



ECE408/CS483/CSE408

Applied Parallel Programming

Lecture 23: Accelerating Matrix Operations

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

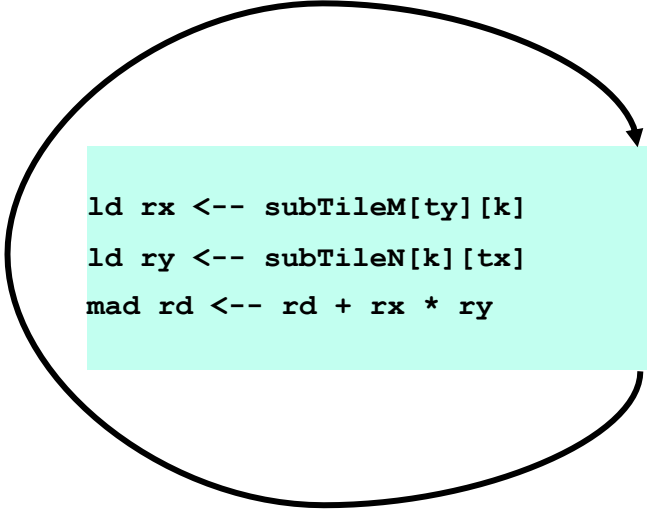
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    // The code assumes that the Width is a multiple of TILE_WIDTH!
8.  for (int q = 0; q < Width/TILE_WIDTH; ++q) {
        // Collaborative loading of M and N tiles into shared memory
9.      subTileM[ty][tx] = M[Row*Width + (q*TILE_WIDTH+tx)];
10.     subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.     __syncthreads();
15. }
16. P[Row*Width+Col] = Pvalue;
}
```

At the Operation Level, per Thread

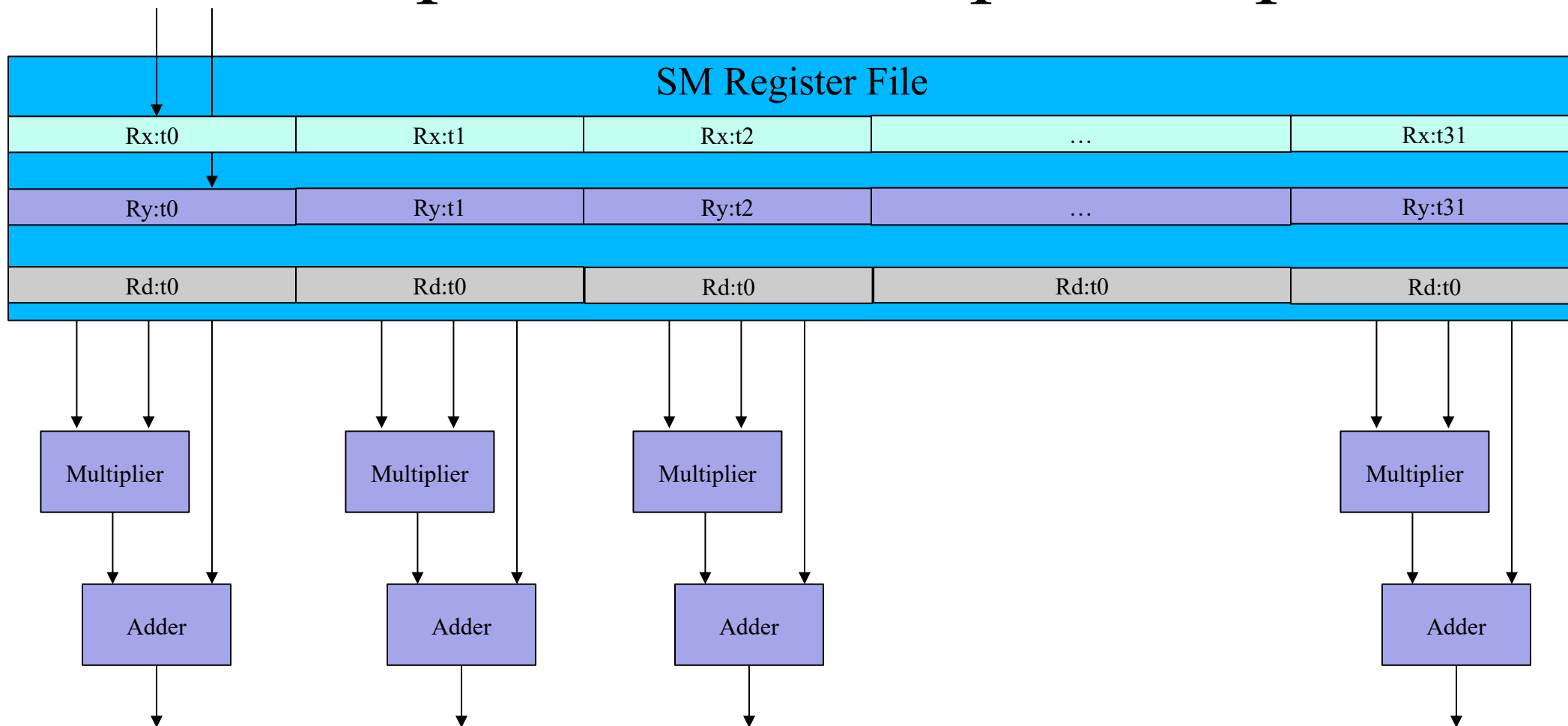
```
12.   for (int k = 0; k < TILE_WIDTH; ++k)
13.       Pvalue += subTileM[ty][k] * subTileN[k][tx];
```



