Asynchronicity and barriers

Prateek Shukla

The systems in synchronous world

In any system there are two major operations

Doing: processing information, solving an equation, cooking food

Fetching: retrieving the data, reading, comprehending the question, bringing the ingredients

In a synchronous world, these happen sequentially. You are "blocked" until the current task finishes.

The synchronous way of cooking food is where while you are cooking food you are not multitasking on any other work. Once the food is cooked then you go to the next task. You are “blocked” from doing anything else

Asynchronicity decouples the Request from the Result.

While cooking food, you go out to work on something else and after some time at some kind of signal you come back and use the ready food.

You accomplished two tasks in the time it took to do one. You have achieved Latency Hiding.

If the cost of fetching a resource is higher than the time it takes to process it, you must overlap the fetching of the next item with the processing of the current item.

The essence of latency hiding is not being “blocked” by an instruction and the ability to put the current job in background and work on something else.

Blocking and non blocking operation

A blocking or synchronous operation pauses the execution of the launching thread (e.g., the CPU host thread) until that operation has fully completed. Control does not return to the thread, so it cannot proceed to the next line of code. This creates an implicit synchronization point, guaranteeing the task is finished before the host continues

A non-blocking or asynchronous operation returns control immediately to the launching thread, before the operation has actually finished.

Asynchrony in GPUs

In modern computing, "Doing" (Math/Compute) is incredibly fast. "Fetching" (Memory Access) is agonizingly slow in comparison.

The time it takes to fetch the data is much higher than the time it takes to compute on that data. Thus the goal of high performance architecture is not just to make the math faster; it is to ensure the math never stops.

The Nvidia H100 is often praised for its raw speed (FLOPS), but its true genius lies in its architecture of asynchronicity. It is designed to ensure that its massive Tensor Cores never have to wait for the data. This is powered by using non blocking instructions for tensor cores and TMA

Async operations follow this pattern

Stage 1: Initialization - A thread fires an asynchronous operation and immediately goes off to the next instruction

Stage 2: Tracking and parallel execution - some part of system tracks the operation (mbarrier for TMA, internal hardware scorecard for wgmma). While these operations are working, other units work in parallel

Stage 3: Synchronization - Once the async operation is done operating in the background

The synchronization step in H100

Warp launches instructions really fast in the order of a few nanoseconds

Once a async instruction reaches the execution unit, it attempts to read the memory immediately

The bottleneck is the memory bandwidth. Moving data from HBM to shared memory/registers is a high-latency operation which take a few hundred ns compared to the logic core.

The two problems

Because the Issuer (~ns) is faster than the Memory Mover (~ hundreds of ns) tensor cores would have to wait for a few hundred ns for the data to arrive if the instructions for copy and compute are issued at the same time. Therefore there is a scope for latency hiding

Also in a big kernel, where instruction queue is full of commands for data that hasn't arrived yet, without some form of synchronization tensor cores might perform operations before the data have even arrived.

We need a way to make sure that the right operations are done on right data and we need to have a system where we can apply latency hiding

What exactly is mbarrier/semaphore

A hardware-accelerated synchronization primitive resident in Shared Memory designed to track the completion of asynchronous memory transactions.

Unlike traditional barriers that block execution until threads arrive, an mbarrier blocks execution until data arrives. It decouples the "Producer" (who issues the copy) from the "Consumer" (who waits for the data), enabling split-phase, fire-and-forget memory pipelines.

Producer sets the expected data transfer size in the barrier and then the hardware does its job in the background and once the transaction is completed the barrier is opened

Why is this a great solution

.The barrier forces the "fast" instruction launcher to respect the "slow" physical hardware by

1. Making sure that the slower operations are launched independently from the faster operations so that by the time the fast unit is launched we have the data

2. Have a way to prevent the fast execution unit from operating on the wrong data

And the way we do that is by using mbarrier.

The Big Picture

Proxies in CUDA

In the context of the NVIDIA H100 and the CUDA PTX memory model, Proxy is a term used to distinguish who is performing memory operations.

