Introduction to WGMMA

Prateek Shukla

4 important innovations in mma in hopper

Apart from having faster tensor cores these are other important innovations:

1. The tensor core operations are now fully asynchronous. The math instruction is now non blocking. This enables multiple in flight ops which means that tensor cores would be active for more time
2. We have now transitioned from using single warps for tensor core operations to using warpgroups which allow the use of much larger tiles
3. The ability to fetch the data directly from shared memory/registers for tiles using asynchronous hardware. In ampere threads could not launch mma.sync operation until the data arrived in the registers. In wgmma we can skip loading registers and directly launch math operations
4. Specialized hardware support for fp8 and sparsity

The paradigm shift: warp vs warp group

In Hopper we have transitioned from having a decade long emphasis on warp as the unit of execution, we changed that to warp group

When you call wgmma you are fusing 4 warps to make a single computational entity which operates on tensor cores. This maximizes the utilization of tensor cores and avoids the hassle of pipelining multiple `mma.sync` operations.

All the 4 warps launch the wgmma instruction together. The warp scheduler checks if all the warps converge at that instruction in PC and then scheduler fuses them and then dispatches the math command to the tensor cores

Once the command is sent to the tensor core the threads go on to work on next instruction.

WGMMA pipeline

Load the data from tiles asynchronously, pre-load the next buffer if nessecery

If loading A tiles into registers then use `ldmatrix` and a bunch of complicated addressing, use the wgmma descriptor for loading b tiles

If not just create a descriptor for A and B tiles.

Launch the wgmma operation.

Collect the results from registers. Launch `wgmma.fence.sync.align` to make the writes to registers visible do some complicated address calculations to move the data to shared memory.

`wgmma.fence.sync.aligned`

wgmma.fence is a register-ordering barrier for `wgmma.mma_async`, not a completion barrier. Its job is to make prior warpgroup register accesses visible in the right order before a later `wgmma.mma_async` uses those same registers.

You need it in two main cases: before the first `wgmma.mma_async` in a warpgroup, and any time a thread has accessed registers that a later `wgmma.mma_async` will reuse as accumulators or A-fragment inputs

What it does not do is order shared-memory writes for the matrix descriptors/data consumed by `wgmma.mma_async`. For that, the PTX docs require an async proxy fence such as `fence.proxy.async` to order prior shared-memory writes against later `wgmma.mma_async` reads

For `wgmma.mma_async.m64n16k16.f32.f16.f16`:

$$D^{f32}_{64\times16} = A^{f16}_{64\times16} \times B^{f16}_{16\times16} + C^{f32}_{64\times16}$$

Element-wise

Each output element is a dot product:

$$D_{i,j} = \sum_{k=0}^{K-1} A_{i,k} \cdot B_{k,j} + C_{i,j}$$

wgmma.mma_async

The core async instruction which tells the tensor cores to do $D = A \times B + C$

A is read directly from shared memory/registers

B is read directly from shared memory

C, D reside in registers

The whole operation looks like this

```
wgmma.mma_async.sync.aligned.shape.dtype.bf16.bf16  d, a-desc, b-desc, scale-d,
imm-scale-a, imm-scale-b, imm-trans-a, imm-trans-b;
```

`.sync` and `.aligned`

`.sync` - this is the SM wide barrier. It mandates that all participating threads in the warpgroup (typically 128 threads, or 4 warps) must reach this instruction before any of them can proceed.

`.aligned` - This qualifier asserts that all threads in the warpgroup are converged (executing in lockstep) at the instruction's memory address (Program Counter) and there is no thread divergence between the warp group

```
wgmma.mma_async.sync.aligned.shape.dtype_d.dtype_a.dtype_b
    {d0, d1, ..., dn},          // Destination registers (accumulator)
    desc_a,                      // Matrix A descriptor (or registers)
    desc_b,                      // Matrix B descriptor
    scale_d,                     // Enable accumulation (0 or 1)
    imm_scale_a,                 // Scale A (1, -1)
    imm_scale_b,                 // Scale B (1, -1)
    imm_trans_a,                 // Transpose A (0 or 1)
    imm_trans_b;                 // Transpose B (0 or 1)
```

The instruction qualifiers

Shape - the shape of your tile, we represent them using m64, n{}, k{}

dtypeD, dtypeA, dtypeB - the datatypes of D, A, B tensors

m64nXkY

For all wgmma instructions, the M dimension is fixed at 64.

