WGMMA Part 2

Prateek Shukla

Grouping wgmma operations

WGMMA launches are async. wgmma.mma_async puts work “in flight” and returns immediately.

If we had to “track every MMA” individually, we’d either pay heavy scoreboard overhead or be forced into overly conservative “wait for everything” barriers.

Groups give us a pipeline-friendly granularity.

`wgmma.commit_group` = “close the current batch” (make it trackable)

`wgmma.wait_group N` = “stall only when too many batches are in flight”

This enables overlap: while group g computes, you can prep/load for group g+1 (address math, TMA, staging, etc.).

`wgmma.commit_group.sync.aligned;`

This is just a way to bundle all the `wgmma.mma_aync` operation which are not yet committed

The hardware can track multiple groups at once. By grouping them, you reduce the overhead of tracking every single matrix multiply individually

You usually issue a set of WGMMA instructions that calculate one "tile" of your output (e.g., a 64x64 chunk), and then commit that tile as one group.

We then wait on these operations

`wgmma.wait_group.sync.aligned N`

This is the synchronization point. It says: "Pause this thread until there are only N committed groups still running."

`wgmma.wait_group.sync.aligned 0` - Wait for **EVERYTHING** to finish. You use this when you are about to read the final result in registers to write it to global memory.

`wgmma.wait_group.sync.aligned N` - keep N groups running in the background while you move on to prepare data for the next group

The “Commit/Wait” Timeline (what actually overlaps)

You don’t just “commit then wait”. You pipeline it.

Issue a bunch of wgmma.mma_async (this is one output tile)

`wgmma.commit_group.sync.aligned;`

Start preparing the NEXT tile (TMA, pointer math, etc.)

`wgmma.wait_group.sync.aligned N;` (keep N groups running)

Drain

wgmma.wait_group.sync.aligned 0;

Now it is safe to read accumulator D and store it out

`wgmma.fence.sync.aligned`

Warpgroup fence for the WGMMA pipeline. It serves two tightly related roles:

Arrive/group-boundary marker for WGMMA issue: WGMMA ops are issued in groups. The fence is used to establish the start of a group’s issuance window.

Register ordering / hazard control for operands used by WGMMA

`wgmma.mma_async` is asynchronous and uses a warpgroup-level execution pipeline. The fence is used to prevent reordering/hazards involving the accumulator registers and any register-resident operand fragments (notably A when it’s register-sourced).

It is not the cross-proxy fence. If you are reading register modified TMA data, you need a cross-proxy fence: fence.proxy.async (or an equivalent acquire/release publication protocol).

The store

When the WGMMA instruction finishes, your result matrix C is sitting in Accumulator Registers. The result is not stored as a contiguous matrix (e.g., Row 0, Row 1, Row 2...). Instead, it is fragmented across the registers of 128 threads in a highly opaque manner

In a normal store (st.shared), if Thread 1 writes data, Thread 1 provides the address, we can't use this because this would require manually managing addresses.

In stmatrix, the data provider and the address provider are decoupled just like `ldmatrix`.

The instruction stmatrix

Every single thread inputs the data which it got in its register from wgmma and then it gives the destination pointer to write this data to

```
asm volatile("stmatrix.sync.aligned.m8n8.x4.trans.shared::cta.b16 [%0], {%1, %2, %3, %4};"
             :: "r"(addr), "r"(data_ptr[0]), "r"(data_ptr[1]), "r"(data_ptr[2]), "r"(data_ptr[3]));
```

.m8n8: The atomic unit is still the 8x8 tile.

.x4: Writes 4 registers per thread (16x16 tile total).

.x1, .x2: Also supported for smaller tiles.

.shared: The destination is always Shared Memory.

`stmatrix.sync.aligned` - we are storing data for a matrix and all warps must synchronize before proceeding and guaranteeing that the every single thread is launching the instruction

.m8n8, .m16n8 tell us about the shape of the matrix stored. Note that .m16n8 shape is valid only for .b8 type in blackwell.

