Speaking Tensor Cores

Vijay Thakkar & Pradeep Ramani
(Representing the CUTLASS Team @NVIDIA)

June 7th, 2024

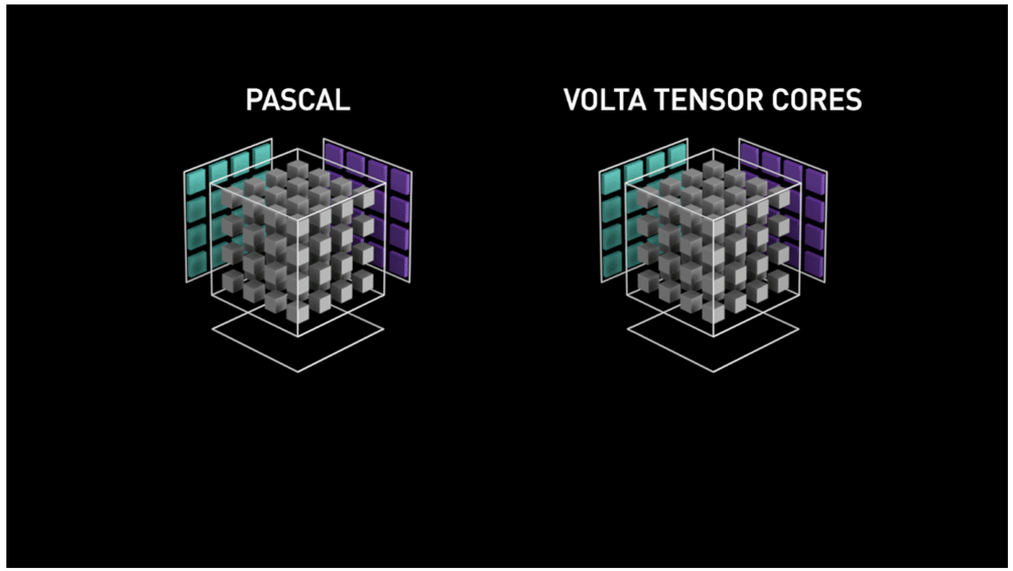
Agenda

- Why Tensor Cores? What make them hard?
- CUTLASS 3.x
- Writing custom kernels with CUTLASS
- Programming Model
- Anatomy of a high performance GEMM
- Demo of a 100 line Hopper GEMM
- Q&A

What are Tensor Cores?

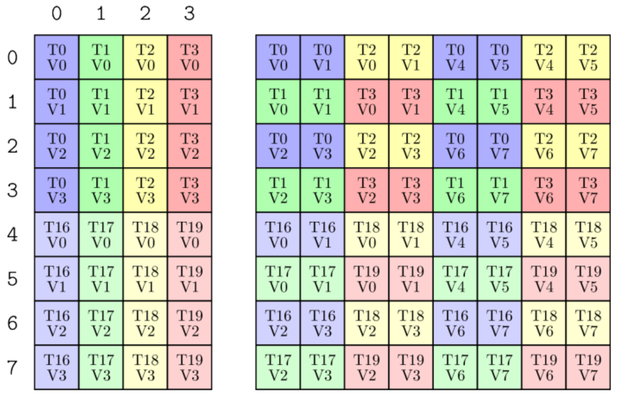
What is a “Tensor Core”

- Hardware block that accelerates MatMul
- FMAs accelerate vector dot products - O(N) operations
- Tensor cores accelerate matrix multiplies - $O(N^3)$ operations
- Increases flop / byte ratio – more temporal and spatial reuse



Tensor Cores

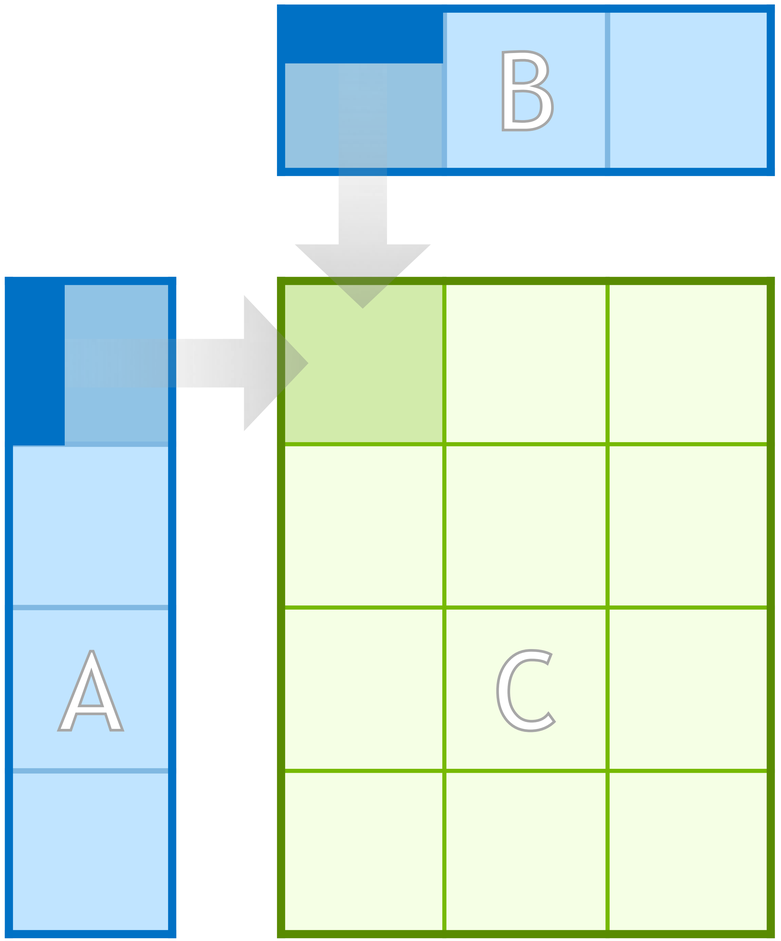
MMA instruction evolution over the years



What’s so hard about MatMuls?

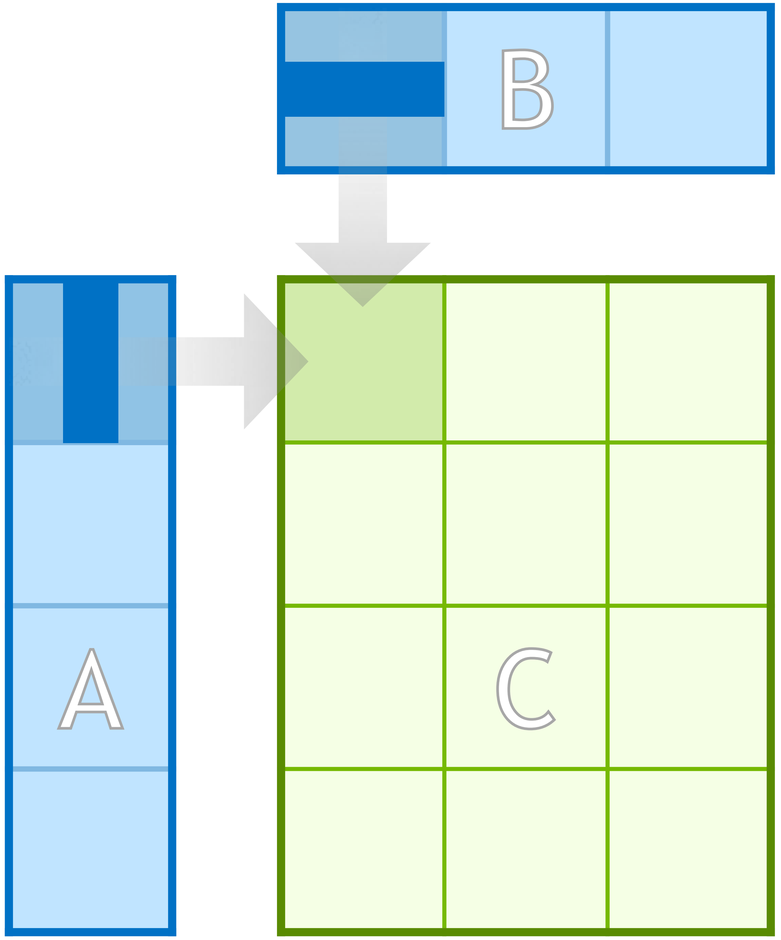
How We Think About Linear Algebra

Accumulated Inner Product of Outer Products



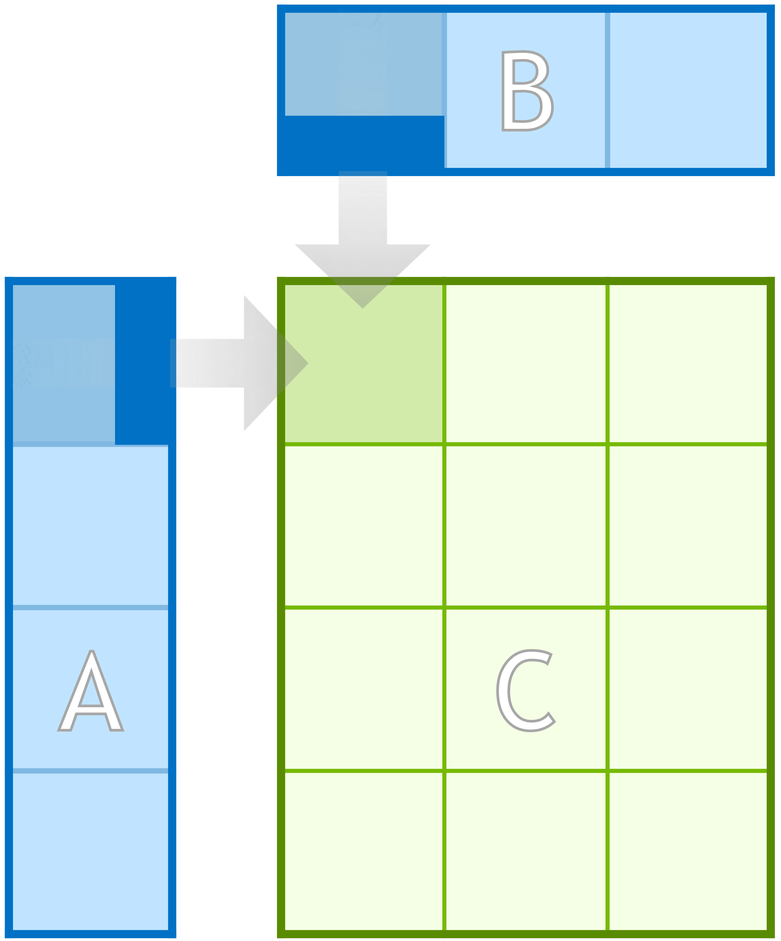
How We Think About Linear Algebra

Accumulated Inner Product of Outer Products



How We Think About Linear Algebra

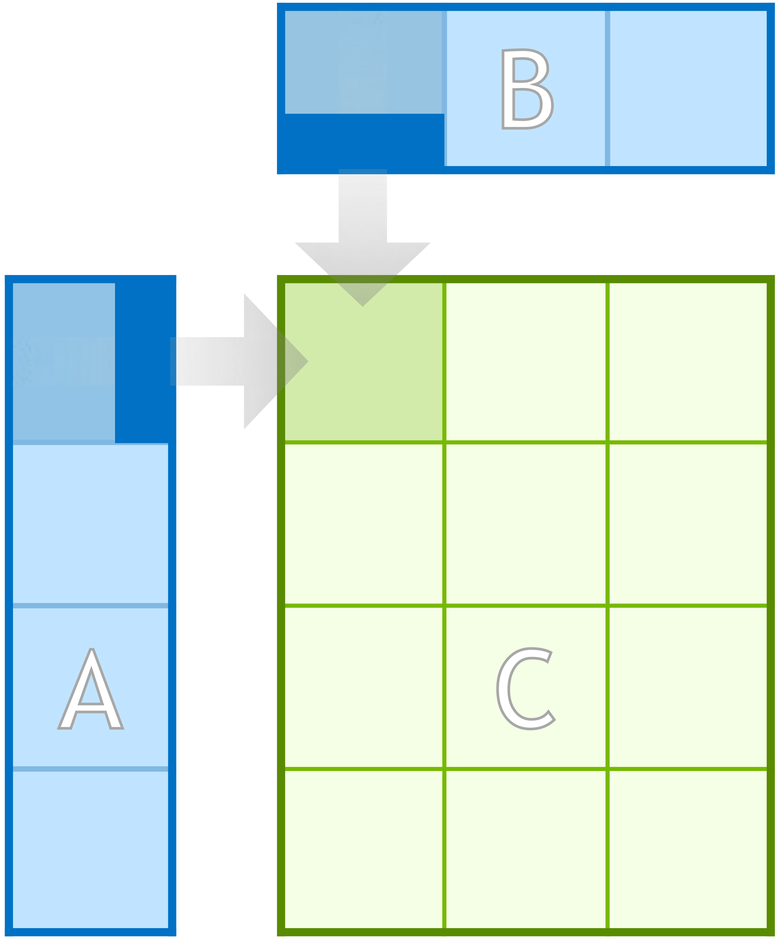
Accumulated Inner Product of Outer Products



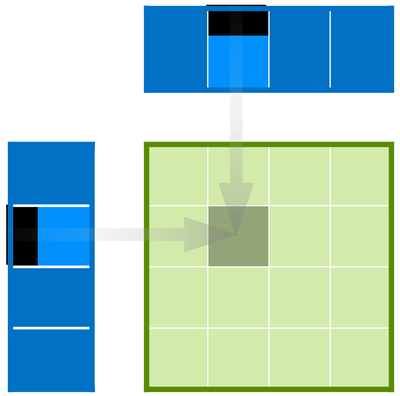

```
for (int m = 0; m < M; ++m) {
  for (int n = 0; n < N; ++n) {
    for (int k = 0; k < K; ++k) {

      C(m,n) += A(m,k) * B(n,k);

    }  // end k
  }  // end n
}  // end m
```

How We Think About Linear Algebra

Parallelism and locality are hierarchical …

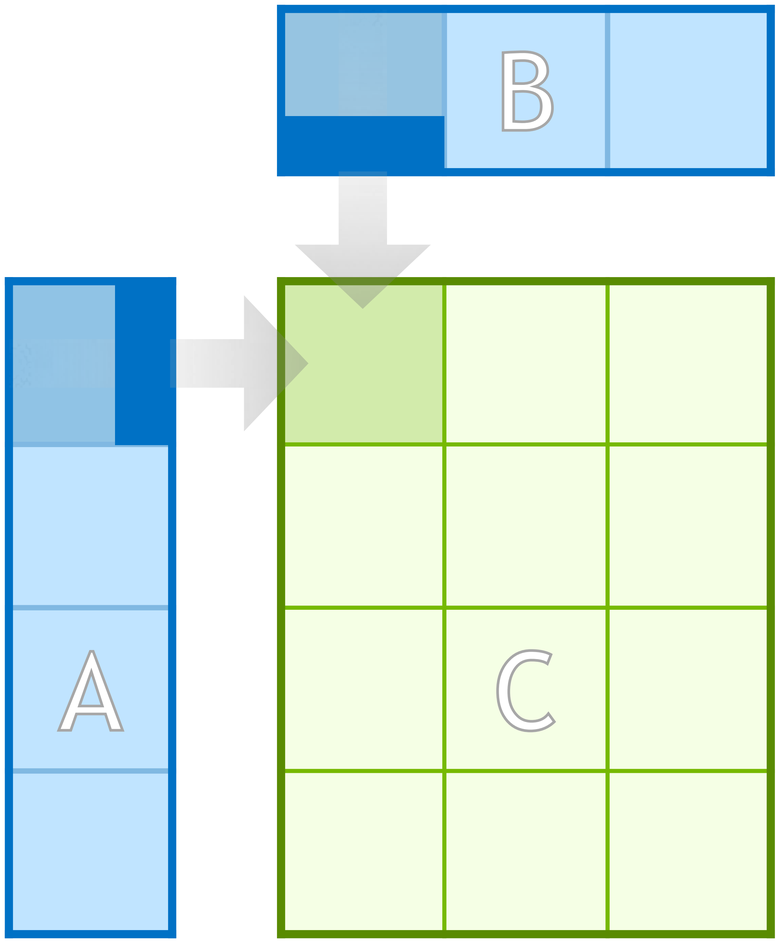



```
for (int cta_n = 0; cta_n < GemmN; cta_n += CtaN) {
  for (int cta_m = 0; cta_m < GemmM; cta_m += CtaM) {
    for (int cta_k = 0; cta_k < GemmK; cta_k += CtaK) {

      // Copy from global to shared

      for (int warp_n = 0; warp_n < CtaN; warp_n += WarpN) {
        for (int warp_m = 0; warp_m < CtaM; warp_m += WarpM) {
          for (int warp_k = 0; warp_k < CtaK; warp_k += WarpK) {

            gemm(a, b, c);
```

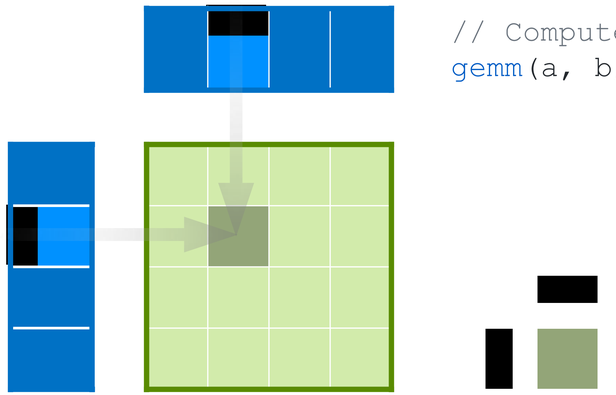


How We Think About Linear Algebra

… often with multiple nested hierarchies of both threads and data

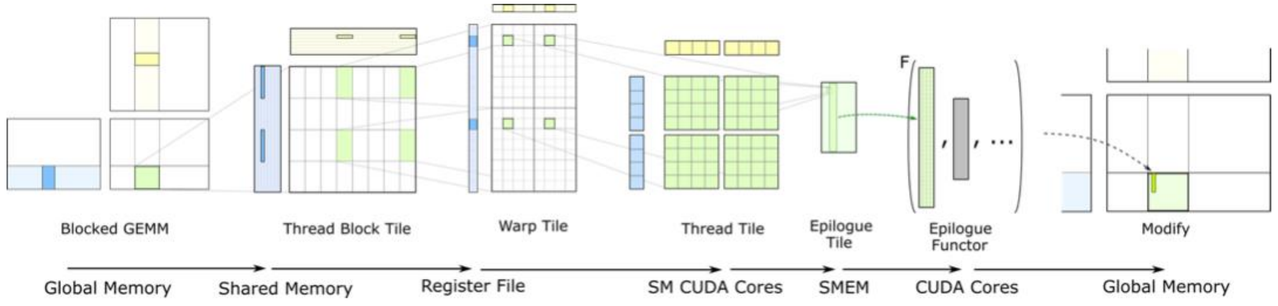



```
for (int cta_n = 0; cta_n < GemmN; cta_n += CtaN) {
  for (int cta_m = 0; cta_m < GemmM; cta_m += CtaM) {
    for (int cta_k = 0; cta_k < GemmK; cta_k += CtaK) {

      // Copy from global to shared

      for (int warp_n = 0; warp_n < CtaN; warp_n += WarpN) {
        for (int warp_m = 0; warp_m < CtaM; warp_m += WarpM) {
          for (int warp_k = 0; warp_k < CtaK; warp_k += WarpK) {

            // Copy from shared to registers

            for (int mma_k = 0; mma_k < WarpK; mma_k += MmaK) {
              for (int mma_n = 0; mma_n < WarpN; mma_n += MmaN) {
                for (int mma_m = 0; mma_m < WarpM; mma_m += MmaM) {

                  // Compute in registers
                  gemm(a, b, c);
```



Architectural Complications …

Push us away from coordinates to index bookkeeping




```
for (int cta_n = 0; cta_n < GemmN; cta_n += CtaN) {
  for (int cta_m = 0; cta_m < GemmM; cta_m += CtaM) {
    for (int cta_k = 0; cta_k < GemmK; cta_k += CtaK) {

      // Copy from global to shared

      for (int warp_n = 0; warp_n < CtaN; warp_n += WarpN) {
        for (int warp_m = 0; warp_m < CtaM; warp_m += WarpM) {
          for (int warp_k = 0; warp_k < CtaK; warp_k += WarpK) {

            // Copy from shared to registers

            for (int mma_k = 0; mma_k < WarpK; mma_k += MmaK) {
              for (int mma_n = 0; mma_n < WarpN; mma_n += MmaN) {
                for (int mma_m = 0; mma_m < WarpM; mma_m += MmaM)

                  // Compute in registers
                  gemm(a, b, c);
```

Figure 3b: Maxwell thread assignments for LDS.U matching

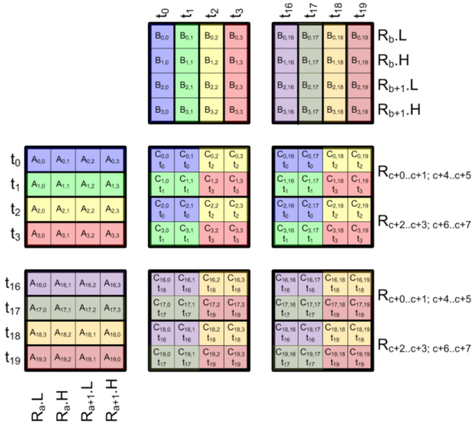
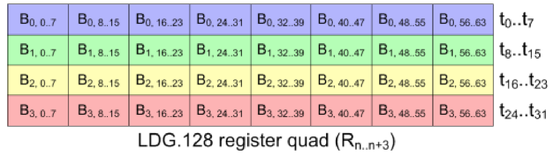
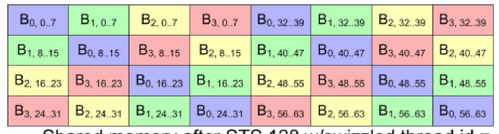


Figure 1b: HMMA.884.F32.TN data layout.
Again, ignore the apparent isomorphism
between thread numbers and matrix indices.



LDG.128 register quad ($R_{n..n+3}$)



Shared memory after STS.128 w/swizzled thread id =
(tid[1:0] << 3) | (tid & 4) | (tid[4:3] ^ tid[1:0])

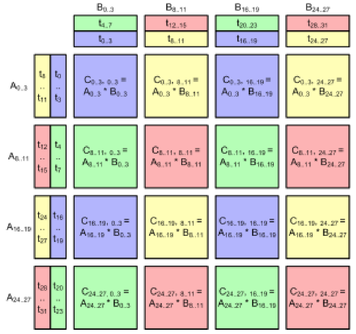
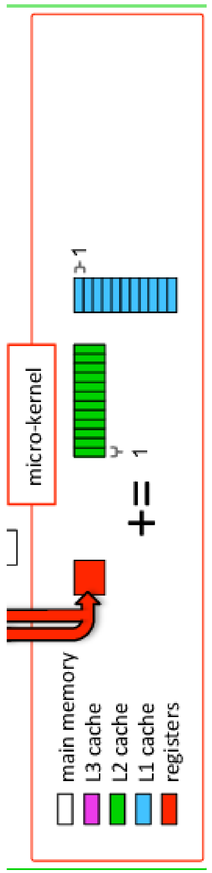


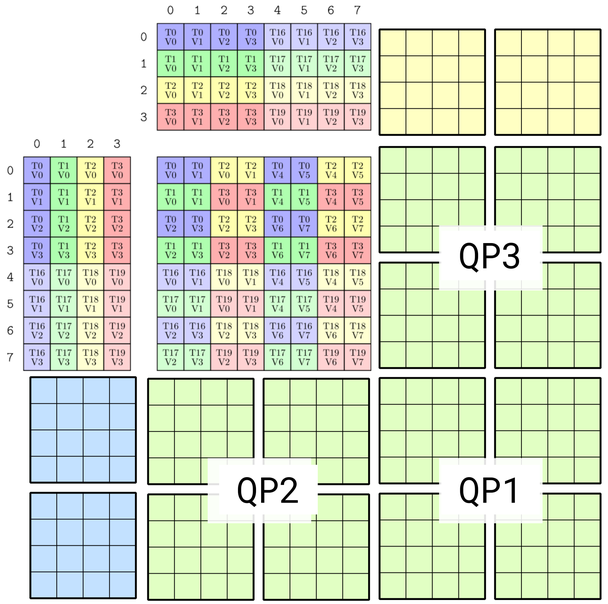
Figure 2a: Composing sparse A[16,4] * B[4, 16] = C[16, 16]
warp-wide multiply from four A[8, 4] * B[4, 8] = C[8, 8] 8-thread multiplies.