The K dimension (inner product dimension) is determined strictly by the precision of the input matrices A and B.

The N dimension is the most flexible. It determines how many columns of matrix B (and C) are processed.

N must be a multiple of the underlying memory allocation block size (typically multiples of 8 or 16 depending on the type).

Valid N values range from 8 up to 256.

The fixed m

The hardware is designed to hold 4 registers per thread. This means that each warpgroup can hold 128 * 4 = 512 registers

This means that if we use the smallest tile m = 64, n = 8 we get 64 * 8 = 512 elements which is exactly the capacity of a warpgroup

The hardware physically maps the data from A to tensor core input B. At M = 64, The hardware has a perfect, static map. This requires zero decision logic at runtime.

If we have m configurable then you need a massive Crossbar Switch (a complex multiplexer) between the registers and the math units to dynamically reroute the data based on your requested which is expensive

Operand location

Matrix A - registers/shared memory

Matrix B - shared memory

Matrix C/D - Registers only

scale_d

Just specifies if we should have the accumulate(D = A x B + C) or overwrite(D = A x B) operation

Scale_d = 1 - accumulate

Scale_d = 0 - overwrite

scale_a/b

These are the scaling factors to scale the elements of A/B by 1 or -1

The register operand d

These are the registers in which the outputs are gonna be stored

In the PTX instruction string, operand d is the first operand. It must be declared as a vector (tuple) of registers enclosed in braces {}.

Number of output operands can be calculated by -

Total elements in the output / total number of threads

```
asm volatile(
  "wgmma.mma_async.sync.aligned.m64n16k16.f32.f16.f16 "
  "{%0, %1, %2, %3, %4, %5, %6, %7}, "  // Operand d (Accumulator): 8 registers
  "{%8, %9, %10, %11}, "                // Operand a
  "%12, "                               // Operand b (descriptor)
  ...
  : "+f"(d0), "+f"(d1), "+f"(d2), "+f"(d3), "+f"(d4), "+f"(d5), "+f"(d6), "+f"(d7)
  : ...
);
```

The location of A

The tiles of A can either reside in shared memory or registers. And thus there are two ways to tell wgmma where A tiles resides

a. Registers: If the tiles reside in the registers then we pass in the registers to the wgmma instructions
b. Shared memory: We pass in the wgmma descriptor

A in registers

This is a good approach when you are reusing the data because loading A and B from shared memory doubles the pressure on the banks, multiple writes into shared memory would be costly therefore better to load from registers.

Moving data from shared memory is expensive. Moving data from registers is cheap. Loading from registers keeps the reusable data where it can be accessed faster.

The data for registers comes from shared memory, the instruction `ldmatrix` takes the unswizzled addresses and loads the data into registers

If you are not reusing the data then loading A from registers does not make sense because there is instruction overhead and register pressure

Mechanism for loading data from registers

Every thread provides the pointer for loading the data into registers and the ldmatrix instruction populate the registers of the target threads with the data

```
ldmatrix.sync.aligned.m8n8.x1.shared::cta.b16 {d}, [addr];
```

```
ldmatrix.sync.aligned.m8n8.x2.trans.shared.b16 {d0, d1}, [addr];
```

```
ldmatrix.sync.aligned.m8n8.x4.b16 {d0, d1, d2, d3}, [addr];
```

Only for blackwell -

```
ldmatrix.sync.aligned.m16n16.x1.trans.shared.b8 {d0, d1}, [addr];
```

```
ldmatrix.sync.aligned.m16n16.x2.trans.shared::cta.b8 {d0, d1, d2, d3}, [addr];
```

```
ldmatrix.sync.aligned.m16n16.x2.trans.shared::cta.b8x16.b6x16_p32 {d0, d1, d2, d3}, [addr];
```

`ldmatrix`

It is the primary mechanism for populating registers with Matrix A data in the WGMMA pipeline.

