Lecture 3: Multi-GPU Parallelization Techniques

Edouard Oyallon
edouard.oyallon@cnrs.fr
CNRS, ISIR

What you will learn today

- 3D parallelism: how Tensor (TP), Pipeline (PP), and Data (DDP) parallelism combine in one scalable setup

- Pipeline parallelism strategies: different scheduling “recipes” for organizing the forward and backward passes

- Expert parallelism: why experts can be *cheaper per parameter* breaking the "6 FLOPs per parameter" rule of thumb

- Model sharding: splitting optimizer state across devices so bigger models can fit and train

Model Flop Utilization

- We're concerned with strategies which allow to process big models and numerous data in in parallel.

- Training is typically throughput-driven (time to process tokens), not latency-driven (number of tokens processed per seconds).

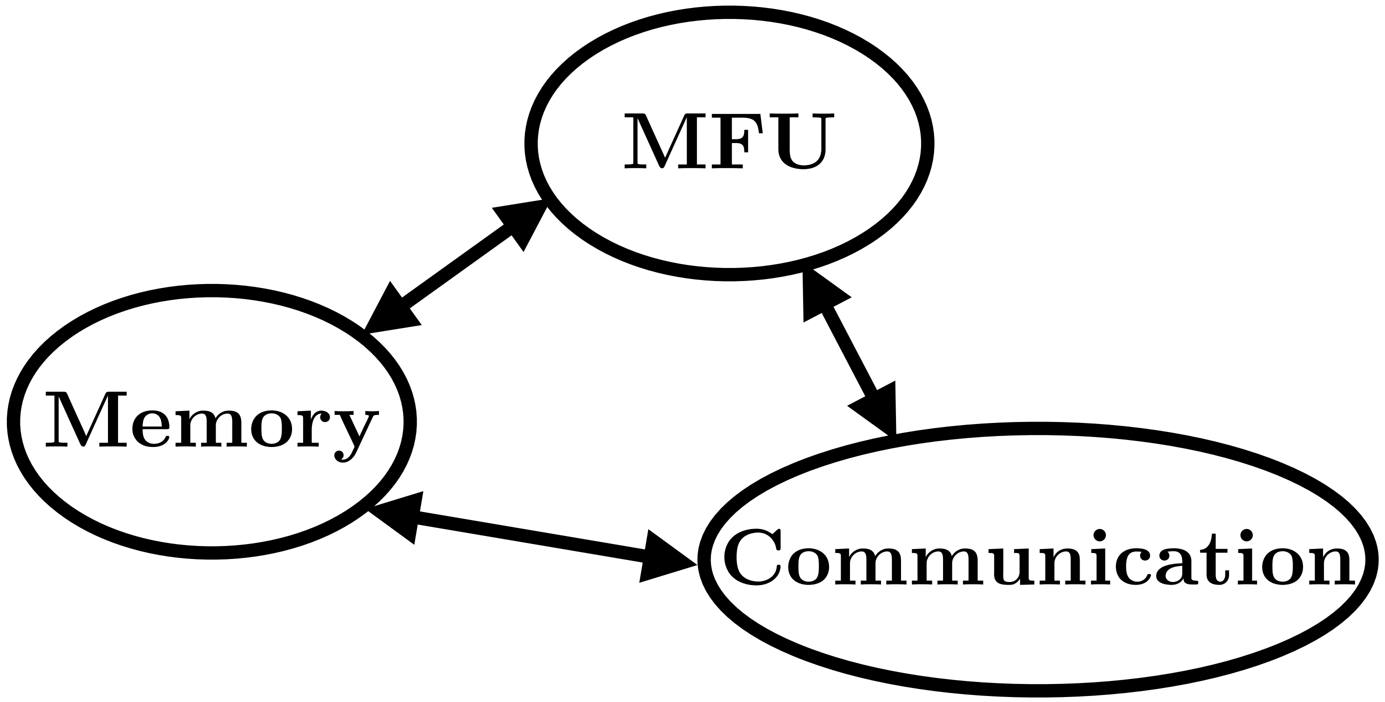
- It is thus necessary to find parallelisation techniques that have the highest throughput, which is directly connected to MFU:

$$\text{MFU} = \frac{\text{FLOPs your model actually executes per second}}{\text{Theoretical peak FLOPs of the hardware}}$$

and equivalently,

$$\text{tokens/s} = \text{MFU} \times \frac{\text{peak FLOPs}}{\text{FLOPs per token}}$$

Compute–Memory–Communication triangle



- **LLMs are huge:** billions of parameters → **memory doesn't fit** on one node
- **Training is compute-heavy:** forward/backward passes require **massive FLOPs** over large datasets
- **Training is bandwidth-heavy:** gradients/activations/optimizer states create **a lot of communication**
- Scaling training is balancing the compute–memory–communication tradeoff: increasing parallelism reduces one bottleneck while often increasing another.

