Lecture 8: Parallelism Basics

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

Section 1: Compute and Memory Ceilings, Interconnect Hierarchy, and Collective Primitives

Training a frontier language model fails immediately on a single accelerator for two independent reasons: memory capacity and compute throughput. On the memory side, mixed-precision training with reduced optimizer state requires roughly 12 bytes per parameter (we will derive the exact 12–16 bytes/parameter breakdown in Section 2). A single NVIDIA B300 GPU carries $288\ \mathrm{GB}$ of high-bandwidth memory (HBM), which can hold at most $288/12 = 24\mathrm{B}$ dense parameters before accounting for a single byte of activations. Meanwhile, frontier models have grown far past that single-device ceiling.

Consider the parameter-scaling curve across model generations (plotted with model generation along the horizontal axis and parameter count on a logarithmic vertical axis spanning $10\mathrm{B}$, $100\mathrm{B}$, $1\mathrm{T}$, and $3\mathrm{T}$). A horizontal dashed line sits near $24\mathrm{B}$, marking the capacity of a single $288\ \mathrm{GB}$ B300 at 12 bytes per parameter. Every landmark model sits well above that line: historical baselines include GPT-3 at $175\mathrm{B}$ (2020), MT-NLG at $530\mathrm{B}$ (2021), and DeepSeek-V3 at $671\mathrm{B}$ (Dec 2024); current 2026 frontier Mixture-of-Experts (MoE) models span DeepSeek-V4.1-Flash at $552\mathrm{B}$ backbone parameters (Sept 2026) up to Kimi K3 at $2.8\mathrm{T}$ total parameters (July 2026). Even if a single device somehow held all parameters in memory, the compute ceiling would remain insurmountable: frontier runs train on tens of trillions of tokens—DeepSeek-V4.1-Flash is pretrained on $45\mathrm{T}$ tokens—requiring exaflops of sustained cluster compute. The datacenter, not a single chip, is the unit of compute.

Across a cluster of N GPUs, distributed training targets two scaling goals simultaneously:

- **Linear memory scaling**: the maximum trainable model size grows proportionally to N.
- **Linear compute scaling**: useful floating-point operations per second (FLOP/s) grow proportionally to N.

Reaching both goals requires understanding the network connecting those N GPUs, because communication speed varies by more than an order of magnitude depending on where two GPUs sit in the cluster.

Scale-Up vs. Scale-Out Interconnects

Modern training clusters are built as a two-tier interconnect hierarchy rather than a flat pool of uniformly connected devices.

DEFINITION

Scale-up domain: A tightly coupled group of GPUs connected through a high-bandwidth, low-latency fabric—specifically NVLink and NVSwitch—so that any GPU in the domain can communicate directly with any other at full link speed.

DEFINITION

Scale-out fabric: The inter-rack network (such as InfiniBand or RDMA over a fat-tree topology) that connects separate scale-up domains across the datacenter at significantly lower per-GPU bandwidth than the intra-domain fabric.

To see this hierarchy concretely, visualize two adjacent GB200 or GB300 NVL72 racks bridged by a scale-out network. Inside Rack 0 sit 72 GPUs (GPU 0 through GPU 71), holding 13.8 to 20.7 TB of aggregate HBM and drawing 145 kW all-in on a GB200 NVL72 rack. Every GPU connects downward into a shared **NVSwitch Fabric (NVLink 5)** providing 900 GB/s unidirectional (1.8 TB/s bidirectional) bandwidth per GPU. Multiplying 72 GPUs $\times$ 900 GB/s gives 64.8 TB/s of rack total unidirectional bandwidth. Rack 1 is an identical 72-GPU all-to-all NVLink domain. Between Rack 0 and Rack 1 sits the scale-out InfiniBand/RDMA fat-tree bridge, which delivers 50 GB/s unidirectional (400 Gb/s) per GPU—exactly $18\times$ slower than intra-rack NVLink. For historical context, earlier HGX A100, H100, and B200 systems had only 8 GPUs per NVLink domain (a single node), whereas NVL72 expands that uniform high-speed domain to 72 GPUs.

Scale-up domains do not expand infinitely across an entire datacenter because physical constraints—copper reach, switch radix, cooling, and rack power (145 kW on GB200 NVL72)—cap how many GPUs can share one NVSwitch fabric.

COMMON MISCONCEPTION

You might think that the 64.8 TB/s figure for a 72-GPU NVL72 rack is the rack's bisection bandwidth. That breaks because 72×900 GB/s $= 64.8$ TB/s is the **rack total** unidirectional injection bandwidth across all 72 GPUs combined, whereas bisection bandwidth measures traffic crossing a cut between two halves of the rack (36 GPUs talking to the other 36 GPUs). Never call the 72-GPU sum "bisection" bandwidth.

This $18\times$ bandwidth cliff dictates the fundamental **placement rule** of distributed training: keep bandwidth-heavy parallelism axes—Tensor Parallelism (TP), Expert Parallelism (EP), and Context Parallelism (CP) all-to-all—**inside the NVLink scale-up domain**, and place lower-bandwidth axes—Pipeline Parallelism (PP) and Data Parallelism (DP)—**across racks** over the scale-out RDMA fabric.

Network Topology and Traffic Patterns

Bandwidth alone does not determine communication speed; network topology must match the communication traffic pattern. Compare two canonical topologies:

- **Direct mesh / torus**: GPUs sit on a grid (such as a 3×3 mesh) with direct links only to immediate neighbors. This topology uses low-radix switches and cheap links, making it effective for structured neighbor traffic such as rings and dense Tensor Parallelism. However, irregular routing between arbitrary pairs requires traversing many hops.
- **Switched all-to-all (NVSwitch / Leaf-Spine)**: Every GPU connects via a single link up to a central switch fabric (NVSwitch inside a rack, or a leaf-spine fat-tree across racks), so reaching any other GPU takes one hop up and one hop down with uniform latency for every pair.

No single network fits every traffic pattern equally cheaply, but switched all-to-all topologies are essential for unstructured **MoE expert routing**, where tokens jump dynamically to arbitrary GPUs. Blackwell clusters combine an NVSwitch scale-up rack with a switched fat-tree RDMA scale-out fabric, placing irregular all-to-all traffic inside the NVSwitch domain.

Collective Communication Primitives

Every distributed training strategy decomposes into point-to-point sends between specific pairs of GPUs plus six standard **collective communication primitives** operating over a group of ranks (GPUs). Consider 4 ranks (numbered $0, 1, 2, 3$), each holding a column of tensor data before and after the operation:

> **DEFINITION**
> **Collective communication primitive**: A coordinated data movement or reduction operation executed jointly by all GPUs (ranks) in a process group, abstracting away individual point-to-point transfers.

1. **Broadcast**: One root rank (say Rank 1) holds tensor A, and the other ranks are empty. After Broadcast, all 4 ranks hold identical copies of A.
2. **Reduce**: The 4 ranks hold distinct tensors A, B, C, D of the same shape. Reduce sums them elementwise onto a single root rank, which ends with $\Sigma = A + B + C + D$.
3. **All-Reduce**: Each rank starts with A, B, C, D, and after the collective every rank holds the complete elementwise sum $\Sigma = A + B + C + D$. This is the core primitive of Distributed Data Parallelism (DDP) and Tensor Parallelism (TP).
4. **All-Gather**: Each rank starts with a distinct shard (slice) of a tensor—Rank 0 holds shard 0, Rank 1 holds shard 1, Rank 2 holds shard 2, Rank 3 holds shard 3. After All-Gather, every rank holds the full concatenation of all 4 shards. Used in ZeRO/FSDP and Sequence Parallelism (SP).
5. **Reduce-Scatter**: Each rank starts with a full tensor partitioned into 4 slices (Rank 0 holds A_0, A_1, A_2, A_3; Rank 1 holds B_0, B_1, B_2, B_3; and so on). Reduce-Scatter sums across ranks elementwise and scatters the result so that Rank i receives only the reduced i-th shard $\Sigma_i = A_i + B_i + C_i + D_i$.
6. **All-to-All**: Each rank holds 4 distinct chunks addressed to the 4 destination ranks (Rank 0 holds chunks $0{\to}0, 0{\to}1, 0{\to}2, 0{\to}3$, and similarly for Ranks $1, 2, 3$). After All-to-All, Rank j holds the chunks addressed to j from every rank ($0{\to}j, 1{\to}j, 2{\to}j, 3{\to}j$). Geometrically, All-to-All is a

distributed matrix transpose across ranks. Used in MoE Expert Parallelism (EP) and Ulysses Context Parallelism (CP).

The Key Collective Identity

The single most important algebraic identity in distributed training links All-Reduce, Reduce-Scatter, and All-Gather:

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

Trace the transformation across $N = 4$ ranks to see why this holds identically. Start with 4 full tensors A, B, C, D, each of total size Ψ bytes, split into $N = 4$ equal shards of size Ψ/N. First, run **Reduce-Scatter**: every rank sends its $N - 1$ unowned shards and receives the contributions needed to sum its own shard, ending with Rank i holding Σ_i (size Ψ/N). Next, run **All-Gather** on those four shards $\Sigma_0, \Sigma_1, \Sigma_2, \Sigma_3$: every rank broadcasts its Ψ/N shard to the other $N - 1$ ranks, so every rank finishes with the concatenated full sum $\Sigma = [\Sigma_0, \Sigma_1, \Sigma_2, \Sigma_3]$—the exact output of All-Reduce.

Now quantify the per-GPU communication volume (bytes sent or received by each rank) for each half. In Reduce-Scatter, a rank keeps its own $1/N$ shard locally and must send the remaining $N - 1$ shards of size Ψ/N across the network:

$$V_{\text{RS}} = \frac{N-1}{N}\,\Psi \approx \Psi$$

In All-Gather, a rank already owns its $1/N$ reduced shard and must receive the other $N - 1$ reduced shards of size Ψ/N:

$$V_{\text{AG}} = \frac{N-1}{N}\,\Psi \approx \Psi$$

Summing the two halves gives the total per-GPU communication volume of All-Reduce:

$$V_{\text{AR}} = V_{\text{RS}} + V_{\text{AG}} = 2\,\frac{N-1}{N}\,\Psi \approx 2\Psi$$

Let us unpack every term in $2\frac{N-1}{N}\Psi$:

- Ψ is the size in bytes of the tensor being reduced.
- N is the number of participating ranks in the collective group.
- The factor $(N-1)/N$ accounts for the $1/N$ slice that stays local on each GPU and does not traverse the network; as N grows, $(N-1)/N \to 1$.
- The leading factor of 2 reflects the two full phases: one pass of size $\frac{N-1}{N}\Psi$ to reduce each shard onto its owner, and a second pass of size $\frac{N-1}{N}\Psi$ to replicate every reduced shard back to all ranks.

In the **bandwidth-limited regime** (large tensors where transfer time dominates startup latency), executing All-Reduce as Reduce-Scatter followed by All-Gather is bandwidth-optimal. That fact unlocks a powerful systems trick we will rely on repeatedly in Section 2 (ZeRO) and Section 5 (Sequence Parallelism): if you insert local computation **between** Reduce-Scatter and All-Gather—such as updating

only your $1/N$ shard of optimizer states in ZeRO, or running pointwise LayerNorm and Dropout on your $1/N$ sequence slice in Sequence Parallelism—you gain N-way sharded memory at **zero extra communication bandwidth** compared to an unsharded All-Reduce.

WORKED EXAMPLE

Comparing All-Reduce Time Inside NVLink vs. Across RDMA

Suppose $N = 64$ GPUs execute an All-Reduce on a BF16 tensor of 1B elements, so $\Psi = 2\ \text{GB}$.

1. **Per-GPU volume for each half**:

$$V_{\text{RS}} = V_{\text{AG}} = \frac{64-1}{64} \times 2\ \text{GB} = \frac{63}{64} \times 2\ \text{GB} = 1.96875\ \text{GB}$$

2. **Total per-GPU All-Reduce volume**:

$$V_{\text{AR}} = 2 \times 1.96875\ \text{GB} = 3.9375\ \text{GB} \approx 2\Psi = 4\ \text{GB}$$

3. **Transfer time inside a GB200/GB300 NVL72 domain** ($B_{\text{NVLink}} = 900\ \text{GB/s}$ unidirectional per GPU):

$$t_{\text{NVLink}} = \frac{3.9375\ \text{GB}}{900\ \text{GB/s}} = 4.375\ \text{ms}$$

4. **Transfer time across racks over scale-out RDMA** ($B_{\text{RDMA}} = 50\ \text{GB/s}$ unidirectional per GPU):

$$t_{\text{RDMA}} = \frac{3.9375\ \text{GB}}{50\ \text{GB/s}} = 78.75\ \text{ms}$$

Interpretation: Moving the exact same $2\ \text{GB}$ All-Reduce across the scale-out boundary slows communication by exactly $78.75/4.375 = 18\times$. Furthermore, splitting that All-Reduce into a $1.96875\ \text{GB}$ Reduce-Scatter followed by local sharded work and a $1.96875\ \text{GB}$ All-Gather costs the exact same $4.375\ \text{ms}$ (or $78.75\ \text{ms}$ over RDMA) in bandwidth time.

KEY IDEA

- **Interconnect placement governs distributed design**: NVL72 scale-up domains deliver $900\ \text{GB/s}$ unidirectional bandwidth across 72 GPUs via NVSwitch, whereas scale-out RDMA delivers $50\ \text{GB/s}$ ($18\times$ slower), so bandwidth-heavy and irregular all-to-all traffic must stay inside the NVLink domain.
- **The collective identity is a free memory lever**: Because $\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$ with optimal bandwidth $2\frac{N-1}{N}\Psi \approx 2\Psi$, inserting local computation between the two halves yields sharded memory without adding communication bytes.

We now have the hardware fabric and communication primitives; the next question is how to put them to work to parallelize training, starting with the simplest strategy—splitting the data batch across GPUs—and seeing exactly where it breaks.

✎ Check Your Understanding

1. A cluster runs on GB300 NVL72 racks ($900\ \text{GB/s}$ unidirectional per GPU inside the rack, $50\ \text{GB/s}$ unidirectional per GPU over inter-rack RDMA). Fill in the blank: Placing a bandwidth-heavy collective across racks instead of inside one NVLink domain reduces per-GPU link bandwidth by a factor of _____.