Unlike a standard `ld.shared` (which loads linear data), ldmatrix loads data in opaque register patterns designed to align physically with the Tensor Core's input lanes

The threads which point to the data are sometimes not the threads which end up with the data.

`.m8n8` is the geometric shape of the matrix tile being loaded from Shared Memory into the Warp's registers.

The qualifiers .x1, .x2, and .x4 dictate the vectorization width and the number of matrix fragments loaded per instruction.

The x{1, 2, 4}

.x{1, 2, 4} tells us what number of core matrices of size 8 x 8 we can move to registers for the computation

.x4 - This means loading 4 registers per thread or loading 4 core matrices of 8x8 dimension Most common because it saturates the register bandwidth by loading 16 * 16 * 4 = 1024 bytes of data from shared memory to registers

.x2 - Loading 512 bytes of data i.e loading 2 registers per thread. Generally use it for loading smaller tiles

.x1 - 1 register per thread, mostly for boundary handling and other works.

How a 16x16 tile is loaded into registers from shared memory

Whole operation works in two phases:

The Address Phase: Who provides the pointer? (The "Gather")

The Register Phase: Who holds the result? (The "Destination")

With x4 we are working with 4 section of 8x8 tiles

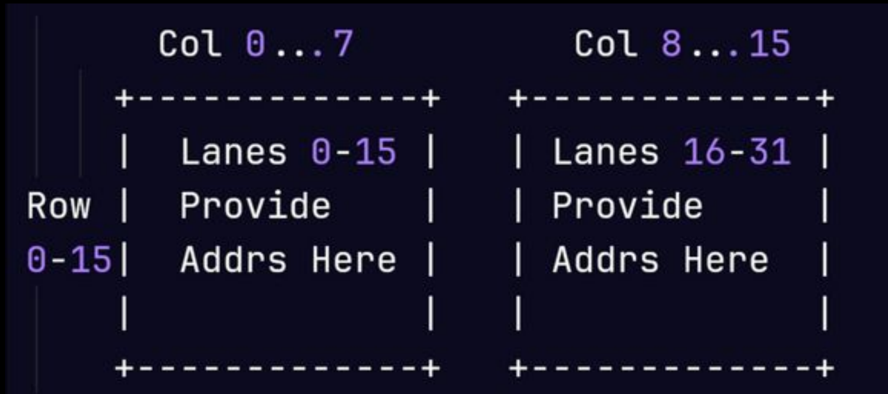
Quadrant	Matrix ID	Global Rows	Global Cols
Top-Left	**M0**	0 - 7	0 - 7
Top-Right	**M1**	0 - 7	8 - 15
Bottom-Left	**M2**	8 - 15	0 - 7
Bottom-Right	**M3**	8 - 15	8 - 15

The "Split-Warp" Strategy

We don't calculate addresses linearly. We split the warp into two "Vertical Strips" (Left/Right) because ldmatrix loads 8-column chunks.

Left Half (Cols 0-7): This is controlled by the first 16 threads (Lanes 0-15).

Right Half (Cols 8-15): This is controlled by the next 16 threads (Lanes 16-31).



Phase 1: Address Responsibility (which thread points)

In this phase, the hardware looks at the address register in each thread to decide where to read from Shared Memory. The warp is split into 4 groups of 8 threads.

```
Threads 0-7:   provide addresses for M0 (Top-Left)     → rows 0-7,  cols 0-7

Threads 8-15:  provide addresses for M2 (Bottom-Left)  → rows 8-15, cols 0-7

Threads 16-23: provide addresses for M1 (Top-Right)    → rows 0-7,  cols 8-15

Threads 24-31: provide addresses for M3 (Bottom-Right) → rows 8-15, cols 8-15
```

* @param tile_row_offset Global row offset of this tile in the larger matrix

* @param tile_col_offset Global col offset of this tile in the larger matrix

Phase 2: Register Responsibility (who holds the data)

Once the data is fetched, ldmatrix distributes it across the warp so that it is ready for Tensor Core math. The distribution pattern is Row-Striped across Groups of 4 Threads.

