1 5 9 3 7	-
	T12 T16 T20 T24 T28	T13 T17 T21 T25 T29	T14 T18 T22 T26 T30	T11 T15 T19 T23 T27 T31	T8 T12 T16 T20 T24 T28	T9 T13 T17 T21 T25 T29	T10 T14 T18 T22 T26 T30	T11 T15 T19 T23 T27 T31	T T T T T	8 12 16 20 24 28	T T T T T T	9 13 17 21 25 29	Ti T1 T1 T2 T2 T3	.0 4 8 2 2 6 30 2	T1 T1 T1 T2 T2 T3	1 5 9 3 7 1 3	Ē
	T12 T16 T20 T24 T28 T0	T13 T17 T21 T25 T29 T1	T14 T18 T22 T26 T30 T2	T11 T15 T19 T23 T27 T31 T3	T8 T12 T16 T20 T24 T28	T9 T13 T17 T21 T25 T29 T1	T10 T14 T18 T22 T26 T30 T2	T11 T15 T19 T23 T27 T31 T3	T T T T T	8 12 16 20 24 28 0	т т: т: т: т: т: т: т	9 13 17 21 25 29	T1 T1 T1 T2 T2 T3 T3	0 4 8 22 26 60 22 5	T1 T1 T1 T2 T2 T3 T3	1 5 9 3 7 1 3 7	-
	T12 T16 T20 T24 T28 T0 T4	T13 T17 T21 T25 T29 T1 T5	T14 T18 T22 T26 T30 T2 T2 T6	T11 T15 T19 T23 T27 T31 T3 T7	T8 T12 T16 T20 T24 T28 T0 T4	T9 T13 T17 T21 T25 T29 T1 T5	T10 T14 T18 T22 T26 T30 T2 T6	T11 T15 T19 T23 T27 T31 T3 T7 T7	ו ד ד ד ד ד ד 1	 12 16 20 24 28 0 4 	T TI TI TI TI TI T T	9 13 17 21 25 29 1 5	Ti T11 T11 T2 T2 T3 T3 T, T0	0 4 8 2 2 6 30 2 2 5 .0	TT T1 T1 T1 T2 T2 T3 T3 T3	1 5 9 3 7 1 1 7 1	-
	T12 T16 T20 T24 T28 T0 T4 T8	T13 T17 T21 T25 T29 T1 T5 T9	T14 T18 T22 T26 T30 T2 T2 T6 T10	T11 T15 T19 T23 T27 T31 T3 T7 T11	T8 T12 T16 T20 T24 T28 T0 T4 T8	T9 T13 T17 T21 T25 T29 T1 T5 T9	T10 T14 T18 T22 T26 T30 T2 T6 T10	T11 T15 T19 T23 T27 T31 T3 T3 T7 T11	T T T T T T T T T	 12 16 20 24 28 0 4 8 	T TI TI TI TI TI T T	9 13 17 21 25 29 1 5 9 13	Ti T11 T11 T22 T22 T33 T3 T11 T11	0 4 8 22 26 30 2 5 4	TT T1 T1 T1 T2 T2 T3 T3 T3 T3 T1 T1	1 5 9 3 7 1 1 5	·
	T12 T16 T20 T24 T28 T0 T4 T8 T12	T13 T17 T21 T25 T29 T1 T5 T9 T13	T14 T18 T22 T26 T30 T2 T6 T10 T14	T11 T15 T19 T23 T27 T31 T3 T7 T11 T15	T8 T12 T16 T20 T24 T28 T0 T4 T8 T12	T9 T13 T17 T21 T25 T29 T1 T5 T9 T13	T10 T14 T18 T22 T26 T30 T2 T6 T10 T14	T11 T15 T19 T23 T27 T31 T3 T7 T11 T15	ו ד ד ד ד ד 1 1 ד ד	 12 16 20 24 28 0 4 74 78 12 	т т: т: т: т: т: т т т т т:	9 13 17 21 25 29 11 5 9 13 17	T(T1 T1 T1 T2 T2 T2 T3 T3 T0 T1 T1 T1	0 4 8 22 6 6 0 4 8	Ti T1 T1 T1 T2 T2 T3 T3 T3 T1 T1 T1	1 5 9 3 7 1 3 7 1 5 9	
	T12 T16 T20 T24 T28 T0 T4 T8 T12 T16	T13 T17 T21 T25 T29 T1 T5 T9 T13 T17	T14 T18 T22 T26 T30 T2 T6 T10 T14 T18	T11 T15 T19 T23 T27 T31 T7 T11 T15 T19	T8 T12 T16 T20 T24 T28 T0 T4 T8 T12 T16	T9 T13 T17 T21 T25 T29 T1 T5 T9 T13 T17	T10 T14 T18 T22 T26 T30 T2 T6 T10 T14 T18	T11 T15 T19 T23 T27 T31 T3 T7 T11 T15 T19	г Т Т Т Т Т Т Т Т Т	12 16 20 24 28 0 4 18 12 16	T T: T: T: T: T: T: T: T:	9 13 17 21 25 29 11 5 9 13 17	Ti T1 T1 T1 T2 T2 T3 T3 T3 T1 T1 T1 T1	0 4 8 22 26 6 6 0 2 5 6 .0 4 8 8 22	Ti T1 T1 T1 T2 T2 T3 T3 T3 T1 T1 T1 T1 T1	1 5 9 3 7 1 3 7 1 5 9 3 3	Ē

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(

64 bits

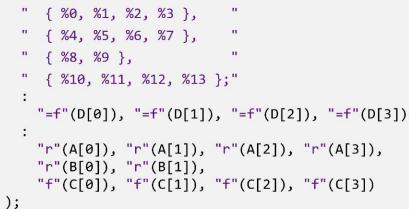
64 bits

r₂ r₃

r₁

ro

"mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "



S8 * S8 + S32

16-b	vy-8	8-b	y-3	32	32	pits	b ₀ b ₁ b ₂ b ₃	т0 т1 т2 т3	77	т9 т10 т11	T13 T14 T15	т17 т18 т19	T21 T22 T23		T29 T30 T31	
. 3	2 bits							т1 т2		т9 т10				T25 T26	_	
b ₀ b	1 b ₂	b ₃						тз	т7					т27		•
то	T1	T2	T3	Т0	T1	T2	Т3	٦	0	Т	1	т	2	T:	3	
Т4	Т5	Т6	T7	T4	T5	Т6	Т7		Т4	Т	5	т	6	т;	7	_
Т8	Т9	T10	T11	Т8	Т9	T10	T11		т8	т	9	tΤ	0	Т1	1	
T12	T13	T14	T15	T12	T13	T14	T15	1	12	T	13	Т1	4	T1	5	
T16	T17	T18	T19	T16	T17	T18	T19	٦	16	T	17	TI	8	Τ1	9	
T20	T21	T22	T23	T20	T21	T22	T23	1	20	T:	21	T2	22	Т2	3	
T24	T25	T26	T27	T24	T25	T26	T27	1	24	T:	25	T2	26	T2	7	
T28	T29	T30	T31	T28	T29	T30	T31	1	28	T.	29	T3	80	Т3	1	+
то	T1	T2	Т3	Т0	T1	T2	Т3	٦	0	Т	1	т	2	T3	3	
Т4	T5	Т6	T7	T4	Т5	Т6	T7		Т4	т	5	т	6	Т	7	
Т8	Т9	T10	T11	Т8	Т9	T10	T11		т8	Т	9	TI	.0	Т1	1	
T12	T13	T14	T15	T12	T13	T14	T15	٦	12	T	13	TI	.4	Τ1	5	
T16	T17	T18	T19	T16	T17	T18	T19	1	16	T	17	TI	.8	T1	9	
T20	T21	T22	T23	T20	T21	T22	T23	1	20	T	21	T2	22	Т2	3	
T24	T25	T26	T27	T24	T25	T26	T27	٦	24	T:	25	T2	26	Т2	7	
T28	T29	T30	T31	T28	T29	T30	T31	٦	28	T.	29	T3	80	Т3	1	

