cuTensorMap

Prateek Shukla

What is TMA

TMA is the new unit which enables fully asynchronous data movements with minimal thread involvement

A single thread launches TMA instructions and immediately continues execution, the whole operation is handled by the hardware in the background

Instead of address calculation you get descriptors

TMA can transfer data to shared memory of multiple SMs simultaneously

Can handle 1d-5d tensors

How does H100 gets the perfect asynchronicity

We need all the units to be busy all the time

H100 gets it right by having multiple buffers loading using TMA and having tensor cores do the operations simultaneously

The more important point is that you don’t need a lot of complex engineering. Descriptors handle everything from data to layout to optimizations, leaving a lot of the heavy lifting to the hardware

Why do we need Descriptors

Descriptors are the piece of the puzzle which enable the asynchronous nature of H100.

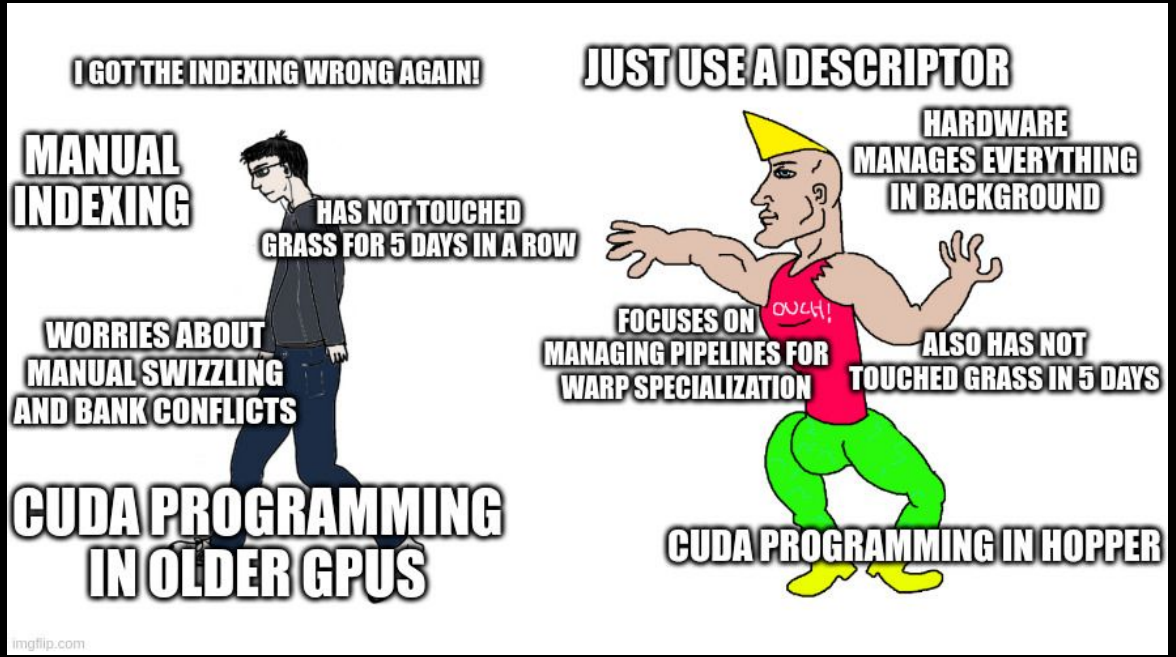
In earlier times the whole fetching of data would happen using indexing where threads would need to compute the address of the data which they wanted to fetch and then data was fetched.

This was fixed a bit in ampere where the copy instruction was non blocking but threads still had to compute the addresses for every 16 bytes. SMs were tethered to the copy engine.

