Multi-GPU Parallelism: Collectives, Interconnects, CUDA Execution, and Distributed Training

Language Modeling from Scratch · lecture notes · rebuilt for NVIDIA Blackwell, late 2026

Section 1: The Multi-GPU Scaling Problem, Training Memory, and the Interconnect Hierarchy

KEY IDEA

A single modern GPU delivers enormous floating-point throughput, yet training a frontier language model on one device fails for two independent reasons: **memory capacity** and **compute throughput**. Even as each hardware generation brings faster processors and larger memory stacks, frontier models continue to grow alongside them, making multi-level parallelism a permanent requirement rather than a temporary workaround.

DEFINITION

To see why a single GPU runs out of memory first, examine what fills a GPU's High Bandwidth Memory (HBM) during training. Imagine the total HBM footprint as a single horizontal bar divided into two contiguous regions: **Model State** on the left (static with respect to batch size, scaling with parameter count) and **Dynamic State** on the right (scaling with batch size, sequence length, hidden width, and depth).

The **Model State** portion consists of three blocks:

1. **Parameters**: the model weights themselves. In sparse Mixture-of-Experts (MoE) architectures, this block includes every expert's weights even though only a small subset of experts is active for any given token.
2. **Gradients**: one gradient value per trainable parameter, having the exact same tensor shapes as the parameters.
3. **Optimizer States (AdamW)**: recall that AdamW tracks per-parameter running averages of the first moment (momentum) and second moment (variance), and in mixed-precision training also maintains full-precision master copies of the weights. Together, these optimizer states typically occupy more memory than the parameters and gradients combined.

DEFINITION

The **Dynamic State** portion consists of **Saved Activations**: intermediate outputs from the forward pass that must remain resident in HBM so the backward pass can compute parameter gradients. Saved activations grow proportionally to $\text{batch size} \times \text{sequence length} \times \text{hidden width} \times \text{number of layers}$.

WORKED EXAMPLE

Consider a concrete example to quantify this footprint. Suppose we train a dense model with 10 billion parameters using mixed precision (2 bytes per BF16 parameter, 2 bytes per BF16 gradient, and 12 bytes per parameter in FP32 for AdamW's master weight copy, momentum, and variance). The static Model State alone requires:

$$\text{Model State} = 10 \times 10^9 \text{ params} \times (2 + 2 + 12) \text{ bytes/param} = 160 \times 10^9 \text{ bytes} = 16$$

Before storing a single saved activation for the backward pass, 160 GB of HBM is already consumed. Scaling to a 70-billion-parameter model pushes Model State alone to $70 \times 16 = 1{,}120$ GB—far beyond any single GPU's HBM capacity. Even if a smaller model does fit on one GPU, training it on trillions of tokens with a single device's floating-point operations per second (FLOPs) would take months or years; combining FLOPs across many GPUs is required to shorten training time.

COMMON MISCONCEPTION

You might assume that once we distribute tensors across multiple GPUs, training speeds up linearly with GPU count. That breaks down because of the unifying theme of modern machine learning systems: **compute (Tensor Cores and ALUs) is far from data, so you must orchestrate computation to avoid data-transfer bottlenecks.**

DEFINITION

Recall that on a single GPU, we fight this bottleneck by minimizing traffic between off-chip HBM and on-chip SRAM using **kernel fusion** and **tiling** into on-chip SRAM. Across multiple GPUs, the exact same principle applies at a larger scale: we minimize cross-GPU and cross-node traffic using **replication** (copying data onto multiple GPUs so each can compute locally without fetching it) and **sharding** (splitting a tensor across GPUs so no single GPU must store the whole object). Different parallelism strategies shard different regions of the HBM footprint bar: Data Parallelism and Fully Sharded Data Parallelism (FSDP) shard batches (and, for FSDP, model state); Tensor and Expert Parallelism shard model width; Pipeline Parallelism shards model depth.

DEFINITION

Every decision about whether to replicate or shard is governed by the **Generalized Memory and Interconnect Hierarchy**. Picture this hierarchy as a four-tier pyramid widening from top to bottom, with a vertical arrow along its side pointing downward labeled **Capacity ↑, Bandwidth ↓**:

- **Tier 1 (Top, narrowest, fastest) — L1 Cache / Shared Memory**: Single GPU, on-chip SRAM. Smallest capacity, highest bandwidth, lowest latency.
- **Tier 2 — HBM**: Single GPU, off-chip DRAM. On an NVIDIA Blackwell B200 GPU, HBM provides **8 TB/s** of bandwidth.
- **Tier 3 — NVLink 5.0 / NVSwitch**: Intra-node (8-GPU node) or rack-scale (72-GPU NVL72 rack) GPU-to-GPU interconnect. On a B200, NVLink 5.0 provides **900 GB/s per direction** per GPU (1.8 TB/s counting both directions).
- **Tier 4 (Bottom, widest, slowest) — InfiniBand / Ethernet**: Inter-node (Multi-Node Pod) and inter-pod (Multi-Pod Cluster) network links. An InfiniBand pod link provides roughly **~0.05 TB/s** (50 GB/s), with Ethernet connecting pods across the cluster.

WORKED EXAMPLE

Every tier trades bandwidth for capacity and physical reach, and crossing a boundary drops bandwidth sharply. Let us compute the exact transfer time of moving a 1 GB (0.001 TB) tensor across three tiers to make these gaps concrete:

- **Within B200 HBM (8 TB/s)**: $t = \frac{0.001\text{ TB}}{8\text{ TB/s}} = 0.125\text{ ms}$.
- **Across B200 NVLink 5.0 (0.9 TB/s per direction)**: $t = \frac{0.001\text{ TB}}{0.9\text{ TB/s}} \approx 1.11\text{ ms}$ ($8.9\times$ slower than HBM).
- **Across InfiniBand (0.05 TB/s)**: $t = \frac{0.001\text{ TB}}{0.05\text{ TB/s}} = 20.0\text{ ms}$ ($18\times$ slower than NVLink 5.0 and $160\times$ slower than HBM).

KEY IDEA

This $18\times$ cliff between NVLink 5.0 and InfiniBand dictates every distributed training architecture we will build: parallelism strategies must match communication frequency to tier speed, keeping frequent, fine-grained transfers inside NVLink and pushing infrequent transfers across InfiniBand or Ethernet.

Before we can design communication algorithms or sharding schemes on top of this hierarchy, we need to understand how software actually executes work on one GPU and coordinates execution across many GPUs simultaneously.

✎ Check Your Understanding

1. **Yes/No with justification:** If a model's parameters fit comfortably inside a single GPU's HBM (say, 40 GB of weights on an 80 GB GPU), is it guaranteed that you can train the model on that single GPU?
   Answer: No. Training HBM must also hold gradients, AdamW optimizer states (momentum, variance, and master copies), and saved forward activations, which together easily exceed the size of the parameters alone.
2. **Quick computation:** Moving a $2\ \text{GB}$ tensor takes how many times longer over a $0.05\ \text{TB/s}$ InfiniBand link than over a $0.9\ \text{TB/s}$ (per direction) B200 NVLink 5.0 link?
   Answer: $0.9/0.05 = 18$ times longer ($40\ \text{ms}$ vs. $2.22\ \text{ms}$), because transfer time is inversely proportional to link bandwidth.

Section 2: Multi-GPU Execution Model — CUDA Streams, Processes, Ranks, and Synchronization

Writing distributed GPU code without understanding how the host CPU dispatches work to a GPU—and how multiple GPUs coordinate—leads directly to race conditions, broken benchmarks, and hidden hardware contention. We need an accurate mental model of both single-GPU asynchronous execution and multi-GPU process coordination.

Single-GPU Asynchronous Execution and CUDA Streams

Picture the hardware and software boundary inside a single Blackwell GPU node as two connected boxes: the **Host CPU** (running Python / PyTorch) on the left, and the **Blackwell GPU (CUDA Device)** on the right.

DEFINITION

When Python executes a matrix multiplication (`Launch GEMM`) or a collective communication call (`Launch NCCL op`), the Host CPU does **not** wait for the GPU to finish the math. Instead, CUDA kernel launches are **asynchronous**: the CPU simply pushes a command descriptor across an arrow labeled `async` into a first-in, first-out (FIFO) queue on the GPU called a **CUDA stream**, and Python immediately returns to execute the next line of code.

Inside the GPU box, we maintain at least two distinct streams:

DEFINITION

- **Compute Stream**: a FIFO queue holding math kernels such as matrix multiplication (`matmul`), `GeLU` , and attention.

DEFINITION

- **Comm Stream (NCCL)**: a separate FIFO queue holding collective communication kernels (`send` , `recv` , `reduce`).

DEFINITION

Work placed in the **same stream** executes strictly in FIFO order: kernel 2 will not begin until kernel 1 finishes. Work placed in **separate streams** can run **concurrently** on the GPU. When a kernel in the Comm Stream depends on the output of a kernel in the Compute Stream (for example, sending a gradient only after the backward matmul finishes computing it), the two streams coordinate via a **CUDA Event**—a lightweight hardware marker recorded in one stream that the other stream waits on before proceeding, without involving the Host CPU.

COMMON MISCONCEPTION

From both streams, ready kernels are dispatched to the GPU's **Streaming Multiprocessors (SMs)**, which contain the **Tensor Cores** and **L1 / Shared Memory (SMEM)**, and connect to **HBM** (8 TB/s on B200, which in turn connects outward to NVLink 5.0 and PCIe/RDMA). Notice a critical hardware fact highlighted in the architecture: **NCCL communication runs as CUDA kernels on the SMs, so both math kernels and NCCL kernels share the same SMs and HBM bandwidth.** You might assume that communication runs on a completely separate dedicated DMA chip with zero impact on math kernels, but that is false for NCCL collectives—overlapping compute and communication shares SM and HBM resources, a trade-off we will quantify in Section 9.

KEY IDEA

Because launches are asynchronous, if the Host CPU wants to wait until every queued kernel on this GPU has actually finished executing, it must call `torch.cuda.synchronize()` . Without `torch.cuda.synchronize()` , a Python timer placed around GPU calls measures only the microseconds of CPU launch overhead, not GPU execution time.

Multi-GPU Execution: Processes, Ranks, and Barriers

COMMON MISCONCEPTION

Now scale from one GPU to W GPUs (say, $W = 4$). You might think the cleanest approach is to run a single Python process with four threads, each managing one GPU. That fails in Python because of the **Global Interpreter Lock (GIL)**: only one thread can execute Python bytecode at a time, causing severe CPU-side launch contention across GPUs, and sharing a single process complicates CUDA device context isolation.

Instead, distributed GPU training runs **one OS process per GPU**:

- Process 0 (`rank = 0` , designated `master` for rendezvous coordination) owns `GPU 0` (`cuda:0`).

- Process 1 (`rank = 1`) owns `GPU 1` (`cuda:1`).
- Process 2 (`rank = 2`) owns `GPU 2` (`cuda:2`).
- Process 3 (`rank = 3`) owns `GPU 3` (`cuda:3`).

DEFINITION

All four GPUs attach underneath to a shared **NCCL Communicator** running over NVLink, NVSwitch, or RDMA. Every process executes the exact same Python script, branching only on its integer `rank` $(0, 1, \ldots, W-1)$ out of the total **world size** W to determine which shard of data or weights it owns. This execution model is called **SPMD (Single Program, Multiple Data)**.

Because we now have both asynchronous GPU streams *and* independent OS processes, there are **two distinct synchronization primitives** that students routinely confuse:

DEFINITION

- `torch.cuda.synchronize()` : blocks **this** process's Host CPU until **this** GPU's queued CUDA kernels finish. It does nothing to other processes or other GPUs.

DEFINITION

- `dist.barrier()` : blocks **all** W **ranks** until every rank has reached the barrier call.

COMMON MISCONCEPTION

You might think calling `torch.cuda.synchronize()` aligns all GPUs, or that calling `dist.barrier()` guarantees all prior GPU math is done before a timer stops. Both assumptions are incomplete. Because ranks execute asynchronously, without a barrier or collective dependency one rank can race far ahead of another.

