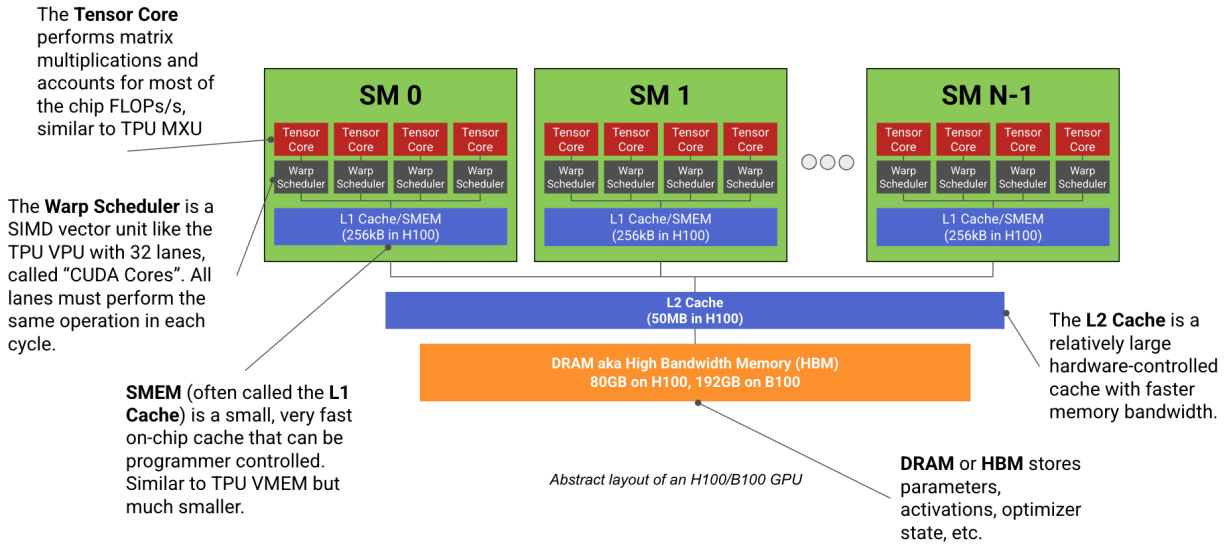
How to Think About GPUs

jax-ml.github.io/scaling-book/gpus

Authors Affiliation Jacob Austin Google DeepMind Swapnil Patil Adam Paszke Reiner Pope MatX Published Aug. 18, 2025



We love TPUs at Google, but GPUs are great too. This chapter takes a deep dive into the world of NVIDIA GPUs – how each chip works, how they're networked together, and what that means for LLMs, especially compared to TPUs. This section builds on Chapter 2 and Chapter 5, so you are encouraged to read them first.

What Is a GPU?

A modern ML GPU (e.g. H100, B200) is basically a bunch of compute cores that specialize in matrix multiplication (called **Streaming Multiprocessors** or **SMs**) connected to a stick of fast memory (called **HBM**). Here's a diagram:

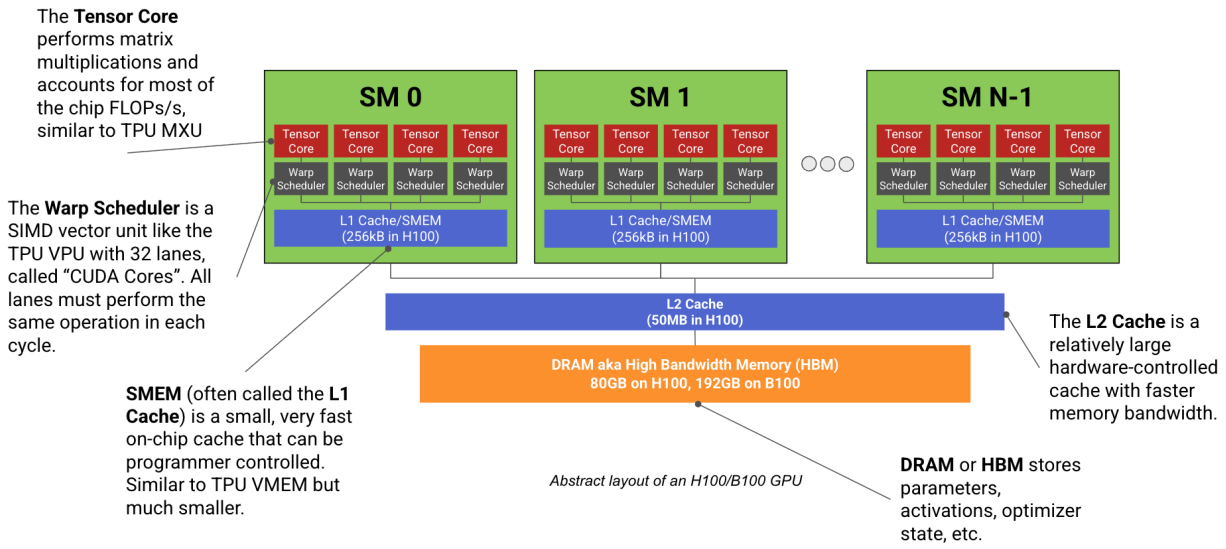


Figure: a diagram showing the abstract layout of an H100 or B200 GPU. An H100 has 132 SMs while a B200 has 148. We use the term 'Warp Scheduler' somewhat broadly to describe a set of 32 CUDA SIMD cores *and* the scheduler that dispatches work to them. Note how much this looks like a TPU!

Each SM, like a TPU's Tensor Core, has a dedicated matrix multiplication core (unfortunately also called a **Tensor Core**

[1] The GPU Tensor Core is the matrix multiplication sub-unit of the SM, while the TPU TensorCore is the umbrella unit that contains the MXU, VPU, and other components.
), a vector arithmetic unit (called a **Warp Scheduler**
[2] NVIDIA doesn't have a good name for this, so we use it only as the best of several bad options. The Warp Scheduler is primarily the unit that dispatches work to a set of CUDA cores, but we use it here to describe the control unit and the set of cores it controls.
), and a fast on-chip cache (called **SMEM**). Unlike a TPU, which has at most 2 independent "Tensor Cores", a modern GPU has more than 100 SMs (132 on an H100). Each of these SMs is much less powerful than a TPU Tensor Core but the system overall is more flexible. Each SM is more or less totally independent, so a GPU can do hundreds of separate tasks at once.
[3] Although SMs are independent, they are often forced to coordinate for peak performance because they all share a capacity-limited L2 cache.
Let's take a more detailed view of an H100 SM:

Figure: a diagram of an H100 SM (source) showing the 4 *subpartitions*, each containing a Tensor Core, Warp Scheduler, Register File, and sets of CUDA Cores of different precisions. The 'L1 Data Cache' near the bottom is the 256kB SMEM unit. A B200 looks similar, but adds a substantial amount of Tensor Memory (TMEM) for feeding the bulky Tensor Cores.

Each SM is broken up into 4 identical quadrants, which NVIDIA calls **SM subpartitions**, each containing a Tensor Core, 16k 32-bit registers, and a SIMD/SIMT vector arithmetic unit called a Warp Scheduler, whose lanes NVIDIA calls **CUDA Cores**. The core component of each partition is arguably the Tensor Core, which performs matrix multiplications and makes up the vast majority of its FLOPs/s, but it's not the only component worth noting.

- **CUDA Cores:** each subpartition contains a set of ALUs called CUDA Cores that do SIMD/SIMT vector arithmetic. Each subpartition contains 32 fp32 cores (and a smaller number of int32 and fp64 cores) that all execute the same instruction in each cycle. Like the TPU's VPU, CUDA cores are responsible for ReLUs, pointwise vector operations, and reductions (sums).

  [4] Historically, before the introduction of the Tensor Core, the CUDA cores were the main component of the GPU and were used for rendering, including ray-triangle intersections and shading. On today's gaming GPUs, they still do a bulk of the rendering work, while TensorCores are used for up-sampling (DLSS), which allows the GPU to render at a lower resolution (fewer pixels = less work) and upsample using ML.
- **Tensor Core (TC):** each subpartition has its own Tensor Core, which is a dedicated matrix multiplication unit like a TPU MXU. The Tensor Core represents the vast majority of the GPUs FLOPs/s (e.g. on an H100, we have 990 bf16 TC TFLOP/s compared to just 66 TFLOPs/s from the CUDA cores).
  - 990 bf16 TFLOPs/s with 132 SM running at 1.76GHz means each H100 TC can do `7.5e12 / 1.76e9 / 4 ~ 1024` bf16 FLOPs/cycle, roughly an 8x8x8 matmul. [5] NVIDIA doesn't share many TC hardware details, so this is more a guess than definite fact – certainly, it doesn't speak to how the TC is implemented. We know that a V100 can perform 256 FLOPs/TC/cycle. An A100 can do 512, H100 can do 1024, and while the B200 details aren't published, it seems likely it's about 2048 FLOPs/TC/cycle, since `2250e12 / (148 * 4 * 1.86e9)` is about 2048. Some more details are confirmed here.
  - Like TPUs, GPUs can do lower precision matmuls at higher throughput (e.g. H100 has 2x fp8 FLOPs/s vs. fp16). Low-precision training or serving can be significantly faster.
  - Each GPU generation since Volta has increased the TC size over the previous generation (good article on this). With B200 the TC has gotten so large it can no longer fit its inputs in SMEM, so B200s introduce a new memory space called TMEM. [6] In Ampere, the Tensor Core could be fed from a single warp, while in Hopper it requires a full SM (warpgroup) and in Blackwell it's fed from 2 SMs. The matmuls have also become so large in Blackwell that the arguments (specifically, the accumulator) no longer fit into register memory/SMEM, so Blackwell adds TMEM to account for this.

CUDA cores are much more flexible than a TPU's VPU: GPU CUDA cores use what is called a SIMT (*Single Instruction Multiple Threads*) programming model, compared to the TPU's SIMD (*Single Instruction Multiple Data*) model. Like ALUs in a TPU's VPU, CUDA cores within a subpartition must execute the same operation in each cycle (e.g. if one core is adding two floats, then every other CUDA core in the subpartition must also do so). Unlike the VPU, however, each CUDA core (or "thread" in the CUDA programming model) has its own instruction pointer and can be *programmed* independently. When two threads in the same warp are instructed to perform different operations, you effectively do *both* operations, masking out the cores that don't need to perform the divergent operation.

