Better Tensor Core Support in TVM with CUTLASS

Leyuan Wang, Dec. 16, 2021

ByteDance 字节跳动

Outline

I. Motivation

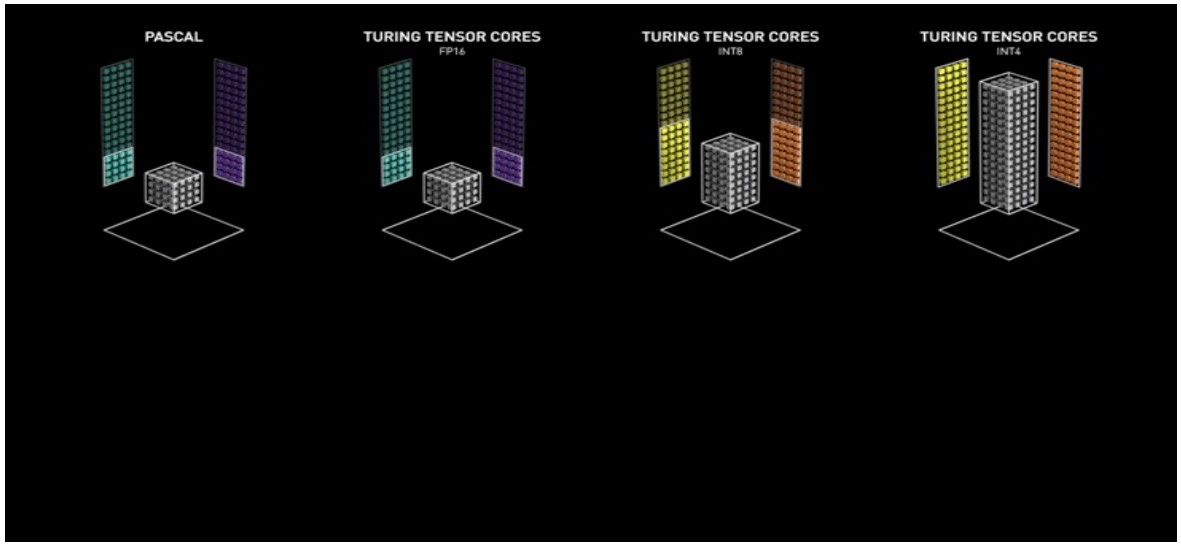
II. Why CUTLASS

III. Integration: BYOC CUTLASS

IV. Some performance numbers

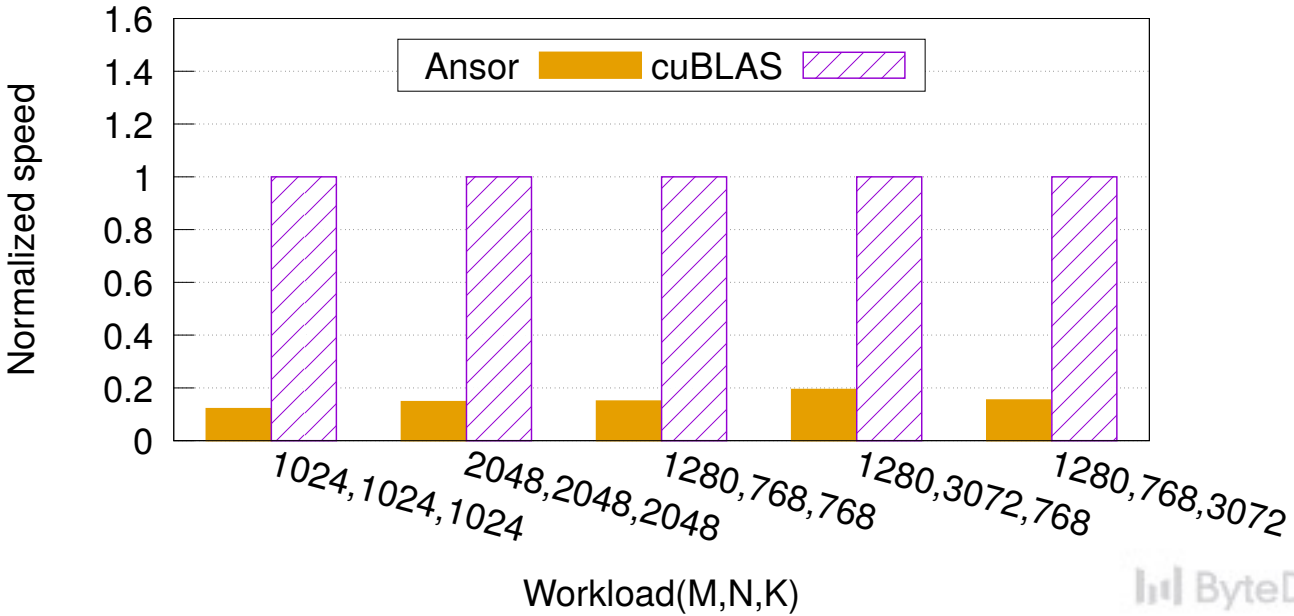
Motivation

- Nvidia GPUs are mostly used in machine learning cloud computing
- Tensor Core acceleration for mixed precision (e.g. fp16) deep learning model inference is widely used.



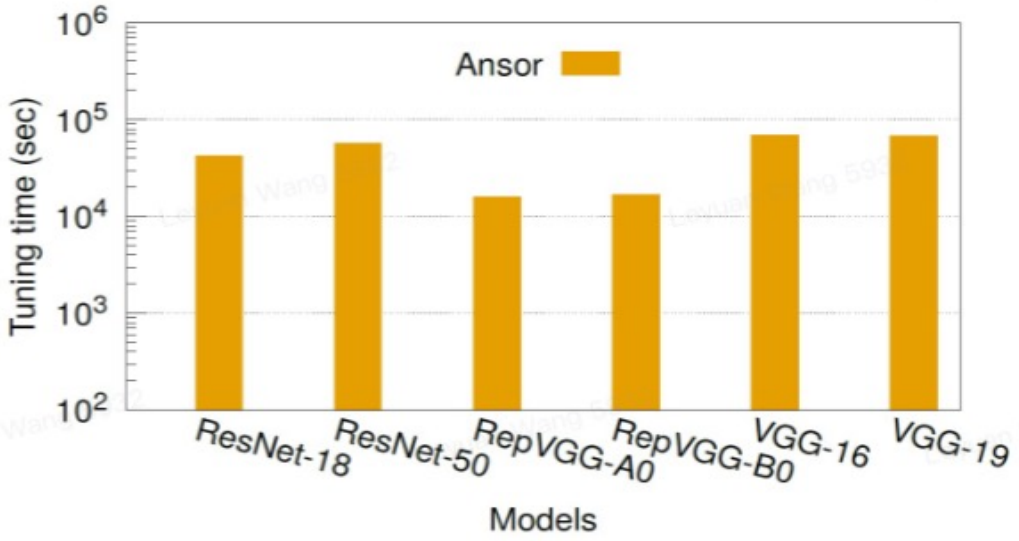
Motivation

- Currently there's performance gap between TVM and vendor libraries on compute intensive operators (e.g. gemm, conv, etc.) in deep learning workloads using tensor cores on NVIDIA GPUs



Motivation

- Long tuning time for searching best-performance compute intensive operators in current autotvm.
- A tophub data base can’t solve the whole problem because of dynamic workloads and cost to maintain.



Outline

I. Motivation

II. Why CUTLASS

III. Integration: BYOC CUTLASS

IV. Some performance numbers

Why choosing CUTLASS

- An open-source CUDA C++ templated header library, containing many modularized and customizable components.
- For single compute intensive kernels, CUTLASS can compete with SOTA close-source vendor libraries on NVIDIA GPUs.
- Support epilogue fusion, persistent kernel fusion, etc. which is a beneficial feature compared with other vendor libraries.
- Support different functionalities such as GEMM and conv, etc..
- Support different data type tensor core instructions (b1, int4, int8, fp16, bf16, fp32, tf32, fp64, complex, quaternion).
- Support different generations of hardware (Volta, Turing, Ampere, etc.).
- Support different versions of CUDA compilers.

Outline

I. Motivation

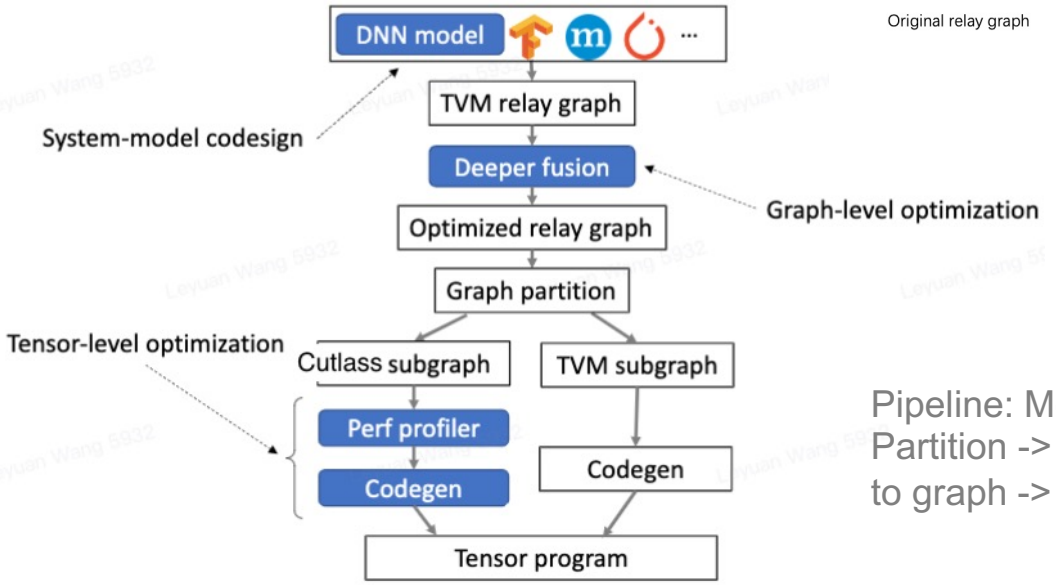
II. Why CUTLASS

III. Integration: BYOC CUTLASS

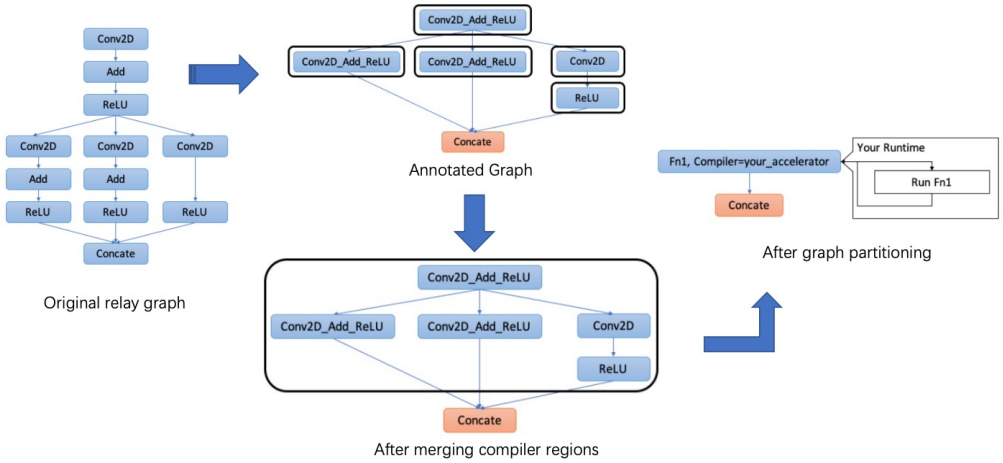
IV. Some performance numbers

Integration method

BYOC (Bring Your Own Codegen)



System Flow Graph



BYOC Pipeline

Pipeline: MergeComposite -> Annotate Target -> Graph Partition -> Tuner -> Codegen -> Compile -> Write back to graph -> Runtime

Outline

I. Motivation

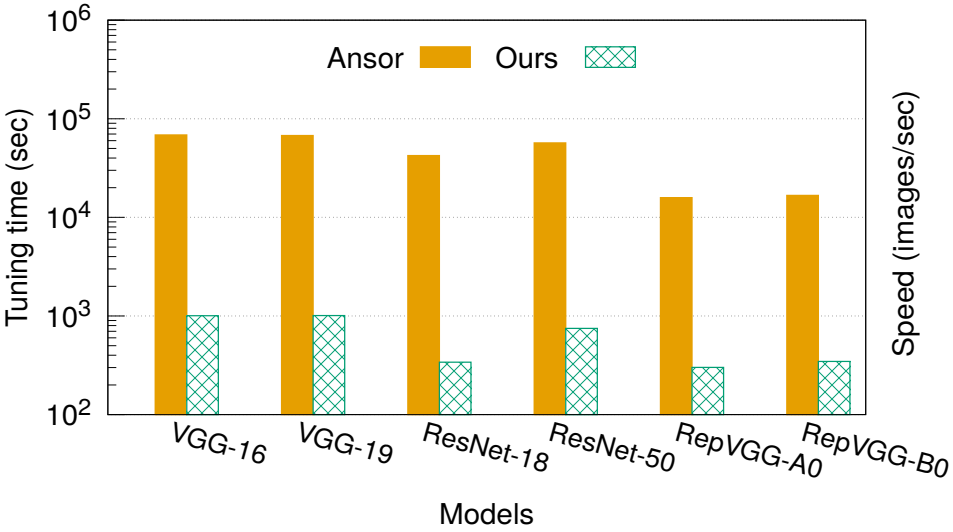
II. Why CUTLASS

III. Integration: BYOC CUTLASS

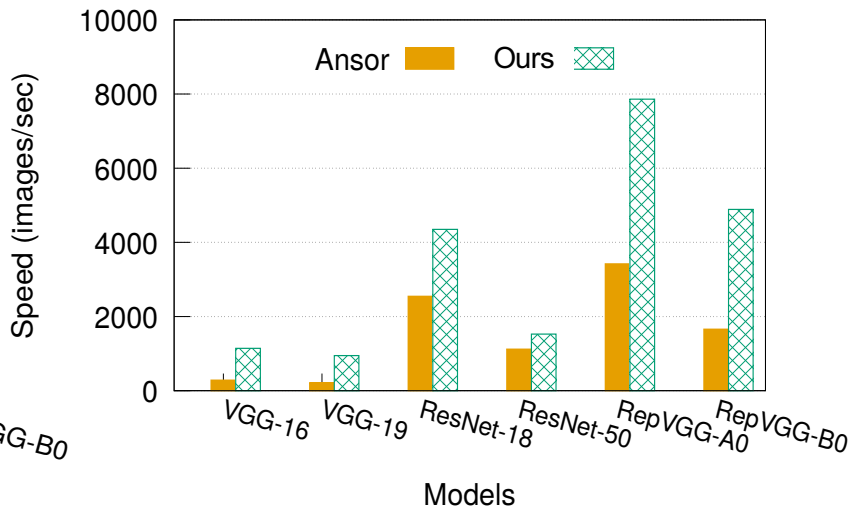
IV. Some performance numbers

Less Tuning Time, Faster Speed

- Baseline: Ansor tune w/ trials = 900 * # of tasks
- We finish tuning within 20 minutes for all models while Ansor takes 12 hours on average



