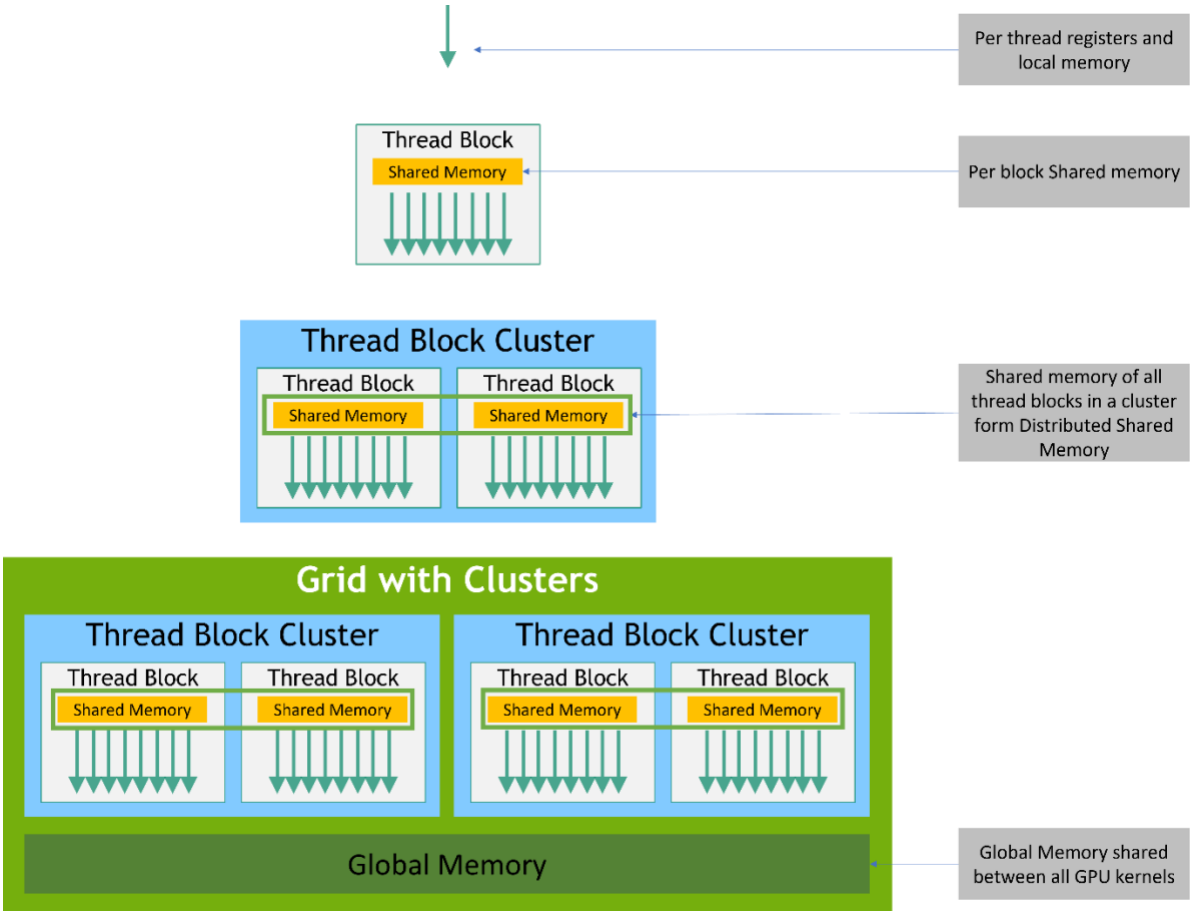
NVIDIA GPU Architecture

1 OVERVIEW

NVIDIA architecture is made up of a hierarchical design. Compute units consist of streaming multiprocessors, which are composed of groups of threads called warps. The threads within a warp are guaranteed to be co-scheduled, allowing efficient communication across threads.



Hopper introduces thread block clusters which are guaranteed to be concurrently scheduled into an SM, leveraging an additional level of hierarchy that permits wider collective operations. Also, shared memory among thread blocks is unified into **distributed shared memory (DSMEM)**, accessible to threads within the cluster.

2 ARCHITECTURE FEATURES

2.1 TENSOR CORES

2.2 COLLECTIVE OPERATIONS

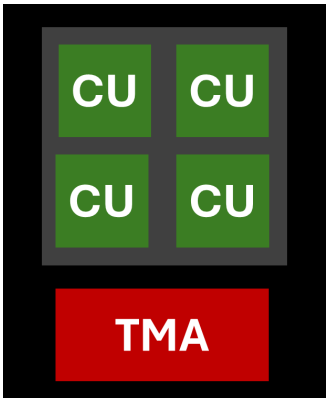
2.3 DPX (DYNAMIC PROGRAMMING ACCELERATION)

[CUDA DOCS] [CUDA INTRINSICS]

Provides operations to find the **maximum/minimum** for up to three 16 and 32-bit signed or unsigned integer parameters, with optional **ReLU** (clamping to zero).