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

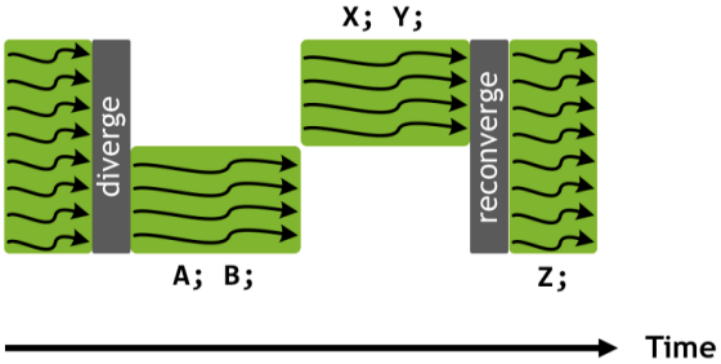


Figure: an example of warp divergence within a set of threads (source). White spaces indicate stalls of at least some fraction of the physical CUDA cores

This enables flexible programming at the thread level, but at the cost of silently degrading performance if warps diverge too often. Threads can also be more flexible in what memory they can access; while the VPU can only operate on contiguous blocks of memory, CUDA cores can access individual floats in shared registers and maintain per-thread state.

CUDA core scheduling is also more flexible: SMs run a bit like multi-threaded CPUs, in the sense that they can "schedule" many programs (**warps**) concurrently (up to 64 per SM) but each *Warp Scheduler* only ever executes a single program in each clock cycle.

[7] Warps scheduled on a given SM are called "resident".
The Warp Scheduler automatically switches between active warps to hide I/O operations like memory loads. TPUs are generally single threaded by comparison.

Memory

Beyond the compute units, GPUs have a hierarchy of memories, the largest being HBM (the main GPU memory), and then a series of smaller caches (L2, L1/SMEM, TMEM, register memory).

- **Registers:** Each subpartition has its own register file containing 16,384 32-bit words on H100/B200 (`4 * 16384 * 4 = 256kB` per SM) accessible by the CUDA cores.
  Each CUDA core can only access up to 256 registers at a time, so although we can schedule up to 64 "resident warps" per SM, you can only fit 8 (`256e3 / (8 * 32 * 256)`) at a time if each thread uses 256 registers.
- **SMEM (L1 Cache):** each SM has its own 256kB on-chip cache called SMEM, which can either be programmer controlled as "shared memory" or used by the hardware as an on-chip cache. SMEM is used for storing activations and inputs to TC matmuls.

- **L2 Cache:** all SMs share
  [8] Technically, the L2 cache is split in two, so half the SMs can access 25MB a piece on an H100. There is a link connecting the two halves, but at lower bandwidth.
  a relatively large ~50MB L2 cache used to reduce main memory accesses.
    - This is similar in size to a TPU's VMEM but it's **much** slower and isn't programmer controlled. This leads to a bit of "spooky action at a distance" where the programmer needs to modify memory access patterns to ensure the L2 cache is well used.
      [9] The fact that the L2 cache is shared across all SMs effectively forces the programmer to run the SMs in a fairly coordinated way anyway, despite the fact that, in principle, they are independent units.
    - NVIDIA does not publish the L2 bandwidth for their chips, but it's been measured to be about 5.5TB/s. Thus is roughly 1.6x the HBM bandwidth but it's full-duplex, so the effective bidirectional bandwidth is closer to 3x. By comparison, a TPU's VMEM is 2x larger *and* has much more bandwidth (around 40TB/s).
- **HBM:** the main GPU memory, used for storing model weights, gradients, activations, etc.
    - The HBM size has increased a lot from 32GB in Volta to 192GB in Blackwell (B200).
    - The bandwidth from HBM to the CUDA Tensor Core is called HBM bandwidth or memory bandwidth, and is about 3.35TB/s on H100 and 9TB/s on B200.

Summary of GPU specs

Here is a summary of GPU specs for recent models. The number of SMs, clock speed, and FLOPs differ somewhat between variants of a given GPU. Here are memory capacity numbers:

GPU	Generation	Clock Speed	SMs/chip	SMEM capacity/SM	L2 capacity/chip	HBM capacity/chip
V100	Volta	1.25GHz/1.38HGz	80	96kB	6MB	32GB
A100	Ampere	1.10GHz/1.41GHz	108	192kB	40MB	80GB
H100	Hopper	1.59GHz/1.98GHz	132	256kB	50MB	80GB
H200	Hopper	1.59GHz/1.98GHz	132	256kB	50MB	141GB
B200	Blackwell	?	148	256kB	126MB	192GB

All generations have 256kB of register memory per SM. Blackwell adds 256kB of TMEM per SM as well. Here are the FLOPs and bandwidth numbers for each chip:

GPU	Generation	HBM BW/chip	FLOPs/s/chip (bf16/fp16)	FLOPs/s/chip (fp8/int8)	FLOPs/s/chip (fp4)
V100	Volta	9.0e11	—	—	—
A100	Ampere	2.0e12	3.1e14	6.2e14	—
H100	Hopper	3.4e12	9.9e14	2.0e15	—
H200	Hopper	4.8e12	9.9e14	2.0e15	—
B200	Blackwell	8.0e12	2.3e15	4.5e15	9.0e15

We exclude B100 since it wasn't mass-produced.

[10] While NVIDIA made a B100 generation, they were only briefly sold and produced, allegedly due to design flaws that prevented them from running close to their claimed specifications. They struggled to achieve peak FLOPs without throttling due to heat and power concerns.
Some specs depend slightly on the precise version of the GPU, since NVIDIA GPUs aren't as standard as TPUs.
Here's a helpful cheat sheet comparing GPU and TPU components:

GPU	TPU	What is it?
Streaming Multiprocessor (SM)	Tensor Core	Core "cell" that contains other units
Warp Scheduler	VPU	SIMD vector arithmetic unit
CUDA Core	VPU ALU	SIMD ALU
SMEM (L1 Cache)	VMEM	Fast on-chip cache memory
Tensor Core	MXU	Matrix multiplication unit
HBM (aka GMEM)	HBM	High bandwidth high capacity memory

GPUs vs. TPUs at the chip level

GPUs started out rendering video games, but since deep learning took off in the 2010s, they've started acting more and more like dedicated matrix multiplication machines – in other words, more like TPUs.

[11] Before the deep learning boom, GPUs ("Graphics Processing Units") did, well, graphics – mostly for video games. Video games represent objects with millions of little triangles, and the game renders (or "rasterizes") these triangles into a 2D image that gets displayed on a screen 30-60 times a second (this frequency is called the framerate). Rasterization involves projecting these triangles into the coordinate frame of the camera and calculating which triangles overlap which pixels, billions of times a second. As you can imagine, this is very expensive, and it's just the beginning. You then have to color each pixel by combining the colors of possibly several semi-opaque triangles that intersect the ray. GPUs were designed to do these operations extremely fast, with an eye towards versatility; you need to run many different GPU workloads (called "shaders") at the same time, with no single operation dominating. As a result, consumer graphics-focused GPUs can do matrix multiplication, but it's not their primary function.
To an extent, this history explains why modern GPUs look the way they do. They weren't designed purely for LLMs or ML models but as general-purpose accelerators, and the hardware aims for level of "generality" that can be both a blessing and a curse. GPUs much more often "just work" when applied to new tasks and lean far less on a good compiler than TPUs do. But this also makes them much harder to reason about or get roofline performance out of, since so many compiler features can cause bottlenecks.
GPUs are more modular. One key difference is that TPUs have 1-2 big Tensor Cores, while GPUs have hundreds of small SMs. Likewise, each Tensor Core has 4 big VPU with 1024 ALUs each, while GPUs have an H100 has 132 * 4 = 528 small independent SIMD units. Here is a 1:1 comparison of GPUs to TPU that highlights this point:

GPU	TPU	H100 #	TPU v5p #
SM (streaming multiprocessor)	Tensor Core	132	2
Warp Scheduler	VPU	528	8

GPU	TPU	H100 #	TPU v5p #
SMEM (L1 cache)	VMEM	32MB	128MB
Registers	Vector Registers (VRegs)	32MB	256kB
Tensor Core	MXU	528	8

This difference in modularity on the one hand makes TPUs much cheaper to build and simpler to understand, but it also puts more burden on the compiler to do the right thing. Because TPUs have a single thread of control and only support vectorized VPU-wide instructions, the compiler needs to manually pipeline all memory loads and MXU/VPU work to avoid stalls. A GPU programmer can just launch dozens of different kernels, each running on a totally independent SM. On the other hand, those kernels might get horrible performance because they are thrashing the L2 cache or failing to coalesce memory loads; because the hardware controls so much of the runtime, it becomes hard to reason about what's going on behind the scenes. As a result, TPUs can often get closer to peak roofline performance with less work.

TPUs have a lot more fast cache memory. TPUs also have a lot more VMEM than GPUs have SMEM (+TMEM), and this memory can be used for storing weights and activations in a way that lets them be loaded and used extremely fast. This can make them faster for LLM inference if you can consistently store or prefetch model weights into VMEM.

Quiz 1: GPU hardware

Here are some problems to work through that test some of the content above. Answers are provided, but it's probably a good idea to try to answer the questions before looking, pen and paper in hand.

Question 1 [CUDA cores]: How many fp32 CUDA cores does an H100 have? B200? How does this compare to the number of independent ALUs in a TPU v5p?

▶ Click here for the answer.

Question 2 [Vector FLOPs calculation]: A single H100 has 132 SMs and runs at a clock speed of 1.59GHz (up to 1.98GHz boost). Assume it can do one vector op per cycle per CUDA core. How many vector fp32 FLOPs can be done per second? With boost? How does this compare to matmul FLOPs?

▶ Click here for the answer.

Question 3 [GPU matmul intensity]: What is the peak fp16 matmul intensity on an H100? A B200? What about fp8?

▶ Click here for the answer.

Question 4 [Matmul runtime]: Using the answer to Question 3, how long would you expect a `fp16[64, 4096] * fp16[4096, 8192]` matmul to take on a single B200? How about `fp16[512, 4096] * fp16[4096, 8192]`?

▶ Click here for the answer.

Question 5 [L1 cache capacity]: What is the total L1/SMEM capacity for an H100? What about register memory? How does this compare to TPU VMEM capacity?

▶ Click here for the answer.

Question 6 [Calculating B200 clock frequency]: NVIDIA reports here that a B200 can perform 80TFLOPs/s of vector fp32 compute. Given that each CUDA core can perform 2 FLOPs/cycle in a FMA (fused multiply add) op, estimate the peak clock cycle.