```
ld rx <-- subTileM[ty][k]      // Load into register rx
ld ry <-- subTileN[k][tx]      // Load into register ry
mad rd <-- rd + rx * ry        // Multiply Add Instruction
```

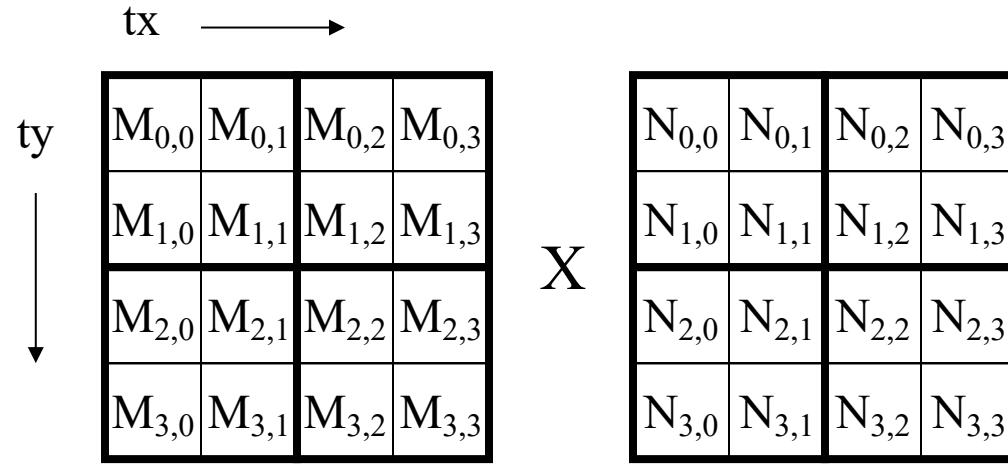
Each thread is able to complete the dot product in $O(k)$ cycles.

At the Operation Level, per Warp



With enough threadblocks, we have a throughput of $2 \times \text{width of SM FP Ops}$ per cycle

Let's look at a 4x4 tile example

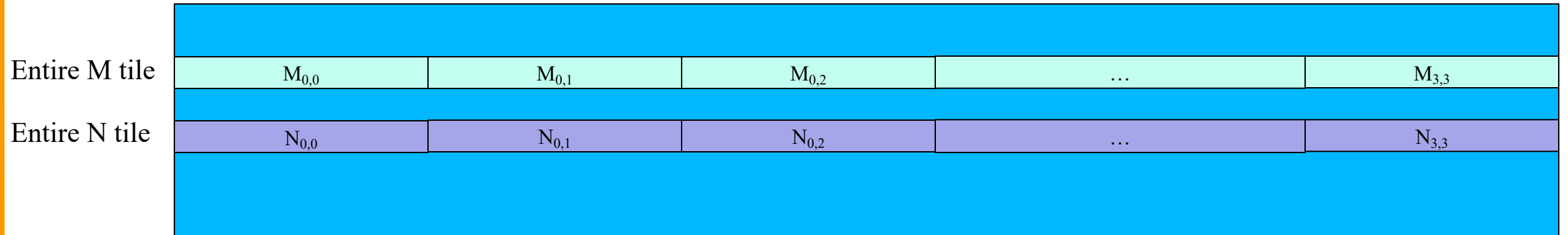