.x{n} tells us how many registers each thread holds. Greater N means larger tile

.b16: This denotes the data type and size of the elements being stored

```
// Store a single 8x8 matrix using 64-bit addressing
.reg .b64 addr;
.reg .b32 r;
stmatrix.sync.aligned.m8n8.x1.shared.b16 [addr], {r};

// Store two 8x8 matrices in column-major format
.reg .b64 addr;
.reg .b32 r<2>;
stmatrix.sync.aligned.m8n8.x2.trans.shared::cta.b16 [addr], {r0, r1};

// Store four 8x8 matrices
.reg .b64 addr;
.reg .b32 r<4>;
stmatrix.sync.aligned.m8n8.x4.b16 [addr], {r0, r1, r2, r3};

// Store a single 16x8 matrix using generic addressing
.reg .b64 addr;
.reg .b32 r;
stmatrix.sync.aligned.m16n8.x1.trans.shared.b8 [addr], {r};

// Store two 16x8 matrices
.reg .b64 addr;
.reg .b32 r<2>;
stmatrix.sync.aligned.m16n8.x2.trans.shared::cta.b8 [addr],{r0, r1};

// Store four 16x8 matrices
.reg .b64 addr;
.reg .b32 r<4>;
stmatrix.sync.aligned.m16n8.x4.b8 [addr], {r0, r1, r2, r3};
```

`stmatrix` and fp32

The most critical thing to understand is that stmatrix does not support f32 data. It only supports .b16 (packed 16-bit) and .b8 (packed 8-bit) types. This creates a divergence in how you handle the output of wgmma.

When using FP32 Accumulator (.f32), Register State: The accumulator resides in unpacked .f32 registers. You cannot feed these registers to stmatrix. You must downcast and pack them first(yes this would give you lossy results)

If using FP16 Accumulator (.f16), the accumulator resides in packed .b32 registers (treated as .f16x2). One 32-bit register holds two elements thus there is no requirement for conversion and packing

`stmatrix`

Like `ldmatrix, stmatrix` is cooperative. Every single thread in the warp (0 to 31) provides its own pointer. Each thread gives the shared memory address in which its data belongs.

Because the threads in a warp hold data in a very specific, non-linear pattern the addresses they generate must be equally non-linear to ensure the data lands in shared memory as a contiguous, clean matrix

When you execute stmatrix, The instruction writes these elements into SMEM in row major Order (contiguous rows) starting at [addr].

The m8n8 Tile Structure

The instruction operates on an 8x8 Tile of 16-bit elements.

Threads 0-31: The warp cooperates.

x1 Variant: Each thread holds part of one 8x8 matrix.

x4 Variant: Each thread holds data for four 8x8 matrices

Like ldmatrix, it assumes specific threads hold specific rows. Threads 0-7 hold Row 0, etc.

Phase 1: The Address Responsibility (who points where?)

This works exactly like ldmatrix.

```
Threads 0-7: Point to Rows 0-7 (Left Half, Cols 0-7).

Threads 8-15: Point to Rows 8-15 (Left Half, Cols 8-15).

Threads 16-23: Point to Rows 0-7 (Right Half, Cols 0-7).

Threads 24-31: Point to Rows 8-15 (Right Half, Cols 8-15).
```

The Register Source (who holds what)

The "Striped" Layout Thread 0 acts as the "Column 0 & 1" owner for the entire tile. It does not see continuous rows; it sees vertical slices.

Thread 0 Register Map:

```
Reg r0: Matrix A (Top-Left) → Row 0, Cols [0, 1]

Reg r1: Matrix B (Bot-Left) → Row 8, Cols [0, 1]

Reg r2: Matrix C (Top-Right) → Row 0, Cols [8, 9]

Reg r3: Matrix D (Bot-Right) → Row 8, Cols [8, 9]
```

The Serialization Loop

Since stmatrix.x4 consumes only 4 registers per thread per call. But the accumulator for one output tile is bigger. So we do a loop over “slices” of the accumulator:

Slice i → pack into {r0,r1,r2,r3}

stmatrix(...)