▶ Click here for the answer.

Question 7 [Estimating H100 add runtime]: Using the figures above, calculate how long it ought to take to add two `fp32[N]` vectors together on a single H100. Calculate both

T_\text{math}Tmath T_\text{math}
and
T_\text{comms}Tcomms T_\text{comms}
. What is the arithmetic intensity of this operation? If you can get access, try running this operation in PyTorch or JAX as well for `N = 1024` and `N=1024 * 1024 * 1024`. How does this compare?

▶ Click here for the answer.

Networking

Networking is one of the areas where GPUs and TPUs differ the most. As we've seen, TPUs are connected in 2D or 3D tori, where each TPU is only connected to its neighbors. This means sending a message between two TPUs must pass through every intervening TPU, and forces us to use only uniform communication patterns over the mesh. While inconvenient in some respects, this also means the number of links per TPU is constant and we can scale to arbitrarily large TPU "pods" without loss of bandwidth.

GPUs on the other hand use a more traditional hierarchical tree-based switching network. Sets of 8 GPUs called **nodes** (up to 72 for GB200

[13] The term node is overloaded and can mean two things: the NVLink domain, aka the set of GPUs fully connected over NVLink interconnects, or the set of GPUs connected to a single CPU host. Before B200, these were usually the same, but in GB200 NVL72, we have an NVLink domain with 72 GPUs but still only 8 GPUs connected to each host. We use the term node here to refer to the NVLink domain, but this is controversial.
) are connected within 1 hop of each other using high-bandwidth interconnects called NVLinks, and these nodes are connected into larger units (called **SUs** or Scalable Units) with a lower bandwidth InfiniBand (IB) or Ethernet network using NICs attached to each GPU. These in turn can be connected into arbitrarily large units with higher level switches.

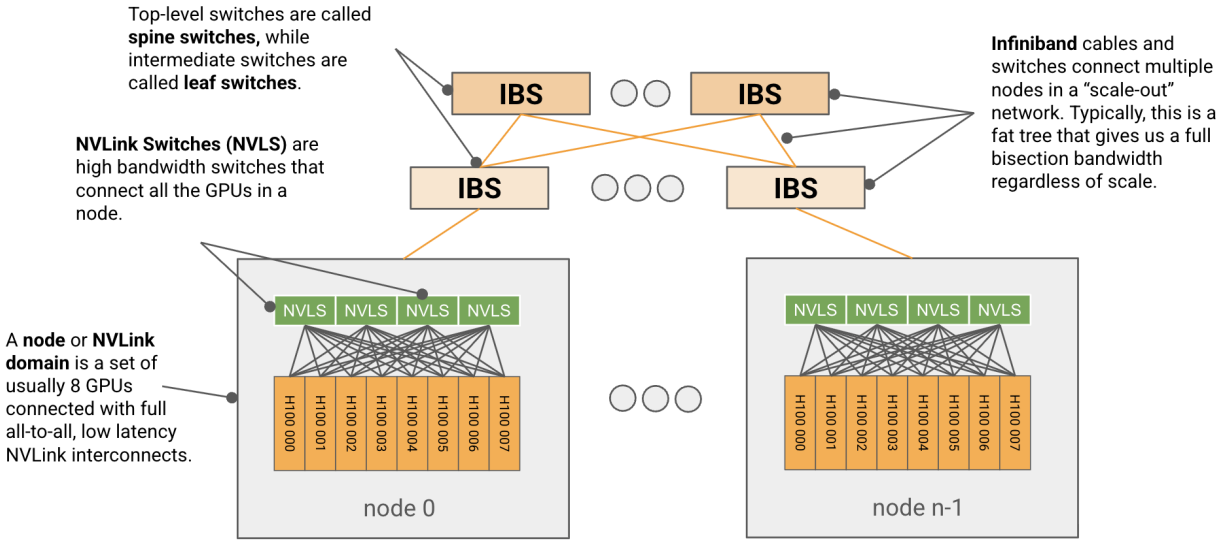


Figure: a diagram showing a typical H100 network. A set of 8 GPUs is connected into a node or NVLink domain with NVSwitches (also called NVLink switches), and these nodes are connected to each other with a switched InfiniBand fabric. H100s have about 450GB/s of egress bandwidth each in the NVLink domain, and each node has 400GB/s of egress bandwidth into the IB network.

At the node level

A GPU node is a small unit, typically of 8 GPUs (up to 72 for GB200), connected with all-to-all, full-bandwidth, low latency NVLink interconnects.

[14] NVLink has been described to me as something like a souped-up PCIe connection, with low latency and protocol overhead but not designed for scalability/fault tolerance, while InfiniBand is more like Ethernet, designed for larger lossy networks.
Each node contains several high-bandwidth NVSwitches which switch packets between all the local GPUs. The actual node-level topology has changed quite a bit over time, including the number of switches per node, but for H100, we have 4 NVSwitches per node with GPUs connected to them in a 5 + 4 + 4 + 5 link pattern, as shown:

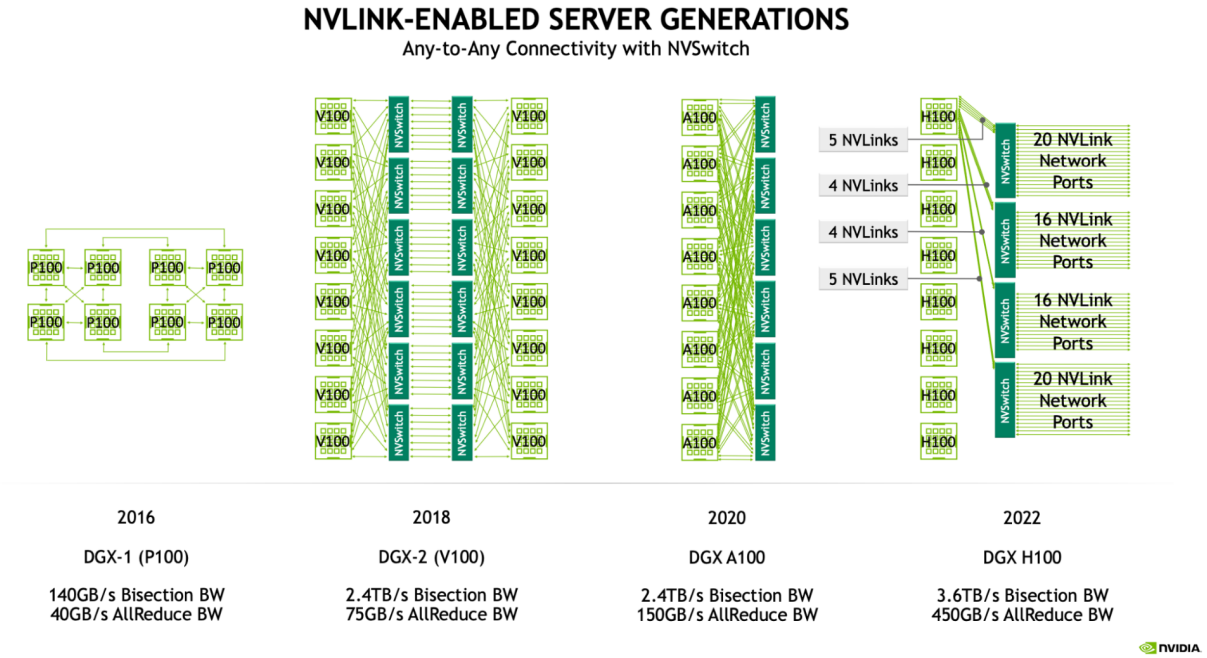


Figure: node aka NVLink domain diagrams from Pascall (P100) onward. Since Volta (V100), we have had all-to-all connectivity within a node using a set of switches. The H100 node has 4 NVSwitches connected to all 8 GPUs with 25GB/s links.

For the Hopper generation (NVLink 4.0), each NVLink link has 25GB/s of full-duplex

[15] Full-duplex here means 25GB/s each way, with both directions independent of each other. You can send a total of 50GB/s over the link, but at most 25GB/s in each direction.
bandwidth (50GB/s for B200), giving us `18 * 25=450GB/s` of full-duplex bandwidth from each GPU into the network. The massive NVSwitches have up to 64 NVLink ports, meaning an 8xH100 node with 4 switches can handle up to `64 * 25e9 * 4=6.4TB/s` of bandwidth. Here's an overview of how these numbers have changed with GPU generation:

NVLink Gen	NVSwitch Gen	GPU Generation	NVLink Bandwidth (GB/s, full-duplex)	NVLink Ports / GPU	Node GPU to GPU bandwidth (GB/s full-duplex)	Node size (NVLink domain)	NVSwitches per node
3.0	2.0	Ampere	25	12	300	8	6
4.0	3.0	Hopper	25	18	450	8	4
5.0	4.0	Blackwell	50	18	900	8/72	2/18

Blackwell (B200) has nodes of 8 GPUs. GB200NVL72 support larger NVLink domains of 72 GPUs. We show details for both the 8 and 72 GPUs systems.

Quiz 2: GPU nodes

Here are some more Q/A problems on networking. I find these particularly useful to do out, since they make you work through the actual communication patterns.

Question 1 [Total bandwidth for H100 node]: How much total bandwidth do we have per node in an 8xH100 node with 4 switches? *Hint:* consider both the NVLink and NVSwitch bandwidth.

▶ Click here for the answer.

Question 2 [Bisection bandwidth]: Bisection bandwidth is defined as the smallest bandwidth available between any even partition of a network. In other words, if split a network into two equal halves, how much bandwidth crosses between the two halves? Can you calculate the bisection bandwidth of an 8x H100 node? *Hint:* bisection bandwidth typically includes flow in both directions.

▶ Click here for the answer.

Question 3 [AllGather cost]: Given an array of B bytes, how long would a (throughput-bound) AllGather take on an 8xH100 node? Do the math for bf16[D_X, F] where `D=4096`, `F=65,536`. *It's worth reading the TPU collectives section before answering this. Think this through here but we'll talk much more about collectives next.*

▶ Click here for the answer.

Beyond the node level

Beyond the node level, the topology of a GPU network is less standardized. NVIDIA publishes a reference DGX SuperPod architecture that connects a larger set of GPUs than a single node using InfiniBand, but customers and datacenter providers are free to customize this to their needs.