```
for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += subTileM[ty][k] * subTileN[k][tx];
```

Shared Memory Loads into Register File by Loop Iteration

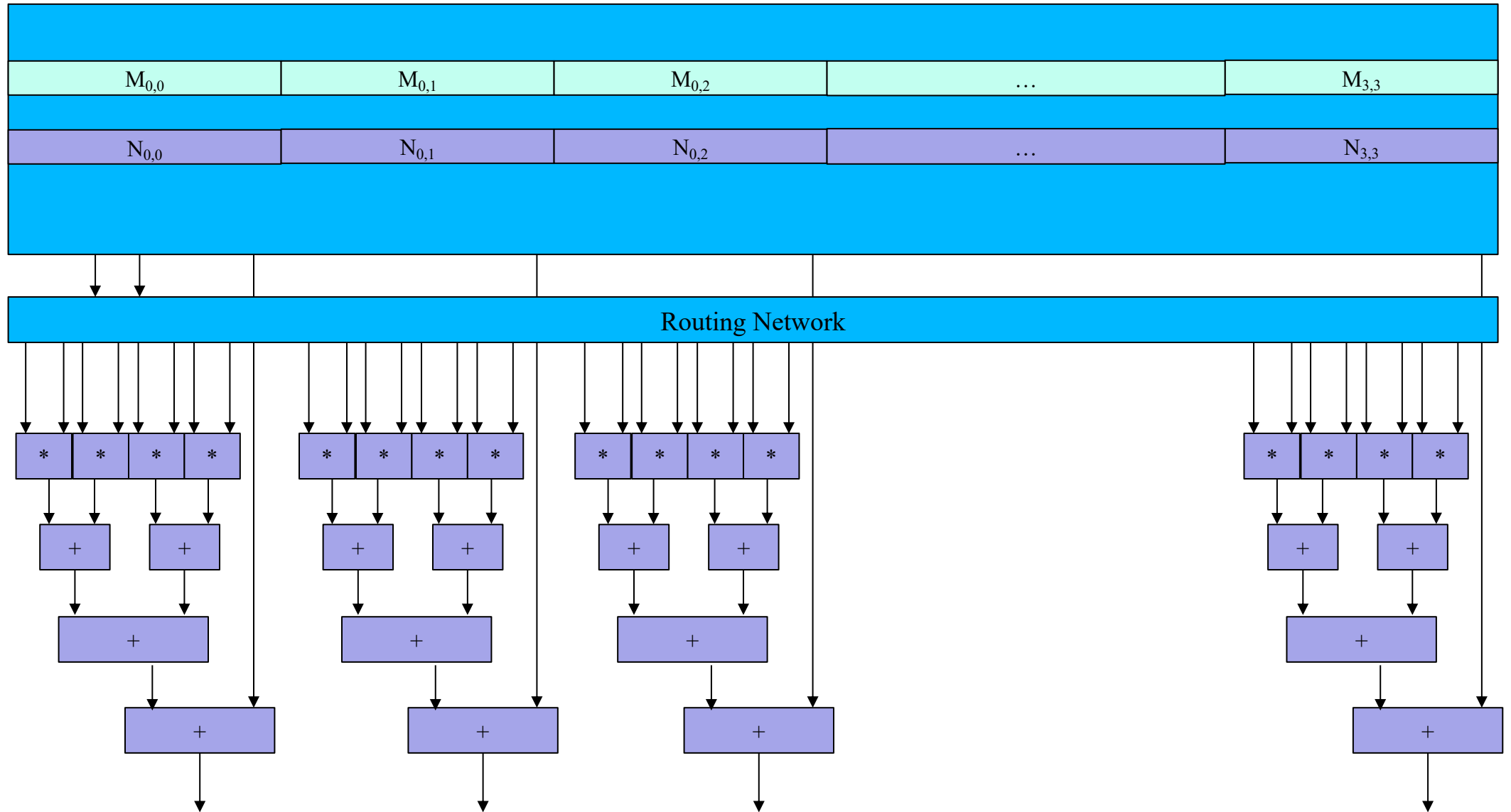
	t0	t1	t2	t3	t4	t5
$k = 0$	$M_{0,0}, N_{0,0}$	$M_{0,0}, N_{0,1}$	$M_{0,0}, N_{0,2}$	$M_{0,0}, N_{0,3}$	$M_{1,0}, N_{0,0}$	$M_{1,0}, N_{0,1}$
$k = 1$	$M_{0,1}, N_{1,0}$	$M_{0,1}, N_{1,1}$	$M_{0,1}, N_{1,2}$	$M_{0,1}, N_{1,3}$	$M_{1,1}, N_{1,0}$	$M_{1,1}, N_{1,1}$
$k = 2$	$M_{0,2}, N_{2,0}$	$M_{0,2}, N_{2,1}$	$M_{0,2}, N_{2,2}$	$M_{0,2}, N_{2,3}$	$M_{1,2}, N_{2,0}$	$M_{1,2}, N_{2,1}$

More Efficient Pattern