Now with H100s The descriptor encapsulates the entire transfer state. Because all the information required to complete the job is contained in that 128-byte object in memory, the hardware can do the job in the background without SM’s involvement

How to use TMA for copies

- Use CUDA API to create a cuTensorMap
- Encode it with information required
- Launch the operation



cuTensorMap

It is a descriptor which stores the information about -

- Memory base pointer (device address)
- Tensor shape (number of elements per dimension)
- Strides (in bytes)
- Data type
- Alignment and swizzling
- Memory space (device, host, or unified)
- Order and rank (up to 32 dimensions)
- Optionally: tiling or “interleaved” layouts

Creating a tensormap object

```
CUtensorMap tma_desc;
```

Created in host memory as a local variable, it has a size of 128 bytes and It must be 128B aligned. It is not a regular pointer but a specific data structure created by nvidia

`cuTensorMapEncodeTiled()` function is used to encode the values in the tensormap.

```
CUresult cuTensorMapEncodeTiled(
    CUtensorMap* tensorMap,                // [out] Descriptor to initialize
    CUtensorMapDataType tensor_data_type,  // Data type of elements
    cuuint32_t tensorRank,                 // Number of dimensions (1-5)
    void* globalAddress,                   // Base pointer to global memory
    const cuuint64_t* globalDim,           // Global tensor dimensions
    const cuuint64_t* globalStrides,       // Strides between dimensions (in **bytes**)
    const cuuint32_t* boxDim,              // Shared memory tile dimensions (in **elements**)
    const cuuint32_t* elementStrides,      // Stride within tile (in **element units**)
    CUtensorMapInterleave interleave,      // Interleave pattern for sub-4-byte types
    CUtensorMapSwizzle swizzle,            // Swizzle pattern for bank conflict avoidance
    CUtensorMapL2promotion l2Promotion,    // L2 cache promotion policy
    CUtensorMapFloatOOBfill oobFill        // Out-of-bounds fill behavior
);
```

```
CUtensorMap* tensorMap
```

This is the tensormap object which we created using

`CUtensorMap tma_desc` which is gonna be filled with the encoded descriptor.

```
CUtensorMapDataType tensor_data_type
```

Determines the datatype of the actual data which you are gonna copy from HBM

TMA engine uses this to automatically get memory alignment and transfer size

Datatypes

- `CU_TENSOR_MAP_DATA_TYPE_UINT8` - Unsigned 8-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_UINT16` - Unsigned 16-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_UINT32` - Unsigned 32-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_INT32` - Signed 32-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_UINT64` - Unsigned 64-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_INT64` - Signed 64-bit integer
- `CU_TENSOR_MAP_DATA_TYPE_FLOAT16` - 16-bit floating-point (half precision)
- `CU_TENSOR_MAP_DATA_TYPE_FLOAT32` - 32-bit floating-point (single precision)
- `CU_TENSOR_MAP_DATA_TYPE_FLOAT64` - 64-bit floating-point (double precision)
- `CU_TENSOR_MAP_DATA_TYPE_BFLOAT16` - 16-bit brain floating-point
- `CU_TENSOR_MAP_DATA_TYPE_FLOAT32_FTZ` - 32-bit float with flush-to-zero mode
- `CU_TENSOR_MAP_DATA_TYPE_TFLOAT32` - TensorFloat-32 format
- `CU_TENSOR_MAP_DATA_TYPE_TFLOAT32_FTZ` - TensorFloat-32 with flush-to-zero mode

Flush-to-zero (FTZ) mode is a floating-point arithmetic optimization that handles denormalized (subnormal) numbers by replacing them with zero instead of computing them normally.

Tensor Rank and Global Memory address

- Tensor Rank is number of dimensions of the tensor, not the matrix rank from linear algebra
- Global Memory address is the memory address of the tensor in the HBM

The global address must be 16 byte aligned for efficient memory access patterns in Hardware

Global Dimension (globalDim)

Array that specifies the size of each dimension of the tensor in terms of number of elements (not bytes).

For a r dimension array we arrange data in following way

`globalDim[0]` = innermost dimension

`globalDim[r]` = outermost dimension

Note that every element of this array ranges from 0 to 2 ^ 32 (around 4 billion)

```
const cuuint64_t* globalStrides
```

These are byte strides between elements in each dimension

how many bytes hardware must skip to move from one coordinate to the next along specific dimensions.

The stride for the innermost dimension is implicit based on element size and the size of this array is (`rank - 1`)

`globalStrides[0]` is the byte stride for `globalDim[1]` (second-innermost dim).

`globalStrides[rank-2]` is the byte stride for `globalDim[rank-1]` (outermost dim).

```
const cuuint32_t* elementStrides
```

It is the number of elements you want to skip along each dimension when you are doing the async copy.