[16] For instance, Meta trained LLaMA-3 on a datacenter network that differs significantly from this description, using Ethernet, a 3 layer switched fabric, and an oversubscribed switch at the top level. Here is a diagram for a reference 1024 GPU H100 system, where each box in the bottom row is a single 8xH100 node with 8 GPUs, 8 400Gbps CX7 NICs (one per GPU), and 4 NVSwitches.

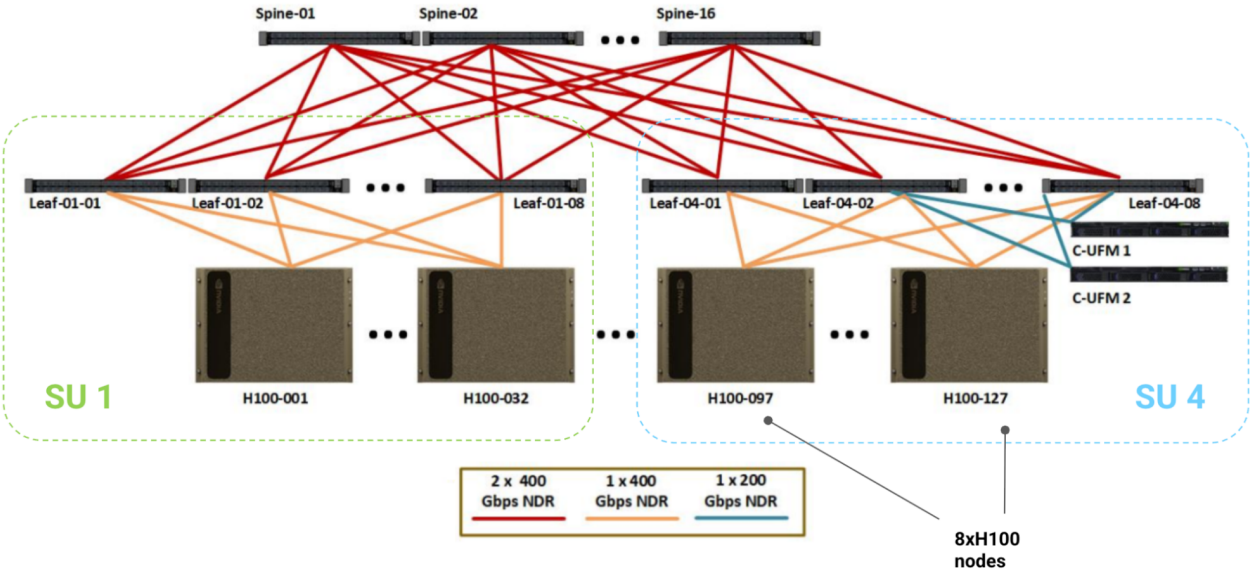


Figure: diagram of the reference 1024 H100 DGX SuperPod with 128 nodes (sometimes 127), each with 8 H100 GPUs, connected to an InfiniBand scale-out network. Sets of 32 nodes (256 GPUs) are called 'Scalable Units' or SUs. The leaf and spine IB switches provide enough bandwidth for full bisection bandwidth between nodes.

Scalable Units: Each set of 32 nodes is called a "Scalable Unit" (or SU), under a single set of 8 leaf InfiniBand switches. This SU has 256 GPUs with 4 NVSwitches per node and 8 Infiniband leaf switches. All the cabling shown is InfiniBand NDR (50GB/s full-duplex) with 64-port NDR IB switches (also 50GB/s per port). *Note that the IB switches have 2x the bandwidth of the NVSwitches (64 ports with 400 Gbps links).*

SuperPod: The overall SuperPod then connects 4 of these SUs with 16 top level "spine" IB switches, giving us 1024 GPUs with 512 node-level NVSwitches, 32 leaf IB switches, and 16 spine IB switches, for a total of 512 + 32 + 16 = 560 switches. Leaf switches are connected to nodes in sets of 32 nodes, so each set of 256 GPUs has 8 leaf switches. All leaf switches are connected to all spine switches.

How much bandwidth do we have? The overall topology of the InfiniBand network (called the "scale out network") is that of a **fat tree**, with the cables and switches guaranteeing full bisection bandwidth above the node level (here, 400GB/s). That means if we split the nodes in half, each node can egress 400GB/s to a node in the other partition at the same time. More to the point, this means we should have a roughly constant AllReduce bandwidth in the scale out network! While it may not be implemented this way, you can imagine doing a ring reduction over arbitrarily many nodes in the scale-out network, since you can construct a ring including every one.

Level	GPUs	Switches per Unit	Switch Type	Bandwidth per Unit (TB/s, full-duplex)	GPU-to-GPU Bandwidth (GB/s, full-duplex)	Fat Tree Bandwidth (GB/s, full-duplex)
Node	8	4	NVL	3.6	450	450
Leaf	256	8	IB	12.8	50	400
Spine	1024	16	IB	51.2	50	400

By comparison, a TPU v5p has about 90GB/s egress bandwidth per link, or 540GB/s egress along all axes of the 3D torus. This is not point-to-point so it can only be used for restricted, uniform communication patterns, but it still gives us a much higher TPU to TPU bandwidth that can scale to arbitrarily large topologies (at least up to 8960 TPUs).

The GPU switching fabric can in theory be extended to arbitrary sizes by adding additional switches or layers of indirection, at the cost of additional latency and costly network switches.

Takeaway: Within an H100 node, we have a full fat tree bandwidth of 450GB/s from each GPU, while beyond the node, this drops to 400GB/s node-to-node. This will turn out to be critical for communication primitives.

GB200 NVL72s: NVIDIA has recently begun producing new GB200 NVL72 GPU clusters that combine 72 GPUs in a single NVLink domain with full 900GB/s of GPU to GPU bandwidth. These domains can then be linked into larger SuperPods with proportionally higher (9x) IB fat tree bandwidth. Here is a diagram of that topology:

Figure: a diagram showing a GB200 DGX SuperPod of 576 GPUs. Each rack at the bottom layer contains 72 GB200 GPUs.

Counting the egress bandwidth from a single node (the orange lines above), we have `4 * 18 * 400 / 8 = 3.6TB/s` of bandwidth to the leaf level, which is 9x more than an H100 (just as the node contains 9x more GPUs). That means the critical node egress bandwidth is much, *much* higher and our cross-node collective bandwidth can actually be *lower* than within the node. See Appendix A for more discussion.

Node Type	GPUs per node	GPU egress bandwidth	Node egress bandwidth
H100	8	450e9	400e9

Node Type	GPUs per node	GPU egress bandwidth	Node egress bandwidth
B200	8	900e9	400e9
GB200 NVL72	72	900e9	3600e9

Takeaway: GB200 NVL72 SuperPods drastically increase the node size and egress bandwidth from a given node, which changes our rooflines significantly.

Quiz 3: Beyond the node level

Question 1 [Fat tree topology]: Using the DGX H100 diagram above, calculate the bisection bandwidth of the entire 1024 GPU pod at the node level. Show that the bandwidth of each link is chosen to ensure full bisection bandwidth. *Hint: make sure to calculate both the link bandwidth and switch bandwidth.*

▶ Click here for the answer.

Question 2 [Scaling to a larger DGX pod]: Say we wanted to train on 2048 GPUs instead of 1024. What would be the simplest/best way to modify the above DGX topology to handle this? What about 4096? *Hint: there's no single correct answer, but try to keep costs down. Keep link capacity in mind. This documentation may be helpful.*

▶ Click here for the answer.

How Do Collectives Work on GPUs?

GPUs can perform all the same collectives as TPUs: ReduceScatters, AllGathers, AllReduces, and AllToAlls. Unlike TPUs, the way these work changes depending on whether they're performed at the node level (over NVLink) or above (over InfiniBand). These collectives are implemented by NVIDIA in the NVSHMEM and NCCL (pronounced "nickel") libraries. NCCL is open-sourced here. While NCCL uses a variety of implementations depending on latency requirements/topology (details), from here on, we'll discuss a theoretically optimal model over a switched tree fabric.

Intra-node collectives

AllGather or ReduceScatter: For an AllGather or ReduceScatter at the node level, you can perform them around a ring just like a TPU, using the full GPU-to-GPU bandwidth at each hop. Order the GPUs arbitrarily and send a portion of the array around the ring using the full GPU-to-GPU bandwidth.

[17] You can also think of each GPU sending its chunk of size
\text{bytes} / Nbytes/N \text{bytes} / N
to each of the other
N - 1N−1 N - 1
GPUs, for a total of
(N - 1) * N * bytes / N(N−1)∗N∗bytes/N (N - 1) * N * bytes / N
bytes communicated, which gives us
The cost of each hop is
T_\text{hop} = \text{bytes} / (N * \text{GPU egress bandwidth})Thop
=bytes/(N∗GPU egress bandwidth) T_\text{hop} = \text{bytes} / (N * \text{GPU egress bandwidth})
, so the overall cost is
TAG or RS comms=bytes⋅(N−1)N⋅GPU egress bandwidth→bytesGPU egress bandwidth

You'll note this is exactly the same as on a TPU. For an AllReduce, you can combine an RS + AG as usual for twice the cost.

Figure: bandwidth-optimal 1D ring AllGather algorithm. For B bytes, this sends V / X bytes over the top-level switches X - 1 times.