2.4 TMA (TENSOR MEMORY ACCELERATOR)

2.4.1 Overview



Beginning with **Hopper (sm_90)** memory transfers between **GMEM↔SMEM** can be offloaded to the TMA unit. It has the advantage to issue asynchronous memory operations which imply that compute units can work continuously without the need for blocking for mem ops. And frees the need of using register to issue such operations.

*"Additional optimizations can be made such as kernel specialization, that can enable techniques such as **producer-consumer** pipelines in which a set of threads can be pushing data from **GMEM** to **SMEM** while consumers are in charge of compute."*

2.4.2 Examples

2.4.2.1 Async bulk copy

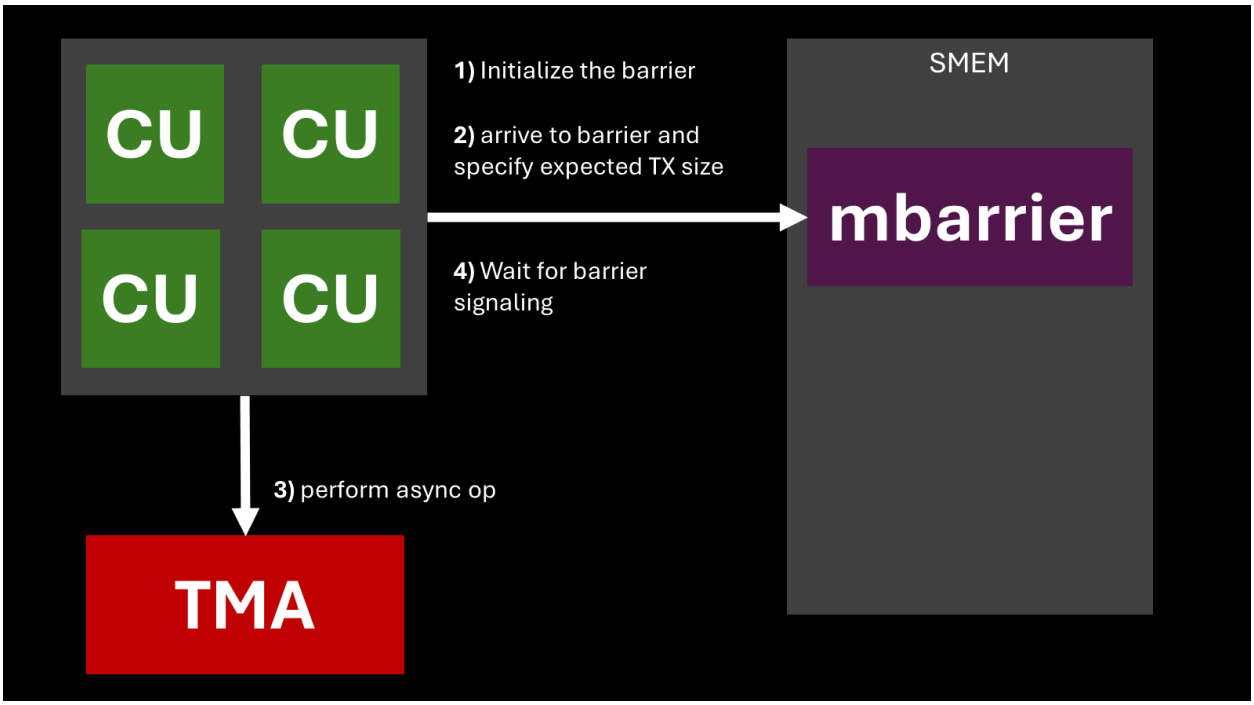
2.4.2.2 GMEM to SMEM

```
if (idx == 0) {
  mbarrier.init
  mbarrier.arrive.expect_tx

  // trigger async mem-op
  cp.async.bulk
}
// ensure thread convergence
__syncthreads()
mbarrier.try_wait
```

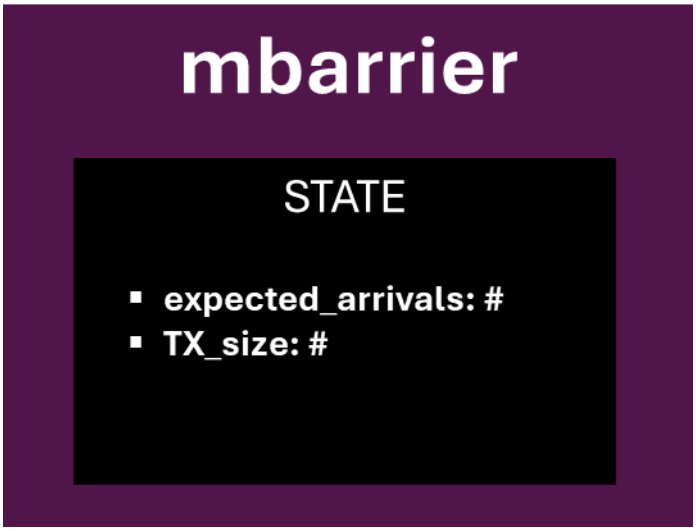
In the example above, we execute a **cp.async.bulk** operation that is offloaded to the TMA. Therefore, we need to synchronize our code using a barrier, which is activated when all the threads specified during initialization reach the barrier and when the TX payload is complete.