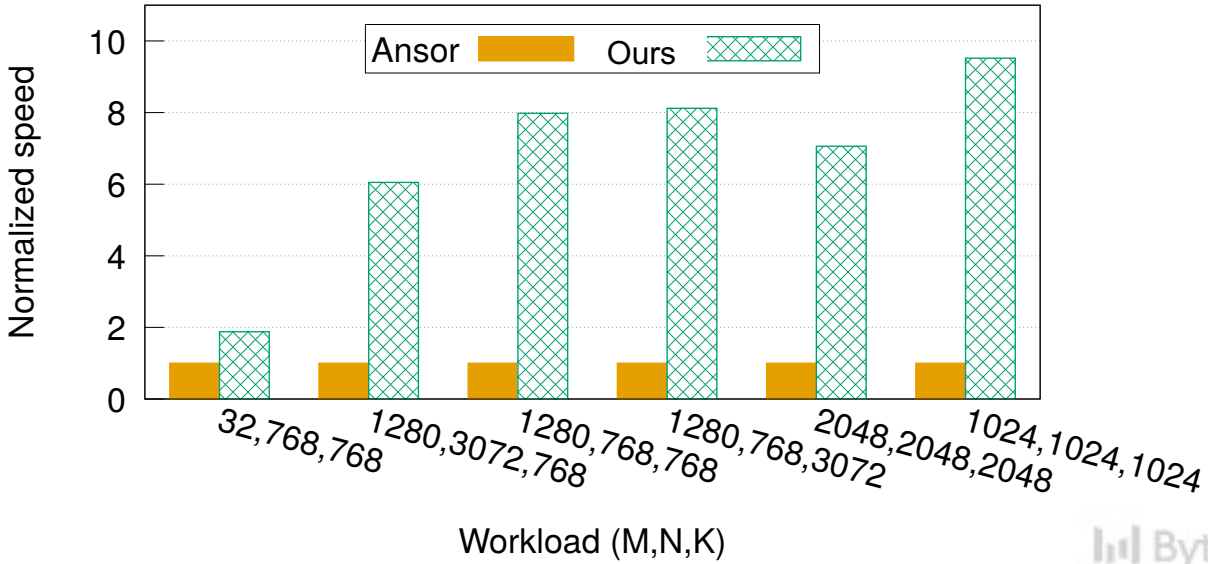
End-to-end model tuning time



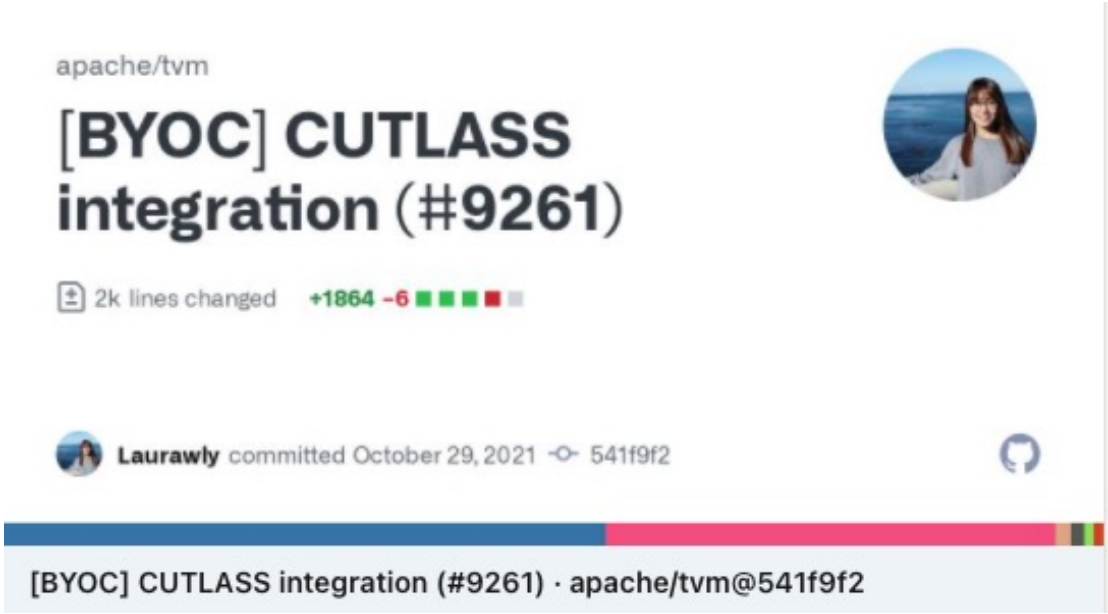
End-to-end model inference speed

GEMM Performance

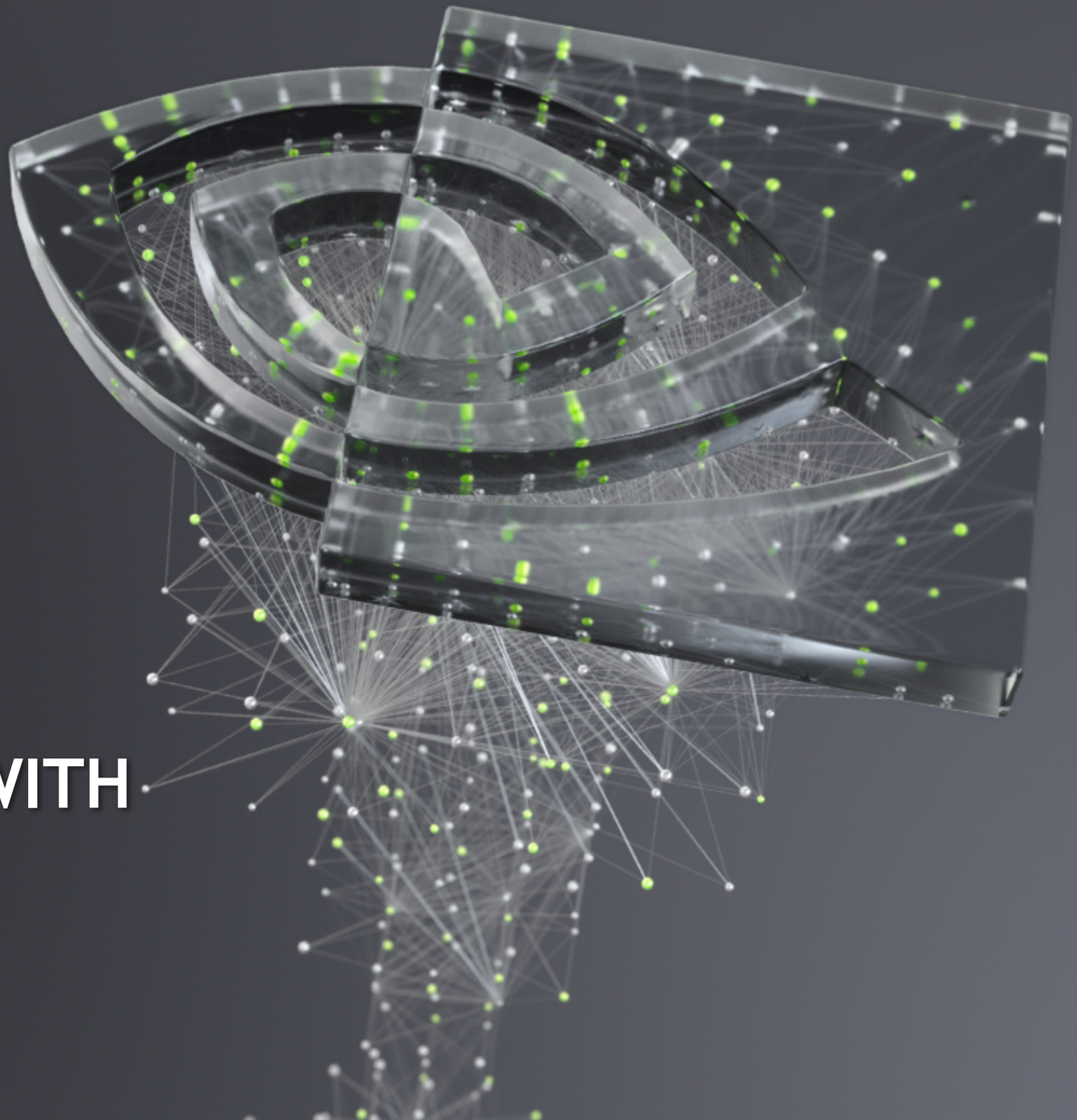
- Data type: FP16
- Baseline: Tune each operator for 2000 trials using Ansor
- GEMM workloads: square matrix + Bert GEMMS with bs 32, seq_length 40
- Hardware: NVIDIA T4 GPU



CUTLASS integration is merged into TVM



THANKS.



CUTLASS + TVM: FUSION WITH GEMM AND CONVOLUTION

Andrew Kerr, December 16, 2021

ACKNOWLEDGEMENTS

CUTLASS GitHub Community

19.5K clones, 1.4K stars, and many active users

CUTLASS Team

Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Duane Merrill, Pradeep Ramani, Cris Cecka, Vijay Thakkar

Contributors

Timothy Costa, Naila Farooqui, Markus Hohnerbach, Alan Kaatz, Wei Liu, Piotr Majcher, Dhiraj Reddy Nallapa, Mathew Nicely, Kyrylo Perelygin, Aniket Shivam, Paul Springer, Pawel Tabaszewski, Chinmay Talegaonkar, John Tran, Jin Wang, Yang Xu, Scott Yokim

Acknowledgements

Olivier Giroux, Mostafa Hagog, Bryce Lelbach, Julien Demouth, Joel McCormack, Aartem Belevich, Peter Han, Timmy Liu, Yang Wang, Nich Zhao, Jack Yang, Vicki Wang, Junkai Wu, Ivan Yin, Aditya Alturi, Shang Zhang, Takuma Yamaguchi, Stephen Jones, Luke Durant, Harun Bayraktar

AGENDA

Overview

NVIDIA Ampere Architecture and CUTLASS

Abstractions for Tensor Cores

Accelerated matrix operations

Efficient Epilogues

Data exchange for efficient access to Global Memory

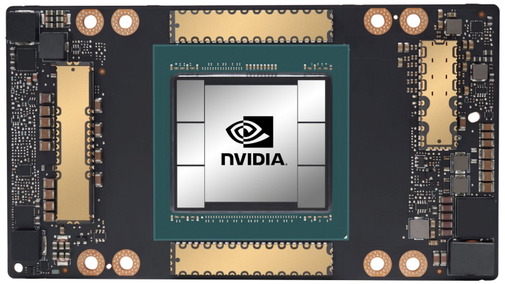
Epilogue Fusion Design Patterns

Opportunities for fusing custom operations with GEMM and CONV

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

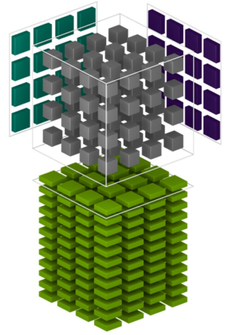
Additional Data Types and Mode

- Bfloat16, double, Tensor Float 32

Asynchronous copy

- Copy directly into shared memory – deep software pipelines

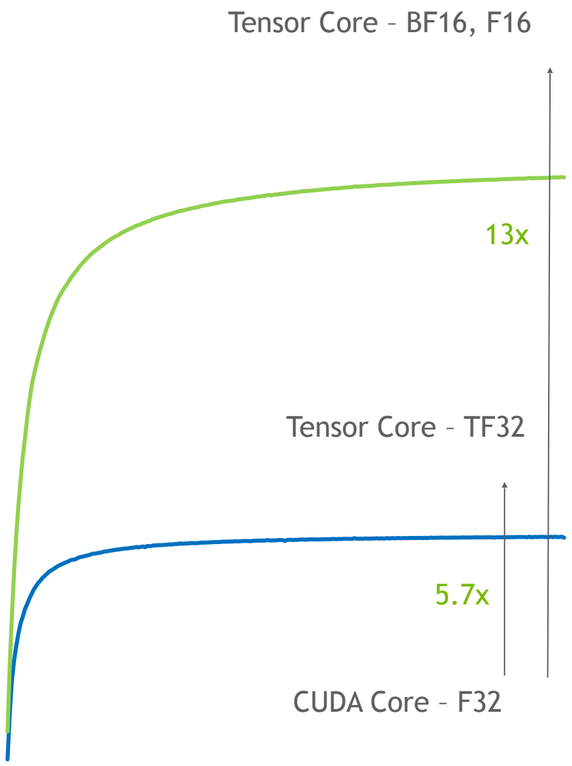
Many additional new features – see “Inside NVIDIA Ampere Architecture”



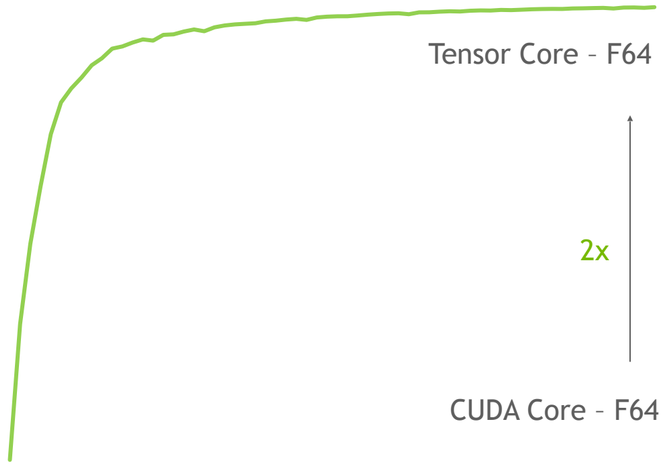
(Slides borrowed heavily from GTC’2020 and GTC’2021 CUTLASS presentations)

CUTLASS PERFORMANCE ON NVIDIA AMPERE ARCHITECTURE

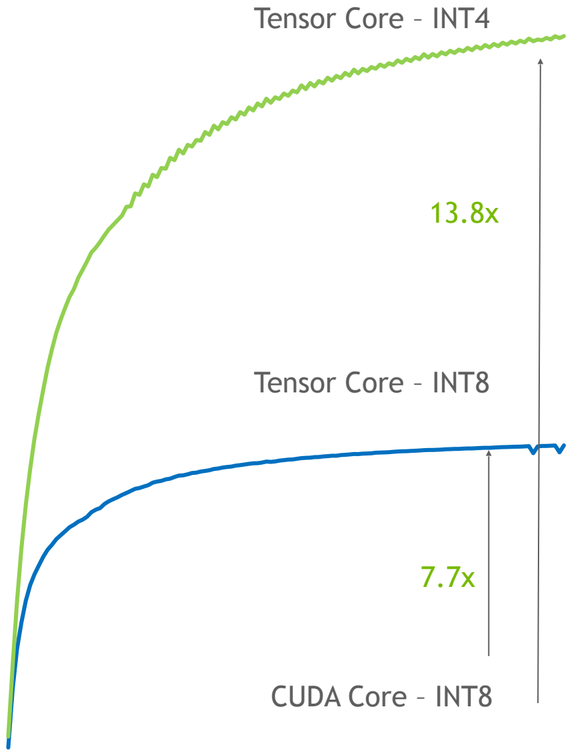
Mixed Precision Floating Point



Double Precision Floating Point



Mixed Precision Integer



m=3456, n=4096

CUTLASS

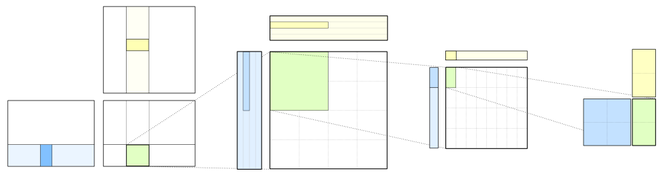
CUDA C++ Template Library for Deep Learning and High Performance Computing

CUTLASS: optimal CUDA C++ templates at all scopes and scales

Device	{ GEMM, Convolution, Reductions } x { all data types } x { SIMT, Tensor Cores } x { all architectures }
Kernel	GEMM, Batched GEMM, Convolution, Reduction, Fused output operations, Fused input operations
Threadblock	Pipelined Matrix Multiply, Epilogue, Collective access to tensors, Convolution matrix access
Warp	Tensor Core Multiply-Add operations, Efficient access to permuted tensor layouts
Thread	Numeric conversion, <functional> operators on arrays, complex<T>, fast math algorithms
Architecture intrinsic	Templates wrapping architecture-specific PTX instructions (e.g. mma, cp.async, ldmatrix, cvt)

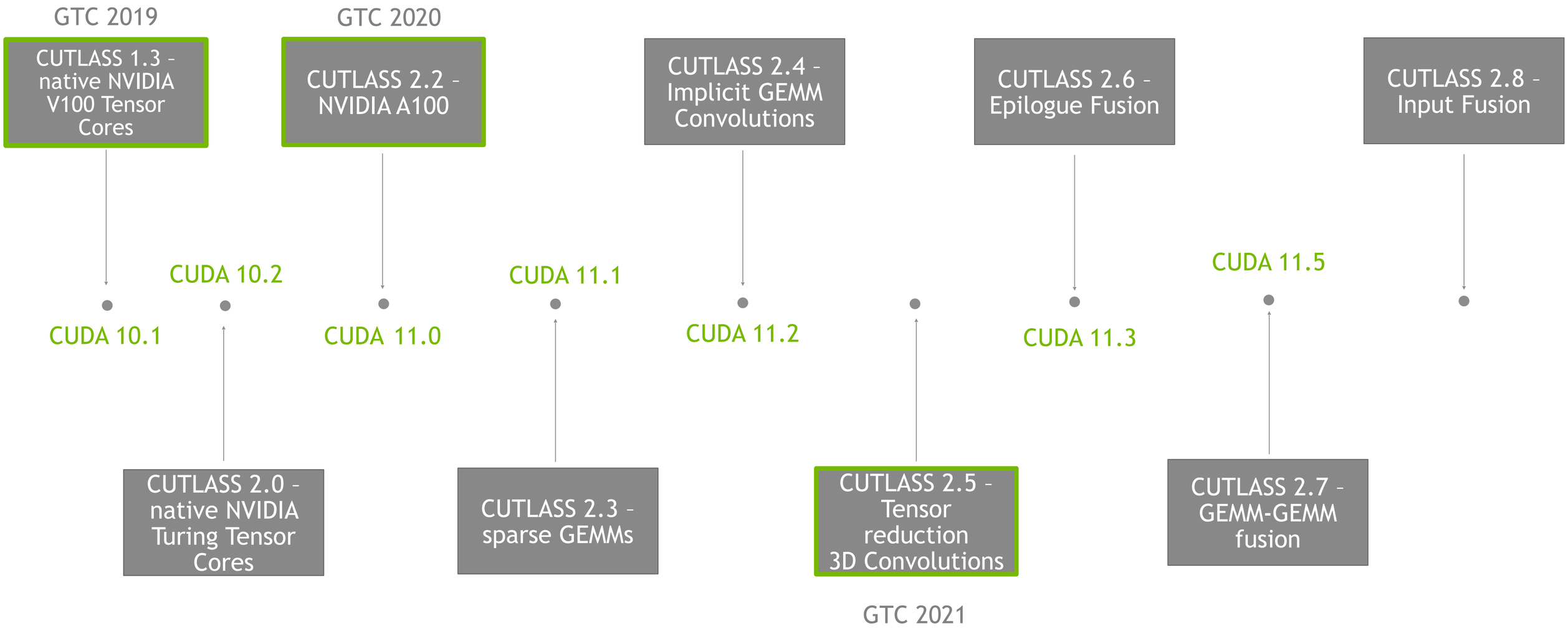
Open source: https://github.com/NVIDIA/cutlass

- Latest revision: CUTLASS 2.8
- Documentation: https://github.com/NVIDIA/cutlass#documentation
- Functionality: https://github.com/NVIDIA/cutlass/blob/master/media/docs/functionality.md