No Trivial Way To Express CPU like SIMD Loop on GPUs



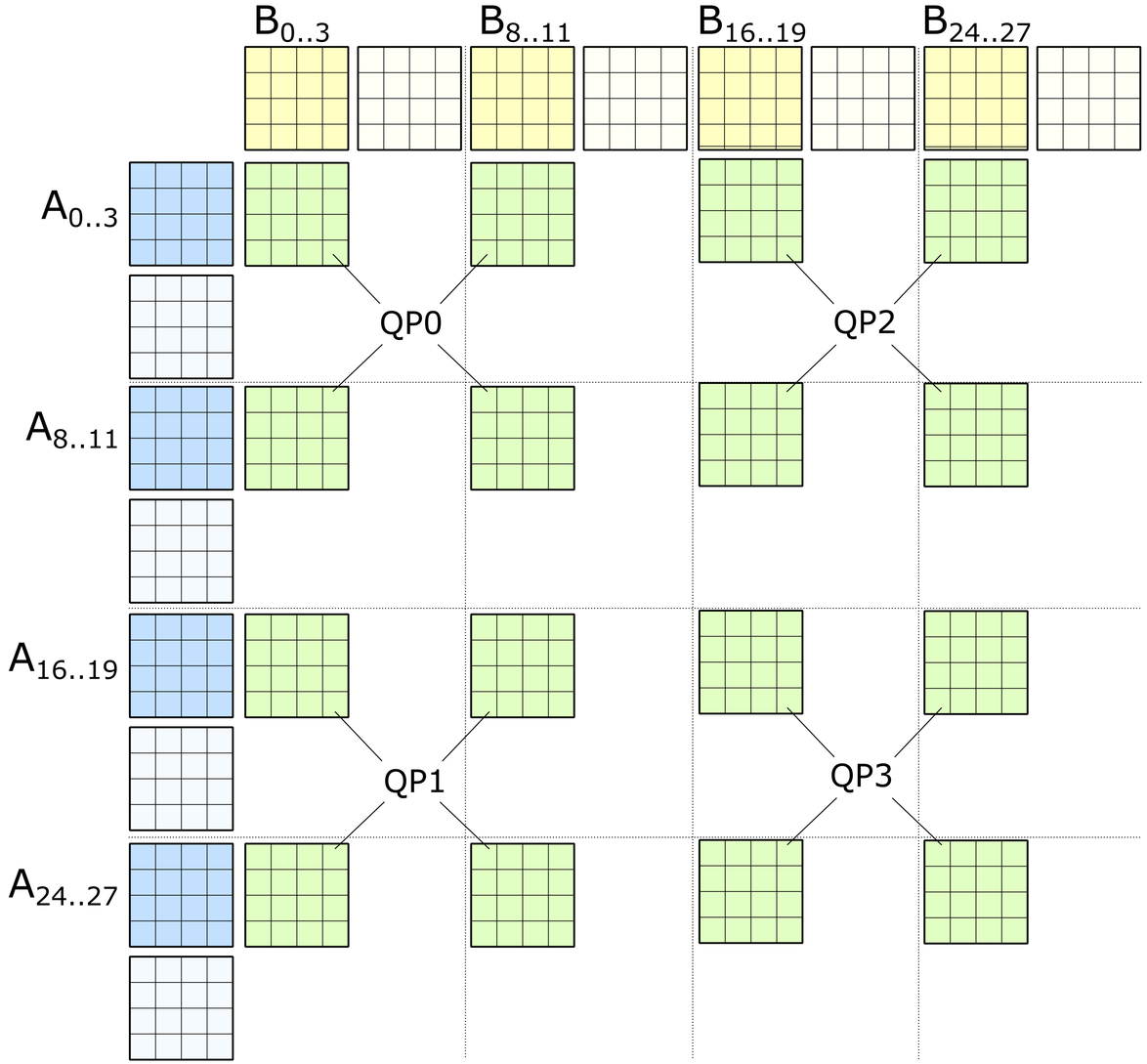
CPU

GPU

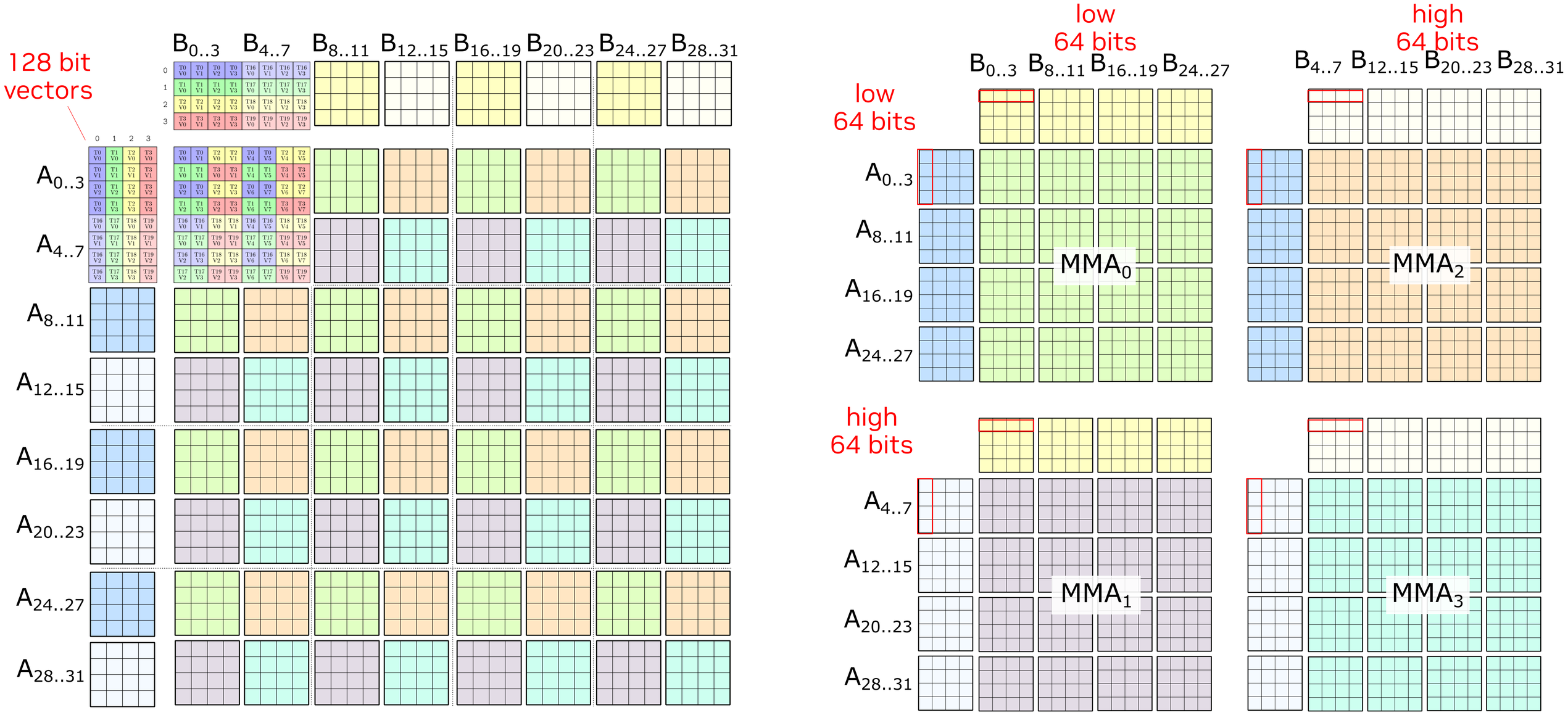


1 x SM70 MMA: 16-by-16-by-4

`SM70_8x8x4_F32F16F16F32_NT`
`Tiled in a 2x2 shape`



Volta µKernel: Spatially Interleaved MMAs for 128 bit smem loads



2x2 Spatial interleaving allows 128 smem loads,
Allowing us to feed 2 MMAs from a single load

Andrew Kerr, GTC 2019

Ad-hoc partitioning doesn't scale

Bespoke layouts/iterators

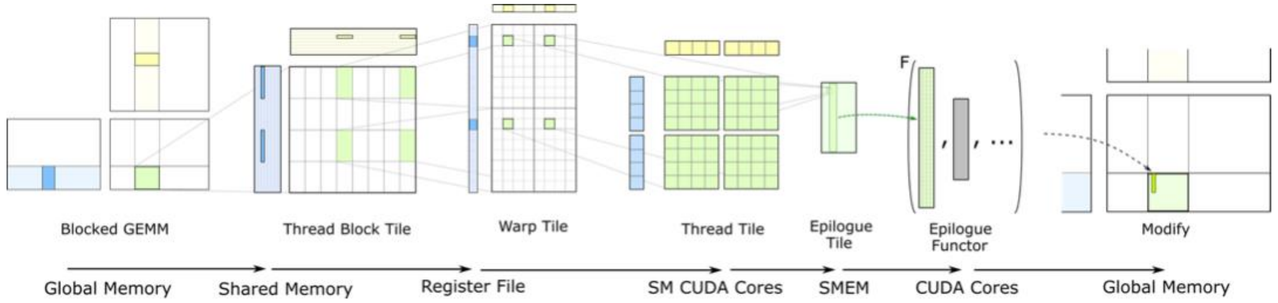
- CUTLASS 2.x implements custom layout functors
  - Mapping logical coord -> index
- These iterators implement post-partitioned layouts
- Thread layouts are implicit in the index computations
- They cannot partition arbitrary input tensors
- Each new arch requires new layouts to be defined
- Indexing math is hard to implement, harder still to maintain

An excerpt out of CUTLASS 2.x's Volta SMEM access iterator

```
/// Returns the offset of a coordinate in linear memory.
CUTLASS_HOST_DEVICE
LongIndex operator()(TensorCoord const &coord) const {
    int vec_contiguous_idx = coord.contiguous() / kElementsPerAccess;
    int vec_strided_idx = coord.strided() / kFactor;

    int tile_contiguous_idx =
        vec_contiguous_idx / (TileShape::kContiguous / kFactor);
    int tile_contiguous_residual =
        vec_contiguous_idx % (TileShape::kContiguous / kFactor) +
        ((coord.strided() % kFactor) * (TileShape::kContiguous / kFactor));
    int tile_strided_residual = vec_strided_idx % TileShape::kStrided;

    int partition_contiguous_idx =
        tile_contiguous_residual / PartitionShape::kContiguous;
    int partition_strided_idx =
        tile_strided_residual / PartitionShape::kStrided;
    int partition_contiguous_residual =
        tile_contiguous_residual % PartitionShape::kContiguous;
    int partition_strided_residual =
        tile_strided_residual % PartitionShape::kStrided;

    int permuted_vec_contiguous_within_partition =
        partition_contiguous_residual ^ (partition_strided_residual % 4);
    int permuted_partition_contiguous_within_tile =
        partition_contiguous_idx ^ (partition_strided_idx % 2);

    int element_contiguous = (tile_contiguous_idx * TileShape::kContiguous +
                              permuted_partition_contiguous_within_tile *
                                  PartitionShape::kContiguous +
                              permuted_vec_contiguous_within_partition) *
                                 kElementsPerAccess +
                             (coord.contiguous() % kElementsPerAccess);

    int element_strided = vec_strided_idx;
    return element_contiguous + element_strided * stride_[0] * kFactor;
```

Problem 1: Complicated Partitioning Patterns

Prevent us from writing canonical loops for all MMAs



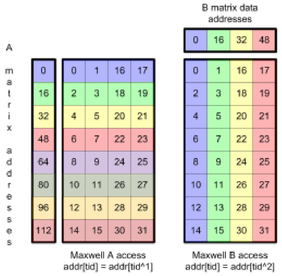


Figure 3b: Maxwell thread assignments for LDS.U matching

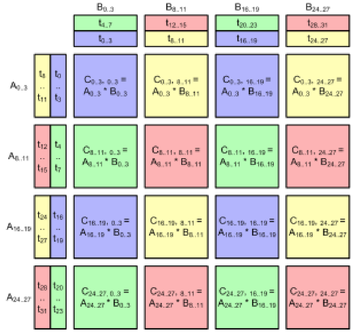


Figure 2a: Composing sparse A[16,4] * B[4, 16] = C[16, 16]
warp-wide multiply from four A[8, 4] * B[4, 8] = C[8, 8] 8-thread multiplies.

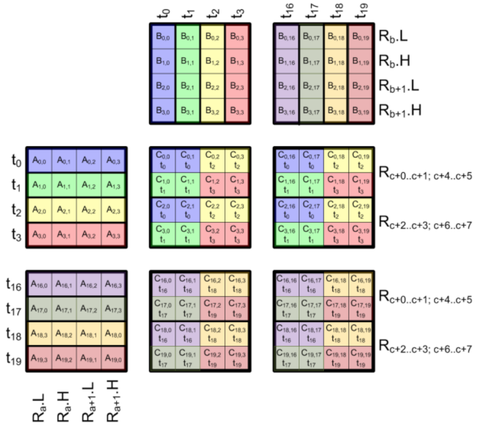
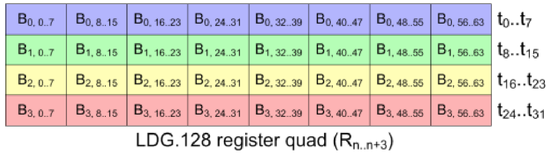
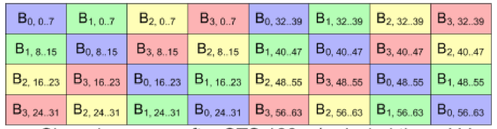
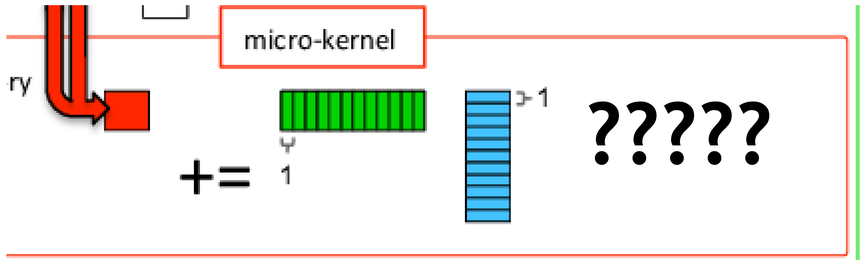


Figure 1b: HMMA.884.F32.TN data layout.
Again, ignore the apparent isomorphism
between thread numbers and matrix indices.





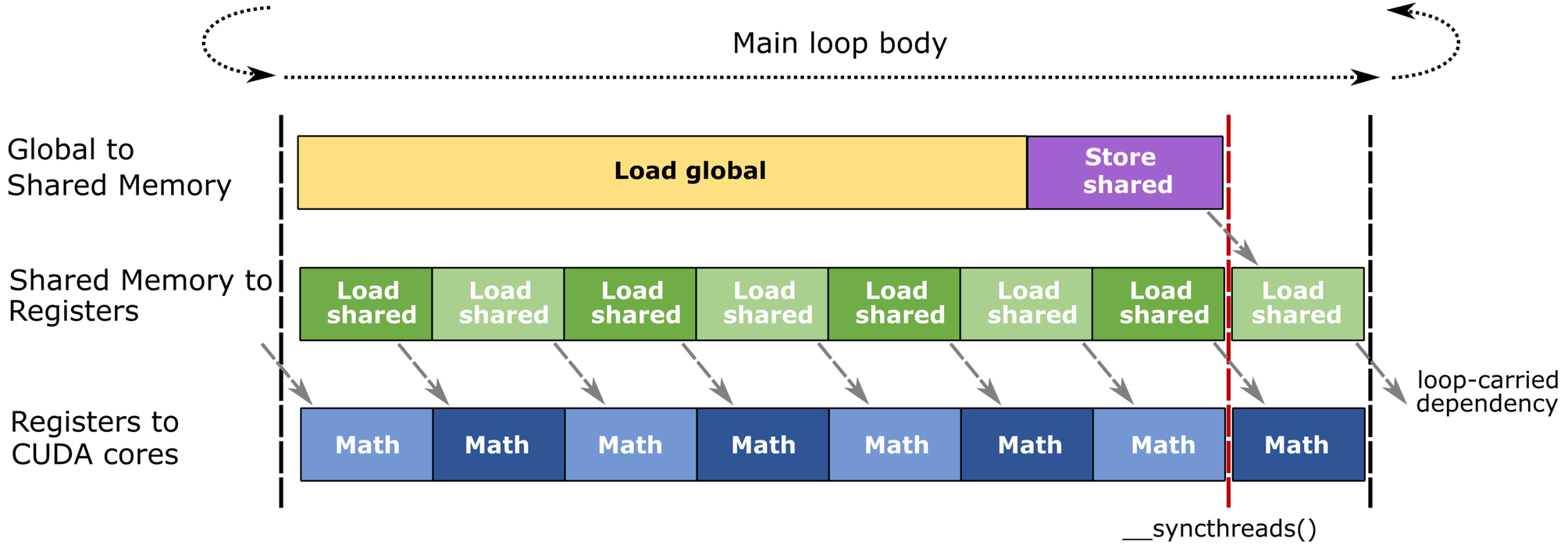



```
for (int cta_n = 0; cta_n < GemmN; cta_n += CtaN) {
  for (int cta_m = 0; cta_m < GemmM; cta_m += CtaM) {
    for (int cta_k = 0; cta_k < GemmK; cta_k += CtaK) {

      // Copy from global to shared

      for (int warp_n = 0; warp_n < CtaN; warp_n += WarpN) {
        for (int warp_m = 0; warp_m < CtaM; warp_m += WarpM) {
          for (int warp_k = 0; warp_k < CtaK; warp_k += WarpK) {

            // Copy from shared to registers

            for (int mma_k = 0; mma_k < WarpK; mma_k += MmaK) {
              for (int mma_n = 0; mma_n < WarpN; mma_n += MmaN) {
                for (int mma_m = 0; mma_m < WarpM; mma_m += MmaM)

                  // Compute in registers
                  gemm(a, b, c);
```

Ampere Mainloop

ASYNC in SMEM and ILP in RMEM

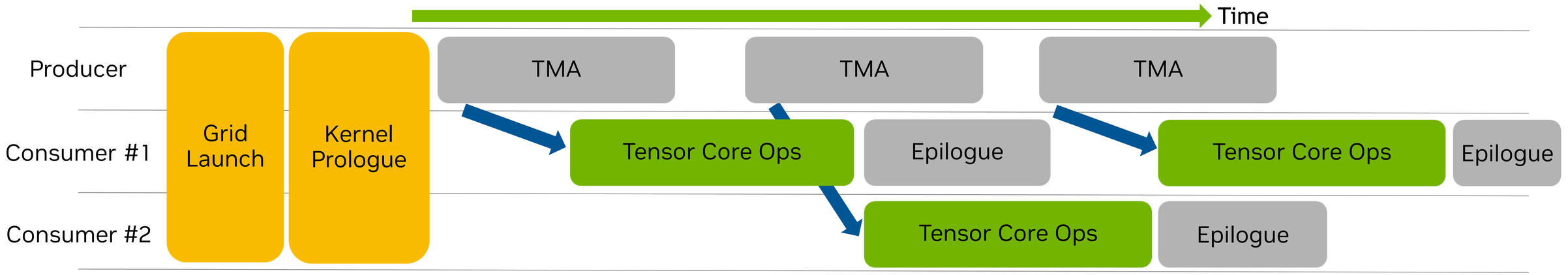
- Instruction interleaving between math and copy critical for speed of light performance
- Async pipeline feeds loads->compute
- Staged via local memories (smem and rmem)



Problem 2: Programmer Managed Asynchrony

GPUs require deeply async, managed, producer/consumer software pipelines

- Feeding the tensor cores constantly is hard – requires managing asynchrony and deep software pipelines
- With newer architectures like Hopper, even the MMA instruction is asynchronous
- Concurrency programming - Writing kernels isn't just about getting the layouts right anymore



Producer	Consumer #1	Consumer #2
PersistentTileSchedulerSm90 **scheduler(problem_shape, blk_shape, cluster_shape)**		

`// Data in via TMA`
 	
`while (work_tile_info.`**`is_valid_tile`**`) {`
`  collective_mainloop.`**`load()`**
`  scheduler.`**`advance_to_next_work()`**
`  work_tile_info = scheduler.`**`get_current_work()`**
`}`
	`// Mainloop, epilogue, and data out`
	
	`while (work_tile_info.`**`is_valid_tile`**`) {`
	`  collective_mainloop.`**`compute()`**
	`  scheduler.`**`advance_to_next_work`**`(NumConsumers)`
	`  work_tile_info = scheduler.`**`get_current_work()`**
	`}`	

CUTLASS 3.x

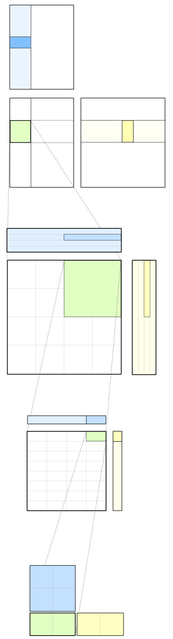
CUTLASS

CUDA C++ Template Library for High Performance Linear Algebra

Tensor core computations at all scopes and scales, decomposed into their “moving parts”

Provides a native **tile-based programming model for GPU kernels**

Device	{ GEMM, Convolution, Reductions , BLAS3 } x { all data types } x { SIMT, Tensor Cores } x { all architectures }
Kernel	GEMM, Batched GEMM, Convolution, Reduction, Fused output operations, Fused input operations
Collective	CUTLASS **temporal micro-kernels** (async producer/consumer pipelines orchestrating spatial micro-kernels)
Atom	CuTe **spatial micro-kernels** (Tiled MMA / Copy)
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

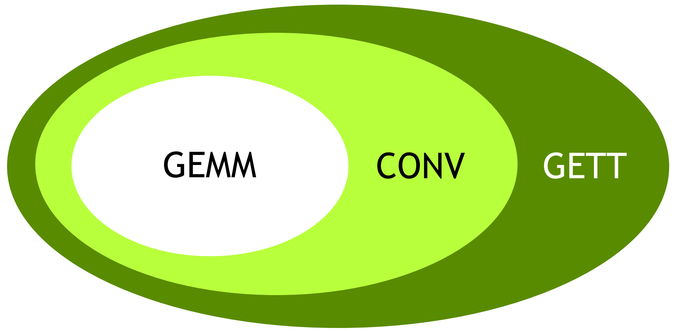
- 4.7K stars, 3M clones/month, 100+ contributors, and many active users
- Latest revision: CUTLASS 3.5
- Documentation: https://github.com/NVIDIA/cutlass#documentation
- Presented: GTC’18, GTC’19, GTC’20, GTC’21, GTC'22 , GTC’22, GTC’23, GTC’24
- Come join the CUTLASS channel in our discord: https://discord.gg/CVEJqWtU

CUTLASS 3: Design Goals

- Public Tensor Core programming model for NVIDIA GPUs
  - Serve as a production grade example for the world
- Extreme focus on developer productivity for custom kernels
  - Allow customizing any layer in the hierarchy while preserving composability with other layers
- If it compiles, it will be correct – actionable static assert messages otherwise
  - Static asserts at every layer to ensure layout and dispatch compatibilities
- Single, clear points of customization and dispatch to flatten the learning curve
  - Reduce API surface area with fewer named types

3.x Feature Highlights

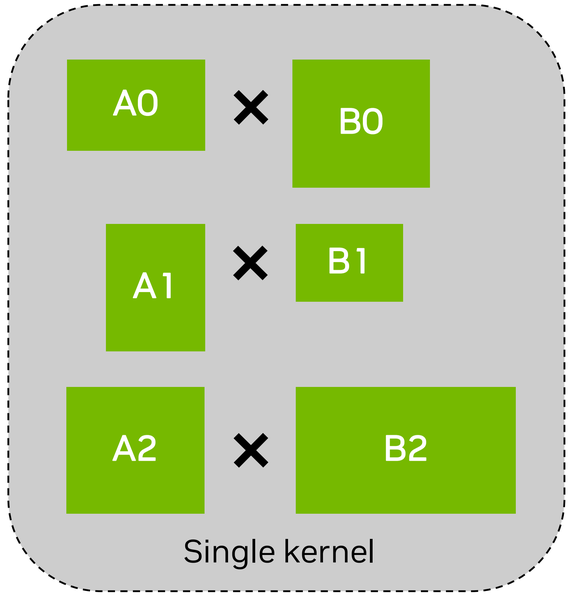
Convolutions in CUTLASS 3



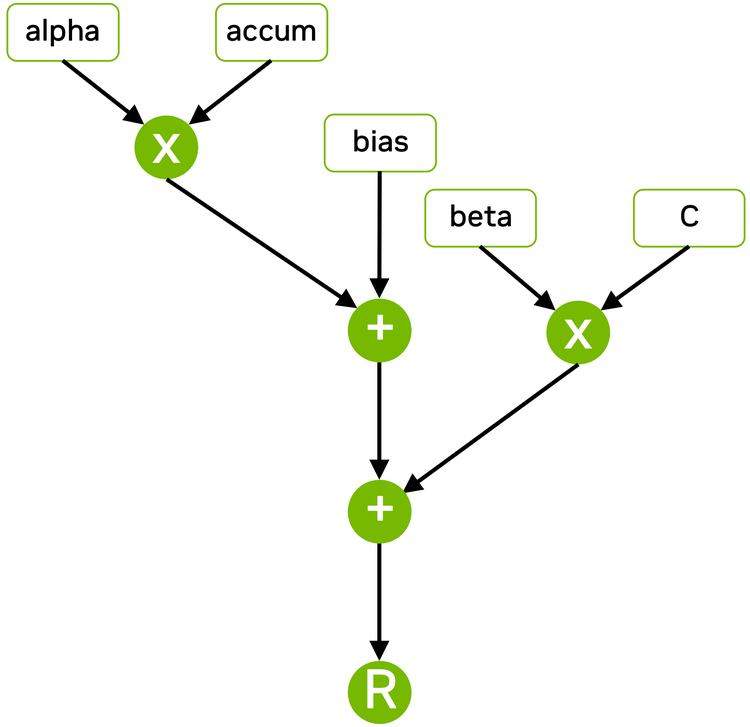
Features for LLMs

Mixed-input GEMM

Grouped GEMM

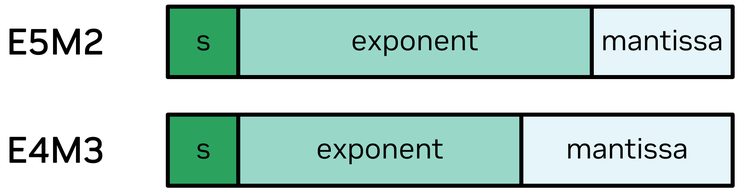


Epilogue Fusions via Visitor Tree



More CUTLASS Feature Highlights

FP8 GEMMs and convolutions for Hopper and Ada



Tile schedulers for composable load balancing

```
using GemmKernel = cutlass::gemm::kernel::GemmUniversal<
  ProblemShape,
  CollectivMainloop,
  CollectiveEpilogue,
  cutlass::gemm::StreamKScheduler
>;
```

