cp.async.bulk

Prateek Shukla

`cp.async.bulk` operations

`cp.async.bulk` is a family of PTX instructions for hardware-accelerated, asynchronous bulk memory transfers on Hopper H100 GPUs.

These operations are offloaded to dedicated hardware and execute independently of the SM's compute pipeline, allowing computation to proceed while data transfers happen in the background

`cp.async.bulk` can handle large, multi-dimensional tensor transfers efficiently from 1D to 5D tensors containing hundreds or thousands of bytes.

`cp.async.bulk` operations require barrier objects for coordination, ensuring proper ordering between asynchronous transfers and compute operations

cp.async.bulk vs cp.async(ampere)

cp.async in ampere

Uses the Load/Store Unit (LSU)

It is "asynchronous" because the thread issues the instruction and moves on, but the thread is still responsible for calculating the address and issuing the command for every 16 bytes of data.

cp.async.bulk in hopper

Uses the Tensor Memory Accelerator (TMA)

A single thread issues one instruction to copy an entire tile and the TMA handles all the address calculations, loop unrolling, and movement in the background

Some more key differences

cp.async (ampere)

To copy a 4KB tile of data, every thread in the warp has to loop and issue multiple cp.async instructions. This burns register cycles and instruction cache.

Uses cp.async.commit_group and wait_group

cp.async.bulk (hopper)

One single thread can initiate the transfer for the whole block. The other 31 threads in the warp (or 127 in the block) effectively do nothing related to memory copy initiation.

With mbarrier The TMA hardware updates the barrier's "transaction count" automatically as bytes arrive

Layouts of cp.async.bulk

1. The tensor layout

This is the layout used when you have a TMA descriptor (Tensor Map) and want to copy a specific multidimensional tile.

```
cp.async.bulk.tensor.ndim.dst.src.barrier_type{.cache_hint}{.multicast}

    dst_addr, tensor_map, coordinate_array, mbarrier_addr;
```

2. The Raw Layout (Linear)

This is the layout used for simple, contiguous byte copies

```
cp.async.bulk.dst.src.barrier_type{.cache_hint}

    dst_addr, src_addr, size, mbarrier_addr;
```

cp.async.tensor

Adding .tensor in the instruction specifies that the operation is tensor-aware, meaning the instruction works on multidimensional tensor data rather than just flat arrays

This opens up a bunch of other really important options which we can set in the instruction. This is also what enables the use of cuTensorMap

`cp.async.bulk.tensor.{1d,2d,3d,4d,5d}`

1d/2d/3d tells the hardware how many numbers (indices) it needs to read from you to locate the tile.

So basically once we get the n number of indices which are required to locate the tile in the tensor then we can grab the tile as per the information given in cuTensorMap

Nd in here means the tensor have N dimensions

We can have tensors with up to 5 dimensions

The state space of source and destination

In `cp.async.bulk.tensor.{dim}.{space1}.{space2}`

- `space` represents the state space of the source tensor
- `space2` represents the state space of the destination tensor

These can be anything in ones which we discussed, `global`, `shared::cta`, `shared::cluster` etc

The load mode {`.tile, .im2col, .im2col::w …..`}

This modifier is critical because it tells the TMA hardware how to interpret the coordinates you provide and how to transform the data on the fly

`.tile` - The TMA uses the coordinates provided in `tensorCoords` to calculate a base address and follows the strides defined in the tensorMap to fetch a dense, contiguous multi-dimensional box (a tile) of data

`im2col`

It performs a hardware-accelerated `im2col` transformation during the fetch. In past you had to write a kernel to copy pixels from an Image layout (N,C,H,W) into a Column layout (Matrix). This wasted memory bandwidth and register space

Now you can just give the coordinates of the top-left corner of a convolution window and the TMA grab the pixels required for the filter, unroll them, and place them in L2 as if they were a column of a matrix, this speeds up the operation.