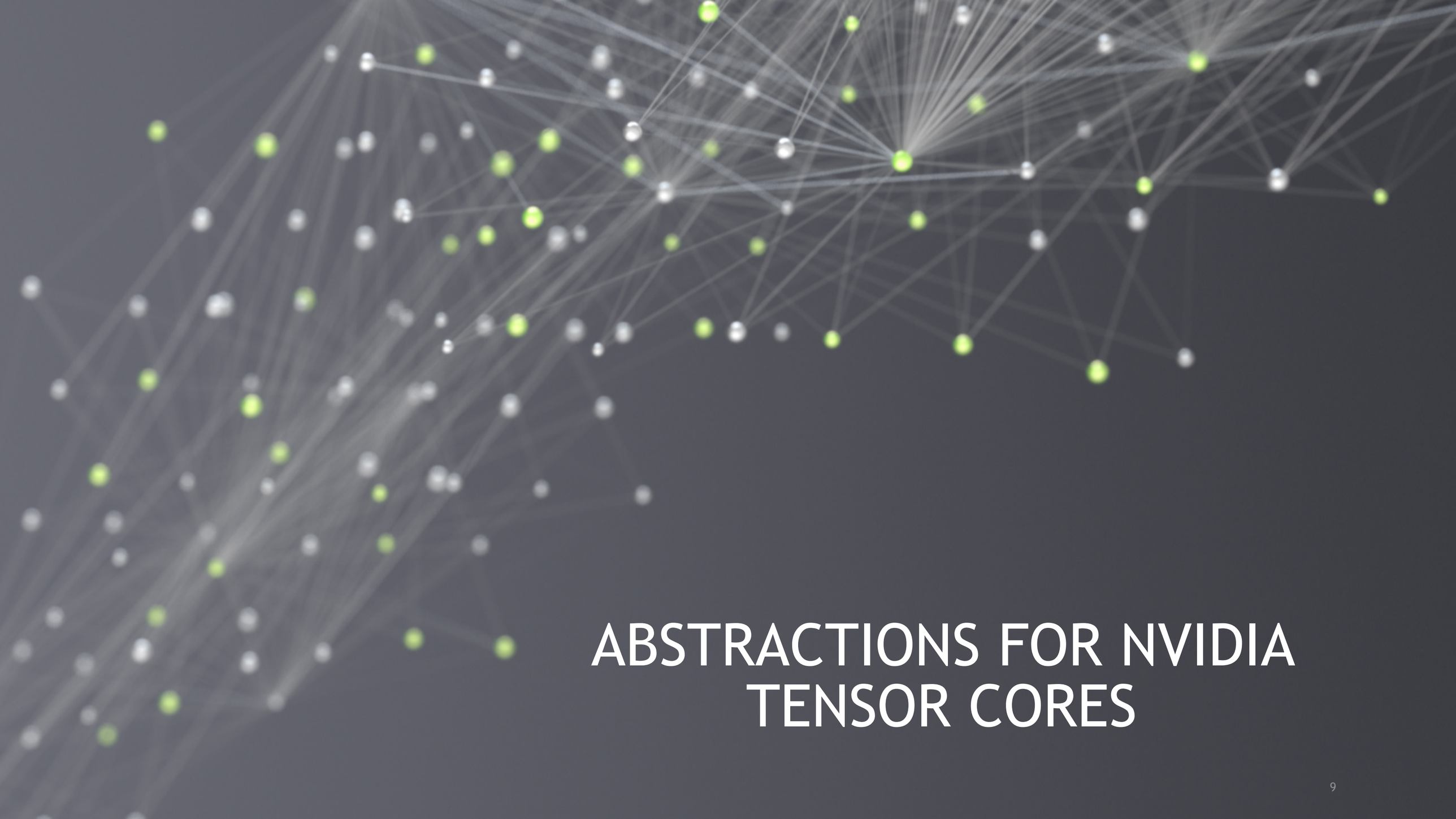


CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



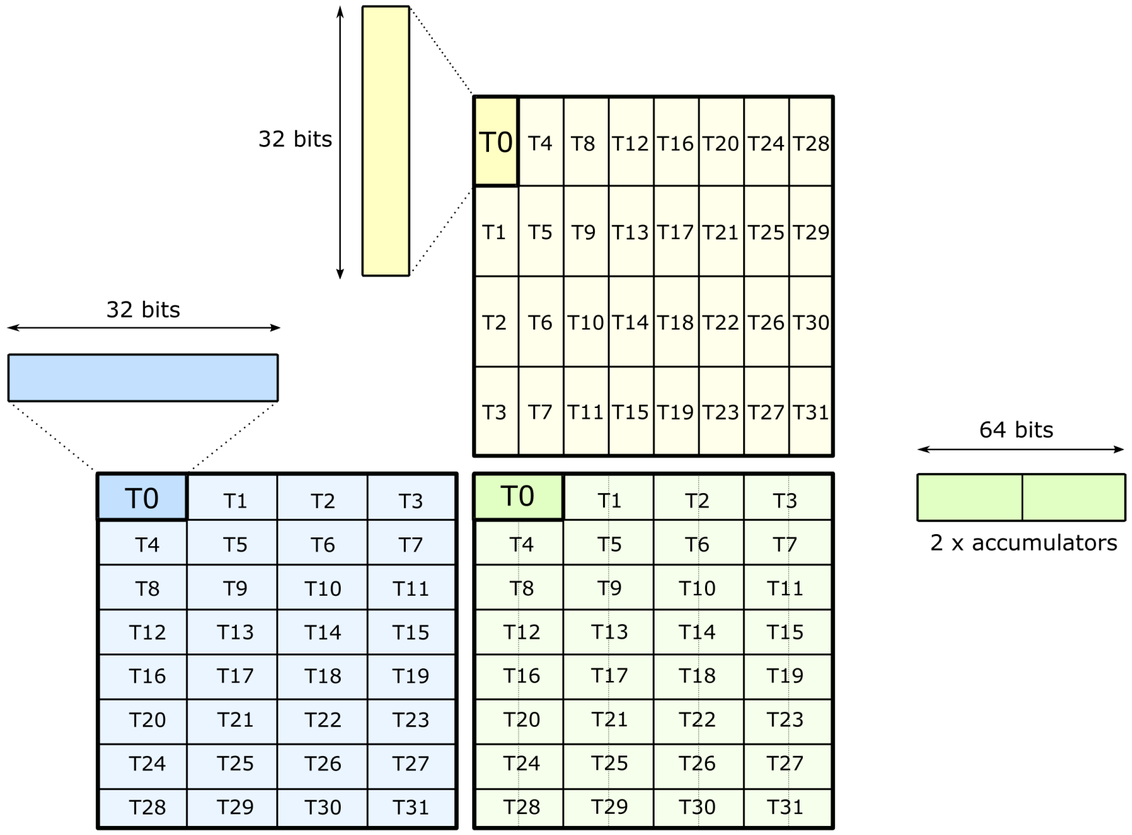
https://github.com/NVIDIA/cutlass



ABSTRACTIONS FOR NVIDIA TENSOR CORES

HELLO WORLD: TENSOR CORES

Map each thread to coordinates of the matrix operation
Load inputs from memory
Perform the matrix operation
Store the result to memory



CUDA example

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

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

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

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

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

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

PERFORMANCE IMPLICATIONS

Load A and B inputs from memory: 2 x 4B per thread
Perform one Tensor Core operation: 2048 flops per warp

2048 flops require 256 B of loaded data
➔ 8 flops/byte

NVIDIA A100 Specifications:

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

➔ 400 flops/byte

8 flops/byte * 1.6 TB/s ➔ 12 TFLOP/s

This kernel is global memory bandwidth limited.

CUDA example

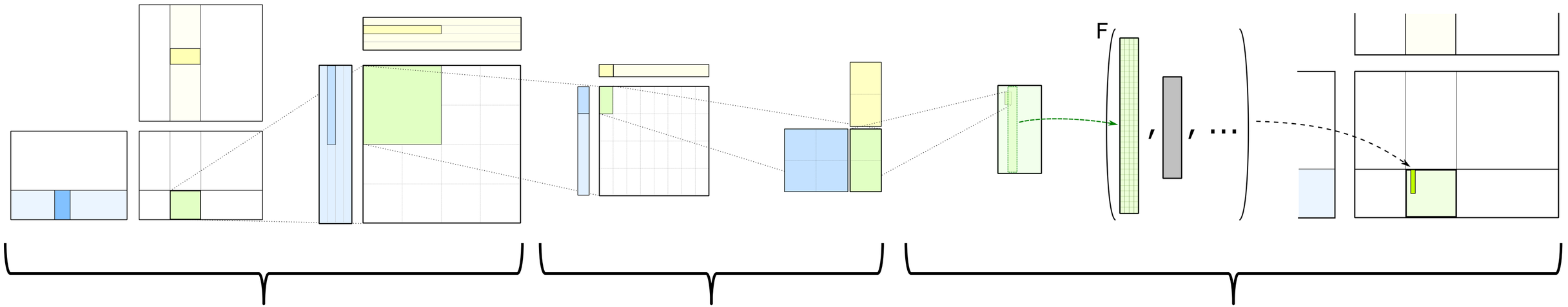
```
__global__ void tensor_core_example_8x8x16(
  int32_t        *D,
  uint32_t const *A,
  uint32_t const *B,
  int32_t const  *C) {

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

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

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

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

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

CUTLASS GEMM MODEL

Decoupling application-specific behavior, Tensor Core operations, and Fused output operations

Matrix Multiply-Accumulate Phase

Epilogue Phase



Global Memory → Shared Memory

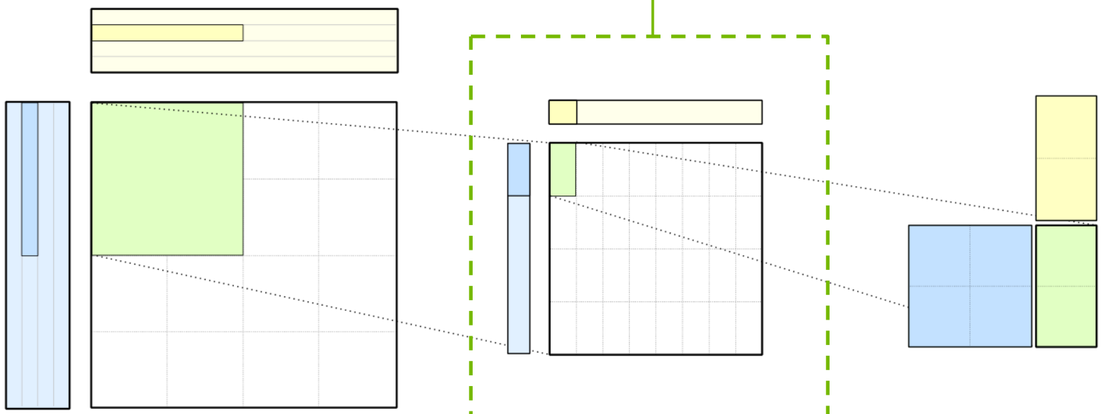
Shared Memory → Tensor Cores

Accumulators → Global Memory

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

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

CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES



Shared Memory

Warp-level matrix multiply

Tensor Cores

```
using MmaWarp = 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[MmaWarp::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[MmaWarp::Shape::kN * GemmK];

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

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

MmaWarp mma_warp;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += MmaWarp::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
                        //   and B matrices

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

CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES

Tile Iterator Constructors:
Initialize pointers into permuted Shared Memory buffers

Fragments:
Register-backed arrays holding each thread's data

Tile Iterator:
load() - Fetches data from permuted Shared Memory buffers

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

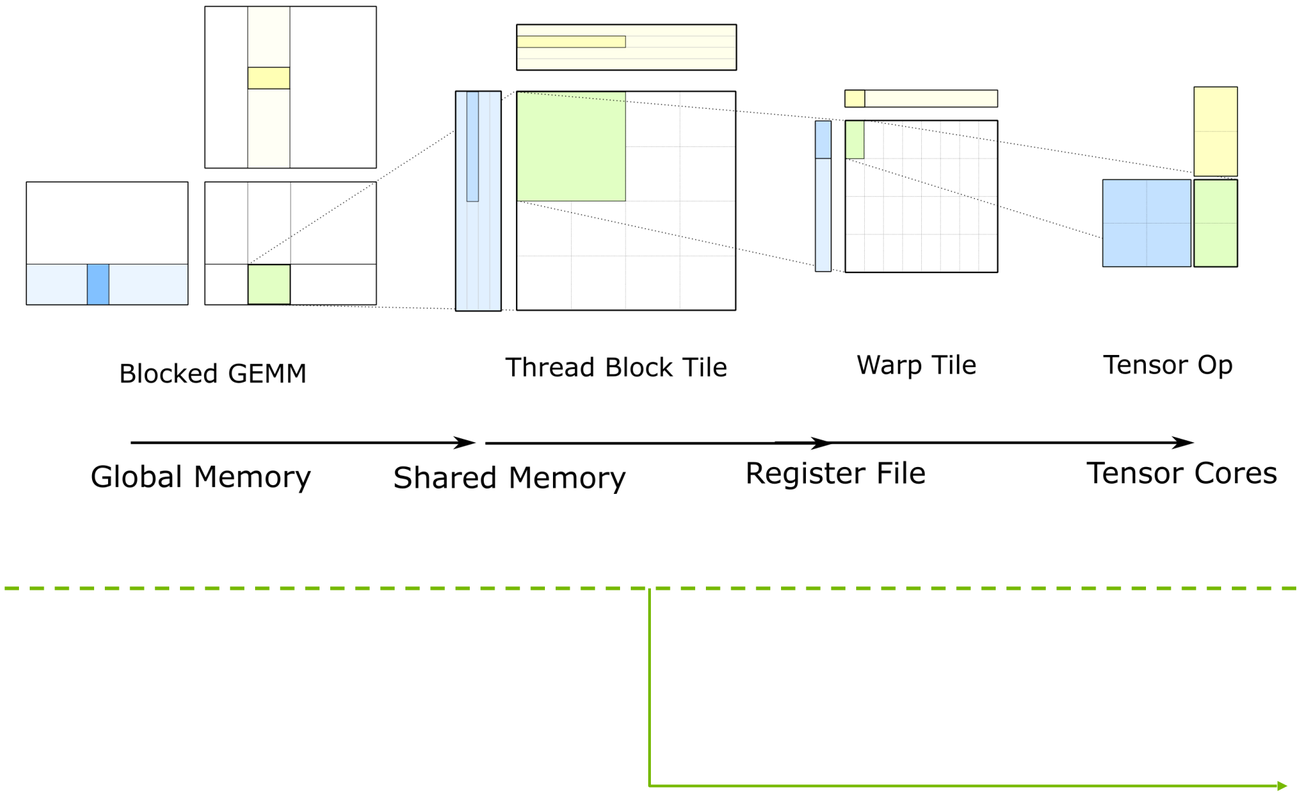
Warp-level matrix multiply:
Decomposes a large matrix multiply into Tensor Core operations

```
using MmaWarp = 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[MmaWarp::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[MmaWarp::Shape::kN * GemmK];

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

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

MmaWarp mma_warp;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += MmaWarp::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
                        //   and B matrices

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

CUTLASS: ABSTRACTION FOR PIPELINED MATRIX MULTIPLY

Threadblock-scoped operator for Matrix Multiply-Accumulate

`cutlass::gemm::threadblock::MmaMultistage`



“Kernel scoped” device code

```
// Compute position within threadblock
int thread_idx = threadIdx.x;

// Construct iterators to A and B operands
typename MmaMultistage::IteratorA iterator_A(
  params.params_A,
  ptr_A,
  {params.problem_size.m(), problem_size_k},
  thread_idx,
  tb_offset_A);

typename MmaMultistage::IteratorB iterator_B(
  params.params_B,
  ptr_B,
  {problem_size_k, params.problem_size.n()},
  thread_idx,
  tb_offset_B);

// Broadcast the warp_id computed by lane 0 to ensure dependent co
// is compiled as warp-uniform.
int warp_idx = __shfl_sync(0xffffffff, threadIdx.x / 32, 0);

int lane_idx = threadIdx.x % 32;

//
// Construct multistage, pipelined thread-scoped matrix multiply
//
MmaMultistage mma_multistage(
  shared_storage.main_loop, thread_idx, warp_idx, lane_idx);

typename MmaMultistage::FragmentC accumulators;

accumulators.clear();

// Compute matrix multiply
mma_multistage(
  gemm_k_iterations,
  accumulators,
  iterator_A,
  iterator_B,
  accumulators);
```

CUTLASS: ABSTRACTION FOR PIPELINED MATRIX MULTIPLY

Tile Iterator Constructors:
Initialize pointers and predicates into Global Memory

CUDA optimization technique:
Compute warp index as a provably warp-uniform value for architecture-specific optimization by the compiler

Construct matrix multiply-accumulate operator:
Constructs warp-level tile iterators, Shared Memory store iterators, and stage counters

Accumulator Fragment:
Register-backed arrays holding Accumulators

Threadblock-level matrix multiply:
Global to Shared Memory, Shared Memory to Registers, Tensor or CUDA Core multiply-add operations

“Kernel scoped” device code

```
// Compute position within threadblock
int thread_idx = threadIdx.x;

// Construct iterators to A and B operands
typename MmaMultistage::IteratorA iterator_A(
  params.params_A,
  ptr_A,
  {params.problem_size.m(), problem_size_k},
  thread_idx,
  tb_offset_A);

typename MmaMultistage::IteratorB iterator_B(
  params.params_B,
  ptr_B,
  {problem_size_k, params.problem_size.n()},
  thread_idx,
  tb_offset_B);

// Broadcast the warp_id computed by lane 0 to ensure dependent co
// is compiled as warp-uniform.
int warp_idx = __shfl_sync(0xffffffff, threadIdx.x / 32, 0);

int lane_idx = threadIdx.x % 32;

//
// Construct multistage, pipelined thread-scoped matrix multiply
//
MmaMultistage mma_multistage(
  shared_storage.main_loop, thread_idx, warp_idx, lane_idx);

typename MmaMultistage::FragmentC accumulators;

accumulators.clear();