It is also an array of size `rank` and have specific requirements -

- It must contain non zero values
- Stride value should be less than or equal to 8
- Array size = `rank`

Also it is calculated in number of elements and not bytes

```
const cuuint32_t* boxDim
```

It is the tile size. It is an array of size `rank` with one entry per tensor dimension matching the tensor rank.

Specifies the size (in elements) of the traversal box the chunk of data transferred per TMA operation from global memory to shared memory

Measured in elements (not bytes), corresponding to the data type being transferred

Inner dimension must be ≤ swizzle size

Shared memory

Shared memory is the on-chip scratchpad attached to an SM. Your thread block gets a region of it, and every thread in that block can read or write any address in that region. Programmatically it looks like one flat address space.

What the hardware does underneath is split that address space into 32 banks. A bank is just one independently serviceable lane of the shared-memory array. The reason there are 32 is simple: a warp has 32 threads, so the ideal case is one thread per bank.

Important point: banks are not 32 separate arrays you index manually. You still just compute an address. The bank is chosen from the address.

Swizzling and bank conflicts

All 32 banks can be accessed simultaneously in one cycle. a warp issues one logical load/store instruction, but the SMEM subsystem may execute it in several rounds:

round 1: one subset of requests from each bank

round 2: the remaining conflicting requests and so on

However, if multiple threads access the same bank, the accesses serialize causing bank conflicts. The normal way to access data in shared memory are strided access patterns this causes in serialization.

1 request → no conflict, 2 requests → 2-way conflict, 4 requests → 4-way conflict, 8 requests → 8-way conflict, 32 requests → worst-case conflict

How swizzling works

It's an address-mapping function. It takes the logical address (the GmemAddress your program thinks it's writing to) and "shuffles" its bits to create a physical address (the SmemAddress where the data actually lands).

We need to make the sequential access patterns and spread them across all the banks to avoid conflicts

To do this we have 3 modes of swizzling 32B, 64B, 128B

The 128B, 64B, and 32B layouts define the span or chunk size of the memory swizzling and, consequently, the size of the memory transactions. Choosing the right layout is a critical performance tuning step based on your application's memory access patterns.

The bank equation

$$\text{bank} = \left\lfloor \frac{a'}{4} \right\rfloor \bmod 32$$

where a' is the actual shared-memory byte address used by hardware. If swizzle is enabled, a' is the swizzled address, not the logical unswizzled one. This 32-bank, 4-byte-step model is the standard way to reason about conflicts and matches the address-bit view you were using earlier

That means bank ID comes from address bits a'[6:2]:

then it repeats every 32 * 4 = 128 bytes.

Swizzle formula

```
a'= a^((a & Y_mask) >> 3)
```

where a is the byte address, Y_mask is 1<< 7 and the low 4 bits a[3:0] are never touched.

32B: a[4]' = a[4] xor a[7]

64B: a[5:4]' = a[5:4] xor a[8:7]

128B: a[6:4]' = a[6:4] xor a[9:7]

the bits that are XORed are the low bits of the swizzle-pattern row index in shared memory, not inherently the row of your mathematical matrix. They become your matrix-row bits only if your layout maps matrix rows onto those shared-memory pattern rows.

Swizzle Span and atom

For SM90, the full swizzle pattern repeats every:

32B swizzle: 256 bytes

64B swizzle: 512 bytes

128B swizzle: 1024 bytes

Each row is 128 byte, each row is split into 8 cells of 16 bytes each. The swizzle permutes those 8 cells from row to row. So `repeat_bytes = 128B * distinct_rows`

A swizzle row is not your matrix row. It is a 128 byte shared memory line. The swizzle only permutes those 16B chunks. Inside the swizzle atom, elements are not randomly shuffled the hardware/layout keeps small contiguous groups intact, it only permutes those groups as units

128B swizzling

Span = 128 Bytes, Atom = 16 Bytes

The hardware thinks of the destination shared-memory region as:

Rows of 128 bytes, each row split into 8 cells of 16 bytes each, cell index inside a row: x = 0..7 row index: y = 0, 1, 2, ...