With two loads from shared memory across a warp (only 16 threads needed), we can load the entire 4x4 subtiles of M and N

Optimized Hardware



Notes on Optimized HW

- If we optimize for 16-bit types (and smaller)
 - We need $16b * 32 \text{ threads} = 512 \text{ bits per register}$
 - 4 16 bit multipliers per lane, but 32 bit adders
- Max Throughput is now increased by 4x

Nvidia Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 or FP32

Basic Tensor Operator: Matrix Multiply + Accumulate



CUDA TENSOR CORE PROGRAMMING

WMMA Matrix Multiply and Accumulate Operation

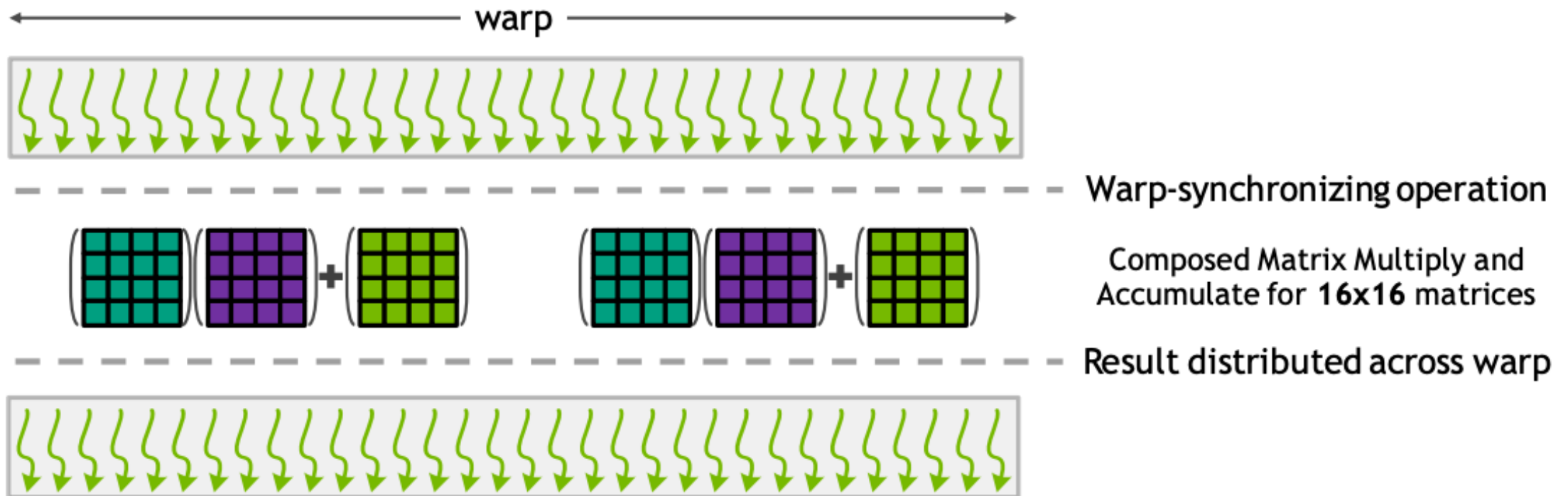
Warp-level operation to perform matrix multiply and accumulate

