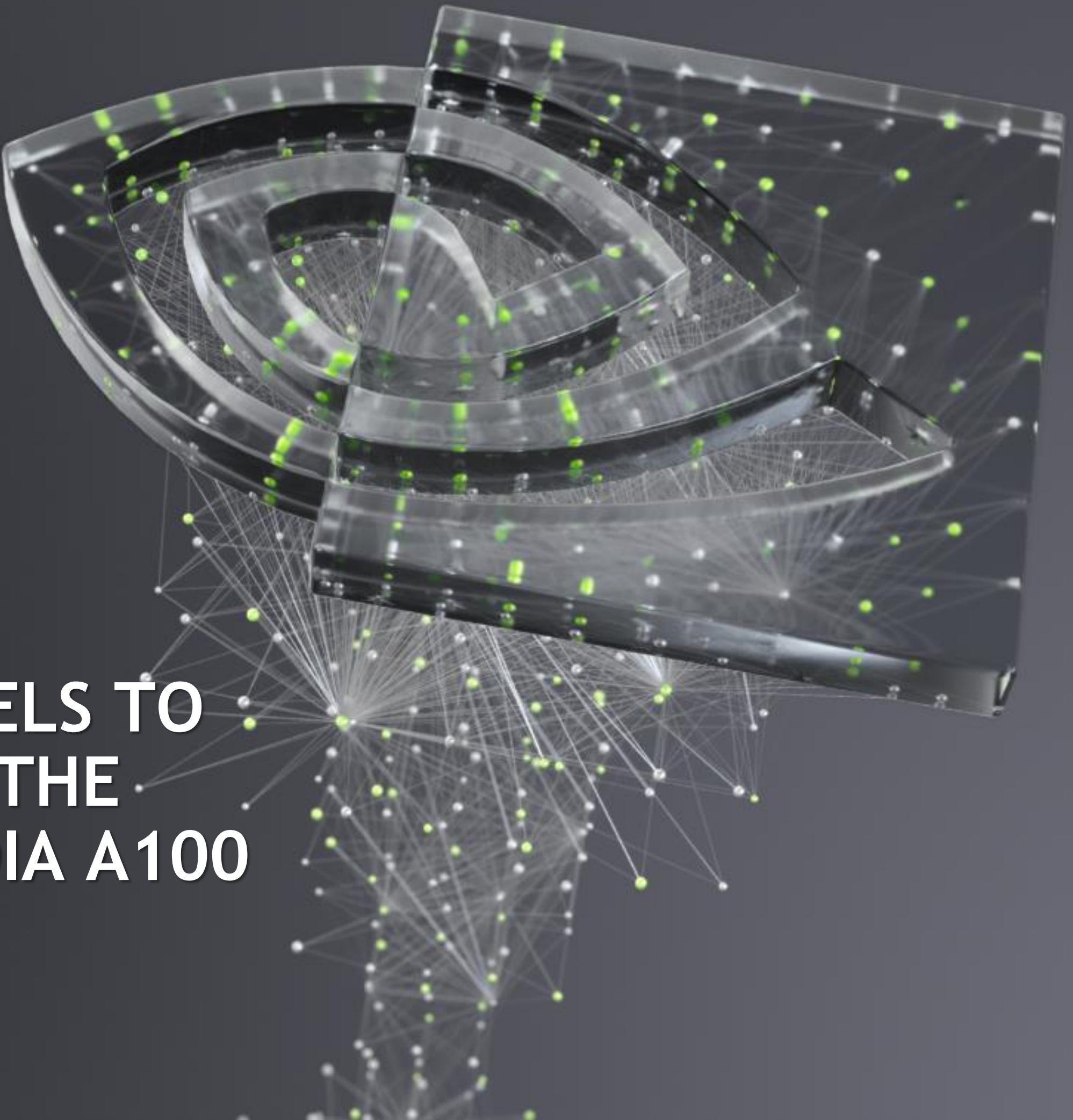


DEVELOPING CUDA KERNELS TO PUSH TENSOR CORES TO THE ABSOLUTE LIMIT ON NVIDIA A100

Andrew Kerr, May 21, 2020





ACKNOWLEDGEMENTS

CUTLASS Team

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

Contributors

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

Acknowledgements

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



AGENDA

Overview

NVIDIA Ampere Architecture and CUTLASS 2.2

Tensor Cores on NVIDIA Ampere Architecture

Accelerated matrix operations

Efficient data movements for Tensor Cores

Strategies for maximizing performance

CUTLASS on NVIDIA A100

Optimal CUDA C++ templates for Tensor Cores



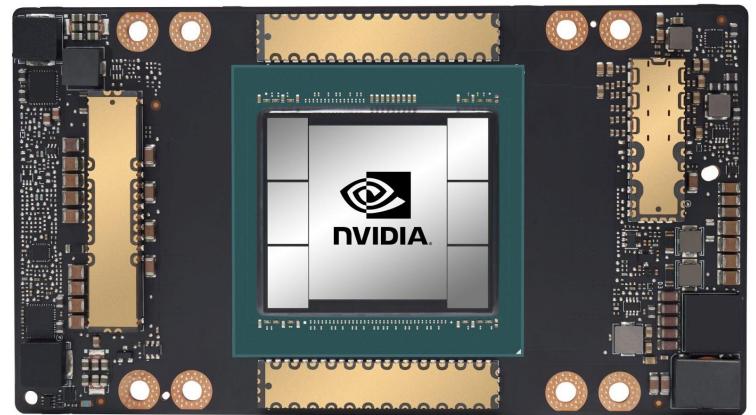
OVERVIEW

NVIDIA AMPERE ARCHITECTURE

NVIDIA A100

New and Faster Tensor Core Operations

- Floating-point Tensor Core operations **8x** and **16x** faster than F32 CUDA Cores
- Integer Tensor Core operations **32x** and **64x** faster than F32 CUDA Cores
- New IEEE double-precision Tensor Cores **2x** faster than F64 CUDA Cores

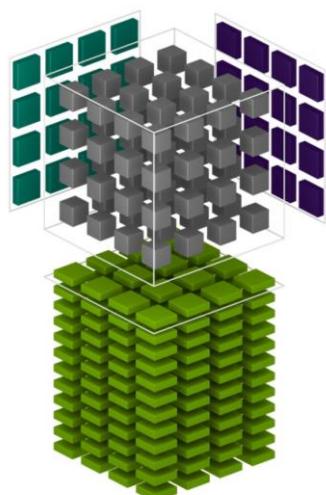


Additional Data Types and Mode

- Bfloat16, double, Tensor Float 32

Asynchronous copy

- Copy directly into shared memory - deep software pipelines



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

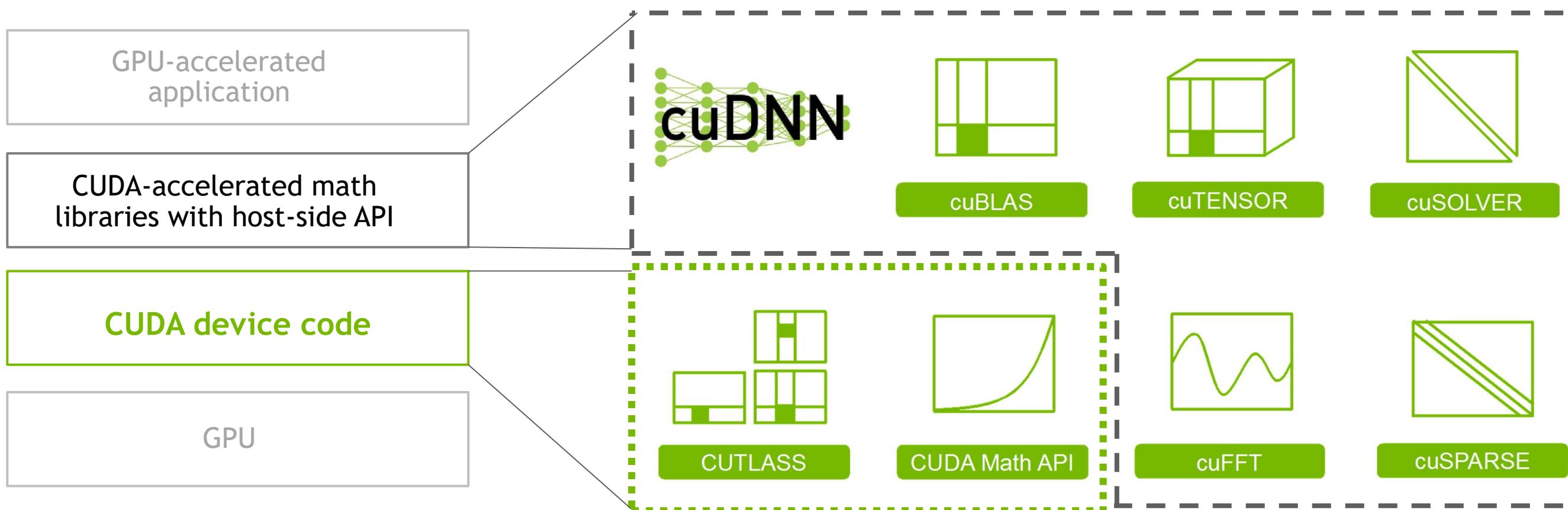
PROGRAMMING NVIDIA AMPERE ARCHITECTURE

Deep Learning and Math Libraries using Tensor Cores (with CUDA kernels under the hood)

- cuDNN, cuBLAS, cuTENSOR, cuSOLVER, cuFFT, cuSPARSE
- “CUDNN V8: New Advances in Deep Learning Acceleration” (GTC 2020 - S21685)
- “How CUDA Math Libraries Can Help you Unleash the Power of the New NVIDIA A100 GPU” (GTC 2020 - S21681)
- “Inside the Compilers, Libraries and Tools for Accelerated Computing” (GTC 2020 - S21766)

CUDA C++ Device Code

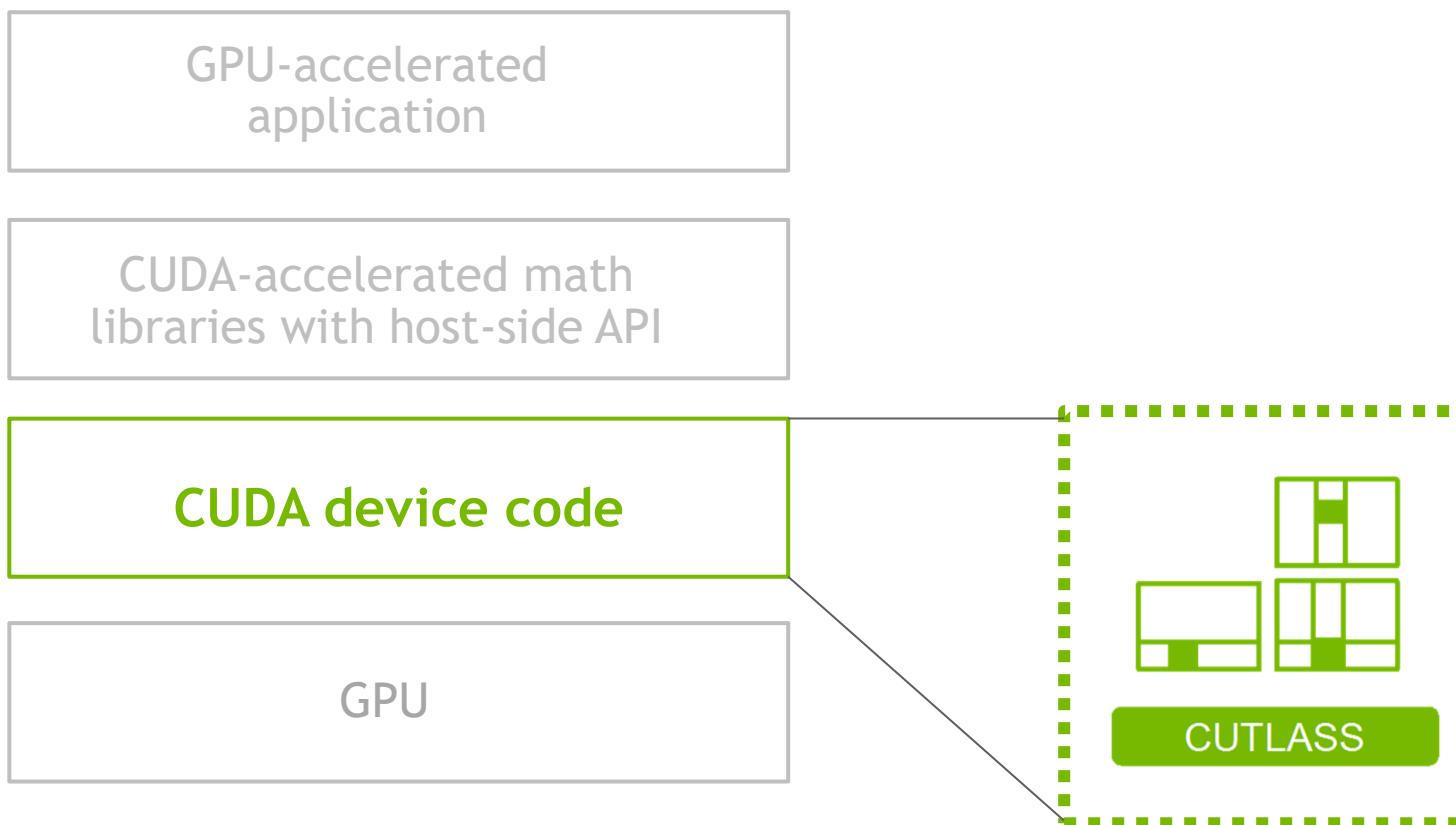
- CUTLASS, CUDA Math API, CUB, Thrust, libcu++



PROGRAMMING NVIDIA AMPERE ARCHITECTURE with CUDA C++

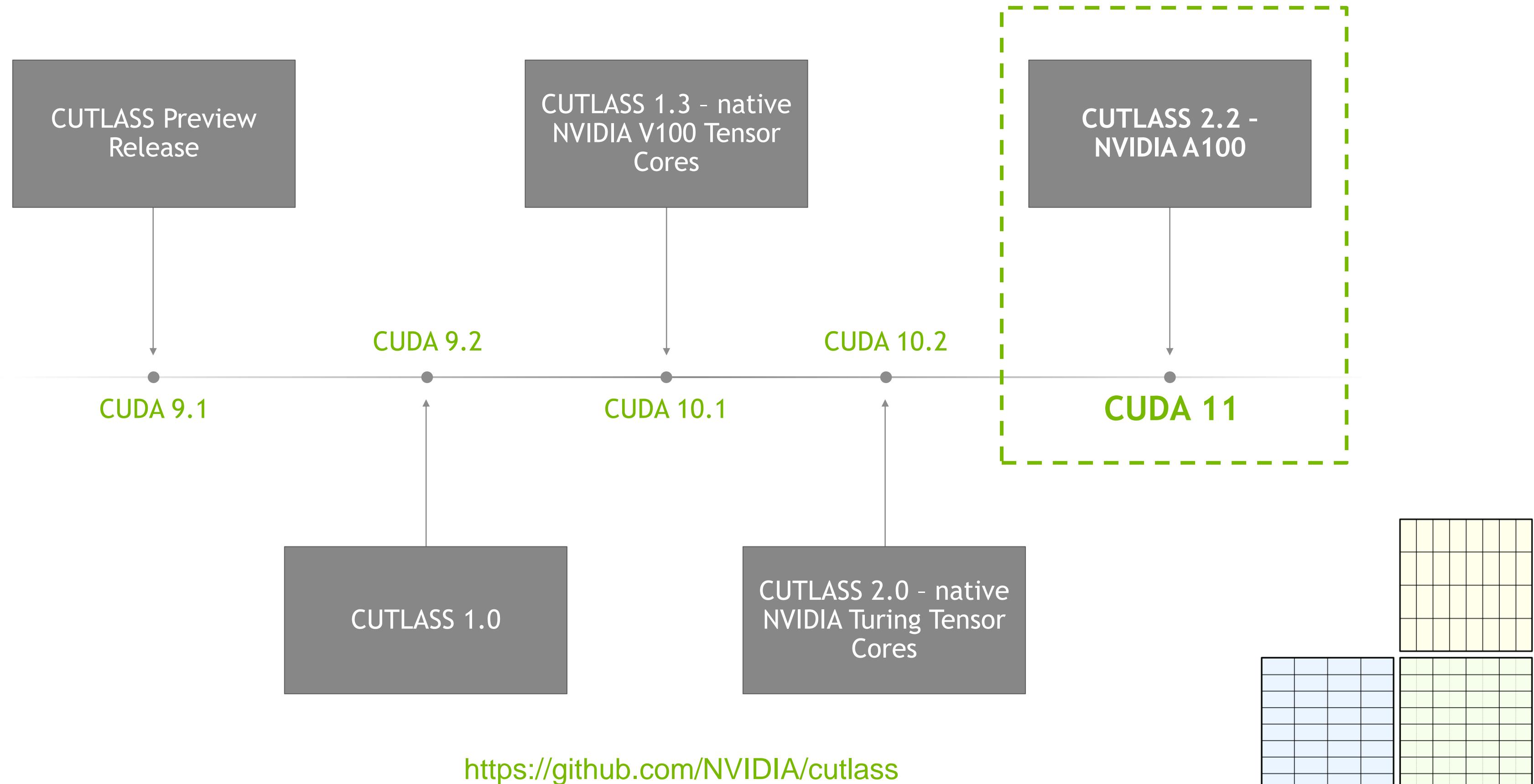


This is a talk for CUDA programmers



CUTLASS

CUDA C++ Templates for Deep Learning and Linear Algebra



CUTLASS

What's new?

CUTLASS 2.2: optimal performance on NVIDIA Ampere Architecture

- Higher throughput Tensor Cores: more than 2x speedup for all data types
- New floating-point types: bfloat16, Tensor Float 32, double
- Deep software pipelines with `cp.async`: efficient and latency tolerant

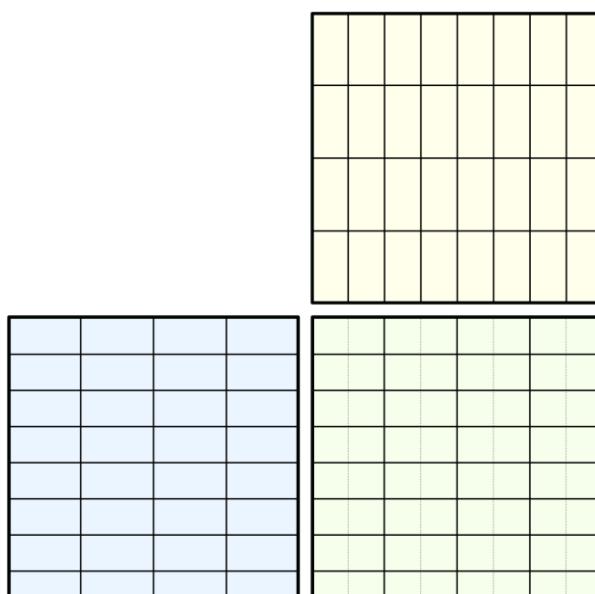
CUTLASS 2.1

- Planar complex: complex-valued GEMMs with batching options targeting Volta and Turing Tensor Cores
- BLAS-style host side API

CUTLASS 2.0: significant refactoring using modern C++11 programming

- Efficient: particularly for Turing Tensor Cores
- Tensor Core programming model: reusable components for linear algebra kernels in CUDA
- Documentation, profiling tools, reference implementations, SDK examples, more..

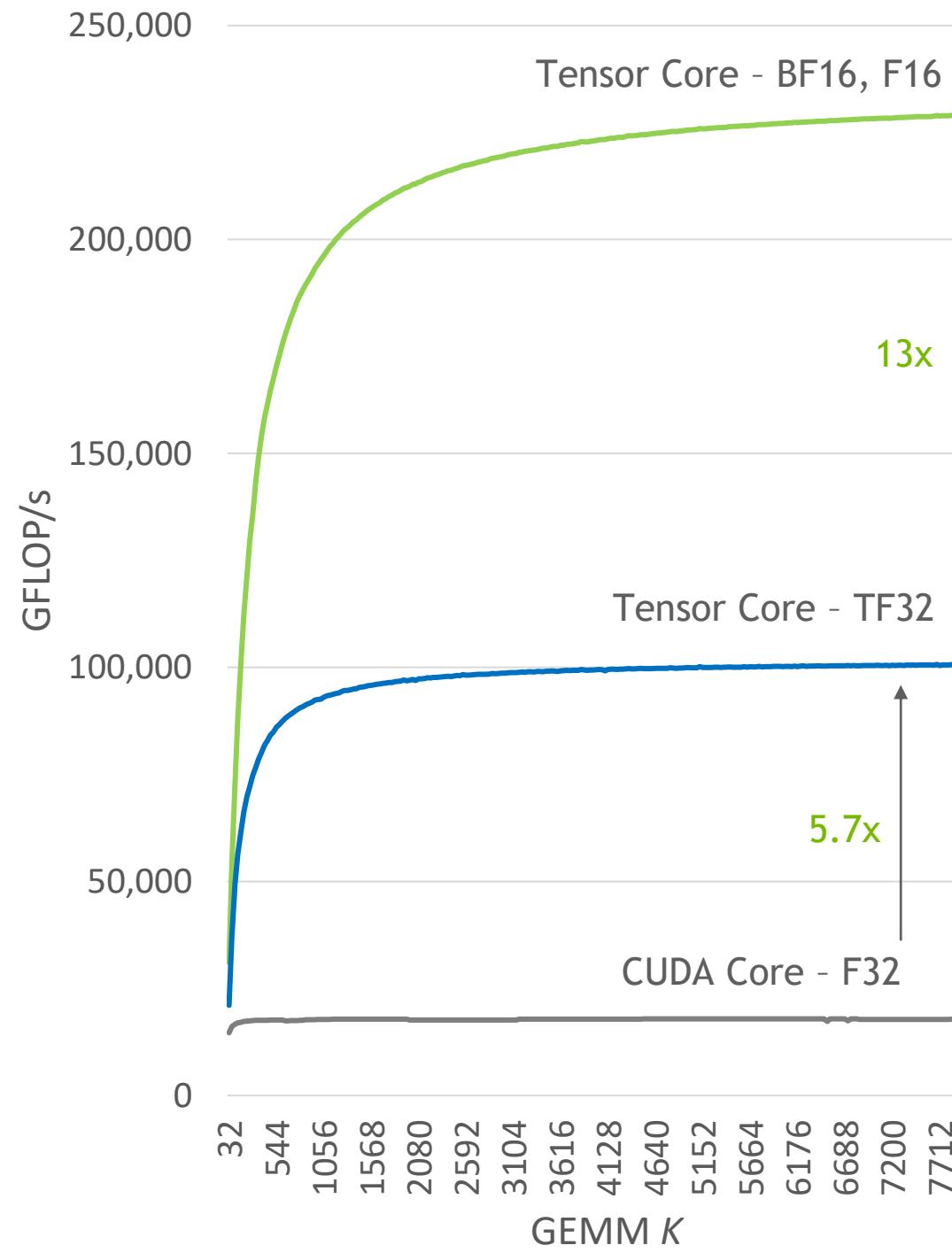
<https://github.com/NVIDIA/cutlass>



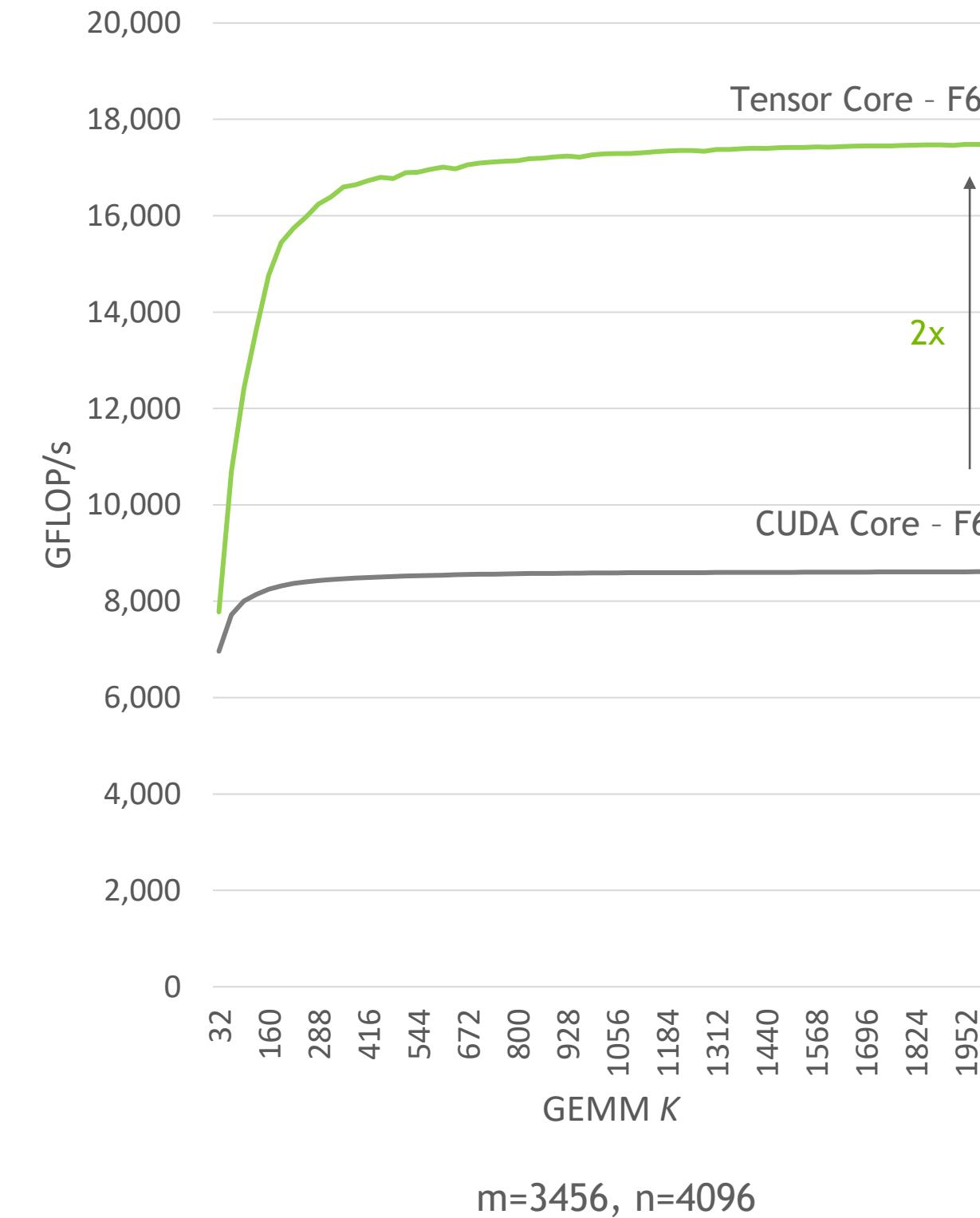
CUTLASS PERFORMANCE ON NVIDIA AMPERE ARCHITECTURE

CUTLASS 2.2 - CUDA 11 Toolkit - NVIDIA A100

Mixed Precision Floating Point



Double Precision Floating Point





TENSOR CORES ON NVIDIA AMPERE ARCHITECTURE

WHAT ARE TENSOR CORES?

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

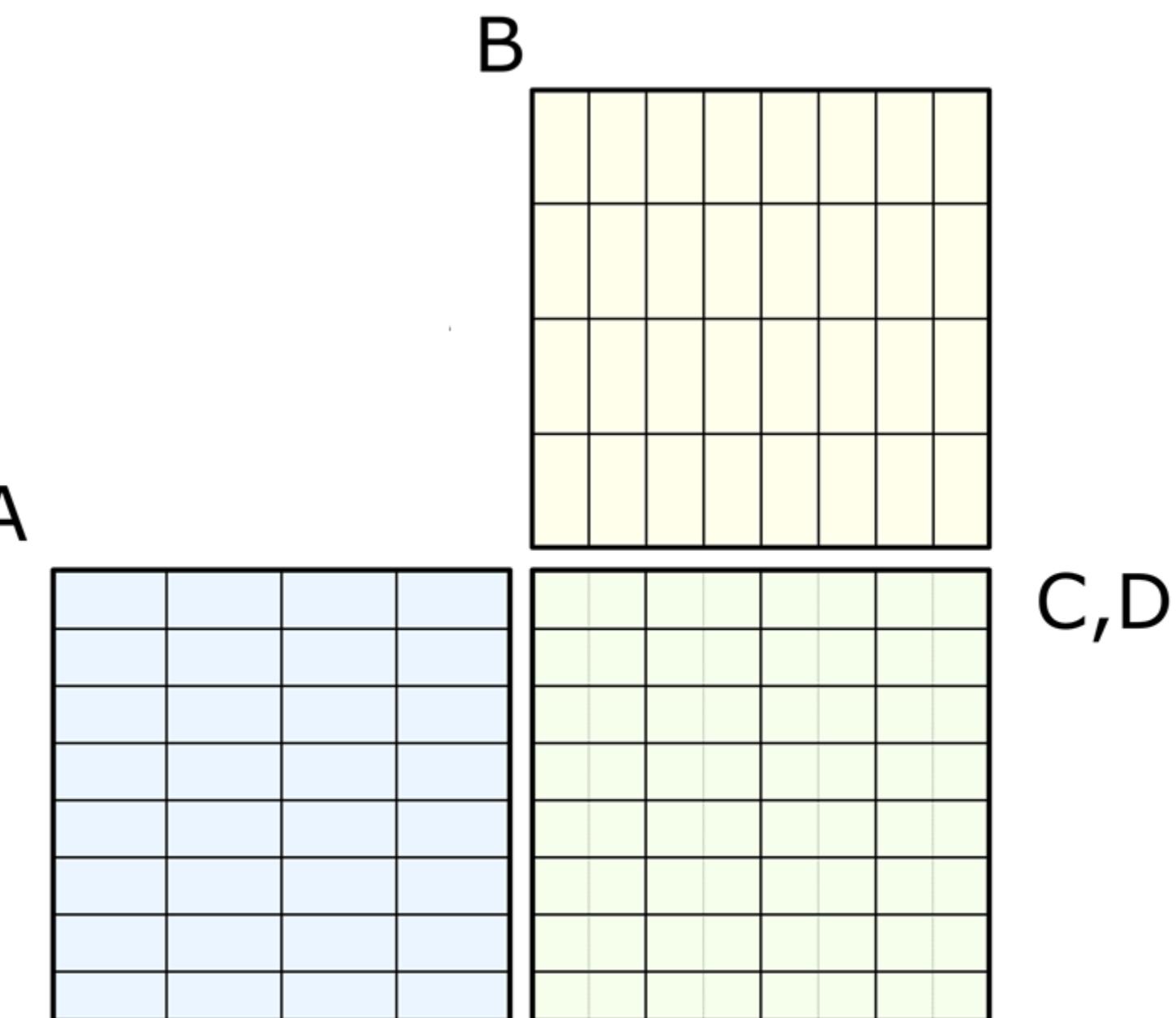
- Matrix multiply-add
- XOR-POPC

Input Data types: A, B

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

Accumulation Data Types: C, D

- half, float, int32_t, double



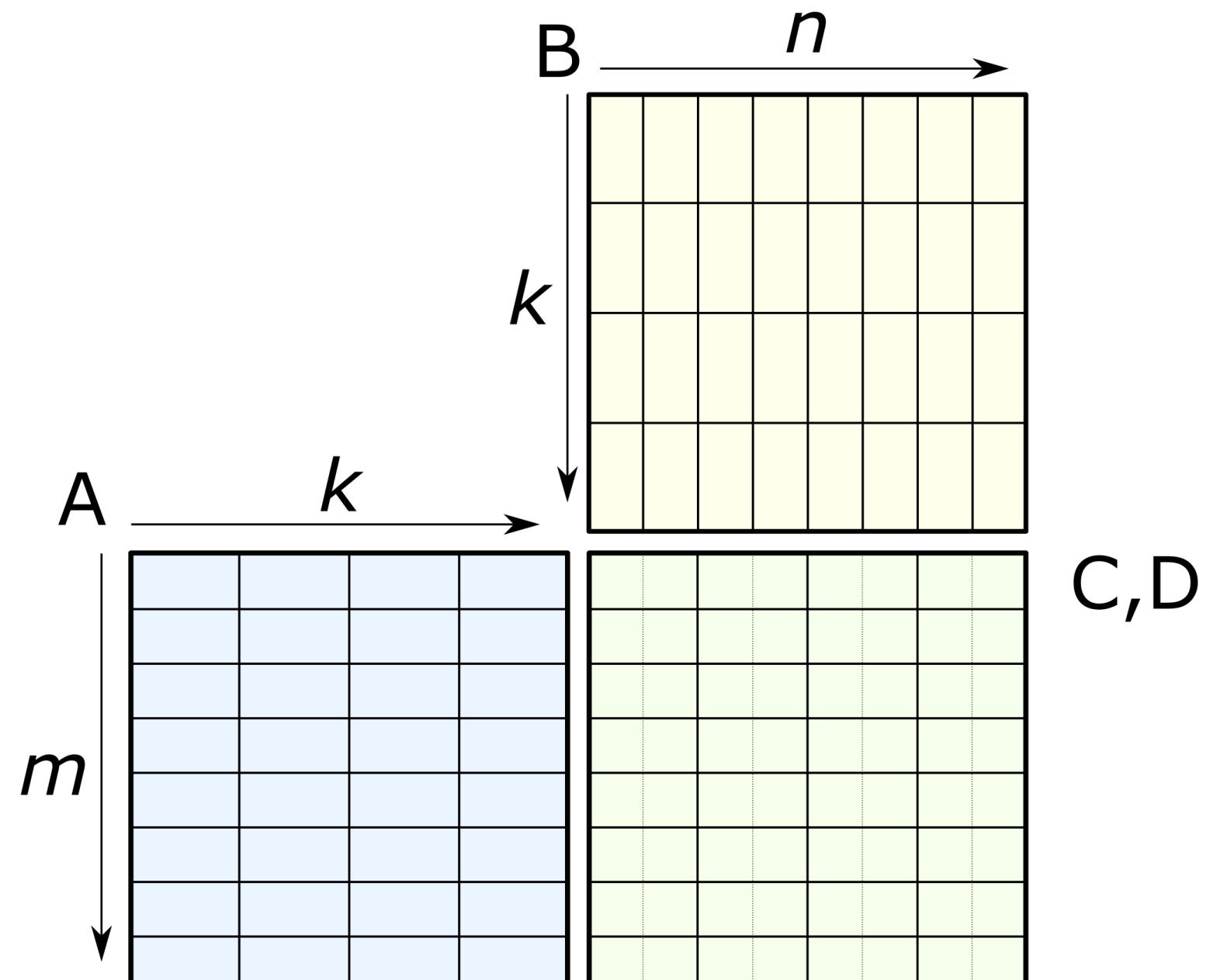
WHAT ARE TENSOR CORES?