If you're concerned about latency (e.g. if your array is very small), you can do a tree reduction, where you AllReduce within pairs of 2, then 4, then 8 for a total of

\log(N)log(N) \log(N)
hops instead of
N - 1N−1 N - 1
, although the total cost is still the same.
Takeaway: the cost to AllGather or ReduceScatter an array of B bytes within a single node is about

T_\text{comms} = B * (8 - 1) / (8 * W_\text{GPU egress}) \approxeq B / W_\text{GPU egress}Tcomms=B∗(8−1)/(8∗WGPU egress)≈B/WGPU egress T_\text{comms} = B * (8 - 1) / (8 * W_\text{GPU egress}) \approxeq B / W_\text{GPU egress}
. This is theoretically around
B / \text{450e9}B/450e9 B / \text{450e9}
on an H100 and
B / \text{900e9}B/900e9 B / \text{900e9}
on a B200. An AllReduce has 2x this cost unless in-network reductions are enabled.
Pop Quiz 1 [AllGather time]: Using an 8xH100 node with 450 GB/s full-duplex bandwidth, how long does AllGather(bf16[B_X, F]) take? Let

B=1024B=1024 B=1024
,
F=16,384F=16,384 F=16,384
.
▶ Click here for the answer.
AllToAlls: GPUs within a node have all-to-all connectivity, which makes AllToAlls, well, quite easy. Each GPU just sends directly to the destination node. Within a node, for B bytes, each GPU has

B / NB/N B / N
bytes and sends
(B / N^2)(B/N2) (B / N^2)
bytes to
N - 1N−1 N - 1
target nodes for a total of

$$T_{\text{AllToAll comms}} = B \cdot (N-1)W \cdot N2 \approx BW \cdot N$$

Compare this to a TPU, where the cost is

B / (4W)B/(4W) B / (4W)
. Thus, within a single node, we get a 2X theoretical speedup in runtime (
B / 4WB/4W B / 4W
vs.
B / 8WB/8W B / 8W
).
For Mixture of Expert (MoE) models, we frequently want to do a *sparse or ragged AllToAll,* where we guarantee at most

kk k
of
NN N
shards on the output dimension are non-zero, that is to say
T_\text{AllToAll}_X \rightarrow K[B, N]

where at most
kk k
of
NN N
entries on each axis are non-zero. The cost of this is reduced by
k/Nk/N k/N
, for a total of about
\min(k/N, 1) \cdot B / (W \cdot N)min(k/N,1)·B/(W·N) \min(k/N, 1) \cdot B / (W \cdot N)
. For an MoE, we often pick the non-zero values independently at random, so there's some chance of having fewer than
kk k
non-zero, giving us approximately
(N-1)/N \cdot \min(k/N, 1) \cdot B / (W \cdot N)(N−1)/N·min(k/N,1)·B/(W·N) (N-1)/N \cdot \min(k/N, 1) \cdot B / (W \cdot N)
.
[18] The true cost is actually

$$(1-(Z-1Z)K)\cdot Z-1Z$$

the expected number of distinct outcomes in
KK K
dice rolls, but it is very close to the approximation given. See the Appendix for more details.
Pop Quiz 2 [AllToAll time]: Using an 8xH100 node with 450 GB/s unidirectional bandwidth, how long does $\text{AllToAll}_{X->N}$(bf16[B_X, N]) take? What if we know only 4 of 8 entries will be non-zero?

▶ Click here for the answer.
Takeaway: The cost of an AllToAll on an array of

BB B
bytes on GPU within a single node is about
T_\text{comms} = (B \cdot (8 - 1)) / (8^2 \cdot W_\text{GPU egress}) \approx B / (8 \cdot W_\text{GPU egress})Tcomms=(B·(8−1))/(82·WGPU egress)≈B/(8·WGPU egress) T_\text{comms} = (B \cdot (8 - 1)) / (8^2 \cdot W_\text{GPU egress}) \approx B / (8 \cdot W_\text{GPU egress})
. For a ragged (top-
kk k
) AllToAll, this is decreased further to
(B \cdot k) / (864 \cdot W_\text{GPU egress})(B·k)/(864·WGPU egress) (B \cdot k) / (864 \cdot W_\text{GPU egress})
.
Empirical measurements: here is an empirical measurement of AllReduce bandwidth over an 8xH100 node. The Algo BW is the measured bandwidth (bytes / runtime) and the Bus BW is calculated as

2 \cdot W \cdot (8 - 1) / 82·W·(8−1)/8 2 \cdot W \cdot (8 - 1) / 8
, theoretically a measure of the actual link bandwidth. You'll notice that we do achieve close to 370GB/s, less than 450GB/s but reasonably close, although only around 10GB/device. This means

although these estimates are theoretically correct, it takes a large message to realize it.

Figure: AllReduce throughput for an 8xH100 node with SHARP disabled. The blue curve is the empirical link bandwidth, calculated as
2 * \text{bytes} * (N - 1) / (N * \text{runtime})2∗bytes∗(N−1)/(N∗runtime) 2 * \text{bytes} * (N - 1) / (N * \text{runtime})
from the empirical measurements. Note that we do not get particularly close to the claimed bandwidth of 450GB/s, even with massive 10GB arrays.

This is a real problem, since it meaningfully complicates any theoretical claims we can make, since e.g. even an AllReduce over a reasonable sized array, like LLaMA-3 70B's MLPs (of size `bf16[8192, 28672]`, or with 8-way model sharding, `bf16[8192, 3584] = 58MB`) can achieve only around 150GB/s compared to the peak 450GB/s. By comparison, TPUs achieve peak bandwidth at much lower message sizes (see Appendix B).

Takeaway: although NVIDIA claims bandwidths of about 450GB/s over an H100 NVLink, it is difficult in practice to exceed 370 GB/s, so adjust the above estimates accordingly.

In network reductions: Since the Hopper generation, NVIDIA switches have supported "SHARP" (Scalable Hierarchical Aggregation and Reduction Protocol) which allows for "in-network reductions". This means *the network switches themselves* can do reduction operations and multiplex or "MultiCast" the result to multiple target GPUs:

Figure: an AllReduce without SHARP has 2x the theoretical cost because it has to pass through each GPU twice. In practice, speedups are only about 30% (from NCCL 2.27.5).

Theoretically, this close to halves the cost of an AllReduce, since it means each GPU can send its data to a top-level switch which itself performs the reduction and broadcasts the result to each GPU without having to egress each GPU twice, while also reducing network latency.

TSHARP AR comms=bytesGPU egress bandwidth

Note that this is exact and not off by a factor of

1/N1/N 1/N
, since each GPU egresses
B \cdot (N - 1) / NB·(N−1)/N B \cdot (N - 1) / N
first, then receives the partially reduced version of its local shard (ingress of
B/NB/N B/N
), finishes the reductions, then egresses
B/NB/N B/N
again, then ingresses the fully reduced result (ingress of
B \cdot (N - 1) / NB·(N−1)/N B \cdot (N - 1) / N
), resulting in exactly
BB B
bytes ingressed.
However, in practice we see about a 30% increase in bandwidth with SHARP enabled, compared to the predicted 75%. This gets us up merely to about 480GB/s effective collective bandwidth, not nearly 2x.

Figure: empirical measurements of AllReduce algo bandwidth with and without NVIDIA SHARP enabled within a node. The gains amount to about 30% throughput improvement at peak, even though algorithmically it ought to be able to achieve closer to a 75% gain.

Takeaway: in theory, NVIDIA SHARP (available on most NVIDIA switches) should reduce the cost of an AllReduce on

BB B
bytes from about
2 * B / W2∗B/W 2 * B / W
to
B / WB/W B / W
. However, in practice we only see a roughly 30% improvement in bandwidth. Since pure AllReduces are fairly rare in LLMs, this is not especially useful.

Cross-node collectives

When we go beyond the node-level, the cost is a bit more subtle. When doing a reduction over a tree, you can think of reducing from the bottom up, first within a node, then at the leaf level, and then at the spine level, using the normal algorithm at each level. For an AllReduce especially, you can see that this allows us to communicate less data overall, since after we AllReduce at the node level, we only have to egress

BB B
bytes up to the leaf instead of
B * NB∗N B * N
.
How costly is this? To a first approximation, because we have full bisection bandwidth, the cost of an AllGather or ReduceScatter is roughly the buffer size in bytes divided by the node egress bandwidth (400GB/s on H100) *regardless of any of the details of the tree reduction.*

TAG or RS comms=bytesWnode egress=H100bytes400e9

where

W_\text{node}Wnode W_\text{node}
egress is generally 400GB/s for the above H100 network (8x400Gbps IB links egressing each node). The cleanest way to picture this is to imagine doing a ring reduction over *every node in the cluster*. Because of the fat tree topology, we can always construct a ring with
W_\text{node}Wnode W_\text{node}
egress between any two nodes and do a normal reduction. The node-level reduction will (almost) never be the bottleneck because it has a higher overall bandwidth and better latency, although in general the cost is

Ttotal=max(Tcomms at node,Tcomms in scale-out network)=max[bytesWGPU egress,bytesWnode egress]

▶ You can see a more precise derivation here.
Other collectives: AllReduces are still 2x the above cost unless SHARP is enabled. NVIDIA sells SHARP-enabled IB switches as well, although not all providers have them. AllToAlls do change quite a bit cross-node, since they aren't "hierarchical" in the way AllReduces are. If we want to send data from every GPU to every other GPU, we can't use take advantage of the full bisection bandwidth at the node level. That means if we have an N-way AllToAll that spans

M = N / 8M=N/8 M = N / 8
nodes, the cost is

TAllToAll comms=B·(M−1)M2·Wnode egress≈BM·Wnode egress

which effectively has 50GB/s rather than 400GB/s of bandwidth. We go from

B / (8 * \text{450e9})B/(8∗450e9) B / (8 * \text{450e9})
within a single H100 node to
B / (2 \cdot \text{400e9})B/(2·400e9) B / (2 \cdot \text{400e9})
when spanning 2 nodes, a more than 4x degradation.
Here is a summary of the 1024-GPU DGX H100 SuperPod architecture:

Level	Number of GPUs	Degree (# Children)	Switch Bandwidth (full-duplex, TB/s)	Cable Bandwidth (full-duplex, TB/s)	Collective Bandwidth (GB/s)
Node	8	8	6.4	3.6	450
Leaf (SU)	256	32	25.6	12.8	400
Spine	1024	4	51.2	51.2	400

We use the term “Collective Bandwidth” to describe the effective bandwidth at which we can egress either the GPU or the node. It’s also the

\text{bisection bandwidth} * 2 / Nbisection bandwidth∗2/N \text{bisection bandwidth} * 2 / N
.
Takeaway: beyond the node level, the cost of an AllGather or AllReduce on B bytes is roughly

B / W_\text{node egress}B/Wnode egress B / W_\text{node egress}
, which is
B / \text{400e9}B/400e9 B / \text{400e9}
on an H100 DGX SuperPod. The overall topology is a fat tree designed to give constant bandwidth between any two pairs of nodes.
Reductions when array is sharded over a separate axis: Consider the cost of a reduction like

AllReduceX(A[IY,J] {UX})

where we are AllReducing over an array that is itself sharded along another axis

YY Y
. On TPUs, the overall cost of this operation is reduced by a factor of
1 / Y1/Y 1 / Y
compared to the unsharded version since we’re sending
1 / Y1/Y 1 / Y
as much data per axis. On GPUs, the cost depends on which axis is the “inner” one (intra-node vs. inter-node) and whether each shard spans more than a single node. Assuming
YY Y
is the inner axis here, the overall cost is reduced effectively by
YY Y
, but only if
YY Y
spans multiple nodes:

Tcomms at node=bytes·Dnodemin(Y,Dnode)·WGPU egress

Tcomms in scale-out network=bytes·NY·Wnode egress

$$T_{total} = \max(T_{comms\ at\ node}, T_{comms\ in\ scale\text{-}out\ network})$$

where N is the number of GPUs and again D is the number of GPUs in a node (the degree of the node). As you can see, if

$Y < D_{node}$
, we get a win at the node level but generally don't see a reduction in overall runtime, while if
$Y > D_{node}$
, we get a speedup proportional to the number of nodes spanned.
If we want to be precise about the ring reduction, the general rule for a tree $\text{AllGather}_X(A_Y \{ U_X \})$ (assuming Y is the inner axis) is

$$T_{AR\ or\ RS\ comms} = \text{bytes} \cdot \max_{\text{depth } i} [D_{i-1} D_i \cdot \max(Y, S_i - 1) \cdot W_{\text{link } i}]$$

where

S_i
is $M * N * \ldots$, the size of the subnodes below level i in the tree. This is roughly saying that the more GPUs or nodes we span, the greater our available bandwidth is, but only within that node.
Pop Quiz 3 [Sharding along 2 axes]: Say we want to perform

$\text{AllGather}_X(\text{bf16}[D_X, F_Y])$
where
Y
is the inner axis over a single SU (256 chips). How long will this take as a function of
D
,
F
, and
Y
?
▶ Click here for the answer.
Takeaway: when we have multiple axes of sharding, the cost of the outer reduction is reduced by a factor of the number of nodes spanned by the inner axis.