In this case, the barrier is initialized with a count of 1, representing the thread that triggers the operation. This thread then calls the arrive operation to set up the expected payload size. We subsequently wait on the barrier, which will be triggered once the TMA completes the memory operation and signals the barrier.



3 APPENDIX

3.1 MBARRIER (MEMORY BARRIER)



A barrier is a synchronization mechanism that ensures that multiple threads or processes reach a certain point of execution before any of them continue. It is used to coordinate the activities of concurrent tasks, allowing them to wait for each other at a designated point, often referred to as a "barrier." Once all participating threads have reached the barrier, they are allowed to proceed with their execution simultaneously.

The **mbarrier** object is initialized with the total number of threads expected to reach it. Since Hopper, the **mbarrier** also tracks information about memory transactions, which are signaled during asynchronous memory operations.

3.2 GPU BINARY PORTABILITY

"Unlike CPUs, GPUs have not adopted common long-lived architectures. That means that, while the CPU code for an application compiled once 10+ years ago may run on a brand new device, the GPU code would not have the same portability if it were compiled to the GPU's native Instruction Set Architecture (ISA).

To solve this problem, GPU programming models often defer generating native ISA to runtime when the GPU driver can optimize for the GPU's specific hardware capabilities. Graphics programming APIs support either shipping shader programs in source code form, or in a Virtual ISA that abstracts common hardware features at a higher level. Both forms allow the GPU driver to lower and optimize the program more efficiently, but the use of a virtual ISA reduces the time it takes the driver to generate native code. As such, virtual ISAs have become the preferred method for shipping shader programs for high-performance applications." **(FROM: DirectX Adopting SPIR-V as the Interchange Format of the Future)**

4 RESOURCES

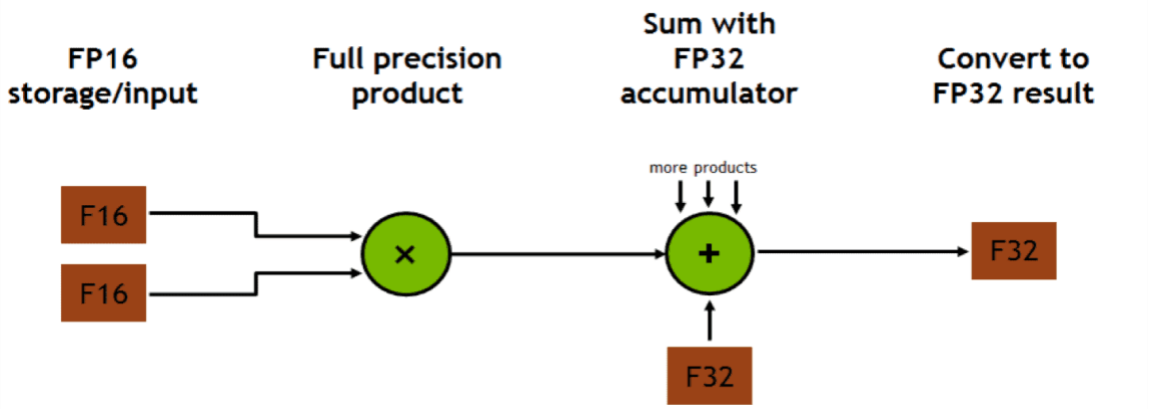
- CUDA Programming Model for Hopper Architecture | GTC 2022 (VIDEO)🔥
- Advanced Strategies for High-Performance GPU Programming with NVIDIA CUDA | GTC 2024 (VIDEO)🔥
- Boosting CUDA Efficiency with Essential Techniques for New Developers | GTC 2024 (VIDEO)🔥
- Lecture 23: Tensor Cores | GPU MODE 🔥
- A history of NVidia Stream Multiprocessor | FABIEN SLANGARD BLOG🔥