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

- Matrix multiply-add
- XOR-POPC

M -by- N -by- K matrix operation

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



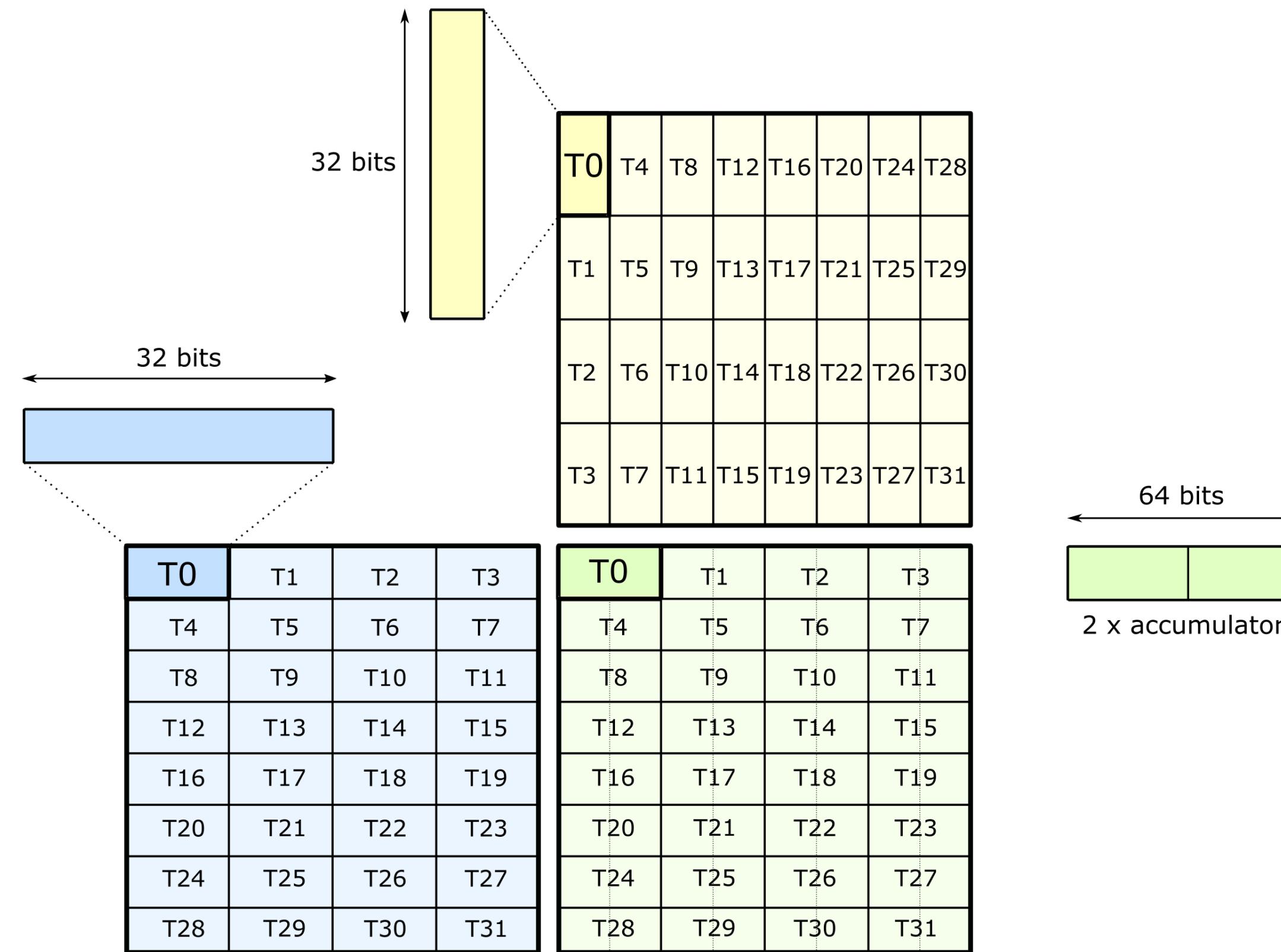
NVIDIA AMPERE ARCHITECTURE - TENSOR CORE OPERATIONS

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

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

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

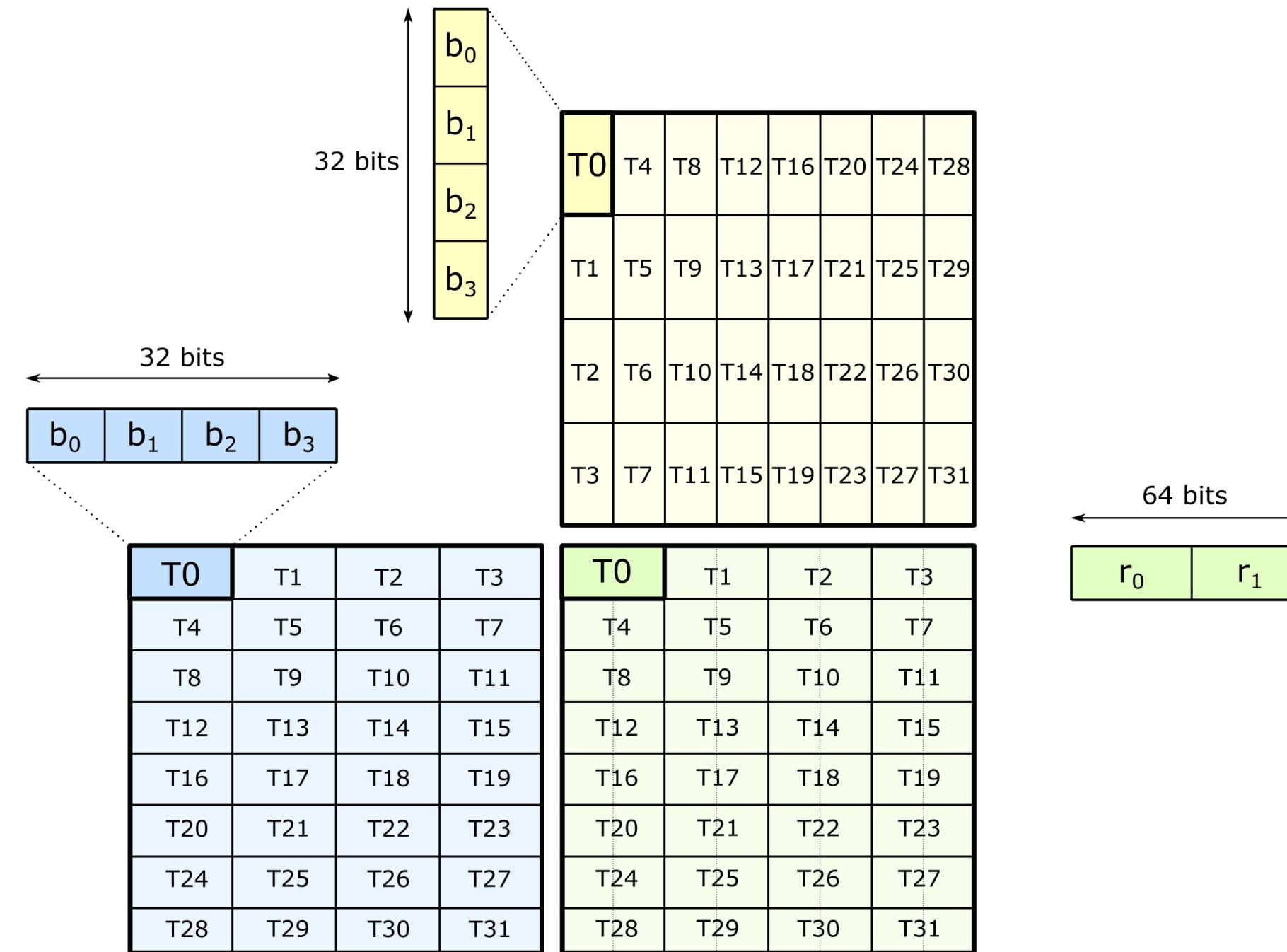
TENSOR CORE OPERATION: FUNDAMENTAL SHAPE



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

$S8 * S8 + S32$

8-by-8-by-16

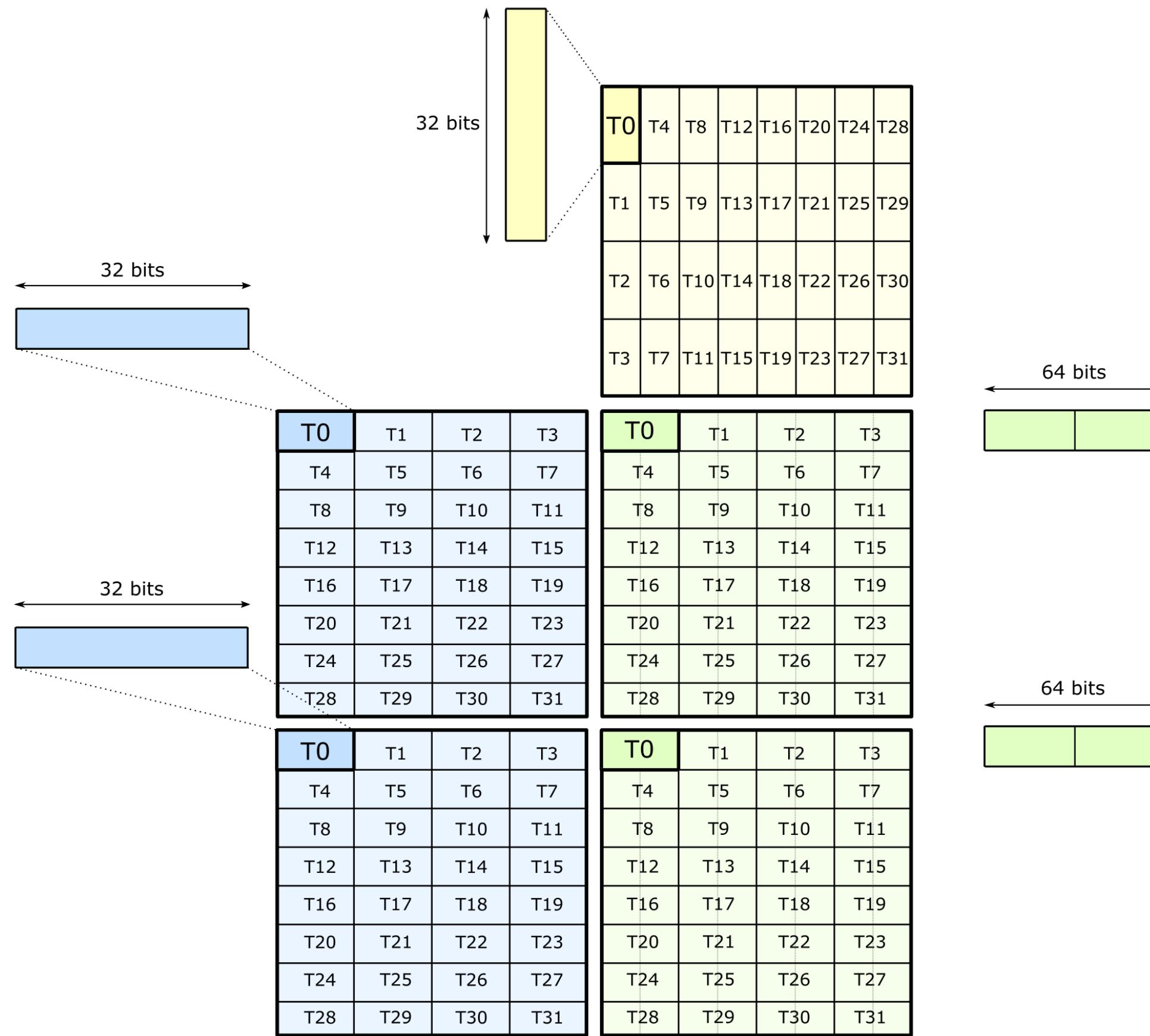
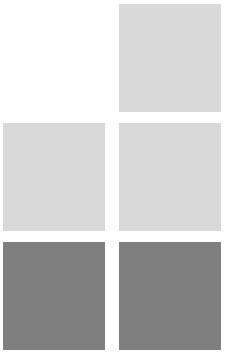


mma.sync.aligned
(via inline PTX)

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

```
// Example targets 8-by-8-by-16 Tensor Core operation
asm(
    "mma.sync.aligned.m8n8k16.row.col.s32.s8.s8.s32 "
    " { %0, %1 }, "
    " %2,
    " %3,
    " { %4, %5 }; "
    :
    "=r"(D[0]), "=r"(D[1])
    :
    "r"(A),    "r"(B),
    "r"(C[0]), "r"(C[1])
);
```

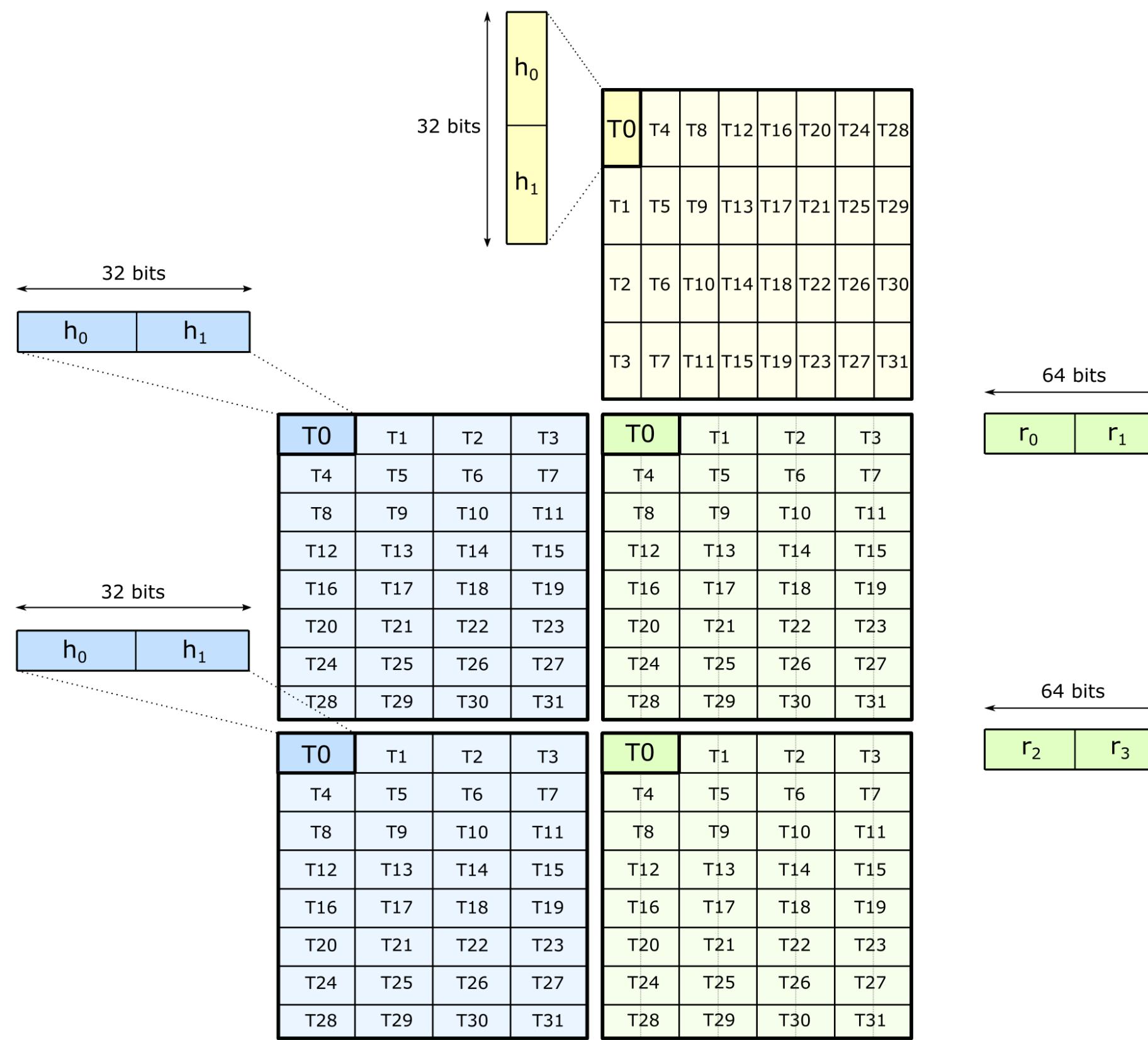
EXPANDING THE M DIMENSION



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

F16 * F16 + F32

16-by-8-by-8

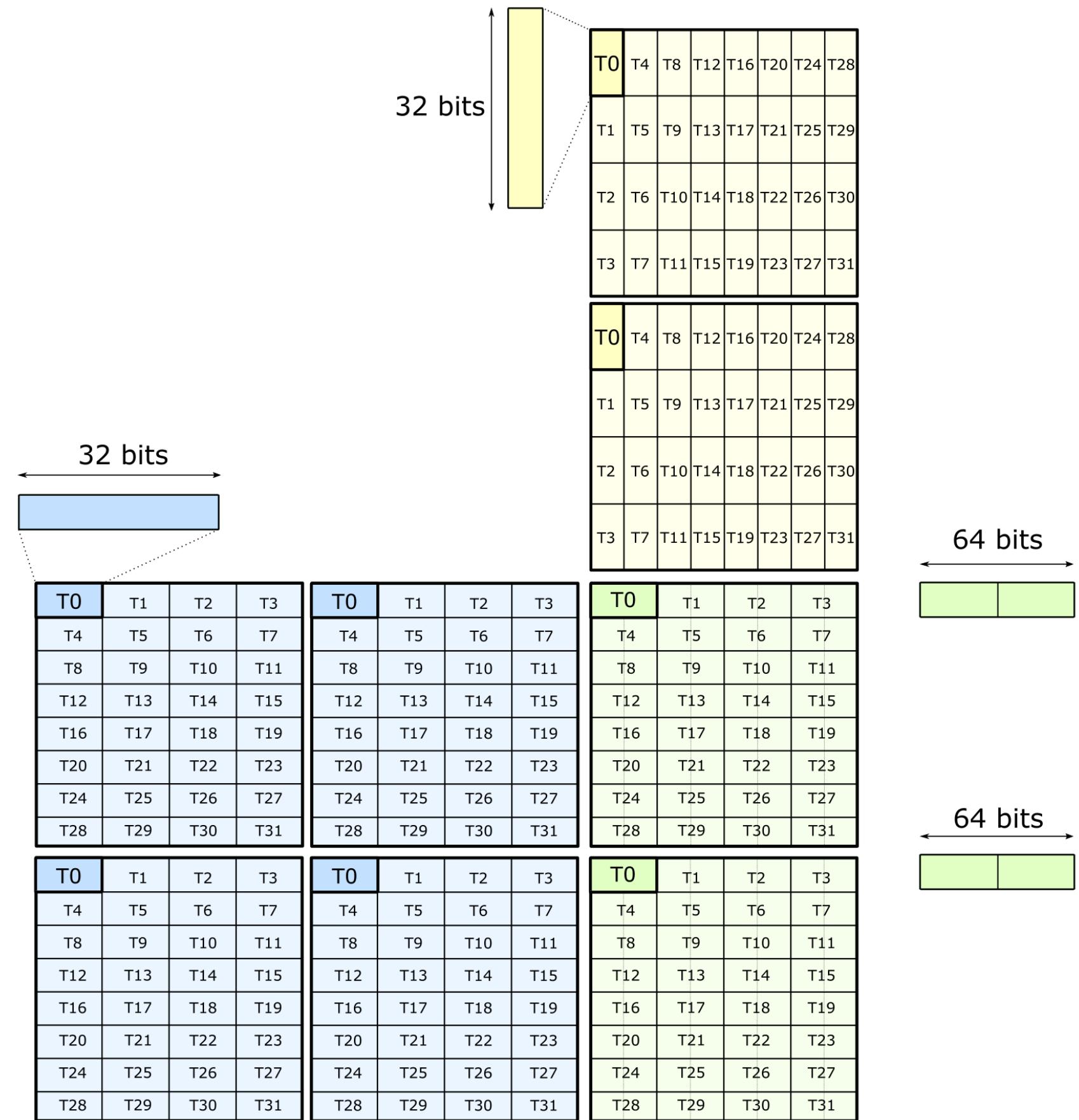


mma.sync.aligned (via inline PTX)

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

```
// Example targets 16-by-8-by-8 Tensor Core operation
asm(
    "mma.sync.aligned.m16n8k8.row.col.f32.f16.f16.f32 "
    " { %0, %1, %2, %3 }, "
    " { %4, %5}, "
    " %6,
    " { %7, %8, %9, %10 };"
    :
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
    :
    "r"(A[0]), "r"(A[1]),
    "r"(B),
    "f"(C[0]), "f"(C[1])
);
```

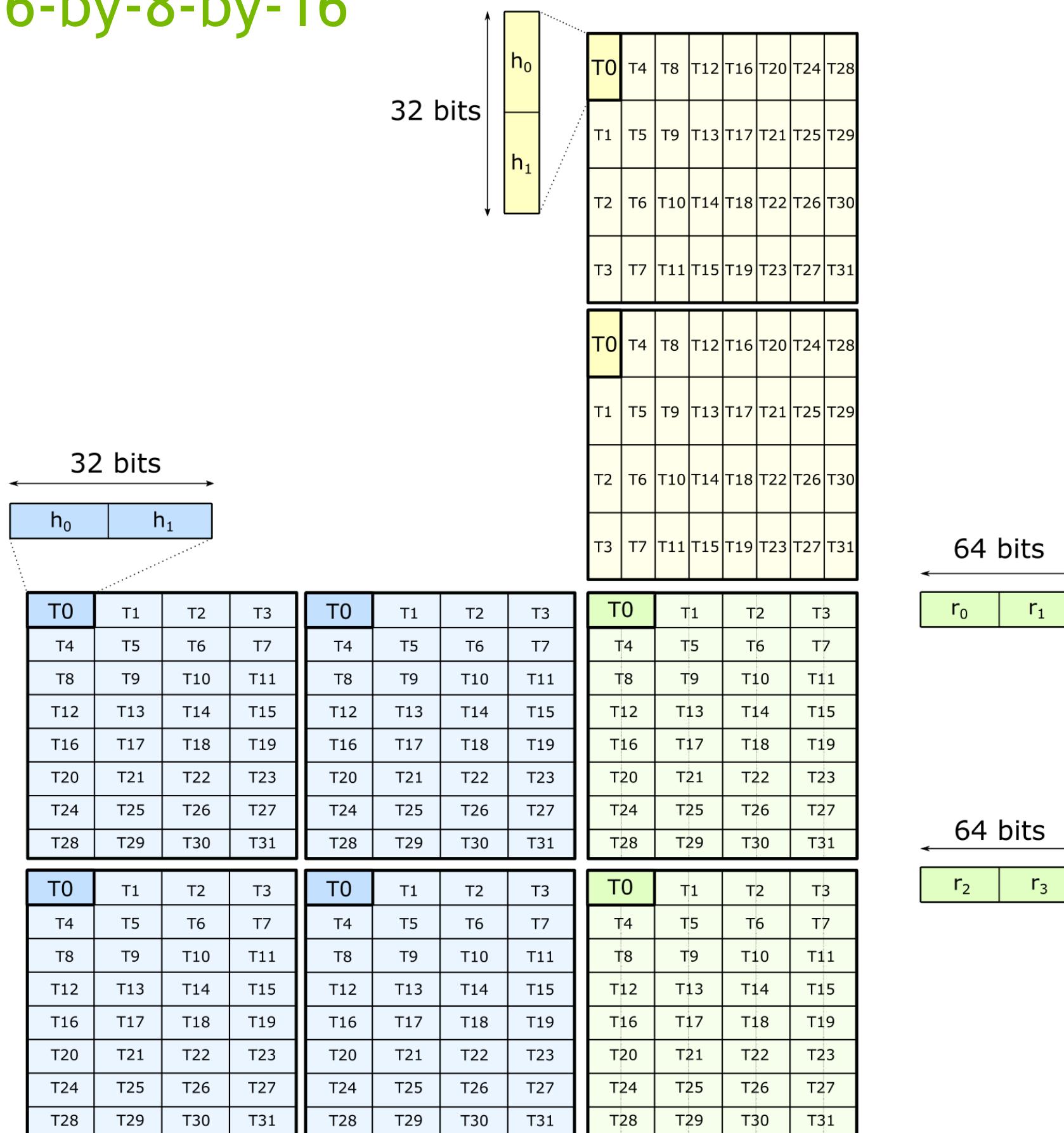
EXPANDING THE K DIMENSION



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

F16 * F16 + F32

16-by-8-by-16



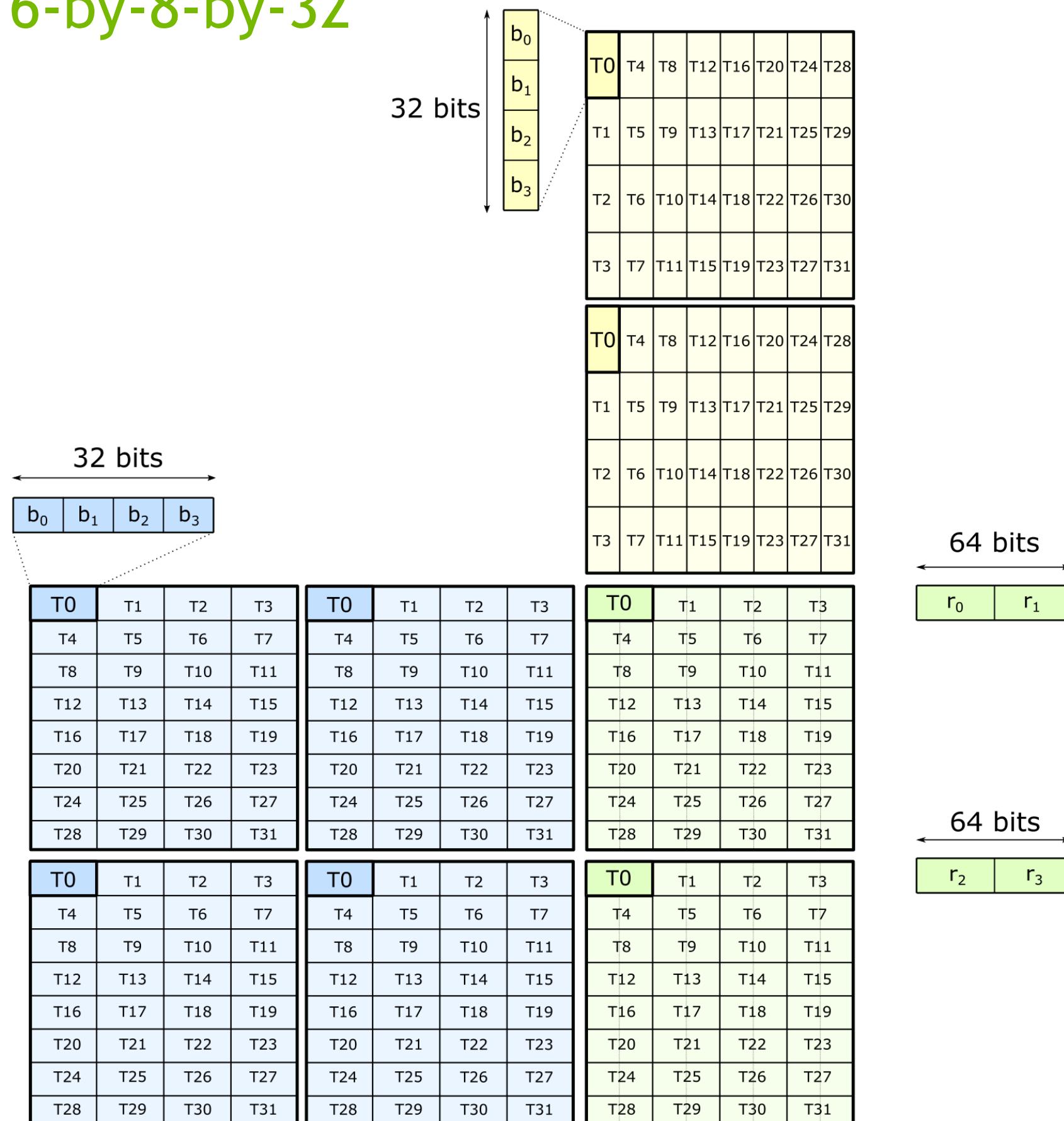
mma.sync.aligned (via inline PTX)