Slice i+1 → pack → stmatrix(...)

This is why the epilogue is usually a loop, not a single store.

The big question

The “atom” we are swizzling is 16B.

16B matters because a single row of an m8n8 bf16 tile is 8 elements = 16 bytes. So each row is naturally a 16B chunk.

The trick is that We swizzle the row-atoms, not individual elements

Even when atoms are permuted, the data inside each 16B atom stays row-major

That’s why TMA can “unswizzle” back to a clean row-major matrix.

FP8

FP8 Tensor Cores deliver theoretically double the TFLOPs of BF16/FP16, reaching ~3958 TFLOPs on H100(Numbers shown with sparsity. Dense is ~½ these values). This comes due to the lower precision

Halving the bit-width reduces global memory bandwidth pressure and allows caching 2x larger models/batches in L2.

Most of the win is in matmul-heavy parts (linear layers, attention projections, MLPs). Many pipelines still keep some ops in BF16/FP16 (softmax-ish, norms, weird reductions).

The paradigm shift

Until roughly 2022, the standard for AI was FP32 or FP16/BF16. FP8 is primarily an industrial advancement driven by Nvidia's H100 (Hopper) architecture which allowed researchers to train models twice as fast using half the memory.

A lot of really big models like deepseek V3 used fp8 for training and kimi k2 famously used int4 Quantization-Aware Training.

Unlike INT8, FP8 is a "floating point", meaning it can represent a wide dynamic range. Because 8 bits is very cramped, engineers split FP8 into two specialized formats to balance Precision vs. Range: e4m3 and e5m2

Floating Point Representation (IEEE 754)

A floating point number is encoded as three fields:

$$(-1)^S \times 2^{E - \text{bias}} \times (1 + M)$$

Field	Purpose
Sign (S)	$\pm$
Exponent (E)	Dynamic range
bias	Offset calculated at $2^{e-1} - 1$, where e is the number of exponent bits for a specific floating point format
Mantissa (M)	Fractional precision

Wgmma on lower precision

We have 2 lower precisions

- e4m3 -
  - 4-bit exponent and 3-bit mantissa
  - Range: ≈ ±448 (max normal value 448, min normal 0.015625)
  - Primarily used for Weights and Activations (forward pass) because it has good dynamic range and represents NaN/Inf.
- e5m2 -
  - 5-bit exponent and 2-bit mantissa
  - Range: ≈ ±57344 (much larger than E4M3)
  - Primarily used for training gradients and back-propagation because it can represent very large numbers needed during gradient updates.

Saturation

In FP32, your dynamic range is massive (~10^38). You effectively never hit the ceiling

In FP8 E4M3, the "ceiling" is 448. If your matrix multiplication results in a value of 450, and you don't handle it, you hit saturation. When you cast a float down to FP8, the hardware has to decide what to do with that 450. It usually has two modes (controlled by intrinsics)

The value clamps to the maximum representable number (448). This distorts your data (clipping), but keeps the math "stable."

The value becomes Infinity (or NaN in E4M3, which has no Inf representation). This destroys your training run immediately

Scaling Factors

Since we can't change the physics of 8 bits, we have to cheat. We use Scaling Factors. This is the core curriculum of FP8 on H100.

Tensor-wise Scaling: You calculate one single scaling factor (max value) for the entire tensor matrix. This is simplest to implement with lowest overhead

Vector-wise Scaling: You assign a unique scaling factor to each row or column.

Block-wise Scaling (MXFP8 / Micro-scaling): The matrix is chopped into small tiles (e.g., 16×16 or 32×32). Each tile gets its own scale.

Others like SmoothQuant, Delayed Scaling etc

Packing

FP8 does not exist as a standalone object during transport. The H100 memory controller does not wake up for anything less than 32 bytes (a "sector"). If you ask for 1 byte (one FP8 value), the GPU fetches 32 bytes, gives you the 1 you asked for, and throws the other 31 away.

To fix the waste, we have the Packed Types e4m3x2/4 and e5m2x2/4. These are distinct PTX data types that the hardware understands.