🔥: **Must read**

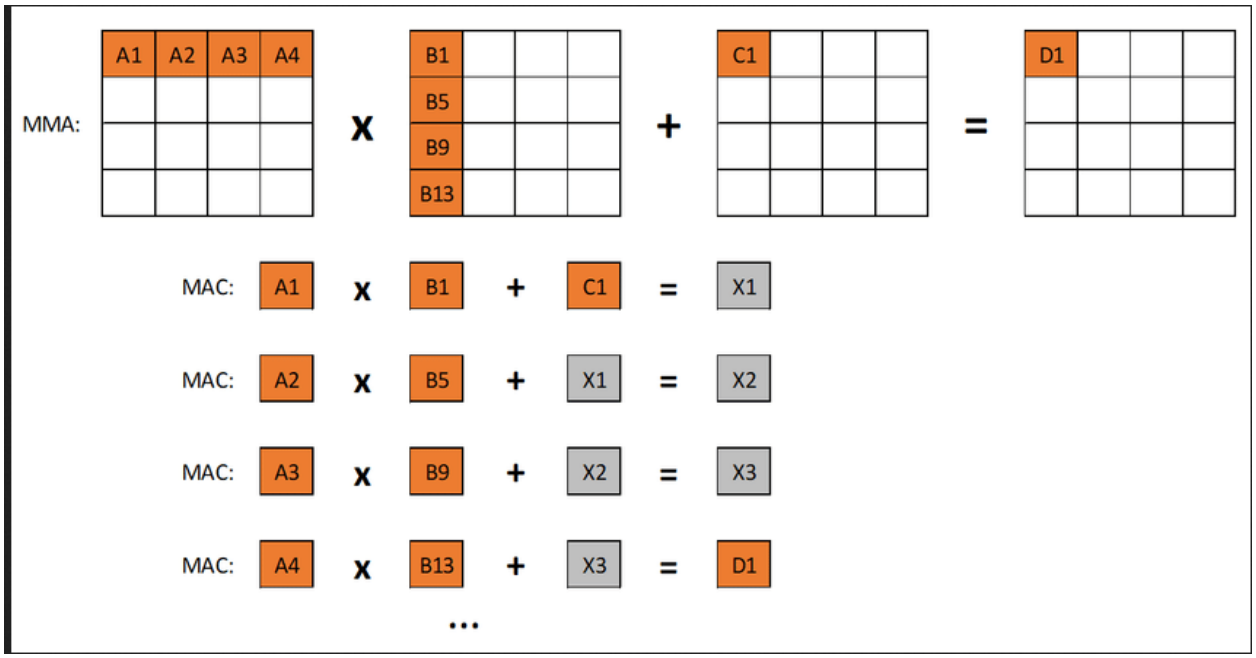
TODO

- FMA

result=a×b+c The advantage of FMA is its precision and performance. By performing the multiply and add operations in one step, it avoids rounding errors that would occur if the operations were performed separately.



- DPX
- MMA
  mma stands for **Matrix Multiply-Accumulate**. multiplying two matrices and adding the result to a third matrix.