For ANY 8x8 matrix (M0, M1, M2, or M3), the rows are distributed as follows:

Row 0 goes to Threads 0-3

Row 1 goes to Threads 4-7

Row 2 goes to Threads 8-11

...

Row 7 goes to Threads 28-31

Layout of a single core matrix

Row\Col	0	1	2	3	4	5	6	7
0	T0:r	T0:r	T1:r	T1:r	T2:r	T2:r	T3:r	T3:r
1	T4:r	T4:r	T5:r	T5:r	T6:r	T6:r	T7:r	T7:r
2	T8:r	T8:r	T9:r	T9:r	T10:r	T10:r	T11:r	T11:r
3	T12:r	T12:r	T13:r	T13:r	T14:r	T14:r	T15:r	T15:r
4	T16:r	T16:r	T17:r	T17:r	T18:r	T18:r	T19:r	T19:r
5	T20:r	T20:r	T21:r	T21:r	T22:r	T22:r	T23:r	T23:r
6	T24:r	T24:r	T25:r	T25:r	T26:r	T26:r	T27:r	T27:r
7	T28:r	T28:r	T29:r	T29:r	T30:r	T30:r	T31:r	T31:r

The address calculation

The tile lives in shared memory in a swizzled layout (chosen to avoid bank conflicts), so the bytes for a “logical row” are not stored contiguously in physical address order.

Each thread starts from its logical coordinate (e.g., “I need row r, column block c”).

It then computes the physical shared-memory address for that fragment using the swizzle mapping (often an XOR-based remap)

smem_addr = base + (row_offset ^ swizzle_mask) + col_offset

That computed address is the pointer the thread provides to ldmatrix.

Because the pointers are swizzle-mapped, the accesses land in different banks, so the warp hits minimal/zero bank conflicts.

ldmatrix loads 16 bytes per thread and reorders it in fragmented layout

Address bits breakdown

To understand how swizzle formula works we need to understand the address layout

Bits [0-1]: Byte offset within a 4-byte word. Irrelevant for bank selection.

Bits [2-6]: The Bank Index. These 5 bits determine which of the 32 banks (0-31) the data lands in.

Specifically, bits [4-6] control the upper 3 bits of the bank index.

Bits [7-9]: The Row Index. Since the pitch is 128 bytes (2^7), bit 7 is the first bit that changes when you move to the next row.

We want the Bank (controlled by bits 4-6) to change based on the Row (controlled by bits 7-9).

Swizzled address calculation

```
Physical_Address = (Base_Address + Linear_Offset) ^ Swizzle_Mask;
```

Where $$\text{Swizzle_Mask} = \left(\frac{\text{Linear_Offset}}{128} \,\&\, 7\right) \times 16$$

$$\text{Linear_Offset} = (\text{Row} \times \text{Stride_in_Bytes}) + (\text{Col} \times \text{Bytes_Per_Element})$$

```
// 128 bytes = 2^7. We take bits [7,8,9], and shift them to positions [4,5,6].
uint32_t mask = ((linear_offset >> 7) & 0x7) << 4;
// Final Pointer
uint32_t smem_ptr = (base_ptr + linear_offset) ^ mask;
```

Packing

The NVIDIA GPU architecture does not have 16-bit registers; it uses 32-bit registers. Therefore, bf16 data must always be packed in pairs when residing in registers.

Container: One .b32 (32-bit) register holds two bf16 elements.

Layout:

- Bits 0-15 (LO): Element N (Even index)
- Bits 16-31 (HI): Element N+1 (Odd index)

When moving this data, you often use .b32 type instructions. When computing, you use .bf16x2.

ldmatrix packing

If you use `ldmatrix.sync.aligned.m8n8.x1.b16`, each thread receives one 32-bit register.

This register contains two bf16 elements packed as described above.

If you use .x2 or .x4, you get 2 or 4 registers, each containing a packed bf16 pair.