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(

64 bits

64 bits

r₂

r₃

 r_1

ro

"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])
);
```

HALF-PRECISION : F16 * F16 + F16

	16	-b	y-8	3-b	y-1	6 32 I	oits	h ₀		T9 T13 T10 T14 T11 T15	T17 T21 T18 T22 T19 T23			
	0.00	2 bits							Т1 T5 T2 T6	T9 T13		т25 т29 т26 т30	 	
	h _o	r	1 1						тз т7	т11 т15	т19 т23	T27 T31	32 bits	
È	TO	T1	T2	Т3	TO	T1	T2	тз	ТО	T1	T2	ТЗ	h ₀ h ₁	
	T4	T5	T6	T7	T4	T5	T6	T7	T4	T5	T6	T7	·	
	Т8	Т9	T10	T11	T8	T9	T10	T11	тв	Т9	T10	T11	C[0]	i
	T12	T13	T14	T15	T12	T13	T14	T15	T12	T13	T14	T15		-
	T16	T17	T18	T19	T16	T17	T18	T19	T16	T17	T18	T19		-
	T20	T21	T22	T23	T20	T21	T22	T23	T20	T21	T22	T23		1
	T24	T25	T26	T27	T24	T25	T26	T27	T24	T25	T26	T27	32 bits	
	T28	T29	T30	T31	T28	T29	T30	T31	T28	T29	T30	T31		
Г	то	T1	T2	Т3	TO	T1	T2	Т3	TO	T1	T2	Т3	h ₂ h ₃	
	T4	T5	Т6	T7	T4	T5	Т6	T7	T4	T5	Т6	T7	C[1]	
	Т8	Т9	T10	T11	Т8	Т9	T10	T11	Т8	Т9	T10	T11	-1.1	ł
	T12	T13	T14	T15	T12	T13	T14	T15	T12	T13	T14	T15		F
	T16	T17	T18	T19	T16	T17	T18	T19	T16	T17	T18	T19		
	T20	T21	T22	T23	T20	T21	T22	T23	T20	T21	T22	T23		
	T24	T25	T26	T27	T24	T25	T26	T27	T24	T25	T26	T27		
	T28	T29	T30	T31	T28	T29	T30	T31	T28	T29	T30	T31		

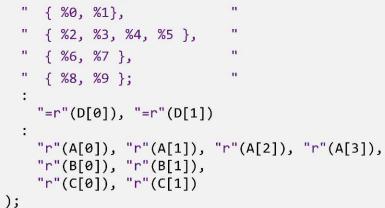
mma.sync.aligned (via inline PTX)

<pre>uint32_t D[2];</pre>	//	two	registers	needed	(VS.	four)
<pre>uint32_t const A[4];</pre>						
<pre>uint32_t const B[2];</pre>						
<pre>uint32_t const C[2];</pre>	//	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 "



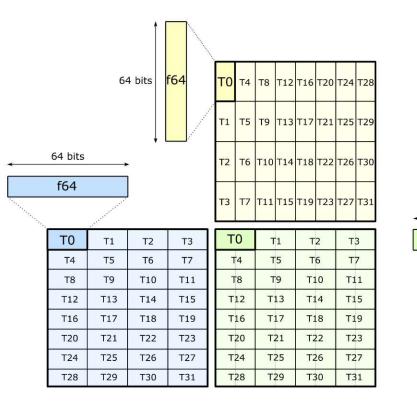
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-instructions-mma-and-friends

DOUBLE-PRECISION: F64 * F64 + F64 mma.sync.aligned **8-by-8-by-4**

128 bits

f64₀

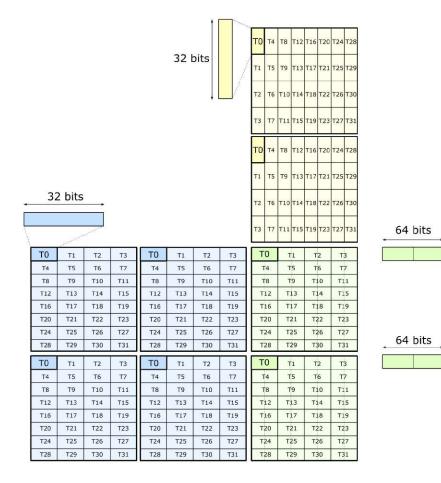
f64₁



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

// Example targets 8-by-8-by-4 Tensor Core operation

CUTLASS: wraps PTX in template *m*-by-*n*-by-*k*



cutlass::arch::Mma

```
/// Matrix multiply-add operation
template <</pre>
```

```
/// 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,
```

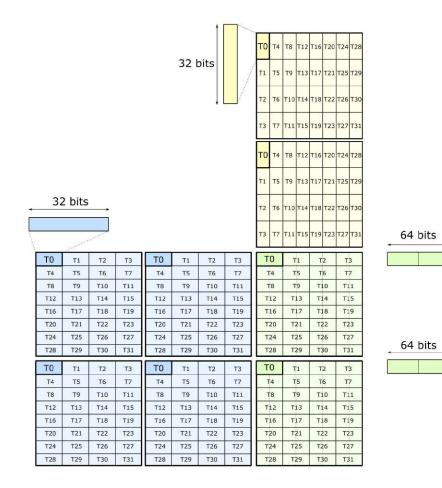
```
/// Inner product operator
typename Operator
```

```
>
```

struct Mma;

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

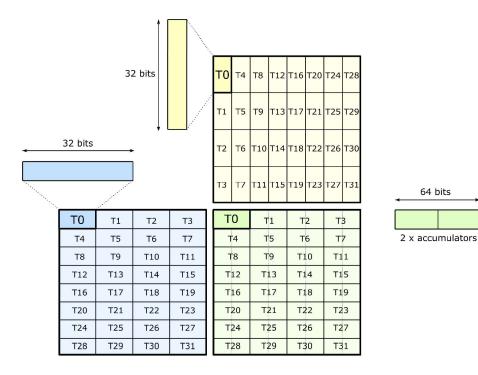
HELLO WORLD: TENSOR CORES

Map each thread to coordinates of the matrix operation

Load inputs from memory

Perform the matrix operation

Store the result to memory



64 bits

}

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,
  н
       %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

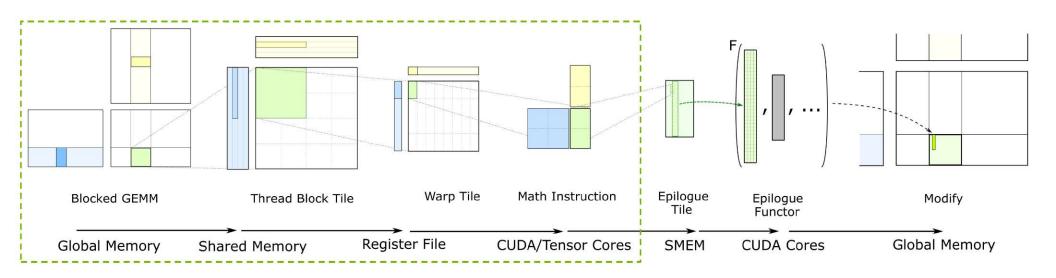
// Compute the coordinates for the accumulator matrices int c_row = threadIdx.x / 4; int c_col = 2 * (threadIdx.x % 4);

```
// 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

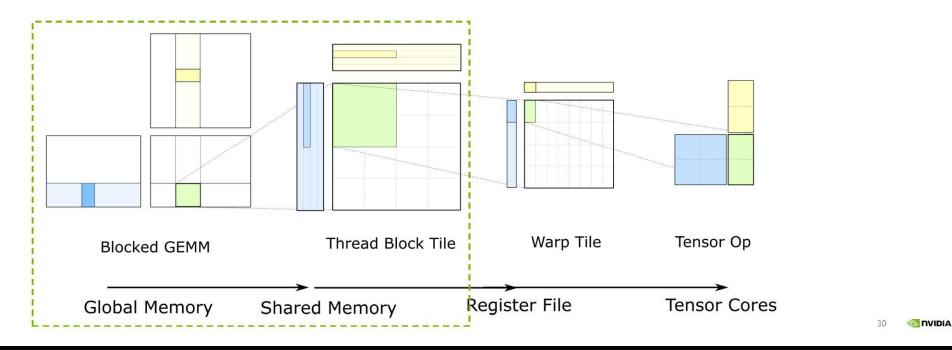
See CUTLASS GTC 2018 talk for more details about this model.