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

```
// Example targets 16-by-8-by-32 Tensor Core operation
asm(
    "mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 "
    " { %0, %1, %2, %3 }, "
    " { %4, %5, %6, %7 }, "
    " { %8, %9 },
    " { %10, %11, %12, %13 };"
    :
    "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
    :
    "r"(A[0]), "r"(A[1]), "r"(A[2]), "r"(A[3]),
    "r"(B[0]), "r"(B[1]),
    "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3])
);
```

$S8 * S8 + S32$

16-by-8-by-32



mma.sync.aligned
(via inline PTX)

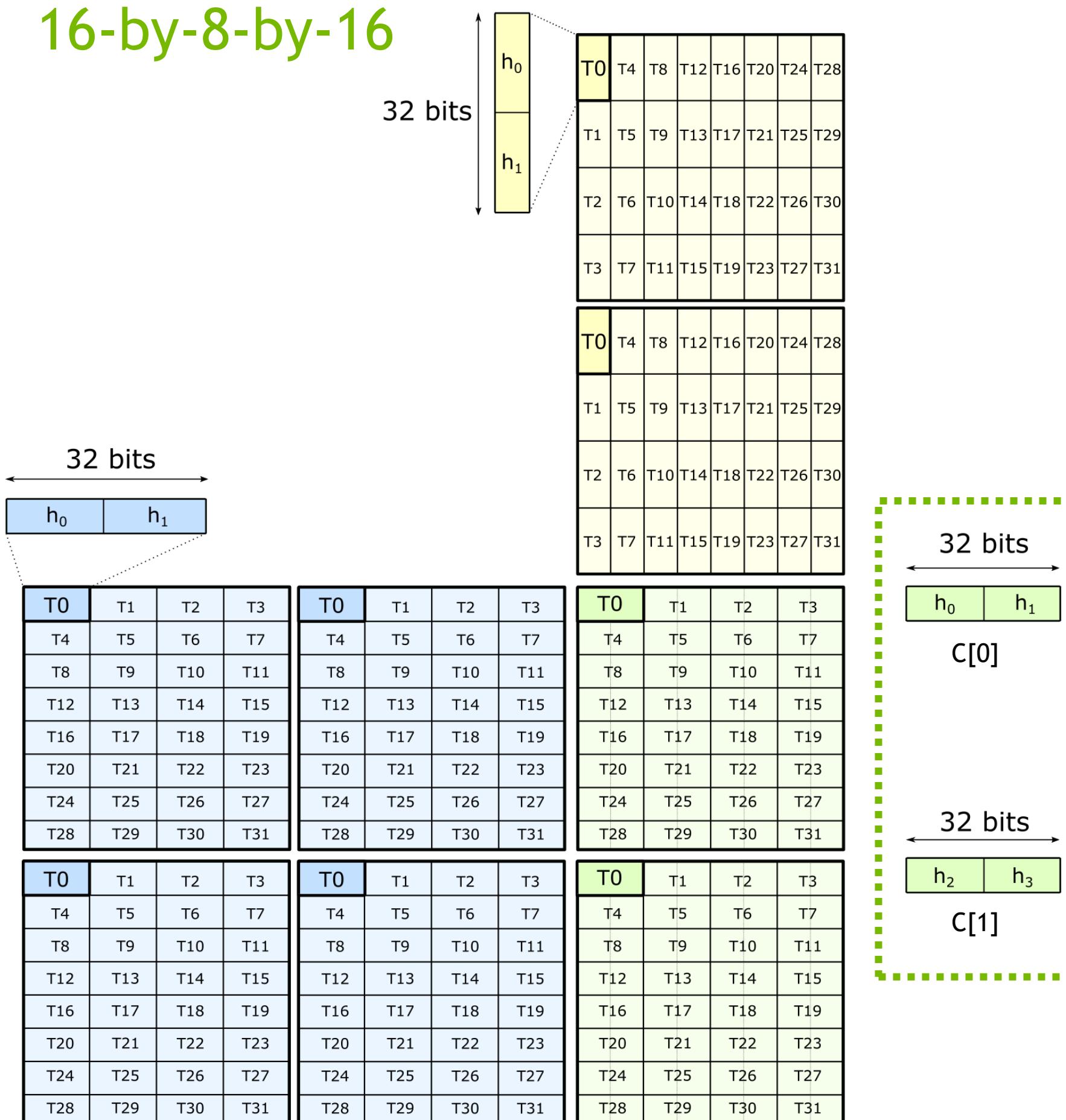
```
int32_t      D[4];
uint32_t const A[4];
uint32_t const B[2];
int32_t const C[4];
```

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

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

HALF-PRECISION : F16 * F16 + F16

16-by-8-by-16



mma.sync.aligned
(via inline PTX)

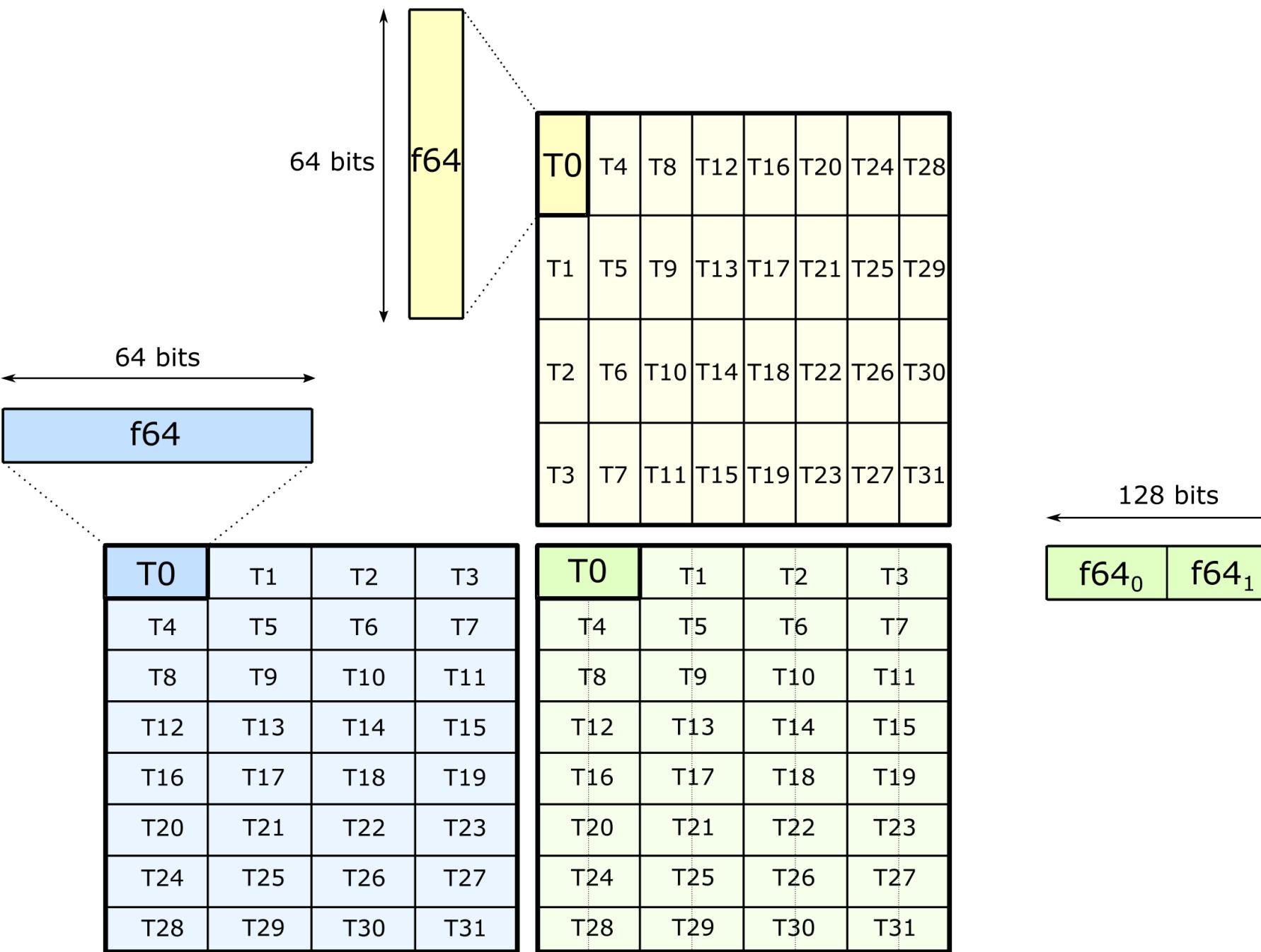
```
uint32_t      D[2]; // two registers needed (vs. four)
uint32_t const A[4];
uint32_t const B[2];
uint32_t const C[2]; // two registers needed (vs. four)
```

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

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

DOUBLE-PRECISION: F64 * F64 + F64

8-by-8-by-4



mma.sync.aligned
(via inline PTX)

```

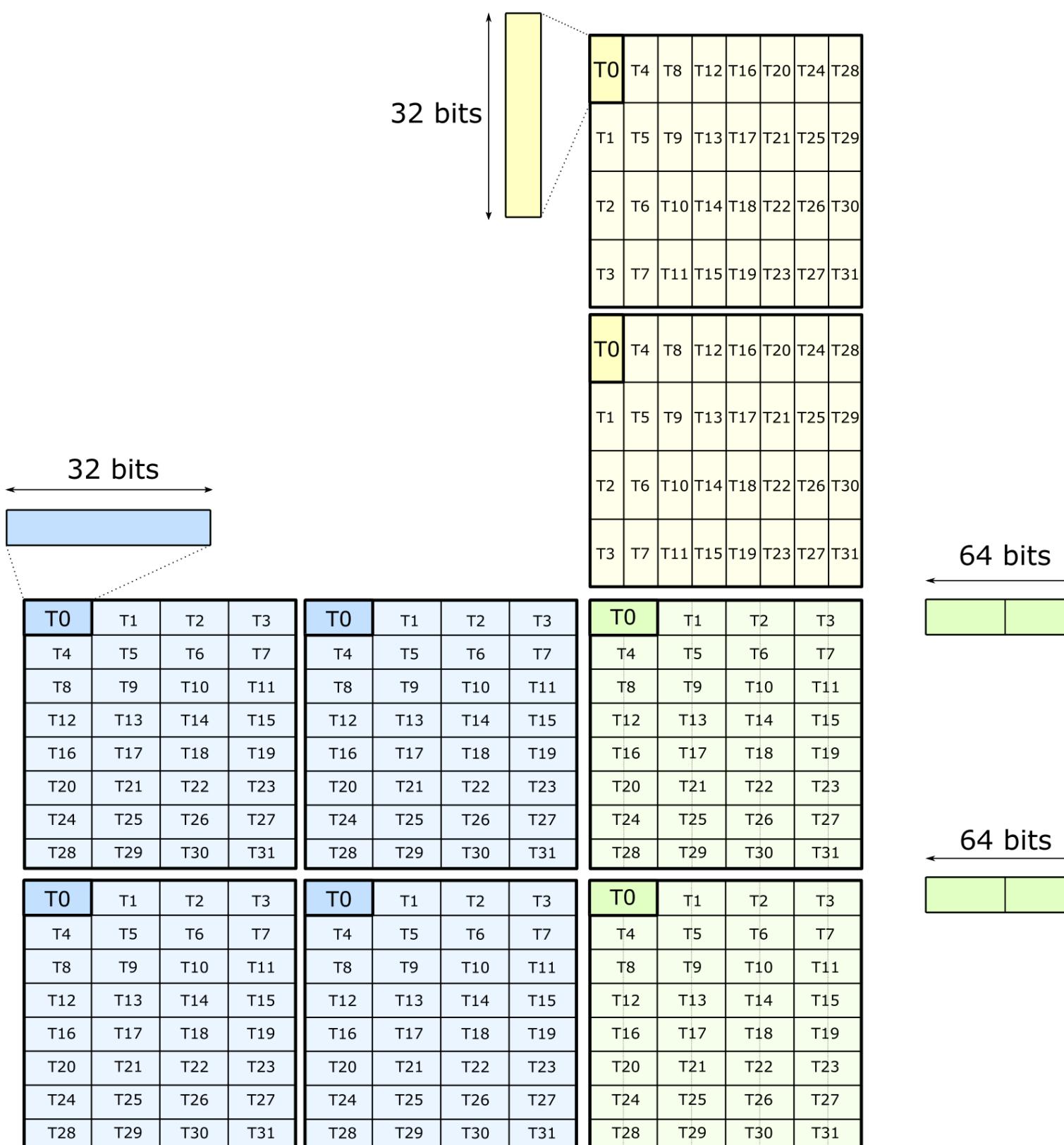
uint64_t      D[2];      // two 64-bit accumulators
uint64_t const A;        // one 64-bit element for A operand
uint64_t const B;        // one 64-bit element for B operand
uint64_t const C[2];     // two 64-bit accumulators

// Example targets 8-by-8-by-4 Tensor Core operation
asm(
    "mma.sync.aligned.m8n8k4.row.col.f64.f64.f64.f64 "
    " { %0, %1}, "
    " %2, "
    " %3, "
    " { %4, %5 }; "
    "=l"(D[0]), "=l"(D[1])
    :
    "l"(A),
    "l"(B),
    "l"(C[0]), "l"(C[1])
);

```

CUTLASS: wraps PTX in template

m-by-n-by-k

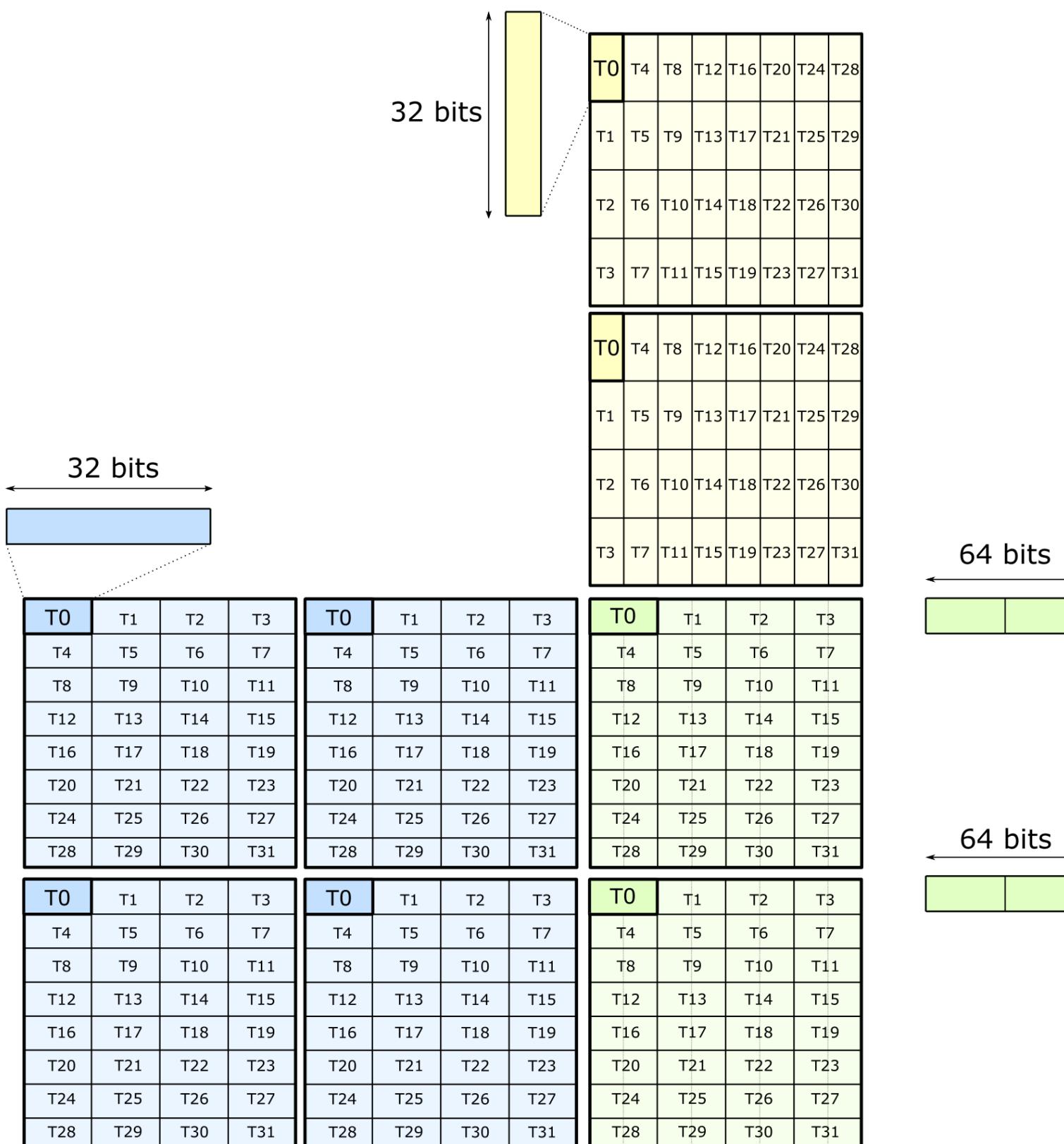


`cutlass::arch::Mma`

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

CUTLASS: wraps PTX in template

16-by-8-by-16



cutlass::arch::Mma

```
__global__ void kernel() {  
  
    // arrays containing logical elements  
    Array<half_t, 8> A;  
    Array<half_t, 4> B;  
    Array< float, 4> C;  
  
    // define the appropriate matrix operation  
    arch::Mma< GemmShape<16, 8, 16>, 32, ... > mma;  
  
    // in-place matrix multiply-accumulate  
    mma(C, A, B, C);  
  
    ...  
}
```



EFFICIENT DATA MOVEMENT
FOR TENSOR CORES

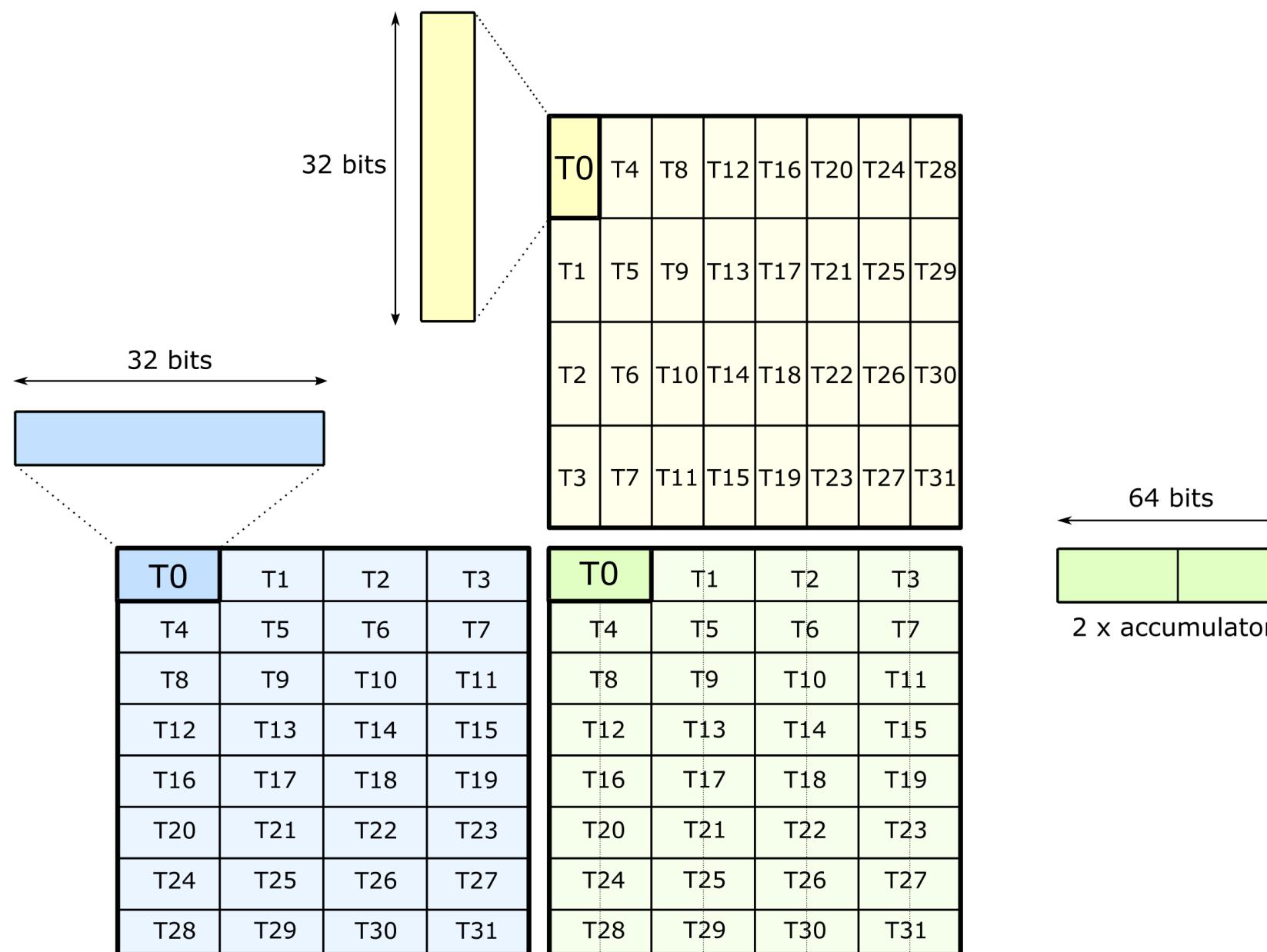
HELLO WORLD: TENSOR CORES

Map each thread to coordinates of the matrix operation

Load inputs from memory

Perform the matrix operation

Store the result to memory



CUDA example

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

    // Compute the coordinates of accesses to A and B matrices
    int outer = threadIdx.x / 4;           // m or n dimension
    int inner = threadIdx.x % 4;           // k dimension

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

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

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

PERFORMANCE IMPLICATIONS

Load A and B inputs from memory: $2 \times 4B$ per thread

Perform one Tensor Core operation: 2048 flops per warp

2048 flops require 256 B of loaded data

→ 8 flops/byte

NVIDIA A100 Specifications:

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

→ 400 flops/byte

8 flops/byte * 1.6 TB/s → 12 TFLOP/s

This kernel is global memory bandwidth limited.

CUDA example

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

    // Compute the coordinates of accesses to A and B matrices
    int outer = threadIdx.x / 4;      // m or n dimension
    int inner = threadIdx.x % 4;      // k dimension

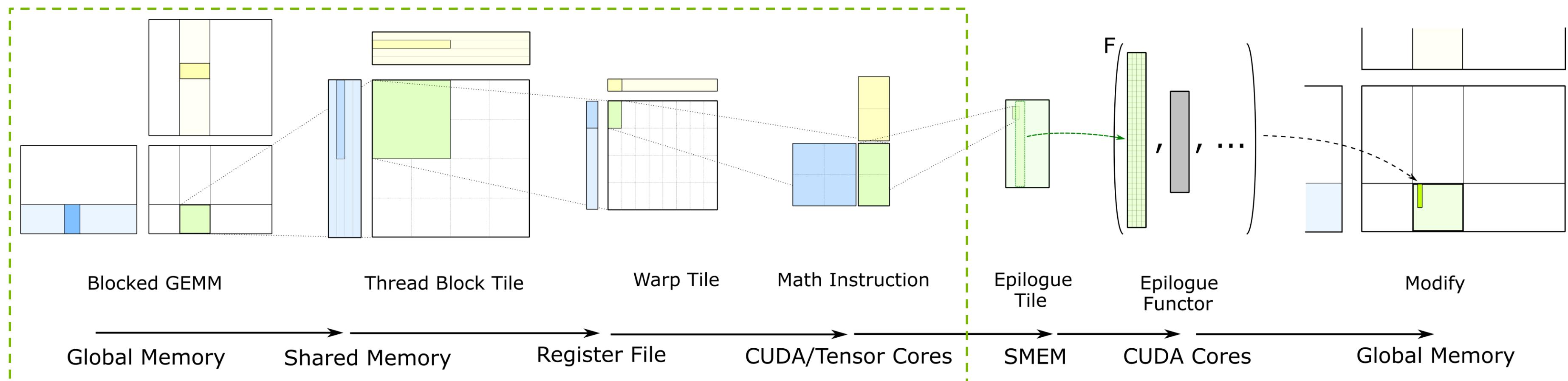
    // Compute the coordinates for the accumulator matrices
    int c_row = threadIdx.x / 4;
    int c_col = 2 * (threadIdx.x % 4);

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

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

FEEDING THE DATA PATH

Efficient storing and loading through Shared Memory



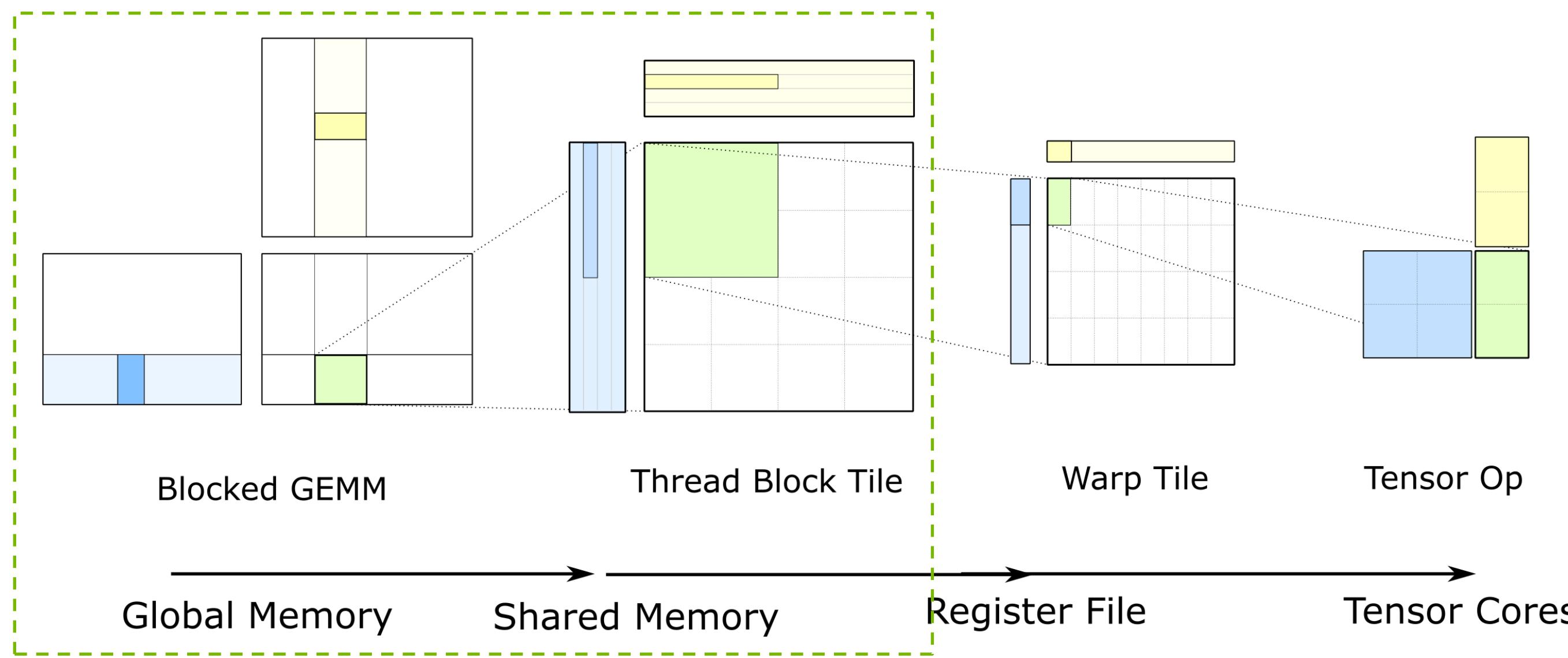
Tiled, hierarchical model: reuse data in Shared Memory and in Registers

See [CUTLASS GTC 2018 talk](#) for more details about this model.