Quiz 4: Collectives

Question 1 [SU AllGather]: Consider only a single SU with M nodes and N GPUs per node. Precisely how many bytes are ingressed and egressed by the node level switch during an AllGather? What about the top-level switch?

▶ Click here for the answer.
Question 2 [Single-node SHARP AR]: Consider a single node with N GPUs per node. Precisely how many bytes are ingressed and egressed by the switch during an AllReduce using SHARP (in-network reductions)?

▶ Click here for the answer.
Question 3 [Cross-node SHARP AR]: Consider an array $\text{bf16}[D_X, F_Y]$ sharded over a single node of N GPUs. How long does $\text{AllReduce}(\text{bf16}[D, F_Y] \{ U_X \})$ take? You can assume we do in-network reductions. Explain how this differs if we have more than a single node?

▶ Click here for the answer.

Question 4 [Spine level AR cost]: Consider the same setting as above, but with

Y = 256Y=256 Y = 256
(so the AR happens at the spine level). How long does the AllReduce take? Again, feel free to assume in-network reductions.

▶ Click here for the answer.

Question 5 [2-way AllGather cost]: Consider the cost of an AllGather over exactly 2 nodes. What is it, precisely? Make sure to calculate the precise cost and not the approximation.

▶ Click here for the answer.

Rooflines for LLM Scaling on GPUs

Now let's look at what this has all been building towards: understanding rooflines for LLM scaling on GPU. This is to complement the TPU training chapter here. As we did there, the goal here is to look at the total

T_\text{math}Tmath T_\text{math}
and
T_\text{comms}Tcomms T_\text{comms}
for different parallelism strategies and understand at what point
T_\text{comms} > T_\text{math}Tcomms>Tmath T_\text{comms} > T_\text{math}
. As before, we consider only the MLP block with operations

$$\text{MLP}(x) \equiv x[B,D] * DWin[D,F] \cdot FWout[F,D]$$

where

BB B
is the global batch size **in tokens** (i.e.
B = \text{batch size} \cdot \text{sequence length}B=batch size·sequence length B = \text{batch size} \cdot \text{sequence length}
).
Here we'll reproduce the table above showing effective bandwidths at both the GPU and node level:

Node Type	GPUs per node	GPU egress bandwidth	Node egress bandwidth
H100	8	450e9	400e9
B200	8	900e9	400e9
GB200 NVL72	72	900e9	3600e9

Note: Both the GPU and node egress bandwidths determine rooflines for our LLMs. We'll use the term

W_\text{collective}Wcollective W_\text{collective}
to describe either the GPU or node bandwidths depending on whether we are operating within or above the node level.
Let's look at the compute communication rooflines as we did for TPUs for **data parallelism, tensor parallelism, pipeline parallelism, expert parallelism,** and combinations thereof. For the rest of this section we'll focus on H100 rooflines for specific calculations. GB200-NVL72 has the same general rooflines but because we have a larger node egress bandwidth, we can sometimes be bottlenecked at the node level instead.

Data Parallelism

As noted before, DP and ZeRO sharding involve either a weight AllReduce or a ReduceScatter + AllGather in the backward pass. Since these both have the same cost, to be compute-bound for pure data parallelism or FSDP *without in-network reductions*, we have, per layer, in the backward pass, with an axis of size X:

$$T_{math} = 2 \cdot 2 \cdot 2 \cdot BDFX \cdot C$$

$$T_{comms} = 2 \cdot 2 \cdot 2 \cdot DFW_{collective}$$

Therefore, for

$T_\text{math} > T_\text{comms}$ $T_{math} > T_{comms}$
, we need
$B / (XC) > 1 / W_\text{collective}$ $B/(XC) > 1/W_{collective}$
or

$$BX > CW_{collective}$$

where

$W_\text{collective}$ $W_{collective}$
is either the GPU or node level egress bandwidth depending on whether we're sharding within a node or across nodes. Thus:

- **Within a node**, we just need the per-GPU **token** batch size > $\text{990e12} / \text{450e9} = 2200$ $990e12/450e9 = 2200$.
- **Within an SU or at the spine level**, BS > $\text{990e12} / \text{400e9} = 2475$ $990e12/400e9 = 2475$.

This is quite a bit higher than on a TPU, where the number is 850 with all three axes. For instance, LLaMA-3, which trained on 16000 H100s would need a batch size of at least 40M tokens (for reference, they used 16M). DeepSeek v3 trained on 2048 H800 GPUs with lower 300GB/s of bandwidth (instead of 450GB/s on H100) would need

$\text{990e12} / \text{300e9} = 3300$ $990e12/300e9 = 3300$
tokens per GPU, or about 6.7M (in practice, they used 4M).
With in-network reductions enabled and using pure data parallelism, theoretically we have 2x the AllReduce bandwidth, which would halve both of these numbers. However, in practice the benefit is closer to 30%, which only really makes up for the fact that we typically struggle to reach the reported numbers. Furthermore, because pure data parallelism is rarely useful, this basically doesn't matter in practice.

MoE models: For a Mixture of Experts (MoE) model, where we have E experts and k experts per token, this increases to

$$T_{math} = 2 \cdot 2 \cdot 2 \cdot k \cdot BDFX \cdot C$$

$$T_{comms} = 2 \cdot 2 \cdot 2 \cdot EDFW_{collective}$$

which inflates the per-GPU token batch size by a factor of

E/k E/k

, i.e.

$$BX > EkCW_{collective}$$

For example, the new OpenAI OSS model with

$k=4$
and
$E=128$
, this increases to `32 * 2475 = 79,200` across nodes, a kind of ridiculously high number.
What happens when X is small? When we do only e.g. 2-node data parallelism, we benefit from the

$(X - 1) / X$
scaling, which gives us

$$T_{math} = 2 \cdot 2 \cdot 2 \cdot BDFN * C$$

$$T_{comms} = 2 \cdot 2 \cdot 2 \cdot DF \cdot (X-1)X \cdot W_{collective}$$

where X is the number of nodes and

$N = 8 \cdot X$
. Then for a dense model we have
$B / N > \alpha \cdot (X - 1) / X$
, or e.g.
$B / N > \text{1237}$
, half the above value. You'll notice 2-way data parallelism fairly often for this reason.
Takeaway: Data parallelism and ZeRO sharding require a per-GPU batch size of about 2500 tokens to be compute-bound on an H100 or B200, assuming perfect overlap and FLOPs utilization. For MoE models, this increases by a factor of

E / k
, the ratio of total to activated parameters. When doing a small amount of data parallelism, the critical batch size decreases.

Tensor Parallelism

Tensor parallelism requires an AllGather and ReduceScatter over the activations, which we need to overlap with the MLP FLOPs. In other words, in the forward pass, we have

$$T_{math} = 2 \cdot 2 \cdot BDFY \cdot C$$

$$T_{comms} = 2 \cdot 2 \cdot BDW_{collective}$$

which to be compute-bound gives us the rule

$$Y < F \cdot \frac{W_{collective}}{C}$$

Within a node, this gives us about

$F / 2200$
or
$F / 2475$
beyond a node. For
$F=\text{28000}$

like LLaMA-3, this is about 11-way TP (or rounding down, about 8-way, which is how large a node is). As with above, we get an extra 2X bandwidth when we span exactly 2 nodes, so we can generally do 16-way data parallelism ($F > 2475 \cdot (Y - 8)$), which gives us up to 19-way model parallelism in theory.
Takeaway: Tensor parallelism over an axis of size Y with feed-forward dimension F becomes communication-bound when the

$Y > F / 2475$, which generally constrains us to only intra-node TP or at most 2-node TP.

Expert Parallelism

As we've already noted above, Mixture of Expert (MoE) models come with E times more model weights with only k times more FLOPs, making data parallelism significantly harder. We can mitigate this somewhat by sharding the our weights along the expert dimension, i.e. $W_{in}[E_Z, D, F]$. To do the MLP block, we need to introduce 2x AllToAll to send our activations to the corresponding experts.

As noted above, the cost of this $\text{AllToAll}_{Z->k}([B, D, k])$ if it spans multiple nodes is roughly

$T_\text{AllToAll} = 2 \cdot B \cdot D \cdot (Z-8)/Z \min(8 * k / Z, 1)$, so for pure expert parallelism we need

$$T_{math} = 4 \cdot B \cdot k \cdot D \cdot F / Z \cdot C$$

$$T_{comms} = 4 \cdot B \cdot D \cdot (Z-8) / W \cdot Z \cdot \min(8 \cdot k / Z, 1)$$

We either need

$K > Z/8$
with
$F > \alpha \cdot (Z - 8)/k$
or
$Z \gg K$
and
$F > 8 \cdot \alpha$
, where
$\alpha = C/W$
. This gives you two domains in which expert parallelism is possible, one with a small amount of expert parallelism (roughly 2-node) and small
F
, or one with large
F
and
Z
arbitrarily large (up to E-way expert parallelism).
You'll see both cases in practice, either a small amount of expert-parallelism (like DeepSeek v3 which has very small F and relatively small, restricted cross-node expert parallelism), or models with large F, in which case we can do significant cross-node EP alongside TP.