```
mma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

$$D = \begin{pmatrix} \text{Teal 8x8 Grid} \end{pmatrix} + \begin{pmatrix} \text{Purple 8x8 Grid} \end{pmatrix} + \begin{pmatrix} \text{Green 8x8 Grid} \end{pmatrix}$$

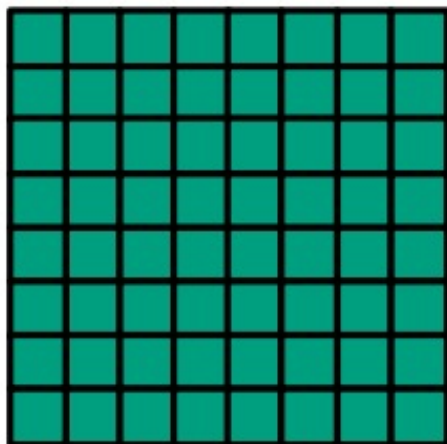
TENSOR SYNCHRONIZATION

Full Warp 16x16 Matrix Math



CUDA TENSOR CORE PROGRAMMING

WMMA datatypes



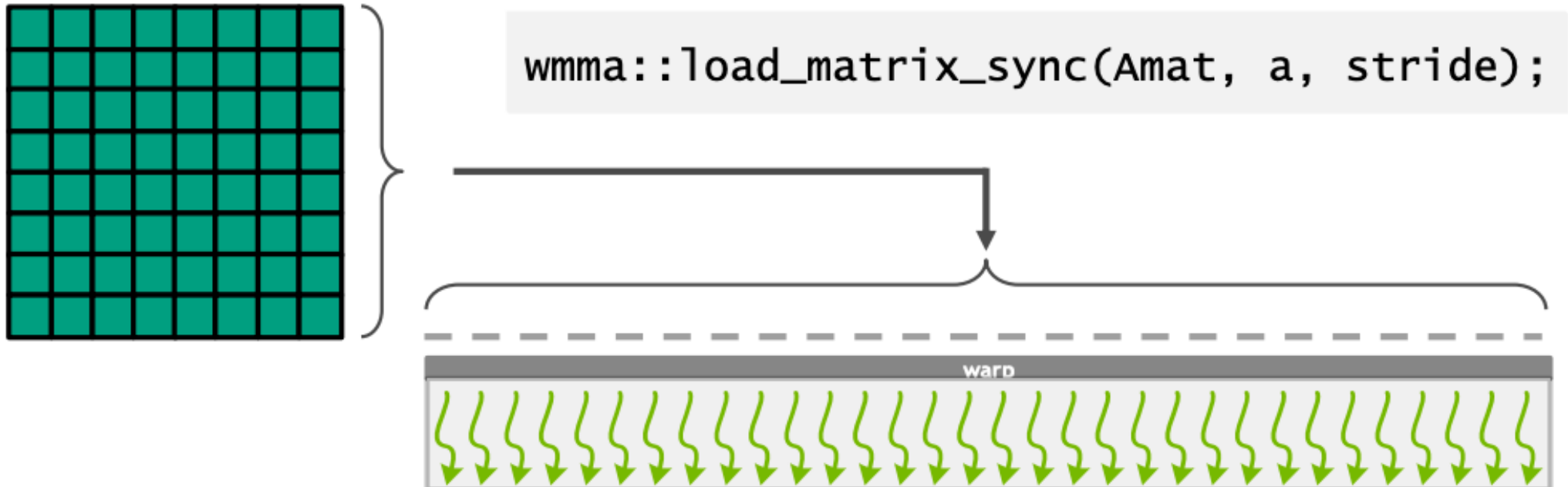
Per-Thread fragments to hold components of matrices for use with Tensor Cores

```
wmma::fragment<matrix_a, ...> Amat;
```