Answer: $18\times$ (since $900\ \text{GB/s} / 50\ \text{GB/s} = 18$).

2. Yes or no with justification: Does replacing a single All-Reduce of a tensor of size Ψ with a Reduce-Scatter followed by local per-rank computation and an All-Gather increase the total network byte volume per GPU?

Answer: No. Reduce-Scatter and All-Gather each move $\frac{N-1}{N}\Psi$ bytes per GPU, summing to $2\frac{N-1}{N}\Psi$, which is identical to the bandwidth-optimal volume of All-Reduce.

Section 2: Data Parallelism, Static Memory Accounting, and the ZeRO / FSDP Progression

The most direct way to harness M GPUs is to replicate the entire model on every GPU and divide the training batch across them.

DEFINITION

Distributed Data Parallelism (DDP): A parallelism strategy in which all M GPUs hold identical copies of model parameters, compute gradients independently on disjoint slices of B/M examples from a global batch of size B, and synchronize gradients via All-Reduce before taking an identical optimizer step.

Mathematically, mini-batch gradient descent updates parameters θ_t using the sum (or average) of gradients across all B examples in the batch:

$$\theta_{t+1} = \theta_t - \eta \sum_{i=1}^{B} \nabla f(x_i)$$

Decomposing the equation term by term:

- θ_t and θ_{t+1} are the parameter vectors at training steps t and $t+1$.
- η is the learning rate.

- $\nabla f(x_i)$ is the loss gradient evaluated on training example x_i.
- Because summation is associative and commutative, $\sum_{i=1}^{B} \nabla f(x_i)$ partitions cleanly into M independent partial sums of B/M examples each, computed in parallel on the M GPUs and summed via a single **gradient All-Reduce** of volume 2Ψ bytes per step (recall Section 1).

Naïve DDP achieves linear compute scaling as long as the per-GPU micro-batch B/M is large enough to saturate each GPU's matrix units. However, DDP hits **two hard walls**:

1. **The batch wall**: Obviously $M \leq B$ (you cannot give a GPU fewer than one example), and more subtly, optimization efficiency stalls once the global batch size B exceeds the problem's **critical batch size (noise scale)** $\mathcal{B}$. Visualize a log-log plot with normalized batch size $B/\mathcal{B}$ on the horizontal axis (ranging from 10^{-2} to 10^{2}, with a vertical dashed line at $B/\mathcal{B} = 10^{0} = 1$) and training speed on a logarithmic vertical axis. To the left of 10^{0}, the curve rises steeply in a straight line—this is the **perfect scaling** region where doubling B doubles training speed. Right around $B/\mathcal{B} = 10^{0}$, the curve bends over and flattens horizontally into the **ineffective scaling** region: pushing $B \gg \mathcal{B}$ burns extra FLOPs without meaningfully reducing the number of optimization steps needed to reach a target loss.
2. **Zero memory scaling**: Every GPU holds a full copy of the model weights, gradients, and optimizer state, so static memory per GPU does not shrink at all as M grows.

Why Naïve DP Runs Out of Memory: Static Memory Accounting

To see why replicating the entire training state on every GPU is so wasteful, we must account for every byte of static memory per parameter under mixed-precision AdamW (where forward and backward passes run in BF16/FP16 while the optimizer maintains high precision for numerical stability). Mixed-precision AdamW stores **five parameter-sized tensors** on every GPU:

1. **Working parameters (BF16)**: 2 bytes/param.
2. **Gradients (BF16)**: 2 bytes/param.
3. **FP32 master weights**: 4 bytes/param (required so small weight updates are not lost to BF16 rounding).
4. **Adam first moment** m: 4 bytes/param in FP32 (or 2 bytes/param in BF16).
5. **Adam second moment** v: 4 bytes/param in FP32 (or 2 bytes/param in BF16).

Group the last three tensors as the **optimizer state** of size K bytes per parameter. Visualize two horizontal stacked memory bars:

- **Standard FP32 Optimizer State ($K = 12$)**: Params (2 B) + Grads (2 B) + FP32 Master Weights (4 B) + Adam m (4 B) + Adam v (4 B) $= (2 + 2 + 12)\Psi_{\text{count}} = 16$ bytes/param. Optimizer state accounts for $12/16 = 75\%$ of static HBM.
- **Reduced State — BF16 Moments / Kahan Baseline ($K = 8$)**: Params (2 B) + Grads (2 B) + FP32 Master Weights (4 B) + BF16 m (2 B) + BF16 v (2 B) $= (2 + 2 + 8)\Psi_{\text{count}} = 12$ bytes/param (or pure BF16 with Kahan summation achieving the same $12\ \text{B/param}$ footprint). Optimizer state still consumes $8/12 = 66.7\%$ of static HBM.

Replicating 67% to 75% of static HBM across all N_d data-parallel ranks throws away most of the cluster's memory storing identical copies of optimizer states that are only touched once per step.

ZeRO: Sharding Redundant State Across DP Ranks

ZeRO (Zero Redundancy Optimizer) eliminates that waste in three progressive stages across N_d data-parallel ranks by exploiting the collective identity from Section 1 ($\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$). Let Ψ denote the number of model parameters (so BF16 parameters occupy 2Ψ bytes, BF16 gradients occupy 2Ψ bytes, and optimizer states occupy $K\Psi$ bytes; note that in communication volume expressions, Ψ is also used for the BF16 tensor byte size when stating 2Ψ per All-Reduce—we will always keep units explicit so there is no ambiguity).

DEFINITION

ZeRO Stage 1 (P_{os}): Shards only the optimizer states across the N_d data-parallel ranks, reducing static memory per GPU from $(4 + K)\Psi$ bytes to $(4 + K/N_d)\Psi$ bytes at the exact same communication volume as DDP.

DEFINITION

ZeRO Stage 2 (P_{os+g}): Shards both optimizer states and gradients across the N_d data-parallel ranks by reduce-scattering gradients layer by layer during the backward pass and immediately freeing unowned gradients, lowering static memory to $(2 + (2 + K)/N_d)\Psi$ bytes.

DEFINITION

ZeRO Stage 3 (FSDP — Fully Sharded Data Parallel): Shards parameters, gradients, and optimizer states across all N_d data-parallel ranks so static memory scales linearly as $\frac{(4+K)\Psi}{N_d}$, materializing full weights one block at a time via All-Gather for forward and backward passes.

In practice, Megatron Core's **distributed optimizer** implements ZeRO-1/2 semantics, while PyTorch **FSDP** implements ZeRO-3.

Trace the four-step execution cycle of **ZeRO Stage 1 and Stage 2** to see why they are "free" memory wins over DDP:

1. **Backward**: Every GPU computes full gradients on its local micro-batch.
2. **Reduce-Scatter gradients** ($1\times$ parameter-tensor bytes, i.e., $2\Psi_{\text{count}}$ bytes in BF16, denoted Ψ_{bytes}): Each rank receives the summed gradient for only its owned $1/N_d$ parameter slice. In **Stage 2**, this Reduce-Scatter runs layer by layer during backward and **immediately frees** unowned gradients rather than holding the full gradient buffer until backward finishes.
3. **Local optimizer step**: Each rank updates only its $1/N_d$ slice of FP32 master weights and moments (m, v), and casts the updated slice back to BF16 working weights.
4. **All-Gather parameters** (Ψ_{bytes}): All ranks gather the updated BF16 parameter slices so every GPU holds the updated full model for the next forward pass.

Comparing communication volume across DDP, ZeRO-1, and ZeRO-2 (writing Ψ_{bytes} for the size of one working-parameter tensor, which the slides denote Ψ):

Method	Collectives per Step	Comm Volume	Static Memory per GPU (bytes, Ψ params)
Naïve DDP	1 All-Reduce (grads)	$2\Psi_{\text{bytes}}$	$(2+2+K)\Psi$
ZeRO-1 (P_{os})	1 Reduce-Scatter + 1 All-Gather	$2\Psi_{\text{bytes}}$ (free)	$2\Psi + 2\Psi + \frac{K\Psi}{N_d}$
ZeRO-2 (P_{os+g})	Per-layer Reduce-Scatter + All-Gather	$2\Psi_{\text{bytes}}$ (almost free)	$2\Psi + \frac{(2+K)\Psi}{N_d}$
ZeRO-3 (FSDP)	Forward AG + Backward AG + Grad RS	$3\Psi_{\text{bytes}}$ ($1.5\times$ DDP)	$\frac{(2+2+K)\Psi}{N_d}$

Because Reduce-Scatter plus All-Gather matches All-Reduce bandwidth exactly, ZeRO-1 should almost always replace plain DDP.

> **COMMON MISCONCEPTION**
> You might think that ZeRO-1 and ZeRO-2 must incur extra communication overhead because they perform two separate collectives (Reduce-Scatter and All-Gather) instead of DDP's single All-Reduce. That breaks because an All-Reduce *is* internally a Reduce-Scatter followed by an All-Gather. ZeRO-1 simply moves the local optimizer update into the gap between those two halves, transmitting the exact same $2\frac{N_d-1}{N_d}\Psi_{\text{bytes}} \approx 2\Psi_{\text{bytes}}$ across the network.

ZeRO Stage 3 (FSDP) and Communication Prefetching

In ZeRO-2, every GPU still stores a full copy of the 2Ψ BF16 parameters. **ZeRO Stage 3 (FSDP)** shards those parameters across the N_d ranks as well, achieving true $1/N_d$ static memory scaling. To keep peak memory bounded to a single layer rather than the entire model, FSDP wraps each Transformer layer as an **FSDP block** and executes two block-level loops:

- **Forward (per FSDP block)**: (1) **All-Gather weights** for the current block (comm cost: Ψ_{bytes} across the full model) $\rightarrow$ (2) **Local Forward** pass using those materialized weights $\rightarrow$ (3) **Free Full Weights** (keeping only the local $1/N_d$ shard) $\rightarrow$ (4) advance to the next block.
- **Backward + Update (per FSDP block)**: (1) **All-Gather weights** again because full weights were discarded after forward (comm cost: Ψ_{bytes}) $\rightarrow$ (2) **Local Backward** to compute local gradients $\rightarrow$ (3) **Reduce-Scatter** gradients onto owning ranks (comm cost: Ψ_{bytes}) $\rightarrow$ (4) **Free Full Weights** $\rightarrow$ (5) **Local Optimizer Step** on each rank's owned $1/N_d$ shard.

Total communication per step is $\Psi_{\text{bytes}} + \Psi_{\text{bytes}} + \Psi_{\text{bytes}} = 3\Psi_{\text{bytes}}$, which is $1.5\times$ DDP.

Even that $3\Psi_{\text{bytes}}$ cost does not have to stall the GPU if we separate the **GPU Compute Stream** and the **GPU Communication Stream** and overlap communication with matrix multiplications (GEMMs). Trace the timeline for a three-weight computation $(W_1W_0 + W_2W_0)x = y$:

- **Forward pass**: The communication stream first runs `AG W0` (exposed startup latency). As soon as `W0` arrives, the compute stream runs `FWD 0` while the communication stream simultaneously prefetches `AG W1`. Next, `FWD 1` runs on compute while `AG W2` runs on comm; once `FWD 1` completes, the compute stream frees `W1` and immediately runs `FWD 2`.
- **Backward pass**: The communication stream starts `AG W2` (if not still resident); once ready, compute runs `BWD 2` while comm prefetches `AG W1` (or reduces `g2`). Specifically, as each backward block finishes—`BWD 2`, then `BWD 1`, then `BWD 0`—the compute stream frees the materialized weight and immediately launches `RS g2`, `RS g1`, and `RS g0` on the communication stream in parallel with the subsequent backward blocks and `AG` prefetches.

When layer compute time meets or exceeds layer communication time ($T_{\text{compute}} \geq T_{\text{comm}}$), **only the very first All-Gather and the very final Reduce-Scatter remain exposed**. We will generalize this overlap principle to all six parallelism axes in Section 8.

WORKED EXAMPLE

ZeRO Capacity on 8× NVIDIA B300 (288 GB) vs. Historical 8× A100 (80 GB)

Assume reduced optimizer state at $12\ \text{B/param}$ (2 B params + 2 B grads + 8 B optimizer state, so $K = 8$), ignoring activations for the moment, across $N_d = 8$ GPUs. Compute per-parameter memory and maximum trainable model size on $8 \times \text{B300}$ ($288\ \text{GB}$ HBM per GPU) and on historical $8 \times \text{A100}$ ($80\ \text{GB}$ HBM per GPU):

1. **Baseline DDP**:

$$\text{Bytes/param} = 12.00 \implies \text{Max B300 model} = \frac{288\ \text{GB}}{12.00\ \text{B/p}} = 24\text{B} \quad (\text{A100: } \frac{80}{12} = 6$$

2. **ZeRO Stage 1**:

$$\text{Bytes/param} = 2 + 2 + \frac{8}{8} = 5.00 \implies \text{Max B300 model} = \frac{288}{5.00} = 57.6\text{B} \quad (\text{A100}$$

3. **ZeRO Stage 2**:

$$\text{Bytes/param} = 2 + \frac{2+8}{8} = 3.25 \implies \text{Max B300 model} = \frac{288}{3.25} \approx 88.6\text{B} \quad (\text{A100}$$

4. **ZeRO Stage 3 (FSDP)**:

$$\text{Bytes/param} = \frac{12}{8} = 1.50 \implies \text{Max B300 model} = \frac{288}{1.50} = 192\text{B} \quad (\text{A100: } \frac{80}{1.5} \approx$$

Interpretation: Even with full ZeRO-3 sharding across $8 \times \text{B300}$ GPUs and zero space reserved for activations, the 192B ceiling is still far too small for historical DeepSeek-V3 (671B, Dec 2024) or 2026 frontier models such as DeepSeek-V4.1-Flash (552B) and Kimi K3 (2.8T). Moreover, ZeRO-3

does not shard activations, so peak HBM grows with sequence length and micro-batch size, and scaling pure ZeRO-3 to very large GPU counts under a fixed global batch size B shrinks the per-GPU micro-batch B/N_d until $T_{\text{compute}} < T_{\text{comm}}$ and communication can no longer be hidden.

