NVIDIA

CUTLASS: CUDA TEMPLATE LIBRARY FOR DENSE LINEAR ALGEBRA AT ALL LEVELS AND SCALES

Jeng Bai-Cheng(Ryan), 21 Nov.

OUTLINE

CUTLASS Introduction

Hierarchical GEMM on GPUs

GEMM Epilogue and Kernel Fusion

Implementation details

Performance and Optimization

MOTIVATION

Productivity Challenges in Deep Learning

Problem:

Multiplicity of Algorithms and Data Types

- GEMM, Convolution, Back propagation
- Mixed precision arithmetic

Kernels specialized for layout and problem size

- NT, TN, NCHW, NHWC

Kernel Fusion

- Custom operations composed with GEMM and convolution

Solution:

Template Library for Linear Algebra Computations in CUDA C++

- Thread-wide, warp-wide, block-wide, device-wide

Data movement and computation primitives

- Iterators, matrix fragments, matrix computations

Inspired by CUB

EXPECTATIONS FOR THE COMPILER

Loop Unrolling: induction variables as constants, arrays as registers

```
const int M, N;
float A[M], B[N], C[M * N];

for (int j = 0; j < N; ++j)
    for (int i = 0; i < M; ++i)
        C[j * M + i] += A[i] * B[j];
```

```
FFMA R79, R23, R16, R79
FFMA R72, R24, R17, R72
FFMA R73, R25, R17, R73
FFMA R74, R26, R17, R74
FFMA R75, R27, R17, R75
...
```

Constant folding: evaluate expressions at compile-time, propagate constants

```
float *smem_ptr[2] = ...;      // *two* SMEM pointers

int const M, Delta;
float A[M];

for (int i = 0; i < M; ++i)
    A[i] = smem_ptr[i & 1][ (i >> 1) * Delta ];
```

```
LDS R28, [R84]
LDS R29, [R85]
LDS R30, [R84 + 128]
LDS R31, [R85 + 128]
```

Function inlining: insert device function into caller’s context and optimize

```
PredicateIterator pred_it = predicates.begin();
int const M;
for (int i = 0; i < M; ++i) {
    if (*pred_it) {            // PredicateIterator::operator*()
      A[i] = ptr[i * Delta];
    }
    ++pred_it;                 // PredicateIterator::operator++()
}
```

```
R2P PR, R112, 0xc
@P1 LDG.E.SYS R105, [R16]
@P2 LDG.E.SYS R106, [R16 + 128]
@P3 LDG.E.SYS R107, [R16 + 256]
@P4 LDG.E.SYS R108, [R16 + 384]
```

GEMM TEMPLATE KERNEL

CUTLASS provides building blocks for efficient device-side code

- Helpers simplify common cases

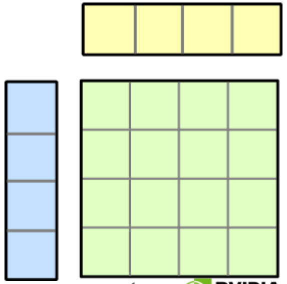
```
typedef cutlass::gemm::SgemmTraits<
  // layout of A matrix
  cutlass::MatrixLayout::kColumnMajor,
  // layout of B matrix
  cutlass::MatrixLayout::kColumnMajor,
  // threadblock tile size
  cutlass::Shape<8, 128, 128>
>
GemmTraits;

typedef cutlass::gemm::Gemm<GemmTraits> Gemm;
typename Gemm::Params params;

int result = params.initialize(
  M, // GEMM M dimension
  N, // GEMM N dimension
  K, // GEMM K dimension
  alpha, // scalar alpha
  A, // matrix A operand
  lda,
  B, // matrix B operand
  ldb,
  beta, // scalar beta
  C, // source matrix C
  ldc,
  C,
  ldc);

Gemm::launch(params);
```

IMPLEMENTED COMPUTATIONS

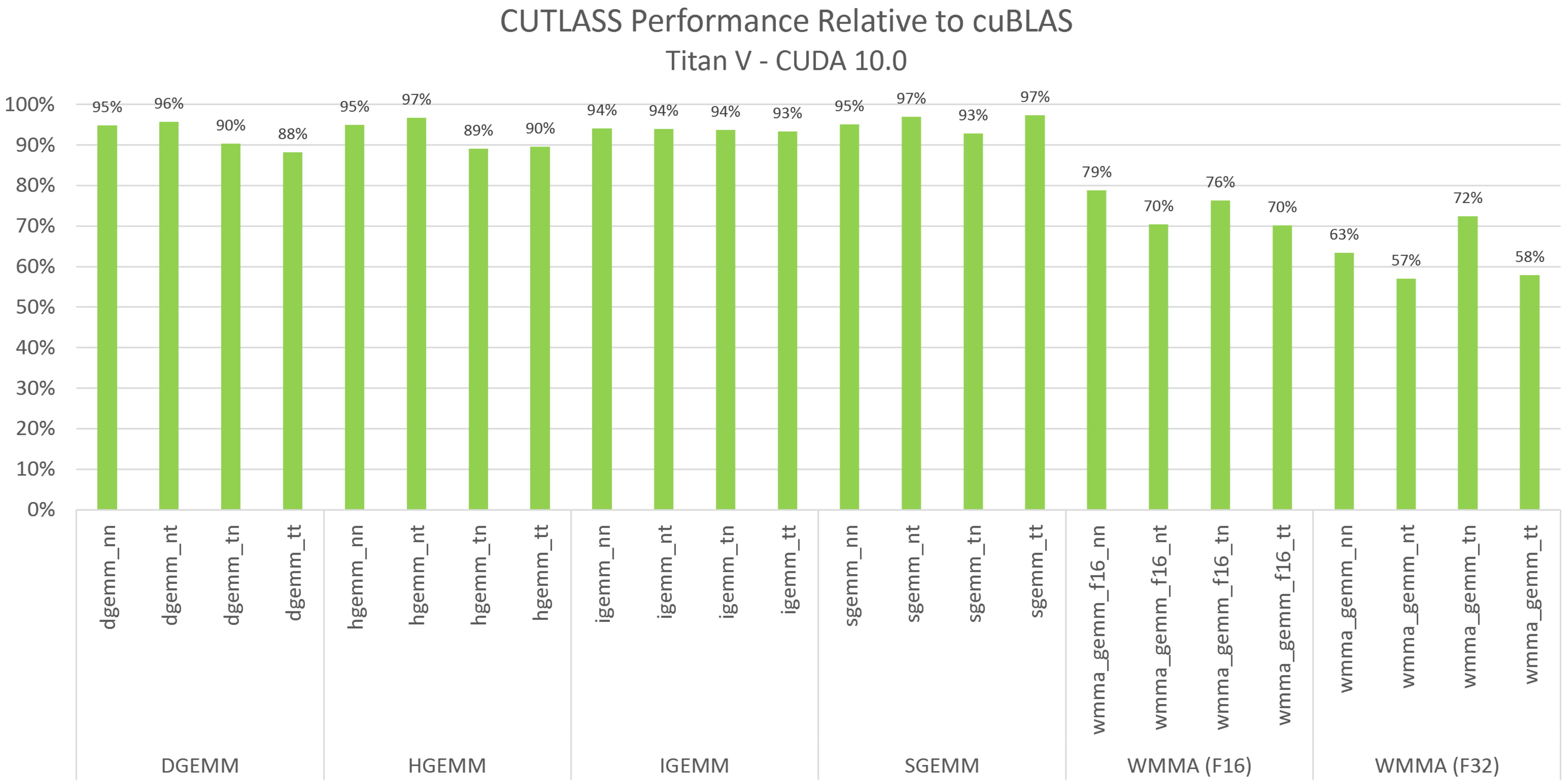
CUTLASS v1.2

	A	B	C	Accumulator
SGEMM	float	float	float	float
	half	half	half, float	float
DGEMM	double	double	double	double
HGEMM	half	half	half	half
IGEMM	int8_t	int8_t	int8_t	int32_t
	int8_t	int8_t	float	int32_t
WMMA GEMM	half	half	half	half
	half	half	half, float	float
	int8_t	**int8_t**	**int32_t**	**int32_t**
	int4_t	**int4_t**	**int32_t**	**int32_t**
	bin1_t	**bin1_t**	**int32_t**	**int32_t**

+ Batched strided; optimizations for small GEMM



CUTLASS PERFORMANCE – TITAN V



HIERARCHICAL GEMM ON GPUS

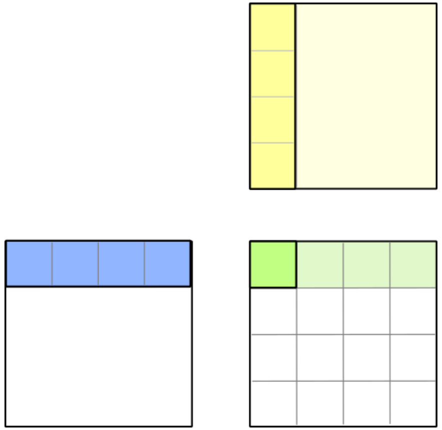
GENERAL MATRIX PRODUCT

Basic definition

General matrix product

$$C = \alpha \, \mathrm{op}(A) * \mathrm{op}(B) + \beta \, C$$

C is *M*-by-*N*, op(*A*) is *M*-by-*K*, op(*B*) is *K*-by-*N*



Compute independent dot products

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
       for (int k = 0; k < K; ++k)
           C[i][j] += A[i][k] * B[k][j];
```

Inefficient due to large working sets to hold parts of *A* and *B*

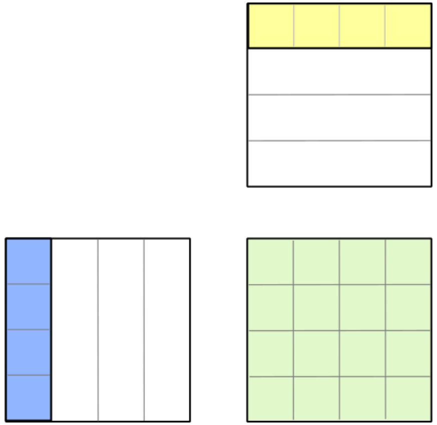
GENERAL MATRIX PRODUCT

Accumulated outer products

General matrix product

$$C = \alpha \ \mathrm{op}(A) * \mathrm{op}(B) + \beta \ C$$

C is M-by-N, op(A) is M-by-K, op(B) is K-by-N



~~Compute independent dot products~~

```
// Independent dot products
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
       for (int k = 0; k < K; ++k)
           C[i][j] += A[i][k] * B[k][j];
```

Permute loop nests

```
// Accumulated outer products
for (int k = 0; k < K; ++k)
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Load elements of *A* and *B* exactly once