// Compute matrix multiply
mma_multistage(
  gemm_k_iterations,
  accumulators,
  iterator_A,
  iterator_B,
  accumulators);
```

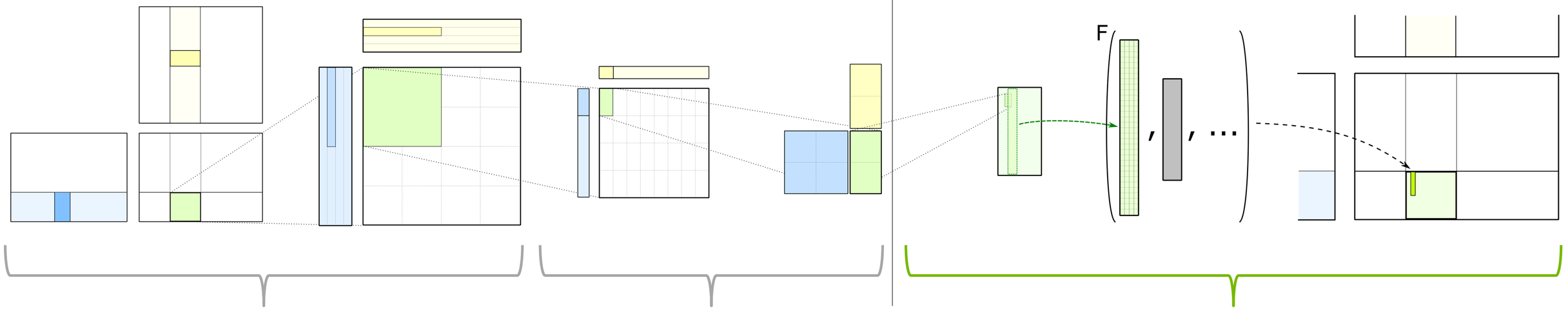
EFFICIENT EPILOGUES

EPILOGUE PHASE

Efficiently storing GEMM output to Global Memory

Matrix Multiply-Accumulate Phase

Epilogue Phase



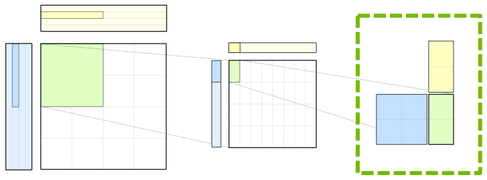
Global Memory → Shared Memory

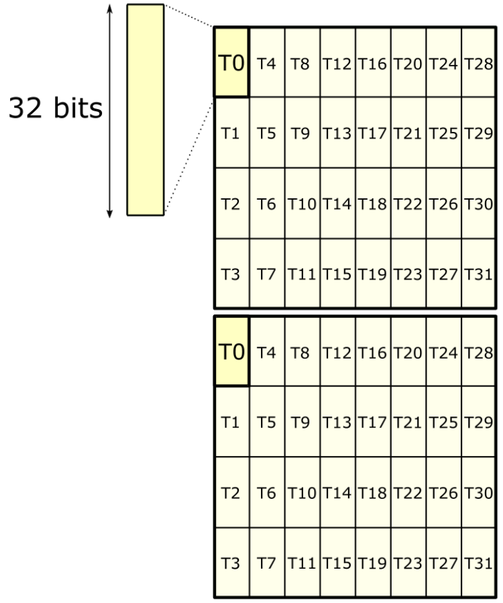
Shared Memory → Tensor Cores

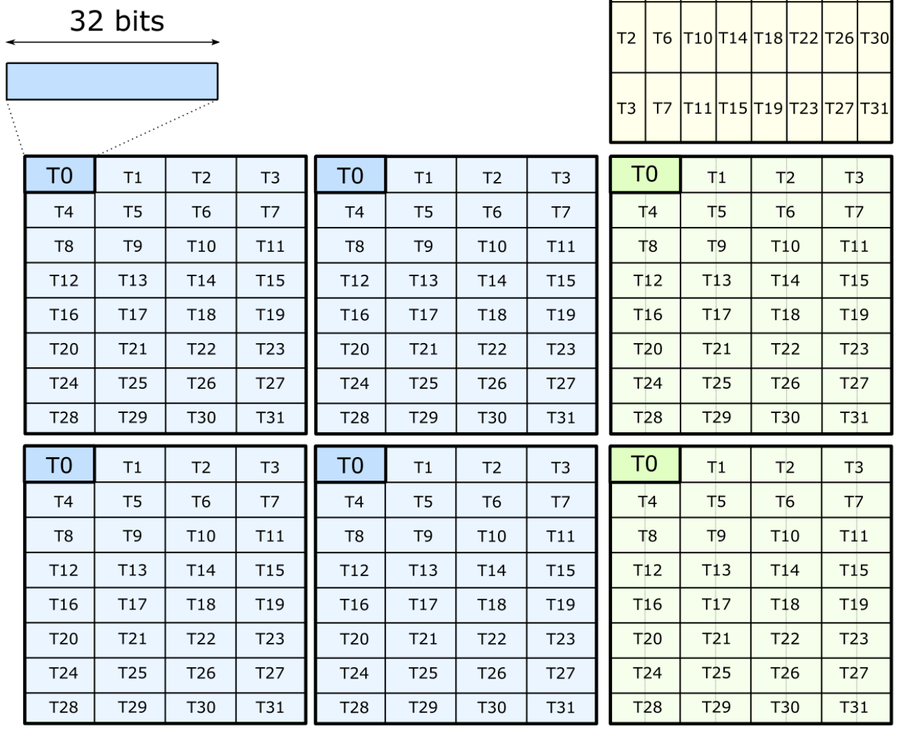
Accumulators → Global Memory

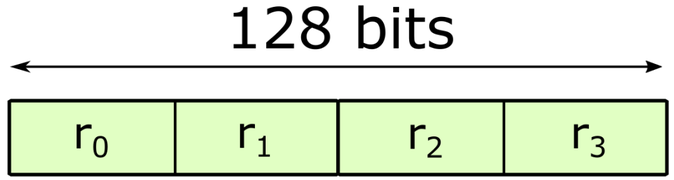
- Rearrangement of accumulators
- Fused math operations
- Type conversions and packing
- Coalesced access to Global Memory

TENSOR CORE REGISTER ARRANGEMENT





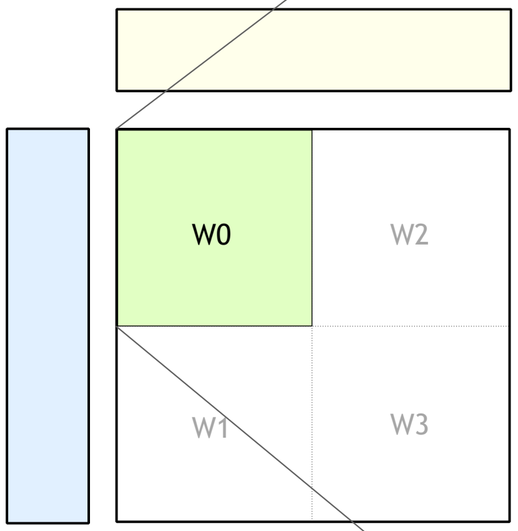




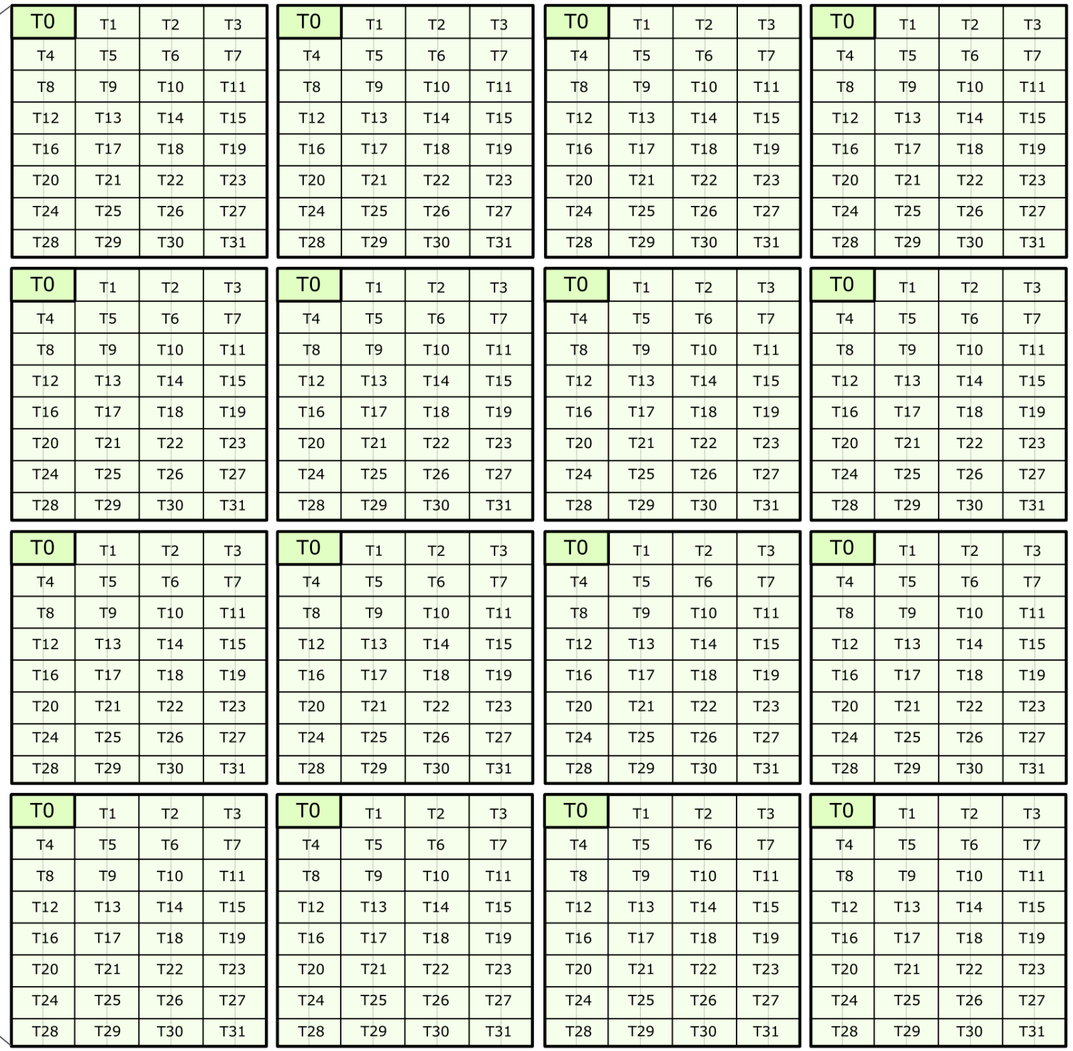

```
uint32_t const A[4];
uint32_t const B[2];

float    const C[4];
float          D[4];


// Example targets 16-by-8-by-32 Tensor Core
operation

asm(

"mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32
"
  "   { %0, %1, %2, %3 },     "
  "   { %4, %5, %6, %7 },     "
  "   { %8, %9 },             "
  "   { %10, %11, %12, %13 };"
  :
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
  :
    "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
    "r"(B[0]), "r"(B[1]),

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

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

WARP-WIDE ARRANGEMENT OF ACCUMULATORS



Threadblock Tile



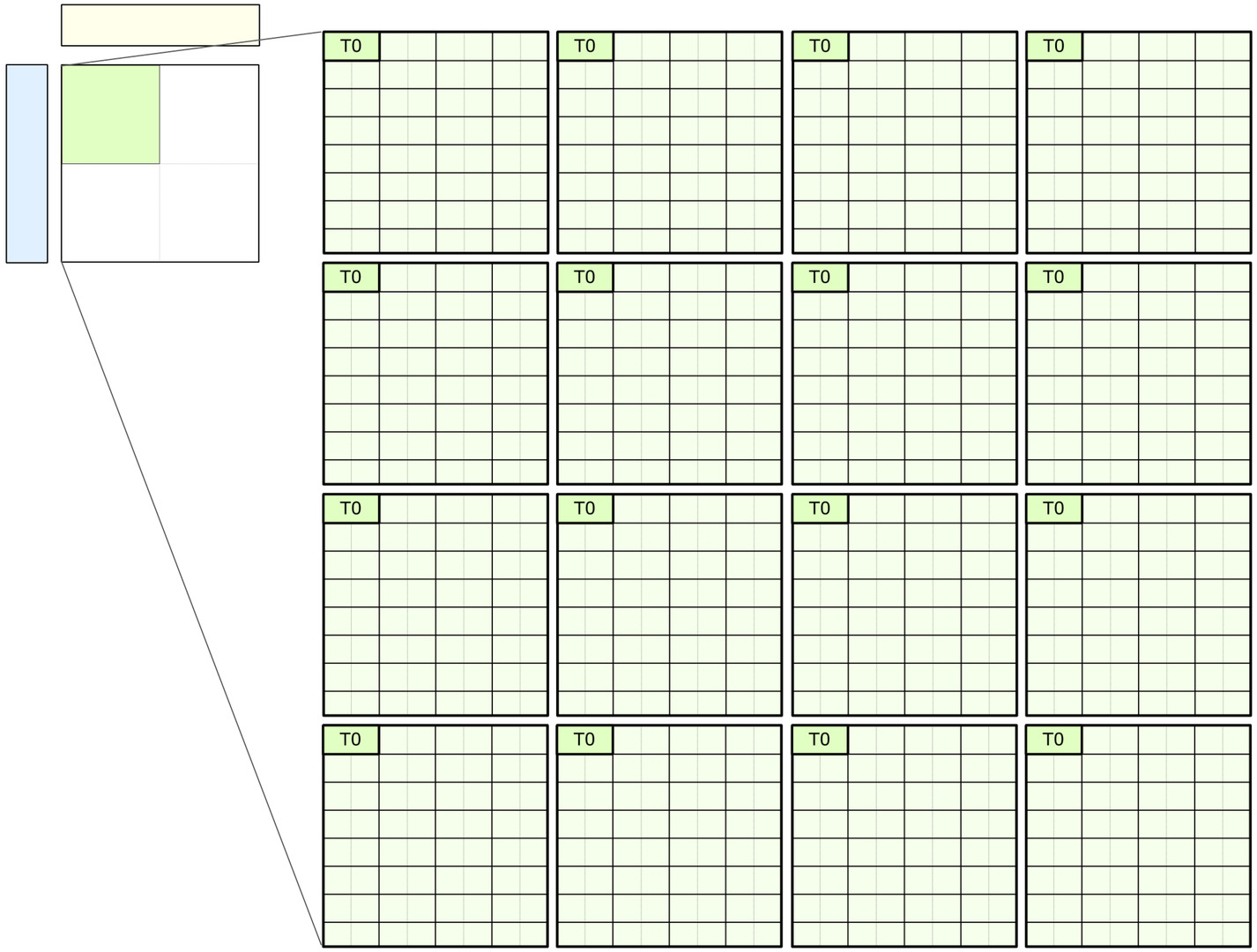
Warp Tile
32-by-32

Typical threadblock configurations.

Threadblock Shape	Warp Shape	Accumulator Count
64-by-64	32-by-32	32
128-by-64	64-by-32	64
128-by-128	64-by-64	128
256-by-128	64-by-64	128

Four warps per threadblock to utilize SM

Directly Store Accumulators



“Warp scoped” device code

```
int const kWarpCountM = 2;
int const kWarpCountN = 2;

int const kMmaCountM = 2;
int const kMmaCountN = 4;

int const kMmaShapeM = 16;
int const kMmaShapeN = 8;

// Accumulator array in Registers
float accumulators[kMmaCountM * kMmaCountN * 4];

// Output matrix in Global Memory
float *D_gmem = ...;
int    ldm    = ...;

// Indexing logic — map threads to coordinates
int warp_idx   = (threadIdx.x / 32);
int thread_idx = (threadIdx.x % 32);

int warp_m     = (warp_idx % kWarpCountM) *
                 (kWarpCountM * kMmaCountM * kMmaShapeM);

int warp_n     = (warp_idx / kWarpCountN) *
                 (kWarpCountN * kMmaCountN * kMmaShapeN);

int thread_m   = (thread_idx / 4);
int thread_n   = (thread_idx % 4) * 2;


// Loop over MMA instructions in the M dimension
#pragma unroll
for (int mma_m = 0; mma_m < kMmaCountM; ++mma_m) {

  // Loop over MMA instructions in the N dimension
  #pragma unroll
  for (int mma_n = 0; mma_n < kMmaCountN; ++mma_n) {

    // Loop over accumulators within one MMA instruction
    #pragma unroll
    for (int i = 0; i < 4; ++i) {

      // Compute index in accumulator array
      int r_idx = i + mma_m * 4 + mma_n * 4 * kMmaCountM;

      // Compute output location within threadblock
      int m = warp_m + thread_m + mma_m * kMmaShapeM + (i / 2) * 8;
      int n = warp_n + thread_n + mma_n * kMmaShapeN + (i % 2);

      // Store to Global Memory
      D_gmem[m + n * ldm] = accumulators[r_idx];
    }
  }
}
```

EFFICIENT EPILOGUES

Directly storing accumulators is not efficient for all data types and configurations

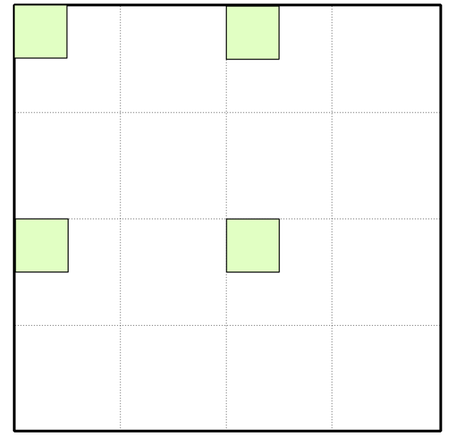
- Each warp writes to a disjoint 8-by-8 tile at a time
- Difficult for the hardware coalesce into accesses spanning large cache lines
- Mixed precision narrows the memory access size, resulting in small store operations

CUTLASS epilogues exchange accumulators through Shared Memory first

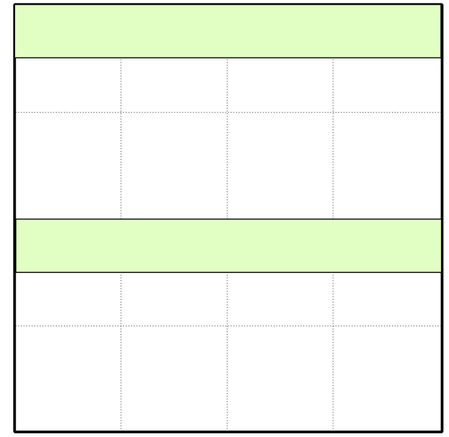
Objectives for efficiently accessing Global Memory

- Threads striped over (a few) large, contiguous tiles
- Store to whole cache lines, limited only by the width of Threadblock tiles
- Support larger vector access sizes
- Take advantage of data exchange by fusing other operations