KEY IDEA

- **Optimizer state dominates static memory**: Mixed-precision AdamW requires 12–$16\ \text{B/param}$, with 67%–75% tied up in FP32 master weights and moments (m, v).
- **ZeRO trades zero-to-modest communication for linear static memory scaling**: ZeRO-1/2 shard optimizer state and gradients at DDP's 2Ψ volume, while ZeRO-3 (FSDP) shards parameters too at 3Ψ volume hidden behind compute via stream prefetching—but none of them shard activations or escape the global batch-size wall.

Before introducing model parallelism to break past ZeRO-3's limits, there is a direct, orthogonal lever that shrinks both static memory and collective communication bytes at the same time: lowering the numerical precision of training itself.

✎ Check Your Understanding

1. Under mixed-precision AdamW with standard FP32 optimizer state ($K = 12$, total $16\ \text{B/param}$), what is the static memory per parameter on each GPU when running ZeRO Stage 1 across $N_d = 16$ ranks?

Answer: $2 + 2 + 12/16 = 4.75\ \text{B/param}$, using the ZeRO-1 formula $2 + 2 + K/N_d$.

2. What happens to FSDP (ZeRO-3) efficiency if you keep global batch size B fixed and keep increasing N_d so that the per-GPU micro-batch B/N_d becomes very small?

Answer: Local layer compute time T_{compute} shrinks proportionally to B/N_d, while per-layer All-Gather and Reduce-Scatter volume stays fixed at $3\Psi_{\text{bytes}}$; once $T_{\text{compute}} < T_{\text{comm}}$, prefetching can no longer hide communication and GPUs stall waiting on the network.

Section 3: Low-Precision Training as a Parallelism Lever

When a distributed training run is bottlenecked by memory capacity and collective bandwidth, every byte trimmed from a parameter, activation, or gradient pays off twice: it accelerates Tensor Core GEMMs **and** cuts both HBM footprint and network bytes moved by collectives.

Recall from Lecture 5 how **BF16** became the baseline training format by matching FP32's exponent range, which eliminated FP16 loss scaling while holding parameters in $2\ \text{B/param}$. To push further, hardware and training recipes have progressed through three finer-grained low-precision formats:

Format	Era / Hardware	Key Mechanism
BF16	Standard (recap Lecture 5)	Matches FP32 exponent range, eliminating FP16 loss scaling ($2\ \mathrm{B/param}$)
FP8 (block-scaled)	Hopper (historical DeepSeek-V3, Dec 2024)	Fine-grained tile/block scaling stabilizes forward and backward GEMMs ($1\ \mathrm{B/param}$)
MXFP8	Blackwell native (2025–26)	Hardware microscaling FP8 with shared block exponents
NVFP4	Blackwell native (2025–26)	4-bit GEMMs + block scales, random Hadamard transforms, and stochastic rounding

The direct parallelism win is immediate: storing working parameters in FP8 ($1\ \mathrm{B/param}$ instead of BF16's $2\ \mathrm{B/param}$) cuts FSDP / ZeRO parameter All-Gather bytes by $2\times$ and shrinks per-parameter static memory directly.

However, naïvely quantizing every operation in a Transformer to FP8 or 4-bit causes training divergence. Stable low-precision training always preserves **high-precision islands**: the **MoE router, normalization layers, embeddings, LM head, and the first and last blocks** stay in BF16 or FP32 for numerical stability.

> **DEFINITION**
> **High-precision islands**: Specific numerically sensitive modules in a low-precision training run—including MoE routers, normalization layers, embeddings, the LM head, and early/late Transformer blocks—that remain in BF16 or FP32 to prevent divergence while bulk linear GEMMs execute in FP8 or FP4.

Blackwell NVFP4 Pretraining Recipe

Pushing all the way to 4-bit training requires careful control of quantization geometry and rounding noise. The **Blackwell NVFP4 pretraining recipe** (NVIDIA, Sept 2025) lays out four distinct precision tiers across the model:

1. **BF16 / Original Precision (~16% of blocks plus sensitive ops)**: The **first 2 and last 8 blocks**, along with **attention GEMMs, embeddings, the LM head, and normalization layers**, stay in BF16/original precision.
2. **NVFP4 Linear GEMMs (~84% of layers)**: Bulk feed-forward and projection linear GEMMs run in NVFP4 using asymmetric block scaling tailored to each tensor's role:
   - **Weights (16×16 2D blocks)**: Quantizing weights in square 2D tiles ensures that the forward GEMM (which reads W) and the backward activation-gradient GEMM (which reads $W^\top$) see the **exact same quantized weight values**, preventing subtle forward/backward mismatch.

- **Activations and Gradients (1×16 1D blocks)**: Fine-grained 1D micro-blocks along the inner GEMM dimension capture local dynamic range tightly.

3. **Targeted noise control**:
   - **Random Hadamard transforms** are applied to **weight-gradient GEMM inputs only**, spreading extreme outlier magnitudes across channels before 4-bit quantization.
   - **Stochastic rounding** is applied to **gradients only**, ensuring that rounding errors remain zero-mean (unbiased) across micro-batches and steps rather than systematically zeroing out small gradients.
4. **FP32 Master State**: **Master weights, accumulated gradients, and optimizer states** remain in FP32 so small updates accumulate accurately over trillions of tokens.

In a 12B-parameter / 10T-token pretraining test, this NVFP4 recipe achieved **62.58 MMLU-Pro** compared to **62.62** for FP8—virtually matching 8-bit quality at 4-bit GEMM precision. In production, this recipe pretrained **Nemotron 3 Super & Ultra** (early 2026), and **Kimi K3** (July 2026) trains its MoE experts with **MXFP4 quantization-aware training (QAT)**.

COMMON MISCONCEPTION

You might think that training in FP8 or NVFP4 cuts total static training memory per GPU by $2\times$ or $4\times$. That breaks because the dominant portion of static memory—FP32 master weights, gradients, and optimizer states ($K = 8$ to $12\ \text{B/param}$)—stays in higher precision for numerical stability. Lowering working weights from BF16 ($2\ \text{B}$) to FP8 ($1\ \text{B}$) halves the **working-weight memory and FSDP All-Gather traffic**, but does not shrink the 8–$12\ \text{B/param}$ optimizer state unless you also shard or compress optimizer states.

WORKED EXAMPLE

Quantifying the FP8 Working-Weight Win Under ZeRO-3

Consider a 70B-parameter model running ZeRO-3 (FSDP) across $N_d = 8$ GPUs with $2\ \text{B}$ BF16 gradients and $K = 8\ \text{B}$ optimizer state (FP32 master weights + BF16 moments). Compare BF16 working weights ($2\ \text{B/param}$) against FP8 working weights ($1\ \text{B/param}$):

1. **Weight All-Gather volume per step** (one All-Gather in forward + one in backward, using $\frac{N_d-1}{N_d} \approx 1$ for simplicity, or exact $\frac{7}{8}$):
   - **BF16 working weights**: One full weight tensor is $70\text{B} \times 2\ \text{B} = 140\ \text{GB}$. Across forward and backward, each GPU moves $2 \times \frac{7}{8} \times 140\ \text{GB} = 245\ \text{GB}$ of parameter All-Gathers.
   - **FP8 working weights**: One full weight tensor is $70\text{B} \times 1\ \text{B} = 70\ \text{GB}$. Across forward and backward, each GPU moves $2 \times \frac{7}{8} \times 70\ \text{GB} = 122.5\ \text{GB}$—an exact $2\times$ **reduction** in weight communication bytes.
2. **Static memory per GPU under ZeRO-3 ($N_d = 8$)**:
   - **BF16 working weights**: $\frac{2+2+8}{8} = 1.50\ \text{B/param} \implies 70\text{B} \times 1.50 = 105\ \text{GB/GPU}$.

- **FP8 working weights**: $\frac{1+2+8}{8} = 1.375\ \text{B/param} \implies 70\text{B} \times 1.375 = 96.25\ \text{GB/GPU}$.

Interpretation: Switching working weights to FP8 cuts FSDP weight All-Gather bandwidth in half ($245\ \text{GB} \to 122.5\ \text{GB}$), which directly relaxes the overlap condition $T_{\text{compute}} \geq T_{\text{comm}}$. At the same time, total static HBM only drops by 8.3% ($105\ \text{GB} \to 96.25\ \text{GB}$) because FP32 master weights, moments, and BF16 gradients still account for $10\ \text{B/param}$ of the sharded state.

KEY IDEA

- **Low precision is a direct bandwidth and compute multiplier**: Moving working weights from BF16 to FP8/MXFP8/NVFP4 accelerates Tensor Core GEMMs and halves (or better) parameter All-Gather traffic.
- **Stability requires selective precision and structured quantization**: Stable 4-bit training keeps sensitive islands in BF16/FP32, uses 16×16 2D weight blocks for forward/backward consistency, 1×16 activation/gradient blocks, Hadamard transforms on weight-gradient inputs, and stochastic rounding on gradients.

Even with FP8/NVFP4 and ZeRO-3 combined, data parallelism alone still forces every rank to process its own slice of the global batch and leaves activations unsharded. To scale model capacity without growing the global batch size—and to cut activation memory—we must move beyond data parallelism to **model parallelism**.

✎ Check Your Understanding

1. Why does the Blackwell NVFP4 pretraining recipe quantize weights in 16×16 2D blocks rather than 1×16 1D blocks?

Answer: Using symmetric 16×16 2D blocks guarantees that the forward GEMM (reading W) and the backward activation-gradient GEMM (reading $W^\top$) see identical quantized weight values.

2. Yes or no with justification: Under the NVFP4 pretraining recipe, are random Hadamard transforms and stochastic rounding applied to the forward-pass weight tensors?

Answer: No. Random Hadamard transforms are applied only to weight-gradient GEMM inputs to smooth outliers, and stochastic rounding is applied only to gradients to keep them unbiased.

Section 4: Pipeline Parallelism and Bubble-Eliminating Schedules

When a model outgrows what data parallelism and low precision can sustain—or when the global batch size B hits the critical batch size $\mathcal{B}$—we split the model itself across devices.

> **DEFINITION**
> **Model parallelism**: A family of strategies that shard model parameters across GPUs (like ZeRO-3) but communicate **activations** between GPUs instead of weights, allowing memory capacity to scale without increasing the global batch size.

Model parallelism splits a Transformer along four distinct architectural axes:

1. **Pipeline Parallelism (PP — Depth)**: Assigns contiguous groups of layers (e.g., layers 0–3, 4–7, 8–11, 12–15) to different GPUs, communicating point-to-point activations between stages. Low communication volume makes PP ideal for slower inter-rack RDMA links.
2. **Tensor + Sequence Parallelism (TP + SP — Width)**: Splits individual weight matrices across GPUs (column-parallel followed by row-parallel), requiring All-Reduce (or All-Gather + Reduce-Scatter) inside every layer. Needs a high-bandwidth NVLink domain.
3. **Expert Parallelism (EP — MoE)**: Places disjoint subsets of MoE experts on different GPUs, routing tokens with All-to-All dispatch and combine. Rack-wide EP is standard on NVL72.
4. **Context Parallelism (CP — Sequence)**: Shards the sequence dimension and KV cache across GPUs for long-context training up to 1M tokens.

We begin with **Pipeline Parallelism (PP)**, the natural cross-rack dimension.

Naïve Layer-Wise Execution vs. GPipe Micro-Batching

Suppose we split a model's depth evenly across $p = 4$ GPUs (GPU 0 through GPU 3). Under **naïve layer-wise execution**, GPU 0 runs the forward pass (`F`) on the entire batch and hands the output activations to GPU 1, which runs `F` and hands off to GPU 2, then GPU 3. GPU 3 then computes the loss and runs backward (`B`), passing gradients back up to GPU 2, GPU 1, and finally GPU 0. Look at the utilization diagram across 4 rows (GPU 0 to GPU 3) and 8 time slots: in every single time slot, **only one GPU is busy** while the other 3 sit idle. Active utilization is $1/p = 1/4$, leaving $(p-1)/p = 3/4$ of all GPU cycles wasted.

To fill those idle slots, **GPipe** divides the batch into m smaller **micro-batches** ($F_0, F_1, \ldots, F_{m-1}$) and feeds them back-to-back into stage 0. Trace GPipe with $p = 4$ GPUs and $m = 4$ micro-batches:

- GPU 0 executes `F0`, `F1`, `F2`, `F3` back-to-back.
- One time slot after `F0` starts on GPU 0, GPU 1 starts `F0`, then `F1`, `F2`, `F3`; GPU 2 starts one slot later; GPU 3 starts one slot after that.
- Once GPU 3 finishes `F3`, it immediately starts backward passes `B3`, `B2`, `B1`, `B0` (note backward blocks are wider than forward blocks because backward computes two GEMMs—input gradient and weight gradient—per forward GEMM).
- Backward gradients propagate back up to GPU 2, GPU 1, and GPU 0. Only after all m micro-batches finish backward does every GPU execute a synchronized optimizer `Step`.

Between the end of GPU 0's forward passes and the arrival of its first backward pass—plus the warmup ramp at the start and cooldown ramp at the end—sits empty white space called the **pipeline bubble**.

DEFINITION

Pipeline bubble: Idle GPU time at the start (warmup), middle, and end (cooldown) of a pipeline schedule caused by sequential data dependencies across the p pipeline stages.

Count the time units per GPU to derive the **bubble-to-compute ratio**. Let one forward-plus-backward micro-batch take $t_f + t_b$ on a single stage. Total useful compute time per GPU across m micro-batches is $m(t_f + t_b)$. During warmup, the last stage waits $(p-1)t_f$ before starting `F0`; during cooldown, the first stage waits $(p-1)t_b$ after finishing `F_{m-1}` before receiving `B0` (or symmetrically across stages, total idle time per GPU is $(p-1)(t_f + t_b)$). Taking the ratio of idle bubble time to useful compute time gives:

$$\text{Bubble-to-compute ratio} = \frac{(p-1)(t_f + t_b)}{m(t_f + t_b)} = \frac{p-1}{m}$$

Breaking this expression apart:

- $p - 1$ is the number of stage-to-stage pipeline hops needed to fill and drain the pipeline.
- m is the number of micro-batches executed before each synchronous optimizer step.
- The common factor $(t_f + t_b)$ cancels out cleanly, leaving a pure ratio of stage count to micro-batch count.

Hiding the bubble requires $m \gg p$. That explains an empirical scaling rule: at a small batch size like 8, throughput drops sharply as PP increases because m cannot stay large; at batch size 128, $m \gg p$ holds and throughput stays nearly flat.