WORKED EXAMPLE

Let us trace a concrete example across two GPUs (`rank 0` and `rank 1`) to see both primitives in action. Suppose both ranks launch a GEMM on their Compute Stream that takes 2.0 ms on GPU 0 and 3.5 ms on GPU 1 (perhaps due to a momentary scheduling jitter), and Python launch overhead is 0.05 ms:

1. At $t = 0.00$ ms, both CPUs launch the GEMM. At $t = 0.05$ ms, both CPUs return from the Python call while the GPUs have barely started. If Rank 0 stops a host timer now (omitting `torch.cuda.synchronize()`), it falsely reports 0.05 ms instead of 2.0 ms.
2. If both ranks call `torch.cuda.synchronize()`, Rank 0's CPU unblocks at $t = 2.00$ ms and Rank 1's CPU unblocks at $t = 3.50$ ms. If Rank 0 immediately reads its timer without calling `dist.barrier()`, it reports 2.00 ms and moves on to the next iteration while Rank 1 is still 1.50 ms behind.

3. Calling `torch.cuda.synchronize()` **and then** `dist.barrier()` ensures that each GPU finishes its own stream locally and that no rank proceeds past $t = 3.50$ ms until every rank is finished. We will use this exact two-step synchronization pattern when benchmarking collectives in Section 5.

Now that we have W SPMD processes running concurrently on W GPUs connected by an NCCL communicator, they must exchange tensors during training. Writing manual send-and-receive loops for every pair of GPUs is error-prone and leaves network bandwidth unused. We instead express multi-GPU data movement through a small set of structured **collective communication primitives**.

✎ Check Your Understanding

1. **Fill-in-the-blank:** To make a kernel on a GPU's NCCL Comm Stream wait for a matrix multiplication on the Compute Stream without blocking the Host CPU, the two streams coordinate via a _____.
   Answer: CUDA Event. Recording an event in the Compute Stream and having the Comm Stream wait on it enforces ordering directly on the GPU while allowing the Host CPU to keep enqueuing work asynchronously.
2. **What would happen if…** you replaced the one-process-per-GPU SPMD design with a single Python process using 8 threads to control 8 GPUs?
   Answer: The threads would contend on Python's Global Interpreter Lock (GIL), serializing CPU kernel-launch code and losing CUDA device context isolation across GPUs.

Section 3: Collective Communication Primitives

DEFINITION

When W GPUs need to combine gradients or assemble sharded weight matrices, manually scheduling point-to-point `send` and `recv` messages between pairs of ranks is both tedious to program and difficult to optimize across different hardware topologies. **Collective operations** solve this by declaring *what* global communication pattern all W ranks in a process group should perform, leaving the collective library to schedule the underlying transfers cleanly and at peak hardware speed.

Every collective follows three simple naming rules:

- **Reduce** combines tensors elementwise across ranks using an associative and commutative operator (such as `sum`, `min`, or `max`).
- **Scatter** is the exact inverse of **Gather** (splitting one tensor across ranks vs. assembling shards from ranks into one tensor).

- The prefix **All-** means **every rank** receives the final result, rather than only a single root rank.

LLM training relies on three families of collectives: four **rooted foundations** (Broadcast, Scatter, Gather, Reduce), three symmetric **workhorses** (All-Gather, Reduce-Scatter, All-Reduce), and one **MoE routing primitive** (All-to-All). Let us examine each family with a concrete $W = 4$ example (ranks `R0`, `R1`, `R2`, `R3`).

Rooted Foundations: Broadcast, Scatter, Gather, and Reduce

DEFINITION

In a **rooted collective**, one designated rank (typically `R0`) acts as the sole source or sole destination.

DEFINITION

- **Broadcast (R0 → all ranks)**: Before the operation, `R0` holds the tensor `[0, 1, 2, 3]` and `R1`, `R2`, `R3` have empty buffers. Arrows fan out from `R0` to every rank. Afterward, all four ranks hold an identical copy of `[0, 1, 2, 3]`. Use case: rank 0 loads initial model weights or a checkpoint from disk and broadcasts them to all other ranks so training starts from identical parameters.

DEFINITION

- **Scatter (split from R0 → all ranks)**: Before the operation, `R0` holds four contiguous chunks `[0]`, `[1]`, `[2]`, `[3]`, while `R1` – `R3` are empty. Arrows fan out from `R0` sending chunk r to rank r. Afterward, `R0` holds `[0]`, `R1` holds `[1]`, `R2` holds `[2]`, and `R3` holds `[3]`. Scatter is the conceptual stepping stone to Reduce-Scatter.

DEFINITION

- **Gather (inverse of Scatter → R0)**: Before the operation, `R0` holds `[0]`, `R1` holds `[1]`, `R2` holds `[2]`, and `R3` holds `[3]`. Arrows converge from all four ranks onto `R0`. Afterward, `R0` holds the concatenated tensor `[0, 1, 2, 3]` while `R1` – `R3` hold nothing new. Gather is the conceptual stepping stone to All-Gather.

DEFINITION

- **Reduce (combine with op → R0)**: Before the operation, `R0` holds `[0]`, `R1` holds `[1]`, `R2` holds `[2]`, and `R3` holds `[3]`. All four arrows feed into a reduction node (`+` for `SUM`), which outputs the single combined result to `R0`. Afterward, `R0` holds `[6]` (since $0 + 1 + 2 + 3 = 6$) and `R1` – `R3` do not receive the sum. Reduce is the stepping stone to All-Reduce.

COMMON MISCONCEPTION

Notice a subtle mathematical requirement on Reduce: the operator **must be associative and commutative** (such as sum, min, or max). You might think any binary operation like subtraction could be plugged into a reduction, but that breaks because the collective library (NCCL) decomposes reductions across rings or trees and reorders partial reductions freely to maximize bandwidth. Only associative, commutative operations guarantee the same logical result regardless of tree or ring order (up to minor floating-point rounding differences).

KEY IDEA

Rooted collectives are useful at startup and checkpointing, but training hot loops almost exclusively use symmetric **"All-"** variants so that no single rank acts as a bottleneck and all ranks remain symmetric in SPMD execution.

The Workhorses: All-Gather, Reduce-Scatter, and All-Reduce

1. All-Gather

DEFINITION

All-Gather performs a Gather whose destination is **every rank**, not just rank 0. Picture four input rows on the left— `R0` holds `[0]`, `R1` holds `[1]`, `R2` holds `[2]`, `R3` holds `[3]` —feeding into an `All-Gather` block, with arrows fanning out to four output rows on the right where **every rank** now holds the full concatenated tensor `[0, 1, 2, 3]`.

If the full output tensor has size S bytes, each rank already possesses its own shard of size S/W bytes and must receive the remaining $W-1$ shards from its peers. Therefore, the data volume moved per rank in an All-Gather is:

$$V_{\text{AG}} = \frac{W-1}{W} S$$

Let us break this formula down:

- S is the total byte size of the assembled output tensor.
- W is the world size (total number of participating ranks).
- S/W is the size of a single rank's shard.
- $(W-1)$ is the number of *other* ranks whose shards each rank must receive (and equivalently, the number of shards each rank must send out in a bandwidth-optimal schedule).
- Together, $\frac{W-1}{W} S$ is the exact volume of data each rank transfers over the interconnect to assemble the full S-byte tensor.

WORKED EXAMPLE

> For example, with $W = 4$ ranks and an assembled layer weight tensor of $S = 400\ \text{MB}$, each rank starts with $100\ \text{MB}$ and moves $\frac{4-1}{4} \times 400\ \text{MB} = 300\ \text{MB}$ across the network—not $400\ \text{MB}$, because it already holds its own $100\ \text{MB}$ shard locally.

All-Gather has two major LLM training use cases that we will implement later:

- **FSDP / ZeRO (Section 6)**: each GPU permanently stores only $1/W$ of a layer's weights, then runs All-Gather just before forward or backward compute to temporarily materialize the full weights.
- **Tensor Parallelism (Section 7)**: reassembling sharded activation columns across ranks after a column-sharded matrix multiplication.

2. Reduce-Scatter

> **DEFINITION**
> Reduce-Scatter is the exact communication dual of All-Gather. Each rank starts with a **full** tensor of size S bytes divided into W slices. Reduce-Scatter reduces each slice across all ranks (elementwise sum across the matching column) and then **scatters** one reduced slice to each rank so that rank r ends up with only slice r of the reduced tensor (size S/W bytes).

> **WORKED EXAMPLE**
> Trace the concrete $W = 4$ example from the diagram:
>
> - Input on `R0`: `[0, 1, 2, 3]`
> - Input on `R1`: `[1, 2, 3, 4]`
> - Input on `R2`: `[2, 3, 4, 5]`
> - Input on `R3`: `[3, 4, 5, 6]`
>
> Summing each column across all four ranks and assigning column r to rank r:
>
> - Column 0 $\rightarrow$ `R0`: $0 + 1 + 2 + 3 = 6$, so `R0` outputs `[6]`.
> - Column 1 $\rightarrow$ `R1`: $1 + 2 + 3 + 4 = 10$, so `R1` outputs `[10]`.
> - Column 2 $\rightarrow$ `R2`: $2 + 3 + 4 + 5 = 14$, so `R2` outputs `[14]`.
> - Column 3 $\rightarrow$ `R3`: $3 + 4 + 5 + 6 = 18$, so `R3` outputs `[18]`.

Just like All-Gather, each rank must send out the $W - 1$ slices owned by other ranks and receive $W - 1$ partial contributions for its own slice, so the data volume moved per rank is identical:

$$V_{\text{RS}} = \frac{W - 1}{W} S$$

For a $400\ \text{MB}$ gradient tensor across $W = 4$ ranks, Reduce-Scatter moves $300\ \text{MB}$ per rank and leaves each rank holding only its $100\ \text{MB}$ summed gradient shard. Its primary use case is summing backward gradients across data-parallel workers while keeping storage distributed.

3. All-Reduce = Reduce-Scatter + All-Gather

DEFINITION

In an **All-Reduce**, every rank starts with a full tensor of size S bytes and every rank ends with the **full** elementwise reduced tensor of size S bytes. You might imagine implementing All-Reduce as a rooted Reduce to `R0` followed by a Broadcast from `R0`, but that would funnel all W ranks' traffic through `R0`'s single network interface, collapsing throughput by a factor of W.

WORKED EXAMPLE

Instead, bandwidth-optimal All-Reduce decomposes into two consecutive symmetric stages: **Reduce-Scatter followed by All-Gather**. Trace our exact same $W = 4$ inputs through this two-stage pipeline:

1. **Inputs (4 ranks)**: `R0: [0, 1, 2, 3]`, `R1: [1, 2, 3, 4]`, `R2: [2, 3, 4, 5]`, `R3: [3, 4, 5, 6]`.
2. **Stage 1 — Reduce-Scatter** ($\frac{W-1}{W}S$ bytes per rank): `R0: [6]`, `R1: [10]`, `R2: [14]`, `R3: [18]`.
3. **Stage 2 — All-Gather** ($\frac{W-1}{W}S$ bytes per rank): every rank collects all four reduced shards, so `R0`, `R1`, `R2`, and `R3` all hold `[6, 10, 14, 18]`.

Adding the per-rank communication volume of both stages gives the exact volume of All-Reduce:

$$V_{\text{AR}} = V_{\text{RS}} + V_{\text{AG}} = \frac{2(W-1)}{W}S$$

WORKED EXAMPLE

For our $S = 400\ \text{MB}$ tensor across $W = 4$ ranks, Reduce-Scatter moves $300\ \text{MB}$ and All-Gather moves $300\ \text{MB}$, so total All-Reduce volume is $600\ \text{MB}$ per rank—**exactly $2\times$ the volume of Reduce-Scatter**.