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



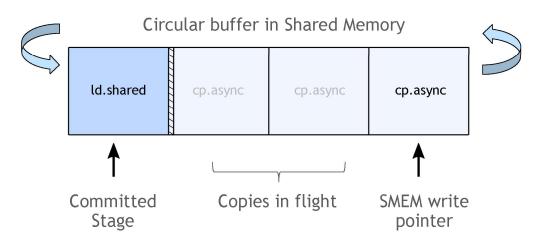
ASYNCHRONOUS COPY: EFFICIENT PIPELINES

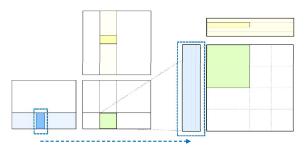
New NVIDIA Ampere Architecture feature: cp.async

- 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: $L2 \rightarrow L1 \rightarrow RF \rightarrow SMEM$ becomes $L2 \rightarrow SMEM$
- 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

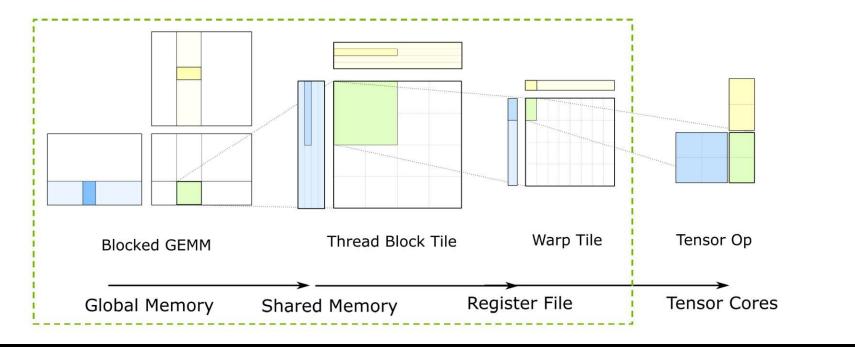




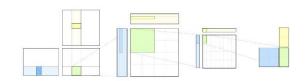
FEEDING THE DATA PATH

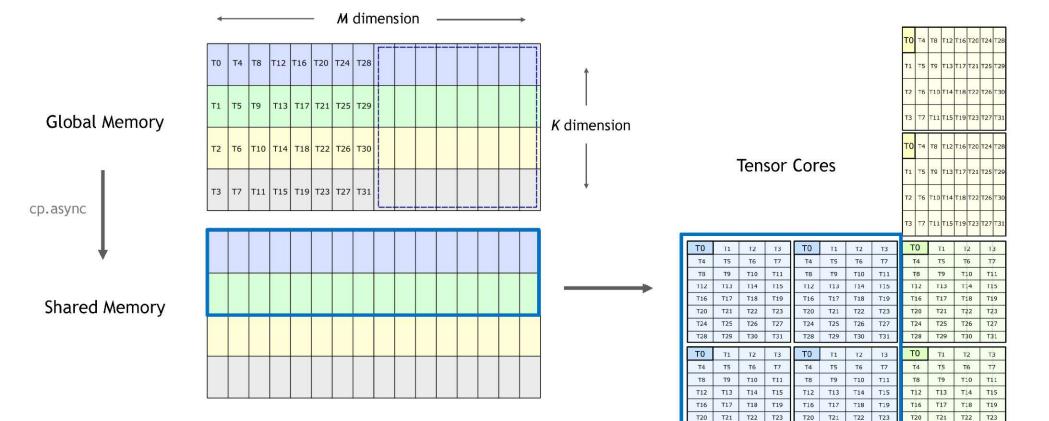
Move data from Global Memory to Tensor Cores as efficiently as possible

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



GLOBAL MEMORY TO TENSOR CORES





T24 T25 T26 T27

T28 T29 T30 T31

T24 T25 T26 T27

T28 T29 T30 T31

T24 T25 T26 T27

T28

T29 T30 T31

LDMATRIX: FETCH TENSOR CORE OPERANDS

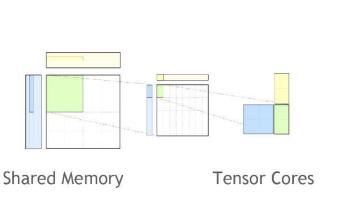
PTX instruction to load a matrix from Shared Memory

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



Core o	perations	5										11 1	2 19 113	117 121	1 T25 T29
	Memory nters						Memory nters						76 T10 T14		
то	\longrightarrow	TO	T1	T2	Т3	T16		Т0	T1	T2	Т3	то	T1	Т2	тз
T1	\rightarrow	Т4	T5	Т6	T7	T17	\longrightarrow	T4	T5	Т6	T7	T4	T5	Т6	Т7
T2	\rightarrow	Т8	Т9	T10	T11	T18	\longrightarrow	T8	Т9	T10	T11	т8	Т9	T10	T11
Т3	\longrightarrow	T12	T13	T14	T15	T19	\longrightarrow	T12	T13	T14	T15	T12	. T13	T14	T15
T4	\longrightarrow	T16	T17	T18	T19	T20	\longrightarrow	T16	T17	T18	T19	т16	5 T17	T18	T19
Т5	\rightarrow	T20	T21	T22	T23	T21	\longrightarrow	T20	T21	T22	T23	T20	T21	T22	T23
Т6	\longrightarrow	T24	T25	T26	T27	T22	\longrightarrow	T24	T25	T26	T27	T24	T25	T26	T27
Τ7	\rightarrow	T28	T29	T30	T31	T23	\longrightarrow	T28	T29	T30	T31	T28	т29	T30	T31
Т8	\longrightarrow	т0	T1	T2	Т3	Т24	\longrightarrow	Т0	T1	T2	T3	то	T1	T2	Т3
Т9	>	T4	T5	Т6	T7	T25	\rightarrow	T4	T5	T6	T7	T4	Т5	Т6	Т7
T10	\longrightarrow	Т8	Т9	T10	T11	T26	\longrightarrow	Т8	Т9	T10	T11	Т8	Т9	T10	T11
T11	\longrightarrow	T12	T13	T14	T15	T27	\longrightarrow	T12	T13	T14	T15	T12	T13	T14	T15
T12	\longrightarrow	T16	T17	T18	T19	T28	\longrightarrow	T16	T17	T18	T19	T16	T17	T18	T19
T13	\longrightarrow	T20	T21	T22	T23	Т29	\longrightarrow	T20	T21	T22	T23	T20	T21	T22	T23
T14	\rightarrow	T24	T25	T26	T27	Т30	\longrightarrow	T24	T25	T26	T27	T24	T25	T26	T27
T15	\rightarrow	T28	T29	T30	T31	T31	>	T28	T29	T30	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