If the base shared-memory address is aligned to the swizzle's full pattern boundary, the physical destination for logical cell (y, x) is: phys_addr = base + 128*y + 16*x

We apply the swizzle formula like this:

```
a[6:4]' = a[6:4] ^ a[9:7]
```

the whole 128B row is treated as 8 swizzlable 16B cells. Row id modulo 8 selects the permutation, the pattern repeats every 8 rows. Total repeat size = 8 * 128B = 1024B

64B swizzling

64B swizzle operates on groups of 4 16B cells.

cell index inside a row: x = 0..3

row index: y = 0, 1, 2, …

the physical destination for logical cell (y, x) is:

```
phys_addr = base + 128*y + 16*x'
```

```
x' = x ^ (y & 3)
```

a[5:4]' = a[5:4] ^ a[8:7]

The left 64B half of the row is permuted by the row id mod 4, the right 64B half is permuted the same way. The pattern repeats every 4 rows, total repeat size = 4 * 128B = 512B

32B swizzling

32B swizzle operates on pairs of 16B cells.

cell index inside a row: x = 0-1

row index: y = 0, 1, 2, …

the physical destination for logical cell (y, x) is:

phys_addr = base + 128*y + 16*x'

x' = x ^ (y & 1)

```
a4' = a[4] ^ a[7]
```

inside each 32B region, the two 16B cells swap on every other 128B row. The full pattern repeats every 2 rows. since each row is 128B, the repeat period is 256B

Setting up swizzling

```
typedef enum CUtensorMapSwizzle_enum {
            CU_TENSOR_MAP_SWIZZLE_NONE = 0,
            CU_TENSOR_MAP_SWIZZLE_32B,                  // Swizzle 16B chunks within 32B  span
            CU_TENSOR_MAP_SWIZZLE_64B,                  // Swizzle 16B chunks within 64B  span
            CU_TENSOR_MAP_SWIZZLE_128B,                 // Swizzle 16B chunks within 128B span
            CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B,        // Swizzle 32B chunks within 128B span
            CU_TENSOR_MAP_SWIZZLE_128B_ATOM_32B_FLIP_8B, // Swizzle 32B chunks within 128B span,
additionally swap lower 8B with upper 8B within each 16B for every alternate row
            CU_TENSOR_MAP_SWIZZLE_128B_ATOM_64B         // Swizzle 64B chunks within 128B span
        } CUtensorMapSwizzle;
```

Interleaving

Data in Global Memory is not always linear. Libraries like cuDNN primarily use primarily uses the NCHW memory layout for optimal GPU performance in convolutions and other operations.

The `CUtensorMapInterleave` parameter informs the TMA how to decode the address space. It maps the physical, interleaved arrangement in HBM to a logical, linear reconstruction in Shared Memory

The Hopper TMA supports specific interleaving patterns designed for NC/xHWCx layouts:

`CU_TENSOR_MAP_INTERLEAVE_16B`

`CU_TENSOR_MAP_INTERLEAVE_32B`

The two major modes

Layout: NC/8HWC8 (Vector of 8 channels).

Math: 8 channels×2 bytes (FP16)=16 bytes.

Behavior: Data is accessed in 16-byte interleaved chunks

Layout: NC/16HWC16 (Vector of 16 channels).

Math: 16 channels×2 bytes=32 bytes.

Behavior: Data is accessed in 32-byte interleaved chunks.

Note: If the channel count isn't a multiple of the slice, the last slice must be zero-padded to maintain the interleave granularity.

Interleaving and swizzling

Coupling: Interleaving and Swizzling are not independent.

The Requirement: The documentation states:

"When interleave is `CU_TENSOR_MAP_INTERLEAVE_32B`, the swizzle parameter must be set to `CU_TENSOR_MAP_SWIZZLE_32B`."