The address in shared memory must be 16-byte (128-bit) aligned.

Using `.trans` allows you to load Row-Major data as Column-Major (or vice-versa) directly into registers without manual shuffling.

```
lane    = threadIdx.x & 31            // 0..31 within warp
warp    = threadIdx.x >> 5            // warp id within CTA

wg_id   = warp >> 2                   // warpgroup id (each has 4 warps)
w_in_wg = warp & 3                    // 0..3 warp index inside warpgroup


m_base = m_block + w_in_wg * 16       // each warp handles 16 rows of the m64 tile
k0     = k_block                      // start column for k16 window (0..15)

// ThunderKittens ldmatrix layout within the warp:
row_in_16 = lane & 15                 // 0..15 (row within the warp's 16-row slice)
col_half  = lane >> 4                 // 0 for lanes 0..15, 1 for lanes 16..31
col_in_16 = col_half * 8              // 0 or 8 (left/right 8 cols of the 16-col window)

// Each lane provides a row-start pointer for an 8-wide segment (16B for bf16/f16):
logical_row_start_ptr = &A[m_base + row_in_16][k0 + col_in_16]

// Apply the shared swizzle (your idx())
addr = swizzle_idx(shared_base, {m_base + row_in_16, k0 + col_in_16})
// or: addr = src.idx(shared_addr, {row, col}) if you're using TK's src.idx()

(d0,d1,d2,d3) = ldmatrix.sync.aligned.m8n8.x4.shared.b16(addr)
```

`.sync.aligned` and `.trans`

.sync: It acts as a mini-barrier. The hardware needs to guarantee that T16 is ready to have its register overwritten by data requested by T0. If T16 was busy doing something else, the hardware writer would corrupt T16's state

.aligned: All threads must execute it together so the hardware knows the Warp is ready.

.trans: this tells whether to transpose the matrix while loading the data.

Because the registers of a thread is not visible to others, this whole instruction is carried out by hardware internally.

How they plug into wgmma instructions

When wgmma reads the registers `{ra0, ra1, ra2, ra3}` from a specific thread (say, Thread 0 of Warp 0), it expects them to contain specific fragments of the matrix.

Crucially, **wgmma is hardwired to expect the EXACT layout that ldmatrix produces.**

The Tensor Core hardware unwires these registers and re-maps them to the physical dot-product units.

Instruction: `wgmma... {acc}, {ra0...ra3}, desc_b …`

Hardware grabs `{ra0...ra3}` from Warp 0. It knows this is Rows 0-15 and others from the warpgroups and executes wgmma with data from B.

The wgmma descriptor

This is the 64 bit descriptor which contains information about the data stored, strides and swizzle layouts

The descriptor packs 5 distinct fields. Consisting of the address, LBO, SBO, Matrix base offset and the swizzle layout

Bit-field	Size in bits	Description
13–0	14	matrix-descriptor-encode(Matrix start address)
29–16	14	matrix-descriptor-encode (Leading dimension byte offset)
45–32	14	matrix-descriptor-encode (Stride dimension byte offset)
51–49	3	Matrix base offset. This is valid for all swizzling modes except the no-swizzle mode.
63–62	2	Specifies the swizzling mode to be used: › 0: No swizzle › 1: 128-Byte swizzle › 2: 64-Byte swizzle › 3: 32-Byte swizzle

The blueprint

A single 64-bit register encodes the base address, K stride, M/N stride, matrix base offset and swizzle mode. This compact structure allows efficient memory access patterns for tensor operations.

Every stride is pre-divided by 16 to fit into the 14-bit fields(last 4 bits are useless because the shared memory pointer is 16 byte aligned). The hardware multiplies by 16 at runtime, ensuring 16-byte alignment and reducing address latency, crucial for high-performance tensor-core operations.

The base address must be 16-byte aligned. This alignment ensures that the preprocessed strides can be correctly interpreted by the hardware, maintaining efficient memory access and avoiding performance penalties.

The base shared memory address

The base address is the first 14 bits of the descriptor. This is the shared memory address of the tile which we are gonna load.

