Multi GPU

Prateek Shukla

Let’s train a 1T parameter model on 10T tokens with H100

Empirical observation shows it takes roughly 6 operations (FLOPs) per parameter for every token of training data (forward pass + backward pass + update).

Total FLOPs=6×(10^12)×(10^13)=6×10^25 FLOPs

Let’s say we can get 1000tflops per second

$$\text{Time} = \frac{6 \times 10^{25}}{10^{15}} = 6 \times 10^{10} \text{ seconds}$$

That’s around 1900 years!!

This is why we need multiple GPUs

Multiple GPUs

Instead of spending thousands of years waiting for operations to complete we can just connect multiple GPUs together and get higher throughput and run bigger models. This is powered by two technologies

NVLink Gen 4

Direct GPU-to-GPU interconnect bypassing the CPU.

- 900 GB/s Bidirectional Bandwidth.
- 7x Faster than PCIe Gen 5.

NVSwitch

Enables every GPU in a node to talk to every other GPU at full speed simultaneously

H100 clusters vs older clusters

In traditional computing, GPUs are discrete units that communicate over slower PCIe lanes. The H100 paradigm shifts this by creating a "mesh" where every GPU can access the memory of every other GPU at extremely high speeds.

An 8-GPU cluster can behave as if it has a single pool of memory and compute cores

This architecture allows systems to scale from 8 GPUs in a single server to clusters of 256 GPUs (using the NVLink Switch System) that communicate at native chip speeds.

H100 DGX system or a node

It features 8 NVIDIA H100 GPUs using the SXM5 form factor.

The board includes 4 third-generation NVSwitches

The GPUs and switches are connected via fourth-generation NVLink, providing a massive 900 GB/s bandwidth per GPU

8 Network interface cards(NICs) NVIDIA ConnectX-7: specialized processor designed to offload networking tasks from the main CPU.

4x OSFP (Octal Small Form-factor Pluggable) cages

The Nvidia H100 Superpod

An nvidia DGX h100 superpod have around 127-128 GPUs organized in 4 SUs(Scalable units) each having 32 DGX H100 nodes.

The number 32 allows for a perfectly balanced "Level 1" network using standard switch port counts (typically 64 ports per switch).

The H100 SuperPod is designed around the fact that a single DGX H100 node has 8 separate NVIDIA ConnectX-7 Network Interfaces (HCAs) for compute, one for each GPU.

Nvlink

Each H100 GPU features 900 GB/s of bidirectional bandwidth. This is approximately 7x faster than the standard PCIe Gen5 interface used to connect the GPU to the CPU. Nvlink allows GPUs to use this to communicate between their HBM and skip the CPU path.

H100 uses 18 individual NVLink "links" to achieve this speed. Unlike previous generations, the 4th Gen NVLink is more density-optimized, using only two high-speed differential pairs per link (down from four), allowing more links to be packed into the same space.

Nvswitch 3

On a standard 8-GPU HGX board, four NVSwitch chips are used to connect all eight H100 GPUs. 4 - 5 Nvlink ports are connected to a single switch

The switches create a fully non-blocking interconnect. This means every GPU can talk to any other GPU at full 900 GB/s speed simultaneously, without waiting in line for data lanes to clear

NVIDIA uses NVSwitch chips in external trays called NVLink Switch Systems. These connect up to 256 H100 GPUs (32 server racks) into a single "SuperPOD”

The NVSwitch itself has ALUs which can perform additions/reductions in the switch itself, this is one of the most important feature which results in massive performance gains.

Connectx

Inside the server box, GPUs talk via NVLink (900 GB/s). Once data needs to leave the box to reach another server, it hits ConnectX-7. There are also 8 Compute Network Interfaces (ConnectX-7) in a DGX.

This is the bottleneck. Your training speed is determined by how efficiently you manage this 18x drop in bandwidth. ConnectX-7 exists to minimize the penalty of leaving the box.

ConnectX7 allows the network to read/write GPU memory without the CPU knowing.

The NIC works with the network switches to sum gradients in flight. This turns the network traffic from O(N) (linear growth with cluster size) to O(1) (constant traffic). This is the only reason large-scale training scales linearly

The OSPF cages

These 4 physical cages provide 8 separate 400Gb/s network links (totaling 3.2 Tb/s of bandwidth) by using "Twin-port" technology. They connect internally to 8x NVIDIA ConnectX-7 network cards.

This allows for GPU Direct RDMA, enabling GPUs to talk to GPUs in other DGX nodes without burdening the system CPU.

The 4 OSFP cages are exclusively for the Compute Fabric. Storage traffic does not touch those cables.

The storage fabric

This helps in loading massive datasets into the GPUs and saving checkpoints

This uses separate PCIe cards located in the standard PCIe slots on the back of the chassis, not the OSFP cages.

This is usually a Fat Tree or standard leaf-spine network. unlike the "Rail" network, any DGX node needs to be able to talk to any storage array.