KEY IDEA

All-Reduce is the core collective of Distributed Data Parallelism (DDP): after the backward pass, all workers All-Reduce their gradients so every rank holds identical gradients. Even more profoundly, the identity $\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$ provides the **key systems insight behind ZeRO and FSDP** (Section 6): breaking All-Reduce apart lets us insert the optimizer step *in the middle*, updating sharded parameters on each rank's $1/W$ slice before gathering weights for the next forward pass.

The MoE Routing Primitive: All-to-All

DEFINITION

All-to-All is the most general collective: instead of sending the *same* shard to every peer (as in All-Gather), each rank sends a **distinct** tensor slice to every other rank.

WORKED EXAMPLE

When all slices have the same size, All-to-All is a **distributed matrix transpose**. Visualize a 4×4 grid where each row is a sender rank and each column j is color-coded for destination rank j:

- Before All-to-All:
  - `R0 : [0, 1, 2, 3]`
  - `R1 : [4, 5, 6, 7]`
  - `R2 : [8, 9, 10, 11]`
  - `R3 : [12, 13, 14, 15]`
- After All-to-All (column $j \rightarrow$ rank j):
  - `R0 : [0, 4, 8, 12]`
  - `R1 : [1, 5, 9, 13]`
  - `R2 : [2, 6, 10, 14]`
  - `R3 : [3, 7, 11, 15]`

DEFINITION

All-to-All is essential for **sparse Mixture-of-Experts (MoE)** layers (Section 9): each rank starts with a batch slice of tokens and owns a subset of experts, and All-to-All routes each token to the GPU hosting its assigned expert. Unlike the balanced transpose above, MoE routing also requires **unbalanced splits**—for instance, if 60% of all tokens route to experts on `R2` while `R0` receives only 10%, `R2` becomes an overloaded straggler that stalls the entire layer. Furthermore, because every rank simultaneously talks to every other rank with unique data, All-to-All heavily stresses each GPU's **per-direction link bandwidth**, because every GPU sends and receives a slice from every peer at once.

These logical collectives define *what* data moves between ranks, but how fast they execute depends entirely on the physical wires, switches, and network protocols underneath them. We turn to the hardware interconnect fabric next.

✎ Check Your Understanding

1. **Quick computation:** Suppose $W = 8$ GPUs perform a Reduce-Scatter and then an All-Reduce on a tensor of size $S = 800\ \text{MB}$. How many megabytes does each rank transfer in Reduce-Scatter, and how many in All-Reduce?
   Answer: In Reduce-Scatter, each rank transfers $\frac{W-1}{W} S = \frac{7}{8} \times 800 = 700\ \text{MB}$. In All-Reduce, each rank transfers $\frac{2(W-1)}{W} S = 2 \times 700 = 1{,}400\ \text{MB}$.
2. **Yes/No with justification:** In a balanced All-to-All across 4 ranks, does every rank end up with the same output tensor?
   Answer: No. A balanced All-to-All performs a distributed matrix transpose where rank j receives column j from every rank, so each rank holds a distinct output tensor (unlike All-Gather or All-Reduce, where all ranks end with identical outputs).

Section 4: GPU Interconnects, Hardware Topology, and NCCL

Logical collective operations only run as fast as the physical interconnect topology allows. Moving from a classic multi-GPU workstation to a modern NVIDIA Blackwell datacenter cluster replaces shared buses and CPU-mediated networking with dedicated all-to-all GPU switch fabrics and CPU-bypassing remote memory access.

Classic Workstation vs. Blackwell Datacenter Fabric

Compare every scope of the interconnect hierarchy side by side:

Scope	Classic Workstation / Baseline	Blackwell Datacenter Fabric
Inside a GPU	On-chip SRAM ↔ HBM	B200 on-chip SRAM ↔ **HBM at 8 TB/s**
Intra-node / Rack	Shared **PCIe bus** (v7.0, 16 lanes = **242 GB/s**)	**NVLink 5.0 + NVSwitch** (**900 GB/s per direction** per B200; up to **72 GPUs** in GB200/GB300 NVL72)
Inter-node (Pod)	Standard Ethernet (**~200 MB/s** home baseline, via CPU)	**InfiniBand** (**~0.05 TB/s**, via HCA + RDMA) or **RoCE** across **256 nodes/pod**
Inter-pod (Cluster)	N/A	**Ethernet** connecting N pods

KEY IDEA

Classic workstations hang multiple GPUs off a tree-structured PCIe bus where peers contend for shared links, and route inter-machine traffic through the CPU and OS network stack. Blackwell datacenter systems eliminate both bottlenecks using **NVSwitch** inside the fast domain and **RDMA** across nodes.

WORKED EXAMPLE

To appreciate the magnitude of these bandwidth differences, calculate the time required to transmit a $100\ \text{MB}$ ($0.1\ \text{GB}$) tensor across each interconnect:

- **NVLink 5.0 (900 GB/s per direction)**: $t = \frac{0.1\ \text{GB}}{900\ \text{GB/s}} \approx 0.111\ \text{ms}$.
- **PCIe v7.0 x16 (242 GB/s)**: $t = \frac{0.1\ \text{GB}}{242\ \text{GB/s}} \approx 0.413\ \text{ms}$ ($3.7\times$ slower than NVLink 5.0, even before PCIe tree contention).
- **InfiniBand (~0.05 TB/s = 50 GB/s)**: $t = \frac{0.1\ \text{GB}}{50\ \text{GB/s}} = 2.0\ \text{ms}$ ($18\times$ slower than NVLink 5.0).
- **Home Ethernet (~200 MB/s = 0.2 GB/s)**: $t = \frac{0.1\ \text{GB}}{0.2\ \text{GB/s}} = 500\ \text{ms}$ ($4{,}500\times$ slower than NVLink 5.0).

Intra-Domain Shift: From 8-GPU Nodes to 72-GPU NVL72 Racks

Look closely at how the high-speed NVLink domain has expanded between generations:

DEFINITION

- **Classic 8-GPU Node (Ampere/Hopper era and B200 server)**: Four GPUs (`GPU 0` – `GPU 3`) sit along the top edge and four GPUs (`GPU 4` – `GPU 7`) sit along the bottom edge, all wired directly into a central **NVSwitch Fabric** that is non-blocking: each B200 gets its full 900 GB/s per direction. Any GPU can talk to any of the other 7 GPUs at full speed simultaneously. However, this high-speed NVLink domain was strictly capped at **8 GPUs**—scaling to a 9th GPU forced traffic across the node boundary onto slower InfiniBand (~0.05 TB/s).

DEFINITION

- **Blackwell GB200 / GB300 NVL72 Rack**: Picture a full rack containing **18 compute trays** stacked vertically (`Tray 1` through `Tray 18`), each holding **4 Blackwell GPUs** (two Grace-Blackwell superchips). All 18 trays plug horizontally into a vertical **Rack-Scale NVSwitch** spine running down the side of the rack. Because $4\ \text{GPUs/tray} \times 18\ \text{trays/rack} = 72\ \text{GPUs}$, the entire rack forms **one single 72-GPU NVLink 5.0 domain**. This $9\times$ expansion of the NVSwitch domain dramatically extends the reach of communication-heavy strategies—specifically Tensor Parallelism (Section 7) and Expert Parallelism (Section 9)—without hitting the InfiniBand cliff at 8 GPUs.

Scaling Out: Pods, Clusters, and Bypassing the CPU via RDMA

Beyond a single 8-GPU node or 72-GPU NVL72 rack, datacenter clusters scale hierarchically in two outer tiers:

DEFINITION

- **Intra-pod (Multi-Node)**: Up to **256 nodes** (`Node 0`, `Node 1`, …, `Node 255`, each containing its own intra-node NVLink domain) connect downward into a pod-level **InfiniBand Fabric** (~0.05 TB/s). The physical hardware path out of a GPU is: `GPU` → `PCIe` → `HCA (Host Channel Adapter, an InfiniBand NIC)` → `InfiniBand Cable`.

DEFINITION

- **Inter-pod (Cluster)**: Multiple pods (`Pod 1` through `Pod N`, each 256 nodes) link horizontally through a datacenter **Ethernet** cluster switch (routed via `PCIe` → `CPU`). Large-scale training must minimize cross-pod and cross-node bytes relative to intra-domain NVLink traffic.

Why does datacenter inter-node networking rely on InfiniBand or RoCE rather than standard TCP/IP over Ethernet? Contrast the two data paths from `GPU 0 HBM` on one node to `GPU 1 HBM` on another node:

1. **Standard Ethernet Path (CPU in the loop)**: `GPU 0 HBM` → `CPU 0 Kernel Buffer` (copies data into host memory and builds TCP packets) → `NIC 0 ring buffer` → wire → `NIC 1 ring buffer` → `CPU 1 Kernel` (unpacks TCP packets in OS kernel space) → `GPU 1 HBM`. Every transfer suffers multiple memory copies, OS kernel transitions, and CPU interrupts.
2. **GPUDirect RDMA Path (InfiniBand or RoCE — CPU bypassed)**: `GPU 0 HBM` → `HCA / RDMA NIC` (direct PCIe DMA straight out of GPU memory) → `IB / RoCE Fabric` → `HCA / RDMA NIC` (direct PCIe DMA straight into remote GPU memory) → `GPU 1 HBM`. Both `CPU 0` and `CPU 1` sit completely idle during the transfer.

DEFINITION

RDMA (Remote Direct Memory Access) allows one GPU to read or write another GPU's HBM across the network with zero CPU copies and zero OS kernel transitions. **InfiniBand** supports RDMA natively; standard Ethernet does not. **RoCE (RDMA over Converged Ethernet)** adds CPU-bypassing RDMA on top of Ethernet hardware. You might assume RoCE and InfiniBand are identical in performance because both support RDMA; in practice, RoCE is cheaper and widely deployed (for example, at Meta), but typically weaker in congestion control and tail latency than native InfiniBand.

NCCL: Translating Collectives to Blackwell Kernels

DEFINITION

With such a complex physical stack—NVLink, NVSwitch, PCIe trees, HCAs, InfiniBand, and RoCE—how does a single PyTorch call like `dist.all_reduce` know how to route packets efficiently? That job belongs to **NCCL (NVIDIA Collective Communication Library)**, which sits as the middle layer in a three-tier software-to-hardware stack:

1. **Top Layer — PyTorch (`torch.distributed`)**: exposes high-level collective APIs (`all_reduce`, `reduce_scatter`, `all_to_all`).
2. **Middle Layer — NCCL**: performs three jobs at runtime:
   - *Detects physical topology* automatically (discovering intra-node NVLink/NVSwitch links, PCIe paths, and inter-node InfiniBand/RoCE NICs).
   - *Selects the optimal algorithm* (Ring, Tree, or Switch-based schedule) depending on message size and world size.
   - *Launches CUDA kernels on GPU SMs* (enqueued on a CUDA stream) that actively stream packets across the links.
3. **Bottom Layer — Blackwell Hardware Fabric**: NVLink 5.0, NVSwitch, and GPUDirect RDMA move the bits.

KEY IDEA

Because NCCL inspects the physical topology at runtime, you do not rewrite your collective code when scaling from 4 GPUs on one node to a 72-GPU NVL72 rack or a 256-node pod—the same PyTorch code adapts automatically.

Next, let us see how to program these collectives directly in `torch.distributed`, benchmark them without falling into asynchronous-timing traps, and derive the exact bus-bandwidth equations used to verify whether our code is saturating the hardware fabric.

> **✎ Check Your Understanding**
>
> 1. **Fill-in-the-blank:** In a Blackwell GB200/GB300 NVL72 rack, _____ compute trays of 4 GPUs each connect through a rack-scale NVSwitch spine, expanding the single NVLink 5.0 domain from the classic 8 GPUs up to _____ GPUs.
>   **Answer:** 18 compute trays; 72 GPUs ($18 \times 4 = 72$).
> 2. **Yes/No with justification:** When two GPUs on different nodes transfer activations using GPUDirect RDMA over InfiniBand, do the host CPUs copy the tensor through OS TCP kernel buffers?
>   **Answer:** No. GPUDirect RDMA enables the HCA/RDMA NIC to perform direct PCIe DMA between each GPU's HBM and the network fabric, completely bypassing CPU memory copies and OS kernel transitions.