The x4 Pack (The Register Filler) Type: .e4m3x4 or .e5m2x4

The x2 Pack (The Math Input) Type: .e4m3x2 or .e5m2x2

The x4 Pack

These are packed datatypes which contain 4 fp8 values: e4m3x4 and e5m2x4

Total Size: 8 bits×4=32 bits.

Why it exists: It perfectly fills one standard GPU register.

The Layout (Little Endian):

[Element 3|Element 2|Element 1|Element 0]

<-- MSB (31) LSB (0) -->

This is your primary format for storage and transport. When you are moving data around or doing simple math (like finding max values), you want x4.

The x2 Pack

Packed datatypes which contain 2 fp8 elements: .e4m3x2 or .e5m2x2

Total Size: 8 bits×2=16 bits.

This matches the size of FP16 (half).

H100 is designed to transition people from FP16 to FP8.

The hardware has dedicated logic to take a 32-bit register holding two FP16s and compress them directly into a 16-bit register holding two FP8s.

The Tensor Core often ingests data in these 16-bit chunks.

Fp8 conversion/quantization

If your source data is FP16/FP32, you'll convert to FP8 via `cvt` (often vector forms like `.e4m3x2/.e5m2x2`) and usually with .satfinite so out-of-range values clamp:

decide your quantization policy (rounding mode, satfinite or not, any per-tensor/per-channel scaling) before feeding WGMMA.

PTX describes saturation behavior: if |input| exceeds the destination max normal, the result becomes sign-preserved max normal.

PTX includes `cvt.satfinite.{e4m3x2,e5m2x2}.{f32,f16x2}` support for sm_90 or higher

`` `cvt.rn.satfinite.e4m3x2.f16x2` `` This instruction takes the 2 fp16 values, Saturates them, Rounds them and the packs them

Hopper 4th generation tensor cores with fp8

Inputs are 8-bit (FP8), but accumulation happens in 32-bit (FP32) registers for numerical stability.

Native support for both E4M3 and E5M2 inputs via WGMMA

One WGMMA instruction issues a massive chunk of math, hiding latency effectively

While the register is FP32, it acts like FP22((~8-bit exponent + ~13-bit mantissa) , a known hardware trait.

We essentially treat FP8 as a compressed storage format while performing math in a "pseudo-FP32" space.

The FP8 Instruction Syntax

```
wgmma.mma_async.sync.aligned.m64nNk32.f32.e4m3.e4m3 D, A, B_desc, scale_D, scale_A, scale_B;
```

m64nNk32: Fixed M=64, K=32 (for 8-bit). N is variable (e.g., 8, 16, ... 256).

.f32: The accumulator D type (Single Precision).

.e4m3.e4m3: Input types for Matrix A and B. FP8 is special in that A and B may be different FP8 formats (e.g. .e4m3 × .e5m2). The PTX examples show mixed .e4m3/.e5m2 pairings.

scale_*: these are the scaling factors, the sign flippers which can go from -1 to 1({-1, 1} for A and B and {0, 1} for scale_d).

K = 32 is the crucial difference, this means we consume 32 columns of A in one go

A on registers

For FP8 RS WGMMA, PTX shows the A operand as a vector expression of registers passed directly to WGMMA.

The “obvious” way (load true 8-bit elements with ldmatrix … .b8) is not an option on SM90a PTX notes say .b8 with ldmatrix is supported on sm_100a / later, not Hopper. You can’t use ldmatrix with fp8

That’s why most Hopper FP8 WGMMA implementations do either:

SS path: use descriptors for A and B or

RS path: load A into regs via plain shared loads (ld.shared.b32 / vectorized), already arranged in the packing WGMMA wants, not via ldmatrix.