CUDA TENSOR CORE PROGRAMMING

WMMA load and store operations

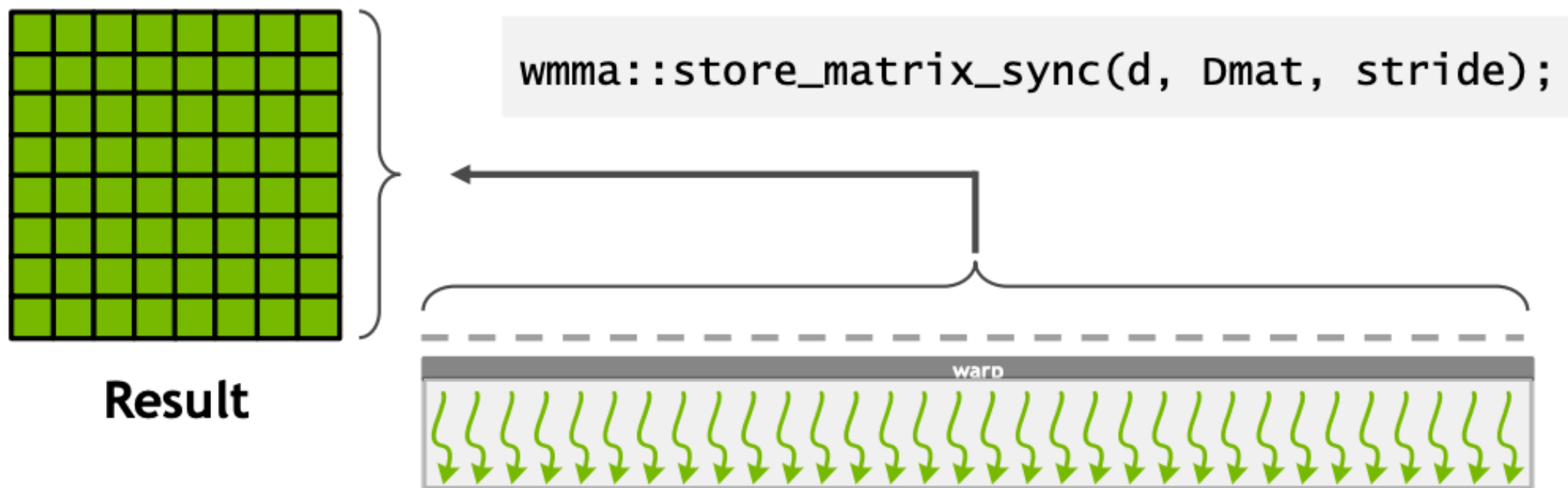
Warp-level operation to fetch components of matrices into fragments