Why does PP shine across racks over $50\ \text{GB/s}$ RDMA? First, it shards static weights by $1/p$. Second, the only network traffic between adjacent stages is a **point-to-point activation tensor** of size bsh elements per micro-batch (where b is micro-batch size, s is sequence length, and h is hidden dimension)—orders of magnitude smaller than an All-Reduce of model weights or per-layer Tensor Parallelism collectives.

Cutting Activation Memory and Bubbles: 1F1B and Interleaved Virtual Stages

GPipe hides the bubble when m is large, but creates a severe memory crisis: GPU 0 executes **all m forward passes before running a single backward pass**, so it must keep saved activations for all m micro-batches in HBM simultaneously—activation memory scales as $O(m)$.

The **1F1B (One Forward, One Backward)** schedule solves this memory blowup without changing the bubble ratio. Trace GPU 0 and GPU 3 in a $p = 4$ pipeline:

- **Warmup**: GPU 0 launches just $p = 4$ forward passes (`F0`, `F1`, `F2`, `F3`)—just enough to fill the 4-stage pipeline so `F0` reaches GPU 3.
- **Steady state**: As soon as GPU 3 finishes `F0`, it runs `B0` immediately and then alternates strictly `F1, B1, F2, B2, F3, B3, F4, B4, F5, B5`. Upstream on GPU 0, as soon as `B0` arrives back from GPU 1, GPU 0 finishes `B0`, **immediately frees micro-batch 0's activations**, and only then launches `F4`, alternating `B0, F4, B1, F5, B2...`.

- **Cooldown**: After all forwards finish, remaining backward passes (`B3, B4, B5`) drain out.

Because every GPU frees a micro-batch's activations as soon as its backward completes, peak in-flight micro-batches drop from $O(m)$ to $O(p)$.

> **DEFINITION**
> **1F1B (One Forward, One Backward) schedule**: A pipeline schedule that warms up with only p in-flight micro-batches and then alternates one forward and one backward pass in steady state, capping peak stored activation memory at $O(p)$ instead of GPipe's $O(m)$.

To shrink the bubble itself without growing m, **Interleaved 1F1B** assigns v **non-contiguous virtual stages** to every GPU. Visualize 4 GPUs with $v = 2$ virtual stages each:

- GPU 0 holds Stages 0 and 4.
- GPU 1 holds Stages 1 and 5.
- GPU 2 holds Stages 2 and 6.
- GPU 3 holds Stages 3 and 7.

Activations flow $0 \to 1 \to 2 \to 3$, then loop back via a dashed wrap-around link from GPU 3 back to GPU 0 to run Stages $4 \to 5 \to 6 \to 7$. Because each stage is now $v\times$ smaller, micro-batches travel around the pipeline v times in smaller chunks, making warmup and cooldown $v\times$ faster. The bubble-to-compute ratio shrinks by $v\times$:

$$\text{Interleaved bubble-to-compute ratio} = \frac{p-1}{v \cdot m}$$

The tradeoff is clear: looping around the ring v times multiplies point-to-point activation communication by $v\times$.

Zero-Bubble Pipelining and DualPipeV

Can we drive the pipeline bubble all the way toward zero without increasing m or v? Every backward pass through a linear layer $y = \sigma(Wx)$ (where $z = Wx$ and ∇z is the upstream gradient) actually consists of **two mathematically independent GEMMs**:

1. **Input gradient (B, urgent)**: $\nabla x = W^\top \nabla z$. The upstream pipeline stage is blocked waiting for ∇x before it can run its own backward pass.
2. **Weight gradient (W, flexible)**: $\nabla W = \nabla z \cdot x^\top$. Nothing upstream depends on ∇W; it only needs to finish before the optimizer step at the end of the iteration.

Zero-Bubble Pipelining (ZB-H1 / ZB-H2) decouples B from W. Trace the ZB schedule diagram: a stage executes `F0, F1, F2, F3`, then runs urgent `B0` immediately to unblock the upstream stage, and packs flexible `W0, W1, W2, W3` blocks into the idle gaps during steady state and cooldown right before the optimizer step.

Building further, **DualPipe** runs a bidirectional schedule that overlaps forward and backward passes moving from both ends of the pipeline simultaneously. **DualPipeV** cuts DualPipe in half (a V-shaped folding of stages across devices), delivering bidirectional forward/backward overlap while requiring **half the devices**. In practice, PyTorch's pipelining library ships GPipe, 1F1B, Interleaved 1F1B, ZeroBubble variants, and DualPipeV out of the box.

COMMON MISCONCEPTION

You might think that switching from GPipe to standard 1F1B reduces the pipeline bubble. That breaks because 1F1B executes the exact same number of forward and backward blocks with the exact same warmup and cooldown dependencies, so its bubble-to-compute ratio stays $\frac{p-1}{m}$. What 1F1B actually cuts is **peak in-flight activation memory**, from $O(m)$ down to $O(p)$; shrinking the bubble itself requires Interleaved 1F1B (v virtual stages), Zero-Bubble (B/W splitting), or DualPipe/DualPipeV.

WORKED EXAMPLE

Comparing Pipeline Schedules for $p = 8$ Stages and $m = 16$ Micro-Batches

Suppose a model is split across $p = 8$ pipeline stages and processes $m = 16$ micro-batches per optimizer step.

1. **Naïve layer-wise ($m = 1$):**

$$\text{Active fraction} = \frac{1}{p} = \frac{1}{8} = 12.5\%, \qquad \text{Idle fraction} = \frac{7}{8} = 87.5\%$$

2. **GPipe ($m = 16$):**

$$\text{Bubble-to-compute ratio} = \frac{p-1}{m} = \frac{8-1}{16} = \frac{7}{16} = 43.75\%$$

   Peak micro-batch activations held simultaneously on Stage $0 = m = 16$.

3. **1F1B ($m = 16$):**

$$\text{Bubble-to-compute ratio} = \frac{7}{16} = 43.75\%$$

   Peak micro-batch activations held simultaneously on Stage 0 drops to $p = 8$—a $2\times$ **reduction** in activation memory.

4. **Interleaved 1F1B ($v = 2$ virtual stages per GPU, $m = 16$):**

$$\text{Bubble-to-compute ratio} = \frac{p-1}{v \cdot m} = \frac{7}{2 \times 16} = \frac{7}{32} = 21.875\%$$

   Bubble overhead is cut in half, at the cost of $2\times$ point-to-point activation transfers across stage boundaries.

> **Interpretation**: Raising m from 1 to 16 transforms a mostly idle pipeline into a practical one; 1F1B halves peak activation memory; and adding $v = 2$ virtual stages halves the remaining bubble again.

KEY IDEA

- **Pipeline Parallelism is the low-bandwidth cross-rack axis**: It shards parameters by $1/p$ while transmitting only bsh point-to-point activations per micro-batch over RDMA.
- **Schedule engineering attacks memory and bubbles separately**: 1F1B cuts activation memory from $O(m)$ to $O(p)$, Interleaved 1F1B cuts the bubble to $\frac{p-1}{v \cdot m}$ at $v\times$ P2P traffic, and Zero-Bubble / DualPipeV eliminate idle gaps by decoupling $\nabla_x L$ (B) from $\nabla_W L$ (W) and overlapping bidirectional flows.

Pipeline Parallelism still requires $m \gg p$ micro-batches to remain efficient and leaves every individual layer's weights and single-micro-batch activations intact on one GPU. When a single layer is too wide, batch sizes are small, or bubbles are unacceptable, we must split the matrix multiplications *inside* each layer using **Tensor Parallelism**.

✎ Check Your Understanding

1. A $p = 4$ pipeline uses Interleaved 1F1B with $v = 2$ virtual stages per GPU and $m = 12$ micro-batches. Compute the bubble-to-compute ratio.

Answer: $\frac{p-1}{v \cdot m} = \frac{4-1}{2 \times 12} = \frac{3}{24} = 12.5\%$.

2. In Zero-Bubble pipelining, why can the weight-gradient computation W ($\nabla W = \nabla z \cdot x^\top$) be deferred into idle cooldown gaps while the input-gradient computation B ($\nabla x = W^\top \nabla z$) cannot?

Answer: The upstream pipeline stage is blocked waiting for ∇x to run its own backward pass, whereas no other stage depends on ∇W until the final optimizer step.

Section 5: Tensor Parallelism, Activation Memory, Sequence Parallelism, and Selective Recomputation

When you cannot afford pipeline bubbles—or when a single micro-batch's activations and layer weights exceed one GPU's HBM—you shard **width** instead of depth by splitting individual linear layers and attention heads across t GPUs.

DEFINITION

Tensor Parallelism (TP): An intra-layer model parallelism strategy that partitions weight matrices by columns and rows (and attention layers by heads) across t GPUs, synchronizing partial sums inside each Transformer block via All-Reduce.

Splitting Matrix Multiplications in MLPs and Self-Attention

A Transformer layer contains two compute-heavy sub-blocks: an MLP (two linear projections separated by an elementwise nonlinearity such as GeLU or SwiGLU) and Multi-Head Self-Attention. Pairing a **column-parallel** projection with a **row-parallel** projection eliminates any communication in the middle of the block.

Trace both sub-blocks across $t = 2$ GPUs (using standard matrix-block multiplication from linear algebra):

1. **MLP: Column-Parallel $A = [A_1, A_2]$ → Row-Parallel $B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$:**
   - The input activation X passes through an operator f to both GPUs.
   - **GPU 0** multiplies XA_1 and applies the elementwise activation $Y_1 = \mathrm{Act}(XA_1)$ (GeLU or SwiGLU). Because the activation is purely elementwise, **no synchronization is needed** after the first GEMM!
   - **GPU 1** independently computes $Y_2 = \mathrm{Act}(XA_2)$.
   - Next, the second matrix B is partitioned by rows into B_1 (on GPU 0) and B_2 (on GPU 1). By block matrix multiplication:

$$Z = YB = \begin{bmatrix} Y_1 & Y_2 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = Y_1 B_1 + Y_2 B_2$$

   - GPU 0 computes $Y_1 B_1$ and GPU 1 computes $Y_2 B_2$. Operator g sums these two partial products via **All-Reduce**, producing the exact full output Z on both GPUs.

2. **Self-Attention: Split Heads Across GPUs**:
   - Attention heads are mathematically independent until the final output projection W_O.
   - Partition the Q, K, V projection matrices **column-parallel** by head so GPU 0 computes (Q_1, K_1, V_1) for the first half of the heads and GPU 1 computes (Q_2, K_2, V_2) for the second half.
   - Each GPU runs full local attention (Attn_1 on GPU 0, Attn_2 on GPU 1) producing local head outputs O_1, O_2 with **zero communication**.
   - Partition W_O **row-parallel** into W_{O1} and W_{O2} so GPU 0 computes $O_1 W_{O1}$ and GPU 1 computes $O_2 W_{O2}$, then operator g runs an **All-Reduce** to produce $Z = O_1 W_{O1} + O_2 W_{O2}$.

Notice the clean **conjugate operators f and g** at the entry and exit of each sub-block:

- **Forward pass**: $f = \text{identity}$ (input X is already replicated on all TP ranks) and $g = \text{All-Reduce}$ (sum partial outputs).
- **Backward pass**: the two operators flip—$g = \text{identity}$ (upstream gradient ∇Z is already replicated, and differentiating $Y_1 B_1 + Y_2 B_2$ simply passes ∇Z to both branches) and $f = \text{All-Reduce}$ (gradients from both branches with respect to shared input X must be summed: $\nabla X = \nabla X_1 + \nabla X_2$).

Summarizing the block split across a Transformer layer:

- **Column-parallel**: QKV projections and MLP up-projection (and gate projection in SwiGLU).
- **Row-parallel**: Attention output projection W_O and MLP down-projection.
- **Replicated (under TP alone)**: LayerNorm/RMSNorm, Dropout, and MoE routers.

Tensor Parallelism Tradeoffs and Domain Limits

Compared to Pipeline Parallelism, TP has zero bubbles, works cleanly at small batch sizes, and wraps layers straightforwardly. Its weakness is **heavy per-layer communication**. Every Transformer layer performs 4 All-Reduces across forward and backward (2 in forward: one after Attention, one after MLP; 2 in backward: one before MLP, one before Attention) on activation tensors of size bsh in BF16 ($2bsh$ bytes). Since each All-Reduce moves $2\frac{t-1}{t}$ times the tensor byte size (Section 1), total TP communication per layer is:

$$\text{TP comm per layer (bytes)} = 4 \times 2\frac{t-1}{t} \times (2bsh) \quad \text{or in elements: } 8bsh\frac{t-1}{t} \text{ per layer}$$

(Specifically, the slides contrast **PP** at bsh P2P per micro-batch across stage boundaries against **TP** at $8bsh\frac{t-1}{t}$ All-Reduce **per layer**—note that 4 All-Reduces $\times 2\frac{t-1}{t} \times bsh = 8bsh\frac{t-1}{t}$ elements, or twice that in BF16 bytes). Because this All-Reduce sits on the critical path of every single layer, crossing onto a slow interconnect destroys throughput.

Visualize the **Historical 8-GPU Node Cliff** (Narayanan et al., 2021, on a 3B model across HGX A100 nodes with 8 NVLink GPUs per node). A bar chart plots **Tensor Parallel Size (TP)** on the horizontal axis ($2, 4, 8, 16, 32$) against **Tokens/sec/GPU** on the vertical axis:

- From $\text{TP} = 2$ to $\text{TP} = 4$, per-GPU throughput dips modestly by -10.8%.
- From $\text{TP} = 4$ to $\text{TP} = 8$, it dips another -12.2%.
- Immediately past $\text{TP} = 8$, a vertical red dashed line marks **Crosses 8-GPU NVLink boundary**.
- At $\text{TP} = 16$, throughput plunges by -42.7%; at $\text{TP} = 32$, it collapses by -65.6%.

That historical result spawned a folklore rule that "TP must never exceed 8." The **2026 rule** is more fundamental: **TP is bounded by the NVLink domain (and by keeping GEMM widths large enough to saturate Tensor Cores), not by the number 8.** On a 72-GPU NVL72 rack, NVLink no longer cuts off at 8, though as we will see in Section 6 and Section 9, shrinking GEMM width still keeps practical TP small (1–4) on modern MoEs.

