Lesson 1 - Introduction to H100

The specs

- Released in September 2022 | based on TSMC 4N process node
- Based on Hopper Architecture
- 2 Form factor: PCIe(300W) and SXM(700W)
- 80 GB HBM 3 Memory (3.35 TB/s)
- 132 SMs
- 528 Tensor cores (4 Per SM)
- 80B transistors on a custom 4N node

Biggest improvement is introduction of Asynchronous features

The TMA

Each SM has a TMA Unit. This is the new unit to offload the tensor copy operations from the SMs.

Created to speed up memory bound operations

Previously we had to do per thread math, looping over regions, launch multiple instructions to work with global memory -> shared memory transfers

With TMA, we get descriptors to do things asynchronously, A single thread launches the whole operation for data movement.

The hardware handles stride/offset/bounds and the data movement in the background

Synchronous copies vs Async copies

A copy is synchronous if the thread (or warp) that issues it cannot make forward progress until the copy is done. It has a simple mental model and you know that once the control gets back to thread the operation is done

A copy is asynchronous if issuing it only starts the transfer, and the issuer can keep doing other work while the copy is in flight, then later wait/check for completion right before using the data.

This is awesome because it enables overlap (hide memory latency under compute). Better for pipelined kernels (GEMM/attention): while computing tile i, you fetch tile i+1.

The only problem is increased complexity with bookkeeping

The fourth generation Tensor Cores

4 tensor cores per SM

Warp Group Matrix Multiply Accumulate (wgmma) - 4 warps work on a MMA

Sparse tensor cores which can do really fast MMA operations on structured sparse data(data which has zeros in a pre-defined layout)

Special instructions and support for fp8. Give 1,979 TFLOPS (big improvement over previous generation of GPUs)

1024 FLOPS/cycle (Dense) for fp16 and 2048 FLOPS/cycle for FP8.

Other things

- Transformer Engine
- 4th-Gen NVLink Interconnect
- DPX Instructions
- Massive 50MB L2 Cache

The whole organization of a GPU

A whole GPU die is divided into -

- Nvidia Gigathread Engine
- 8 GPCs
- HBM3 stacks
- HBM3 Memory controllers
- PCI express 5.0 host interface
- Nvlink switches, ports, hub
- L2 cache slices

Gigathread Engine

The GigaThread Engine is the hardware that takes a kernel launch and hands out thread blocks (CTAs) to SMs.

It tracks which CTAs are not yet started, which are running, which finished. When an SM has capacity for another CTA, the GigaThread Engine (and related front-end logic) assigns the next CTA to that SM.

It enforces constraints like occupancy limits, and on Hopper it also understands clusters (more below).

The HBM3

Off-chip device memory. 80GB at 3.35 TB/sec

Divided in 5 stacks (actually 6 but one is disabled for yield reasons)

5120 bus width which enables TMA to move 128 bytes of data at once

Connected via 10 independent 512-bit memory controllers

SM ⟶ (L1D / coalescer) ⟶ L2 ⟶ memory partition/crossbar ⟶ memory controller ⟶ HBM stack

A GPU memory controller block is doing the critical work:

DRAM protocol + scheduling (activate/precharge/read/write timing, reordering to maximize row hits). mapping addresses ⟶ channels/banks/rows/cols

L2 Cache

50 MB, divided into 25 MB partitions

L2 cache is partitioned, and SMs in a given GPC have a “closer” path to the L2 partition they’re directly connected to. When your accesses mostly hit lines that live in that nearby partition, you get better effective latency/bandwidth

128 byte lines and 32 byte sectors. A memory request can touch 1–4 sectors of a single 128B line. You can be “uncoalesced” in a way that explodes sectors/request.

It absorbs and coalesces messy, small writes from the SMs (via L1) and turns them into clean, large, efficient writes to HBM.

GPC - Graphics Processing cluster

It is a group of 18 SMs.

Each GPC is connected to its own chunk of L2 cache. When there is a request to fetch something to shared memory, the data flows from HBM -> L2 -> L1 and GPC's handle all that

GPCs also enable the use of distributed shared memory between the SMs. There is no DSMEM outside a GPC

They have TPCs, which basically hold 2 SMs and their only job is to make sure that the communication between the SMs inside the TPC is really fast. This connection is for DSMEM

NvLink 4.0

18 NVLink 4.0 lanes which give 900 GB/s of GPU - GPU bandwidth

organized into 9 sub-links, each providing 100 GB/s bidirectional bandwidth(50GB/s per direction)

PAM4 signaling which basically makes use of 4 different voltage levels instead of 2. This doubles the double the bits per second between GPU

You can directly connect to 8 GPUs and use NVSwitch fabric to connect up to 256 GPUs

Connects directly with TMA for bulk copies in a single DGX Node

The SMs

The fundamental execution unit of the GPU which executes thread blocks of a CUDA kernels. Following are its components -

- FP32 CUDA Cores, Int/FP64 units
- 4th Generation Tensor Cores
- Shared Memory/L1 cache
- L1 instruction cache
- Warp Scheduler
- Dispatch Units
- Registers
- L0 Instruction cache