Takeaway: if

F < 8 * C / W_\text{node}F<8∗C/Wnode F < 8 * C / W_\text{node}
, expert parallelism can span 1-2 nodes with similar (slightly lower) cost to TP, or if
F > 8 * C / W_\text{node}F>8∗C/Wnode F > 8 * C / W_\text{node}
, we can do a significant amount of expert parallelism (up to
EE E
nodes) with relatively low cost.

Pipeline Parallelism

Pipeline parallelism splits layers across nodes with an extremely low communication cost, since we are just sending small microbatches of activations every couple layers. Historically pipelining has suffered from “pipeline bubbles”, but with new zero-bubble pipelining approaches, it is typically possible to do without.

The overall communication cost of pipelining is tiny: with

N_\text{MB}NMB N_\text{MB}
microbatches and
N_\text{stages}Nstages N_\text{stages}
, we have
T_\text{comms per hop} = 2 \cdot B \cdot D / (W \cdot N_\text{MB})Tcomms per hop=2·B·D/(W·NMB) T_\text{comms per hop} = 2 \cdot B \cdot D / (W \cdot N_\text{MB})
and
N_\text{MB} + N_\text{stages} - 2NMB+Nstages−2 N_\text{MB} + N_\text{stages} - 2
hops, so roughly

$$T_{\text{total PP comms}} = 2BDW \cdot N_{\text{microbatches}} \cdot (N_{\text{microbatches}} + N_{\text{stages}} - 2)$$

$$T_{\text{per-layer comms}} \approx 1.5 \cdot 2BDW \cdot N_{\text{layers}}$$

Since we are dividing by

N_\text{layers}Nlayers N_\text{layers}
, this is vastly smaller than any of the other costs. In other words, from a communication standpoint, pipelining is basically free. So why don’t we just do pipelining? There are a few reasons:
(1) **Code complexity:** pipelining doesn’t fit nicely as nicely into automatic parallelism frameworks (like XLA’s GSPMD) as other approaches. Because it introduces microbatching to hide pipeline bubbles, it changes the structure of the program, and custom zero-bubble pipeline schedules exacerbate this problem by requiring complicated interleaving of the forward and backward pass.

(2) **Pipelining makes data parallelism and FSDP hard:** probably the biggest reason not to do pipelining is that it plays badly with FSDP and data parallelism. ZeRO-3 sharding in particular works badly, since it requires us to AllGather the weights on every microbatch which doesn’t work when we have only

B / N_\text{microbatches}B/Nmicrobatches B / N_\text{microbatches}
tokens to amortize the AllGather cost. Furthermore, during the backward pass, *we can’t AllReduce or ReduceScatter the gradients until the last microbatch has passed a given stage, which means we have significant non-overlapped communication time.*

Figure: an example 2 stage, 2 microbatch pipeline. F denotes a stage forward pass and B is a stage backward pass (2x the cost). G denotes the data-parallel AllReduces, which can be significantly longer than the time of a single microbatch.

(3) **Pipeline bubbles and step imbalance:** As you can see in the (bad) pipeline schedule above, it is easy to have significant bubbles (meaning wasted compute) during a naive pipeline schedule. Above, the second stage is idle on step 0, the first stage is idle from step 2 to 3, and the second stage is again idle on the last step. While we can avoid these somewhat with careful scheduling, we still often have some bubbles. We also have to pass activations from one stage to the next on the critical path, which can add overhead:

Figure: an example pipeline showing transfer cost in red. This shifts stages relative to each other and increases the pipeline bubble overhead.

There are workarounds for each of these issues, but they tend to be complicated to implement and difficult to maintain, but pipelining remains a technique with low communication cost relative to other methods.

Caveat about latency: As noted before, GPUs struggle to achieve full AllReduce bandwidth even with fairly large messages. This means even if we in theory can scale e.g. expert-parallel AllToAlls across multiple nodes, we may struggle to achieve even 50% of the total bandwidth. This means we do try to keep TP or EP within a smaller number of nodes to minimize latency overhead.

Examples

What does DeepSeek do? For reference, DeepSeek V3 is trained with 2048 H800 GPUs with:

- 64-way Expert Parallelism (EP) spanning 8 nodes
- 16-way Pipeline Parallelism (PP)
- 2-way ZeRO-1 Data Parallelism (DP)

They had a steady state batch size of `4096 * 15360 = 62,914,560` tokens, or 30k tokens per GPU. You can see that this is already quite large, but their model is also very sparse (k=8, E=256) so you need a fairly large batch size. You can see that with 64-way EP and 16-way PP, we end up with 1024-way model parallelism in total, which means the AllReduce is done at the spine level, and because it's only 2-way, we end up with

2 / (2 - 1) = 22/(2−1)=2 2 / (2 - 1) = 2
times more bandwidth in practice. This also helps reduce the cost of the final data-parallel AllReduce overlapping with the final pipeline stages.
What does LLaMA-3 do? LLaMA-3 trains with a BS of 16M tokens on 16k GPUs, or about 1k tokens per GPU. They do:

- 8-way Tensor Parallelism within a node (TP)
- 16-way Pipeline Parallelism (PP)
- 128-way ZeRO-1 Data Parallelism

This is also a dense model so in general these things are pretty trivial. The 16-way PP reduces the cost of the data parallel AllReduce by 16x, which helps us reduce the critical batch size.

TLDR of LLM Scaling on GPUs

Let's step back and come up with a general summary of what we've learned so far:

- **Data parallelism or FSDP (ZeRO-1/3) requires a local batch size of about 2500 tokens per GPU**, although in theory in-network reductions + pure DP can reduce this somewhat.

- **Tensor parallelism is compute-bound up to about 8-ways** but we lack the bandwidth to scale much beyond this before becoming comms-bound. This mostly limits us to a single NVLink domain (i.e. single-node or need to use GB200NVL72 with to 72 GPUs).
- **Any form of model parallelism that spans multiple nodes can further reduce the cost of FSDP**, so we often want to mix PP + EP + TP to cross many nodes and reduce the FSDP cost.
- **Pipeline parallelism works well if you can handle the code complexity of zero-bubble pipelining and keep batch sizes fairly large to avoid data-parallel bottlenecks.** Pipelining usually makes ZeRO-3 impossible (since you would need to AllGather on each pipeline stage), but you can do ZeRO-1 instead.

At a high level, this gives us a recipe for sharding large models on GPUs:

- For relatively small dense models, aggressive FSDP works great if you have the batch size, possibly with some amount of pipelining or tensor parallelism if needed.
- For larger dense models, some combination of 1-2 node TP + many node PP + pure DP works well.
- For MoEs, the above rule applies but we can also do expert parallelism, which we prefer to TP generally. If
  $F > 8 * C / W_{node}$
  , we can do a ton of multi-node expert parallelism, but otherwise we're limited to roughly 2-node EP.

Quiz 5: LLM rooflines

Question 1 [B200 rooflines]: A B200 DGX SuperPod (**not GB200 NVL72**) has 2x the bandwidth within a node (900GB/s egress) but the same amount of bandwidth in the scale-out network (400GB/s) (source). The total FLOPs are reported above. How does this change the model and data parallel rooflines?

▶ Click here for the answer.

Question 2 [How to shard LLaMA-3 70B]: Consider LLaMA-3 70B, training in bfloat16 with fp32 optimizer state with Adam.

1. At a minimum, how many H100s would we need simply to store the weights and optimizer?
2. Say we want to train on 4096 H100 GPUs for 15T tokens. Say we achieved 45% MFU (Model FLOPs Utilization). How long would it take to train?
3. LLaMA-3 70B has `F = 28,672` and was trained with a batch size of about 4M tokens. What is the most model parallelism we could do without being comms-bound? With this plus pure DP, could we train LLaMA-3 while staying compute-bound? What about ZeRO-3? What about with 8-way pipelining?

▶ Click here for the answer.

Question 3 [Megatron-LM hyperparams]: Consider this figure from the Megatron-LM repository highlighting their high MFU numbers.

Model size	Attention heads	Hidden size	Number of layers	Tensor MP size	Pipeline MP size	Data-parallel size	Number of GPUs	Batch size	Per-GPU teraFLOP/s	MFU	Aggregate petaFLOP/s
1.7B	16	2048	24	1	1	48	48	192	408.8	41%	19.6
7.1B	32	4096	30	2	1	48	96	192	465.9	47%	44.7
16B	48	6144	32	4	1	48	192	192	489.1	49%	93.9
32B	56	7168	48	8	1	48	384	192	459.6	46%	176.5
70B	64	8192	84	8	2	48	768	384	419.7	42%	322.3
119B	80	10240	92	8	4	48	1536	768	420.5	43%	645.9
177B	96	12288	96	8	6	48	2304	1152	432.8	44%	997.2
314B	128	16384	96	8	8	48	3072	1536	474.4	48%	1457.4
462B	144	18432	112	8	16	48	6144	3072	459.9	47%	2825.6

Note that their sequence length is 4096 everywhere. For the 16B, 70B, and 314B models, what is the per-GPU token batch size? Assuming data parallelism is the outermost axis and assuming bfloat16 reductions, determine whether each of these is theoretically compute-bound or communication-bound, and whether there is a more optimal configuration available?

▶ Click here for the answer.

Acknowledgements and Further Reading

This chapter relied heavily on help from many knowledgeable GPU experts, including:

- Adam Paszke, who helped explain the realities of kernel programming on GPUs.
- Swapnil Patil, who first explained how GPU networking works.
- Stas Bekman, who pointed out that the empirical realities of GPUs are often different from the purported specs.
- Reiner Pope, who helped clarify how GPUs and TPUs compare at a hardware level.
- Frédéric Bastien, who gave detailed feedback on the chip-level story.
- Nouamane Tazi, whose experience with LLM training on GPUs helped improve the roofline section.
- Sanford Miller, who helped me understand how GPUs are networked and how NVIDIA's specifications compare to what's often deployed in the field.