Improved Python interface support and addition of PyPI wheel

```
pip install nvidia-cutlass
```

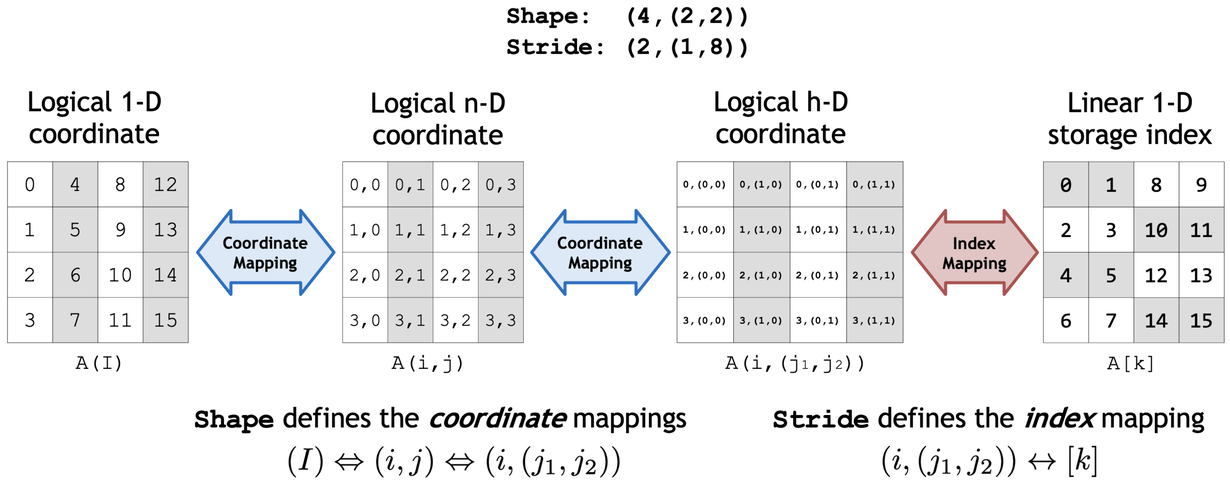
More on GitHub

- Expanded compiler support
- Narrow-alignment GEMMs
- Improved documentation

CuTe Layouts and Layout Algebra

cutlass/include/cute

- **`cute::Layout<Shape, Stride>`**
  - Compact representation of all affine layouts
  - Hierarchical and multi-modal shapes & strides
  - Layouts always maintain logical consistency
  - Can explicitly define thread layouts
  - Handles all coord->mapping
  - Compose with swizzle functors



`Shape` defines the ***coordinate*** mappings

$$(I) \Leftrightarrow (i,j) \Leftrightarrow (i,(j_1,j_2))$$

`Stride` defines the ***index*** mapping

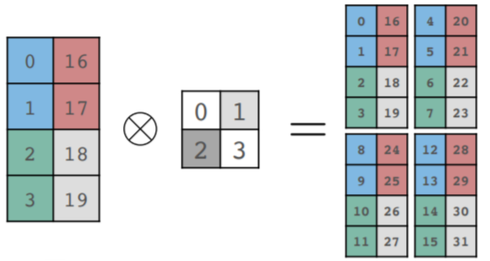
$$(i,(j_1,j_2)) \leftrightarrow [k]$$

- Formalized algebra of layouts
  - Allows for composability at all layers
  - Build complicated layouts from simple invariant layouts

Logical product:

$$f_A \otimes g_B = (f_A, f_A^* \circ g_B) \rightarrow (f_A, h_{B'})$$

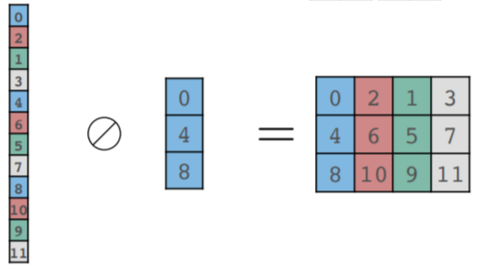
"Produce a layout where every element of layout B is a layout A."



Logical divide:

$$f_A \oslash g_B = f_A \circ (g_B, g_B^*) \rightarrow (h_{B'}, \ell_C)$$

"Produce a layout of Bs from a layout A."



Layout And Tensors

Why CuTe?

- Layout subsumes every CUTLASS-2 iterator
  - Condense complexity to a single impl. and vocab type
- Formal algebra to manipulate Layout
  - composition
  - complement
  - right_inverse, left_inverse
  - “product”
  - “divide”
- Layout for both threads and data

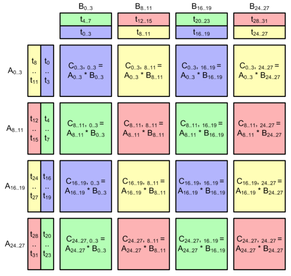
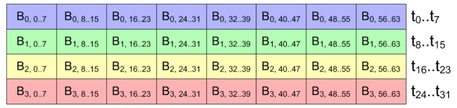
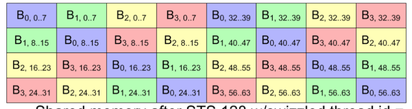


Figure 2a: Composing sparse A[16,4] * B[4, 16] = C[16, 16]
warp-wide multiply from four A[8, 4] * B[4, 8] = C[8, 8] 8-thread multiplies.



LDG.128 register quad ($R_{n..n+3}$)



Shared memory after STS.128 w/swizzled thread id =
(tid[1:0] << 3) | (tid & 4) | (tid[4:3] ^ tid[1:0])

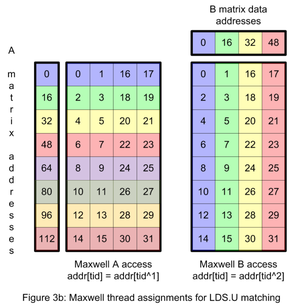


Figure 3b: Maxwell thread assignments for LDS.U matching

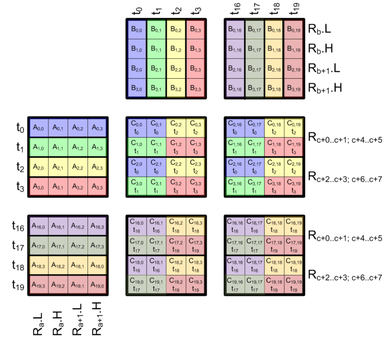
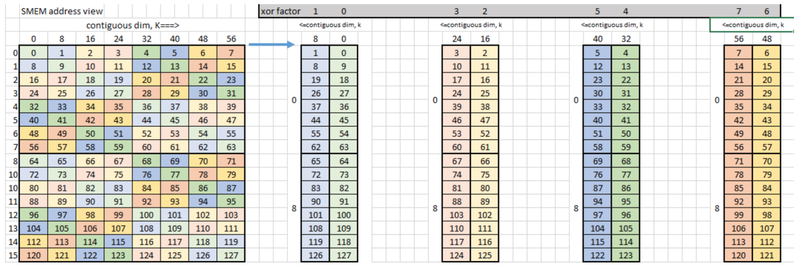
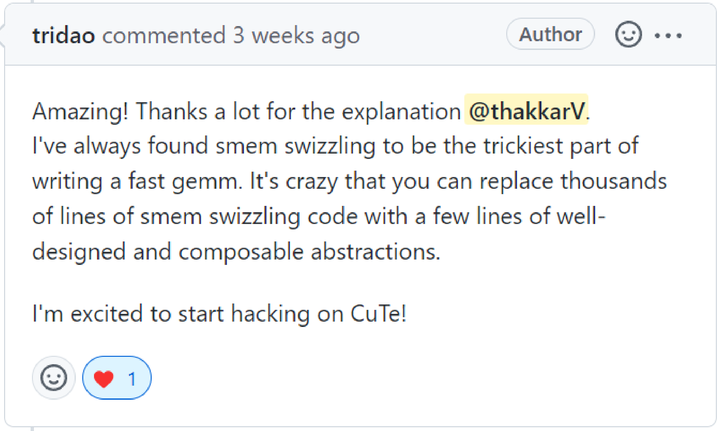


Figure 1b: HMMA.884.F32.TN data layout.
Again, ignore the apparent isomorphism
between thread numbers and matrix indices.





tridao commented 3 weeks ago — Author

Amazing! Thanks a lot for the explanation @thakkarV.
I've always found smem swizzling to be the trickiest part of writing a fast gemm. It's crazy that you can replace thousands of lines of smem swizzling code with a few lines of well-designed and composable abstractions.

I'm excited to start hacking on CuTe!

1

CUTLASS 2.x

```
RowMajor
ColumnMajor
RowMajorInterleaved
ColumnMajorInterleaved
PitchLinear
TensorNCHW
VoltaTensorOpMultiplicandCongruous
ColumnMajorVoltaTensorOpMultiplicandCongruous
RowMajorVoltaTensorOpMultiplicandCongruous
VoltaTensorOpMultiplicandBCongruous
ColumnMajorVoltaTensorOpMultiplicandBCongruous
RowMajorVoltaTensorOpMultiplicandBCongruous
VoltaTensorOpMultiplicandCrosswise
ColumnMajorVoltaTensorOpMultiplicandCrosswise
RowMajorVoltaTensorOpMultiplicandCrosswise
Many, many more …
```

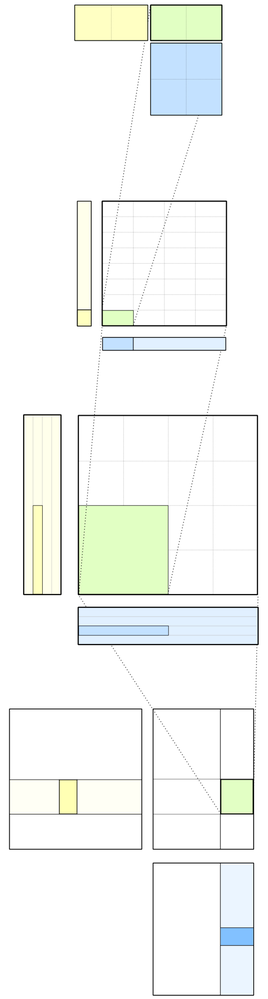
CUTLASS 3.x

Layout<Shape, Stride>

CUTLASS 3 Conceptual Hierarchy

CUTLASS 3.x computation hierarchy is not centered around the hardware hierarchy

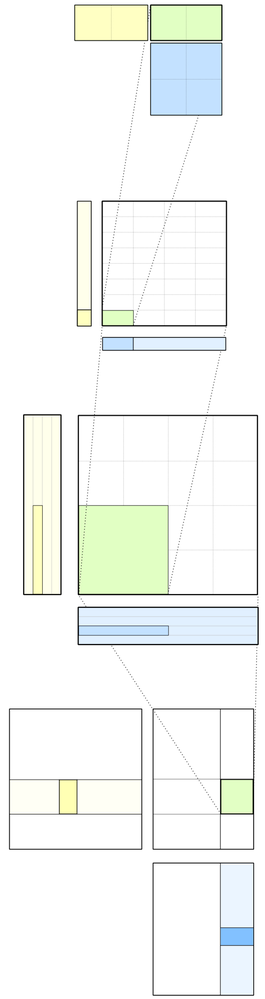
- **Atom layer:** Architecture instructions and associated meta-information
  - Smallest set of threads and values that must participate in an architecture accelerated specified math/copy op
- **Tiled MMA/Copy**: Spatial Microkernel layer
  - **Describes the complete spatial tiling of a math/copy operation**
  - Write canonical loops with arch specific instructions
- **Collective layer:** Temporal Microkernel layer
  - **Describes the complete temporal tiling of spatial microkernels and computing one output tile**
  - Abstract complex arch-specific synchronization, warp-specialization, pipelining, and instruction interleaving
- **Kernel layer**: Outermost loops around collectives
  - Conceptually: A collection of all threadblock/clusters in the grid
  - Responsible for load balancing across tiles, thread marshalling, grid planning, and arguments construction
- **Device layer:** host side setup and interface



CUTLASS 3 API

Single points of entry at each abstraction layer

- Spatial microkernels: `cute::Tiled{Mma|Copy}<>`
  - Robust representation power across a wide range GPU architectures
- Temporal Microkernels: `collective::Collective{Mma|Conv|Epilogue|Transform}<>`
  - Dispatched against by policies that also define the set of kernel schedules they can be composed with
- Kernel layer: `kernel::{Gemm|Conv}Universal<>`
  - Treats GEMM as a composition of a collective mainloop and a collective epilogue
  - **Tile Schedulers** are a first class at this level – decide which tile coords map to which program ID
- Device layer: `device::{Gemm|Conv}UniversalAdapter<>`
  - Just a handle object to the kernel
- cutlass::Pipeline: used for abstracting synchronization across or within the layers
- Static asserts everywhere to guard against invalid compositions or incorrect layouts




NVIDIA
Off the shelf CUTLASS GEMM

CUTLASS 3.0 GETT

CUTLASS 3.0 Hopper GEMMs are GETTs in Disguise

```
// Build the mainloop type
using CollectiveMainloop = typename cutlass::gemm::collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    ElementA, StrideA, AlignmentA,
    ElementB, StrideB, AlignmentB,
    ElementAccumulator,
    TilesShape, ClusterShape,
    cutlass::gemm::collective::StageCountAuto,
    cutlass::gemm::collective::KernelScheduleAuto
  >::CollectiveOp;

// Build the epilogue type
using CollectiveEpilogue = typename cutlass::epilogue::collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    TileShape, ClusterShape,
    cutlass::epilogue::collective::EpilogueTileAuto,
    ElementAccumulator, ElementCompute,
    ElementC, StrideC, AlignmentC,
    ElementD, StrideD, AlignmentD,
    cutlass::epilogue::collective::EpilogueScheduleAuto
  >::CollectiveOp;

// Compose both at the kernel layer, and define the type of our problem shape tuple
using GettKernel = cutlass::gemm::kernel::GemmUniversal<
    ProblemShape_MNKL, // still a rank-4 tuple, but now is hierarchical
    CollectiveMainloop,
    CollectiveEpilogue>;

// Device layer handle to the kernel
using Gett = cutlass::gemm::device::GemmUniversalAdapter<GettKernel>;
```

CUTLASS 3: Collective API

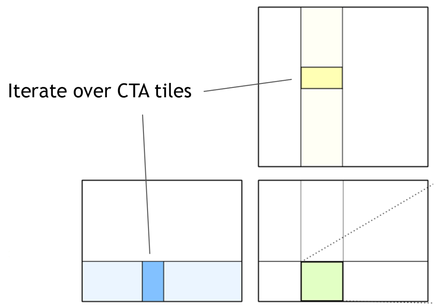
`cutlass::collective::{Gemm|Conv|Epilogue}`

- Implements a temporal microkernel
  - Computes one output tile
  - Can be warp-specialized
  - Can have async spatial microkernels (Producer-Consumer pipelines)

- Dispatch policies used for selecting between specializations
  - One dispatch policy per mainloop variant
  - Can possibly compose with more than one kernel schedule! (next slide)

```
// Dispatch policy examples
// 2 stage pipeline through 1 stage in smem, 1 in rmem, with predicated gmem loads
struct MainloopSm70TwoStage;

// n-buffer in smem (cp.async), pipelined with registers, WITHOUT predicated gmem loads
struct MainloopSm80CpAsyncUnpredicated;

// n-buffer in smem (TMA), pipelined with Hopper GMMA and TMA, Warp specialized dynamic schedule
Struct MainloopSm90TmaGmmaWarpSpecialized;
```

```
template <
  class DispatchPolicy,
  class TileShape,
  class ElementA,
  class SmemLayoutA,
  class ElementB,
  class SmemLayoutB,
  class ElementC,
  class ArchTag,
  class TiledMma,
  class GmemCopyAtomA,
  class SmemCopyAtomA,
  class GmemCopyAtomB,
  class SmemCopyAtomB
>
struct CollectiveMma;
```



Multiply tiles together
Add them to output tile

CUTLASS 3: Kernel API

`cutlass::gemm::kernel::GemmUniversal<>`

- Each GEMM kernel is a composition of three parts:
  - Collective mainloop (gemm or conv)
  - Collective epilogue
  - Tile scheduler

- Implements:
  - Grid planning logic and kernel global entry point
  - Thread marshalling for warp-specialized collectives
  - Block/Cluster wide swizzling
  - Load balancing schedules (split-K, stream-K)

- Selection via mainloop dispatch policy's `Schedule` tag
  - SM90 cp.async + GMMA multistage
  - SM90 TMA + GMMA multistage
  - SM90 TMA + GMMA + Warp-Specialized non-persistent
  - SM90 TMA + GMMA + Warp-Specialized + Pingpong Persistent
  - SM90 TMA + GMMA + Warp-Specialized + Cooperative Persistent + StreamK

```
template <
  class ProblemShapeOrThreadblockMma_,
  class CollectiveMainloopOrEpilogue_,
  class CollectiveEpilogueOrThreadblockSwizzle_,
  class GridSwizzle_ = void,
  class Enable = void
>
class GemmUniversal;
```

Kernel layer back-compatible API entry point

Sensible defaults with incremental opt-ins

`cutlass::gemm::collective::CollectiveBuilder<>`

"I just want a Hopper collective":

```
using CollectiveOp = typename collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    half_t, LayoutA, 8,
    half_t, LayoutB, 8,
    float,
    Shape<_128,_128,_64>, Shape<_1,_2,_1>,
    gemm::collective::StageCountAuto,
    gemm::collective::KernelScheduleAuto
  >::CollectiveOp;
```

"I want a Hopper collective but composed with persistent kernel and 5 stages":

```
using CollectiveOp = typename collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    half_t, LayoutA, 8,
    half_t, LayoutB, 8,
    float,
    Shape<_128,_128,_64>, Shape<_1,_2,_1>,
    gemm::collective::StageCount<5>,
    gemm::KernelTmaWarpSpecializedPersistent
  >::CollectiveOp;
```

```
template <
  class ArchTag,
  class OpClass,
  class ElementA,
  class GmemLayoutA,
  int   AlignmentA,
  class ElementB,
  class GmemLayoutB,
  int   AlignmentB,
  class ElementAccumulator,
  class TileShape_MNK,
  class ClusterShape_MNK,
  class StageCountType,
  class KernelScheduleType,
  class Enable = void
>
struct CollectiveBuilder;
```

https://github.com/NVIDIA/cutlass/tree/master/examples/49_hopper_gemm_schedules_with_collective_builder

Builders do the heavy lifting …

```
using CollectiveOp = typename collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    half_t, LayoutA, 8,
    half_t, LayoutB, 8,
    float,
    Shape<_128,_128,_64>, Shape<_2,_1,_1>,
    gemm::collective::StageCountAuto,
    gemm::collective::KernelScheduleAuto
  >::CollectiveOp;
```

Automatically maps down to the best mainloop config:
stage count, GMMA instruction, smem layouts, TMA mcast, kernel schedule, mainloop type …

```
cutlass::gemm::collective::CollectiveMma<
    cutlass::gemm::MainloopSm90TmaGmmaWarpSpecialized<
        7, Shape<_2,_1,_1>, cutlass::gemm::KernelTmaWarpSpecializedCooperative>,
    Shape<_128, _128, _64>,
    half_t, Stride<int64_t, _1, int64_t>,
    half_t, Stride<int64_t, _1, int64_t>,
    TiledMMA<
        MMA_Atom<SM90_64x128x16_F32F16F16_SS<GMMA::Major::K, GMMA::Major::K>>,
        Layout<Shape<_2, _1, _1>>,
    SM90_TMA_LOAD,
    ComposedLayout<Swizzle<3, 4, 3>, smem_ptr_flag_bits<16>, Layout<tuple<_8, _64>, tuple<_64, _1>>>,
    void, identity,
    SM90_TMA_LOAD_MULTICAST,
    ComposedLayout<Swizzle<3, 4, 3>, smem_ptr_flag_bits<16>, Layout<tuple<_8, _64>, tuple<_64, _1>>>,
    void, identity>
```

Convolutions provide a familiar builder API

`cutlass::conv::collective::CollectiveBuilder<>`

"I just want a Hopper Fprop collective for NWC layout tensors (1-D Fprop)":

```
using CollectiveOp = typename conv::collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    conv::Operator::kFprop,
    half_t, layout::TensorNWC, 8,
    half_t, layout::TensorNWC, 8,
    float,
    Shape<_64,_128,Shape<_64>>, Shape<_2,_1,_1>,
    conv::collective::StageCountAuto,
    conv::collective::KernelScheduleAuto
  >::CollectiveOp;
```

"I want a Hopper Dgrad collective on NDHWC layout tensors (3-D Dgrad)":

```
using CollectiveOp = typename conv::collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    conv::Operator::kDgrad,
    half_t, layout::TensorNDHWC, 8,
    half_t, layout::TensorNDHWC, 8,
    float,
    Shape<_64,_128,Shape<_64>>, Shape<_2,_1,_1>,
    conv::collective::StageCount<5>,
    conv::collective::KernelScheduleAuto
  >::CollectiveOp;
```

```
template <
  class ArchTag,
  class OpClass,
  conv::Operator,
  class ElementA,
  class GmemLayoutA,
  int AlignmentA,
  class ElementB,
  class GmemLayoutB,
  int AlignmentB,
  class ElementAccumulator,
  class TileShape_MNK,
  class ClusterShape_MNK,
  class StageCountType,
  class KernelScheduleType,
  class Enable = void
>
struct CollectiveBuilder;
```