We extract the 14 bits from the address and then use them as the 14 bits of the descriptor

```
uint32_t smem_offset;

asm volatile (
    "{ .reg .u64 %addr_val; "
    "  cvta.to.shared.u64 %addr_val, %1; "
    "  cvt.u32.u64 %0, %addr_val; }"
    : "=r"(smem_offset)
    : "l"(ptr)
);
uint64_t encoded_addr = (uint64_t)(smem_offset >> 4) & 0x3FFF;
```

Leading byte offset (LBO)

the byte distance in shared memory between two core-matrix columns that are adjacent along the operand’s “leading” direction

LBO occupies bits 16-29. To encode the K stride, divide the desired byte stride by 16, mask to 14 bits, then shift left by 16

LBO for K major swizzled layouts is not used. Hardware assume its 1

For MN Major, LBO = offset from the first (swizzle-byte-size/16) rows to the next (swizzle-byte-size/16) rows

For 128B swizzle, swizzle-byte-size/16 = 128/16 = 8, so it’s “jump by 8 rows” in that normalized layout.

```
uint64_t lbo = (k_stride >> 4) & 0x3FFF;
descriptor |= lbo << 16;
```

Stride Byte Offset (SBO)

It's the other stride WGMMA uses (along with LBO) to navigate a matrix operand in shared memory. SBO is the byte distance in SMEM to jump "one core-matrix block" in the other (non-leading) direction.

SBO occupies bits 32-45. Encode the M/N stride similarly:

For K major swizzled layout, SBO is offset from the first 8 rows to the next 8 rows

For MN Major swizzled layout, SBO is offset from the first 8 columns to the next 8 columns

```
uint64_t sbo = (mn_stride >> 4) & 0x3FFF;
descriptor |= sbo << 32;
```

Matrix base offset

When SMEM is swizzled (e.g., 128B swizzle), the mapping from a logical address to the physical SMEM bank/line uses a repeating permutation pattern. That pattern repeats every 128B for a 128B swizzle pattern

If your matrix start address is exactly at the “pattern start” boundary for that swizzle, then base_offset = 0. Otherwise, you must tell hardware which 128B chunk within the repeating region you’re starting in. That’s what the base offset encodes.

Swizzle mode bits

If the data is the shared memory is swizzled then we need to apply the swizzled layouts to the tensors to read the data correctly

If we don’t set the correct swizzle layout correctly then wgmma units read the scrambled data linearly

We have 4 swizzle modes, bits 61-63 control the swizzle mode. Use 00 for no swizzle 01 for 128B, 10 for 64B, 11 for 32B

```
uint64_t swz = 3;
descriptor |= swz << 61
```