There's a great deal of good reading on GPUs, but some of my favorites include:

- SemiAnalysis' History of the NVIDIA Tensor Core: a fantastic article describing how GPUs transformed from video game engines to ML accelerators.
- SemiAnalysis' Analysis of Blackwell Performance: worth reading to understand the next generation of NVIDIA GPUs.
- H100 DGX SuperPod Reference: dry but useful reading on how larger GPU clusters are networked. Here is a similar document about the GB200 systems.
- Hot Chips Talk about the NVLink Switch: fun reading about NVLink and NCCL collectives, especially including in-network reductions.
- DeepSeek-V3 Technical Report: a good example of a large semi-open LLM training report, describing how they picked their sharding setup.
- How to Optimize a CUDA Matmul: a great blog describing how to implement an efficient matmul using CUDA Cores, with an eye towards cache coherence on GPU.
- HuggingFace Ultra-Scale Playbook: a guide to LLM parallelism on GPUs, which partly inspired this chapter.
- Making Deep Learning Go Brrrr From First Principles:: a more GPU and PyTorch-focused tutorial on LLM rooflines and performance engineering.

Appendix A: How does this change with GB200?

Blackwell introduces a bunch of major networking changes, including NVLink 5 with twice the overall NVLink bandwidth (900GB/s). B200 still has 8-GPU nodes, just like H100s, but GB200 systems (which combine B200 GPUs with Grace CPUs) introduce much larger NVLink domain (72 GPUs in NVL72 and in theory up to 576). This bigger NVLink domain also effectively increases the node egress bandwidth, which reduces collective costs above the node level.

Figure: a diagram showing how a GB200 NVL72 unit is constructed, with 18 switches and 72 GPUs.

Within a node, this increased bandwidth (from 450GB/s to 900GB/s) doesn't make much of a difference because we also double the total FLOPs/s of each GPU. Our rooflines mostly stay the same, although because NVLink has much better bandwidth, Expert Parallelism becomes easier.

Beyond a node, things change more. Here's a SuperPod diagram from here.

Figure: a diagram showing a GB200 DGX SuperPod of 576 GPUs.

As you can see, the per-node egress bandwidth increases to `4 * 18 * 400 / 8 = 3.6TB/s`, up from 400GB/s in H100. This improves the effective cross-node rooflines by about 4x since our FLOPs/chip also double. Now we may start to worry about whether we're bottlenecked at the node level rather than the scale-out level.

Grace Hopper: NVIDIA also sells GH200 and GB200 systems which pair some number of GPUs with a Grace CPU. For instance, a GH200 has 1 H200 and 1 Grace CPU, while a GB200 system has 2 B200s and 1 Grace CPU. An advantage of this system is that the CPU is connected to the GPUs using a full bandwidth NVLink connection (called NVLink C2C), so you have very high CPU to GPU bandwidth, useful for offloading parameters to host RAM. In other words, for any given GPU, the bandwidth to reach host memory is identical to reaching another GPU's HBM.

Appendix B: More networking details

Here's a diagram of an NVLink 4 switch. There are 64 overall NVLink4 ports (each uses 2 physical lanes), and a large crossbar that handles inter-lane switching. TPUs by contrast use optical switches with mirrors that can be dynamically reconfigured.

Figure: a lower level view of a single NVLink4 Switch.

At each level we can be bottlenecked by the available link bandwidth or the total switch bandwidth.

- **Node level:** at the node level, we have 4 * 1.6TB/s = 6.4TB/s of NVSwitch bandwidth, but each of our 8 GPUs can only egress 450GB/s into the switch, meaning we actually have a peak bandwidth of 450e9 * 8 = 3.6TB/s (full-duplex) within the node.
- **SU/leaf level:** at the SU level, we have 8 switches connecting 32 nodes in an all-to-all fashion with 1x400 Gbps Infiniband. This gives us 8 * 32 * 400 / 8 = 12.8TB/s of egress bandwidth from the nodes, and we have 8 * 1.6TB/s = 12.8TB/s at the switch level, so both agree precisely.
- **Spine level:** at the spine level, we have 16 switches connecting 32 leaf switches with 2x400 Gbps links, so we have 32 * 16 * 400 * 2 / 8 = 51.2TB/s of egress bandwidth. The 16 switches give us 16 * 1.6TB/s = 25.6TB/s of bandwidth, so this is the bottleneck at this level.

Per GPU, this gives us 450GB/s of GPU to GPU bandwidth at the node level, 50GB/s at the SU level, and 25 GB/s at the spine level.

GPU empirical AR bandwidth:

Figure: AllReduce bandwidth on an 8xH100 cluster (intra-node, SHARP disabled).

TPU v5p bandwidth (1 axis):

Figure: AllReduce bandwidth on a TPU v5p 4x4x4 cluster (along one axis).

Here's AllGather bandwidth as well:

Figure: AllGather bandwidth on an 8xH100 cluster (intra-node).

Figure: AllGather bandwidth on a TPU v5e 8x16 cluster (along one axis).

More on AllToAll costs:

Here we can compare the approximation

\min(K / Z) * (Z - 1) / Zmin(K/Z)∗(Z−1)/Z \min(K / Z) * (Z - 1) / Z
to the true value of
(1 - ((Z - 1) / Z) ** K) * (Z - 1) / Z(1−((Z−1)/Z)∗∗K)∗(Z−1)/Z (1 - ((Z - 1) / Z) ** K) * (Z - 1) / Z
. They're similar except for small values of
ZZ Z
.

Figure: a comparison of the approximate and true cost of a ragged AllToAll as the number of shards increases.

Footnotes

1. The GPU Tensor Core is the matrix multiplication sub-unit of the SM, while the TPU TensorCore is the umbrella unit that contains the MXU, VPU, and other components.[↩]
2. NVIDIA doesn't have a good name for this, so we use it only as the best of several bad options. The Warp Scheduler is primarily the unit that dispatches work to a set of CUDA cores, but we use it here to describe the control unit and the set of cores it controls.[↩]
3. Although SMs are independent, they are often forced to coordinate for peak performance because they all share a capacity-limited L2 cache.[↩]
4. Historically, before the introduction of the Tensor Core, the CUDA cores were the main component of the GPU and were used for rendering, including ray-triangle intersections and shading. On today's gaming GPUs, they still do a bulk of the rendering work, while TensorCores are used for up-sampling (DLSS), which allows the GPU to render at a lower resolution (fewer pixels = less work) and upsample using ML.[↩]
5. NVIDIA doesn't share many TC hardware details, so this is more a guess than definite fact – certainly, it doesn't speak to how the TC is implemented. We know that a V100 can perform 256 FLOPs/TC/cycle. An A100 can do 512, H100 can do 1024, and while the B200 details aren't published, it seems likely it's about 2048 FLOPs/TC/cycle, since `2250e12 / (148 * 4 * 1.86e9)` is about 2048. Some more details are confirmed here.[↩]
6. In Ampere, the Tensor Core could be fed from a single warp, while in Hopper it requires a full SM (warpgroup) and in Blackwell it's fed from 2 SMs. The matmuls have also become so large in Blackwell that the arguments (specifically, the accumulator) no longer fit into register memory/SMEM, so Blackwell adds TMEM to account for this.[↩]
7. Warps scheduled on a given SM are called "resident".[↩]
8. Technically, the L2 cache is split in two, so half the SMs can access 25MB a piece on an H100. There is a link connecting the two halves, but at lower bandwidth.[↩]

9. The fact that the L2 cache is shared across all SMs effectively forces the programmer to run the SMs in a fairly coordinated way anyway, despite the fact that, in principle, they are independent units.[↩]
10. While NVIDIA made a B100 generation, they were only briefly sold and produced, allegedly due to design flaws that prevented them from running close to their claimed specifications. They struggled to achieve peak FLOPs without throttling due to heat and power concerns.[↩]
11. Before the deep learning boom, GPUs ("Graphics Processing Units") did, well, graphics – mostly for video games. Video games represent objects with millions of little triangles, and the game renders (or "rasterizes") these triangles into a 2D image that gets displayed on a screen 30-60 times a second (this frequency is called the framerate). Rasterization involves projecting these triangles into the coordinate frame of the camera and calculating which triangles overlap which pixels, billions of times a second. As you can imagine, this is very expensive, and it's just the beginning. You then have to color each pixel by combining the colors of possibly several semi-opaque triangles that intersect the ray. GPUs were designed to do these operations extremely fast, with an eye towards versatility; you need to run many different GPU workloads (called "shaders") at the same time, with no single operation dominating. As a result, consumer graphics-focused GPUs can do matrix multiplication, but it's not their primary function.[↩]
12. It's notable that this intensity stays constant across recent GPU generations. For H100s it's 33.5 / 3.5 and for B200 it's 80 / 8. Why this is isn't clear, but it's an interesting observation.[↩]
13. The term node is overloaded and can mean two things: the NVLink domain, aka the set of GPUs fully connected over NVLink interconnects, or the set of GPUs connected to a single CPU host. Before B200, these were usually the same, but in GB200 NVL72, we have an NVLink domain with 72 GPUs but still only 8 GPUs connected to each host. We use the term node here to refer to the NVLink domain, but this is controversial.[↩]
14. NVLink has been described to me as something like a souped-up PCIe connection, with low latency and protocol overhead but not designed for scalability/fault tolerance, while InfiniBand is more like Ethernet, designed for larger lossy networks.[↩]
15. Full-duplex here means 25GB/s each way, with both directions independent of each other. You can send a total of 50GB/s over the link, but at most 25GB/s in each direction.[↩]
16. For instance, Meta trained LLaMA-3 on a datacenter network that differs significantly from this description, using Ethernet, a 3 layer switched fabric, and an oversubscribed switch at the top level.[↩]
17. You can also think of each GPU sending its chunk of size
\text{bytes} / Nbytes/N \text{bytes} / N
to each of the other
N - 1N−1 N - 1
GPUs, for a total of
(N - 1) * N * bytes / N(N−1)∗N∗bytes/N (N - 1) * N * bytes / N
bytes communicated, which gives us[↩]
18. The true cost is actually
$$(1-(Z-1Z)K)\cdot Z-1Z$$
the expected number of distinct outcomes in
KK K
dice rolls, but it is very close to the approximation given. See the Appendix for more details.[↩]

Miscellaneous

*Work done at Google DeepMind, now at MatX.

Citation

For attribution in academic contexts, please cite this work as:

```
Austin et al., "How to Scale Your Model", Google DeepMind, online, 2025.
```

or as a BibTeX entry:

```
@article{scaling-book,
  title = {How to Scale Your Model},
  author = {Austin, Jacob and Douglas, Sholto and Frostig, Roy and Levskaya, Anselm and
Chen, Charlie and Vikram, Sharad
  and Lebron, Federico and Choy, Peter and Ramasesh, Vinay and Webson, Albert and Pope,
Reiner},
  publisher = {Google DeepMind},
  howpublished = {Online},
  note = {Retrieved from https://jax-ml.github.io/scaling-book/},
  year = {2025}
}
```