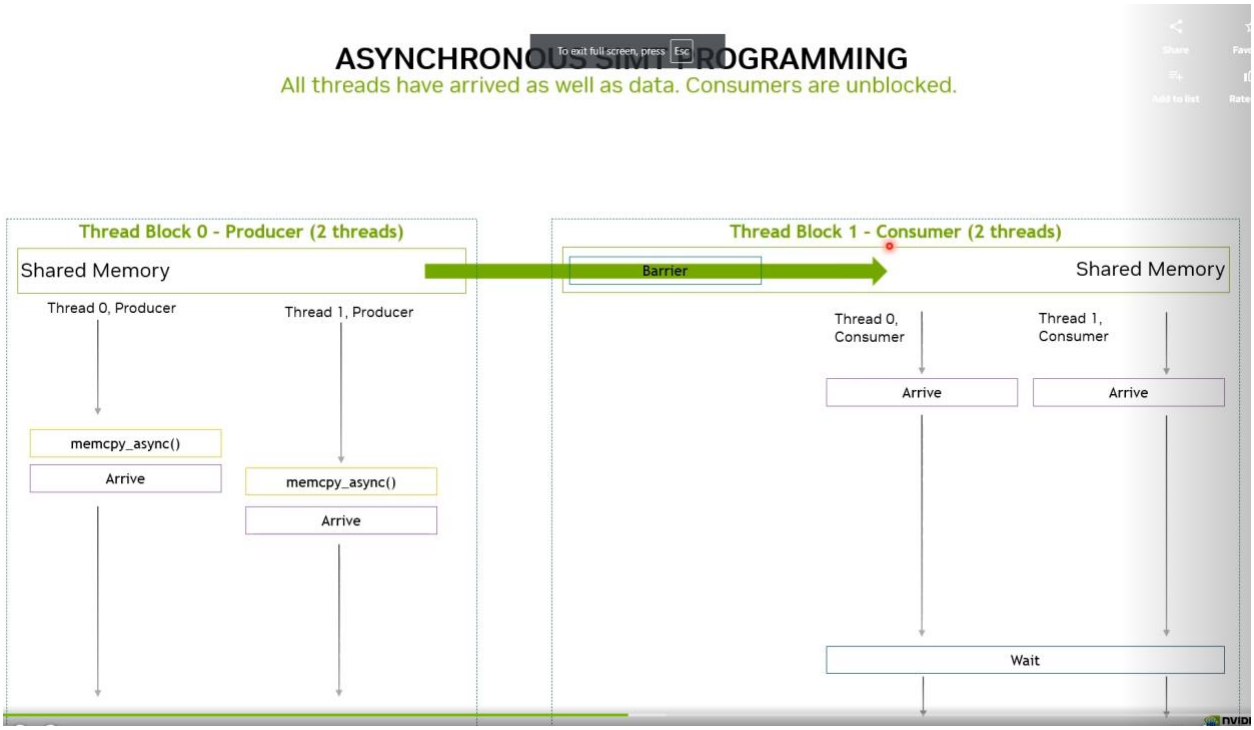


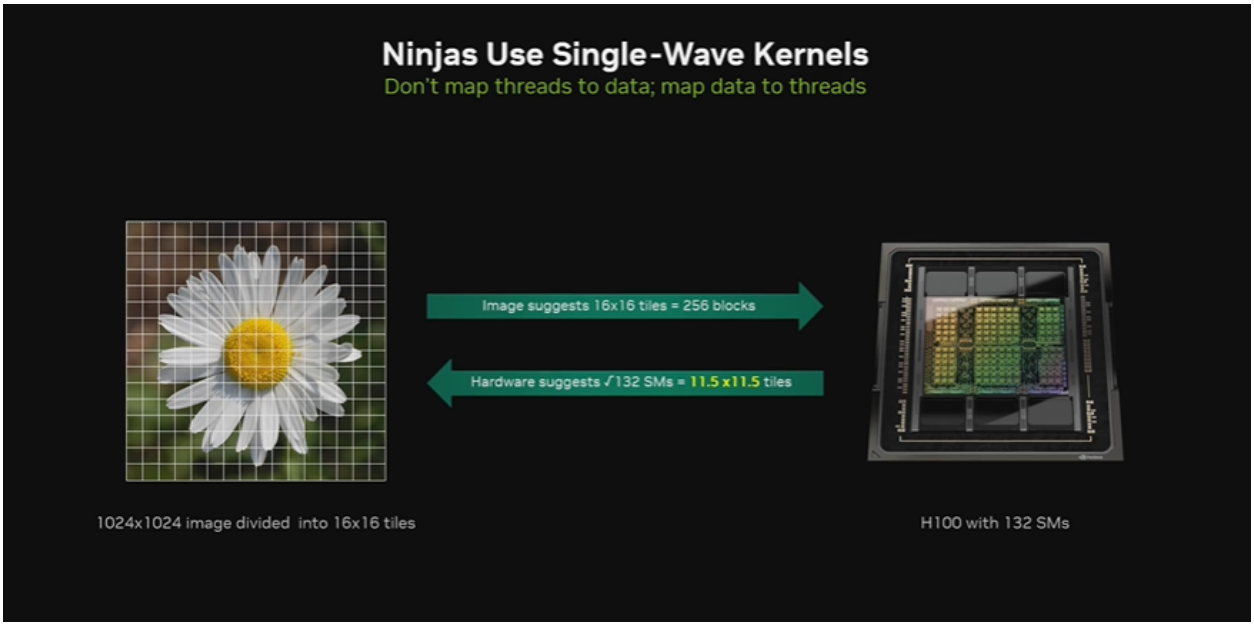
- WGMMA

WMMA simplifies the matrix operation process by operating at a warp level rather than explicitly defining all individual PTX operations.

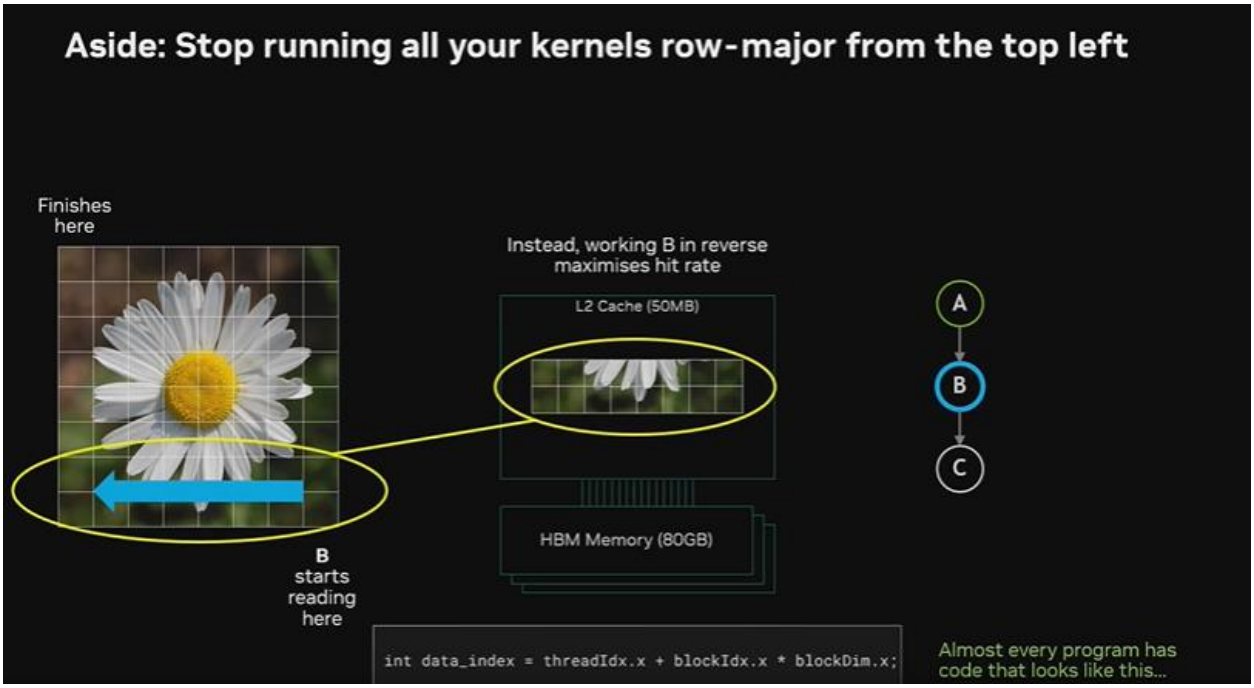
- LTO: JIT LTO & nvJitLink
- Hierarchy: Thread, Block, Thread Block Cluster[1], Grid. RMEM LMEM SMEM DSMEM (Distributed shared memory)[1], GMEM.
- TMA: Tensor Memory Accelerator[1]. Simply an async mem engine.
- Asynchronous Transaction barriers[1]