FEEDING THE DATA PATH

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

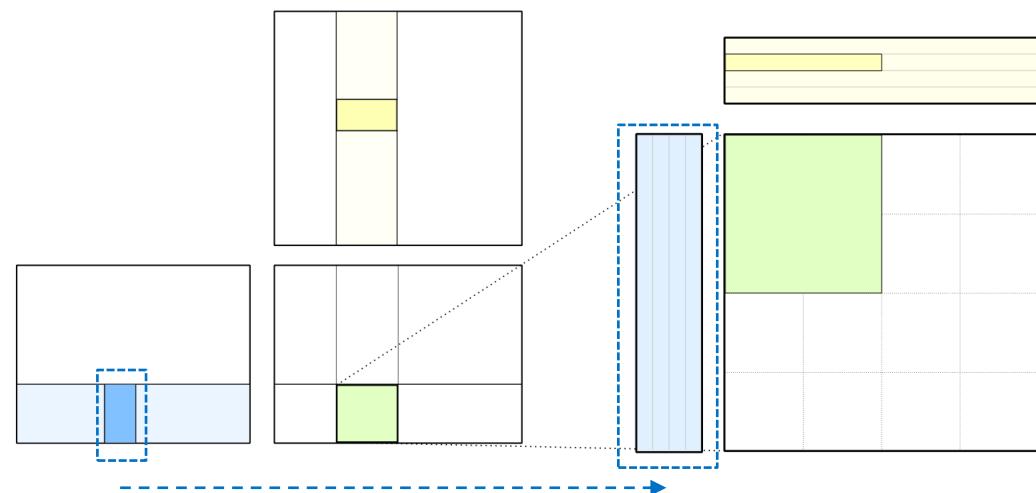
- Latency-tolerant pipeline from Global Memory
- Conflict-free Shared Memory stores
- Conflict-free Shared Memory loads



ASYNCHRONOUS COPY: EFFICIENT PIPELINES

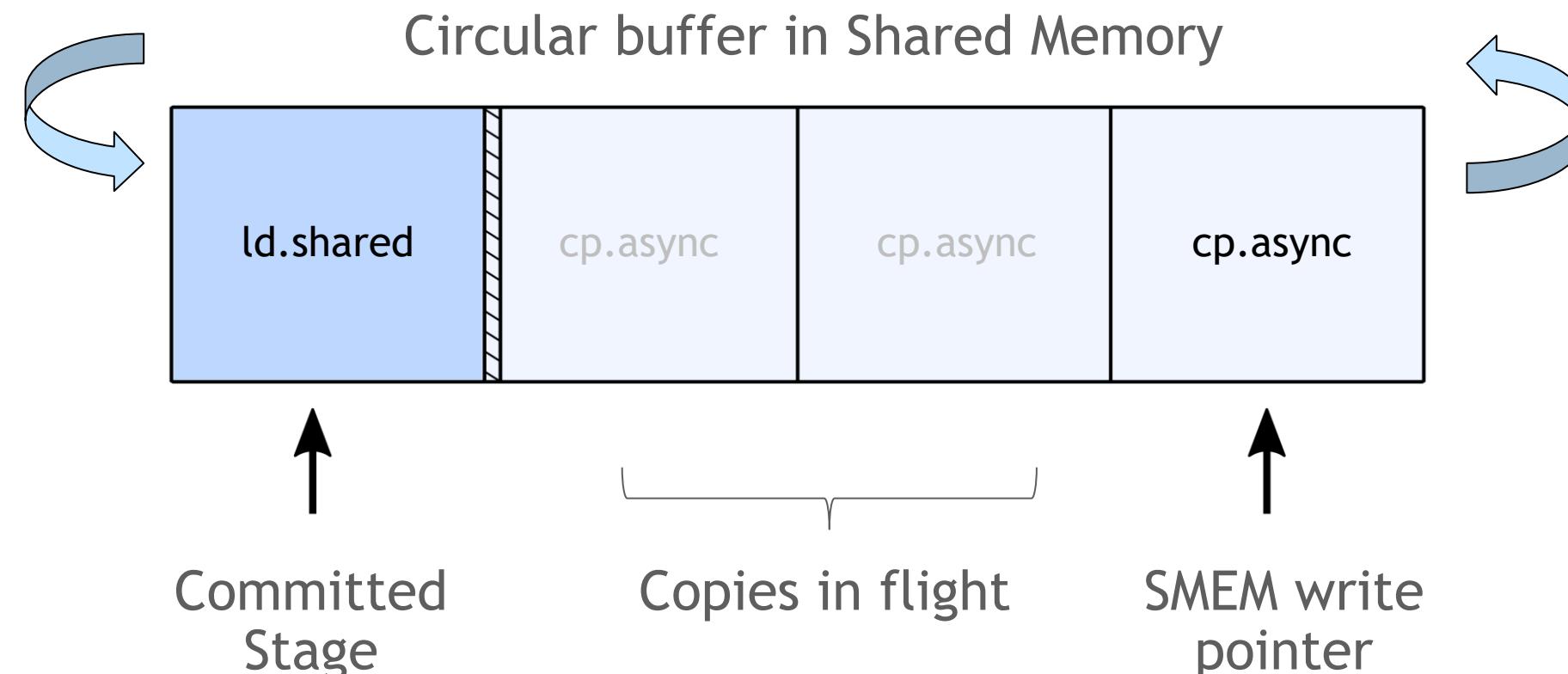
New NVIDIA Ampere Architecture feature: cp.async

- Asynchronous copy directly from Global to Shared Memory
- See “*Inside the NVIDIA Ampere Architecture*” for more details (GTC 2020 - S21730)



Enables efficient software pipelines

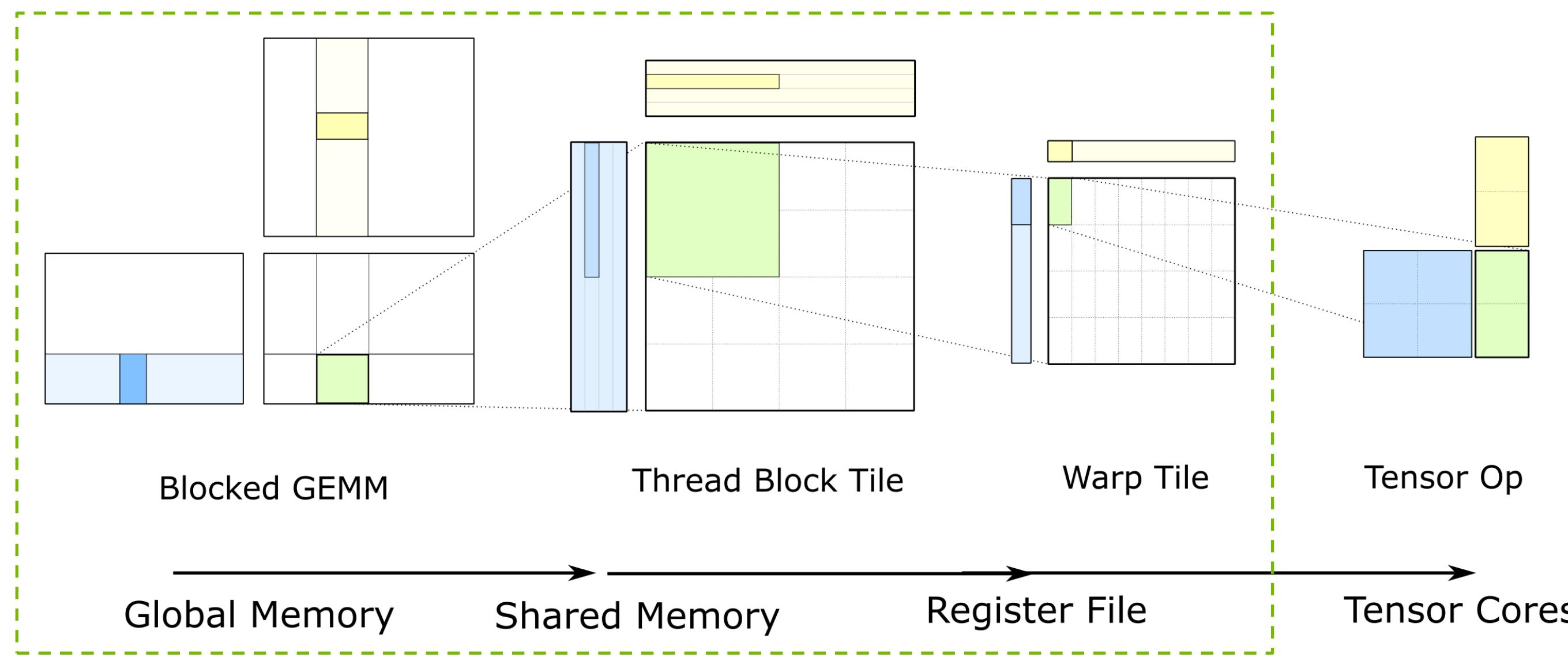
- Minimizes data movement: L2 → L1 → RF → SMEM becomes L2 → SMEM
- Saves registers: RF no longer needed to hold the results of long-latency load instructions
- Indirection: fetch several stages in advance for greater latency tolerance



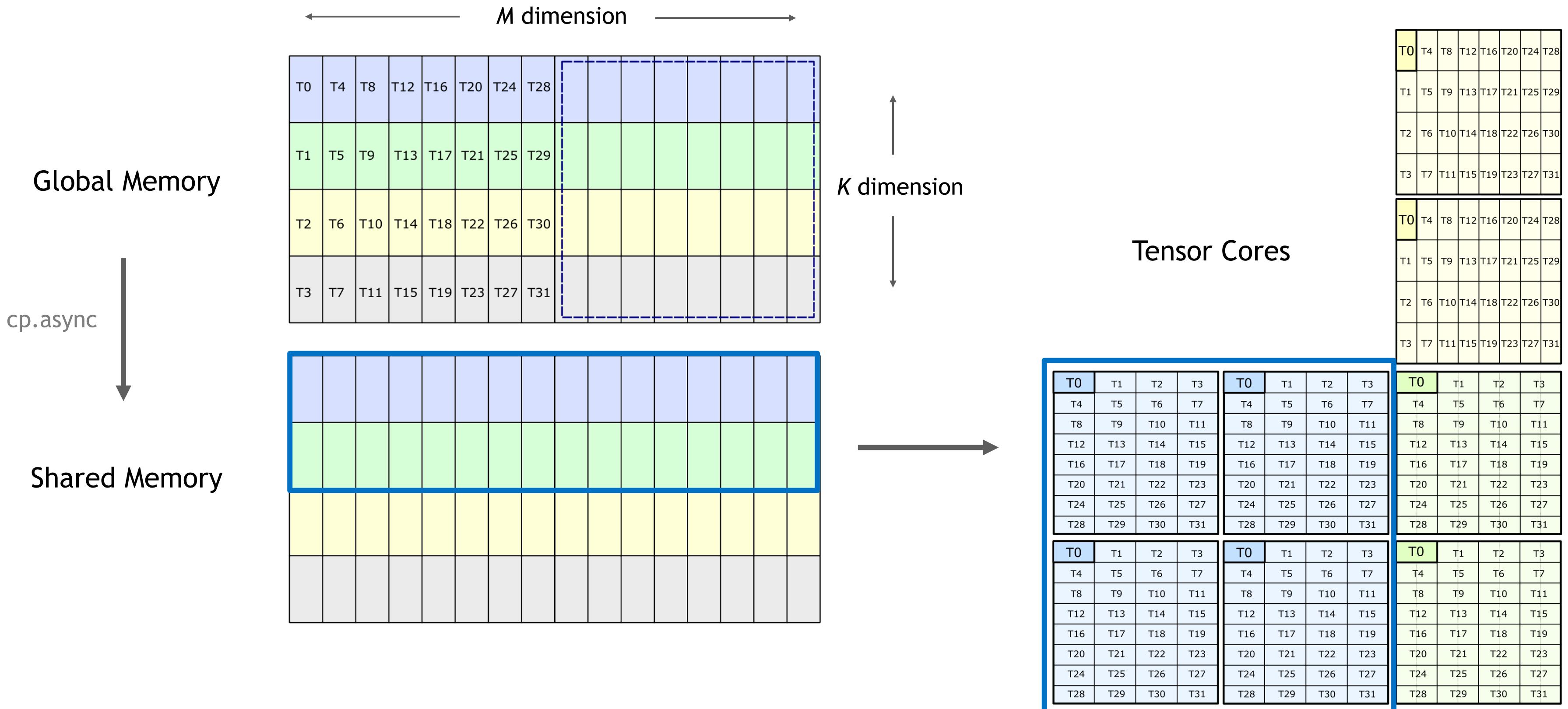
FEEDING THE DATA PATH

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

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



GLOBAL MEMORY TO TENSOR CORES



LDMATRIX: FETCH TENSOR CORE OPERANDS

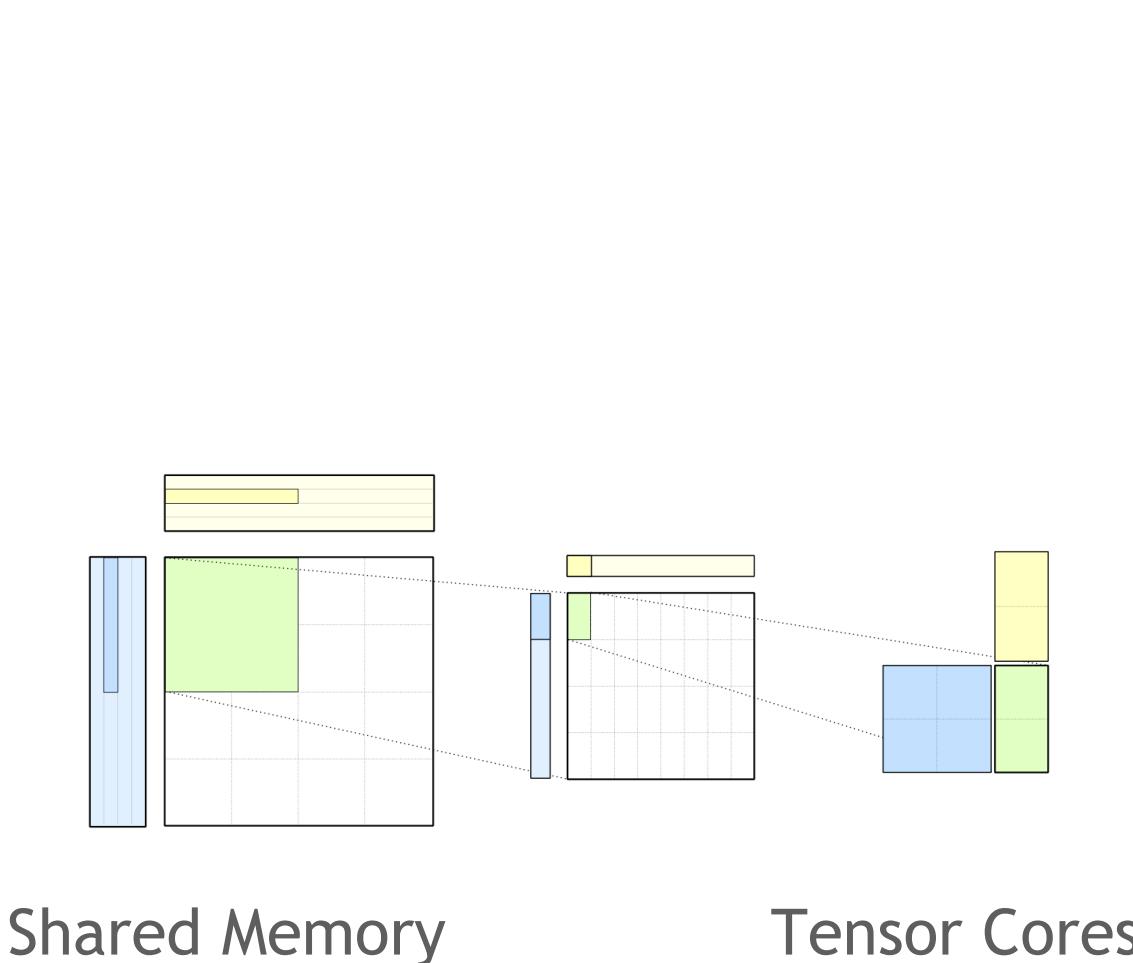
PTX instruction to load a matrix from Shared Memory

Each thread supplies a pointer to 128b row of data in Shared Memory

Each 128b row is broadcast to groups of four threads

(potentially different threads than the one supplying the pointer)

Data matches arrangement of inputs to Tensor Core operations



Shared Memory Pointers	
T0	→
T1	→
T2	→
T3	→
T8	→
T9	→
T10	→
T11	→
T12	→
T13	→
T14	→
T15	→
T16	→
T17	→
T18	→
T19	→
T20	→
T21	→
T22	→
T23	→
T24	→
T25	→
T26	→
T27	→
T28	→
T29	→
T30	→
T31	→

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				

LDMATRIX: PTX INSTRUCTION

PTX instruction to load a matrix from SMEM

Each thread supplies a pointer to 128b row of data in Shared Memory

Each 128b row is broadcast to groups of four threads

(potentially different threads than the one supplying the pointer)

Data matches arrangement of inputs to Tensor Core operations

```
// Inline PTX assembly for ldmatrix

uint32_t R[4];
uint32_t smem_ptr;

asm volatile (
    "ldmatrix.sync.aligned.x4.m8n8.shared.b16 "
    "{%0, %1, %2, %3}, [%4];"
    :
    "=r"(R[0]), "=r"(R[1]), "=r"(R[2]), "=r"(R[3])
    :
    "r"(smem_ptr)
);
```

Shared Memory
Pointers

T0 →
T1 →
T2 →
T3 →
T4 →
T5 →
T6 →
T7 →

T8 →
T9 →
T10 →
T11 →
T12 →
T13 →
T14 →
T15 →

T16 →
T17 →
T18 →
T19 →
T20 →
T21 →
T22 →
T23 →

T24 →
T25 →
T26 →
T27 →
T28 →
T29 →
T30 →
T31 →

T24 →
T25 →
T26 →
T27 →
T28 →
T29 →
T30 →
T31 →

T24 →
T25 →
T26 →
T27 →
T28 →
T29 →
T30 →
T31 →

T24 →
T25 →
T26 →
T27 →
T28 →
T29 →
T30 →
T31 →

Matrix loaded by warp

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

Data loaded by T0

h ₀	h ₁
R0	

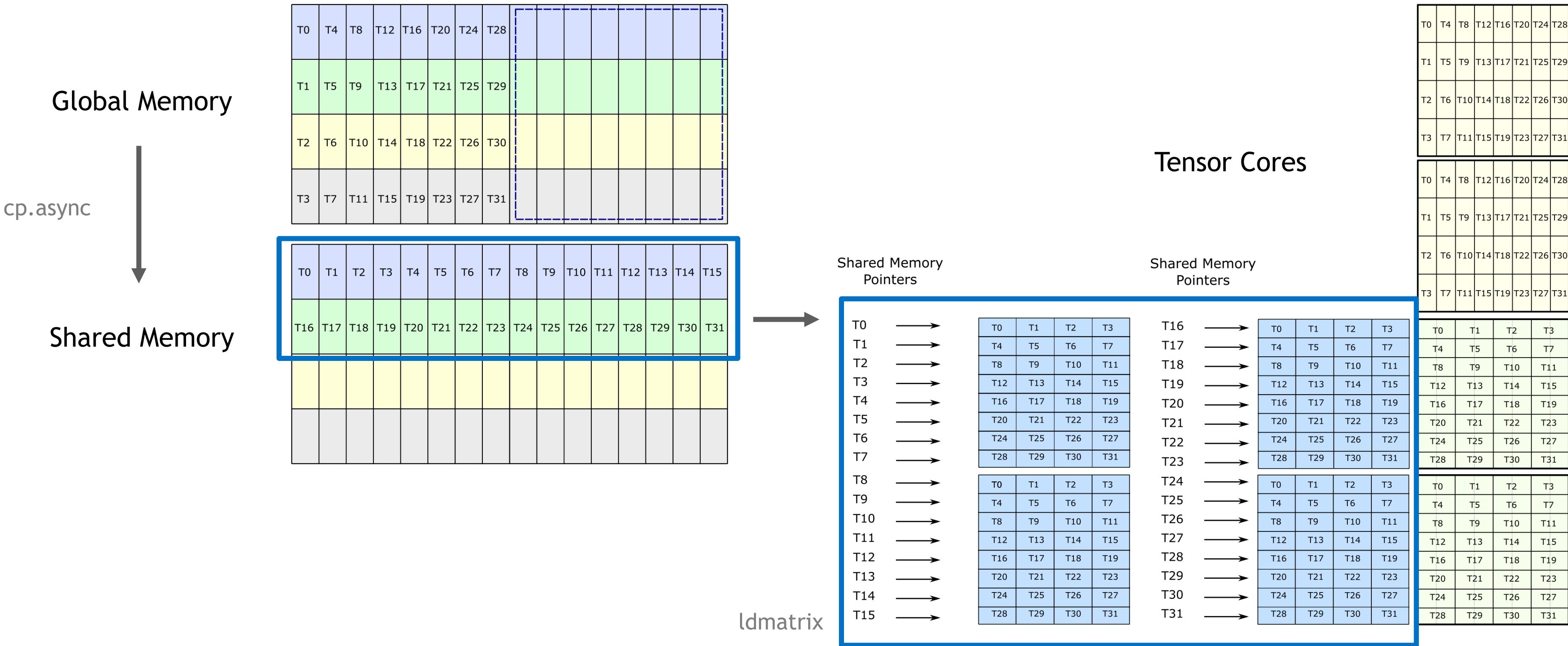
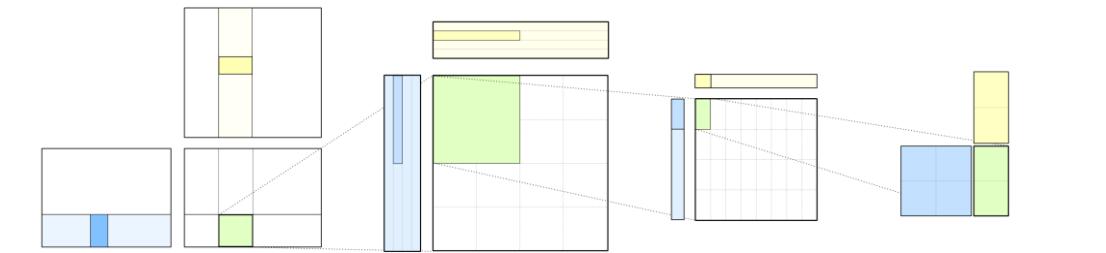
h ₂	h ₃
R1	

h ₄	h ₅
R2	

h ₆	h ₇
R3	

32 bits

GLOBAL MEMORY TO TENSOR CORES



NVIDIA AMPERE ARCHITECTURE - SHARED MEMORY BANK TIMING

Bank conflicts between threads in the same phase

4B words are accessed in 1 phase

8B words are accessed in 2 phases:

- Process addresses of the first 16 threads in a warp
- Process addresses of the second 16 threads in a warp

Phase 0: T0 .. T7

Phase 1: T8 .. T15

Phase 2: T16 .. T23

Phase 3: T24 .. T31

16B words are accessed in 4 phases:

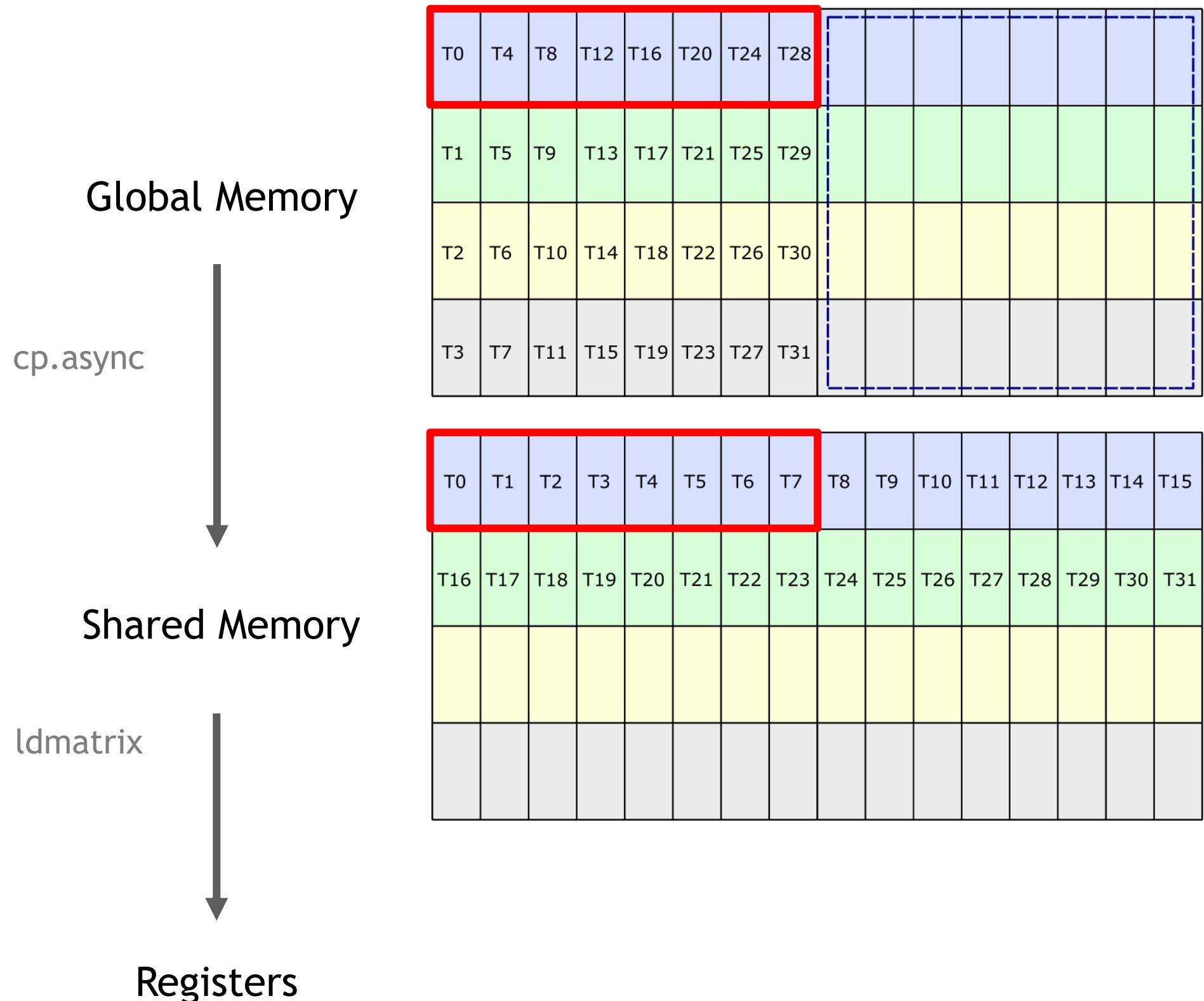
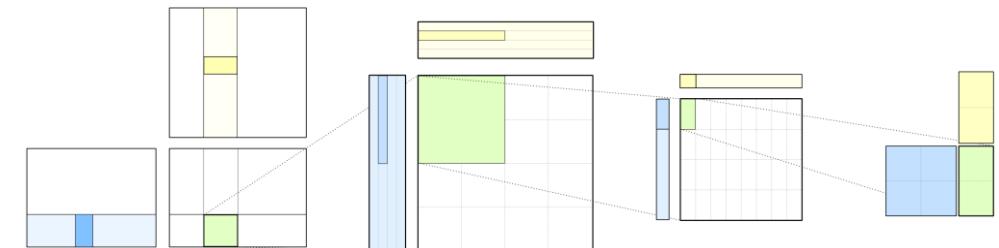
- Each phase processes **8 consecutive threads** of a warp

128 bit access size

Slide borrowed from: Guillaume Thomas-Collignon and Paulius Micikevicius. "Volta Architecture and performance optimization." GTC 2018.

<http://on-demand.gputechconf.com/gtc/2018/presentation/s81006-volta-architecture-and-performance-optimization.pdf>

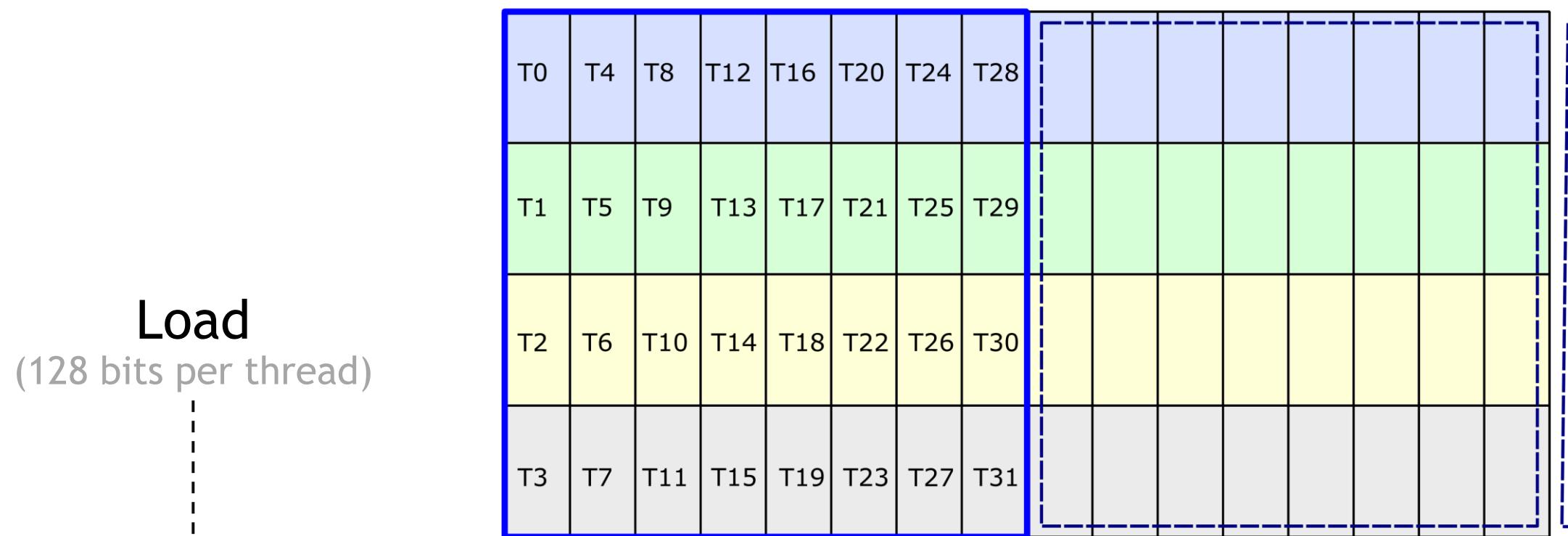
GLOBAL MEMORY TO TENSOR CORES



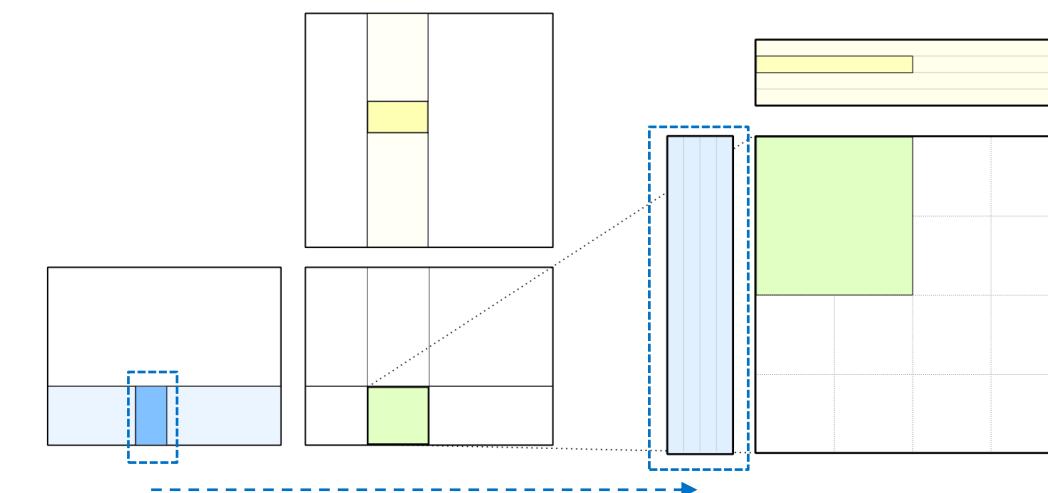
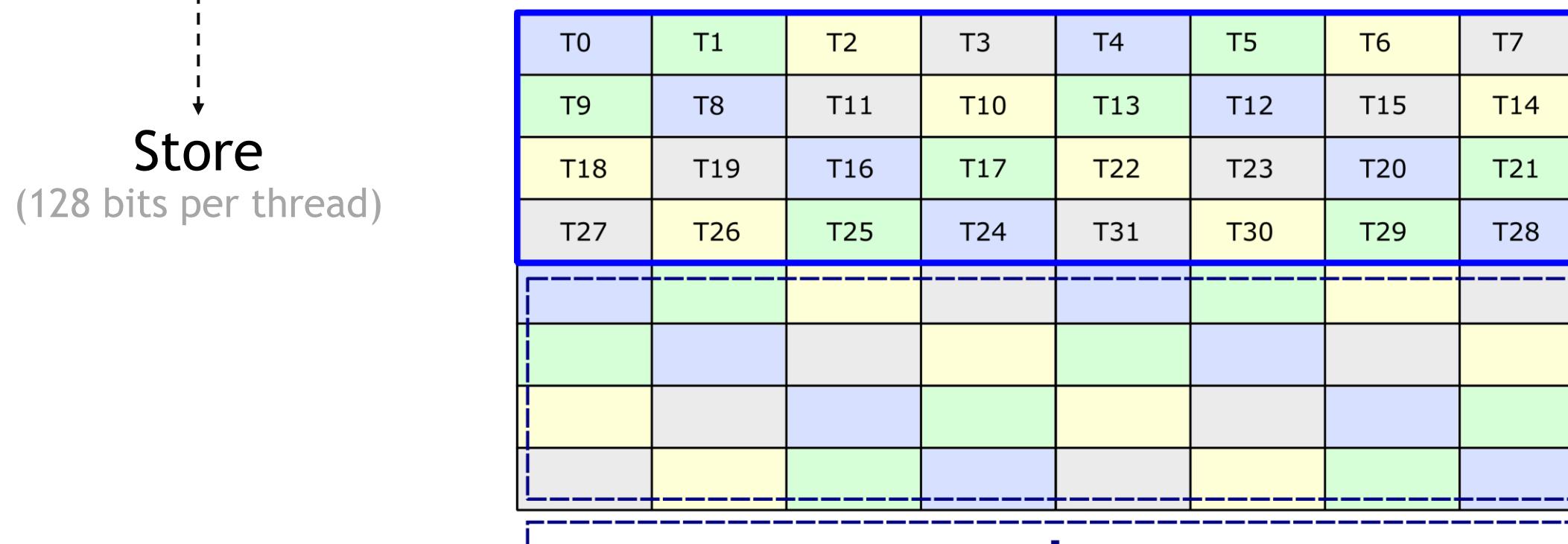
Bank conflict on either store or load from Shared Memory