TO T12 T17 T21 T25

LDMATRIX: PTX INSTRUCTION

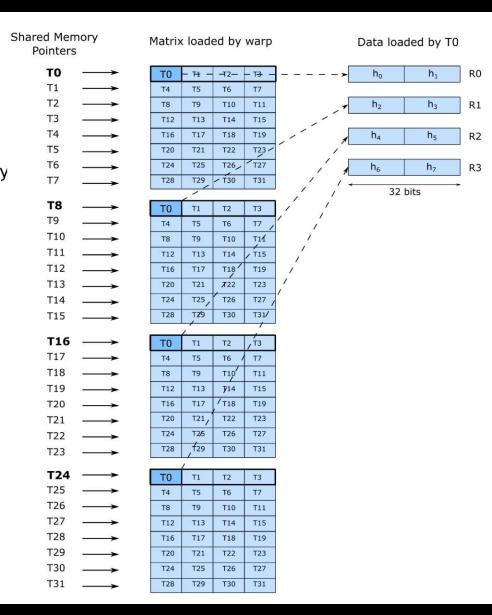
PTX instruction to load a matrix from SMEM

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

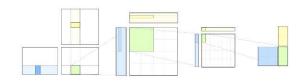
```
// Inline PTX assembly for ldmatrix
```

```
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



Global Memory	T0 T4 T8 T12 T16 T20 T24 T28 Image: Comparison of the temperature T1 T5 T9 T13 T17 T21 T25 T29 Image: Comparison of temperature Image: Comparison of temperature T2 T6 T10 T14 T18 T22 T26 T30 Image: Comparison of temperature Image: Comparison of temperature		Tensor Cores	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
cp.async	T3 T7 T11 T15 T19 T23 T27 T31	Shared Memory	Shared Memory	T1 T5 T9 T13 T17 T21 T25 T29 T2 T6 T10 T14 T18 T22 T26 T30
Shared Memory	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	Pointers T0 \longrightarrow T0 T1 T1 \longrightarrow T4 T5 T2 \longrightarrow T8 T9	Pointers $\begin{array}{c ccccccccccccccccccccccccccccccccccc$	T3 T7 T11 T15 T19 T23 T27 T31 T0 T1 T2 T3 T4 T5 T6 T7
		$\begin{array}{cccccccccccccccccccccccccccccccccccc$	T14 T15 T19 T12 T13 T14 T15 T18 T19 T20 T15 T17 T18 T18 T22 T23 T21 T20 T20 T21 T22 T22 T23	5 T12 T13 T14 T15 7 T16 T17 T18 T19 8 T20 T21 T22 T23 72 T24 T25 T26 T27
		$\begin{array}{cccccccccccccccccccccccccccccccccccc$	$\begin{array}{c ccccccccccccccccccccccccccccccccccc$	

ldmatrix

T12

T13

T14

T15 ____

T16 T17 T18 T19

T20 T21

T24 T25

T28

T29

T28

T29

Т30

T31

T22 T23

T26 T27

T30 T31

T16 T17 T18 T19

T20 T21 T22 T23

T24 T25 T26 T27

T28 T29

->

T16

T20 T21 T22 T23

⊤24

T28

T30 T31

T17 T18 T19

T26 T27

T31

T29 T30

T25

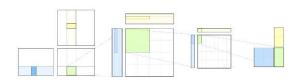
NVIDIA AMPERE ARCHITECTURE - SHARED MEMORY BANK TIMING

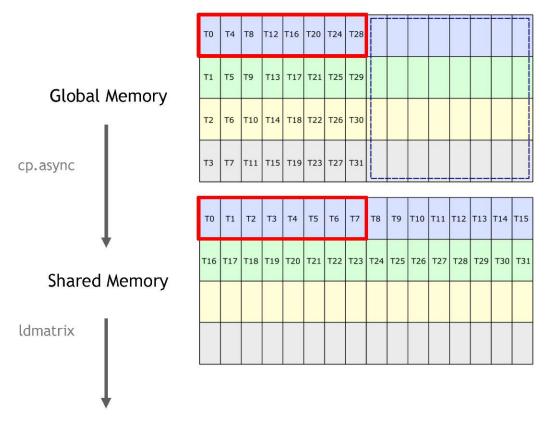
Bank conflicts between threads in the same p	Phase 0: T0 T7						
4B words are accessed in 1 phase		Phase 1: T8 T15					
8B words are accessed in 2 phases:		Phase 2: T16 T23					
 Process addresses of the first 16 threads in a warp 	Phase 3: T24 T31						
 Process addresses of the second 16 threads in a w 							
16B words are accessed in 4 phases:	ss 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

GLOBAL MEMORY TO TENSOR CORES

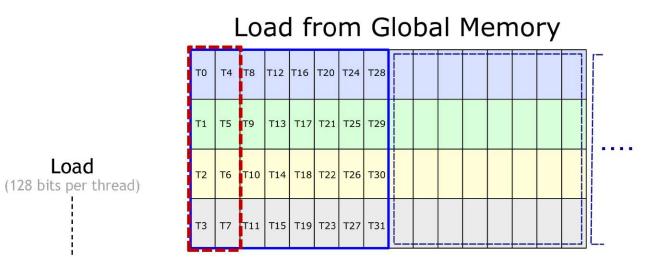


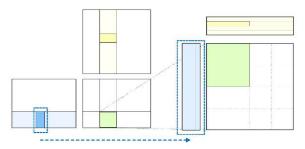


Bank conflict on either store or load from Shared Memory

Registers

GLOBAL TO SHARED MEMORY Load from Global Memory TO Τ4 T8 T12 T16 T20 T24 T28 Т5 T13 T17 T21 T25 T29 **T9** T1 Load Т6 T10 T14 T18 T22 T26 T30 T2 (128 bits per thread) T11 T15 T19 T23 T27 T31 Т3 T7 Permuted Shared Memory layout XOR function maps thread index to Shared Memory location Store to Shared Memory Τ1 T2 Т3 Τ4 Τ5 Τ6 Τ7 T0 Т9 Т8 T11 T15 T10 T13 T12 T14 Store T18 T16 T17 T22 T23 T20 T19 T21 (128 bits per thread) T27 T26 T25 T24 T30 T29 T28 T31





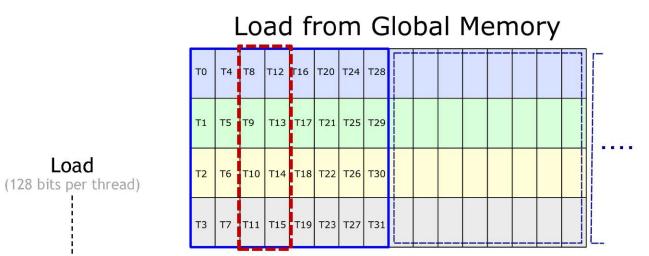
Store to Shared Memory

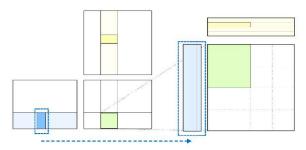
т0	T1	T2	Т3	T4	Т5	Т6	T7
Т9	Т8	T11	T10	T13	T12	T15	T14
T18	T19	T16	T17	T22	T23	T20	T21
T27	T26	T25	T24	T31	Т30	T29	T28
							7

Phase	0:	тө	••	Т7
Phase	1:	Т8	• •	T15
Phase	2:	T16	• •	T23
Phase	3:	T24		T31

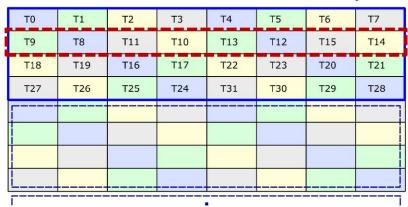


Load





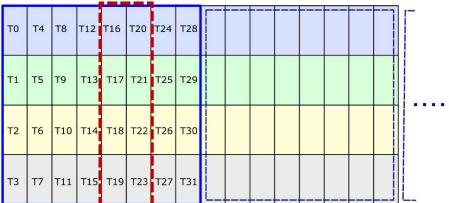