Reasoning: The hardware path for de-interleaving 32B chunks from Global Memory feeds directly into the swizzling logic for 32B atoms in Shared Memory. They operate in lockstep to map the complex global layout to a bank-conflict-free shared layout

A few important points

If using 32B interleaving then your global address should be 32B aligned and if using 16B interleaving then global address should be 16B aligned

Also your global strides should be a multiple of 16 if using 16B interleaving and global strides should be multiple of 32 if using 32B interleaving

Also if you are using interleaving the dimensionality must be greater than or equal to 3

Usage

```
CU_TENSOR_MAP_INTERLEAVE_NONE = 0,
CU_TENSOR_MAP_INTERLEAVE_16B,
CU_TENSOR_MAP_INTERLEAVE_32B
```

L2 promotion

Fetching data from HBM is slow compared to fetching data from the L2 cache.

Prefetching anticipates future data needs by issuing non-blocking transfers from HBM to L2 cache

Without promotion (`CU_TENSOR_MAP_L2_PROMOTION_NONE`), the memory controller uses a standard cache line fetch size (typically 32 bytes). This is inefficient because the L2 cache manages data in 128-byte lines.

By promoting the fetch size to 128B or 256B, a single TMA request ensures that a larger contiguous block of data is brought into the L2 cache in one go.

This maximizes bandwidth efficiency for predictable patterns like GEMMs, ensuring data is resident in L2 exactly when the compute units need it.

L2 Promotion Enumeration, Hardware Implications

Enumerator	Value	Hardware Action	Use Case
`L2_PROMOTION_NONE`	0	**32B Fetch:** Standard cache line.	Sparse/Random access.
`L2_PROMOTION_L2_64B`	1	**64B Fetch:** Fetches aligned 64B sector.	Moderate density.
`L2_PROMOTION_L2_128B`	2	**128B Fetch:** Fetches aligned 128B sector.	Matches large vector loads/ tile widths.
`L2_PROMOTION_L2_256B`	3	**256B Fetch:** Max fetch size.	**Dense GEMM/Convolution.**

`CU_TENSOR_MAP_L2_PROMOTION_L2_64B` and `CU_TENSOR_MAP_L2_PROMOTION_NONE`

`CU_TENSOR_MAP_L2_PROMOTION_NONE:` Uses the default 32-byte sector fetch. If your tensor is very sparse or the stride is massive, fetching neighbors is wasteful. Use this to avoid "over-fetching" (wasting bandwidth on useless data).

`CU_TENSOR_MAP_L2_PROMOTION_L2_64B:` Fetches two adjacent 32B sectors (64 bytes total).

useful for specific half (FP16) tensor shapes where the inner dimension is exactly 64 bytes (32 elements)

CU_TENSOR_MAP_L2_PROMOTION_L2_128B

Fetches the full 128-byte cache line in one shot.

Reduces DRAM command overhead by 75% (1 command instead of 4).

Alignment: Matches the standard vector load size of CUDA (ld.global.v4 = 16B × 32 threads = 512B, often split into 128B chunks).

Recommendation: This is the standard default for most dense FP16/BF16 GEMM operations.

CU_TENSOR_MAP_L2_PROMOTION_L2_256B

Fetches two full cache lines (256 bytes) in a single logical transaction.

The H100 memory controller is robust enough to handle multi-line prefetching

Requires high data density. Your application must consume this data immediately to justify the cache space.

Highest risk of cache pollution if the data isn't used.

A small but important note

Rule of Thumb:

Dense GEMM/Conv: Always use L2_128B or L2_256B. Maximize bus efficiency.

Embedding Lookups / Sparse: Use NONE (32B). Don't fetch what you won't touch.

Tuning Variable: The choice depends on Tensor_Inner_Dim_Bytes.

If Inner Dim < 64B, L2_128B might be wasteful. Match the promotion size to your contiguous data width.

OOBfill

oobFill helps handle out-of-bounds conditions during tensor copy operations on H100 GPUs.

Automatically fills out-of-bound regions with zeros or NaNs in the destination, not in HBM.

Source data in HBM remains unchanged throughout the process.

Eliminates manual boundary checks in kernels, improving code clarity and performance.

Useful for tiling operations, especially when tiles extend beyond tensor edges.

Choice of fill depends on application needs; zero for padding, NaN for debugging.

usage

```
CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE = 0,
CU_TENSOR_MAP_FLOAT_OOB_FILL_NAN_REQUEST_ZERO_FMA
```

The second statement uses Nan but during FMA operation its treated as zero