The completion mechanism

`cp.async.bulk.tensor.{}.{}.{}.completion_mechanism`

We have two completion mechanisms

- `mbarrier::complete_tx::bytes`
- `.bulk_group`

`mbarrier::complete_tx::bytes`

The `cp.async.bulk` call takes the mbarrier handle (likely in [mbar] parameter) and the completion mechanism specification. The hardware tracks this operation and automatically updates the barrier's tx-count as data moves

When `tx-count` hits zero, the barrier phase flips and waiting threads are released

You need a pointer to the mbarrier and the size of the copy operation as the operands for this operation

bulk_group

`bulk_group` is a much more simpler and lightweight alternative to mbarrier.

Instead of tracking the tx_count, you issue a bunch of copy operations using bulk_group and then then batch them together using commit_group then wait on it using wait_group until only N or less of the most recent bulk async-groups are pending.

```
asm volatile(
    "cp.async.bulk.tensor.1d.global.shared::cta.bulk_group [%0, {%2}], [%1];"
    :
    : "l"(gmem_int_desc), "r"(smem_addr), "r"(dest.crds[0])
    : "memory");
```

L2 cache hinting

```
cp.async.bulk.tensor.1d.shared::cta.global.mbarrier::complete_tx::bytes.L2::cache_hint
```

During our async copies if we let the data flood the L2 cache it will kick out other important cached data which you might be using repeatedly. To prevent one use data to take useful space from L2 cache we use L2 hinting

There are different ways to implement it for loads and stores

evict_first is a way to tell the hardware to throw the data immediately to save the cache space

evict_last is a way to tell the hardware to mark the data persistent and keep it as long as possible

evict_normal is the default behavior

L2 hinting loads(global -> shared/dsmem)

During the global -> shared transfers data is loaded and cached into L2 according to write back cache policy.

For Reused Weights/Data (Best): Use evict_last. This marks the data as "persistent," telling the L2 cache to keep it as long as possible.

For Streaming/One-Pass Data: Use evict_first. Because essentially you are using the data once and then throwing it away

L2 hinting stores (shared -> global)

When writing results back to global memory, you typically don't need to read them again immediately.

For Final Output (Best): Use evict_first. You are writing data to HBM and likely won't read it back on this SM.

This minimizes cache pollution. The data hits L2 (to serve the write) but is immediately marked as the first candidate for eviction, keeping the cache clean for your inputs

Creating a cache policy descriptor

The createpolicy instruction in PTX creates a 64-bit cache policy descriptor that encodes eviction priorities for specific memory access patterns. Here's how it works:

A cache policy descriptor is 64 bit object which determines the cache policy

```
createpolicy.fractional.L2::evict_last.b64 policy_reg, 1.0;
```

```
createpolicy.fractional.L2::evict_first.b64 policy_reg, 1.0;
```

dest is the 64 bit register which will hold the cache policy descriptor

fraction(1.0) determines to what fraction of data policy is applied

Copying unstructured data vs structured (tensors) data

cp.async.bulk (Unstructured) is a hardware-accelerated memcpy. It treats memory as a linear stream of bytes. It does not "know" that your data is a matrix, a 3D volume, or a tiled tensor.

We use `cp.async.bulk.shared/cp.async.bulk.global` to copy unstructured data

cp.async.bulk.tensor (Structured) is a smart, descriptor-based copy. It relies on a CUtensorMap object (opaque handle) created on the host. The hardware "understands" the dimensions, strides, and boundaries of your data.

We use `cp.async.bulk.tensor` to copy structured tensors, cuTensorMap comes to play here

Unstructured copies

```
// global -> shared::cta
cp.async.bulk.dst.src.completion_mechanism{.level::cache_hint}
                     [dstMem], [srcMem], size, [mbar] {, cache-policy}

.dst =                  { .shared::cta }
.src =                  { .global }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
.level::cache_hint =    { .L2::cache_hint }
```

Operands:

dstmem - the destination address in the shared memory

srcMem - the source in shared memory

size - size of the data transferred in bytes

mbar - the pointer to the mbarrier object which you are using

cache_policy - 64 bit pointer to the policy descriptor

```
// shared::cta -> global
cp.async.bulk.dst.src.completion_mechanism{.level::cache_hint}{.cp_mask}
                     [dstMem], [srcMem], size {, cache-policy} {, byteMask}

.dst =                  { .global }
.src =                  { .shared::cta }
.completion_mechanism = { .bulk_group }
.level::cache_hint =    { .L2::cache_hint }
```