If two memory operations happen in the same proxy (e.g., the Generic Proxy), the hardware guarantees they are safe and ordered (mostly).

If operation A is in Proxy 1 and operation B is in Proxy 2, the hardware stops checking. It assumes they are completely unrelated. It lets them run wild, out of order, and in parallel

Generic proxy vs Async proxy

When you write a CUDA kernel, your thread executes instructions sequentially. When the thread reads or writes memory, it is acting as the Generic Proxy. The hardware guarantees that these operations happen in the order you wrote them (mostly) within that specific thread.

The Async Proxy refers to the hardware mechanism performing bulk async copies or tensor core math. When you issue a command like cp.async.bulk or wgmma, the Generic Proxy (thread) just kicks off the command and immediately moves to the next line of code.

The Async Proxy runs completely independently of the Generic Proxy. It does not know what the thread is currently doing, and the thread does not automatically know when the proxy is finished.

Read After Write

The Generic Proxy and Async Proxy have distinct paths to memory. The Generic Proxy operates through the SM's L1 cache, while the Async Proxy bypasses the L1 and interacts directly with the L2 cache or HBM.

Stores issued by the Generic Proxy are initially held in local store buffers or the L1 cache. They are not immediately visible to the rest of the system (including TMA).

If the Generic Proxy writes to a memory address and immediately triggers the Async Proxy to read that same address, the Async Proxy will likely read stale data from L2/DRAM because the new data is still stuck in the Generic Proxy's L1

This is called a read after write hazard

Write after read

A thread (or CTA) does a Generic Proxy read of address X, pulling X into L1 (or otherwise “anchoring” an old version locally).

Later, an Async Proxy write updates X via L2/HBM, without updating/invalidating the Generic Proxy’s L1 view. The Generic Proxy can keep operating on the stale L1 line. Two common failure modes:

A later Generic Proxy store / writeback / eviction can clobber the newer value written by the Async Proxy (stale line wins) or later Generic Proxy loads keep returning the old value even though X was updated via Async Proxy.

This is called a write after read hazard or WAR

fences

A fence is an explicit ordering/visibility point that constrains how memory effects become observable. You use it when the hardware might otherwise let things “pass” each other especially with asynchronous operations that are not automatically ordered with normal loads/stores the way you might assume

NVIDIA explicitly calls out that a proxy fence is required to synchronize across proxies for proper ordering

Also these fences are scoped (e.g., .cta, .cluster, .gpu, .sys) and the scope determines who must see the ordering, tied to a point of coherency in the hierarchy (e.g., L1 vs L2).

There are two types of fences Ordinary fence and cross proxy fence

Release fence (producer-side)

A release fence enforces this one-way rule:

All memory operations (especially writes) that appear before the release in program order become visible-before anything that appears after the release to other threads that synchronize with it (at the chosen scope).

It prevents earlier writes from being delayed/reordered past the release point.

```
__device__ __forceinline__ void ptx_fence_release_cta() {
  asm volatile("fence.release.cta;" ::: "memory");
}

__device__ __forceinline__ void ptx_fence_release_cluster() {   // SM90+
  asm volatile("fence.release.cluster;" ::: "memory");
}
```

Acquire fence (consumer-side)

An acquire fence enforces the opposite one-way rule:

No memory operation that appears after the acquire in program order (especially reads) is allowed to be observed as happening before it.

After the acquire, the thread is guaranteed to observe the writes that were made visible by a matching release (again, within the chosen scope/proxy).

```
__device__ __forceinline__ void ptx_fence_acquire_cta() {
  asm volatile("fence.acquire.cta;" ::: "memory");
}

__device__ __forceinline__ void ptx_fence_acquire_cluster() {   // SM90+
  asm volatile("fence.acquire.cluster;" ::: "memory");
}
```

Cross proxy fence

If you access the same location across multiple proxies, you need a cross-proxy fence. For the async proxy, use fence.proxy.async to synchronize memory between generic and async proxy.