CUDA TENSOR CORE PROGRAMMING

WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments



TENSOR CORE EXAMPLE

Create Fragments

Initialize Fragments

Perform MatMul

Store Results

```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
    wmma::fragment<matrix_a, ...> Amat;
    wmma::fragment<matrix_b, ...> Bmat;
    wmma::fragment<matrix_c, ...> Cmat;

    wmma::load_matrix_sync(Amat, a, 16);
    wmma::load_matrix_sync(Bmat, b, 16);
    wmma::fill_fragment(Cmat, 0.0f);

    wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

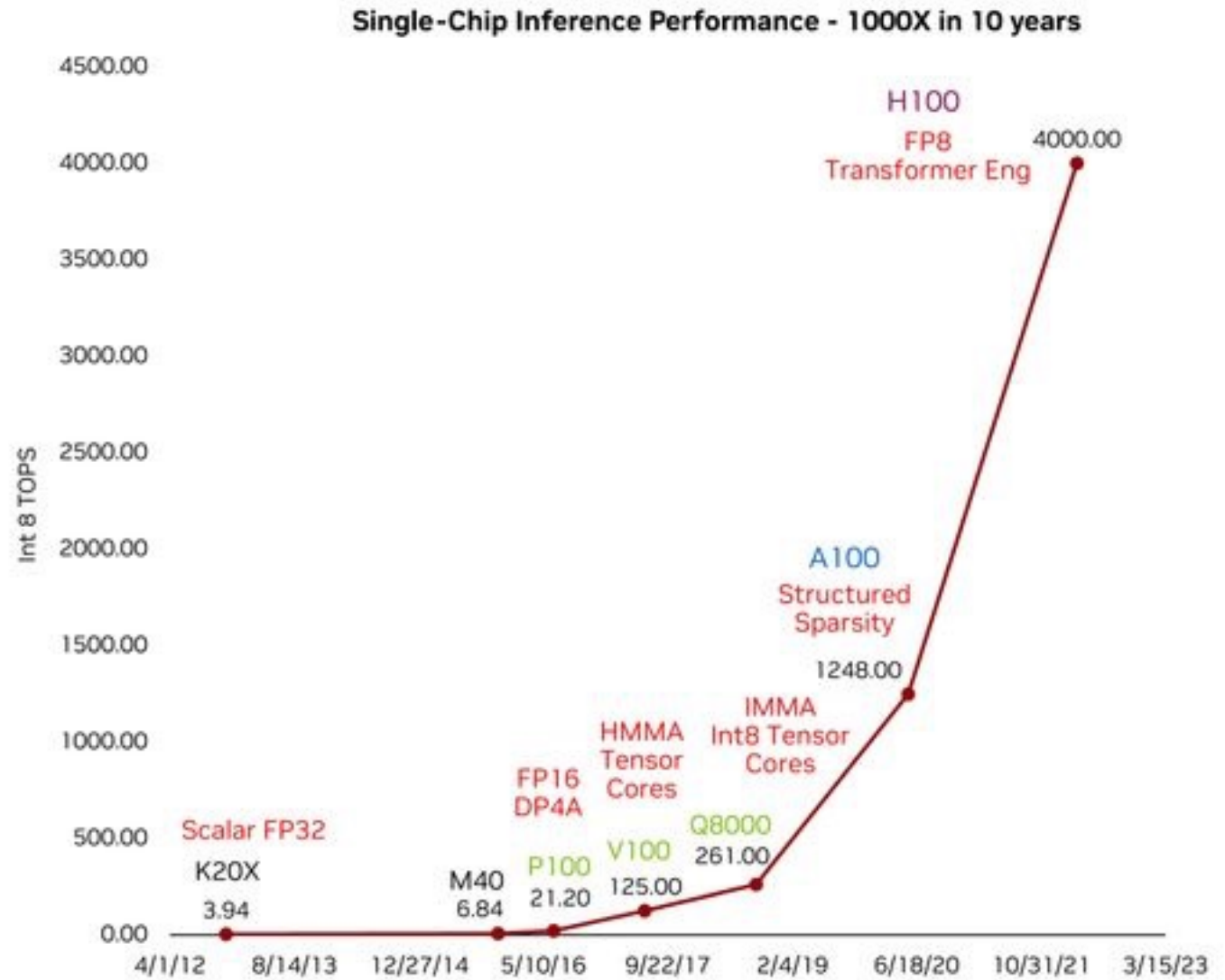
    wmma::store_matrix_sync(d, Cmat, 16,
        wmma::row_major);
}
```

