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

Andrew Kerr, May 21, 2020
ACKNOWLEDGEMENTS

CUTLASS Team
Andrew Kerr, Haicheng Wu, Manish Gupta, Duane Merrill, Pradeep Ramani

Contributors
Mostafa Hagog, Timothy Costa, Alan Kaatz, John Tran, Stephen Jones, Kyrylo Perelygin, Luke Durant, Piotr Majcher, Paul Springer, Markus Hohnerbach

Acknowledgements
Joel McCormack, Julien Demouth, Olivier Giroux, Bryce Lelbach, Cris Cecka
AGENDA

Overview
NVIDIA Ampere Architecture and CUTLASS 2.2

Tensor Cores on NVIDIA Ampere Architecture
Accelerated matrix operations

Efficient data movements for Tensor Cores
Strategies for maximizing performance

CUTLASS on NVIDIA A100
Optimal CUDA C++ templates for Tensor Cores
OVERVIEW
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, libc++
This is a talk for CUDA programmers
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

Mixed Precision Floating Point

- Tensor Core - BF16, F16
- Tensor Core - TF32
- CUDA Core - F32

Double Precision Floating Point

- Tensor Core - F64
- CUDA Core - F64

Mixed Precision Integer

- Tensor Core - INT4
- TensorFlow - INT8
- CUDA Core - INT8

m=3456, n=4096
TENSOR CORES ON NVIDIA AMPERE ARCHITECTURE
WHAT ARE TENSOR CORES?

Matrix operations: \( D = \text{op}(A, B) + C \)
- 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: \( D = \text{op}(A, B) + C \)
- 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

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


* Instructions with equivalent functionality for Turing and Volta differ in shape from the NVIDIA Ampere Architecture in several cases.
Warp-wide Tensor Core operation: 8-by-8-by-128b
\texttt{S8 \times S8 + S32} \\
8-by-8-by-16

\texttt{mmasync.aligned} \\
(via inline PTX)

\begin{verbatim}
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])
);
\end{verbatim}

EXPANDING THE $M$ DIMENSION

Warp-wide Tensor Core operation: 16-by-8-by-128b
```c
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])
);
```

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)

```c
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"(B[0]), "r"(B[1]),
    "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])
);```

S8 \times S8 + S32

16-by-8-by-32

mma.sync.aligned
(via inline PTX)

\begin{verbatim}
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"(B[0]), "r"(B[1]),
    "r"(C[0]), "r"(C[1]), "r"(C[2]), "r"(C[3])
);
\end{verbatim}

HALF-PRECISION: \( F_{16} \times F_{16} + F_{16} \)

16-by-8-by-16

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"(B[0]), "=r"(B[1]),
      ":=r"(C[0]), "=r"(C[1])
);
**DOUBLE-PRECISION: F64 * F64 + F64**

8-by-8-by-4

```c
// 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])
    :
    "1"(A),
    "1"(B),
    "1"(C[0]), ":1"(C[1])
);
```

CUTLASS: wraps PTX in template

\[ m \times n \times k \]

```cpp
/// 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
```cpp
__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);

    ...
}
```

CUTLASS: wraps PTX in template

16-by-8-by-16

[Diagram of 16-by-8-by-16 matrix multiplication]

---

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

```c
__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\(C[cd_idx]\), \"C[cd_idx + 1]\)"
    );
}
```
CUDA example

```c
__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.
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

Circular buffer in Shared Memory
- Committed Stage
- Copies in flight
- SMEM write pointer
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

Global Memory

Shared Memory

Tensor Cores

$M$ dimension

$K$ dimension

$T0$ $T4$ $T8$ $T12$ $T16$ $T20$ $T24$ $T28$

$T1$ $T5$ $T9$ $T13$ $T17$ $T21$ $T25$ $T29$

$T2$ $T6$ $T10$ $T14$ $T18$ $T22$ $T26$ $T30$

$T3$ $T7$ $T11$ $T15$ $T19$ $T23$ $T27$ $T31$

cp.async
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
LDMATRIX: PTX INSTRUCTION

PTX instruction to load a matrix from SMEM

Each thread supplies a pointer to a 128-byte row of data in Shared Memory.

Each 128-byte row is broadcast to groups of four threads (potentially different threads than the one supplying the pointer).

Data matches the arrangement of inputs to Tensor Core operations.

// Inline PTX assembly for ldmatrix

```c
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

cp.async

Shared Memory

ldmatrix

Tensor Cores

Shared Memory Pointers

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
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