Operands:

Dst: the destination address in the global

Src: The source from where the copy is done

Size: size of transaction

Cache_policy: the descriptor for the cache policy

Mask: for masked writes, specifies which bytes to write for the destination

```
// shared::cta -> shared::cluster
cp.async.bulk.dst.src.completion_mechanism [dstMem], [srcMem], size, [mbar]

.dst =                  { .shared::cluster }
.src =                  { .shared::cta }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
```

dst - the destination address in the dsmem

src - the source address in the shared memory

size - copy size

completion_mechanism - mbarriers

The point here is that you are not transferring data to the shared memory allocated to some other block but you are moving data to a location in distributed shared memory which is pooled by all the blocks

Origins of multicast

Computing data is quicker than moving data. Especially moving data from HBM is really time and energy intensive. Computation can be done a lot more quicker

A lot of deep learning applications use gemms and in gemm many different blocks need to read the same chunk of Matrix A to multiply it against their specific chunk of Matrix B. If SM0, SM1, SM2, and SM3 all need "Tile A0".

In the previous generations if you had 4 SMs that all needed the same piece of data (a weight matrix for a neural network layer), all 4 SMs had to individually request that data from the global memory (which was really expensive).

Why not fetch the data once from HBM and copy it to all 4 SMs simultaneously as it flows through the wire?

Breakdown of the operation

The TMA reads size bytes from Global Memory (srcMem) into the L2 cache. The cache hint tells L2 to persist this line, optimizing bandwidth for subsequent accesses by other waves.

The L2 cache controller reads the data once and broadcasts it over the cluster crossbar, simultaneously targeting the SMEM banks of every block defined in the multicast mask.

The L2 cache controller reads the data once and broadcasts it over the cluster crossbar, simultaneously targeting the SMEM banks of every block defined in the multicast mask.

Upon completion, the TMA uses the mbar pointer (also multicast-encoded) to atomically add size bytes to the transaction count of the mbarrier in every participating block simultaneously

single leader thread issues this non-blocking instruction. The TMA hardware manages the entire fetch-broadcast-signal pipeline independently, allowing threads in all blocks to compute or sleep while waiting

Mbarriers and multicast

Its really easy to shoot yourself in the foot here because there are things to understand about using mbarriers for specific thread blocks in the clusters.

the mbarrier object is replicated at the same relative memory offset within the Shared Memory of every participating CTA. When the producer issues the TMA instruction, the hardware broadcasts data to all CTAs in the ctaMask and automatically signals the mbarrier at that specific address in each destination CTA

The hardware knows the group members immediately from the instruction's mask. Receiver CTAs do not need to 'arrive' to form the group; they simply wait on their local mbarrier instance to know when valid data has landed

The first thread (or any single thread) of all the blocks in the cluster call `expect_tx` with the amount of data they are expecting and they all are pointing to the mbarrier in their own shared memory. The way `cp.async` tracks the barrier is by using offset to get to the mbarrier for each block

A single thread in the whole block calls `cp.async.bulk` with multicast and the mask for all the blocks in that clusters, it points to its own barrier

If you want the data to be transferred to blocks 0 and 3 and not in block 1 and 2 then you must not call mbarrier from block 1 and 2 and put them in mask

All consumer warps in all the blocks spin on their local barrier `mbarrier.try_wait`

The arrival counts are managed by the TMA itself for the masked blocks. TMA automatically signals "remote arrival" to the mbarrier at the specified offset

A small note

You need to manually manage the shared memory layout when using dynamic shared memory

Instead of doing this - `extern __shared__ char smem[];` and the adding mbarrier

Use this - `uint64_t* bar_ptr = reinterpret_cast<uint64_t*>(smem);`

```
int tma_alignment = 128;

int data_offset = (sizeof(uint64_t)+ tma_alignment - 1) & ~(tma_alignment - 1);

half* tile_ptr = reinterpret_cast<half*>(smem + data_offset);
```

All participating CTAs (say, 16 CTAs running on 16 different SMs within the same cluster) must first establish themselves as a multicast group. Each CTA allocates the same mbarrier object in shared memory and calls mbarrier.init.shared.b64 with an arrival count, then executes arrive() on that mbarrier. This arrival isn't just synchronization—it's hardware registration. The memory subsystem now knows these 16 CTAs form a logical group that will receive identical data.