It drains/orders the generic-proxy shared-memory write visibility so the async proxy doesn’t read an older view.

it’s not a “collective flush.” It’s per-thread ordering, so you still need a block sync so all writers have done it before the elected thread launches TMA (the CUDA example uses __syncthreads() around the setup/coordination).

```
__device__ __forceinline__ void fence_proxy_async_shared_cta_ptx() {
  asm volatile("fence.proxy.async.shared::cta;" ::: "memory");
}
```

The whole mbarrier pipeline

Step 1: initialize the mbarrier in shared memory and use mbarrier.init to create a barrier with expected thread arrival count. Thread arrival count make sure that a specific number of threads have issued the copy instructions.

Step 2: Set the expected transaction count for the async op you are gonna launch, using mbarrier.arrive. Everytime you do this you are logging an instruction thus reducing the arrival count.

Step 3: launch the async operation

Step 4: launch other operations and wait (threads sleep till thread arrival and expected_tx = 0)

Step 5: flip the phase, reset the arrival count, do the next operation

Phase and expected transaction count

A barrier's phase refers to its current reusable state or cycle. It is a single bit which flips whenever the barrier completes a cycle.

Transaction count represent the size of the asynchronous operations which you are performing. Expected transaction count is the amount of ‘work’ which is yet to be done with the async operation.

Because these are async ops, the hardware automatically decrements the transaction count as the async operation progresses. We attach the barrier to the async operation to make sure of this behavior

You can ‘reuse’ a barrier by flipping its phase at the end of operation and increasing the transaction count. The thread arrival count is automatically resetted as per the value set during initialization.

Initializing the mbarrier

This is the only instruction where most bugs originate. Get this wrong, and nothing else matters

```
mbarrier.init{.space}.b64 [addr], count
```

addr is just the memory address of the mbarrier in the state space

Count is the expected thread arrival count in the barrier, this is the value to which the barrier will reset when you reuse the barrier by flipping the phase.

Some things to keep in mind

Generally the scope is shared::cta so that only the threads in the same thread block can “see” the barrier

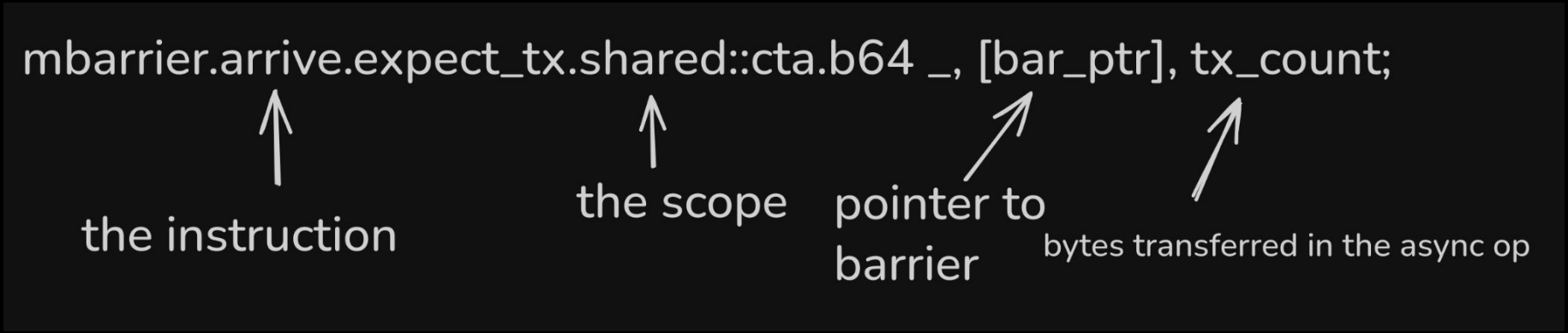
The address is shared memory pointer which is created using `__cvta`

You can use `shared::cluster` to make the barrier visible to all the threads in a cluster

You must use the mapa ptx instruction to get the address in the cluster

Mbarrier objects must be 64 bit aligned, unaligned access may cause silent corruption

Setting expected transaction count



Decrementing the arrival count by 1

adding the expected transaction bytes with tx_count

The `tx_count` is added to the barrier's pending `tx-count`. If you call arrive.expect_tx twice with 4096, the barrier expects 8192 bytes total

The placeholder _

The _ in the instruction is a placeholder for the `phase_out` which is the way to get a 64 bit encoded token which contains information about the current phase and transaction count of the barrier

It was there for some complicated synchronization issues

We discard phase_out because we are actually writing this information into the producer threads. Register pressure is expensive. it is cheaper to toggle a 1-bit integer than writing a 64 bit token in registers.