CUDA C++

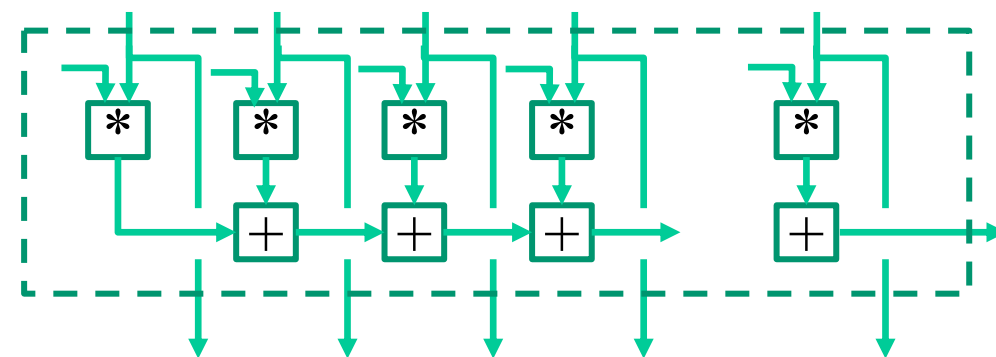
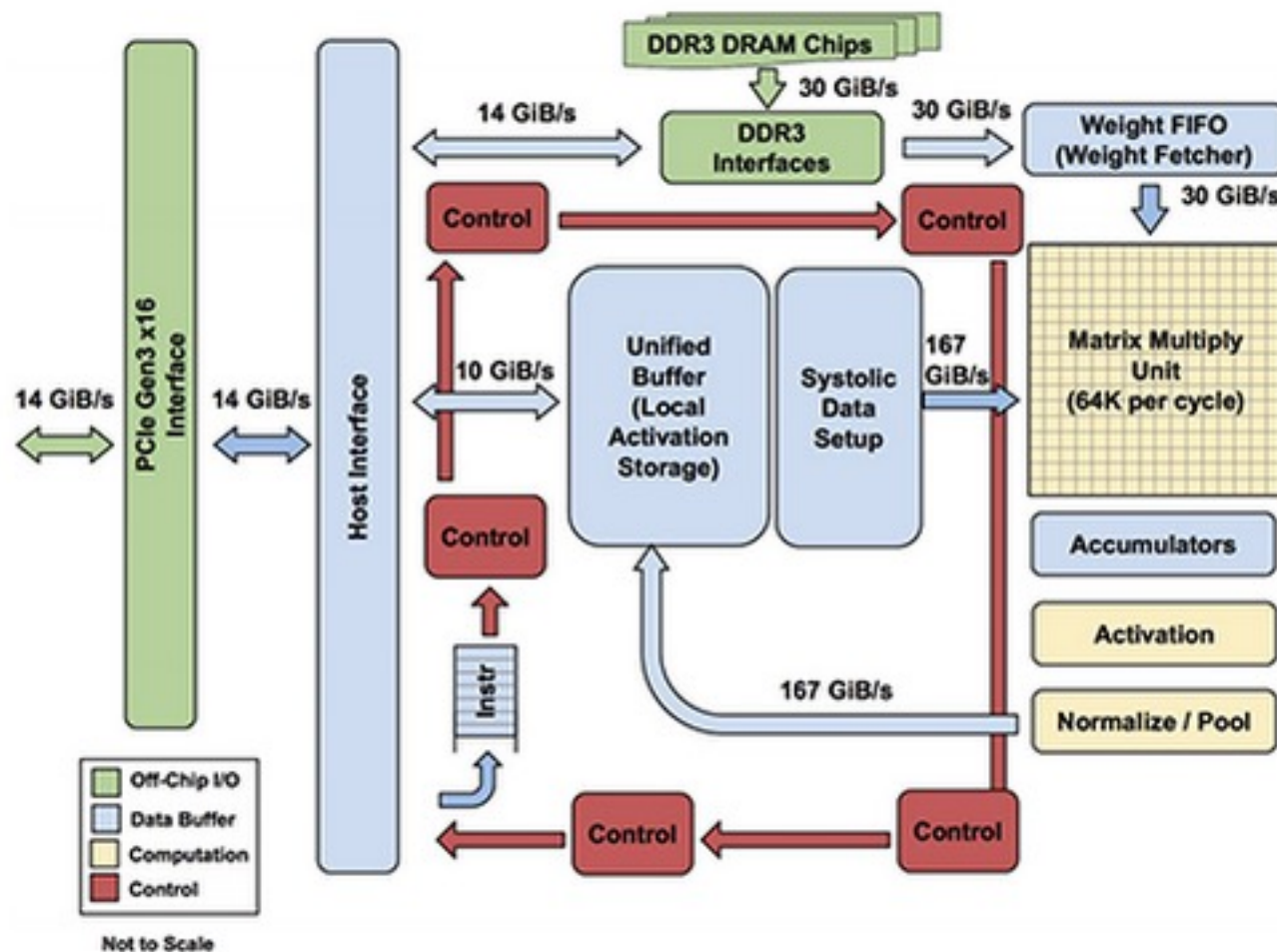
Warp-Level Matrix Operations

Gains from

- Number Representation
 - FP32, FP16, Int8
 - (TF32, BF16)
 - ~16x
- Complex Instructions
 - DP4, HMMA, IMMA
 - ~12.5x
- Process
 - 28nm, 16nm, 7nm, 5nm
 - ~2.5x
- Sparsity
 - ~2x
- Model efficiency has also improved – overall gain > 1000x



Google TPU (v1 in 2016)



Intel Advance Matrix Extensions (AMX)

- Introduced in 2020
- Introduced into x86 processors
- Usable directly from CPU code

```
// Load tile configuration
init_tile_config (&tile_data);

// Init src matrix buffers with data
init_buffer (src1, 2);
print_buffer(src1, rows, colsb);

init_buffer (src2, 2);
print_buffer(src2, rows, colsb);

// Init dst matrix buffers with data
init_buffer32 (res, 0);

// Load tile rows from memory
_tile_loadd (2, src1, STRIDE);
_tile_loadd (3, src2, STRIDE);
_tile_loadd (1, res, STRIDE);

// Compute dot-product of bytes in tiles with a
// source/destination accumulator
_tile_dpssd (1, 2, 3);

// Store the tile data to memory
_tile_stored (1, res, STRIDE);
```