The Other Half of Memory: Dynamic Activations

Sharding parameters with ZeRO-3, PP, or TP only solves half the memory equation. During the forward pass, PyTorch must save intermediate **activations** so the backward pass can compute weight and input gradients. At long sequence lengths or large micro-batches, stored activations easily exceed static parameter memory.

Korthikanti et al. (2022) derived the exact per-layer activation memory (in bytes, assuming BF16 activations and 1-byte dropout masks) for a standard Transformer layer with sequence length s, micro-batch size b, hidden size h, and a attention heads:

$$\text{Activations per layer (no parallelism)} = sbh\left(34 + 5\frac{as}{h}\right)$$

Decompose every piece of this formula:

- sbh is the number of elements in a single hidden-state tensor of shape (s, b, h).
- The factor 34 sums the linear-in-s activation buffers stored across LayerNorm, QKV/attention projections, MLP up/activation/down projections, and dropout masks.
- The second term $5\frac{as}{h}$ multiplies by sbh to give $5abs^2$ bytes—**quadratic in sequence length s**—coming from saving the $a \times s \times s$ attention score matrix, softmax output, and attention dropout mask.

Now apply **Tensor Parallelism of degree t** alone:

$$\text{Activations per layer (TP only)} = sbh\left(\mathbf{10} + \frac{24}{t} + 5\frac{as}{ht}\right)$$

Two critical bottlenecks jump out of this equation:

1. The **quadratic attention term** $5\frac{as}{ht}$ still blows up as s^2 when sequences grow to tens or hundreds of thousands of tokens. **FlashAttention / selective recomputation** eliminates this entire term by never materializing or storing the $s \times s$ attention matrices for backward, instead recomputing the cheap attention scores/softmax on the fly during backward.
2. The **stubborn $10sbh$ floor** does not divide by t at all! Those $10sbh$ bytes come from operations that remain **replicated** on every TP rank: LayerNorm ($4sbh$), Dropout ($2sbh$), and the replicated inputs to Attention and MLP ($4sbh$). Even as $t \to \infty$, activation memory under TP alone hits a brick wall at $10sbh$.

Sequence Parallelism and Selective Recomputation

How do we shard that stubborn $10sbh$ floor across the t TP ranks without adding any new communication? By applying the Section 1 collective identity ($\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$) directly to Tensor Parallelism!

DEFINITION

Sequence Parallelism (SP): A companion to Tensor Parallelism that splits the replicated pointwise regions of a Transformer layer (LayerNorm, Dropout, and residual connections) along the sequence dimension across the t TP ranks, replacing each TP All-Reduce with a Reduce-Scatter before the pointwise region and an All-Gather after it at identical communication volume.

Trace one complete Transformer layer under **TP + Sequence Parallelism**:

1. **Seq Parallel region** (activations sharded to length s/t on each GPU): **Norm** runs locally on each rank's s/t slice.
2. **Operator g = All-Gather** (forward): gathers the full sequence s onto all t ranks right before entering the column-parallel Attention block.
3. **Tensor Parallel region** (width sharded by t): **Attn** + **Proj** execute across the t ranks.
4. **Operator $\bar{g}$ = Reduce-Scatter** (forward): sums the row-parallel partial outputs and immediately scatters them along the sequence dimension so each rank receives only its s/t slice.
5. **Seq Parallel region**: **Drop + Residual Add** and the second **Norm** run locally on the s/t slice.
6. **Operator g = All-Gather**: gathers the full sequence s for the MLP.
7. **Tensor Parallel region**: **MLP (Col $\rightarrow$ Row)** executes across the t ranks.
8. **Operator $\bar{g}$ = Reduce-Scatter**: sums and scatters the MLP output back to s/t for the final **Drop + Residual Add**.

In the backward pass, g and $\bar{g}$ reverse roles ($g = \text{Reduce-Scatter}$, $\bar{g} = \text{All-Gather}$). Every pair of $(\text{Reduce-Scatter}, \text{All-Gather})$ moves the exact same byte volume as one TP All-Reduce—**zero extra bandwidth**—while dividing the $10sbh$ floor by t!

Putting TP, Sequence Parallelism, and Selective Recomputation together yields a clean progression:

Configuration	Activations / Layer (bytes)	Scaling Behavior
No parallelism	$sbh\left(34 + 5\frac{as}{h}\right)$	None; quadratic in s
TP only	$sbh\left(10 + \frac{24}{t} + 5\frac{as}{ht}\right)$	Hits $10sbh$ floor
TP + Sequence Parallel (SP)	$sbh\left(\frac{34}{t} + 5\frac{as}{ht}\right)$	Shards norms and dropout
TP + SP + Selective Recomputation	$\mathbf{sbh\left(\frac{34}{t}\right)}$	**True linear scaling ($1/t$)**

Selective recomputation often **pays for itself in wall-clock speed**: although recomputing attention softmax during backward adds a small amount of extra FLOPs, the massive HBM freed unlocks larger micro-batches and higher net GPU throughput.

COMMON MISCONCEPTION

You might think that Sequence Parallelism (SP) adds a new parallelism dimension requiring extra GPUs beyond TP. That breaks because SP reuses the **exact same t GPUs** as Tensor Parallelism: every rank simply alternates between being sharded along hidden width (inside GEMMs) and sharded along sequence length (inside norms and dropout), using the two halves of the existing TP All-Reduce to switch layouts.

WORKED EXAMPLE

Activation Memory Reduction Across TP, SP, and Selective Recomputation

Consider an 80-layer Transformer with micro-batch $b = 1$, sequence length $s = 8192$, hidden size $h = 8192$, and $a = 64$ heads, parallelized across $t = 4$ TP GPUs.

1. **Compute base tensor unit sbh and quadratic factor $5as/h$:**

$$sbh = 8192 \times 1 \times 8192 = 2^{26} \text{ bytes} = 64 \text{ MiB} = \frac{1}{16} \text{ GiB}$$

$$5\frac{as}{h} = 5 \times \frac{64 \times 8192}{8192} = 320$$

2. **No parallelism**:

$$\text{Per layer} = \frac{1}{16} \text{ GiB} \times (34 + 320) = \frac{354}{16} = 22.125 \text{ GiB} \implies 80 \text{ layers} = 1{,}770 \text{ GiB}$$

3. **TP only ($t = 4$)**:

$$\text{Per layer} = \frac{1}{16} \text{ GiB} \times \left(10 + \frac{24}{4} + \frac{320}{4}\right) = \frac{96}{16} = 6.0 \text{ GiB} \implies 80 \text{ layers} = 480 \text{ GiB}$$

   Notice that even if we dropped the 80 quadratic term with selective recomputation, TP alone would remain stuck at $(10 + 6)/16 = 1.0$ GiB/layer (80 GiB), and even at $t = 64$ it could never drop below the $10sbh$ floor (50 GiB).

4. **TP + SP ($t = 4$)**:

$$\text{Per layer} = \frac{1}{16} \text{ GiB} \times \left(\frac{34}{4} + \frac{320}{4}\right) = \frac{88.5}{16} = 5.53125 \text{ GiB} \implies 80 \text{ layers} = 442.5 \text{ GiB}$$

5. **TP + SP + Selective Recomputation ($t = 4$)**:

$$\text{Per layer} = \frac{1}{16} \text{ GiB} \times \left(\frac{34}{4}\right) = \frac{8.5}{16} = 0.53125 \text{ GiB} \implies 80 \text{ layers} = 42.5 \text{ GiB}$$

Interpretation: Combining TP=4, SP, and selective recomputation slashes activation memory across 80 layers from an impossible $1{,}770$ GiB down to 42.5 GiB—a $41.6\times$ **reduction**—leaving ample room for weights and larger micro-batches inside a 288 GB B300.

KEY IDEA

- **TP shards matrix multiplies inside a layer at high per-layer bandwidth**: Pairing column-parallel with row-parallel GEMMs needs 4 All-Reduces per layer ($8bsh\frac{t-1}{t}$), confining TP strictly inside the NVLink domain.
- **TP + SP + selective recomputation achieves true $1/t$ activation scaling**: Splitting TP's All-Reduce into Reduce-Scatter and All-Gather shards the stubborn $10sbh$ norm/dropout/input floor for free, while selective recomputation removes the quadratic $5\frac{as}{ht}$ attention term.

Dense MLPs shard cleanly with TP and SP, but 2026 frontier models—such as GLM-5.3 (256 experts), DeepSeek-V4.1-Flash (384 routed experts), and Kimi K3 (896 routed experts)—replace dense MLPs with sparse Mixture-of-Experts (MoE) layers. Slicing hundreds of already-narrow experts across GPUs with TP destroys Tensor Core utilization, which forces us to introduce **Expert Parallelism**.

✎ Check Your Understanding

1. In Tensor Parallelism for an MLP block ($A = [A_1, A_2]$ followed by GeLU/SwiGLU and $B = [B_1; B_2]$), why is no collective communication required between the first GEMM and the activation function?

Answer: The first GEMM is partitioned column-parallel, producing disjoint subsets of output channels on each GPU; because GeLU/SwiGLU operates elementwise, each GPU applies it locally to its own output channels with no synchronization.

2. Fill in the blank: Under TP alone, activation memory per layer hits a non-sharded floor of _____ bytes coming from LayerNorm, Dropout, and replicated Attention/MLP inputs; adding Sequence Parallelism divides this floor by t at zero extra communication volume.

Answer: $10sbh$.

Section 6: Expert Parallelism, MoE Parallel Folding, and Fused Token Dispatch

In a sparse Mixture-of-Experts (MoE) layer, each token activates only a small subset of experts (for example, 6 active out of 384 routed experts in DeepSeek-V4.1-Flash, or 16 active out of 896 in Kimi K3). Rather than slicing every single expert matrix across all GPUs with Tensor Parallelism, **Expert Parallelism (EP)** keeps each expert's GEMM matrices intact and assigns different subsets of experts to different GPUs.

> **DEFINITION**
> **Expert Parallelism (EP)**: A model parallelism strategy for MoE layers that places disjoint subsets of intact expert feed-forward networks on different ranks and uses All-to-All collectives to route tokens to their assigned experts and return the outputs.

Trace the data flow of an MoE layer across EP ranks (say Rank 0 holding Experts $0 \dots k-1$ through Rank $E-1$ holding the last k experts):

1. **Router (FP32)**: On each rank, a lightweight router scores every local token against all experts and selects the top-K experts for each token.
2. **All-to-All Dispatch**: Every rank exchanges tokens with every other rank so that each token lands on the GPU owning its chosen expert(s).
3. **Local Expert FFNs**: Each rank runs the intact feed-forward GEMMs for its local experts on the incoming batch of tokens.
4. **All-to-All Combine**: Each rank sends the computed expert outputs back to the tokens' originating ranks, where they are weighted by router probabilities and summed.

This two-collective sequence repeats in the backward pass, totaling **4 All-to-All collectives per MoE layer** (dispatch and combine in forward; combine-grad and dispatch-grad in backward).

Why does **EP beat TP** for MoE experts?

1. **Larger local GEMMs**: Keeping each expert intact preserves matrix dimensions, achieving much higher Tensor Core utilization than slicing narrow experts across TP ranks.
2. **Lower communication**: Routing only the active tokens to their assigned experts moves far fewer bytes than All-Reducing every expert's output across TP.
3. **Cleaner overlap graph**: The token-routing All-to-Alls decouple cleanly from surrounding attention and backward compute for hiding communication.
4. **Zero local token permutation at full EP**: When $\mathrm{EP} = \mathrm{num_experts}$ (one expert per GPU), each rank no longer needs to permute and unpermute tokens among multiple local experts.

As a historical check, even on the relatively small 8-expert Mixtral $8 \times 7\mathrm{B}$, running `EP8×TP1` outperformed `EP4×TP2`.

Decoupling Attention and Experts: MoE Parallel Folding

MoE sparsity applies only to MLP layers, **not** to attention layers. That creates an architectural clash: attention wants Tensor Parallelism and Context Parallelism (especially at $128\mathrm{K}$–$1\mathrm{M}$ token contexts), whereas 256–896 experts want wide Expert Parallelism across an entire NVL72 rack.

In an **outdated coupled layout**, EP splits were carved directly out of the Data Parallelism (DP) group, imposing the hard constraint:

$$\mathrm{EP} \leq \mathrm{DP}$$

Look at what breaks when you enable long-context training: increasing CP shrinks DP (since total GPUs $= \text{TP} \times \text{CP} \times \text{DP} \times \text{PP}$), which immediately crushes the maximum allowable EP and makes it impossible to fit 256–896 experts into HBM!

Megatron Core solves this conflict with **MoE Parallel Folding**: the exact same pool of GPUs is organized into **two independent logical grids**—one for attention and one for experts:

- **Attention grid**: $\text{TP} \times \text{CP} \times \text{DP} \times \text{PP}$
- **Expert grid**: $\text{ETP} \times \text{EP} \times \text{EDP} \times \text{PP}$

> **DEFINITION**
> **MoE Parallel Folding**: A layout architecture (used in Megatron Core) that maps the same physical GPUs onto two decoupled parallelism grids—an attention grid ($\text{TP} \times \text{CP} \times \text{DP} \times \text{PP}$) and an expert grid ($\text{ETP} \times \text{EP} \times \text{EDP} \times \text{PP}$)—eliminating the coupled $\text{EP} \leq \text{DP}$ constraint.

Parallel folding is essential in 2026: frontier models such as GLM-5.3 (256 experts), DeepSeek-V4.1-Flash (384 routed experts), and Kimi K3 (896 routed experts) target 1M context windows. Folding lets attention keep TP and CP while experts set $\text{ETP} = 1$ and span wide EP across a full GB200/GB300 NVL72 rack.

Token Dispatchers: All-Gather, Standard All-to-All, and DeepEP / HybridEP

Implementing token dispatch efficiently across a two-tier NVLink/RDMA fabric has evolved through three dispatcher architectures:

Dispatcher	Mechanism	Best Regime / Tradeoff
All-Gather	Gather all tokens to all EP ranks, compute local experts, Reduce-Scatter	Simple and avoids variable-length sends, but wastes massive bandwidth at high EP
Standard All-to-All	Standard NCCL exchange of routed tokens only	Bandwidth-efficient, but incurs CPU kernel-launch overhead and poor two-tier NVLink/RDMA handling
DeepEP / HybridEP	Fused GPU-initiated kernels: **NVLink** inside domain, **RDMA** across domains	Standard for wide EP; minimizes cross-domain traffic via hierarchical combine

DeepEP V2 / V2.5 (late 2026) rebuilt the dispatcher around a single `EPBuffer` API and dropped NVSHMEM in favor of **NCCL's device API ("Gin" backend)**. It computes streaming multiprocessor (SM) and queue-pair counts analytically, but still occupies GPU SMs to drive communication. That introduces a real **SM tax**: Megatron Core's MoE paper reports that dedicating **20 SMs/GPU** to DeepEP costs roughly **20% GEMM efficiency**, motivating careful overlap and **HybridEP** tuning on Blackwell.