1: Since Hopper


ASYNCHRONOUS SIMT PROGRAMMING
All threads have arrived as well as data. Consumers are unblocked.
Thread Block 0 - Producer (2 threads)
Shared Memory
Thread 0, Producer
Thread 1, Producer
memcpy_async()
Arrive
memcpy_async()
Arrive
Thread Block 1 - Consumer (2 threads)
Barrier
Shared Memory
Thread 0, Consumer
Thread 1, Consumer
Arrive
Arrive
Wait


Ninjas Use Single-Wave Kernels
Don't map threads to data; map data to threads
Image suggests 16x16 tiles = 256 blocks
Hardware suggests √132 SMs = 11.5 x 11.5 tiles
1024x1024 image divided into 16x16 tiles
H100 with 132 SMs

Mmmmmmm



MMmmmm

Context, A goes normal from top to bottom. But B does bottom to the top part. So wen A finishes it can reuse data in cache

TOMORROW – document PTX

Miguel

- 1

Multiply accumulate add

C = C + A*B

C += A*B

For i

For j

For k

C[i, j] += A[i, k] * B[k, j]

Warp mma

Warp group mma

- collective operations (warp collectives) – + Explain an use case.

Warp – 32 threads

SIMT

Ballot

https://github.com/burtscher/ECL-MST -> Dynamic parallelism

https://docs.nvidia.com/cuda/cuda-c-programming-guide

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html 🔥

Isaac

- DPX
- TMA

https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#parallel-synchronization-and-communication-instructions

TMA

Works on memory barrier (mbarrier). Note there also exist a member which is different to **mbar**

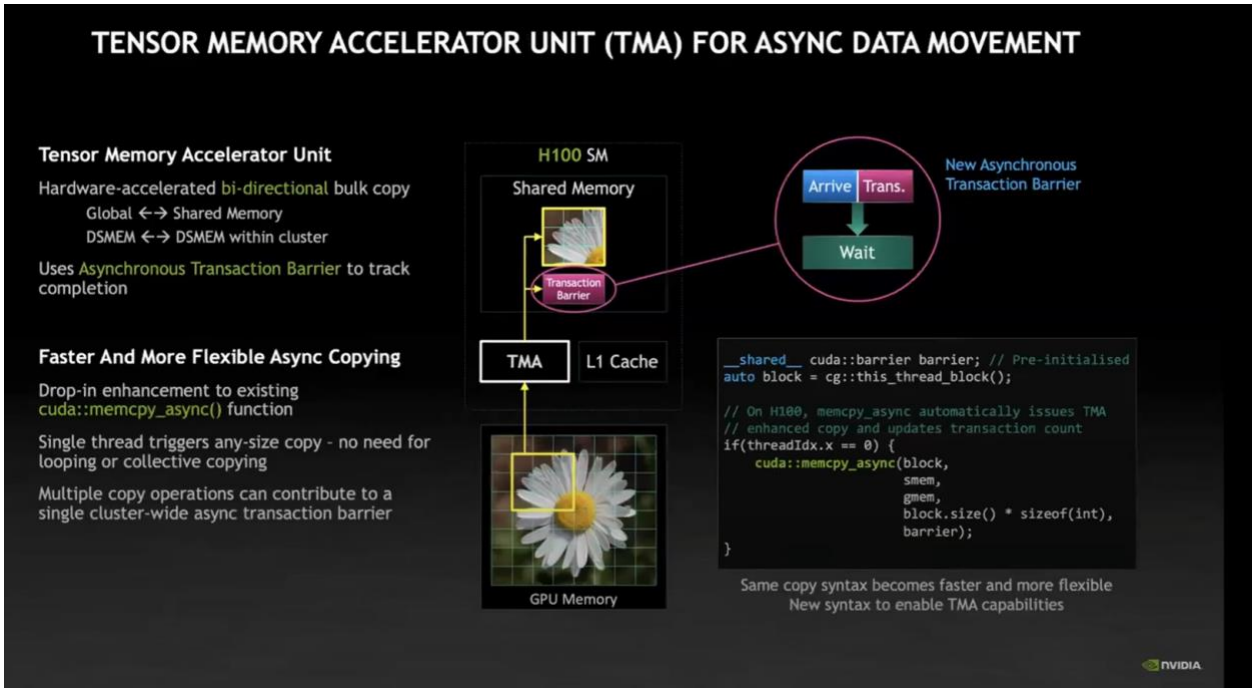
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#parallel-synchronization-and-communication-instructions-mbarrier