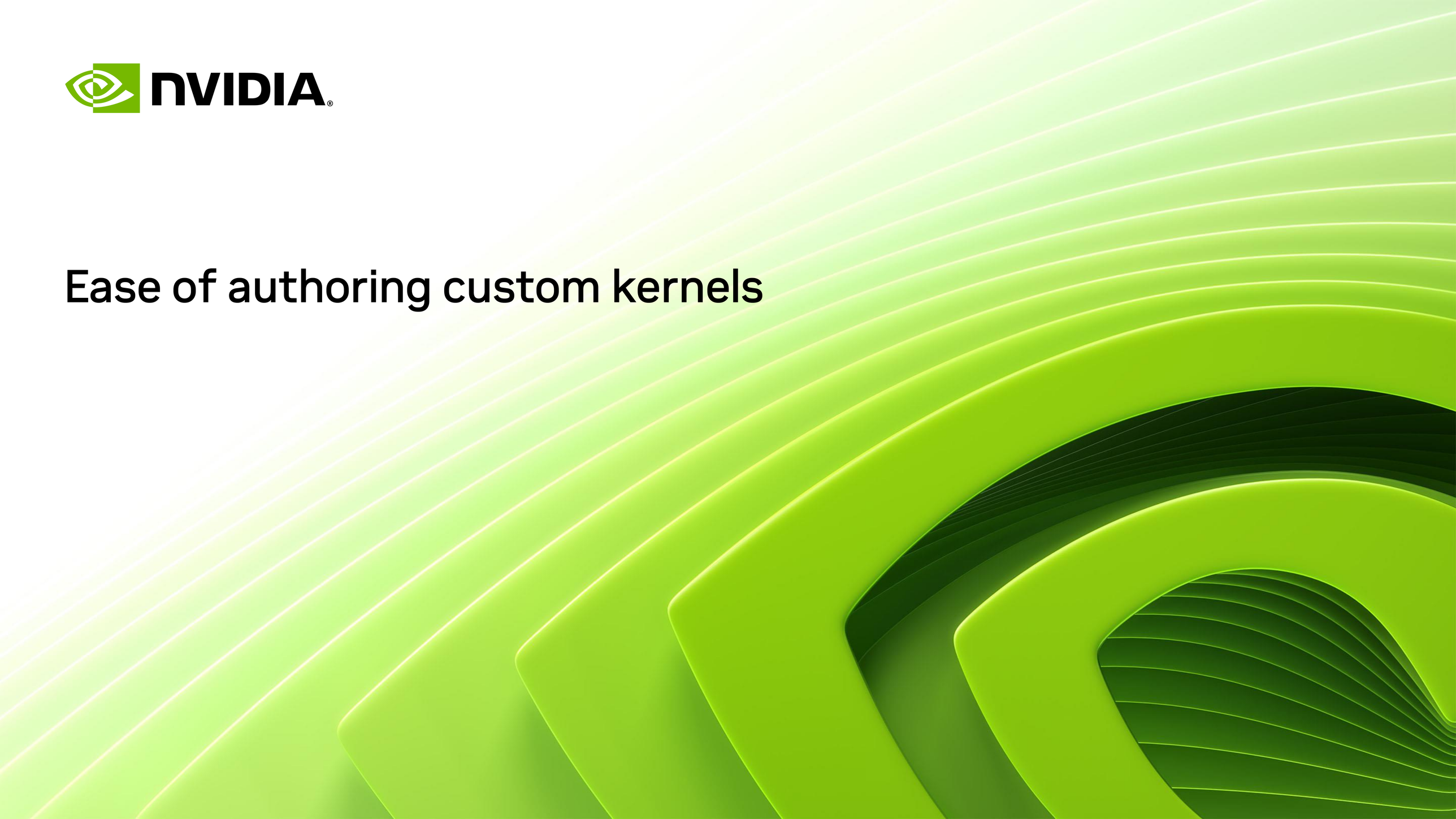
Builders do the heavy lifting …

```
using CollectiveOp = typename conv::collective::CollectiveBuilder<
    arch::Sm90, arch::OpClassTensorOp,
    conv::Operator::kDgrad,
    half_t, layout::TensorNDHWC, 8,
    half_t, layout::TensorNDHWC, 8,
    float,
    Shape<_64,_128,Shape<_64>>, Shape<_2,_1,_1>,
    conv::collective::StageCountAuto,
    conv::collective::KernelScheduleAuto
  >::CollectiveOp;
```

Automatically maps down to the best mainloop config:
mainloop type, stage count, GMMA instruction, TMA instruction, smem layouts, kernel schedule …

```
cutlass::conv::collective::CollectiveConv<
    conv::MainloopSm90TmaGmmaWarpSpecializedImplicitGemm<
        conv::Operator::kDgrad, 8, 3, Shape<_2,_1,_1>,
        cutlass::conv::KernelImplicitTmaWarpSpecializedSm90, 1>,
        Shape<_64,_128,Shape<_64>>,
    cutlass::half_t,
    cutlass::half_t,
    TiledMMA<
        MMA_Atom<SM90_64x128x16_F32F16F16_SS<GMMA::Major::K, GMMA::Major::MN>>,
        Layout<Shape<_1,_1,_1>>>,
    cutlass::conv::collective::detail::Sm90ImplicitGemmTileTraits<
        SM90_TMA_LOAD_IM2COL,
        ComposedLayout<Swizzle<3, 4, 3>, smem_ptr_flag_bits<16>,
           Layout<Shape<_64,_64,_8>, Stride<_64,_1,_4096>>>,
        void>,
    cutlass::conv::collective::detail::Sm90ImplicitGemmTileTraits<
        SM90_TMA_LOAD_MULTICAST,
        ComposedLayout<Swizzle<3, 4, 3>, smem_ptr_flag_bits<16>,
           Layout<Shape<Shape<_64,_2>,_64,_8>, Stride<Stride<_1,_4096>,_64,_8192>>>,
        void>>
```

Ease of authoring custom kernels



Writing Custom Kernels

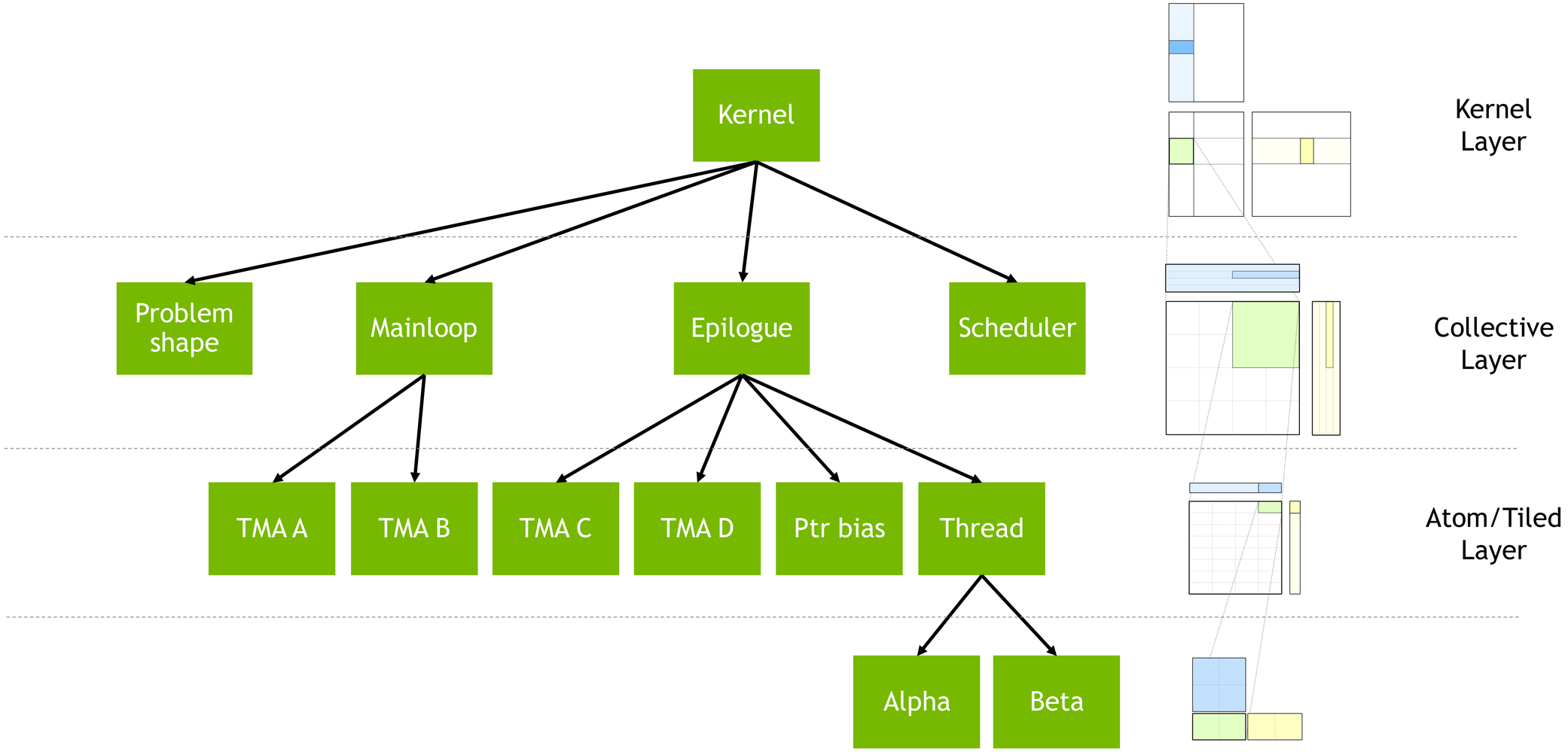
Plug and play with dispatch policies

- Write custom mainloops (micro-kernels), compose with existing schedule (outer loops) via dispatch policies
- Write custom schedules (outer loops), compose with existing mainloops (micro-kernels) via dispatch policies
- Kernel layer totally agnostic of # of in/out tensors and the semantics of the computation itself
- Kernel is a composition of a mainloop, epilogue, and tile scheduler: compose them freely

```
template<
  int Stages_,
  class ClusterShape_ = Shape<_1,_1,_1>,
  class KernelSchedule = KernelTmaWarpSpecialized // or KernelTmaWarpSpecializedPersistent
>
struct MyCustomHopperMainloopWoot {
  constexpr static int Stages = Stages_;
  using ClusterShape = ClusterShape_;
  using ArchTag = arch::Sm90;
  using Schedule = KernelSchedule;
};
```

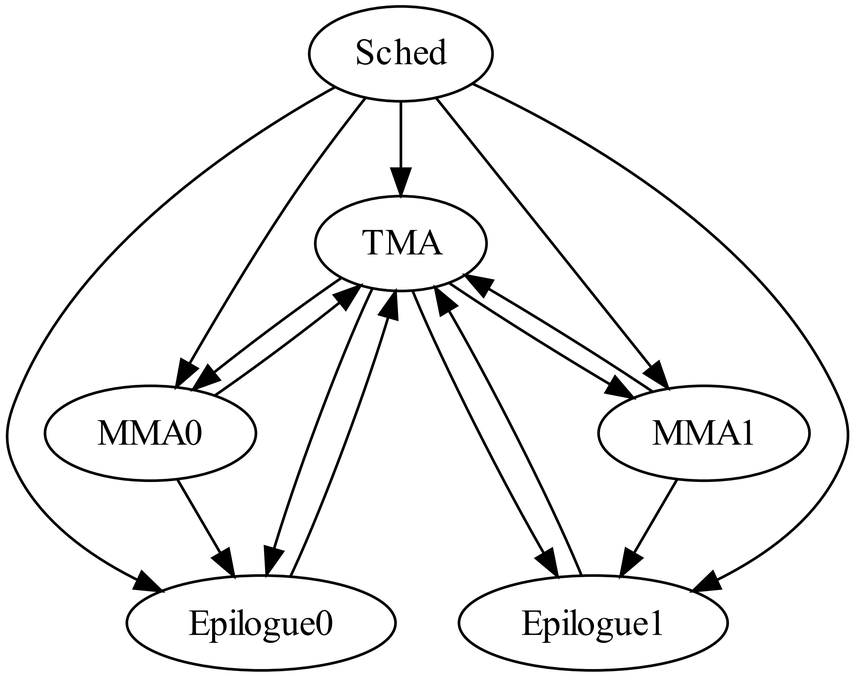
Kernel Params fully congruent with the structure of the kernel

Host Arguments -> Device Params in CUTLASS 3.x

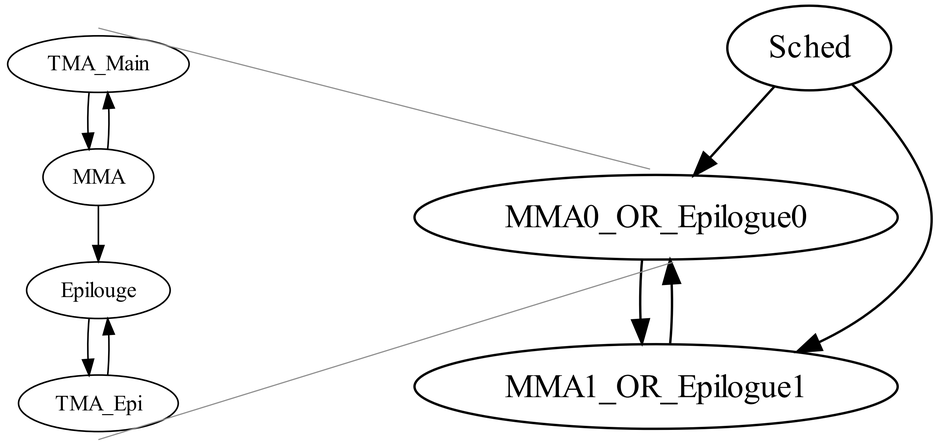


CUTLASS pipelines make synchro easy!

Collectives compose with each other and kernel schedules



Without a hierarchical pipeline model,
Hopper synchro model is too complex to reason about



Pipelines greatly simplify synchronization by grouping the synchro graph nodes hierarchically

CUTLA OFF

CUTLA ON

Custom kernel rules of thumb

What changes when?

- Kernel fusion?
  - Write a custom kernel schedule that compose multiple existing collectives together
  - Get tile schedulers, pipelines, and highly optimized collectives for free

- Mainloop fusion?
  - Customize an existing collective MMA out of CUTLASS
  - Define a new dispatch policy and compose with an existing kernel layer
  - Hopper Mixed Input GEMMs reuse the vanilla kernel layer! (get all outer loop optimizations for free!)
  - Prefer doing mainloop fusions on A tensor only (due to MMA feature supporting A from registers)

- Epilogue fusion?
  - Use epilogue visitor tree for to make your life extremely easy

- Custom load balancing, L2 locality, or other outer loop shenanigans?
  - Write a custom tile scheduler, compose with existing mainloops, epilogue, and entire kernel layers
  - This is how StreamK is implemented in CUTLASS 3.x – it does not need a new kernel layer

Support for a variety of mixed-input combinations

```
using CollectiveMainloop =
  typename cutlass::gemm::collective::CollectiveBuilder<
    cutlass::arch::Sm90, cutlass::arch::OpClassTensorOp,
    ElementA, LayoutA, AlignmentA,
    ElementB, LayoutB, AlignmentB,
    ElementAccumulator,
    TileShape, ClusterShape,
    StageCount,
    KernelScheduleType
  >::CollectiveOp;
```

Requirements:

- Tensor Core instruction exists for wider data type
- Operands meet TMA requirements

FP16 X INT8
FP16 X INT4
BF16 X INT8
FP8 X INT4
INT8 X FP16
INT8 X BF16
INT4 X FP8
★ INT2 X FP16
★ INT1 X FP8
(and more)

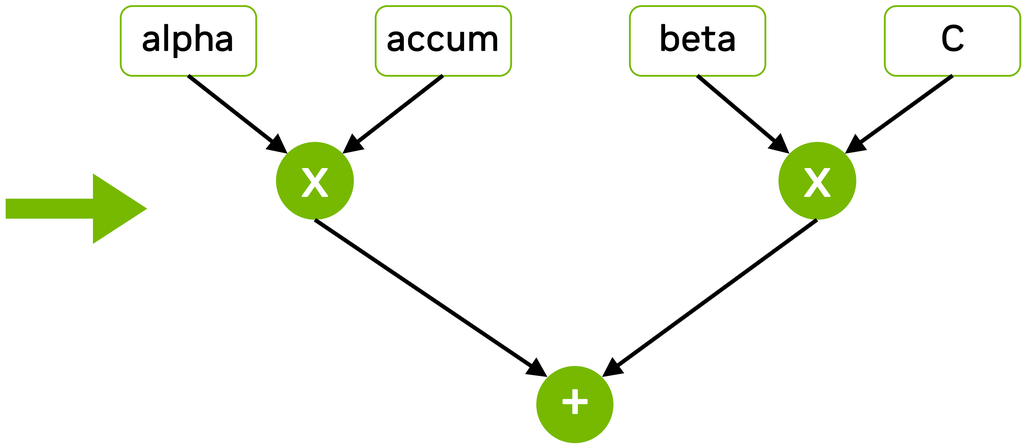
Epilogue Visitor Tree (EVT): building blocks for composing fused epilogues

- Set of primitive nodes that can be composed together to build complex epilogues

Load	Compute	Store
Accumulator Aux tensor Row broadcast Column broadcast Scalar broadcast	Compute (elementwise, binary, ternary)	Aux tensor Row reduction Column reduction Scalar reduction

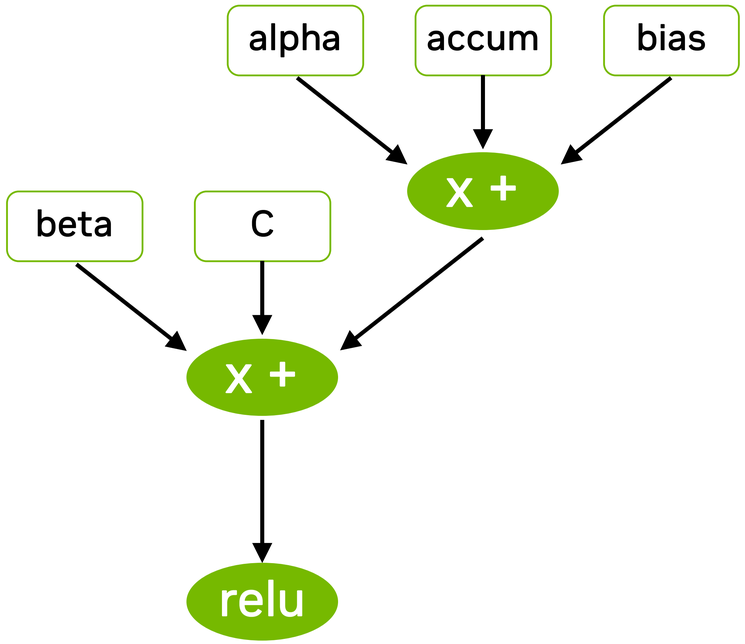
- Compose together nodes in a tree or DAG

(alpha * accumulators) + (beta * C)



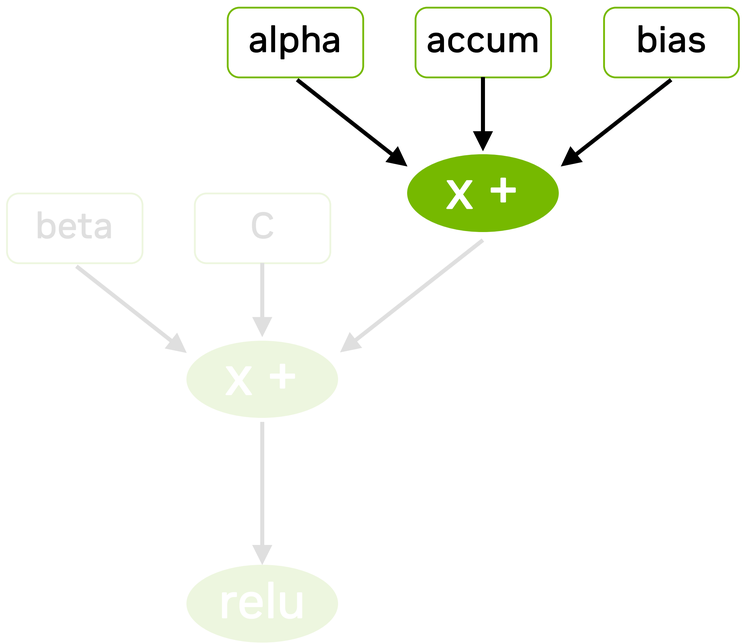
Adding an epilogue visitor tree to a GEMM in CUTLASS 3.x (C++)

ReLU((alpha * accumulators) + bias + (beta * C))



Adding an epilogue visitor tree to a GEMM in CUTLASS 3.x (C++)

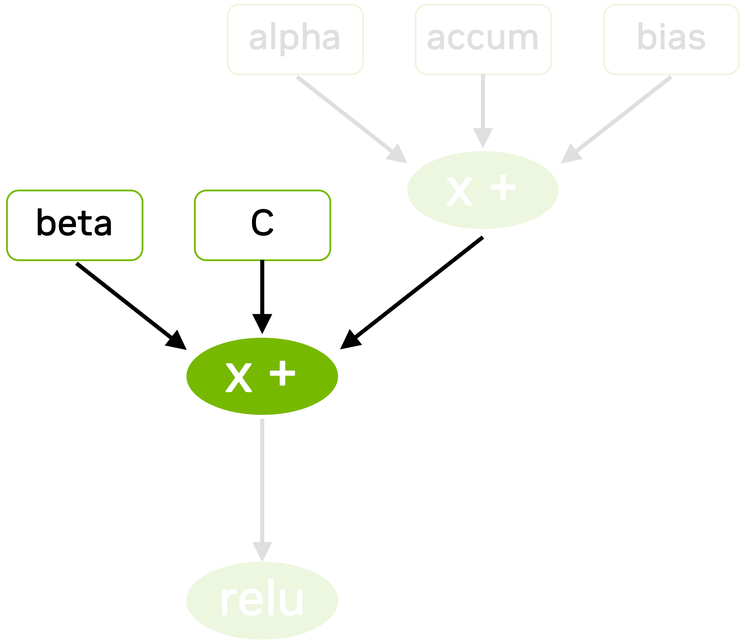
ReLU((alpha * accumulators) + bias + (beta * C))




```
using Alpha = Sm90ScalarBroadcast<ElementScalar>;
using Accum = Sm90AccFetch;
using Bias  = Sm90ColBroadcast<
    0, TileShape, ElementBias, Stride<_1,_0,int>, AlignmentBias>;
using MultiplyAdd = Sm90Compute<
    multiply_add, ElementCompute, ElementCompute, RoundStyle>;
using EVTCompute0 = Sm90EVT<MultiplyAdd, Alpha, Accum, Bias>;
```

Adding an epilogue visitor tree to a GEMM in CUTLASS 3.x (C++)

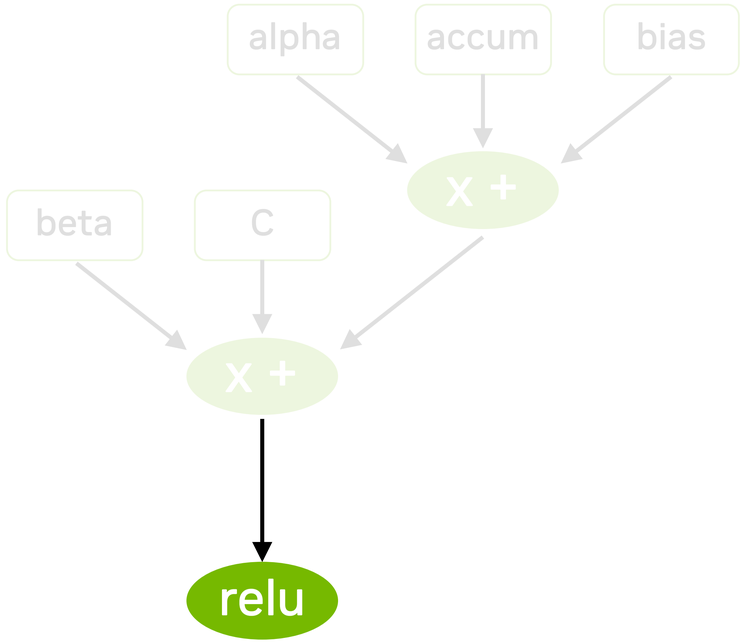
ReLU((alpha * accumulators) + bias + (beta * C))




```
using Alpha = Sm90ScalarBroadcast<ElementScalar>;
using Accum = Sm90AccFetch;
using Bias  = Sm90ColBroadcast<
    0, TileShape, ElementBias, Stride<_1,_0,int>, AlignmentBias>;
using MultiplyAdd = Sm90Compute<
    multiply_add, ElementCompute, ElementCompute, RoundStyle>;
using EVTCompute0 = Sm90EVT<MultiplyAdd, Alpha, Accum, Bias>;


using Beta = Sm90ScalarBroadcast<ElementScalar>;
using C    = Sm90SrcFetch<ElementSource>;
using EVTCompute1 = Sm90EVT<MultiplyAdd, Beta, C, EVTCompute0>;
```

Adding an epilogue visitor tree to a GEMM in CUTLASS 3.x (C++)

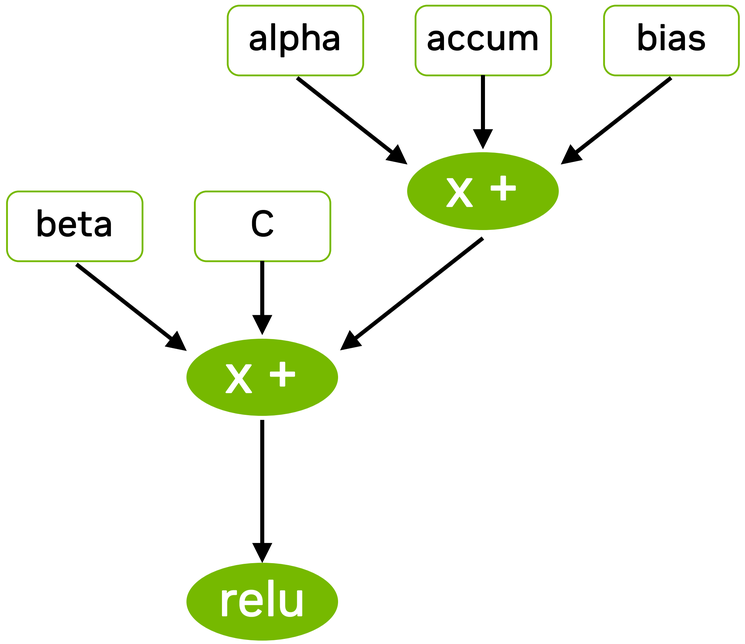
ReLU((alpha * accumulators) + bias + (beta * C))




```
using Alpha = Sm90ScalarBroadcast<ElementScalar>;
using Accum = Sm90AccFetch;
using Bias  = Sm90ColBroadcast<
    0, TileShape, ElementBias, Stride<_1,_0,int>, AlignmentBias>;
using MultiplyAdd = Sm90Compute<
    multiply_add, ElementCompute, ElementCompute, RoundStyle>;
using EVTCompute0 = Sm90EVT<MultiplyAdd, Alpha, Accum, Bias>;


using Beta = Sm90ScalarBroadcast<ElementScalar>;
using C    = Sm90SrcFetch<ElementSource>;
using EVTCompute1 = Sm90EVT<MultiplyAdd, Beta, C, EVTCompute0>;


using ReLUAct = Sm90Compute<
    ReLu, ElementOutput, ElementCompute, RoundStyle>;
using EVTOutput = Sm90EVT<ReLUAct, EVTCompute1>;
```