Hiding All-to-All, Static Shapes, and Router Stability

At wide EP, raw All-to-All dispatch and combine can consume 30%–40% of step time if left unhidden. Three systems techniques bring that cost under control:

1. **Overlapping All-to-All with compute**:
   - **1F1B / DualPipe-style overlap**: Run micro-batch A's forward compute while micro-batch B's backward pass executes its All-to-All token exchange (and vice versa). Megatron Core reports (vendor-reported) that 1F1B All-to-All overlap hides **93%** of expert communication on historical DeepSeek-V3, dropping its share of step time from **30–40% to under 5%**.
   - **Shared-expert overlap**: Models like DeepSeek-V4.1-Flash (1 shared expert) and Kimi K3 (2 shared experts) include always-active shared experts that require no routing; computing the shared expert concurrently during All-to-All dispatch directly masks dispatch latency.
2. **Static-shape MoE via online expert replication**:
   - Because routers send unpredictable numbers of tokens to each expert, dynamic token counts per rank create stragglers (every rank waits for the hottest expert) and **break CUDA graphs**, which require static tensor shapes.
   - **MoonEP, UltraEP, and DeepEP V2.5** replicate hot experts online—using redundant-expert prefetch and gradient reduction—so every rank processes an equal, static token count.
3. **Router stability**:
   - Keep router weights and logits strictly in **FP32**, and synchronize **expert-bias** load-balancing updates across ranks so routing stays balanced without diverging across replicas.

> **COMMON MISCONCEPTION**
>
> You might think that because DeepEP uses fused GPU-initiated kernels to bypass CPU launch overhead, its communication runs completely "for free" alongside GEMMs. That breaks because GPU-initiated communication kernels consume SMs—specifically around 20 SMs/GPU in Megatron Core's measurements—reducing concurrent GEMM efficiency by roughly 20% unless carefully tuned (as in HybridEP) and overlapped across micro-batches.

> **WORKED EXAMPLE**
>
> **Coupled Layout vs. MoE Parallel Folding on 64 GPUs**
>
> Consider training a 384-routed-expert MoE (DeepSeek-V4.1-Flash scale) on $N = 64$ GPUs inside a single NVL72 rack, with $\mathrm{TP} = 2$ and $\mathrm{PP} = 1$. Trace what happens as Context Parallelism increases from $\mathrm{CP} = 1$ (short context) to $\mathrm{CP} = 4$ (long context):
>
> 1. **Short context ($\mathrm{CP} = 1$)**:
>
> $$\mathrm{DP} = \frac{64}{\mathrm{TP} \times \mathrm{CP} \times \mathrm{PP}} = \frac{64}{2 \times 1 \times 1} = 32$$
>
> Under a coupled layout ($\mathrm{EP} \leq \mathrm{DP}$), maximum $\mathrm{EP} = 32$, placing $384/32 = 12$ routed experts per GPU.

2. **Long context ($\mathrm{CP} = 4$) — Coupled layout fails**:

$$\mathrm{DP} = \frac{64}{2 \times 4 \times 1} = 8$$

   Because $\mathrm{EP} \leq \mathrm{DP}$, EP collapses to $\mathrm{EP} = 8$, forcing each GPU to hold $384/8 = 48$ routed experts—a $4\times$ **spike in expert parameter memory** right when long-context activations already pressure HBM!
3. **Long context ($\mathrm{CP} = 4$) — MoE Parallel Folding succeeds**:
   - **Attention grid**: $\mathrm{TP} = 2 \times \mathrm{CP} = 4 \times \mathrm{DP} = 8 = 64$ GPUs.
   - **Expert grid**: $\mathrm{ETP} = 1 \times \mathrm{EP} = 64 \times \mathrm{EDP} = 1 = 64$ GPUs. Each GPU holds only $384/64 = 6$ intact routed experts—an $8\times$ **reduction** in per-GPU expert weights compared to the coupled layout—and 100% of All-to-All traffic stays on the $900\ \mathrm{GB/s}$ NVLink 5 fabric.

Interpretation: MoE Parallel Folding completely decouples long-context attention scaling from expert sharding, letting wide EP stay locked to the full 64-GPU NVLink domain even when CP shrinks DP.

KEY IDEA

- **EP preserves GEMM efficiency while routing tokens**: Assigning intact experts to disjoint ranks via 4 All-to-Alls per MoE layer outperforms slicing experts with TP, and MoE Parallel Folding removes the $\mathrm{EP} \leq \mathrm{DP}$ wall by decoupling attention and expert grids.
- **Production wide-EP depends on fused kernels, overlap, and static shapes**: DeepEP/HybridEP minimize cross-domain traffic (subject to an SM tax), 1F1B and shared-expert overlap hide $> 90\%$ of All-to-All latency, and online hot-expert replication restores static shapes for CUDA graphs.

We have now seen how CP shrinks DP when training on long sequences, motivating Parallel Folding—but how does **Context Parallelism** itself actually parallelize attention across GPUs when sequences reach $128\mathrm{K}$ to $1\mathrm{M}$ tokens?

✎ Check Your Understanding

1. How many All-to-All collectives does a single MoE layer execute per training iteration (forward plus backward)?

Answer: Four All-to-All collectives per MoE layer: Dispatch and Combine in the forward pass, and their two corresponding gradient exchanges in the backward pass.

2. Why do dynamic per-rank token counts hurt MoE training efficiency, and how do MoonEP, UltraEP, and DeepEP V2.5 fix this?

Answer: Uneven dynamic token counts create straggler ranks and break static-shape CUDA graphs; MoonEP, UltraEP, and DeepEP V2.5 replicate hot experts online so every rank processes an equal, static token count.

Section 7: Context Parallelism and Hybrid Long-Context Architectures

Even with TP + SP + selective recomputation (Section 5), sharding sequence activations only by the small TP degree ($t = 1$ to 4) cannot fit 128K to 1M-token sequences in HBM. **Context Parallelism (CP)** introduces a dedicated sequence-sharding axis that partitions both sequence activations and the KV cache across CP ranks, cutting long-context memory by $1/\text{CP}$. Frontier long-context stages rely heavily on CP: Nemotron 3 Super uses $\text{CP} = 64$, while Kimi K3 and DeepSeek-V4.1-Flash target 1M-token contexts.

DEFINITION

Context Parallelism (CP): A parallelism dimension that partitions input sequences, activations, and KV caches along the sequence length s across CP GPUs, cutting long-context activation and KV memory by $1/\text{CP}$.

Because every query token in standard self-attention must attend to key and value tokens located on other GPUs, there are two fundamentally different ways to execute attention across CP ranks: **Ring Attention** and **Ulysses**.

Ring Attention vs. Ulysses

Compare the two mechanisms side by side across a group of CP GPUs:

1. **Ring Attention (Pass KV Chunks)**:
   - Each GPU i permanently owns its local query shard Q_i (of length s/CP) and starts with its local key-value shard (K_i, V_i).

- Imagine GPUs $0, 1, 2$ connected in a directed ring ($0 \to 1 \to 2 \to 0$). In step j, each GPU computes local FlashAttention between its stationary Q_i and the currently resident KV chunk (K_j, V_j), updating its running online softmax accumulators.
- **Simultaneously**, each GPU sends its current (K_j, V_j) chunk to the next neighbor in the ring and receives the next KV chunk from its predecessor.
- After CP ring steps, every query shard has attended to every KV shard, overlapping KV transfers directly behind local FlashAttention compute.
- **Key strengths**: Ring Attention has **no head-count ceiling** and works seamlessly with Grouped-Query Attention (GQA) and Multi-Head Latent Attention (MLA).

2. **Ulysses (All-to-All Over Heads)**:
   - Before attention, every GPU holds a **sequence-sharded** tensor of shape $(s/\mathrm{CP}, H)$, containing all H attention heads for its $1/\mathrm{CP}$ sequence slice.
   - Ulysses executes an **All-to-All transpose** across the CP ranks, transforming the layout from sequence-sharded $(s/\mathrm{CP}, H)$ into **head-sharded** $(s, H/\mathrm{CP})$—every GPU now holds the **entire sequence of length** s for its assigned subset of H/CP heads.
   - Each GPU runs standard local full-sequence attention on its heads (no cross-GPU attention protocol required!), and then runs a second **All-to-All transpose** back from $(s, H/\mathrm{CP})$ to $(s/\mathrm{CP}, H)$.
   - **Key strengths**: Low communication volume per step and pairs cleanly with **sparse attention** kernels (since each GPU sees the entire sequence for its heads).
   - **Hard ceiling**: Ulysses is bounded by the KV head count—$\mathrm{CP} \leq H_{\mathrm{KV}}$—because pushing $\mathrm{CP} > H_{\mathrm{KV}}$ forces you to duplicate KV heads across multiple GPUs, wasting memory and bandwidth.

Hierarchical Context Parallelism and Late-2026 Hybrid Architectures

Just as MoE dispatch matches the two-tier cluster interconnect, **Hierarchical Context Parallelism** combines Ulysses and Ring Attention across the two network tiers:

- Run **Ulysses All-to-All** inside the high-bandwidth **NVLink domain** (an NVL72 rack).
- Run **Ring Attention** across NVLink domains over lower-bandwidth **RDMA**, where ring point-to-point KV transfers overlap cleanly with FlashAttention compute.

At the architectural level, late-2026 frontier models reduce long-context communication even further by mixing **linear-time (recurrent/state-space) attention** with periodic full or sparse attention layers:

- **Kimi K3** (July 2026, $1\mathrm{M}$ context): 3 **Kimi Delta Attention** layers plus 1 **gated MLA** layer per block.
- **Qwen3.8-Flash-Next** (Aug 2026): $3{:}1$ ratio of **Gated DeltaNet** layers to **Qwen Sparse Attention (QSA)** layers.
- **Nemotron 3.5 Super / Lightning** (Oct 2026): hybrid **Mamba-Transformer MoE**.

How do linear-attention and Mamba layers split long contexts across CP ranks? Because they compress past tokens into a fixed-size recurrent state rather than attending over a growing KV cache, **no quadratic KV ring is needed**: they split either **by head** or **by sequence chunk**, passing only compact recurrent

states across CP ranks. This concentrates KV-ring traffic solely onto the periodic full or sparse attention layers, shrinking overall long-context communication dramatically.

COMMON MISCONCEPTION

You might think that Ulysses can always scale to $\mathrm{CP} = 64$ as long as the model has 64 query heads. That breaks under Grouped-Query Attention (GQA) or Multi-Query Attention, where 64 query heads share far fewer KV heads ($H_{\mathrm{KV}} \ll H_Q$). Splitting across more ranks than H_{KV} forces the same KV head to be copied across multiple GPUs, eroding Ulysses's memory and communication advantage; exceeding H_{KV} cleanly requires Ring Attention or Hierarchical CP.

WORKED EXAMPLE

Choosing Ring vs. Ulysses vs. Hierarchical CP at $\mathrm{CP} = 64$

Compare two architectures running a long-context stage with $\mathrm{CP} = 64$:

1. **Case A — DeepSeek-V4.1-Flash (1 KV head shared by 64 query heads)**: Here $H_{\mathrm{KV}} = 1$. Attempting pure Ulysses at $\mathrm{CP} = 64$ violates $\mathrm{CP} \le H_{\mathrm{KV}}$ by a factor of 64, forcing all 64 ranks to replicate the single KV head. Instead, **Ring Attention** (which has no head-count ceiling) circulates KV chunks around the ring and overlaps each hop with local attention compute.
2. **Case B — A GQA model with $H_{\mathrm{KV}} = 8$ KV heads spanning 8 NVLink domains**: Pure Ulysses is capped at $\mathrm{CP}_{\mathrm{Ulysses}} \le H_{\mathrm{KV}} = 8$. With **Hierarchical CP**, we set:

   $$\mathrm{CP}_{\mathrm{Ulysses}} = 8 \text{ (inside each NVLink domain)}, \qquad \mathrm{CP}_{\mathrm{Ring}} = 8 \text{ (across domains over}$$

   Total context parallelism is $8 \times 8 = 64$: every KV head sits on exactly one rank inside each domain (zero KV duplication), high-speed NVLink handles the Ulysses All-to-All transpose, and RDMA carries only overlapped Ring KV hops.

Interpretation: Matching CP algorithms to both H_{KV} and the NVLink/RDMA hierarchy achieves $\mathrm{CP} = 64$ without redundant KV copies or cross-rack All-to-All bottlenecks.

KEY IDEA

- **Ring Attention and Ulysses trade head-count limits against sparse-attention simplicity**: Ring Attention circulates KV blocks with no head-count ceiling (ideal for GQA/MLA and cross-domain RDMA), while Ulysses transposes sequence-sharded tensors to head-sharded tensors via All-to-All (ideal inside NVLink and with sparse attention, up to $\mathrm{CP} \le H_{\mathrm{KV}}$).
- **Hierarchical CP and hybrid linear/sparse architectures dominate 1M-context training**: Combining intra-domain Ulysses with inter-domain Ring—and passing only compact recurrent states across linear-attention/Mamba layers—drastically cuts long-context communication.

We now have all six parallelism dimensions (DDP/ZeRO, FSDP, PP, TP+SP, EP, CP). Yet implementing any of them naïvely leaves Tensor Cores sitting idle during network transfers—or breaks completely when we swap AdamW for modern matrix-orthogonalizing optimizers. Next, we unify the systems rules for

communication–computation overlap and **optimizer-aware parallelism**.