Every CTA in the multicast group creates an identical CUtensorMap descriptor. On the host, you call make_tma_copy(SM90_TMA_LOAD_MULTICAST{}, gmem_tensor, smem_layout, cluster_size) which encodes the tensor geometry, data type, swizzling pattern, and critically, the cluster dimensions. This descriptor is passed to the kernel (marked __grid_constant__) and tells TMA what global memory region to fetch and where in each CTA's shared memory to place it. All 16 CTAs must use this exact same descriptor—any deviation breaks the contract that they want the same data.

Each CTA (typically from a single elected thread per CTA) issues the TMA multicast instruction: cp.async.bulk.tensor.shared.cluster.global.mbarrier.multicast. Notice the .cluster scope and .multicast qualifier—these signal hardware intent. The instruction takes the shared memory destination address, the tensorMap pointer, tensor coordinates, the mbarrier pointer, and crucially, a ctaMask parameter. The ctaMask is a 16-bit bitmask (for a cluster size of 16) where bit i indicates whether CTA i participates—for example, 0xFFFF means all 16 CTAs receive the data.

Here's where the magic happens at the L2 cache level. The L2 cache controller receives 16 seemingly independent requests for the same tensor tile, all tagged with the same mbarrier group ID. The hardware recognizes this pattern and promotes one request as the leader. That leader request triggers a single read from HBM (let's say 1MB of weight data). As data streams into the L2 cache, the cache controller doesn't just send it to one SM—it multicasts (simultaneously forwards) the data to all 16 participating SMs' L1 caches and directly into their shared memory regions. You pay 1MB of HBM bandwidth but deliver 16MB worth of data across the SMs.

The TMA hardware also automatically decrements the transaction byte count (tx-count) on each CTA's mbarrier as data arrives, tracking completion progress.

After issuing the TMA multicast, each CTA executes wait_barrier(tma_load_mbar, phase) or the equivalent mbarrier.try_wait in PTX. This blocks until the mbarrier's transaction count reaches zero—meaning all expected bytes have been delivered to that CTA's shared memory. Once all CTAs pass this barrier, they're guaranteed their shared memory is populated with the data, and compute can begin

```
// global -> shared::cluster
cp.async.bulk.dst.src.completion_mechanism{.multicast}{.level::cache_hint}
                     [dstMem], [srcMem], size, [mbar] {, ctaMask} {, cache-policy}

.dst =                  { .shared::cluster }
.src =                  { .global }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
.level::cache_hint =    { .L2::cache_hint }
.multicast =            { .multicast::cluster  }
```

Dst - the destination address

Src - the global memory pointer

Size - the size of the operation

Mbar - the pointer to mbarrier

Ctamask - the 16 bit mask for multicasting

Cache–policy - the cache policy

Structured copies

```
// global -> shared::cta
cp.async.bulk.tensor.dim.dst.src{.load_mode}.completion_mechanism{.cta_group}{.level::cache_hint}
                                  [dstMem], [tensorMap, tensorCoords], [mbar]{, im2colInfo} {, cache-policy}

.dst =                  { .shared::cta }
.src =                  { .global }
.dim =                  { .1d, .2d, .3d, .4d, .5d }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
.cta_group =            { .cta_group::1, .cta_group::2 }
.load_mode =            { .tile, .tile::gather4, .im2col, .im2col::w, .im2col::w::128 }
.level::cache_hint =    { .L2::cache_hint }
```