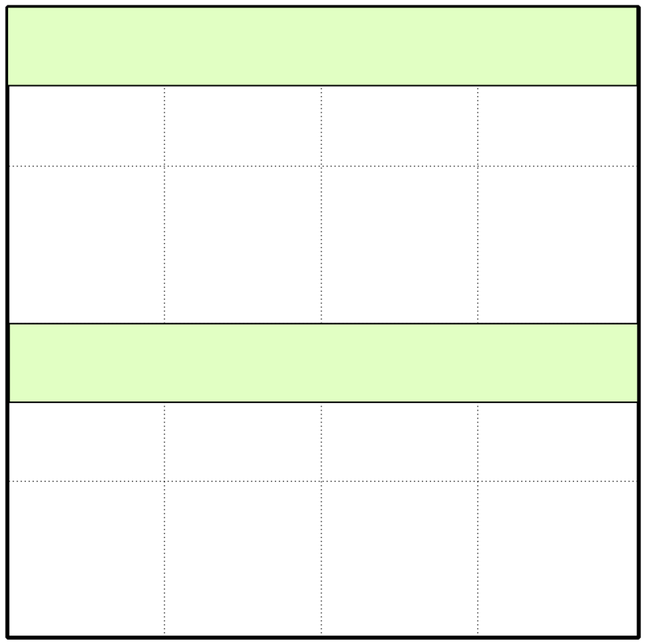
Direct Store access pattern



Coalesced accesses over contiguous tiles

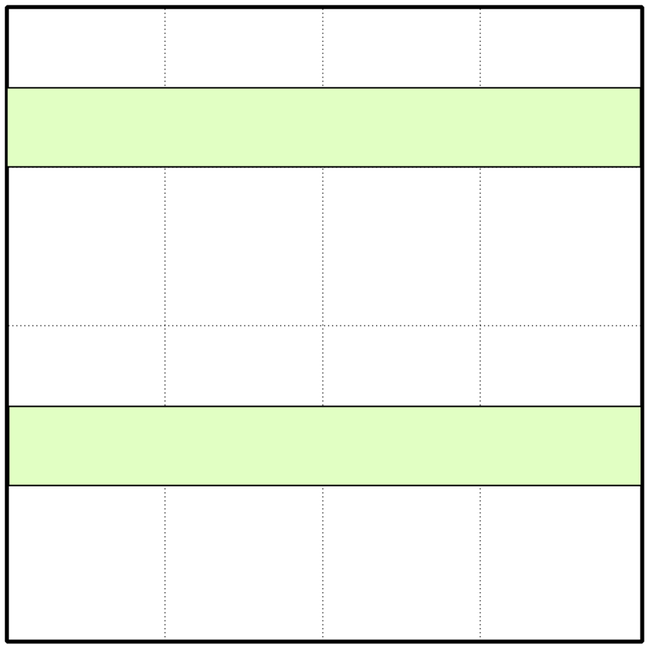


EFFICIENT EPILOGUES



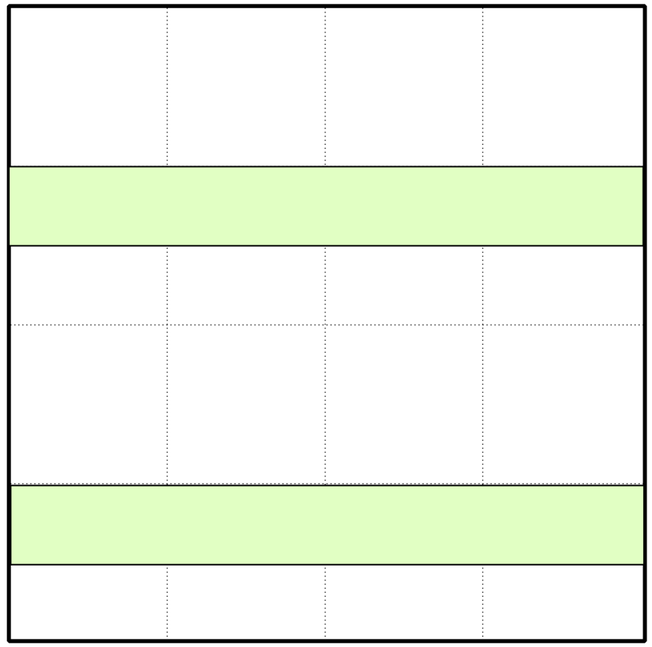
```
for each “slice”
      __syncthreads();
      Store a slice of accumulators to Shared Memory
      __syncthreads();
      Load the slice of accumulators from Shared Memory into registers
      Compute fused operations on accumulator slice and other matrices
      Store to Global Memory
      Advance Global Memory pointers
```

EFFICIENT EPILOGUES



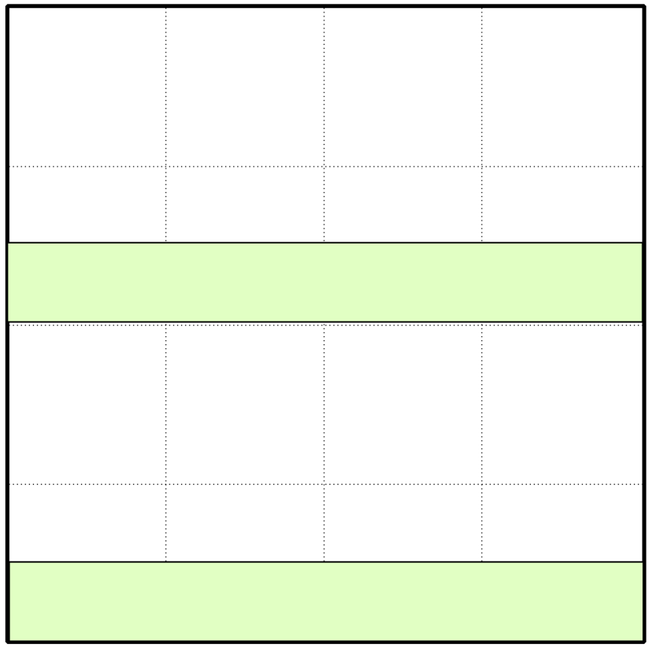
```
for each “slice”
      __syncthreads();
      Store a slice of accumulators to Shared Memory
      __syncthreads();
      Load the slice of accumulators from Shared Memory into registers
      Compute fused operations on accumulator slice and other matrices
      Store to Global Memory
      Advance Global Memory pointers
```

EFFICIENT EPILOGUES



```
for each “slice”
      __syncthreads();
      Store a slice of accumulators to Shared Memory
      __syncthreads();
      Load the slice of accumulators from Shared Memory into registers
      Compute fused operations on accumulator slice and other matrices
      Store to Global Memory
      Advance Global Memory pointers
```

EFFICIENT EPILOGUES



```
for each “slice”
      __syncthreads();
      Store a slice of accumulators to Shared Memory
      __syncthreads();
      Load the slice of accumulators from Shared Memory into registers
      Compute fused operations on accumulator slice and other matrices
      Store to Global Memory
      Advance Global Memory pointers
```

CUTLASS Epilogue

Source fragment
Stores tiles from “C” matrix loaded from GMEM

Accumulator “fragment iterator”
Iterates over “slices” of the overall accumulator fragment

Load from Global Memory
Fetches a tile from the “C” matrix

Store Accumulators to Shared Memory
Copies from accumulator fragment to Shared Memory

Loads Accumulators from Shared Memory
Resulting ownership of data ‘aligned’ to vectors in the output matrix

Elementwise functor

Store to Global Memory

“Threadblock scoped” device code

```
template < ... >
class Epilogue {
public:

  /// Executes the steps of the CUTLASS Epilogue
  CUTLASS_DEVICE
  void operator()(
    OutputOp const &output_op,                    ///< Elementwise functor
    OutputTileIterator destination_iterator,      ///< Tile iterator for destination tensor in Global Memory
    AccumulatorTile const &accumulators,          ///< Accumulators — computed during Matrix Multiply phase
    OutputTileIterator source_iterator            ///< Tile iterator for source tensor in Global Memory
    ) {

    typename OutputTileIterator::Fragment source_fragment;
    source_fragment.clear();

    // Iterator over warp-level accumulator fragment
    AccumulatorFragmentIterator accum_fragment_iterator(accumulators);

    // Iterate over slices of the accumulator
    #pragma unroll
    for (int iter = 0; iter < OutputTileIterator::kIterations; ++iter) {

      // Load the source tensor from Global Memory
      source_iterator.load(source_fragment);
      ++source_iterator;

      __syncthreads();

      // Store slice of accumulators to Shared Memory
      typename AccumulatorFragmentIterator::Fragment accum_fragment;

      accum_fragment_iterator.load(accum_fragment);
      ++accum_fragment_iterator;

      this->warp_tile_iterator_.store(accum_fragment);

      __syncthreads();

      // Load fragments from Shared Memory
      typename SharedLoadIterator::Fragment aligned_accum_fragment;
      shared_load_iterator_.load(aligned_accum_fragment);

      // Compute an elementwise operation on the accumulated result
      typename OutputTileIterator::Fragment output_fragment;

      apply_output_operator_(output_fragment, output_op, aligned_accum_fragment, source_fragment);

      // Store the resulting fragment to Global Memory
      destination_iterator.store(output_fragment);
      ++destination_iterator;
    }
  }
};
```

Combining Matrix Multiply and Epilogue Phases

Accumulator fragment
Contains result of matrix multiply operation

Execution of matrix multiply
Issues Tensor Core operations

Epilogue functor
Performs elementwise operations on result of matrix multiply

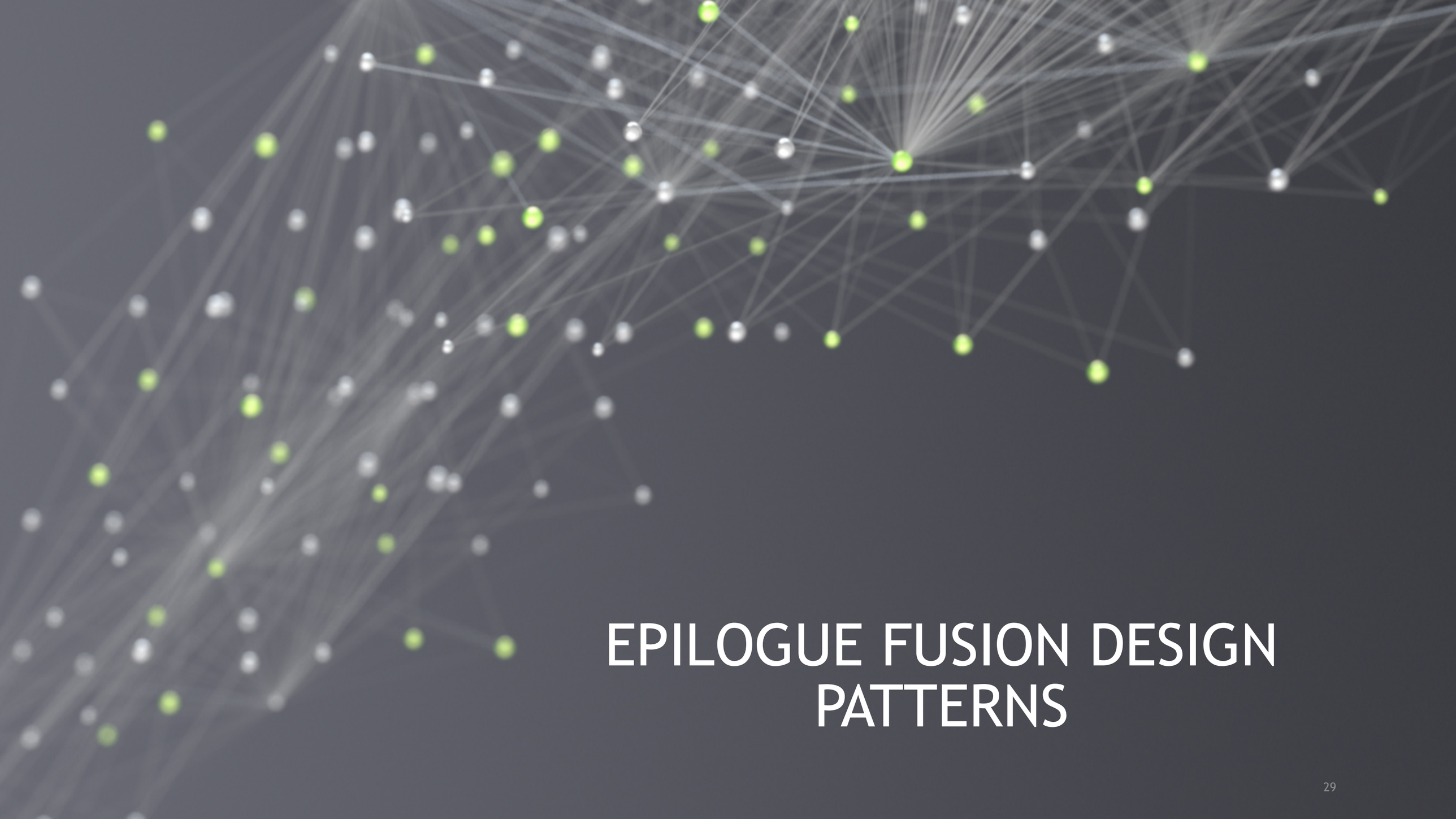
Epilogue output tile iterator
Loads and stores to output matrix in Global Memory

Epilogue constructor
Initializes internal pointers to store and load from Shared Memory

Execution of Epilogue
Transforms accumulators and stores to Global Memory

“Kernel scoped” device code

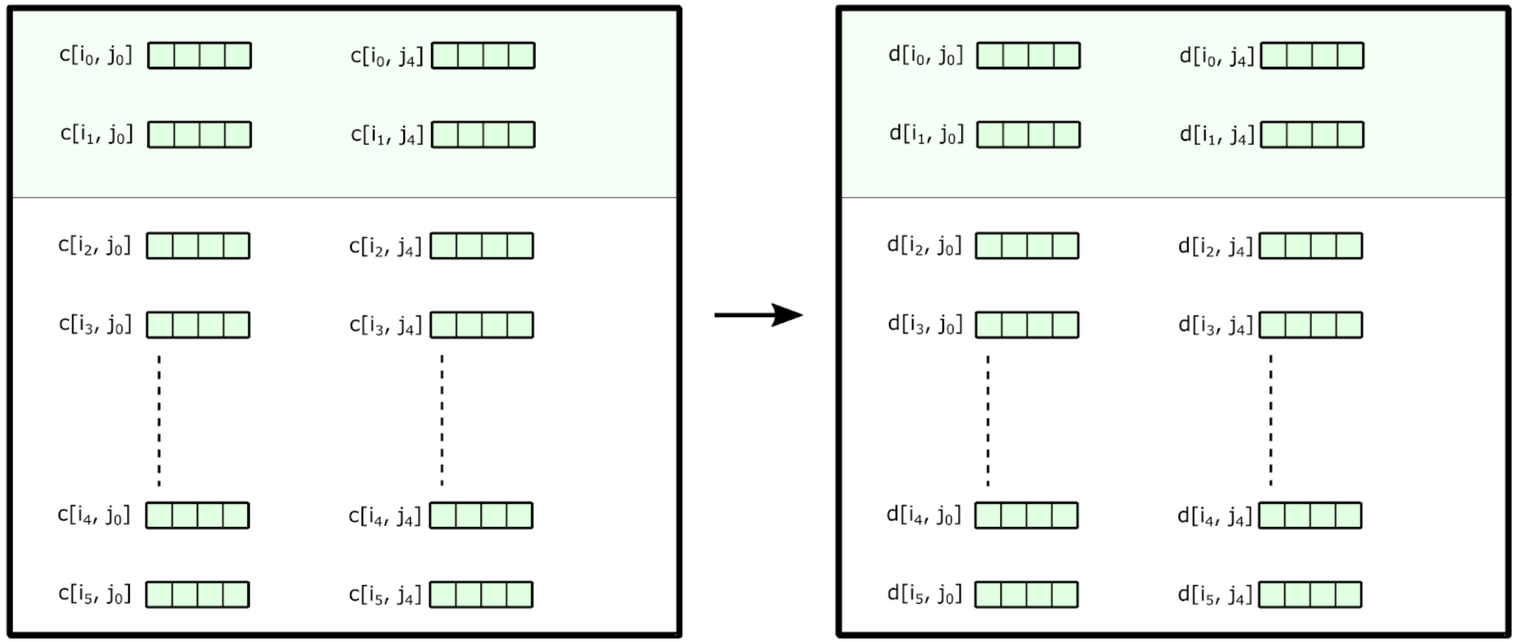
```
//
// Matrix multiply phase
//

MmaMultistage mma_multistage(
  shared_storage.main_loop, thread_idx, warp_idx, lane_idx);

typename MmaMultistage::FragmentC accumulators;

// Execute matrix multiply
mma_multistage(
  gemm_k_iterations,
  accumulators,
  iterator_A,
  iterator_B,
  accumulators);

//
// Epilogue phase
//

EpilogueOutputOp output_op(params.output_op);

// Compute threadblock's location within Global Memory
MatrixCoord threadblock_offset(
  threadblock_tile_offset.m() * MmaMultistage::Shape::kM,
  threadblock_tile_offset.n() * MmaMultistage::Shape::kN
);

// Tile iterator loading from source tensor.
typename Epilogue::OutputTileIterator iterator_C(
  params.params_C,
  ptr_C,
  params.problem_size.mn(),
  thread_idx,
  threadblock_offset
);

Epilogue epilogue(
  shared_storage.epilogue, thread_idx, warp_idx, lane_idx);

// Execute the epilogue phase
epilogue(
  output_op, iterator_C, accumulators, iterator_C);
```



EPILOGUE FUSION DESIGN PATTERNS

ELEMENTWISE OPERATIONS

Fusing operations in the Epilogue



Elementwise operations may be performed on register-backed arrays before storing

Additional tensors may be loaded, subject to register liveness constraints

Vector sizes may be arbitrarily large; this is useful for storing bit-vectors for ReLU calculations (8 x 1b)

Composing GEMM with Epilogue Functor

Define the input, output, and compute types

Define the epilogue functor

Define the device-wide GEMM operator

E.g. GEMM “NN” using Tensor Cores

Construct “Arguments” structure

Problem size, pointers and strides, epilogue functor arguments

Launch the GEMM kernel grid

“Device wide” operator definition and launch in host code

```
using ElementInput = cutlass::half_t;
using ElementOutput = float;
using ElementAccumulator = float;
using ElementEpilogueCompute = float;

using EpilogueFunctor = cutlass::epilogue::thread::LinearCombinationGELU<
    ElementOutput,                     // Output tensor data type
    8,                                 // Epilogue “vector size” — 16 bytes
    ElementAccumulator,                // Accumulator data type — must match mainloop
    ElementEpilogueCompute             // Epilogue “compute type” — type of alpha/beta
>;

using Gemm = cutlass::gemm::device::Gemm<
    ElementInput,  cutlass::layout::ColumnMajor,
    ElementInput,  cutlass::layout::ColumnMajor,
    ElementOutput, cutlass::layout::ColumnMajor,
    ElementAccumulator,
    cutlass::arch::OpClassTensorOp,
    cutlass::arch::Sm80,
    cutlass::gemm::GemmShape<128, 256, 64>,
    cutlass::gemm::GemmShape<64, 64, 64>,
    cutlass::gemm::GemmShape<16, 8, 16>,
    EpilogueFunctor,
    cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<>,
    3
>;

/// Collect arguments
typename Gemm::Arguments arguments{
  problem_size,
  tensor_A.device_ref(),
  tensor_B.device_ref(),
  tensor_C.device_ref(),
  tensor_D.device_ref(),
  {alpha, beta}              // Arguments passed to EpilogueFunctor
};

Gemm gemm_op;

gemm_op(arguments);          // Run the GEMM
```