Store to Shared Memory

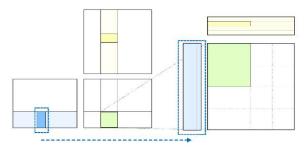


Phase	0:	ТØ		Τ7
Phase	1:	Т8	•••	T15
Phase	2:	T16		T23
Phase	3:	T24		T31



Load from Global Memory





Store to Shared Memory

т0	T1	T2	Т3	T4	T5	Т6	Т7
Т9	Т8	T11	T10	T13	T12	T15	T14
T18	3 T19	T16	T17	Т22	T23	Т20	T21
T27	7 T26	T25	T24	T31	Т30	T29	T28
[7
· · · · · · · · · · · · · · · · · · ·							

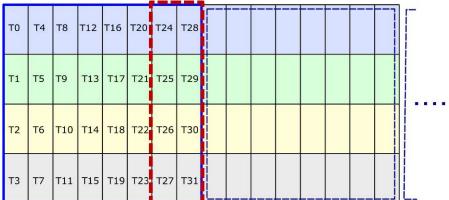
. т	15
. т	23
. т	31
	.т .т .т

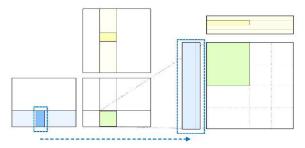


Load

(128 bits per thread)

Load from Global Memory





Store to Shared Memory

Т0	T1	T2	Т3	T4	T5	Т6	Τ7	
Т9	Т8	T11	T10	T13	T12	T15	T14	
T18	T19	T16	T17	T22	T23	T20	T21	
T27	T26	T25	T24	T31	Т30	T29	T28	

Phase	3:	T24	••	T31
Phase	2:	T16	• •	T23
Phase	1:	Т8	• •	T15
Phase	0:	TØ		Τ7



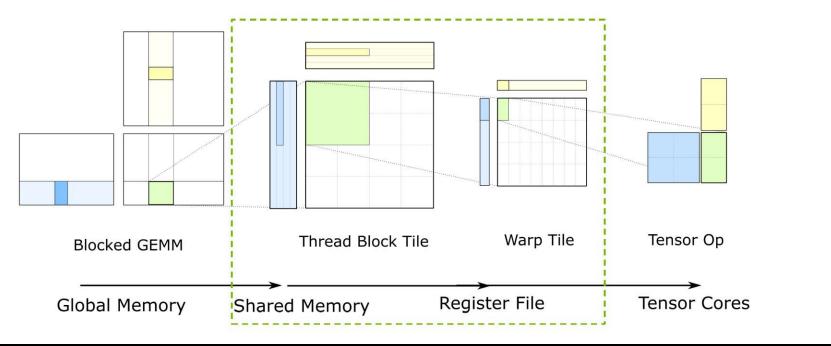
Load

(128 bits per thread)

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



Logical view of threadblock tile

то	т1	т2	тз	Т4	Т5	Т6	т7	Т8	Т9	т10	T11	Т12	Т13	T14	T15
Т16	T17	T18	T19	T20	T21	т22	т23	T24	т25	Т26	т27	т28	т29	т30	Т31

Load Matrix from Shared Memory

то	T16			T1	T17		
T18	T2			T19	Т3		
		T4	T20			T5	T21
		T22	Т6			T23	T7
Т8	T24			Т9	T25		n
T26	T10			T27	T11		
		T12	T28			T13	T29
		Т30	T14			T31	T15
٢							

т0	\rightarrow	то
T1	\rightarrow	T4
Т2	\longrightarrow	тв
Т3	\rightarrow	T12
Т4	\longrightarrow	T16
T5	→	T20
Т6	\longrightarrow	T24
Т7	\longrightarrow	T28
Т8	\rightarrow	то
Т9	\longrightarrow	T4
T10	\longrightarrow	Т8
T11		T12
T12	\rightarrow	T16
T13	>	T20
T14		T24
1 1 4		
T15	\rightarrow	T28

Shared Memory

Pointers

то	T1	T2	Т3
T4	T5	Т6	T7
Т8	Т9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31
		-	
то	T1	T2	T3
T4	T5	Т6	T7
T8	Т9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
120	121	166	
T24	T25	T26	T27

Shared	Memory					T2	T6	T10	T14	T18	T22	T26	T30
Poi	nters					тз	77	т11	т15	т19	т23	т27	Т31
T16	\longrightarrow	то	T1	T2	T3		то	Т	1	Т	2	T.	3
T17	\longrightarrow	T4	Т5	Т6	T7		т4	Т	5	т	6	Т	7
T18	\longrightarrow	Т8	Т9	T10	T11		т8	т	9	т	10	Т1	1
T19	\longrightarrow	T12	T13	T14	T15		F12	т	13	TI	.4	Τ1	5
T20	\longrightarrow	T16	T17	T18	T19		Г16	т	17	TI	18	T1	9
T21	\longrightarrow	T20	T21	T22	T23		F20	T,	21	Т2	22	T2	3
T22	\longrightarrow	T24	T25	T26	T27		r24	T	25	Τ2	26	T2	7
T23	\longrightarrow	T28	T29	T30	T31		F28	T	29	Т3	30	Т3	1
T24	\longrightarrow	т0	T1	T2	Т3		то	Т	1	T	2	T.	3
T25	\longrightarrow	T4	T5	Т6	T7		т4	т	5	т	6	т	7
T26	\rightarrow	Т8	Т9	T10	T11		т8	т	9	TI	0	Т1	1
T27	\rightarrow	T12	T13	T14	T15		Г12	Т	13	т	4	T1	5
T28	\rightarrow	T16	T17	T18	T19	1	Г16	Т	17	TI	8	Τ1	9
T29	\rightarrow	T20	T21	T22	T23	1	F20	Т	21	Т2	22	T2	3
T30	\longrightarrow	T24	T25	T26	T27		r24	T.	25	T2	26	T2	7
T31	\longrightarrow	T28	T29	T30	T31		728	T	29	Т3	30	Т3	1

T0 T4 T8 T12 T16 T20 T24 T2

T1 T5 T9 T13 T17 T21 T25 T2

T2 T6 T10 T14 T18 T22 T26

T3 T7 T11 T15 T19 T23 T27 T31

T0 T4 T8 T12 T16 T20 T24 T28

T1 T5 T9 T13 T17 T21 T25 T29

Logical view of threadblock tile

то	Т1	Т2	Т3	T4	Т5	Т6	Τ7	т8	т9	т10	Т11	T12	Т13	T14	Т15
Т16	T17	T18	Т19	т20	T21	T22	т23	Т24	T25	T26	Т27	т28	т29	т30	Т31

Load Matrix from Shared Memory

то	T16			T1	T17		
T18	T2			T19	Т3		
		T4	T20			T5	T21
		T22	Т6			T23	T7
Т8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28	_		T13	T29
		Т30	T14			T31	T15
Г							
				9			

	т0	\rightarrow	Т
	T1	\rightarrow	т