> **✎ Check Your Understanding**
>
> 1. Fill in the blank: In Ulysses Context Parallelism, an All-to-All collective transposes tensors from shape $(s/\mathrm{CP}, H)$ to shape _____, allowing each GPU to run local full-sequence attention on its assigned heads.
>
> **Answer:** $(s, H/\mathrm{CP})$.
>
> 2. Why do hybrid models such as Kimi K3 (3 Kimi Delta Attention + 1 gated MLA) or Nemotron 3.5 Super (hybrid Mamba-Transformer) incur far less Context Parallelism communication than pure full-attention Transformers?
>
> **Answer:** Their linear-attention or Mamba layers do not require a quadratic KV ring; they split by head or sequence chunk and pass only compact recurrent states across CP ranks, confining KV-ring traffic to the periodic full/sparse attention layers.

Section 8: Universal Communication–Computation Overlap and Optimizer-Aware Parallelism (Distributed Muon)

Prefetching weights in FSDP (Section 2) is not a special-case trick—it is one instance of a universal distributed systems rule: **never leave Tensor Cores idle waiting on the network**. Whenever communication runs sequentially after computation, total step time is the sum:

$$T_{\text{no-overlap}} = T_{\text{compute}} + T_{\text{comm}}$$

When communication executes asynchronously on a separate stream concurrently with independent compute, effective step time collapses to the maximum of the two:

$$T_{\text{overlapped}} = \max(T_{\text{compute}}, T_{\text{comm}})$$

Every single parallelism dimension has a specific compute partner designed to hide its communication:

Parallelism	Communication	Compute That Overlaps It
DDP	Gradient bucket All-Reduce	Earlier layers' backward pass (each bucket fires as soon as its gradients are ready)
FSDP / Dist. Opt	Parameter All-Gather & gradient Reduce-Scatter	Next layer's forward pass & previous layer's backward pass
Tensor Parallel	Activation All-Gather / Reduce-Scatter	Chunked producer/consumer GEMMs along sequence or batch tiles
Pipeline Parallel	Stage-boundary P2P send/recv	Current micro-batch forward/backward on adjacent stages
Expert Parallel	All-to-All Dispatch & Combine	Another micro-batch (1F1B / DualPipe) + always-active shared-expert FFN
Context Parallel	Ring KV send/recv	Local FlashAttention block on the previously arrived KV chunk

As long as $T_{\text{compute}} \geq T_{\text{comm}}$, the communication overhead disappears almost completely from wall-clock step time.

Optimizers Meet Parallelism: Distributed Muon

Until recently, distributed optimizer design assumed every parameter update was purely elementwise, as in AdamW. That assumption breaks with **Muon** (which **Lecture 11** will introduce in full mathematical detail), a matrix-aware optimizer widely adopted in 2026 frontier training—including **DeepSeek-V4** (Muon) and **Kimi K3** (Per-Head Muon).

Why does flat ZeRO/FSDP sharding break for Muon?

- **AdamW is elementwise**: Updating entry W_{ij} depends only on W_{ij}, its scalar gradient g_{ij}, and its scalar moments m_{ij}, v_{ij}. Consequently, ZeRO/FSDP can flatten all model parameters into a single 1D vector and slice it arbitrarily across N_d DP ranks, even cutting straight through the middle of weight matrices.
- **Muon runs Newton–Schulz orthogonalization on whole 2D weight matrices**: Orthogonalizing a matrix couples every row and every column together. You cannot slice a 2D weight matrix into flat 1D shards across DP ranks and orthogonalize each shard independently—the result would not be the orthogonalization of the full matrix.

DEFINITION

Layer-wise distributed optimizer: A ZeRO-style distributed optimizer that partitions parameters at the granularity of whole 2D weight matrices (or full layers) rather than flat 1D slices, enabling matrix-level optimizers like Muon to run Newton–Schulz orthogonalization locally on each rank before All-Gathering updated weights.

The **layer-wise distributed optimizer** resolves this cleanly:

1. Assign **whole parameter matrices / layers** to each DP rank (balancing total bytes across ranks).
2. During backward, Reduce-Scatter each matrix's gradient onto the single DP rank that owns that whole matrix.
3. Each DP rank runs Newton–Schulz orthogonalization locally on its intact 2D matrices, updates them, and then **All-Gathers** the updated weights across DP ranks.
4. When Tensor Parallelism also shards a 2D weight matrix across TP ranks, Megatron Core's layer-wise distributed optimizer provides **three TP modes** to handle Newton–Schulz orthogonalization on TP-sharded weights.

In practice, distributed Muon runs with negligible—or even negative—throughput penalty relative to AdamW: on 256 GB300 GPUs in MXFP8, NVIDIA reports (vendor-reported) historical **Kimi K2** achieving **1,080 TFLOP/s/GPU** with Muon versus **1,051 TFLOP/s/GPU** with AdamW.

COMMON MISCONCEPTION

You might think that communication–computation overlap always eliminates 100% of communication time regardless of network speed. That breaks whenever $T_{\text{comm}} > T_{\text{compute}}$: overlap only reduces step time from $T_{\text{compute}} + T_{\text{comm}}$ to $\max(T_{\text{compute}}, T_{\text{comm}})$. If communication takes longer than compute, the excess $T_{\text{comm}} - T_{\text{compute}}$ remains exposed as idle Tensor Core stalls.

WORKED EXAMPLE

Quantifying Overlap Speedup and Flat-ZeRO Failure for Muon

Part 1 (Overlap): Suppose a layer's compute takes $T_{\text{compute}} = 10$ ms.

- **Compute-bound case ($T_{\text{comm}} = 3.5$ ms)**: Without overlap, step time is $10 + 3.5 = 13.5$ ms. With full overlap, step time drops to $\max(10, 3.5) = 10$ ms—a $1.35\times$ speedup that hides 100% of communication.
- **Communication-bound case ($T_{\text{comm}} = 12$ ms)**: Without overlap, step time is $10 + 12 = 22$ ms. With full overlap, step time drops to $\max(10, 12) = 12$ ms—still a $1.83\times$ speedup, but 2 ms of exposed network stall remain because compute is too short to cover T_{comm}.

Part 2 (Layer-wise sharding for Muon): Consider 8 identical linear layers, each with a 4096×4096 weight matrix ($16{,}777{,}216$ parameters per matrix), trained across $N_d = 8$ DP ranks.

- **Flat 1D ZeRO** slices every 4096×4096 matrix into 8 chunks of 512×4096 rows across the 8 ranks. Running Newton–Schulz independently on eight 512×4096 submatrices cannot capture singular directions spanning all 4096 rows.
- **Layer-wise distributed optimizer** assigns Layer k's entire 4096×4096 matrix to Rank k ($k = 0, \ldots, 7$). Each rank stores the exact same $1/8$ total optimizer memory ($16{,}777{,}216$ parameters), runs exact 2D Newton–Schulz locally, and All-Gathers the updated matrices.

Interpretation: Both overlap and optimizer sharding require respecting the algorithm's dependency structure—overlapping independent streams to reach $\max(T_{\text{compute}}, T_{\text{comm}})$ and aligning shard boundaries to whole 2D matrices for Muon.

KEY IDEA

- **Overlap turns addition into a maximum**: Pairing every parallelism axis's collective with concurrent independent compute cuts effective step time from $T_{\text{compute}} + T_{\text{comm}}$ to $\max(T_{\text{compute}}, T_{\text{comm}})$.
- **Matrix-aware optimizers require matrix-aligned sharding**: Because Muon runs Newton–Schulz orthogonalization on whole 2D weight matrices, flat 1D ZeRO breaks and must be replaced by a layer-wise distributed optimizer (plus TP-aware modes when weights are TP-sharded).

We now understand every individual parallelism mechanism, its overlap partner, and its optimizer constraints. No single dimension solves static memory, long sequences, and compute scaling simultaneously—so next we synthesize all six dimensions into a systematic procedure for designing 5D parallelism layouts on real clusters.

✎ Check Your Understanding

1. A training step has $T_{\text{compute}} = 8$ ms and $T_{\text{comm}} = 6$ ms. Compute the step time without overlap and with full overlap.

Answer: Without overlap, $T_{\text{compute}} + T_{\text{comm}} = 14$ ms; with full overlap, $\max(8, 6) = 8$ ms.

2. Yes or no with justification: Can standard flat 1D ZeRO-1 run Muon's Newton–Schulz orthogonalization locally on each DP rank without extra communication?

Answer: No. Flat 1D ZeRO cuts across 2D weight matrices, whereas Newton–Schulz orthogonalization operates on whole 2D matrices; a layer-wise distributed optimizer must assign whole matrices to each DP rank instead.

Section 9: Composing 5D Parallelism — Guidelines, Real-World Layouts, and Worked Cluster Designs

Before composing a multi-dimensional layout, place all six parallelism dimensions side by side to see what each one scales—and what it costs:

Method	Comm / Sync	Param Mem / Rank	Act / KV Mem / Rank	Main Bandwidth Cost	Scales Batch?	Complexity
DDP / ZeRO-1	Grad RS + Param AG (or AR)	No param split (ZeRO-1 shards opt)	None	2Ψ / step	**Yes ($\propto$ DP)**	Low
FSDP / ZeRO-3	Per-layer AG + RS (overlapped)	$\sim 1/\mathrm{DP}$ (all static state)	None	3Ψ / step	**Yes ($\propto$ DP)**	Moderate
Pipeline (PP)	Stage P2P; bubbles	$\sim 1/\mathrm{PP}$	Schedule-dependent ($O(p)$ in 1F1B)	bsh P2P between stages	Needs large m	High
Tensor + SP	Per-layer AG + RS (blocking if unoverlapped)	$\sim 1/\mathrm{TP}$	$\sim 1/\mathrm{TP}$ (with SP + recompute)	$8bsh\frac{t-1}{t}$ / layer	No	High
Context (CP)	Ring KV or head A2A	None	$\sim 1/\mathrm{CP}$ (seq acts & KV)	KV or head activations / layer	No	High
Expert (EP)	$4\times$ A2A / MoE layer (dispatch + combine)	$\sim 1/\mathrm{EP}$ (experts only)	None (attn); scales expert MLP	Token-routing All-to-All	Needs tokens/exp	High

No single row solves static parameter memory, dynamic activation/KV memory, and cluster compute throughput simultaneously. Frontier training runs compose **4 to 5 dimensions** together.

Megatron Core Composition Guidelines

The single regime variable governing when to add model parallelism is the **batch size per GPU (B/N)**. Pure FSDP works well when B/N is large enough that layer compute hides weight All-Gathers ($T_{\text{compute}} \geq T_{\text{comm}}$). At small B/N (large clusters under a fixed critical global batch size $\mathcal{B}$), pure FSDP becomes communication-bound; introducing model parallelism (TP, PP, EP) raises the batch size per model replica and keeps compute above communication.

Megatron Core distills 5D composition into five concrete engineering rules:

1. **Minimize model parallelism, maximize DP**: Shard optimizer state first (distributed optimizer / ZeRO-1/2), adding TP, PP, and EP only as needed to fit memory and hit the target global batch size.
2. **Keep bandwidth-heavy axes inside the NVLink domain**: Place TP, EP, and Ulysses CP inside the 72-GPU NVL72 rack ($900\ \mathrm{GB/s}$ unidirectional).
3. **Use PP (plus virtual stages) across domains**: Low-volume bsh point-to-point stage transfers fit $50\ \mathrm{GB/s}$ inter-rack RDMA links.
4. **Prefer EP over TP for MoE layers**: Decouple attention and experts via **MoE Parallel Folding** so experts scale to wide EP with $\mathrm{ETP} = 1$.
5. **Enable CP for long contexts** ($\geq 8\mathrm{K}$ up to $1\mathrm{M}$ tokens), and **order process groups from fastest to slowest link**:

   ```
   [TP, CP, PP, DP]
   ```

   When global batch size is small relative to cluster size, use **gradient accumulation** across multiple micro-batches to amortize DP communication and fill pipeline stages.

Real-World Training Layouts: Historical Baselines to 2026

Tracing published cluster layouts across hardware generations reveals how expanding the NVLink domain from 8 GPUs to 72 GPUs reshaped parallelism choices:

Model (Era)	Hardware	TP	PP	CP	EP	DP / Notes
Llama 3 405B (historical, 2024, dense)	16,384 H100	8	16	$1 \rightarrow 16$	–	DP $128 \rightarrow 8$; $8\text{K} \rightarrow 131\text{K}$ ctx; 38–43% BF16 MFU
DeepSeek-V3 (historical, Dec 2024, 671B)	H800 (8-GPU nodes)	1	16	1	64	ZeRO-1; EP across 8 nodes + 1F1B A2A overlap
Qwen3 235B-A22B (historical, May 2025)	512 GPUs	2	8	1	32	Qwen3 30B-A3B: TP1/PP1/EP8 (single 8-GPU node)
Nemotron 3 Super (early 2026, LC)	GB200 NVL72	2	–	64	64	120B-A12B long-context stage; folded CP64 & EP64
Wide-EP Blackwell shape (2026)	GB200/GB300 NVL72	1	4	var	64	EP64 inside 1 NVL72 rack via HybridEP; PP across racks

Two clear **2026 architectural shifts** stand out in this table:

- **Wide rack-local EP with minimal TP**: 72-GPU NVLink domains let **EP expand to 64 inside a single rack** while **TP shrinks to 1–2**, preserving larger GEMMs and eliminating blocking per-layer TP syncs.
- **DP-to-CP trading for long context**: Long-context training stages keep the total GPU count fixed and trade DP directly for large CP (16–64).

> **COMMON MISCONCEPTION**
>
> You might think that trading DP for CP during a long-context training stage (for instance, moving from `DP=16, CP=1` to `DP=4, CP=4` on the same cluster) increases static optimizer memory per GPU by $4\times$ because DP shrank by $4\times$. That breaks in Megatron Core because the distributed optimizer shards optimizer states across the **entire data-and-context replica group** $\text{DP} \times \text{CP}$, not just DP—since all CP ranks in a group hold identical copies of the model weights. Static memory per GPU stays completely unchanged!

Worked Cluster Layouts

We now apply the five Megatron Core guidelines end-to-end to two concrete cluster designs: a dense 70B model and a 552B–2.8T frontier MoE.

WORKED EXAMPLE

Worked Layout 1: Dense 70B Transformer on 64 Blackwell GPUs

Setup: Train a 70B-parameter dense Transformer at $12\ \text{B/param}$ ($2\ \text{B}$ params + $2\ \text{B}$ grads + $8\ \text{B}$ optimizer state) on 64 Blackwell GPUs (GB200 $192\ \text{GB}$ HBM or B300 $288\ \text{GB}$ HBM), using Sequence Parallelism (SP) and Megatron Core's distributed optimizer (ZeRO-1 semantics).