GEMM + Elementwise example functor: LinearCombinationGELU

"Params" nested structure contains arguments passed from caller

Constructor optionally fetches scalars via device pointers

Function call operator performs elementwise arithmetic and conversion

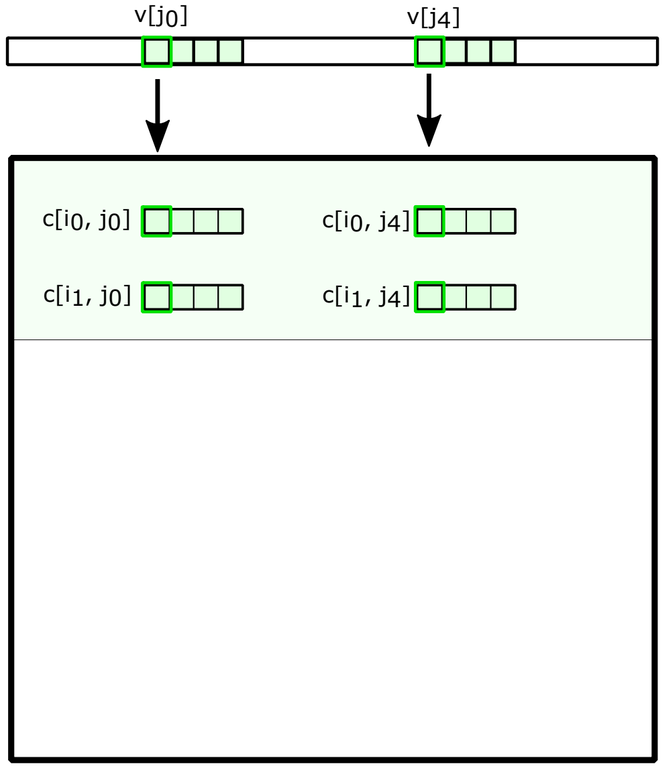
Returned value stored to Global Memory by Epilogue

"Thread-wide" epilogue functor

```
class LinearCombinationGELU {
public:

  /// Host-constructable parameters structure
  struct Params {

    ElementCompute alpha;
    ElementCompute beta;
    ElementCompute const *alpha_ptr;
    ElementCompute const *beta_ptr;
    ...
  };

  ElementCompute alpha_, _beta;

  /// Constructs the function object, possibly loading from pointers in host memory
  CUTLASS_HOST_DEVICE
  LinearCombinationGELU(Params const &params) {

    alpha_ = (params.alpha_ptr ? *params.alpha_ptr : params.alpha);
    beta_ = (params.beta_ptr ? *params.beta_ptr : params.beta);
  }

  /// Computes: D = gelu( alpha * accumulator )
  CUTLASS_HOST_DEVICE
  FragmentOutput operator()(FragmentAccumulator const &accumulator) const {

    // Convert source to interal compute numeric type
    NumericArrayConverter<ElementCompute, ElementAccumulator, kCount, Round>
accumulator_converter;

    ComputeFragment converted_accumulator = accumulator_converter(accumulator);

    // Perform elementwise operations
    ComputeFragment intermediate;

    multiplies<ComputeFragment> mul_add_accumulator;
    GELU_Taylor<ComputeFragment> gelu;

    intermediate = mul_add_accumulator(alpha_, converted_accumulator);    // D = alpha *
Accum
    intermediate = gelu(intermediate);

    // Convert to destination numeric type
    NumericArrayConverter<ElementOutput, ElementCompute, kCount, Round>
destination_converter;

    return destination_converter(intermediate);
```

BROADCASTING VECTORS

Fusing operations in the Epilogue



Vector is broadcast over columns

```
# Matrix product computation
AB[m, n] = sum_k( A[m, k] * B[k, n] )

# Elementwise operations with broadcast
Z[m, n] = OutputOp_0( AB[m, n], C[m, n], V[n] )
T[m, n] = OutputOp_1( AB[m, n], C[m, n], V[n] )
```

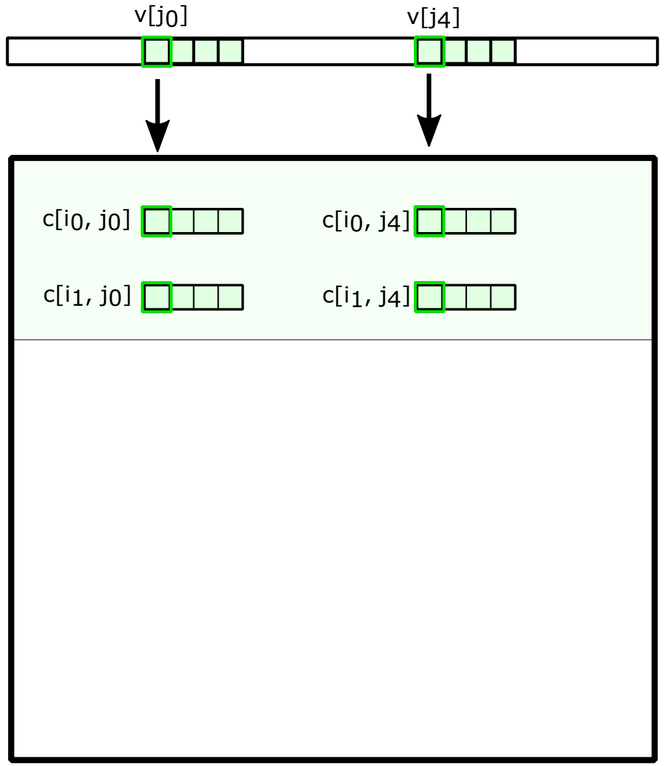
Elementwise operation may output two tensors Z and T

Vectors may be loaded prior to or during the Epilogue steps and broadcast to elementwise operations

See include/cutlass/epilogue/threadblock/epilogue_with_broadcast.h

BROADCASTING VECTORS

Fusing operations in the Epilogue




```
// Epilogue computes: Z, T = OutputOp(AB, C, V)
struct EpilogueWithBroadcast {

  CUTLASS_DEVICE void operator()(
    OutputOp const &output_op,                      ///< Output operator
    ElementVector const * broadcast_ptr,            ///< Broadcast vector "V"
    ...
  ) {

  // Load the broadcast vector at the start of the epilogue
  BroadcastFragment broadcast_fragment;

  load_broadcast_fragment_(
    broadcast_fragment, broadcast_ptr, problem_size, threadblock_offset);

  ...


  // Iterate over accumulator tile
  #pragma unroll
  for (int iter = 0; iter < OutputTileIterator::kIterations; ++iter) {
    ...

    // Perform elementwise operation with broadcast vector
    #pragma unroll
    for (int i = 0; i < kOutputOpIterations; ++i) {

      output_op(
        frag_Z_ptr[i],
        frag_T_ptr[i],
        frag_AB_ptr[i],
        broadcast_fragment[i % ThreadMap::Iterations::kColumn]);
    }

    ...
  }
}
```

Vectors may be loaded prior to or during the Epilogue steps and broadcast to elementwise operations

See [include/cutlass/epilogue/threadblock/epilogue_with_broadcast.h]

GEMM + Broadcast + Elementwise

Define the epilogue functor for fused broadcast + elementwise

Define the device-wide GEMM operator

E.g. GEMM “NN” using Tensor Cores

Construct “Arguments” structure

Additional arguments: pointers, leading dimensions, and batch strides

“Device wide” operator definition and launch in host code

```
// GEMM with fused broadcast operation
//
// Computes the following:
//
//   Z[m, n] = OutputOp0( AB[m, n], C[m, n], V[n] )
//   T[m, n] = OutputOp1( AB[m, n], C[m, n], V[n] )

using EpilogueOutputOp = cutlass::epilogue::thread::LinearCombinationBiasElementwise<
  ElementOutput,
  ElementAccumulator,
  ElementCompute,
  ElementZ,
  ElementT,
  8,
  cutlass::epilogue::thread::GELU_taylor<float>
>;

using GemmKernel =
  typename cutlass::gemm::kernel::DefaultGemmWithBroadcast<
    cutlass::half_t, cutlass::layout::ColumnMajor, cutlass::ComplexTransform::kNone, 8,
    cutlass::half_t, cutlass::layout::ColumnMajor, cutlass::ComplexTransform::kNone, 8,
    cutlass::half_t, cutlass::layout::ColumnMajor,
    float,
    cutlass::arch::OpClassTensorOp,
    cutlass::arch::Sm80,
    cutlass::gemm::GemmShape<128, 128, 32>,
    cutlass::gemm::GemmShape<64, 64, 32>,
    cutlass::gemm::GemmShape<16, 8, 16>,
    EpilogueOutputOp,
    cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<8>,
    3,
    cutlass::arch::OpMultiplyAdd
>::GemmKernel;

using Gemm = cutlass::gemm::device::GemmUniversalAdapter<GemmKernel>;

typename Gemm::Arguments arguments{
  ...
  tensor_Broadcast.device_data(),
  tensor_T.device_data(),
  ...
  tensor_Z.layout().stride(0),
  tensor_T.layout().stride(0),
};
```

GEMM + Broadcast + Elementwise example functor: LinearCombinationBiasElementwise

Functor outputs `frag_Z` and `frag_T`

`LinearCombinationBiasElementwise` is specialized for linear scaling with broadcast

Function call operator performs elementwise arithmetic and conversion

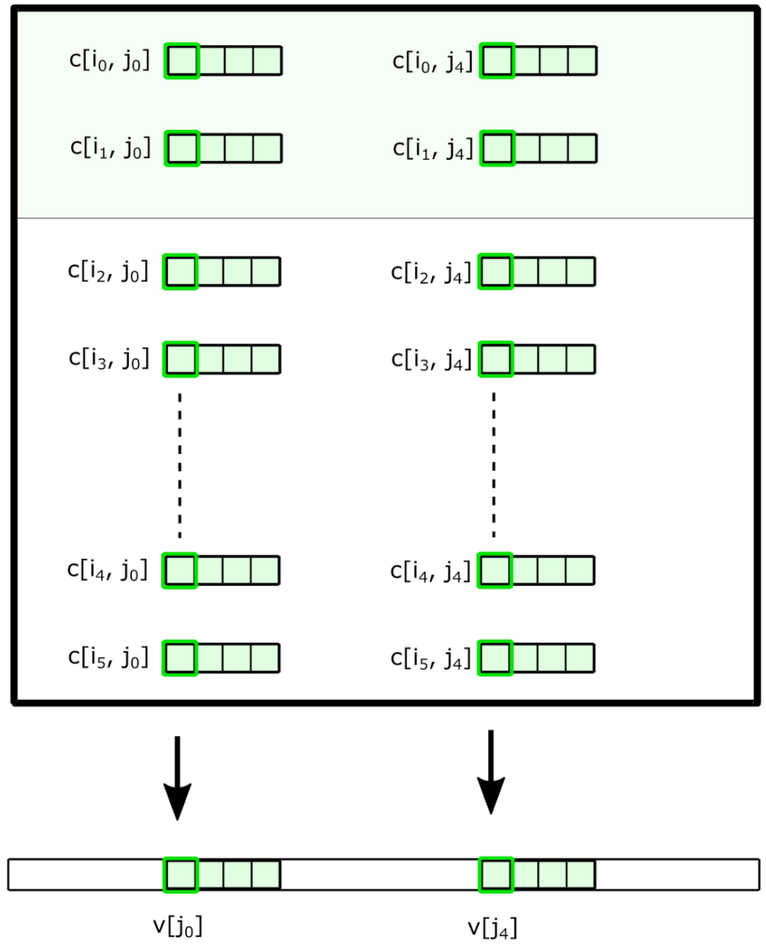
Output arguments are stored to Global Memory by EpilogueWithBroadcast

“Thread-wide” epilogue functor

```
template <
  ...
  typename ElementwiseOp_ = Identity<ElementCompute_>,
  typename BinaryOp_ = plus<ElementCompute_>
>
class LinearCombinationBiasElementwise {
public:

  /// Applies the operation when is_source_needed() is false
  CUTLASS_HOST_DEVICE
  void operator()(
    FragmentZ &frag_Z,
    FragmentT &frag_T,
    FragmentAccumulator const &AB,
    FragmentCompute const &V) const {

    ElementwiseOp elementwise_op;
    BinaryOp binary_op;

    NumericArrayConverter<
      ElementCompute, ElementAccumulator, kElementsPerAccess> accumulator_converter;

    FragmentCompute tmp_Accum = accumulator_converter(AB);
    FragmentCompute result_Z;
    FragmentCompute result_T;

    CUTLASS_PRAGMA_UNROLL
    for (int i = 0; i < kElementsPerAccess; ++i) {
      ElementCompute z = binary_op(alpha_ * tmp_Accum[i], V[i]);
      result_Z[i] = z;
      result_T[i] = elementwise_op(z);
    }

    NumericArrayConverter<ElementZ, ElementCompute, kElementsPerAccess> convert_z;
    frag_Z = convert_z(result_Z);

    NumericArrayConverter<ElementT, ElementCompute, kElementsPerAccess> convert_t;
    frag_T = convert_t(result_T);
  }
};
```

REDUCTIONS OVER GEMM OUTPUT

Fusing operations in the Epilogue



Columns reduced by custom reduction operator

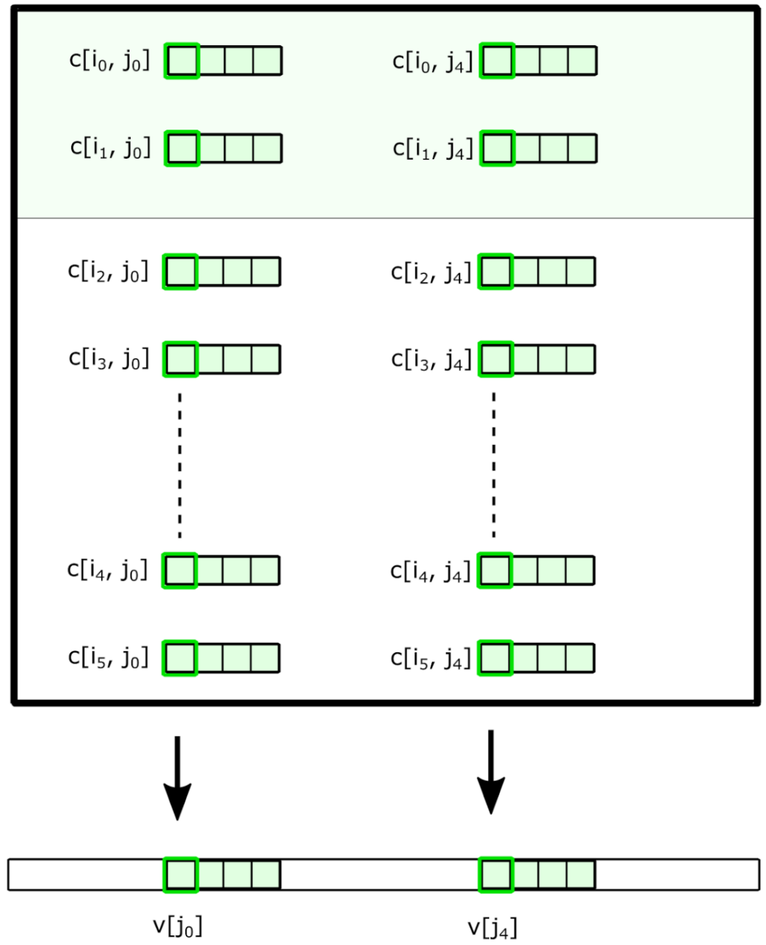
```
# Matrix product computation
AB[m, n] = sum_k( A[m, k] * B[k, n] )

# Elementwise operation
Z[m, n] = OutputOp( AB[m, n], C[m, n], T[m, n] )

# Reduction operation
V[n]    = Reduce_m( Z[m, n], identity_element )
```

Reductions over Threadblock tile may be fused into Epilogue

See [include/cutlass/epilogue/threadblock/epilogue_with_reduction.h]

REDUCTIONS OVER GEMM OUTPUT

Fusing operations in the Epilogue




```
// Epilogue with reduction
struct EpilogueWithReduction {

  CUTLASS_DEVICE void operator()(
    OutputOp const &output_op,                      ///< Output operator
    ElementVector * reduction_output_ptr,           ///< Reduction output vector "T"
    ...
  ) {

    ReductionFragment reduction_fragment;
    reduction_fragment.clear();

    // Iterate over accumulator tile
    for (int iter = 0; iter < OutputTileIterator::kIterations; ++iter) {
      ...

      // Elementwise operation
      for (int i = 0; i < kOutputOpIterations; ++i) {
        compute_fragment[i] = output_op(accumulator_fragment[i]);
      }

      // Partial reduction within a thread
      for (int column = 0; column < ReductionDetail::kColumnsPerThread; ++column) {
        for (int row = 0; row < ReductionDetail::kRowsPerThread; ++row) {

          reduction_fragment[column] = reduction_op(
            reduction_fragment[column],
            compute_fragment[row * ReductionDetail::kColumnsPerThread + column]);
        }
      }

      ...
    }

    __syncthreads();

    // Data exchange through Shared Memory, serial reduction, and store to Global
Memory
    reduction_(problem_size, threadblock_offset, reduction_output_ptr,
reduction_fragment);
  }
};
```

Reductions over Threadblock tile may be fused into Epilogue

See [include/cutlass/epilogue/threadblock/epilogue_with_reduction.h]

GEMM + Elementwise + Reduction

Define the elementwise functor

Define the reduction functor

Define the device-wide GEMM operator
E.g. GEMM “NN” using Tensor Cores

Specify elementwise and reduction functors

Reduction is performed over one threadblock
Dependent kernel launch needed for final reduction

“Device wide” operator definition and launch in host code

```
// GEMM with fused partial reduction
//
// Computes the following:
//
//   Z[m, n] = OutputOp( AB[m, n], C[m, n], T[m, n] )
//
//   V[n]    = Reduce_m(  Z[m, n], identity_element )

using EpilogueOutputOp = cutlass::epilogue::thread::LinearCombinationDReluConditionalBits<
  ElementCompute,
  ElementAccumulator,
  ElementOutput,
  8
>;

using ReductionOp = cutlass::plus<float>;


using GemmKernel =
  typename cutlass::gemm::kernel::DefaultGemmWithReduction<
    cutlass::half_t, cutlass::layout::ColumnMajor, cutlass::ComplexTransform::kNone, 8,
    cutlass::half_t, cutlass::layout::ColumnMajor, cutlass::ComplexTransform::kNone, 8,
    cutlass::half_t, cutlass::layout::ColumnMajor,
    float,
    cutlass::arch::OpClassTensorOp,
    cutlass::arch::Sm80,
    cutlass::gemm::GemmShape<128, 128, 32>,
    cutlass::gemm::GemmShape<64, 64, 32>,
    cutlass::gemm::GemmShape<16, 8, 16>,

    EpilogueOutputOp,                              // Elementwise functor

    ReductionOp,                                   // Reduction functor

    cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<8>,
    5,
    cutlass::arch::OpMultiplyAdd
>::GemmKernel;

using Gemm = cutlass::gemm::device::GemmUniversalAdapter<GemmKernel>;
```

GEMM + Elementwise + Reduction example functor: LinearCombinationDReluConditionalBits

Functor outputs one fragment prior to reduction

`LinearCombinationDReluConditionalBits` accepts an optional tensor of 1b condition bits produced by the forward pass

Condition bits are unpacked into `bool` elements and used to conditionally output *dy*=0.