The rail aligned system

In a standard network, a server might have one cable that carries traffic for all its components. In a DGX SuperPOD, the network is physically split into 8 parallel, isolated networks these are the "Rails."

Inside a Node, there are 8 GPUs (numbered 0 to 7). Rail 1 connects GPU 0 from every single node in the cluster to the same set of switches. Rail 2 connects GPU 1 from every single node to a different set of switches. And so on

Traffic on Rail 1 never interferes with traffic on Rail 2 at the leaf switch level. This creates 8 independent planes of connectivity across the entire cluster.

The rail system leverages NVIDIA's SHARP technology, which offloads data operations to the network switches themselves.

The rail optimized fat tree

The SuperPOD uses a 3-tier hierarchy (Leaf, Spine, Core) to connect the SUs together

Layer 1: The Leaf Layer (Connection to Nodes)

Layer 2: The Spine Layer (Connection between SUs)

Layer 3: The Core / Super-Spine Layer (Maximum Scale)

These layers are nothing but a bunch of Quantumn 2 infiniband switches connected together

Quantum-2 QM9700 InfiniBand Switch

It provides ultra high bandwidth low latency GPU-GPU interconnect

It has 64 ports of 400Gb/s connectivity

When you look at the front of the switch, you will see 32 cages. These use the OSFP (Octal Small Form-factor Pluggable) standard. Each OSFP cage actually carries two separate 400Gb/s links. 32 Cages × 2 Links = 64 Logical Ports.

Since the QM9700 has 64 ports, it is the perfect size for a SU of 32 nodes.

32 Ports (Downlinks): Connect to the 32 Nodes in the SU (e.g., Rail 1 connects to GPU 0 on all 32 nodes).

32 Ports (Uplinks): Connect "up" to the Spine Switches (Layer 2) to talk to other SUs.

The leaf layer

This is where the cables physically leave the back of the DGX H100 server.

Rail Segregation: There are 8 separate "planes" of switches:

Rail 1 Switches: Connect only to the 1st network card (GPU 0) of every node.

Rail 8 Switches: Connect only to the 8th network card (GPU 7) of every node.

The leaf switches handle traffic within the local Scalable Unit. If GPU 0 on Node 1 needs to talk to GPU 0 on Node 2 (in the same SU), traffic goes Node -> Leaf Switch -> Node. It never needs to go higher up the chain.

The QM9700 uses SHARPv3, which is 32x more capable than the previous generation, allowing it to handle complex AI math.

Adaptive routing in the leaf layer

In the leaf layer, adaptive routing is critical for upstream traffic.

When a packet arrives at a leaf switch from a GPU and needs to travel to a different SU, it must go up to a spine switch. In a non-blocking or even oversubscribed fat-tree, there are multiple available spine switches it can use.

Instead of using a static hash (which always sends a specific flow to the same spine switch, potentially causing collisions), the Quantum-2 switch hardware monitors the queue depth and congestion levels of all uplink ports

The switch dynamically sends packets to the least congested spine switch link on a per-packet or per-message basis. This ensures that even if one path to the spine is clogged, traffic flows smoothly through others.

The spine layer

The Spine layer connects the Leaf switches together. This allows nodes in SU-1 to talk to nodes in SU-2.

Spine Groups: The spine switches are also organized into groups that align with the rails.

Spine Group 1: Connects only to the Leaf switches that handle Rail 1.

Isolation: This ensures that traffic from Rail 1 never accidentally "leaks" over to Rail 2's cables, which would cause congestion (blocking).

If GPU 0 in SU-1 needs to talk to GPU 0 in SU-2, the traffic flows: Node (SU1) -> Leaf (Rail 1) -> Spine (Group 1) -> Leaf (Rail 1) -> Node (SU2)

Adaptive routing in the spine layer

In the spine layer, adaptive routing manages traffic traversing the core of the network.

Typically, in a standard 2-layer fat-tree, there is only one path "down" from a specific spine switch to a specific destination leaf switch. However, adaptive routing is still vital here for handling faults and multi-pathing if parallel links exist

If the buffers towards a specific leaf switch are full, the spine switches communicate this backpressure. Adaptive routing features in Quantum-2 (like SHIELD) help isolate this congestion so it doesn't spread to other unaffected traffic flows in the spine.

The Core / Super-Spine Layer (Maximum Scale)

For massive clusters (e.g., 127 nodes or more), a third layer is added.

Function: This layer connects multiple "Pods" (clusters of SUs) together.

Optics: At this level, the system often switches to 800G optical transceivers to reduce the number of cables required, while logically splitting them back into two 400G links.

SHIELD

In traditional InfiniBand networks, if a cable breaks or a link flaps, it takes around 5 - 30 seconds for the software controller to calculate a new routing table and then push it to all the switches. This crashes entire training run

SHIELD moves this recovery logic directly into the switch hardware ASIC.