GLOBAL TO SHARED MEMORY

Load from Global Memory



Store to Shared Memory

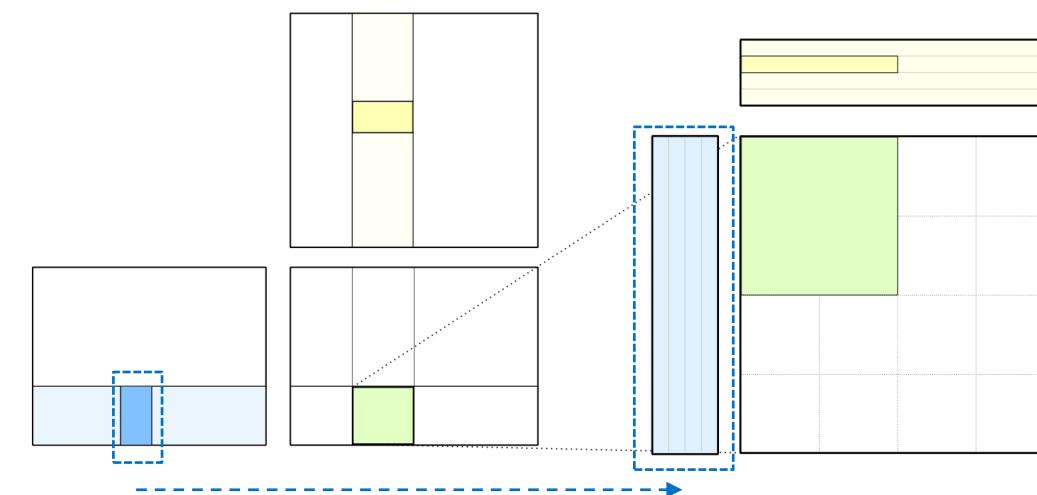
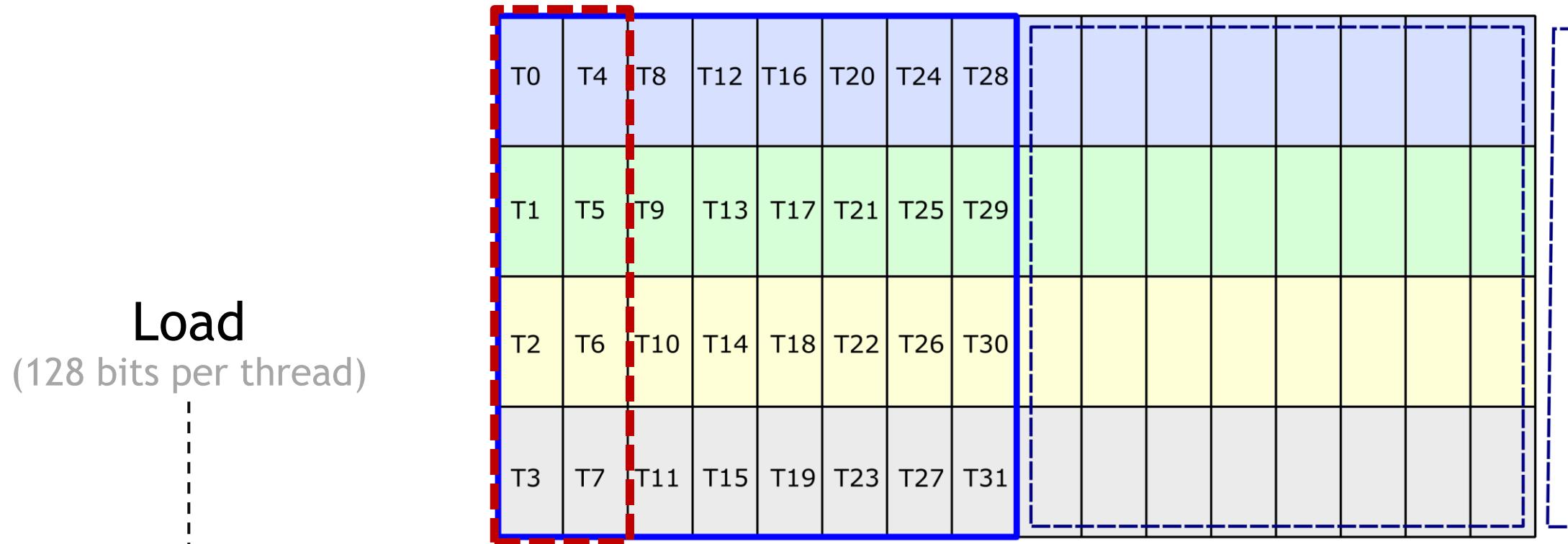


Permuted Shared Memory layout

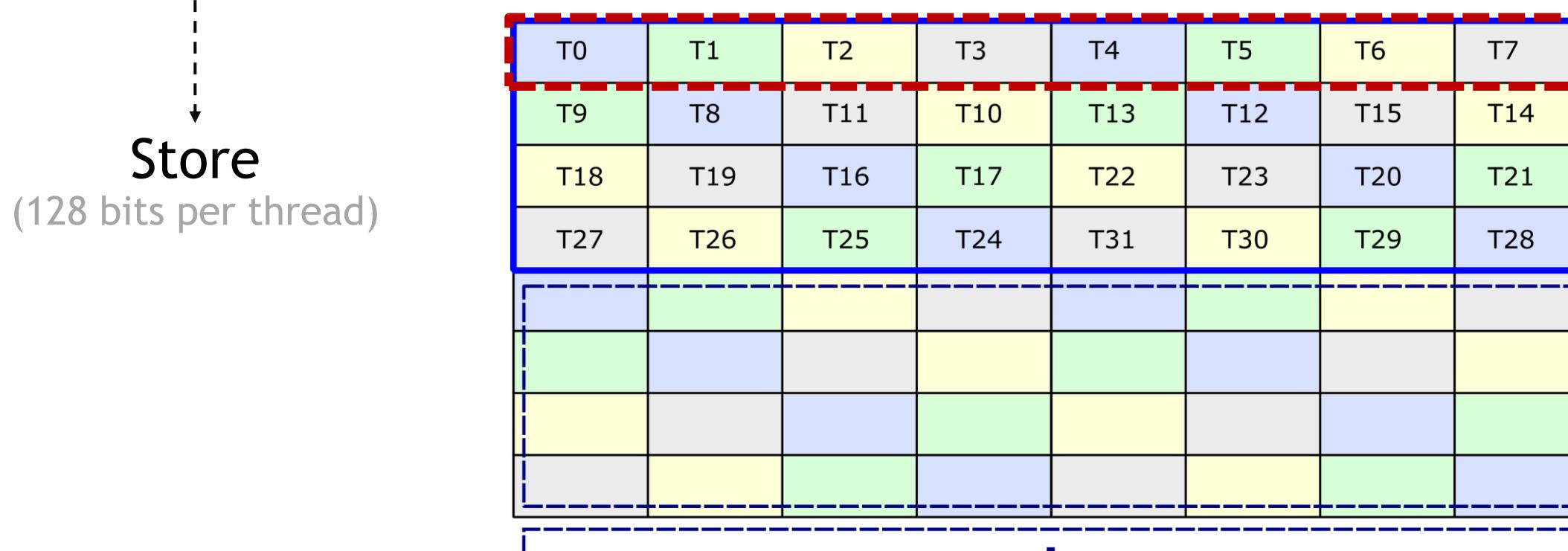
XOR function maps thread index to Shared Memory location

GLOBAL TO SHARED MEMORY

Load from Global Memory



Store to Shared Memory



Phase 0: T0 .. T7

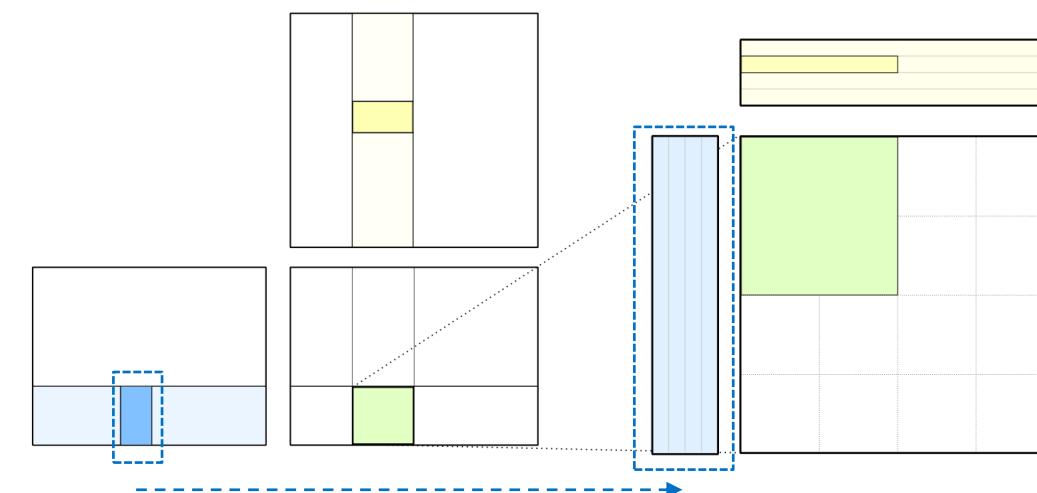
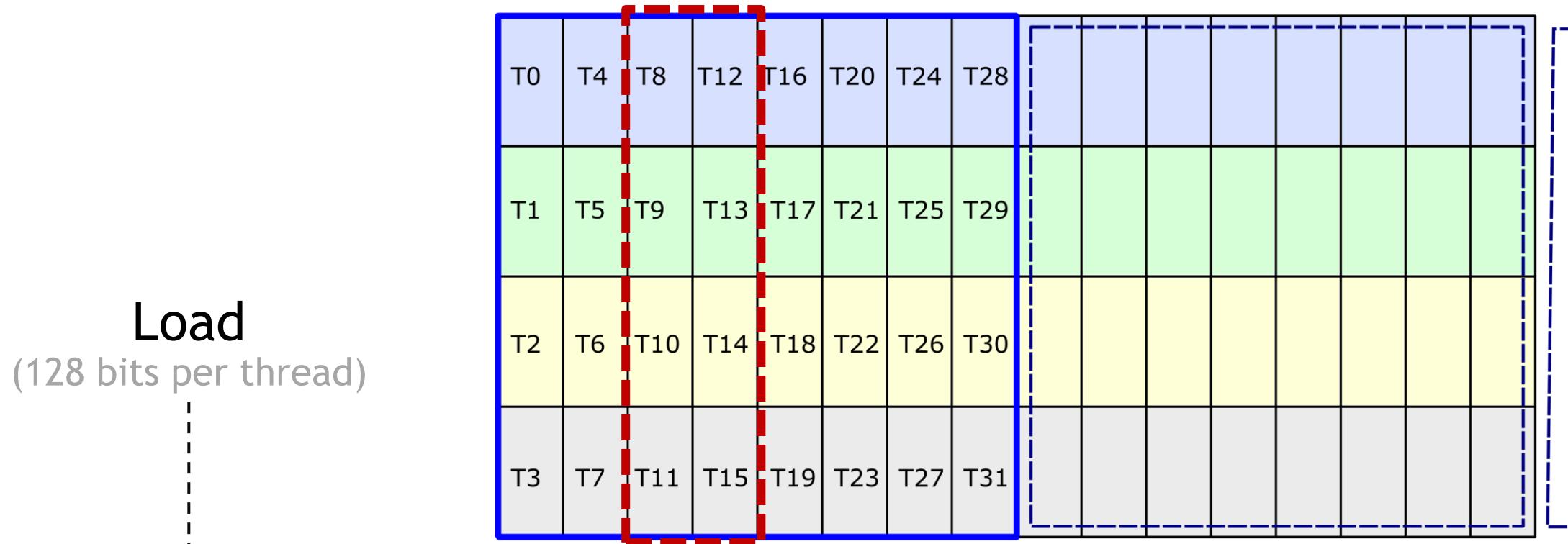
Phase 1: T8 .. T15

Phase 2: T16 .. T23

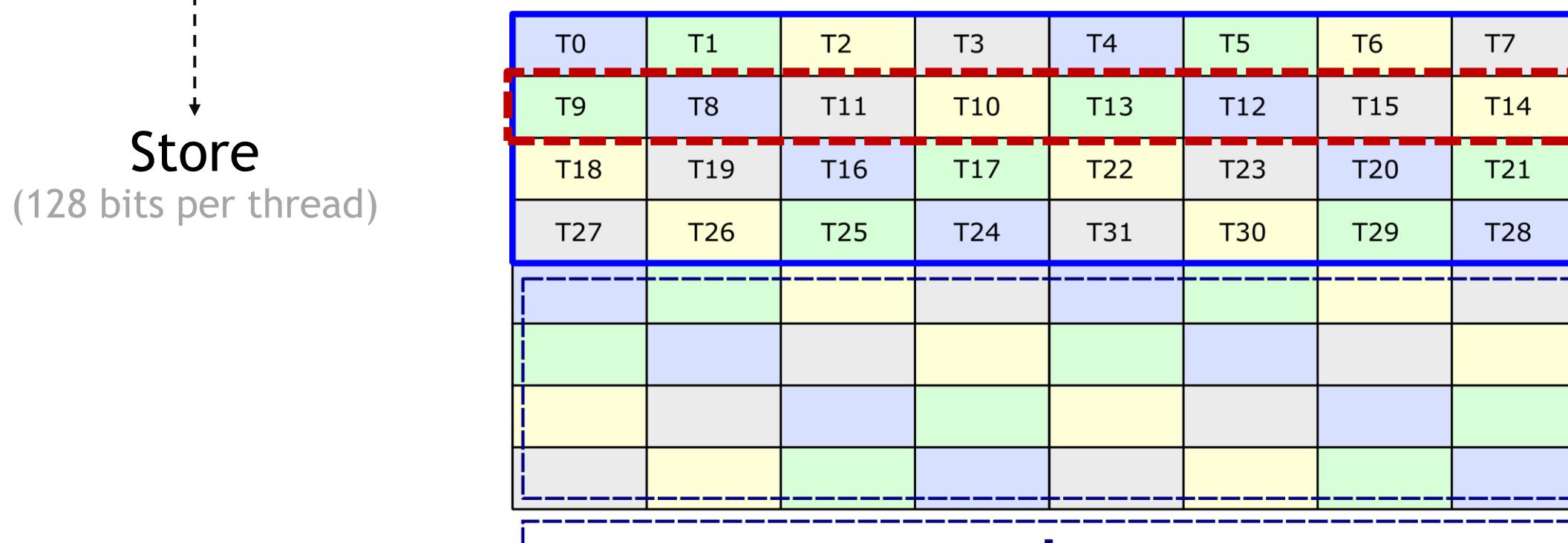
Phase 3: T24 .. T31

GLOBAL TO SHARED MEMORY

Load from Global Memory



Store to Shared Memory



Phase 0: T0 .. T7

Phase 1: T8 .. T15

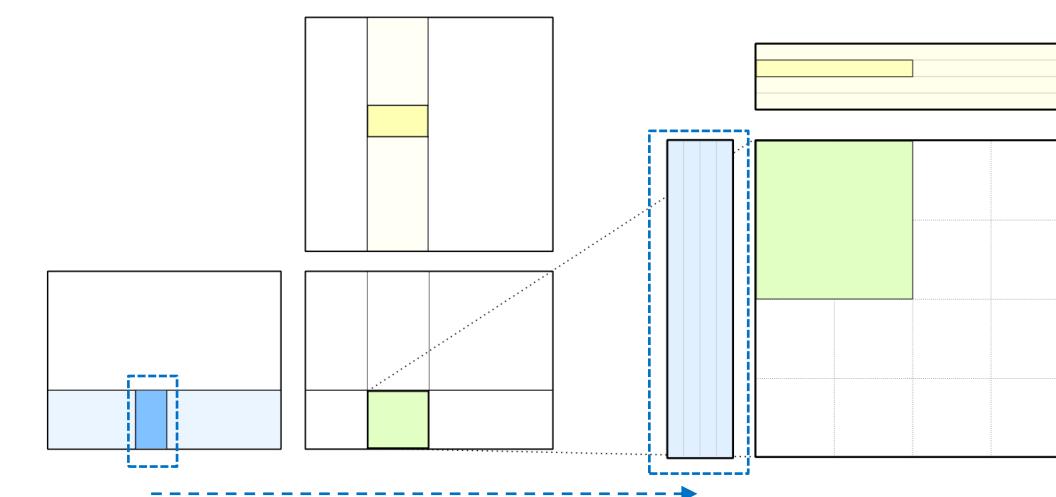
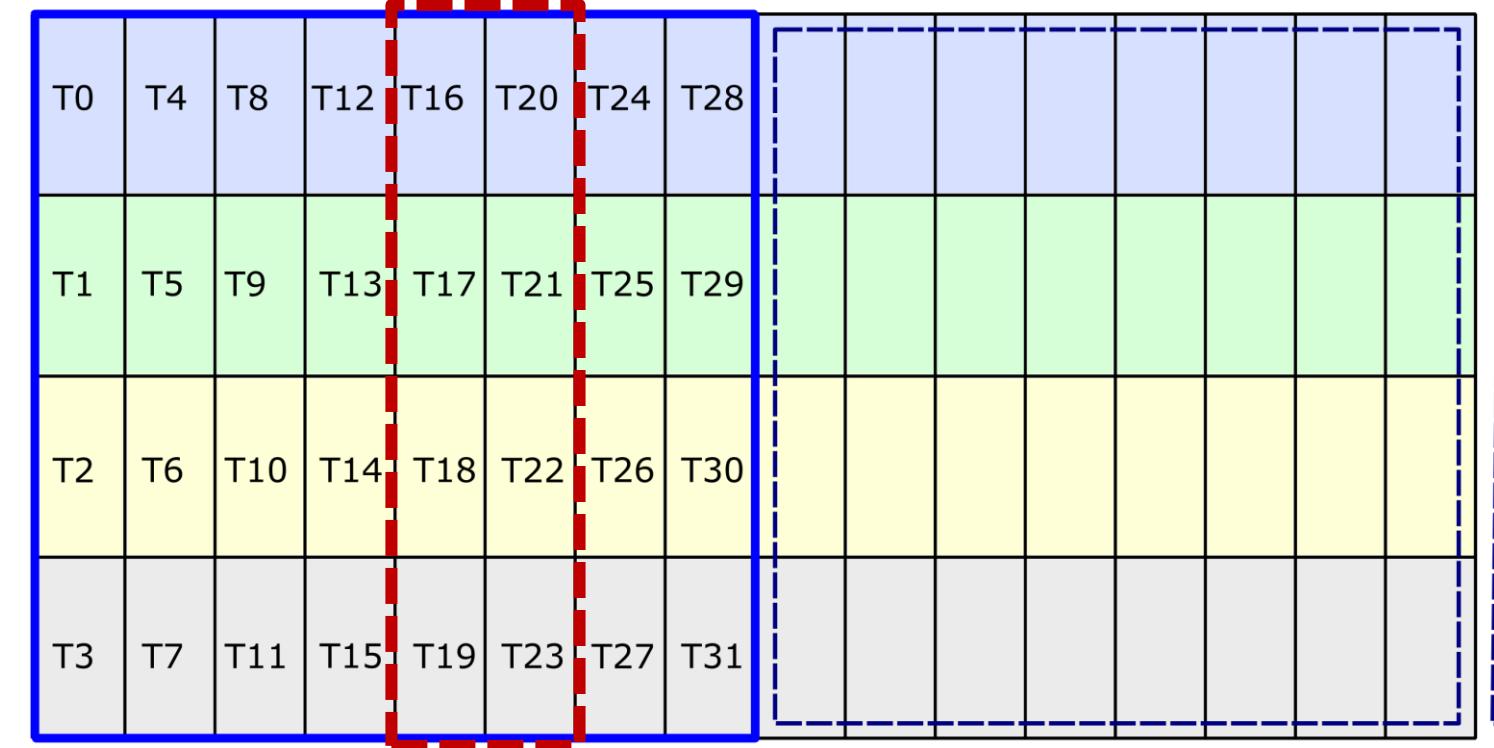
Phase 2: T16 .. T23

Phase 3: T24 .. T31

GLOBAL TO SHARED MEMORY

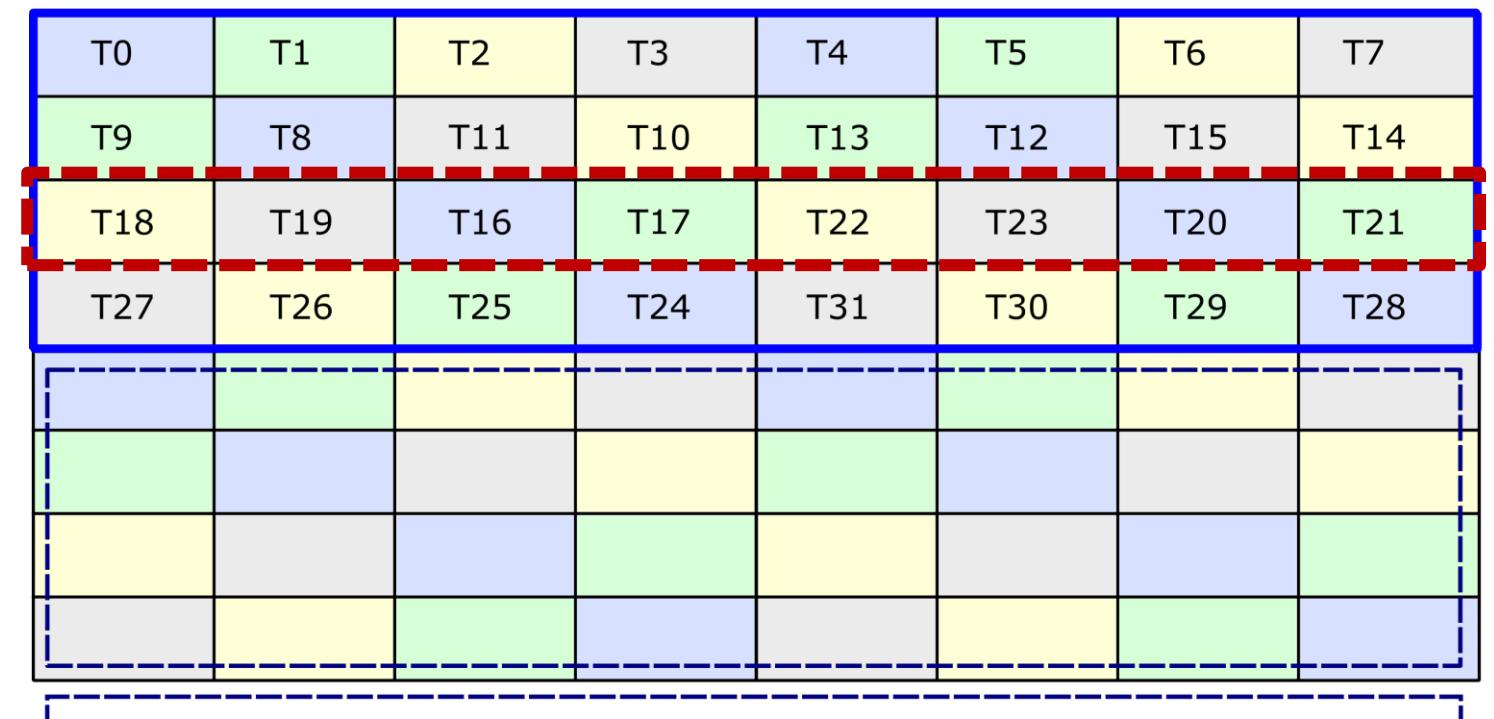
Load from Global Memory

Load
(128 bits per thread)



Store to Shared Memory

Store
(128 bits per thread)

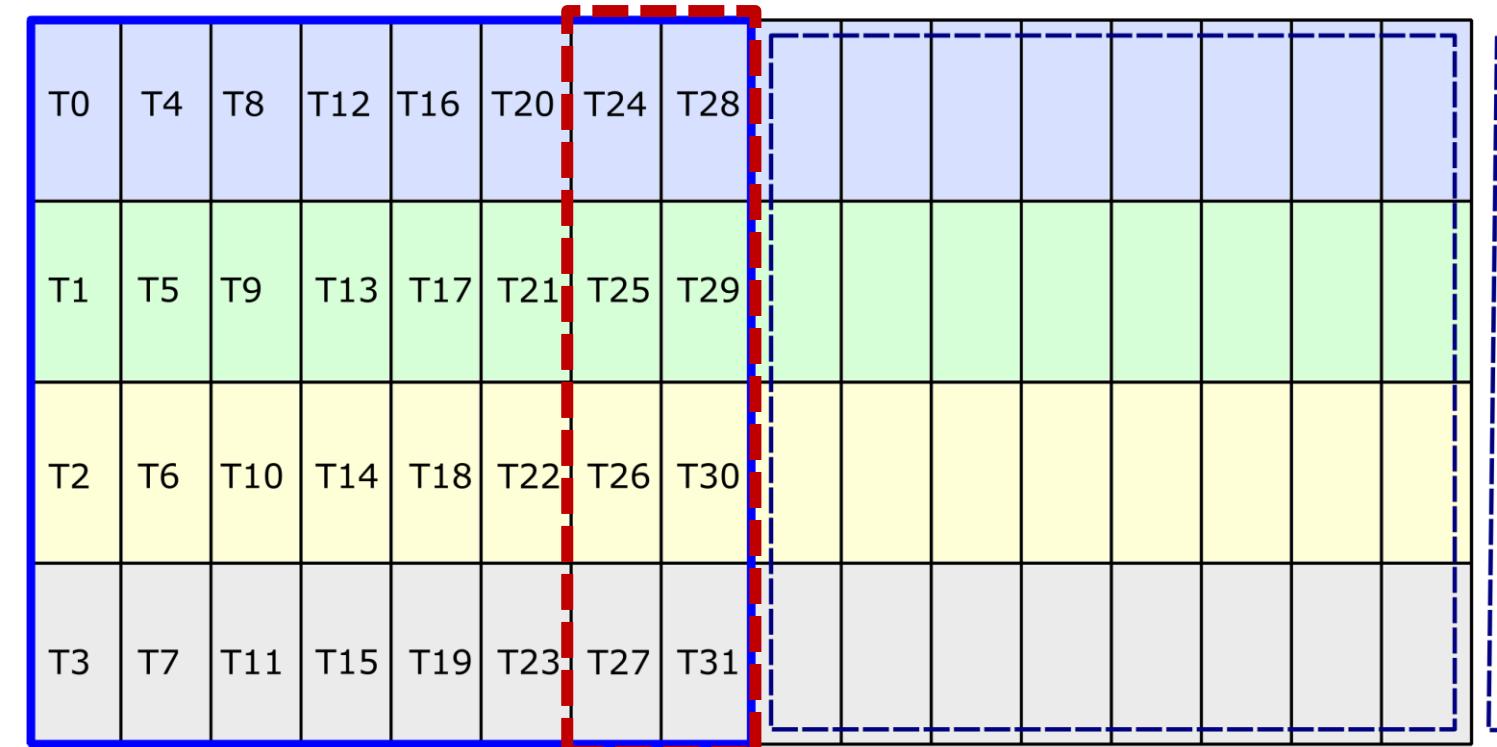


Phase 0: T0 .. T7
Phase 1: T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31

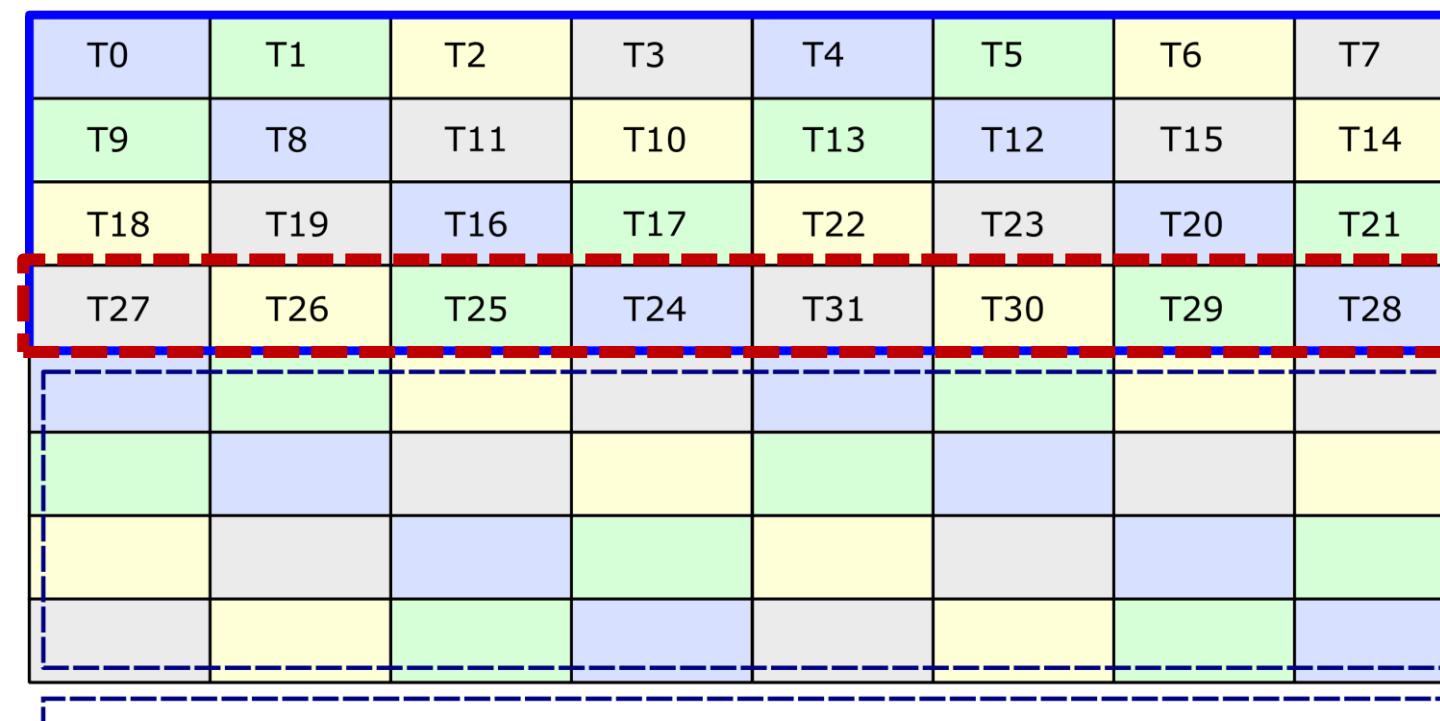
GLOBAL TO SHARED MEMORY

Load from Global Memory

Load
(128 bits per thread)



Store
(128 bits per thread)