1. **Short-context stage (`TP=4, PP=1, CP=1, DP=16`)**:
   - Check GPU count: $4 \times 1 \times 1 \times 16 = 64$ GPUs.
   - Parameters per GPU: $70\text{B}/\text{TP} = 70\text{B}/4 = 17.5\text{B}$.
   - Optimizer shard factor: $\text{DP} \times \text{CP} = 16 \times 1 = 16$.
   - Static memory per GPU:

$$17.5\text{B} \times \left(2 + 2 + \frac{8}{16}\right)\ \text{B/param} = 17.5\text{B} \times 4.5\ \text{B/param} = \mathbf{78.75}\ \text{GB}$$

   - **Justification**: `TP=4` stays comfortably inside the NVLink domain and cuts activations $4\times$ (with SP + selective recomputation); `PP=1` eliminates pipeline bubbles entirely; `DP=16` maximizes batch throughput.

2. **Long-context stage (`TP=4, PP=1, CP=4, DP=4`)**:
   - Check GPU count: $4 \times 1 \times 4 \times 4 = 64$ GPUs.
   - Parameters per GPU: still $70\text{B}/4 = 17.5\text{B}$.
   - Optimizer shard factor: Megatron Core shards optimizer state across $\text{DP} \times \text{CP} = 4 \times 4 = 16$ (identical to the short-context stage!).
   - Static memory per GPU:

$$17.5\text{B} \times \left(2 + 2 + \frac{8}{16}\right)\ \text{B/param} = \mathbf{78.75}\ \text{GB}$$

   - **Justification**: Trading DP for `CP=4` cuts long-context activations and KV cache by an additional $4\times$ without adding pipeline bubbles or increasing static memory.

Interpretation: In both stages, static memory is just $78.75\ \text{GB/GPU}$, leaving over $110\ \text{GB}$ on GB200 ($192\ \text{GB}$) and over $200\ \text{GB}$ on B300 ($288\ \text{GB}$) for dynamic activations with selective recomputation.

WORKED EXAMPLE

Worked Layout 2: Frontier 552B MoE (DeepSeek-V4.1-Flash Scale) on GB300 NVL72 — and Scaling to 2.8T (Kimi K3)

Setup: Train a 552B sparse MoE matching **DeepSeek-V4.1-Flash** scale (384 routed experts, 6 active, 1 shared expert, 1M context) on 64 GPUs inside a single **GB300 NVL72 rack** ($900\ \text{GB/s}$ unidirectional NVLink 5, $288\ \text{GB}$ HBM/GPU).

Visualize the **MoE Parallel Folding** architecture spanning those same 64 GPUs inside one NVSwitch domain:

- **Left grid — Attention Grid (`TP=2 × CP=4 × DP=8`)**:
  - Product check: $2 \times 4 \times 8 = 64$ GPUs.
  - `TP=2` (+ SP) and `CP=4` jointly shard long-context activations ($8\times$ activation reduction).
  - `DP=8` keeps global batch throughput high.
  - The distributed optimizer shards attention optimizer states across $\text{DP} \times \text{CP} = 8 \times 4 = 32$ ranks.
- **Right grid — Expert Grid (`ETP=1 × EP=64 × EDP=1`)**:
  - Product check: $1 \times 64 \times 1 = 64$ GPUs.
  - Routed experts per GPU: $384/64 = \mathbf{6}$ **intact routed experts per GPU** (no TP slicing of expert GEMMs).
  - Notice $\text{EP} = 64 > \text{DP} = 8$—directly enabled by MoE Parallel Folding.
  - 100% of All-to-All dispatch and combine stays on the $900\ \text{GB/s}$ NVLink 5 fabric via HybridEP / DeepEP.
- **Overlap and stability**: Overlap All-to-All with 1F1B compute plus the always-active shared expert; keep the router in FP32 and replicate hot experts online for static CUDA-graph shapes.

Scaling up to 2.8T (Kimi K3, July 2026, 896 routed experts):

- Keep $\text{EP} = 64$ **inside each NVL72 rack**, giving $896/64 = \mathbf{14}$ **routed experts per GPU**, so every All-to-All stays rack-local on NVLink 5.
- Add **Pipeline Parallelism (PP) across NVL72 racks** over $50\ \text{GB/s}$ RDMA to shard the 2.8T total model depth across racks.
- Add **Data Parallelism (DP) across multi-rack replicas** to scale cluster compute linearly.

Interpretation: Every communication pattern sits on the hardware tier built for it—All-to-All and TP/CP inside NVL72, P2P PP and DP across racks—allowing the same design pattern to scale seamlessly from 552B on one rack to 2.8T across a multi-rack cluster.

KEY IDEA

- **Compose 5D parallelism from fastest to slowest link (`[TP, CP, PP, DP]`)**: Keep TP, EP, and Ulysses CP inside the NVL72 NVLink domain, place PP (+ virtual stages) and DP across racks over RDMA, and minimize model parallelism whenever B/N allows DP/FSDP to stay compute-bound.
- **Modern NVL72 clusters shift capacity from TP into wide rack-local EP**: Using MoE Parallel Folding (`ETP=1, EP=64`), experts stay intact inside one rack while attention independently balances `TP`, `CP`, and `DP`, with optimizer states sharded across $\text{DP} \times \text{CP}$.

Even the best-designed 5D layout achieves zero useful progress if hardware faults interrupt a multi-week cluster run every hour—or worse, silently poison the model weights without crashing. Our final section addresses how to keep thousand-GPU runs alive.

✎ Check Your Understanding

1. In Megatron Core's link-speed ordering `[TP, CP, PP, DP]`, why are TP and CP placed innermost while PP and DP are placed outermost?

Answer: TP and (Ulysses) CP require high-bandwidth, frequent intra-layer collectives and therefore belong on the fastest intra-rack NVLink fabric, whereas PP (low-volume bsh point-to-point transfers) and DP (amortized step-level gradient sync) tolerate slower inter-rack RDMA links.

2. A 70B dense model runs on 64 GPUs with `TP=4, PP=1, CP=2, DP=8` using Megatron Core's distributed optimizer at $12\ \text{B/param}$ ($2\ \text{B}$ params, $2\ \text{B}$ grads, $8\ \text{B}$ opt state). What is the static memory per GPU?

Answer: $78.75\ \text{GB}$. Each GPU holds $70\text{B}/4 = 17.5\text{B}$ parameters, and optimizer state shards across $\text{DP} \times \text{CP} = 8 \times 2 = 16$, giving $17.5\text{B} \times (2 + 2 + 8/16) = 78.75\ \text{GB}$.

Section 10: Keeping Long Runs Alive — Failures, Fault Tolerance, and Silent Data Corruption

At tens of thousands of GPUs running continuously for months, hardware failures are not edge cases—they are a routine part of every training day. Examine the empirical failure breakdown from the historical 54-day **Llama 3 405B** run across $16{,}384$ H100 GPUs, visualized as a horizontal bar chart of failure causes (approximately **78% hardware-related**):

- **Faulty GPU**: **30.1%** (largest single cause)
- **HBM3 Memory**: **17.2%**
- **Software Bug**: **12.9%**
- **SRAM / System Processes**: **8.6%**

- **Switch / Cable**: **8.4%**
- **Host Maintenance**: **7.6%**
- **Silent Data Corruption (SDC)**: **1.4% (6 incidents)**

While SDC accounts for only 1.4% of incidents, it is uniquely dangerous: unlike a crashed GPU or dropped switch link that immediately halts the job, **Silent Data Corruption silently corrupts tensors or weights without crashing**, degrading loss or destabilizing training hours or days later.

Keeping frontier runs productive across thousands to $O(100\text{K})$ GPUs requires four fault-tolerance mechanisms working together:

1. **Asynchronous reshardable checkpoints**:
   - Writing terabytes of weights and optimizer state synchronously to remote storage stalls the entire cluster during every checkpoint.
   - Instead, GPUs quickly snapshot state to **host RAM** and flush to persistent storage asynchronously in the background while training resumes immediately.
   - Checkpoints must be **reshardable** so a job interrupted on N GPUs can restore cleanly into a **different TP / PP / DP layout** (for example, resuming with fewer healthy nodes or transitioning into a long-context CP stage).
2. **Hangs, stragglers, and in-process restart**:
   - A single slow (straggler) or hung GPU stalls every collective across the entire synchronous training job.
   - Cluster runtimes monitor step latency to detect hung or slow ranks automatically and **swap faulty nodes in-process** without tearing down and relaunching the whole distributed job from scratch.
3. **Replica-level fault tolerance (`torchft` / `FT-HSDP`)**:
   - In standard synchronous training, a single GPU failure anywhere in the cluster kills all N GPUs.
   - **Replica-level fault tolerance** (`torchft`, `FT-HSDP`) isolates failures to a single Data Parallel replica: the remaining healthy DP replicas continue making forward training progress while the affected replica recovers and rejoins.
   - At $O(100\text{K})$ **GPUs**, `FT-HSDP` lifts effective training time (goodput) from **44% to 80%**.

> **DEFINITION**
> **Replica-level fault tolerance**: A resilience architecture (such as `torchft` or `FT-HSDP`) that confines a hardware failure to a single data-parallel replica rather than halting the entire cluster, allowing healthy replicas to keep training while the failed replica recovers.

4. **Silent Data Corruption (SDC) detection**:

- **DEFINITION**

  Silent Data Corruption (SDC): A hardware fault in which a GPU produces an incorrect numerical result in compute or memory without raising an error or crashing the process, silently poisoning model weights or gradients.

- When an anomalous loss spike or NaN occurs, the standard diagnostic is to **re-run the suspect step** on clean hardware to separate bad hardware from bad training data.
- At production scale, **AEGIS** (OSDI '26) detects SDCs automatically during training: across **35 million GPU-hours**, AEGIS caught **18 SDCs across 13 faulty GPUs** at just **0.86% overhead**.

COMMON MISCONCEPTION

You might think that frequent synchronous checkpointing to disk solves cluster reliability at $100{,}000$ GPUs. That breaks because as GPU count grows, the mean time between cluster-wide failures shrinks inversely with N until full-cluster stop-and-restart cycles consume more time than actual training; keeping goodput high at $O(100\text{K})$ GPUs requires asynchronous host-RAM checkpoints, in-process node swaps, and replica-level isolation (`torchft / FT-HSDP`) so one bad chip never halts all $100{,}000$ GPUs.

WORKED EXAMPLE

Why Full-Cluster Restarts Collapse at 100,000 GPUs — and How Replica Isolation Recovers Goodput

Suppose individual hardware components fail at a constant rate such that a $16{,}000$-GPU cluster experiences one failure every 4 hours (240 minutes), and recovering a full-cluster failure (detecting the hang, draining, restarting, and replaying steps since the last checkpoint) costs 30 minutes of lost training time.

1. **At $16{,}000$ GPUs (full-cluster restart)**:

$$\text{Effective training time} = \frac{240}{240 + 30} = \frac{240}{270} \approx 88.9\%$$

2. **Scale the exact same hardware reliability to $100{,}000$ GPUs ($6.25\times$ more devices)**:

$$\text{Cluster MTBF} = \frac{240 \text{ min}}{6.25} = 38.4 \text{ minutes}$$

   Under a full-cluster restart costing 30 minutes (or slightly longer at larger scale—say 49 minutes):

$$\text{At 30 min recovery: } \frac{38.4}{38.4 + 30} = 56.1\%, \qquad \text{At 49 min recovery: } \frac{38.4}{38.4 + 49} = 43.$$

3. **With replica-level fault tolerance (`FT-HSDP`)**: Suppose the $100{,}000$ GPUs are divided into 10 independent DP replicas of $10{,}000$ GPUs each. A failure halts only 1 replica (10% of the cluster) while the other 9 replicas keep training uninterrupted; even during recovery, 90% of the cluster

remains productive—explaining how `FT-HSDP` raises effective training time from 44% **to** 80% at $O(100\text{K})$ GPUs.

Interpretation: Linear hardware scaling turns occasional single-node failures into continuous cluster-wide interruptions; only isolating fault domains at the replica level preserves high effective utilization at $O(100\text{K})$ scale.

KEY IDEA

- **Fault tolerance is mandatory at frontier scale**: Roughly 78% of interruptions in large runs stem from hardware (GPUs, HBM, switches, cables, SRAM), requiring async reshardable checkpoints, in-process restarts, replica-level isolation (`torchft` / `FT-HSDP`, lifting $O(100\text{K})$-GPU goodput from 44% to 80%), and low-overhead SDC detection (AEGIS: 18 SDCs / 13 faulty GPUs over 35M GPU-hours at 0.86% overhead).
- **End-to-end synthesis of Parallelism Basics**:
  1. Match parallelism to the interconnect: keep TP, EP, and Ulysses CP inside the 72-GPU NVL72 NVLink domain (900 GB/s uni-di) and push PP and DP across racks over RDMA (50 GB/s uni-di).
  2. Exploit $\text{All-Reduce} = \text{Reduce-Scatter} + \text{All-Gather}$ to gain ZeRO-1/2 and Sequence Parallelism memory scaling at zero extra bandwidth.
  3. Stack precision and activation levers: MXFP8/NVFP4 halve memory and comm bytes, while SP + selective recomputation + CP keep activation memory linear.
  4. Fold MoE grids (`TP×CP×DP` vs. `ETP×EP×EDP`), overlap communication with compute across every axis, and align optimizer sharding to whole 2D matrices for Muon.
  5. Design for failure from day one so thousand-GPU runs sustain high goodput without silent corruption.

✎ Check Your Understanding

1. Why are checkpoints in large-scale training designed to be both **asynchronous** and **reshardable**?

Answer: Asynchronous snapshots to host RAM avoid stalling GPUs while flushing terabytes to storage, and reshardability allows a failed or reconfigured run to resume cleanly into a different TP/PP/DP/CP topology.

2. What distinguishes Silent Data Corruption (SDC) from the more common faulty-GPU or HBM3 crash failures, and how is a suspect step diagnosed?

Answer: Unlike hard crashes that halt execution immediately, an SDC silently produces wrong numerical outputs without raising an error, poisoning model weights; operators diagnose it by re-running the suspect step on clean hardware to distinguish bad hardware from bad data (and at scale use continuous detectors like AEGIS).