Synchronization

So till now we have launched the async copy instruction but now how do we know that the operation is completed.

The old way was `barrier.sync` which would block the thread until the operation completes but that destroys asynchronicity

The way we do it in hopper is by having an instruction which can provide true or false if the operation is completed. This allows Latency Hiding. A thread can check the barrier, see it's not ready repeat until the operation ends and the phase flips. This is precisely the job of `mbarrier.try_wait.parity`

mbarrier.try_wait.parity

You pass a phaseParity bit (0 or 1) into the instruction representing the specific execution phase you must verify is fully completed before proceeding

The hardware compares your input parity bit against the mbarrier object’s current internal parity state to determine if that specific phase is still active

If your input parity equals the barrier's current parity, the barrier is still processing that phase, so the instruction returns false (keep waiting). If your input parity differs from the barrier's current parity, the barrier has advanced to the next phase, so the instruction returns true (proceed)

A successful return implies the barrier parity has "flipped," meaning your requested parity now refers to the immediately preceding (and therefore completed) phase

The instruction

```
mbarrier.try_wait.parity{.sem.scope}{.shared{::cta}}.b64  waitComplete, [addr], phaseParity
                                                          {, suspendTimeHint};
```

try_wait: The operation name. It implies an attempt that might involve a temporary suspension of the thread.

waitComplete: (Destination) The boolean result.

1 (True): The barrier has reached the target phase (work is done).

0 (False): The barrier is not done, and the thread has just woken up

suspendTimeHint: Optional immediate value (constant) that tells the GPU scheduler how long to yield the thread if the condition isn't met.

phaseParity: An integer (0 or 1) representing the phase generation you are waiting for.

mbarrier.test_wait.parity

```
mbarrier.test_wait.parity{.sem.scope}{.shared{::cta}}.b64 waitComplete, [addr], phaseParity;
```

test_wait.parity : The variant of the instruction. It tracks the barrier's progress using a "phase parity" bit (0 or 1) rather than a raw integer count.

waitComplete: (Destination) A 1-bit predicate register (boolean). It receives 1 if the barrier is done, 0 if it's still busy.

[addr]: (Source) The pointer/address to the mbarrier object in shared memory.

phaseParity: (Source) An integer (0 or 1) representing the phase generation you are waiting for.

mbarrier wait with .acquire

It does everything which is there in mbarrier.try_wait with a critical distinction.

If the current synchronization round is fully finished, the .acquire semantic creates a strict memory fence, ensuring that all data writes from the async proxy or other threads are guaranteed visible before proceeding.

If you are loading data from Shared Memory into Registers (to feed a Tensor Core manually), the .acquire ensures those LD instructions don't fire until the data is valid. If they fire early, your registers get garbage.

Even though wgmma reads directly from Shared Memory, the instruction itself requires descriptors and memory state to be consistent. The .acquire ensures the dependency chain is respected

The complicated ways to implement it

An important point

In the producer warp which is launching the copies only a single thread is running all the instructions like mbarrier.init, mbarrier.arrive.expect_tx, cp.async.bulk and all that. Other threads in the warp are just lying around