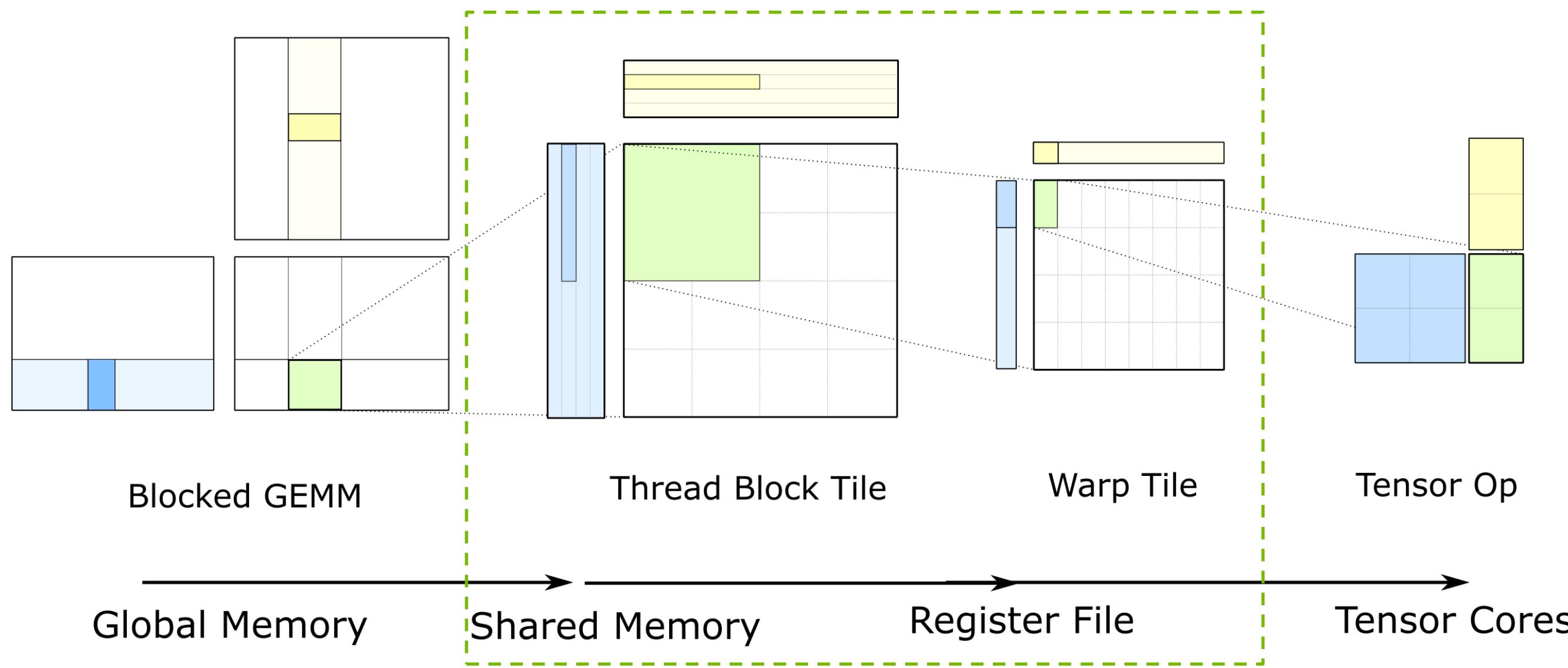


Phase 0: T0 .. T7
Phase 1: T8 .. T15
Phase 2: T16 .. T23
Phase 3: T24 .. T31

FEEDING THE DATA PATH

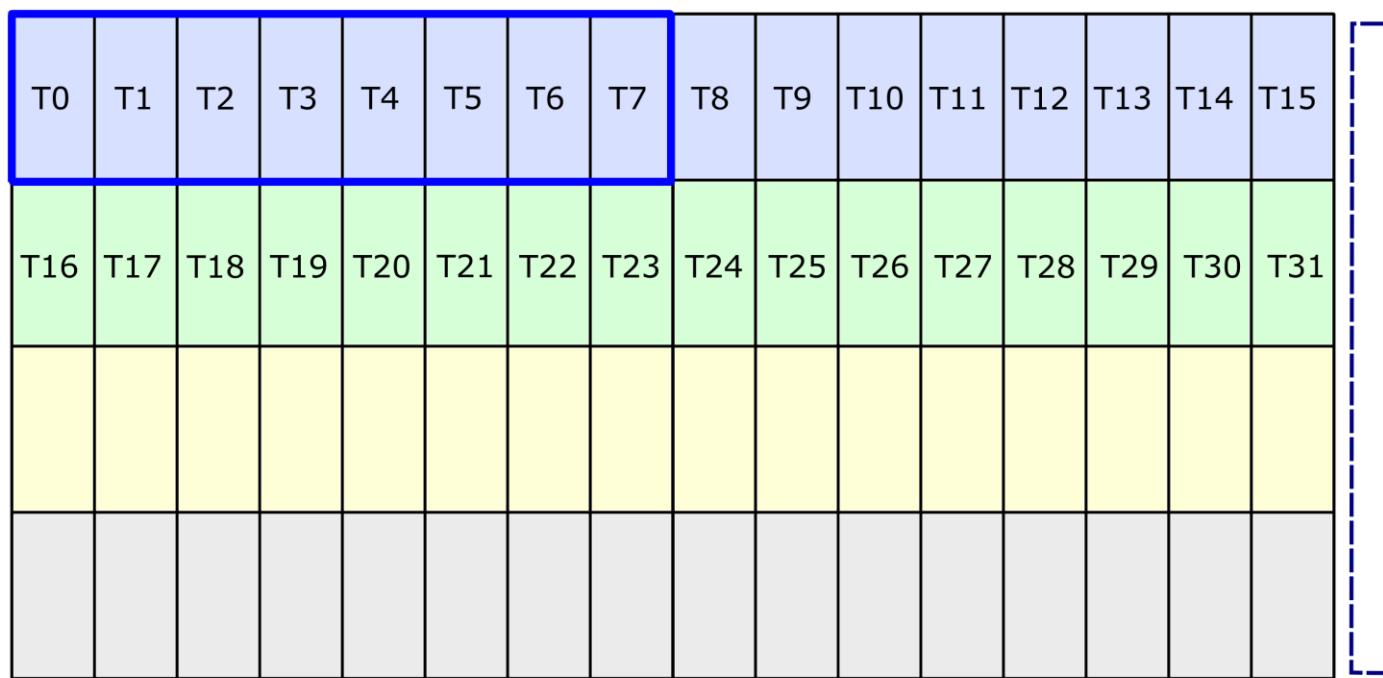
Move data from Global Memory to Tensor Cores as efficiently as possible

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

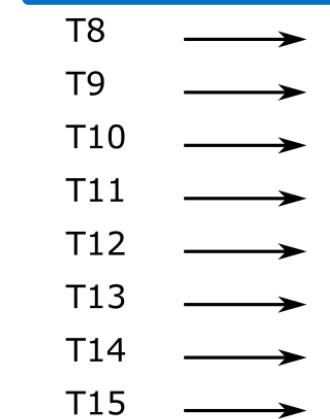
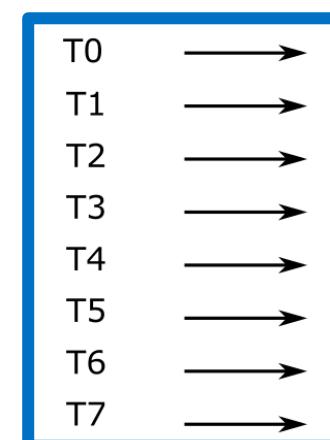


LOADING FROM SHARED MEMORY TO REGISTERS

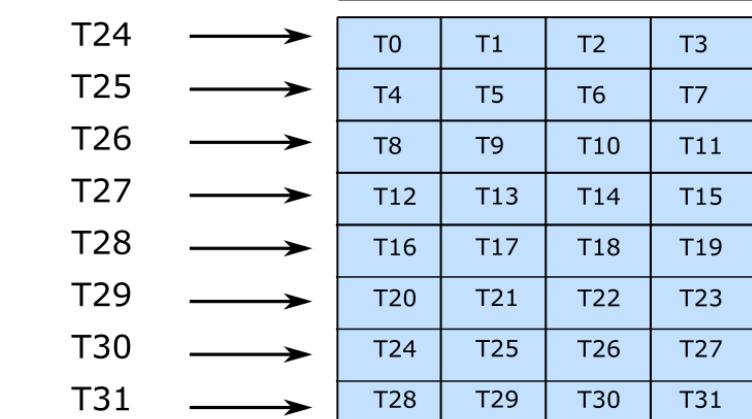
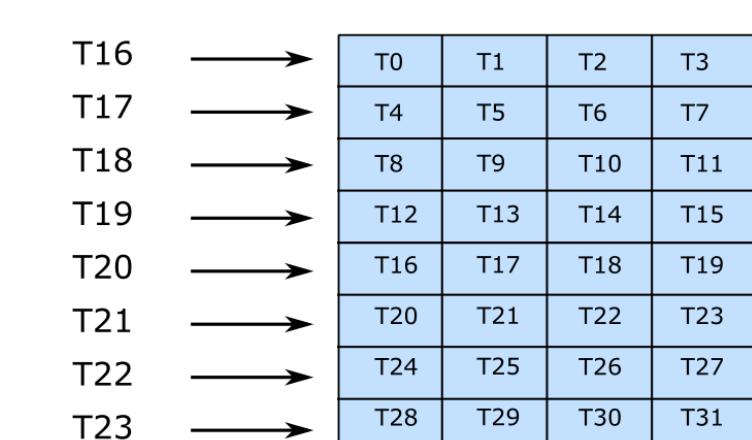
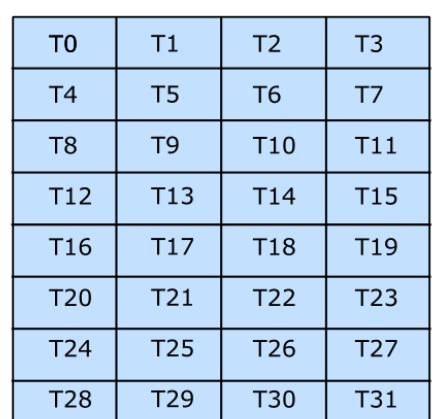
Logical view of threadblock tile



Shared Memory
Pointers



Shared Memory
Pointers



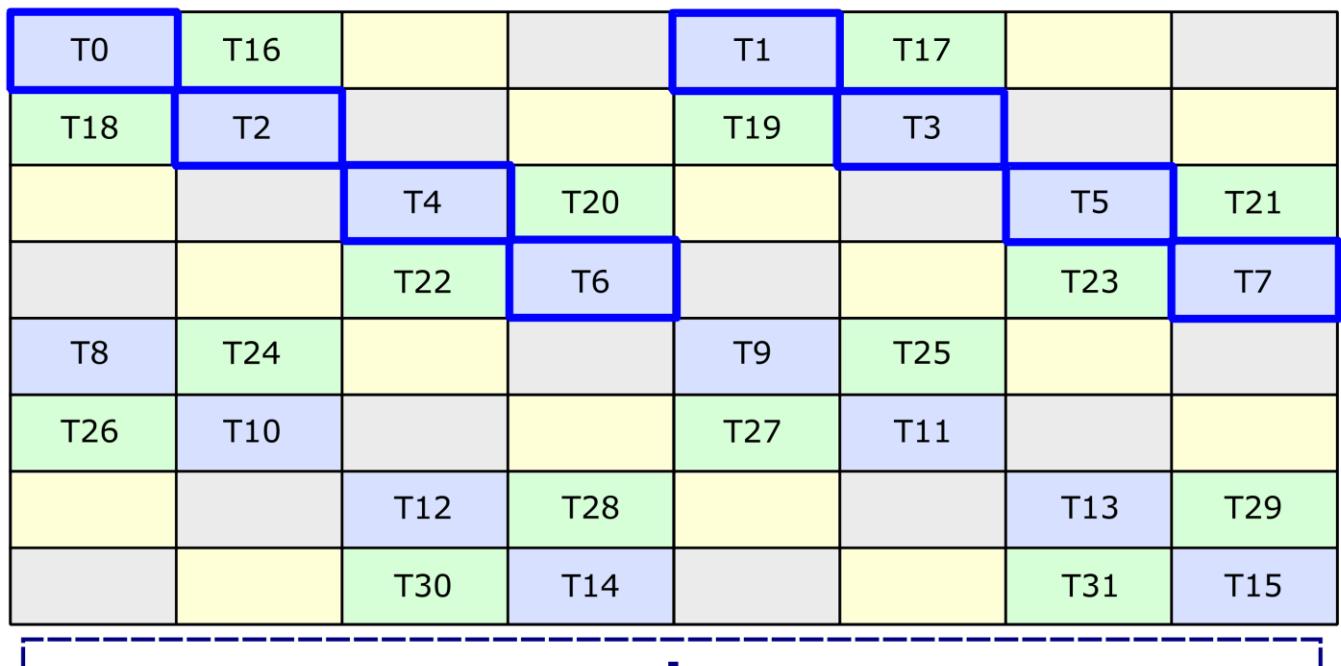
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

Load Matrix from Shared Memory



LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile

T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
T16	T17	T18	T19	T20	T21	T22	T23	T24	T25	T26	T27	T28	T29	T30	T31

T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31

Shared Memory Pointers

T0	→
T1	→
T2	→
T3	→
T4	→
T5	→
T6	→
T7	→
T8	→
T9	→
T10	→
T11	→
T12	→
T13	→
T14	→
T15	→

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31
T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

Shared Memory Pointers

T16	→
T17	→
T18	→
T19	→
T20	→
T21	→
T22	→
T23	→
T24	→
T25	→
T26	→
T27	→
T28	→
T29	→
T30	→
T31	→
T0	→
T1	→
T2	→
T3	→
T4	→
T5	→
T6	→
T7	→
T8	→
T9	→
T10	→
T11	→
T12	→
T13	→
T14	→
T15	→

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31
T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

Load Matrix from Shared Memory

T0	T16			T1	T17		
T18	T2			T19	T3		
		T4	T20			T5	T21
		T22	T6			T23	T7
T8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28			T13	T29
		T30	T14			T31	T15

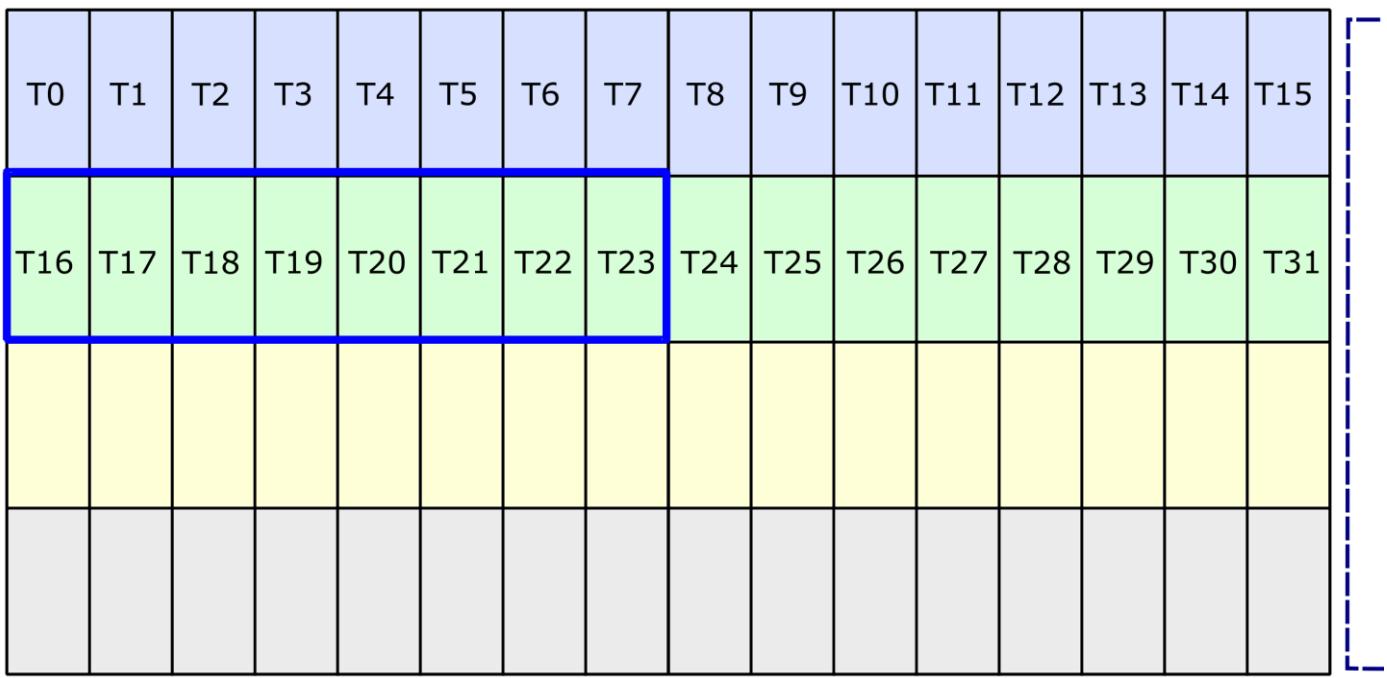
T8	→
T9	→
T10	→
T11	→
T12	→
T13	→
T14	→
T15	→

T0	T1	T2	T3
T4	T5	T6	T7
T8	T9	T10	T11
T12	T13	T14	T15
T16	T17	T18	T19
T20	T21	T22	T23
T24	T25	T26	T27
T28	T29	T30	T31

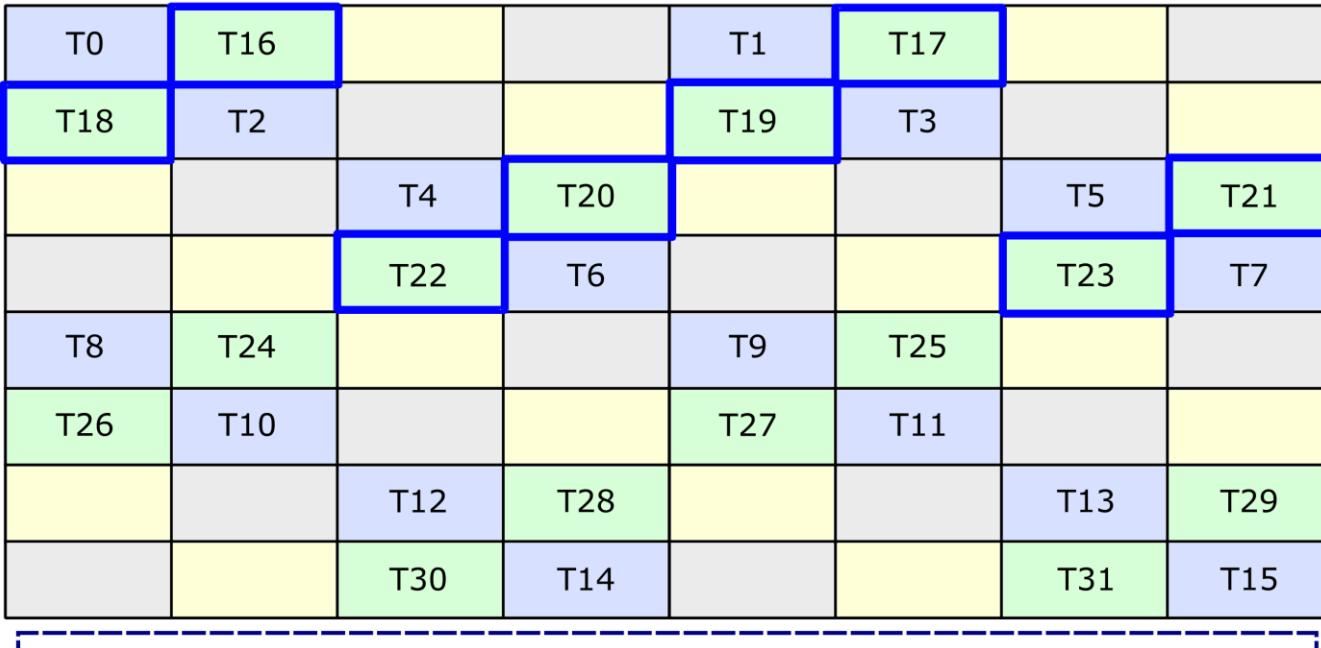
T28	→
T29	→
T30	→
T31	→
T0	→
T1	→
T2	→
T3	→
T4	→
T5	→
T6	→
T7	→
T8	→
T9	→
T10	→
T11	→
T12	→
T13	→
T14	→
T15	→
T16	→
T17	→
T18	→
T19	→
T20	→
T21	→
T22	→
T23	→
T24	→
T25	→
T26	→
T27	→

LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile



Load Matrix from Shared Memory



Shared Memory Pointers

T0 →
T1 →
T2 →
T3 →
T4 →
T5 →
T6 →
T7 →
T8 →
T9 →
T10 →
T11 →
T12 →
T13 →
T14 →
T15 →

T0 T1 T2 T3
T4 T5 T6 T7
T8 T9 T10 T11
T12 T13 T14 T15
T16 T17 T18 T19
T20 T21 T22 T23
T24 T25 T26 T27
T28 T29 T30 T31

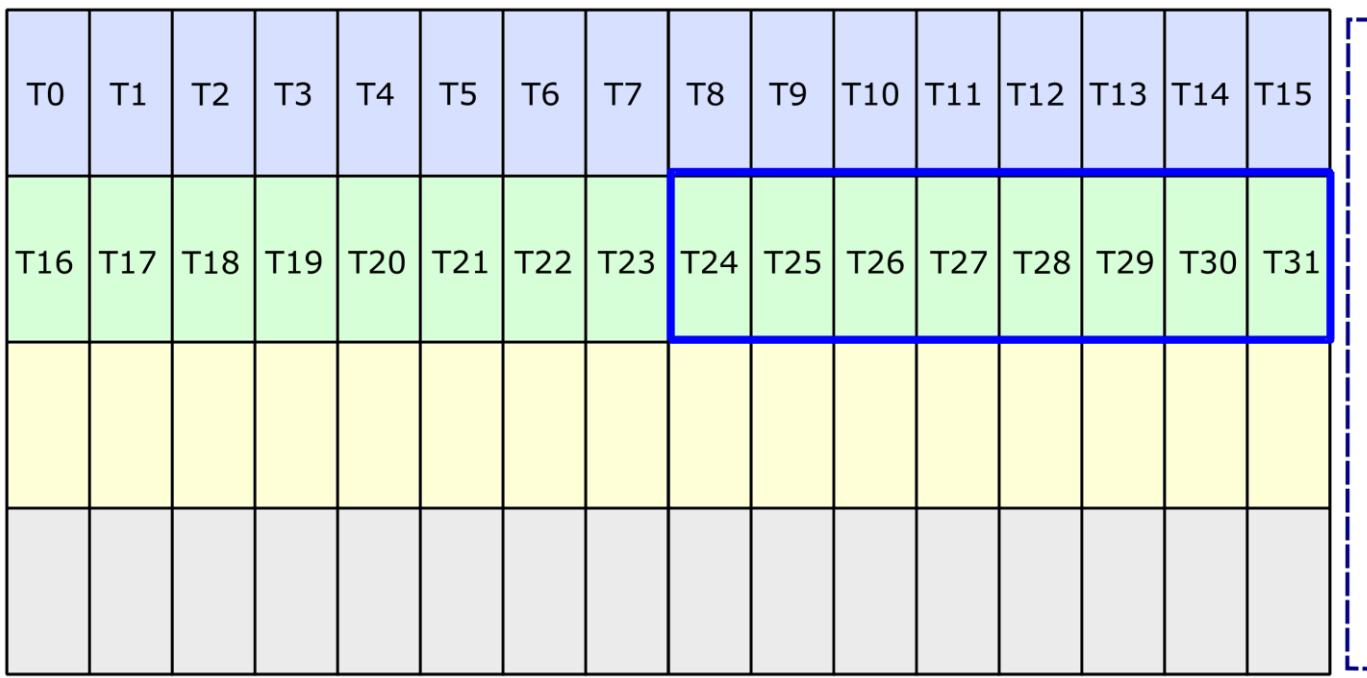
Shared Memory Pointers

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

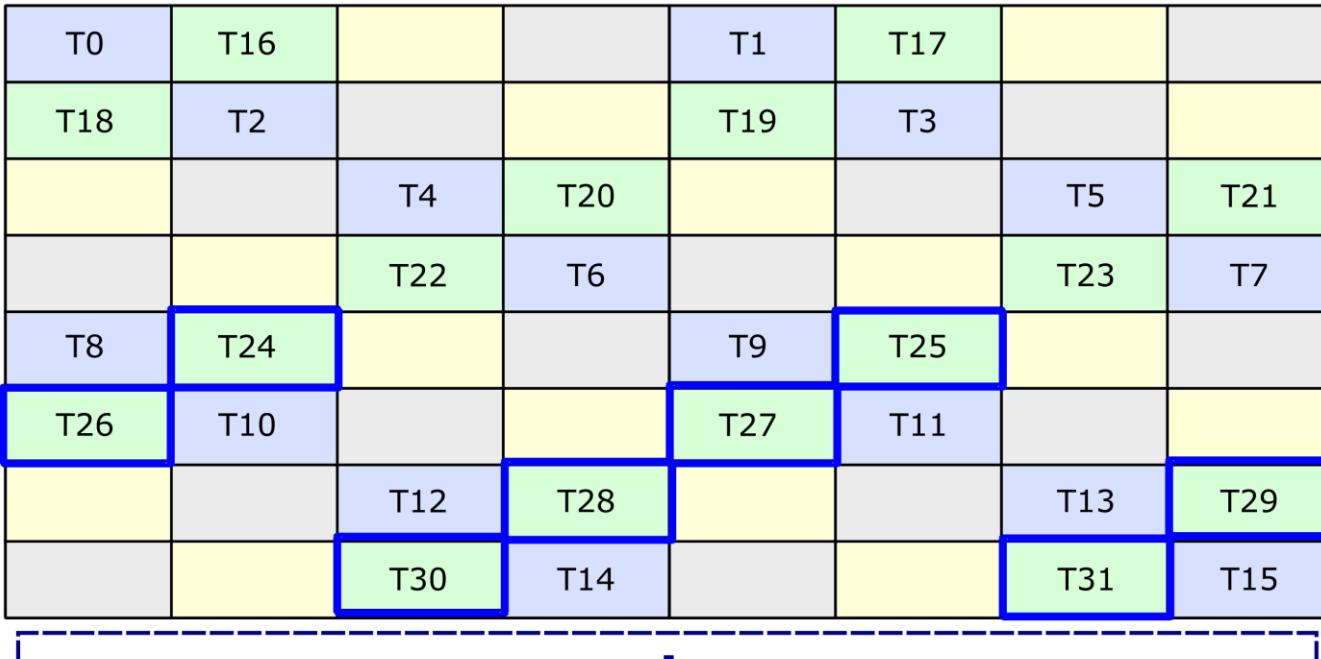
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				

LOADING FROM SHARED MEMORY TO REGISTERS

Logical view of threadblock tile



Load Matrix from Shared Memory



Shared Memory Pointers

T0 →
T1 →
T2 →
T3 →
T4 →
T5 →
T6 →
T7 →
T8 →
T9 →
T10 →
T11 →
T12 →
T13 →
T14 →
T15 →

T0 →
T1 →
T2 →
T3 →
T4 →
T5 →
T6 →
T7 →
T8 →
T9 →
T10 →
T11 →
T12 →
T13 →
T14 →
T15 →
T16 →
T17 →
T18 →
T19 →
T20 →
T21 →
T22 →
T23 →
T24 →
T25 →
T26 →
T27 →
T28 →
T29 →
T30 →
T31 →

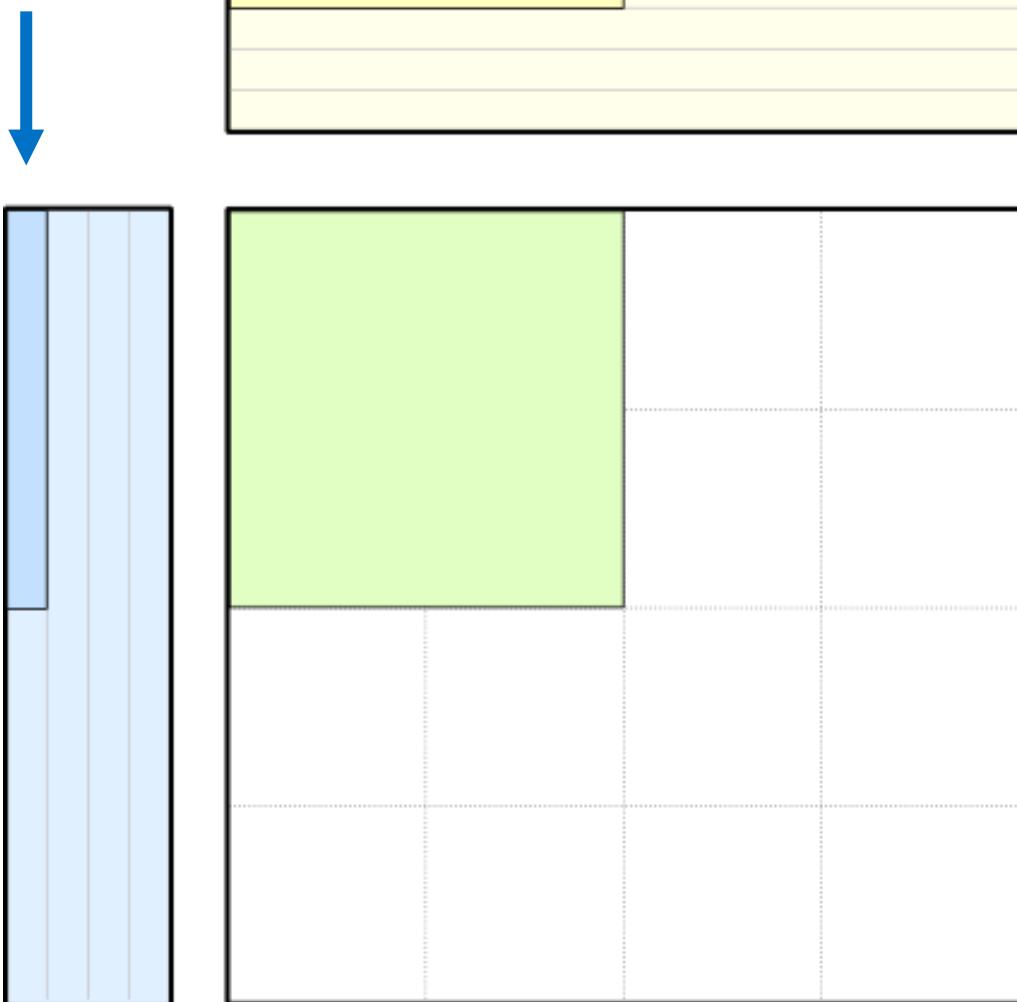
Shared Memory Pointers

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

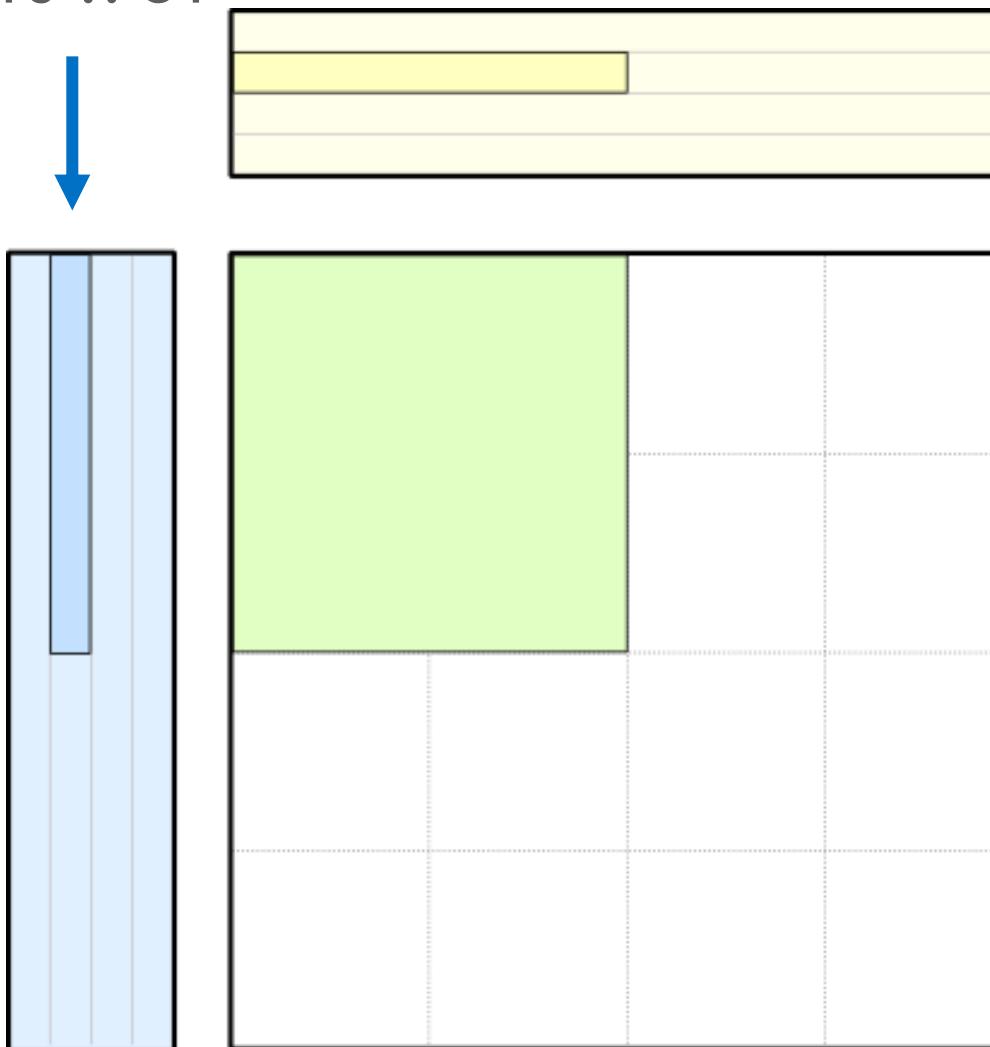
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T4	T8	T12	T16	T20	T24	T28
T1	T5	T9	T13	T17	T21	T25	T29
T2	T6	T10	T14	T18	T22	T26	T30
T3	T7	T11	T15	T19	T23	T27	T31
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				
T0	T1	T2	T3				
T4	T5	T6	T7				
T8	T9	T10	T11				
T12	T13	T14	T15				
T16	T17	T18	T19				
T20	T21	T22	T23				
T24	T25	T26	T27				
T28	T29	T30	T31				