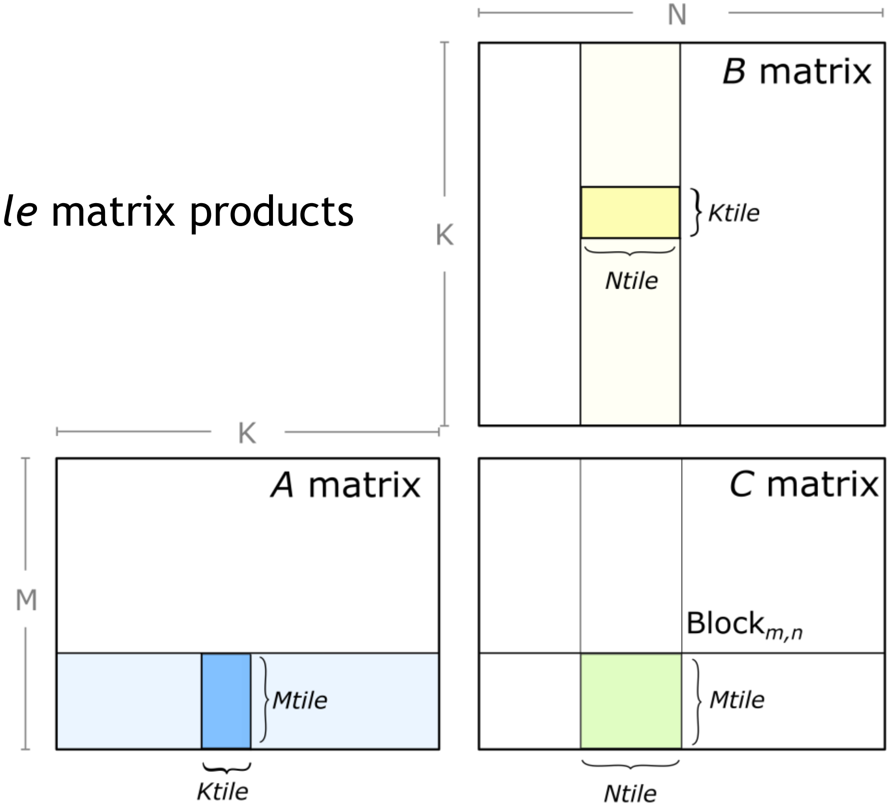
GENERAL MATRIX PRODUCT

Computing matrix product one block at a time

Partition the loop nest into *blocks* along each dimension

- Partition into *Mtile*-by-*Ntile* independent matrix products
- Compute each product by accumulating *Mtile*-by-*Ntile*-by-*Ktile* matrix products

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)
        for (int kb = 0; kb < K; kb += Ktile)
        {
            // compute Mtile-by-Ntile-by-Ktile matrix product
            for (int k = 0; k < Ktile; ++k)
                for (int i = 0; i < Mtile; ++i)
                    for (int j = 0; j < Ntile; ++j)
                    {
                        int row = mb + i;
                        int col = nb + j;

                        C[row][col] +=
                            A[row][kb + k] * B[kb + k][col];
                    }
        }
```



BLOCKED GEMM IN CUDA

Parallelism Among CUDA Thread Blocks

Launch a CUDA kernel grid

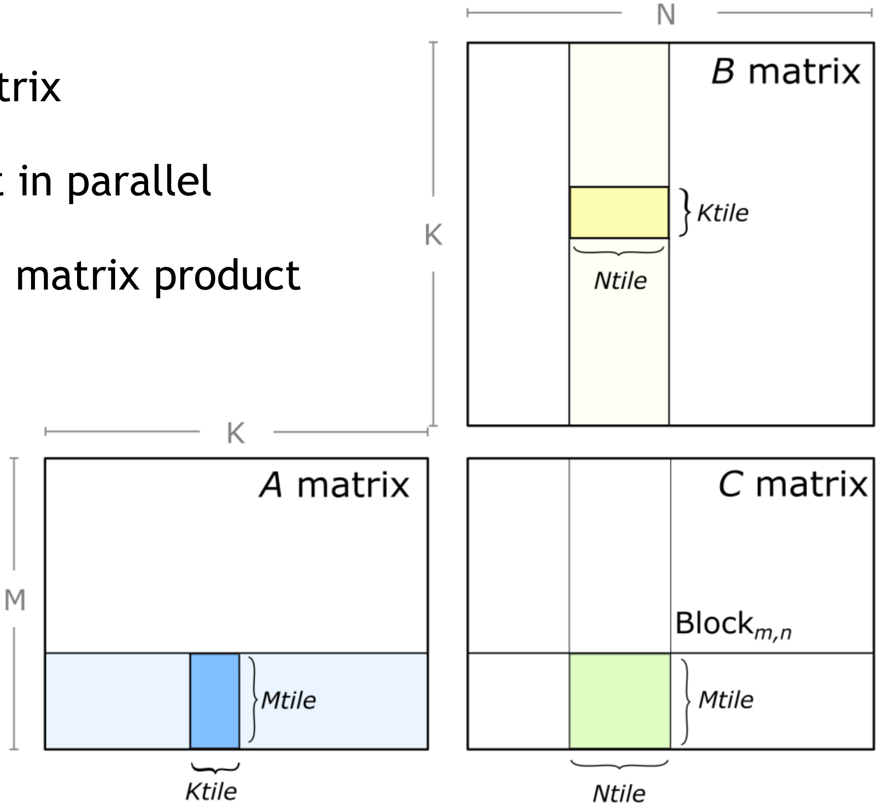
- Assign CUDA thread blocks to each partition of the output matrix

CUDA thread blocks compute *Mtile*-by-*Ntile*-by-*K* matrix product in parallel

- Iterate over *K* dimension in steps, performing an accumulated matrix product

```
for (int mb = 0; mb < M; mb += Mtile)
    for (int nb = 0; nb < N; nb += Ntile)

        for (int kb = 0; kb < K; kb += Ktile)
        {
            .. compute Mtile by Ntile by Ktile GEMM
        }
```

by each CUDA thread block



THREAD BLOCK TILE STRUCTURE

Parallelism Within a CUDA Thread Block

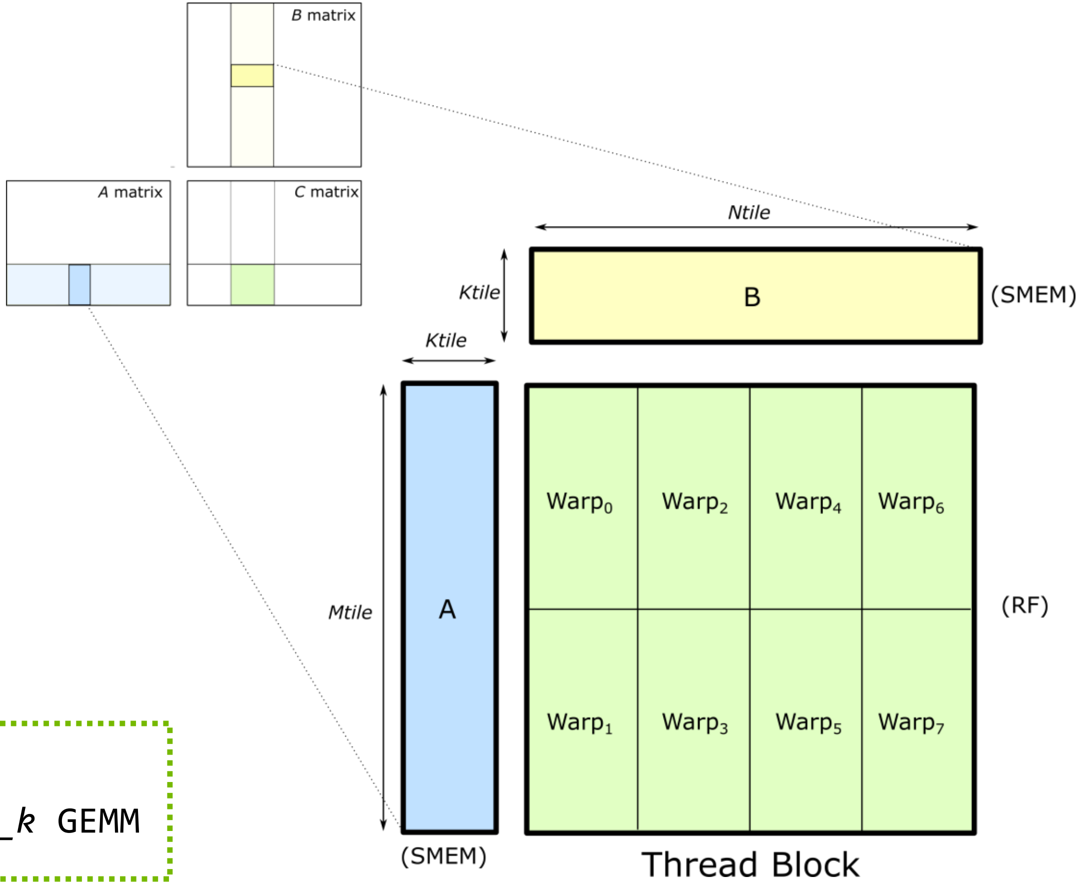
Decompose thread block into warp-level tiles

- Load *A* and *B* operands into Shared Memory (reuse)
- *C* matrix distributed among warps

Each warp computes an independent matrix product

```
for (int kb = 0; kb < K; kb += Ktile)
{
    .. load A and B tiles to shared memory

    for (int m = 0; m < Mtile; m += warp_m)
        for (int n = 0; n < Ntile; n += warp_n)

            for (int k = 0; k < Ktile; k += warp_k)
                .. compute warp_m by warp_n by warp_k GEMM

}
```

by each CUDA warp



WARP-LEVEL TILE STRUCTURE

Warp-level matrix product

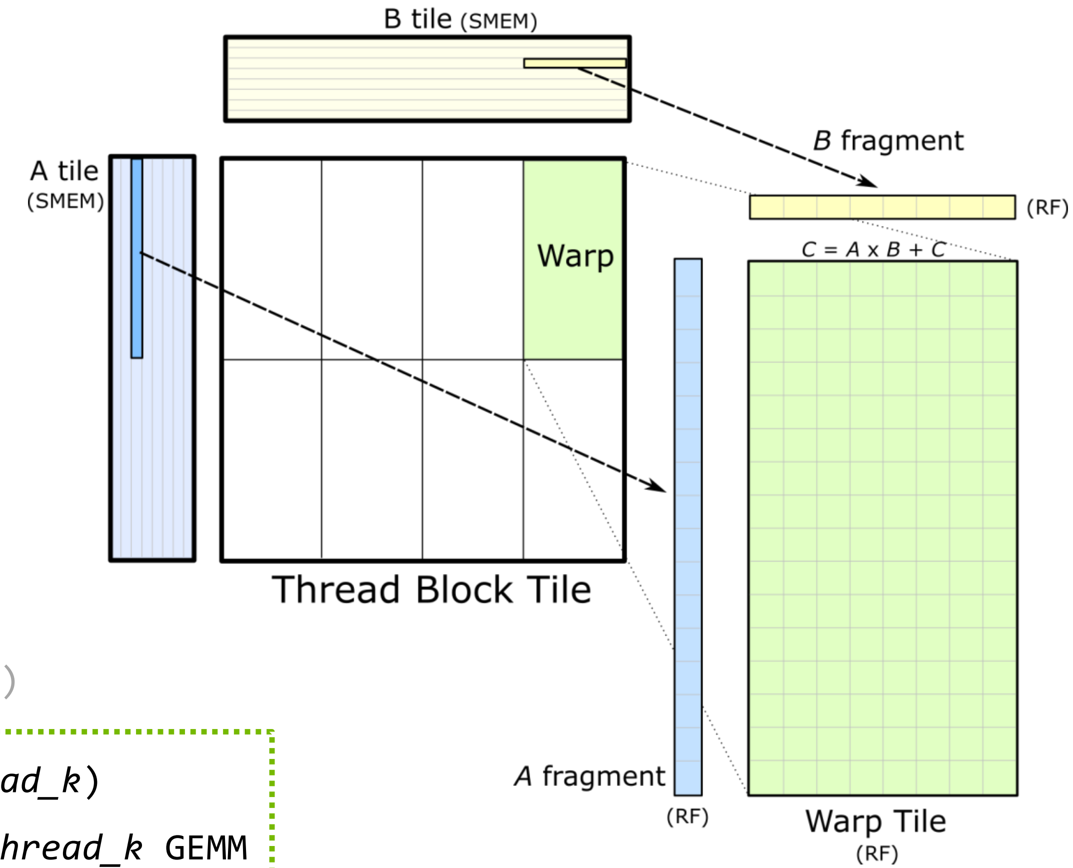
Warps perform an accumulated matrix product

- Load *A* and *B* operands from SMEM into registers
- *C* matrix held in registers of participating threads

Shared Memory layout is *K*-strided for efficient loads