Quadrant/SMSP

Each SM is divided into 4 identical sub-divisions called Quadrants or SMSPs

Each quadrant have 16 fp64 units, 32 fp32 cuda cores, 16 int32 units, 1 tensor core, 8 load/store unit, 4 Special function unit

Apart from these each quadrants have their own warp schedulers, L0 cache, dispatch unit and register file

Each quadrant can dispatch instructions to its local units every cycle, juggling up to 16 warps per quadrant

Special functional Units (SFUs)

16 SFUs per SM (4 per SMSP/quadrant). Each SFU can issue 1 instruction per cycle, giving you 16 operations/cycle per SM

SFUs handle complex mathematical functions like sine, cosine, logarithms, exponentials, square roots, and reciprocals that would be computationally expensive on cores

SFUs sacrifice some precision for massive throughput gains, using polynomial approximations and lookup tables combined with interpolation

SFUs stall if all the threads call math functions

Load/Store Units

They execute the per thread memory instructions like loads, stores, and atomics

Each quadrant contains 8 dedicated load/store units, totaling 32 LSUs per SM and they directly connect to L1, L2

When your warp executes ld/st/atom, the LSU coalesces the 32 thread addresses, forms cache-line/sector requests, and queries the L1 data cache. If it hits, data comes back quickly to registers; if it misses, the request goes onward to L2/DRAM and may fill back into L1.

When accesses are coalesced, the LSU merges the warp's 32 addresses into the fewest possible cache-lines, so it issues fewer requests to L1/L2, reducing replays/stalls and boosting effective bandwidth.

Unified Shared Memory + L1 cache

256 KB size 33 TB/s bandwidth per SM divided into 32 banks, each 32 bits (4 bytes) wide, the shared memory size is 228kb

128B/cycle for loads and stores, conflict-free. TMA async copies hit near-peak bandwidth and overlap compute. Banks interleave consecutive 4-byte words across sequential banks.

L1 cache acts as a coalescing buffer: it gathers the warp's requested data and delivers it efficiently

Cache line = 128 bytes, Sector = 32 bytes, The cache can be filled sector-by-sector

Maximum configurable shared memory capacity is 228 KB per SM

per-thread-block limit is 227 KB due to a 1 KB reservation by CUDA

More on shared memory

Byte address contains information about banks, bankrow (row inside that bank) etc

Static shared memory (compile-time array declarations) is architecturally limited to 48 KB to maintain compatibility with older compute capabilities . To exceed this limit you need to use dynamic shared memory using

```
extern __shared__
```

Optimal tile size is typically 64-128 KB per block for GEMMs. Massive transfers hurt the latency hiding

So if you increase shared memory up, you’re shrinking the space left for L1 caching and vice versa.

Int units

These are the standard integer arithmetic logic units used for memory addressing, loop controls, and general integer math.

Architecture: There are 64 INT32 Cores per SM.

Concurrent Execution: These units can execute simultaneously with floating-point datapaths, allowing the GPU to calculate memory addresses (INT) while processing data (FP) without stalling.

Total Count (SXM5): 8,448 units.

FP32 CUDA Cores

The "CUDA Core" in NVIDIA marketing terminology generally refers to the Single Precision (FP32) floating-point unit. In hopper, these are the primary for general-purpose graphics and compute shading.

Each Streaming Multiprocessor (SM) in the H100 contains 128 FP32 CUDA Cores.

Total Count (SXM5): 16,896 active cores (132 SMs enabled).

These handle standard 32-bit floating-point calculations, essential for most scientific simulations and the "default" precision for general compute tasks.

FP64 Units

The H100 features dedicated FP64 cores separate from the FP32 cores. There are 64 FP64 Cores per SM.

Generally used for scientific computing etc.

Registers

Every thread gets a private set of on-chip registers. They have Fastest bandwidth lowest latency, 256 KB per SM. Up to 128 reads/writes per cycle per SM.

Registers are a frequent limiter. If your kernel uses R regs/thread, then max resident threads is roughly floor(65536 / R) (then rounded/limited by block granularity and other limits). At 128 registers/thread, you're limited to 512 threads/block.

32-bit is the base unit for registers. The hardware register file is effectively 32-bit slots. When working with FP16 or FP8 we have to use packed datatypes to fit 2/4 elements into a single register

More on registers

Register file accesses are 30-50x more power-efficient than shared memory accesses and 1000x+ more efficient than HBM3 accesses!

Register usage is determined at compile time, not runtime

When the compiler runs out of registers, then it spills the data into local memory which is much slower than registers. CUDA 13.0 added support for spilling into shared memory first, and only falling back to local memory if shared memory is insufficient.