Section 5: Programming, Benchmarking, and Modeling Collective Bandwidth in `torch.distributed`

Understanding collectives conceptually and knowing the hardware fabric are not enough until you can write working `torch.distributed` code, measure its true throughput, and compare that measurement against the physical link speed.

Initializing a Process Group and Running Collectives

PyTorch's `torch.distributed` module exposes collectives over pluggable backends: `nccl` for CUDA GPUs and `gloo` as a CPU fallback. Here is the minimal setup and an in-place All-Reduce across 4 GPUs:

```python
import os, torch, torch.distributed as dist

def setup(rank: int, world_size: int):
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "15623"
    backend = "nccl" if torch.cuda.is_available() else "gloo"
    dist.init_process_group(backend, rank=rank, world_size=world_size)

# Example: in-place All-Reduce across 4 CUDA GPUs
data = torch.tensor([0., 1, 2, 3], device=f"cuda:{rank}") + rank
dist.all_reduce(tensor=data, op=dist.ReduceOp.SUM, async_op=False)
# Every rank now holds tensor([6., 10., 14., 18.])
```

Two practical details matter here:

COMMON MISCONCEPTION

- You might think `MASTER_ADDR` and `MASTER_PORT` act as a central server through which every tensor passes during `all_reduce`. That is false—they are used **only to bootstrap initial rendezvous** so the W processes can discover each other and establish NCCL communicators; all actual tensor data flows directly through NCCL over NVLink or RDMA.

- Always call `dist.destroy_process_group()` before process exit to tear down and release NCCL communicators cleanly.

WORKED EXAMPLE

We can also verify our Section 3 identity (All-Reduce $=$ Reduce-Scatter $+$ All-Gather) directly in PyTorch:

```
# Input on each rank r: [r, r+1, r+2, r+3]
inp = torch.arange(world_size, dtype=torch.float32,
                   device=f"cuda:{rank}") + rank
shard = torch.empty(1, device=f"cuda:{rank}")

# Step 1: Reduce-Scatter (each rank keeps 1/world_size of the sum)
dist.reduce_scatter_tensor(output=shard, input=inp, op=dist.ReduceOp.SUM)

# Step 2: All-Gather (every rank collects all reduced shards)
full = torch.empty(world_size, device=f"cuda:{rank}")
dist.all_gather_into_tensor(output_tensor=full, input_tensor=shard)
```

Notice we use `reduce_scatter_tensor` and `all_gather_into_tensor` (operating on single contiguous tensors) rather than list-based variants, which avoids extra memory copies and packing overhead. Tracing all four ranks confirms exact equivalence to `all_reduce`:

Rank	`inp`	`shard` (after Reduce-Scatter)	`full` (after All-Gather)
Rank 0	`[0, 1, 2, 3]`	`[6.]`	`[6., 10., 14., 18.]`
Rank 1	`[1, 2, 3, 4]`	`[10.]`	`[6., 10., 14., 18.]`
Rank 2	`[2, 3, 4, 5]`	`[14.]`	`[6., 10., 14., 18.]`
Rank 3	`[3, 4, 5, 6]`	`[18.]`	`[6., 10., 14., 18.]`

Benchmarking Collectives Correctly on CUDA

Measuring the execution time of a collective on CUDA is notoriously easy to get wrong. Recall from Section 2 that GPU launches are asynchronous and ranks execute independently. Here is the correct benchmark template:

```
data = torch.randn(100 * 1024**2, device=f"cuda:{rank}")

# 1. Warmup: initialize NCCL rings/buffers and JIT/CUDA state
dist.all_reduce(tensor=data, op=dist.ReduceOp.SUM, async_op=False)
torch.cuda.synchronize()    # Wait for local GPU kernels to finish
dist.barrier()              # Align all ranks before starting clock

# 2. Timed collective
t0 = time.time()
dist.all_reduce(tensor=data, op=dist.ReduceOp.SUM, async_op=False)
torch.cuda.synchronize()    # Ensure collective actually finished on GPU!
dist.barrier()              # Wait for the slowest rank
duration = time.time() - t0
```

Every single line outside `t0 = time.time()` and `all_reduce` prevents a specific measurement bug:

- **Omit warmup**: your timer measures one-time NCCL ring/buffer initialization and CUDA JIT compilation overhead instead of steady-state bandwidth.

> **COMMON MISCONCEPTION**
>
> - **Omit `torch.cuda.synchronize()`** : even when `async_op=False` is passed to `dist.all_reduce` , PyTorch with NCCL only ensures the collective is enqueued synchronously on the GPU stream so subsequent GPU operations wait for it—it does **not** block the Host CPU until the GPU finishes! Omitting `torch.cuda.synchronize()` measures only CPU enqueue time, yielding absurdly high, fake bandwidth numbers (such as tens of terabytes per second over InfiniBand).

- **Omit `dist.barrier()`** : fast ranks exit and stop their clocks before straggler ranks complete the collective, under-reporting true end-to-end step time.

Reasoning About Collective Bus Bandwidth

Once you have a valid measurement of tensor size S (in bytes), world size W, and elapsed time t (in seconds), how do you tell whether NCCL is saturating your hardware interconnect (say, NVLink 5.0's 900 GB/s per direction)?

COMMON MISCONCEPTION

You might naively divide S/t (known as *algorithmic bandwidth*). That fails as a hardware utilization metric because a collective moves a different number of bytes per link than S. To understand why, visualize a **Ring Schedule** for $W = 4$ GPUs (`R0` at top, `R1` at right, `R2` at bottom, `R3` at left) connected by clockwise directed arcs `R0` $\to$ `R1` $\to$ `R2` $\to$ `R3` $\to$ `R0` :

1. **Reduce-Scatter phase**: the S-byte tensor is partitioned into W shards of size S/W bytes each. In each step, every rank simultaneously sends one S/W-byte shard clockwise to its neighbor while receiving a shard from its counterclockwise neighbor and summing it into its local buffer. After exactly $W - 1$ steps, every rank holds the complete sum for its assigned S/W-byte shard. Each rank has transmitted $(W-1) \cdot \frac{S}{W} = \frac{W-1}{W} S$ bytes over its outgoing link.
2. **All-Gather phase**: in each of the next $W - 1$ steps, every rank forwards a completed S/W-byte shard clockwise around the ring. After $W - 1$ steps, every rank holds all W completed shards. Each rank has transmitted another $\frac{W-1}{W} S$ bytes.

DEFINITION

Dividing the actual bytes transmitted per rank over its link by the measured time t yields the **collective bus bandwidth**:

$$\mathrm{BW}_{\text{Reduce-Scatter}} = \frac{W-1}{W} \cdot \frac{S}{t} \approx \frac{S}{t}$$

$$\mathrm{BW}_{\text{All-Reduce}} = \frac{2(W-1)}{W} \cdot \frac{S}{t} \approx \frac{2S}{t}$$

Let us decompose the All-Reduce bus bandwidth formula term by term:

- S is the size in bytes of the tensor being reduced on each rank.
- t is the measured wall-clock duration of the collective (properly synchronized with `torch.cuda.synchronize()` and `dist.barrier()`).
- S/t is the raw algorithmic throughput (tensor bytes completed per second).
- $(W-1)/W$ is the per-stage ring transfer factor: because a rank already owns $1/W$ of the tensor, it sends $(W-1)$ shards of size S/W during Reduce-Scatter (and again during All-Gather). As W grows large, $(W-1)/W \to 1$.
- The leading factor of 2 accounts for the two consecutive phases (Reduce-Scatter + All-Gather).
- As a single conceptual unit, $\mathrm{BW}_{\text{All-Reduce}}$ measures the **effective physical link bandwidth achieved per GPU**, normalizing out the world size W so you can compare the result directly against a hardware spec such as NVLink 5.0 or InfiniBand.

KEY IDEA

In the large-message regime (where bandwidth dominates fixed startup latency), bus bandwidth is **independent of world size W and independent of topology (ring or tree)**: All-Reduce moves $2\times$ the data of Reduce-Scatter in $2\times$ the time, so both achieve the exact same bus bandwidth.

WORKED EXAMPLE

Let us work through a concrete numerical example using the tensor from our benchmark code above: `100 * 1024**2` float32 elements on $W = 8$ GPUs.

1. Compute tensor size S in bytes:

$$S = 100 \times 1{,}048{,}576 \text{ elements} \times 4 \text{ bytes/element} = 419{,}430{,}400 \text{ bytes} \approx 0.4194 \text{ G}$$

2. Suppose our synchronized benchmark measures $t = 1.0 \text{ ms} = 0.001 \text{ s}$ for `dist.all_reduce`.
3. A naive algorithmic calculation would report $S/t = 0.4194 \text{ GB}/0.001 \text{ s} = 419.4 \text{ GB/s}$.
4. The true **bus bandwidth** is:

$$\text{BW}_{\text{All-Reduce}} = \frac{2(8-1)}{8} \cdot \frac{0.4194 \text{ GB}}{0.001 \text{ s}} = 1.75 \times 419.4 \text{ GB/s} = 734.0 \text{ GB/s}$$

5. If we instead ran `reduce_scatter_tensor` on the same tensor and hardware at the same 734.0 GB/s bus bandwidth, it would move half the bytes ($\frac{7}{8} \times 0.4194 = 0.3670$ GB) and finish in half the time: $t_{\text{RS}} = 0.3670/734.0 = 0.5$ ms.

We now possess the full foundations: the memory hierarchy, asynchronous CUDA execution, collective primitives, Blackwell interconnect topology, `torch.distributed` APIs, and bus-bandwidth math. Next, we use these primitives to build the actual distributed training strategies along every dimension of a neural network.

✎ Check Your Understanding

1. **What would happen if…** you ran the collective benchmark script with `async_op=False`, kept `dist.barrier()`, but deleted `torch.cuda.synchronize()`?
   Answer: Because NCCL launches asynchronously onto the CUDA stream even when `async_op=False`, the host CPU would stop the timer as soon as the kernel was enqueued rather than finished, measuring tiny CPU launch overhead and producing absurdly inflated fake bandwidth numbers.
2. **Quick computation:** An All-Reduce of a $S = 200$ MB tensor across $W = 4$ GPUs completes in $t = 2$ ms. What is the achieved bus bandwidth in GB/s?
   Answer: $\text{BW}_{\text{All-Reduce}} = \frac{2(4-1)}{4} \cdot \frac{0.2 \text{ GB}}{0.002 \text{ s}} = 1.5 \times 100 \text{ GB/s} = 150 \text{ GB/s}$.

Section 6: Data Parallelism (DDP) and Sharded Data Parallelism (FSDP / ZeRO)

To make every distributed training strategy concrete, we implement each one from raw `torch.distributed` primitives on a single representative model: a deep multi-layer perceptron (MLP) with `batch_size = 128`, `num_dim = 1024`, and `num_layers = 4`. Focusing on a deep MLP is directly representative of large language models because MLPs are the primary compute bottleneck in Transformers (we will bring in attention layers explicitly in Section 9).