ADVANCING TO NEXT K GROUP

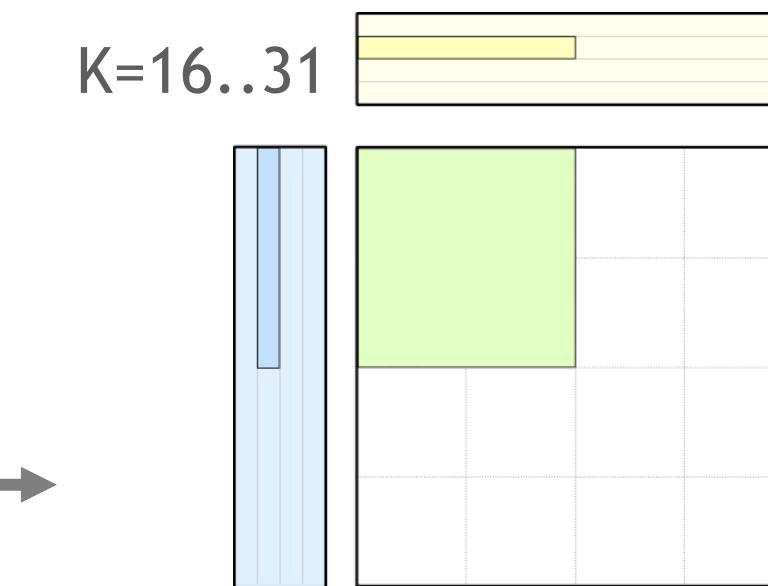
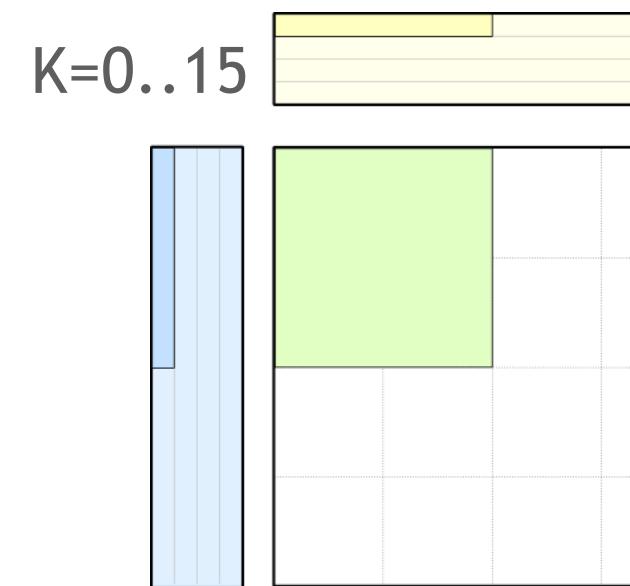
K=0 .. 15



K=16 .. 31



ADVANCING TO NEXT K GROUP



T0	T16			T1	T17		
T18	T2			T19	T3		
		T4	T20		T5	T21	
		T22	T6		T23	T7	
T8	T24			T9	T25		
T26	T10			T27	T11		
		T12	T28		T13	T29	
		T30	T14		T31	T15	

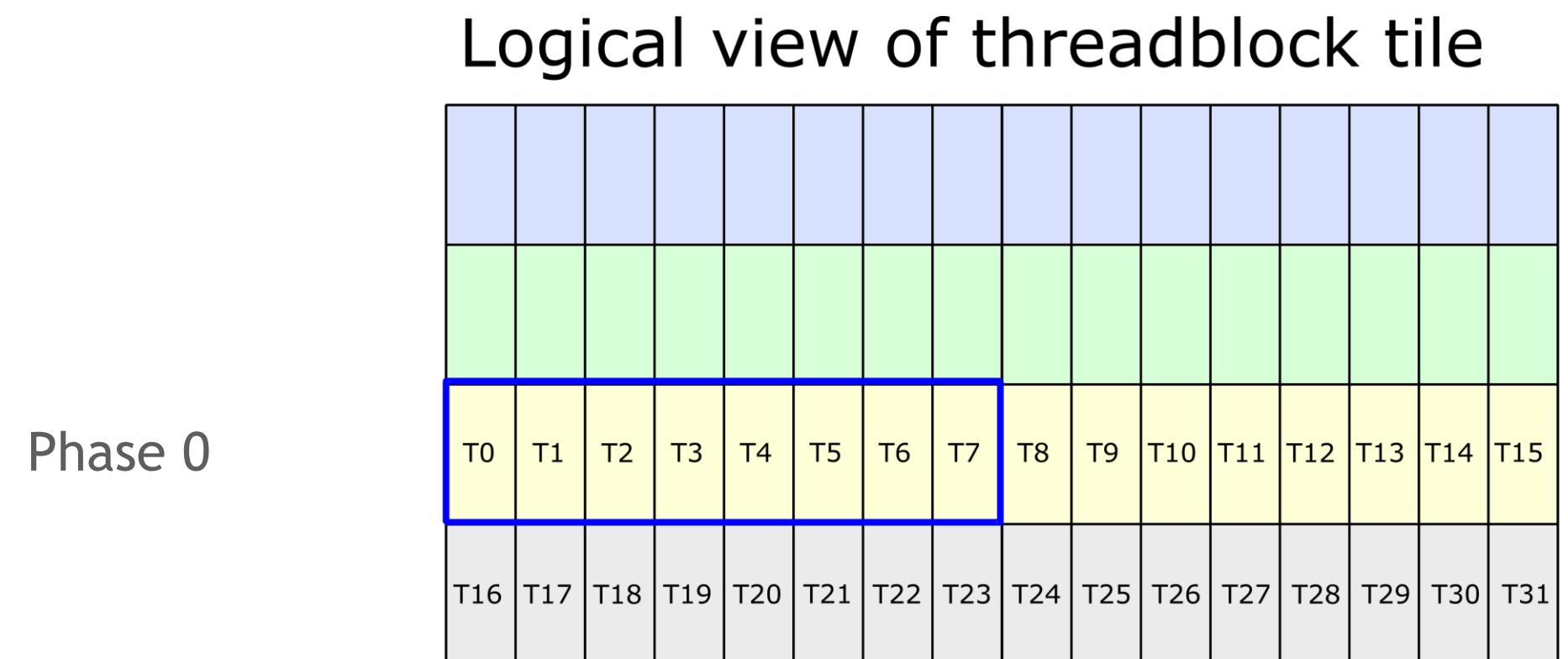
		T0	T16			T1	T17
		T18	T2			T19	T3
		T4	T20			T5	T21
		T22	T6			T23	T7
		T8	T24			T9	T25
		T26	T10			T27	T11
		T12	T28			T13	T29
		T30	T14			T31	T15



smem_ptr = row_idx * 8 + column_idx;

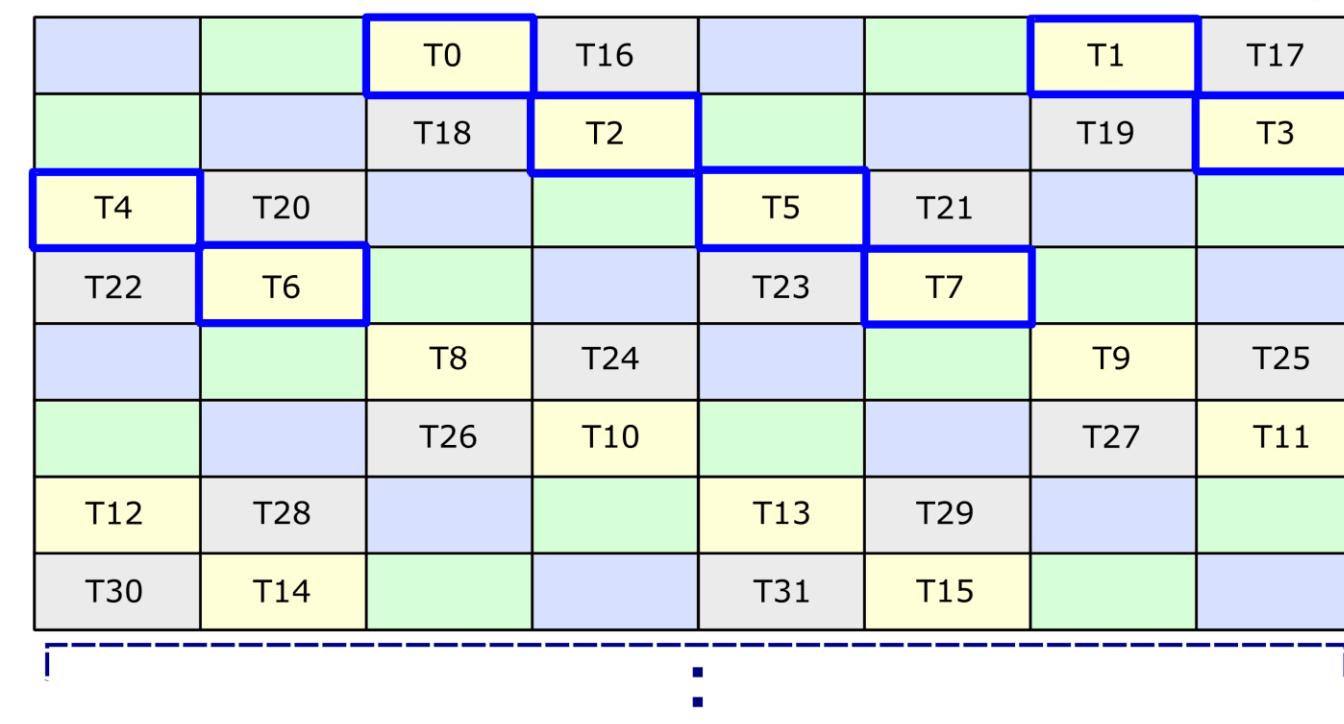
smem_ptr = smem_ptr ^ 2;