```
for (int k = 0; k < Ktile; k += warp_k)
{
    .. load A tile from SMEM into registers
    .. load B tile from SMEM into registers

    for (int tm = 0; tm < warp_m; tm += thread_m)
        for (int tn = 0; tn < warp_n; tn += thread_n)

            for (int tk = 0; tk < warp_k; tk += thread_k)

                .. compute thread_m by thread_n by thread_k GEMM

}
```

by each CUDA thread



THREAD-LEVEL TILE STRUCTURE

Parallelism within a thread

Threads compute accumulated matrix product

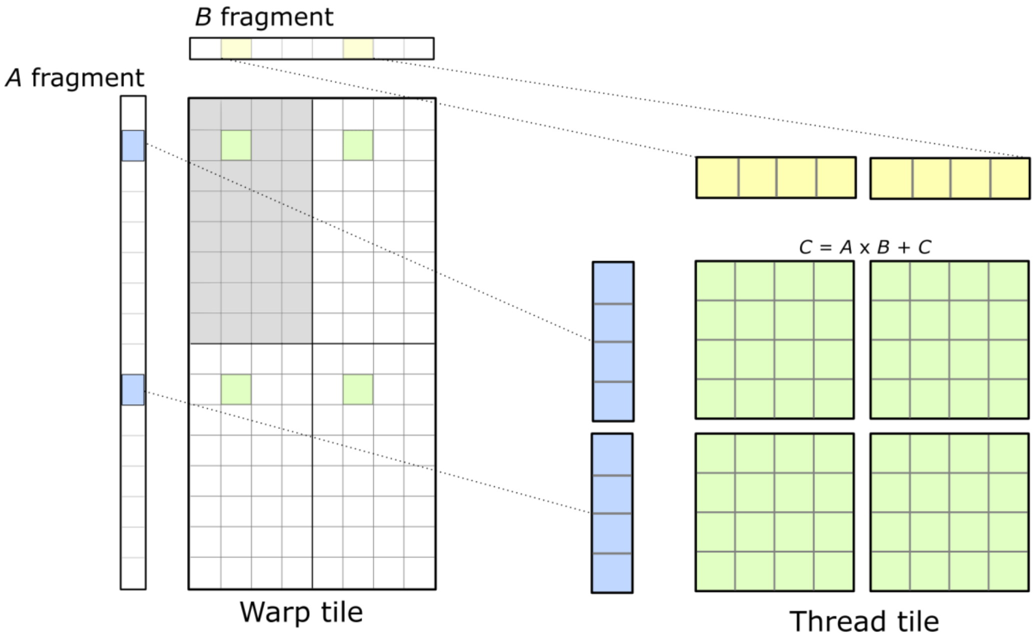
- ***A***, ***B***, and ***C*** held in registers

Opportunity for data reuse:

- *O(M*N)* computations on *O(M+N)* elements

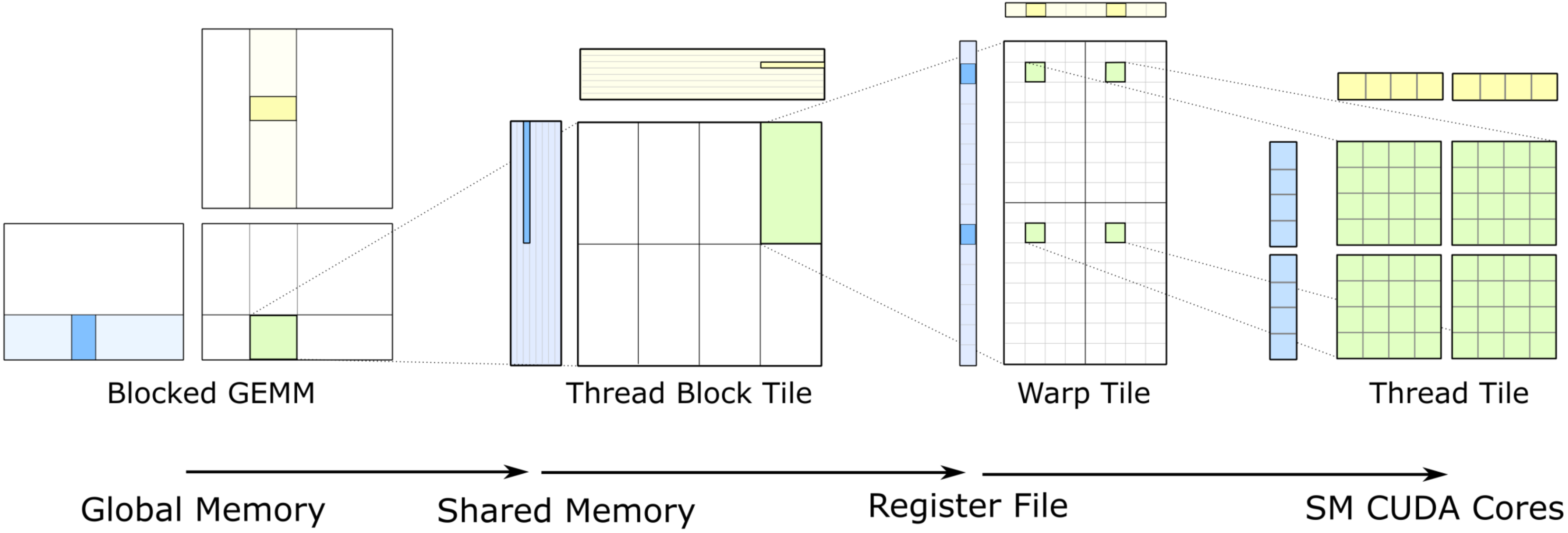
```
for (int m = 0; m < thread_m; ++m)
    for (int n = 0; n < thread_n; ++n)

        for (int k = 0; k < thread_k; ++k)
            C[m][n] += A[m][k] * B[n][k];
```

Fused multiply-accumulate instructions



COMPLETE GEMM HIERARCHY

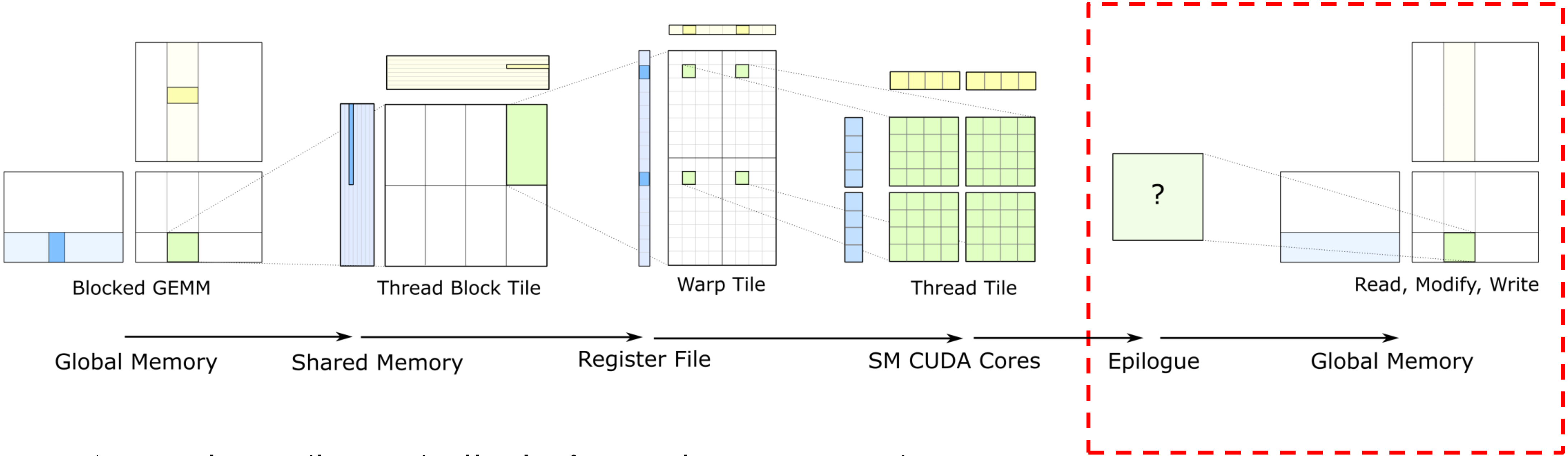
Data reuse at each level of the memory hierarchy



GEMM EPILOGUE AND KERNEL FUSION

(IN)COMPLETE GEMM HIERARCHY

Efficiently update the output matrix

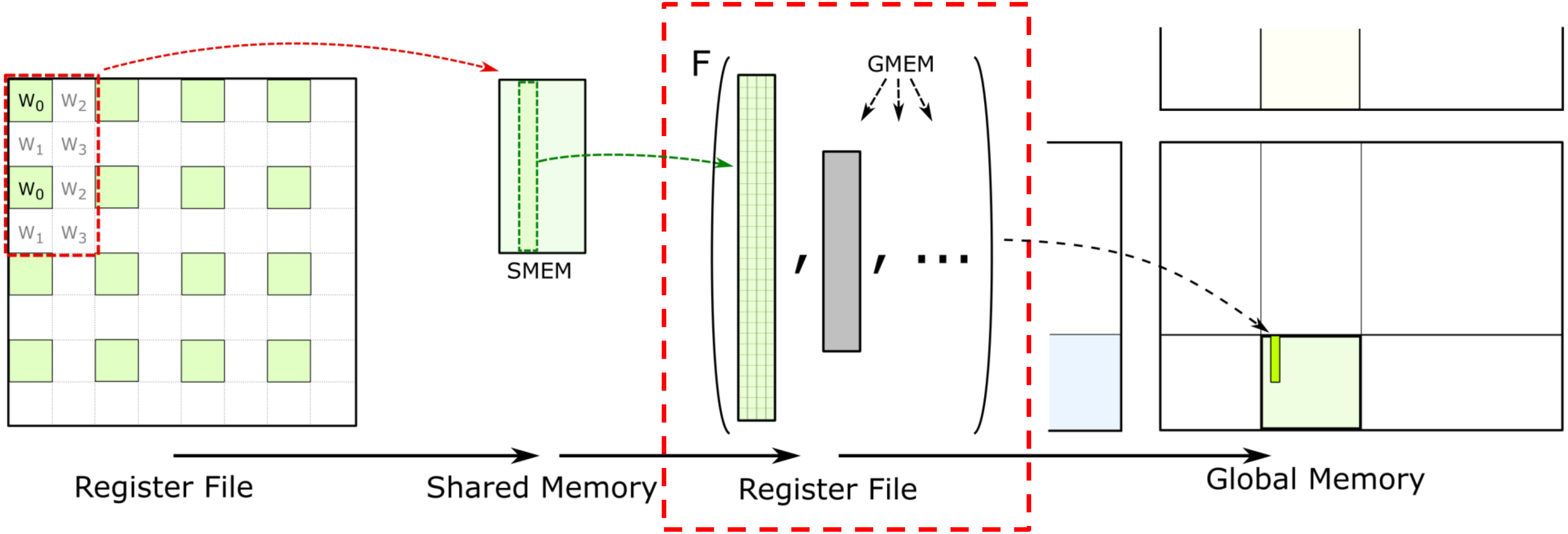


Accumulator tiles typically don’t match output matrix

- Element-wise operation: $C = \alpha AB + \beta C$
- Type Conversion: scale, convert, and pack into vectors
- Layout: C matrix is contiguous

KERNEL FUSION

Custom element-wise operations during epilogue

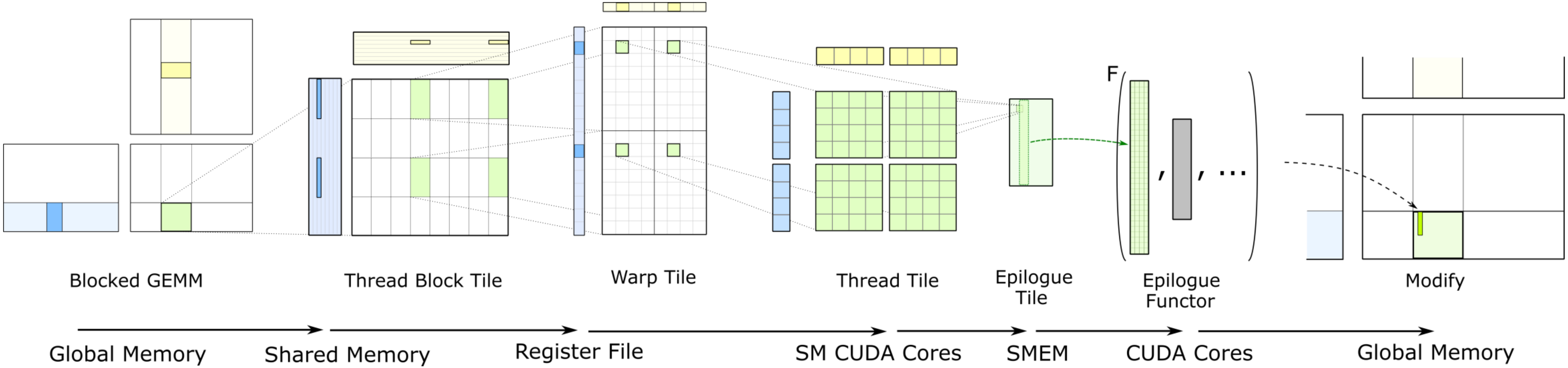


Matrix product may be combined with arbitrary functions

- Element-wise operators: Scaling, bias, activation functions
- Data type conversion: F32->F16, Int32->Int8
- Matrix update operations: reductions across thread blocks

COMPLETE* GEMM DATA FLOW

Embodied by CUTLASS CUDA templates



* Mostly. Not depicted: software pipelining, double-buffering, and more. Read the code. ☺

IMPLEMENTATION DETAILS

GEMM TEMPLATE KERNEL

CUTLASS provides building blocks for efficient device-side code

- Helpers simplify common cases