The Strict K-Major Rule

`wgmma.mma_async` does not expose transpose controls (imm-trans-a/b), unlike FP16/BF16 variants

With no transpose, WGMMA interprets shared operands in the default K-major canonical layout. If you feed MN-major tiles, the instruction can't "reinterpret" them so you must pack/transpose anyway

That transpose/pack step adds extra instructions, sync, and shared traffic, hurting throughput

MN-major also complicates swizzle-atom alignment/divisibility constraints for FP8 (128b atom scaling)

Stage FP8 tiles in K-major (or prepack offline); only use MN-major if you'll transpose during staging

Precision Pitfalls

Even when the wgmma instruction is documented as using an FP32 accumulator, empirical studies have reported that FP8 Tensor Core accumulation can behave like a reduced-precision FP32 variant (~8-bit exponent + ~13-bit mantissa, i.e. “FP22 total bits”), meaning the lowest FP32 mantissa bits may be effectively lost during accumulation.

For very large dot products (large K), this reduced effective mantissa can increase rounding/truncation error, and the error can compound with the reduction depth.

Use K-slicing and/or multi-stage accumulation (accumulate partial sums, then reduce partials in higher precision) to limit the accumulation depth per WGMMA “chunk” and improve numerical stability.

FP8 wgmma instruction

```
wgmma.mma_async.sync.aligned.shape.dtype.atype.btype  d, a-desc, b-desc, scale-d, imm-scale-a, imm-scale-b;

wgmma.mma_async.sync.aligned.shape.dtype.atype.btype  d, a, b-desc, scale-d, imm-scale-a, imm-scale-b;

.shape   = {.m64n8k32, .m64n16k32, .m64n24k32, .m64n32k32,
            .m64n40k32, .m64n48k32, .m64n56k32, .m64n64k32,
            .m64n72k32, .m64n80k32, .m64n88k32, .m64n96k32,
            .m64n104k32, .m64n112k32, .m64n120k32, .m64n128k32,
            .m64n136k32, .m64n144k32, .m64n152k32, .m64n160k32,
            .m64n168k32, .m64n176k32, .m64n184k32, .m64n192k32,
            .m64n200k32, .m64n208k32, .m64n216k32, .m64n224k32,
            .m64n232k32, .m64n240k32, .m64n248k32, .m64n256k32};
.atype  = {.e4m3, .e5m2};
.btype  = {.e4m3, .e5m2};
.dtype  = {.f16, .f32};
```

Sparse WGMMA

This is a special type of WGMMA operation where tensor cores do the same MMA math but the difference is that A is structured sparse (50% zeros).

```
wgmma.mma_async.sp.sync.aligned.shape.dtype.f16.f16  d, a-desc, b-desc, sp-meta, sp-sel, scale-d, imm-scale-a, imm-scale-b, imm-trans-a, imm-trans-b;

wgmma.mma_async.sp.sync.aligned.shape.dtype.f16.f16  d, a, b-desc, sp-meta, sp-sel, scale-d, imm-scale-a, imm-scale-b, imm-trans-b;
```

`descA` (or register fragments for A, depending on RS vs SS form), `descB`(for B)

`sp-meta` (a `.b32` register containing packed indices),

`sp-sel` (a 32b constant, the “sparsity selector”, choosing which threads contribute metadata for a group)

And other operands...

Some important points

The hardware strictly operates as Sparse A × Dense B. Even if Matrix B is mathematically sparse (contains zeros), the Tensor Core treats it as a dense matrix.

We need to create a register named sp_meta for every single thread. You only make sp_meta meaningful in the lanes that HW will read, and in the other lanes you typically set sp_meta = 0 (or anything) because those lanes are ignored for metadata (for these shapes).

```
int lane = laneid();        // 0..31
int q = lane & 3;           // 0,1,2,3  (position within each 4-lane group)

bool meta_lane = (sp_sel == 0) ? (q < 2) : (q >= 2);
uint32_t sp_meta = meta_lane ? real_meta_for_this_lane() : 0;
```

Packing and metadata

The physical rule of NVIDIA's sparse tensor core is 2:4 structured sparsity. We provide a contiguous vector of 4 values called Quartet(I didn’t made this up). You must delete exactly 2 of them and store the two survivors(yes this must happen in global memory or with some transformation in shared memory)

This saves 50% of the bandwidth and storage. If you hand the GPU [8.5, 3.2], it has no idea where they belong. Did 8.5 come from index 0, 1, or 2? This is where Metadata enters.