	T2	\longrightarrow	т
	Т3	\longrightarrow	т
	T4	\longrightarrow	Т
	Т5	>	т
	Т6	>	т
	Т7	\rightarrow	Т
ĺ	T8	\rightarrow	Т
		\rightarrow	
	Т8	\rightarrow	Т
	Т8 Т9		Т
	T8 T9 T10		T T T
	T8 T9 T10 T11		T T T
	T8 T9 T10 T11 T12		T T T T

Shared Memory

Pointers

то	T1	T2	Т3	
T4	T5	Т6	T7	
Т8	Т9	T10	T11	
T12	T13	T14	T15	
T16	T17	T18	T19	
T20	T21	T22	T23	
T24	T25	T26	T27	
T28	T29	T30	T31	
			500	
то	T1	T2	T3	
то т4	T1 T5	T2 T6		
			ТЗ	
T4	T5	Т6	T3 T7	
T4 T8	T5 T9	T6 T10	T3 T7 T11	
T4 T8 T12	T5 T9 T13	T6 T10 T14	T3 T7 T11 T15	
T4 T8 T12 T16	T5 T9 T13 T17	T6 T10 T14 T18	T3 T7 T11 T15 T19	

Shared Memory

Pointers

T16 T17 T18 T19 T20 T21 T22 T23 T24 T25 T26 T27 T28 T29 T30 T31

				TO	T4	T8	T12	T16	T20	T24	120
				т1	т5	т9	т13	т17	т21	т25	т29
				т2	т6	т10	Т14	T18	т22	т26	т30
				тз	т7	т11	т15	т19	т23	т27	т31
				то	т4	тв	т12	т16	т20	т24	т28
				т1	т5	т9	т13	т17	т21	т25	т29
				т2	т6	т10	T14	T18	т22	т26	т30
				тз	т7	т11	T15	т19	т23	т27	Т31
то	T1	T2	T3		го	Т	1	т	2	Т	3
T0 T4	T1 T5	T2 T6	T3 T7		го Г4	-	1 5	т т		T. T	-
						т	-		6		7
T4	T5	Т6	T7	1	F4	T T	5	т	6	т	7
T4 T8	T5 T9	Т6 Т10	T7 T11	ר ד ד	r4 r8	T T T	5 9	T T:	6 10 14	Т Т1	7
T4 T8 T12	T5 T9 T13	T6 T10 T14	T7 T11 T15	T T T	r4 r8 12	T T T	5 9 13	T T: T:	6 10 14 18	T T1 T1	7 11 15
T4 T8 T12 T16	T5 T9 T13 T17	T6 T10 T14 T18	T7 T11 T15 T19	T T T T	r4 r8 12 16	T T T T	5 9 13 17	т тт тт тт	6 10 14 18 22	T T1 T1 T1	7 11 15 19 23
T4 T8 T12 T16 T20	T5 T9 T13 T17 T21	T6 T10 T14 T18 T22	T7 T11 T15 T19 T23	ן ד ד ד ד	r4 r8 12 16 20	т т т. т. т.	75 79 13 17 21	T T T T T T	6 10 14 18 22 26	Т Т1 Т1 Т1 Т2	7 11 15 19 23 27
T4 T8 T12 T16 T20 T24	T5 T9 T13 T17 T21 T25	T6 T10 T14 T18 T22 T26	T7 T11 T15 T19 T23 T27	T T T T T	r4 r8 12 16 20 24	т т т. т. т. т.	5 9 13 17 21 25	т т: т: т: т: т:	6 10 14 18 22 26 30	T T1 T1 T1 T2 T2	7 11 15 19 23 27 31
T4 T8 T12 T16 T20 T24 T28	T5 T9 T13 T17 T21 T25 T29	T6 T10 T14 T18 T22 T26 T30	T7 T11 T15 T19 T23 T27 T31		r4 r8 12 16 20 24 28	T T T T T T T	5 9 13 17 21 25 29	т ті ті ті ті ті	6 10 14 18 22 26 30 2	T T1 T1 T1 T2 T2 T3	7 11 15 19 23 27 81
T4 T8 T12 T16 T20 T24 T28 T0	T5 T9 T13 T17 T21 T25 T29 T1	T6 T10 T14 T18 T22 T26 T30 T2	T7 T11 T15 T19 T23 T27 T31 T3		r4 r8 12 16 20 24 28 r0	т т т т т т т т	-5 -9 13 17 21 25 29 -1	T T1 T1 T2 T2 T2 T2 T2 T2 T2	6 10 14 18 22 26 80 2 2 6	T T1 T1 T1 T2 T2 T3 T3	7 11 5 9 23 27 81 3 7
T4 T8 T12 T16 T20 T24 T28 T0 T4	T5 T9 T13 T17 T21 T25 T29 T1 T5	T6 T10 T14 T18 T22 T26 T30 T2 T2 T6	T7 T11 T15 T19 T23 T27 T31 T3 T7		r4 r8 12 16 20 24 28 r0 r4	т т т. т. т. т. т т т т т	5 9 13 17 21 25 29 71 5	T T T T T T T T T T T	6 10 14 18 22 26 80 2 2 6 10	T T1 T1 T1 T2 T2 T2 T3 T. T.	7 11 5 9 23 27 81 3 7 1
T4 T8 T12 T16 T20 T24 T28 T0 T4 T8	T5 T9 T13 T17 T21 T25 T29 T1 T5 T9	T6 T10 T14 T18 T22 T26 T30 T2 T6 T10	T7 T11 T15 T19 T23 T27 T31 T3 T7 T11		r4 r8 12 16 20 24 28 r0 r4 r8	т т т. т. т. т. т. т.	5 9 113 17 21 25 29 1 1 5 5 9	T T T T T T T T T T T	6 10 14 18 22 26 30 2 2 6 10 14	T T1 T1 T1 T2 T2 T3 T3 T1 T1 T1	7 11 5 9 23 27 81 3 7 11 5
T4 T8 T12 T16 T20 T24 T28 T0 T4 T4 T8 T12	T5 T9 T13 T17 T21 T25 T29 T1 T5 T9 T13	T6 T10 T14 T18 T22 T26 T30 T2 T6 T10	T7 T11 T15 T19 T23 T27 T31 T3 T3 T7 T11 T15		r4 r8 12 16 20 24 28 r0 r4 r8 12	т т т т, т, т, т, т, т, т,	5 9 13 17 21 25 29 71 5 9 13	T T1 T1 T2 T2 T2 T2 T2 T2 T1 T1 T1	6 10 14 18 22 26 80 2 2 6 10 14 14	T T1 T1 T1 T2 T2 T2 T3 T. T T1 T1	7 11 5 9 23 27 31 3 7 11 5 9 9