Before implementing the first strategy, map out all six **axes of parallelism** so you see how each one shards a different tensor dimension and maps to a different interconnect tier:

Strategy	Dimension Sharded	Primary Communication	Interconnect Fit
Data (DDP)	Batch ($B \to B/W$)	All-Reduce gradients	Moderate (IB / NVLink)
Sharded Data (FSDP / ZeRO)	Batch + model state ($1/W$)	All-Gather params + Reduce-Scatter grads	Moderate (overlapped)
Tensor (TP)	Hidden width ($D \to D/W$)	All-Gather / All-Reduce activations per layer	**Fast only** (NVLink / NVL72)
Expert (EP)	MoE experts (width)	All-to-All token dispatch & combine	**Fast** (NVLink / NVL72)
Pipeline (PP)	Depth ($L \to L/W$ layers)	Point-to-point `send` / `recv`	Tolerates slow IB / RoCE
Sequence / Context	Sequence length ($T \to T/W$)	Ring / All-to-All attention states	NVLink / fast IB

Data Parallelism (DDP): Split the Batch

> **DEFINITION**
> The simplest way to harness W GPUs for more FLOPs is **Distributed Data Parallelism (DDP)**: split the global batch B evenly along the batch dimension so each GPU processes B/W examples independently, while every GPU holds a full copy of the model weights and AdamW optimizer state.

Picture the DDP architecture for $W = 4$ GPUs and $B = 128$:

- On the left, the **Global Batch** B splits into four equal micro-slices: B_0 (32 examples) $\to$ GPU 0, B_1 (32) $\to$ GPU 1, B_2 (32) $\to$ GPU 2, and B_3 (32) $\to$ GPU 3.
- In the center, each GPU r holds a complete copy of the weights W and AdamW optimizer state, and executes $\text{Fwd}(B_r) \to \text{Loss } L_r \to \text{Bwd} \to g_r$ with **zero cross-GPU communication** during the entire forward and backward pass.
- On the right, all four gradient sets g_0, g_1, g_2, g_3 flow into a single **All-Reduce (`AVG grads`)** box, which feeds into an **Identical AdamW step** executed on every GPU.

Here is the complete bare-bones PyTorch implementation:

```
local_bs = batch_size // world_size
x_local = data[rank * local_bs : (rank + 1) * local_bs].to(f"cuda:{rank}")

# Every rank holds all parameters and its own AdamW optimizer state
params = [get_init_params(num_dim, num_dim, rank)
          for _ in range(num_layers)]
optimizer = torch.optim.AdamW(params, lr=1e-3)

for step in range(num_steps):
    x = x_local
    for param in params:
        x = F.gelu(x @ param)
    loss = x.square().mean()
    loss.backward()

    # ONLY difference from single-GPU training: average gradients
    for param in params:
        dist.all_reduce(param.grad, op=dist.ReduceOp.AVG)

    optimizer.step()
    optimizer.zero_grad()
```

Why is this mathematically identical to single-GPU training on the full global batch B? Because the loss is a mean over examples, the global loss $L(B)$ equals the average of the per-rank losses:

$$L(B) = \frac{1}{W} \sum_{r=0}^{W-1} L(B_r)$$

By linearity of differentiation, the gradient of the global loss with respect to any parameter θ is the average of the local gradients:

$$g_{\text{global}} = \nabla_\theta L(B) = \frac{1}{W} \sum_{r=0}^{W-1} \nabla_\theta L(B_r) = \frac{1}{W} \sum_{r=0}^{W-1} g_r$$

WORKED EXAMPLE

Let us verify this with a concrete numerical example. Suppose a single scalar weight is $\theta = 2.0$, the loss on example i is $\ell_i(\theta) = (\theta - y_i)^2$, and we have a global batch of 4 targets $y = [1, 3, 5, 7]$ split across $W = 2$ GPUs (`R0` gets $[1, 3]$, `R1` gets $[5, 7]$):

- **Single GPU on full batch**: $L = \frac{1}{4}[(2-1)^2 + (2-3)^2 + (2-5)^2 + (2-7)^2] = \frac{1}{4}(1 + 1 + 9 + 25) = 9.0$. Gradient: $g_{\text{global}} = \frac{1}{4}[2(1) + 2(-1) + 2(-3) + 2(-5)] = \frac{-16}{4} = -4.0$.
- **DDP across 2 GPUs**:
  - `R0` computes $L_0 = \frac{1}{2}[(2-1)^2 + (2-3)^2] = 1.0$, and $g_0 = \frac{1}{2}[2(1) + 2(-1)] = 0.0$.

- `R1` computes $L_1 = \frac{1}{2}[(2-5)^2 + (2-7)^2] = 17.0$, and $g_1 = \frac{1}{2}[2(-3) + 2(-5)] = -8.0$.
- Notice that **local losses differ** ($1.0 \neq 17.0$) because each rank sees different data! Students often panic when logging per-rank losses and seeing different numbers; that is completely expected.
- `dist.all_reduce(param.grad, op=dist.ReduceOp.AVG)` computes $\frac{g_0+g_1}{2} = \frac{0.0+(-8.0)}{2} = -4.0$ on both ranks—matching g_{global} exactly.
- Because both ranks started with identical parameters (via the same random seed or a rank-0 Broadcast from Section 3) and apply the exact same deterministic `optimizer.step()` to the exact same averaged gradient -4.0, **parameters stay identical on every rank—no parameter broadcast is ever needed after the optimizer step.**

Despite its simplicity, DDP immediately hits a **memory wall**. Look at every component of the HBM footprint bar from Section 1 under DDP across W GPUs:

Memory Component	Single GPU	DDP (W GPUs)	Bottleneck?
Activations	Full batch B	B/W (sharded)	Reduced by W
Parameters	Full model	**Full model** (replicated)	**Redundant** $W\times$
Gradients	Full model	**Full model** (replicated)	**Redundant** $W\times$
AdamW Optimizer States	Full model	**Full model** (replicated)	**Redundant** $W\times$

KEY IDEA

DDP speeds up training by adding FLOPs and shrinks activation memory by W, but it **cannot** train a model whose parameters, gradients, and AdamW optimizer states exceed a single GPU's HBM—and it wastes compute duplicating the exact same AdamW update W times.

Breaking the Memory Wall: From DDP to FSDP / ZeRO

How do we eliminate the $W\times$ redundant storage of model state without inventing a brand-new communication pattern? Recall the identity we proved in Section 3:

$$\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$$

DEFINITION

Compare the DDP flow and the **FSDP (Fully Sharded Data Parallel) / ZeRO (Zero Redundancy Optimizer)** flow side by side:

- **DDP (Replicated State — full model state per GPU)**: `Full W, g, Opt stored on every GPU` $\longrightarrow$ `Forward + Backward` $\longrightarrow$ `All-Reduce(g) [= RS + AG]` $\longrightarrow$ `Full AdamW Step (duplicated W×)`.
- **FSDP / ZeRO (Sharded State — $1/W$ model state per GPU)**:
  1. **Persistent state**: each GPU permanently stores only its own $1/W$ **shard** of the parameters W and AdamW optimizer states.
  2. **All-Gather layer weights W_ℓ**: just before computing layer ℓ, GPUs run All-Gather to temporarily reconstruct the full weights W_ℓ of that single layer.
  3. **Layer Compute (and free W_ℓ after!)**: each GPU runs forward (or backward) on its local batch slice B/W using W_ℓ, and immediately discards the gathered remote shards of W_ℓ once layer ℓ finishes.
  4. **Reduce-Scatter gradients g_ℓ**: after backward for layer ℓ, GPUs run Reduce-Scatter to sum gradients across ranks while leaving each rank with **only its own $1/W$ gradient shard**.
  5. **Sharded AdamW**: each GPU updates **only its own $1/W$ shard** of weights and optimizer states—zero duplicated optimizer compute!

COMMON MISCONCEPTION

Now check the communication cost carefully, because students frequently assume FSDP must communicate far more data than DDP. Suppose the model parameters occupy S bytes (with matching S-byte gradients):

- **DDP** runs one All-Reduce on gradients per step:

$$V_{\text{DDP}} = \frac{2(W-1)}{W} S$$

- **Sharding optimizer states and gradients (ZeRO-1 / ZeRO-2)**: each GPU keeps full weights, runs **Reduce-Scatter** on gradients ($\frac{W-1}{W} S$), runs its sharded AdamW step on its $1/W$ slice, and runs **All-Gather** on the updated weights ($\frac{W-1}{W} S$). Total volume:

$$V_{\text{ZeRO-2}} = \frac{W-1}{W} S + \frac{W-1}{W} S = \frac{2(W-1)}{W} S = V_{\text{DDP}}$$

  Sharding optimizer states and gradients costs **zero extra communication** compared to DDP!
- **Sharding parameters too (ZeRO-3 / FSDP)**: because we discard the gathered weights W_ℓ after forward to save HBM, we must run one additional **All-Gather** before backward to rematerialize W_ℓ. Total volume:

$$V_{\text{FSDP}} = \underbrace{\frac{W-1}{W} S}_{\text{Fwd AG}} + \underbrace{\frac{W-1}{W} S}_{\text{Bwd AG}} + \underbrace{\frac{W-1}{W} S}_{\text{Bwd RS}} = \frac{3(W-1)}{W} S = 1.5 \times V_{\text{DDP}}$$

WORKED EXAMPLE

Let us put concrete numbers on this for our 10 B-parameter model ($S = 20$ GB of BF16 weights, 160 GB total static Model State) across $W = 8$ GPUs:

- **DDP**: static Model State per GPU stays 160 GB (does not fit on an 80 GB GPU); communication per step $= \frac{2(7)}{8} \times 20$ GB $= 35$ GB per rank.
- **FSDP (ZeRO-3)**: persistent Model State per GPU drops by $8\times$ to $160/8 = 20$ GB (plus temporary HBM for just one gathered layer at a time); communication per step $= \frac{3(7)}{8} \times 20$ GB $= 52.5$ GB per rank—only $1.5\times$ DDP's communication volume in exchange for an $8\times$ reduction in persistent model-state HBM! (And in Section 9 we will overlap those per-layer All-Gathers and Reduce-Scatters with compute.)

KEY IDEA

Even with FSDP, splitting along the batch dimension runs into a fundamental wall when the global batch size B is small (for example, during low-latency inference decoding where B may be 1 to 16, or when a model is so wide that even a single layer's activations and temporary gathered weights are too large or too slow to compute on one GPU). When splitting the batch is no longer enough, we must shard the computation *inside* each layer—starting along the model's width with **Tensor Parallelism**.

✎ Check Your Understanding

1. **Yes/No with justification:** In DDP, after `optimizer.step()` finishes on all ranks, must rank 0 broadcast the updated weights to the other ranks to keep models in sync?
   Answer: No. Because all ranks start from identical weights and receive identical averaged gradients from `all_reduce(..., ReduceOp.AVG)`, running the deterministic AdamW update locally produces identical updated weights on every rank automatically.
2. **Quick computation:** Moving from DDP to full parameter-sharded FSDP (ZeRO-3) increases per-step communication volume by what multiplicative factor, and what extra collective causes that increase?
   Answer: It increases communication volume by a factor of $1.5\times$ (from $\frac{2(W-1)}{W}S$ to $\frac{3(W-1)}{W}S$), caused by the extra All-Gather required during the backward pass to rematerialize layer weights that were freed after the forward pass.

Section 7: Tensor Parallelism (TP) — Sharding Model Width

DEFINITION

Data parallelism and FSDP both require every GPU to hold a complete layer's weight matrix while executing that layer on its own slice of the batch. That leaves two problems unsolved: first, if the batch size is small (such as during interactive decoding/serving, where latency rather than just throughput dominates), you cannot split the batch across many GPUs; second, if a single layer's hidden dimension D is huge, even one layer's matrix multiplication and activations overwhelm a single GPU's compute or HBM. **Tensor Parallelism (TP)** solves this by sharding each layer's weight matrix along its **hidden width** ($D \to D/W$).