```
typedef cutlass::gemm::SgemmTraits<
  // layout of A matrix
  cutlass::MatrixLayout::kColumnMajor,
  // layout of B matrix
  cutlass::MatrixLayout::kColumnMajor,
  // threadblock tile size
  cutlass::Shape<8, 128, 128>
>
GemmTraits;

typedef cutlass::gemm::Gemm<GemmTraits> Gemm;
typename Gemm::Params params;

int result = params.initialize(
  M, // GEMM M dimension
  N, // GEMM N dimension
  K, // GEMM K dimension
  alpha, // scalar alpha
  A, // matrix A operand
  lda,
  B, // matrix B operand
  ldb,
  beta, // scalar beta
  C, // source matrix C
  ldc,
  C,
  ldc);

Gemm::launch(params);
```

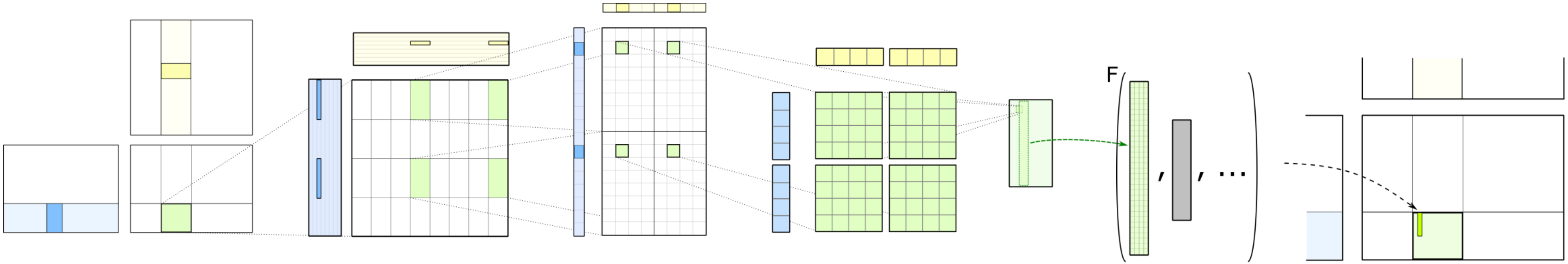
GEMM TEMPLATE KERNEL

GEMM Kernel:

```
/// GEMM kernel with launch bounds specified
template <typename Gemm_>
__global__  __launch_bounds__(Gemm_::kThreads)
void gemm_kernel(typename Gemm_::Params params) {
  // Declare shared memory.
  __shared__ typename Gemm_::SharedStorage shared_storage;

  // Construct the GEMM object.
  Gemm_ gemm(params, shared_storage);
  // Run GEMM.
  gemm.multiply_add();
}
```

COMPLETE GEMM STRUCTURAL MODEL

Embodied by CUTLASS CUDA templates



Blocked GEMM → Thread Block Tile → Warp Tile → Thread Tile → Epilogue Tile → Epilogue Functor → Modify

Global Memory → Shared Memory → Register File → CUDA/Tensor Cores → SMEM → CUDA Cores → Global Memory

Global Load Stream

- GemmGlobalIteratorAb
- Transformer
- GemmSharedStoreTileAb

Shared Load Stream

- GemmSharedLoadTile{A,B}

Matrix Multiply

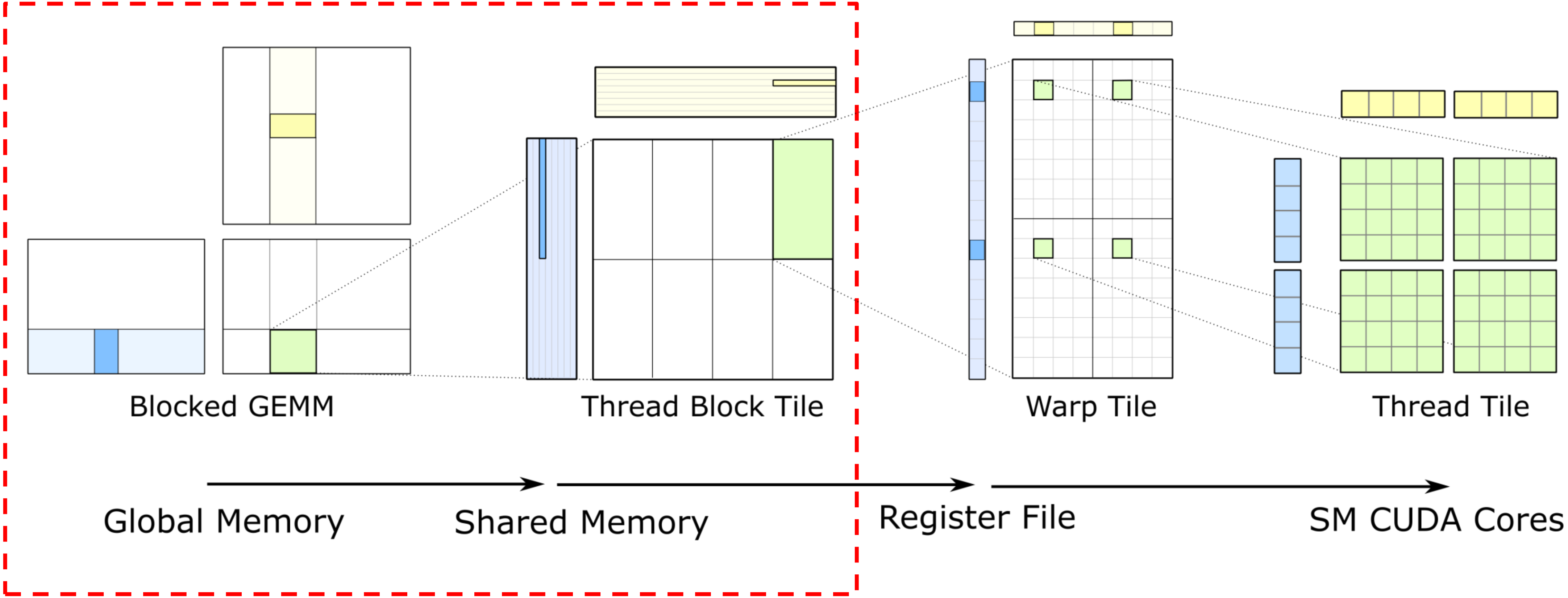
- fma, dp4a, hfma2
- WMMA

Epilogue

- Transformer
- GemmSharedStoreTileD
- GemmSharedLoadTileD
- GemmGlobalIteratorC
- Functor
- GemmGlobalIteratorD

GEMM HIERARCHY: THREAD BLOCKS

Streaming efficiently to shared memory



GEMM TEMPLATE KERNEL

Global Load Stream:

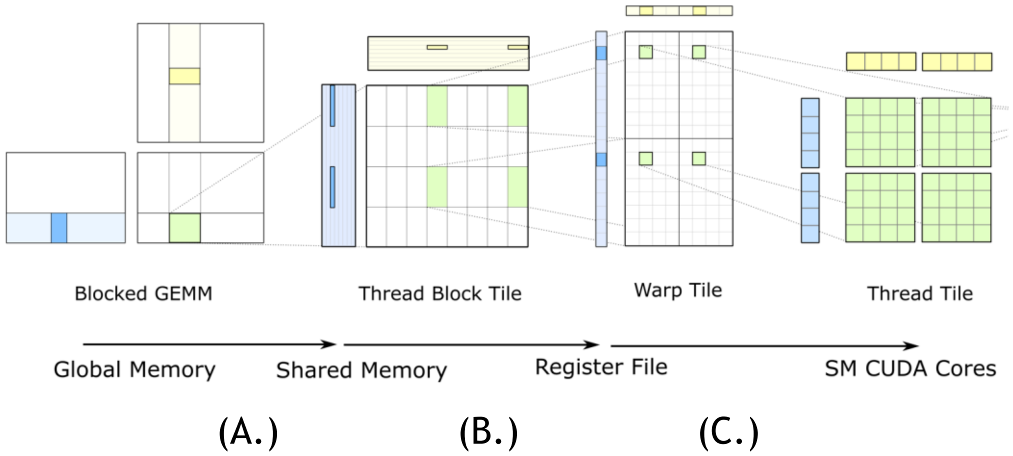
```
CUTLASS_DEVICE void multiply_add() {
  // vector load, 32b, 64b, or 128b
  global_to_shared_stream.copy();
  // vector store
  global_to_shared_stream.commit();
  __syncthreads();