The consumer warps are the ones which launch the wait operation as they are waiting for the tma copy to complete so that they can work on the data

mbarrier reuse pattern

```
int phase_bit = 0;

// 1. Initialize
asm volatile (
    "mbarrier.init.shared::cta.b64 [%0], %1;\n"
    :: "r"(barrier_addr), "r"(initial_count)
);
for (int k = 0; k < K_ITERATIONS; ++k) {
    asm volatile (
        "mbarrier.arrive.expect_tx.shared::cta.b64 _, [%0], %1;\n"
        :: "r"(barrier_addr), "r"(expected_bytes)
    );
    // 3. Launch Async Load (Hardware does the Arrival)
    asm volatile (
        "cp.async.bulk.tensor ... [%0];"
        :: "r"(barrier_addr)
    );
    // 4. Wait for Completion
    asm volatile (
        // loop
        "mbarrier.try_wait.parity.shared::cta.b64 P1, [%0], %1;\n"
        :: "r"(barrier_addr), "r"(phase_bit)
        // loop
    );

    // [CONSUME DATA HERE]

    // 5. Flip Phase for Reuse
    phase_bit ^= 1;
}
```

The phase is declared in registers

and not shared memory!

Each thread keeps its own int phase

phase ^= 1 as the last operation in loop

Is mbarrier only for async bulk copies??

Till now we discussed about mbarriers and cp.async.bulk and it really feels that mbarriers are built for async bulk copies.

Consider this:

Producer writes to Shared Memory

Consumer reads from Shared Memory (using WGMMA).

Producer wants to overwrite that same Shared Memory with the next tile.

If the Producer overwrites sA while the WGMMA is still reading it, your math is garbage. This is a Write-After-Read (WAR) hazard.

How do we prevent wars

We create a barrier with an expected thread arrival count

The Consumer is busy reading the data, producer is spinning on the barrier

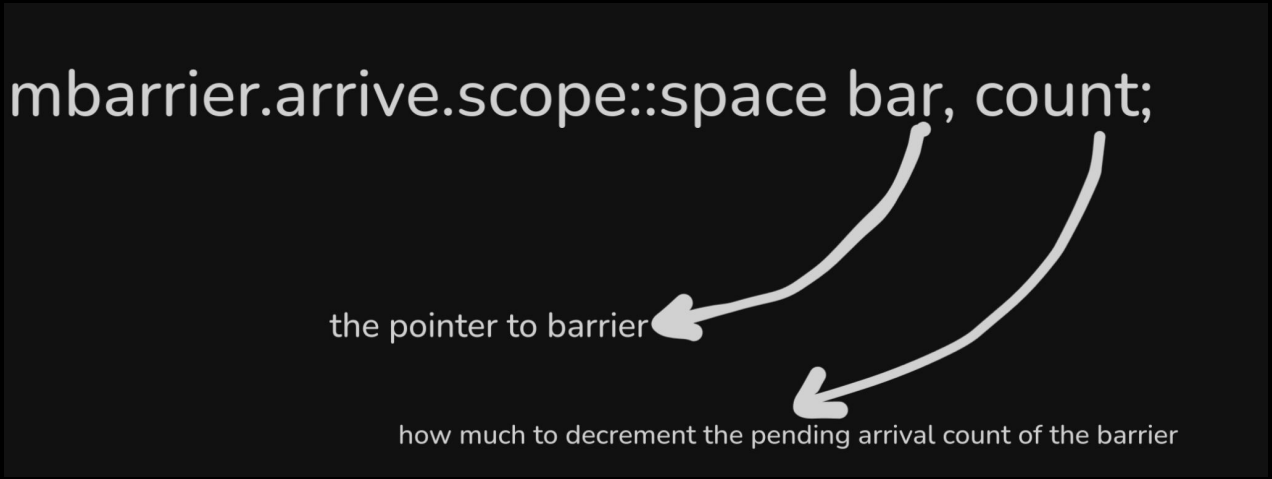
The Consumer finishes its last read instruction

Consumer executes instruction to decrease the thread arrival count

The barrier state flips and now the producer can write the new data

The instruction to decrease the arrival count is ….

mbarrier.arrive.scope



Just decreases the pending arrival thread count of the barrier by count

Compilers and reordering

Compilers often reorder the operations in order to get some efficiency gains

a = 1;

b = 2;

c = a + b;

Compilers might do b = 2 before a = 1 because they think it does not affect the whole c = a + b operation which uses them

Why this is fatal in async gpu ops

In multi-threaded code, reordering breaks correctness because different threads can observe operations in different orders.