Column-Sharded Matrix Multiplication and Activations

Recall our running MLP (`batch_size = 128`, `num_dim = 1024`, `num_layers = 4`) across $W = 4$ GPUs. Picture the linear algebra of a single column-sharded layer across the four GPUs:

- **Input X (replicated)**: every GPU holds the full input activation matrix X of shape $B \times D = 128 \times 1024$.
- **Weight W (sharded by columns)**: instead of storing the full 1024×1024 matrix on every GPU, we slice W vertically into $W = 4$ column blocks:

$$W = \begin{bmatrix} W_0 & W_1 & W_2 & W_3 \end{bmatrix}$$

  where GPU r stores only W_r of shape $D \times (D/W) = 1024 \times 256$.
- **Local MatMul + GeLU**: each GPU multiplies the replicated input X by its local weight shard W_r and applies the elementwise nonlinearity GeLU locally:

$$Y_r = \mathrm{GeLU}(XW_r) \quad \text{of shape } B \times (D/W) = 128 \times 256$$

  Why can we apply GeLU *before* communicating? Because in block matrix multiplication, $X[W_0, W_1, W_2, W_3] = [XW_0, XW_1, XW_2, XW_3]$, so every output column is already completely computed on its owning GPU; because GeLU acts elementwise on each entry independently, $\mathrm{GeLU}([XW_0, \ldots, XW_3]) = [\mathrm{GeLU}(XW_0), \ldots, \mathrm{GeLU}(XW_3)]$.
- **All-Gather + Concat**: before the next layer can multiply by a full-width matrix, all GPUs execute an **All-Gather** on their local shards Y_r and concatenate them along the feature dimension (`dim=1`) to reconstruct the full output $Y = [Y_0, Y_1, Y_2, Y_3]$ of shape $B \times D = 128 \times 1024$ on every rank.

Here is the bare-bones PyTorch implementation:

```
data = data.to(f"cuda:{rank}")          # All ranks hold full batch (128x1024)
local_dim = num_dim // world_size       # Shard hidden dim: 1024 // 4 = 256

# Each rank holds only 1/world_size of every layer's columns
params = [get_init_params(num_dim, local_dim, rank)
          for _ in range(num_layers)]

x = data
for layer in range(num_layers):
    # Local matmul + GeLU -> shape (batch_size, local_dim)
    x = F.gelu(x @ params[layer])

    # Gather activation shards from every rank
    shards = [torch.empty(batch_size, local_dim, device=f"cuda:{rank}")
              for _ in range(world_size)]
    dist.all_gather(tensor_list=shards, tensor=x, async_op=False)

    # Concatenate along feature dim -> shape (batch_size, num_dim)
    x = torch.cat(shards, dim=1)
```

Production Optimization: Column-Sharded Paired with Row-Sharded

In the bare-bones loop above, every single linear layer runs a blocking All-Gather. In production Transformer MLPs (which consist of two consecutive linear projections separated by a nonlinearity: $Z = \mathrm{GeLU}(XW)V$), we can cut communication in half by **pairing a column-sharded first layer W with a row-sharded second layer V**.

Slice $W = [W_0, W_1]$ by columns and $V = \begin{bmatrix} V_0 \\ V_1 \end{bmatrix}$ by rows across $W = 2$ GPUs. Look at the block matrix product:

$$Z = \mathrm{GeLU}(XW)V = \begin{bmatrix} \mathrm{GeLU}(XW_0) & \mathrm{GeLU}(XW_1) \end{bmatrix} \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} = \underbrace{\mathrm{GeLU}(XW_0)V_0}_{\text{computed on GPU 0}} + \underbrace{\mathrm{GeLU}(X}_{\text{computed on}}$$

KEY IDEA

Each GPU computes $Y_r = \mathrm{GeLU}(XW_r)$ locally, skips the All-Gather completely, immediately multiplies $Y_r V_r$ locally to produce a partial $B \times D$ output matrix, and then runs **one single All-Reduce (`SUM`)** across GPUs to add the partial outputs together! Two MLP matrix multiplications thus require only **one All-Reduce** instead of two All-Gathers.

WORKED EXAMPLE

Let us verify this algebra—and expose a classic misconception—with a concrete numerical example. Suppose $B = 1$, $D = 2$, $W = 2$ GPUs, input $X = [1, 2]$, first weight $W = \begin{bmatrix} 1 & 2 \\ 0 & -1 \end{bmatrix}$ split into columns $W_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ (GPU 0) and $W_1 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$ (GPU 1), and second weight $V = \begin{bmatrix} 3 & 1 \\ -2 & 4 \end{bmatrix}$ split into rows $V_0 = \begin{bmatrix} 3 & 1 \end{bmatrix}$ (GPU 0) and $V_1 = \begin{bmatrix} -2 & 4 \end{bmatrix}$ (GPU 1). For simplicity of hand arithmetic, use $\mathrm{ReLU}(u) = \max(0, u)$ as the elementwise nonlinearity:

1. **Full single-GPU computation**:
   - $XW = [1(1) + 2(0),\ 1(2) + 2(-1)] = [1, 0]$.
   - $Y = \mathrm{ReLU}([1, 0]) = [1, 0]$.
   - $Z = [1, 0] \begin{bmatrix} 3 & 1 \\ -2 & 4 \end{bmatrix} = [3, 1]$.
2. **Column-then-row TP across 2 GPUs**:
   - GPU 0: $XW_0 = [1] \implies Y_0 = \mathrm{ReLU}([1]) = [1] \implies Z_0 = [1][3, 1] = [3, 1]$.
   - GPU 1: $XW_1 = [0] \implies Y_1 = \mathrm{ReLU}([0]) = [0] \implies Z_1 = [0][-2, 4] = [0, 0]$.
   - One All-Reduce (`SUM`): $Z = Z_0 + Z_1 = [3, 1] + [0, 0] = [3, 1]$. Exact match!

COMMON MISCONCEPTION

3. **Why you cannot reverse the order (row-sharded first, then elementwise nonlinearity)**: You might think you could row-shard W first, apply GeLU or ReLU locally to the partial dot products, and sum afterward. That breaks immediately because nonlinearities do not distribute over addition: $\mathrm{ReLU}(a+b) \neq \mathrm{ReLU}(a) + \mathrm{ReLU}(b)$ (for example, $\mathrm{ReLU}(2+(-2)) = 0 \neq \mathrm{ReLU}(2) + \mathrm{ReLU}(-2) = 2$). That is why the first projection **must** be column-sharded and the second **must** be row-sharded.

Why NVLink Is Mandatory for Tensor Parallelism

KEY IDEA

Look closely at where communication sits in Tensor Parallelism: **activations must cross GPUs synchronously inside every single Transformer block, in both the forward and backward passes.** Neither the next layer nor the rest of the forward pass can proceed until the activation collective completes, placing communication latency and bandwidth directly on the critical path of matrix multiplication.

WORKED EXAMPLE

Compare how a $32\ \mathrm{MB}$ ($0.032\ \mathrm{GB}$) activation All-Reduce across $W = 8$ GPUs performs across the two Blackwell interconnect tiers from Section 4 (using $\text{volume} = \frac{2(8-1)}{8} \times 0.032\ \mathrm{GB} = 0.056\ \mathrm{GB}$ per rank):

- **Over NVLink 5.0 (900 GB/s per direction)**: $t \approx \frac{0.056\ \mathrm{GB}}{900\ \mathrm{GB/s}} \approx 0.062\ \mathrm{ms}$—fast enough to keep Tensor Cores fed across dozens of layers.
- **Over InfiniBand (~0.05 TB/s = 50 GB/s)**: $t \approx \frac{0.056\ \mathrm{GB}}{50\ \mathrm{GB/s}} = 1.12\ \mathrm{ms}$ per collective—$18\times$ slower! Across 80 Transformer layers (forward + backward), stalling for milliseconds on every single layer destroys GPU utilization.

This explains a major architectural shift on Blackwell hardware:

- On historical Ampere and Hopper systems, high-speed NVLink stopped at **8 GPUs per node**, so Tensor Parallelism stayed strictly within 8 GPUs.
- On **Blackwell GB200/GB300 NVL72**, unifying **72 GPUs** inside a single NVLink 5.0 domain removes the 8-GPU cliff, allowing much wider models and low-latency inference decoding to scale TP well beyond 8 GPUs.

Still, frontier training runs span thousands of GPUs across many nodes and racks connected by InfiniBand or RoCE. If Tensor Parallelism cannot cross slow inter-node links without stalling, how can we shard a model across nodes over slower interconnects? Instead of cutting every layer vertically along its width, we cut the model horizontally along its **depth** using **Pipeline Parallelism**.

✎ Check Your Understanding

1. **Yes/No with justification:** In a two-layer Transformer MLP (`Linear1 -> GeLU -> Linear2`), can you shard `Linear1` by rows, apply `GeLU` locally on each GPU, and All-Reduce afterward without losing mathematical equivalence?
   Answer: No. Row-sharding `Linear1` produces partial sums on each GPU, and because `GeLU` is nonlinear ($\mathrm{GeLU}(a+b) \neq \mathrm{GeLU}(a) + \mathrm{GeLU}(b)$), applying `GeLU` before summing gives the wrong result. `Linear1` must be column-sharded and `Linear2` row-sharded.
2. **Fill-in-the-blank:** Tensor Parallelism requires high-bandwidth _____ interconnects rather than InfiniBand because it runs synchronous activation collectives on the critical path inside every Transformer layer; Blackwell NVL72 expands this fast domain from 8 GPUs to _____ GPUs.
   Answer: NVLink (or NVLink 5.0 / NVSwitch); 72 GPUs.

Section 8: Pipeline Parallelism (PP) — Sharding Depth and Managing Pipeline Bubbles

DEFINITION

When a model is too large to fit inside one fast NVLink domain, or when we want to shard model state across nodes linked by slower InfiniBand (~0.05 TB/s) or RoCE without invoking per-layer collectives, we shard the network along its **depth** ($L \to L/P$ layers per stage). This strategy is **Pipeline Parallelism (PP)**.

Sharding Depth with Point-to-Point Stage Boundaries

Picture our 4-layer MLP partitioned across $P = 2$ pipeline stages (`Rank 0` and `Rank 1`):

- **Rank 0 (Stage 0)** holds only the parameters for **Layer 0** (`MatMul + GeLU`) and **Layer 1** (`MatMul + GeLU`).
- **Stage boundary**: a single point-to-point `dist.send` / `dist.recv` transfers the activations of a micro-batch from Rank 0 to Rank 1.
- **Rank 1 (Stage 1)** holds only the parameters for **Layer 2** (`MatMul + GeLU`) and **Layer 3** (`MatMul + GeLU`).

KEY IDEA

Compare this communication pattern against Tensor Parallelism (Section 7): instead of running an all-to-all or ring collective across all GPUs inside *every* layer, Pipeline Parallelism communicates **only at stage boundaries** using lightweight point-to-point `send` / `recv` calls between adjacent ranks. Because total communication volume is far smaller and occurs only once every L/P layers, Pipeline Parallelism tolerates slower inter-node InfiniBand or RoCE links gracefully.

During the backward pass, data flow simply reverses: each stage receives output gradients from `rank + 1` via `dist.recv`, runs backward through its local layers to accumulate local parameter gradients, and sends input gradients to `rank - 1` via `dist.send`.

The Pipeline Bubble Problem and Micro-Batching

DEFINITION

Pipeline Parallelism introduces a new inefficiency that neither DDP nor TP has: **pipeline bubbles** (idle GPU time caused by sequential dependencies between stages).

WORKED EXAMPLE

Trace a **naive schedule** with $P = 2$ stages and a single batch ($M = 1$):

Stage	t1	t2	t3	t4
`Rank 0 (L0–1)`	Fwd	**idle**	**idle**	Bwd
`Rank 1 (L2–3)`	**idle**	Fwd	Bwd	**idle**

Look at the utilization: Rank 1 sits completely idle at `t1` waiting for Rank 0's activations; Rank 0 sits idle at `t2` and `t3` waiting for Rank 1 to finish forward and backward; Rank 1 sits idle at `t4` while Rank 0 finishes backward. Out of $2 \times 4 = 8$ GPU-time slots, **4 slots (50%) are wasted idle bubbles!** With $P = 8$ stages and $M = 1$, seven out of eight GPUs would sit idle at every instant—an unacceptable 87.5% waste.