Adding an epilogue visitor tree to a GEMM in CUTLASS 3.x (C++)

ReLU((alpha * accumulators) + bias + (beta * C))




```
using EVTOutput = Sm90EVT<ReLUAct, EVTCompute1>;

using CollectiveEpilogue = typename
cutlass::epilogue::collective::CollectiveBuilder<
      cutlass::arch::Sm90, cutlass::arch::OpClassTensorOp,
      TileShape, ClusterShape,
      cutlass::epilogue::collective::EpilogueTileAuto,
      ElementAccumulator, ElementCompute,
      ElementC, LayoutC, AlignmentC,
      ElementD, LayoutD, AlignmentD,
      EpilogueScheduleType,
      EVTOutput
>::CollectiveOp;
```

Pre-baked aliases for common patterns

```
using Alpha = Sm90ScalarBroadcast<ElementScalar>;
using Accum = Sm90AccFetch;
using Bias  = Sm90ColBroadcast<
    0, TileShape, ElementBias, Stride<_1,_0,int>, AlignmentBias>;
using MultiplyAdd = Sm90Compute<
    multiply_add, ElementCompute, ElementCompute, RoundStyle>;
using EVTCompute0 = Sm90EVT<MultiplyAdd, Alpha, Accum, Bias>;


using Beta = Sm90ScalarBroadcast<ElementScalar>;
using C    = Sm90SrcFetch<ElementSource>;
using EVTCompute1 = Sm90EVT<MultiplyAdd, Beta, C, EVTCompute0>;


using ReLUAct = Sm90Compute<ReLu, ElementOutput, ElementCompute, RoundStyle>;
using EVTOutput = Sm90EVT<ReLUAct, EVTCompute1>;
```

```
using EVTOutput = Sm90LinCombPerRowBiasEltAct<
    TileShape, ReLu, ElementOutput, ElementCompute>;
```

Epilogue visitor tree in Python

1. Declare a basic GEMM

```
import cutlass
plan = cutlass.op.Gemm(
  element=torch.float32,
  layout=cutlass.LayoutType.RowMajor)
```

2. Define an epilogue as a Python function

```
def my_epilogue(accum, alpha, C, beta, bias):
  D = relu(alpha * accum + beta * C + bias)
  return D
```

3. Define types and shapes of each EVT operand/output

```
empty_mn = torch.empty(size=(m, n), dtype=torch.float32)
empty_bias = torch.empty(size=(m, 1), dtype=torch.float32)
examples_inputs = {
  "accum": empty_mn, "C": empty_mn, "D": empty_mn,
  "alpha": 1.0, "beta": 1.0, "bias": empty_bias,
}
```

4. Construct the EVT and assign it to the GEMM

```
plan.epilogue_visitor = cutlass.epilogue.trace(
  my_epilogue, examples_inputs)
```

5. Compile and run the kernel

```
A, B, C, D, bias = ...
visitor_args = {
    "alpha": 2.0, "beta": 0.0,
    "C": C, "D": D, "bias": bias
}
plan.run(A, B, C, D, visitor_args=visitor_args)
```

EVT handles writing optimized epilogue loops for you

Consumer store warpgroup pseudocode

```
for (int i = 0; i < NumEpiSubtiles; ++i) {
  Tensor tRS_rAcc_frg_mn = tRS_rAcc_frg(_,i);

  copy(tiled_s2r, ...); // Copy C from smem to registers



  // Compute epilogue on accumulator fragments
  tRS_rD_frg(epi_v) = compute(tRS_rAcc_frg_mn, ...);



  copy(tiled_r2s, ...); // Copy results from register to smem



  copy(params.tma_store_d, ...); // Copy D smem to gmem


}
```

EVT handles writing optimized epilogue loops for you

Consumer store warpgroup pseudocode

```
callbacks.begin();
for (int i = 0; i < NumEpiSubtiles; ++i) {
  Tensor tRS_rAcc_frg_mn = tRS_rAcc_frg(_,i);

  copy(tiled_s2r, ...); // Copy C from smem to registers

  callbacks.previsit(...);

  // Compute epilogue on accumulator fragments
  tRS_rD_frg(epi_v) = callbacks.visit(tRS_rAcc_frg_mn, ...);

  callbacks.reduce(...);  // Smem reduction callback entry

  copy(tiled_r2s, ...); // Copy results from register to smem

  callbacks.postreduce(...);

  copy(params.tma_store_d, ...); // Copy D smem to gmem

  callbacks.tma_store(..);
}
callbacks.end();
```

Using CUTLASS Profiler

Whether you are a C++ user or targeting PTX

- CUTLASS has a python based kernel emitter and a manifest to hold a bunch of kernels
- Autotuning strategy is to stamp out a set of candidates kernels and then …
- Use the CUTLASS profiler to pick the best kernel for your problems of interest
- It is also possible to dump ptx of the best performing kernel with `cuobjdump` or –DCUTLASS_NVCC_KEEP

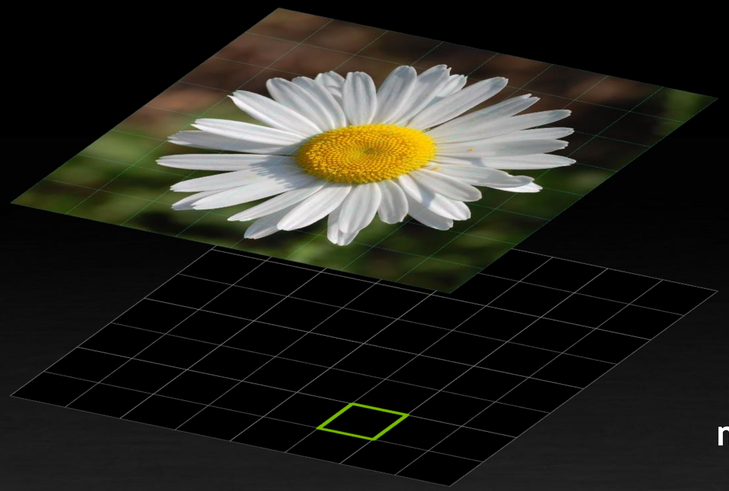
Hopper Programming Model

Courtesy : Stephen Jones

THE CUDA PROGRAMMING MODEL: GRID → BLOCKS → THREADS

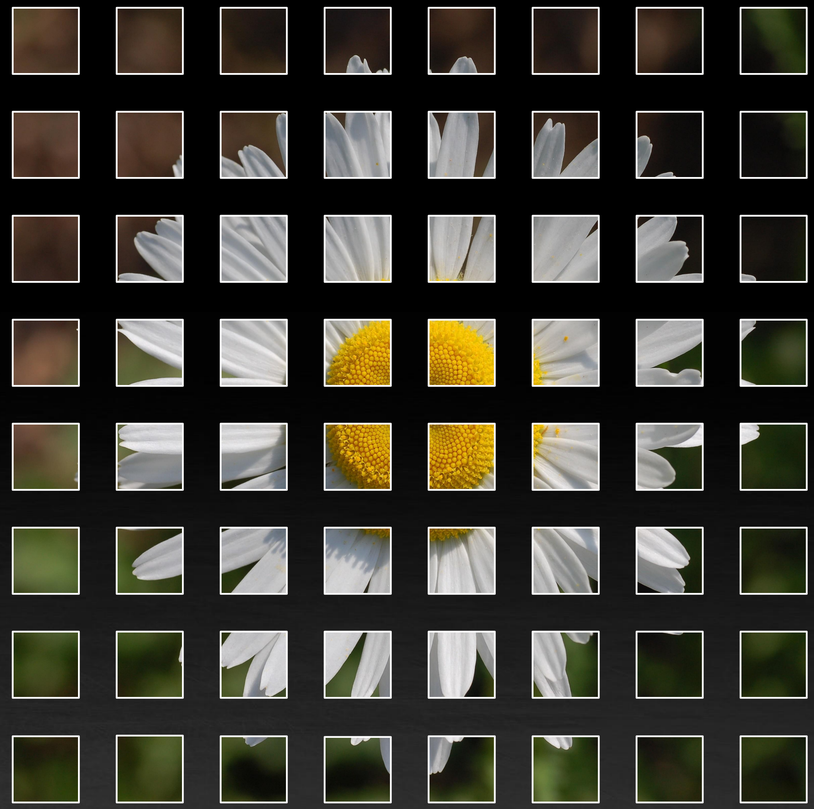
Grid
of work

DIVIDE THE WORK INTO A GRID OF EQUAL BLOCKS

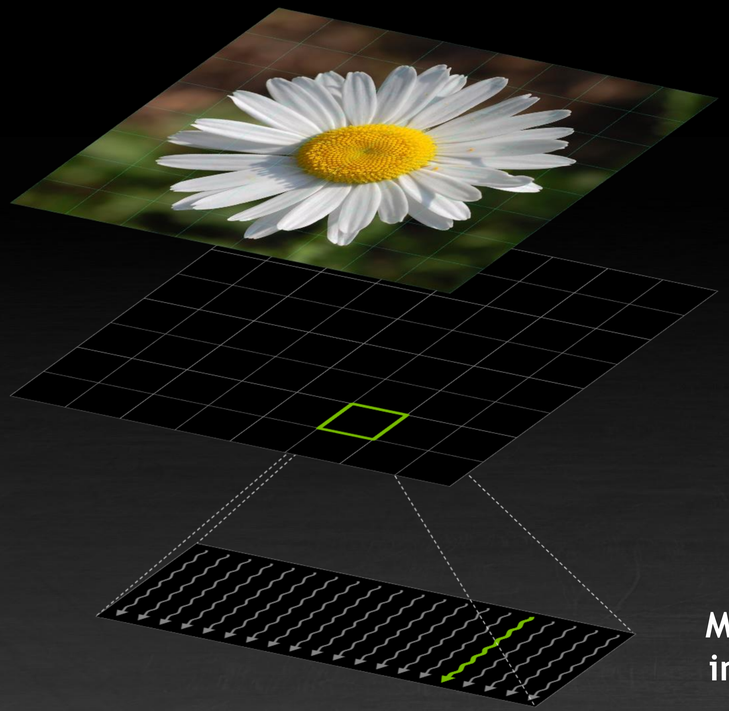


Grid
of work

**Divide into
many Blocks**



EACH BLOCK RUNS AS IF IT'S AN INDEPENDENT PROGRAM



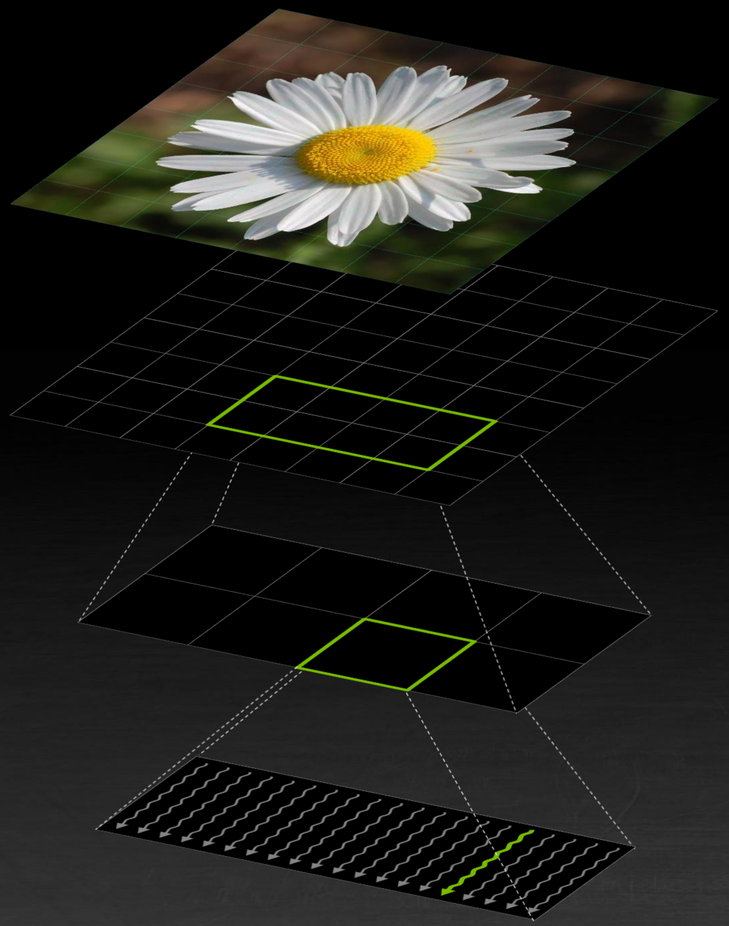
Grid
of work

Blocks
of Threads

**Many Threads
in each Block**

THREAD BLOCK CLUSTER

A collective of blocks, co-scheduled on adjacent multiprocessors

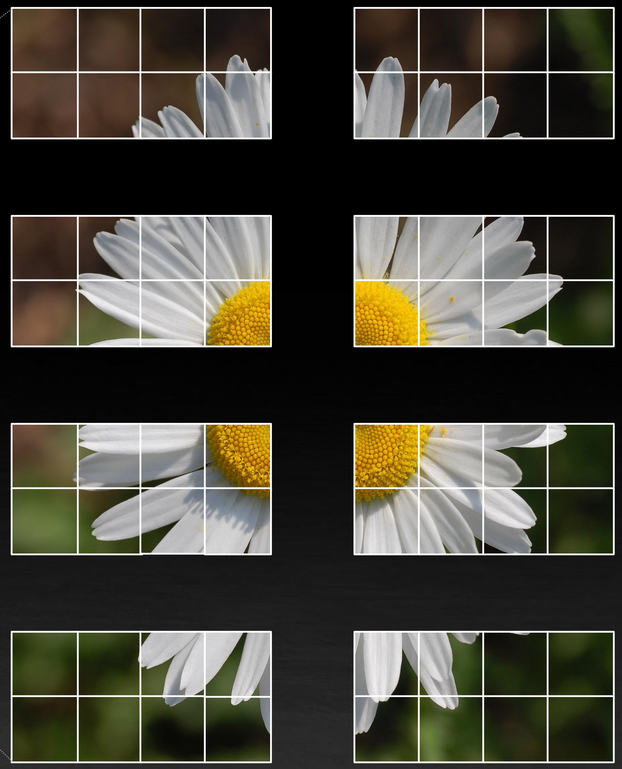


Grid
of work

**Cluster
of Blocks**

Blocks
of Threads

Threads



TAKING ADVANTAGE OF LOCALITY AT A GPU SCALE

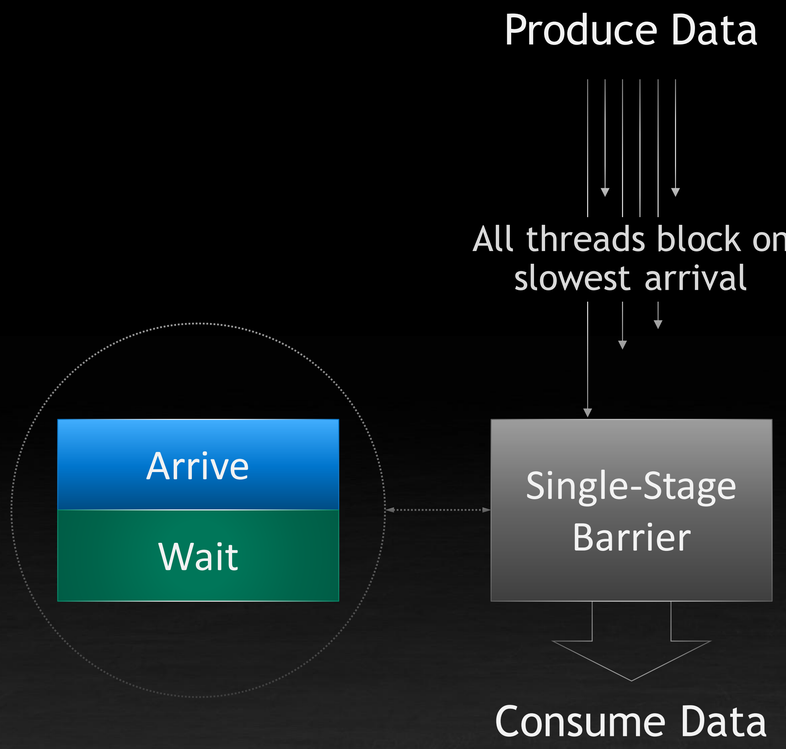
Guaranteed co-located blocks
New tier of guaranteed concurrency
Fast data exchange & sync

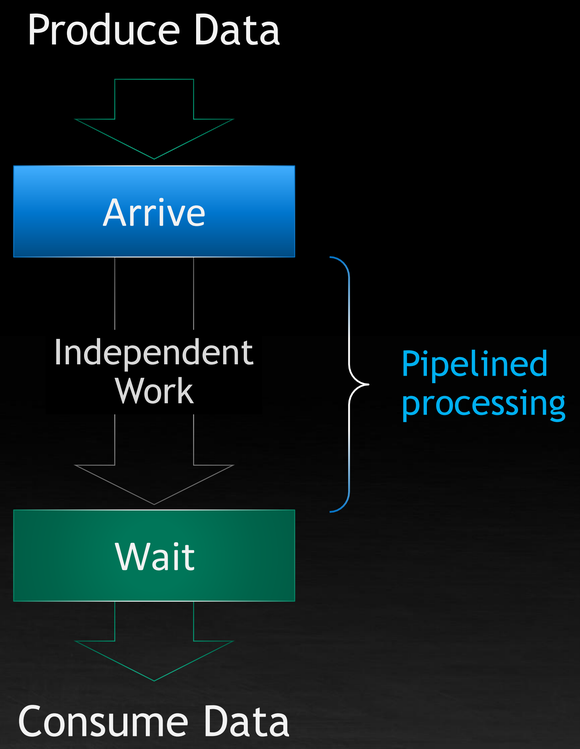
CLUSTER DISTRIBUTED SHARED MEMORY (DSMEM)

Blocks within a cluster are able to access each others' shared memory directly

Full load/store/atomic access to all shared memory between blocks within a cluster

ASYNCHRONOUS BARRIERS



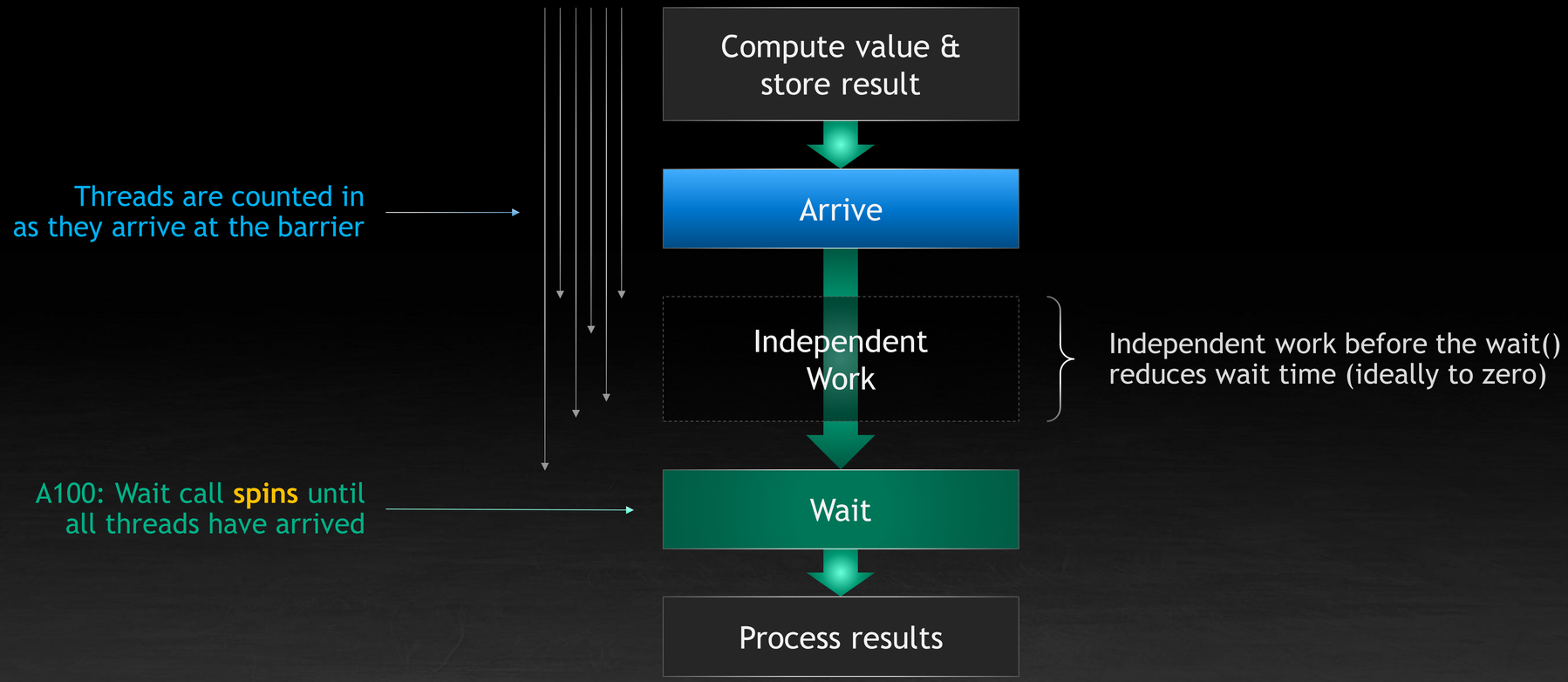


Single-Stage barriers combine back-to-back arrive & wait

Asynchronous barriers enable pipelined processing

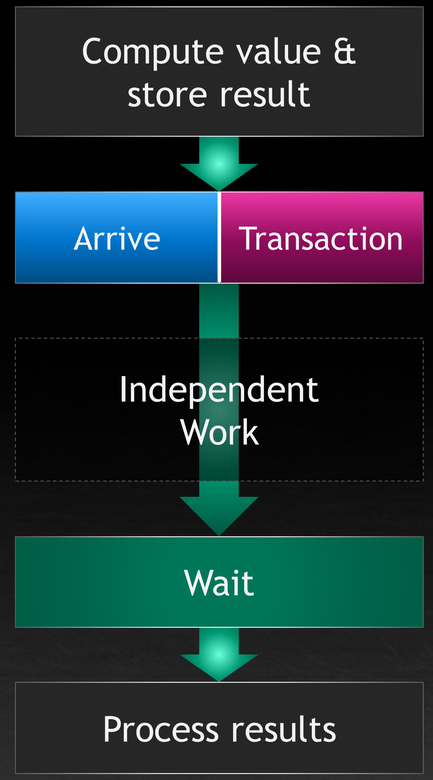
ASYNCHRONOUS BARRIERS

cuda::barrier



ASYNCHRONOUS TRANSACTION BARRIERS

Trigger barrier with both transactions and threads



Arrival count can include threads from **other** blocks in the cluster

Async writes to shared memory increment **transaction count** both within one block or between one or more blocks in a cluster

Wait call **sleeps** until **both** conditions are satisfied:

1. All threads have arrived
2. Transaction count reaches expected value

Drop-in enhancement to existing cuda::barrier

TENSOR MEMORY ACCELERATOR UNIT (TMA) FOR ASYNC DATA MOVEMENT

Tensor Memory Accelerator Unit

Hardware-accelerated bi-directional bulk copy

- Global ←→ Shared Memory
- DSMEM ←→ DSMEM within cluster

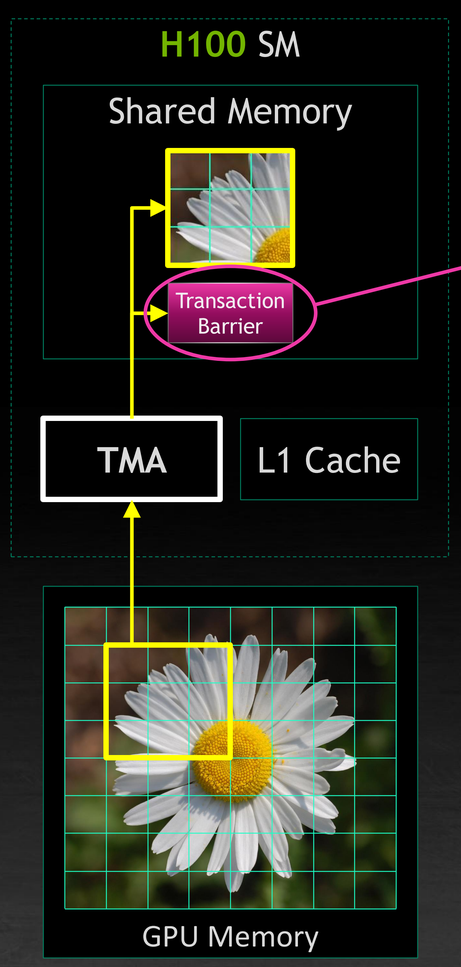
Uses Asynchronous Transaction Barrier to track completion

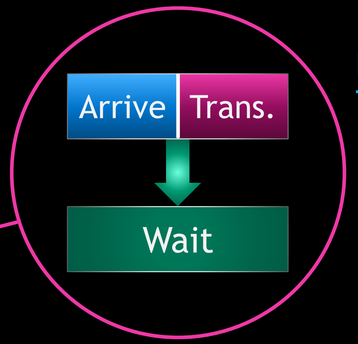
Faster And More Flexible Async Copying

Drop-in enhancement to existing cuda::memcpy_async() function

Single thread triggers any-size copy – no need for looping or collective copying

Multiple copy operations can contribute to a single cluster-wide async transaction barrier





New Asynchronous Transaction Barrier

```
__shared__ cuda::barrier barrier; // Pre-initialised
auto block = cg::this_thread_block();

// On H100, memcpy_async automatically issues TMA
// enhanced copy and updates transaction count
if(threadIdx.x == 0) {
    cuda::memcpy_async(block,
                       smem,
                       gmem,
                       block.size() * sizeof(int),
                       barrier);
}
```

Same copy syntax becomes faster and more flexible
New syntax to enable TMA capabilities

HOPPER TENSOR CORES

NVIDIA - H100

New Faster Tensor Core Instructions

16b Floating-point Tensor Core operations 16x and 32x faster than F32 CUDA Cores

8b Floating-point Tensor Core operations 32x and 64x faster than F32 CUDA Cores

Improved Integer Tensor Core operations 32x and 64x faster than F32 CUDA Cores

Improved Tensor Float 32 Tensor Cores 8x and 16x faster than F32 CUDA Cores

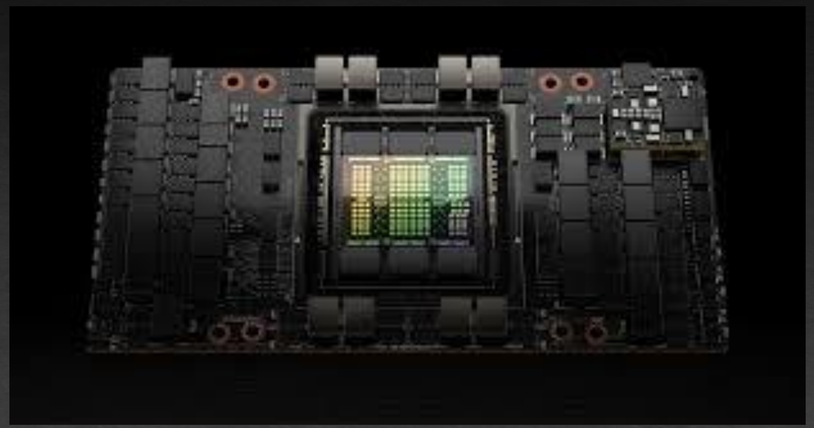
Improved IEEE double-precision Tensor Cores 2x faster than F64 CUDA Cores

Additional Data Types

8bit Floating point types E5M2 and E4M3

Many additional new features – see "NVIDIA H100 Tensor Core GPU Architecture" whitepaper

Applicable with using 2:4 sparse computation



NVIDIA

NVIDIA HOPPER ARCHITECTURE - TENSOR CORE OPERATIONS

Operation	Data Types (A * B + C)	Shape	TFLOPS Hopper (H100)	TFLOPS Ampere (A100)	TFLOPS Volta (V100)
MMA wgmma, mma_sync	F16 * F16 + F16 F16 * F16 + F32 BF16 * BF16 + F32	64-by-N-by-16	1000	312	125
MMA wgmma, mma_sync	TF32 * TF32 + F32	64-by-N-by-8	500	156	N/A
MMA mma_sync	F64 * F64 + F64	16-by-8-by-4/8/16	60	19	N/A
MMA wgmma, mma_sync	S8 * S8 + S32	64-by-N-by-32	2000	624	N/A
MMA wgmma, mma_sync	F8 * F8 + F32	64-by-N-by-32	2000	N/A	N/A

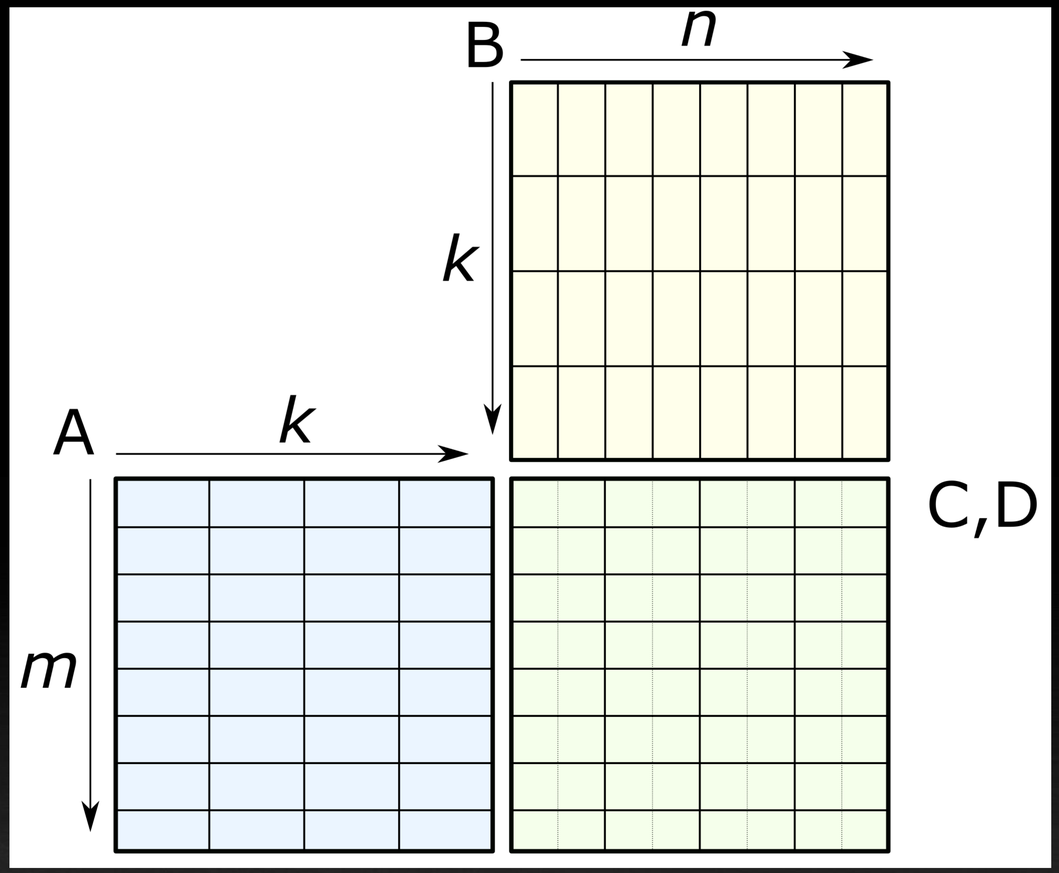
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#asynchronous-warpgroup-level-matrix-instructions

TENSOR CORES OPERATION : FUNDAMENTAL SHAPE

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

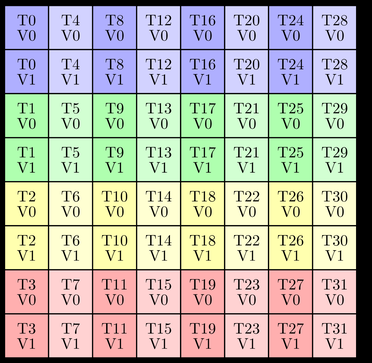
- Matrix multiply-add
- XOR-POPC

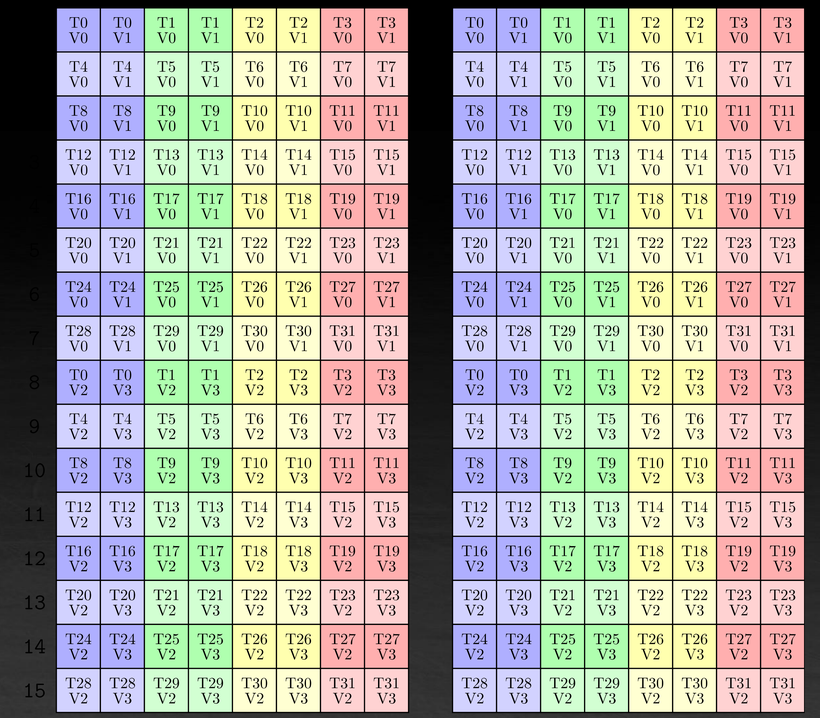
- M-by-N-by-K matrix operation
- Warp-Group-wide, async. collective operation.
- 128 threads within warp-group collectively holds D operand.
- Operands A and B (not both) can optional be loaded from register memory. Loading directly from shared memory using descriptors most optimal.
- Valid value(s) of M = 64; N ∈ [8, 256] and K ∈ [8, 16, 32, 256] based on data type refer ISA documentation for details



AMPERE/TURING FP16 MMA

```
template <>
struct MMA_Traits<SM80_16x8x8_F32F16F16F32_TN>
{
  using ElementDVal = float;
  using ElementAVal = half_t;
  using ElementBVal = half_t;
  using ElementCVal = float;

  using Shape_MNK = Shape<_16,_8,_8>;

  using ThrID = Layout<_32>;


  // (T32,V4) -> (M16,K8)
  using ALayout = Layout<Shape <Shape < _4,_8>, Shape < _2,_2>>,
                         Stride<Stride<_32,_1>, Stride<_16,_8>>>;

  // (T32,V2) -> (N8,K8)
  using BLayout = Layout<Shape <Shape < _4,_8>,_2>,
                         Stride<Stride<_16,_1>,_8>>;

  // (T32,V4) -> (M16,N8)
  using CLayout = Layout<Shape <Shape < _4,_8>, Shape < _2,_2>>,
                         Stride<Stride<_32,_1>, Stride<_16,_8>>>;
};
```

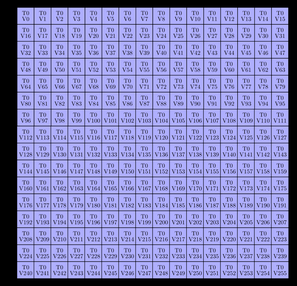


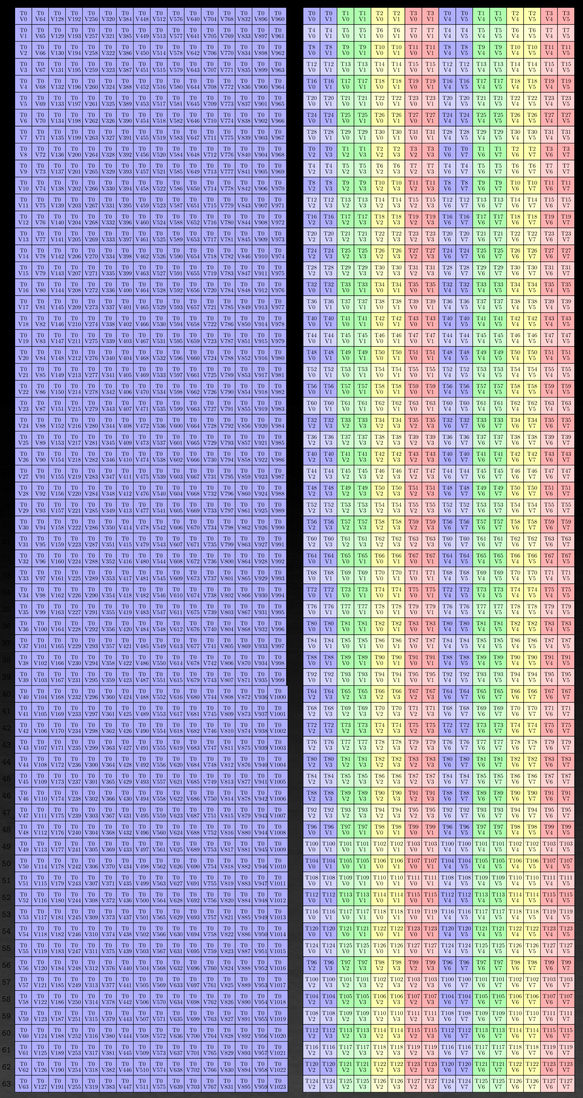


SM80_16x8x8_F32F16F16F32_TN

HOPPER FP16 MMA

```
template <GMMA::Major tnspA, GMMA::Major tnspB, GMMA::ScaleIn scaleA,
GMMA::ScaleIn scaleB>
struct MMA_Traits<SM90_64x16x16_F16F16F16_SS<tnspA, tnspB, scaleA, scaleB>
{
  using ValTypeD = half_t;
  using ValTypeA = half_t;
  using ValTypeB = half_t;
  using ValTypeC = half_t;

  using Shape_MNK = Shape<_64,_16,_16>;

  using ThrID = Layout<_128>;



  using ALayout = GMMA::ABLayout< 64, 16>;


  using BLayout = GMMA::ABLayout< 16, 16>;;


  using CLayout = Layout<Shape <Shape <  _4,_8, _4>,Shape < _2,_2,  _2>>,
                         Stride<Stride<_128,_1,_16>,Stride<_64,_8,_512>>>;
};
```





Hopper MMA :

F16 * F16 + F32

64-by-N-by-16

- **wgmma.mma_async** issues a warp-group wide asynchronous MxNxK matrix multiply and accumulate operation,

  D = A * B + D

  Where A matrix is MxK, B matrix is KxN, and D matrix is MxN.

- Where N ∈ [8, 256], and M = 64.

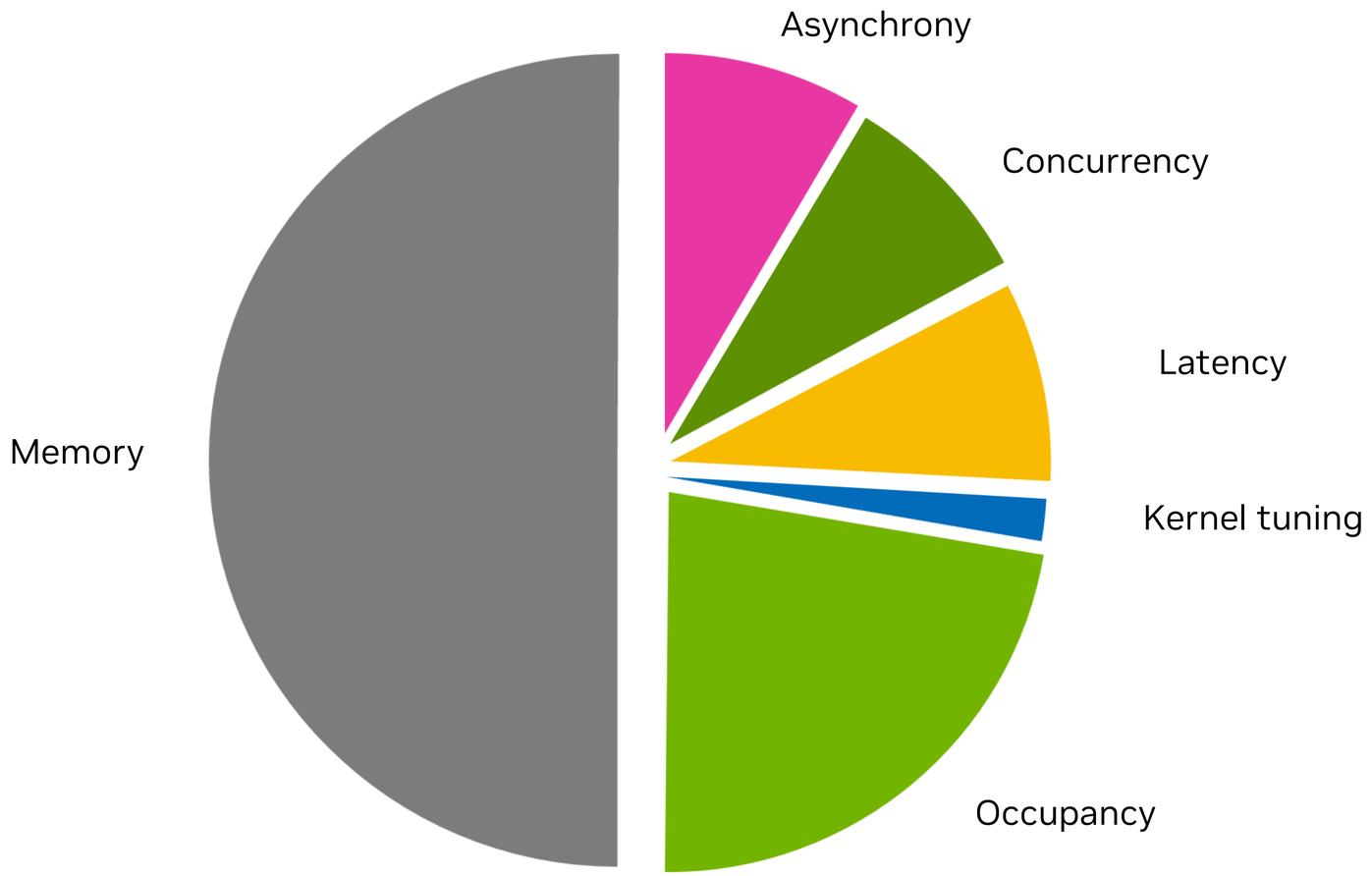
- A and B matrices are loaded directly from Shared Memory using descriptors.
- Transposition and scaling with certain immediate values supported as part of the instruction.

```
  uint64_t descpritor_a, descpritor_b;
  float D[64];
  constexpr int scale_a, scale_b;      // -1 or +1
  constexpr uint32_t scale_d;          //  0 or  1
  constexpr uint32_t trans_a, trans_b; //  0 or  1

  asm (
    "wgmma.mma_async.sync.aligned.m64n64k16.f32.f16.
f16 "
    "{%0,  %1,  %2,  %3,  %4,  %5,  %6,  %7,  "
    " %8,  %9,  %10, %11, %12, %13, %14, %15, "
    ....
    ....
   " %56, %57, %58, %59, %60, %61, %62, %63},"
    " %64,"
    " %65,"
    " %66, %67, %68, %69, %70;\n"
    : "+f"(D[0]),"+f"(D[1]),"+f"(D[2]),"+f"(D[3]),
      "+f"(D[4]),"+f"(D[5]),"+f"(D[6]),"+f"(D[7]),
    ....
    ....
    "+f"(D[60]),"+f"(D[61]),"+f"(D[62]),"+f"(D[63])
    :
    "l"(descriptor_a),"l"(descriptor_b),
    "n"(scale_d), "n"(scale_a), "n"(scale_b),
    "n"(trans_a), "n"(trans_b)
  );
```