Returned value `*dy*` stored to Global Memory by EpilogueWithReduction

“Thread-wide” epilogue functor

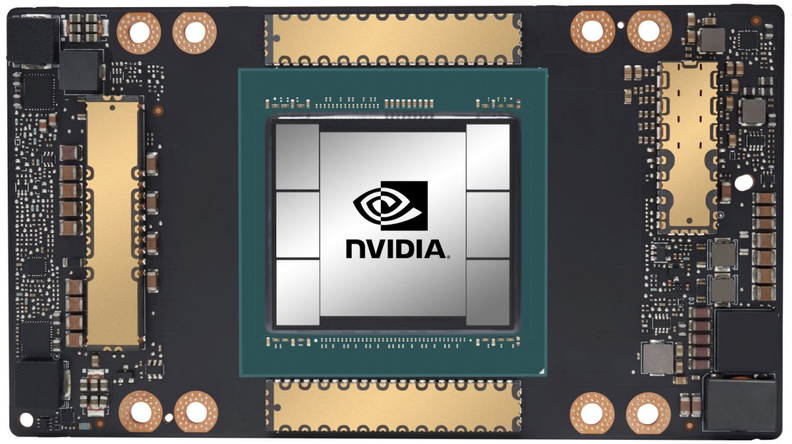
```
class LinearCombinationDReluConditionalBits {
public:

  /// Computes linear scaling: D = alpha * accumulator
  CUTLASS_HOST_DEVICE
  FragmentCompute operator()(
    FragmentAccumulator const &accumulator,
    FragmentTensor const &tensor) const {

    // Convert source to internal compute numeric type
    NumericArrayConverter<
      ElementCompute, ElementAccumulator, kCount, Round> accumulator_converter;

    FragmentCompute converted_accumulator = accumulator_converter(accumulator);

    // Perform binary operations
    multiplies<FragmentCompute> mul_accumulator;

    // dy = alpha * Accum
    FragmentCompute dy = mul_accumulator(alpha_, converted_accumulator);

    // Obtain from packed bits
    bool conditions[kCount];

    UnpackPredicates<kCount> unpack_predicates;

    unpack_predicates(conditions, tensor);

    // dReLU = (cond ? dy : 0)
    CUTLASS_PRAGMA_UNROLL
    for (int i = 0; i < kCount; ++i) {
      if (!conditions[i]) {
        dy[i] = ElementCompute();
      }
    }

    return dy;
  }
};
```

CONCLUSION

CUTLASS PROVIDES REUSABLE TEMPLATES FOR FUSING WITH GEMM AND CONVOLUTION

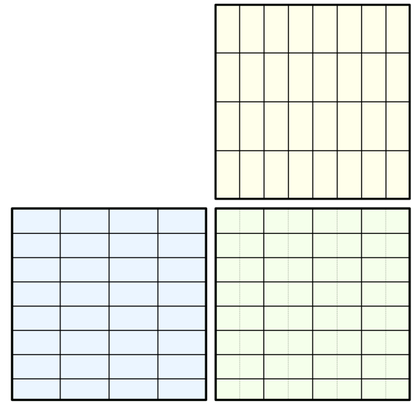
CUTLASS

- CUTLASS provides tuned implementations of GEMM and CONV
- Extensible via custom functors on input and output
- Client defines functors for: elementwise, broadcast, reduction
- Capable of near-peak performance with CUDA 11.5 Toolkit



CUTLASS 2.8: November 2021

- Block-resident CONV-CONV fused kernels
- Emulated single-precision with TensorFloat32
- Implicit GEMM convolution fusion on input
- Grouped GEMM implementation

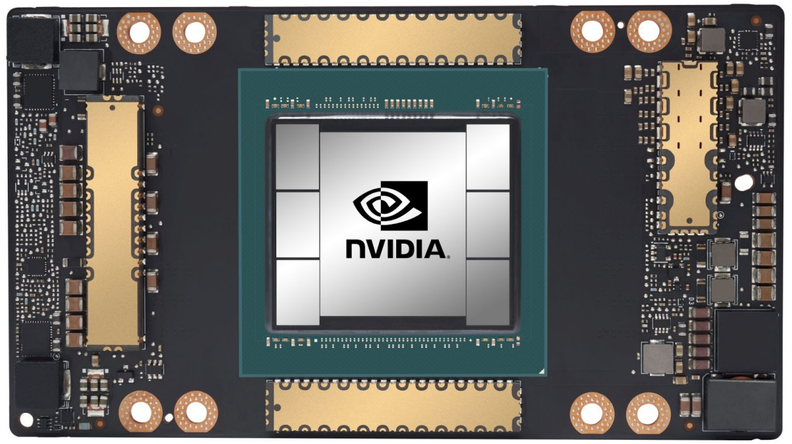


Try it out! https://github.com/NVIDIA/cutlass

REFERENCES

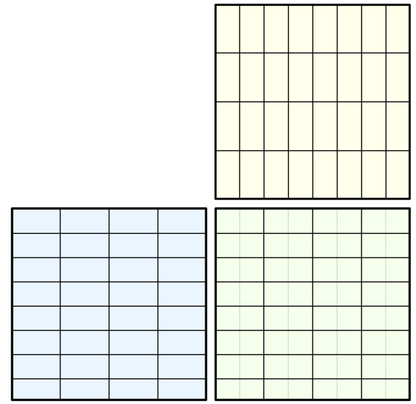
NVIDIA Ampere Architecture:

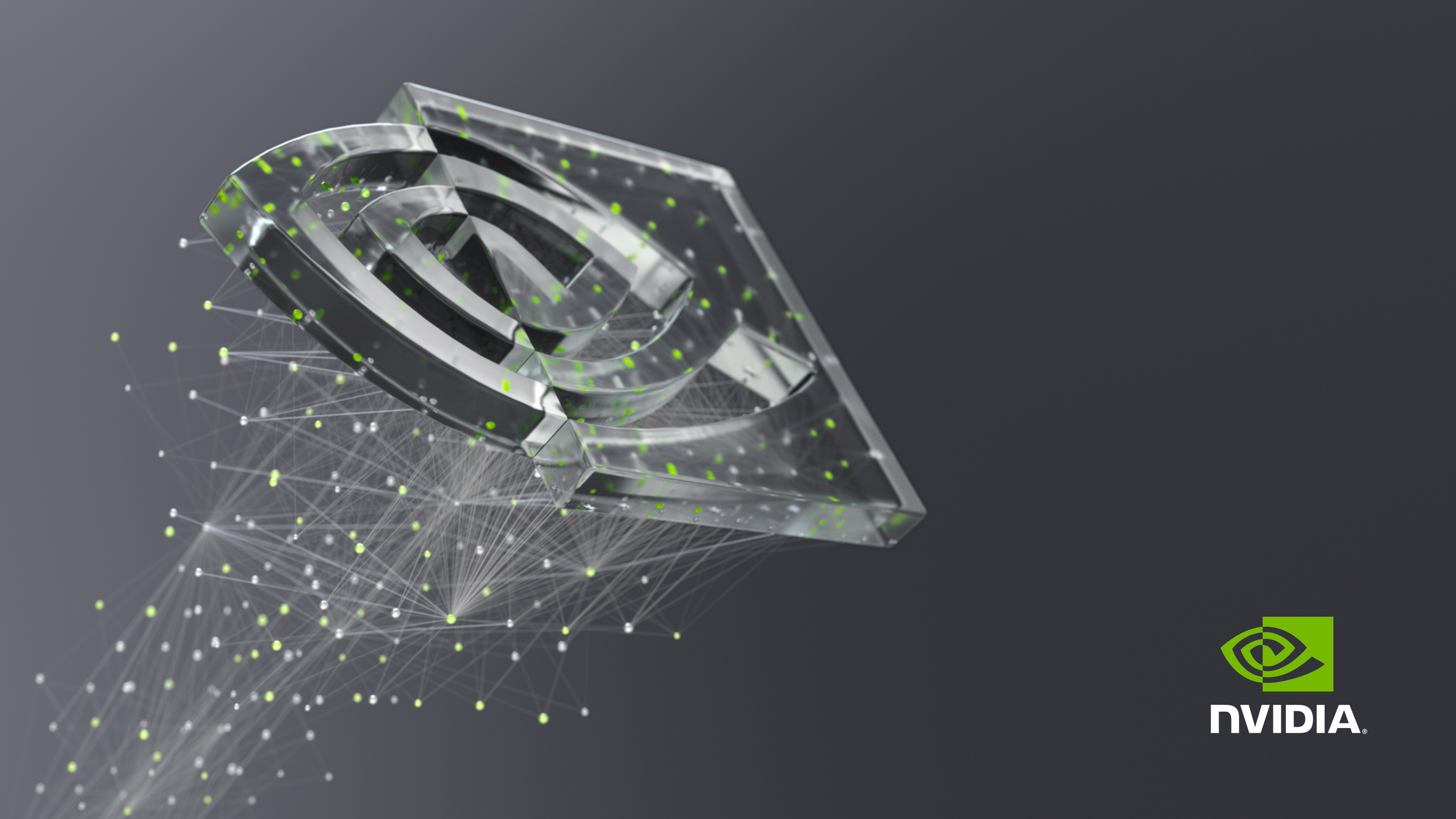
“Inside the NVIDIA Ampere Architecture” (GTC 2020 – S21730)
“NVIDIA Ampere Architecture In-Depth” (blog post)
“CUDA New Features and Beyond” (GTC 2020 – S21760)
“Tensor Core Performance on NVIDIA GPUs” (GTC 2020 – S21929)
“Inside the Compilers, Libraries and Tools for Accelerated Computing” (GTC 2020 - S21766)

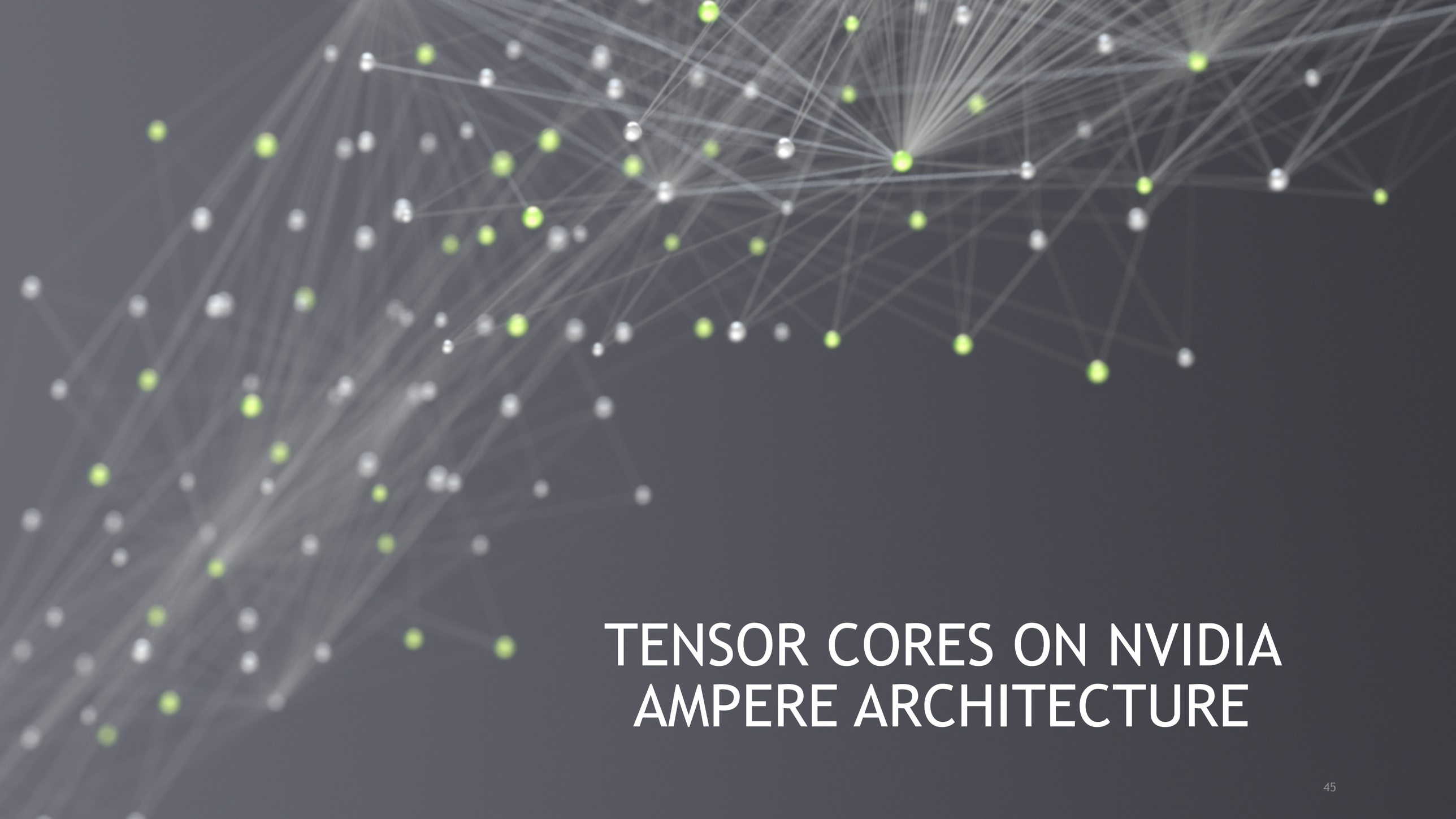


CUTLASS

https://github.com/NVIDIA/cutlass (open source software, New BSD license)
CUTLASS Parallel For All blog post
GTC 2018 talk (S8854) : CUTLASS: Software primitives for dense linear algebra at all levels and scales within CUDA
GTC 2019 talk (S9593) : cuTENSOR:High-performance Tensor Operations in CUDA (joint talk with cuTENSOR)
GTC 2020 talk (S21745) : Developing CUDA kernels to push Tensor Cores to the Absolute Limit on NVIDIA A100
GTC 2021 talk (S31883) : Accelerating Convolution with Tensor Cores in CUTLASS




NVIDIA



TENSOR CORES ON NVIDIA AMPERE ARCHITECTURE

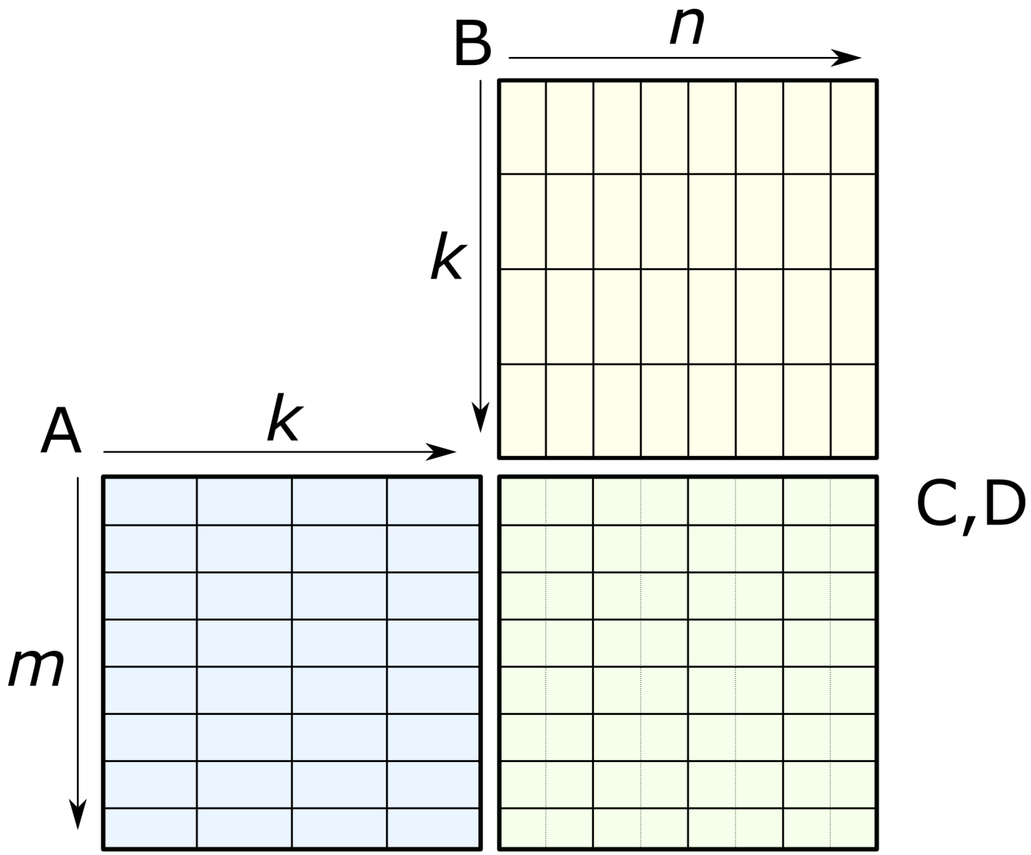
WHAT ARE TENSOR CORES?