Since we are selecting 2 indices out of 4 possible positions (0, 1, 2, 3), we need to encode the positions of the survivors.

sp-sel

spSel tells the hardware who is responsible for providing metadata

In other words, which thread-pair inside each group of 4 consecutive threads (T0–T3) is considered the metadata contributor for cases where only a pair contributes.

TF32 sparse (.m64nNk16 .tf32): spSel must be 0 (threads T0,T1) or 1 (threads T2,T3).

FP16/BF16 sparse (.m64nNk32 .f16/.bf16): spSel must be 0 (T0,T1) or 1 (T2,T3).

FP8 / INT8 sparse (.m64nNk64 .e4m3/.e5m2/.s8/.u8):

all threads contribute metadata, so spSel must be 0 (anything else is undefined).

Configuring sp-sel (The Thread Selector)

Selecting 0 tells the hardware to read metadata registers from threads T0 and T1, ignoring T2 and T3.

Selecting 1 tells the hardware to read metadata registers from threads T2 and T3, ignoring T0 and T1.

Selection Rule 1 (Replicated): If your loader broadcasts metadata to all threads (common), always choose sp-sel=0 for simplicity.

Selection Rule 2 (Sharded): If you split metadata to save registers, you must dynamically match sp-sel to the thread holding the data.

Incorrect configuration causes the Tensor Core to read from empty registers, treating the block as dense or zeroed out.

sp-meta

It tells where are the nonzeros in A

spMeta is a packed bitfield. The exact mapping is shown in the PTX figures, but the rules are:

2:4 structured sparsity (FP16/BF16, FP8, INT8): A is sparse at 2 nonzeros per 4 adjacent elements in each row. Only the two nonzeros are stored, and their positions (0..3) are encoded by two 2-bit indices in the metadata operand.

1:2 structured sparsity (TF32): A is sparse at 1 nonzero per 2 adjacent elements. metadata uses a 4-bit index to indicate which of the 2 positions is present, and only two specific bit-patterns are meaningfu; other values are undefined.

Configuring sp-meta for FP16/BF16/INT8

For standard precisions, a single register load packs enough metadata for exactly four consecutive WGMMA math instructions.

Selection Rule: You must unroll your main loop 4 times and cycle sp-meta through indices 0, 1, 2, 3.

Use sp-meta=0 for the first K-tile, sp-meta=1 for the second, and so on, until the buffer is consumed.

This index increments the internal hardware pointer to the next set of 2-bit indices within the loaded register.

Failing to increment this value forces the hardware to reuse the sparsity pattern of the first tile for all calculations.

Configuring sp-meta for TF32 (The Exception)

TF32 metadata requires a unique selection strategy because the register only holds enough information for two WGMMA instructions.

You strictly cannot use sequential indices (0, 1); you must toggle between the hardcoded hardware bitmasks 14 and 4.

Step 1: For the first K=16 calculation, you must use sp-meta = 0b1110 (14) to decode the lower bits.

Step 2: For the second K=16 calculation, you must use sp-meta = 0b0100 (4) to decode the upper bits.

These specific values are required to align the hardware swizzler with the non-standard 19-bit/32-bit TF32 data format.

Using standard indices like 0 or 1 for TF32 will result in a hardware misalignment and incorrect matrix output.

Valid values of sp-sel and sp-meta

Matrix shape	.atype	Valid values of sp-meta	Valid values of sp-sel
.m64nNk16	.tf32	0b1110 , 0b0100	0 (threads T0, T1) or 1 (threads T2, T3)
.m64nNk32	.f16 / .bf16	0b00, 0b01, 0b10, 0b11	0 (threads T0, T1) or 1 (threads T2, T3)
.m64nNk64	.e4m3 / .e5m2 / .s8 / .u8	0b00, 0b01, 0b10, 0b11	0 (all threads contribute)

Different versions of sparse wgmma