T4 T8 T12 T16 T20 T24 T28 T0 T4 T8 T12 T16	T5 T9 T13 T17 T21 T25 T29 T1 T5 T9 T13 T17	T6 T10 T14 T18 T22 T26 T30 T2 T6 T10 T11 T11 T12 T6 T10 T14	T7 T11 T15 T19 T23 T27 T31 T3 T7 T11 T15 T19		r4 r8 12 16 20 24 28 r0 r4 r8 12 16	т т т. т. т. т. т. т. т. т. т.	5 9 13 17 21 25 29 11 5 9 13 17	T T T T T T T T T T T T T	6 10 14 18 22 26 30 22 6 10 14 18 22 22	T T1 T1 T1 T2 T2 T3 T. T1 T1 T1 T1	7 11 5 .9 23 27 31 3 7 11 5 .9 9 23

Logical view of threadblock tile

то	T1	Т2	Т3	T4	Т5	Т6	т7	Т8	Т9	т10	T11	T12	т13	T14	T15
т16	T17	T18	т19	т20	Т21	T22	т23	T24	т25	т26	т27	т28	т29	т30	Т31

Load Matrix from Shared Memory

то	T16			T1	T17		
T18	T2			T19	Т3		
		T4	T20			T5	T21
		T22	Т6			T23	T7
Т8	T24			Т9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		Т30	T14			T31	T15
F							1

то	\rightarrow	Г
T1	>	F
Т2	\longrightarrow	
Т3		F
T4	>	F
Т5	\rightarrow	F
Т6	\longrightarrow	F
Т7	>	
Т8	\rightarrow	Ē
Т9	\longrightarrow	F
T10	\longrightarrow	F
T11	\longrightarrow	F
T12	\longrightarrow	F
T13		F
T14		

T15

Shared Memory

Pointers

				-
то	T1	T2	Т3	
T4	T5	T6	T7	
Т8	Т9	T10	T11	
T12	T13	T14	T15	
T16	T17	T18	T19	
T20	T21	T22	T23	
T24	T25	T26	T27	
T28	T29	T30	T31	
TO	T1	T2	T3	
T4	T5	T6	T7	
T8	T9	T10	T11	
T12	T13	T14	T15	
T16	T17	T18	T19	
1 million (197	T21	T22	T23	
T20	121	122		
T20 T24	T25	T26	T27	

					то	T4	тв	T12	т16	т20	т24	т28
					т1	т5	т9	т13	т17	т21	т25	т29
Shared Memory					т2	тб	т10	T14	т18	т22	т26	т30
Pointers					тз	77	т11	т15	т19	т23	т27	Т31
T16>	то	T1	T2	T3		то	Т	1	т	2	T	3
T17	T4	Т5	Т6	T7		т4	т	5	т	6	т	7
T18>	Т8	Т9	T10	T11		T8	т	9	T	10	Т1	1
T19	T12	T13	T14	T15	1	12	T	13	TI	.4	Τ1	5
T20	T16	T17	T18	T19	Т	16	T	17	T	18	T1	9
T21 →	T20	T21	T22	T23	T	20	T:	21	т	22	T2	3
T22>	T24	T25	T26	T27	1	24	Т	25	Т	26	T2	7
T23>	T28	T29	T30	T31	٦	28	T	29	T3	30	T3	1
T24>	то	T1	T2	Т3		то	Т	1	т	2	T	3
T25	T4	T5	Т6	T7	1	т4	т	5	т	6	т	7
T26>	Т8	Т9	T10	T11		т8	т	9	т	0	Т1	1
T27>	T12	T13	T14	T15	Т	12	T	13	т	4	T1	5
T28	T16	T17	T18	T19	Т	16	т	17	т	8	T1	9
T29	T20	T21	T22	T23	Т	20	т	21	Т	22	T2	3
тзо ——	T24	T25	T26	T27	Т	24	т	25	Т	26	T2	7
T31	T28	T29	T30	T31	Т	28	Т	29	T3	30	Т3	1
	-				-							_

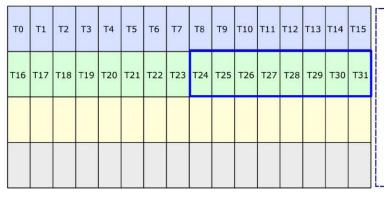
T0 T4 T8 T12 T16 T20 T24 T28

T1 T5 T9 T13 T17 T21 T25 T2

T2 T6 T10 T14 T18 T22 T26

T3 T7 T11 T15 T19 T23 T27 T31

Logical view of threadblock tile



Load Matrix from Shared Memory

то	T16			Τ1	T17		
T18	T2			T19	Т3		
		T4	T20			T5	T21
		T22	Т6			T23	T7
Т8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		Т30	T14			T31	T15
Γ							1

то	\rightarrow	Т
T1	>	т
T2	\longrightarrow	т
Т3	>	т
T4	>	т
T5	>	т
Т6	\longrightarrow	т
Т7	>	Т
Т8	\rightarrow	Т
Т9	\longrightarrow	т
T10	\longrightarrow	т
T11	→	т
T12	>	т
T13	\longrightarrow	т
T14	\longrightarrow	т
		-

T15

Shared Memory

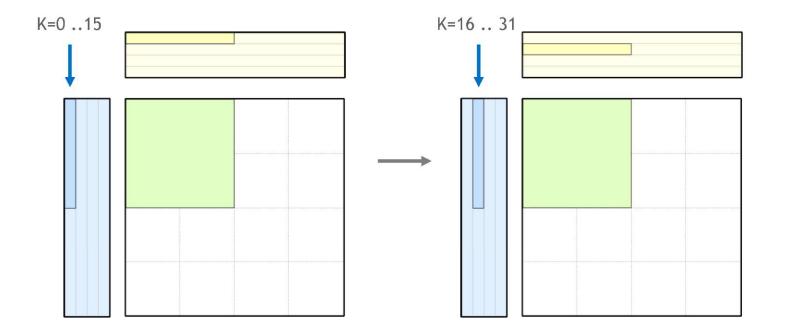
Pointers

то	T1	T2	Т3	T16
T4	T5	T6	T7	T17
T8	Т9	T10	T11	T18
T12	T13	T14	T15	T19
T16	T17	T18	T19	Т20
T20	T21	T22	T23	T21
T24	T25	T26	T27	Т22
T28	T29	T30	T31	1
	100	150	151	T23
				Concession of the local division of the loca
то	T1	T2	тз	Т24
T0 T4	T1 T5			Concession of the local division of the loca
	100 NI	T2	ТЗ	Т24
T4	T5	T2 T6	T3 T7	T24 T25
T4 T8	T5 T9	T2 T6 T10	T3 T7 T11	T24 T25 T26
T4 T8 T12	T5 T9 T13	T2 T6 T10 T14	T3 T7 T11 T15	T24 T25 T26 T27