https://carbon.now.sh/?bg=rgba%28255%2C255%2C255%2C1%29&t=dracula-pro&wt=none&l=auto&width=680&ds=false&dsyoff=20px&dsblur=68px&wc=false&wa=true&pv=56px&ph=56px&ln=false&fl=1&fm=Hack&fs=14px&lh=133%25&si=false&es=2x&wm=false&code=.shared%2520.b64%2520barrierMem%250A.reg%2520.b32%2520%2525r1%250A%250Ambarrier.init.b64%2520%255BbarrierMem%255D%252C%2520%2525r1%250A...%250A%2523%2520TODO%2520put%2520operations%2520on%2520barrier%250A...%250Ambarrier.inval.b64%2520%255BbarrierMem%255D

shared has additional submodifiers

.shared::cta (default when using .shared)

.shared::cluster



1.4.1.2. Tensor Memory Accelerator

The Hopper architecture builds on top of the asynchronous copies introduced by NVIDIA Ampere GPU architecture and provides a more sophisticated asynchronous copy engine: the Tensor Memory Accelerator (TMA).

TMA allows applications to transfer 1D and up to 5D tensors between global memory and shared memory, in both directions, as well as between the shared memory regions of different SMs in the same cluster (refer to Thread Block Clusters). Additionally, for writes from shared memory to global memory, it allows specifying element wise reduction operations such as add/min/max as well as bitwise and/or for most common data types.

This has several advantages:

- Avoids using registers for moving data between the different memory spaces.
- Avoids using SM instructions for moving data: a single thread can issue large data movement instructions to the TMA unit. The whole block can then continue working on other instructions while the data is in flight and only wait for the data to be consumed when actually necessary.
- Enables users to write warp specialized codes, where specific warps specialize on data movement between the different memory spaces while other warps only work on local data within the SM.

This feature will be exposed through `cuda::memcpy_async` along with the `cuda::barrier` and `cuda::pipeline` for synchronizing data movement.

• viaddmax s16x2(a,b,c)

max(a + b, c)

• vimax3 s16x2 relu(a,b,c)

relu(max(max(a, b), c))

5 BINARY FORMATS

5.1 PTX (PARALLEL THREAD EXECUTION)

5.1.1 mbarrier

5.1.1.1 Overview

Phase

Pending arrival count

Next phase expected arrivals

Pending transaction count

5.1.1.2 Lifecycle

A mbarrier is an **opaque object** stored within **shared memory** with data layout of **b64** and **must be 8 byte aligned**. It can be initialized and invalidated with *mbarrier.init and mbarrier.inval*

5.1.1.3 Operations

The supported operations are:

- *mbarrier.expect_tx*
- *mbarrier.complete_tx*
- *mbarrier.arrive*
- *mbarrier.arrive_drop*
- *mbarrier.test_wait*
- *mbarrier.try_wait*
- *mbarrier.pending_count*
- *cp.async.mbarrier.arrive*

Parallel Thread Execution

CTA – cooperative thread array

WARP

Cluster of CTA -shared mem

5.1.1.4 Example

```
.shared .b64 barrierMem
.reg .b32 %r1

mbarrier.init.b64 [barrierMem], %r1
...
# TODO put operations on barrier
...
mbarrier.inval.b64 [barrierMem]
```

5.2 SASS (SOURCE AND ASSEMBLY)

Source and Assembly

6 GPU ARCHITECTURES

6.1 HOPPER SM_90

6.2 BLACKWELL SM_100

ST.ASYNC

CP.ASYNC.BULK

CP.REDUCE.ASYNC.BULK

CP:ASYNC.BULK.TENSOR

RED.ASYNC

Single thread does

mbarrier.init

mbarrier.arrive.expect_tx

cp.async.bulk.tensor

__syncthreads() to ensure thread convergence

mbarrier.try_wait

FENCE -> guarantee memory ordering

8.6. Proxies

A *memory proxy*, or a *proxy* is an abstract label applied to a method of memory access. When two memory operations use distinct methods of memory access, they are said to be different *proxies*.

Memory operations as defined in Operation types use *generic* method of memory access, i.e. a *generic proxy*. Other operations such as textures and surfaces all use distinct methods of memory access, also distinct from the *generic* method.

A *proxy fence* is required to synchronize memory operations across different *proxies*. Although virtual aliases use the *generic* method of memory access, since using distinct virtual addresses behaves as if using different *proxies*, they require a *proxy fence* to establish memory ordering.