To shrink these bubbles, we split the global batch B into M **smaller micro-batches** (each of size B/M) and stream them back-to-back through the pipeline, accumulating gradients across all M micro-batches before calling `optimizer.step()`. Here is the bare-bones PyTorch forward loop:

```
local_layers = num_layers // world_size
local_params = [get_init_params(num_dim, num_dim, rank)
                for _ in range(local_layers)]

# Split batch into micro-batches to shrink pipeline bubbles
micro_bs = batch_size // num_micro_batches
if rank == 0:
    micro_batches = data.to("cuda:0").chunk(num_micro_batches, dim=0)
else:
    micro_batches = [torch.empty(micro_bs, num_dim, device=f"cuda:{rank}")
                     for _ in range(num_micro_batches)]

for x in micro_batches:
    if rank - 1 >= 0:
        dist.recv(tensor=x, src=rank - 1)   # Receive from previous stage
    for param in local_params:
        x = F.gelu(x @ param)               # Run local layers
    if rank + 1 < world_size:
        dist.send(tensor=x, dst=rank + 1)   # Forward to next stage
```

WORKED EXAMPLE

Now trace the timeline for $P = 2$ stages with $M = 4$ **micro-batches** (`F0` – `F3` for forward, `B3` – `B0` for backward):

Stage	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10
`Rank 0 (L0–1)`	F0	F1	F2	F3	**idle**	**idle**	B3	B2	B1	B0
`Rank 1 (L2–3)`	**idle**	F0	F1	F2	F3	B3	B2	B1	B0	**idle**

Watch what happened in the steady state (`t2` – `t4` and `t7` – `t9`): both stages stay busy simultaneously! Count the slots: out of $2 \times 10 = 20$ GPU-time slots, only 4 slots are idle (`Rank 0` at `t5`, `t6` and `Rank 1` at `t1`, `t10`), cutting the bubble fraction from 50% down to $4/20 = 20\%$.

DEFINITION

Let us derive the general **pipeline bubble fraction** formula for P stages and M micro-batches (assuming equal forward and backward step times, say 1 time unit each):

- Every stage must execute M forward micro-batches and M backward micro-batches, requiring $2M$ active time units of useful work per stage.
- To fill the pipeline at the start of forward, the last stage waits $(P-1)$ forward steps before receiving `F0`. Symmetrically, to drain the pipeline at the end of backward, the first stage waits $(P-1)$ backward steps after the last stage begins `B_{M-1}`.
- Total wall-clock time per training step is therefore $T_{\text{total}} = 2M + 2(P-1) = 2(M+P-1)$ time units, of which every stage spends $2(P-1)$ time units idle.
- Dividing idle time by total time yields the bubble fraction:

$$f_{\text{bubble}} = \frac{2(P-1)}{2(M+P-1)} = \frac{P-1}{M+P-1}$$

Break this equation apart term by term:

- P is the number of pipeline stages (GPUs sharding depth).
- M is the number of micro-batches per optimizer step.
- $P-1$ in the numerator is the number of pipeline fill/drain idle steps per pass (zero when $P=1$, because there is no pipeline).
- $M+P-1$ in the denominator is the total number of time steps per pass (useful micro-batch steps M plus pipeline fill/drain latency $P-1$).
- Conceptually, f_{bubble} gives the fraction of total GPU compute capacity lost to pipeline stalls; driving f_{bubble} close to zero requires choosing $M \gg P$.

WORKED EXAMPLE

Check the formula on concrete numbers:

- Our $P=2, M=1$ table: $f_{\text{bubble}} = \frac{2-1}{1+2-1} = \frac{1}{2} = 50\%$.
- Our $P=2, M=4$ table: $f_{\text{bubble}} = \frac{2-1}{4+2-1} = \frac{1}{5} = 20\%$.
- A deeper pipeline with $P=4$ stages:
  - At $M=4$: $f_{\text{bubble}} = \frac{3}{4+3} = \frac{3}{7} \approx 42.9\%$ wasted!
  - At $M=16$: $f_{\text{bubble}} = \frac{3}{16+3} = \frac{3}{19} \approx 15.8\%$.
  - At $M=32$: $f_{\text{bubble}} = \frac{3}{32+3} = \frac{3}{35} \approx 8.6\%$.

Two common misconceptions deserve attention here:

COMMON MISCONCEPTION

1. You might think splitting a batch into M micro-batches changes the mathematical optimization step. For standard LLM layers (which use LayerNorm/RMSNorm rather than cross-example BatchNorm), accumulating gradients across all M micro-batches before `optimizer.step()` produces the **exact same** mathematical gradients as processing the full batch at once.

COMMON MISCONCEPTION

2. You might think you can push M arbitrarily high for free to drive bubbles to zero. That breaks because of **activation memory**: in the naive all-forward-then-all-backward schedule drawn above, `Rank 0` must keep the saved forward activations of **all M in-flight micro-batches** in HBM simultaneously until their corresponding backward passes return (`F0` is computed at `t1` and not freed until `B0` finishes at `t10`). Micro-batching therefore trades saved activation memory for higher GPU utilization.

KEY IDEA

Production pipeline schedules—such as **1F1B (One-Forward-One-Backward)**, **interleaved / virtual stages**, and **dual-direction schedules**—address this trade-off by starting backward passes for early micro-batches as soon as possible (capping in-flight activations at $O(P)$ instead of $O(M)$) and further shrinking bubble overhead.

Up to this point, every collective we wrote in DDP, FSDP, and TP used `async_op=False`, leaving Tensor Cores completely idle while communication executed. Next, we learn how to hide communication latency behind math using CUDA stream overlap, and then extend our parallelism toolkit beyond dense MLPs to the two defining features of modern frontier models: **Sparse Mixture-of-Experts (MoE)** and **long-context Attention**.

✎ Check Your Understanding

1. **Quick computation:** A model is split across $P = 8$ pipeline stages using $M = 24$ micro-batches. What is the pipeline bubble fraction f_{bubble}?
   Answer: $f_{\text{bubble}} = \frac{P-1}{M+P-1} = \frac{8-1}{24+8-1} = \frac{7}{31} \approx 22.6\%$.
2. **What would happen if…** you doubled the number of micro-batches M (holding global batch size B fixed) in the naive all-forward-then-all-backward pipeline schedule?
   Answer: The pipeline bubble fraction $\frac{P-1}{M+P-1}$ would shrink, increasing GPU utilization, while the pipeline would still have to keep activations for all M in-flight micro-batches resident on early stages (at fixed global batch B, total saved activations across the batch stay proportional to B, whereas scaling M by increasing B increases peak activation memory linearly with M).

Section 9: Hiding Communication via CUDA Stream Overlap, Sparse MoE, and Sequence Parallelism

> **KEY IDEA**
> Three critical ingredients separate a textbook dense-MLP prototype from a high-utilization frontier LLM training system: overlapping communication with computation on separate CUDA streams, routing tokens across sharded experts in Sparse Mixture-of-Experts (MoE) models, and parallelizing attention across long sequences.

Hiding Communication via CUDA Stream Overlap

Recall from Section 2 that a GPU can execute kernels from a **Compute Stream** and a **Comm Stream (NCCL)** concurrently, coordinating dependencies through CUDA Events. Look at what happens during a 4-layer backward pass (say, in FSDP) when we run collectives synchronously (`async_op=False`) versus asynchronously (`async_op=True`):

- **Synchronous (`async_op=False`) — Tensor Cores stall on every collective**:
  - `Compute` timeline: `Bwd L3` → **`stall`** → `Bwd L2` → **`stall`** → `Bwd L1` → **`stall`**
  - `NCCL` timeline: (idle) → `RS grad L3` → (idle) → `RS grad L2` → (idle) → `RS grad L1`
  - Every time a layer finishes backward, the Tensor Cores sit completely idle while NCCL runs Reduce-Scatter on that layer's gradients.
- **Overlapped (`async_op=True` on dedicated NCCL stream) — hide latency behind math**:
  - `Compute` timeline: `Bwd L3` → `Bwd L2` → `Bwd L1` → `Bwd L0` (continuous back-to-back math, zero stalls!)
  - `NCCL` timeline: (idle during `Bwd L3`) → `RS grad L3` (runs *while* `Bwd L2` computes!) → `RS grad L2` (runs *while* `Bwd L1` computes!) → `RS grad L1` (runs *while* `Bwd L0` computes!) → `RS grad L0`.

In PyTorch, you trigger overlap by launching collectives asynchronously:

```
work = dist.reduce_scatter_tensor(out_shard, full_grad, async_op=True)
```

and synchronizing via CUDA Events (or `work.wait()`) only when the result is actually required. FSDP applies this in both passes: during forward, it **prefetches** layer $\ell + 1$'s weight All-Gather on the NCCL stream while layer ℓ runs math on the Compute stream; during backward, it launches layer ℓ's gradient Reduce-Scatter while layer $\ell - 1$ computes backward.

COMMON MISCONCEPTION

Watch out for an important **CUDA hardware caveat** that we flagged in Section 2: you might assume that because Compute and NCCL occupy separate CUDA streams, overlapping them gives 100% free communication with zero impact on matrix-multiplication speed. That is false—**NCCL kernels execute on the GPU's SMs and read/write HBM**, so concurrent NCCL kernels compete with math kernels for SM slots and HBM bandwidth. Excessive SM contention can slow down GEMMs.

WORKED EXAMPLE

Let us quantify both the massive speedup and the SM-contention caveat with a concrete 4-layer example. Suppose each layer's backward compute takes 1.0 ms alone, and each layer's Reduce-Scatter takes 0.8 ms alone:

- **Synchronous (`async_op=False`)**: compute and communication serialize completely:

$$T_{\text{sync}} = 4 \times (1.0 + 0.8)\ \text{ms} = 7.2\ \text{ms}$$

- **Ideal zero-contention overlap**: `Bwd L3` (1.0 ms) + `Bwd L2, L1, L0` (3.0 ms, hiding `RS L3, L2, L1`) + trailing `RS L0` (0.8 ms):

$$T_{\text{ideal}} = 1.0 + 3.0 + 0.8 = 4.8\ \text{ms}$$

- **Realistic overlap with 10% SM contention during overlapped layers**: `Bwd L3` runs alone (1.0 ms); `Bwd L2, L1, L0` each slow down by 10% to 1.1 ms while sharing SMs with NCCL (3.3 ms, still easily hiding the 0.8 ms RS); trailing `RS L0` finishes alone (0.8 ms):

$$T_{\text{real}} = 1.0 + 3.3 + 0.8 = 5.1\ \text{ms}$$

Even accounting for SM contention, overlap cuts step time from 7.2 ms down to 5.1 ms—a 29% speedup—while showing why NCCL's SM footprint must be carefully bounded in production kernels.

Sparse MoE and Expert Parallelism (Current Practice)

DEFINITION

So far we have modeled dense MLPs where every token multiplies every weight matrix. Current frontier language models, however, are predominantly **sparse Mixture-of-Experts (MoE)** architectures: each MLP layer is replaced by many parallel "expert" MLPs, and a lightweight learned **Router** selects only the **top-k experts** (for example, $k = 2$ out of 64 experts) for each individual token.

COMMON MISCONCEPTION

Why do frontier models use sparse MoEs? Because they decouple model capacity from compute: multiplying the number of experts multiplies **total parameters** (increasing model knowledge and quality) without multiplying **FLOPs per token**. You might mistakenly think sparse MoEs save HBM memory because each token uses only a few experts; in reality, **all experts' weights, gradients, and AdamW states must still reside in HBM**, creating an intense memory-capacity problem that demands sharding experts across GPUs.

DEFINITION

Expert Parallelism (EP) places distinct experts on different GPUs and routes tokens across GPUs using **two All-to-All collectives** (Section 3) per MoE layer. Trace the full forward pipeline from left to right:

1. **Local Tokens**: each rank holds its local batch slice of tokens (`Rank 0 batch` , `Rank 1 batch` , …, `Rank W-1 batch`).
2. **Router**: scores each token locally and assigns it to its `top-k` target experts.
3. **All-to-All Dispatch**: sends every token across the interconnect to the GPU hosting its assigned expert.
4. **Local Experts**: each GPU executes its local expert MLPs on all incoming tokens (`GPU 0: Exp A` , `GPU 1: Exp B` , …, `GPU r: Exp ...`).
5. **All-to-All Combine**: sends each processed token output back to the originating GPU where its sequence lives.
6. **Out tokens**: each rank holds its updated local tokens and proceeds to the next layer.