T4 T8 T12 T16	T5 T9 T13 T17	T2 T6 T10 T14 T18	T3 T7 T11 T15 T19	T24 T25 T26 T27 T28

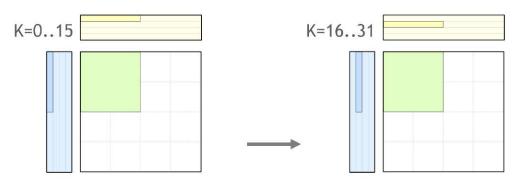
						т1	т5	т9	т13	т17	т21	T25	Т29
						т2	т6	т10	T14	T18	т22	т26	т30
						тз	77	т11	T15	т19	т23	т27	Т31
						то	T4	тв	т12	т16	т20	т24	т28
						т1	т5	т9	T13	T17	т21	т25	т29
Shared	l Memory					т2	т6	т10	т14	т18	т22	т26	т30
Poi	nters					тз	77	т11	т15	т19	т23	т27	Т31
T16	\longrightarrow	TO	T1	T2	T3		то	Т	1	Т	2	T.	3
T17	\rightarrow	T4	T5	Т6	T7	1 12	т4	т	5	т	6	т	7
T18	\rightarrow	Т8	Т9	T10	T11		т8	т	9	T	10	Т1	1
T19	\rightarrow	T12	T13	T14	T15	1	12	т	13	TI	14	T1	5
T20	\longrightarrow	T16	T17	T18	T19	1	16	т	17	T	18	T1	9
T21	\longrightarrow	T20	T21	T22	T23		20	T.	21	т	22	T2	3
T22	\longrightarrow	T24	T25	T26	T27	1	24	T	25	Т	26	T2	7
T23	\longrightarrow	T28	T29	T30	T31	٦	28	T	29	T3	30	Т3	1
T24	>	TO	T1	T2	Т3		то	Т	1	т	2	T	3
T25	\longrightarrow	T4	T5	Т6	T7		T4		5	т	-	т	
T26	\longrightarrow	т8	Т9	T10	T11		т8		9	TI		T1	
T27	\longrightarrow	T12	T13	T14	T15	T	12	Т	13	т	4	T1	5
T28	\longrightarrow	T16	T17	T18	T19	1	16	T	17	T	8	Т1	9
			-	T22	T23		20	Т	21	т	2	T2	3
T29	\longrightarrow	T20	T21	122									- I
Т29 Т30		T20 T24	T21 T25	T26	T27		24		25	Т		T2	

T0 T4 T8 T12 T16 T20 T24 T28

ADVANCING TO NEXT K GROUP



ADVANCING TO NEXT K GROUP



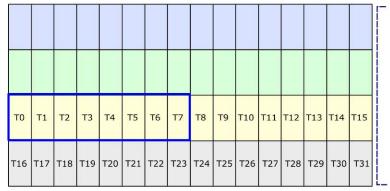
		<u>//</u>					
то	T16			T1	T17		
T18	T2			T19	Т3		
		Т4	T20			Т5	T21
		T22	T6			T23	T7
Т8	T24			Т9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		Т30	T14			T31	T15

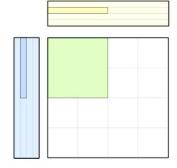
		то	T16			T1	T17
		T18	T2			T19	Т3
T4	T20			Т5	T21		
T22	T6			T23	T7		
		Т8	T24			Т9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
Т30	T14			T31	T15		

smem_ptr = row_idx * 8 + column_idx;

smem_ptr = smem_ptr ^ 2;

Logical view of threadblock tile





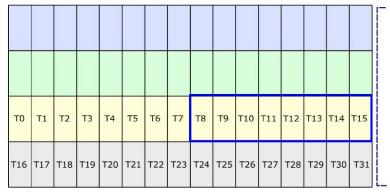


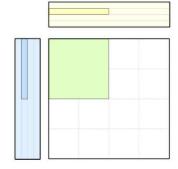
Phase 0

		то	T16			T1	T17
		T18	T2			T19	Т3
T4	T20			Т5	T21		
T22	Т6			T23	T7		
		Т8	T24			Т9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

Phase 1

Logical view of threadblock tile

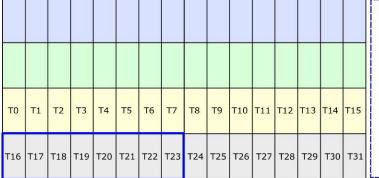


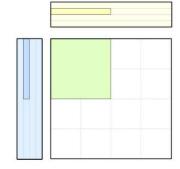




		то	Т16			Т1	T17
		T18	T2			T19	Т3
T4	T20			T5	T21		
T22	T6			T23	T7		
		Т8	T24			Т9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		

Logical view of threadblock tile



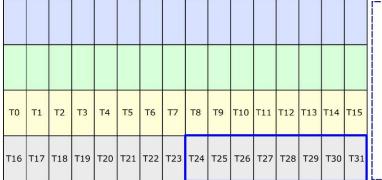


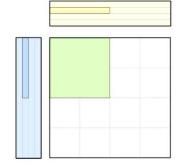


Phase 2

		то	T16			T1	T17
		T18	T2			T19	Т3
T4	T20			Т5	T21		
T22	Т6			T23	T7		
		T8	T24			Т9	T25
		T26	T10			T27	T11
T12	T28			T13	T29		
T30	T14			T31	T15		
			2				

Logical view of threadblock tile





K=16..31

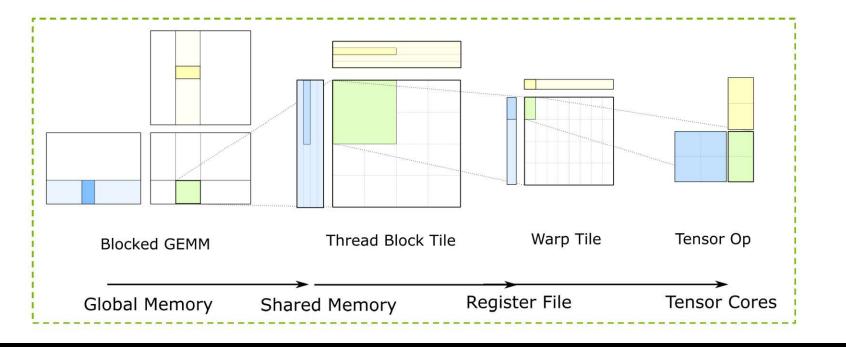
Phase 3

T2 T24	T5 T23	T21 T7	T19 	T3
T24			Т9	T25
T24	T23	77	Т9	T25
T24			Т9	T25
T10			T27	T11
-	T13	T29		
	T31	T15		
<u> </u>			<u> </u>	
		2 10 10 10 10 10 10 10 10 10 10 10 10 10	T31 T15	T31 T15

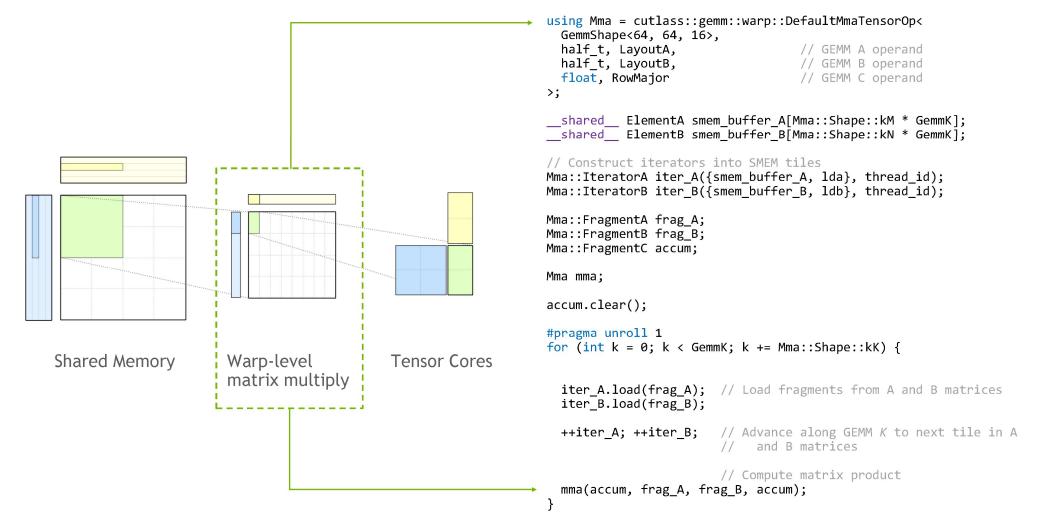
CUTLASS

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

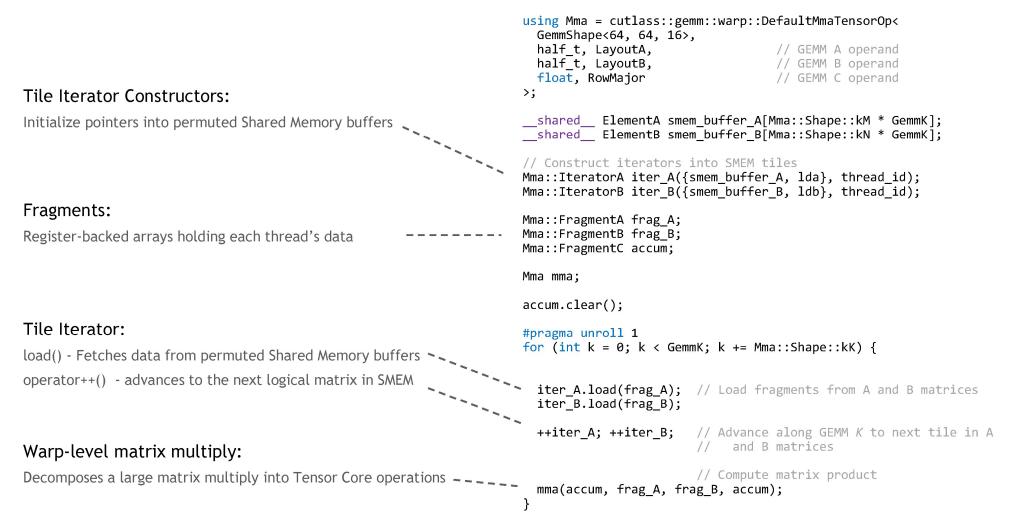
- Latency-tolerant pipeline from Global Memory
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES



CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES



Thank you.

- Md Aamir Raihan, Negar Goli, Tor Aamodt
- Andrew Kerr et al., NVIDIA
- NVIDIA