Ref. **cute::SM90_64x128x16_F32F16F16_SS**

Anatomy of a High Performance GEMM

What I Care About When Programming The GPU

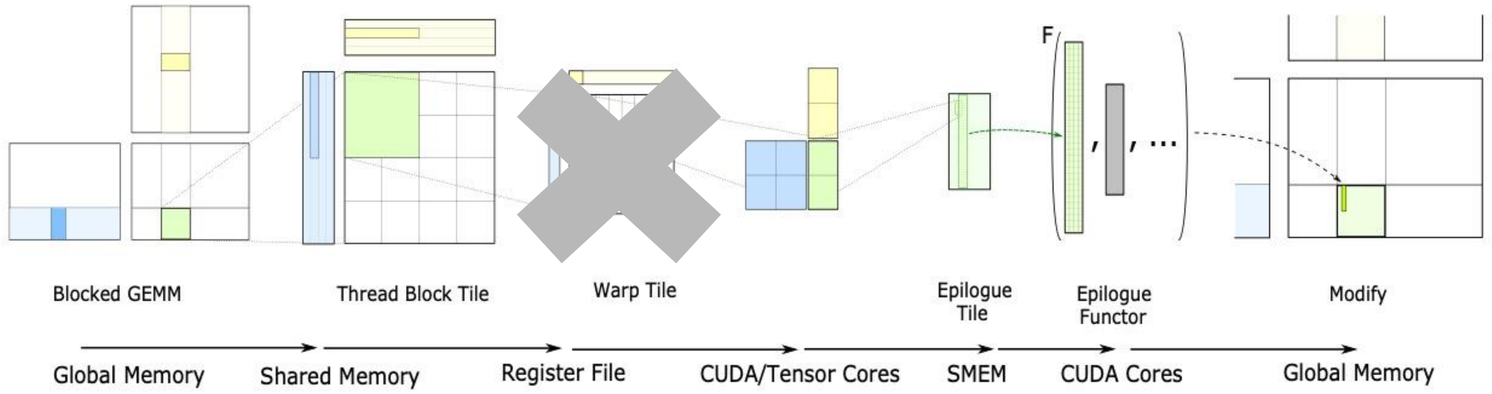


Anatomy of a High Performance Hopper GEMM

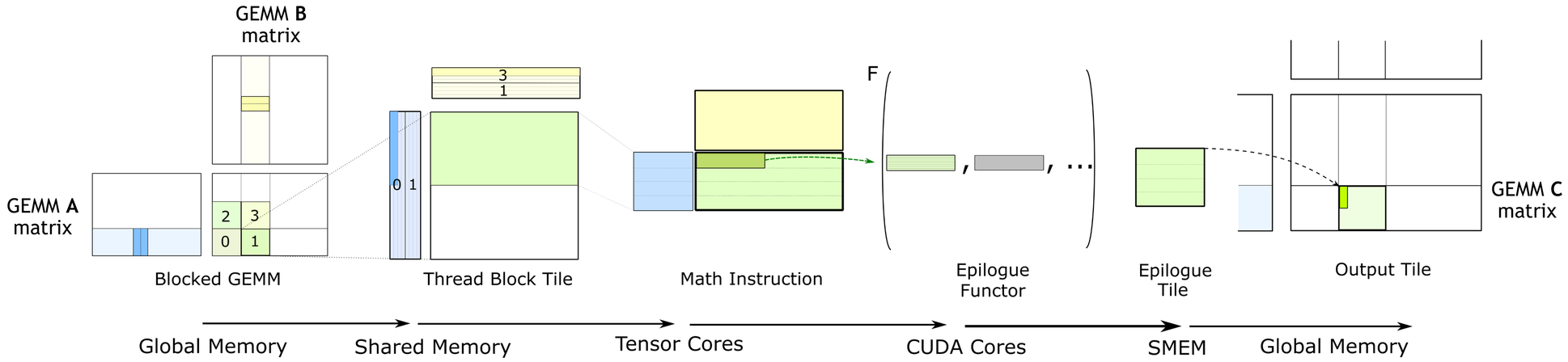
101 Level Optimizations

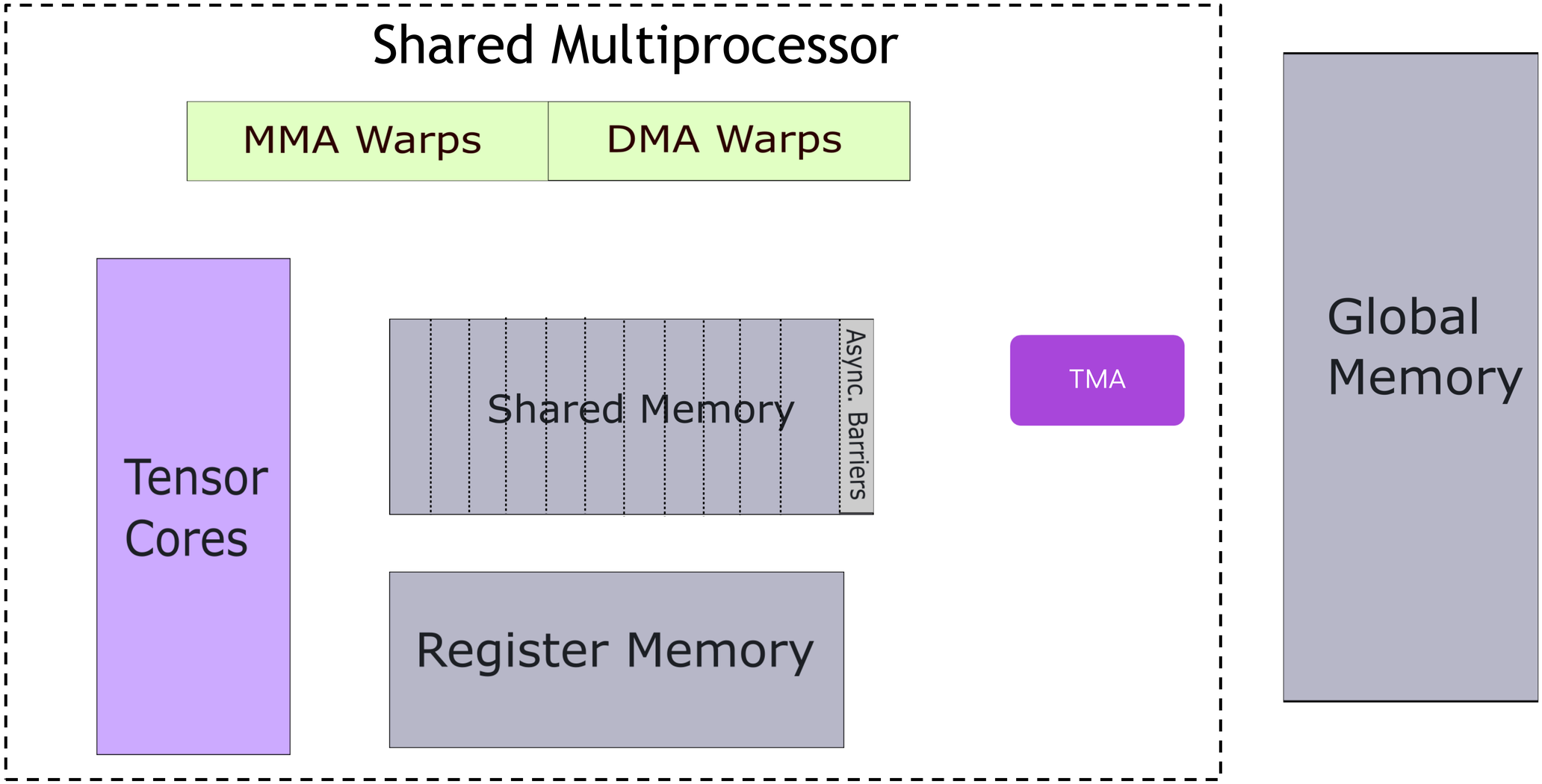
- Global memory accesses via TMA
  - HW supported Out-Of-Bounds determination a.k.a predication
  - Reduced register pressure and ALU ops
- Threadblock clusters used in concert with TMA to allow for programmatic multicast.
  - Amplification of L2 bandwidth via exploitation of temporal / spatial locality at the cost of quantization, occupancy.
- Deep software pipelining of loads via shared memory
  - Better Latency coverage at the cost of fewer ThreadBlocks running concurrently
- Warp-Group MMA operation
  - Multiple asynchronous MMA instructions in flight
- Mbarrier based synchronization between Async. TMA producers & MMA consumers
- Swizzling of data in shared memory to avoid bank conflicts
  - Prescribed together by MMA, TMA the tile size and the data-type

GEMM Data movement

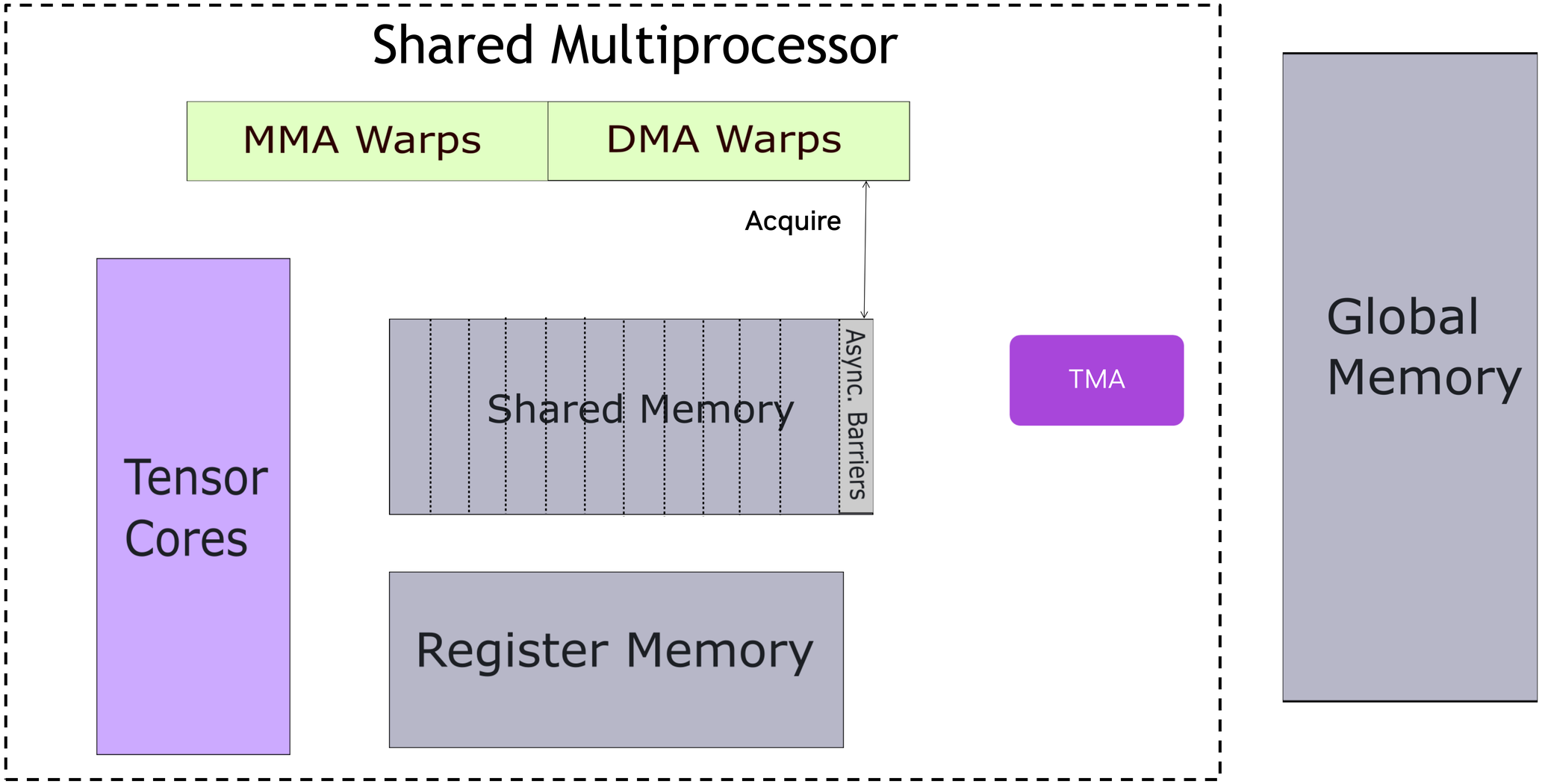


Tensor cores reuse data directly from in shared memory (if possible)