The whole descriptor function

```
constexpr uint64_t SWIZZLE_128B = 1;

__device__ uint64_t make_wgmma_desc(void const* smem_ptr, uint64_t leading_dim_bytes, uint64_t stride_dim_bytes) {
    // 1. Downcast offset using CVTA (Safe)
    uint32_t smem_offset = cast_smem_ptr_to_uint(smem_ptr);

    // 3. Construct Descriptor
    // Address is shifted by 4 (>> 4) because the hardware expects 16-byte chunks
    uint64_t desc = 0;

    // Bits 0-13: Start Address (14 bits)
    desc |= ((uint64_t)smem_offset >> 4) & 0x3FFF;

    // Bits 16-29: Leading Dimension Offset (14 bits)
    // Stride between rows (or columns depending on layout), divided by 16
    desc |= ((leading_dim_bytes >> 4) & 0x3FFF) << 16;

    // Bits 32-45: Stride Dimension Offset (14 bits)
    // Stride between matrices/batches, divided by 16
    desc |= ((stride_dim_bytes >> 4) & 0x3FFF) << 32;

    // Bits 62-63: Swizzle Mode
    // Set to 1 for 128-byte swizzle
    desc |= (SWIZZLE_128B << 62);

    return desc;
}
```

```
asm volatile(
    "{\n"
    "wgmma.mma_async.sync.aligned.m64n128k16.f32.bf16.bf16 "
    "{%0,   %1,   %2,   %3,   %4,   %5,   %6,   %7,   "
    " %8,   %9,   %10,  %11,  %12,  %13,  %14,  %15,  "
    " %16,  %17,  %18,  %19,  %20,  %21,  %22,  %23,  "
    " %24,  %25,  %26,  %27,  %28,  %29,  %30,  %31,  "
    " %32,  %33,  %34,  %35,  %36,  %37,  %38,  %39,  "
    " %40,  %41,  %42,  %43,  %44,  %45,  %46,  %47,  "
    " %48,  %49,  %50,  %51,  %52,  %53,  %54,  %55,  "
    " %56,  %57,  %58,  %59,  %60,  %61,  %62,  %63},"
    " %64,"
    " %65,"
    " %66,    %67,  %68,  %69,  %70;\n"
    "}\n"
    : "+f"(d[0][0]), "+f"(d[0][1]), "+f"(d[0][2]), "+f"(d[0][3]), "+f"(d[0][4]), "+f"(d[0][5]),
      "+f"(d[0][6]), "+f"(d[0][7]), "+f"(d[1][0]), "+f"(d[1][1]), "+f"(d[1][2]), "+f"(d[1][3]),
      "+f"(d[1][4]), "+f"(d[1][5]), "+f"(d[1][6]), "+f"(d[1][7]), "+f"(d[2][0]), "+f"(d[2][1]),
      "+f"(d[2][2]), "+f"(d[2][3]), "+f"(d[2][4]), "+f"(d[2][5]), "+f"(d[2][6]), "+f"(d[2][7]),
      "+f"(d[3][0]), "+f"(d[3][1]), "+f"(d[3][2]), "+f"(d[3][3]), "+f"(d[3][4]), "+f"(d[3][5]),
      "+f"(d[3][6]), "+f"(d[3][7]), "+f"(d[4][0]), "+f"(d[4][1]), "+f"(d[4][2]), "+f"(d[4][3]),
      "+f"(d[4][4]), "+f"(d[4][5]), "+f"(d[4][6]), "+f"(d[4][7]), "+f"(d[5][0]), "+f"(d[5][1]),
      "+f"(d[5][2]), "+f"(d[5][3]), "+f"(d[5][4]), "+f"(d[5][5]), "+f"(d[5][6]), "+f"(d[5][7]),
      "+f"(d[6][0]), "+f"(d[6][1]), "+f"(d[6][2]), "+f"(d[6][3]), "+f"(d[6][4]), "+f"(d[6][5]),
      "+f"(d[6][6]), "+f"(d[6][7]), "+f"(d[7][0]), "+f"(d[7][1]), "+f"(d[7][2]), "+f"(d[7][3]),
      "+f"(d[7][4]), "+f"(d[7][5]), "+f"(d[7][6]), "+f"(d[7][7])
    : "l"(desc_a), "l"(desc_b), "n"(int32_t(ScaleD)), "n"(int32_t(ScaleA)),
      "n"(int32_t(ScaleB)), "n"(int32_t(TransA)), "n"(int32_t(TransB)));
```

What does the wgmma.mma_aync operation does

It asynchronously moves the data from your shared memory/registers to the tensor cores for the operation. All the threads in the consumer warp group are calling wgmma.mma_async

The tensor cores start computing on the data. Then result is going to be written back into the registers of consumer warpgroup based on the warpId, laneId. Its low overhead the hardware is dumping the data to the closes memory

The hardware wants to write results to the threads that are physically closest to the math unit responsible for that chunk. So the threads are interleaved into small blocks where each thread holds non contiguous data in memory

A thread is getting all those registers in output

If we are working with wgmma with m64n256k16 then each thread has 128 elements in its accumulators. Every single WGMMA instruction is launched by every single thread in the warpgroup!

For 128 threads in warpgroup you need, 16 stmatrix instructions with .x4 if accumulators are 16 bit and 32 stmatrix if 32 bit accumulators

In the case of kernels like flashattention 3 one should be careful about the register pressure by the softmax gemm pipeline if using registers for loading the A tiles.