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

- Matrix multiply-add
- XOR-POPC

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

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



NVIDIA AMPERE ARCHITECTURE - TENSOR CORE OPERATIONS

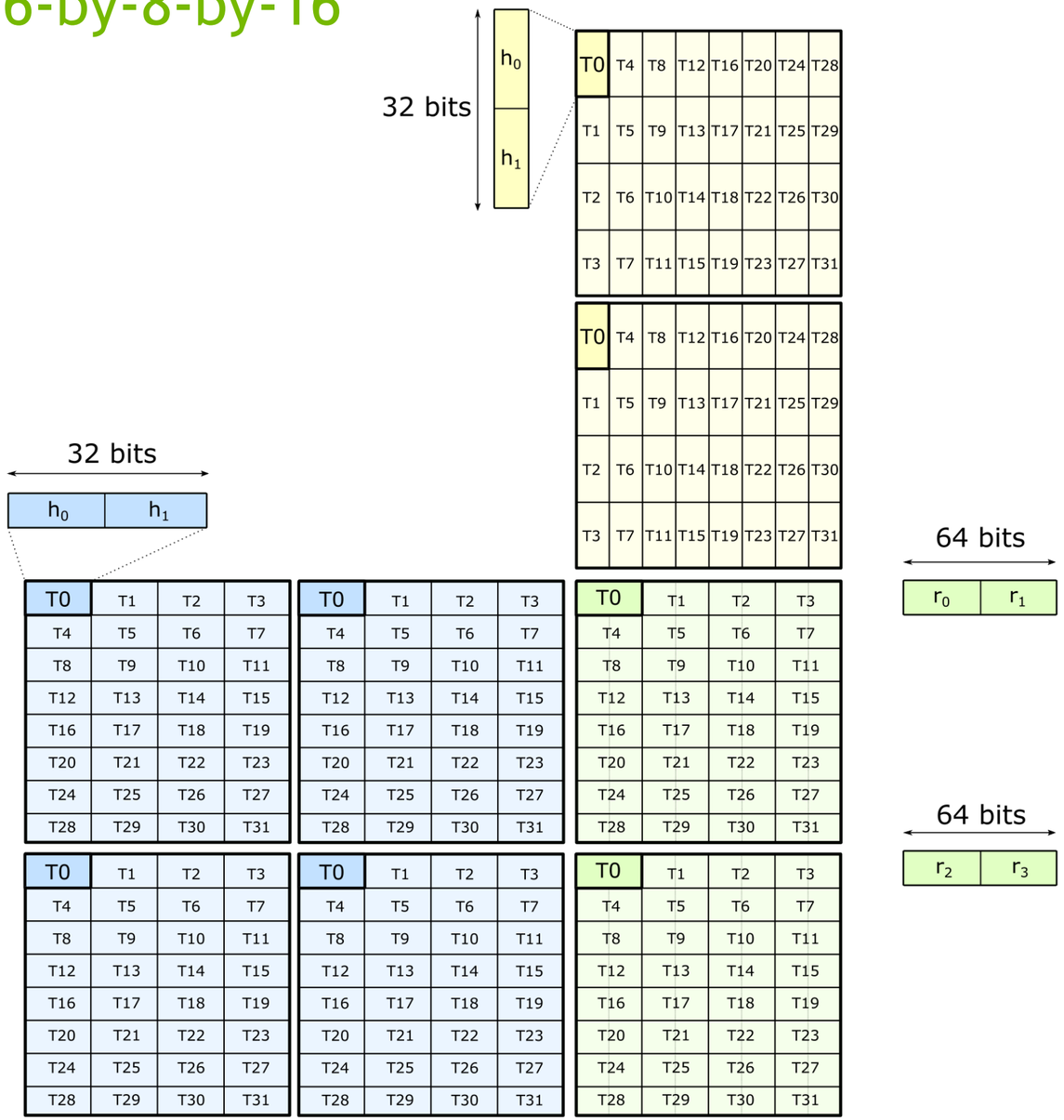
PTX	Data Types (A * B + C)	Shape	Speedup on NVIDIA A100 (vs F32 CUDA cores)	Speedup on Turing* (vs F32 Cores)	Speedup on Volta* (vs F32 Cores)
mma.sync.m16n8k16 mma.sync.m16n8k8	F16 * F16 + F16 F16 * F16 + F32 BF16 * BF16 + F32	16-by-8-by-16 16-by-8-by-8	16x	8x	8x
mma.sync.m16n8k8	TF32 * TF32 + F32	16-by-8-by-8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

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

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

F16 * F16 + F32

16-by-8-by-16



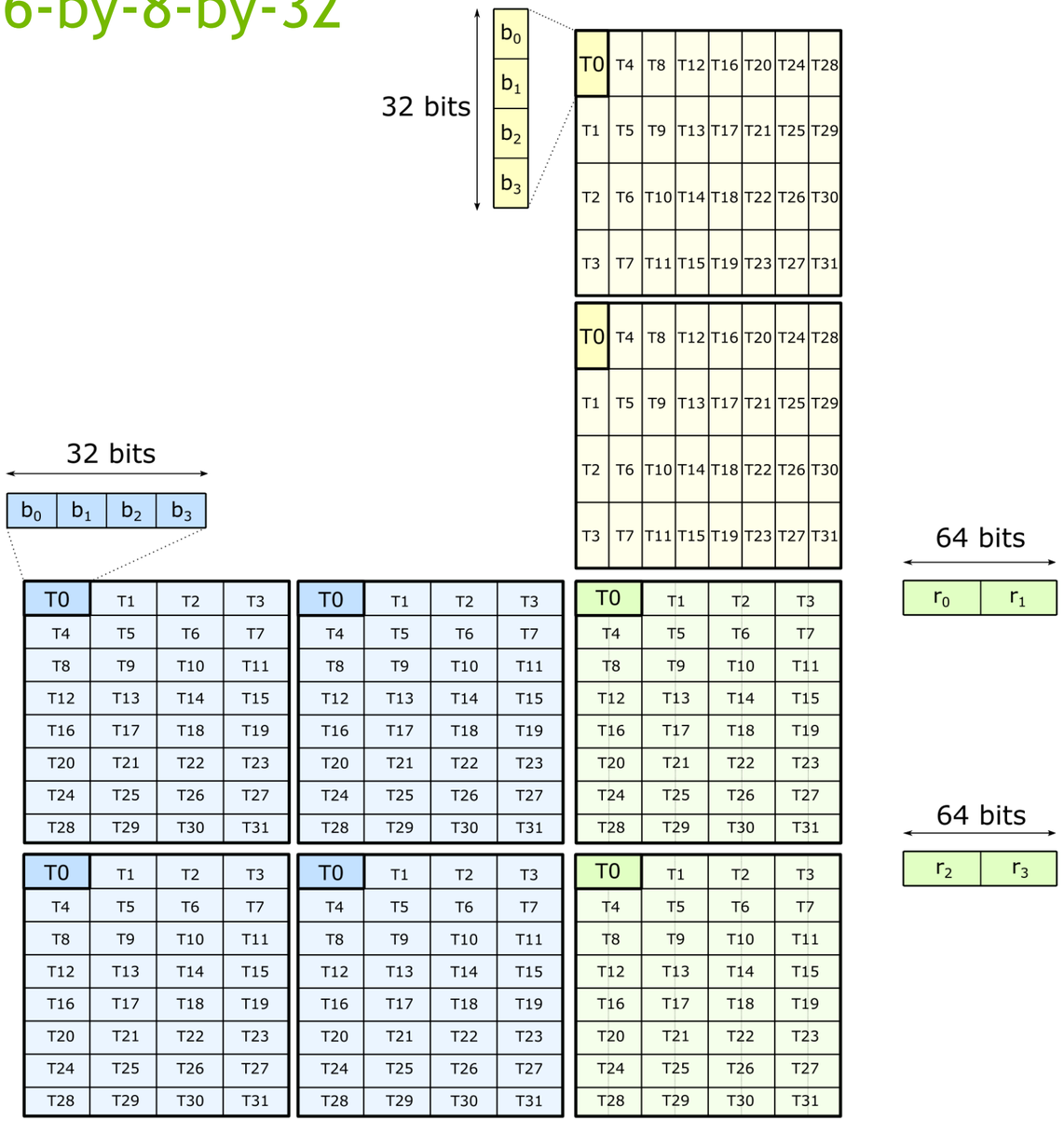
mma.sync.aligned
(via inline PTX)

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


// Example targets 16-by-8-by-32 Tensor Core operation

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

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

S8 * S8 + S32

16-by-8-by-32



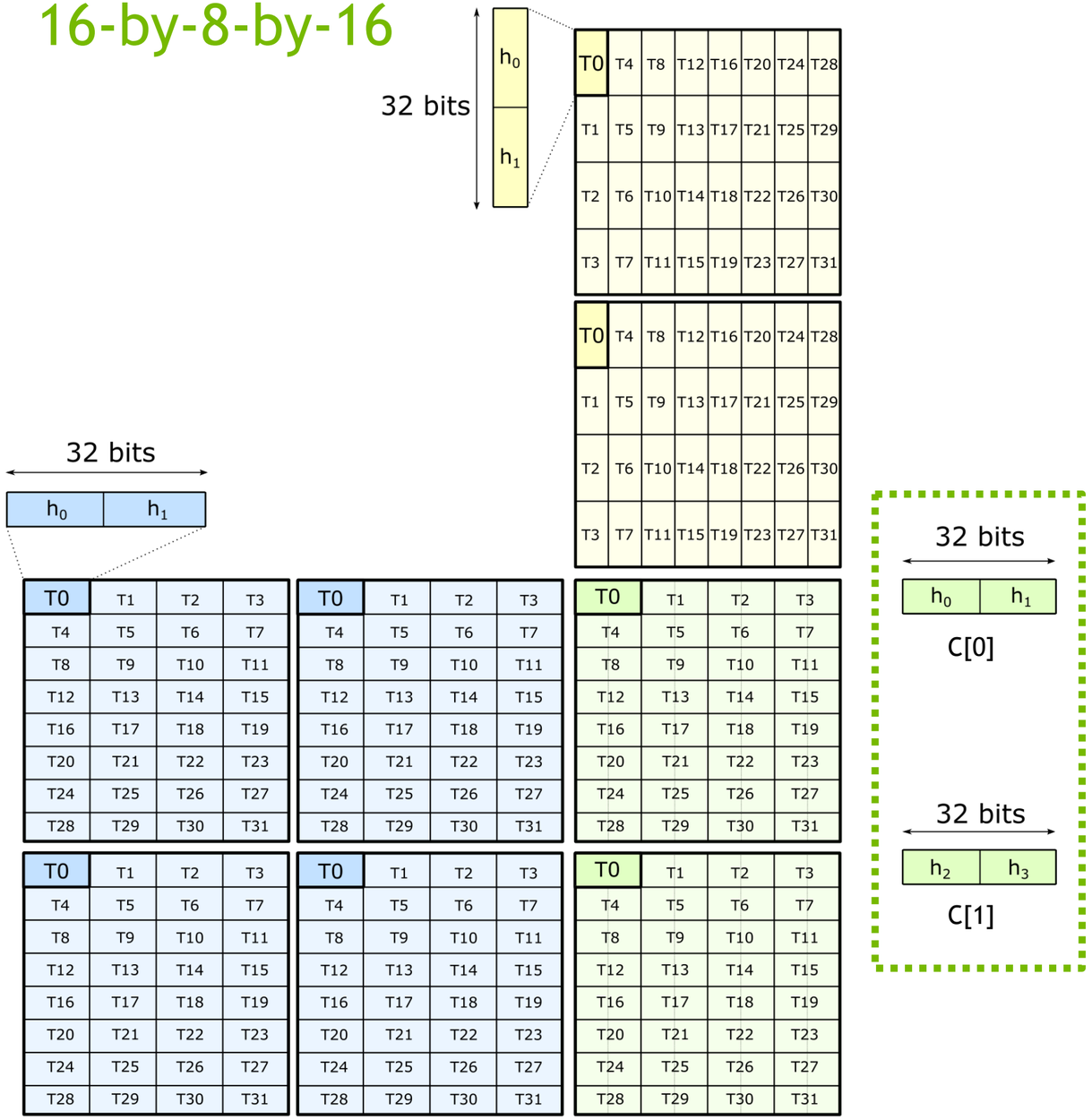
mma.sync.aligned
(via inline PTX)

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


// Example targets 16-by-8-by-32 Tensor Core operation

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

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

HALF-PRECISION : F16 * F16 + F16

16-by-8-by-16



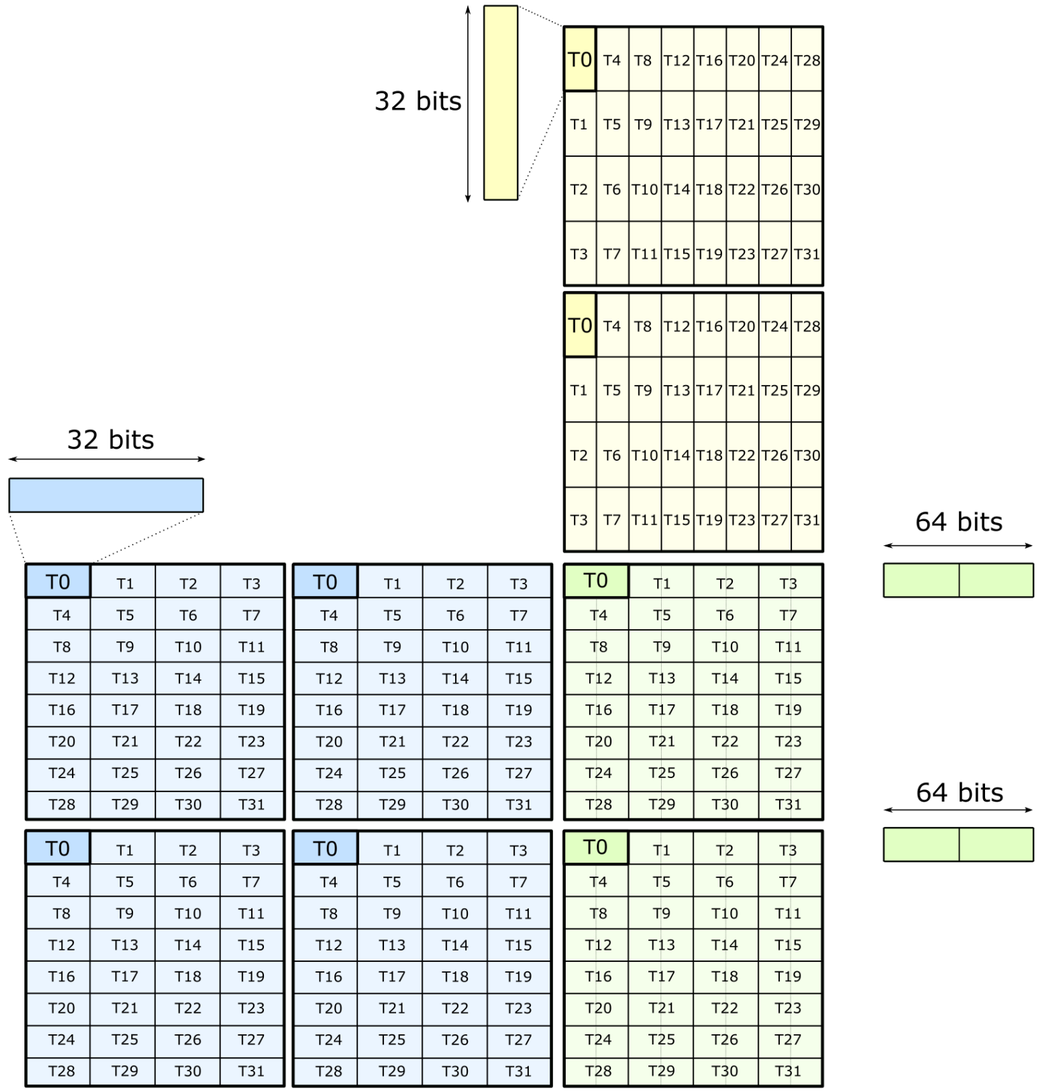
mma.sync.aligned
(via inline PTX)

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


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

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

CUTLASS: wraps PTX in template

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



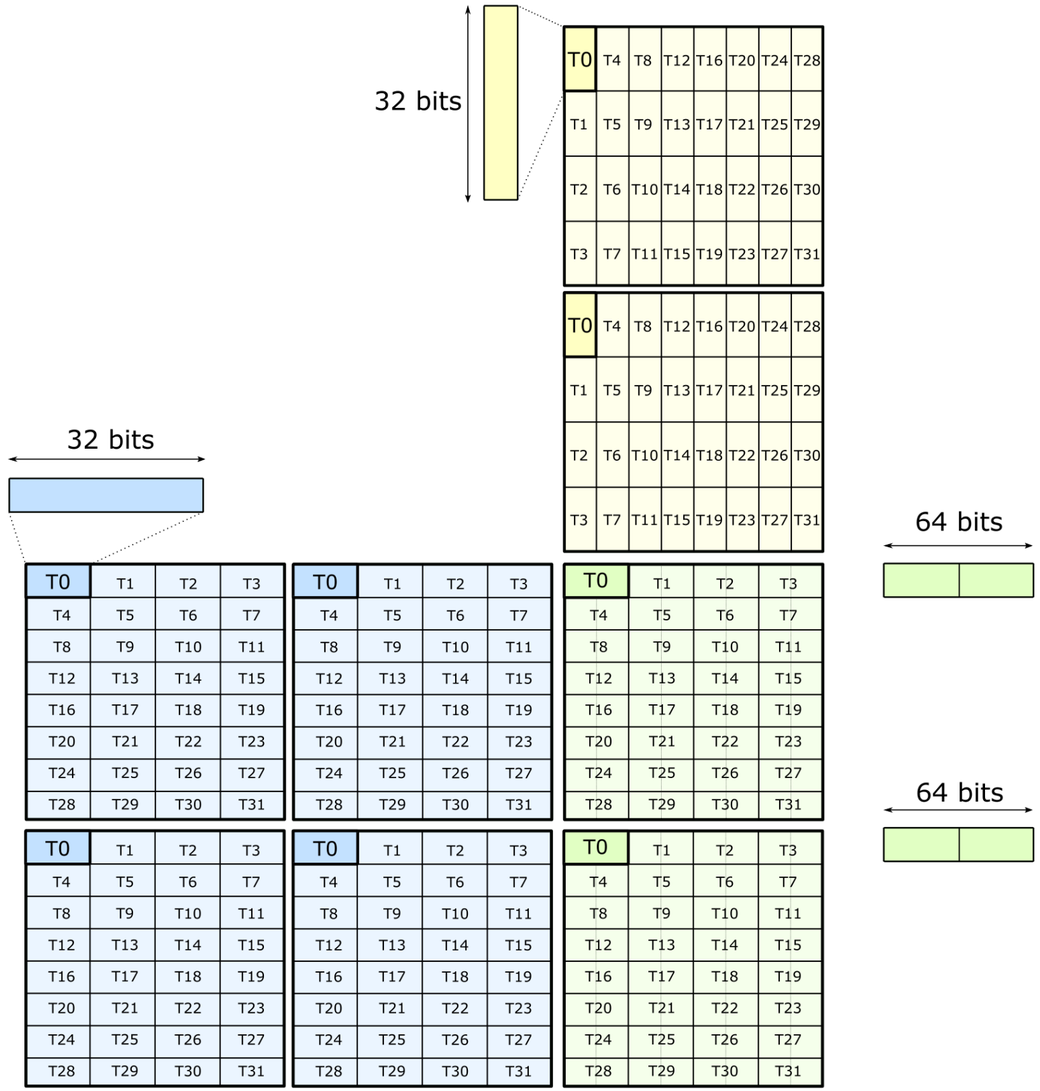
cutlass::arch::Mma

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

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

CUTLASS: wraps PTX in template

16-by-8-by-16



cutlass::arch::Mma

```
__global__ void kernel() {

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

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

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

  ...
}
```

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