128 bit access size

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

GLOBAL MEMORY TO TENSOR CORES

Bank conflict on either store or load from Shared Memory
GLOBAL TO SHARED MEMORY

Load (128 bits per thread)

Permuted Shared Memory layout
XOR function maps thread index to Shared Memory location

Store to Shared Memory

Store (128 bits per thread)
GLOBAL TO SHARED MEMORY

Load from Global Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T4</th>
<th>T8</th>
<th>T12</th>
<th>T16</th>
<th>T20</th>
<th>T24</th>
<th>T28</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T5</td>
<td>T9</td>
<td>T13</td>
<td>T17</td>
<td>T21</td>
<td>T25</td>
<td>T29</td>
</tr>
<tr>
<td>T2</td>
<td>T6</td>
<td>T10</td>
<td>T14</td>
<td>T18</td>
<td>T22</td>
<td>T26</td>
<td>T30</td>
</tr>
<tr>
<td>T3</td>
<td>T7</td>
<td>T11</td>
<td>T15</td>
<td>T19</td>
<td>T23</td>
<td>T27</td>
<td>T31</td>
</tr>
</tbody>
</table>

Load
(128 bits per thread)

Store to Shared Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
<td>T8</td>
<td>T11</td>
<td>T10</td>
<td>T13</td>
<td>T12</td>
<td>T15</td>
<td>T14</td>
</tr>
<tr>
<td>T18</td>
<td>T19</td>
<td>T16</td>
<td>T17</td>
<td>T22</td>
<td>T23</td>
<td>T20</td>
<td>T21</td>
</tr>
<tr>
<td>T27</td>
<td>T26</td>
<td>T25</td>
<td>T24</td>
<td>T31</td>
<td>T30</td>
<td>T29</td>
<td>T28</td>
</tr>
</tbody>
</table>

Store
(128 bits per thread)

Phase 0: T0 .. T7
Phase 1: T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31
GLOBAL TO SHARED MEMORY

Load from Global Memory

Store to Shared Memory

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

Load (128 bits per thread)
Store (128 bits per thread)
# GLOBAL TO SHARED MEMORY

### Load from Global Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T4</th>
<th>T8</th>
<th>T12</th>
<th>T16</th>
<th>T20</th>
<th>T24</th>
<th>T28</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T5</td>
<td>T9</td>
<td>T13</td>
<td>T17</td>
<td>T21</td>
<td>T25</td>
<td>T29</td>
</tr>
<tr>
<td>T2</td>
<td>T6</td>
<td>T10</td>
<td>T14</td>
<td>T18</td>
<td>T22</td>
<td>T26</td>
<td>T30</td>
</tr>
<tr>
<td>T3</td>
<td>T7</td>
<td>T11</td>
<td>T15</td>
<td>T19</td>
<td>T23</td>
<td>T27</td>
<td>T31</td>
</tr>
</tbody>
</table>

Load (128 bits per thread)

### Store to Shared Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
<td>T8</td>
<td>T11</td>
<td>T10</td>
<td>T13</td>
<td>T12</td>
<td>T15</td>
<td>T14</td>
</tr>
<tr>
<td>T18</td>
<td>T19</td>
<td>T16</td>
<td>T17</td>
<td>T22</td>
<td>T23</td>
<td>T20</td>
<td>T21</td>
</tr>
<tr>
<td>T27</td>
<td>T26</td>
<td>T25</td>
<td>T24</td>
<td>T31</td>
<td>T30</td>
<td>T29</td>
<td>T28</td>
</tr>
</tbody>
</table>

Store (128 bits per thread)

### Phases
- **Phase 0**: T0 .. T7
- **Phase 1**: T8 .. T15
- **Phase 2**: T16 .. T23
- **Phase 3**: T24 .. T31
### GLOBAL TO SHARED MEMORY

#### Load from Global Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T4</th>
<th>T8</th>
<th>T12</th>
<th>T16</th>
<th>T20</th>
<th>T24</th>
<th>T28</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>T5</td>
<td>T9</td>
<td>T13</td>
<td>T17</td>
<td>T21</td>
<td>T25</td>
<td>T29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>T6</td>
<td>T10</td>
<td>T14</td>
<td>T18</td>
<td>T22</td>
<td>T26</td>
<td>T30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>T7</td>
<td>T11</td>
<td>T15</td>
<td>T19</td>
<td>T23</td>
<td>T27</td>
<td>T31</td>
</tr>
</tbody>
</table>