  #pragma unroll
  for (; outer_k > 0; outer_k -= Traits::OutputTile::kD) {
    // prefatch next tile
    global_to_shared_stream.copy();
    /***
    compute_tile(param):
      load data from shared memory to register
      compute MAD
      __syncthreads();
    ***/
    global_to_shared_stream.commit();
  }
}
```

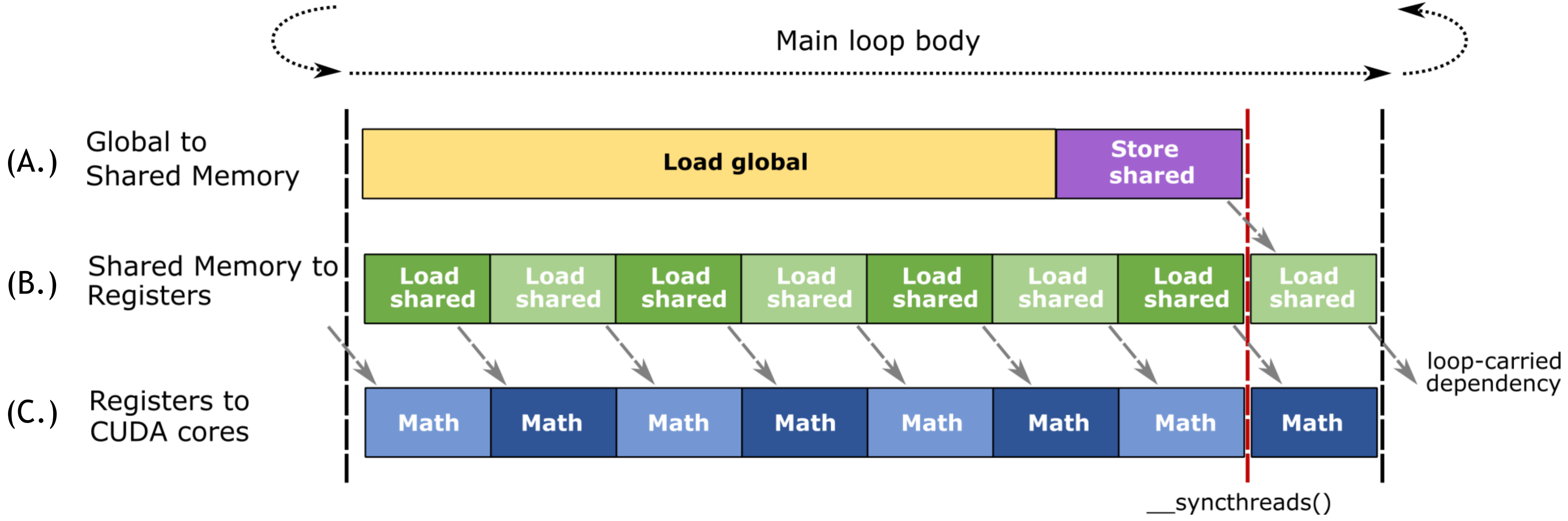
SOFTWARE PIPELINING

Hiding latency through ILP

Double buffer: Load next tile from higher-level memory while computing with current tile

- 2x SMEM
- 2x RF fetch registers



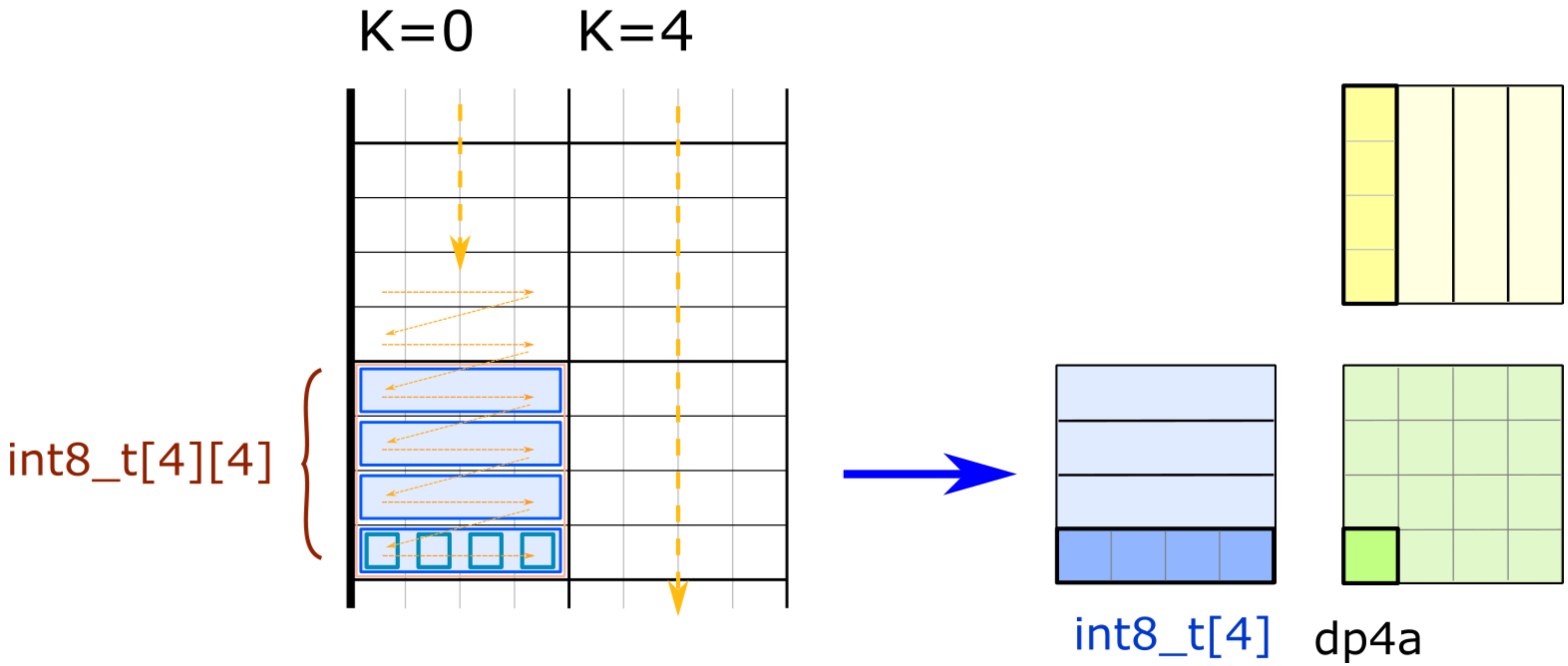


EXAMPLE: IGEMM

Interleaved data layouts for efficient streaming from Shared Memory

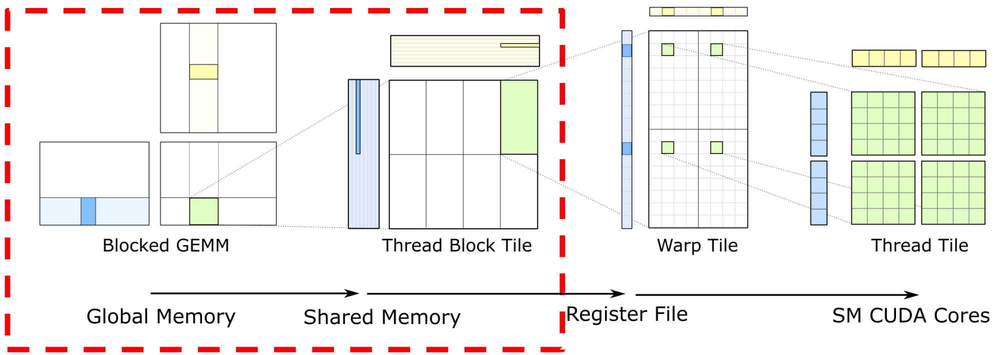
DP4A requires operands to be contiguous along *K* dimension

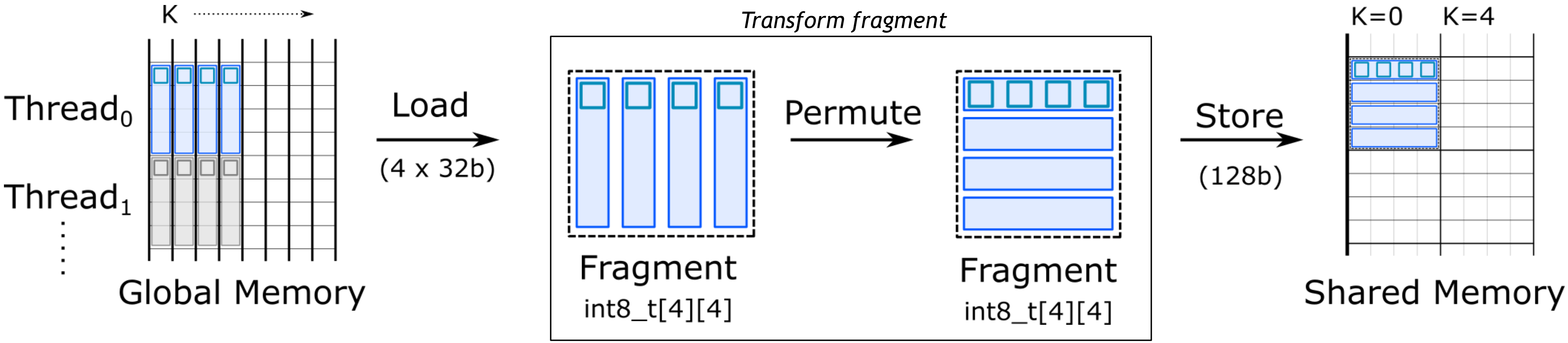
- Efficient fragment loading requires *K*-strided layout in Shared Memory
- Solution: adopt a hybrid SMEM layout



FRAGMENTS

Permute fragments before storing to shared memory

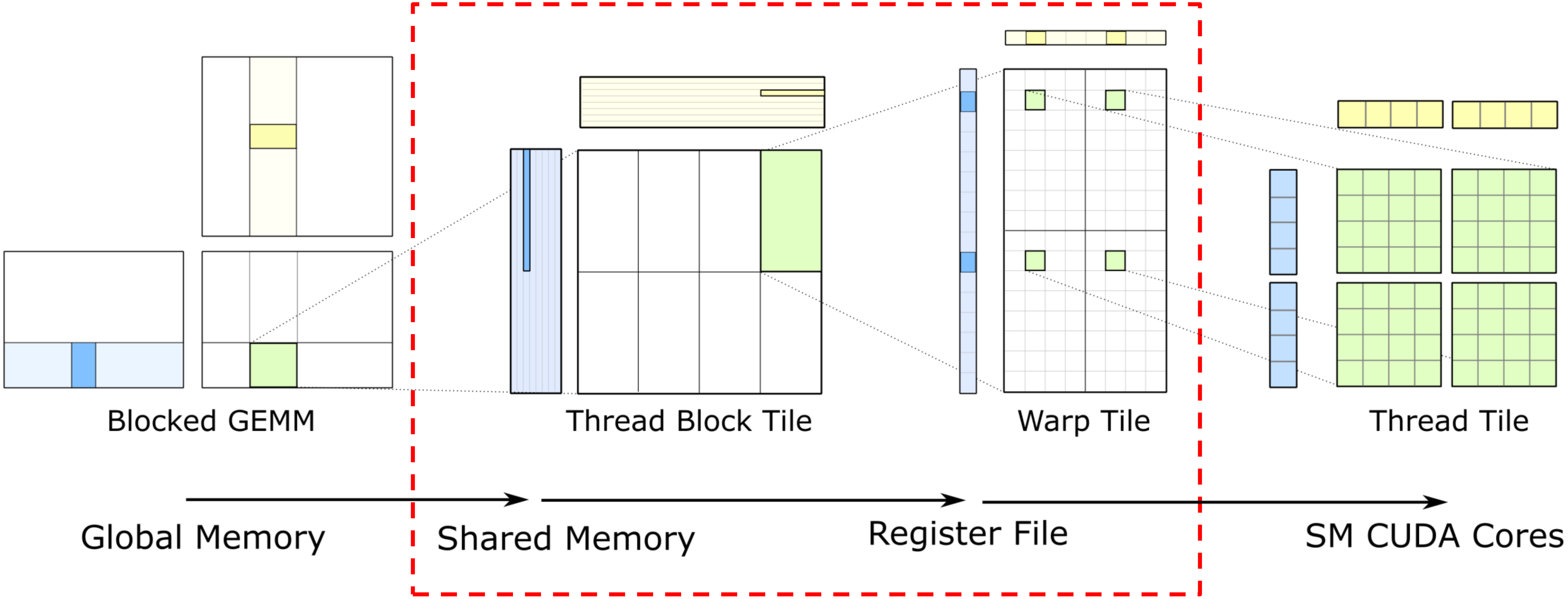




PTX ISA: prmt

GEMM HIERARCHY: WARP TILES

Loading multiplicands into registers



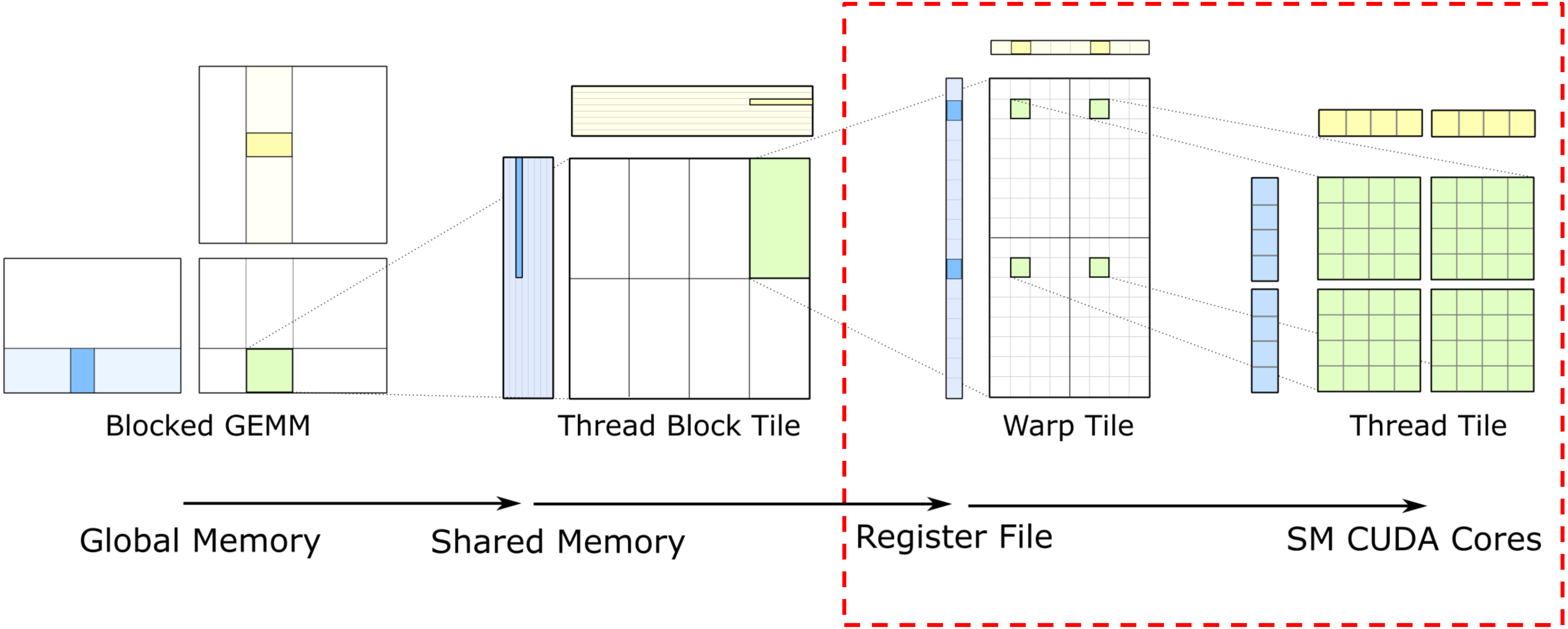
GEMM TEMPLATE KERNEL

Shared Load Stream:

```
CUTLASS_DEVICE void consume_tile(...) {
  shared_load_stream.copy(step);
  #pragma unroll
  for (int step = 0; step < kWarpGemmSteps - 1; ++step) {
    // Trigger the copy from shared memory for the next A/B values.
    shared_load_stream.copy(step + 1);
    // Make sure the values are available for the current iteration to do the multiply-add.
    shared_load_stream.commit(step);
    // Do the math on the fragments of the current iteration.
    multiply_add.multiply_add(shared_load_stream.fragment_a(step),
                              shared_load_stream.fragment_b(step),
                              accumulators,
                              accumulators);
  }
  // Make sure the data from shared memory has been entirely consumed.
  __syncthreads();
}
```

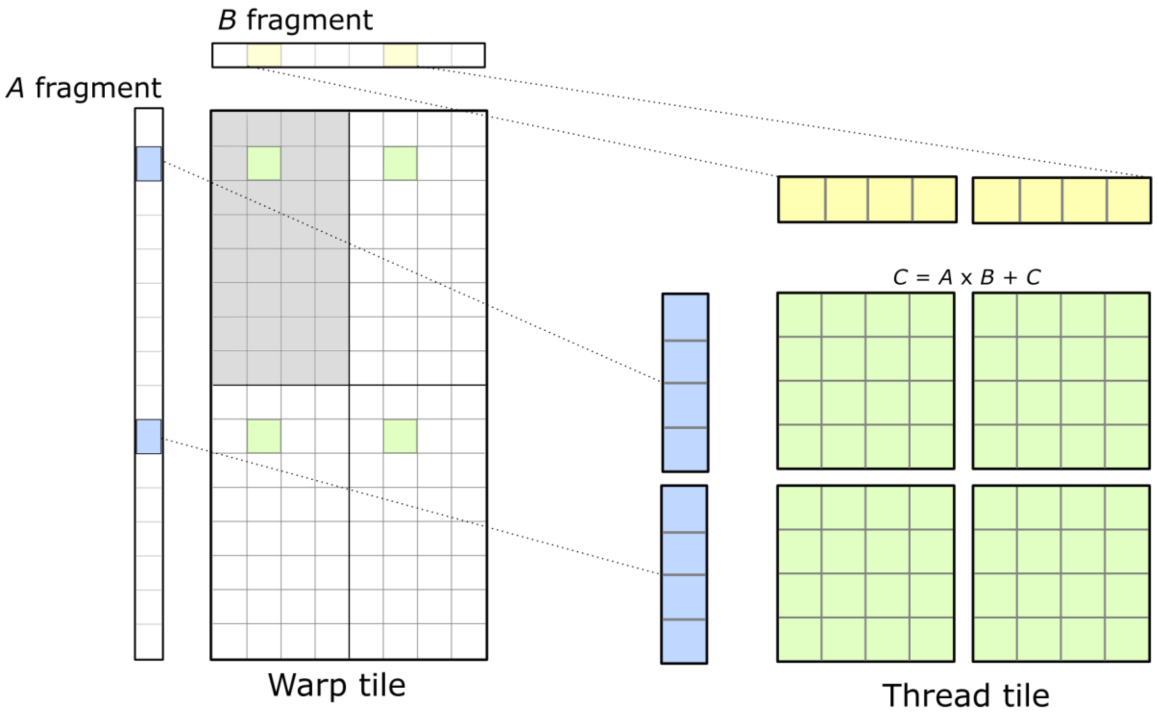
GEMM HIERARCHY: CUDA CORES

Actually doing the math



REGISTERS TO CUDA CORES

Compute matrix multiply-accumulate on fragments held in registers



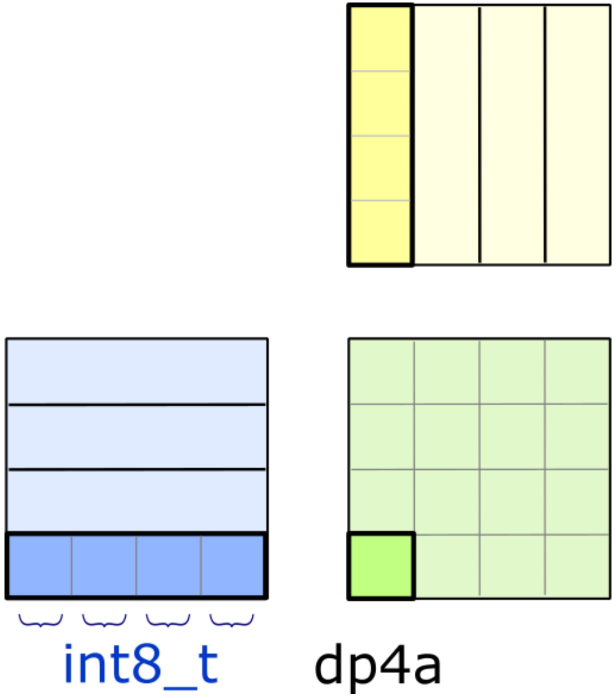
```
// Perform thread-level matrix multiply-accumulate
template <
    typename Shape,
    typename ScalarA,
    typename ScalarB,
    typename ScalarC
>
struct ThreadMultiplyAdd {

    /// Multiply: D = A*B + C
    inline __device__ void multiply_add(
        Fragment<ScalarA, Shape::kW> const & A,
        Fragment<ScalarB, Shape::kH> const & B,
        Accumulators const & C,
        Accumulators & D) {

        // Perform M-by-N-by-1 matrix product using FMA
        for (int j = 0; j < Shape::kH; ++j) {
            for (int i = 0; i < Shape::kW; ++i) {

                D.scalars[j * Shape::kW + i] =

                    // multiply
                    A.scalars[i] * B.scalars[j] +

                    // add
                    C.scalars[j * Shape::kW + i];
            }
        }
    }
};
```

EXAMPLE: IGEMM

Mixed-precision Integer-valued GEMM

DP4A instruction computes 4-element dot product

- *A* and ***B*** are packed vectors of 8-bit integers
- Accumulator is 32-bit signed integer



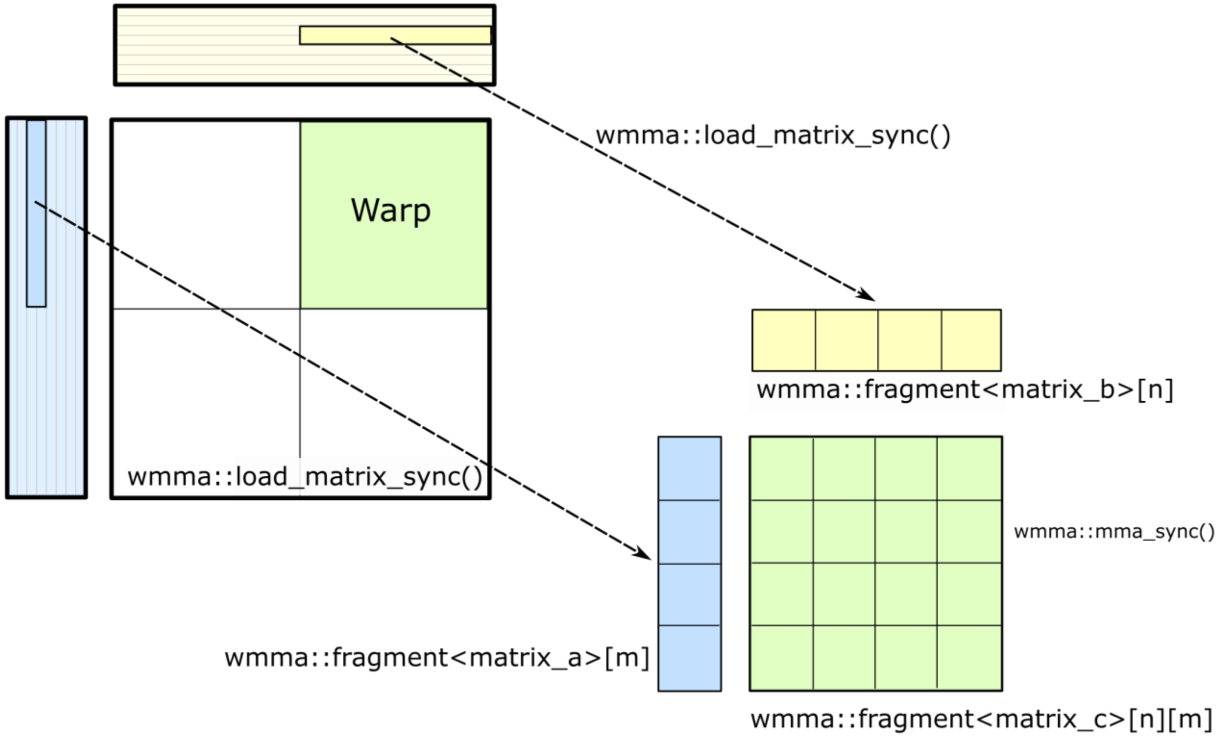
```
/// Perform M-by-N-by-4 matrix product using DP4A
template <typename Shape>
struct ThreadMultiplyAdd<Shape, int8_t, int8_t, int> {