Copy data from global to cta

Operands -

dstMem - pointer to shared memory location

tensorMap, tensorCoord - address to tensorMap, array of box coordinates

srcMem - pointer to the global memory address

cache-policy - pointer to the cache descriptor

```
// global -> shared::cluster
cp.async.bulk.tensor.dim.dst.src{.load_mode}.completion_mechanism{.multicast}{.cta_group}{.level::cache_hint}
                                 [dstMem], [tensorMap, tensorCoords], [mbar]{, im2colInfo}
                                 {, ctaMask} {, cache-policy}

.dst =                  { .shared::cluster }
.src =                  { .global }
.dim =                  { .1d, .2d, .3d, .4d, .5d }
.completion_mechanism = { .mbarrier::complete_tx::bytes }
.cta_group =            { .cta_group::1, .cta_group::2 }
.load_mode =            { .tile, .tile::gather4, .im2col, .im2col::w, .im2col::w::128 }
.level::cache_hint =    { .L2::cache_hint }
.multicast =            { .multicast::cluster  }
```

Operands -

dstMem - pointer to dsmem

tensorMap, tensorCoord - address to tensorMap, 1d vector telling the coordinates

mbar - the pointer to mbarrier object

ctaMask - Mask for selecting the blocks in which we have to copy

cache-policy - pointer to the cache descriptor

```
cp.async.bulk.tensor.dim.dst.src{.load_mode}.completion_mechanism{.level::cache_hint}
                                  [tensorMap, tensorCoords], [srcMem] {, cache-policy}

.dst =                  { .global }
.src =                  { .shared::cta }
.dim =                  { .1d, .2d, .3d, .4d, .5d }
.completion_mechanism = { .bulk_group }
.load_mode =            { .tile, .tile::scatter4, .im2col_no_offs }
.level::cache_hint =    { .L2::cache_hint }
```

TMA stores using bulk_group

tensorMap, tensorCoords - 64 bit pointer to the cuTensorMap, array of box coordinates

srcMem - the memory location from which data is coming from

Cache-policy - pointer to the cache policy descriptor

cp.async.bulk.prefetch

We can prefetch the data to L2 cache for lower latency

We use cp.async.bulk.prefetch.tensor to do that

```
cp.async.bulk.prefetch.L2.src{.level::cache_hint}   [srcMem], size {, cache-policy}

.src =                 { .global }
.level::cache_hint =   { .L2::cache_hint }
```

```
cp.async.bulk.prefetch.tensor.dim.L2.src{.load_mode}{.level::cache_hint} [tensorMap, tensorCoords]
                                                                    {, im2colInfo } {, cache-policy}

.src =                 { .global }
.dim =                 { .1d, .2d, .3d, .4d, .5d }
.load_mode =           { .tile, .tile::gather4, .im2col, .im2col::w, .im2col::w::128 }
.level::cache_hint =   { .L2::cache_hint }
```

A few important points

cp.async.bulk.prefetch is a performance hint, if you launch cp.async.bulk.prefetch and then immediately launch cp.async.bulk.tensor and the data isn’t cached then the device will fetch the data from HBM anyways.

Even though you use the same tensorMap for both the main cp.async op and the prefetch operation there is no effect of the L2 promotion or the swizzling to the way to data is cached in L2

cp.reduce.async

It offloads the atomic accumulation of an entire data tile from the SM to the TMA

$$\text{Global}[i] \leftarrow \text{Global}[i] \ \mathbf{OP} \ \text{Shared}[i]$$

This is literally the work which TMA does with that instruction. Following are two versions

cp.reduce.async.bulk.dst.src.completion_mechanism{.level::cache_hint}.redOp.type [dstMem], [srcMem], size{, cache-policy}

cp.reduce.async.bulk.tensor.dim.dst.src.redOp{.load_mode}.completion_mechanism{.level::cache_hint} [tensorMap, tensorCoords], [srcMem] {,cache-policy}

The redOps and the data types allowed

.add

.min/.max

.inc/.dec

.and

.or

.xor

.f16

.bf16

.b32

.u32

.s32

.b64

.u64

.s64

.f32

.f64