1. Fill SMEM
2. Trigger fence on async proxy
3. cp.async.bulk
4. cp.async.bulk-group // commit prior **cp.async.bulk** ops
5. cp.async.bulk.wait_group{.read} // wait until completion of all ops within the **async-group**

9.7.10.24.1. Completion Mechanisms for Asynchronous Copy Operations

A thread must explicitly wait for the completion of an asynchronous copy operation in order to access the result of the operation. Once an asynchronous copy operation is initiated, modifying the source memory location or reading from the destination memory location before the asynchronous operation completes, will cause unpredictable results.

This section describes two asynchronous copy operation completion mechanisms supported in PTX: Async-group mechanism and mbarrier-based mechanism.

Async-group mechanism

When using the async-group completion mechanism, the issuing thread specifies a group of asynchronous operations, called *async-group*, using a *commit* operation and tracks the completion of this group using a *wait* operation. The thread issuing the asynchronous operation must create separate *async-groups* for bulk and non-bulk asynchronous operations.

A *commit* operation creates a per-thread *async-group* containing all prior asynchronous operations initiated by the executing thread but none of the asynchronous operations following the commit operation. A committed asynchronous operation belongs to a single *async-group*.

When an *async-group* completes, all the asynchronous operations belonging to that group are complete and the executing thread that initiated the asynchronous operations can read the result of the asynchronous operations. All *async-groups* committed by an executing thread always complete in the order in which they were committed. There is no ordering between asynchronous operations within an *async-group*.

A typical pattern of using *async-group* as the completion mechanism is as follows:

- Initiate the asynchronous operations.
- Group the asynchronous operations into an *async-group* using a *commit* operation.
- Wait for the completion of the async-group using the wait operation.
- Once the *async-group* completes, access the results of all asynchronous operations in that *async-group*.

Mbarrier-based mechanism

A thread can track the completion of one or more asynchronous operations using the current phase of an *mbarrier object*. When the current phase of the *mbarrier object* is complete, it implies that all asynchronous operations tracked by this phase are complete, and all threads participating in that *mbarrier object* can access the result of the asynchronous operations.

The *mbarrier object* to be used for tracking the completion of an asynchronous operation can be either specified along with the asynchronous operation as part of its syntax, or as a separate operation. For a bulk asynchronous operation, the *mbarrier object* must be specified in the asynchronous operation, whereas for non-bulk operations, it can be specified after the asynchronous operation.

A typical pattern of using mbarrier-based completion mechanism is as follows:

- Initiate the asynchronous operations.
- Set up an *mbarrier object* to track the asynchronous operations in its current phase, either as part of the asynchronous operation or as a separate operation.
- Wait for the *mbarrier object* to complete its current phase using `mbarrier.test_wait` or `mbarrier.try_wait`.
- Once the `mbarrier.test_wait` or `mbarrier.try_wait` operation returns `True`, access the results of the asynchronous operations tracked by the *mbarrier object*.

https://carbon.now.sh/?bg=rgba%28255%2C255%2C255%2C1%29&t=dracula-pro&wt=none&l=auto&width=414&ds=false&dsyoff=20px&dsblur=68px&wc=false&wa=false&pv=0px&ph=0px&ln=false&fl=1&fm=Hack&fs=14px&lh=133%25&si=false&es=2x&wm=false&code=if%2520%28idx%2520%253D%253D%25200%29%2520%257B%250A%2520%2520mbarrier.init%250A%2520%2520mbarrier.arrive.expect_tx%2520%252F%252F%252013.5%2520GB%250A%2520

%2520%250A%2520%2520%252F%252F%2520trigger%2520async%2520mem-op%250A%2520%2520cp.async.bulk%2520%252F%252F%25201GB%2520%252F%252F%2520inmediate%250A%2520%2520cp.async.bulk%2520%252F%252F%25202%2520GB%2520%252F%252F%2520inmediate%250A%2520%2520cp.async.bulk%2520%252F%252F%252010%2520GB%2520%252F%252F%2520inmediate%250A%2520%2520cp.async.bulk%2520%252F%252F%252050 0%2520MB%2520%252F%252F%2520inmediate%250A%257D%250A%250A%252F%252F%2520COMPUTE%2520%250A%250A%252F%252F%2520COMPUTE%250A%250A%252F%252F%2520COMPUTE%250A%250A%252F%252F%2520ensure%2520thread%2520convergence%250A__syncthreads%28%29%250Ambarrier.try_wait%250A

```
if (idx == 0) {
  mbarrier.init
  mbarrier.arrive.expect_tx // 13.5 GB

  // trigger async mem-op
  cp.async.bulk // 1GB // inmediate
  cp.async.bulk // 2 GB // inmediate
  cp.async.bulk // 10 GB // inmediate
  cp.async.bulk // 500 MB // inmediate
}

// COMPUTE

// COMPUTE

// COMPUTE

// TOTAL CP: 1:30

// TOTAL COMPUTE: 1 m

// CP CP CP // 1:30
// CMP CMP CMP // 1m

// |||||||||| CP
// ||||||x    COMPUTE
// 30


// ensure thread convergence
__syncthreads()
mbarrier.try_wait // 30s

// COMPUTE DATA LOADED PREVIOUSLY
```