    /// Multiply: d = a*b + c
    inline __device__ void multiply_add(
        Fragment<int8_t, Shape::kW * 4> const & A,
        Fragment<int8_t, Shape::kH * 4> const & B,
        Accumulators const & C,
        Accumulators & D) {

        int const* a_int =
            reinterpret_cast<int const*>(&A.scalars[0]);

        int const* b_int =
            reinterpret_cast<int const*>(&B.scalars[0]);

        // Perform M-by-N-by-4 matrix product using DP4A
        for (int j = 0; j < Shape::kH; ++j) {
            for (int i = 0; i < Shape::kW; ++i) {

                // Inline PTX to issue DP4A instruction
                asm volatile(
                    "dp4a.s32.s32 %0, %1, %2, %3;"
                    : "=r"(D.scalars[j * Shape::kW + i])
                    : "r"(a_int[i]),
                      "r"(b_int[j]),
                      "r"(C.scalars[j * Shape::kW + i])
                );
            }
        }
    }
};
```

EXAMPLE: TENSOR CORES

Targeting the CUDA WMMA API

WMMA: Warp-synchronous Matrix Multiply-Accumulate

- API for issuing operations to Tensor Cores


wmma::load_matrix_sync()
Warp
wmma::load_matrix_sync()
wmma::fragment<matrix_b>[n]
wmma::mma_sync()
wmma::fragment<matrix_a>[m]
wmma::fragment<matrix_c>[n][m]

```
/// Perform warp-level multiply-accumulate using WMMA API
template <
    /// Data type of accumulator
    typename ScalarC,

    /// Shape of warp-level accumulator tile
    typename WarpTile,

    /// Shape of one WMMA operation – e.g. 16x16x16
    typename WmmaTile
>
struct WmmaGemmMultiplyAdd {

    /// Compute number of WMMA operations
    typedef typename ShapeDiv<WarpTile, WmmaTile>::Shape
       Shape;

    /// Multiply: D = A*B + C
    inline __device__ void multiply_add(
        FragmentA const & A,
        FragmentB const & B,
        FragmentC const & C,
        FragmentD & D) {

        // Perform M-by-N-by-K matrix product using WMMA
        for (int n = 0; n < Shape::kH; ++n) {
            for (int m = 0; m < Shape::kW; ++m) {

                // WMMA API to invoke Tensor Cores
                nvcuda::wmma::mma_sync(
                    D.elements[n][m],
                    A.elements[k][m],
                    B.elements[k][n],
                    C.elements[n][m]
                );
            }
        }
    }
};
```

EXAMPLE: TENSOR CORES

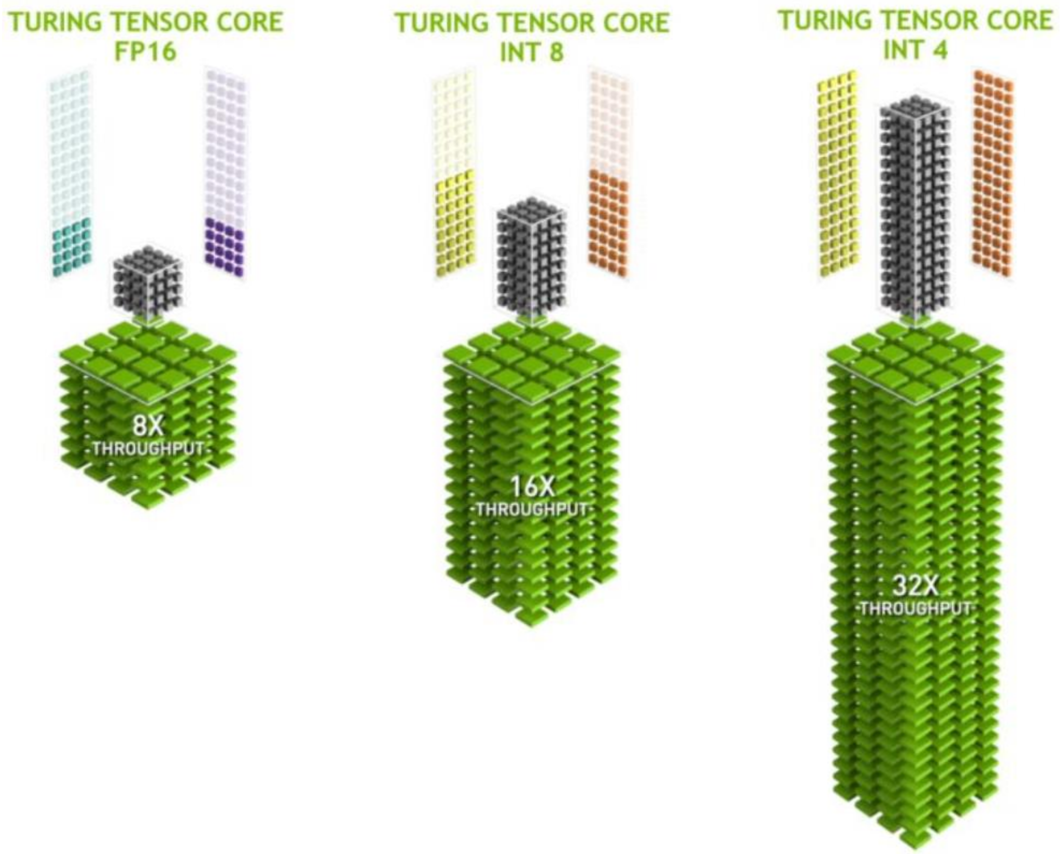
Targeting the CUDA WMMA API

WMMA GEMM targeting TensorCores

Volta Features: FP16

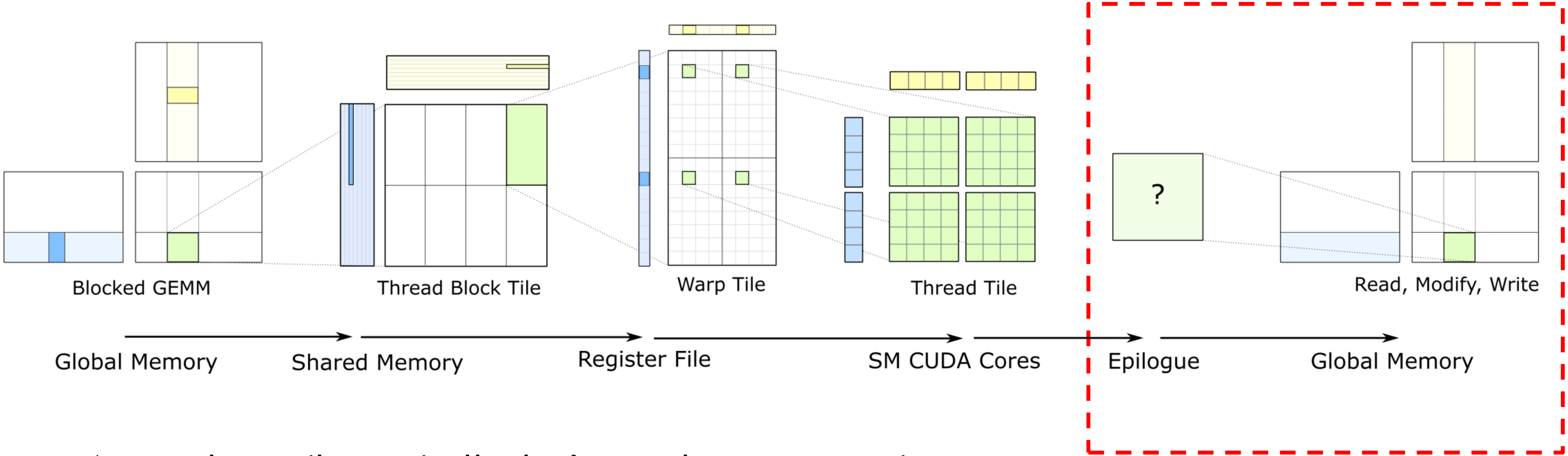
Turing Features: INT8, INT4, 1-bit

cutlass/examples/05_wmma_gemm



(IN)COMPLETE GEMM HIERARCHY

Efficiently update the output matrix



Accumulator tiles typically don’t match output matrix

- Element-wise operation: $D = \alpha AB + \beta C$
- Type Conversion: scale, convert, and pack into vectors
- Layout: *C* matrix is contiguous

GEMM TEMPLATE KERNEL

Epilogue ($D = \alpha AB + \beta C$):

```
CUTLASS_DEVICE void
epilogue_with_or_without_beta(...) {
  for (int h = 0; h < Iterations::kH; ++h) {
    for (int w = 0; w < Iterations::kW; ++w) {
      // LinearScaling
      functor.evaluate(accum, c, d);
    }
  }
}

struct LinearScaling {
  CUTLASS_DEVICE void evaluate(...) {
    FragmentMultiplyAdd mad;
    FragmentB_ tmp;
    mad.multiply(params.beta, old, tmp);
    mad.multiply_add(alpha, accum, tmp, output);
  }
}
```

```
struct FragmentMultiplyAdd {
  CUTLASS_DEVICE void multiply_add(...) {
    for (int j = 0; j < kElements; ++j) {
      d[j] = a * ScalarAlphaBeta(d[j]) +
             ScalarAlphaBeta(c[j]);
    }
  }
}
```

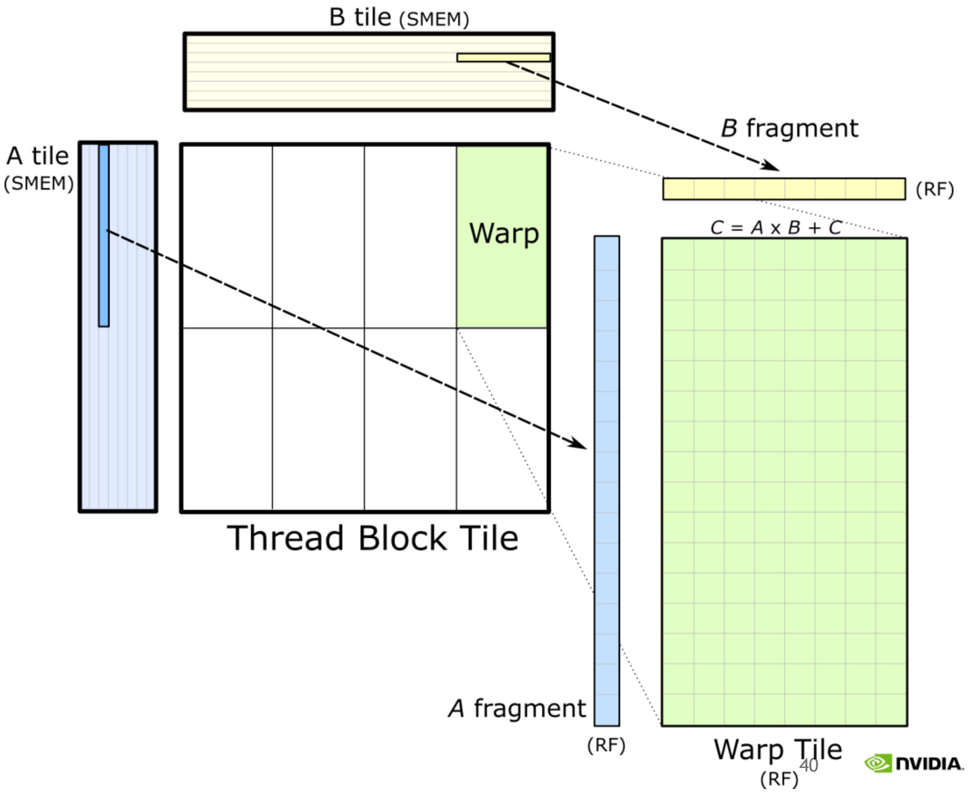
PERFORMANCE AND OPTIMIZATION

CHOOSING TILE SIZES

Typical warp-tile fragment sizes (M-by-N-by-K) :

- SGEMM, DGEMM: 64-by-32-by-1
- HGEMM: 128-by-32-by-1
- IGEMM: 64-by-32-by-4
- WMMA: 64-by-32-by-16, 64-by-64-by-16