Even a small change (say 63 ⟶ 65 registers/thread) can drop resident warps because allocation is rounded to an internal granularity. You'll see fewer active warps and lower performance

Let:

R = registers per thread (32-bit regs, from ptxas)

T = threads per block

RegSM = total 32-bit registers available per SM

warpRegs = 32 * R (one warp has 32 threads)

Then:

Regs needed per block ≈ T * R (plus hardware/allocator rounding)

Max resident blocks from regs = floor(RegSM / (T * R))

Active warps = (resident blocks) * (T / 32)

WGMMA Register Layout

Tensor Core operations require specific register layouts:

Matrix A: Stored in shared memory or registers

Matrix B: Must always be in shared memory (SMEM), never registers

Matrix C: Distributed across 4 warps (128 threads) in registers

When working with fp8/fp16 WGMMA expects the data to be packed into registers i.e 2 FP16 in a single 32 bit register.

L0 instruction cache

The L0 I-Cache acts as a micro-buffer for the instruction stream. Its purpose is to filter requests to the larger L1 cache, allowing the scheduler to issue instructions at the speed of the hardware (every clock cycle) without saturating the shared memory bandwidth.

The L0 cache is designed for speed, not capacity. It can only hold a very small number of instructions (typically fitting just a few tight loops). Aggressive loop unrolling, Inlining functions might exceed the L0 cache size.

Code running on SMSP 0 cannot use the L0 cache of SMSP 1.

What is a warp

A warp is a group of 32 threads that:

- Are created together from a thread block (threads 0-31 = warp 0, 32-63 = warp 1)
- Share a warp scheduler's attention on a per-cycle basis
- Have private registers but shared instruction stream (mostly)

Why 32 Threads?

Balance between control of threads/work per instruction

Memory coalescing lines up cleanly with cache/bus granularities. With 32 threads, a lot of common access patterns become naturally “nice”

What is a warp scheduler

Unit which manages instruction issues.

Selects ready warps for instruction issue each cycle

Handles warp-level branching and divergence

Manages a scoreboard which tracks which warps are truly ready

Ensures source registers are read from the register file or forwarded from the pipeline before issue

It takes care of structural limits like how many long latency ops a warp can have in flight, pipe availability for execution etc

How warps execute

Warp scheduler scans all the 16 warps assigned to the SMSP and picks 1 warp which is ready

Fetch instructions -> Send the selected warp ID + Program counter to the dispatch unit and issues instructions, routing them to appropriate execution pipelines (CUDA cores, Tensor cores, load/store units)

Waits for dispatch feedback

The warp scheduler would always try to launch as many instructions as possible and try to turn all the free warp busy

To maximize hardware utilization, you generally want your block dimensions to be multiples of 4 warps (e.g., 128 threads) so that work is evenly balanced across the four sub-partitions

The Dispatch unit

responsible for the physical act of sending warp’s operations to the appropriate functional units each clock cycle this is done by issuing instructions to execution pipelines

Execution pipelines are physical datapaths to functional units via dispatch ports

For shared SM resources (LSUs, SFUs), the dispatch unit sends instructions directly to each execution unit's dedicated instruction queue/pipeline.

To go further we need to understand what a dispatch port is

The dispatch port

The physical connection point through which the dispatch unit sends instructions to specific execution pipelines(tensor cores, cuda cores etc)

Multiple execution pipelines compete for the same dispatch port, meaning they cannot receive instructions simultaneously

Only one instruction can use a shared dispatch port per clock cycle; the dispatch unit must serialize issuance

Port is only needed for instruction launch, not the full execution duration

L1 Instruction cache

Its job is to buffer executable machine code (SASS instructions) to keep the Warp Schedulers constantly fed.

It decouples the execution units from the memory hierarchy. Without it, the SM would have to fetch instructions from the L2 cache or HBM3 memory (which takes hundreds of cycles), causing massive pipeline stalls

It is optimized to handle jumps, loops, and function calls. When a kernel loops (common in matrix multiplication), the instructions are served directly from the L1 I-Cache, saving bandwidth and power.

How are thread blocks distributed across SMs and SMSPs

The gigathread engine selects the SM which have enough resources to handle your thread block.

The hardware does not keep a single block inside a single SMSP unless the block size is really small. Hardware divides the block into warps and then assignes the warps in a round robin fashion.

Whenever we use __syncthreads() the synchronization is handled by the barrier logic in the shared memory which all the SMSPs share. Other warps are put to sleep until each of the warps have completed their execution.

PCIe 5.0 host interface

It acts as the primary 128 GB/s data highway connecting the H100 to the host CPU and system memory.

It serves as the critical physical bridge connecting the GPU to Network Interface Cards (NICs). It enables GPUDirect RDMA, allowing network cards to read GPU memory directly without burdening the CPU.

Conclusion

The defining characteristic of the H100 is the shift to fully asynchronous execution. We moved from simple synchronous copies to using the TMA to handle data movement in the background, allowing the SMs to focus entirely on compute.

Specialization: The architecture relies on specialized units for every bottleneck:

- TMA for memory bandwidth.
- 4th Gen Tensor Cores for Matrix Math.
- SFUs for complex math

Data flows efficiently from HBM3 (High Bandwidth) → L2 Cache (50MB Unified) → Shared Memory (low latency) → Registers (highest speed).