WORKED EXAMPLE

Load balance is critical in Expert Parallelism. Consider a concrete example with $W = 4$ GPUs (1 expert per GPU), 16 total tokens per batch (4 tokens starting on each GPU), $k = 1$, and expert compute time of 0.1 ms per token:

- **Balanced routing**: each GPU receives exactly 4 tokens after All-to-All Dispatch. All four GPUs compute in parallel and finish in $4 \times 0.1 = 0.4$ ms.
- **Skewed routing**: suppose the router sends 10 tokens to `Exp A` on `GPU 0` and only 2 tokens each to `GPU 1` , `GPU 2` , and `GPU 3` . `GPU 1` – `GPU 3` finish in 0.2 ms and sit idle waiting at the All-to-All Combine barrier until overloaded `GPU 0` finishes in $10 \times 0.1 = 1.0$ ms—a $2.5\times$ slowdown caused solely by routing skew!

KEY IDEA

Because All-to-All Dispatch and Combine happen synchronously twice per MoE layer and load every GPU's NVLink in both directions at once, Expert Parallelism demands ultra-fast interconnects. Historically, EP was constrained by the 8-GPU node boundary; **Blackwell NVL72** lets Expert Parallelism scale cleanly across **72 GPUs** over NVLink 5.0 at 900 GB/s per direction per GPU.

Beyond MLPs: Attention and Sequence / Context Parallelism

MLPs process each token independently—token i never interacts with token j inside an MLP or LayerNorm. Attention layers break that independence: **attention mixes information across the entire sequence length T.**

When training on long documents (tens or hundreds of thousands of tokens), attention activations and Key-Value (KV) states grow with context length T and overflow single-GPU HBM even if the batch size is $B = 1$.

DEFINITION

Sequence / Context Parallelism solves this by sharding every sequence along the sequence length dimension ($T \to T/W$):

COMMON MISCONCEPTION

- **Local operations (zero communication)**: elementwise operations (`LayerNorm`, `dropout`) and MLPs act independently on each token, so each rank runs them locally on its T/W token slice. You might think sharding along T requires communicating inside MLPs; it does not, because MLPs mix features across D, not across T.

- **Attention communication**: because every query token must attend to keys and values across the full sequence T, ranks must communicate during attention using one of two patterns:
  1. **Ring-style K/V block exchange**: each rank holds its local Queries Q_r (T/W tokens) and passes Key/Value blocks (K_j, V_j) around a ring across GPUs, accumulating partial attention outputs as each block passes through.
  2. **All-to-All sequence-to-head transpose**: suppose a sequence has $T = 8{,}192$ tokens and $H = 32$ attention heads across $W = 4$ GPUs. Before attention, each GPU holds $(T/4, H) = (2{,}048 \text{ tokens}, 32 \text{ heads})$. An **All-to-All** transposes the sharding axis from sequence to heads so each GPU temporarily holds $(T, H/4) = (8{,}192 \text{ tokens}, 8 \text{ heads})$! Now each GPU has the **complete sequence** for its 8 assigned heads and runs standard single-GPU FlashAttention locally with zero communication, followed by a second All-to-All to return to $(2{,}048 \text{ tokens}, 32 \text{ heads})$ for the local MLP.

Beyond distributed systems tricks, **current hybrid and sparse attention architectures** attack this bottleneck directly at the model level: interleaving linear/recurrent layers or sparse/sliding-window attention with full attention shrinks KV memory footprint and reduces cross-GPU attention traffic

dramatically.

We have now built every individual parallelism axis—DDP, FSDP/ZeRO, Tensor Parallelism, Pipeline Parallelism, Expert Parallelism, and Sequence/Context Parallelism—along with CUDA stream overlap. Real frontier training runs do not pick just one of these; they compose all of them simultaneously onto a physical cluster. Our final section shows how every axis locks onto its matching hardware tier.

✎ Check Your Understanding

1. **Yes/No with justification:** When FSDP overlaps a gradient Reduce-Scatter on the NCCL stream with the next layer's backward GEMM on the Compute stream, is the GEMM guaranteed to run at the exact same speed as it would with no concurrent communication?
   Answer: No. NCCL communication runs as CUDA kernels on the GPU's SMs and reads/writes HBM, so concurrent NCCL kernels share SM and HBM resources with compute kernels and can slightly slow down GEMMs due to contention.
2. **Fill-in-the-blank:** In Expert Parallelism for a sparse MoE layer, tokens are routed to their assigned expert GPUs using an _____ Dispatch collective and returned to their originating GPUs using an _____ Combine collective.
   Answer: All-to-All; All-to-All.

Section 10: Composing Multi-Dimensional Parallelism and the Systems Trade-Off Triangle

KEY IDEA

No single parallelism strategy can scale a frontier language model to thousands of GPUs on its own: DDP hits the HBM wall; Tensor Parallelism and Expert Parallelism stall if pushed onto slow InfiniBand; Pipeline Parallelism accumulates bubbles if P grows too large relative to M; Sequence Parallelism helps only along context length T. Production training systems therefore compose **multi-dimensional parallelism**, mapping each axis directly onto the physical hardware tier whose bandwidth matches its communication frequency.

Mapping Parallelism Axes to Blackwell Physical Tiers

Visualize the complete cluster architecture as three concentric nested boxes, mirroring the memory and interconnect pyramid from Section 1:

1. **Innermost Tier — Single Blackwell GPU (HBM 8 TB/s · On-chip SRAM)**:
   - Techniques: **Kernel Fusion**, **Tiling**, and **Activation Checkpointing** (discarding intermediate activations in forward and recomputing them in backward to save HBM).

2. **Middle Tier — Blackwell NVLink 5.0 Domain (900 GB/s per direction · up to 72 GPUs in NVL72)**:
   - Techniques: **Tensor Parallelism (TP)**, **Expert Parallelism (EP — All-to-All)**, and **Context / Sequence Parallelism**. All three perform frequent, fine-grained, latency-sensitive communication inside every Transformer block and require NVLink's 900 GB/s-per-direction, non-blocking NVSwitch fabric.
3. **Outermost Tier — Across Pods / Nodes (InfiniBand ~0.05 TB/s · RoCE / Ethernet)**:
   - Techniques: **Data Parallelism (FSDP / ZeRO)** and **Pipeline Parallelism (PP)**. Pipeline Parallelism sends only stage-boundary activations via point-to-point `send` / `recv`, and FSDP overlaps per-layer All-Gathers and Reduce-Scatters with compute—allowing both to tolerate the $18\times$ lower bandwidth of inter-node InfiniBand or RoCE.

WORKED EXAMPLE

Let us work through a concrete cluster-mapping example to see how these dimensions multiply together cleanly. Suppose you are training a long-context sparse MoE model on **16 Blackwell NVL72 racks**, giving $16 \times 72 = 1{,}152$ GPUs total:

- **Inside each 72-GPU NVL72 rack (900 GB/s per direction NVLink 5.0)**:
  - Allocate $D_{\text{TP}} = 8$ GPUs to Tensor Parallelism (or $D_{\text{EP}} = 72$ for MoE layers across the full rack), and $D_{\text{CP}} = 9$ along Context Parallelism for long sequences ($8 \times 9 = 72$ GPUs per rack, keeping every TP All-Reduce, EP All-to-All, and attention exchange strictly inside NVLink 5.0).
- **Across the 16 NVL72 racks (~0.05 TB/s InfiniBand)**:
  - Allocate $D_{\text{PP}} = 4$ Pipeline Parallel stages across 4 racks (sending only stage-boundary activations over InfiniBand), and $D_{\text{FSDP}} = 4$ Sharded Data Parallel replicas across the remaining 4-way dimension ($4 \times 4 = 16$ racks, overlapping FSDP collectives with backward compute).
- Verify total GPU allocation: $D_{\text{TP}} \times D_{\text{CP}} \times D_{\text{PP}} \times D_{\text{FSDP}} = 8 \times 9 \times 4 \times 4 = 1{,}152$ GPUs. Every communication-intensive inner-loop collective stays inside the 900 GB/s-per-direction NVL72 domain, while only lower-frequency or overlapped traffic crosses the 0.05 TB/s InfiniBand fabric.

The Fundamental Systems Trade-Off Triangle

KEY IDEA

At the heart of the innermost tier—and governing every decision across the entire cluster—sits a single three-way trade-off:

$$\textbf{Recompute} \rightleftarrows \textbf{Store (HBM)} \rightleftarrows \textbf{Communicate}$$

For every intermediate tensor during training (activations, weights, gradients, optimizer states, KV caches), you have exactly three currencies you can spend:

- **Store it in HBM**: fastest to access later, zero extra FLOPs, zero network traffic—until you run out of HBM capacity.
- **Recompute it (spend compute)**: discard the tensor (for example, via activation checkpointing) to save HBM, and spend extra Tensor Core FLOPs to recompute it during the backward pass.
- **Shard and communicate it (spend interconnect bandwidth)**: split the tensor across GPUs (FSDP, TP, EP, PP, Sequence Parallelism) to reduce per-GPU HBM, paying interconnect latency and bandwidth when GPUs need data from peers.

WORKED EXAMPLE

Concrete trade-off example: suppose an attention layer's saved activations exceed remaining HBM by 12 GB per GPU on an NVL72 rack. You can either (1) **recompute** them during backward via activation checkpointing (trading Tensor Core FLOPs for HBM), (2) **communicate** by adding $2\times$ Sequence Parallelism across NVLink 5.0 (trading interconnect bandwidth for HBM), or (3) **store** them by freeing 12 GB of model state through FSDP parameter sharding (trading extra All-Gather bandwidth for HBM). You cannot eliminate the cost; you can only shift it onto whichever of the three currencies—compute, HBM, or interconnect bandwidth—has spare headroom.

Explicit Primitives vs. Compiler-Driven Frameworks

DEFINITION

Throughout this lecture, we built DDP, TP, PP, and collective decompositions from explicit PyTorch and NCCL primitives (`all_reduce`, `reduce_scatter_tensor`, `all_gather_into_tensor`, `send`, `recv`) so that every byte moved, every stream dependency, and every synchronization barrier is completely visible. In practice, you will also encounter **compiler-driven distributed stacks** (such as JAX on TPUs), where the programmer annotates how tensors should be sharded across a logical device mesh and the compiler automatically infers and inserts the required All-Gather, Reduce-Scatter, All-Reduce, and All-to-All collectives. Even when using an automated compiler stack, the exact same physical constraints—the memory-interconnect hierarchy, collective bus bandwidth, pipeline bubbles, SM contention, and the Recompute $\rightleftarrows$ Store $\rightleftarrows$ Communicate triangle—govern whether your distributed run saturates the hardware or stalls on data movement.

✎ Check Your Understanding

1. **What would happen if…** on a multi-rack Blackwell cluster, you swapped the placement of Tensor Parallelism and Pipeline Parallelism—putting TP across racks over InfiniBand (~0.05 TB/s) and PP inside a single NVL72 rack (900 GB/s per direction)?
   Answer: Training throughput would collapse. TP performs frequent, synchronous activation collectives on the critical path of every layer, which would stall severely over 0.05 TB/s InfiniBand, while NVLink's 900 GB/s-per-direction bandwidth would be vastly underutilized by PP's infrequent point-to-point stage transfers.
2. **Fill-in-the-blank:** Every distributed training system balances the same three fundamental currencies for any intermediate tensor: _____ (recompute), _____ (store), and _____ (communicate).
   Answer: Compute (FLOPs); HBM (memory capacity); interconnect bandwidth.