```
typedef cutlass::gemm::SgemmTraits<
  // layout of A matrix
  cutlass::MatrixLayout::kColumnMajor,
  // layout of B matrix
  cutlass::MatrixLayout::kColumnMajor,
  // threadblock tile size
  cutlass::Shape<1, 32, 64>
>
GemmTraits;
```



PARALLELIZING REDUCTIONS

GEMM "Split K"

GEMM problems may not be embarrassingly parallel

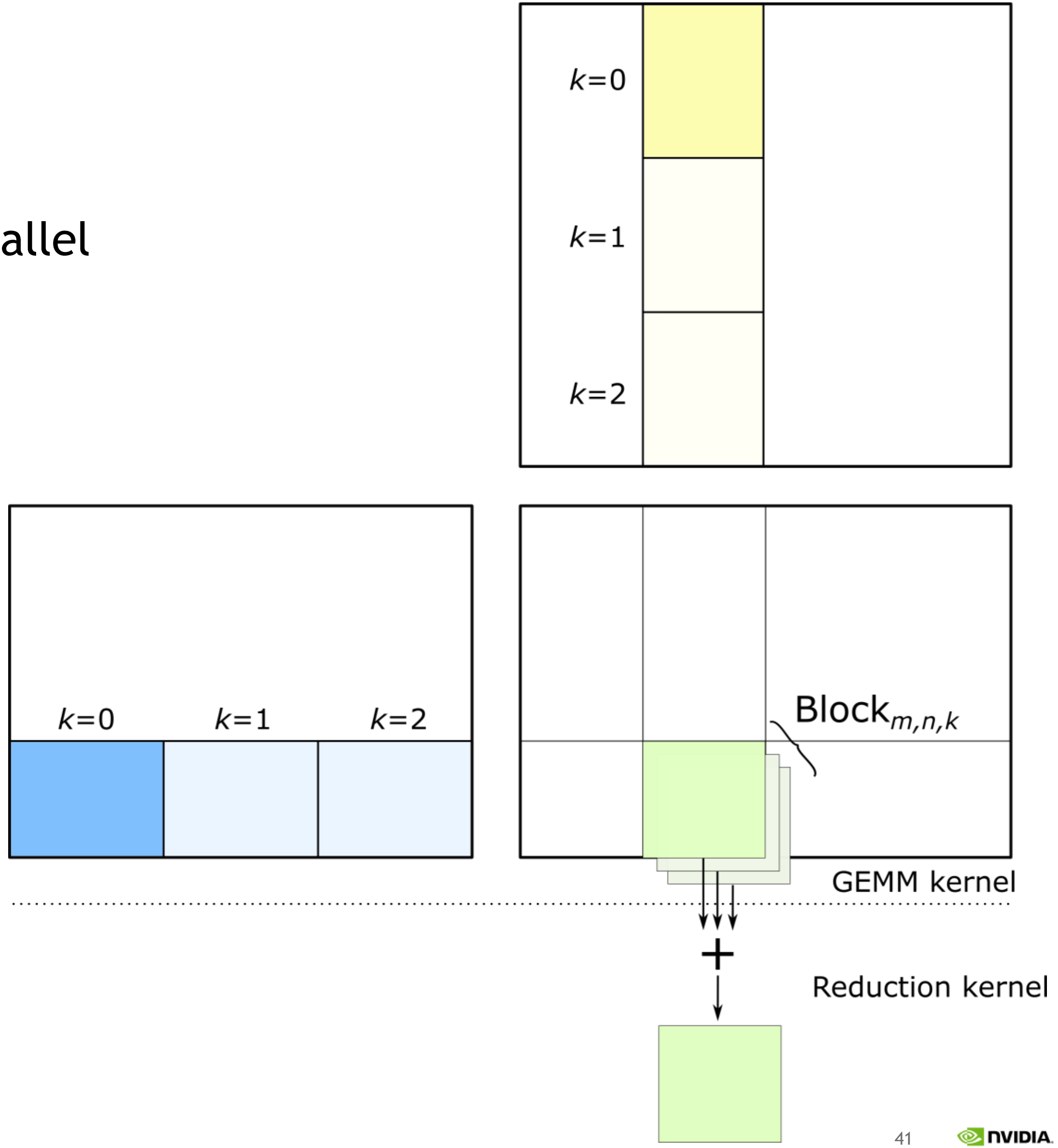
- Large K but small *M*, *N*

Reduction may be parallelized

- Partition GEMM *K* dimension
- Launch multiple threadblocks per *m*, *n* location
- Perform final reduction in separate kernel

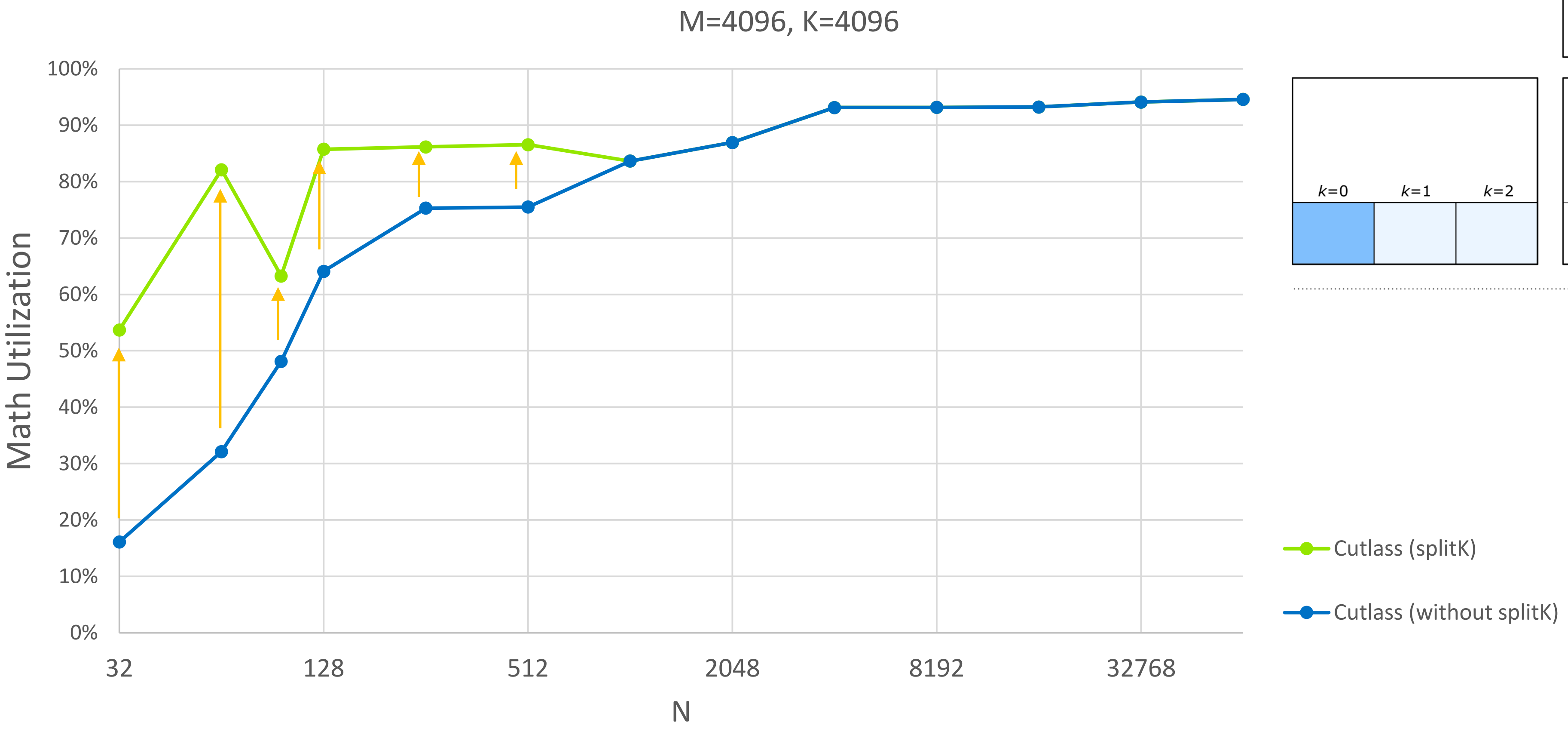
Implications

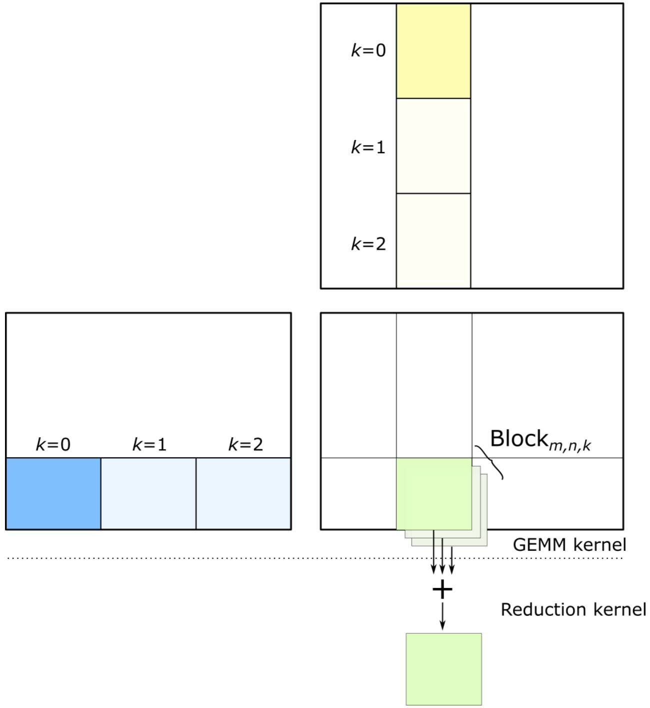
- Workspace needed to store intermediate result
- Epilogue functor executed in reduction kernel
- Deterministic; not bit equivalent with serial reduction



PARALLELIZING REDUCTIONS

GEMM “Split K”





Narrows performance gap for small GEMM problems

GEMM TEMPLATE KERNEL

Split-K GEMM:

```
typedef cutlass::gemm::SgemmTraits<
    cutlass::MatrixLayout::kColumnMajor,
    cutlass::MatrixLayout::kColumnMajor,
    cutlass::Shape<8, 128, 128> >
    SgemmTraits;

typedef
cutlass::reduction::BatchedReductionTraits<
    float, /*type of A*/
    float, /*type of C*/
    float, /*type of D*/
    float, /*type of alpha and beta*/
    float, /*type of accumulation*/
    splits_count /*reduction workload*/
    >
    BatchedReductionTraits;

typedef
cutlass::gemm::SplitkPIGemmTraits<SgemmTraits,
BatchedReductionTraits> deviceGemmTraits;
```

```
// kernel class
typedef typename deviceGemmTraits::KernelClass
deviceGemm;

typename deviceGemm::Params deviceGemmParams(m, n,
k);

int workspace_size =
deviceGemmParams.required_workspace_memory_in_byt
e();
float *workspace_ptr;
cudaMalloc(&workspace_ptr, workspace_size);

// finish init Params
deviceGemmParams.initialize(alpha, A, lda,
                            B, ldb, beta,
                            C, ldc, C, ldc,
                            workspace_ptr);
deviceGemm::launch(deviceGemmParams);
```

CONCLUSION

CONCLUSION

CUTLASS: CUDA C++ Template Library

CUTLASS is an Open Source Project for implementing Deep Learning computations on GPUs

- https://github.com/nvidia/cutlass (3-clause BSD License)

CUTLASS is efficient: >90% cuBLAS performance

Generic programming techniques span Deep Learning design space

- Hierarchical decomposition of GEMM
- Data movement primitives
- Mixed-precision and Volta Tensor Cores

CUTLASS enables developers to compose custom Deep Learning CUDA kernels