Thread a -

d[0][0] = 42.0f;

flag = 1;

Thread b -

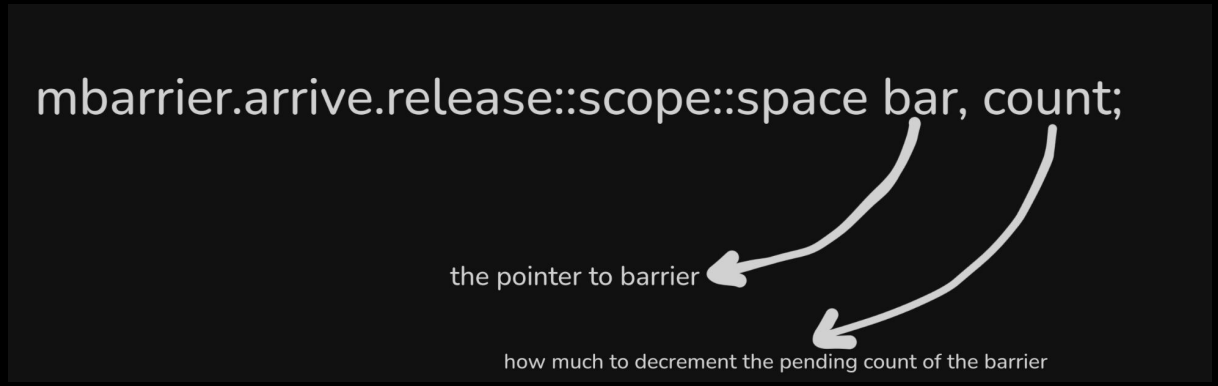
while (flag == 0);

float x = d[0][0];

mbarrier.arrive.release.scope

The .release suffix provides release semantics. That means it acts as a one-way hardware fence where memory writes above this line cannot reorder below it.

Use mbarrier.arrive.release after a thread finishes producing data that other threads will consume:



Destroying a barrier

```
mbarrier.inval{.shared{::cta}}.b64 [addr];
```

It formally invalidates an mbarrier object (a 64-bit synchronization primitive), effectively wiping the hardware's tracking of that barrier's state.

By invalidating the barrier, it frees up the specific shared memory address so it can be safely overwritten or repurposed for other variables later in the kernel.

It specifically converts a generic pointer to a 32-bit shared memory offset to ensure the GPU hardware addresses the correct local memory bank.

The whole pipeline

1. `mbarrier.init`: Threads create a barrier in shared memory set an expected thread count and start in Phase 0
2. `mbarrier.expect_tx`: producer threads tell the barrier to also expect a specific number of bytes from an upcoming async operation
3. `cp.async.bulk` : A thread launches `cp.async` (the copy) and then `mbarrier.try_wait.parity 0`, spins until Phase 0 is done
4. Flip: When all expected bytes (and threads) arrive, barrier auto-flips to phase 1
5. Computation begins
6. Once the computation ends we manually flip the phase and then start working on the next k tile to reuse the barrier

mbarrier.arrive_drop

It acts like a standard arrive operation, decrementing the pending arrival count for the current phase.

But more importantly it also permanently decrements the expected arrival count for the barrier. So if you are flipping the phase and trying to reuse the barrier the "expected arrival count" will be the previous value minus the number of drops that occurred

```
mbarrier.arrive_drop{.sem.scope}{.shared{::cta}}.b64 state,           [addr]{, count};
mbarrier.arrive_drop{.sem.scope}{.shared::cluster}.b64           _,   [addr] {,count};
mbarrier.arrive_drop.expect_tx{.sem.scope}{.shared{::cta}}.b64 state, [addr], tx_count;
mbarrier.arrive_drop.expect_tx{.sem.scope}{.shared::cluster}.b64   _, [addr], tx_count;
mbarrier.arrive_drop.noComplete{.release.cta}{.shared{::cta}}.b64 state,  [addr], count;

.sem   = { .release, .relaxed }
.scope = { .cta, .cluster }
```

`.sem` `.expect_tx` and `.noComplete`

`arrive_drop.expect_tx`: sets the expected transaction count and permanently and temporarily decreases the arrival count by 1

`arrive_drop.sem` - sem can be `.release` or `.relaxed`. `.relaxed` Specifies that no memory ordering is enforced and .release Ensures that all memory writes performed by the thread before the arrival are visible to any thread that waits on the barrier

`.noComplete`: instructs the hardware to perform the arrival (decrementing pending/expected counts) without triggering the phase completion, even if the conditions for completion (pending count reaching zero) are met.