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

.shape   = {.m64n8k16, .m64n16k16, .m64n24k16, .m64n32k16,
            .m64n40k16, .m64n48k16, .m64n56k16, .m64n64k16,
            .m64n72k16, .m64n80k16, .m64n88k16, .m64n96k16,
            .m64n104k16, .m64n112k16, .m64n120k16, .m64n128k16,
            .m64n136k16, .m64n144k16, .m64n152k16, .m64n160k16,
            .m64n168k16, .m64n176k16, .m64n184k16, .m64n192k16,
            .m64n200k16, .m64n208k16, .m64n216k16, .m64n224k16,
            .m64n232k16, .m64n240k16, .m64n248k16, .m64n256k16};
.dtype   = {.f16, .f32};
```

imm-scale-{a,b} - option to “negate” A, this means A becomes -A

This can have two values 1 or -1, no other values because this transformation just requires switching a bit instead of multiple bits

Imm-trans-{a,b} - this is option to transpose the tensor

This can take values 0 or 1

These are stored in bits because aren't enough bits left in the instruction string to store a full 16-bit floating point number

```
wgmma.mma_async.sync.aligned.shape.dtype.bf16.bf16  d, a-desc, b-desc, scale-d, imm-scale-a, imm-scale-b, imm-trans-a, imm-trans-b;

wgmma.mma_async.sync.aligned.shape.dtype.bf16.bf16  d, a, b-desc, scale-d, imm-scale-a, imm-scale-b, imm-trans-b;

.shape   = {.m64n8k16, .m64n16k16, .m64n24k16, .m64n32k16,
            .m64n40k16, .m64n48k16, .m64n56k16, .m64n64k16,
            .m64n72k16, .m64n80k16, .m64n88k16, .m64n96k16,
            .m64n104k16, .m64n112k16, .m64n120k16, .m64n128k16,
            .m64n136k16, .m64n144k16, .m64n152k16, .m64n160k16,
            .m64n168k16, .m64n176k16, .m64n184k16, .m64n192k16,
            .m64n200k16, .m64n208k16, .m64n216k16, .m64n224k16,
            .m64n232k16, .m64n240k16, .m64n248k16, .m64n256k16};
.dtype  = {.f32};
```

```
wgmma.mma_async.sync.aligned.shape.dtype.tf32.tf32  d, a-desc, b-desc, scale-d, imm-scale-a, imm-scale-b;

wgmma.mma_async.sync.aligned.shape.dtype.tf32.tf32  d, a, b-desc, scale-d, imm-scale-a, imm-scale-b;

.shape   = {.m64n8k8, .m64n16k8, .m64n24k8, .m64n32k8,
            .m64n40k8, .m64n48k8, .m64n56k8, .m64n64k8,
            .m64n72k8, .m64n80k8, .m64n88k8, .m64n96k8,
            .m64n104k8, .m64n112k8, .m64n120k8, .m64n128k8,
            .m64n136k8, .m64n144k8, .m64n152k8, .m64n160k8,
            .m64n168k8, .m64n176k8, .m64n184k8, .m64n192k8,
            .m64n200k8, .m64n208k8, .m64n216k8, .m64n224k8,
            .m64n232k8, .m64n240k8, .m64n248k8, .m64n256k8};
.dtype  = {.f32};
```

```
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};
```

```
wgmma.mma_async.sync.aligned.shape{.satfinite}.s32.atype.btype  d, a-desc, b-desc, scale-d;

wgmma.mma_async.sync.aligned.shape{.satfinite}.s32.atype.btype  d, a, b-desc, scale-d;

.shape   = {.m64n8k32, .m64n16k32, .m64n24k32, .m64n32k32,
            .m64n48k32, .m64n64k32, .m64n80k32, .m64n96k32,
            .m64n112k32, .m64n128k32, .m64n144k32, .m64n160k32,
            .m64n176k32, .m64n192k32, .m64n208k32, .m64n224k32};
.atype  = {.s8, .u8};
.btype  = {.s8, .u8};
```

bye