#### Store to Shared Memory

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
<td>T8</td>
<td>T11</td>
<td>T10</td>
<td>T13</td>
<td>T12</td>
<td>T15</td>
<td>T14</td>
</tr>
<tr>
<td>T18</td>
<td>T19</td>
<td>T16</td>
<td>T17</td>
<td>T22</td>
<td>T23</td>
<td>T20</td>
<td>T21</td>
</tr>
<tr>
<td>T27</td>
<td>T26</td>
<td>T25</td>
<td>T24</td>
<td>T31</td>
<td>T30</td>
<td>T29</td>
<td>T28</td>
</tr>
</tbody>
</table>

- **Load** (128 bits per thread)
- **Store** (128 bits per thread)

**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

Load Matrix from Shared Memory
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

Load Matrix from Shared Memory

Shared Memory Pointers

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
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

Load Matrix from Shared Memory

Shared Memory Pointers

- 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

Shared Memory Pointers
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

Load Matrix from Shared Memory

Shared Memory Pointers
ADVANCING TO NEXT K GROUP

K=0 .. 15

K=16 .. 31
ADVANCING TO NEXT K GROUP

smem_ptr = row_idx * 8 + column_idx;

K=0..15

K=16..31

smem_ptr = smem_ptr ^ 2;
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

Load Matrix from Shared Memory

Phase 0

K = 16..31
LOADING FROM SHARED MEMORY TO REGISTERS

Phase 1

Logical view of threadblock tile

Load Matrix from Shared Memory

K = 16..31
LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

Phase 2

K = 16..31

Load Matrix from Shared Memory
LOADING FROM SHARED MEMORY TO REGISTERS

Phase 3

Logical view of threadblock tile

Load Matrix from Shared Memory

K = 16..31
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
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
    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

```cpp
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
    mma(accum, frag_A, frag_B, accum);
}
```
CUTLASS ON NVIDIA A100
CUTLASS RELATIVE PERFORMANCE TO CUBLAS

CUTLASS 2.2 - CUDA 11 Toolkit - NVIDIA A100

cuBLAS

CUTLASS

DGEMM | IGEMM | SGEMM | TensorOp (f16) | TensorOp (f32) | TensorOp (TF32)

99%  | 99%  | 98%  | 99%  | 99%  | 99%  |
95%  | 93%  | 96%  | 95%  | 98%  | 97%  |
98%  | 97%  | 98%  | 97%  | 94%  | 90%  |
90%  | 90%  | 90%  | 90%  | 90%  | 90%  |
80%  | 83%  | 80%  | 80%  | 80%  | 80%  |
CUTLASS RELATIVE PERFORMANCE TO CUBLAS

CUTLASS 2.2 - CUDA 11 Toolkit - Three generations of GPU architectures

CuBLAS

CUTLASS

- 2080Ti
- A100
- TitanV

NN, NT, TN, TT

DGEMM, IGEMM, SGEMM, TensorOp (f16), TensorOp (f32), TensorOp (TF32)
CONCLUSION: NVIDIA A100 IS FAST AND PROGRAMMABLE

Tensor Cores on NVIDIA A100 in CUDA
- Order of magnitude speedup for matrix computations
- Programmable in CUDA via `mma.sync` with zero overhead
- Kernel design can avoid memory bottlenecks
- CUDA 11 Toolkit capable of near-peak performance

CUTLASS 2.2: May 2020
- Open source CUDA C++ template library for CUDA development
- Reusable building blocks for utilizing Tensor Cores on NVIDIA GPUs
- Near-optimal performance on NVIDIA Ampere Architecture

Try it out! [https://github.com/NVIDIA/cutlass](https://github.com/NVIDIA/cutlass)
REFERENCES

NVIDIA Ampere Architecture:

“Inside the NVIDIA Ampere Architecture” (GTC 2020 - S21730)

“NVIDIA Ampere Architecture In-Depth” (blog post)

“CUDA New Features and Beyond” (GTC 2020 - S21760)

“Tensor Core Performance on NVIDIA GPUs” (GTC 2020 - S21929)

“Inside the Compilers, Libraries and Tools for Accelerated Computing” (GTC 2020 - S21766)

CUTLASS

https://github.com/NVIDIA/cutlass (open source software, New BSD license)

GTC 2018 and GTC 2019 talks: GEMM structure and Volta Tensor Cores

CUTLASS Parallel For All blog post