`barrier.cluster`

Standard barriers like bar.sync (`__syncthreads`) only synchronize threads within a single block. They cannot coordinate producer/consumer blocks across a cluster.

`barrier.cluster` is required to synchronize across different thread blocks that are co-scheduled on the same cluster. It also guarantees that any writes to DSMEM made before the barrier are visible to all blocks in the cluster after the barrier

The key usecase is TMA Multicast. The way it works is that in the initial step, the block 0 might be running but block 1 might not be running and if block 0 tries to write into block 1 then the program will crash. By launching this you make sure that every block is physically present.

Two important instructions in `barrier.cluster`

`barrier.cluster.arrive` The thread signals it has reached the barrier. It does not stop. It continues executing independent instructions (math, local register ops) that don't depend on data from other blocks

`barrier.cluster.wait` this is a blocking instruction. The thread stalls here until every other thread/block in the cluster has signaled arrive. Once this unblocks, you are guaranteed that all data written by other blocks is now safe to read

On H100, Because Block A can write to Block B's memory, we need a barrier to prevent Block B from reading before Block A has finished writing

Asynchronous groups

When you launch a `cp.async.bulk` operation using bulk group it looks something like this -

```
asm volatile("cp.async.bulk.tensor.3d.global.shared::cta.tile.bulk_group"
             " [%0, {%2, %3, %4}], [%1];"
             :
             : "l"(tma_ptr), "r"(src_ptr), "n"(0), "r"(global_row_idx),
               "r"(global_col_idx / 64)
```

Bulk_group is attached to a `cp.async.bulk` instruction and this enable us to use `cp.async.bulk.commit_group` and `cp.async.bulk.wait_group`

These are the instructions which we are gonna use for batching a bunch of instructions and then using them

`cp.async.bulk.commit_group`

The way pipeline works in here is that we launch a bunch of operations using `cp.async.bulk` with `bulk_group`

Now there was no `cp.async.bulk.commit_group` operation which was done before so these copies are uncommitted, we can commit i.e batch them into a group, the point of batching them in a group is that we can later make other units wait until the execution of N units is finished

Any instructions launched after this instructions belong to the next group or are just instructions

We can have multiple groups with multiple `cp.async.bulk` operations

`cp.async.bulk.wait_group<N>`

Once we have launched and committed a bunch of operations then we need other units which are gonna use their outputs to wait

The `wait_group<N>` instruction waits until at most N committed groups remain pending. For example, `wait_group<0>` waits for all groups to complete, while `wait_group<2>` tells you to wait till only the two operations out of all the ones you launched are still pending.

The count refers to groups still pending, not groups that have completed. So `wait_group<2>` means "wait until only the 2 most recent committed groups are still pending" all the older groups must been completed

`cp.async.bulk.wait_group.read`

This stalls execution AND enforces an Acquire Fence.

This is really important for read-after-write scenarios because you risk reading the older data

namedbarriers

__syncthreads() is a whole-block rendezvous. Named barriers let you create multiple independent synchronization points inside one block, so different subsets of warps can coordinate without forcing unrelated warps to stop.

PTX explicitly allows different warps to use the same named barrier with different operations, such as mixing .arrive and .sync to build producer/consumer pipelines.

```
"barrier{.cta}.sync{.aligned}"          a{, b};
"barrier{.cta}.arrive{.aligned}"        a, b;

"barrier{.cta}.red.popc{.aligned}.u32" d, a{, b}, {!}c;
"barrier{.cta}.red.op{.aligned}.pred"  p, a{, b}, {!}c;
.op = { .and, .or };
```

a = barrier ID, b = participating thread count