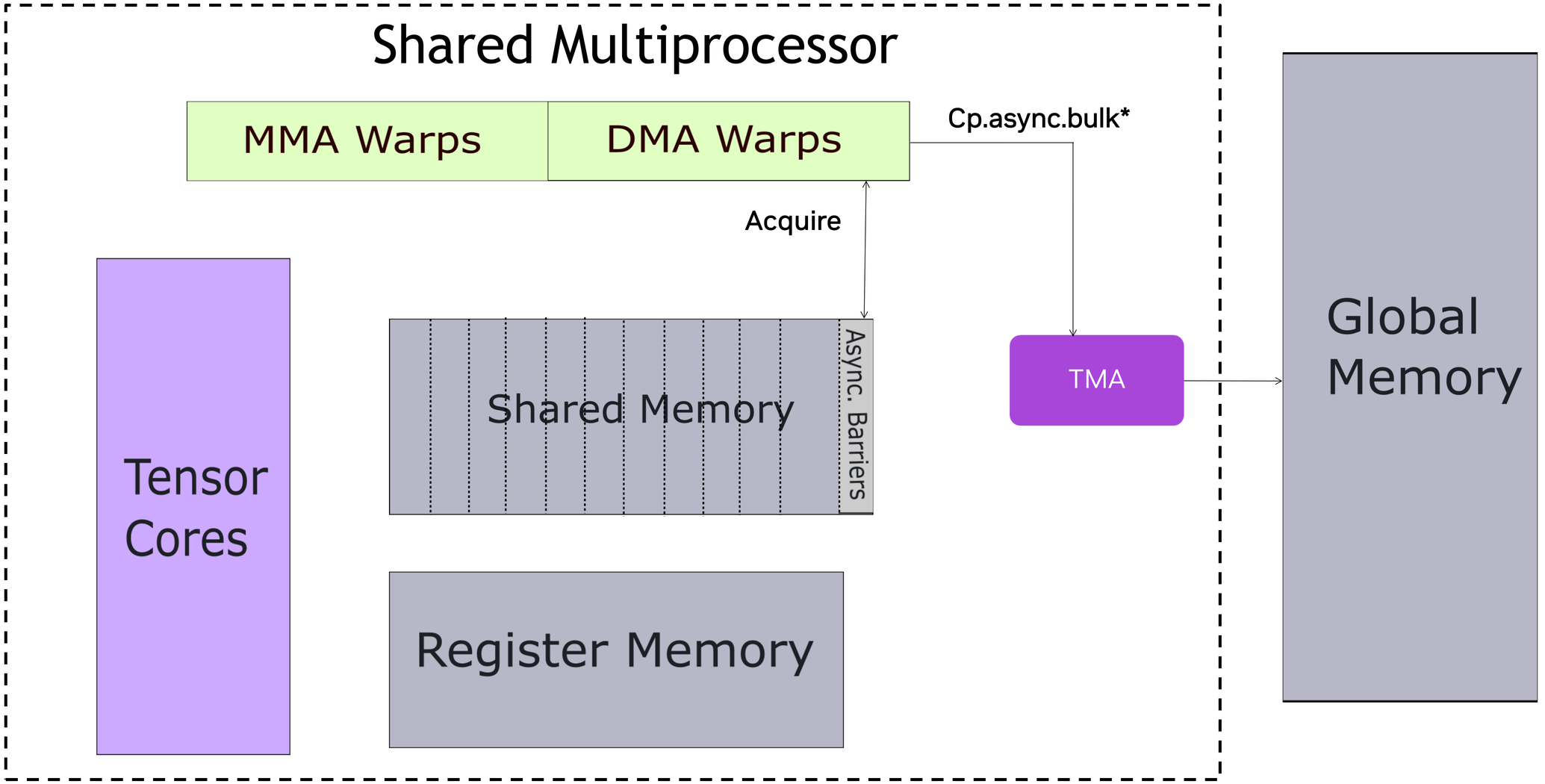




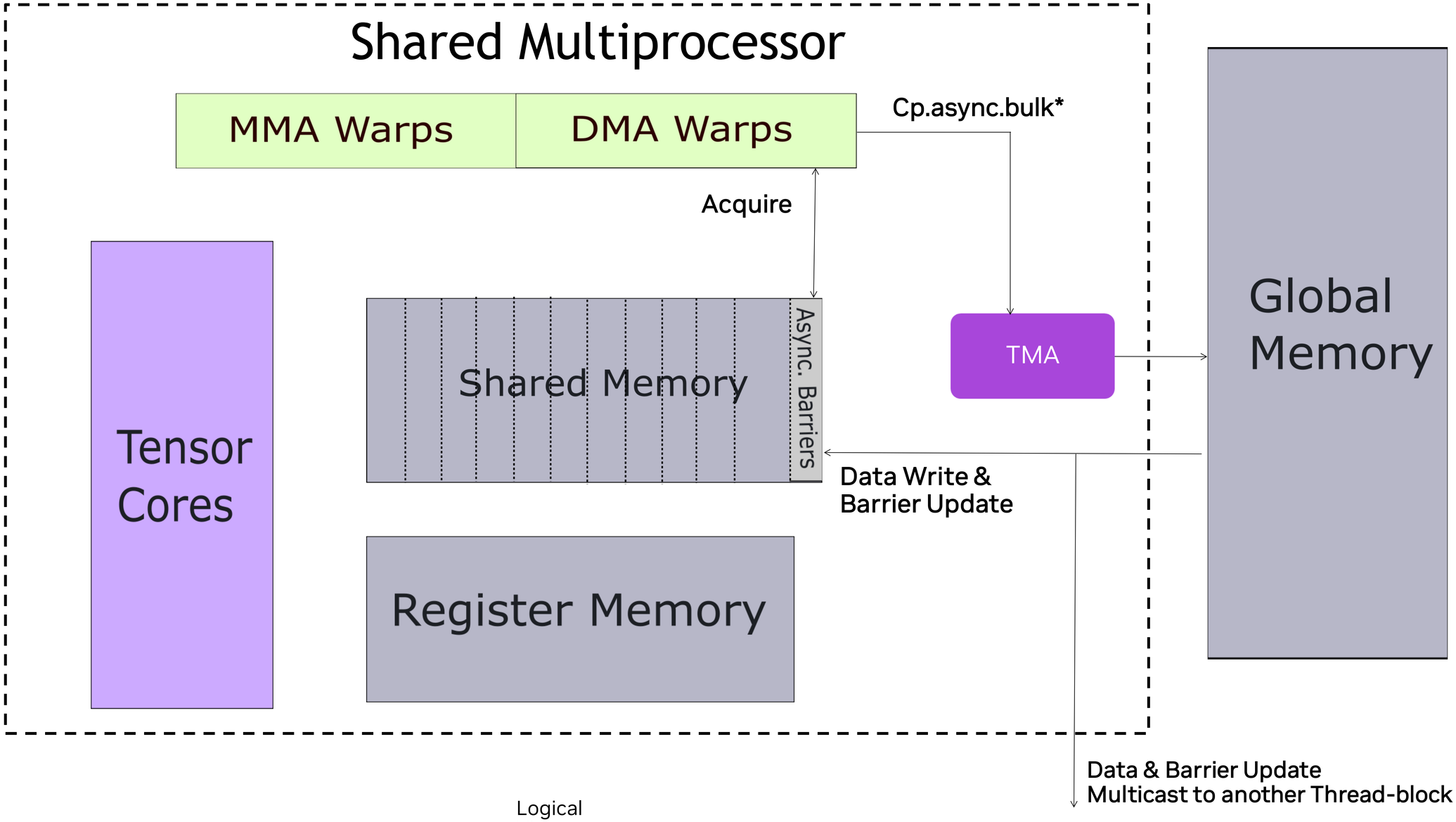
Logical View



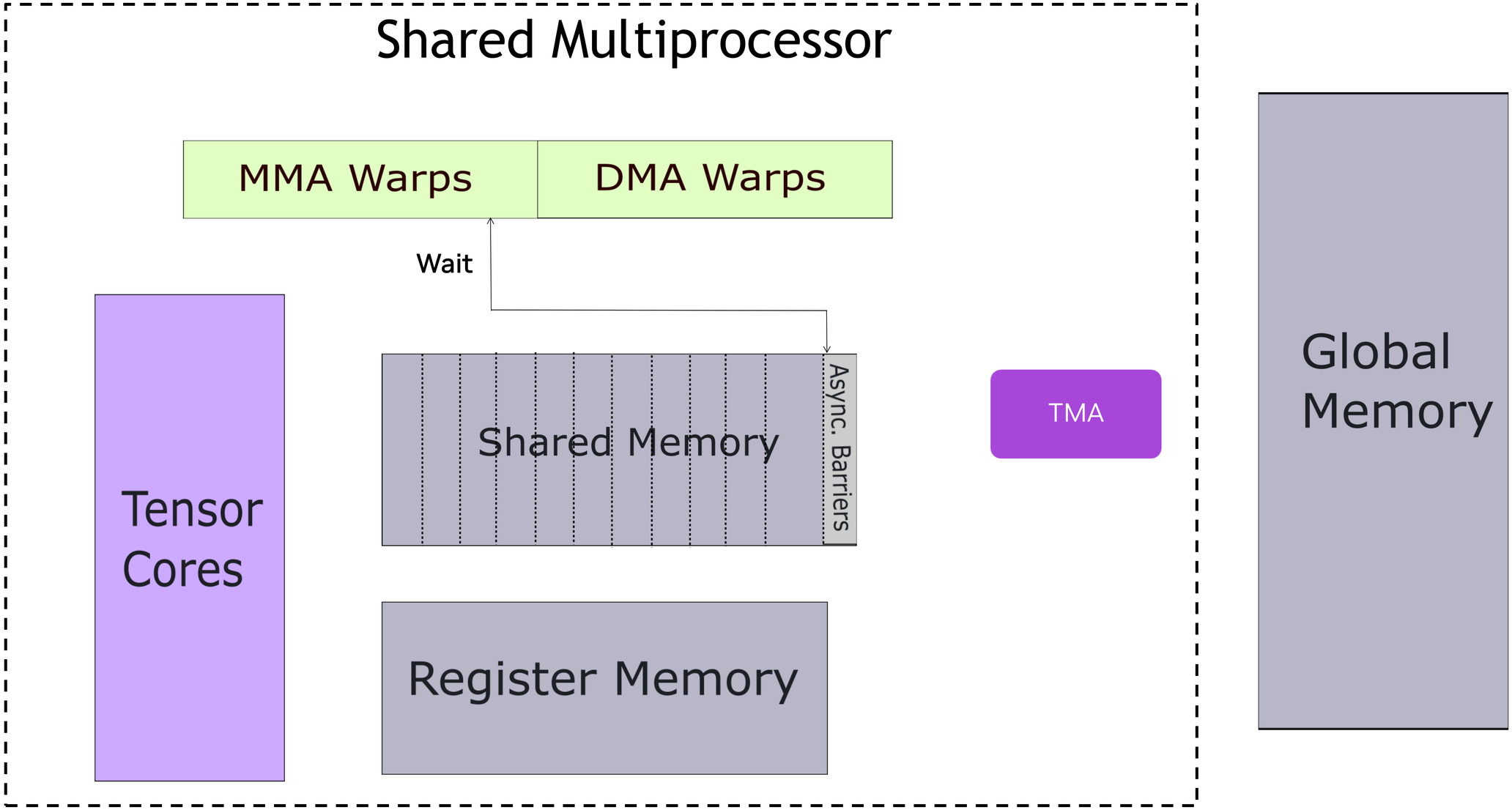
Logical View



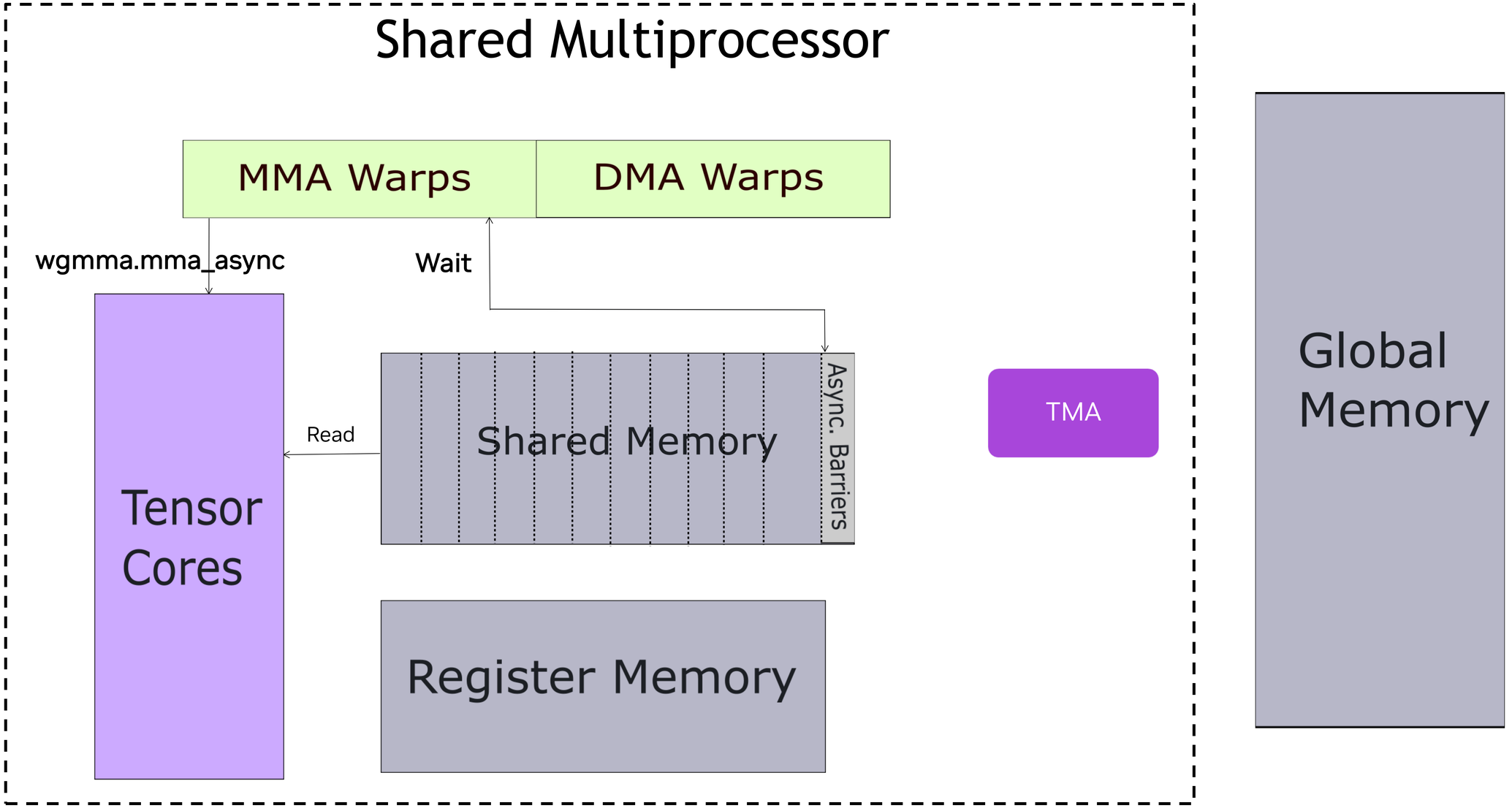
Logical View



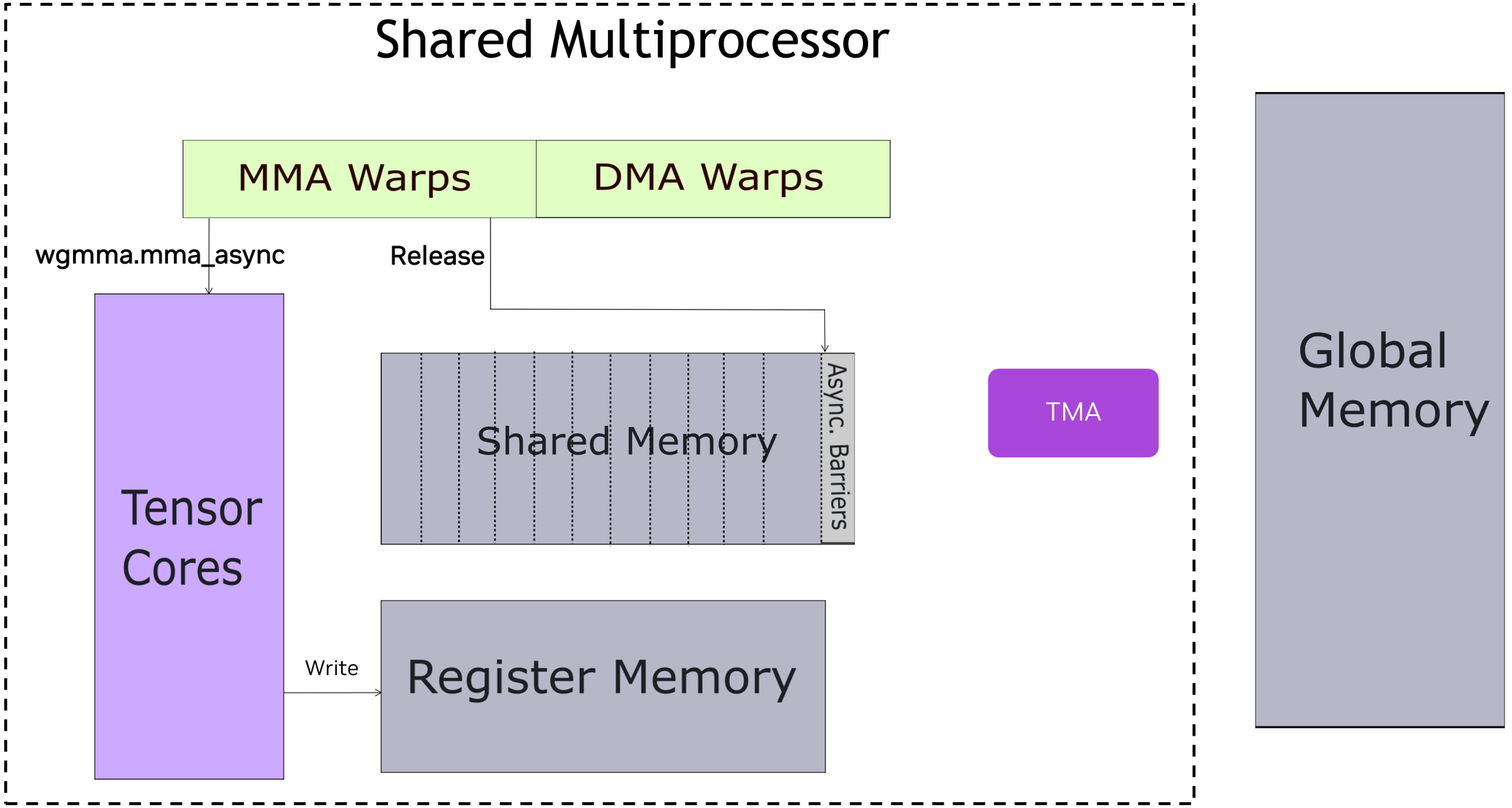
Logical View



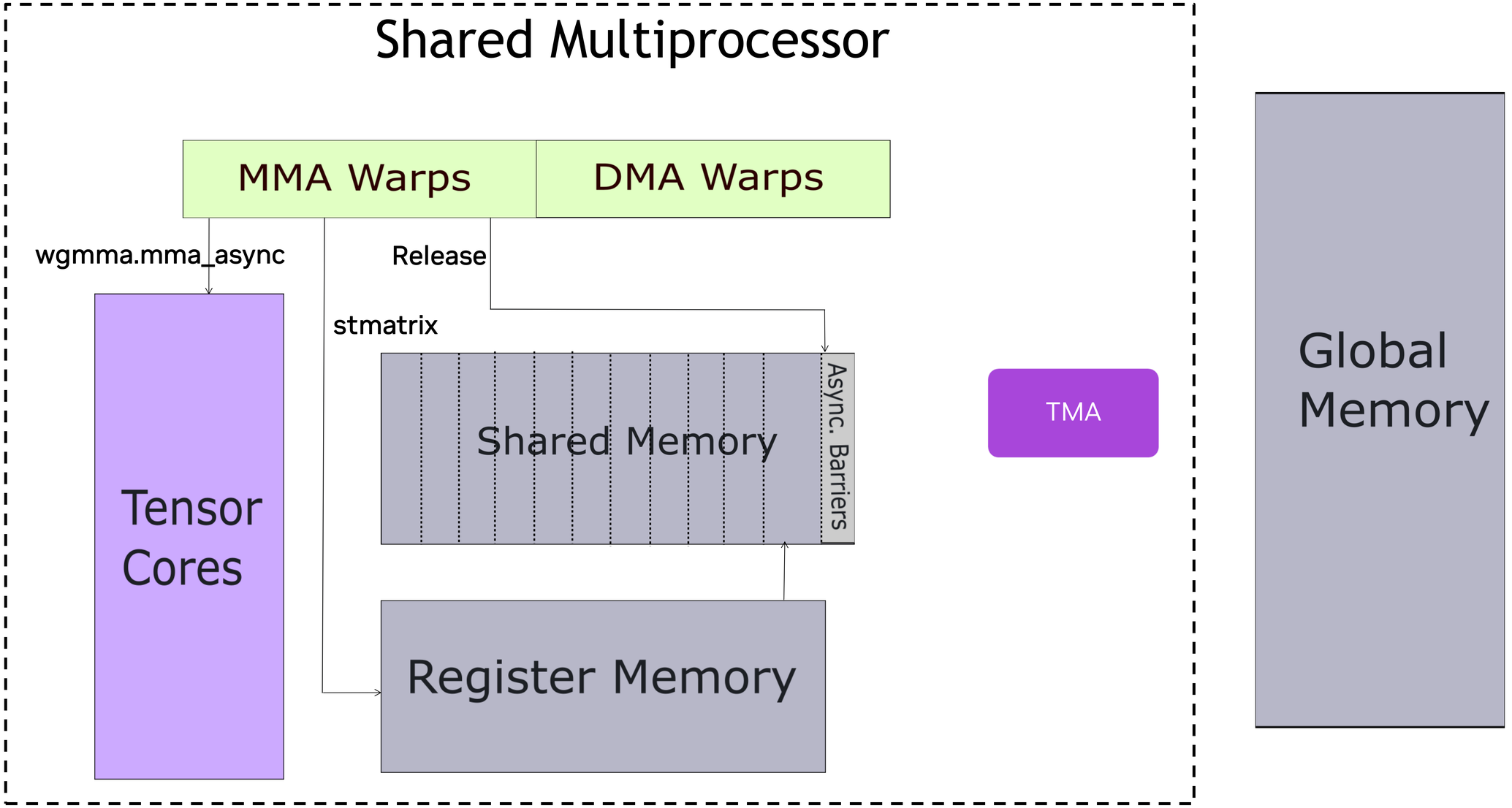
Logical View



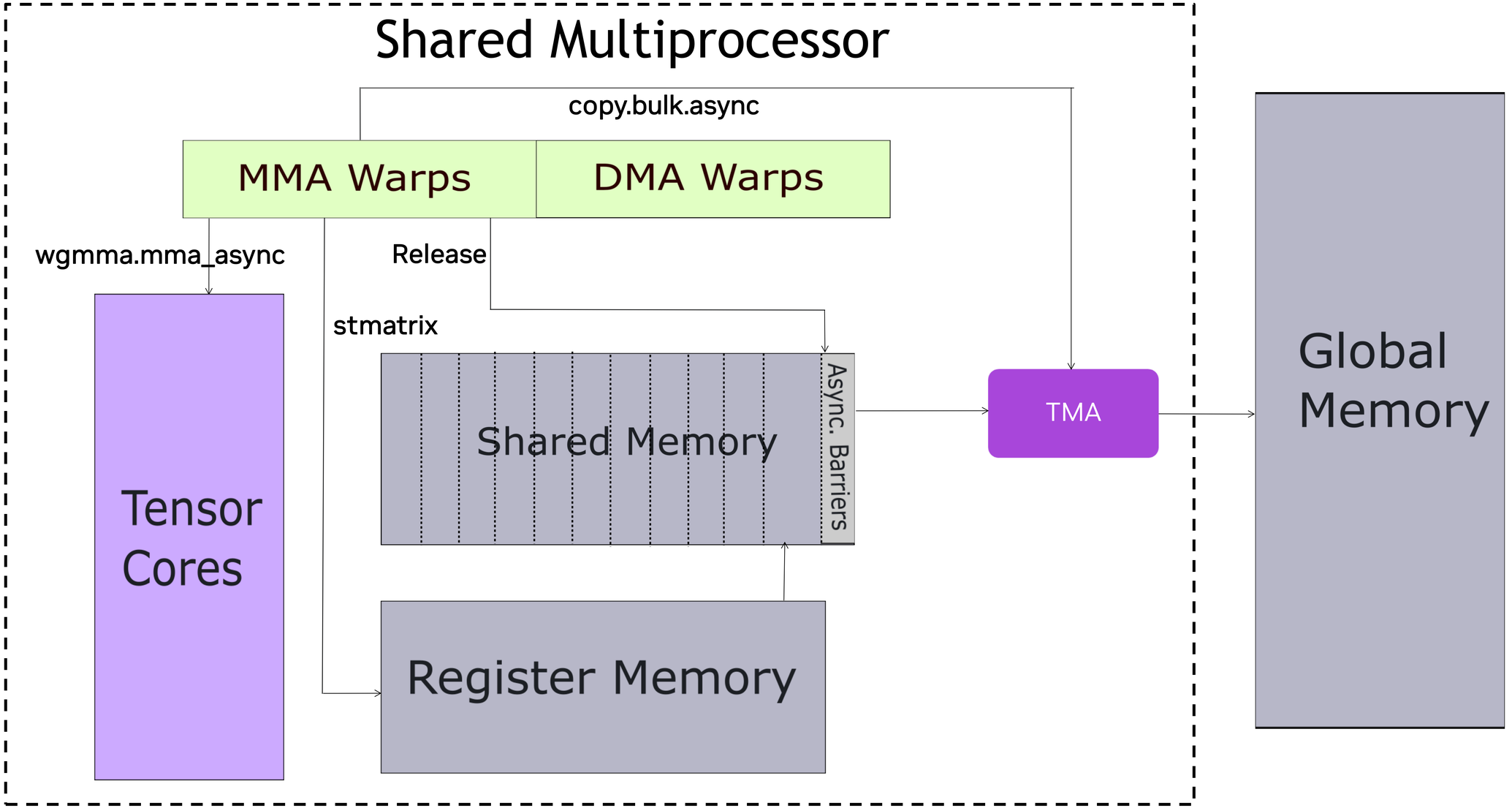
Logical View



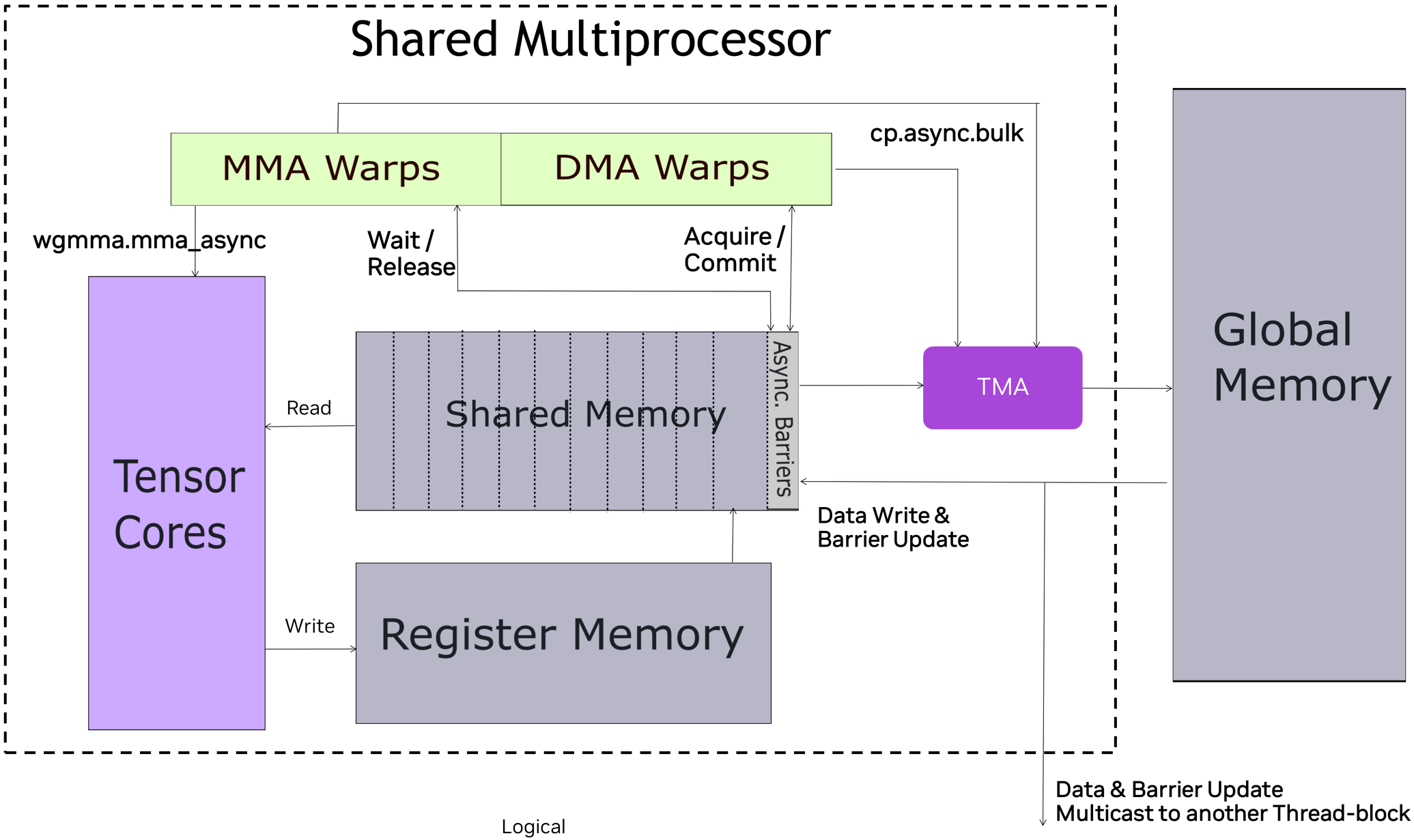
Logical View



Logical View



Logical View



Logical View

Hopper Tensor Core GEMM in ~100 lines

Putting it all together

A **basic** GEMM kernel in ~100 lines

Code Walk Through

Reading material :

- https://github.com/NVIDIA/cutlass/blob/main/media/docs/cute/
- https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51413/
  https://www.nvidia.com/en-us/on-demand/session/gtc24-s61198/
- https://github.com/NVIDIA/cutlass/pull/1578
-

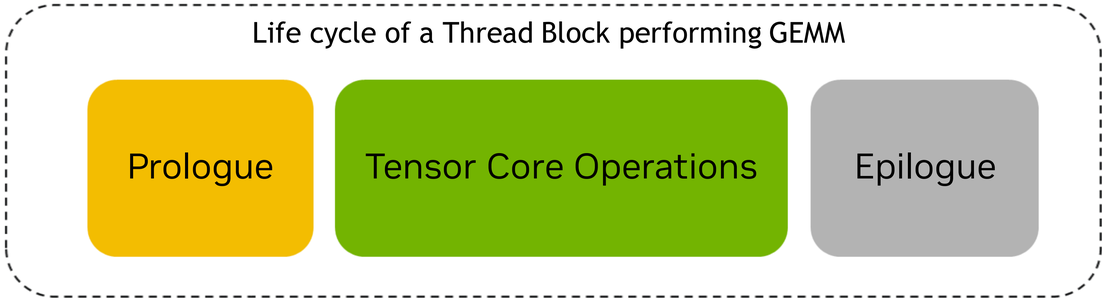
GEMM Advance optimizations

Anatomy of a High Performance Hopper GEMM

201 Level Optimizations

- Warp Specialization
  - Optimal memory and compute throughput across a wide variety of scenarios.
- Persistent schedules
  - Amortizes hardware and software pipelining fixed costs in a kernel
- Epilogue hiding in the shadow of MMAs (a.k.a Ping-Pong Schedule)
  - Critical for small yet compute bound GEMMs
- Maximizing tile size to improve efficiency (a.k.a Cooperative Schedule)
  - Better handling of register pressure
- Efficient usage of synchronization primitives

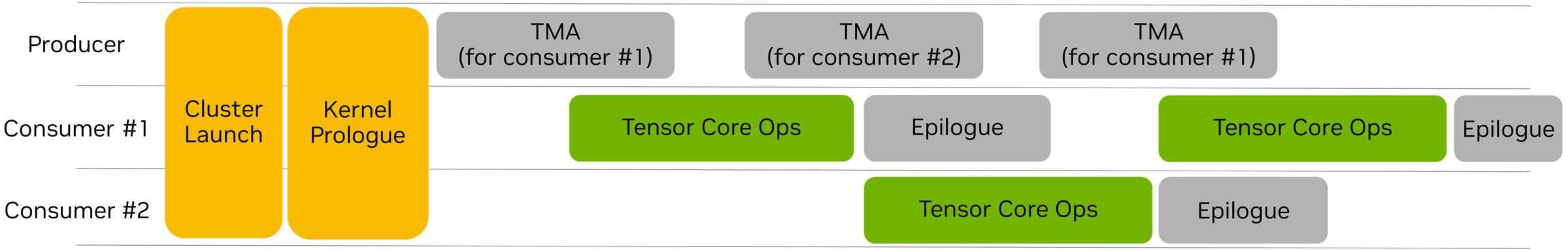
Life of a GEMM kernel



- Prologue and Epilogue are components of the GEMM kernel which involve non-tensor core operations and often latency or bandwidth bound.
- Typically hidden via multiple concurrently running ThreadBlock / SM
- With deep software pipelines – it becomes a tricky problem (due to lack of shared memory capacity)

Warp-Specialized - Persistent Ping-Pong Kernels

- 2 consumers warp-groups *work on two different work tiles* but ping-pong mainloop & epilogue
- Leads to guaranteed overlapping of epilogue of one consumer with the MMAs of the other
- Always keeps the tensor-cores busy – but at the cost of higher register pressure.



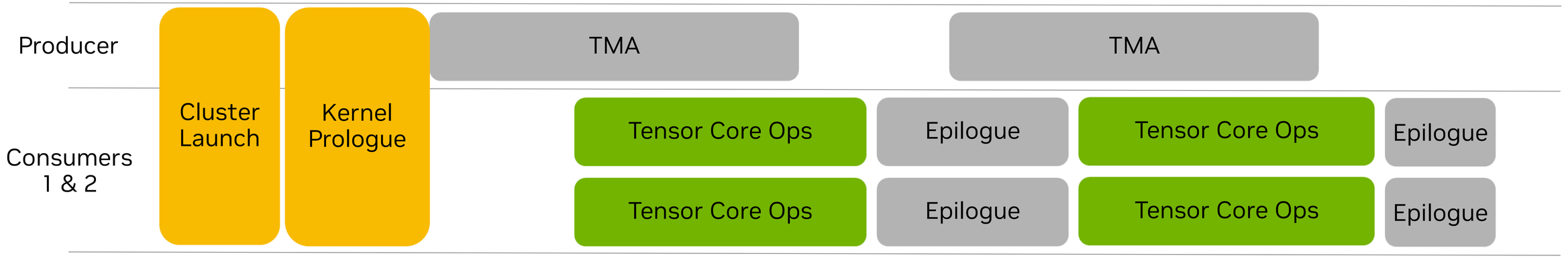
Producer	Consumer #1	Consumer #2
PersistentTileSchedulerSm90 **scheduler(problem_shape, blk_shape, cluster_shape)**		

```
// Data in via TMA

while (work_tile_info.is_valid_tile) {
  collective_mainloop.dma()
  scheduler.advance_to_next_work()
  work_tile_info = scheduler.get_current_work()
}
```

```
// Mainloop, epilogue, and data out

while (work_tile_info.is_valid_tile) {
  collective_mainloop.mma()
  scheduler.advance_to_next_work(NumConsumers)
  work_tile_info = scheduler.get_current_work()
}
```

Warp-Specialized - Persistent Cooperative Kernels

- 2 Consumers each cooperatively each work on *their half of the same work tile*
- 2 consumer warp-groups run in concert and so math and epilogue sections execute ~together
- Reduced register pressure in each consumer – which can optionally be utilized for larger tile sizes



Producer	Consumer #1	Consumer #2
PersistentTileSchedulerSm90 **scheduler(problem_shape, blk_shape, cluster_shape)**		

```
// Data in via TMA

while (work_tile_info.is_valid_tile) {
  collective_mainloop.dma()
  scheduler.advance_to_next_work()
  work_tile_info = scheduler.get_current_work()
}
```

```
// Mainloop, epilogue, and data out

while (work_tile_info.is_valid_tile) {
  collective_mainloop.mma()
  scheduler.advance_to_next_work(NumConsumers)
  work_tile_info = scheduler.get_current_work()
}
```

Anatomy of a High Performance Hopper GEMM

501 Level Optimizations

- Optimal ThreadBlock rasterization & swizzling
  - Encourages exploitation of locality
  - https://github.com/NVIDIA/cutlass/blob/main/media/docs/efficient_gemm.md#threadblock-rasterization
- Stream-K scheduling
  - Finding the optimal Trade-off between occupancy and efficiency
  - https://arxiv.org/abs/2301.03598
- Efficient Input transformations prior to MMA
  - Optimizing for register usage
  - Pipelined RS kernels
  - https://github.com/NVIDIA/cutlass/blob/main/include/cutlass/gemm/collective/sm90_mma_tma_gmma_rs_warpspecialized.hpp
- Optimal instruction sequence generation
  - Prefetching, Cache management

When to use which recipe?

*Caveat: For well behaved kernels

- Persistent kernels are great! Almost always a win
- “Small” K shape with large MN shape -> persistent ping pong schedule to hide prologue/epilogue overheads
  - Heavy epilogue fusions even for large K shape? Pingpong persistent
- “Large” K shape with large MN shape -> Cooperative persistent
  - Multistage persistent for even larger acc tiles
  - Naïve direct store epilogues to give mainloop even more smem capacity can be a win
- Small MN and large K shape -> cooperative with stream K load balancing
- HBM b/w bound MN shape? -> cleverer tile rasterization for L2 locality
- For fp8 training, follow the CUTLASS FP8 recipe that does higher precision accumulation
- For fp8 quant/dequant, see CUTLASS epilogues fusing amax and aux tensors

Other rules of thumb

- Design around 1 CTA / SM occupancy to start with – only go to higher as a last resort
- Cluster sizes of 2 are almost always a win. Start there with TMA multicast on the largest tile dim
- For larger tile sizes, prefer larger WGMMA N shapes over tiling over M
- Use larger swizzles for WGMMA and TMA if possible (a function of tile shapes)

- Source both A and B from smem if you can
- For non-16b input operands that are not K major, prefer the following in order:
  - Use WGMMA variants that source A from RMEM with or without swap AB only
  - Use WGMMA variants that source A from RMEM with swap AB *and* with pipelined and fused transpose for B

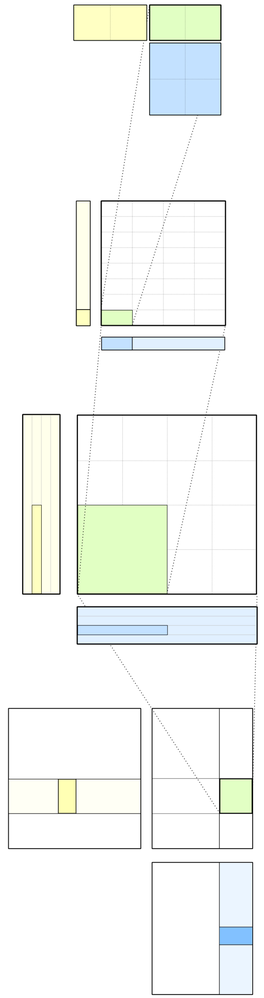
- For epilogue fusions, you have a lot more leeway
- For mainloop fusions
  - If possible, fuse into the epilogue of the previous kernel 🦉
  - Use WGMMA variants that source A from RMEM
  - Avoid fusing things on WGMMA B input as that is a rount trip to smem

CUTLASS

CUDA C++ Template Library for High Performance Linear Algebra

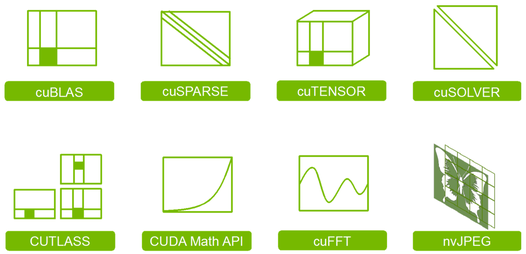
CUTLASS: tensor computations at all scopes and scales, decomposed into their “moving parts”

Device	{ GEMM, Convolution, Reductions , BLAS3 } x { all data types } x { SIMT, Tensor Cores } x { all architectures }
Kernel	GEMM, Batched GEMM, Convolution, Reduction, Fused output operations, Fused input operations
Collective	CUTLASS **temporal micro-kernels** (async producer/consumer pipelines orchestrating spatial micro-kernels)
Atom	CuTe **spatial micro-kernels** (Tiled MMA / Copy)
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

- 4.7K stars, 3M clones/month, 100+ contributors, and many active users
- Latest revision: CUTLASS 3.5
- Documentation: https://github.com/NVIDIA/cutlass#documentation
- Presented: GTC’18, GTC’19, GTC’20, GTC’21, GTC'22 , GTC’22, GTC’23




Thanks !
Q & A