If a switch sees a link go down, it doesn't just drop packets. It immediately checks for alternative valid paths in its local hardware table. If it finds one then it updates its own table to avoid the bad node and goes to healthy neighbor, if it fails then it sends a hardware signal to the neighbour so that no traffic is sent to it in future.

P2p mechanism for communication between GPUs

The P2P (Peer-to-Peer) CUDA mechanism is a feature that allows two GPUs to communicate without using CPUs

We can do p2p memory copy explicitly or we can get p2p direct access and the whole transfer uses Nvlink.

Unified Virtual addressing enables this whole system

Unified Virtual Addressing (UVA)

UVA allows the CPU and GPU to share a single virtual address space.

Before UVA: The CPU had its own memory pointers and the GPU had its own. You had to manually manage which pointer pointed to which physical memory.

With UVA: The system determines specifically where the data is physically located (in the system RAM or on the H100's HBM3 memory) based on the pointer value alone.

You need to enable Peer access using cudaDeviceEnablePeerAccess otherwise this feature can’t work

cudaDeviceEnablePeerAccess

Without this call the GPU might not use Nvlink and go through the PCIe

```
#define GPU_COUNT 8

void enable_full_p2p() {
    // Iterate through each GPU to set it as the "current" device
    for (int i = 0; i < GPU_COUNT; i++) {
        cudaSetDevice(i);

        // Iterate through every *other* GPU to enable it as a peer
        for (int j = 0; j < GPU_COUNT; j++) {
            if (i == j) continue; // Cannot be a peer to yourself

            int can_access = 0;
            cudaDeviceCanAccessPeer(&can_access, i, j);

            if (can_access) {
                cudaError_t err = cudaDeviceEnablePeerAccess(j, 0);

                if (err != cudaSuccess && err != cudaErrorPeerAccessAlreadyEnabled) {
                    printf("Error enabling P2P from %d to %d: %s\n",
                           i, j, cudaGetErrorString(err));
                }
            }
        }
    }
```

A few important points

cudaDeviceEnablePeerAccess(peerDevice, 0) is unidirectional. If you need to copy back and forth or access memory kernels on both sides, you must call it on both devices.

If you don’t enable this then you might fall back to a slower PCIe path

On H100 servers, this function will automatically utilize NVLink (900 GB/s) if available; otherwise, it uses PCIe Gen5

cudaMemCpyPeer

enables the direct transfer of data between the memory of two separate GPUs without involving the host (CPU) memory

```
cudaError_t cudaMemcpyPeer(
    void*        dst,       // Destination device pointer
    int          dstDevice, // Destination device ID (e.g., 1)
    const void*  src,       // Source device pointer
    int          srcDevice, // Source device ID (e.g., 0)
    size_t       count      // Size of data in bytes
);
```

You must enable cudaDeviceEnablePeerAccess to use Nvlink for copies

```
    // Allocate on Device 0 (Source)
    CHECK_CUDA(cudaSetDevice(dev0));
    CHECK_CUDA(cudaMalloc(&d_src, size));

    float *h_src = (float*)malloc(size);
    for (int i = 0; i < 1024; i++) h_src[i] = 1.0f;
    CHECK_CUDA(cudaMemcpy(d_src, h_src, size, cudaMemcpyHostToDevice));

    CHECK_CUDA(cudaSetDevice(dev1));
    CHECK_CUDA(cudaMalloc(&d_dst, size));
    CHECK_CUDA(cudaMemset(d_dst, 0, size));

    int canAccessPeer = 0;
    CHECK_CUDA(cudaDeviceCanAccessPeer(&canAccessPeer, dev0, dev1));

    if (canAccessPeer) {
        printf("Peer access available between Device %d and %d.\n", dev0, dev1);

        CHECK_CUDA(cudaSetDevice(dev0));
        CHECK_CUDA(cudaDeviceEnablePeerAccess(dev1, 0));

        CHECK_CUDA(cudaSetDevice(dev1));
        CHECK_CUDA(cudaDeviceEnablePeerAccess(dev0, 0));
    } else {
        printf("Peer access NOT available. Fallback to system memory copy likely.\n");
    }

    printf("Transferring data from Device %d to Device %d...\n", dev0, dev1);
    CHECK_CUDA(cudaMemcpyPeer(d_dst, dev1, d_src, dev0, size));
```

Truth about raw p2p

On an H100 HGX board, you don't just have 8 GPUs connected in a line. You have a complex mesh connected via NVSwitches.

P2P: You have to manually manage which link you are traversing. If GPU 0 talks to GPU 7, does it go direct? Does it hop through GPU 3?

The H100 has a bidirectional NVLink bandwidth of 900 GB/s. If you issue a standard LD/ST (Load/Store) instruction across the NVLink fabric from an SM, you are likely using a single distinct NVLink lane or a subset of them. You will struggle to saturate that pipe. You are using a straw to drink from a firehose.

These are the reasons because of which we need libraries such as NCCL for managing multi GPU connections