LOADING FROM SHARED MEMORY TO REGISTERS

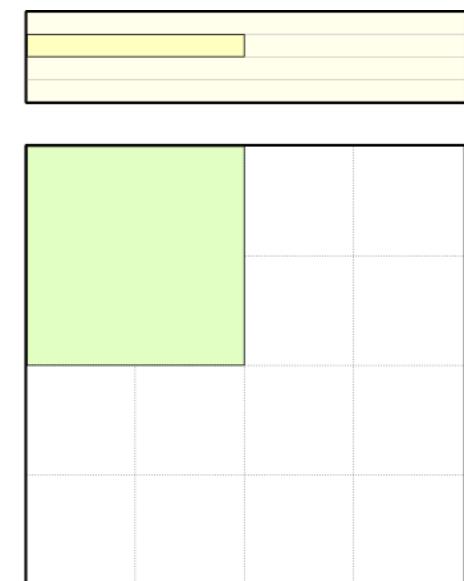


K=16..31

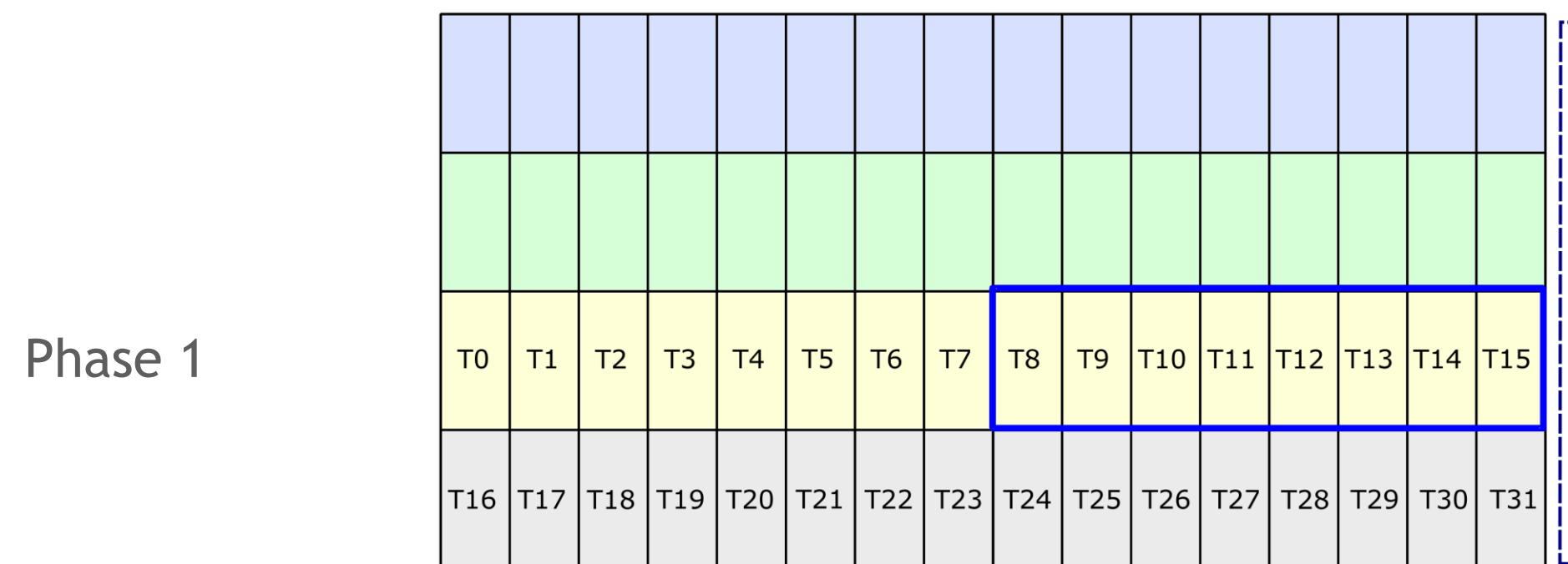
Load Matrix from Shared Memory



LOADING FROM SHARED MEMORY TO REGISTERS

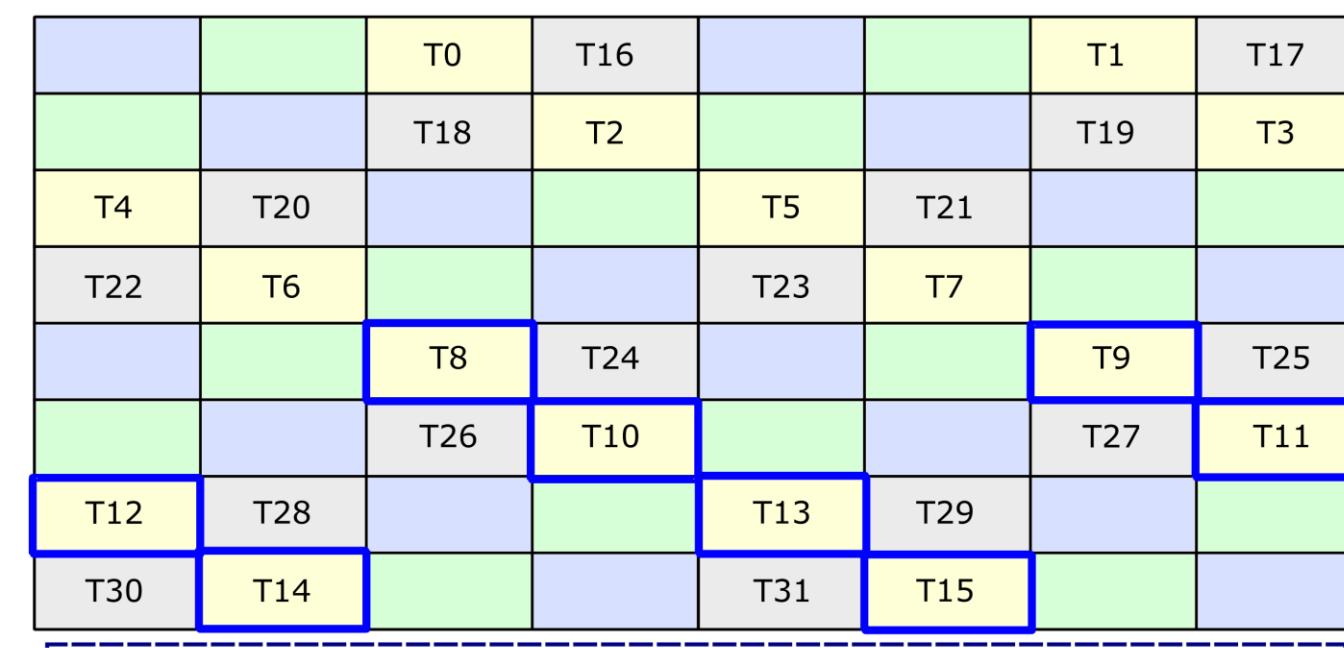


Logical view of threadblock tile

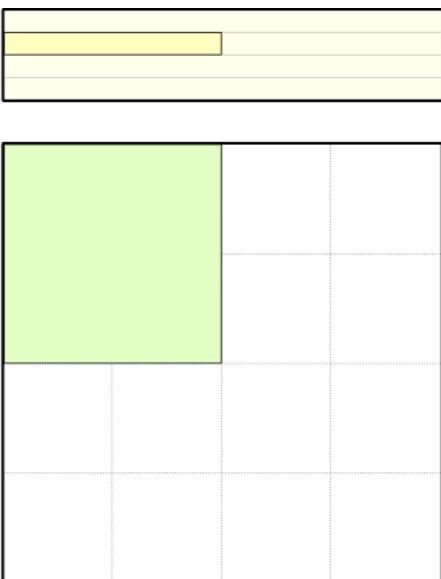


K=16..31

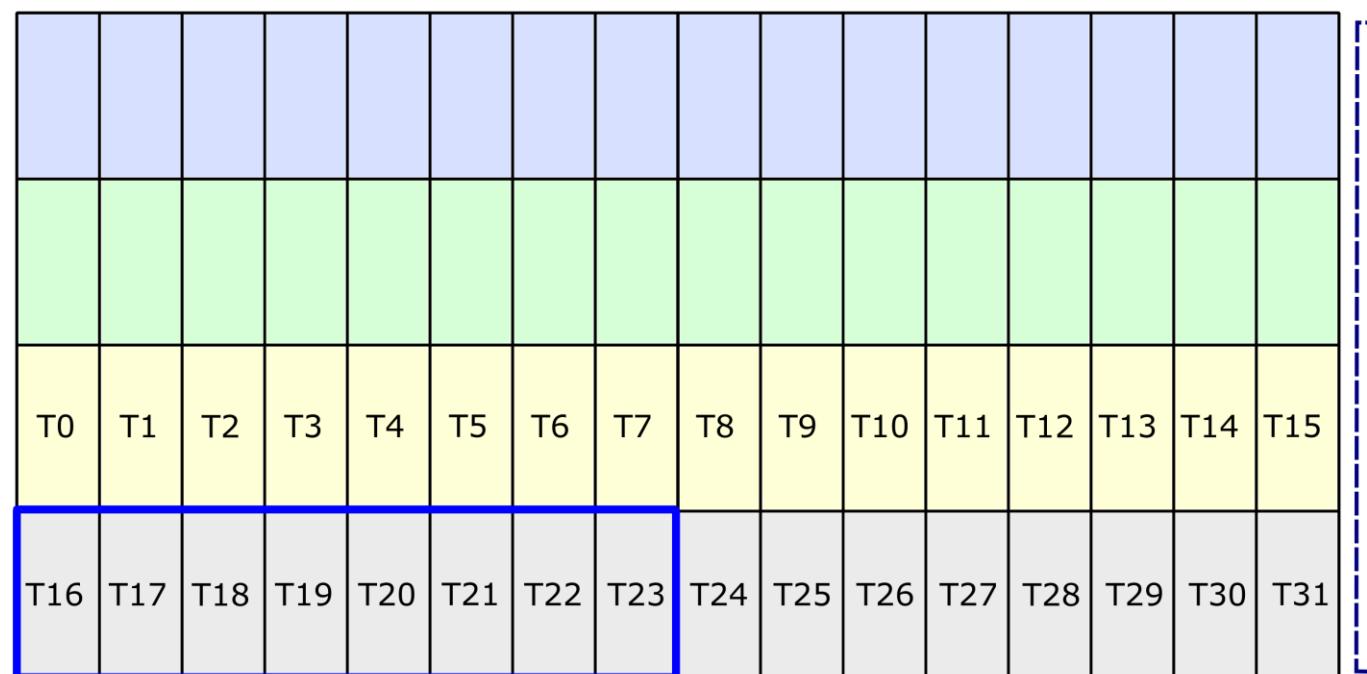
Load Matrix from Shared Memory



LOADING FROM SHARED MEMORY TO REGISTERS

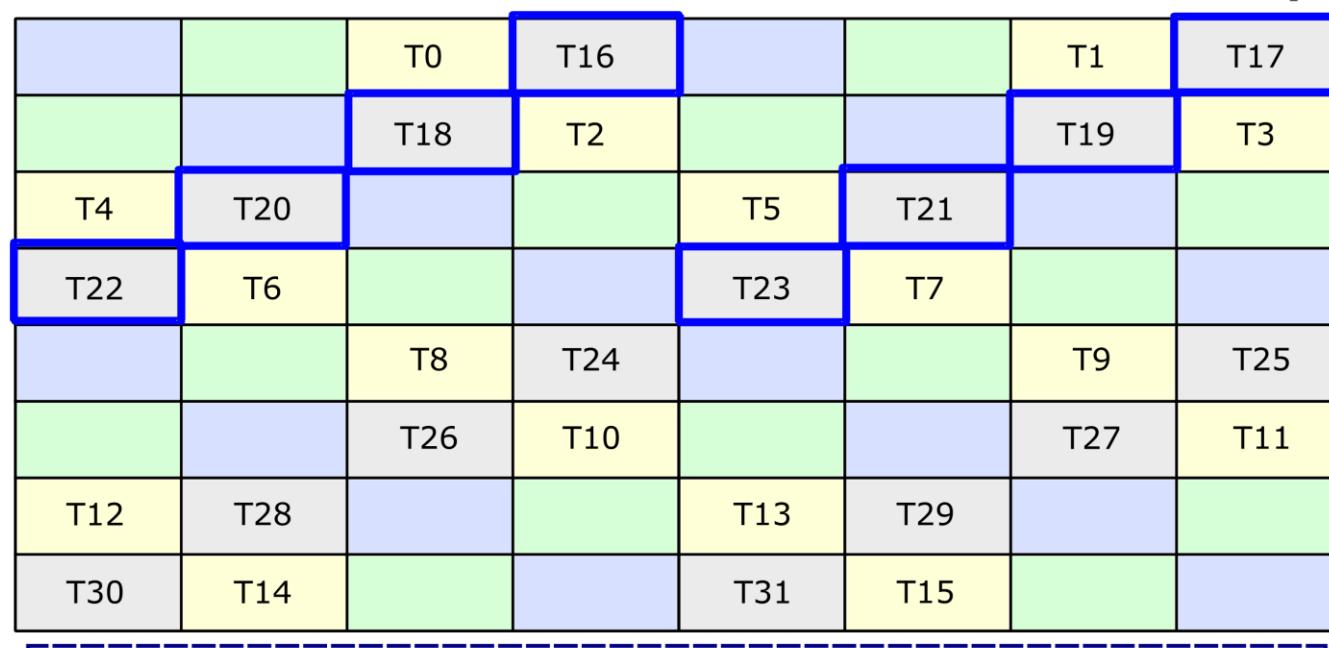


Logical view of threadblock tile

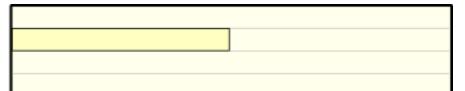


Phase 2

Load Matrix from Shared Memory

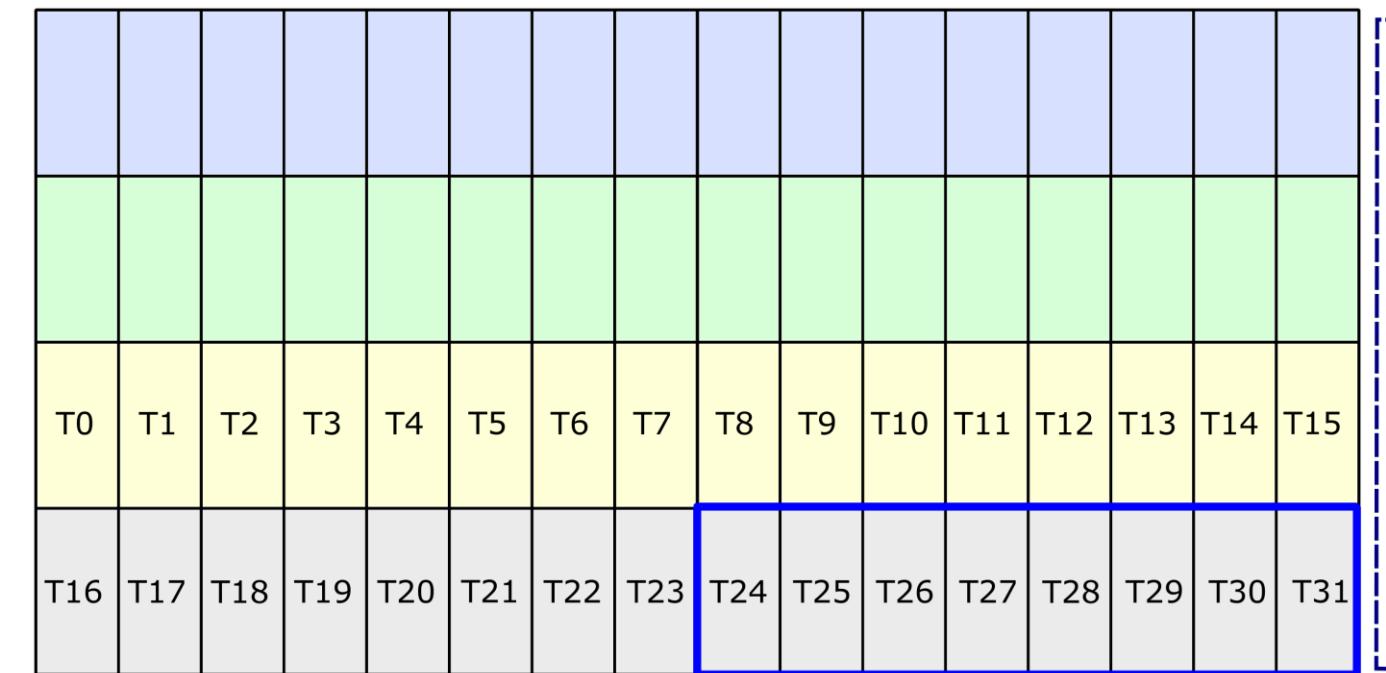


LOADING FROM SHARED MEMORY TO REGISTERS

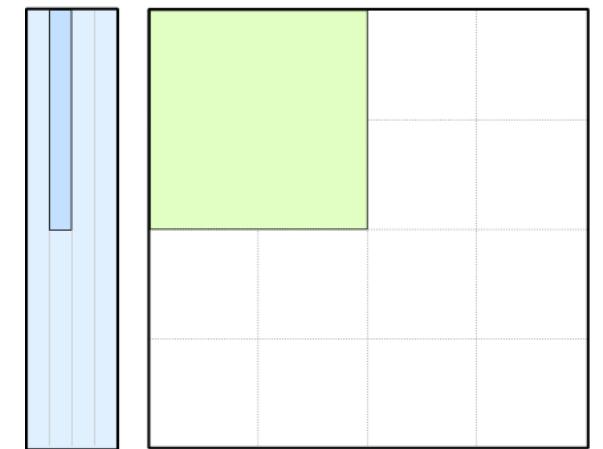


Logical view of threadblock tile

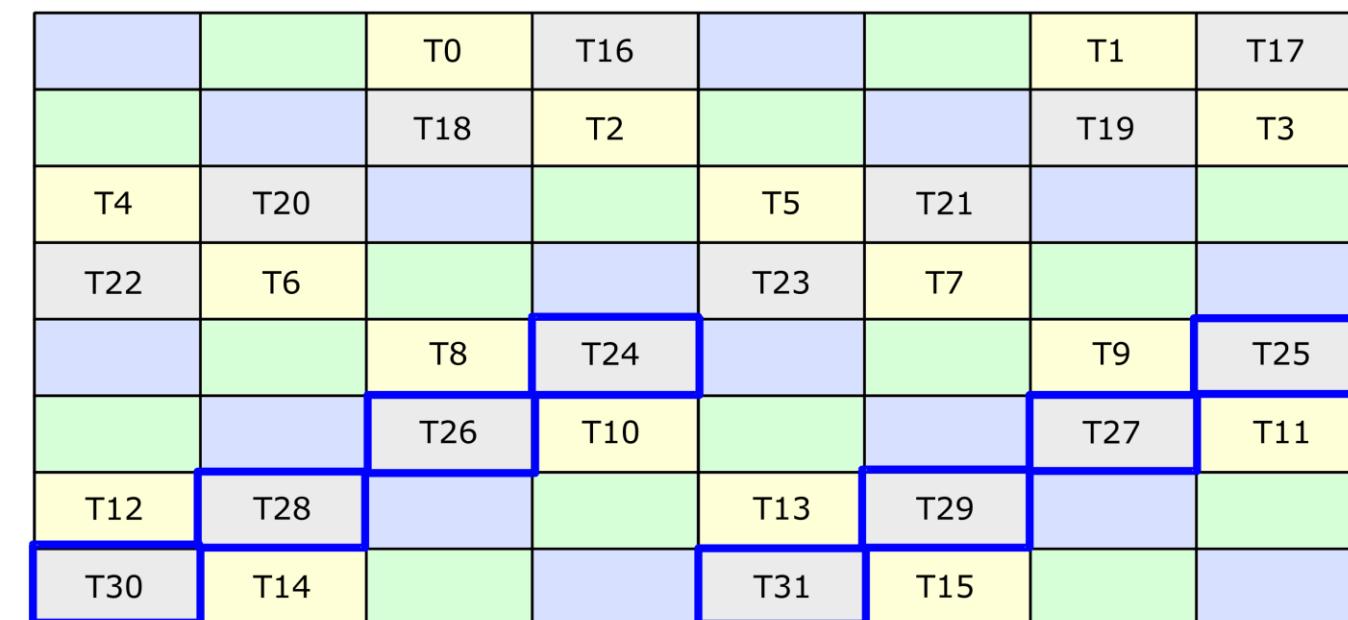
Phase 3



K=16..31



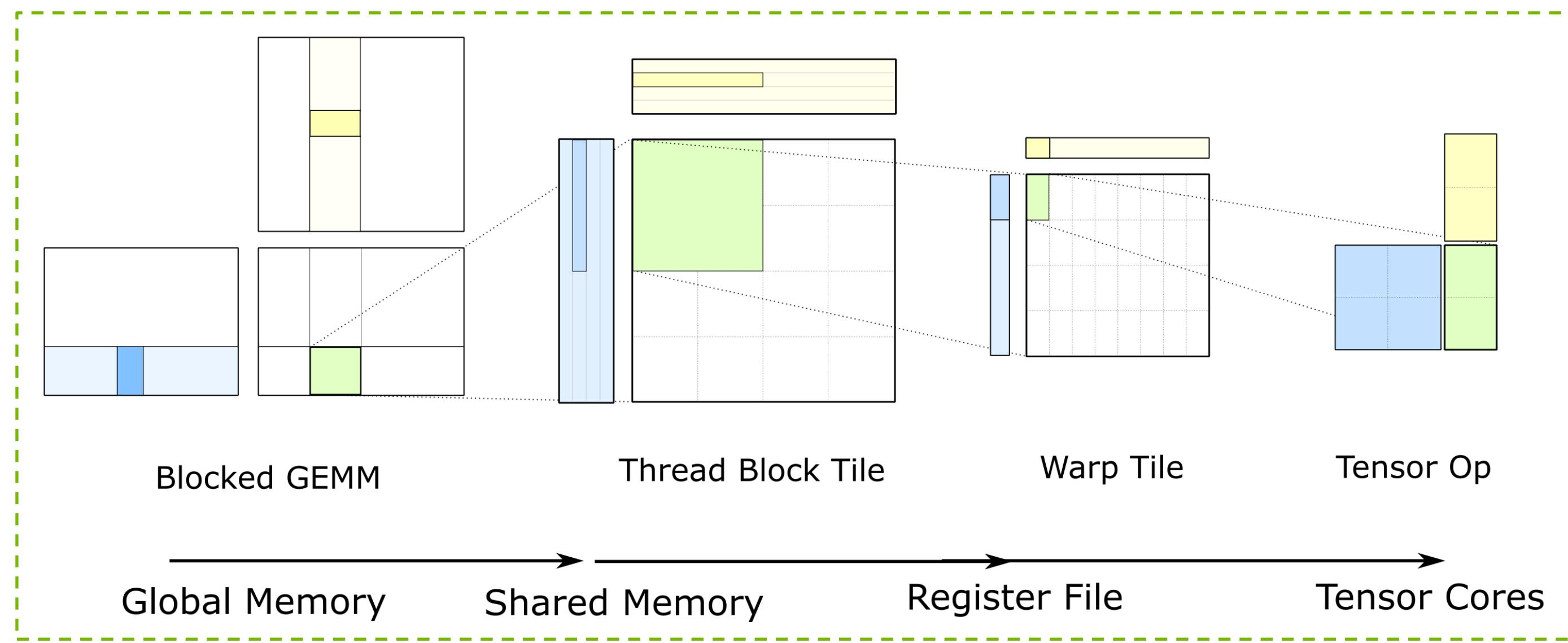
Load Matrix from Shared Memory



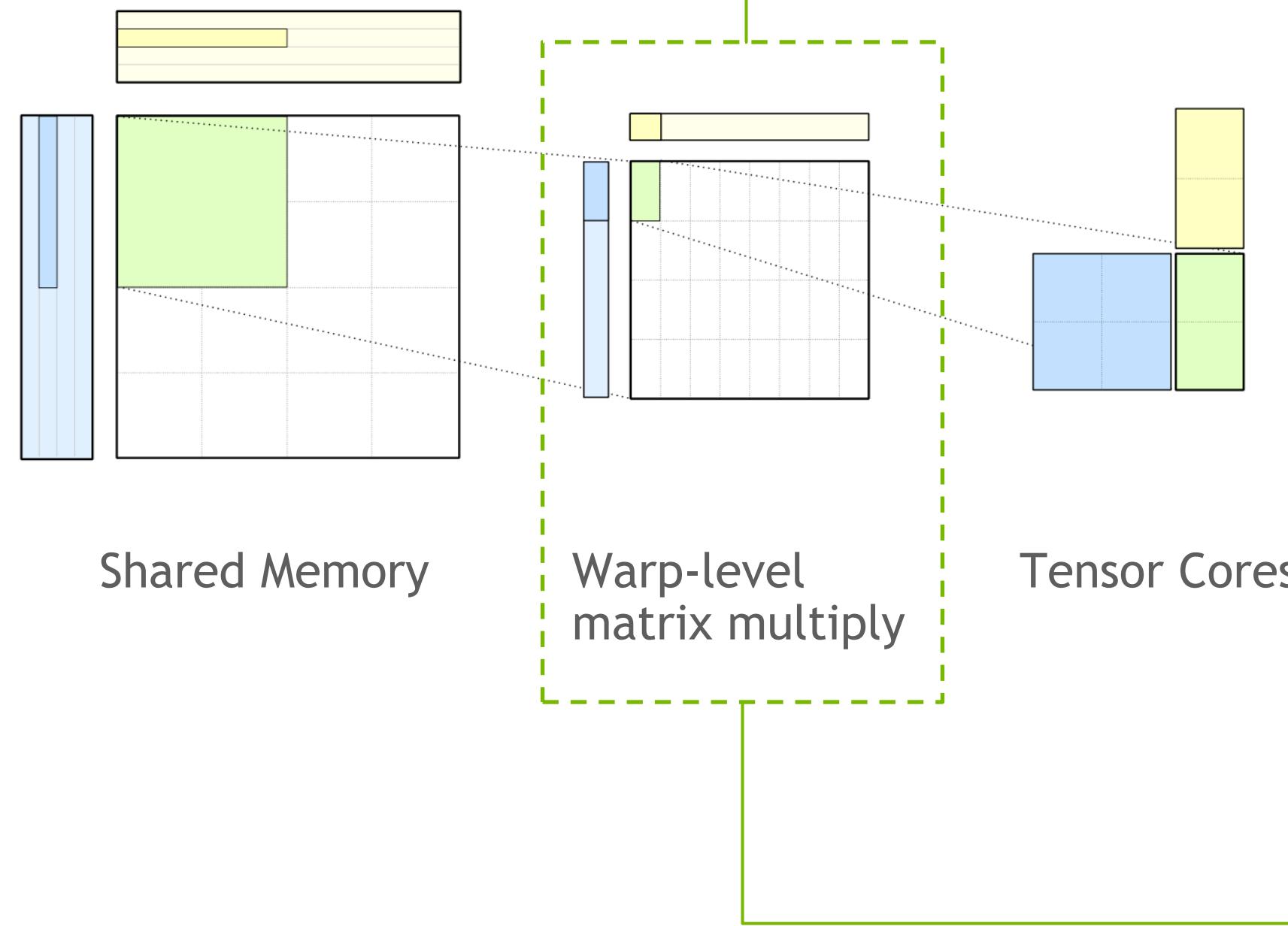
CUTLASS

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

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



CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES



```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
    GemmShape<64, 64, 16>,
    half_t, LayoutA,
    half_t, LayoutB,
    float, RowMajor
>;
```

```
__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];
```

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

```
Mma::FragmentA frag_A;
Mma::FragmentB frag_B;
Mma::FragmentC accum;
```

```
Mma mma;
```

```
accum.clear();
```

```
#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {
```

```
    iter_A.load(frag_A); // Load fragments from A and B matrices
    iter_B.load(frag_B);
```

```
    ++iter_A; ++iter_B; // Advance along GEMM K to next tile in A
                         // and B matrices
```

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

```
}
```

CUTLASS: OPTIMAL ABSTRACTION FOR TENSOR CORES

Tile Iterator Constructors:

Initialize pointers into permuted Shared Memory buffers

Fragments:

Register-backed arrays holding each thread's data

Tile Iterator:

load() - Fetches data from permuted Shared Memory buffers

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

Warp-level matrix multiply:

Decomposes a large matrix multiply into Tensor Core operations

```
using Mma = cutlass::gemm::warp::DefaultMmaTensorOp<
    GemmShape<64, 64, 16>, // GEMM A operand
    half_t, LayoutA, // GEMM B operand
    half_t, LayoutB, // GEMM C operand
    float, RowMajor>;
__shared__ ElementA smem_buffer_A[Mma::Shape::kM * GemmK];
__shared__ ElementB smem_buffer_B[Mma::Shape::kN * GemmK];
// Construct iterators into SMEM tiles
Mma::IteratorA iter_A({smem_buffer_A, lda}, thread_id);
Mma::IteratorB iter_B({smem_buffer_B, ldb}, thread_id);

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

Mma mma;

accum.clear();

#pragma unroll 1
for (int k = 0; k < GemmK; k += Mma::Shape::kK) {
    iter_A.load(frag_A); // Load fragments from A and B matrices
    iter_B.load(frag_B);

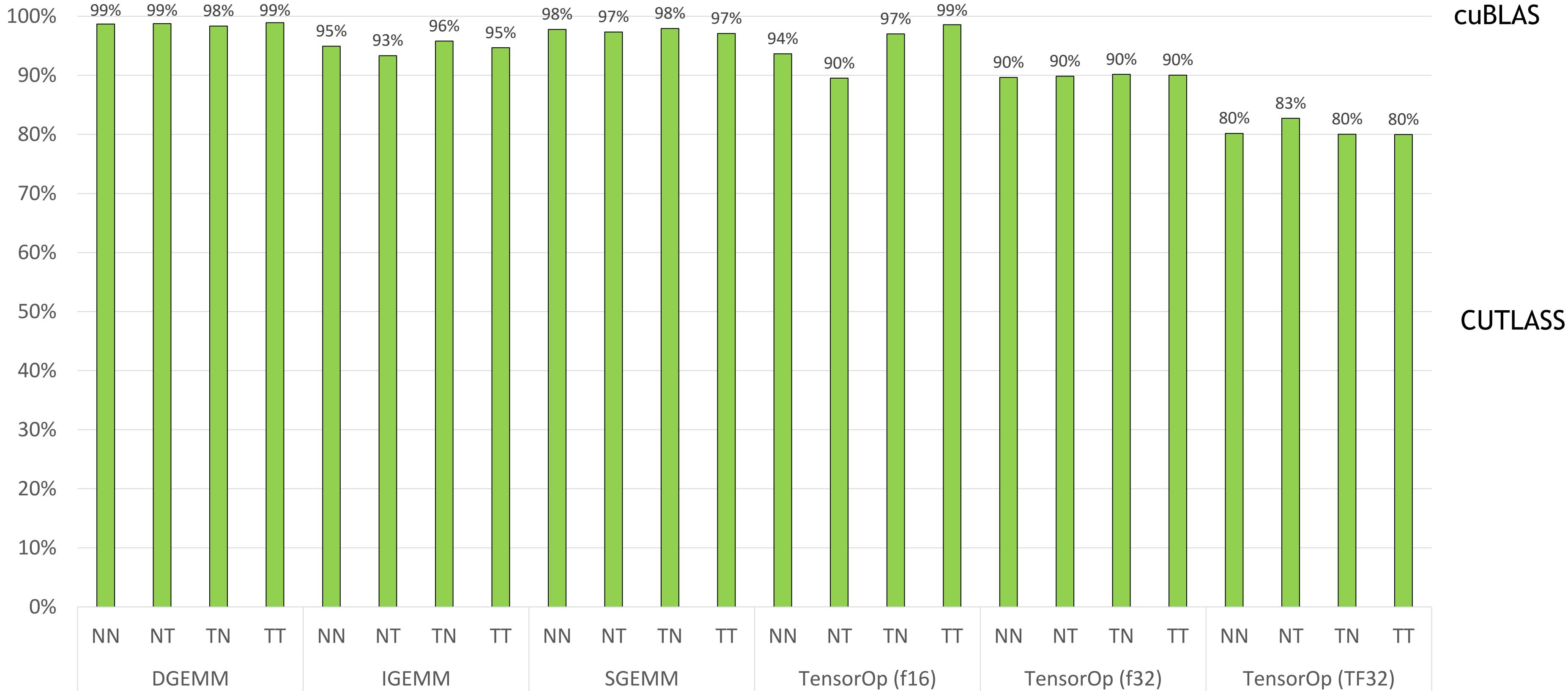
    ++iter_A; ++iter_B; // Advance along GEMM K to next tile in A
                        // and B matrices
    mma(accum, frag_A, frag_B, accum);
}
```



CUTLASS ON NVIDIA A100

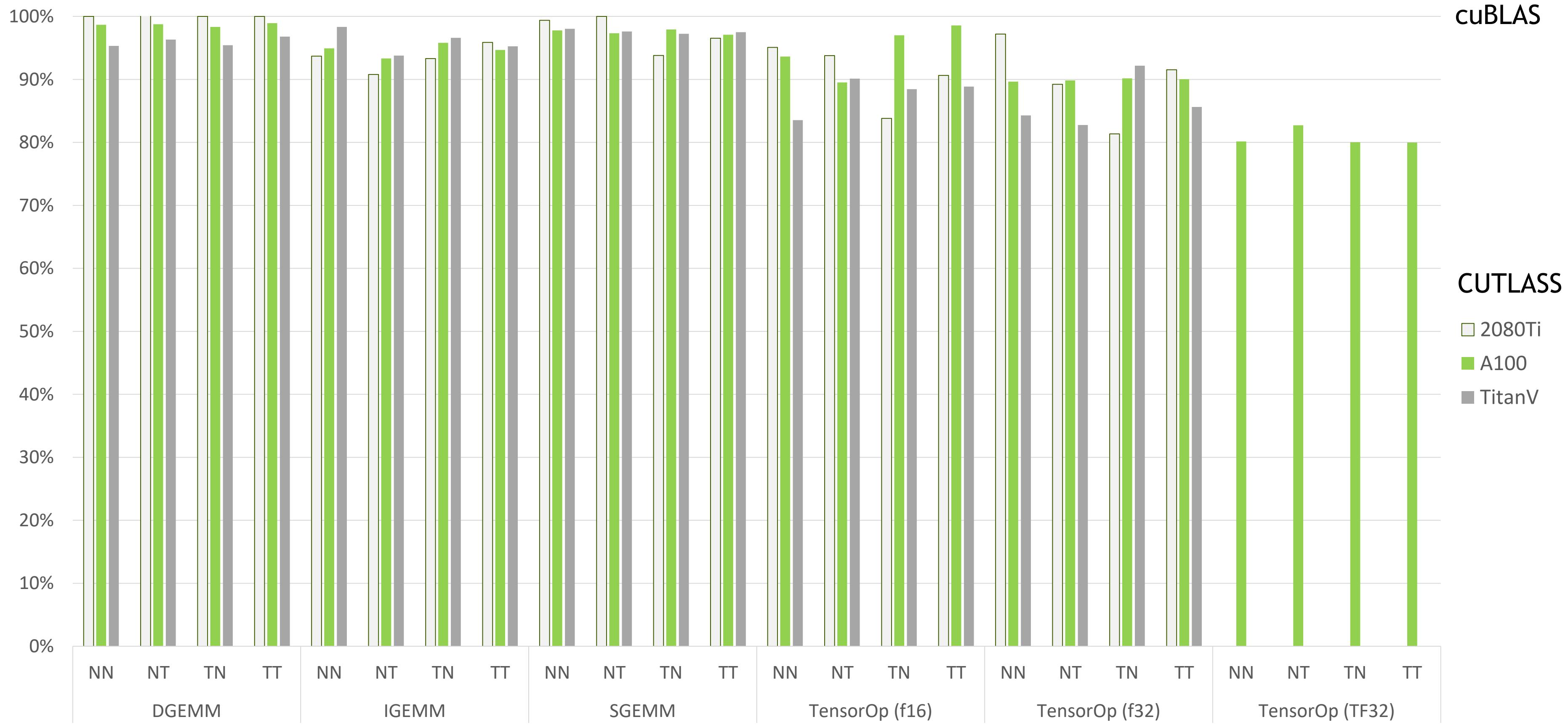
CUTLASS RELATIVE PERFORMANCE TO CUBLAS

CUTLASS 2.2 - CUDA 11 Toolkit - NVIDIA A100



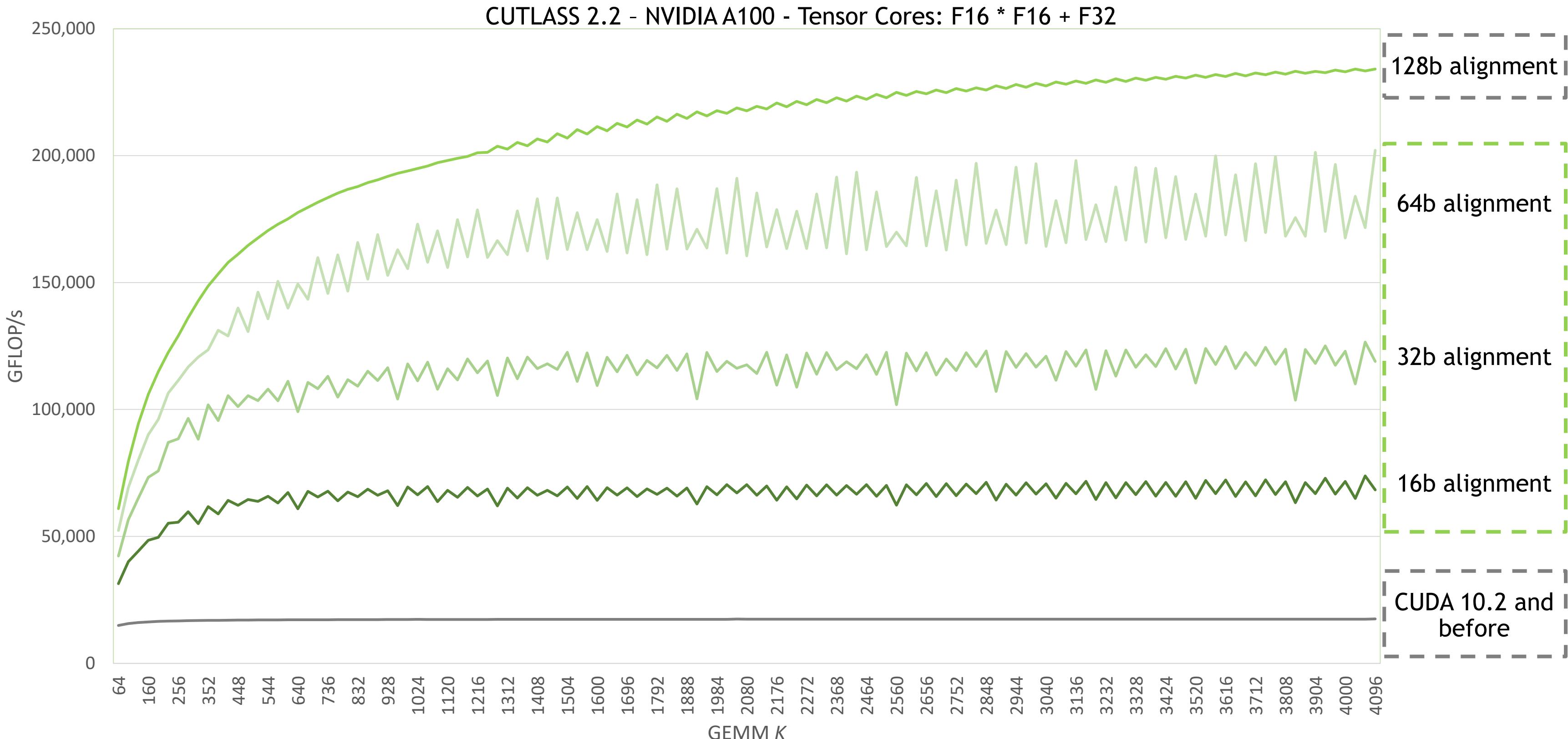
CUTLASS RELATIVE PERFORMANCE TO cuBLAS

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



ARBITRARY PROBLEM SIZE

CUTLASS Templates Cover the Design Space



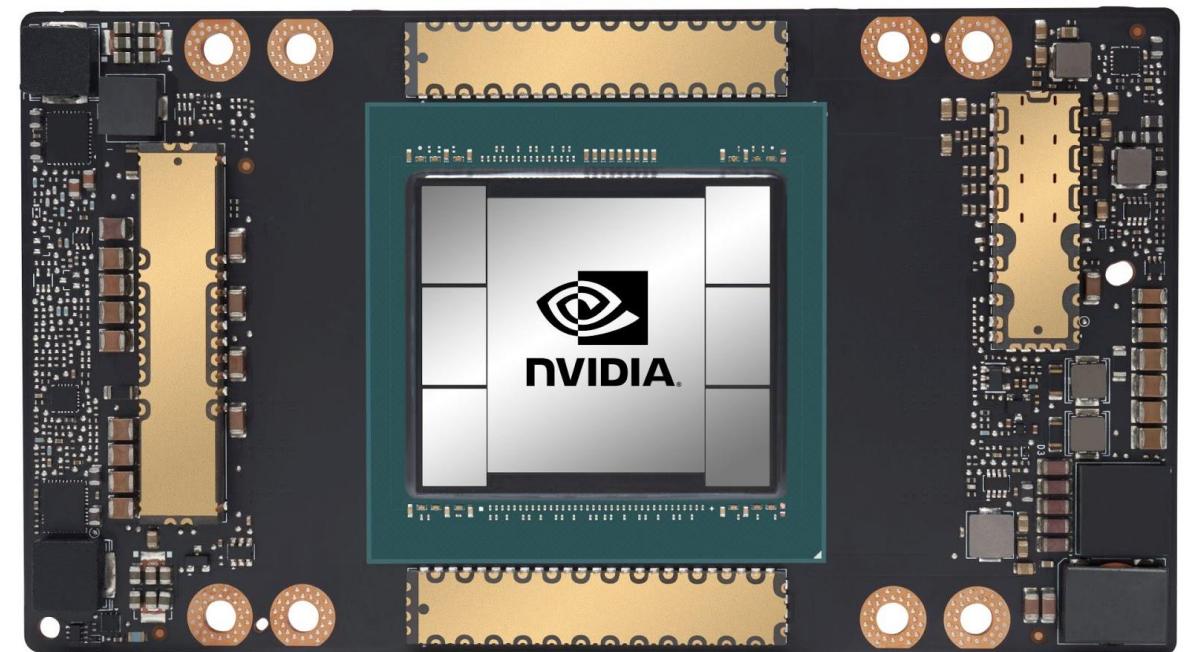


CONCLUSION

CONCLUSION: NVIDIA A100 IS FAST AND PROGRAMMABLE

Tensor Cores on NVIDIA A100 in CUDA

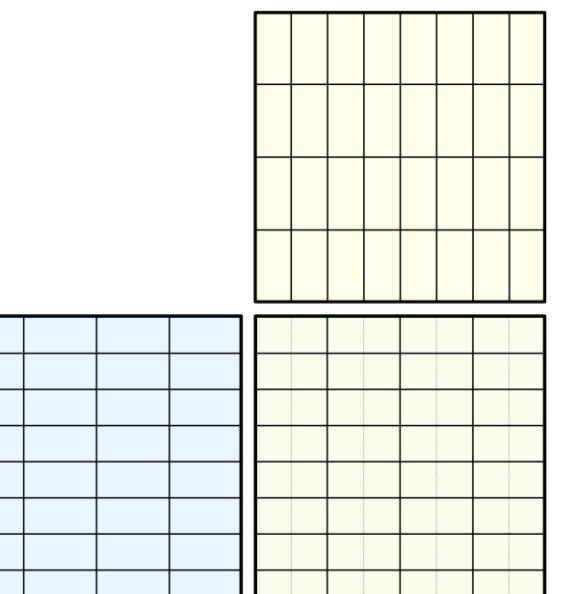
- Order of magnitude speedup for matrix computations
- Programmable in CUDA via `mma.sync` with zero overhead
- Kernel design can avoid memory bottlenecks
- CUDA 11 Toolkit capable of near-peak performance



CUTLASS 2.2: May 2020

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

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



REFERENCES

NVIDIA Ampere Architecture:

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

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

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

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

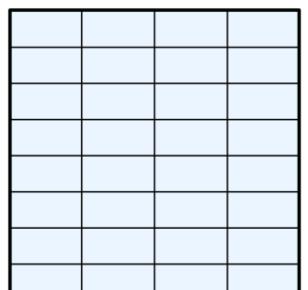
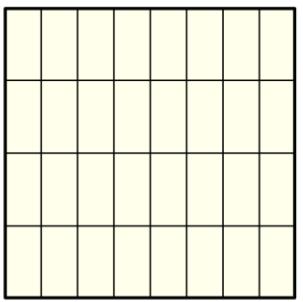
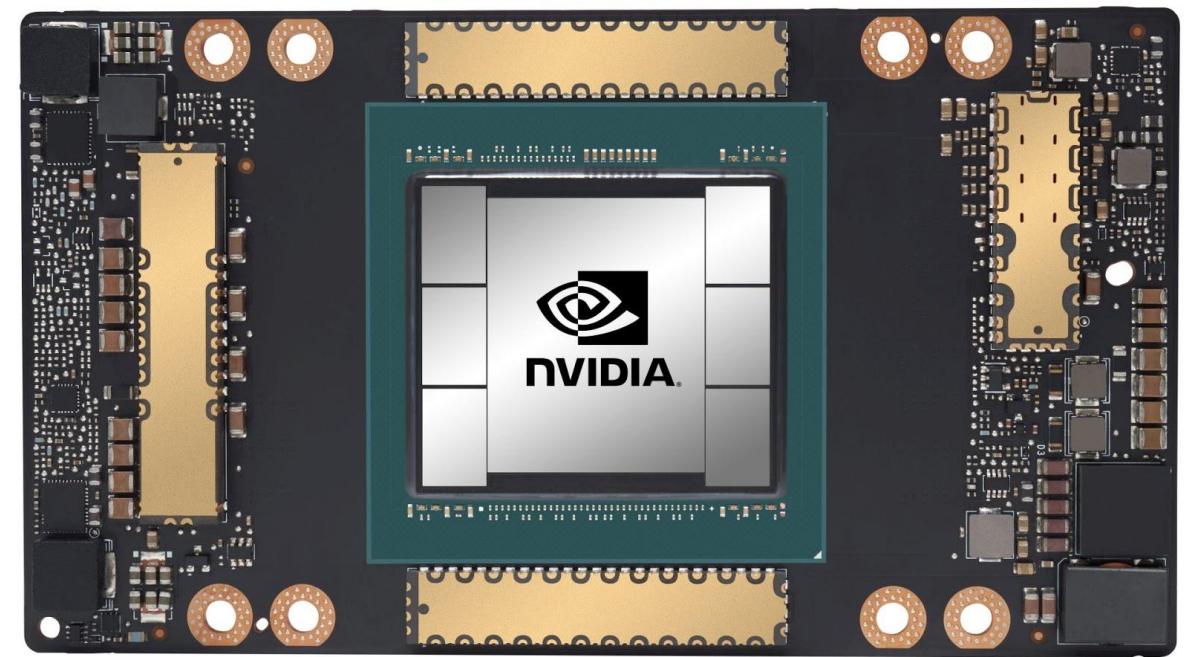
“Inside the Compilers, Libraries and Tools for Accelerated Computing” (GTC 2020 - S21766)

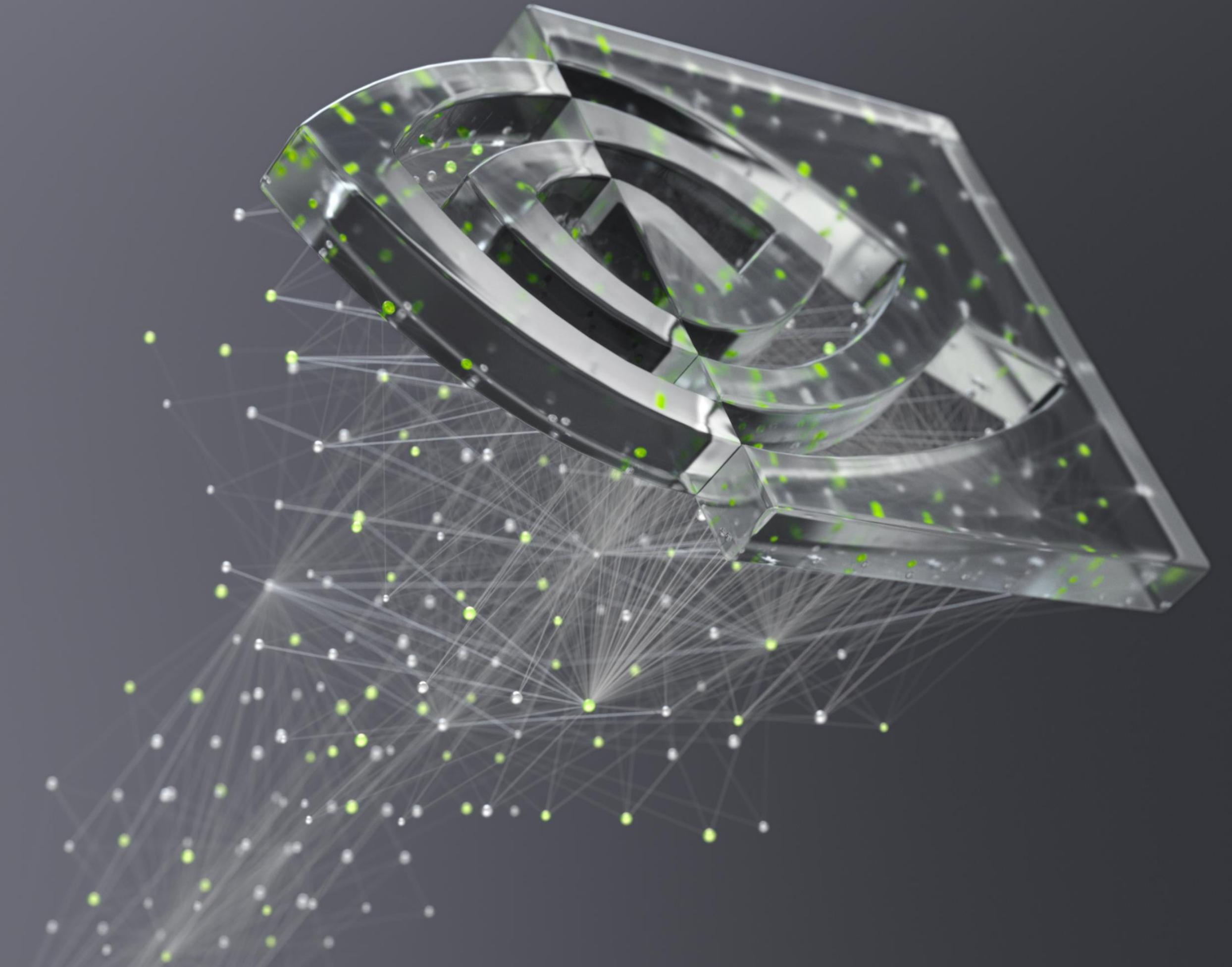
CUTLASS

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

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

CUTLASS Parallel For All [blog post](#)





NVIDIA®