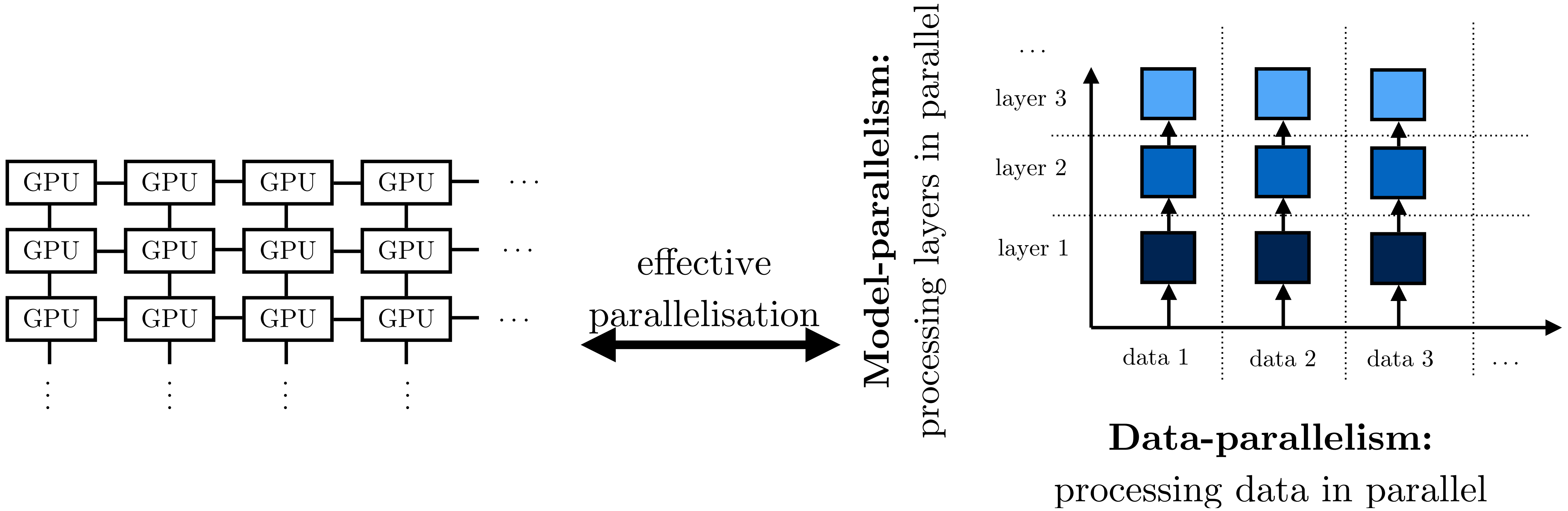
Data Parallelism vs Model Parallelism

- Two guiding ideas shape most parallelism designs:

  - **Data Parallelism (DP):** primarily increases throughput by replicating the model and splitting the data/batch.

  - **Model Parallelism (MP):** primarily increases capacity by splitting the model across devices, and can sometimes improve throughput when compute gains outweigh communication overhead.

- One typically tries to map GPUs to data and layers.


GPU
...
effective parallelisation
Model-parallelism: processing layers in parallel
layer 1
layer 2
layer 3
...
data 1
data 2
data 3
...
Data-parallelism: processing data in parallel

Multi-Axis Parallelism

- **Think "GPU mesh":** arrange GPUs into a logical **grid** (1D/2D/ 3D/…). Each **dimension of the grid** corresponds to one parallelism axis and d-dimensional parallelism can be represented via

$$(i_1, \dots, i_d) \in [n_1] \times \cdots \times [n_d]$$

- Multi-axis parallelism allows to combine multiple parallelisms at once

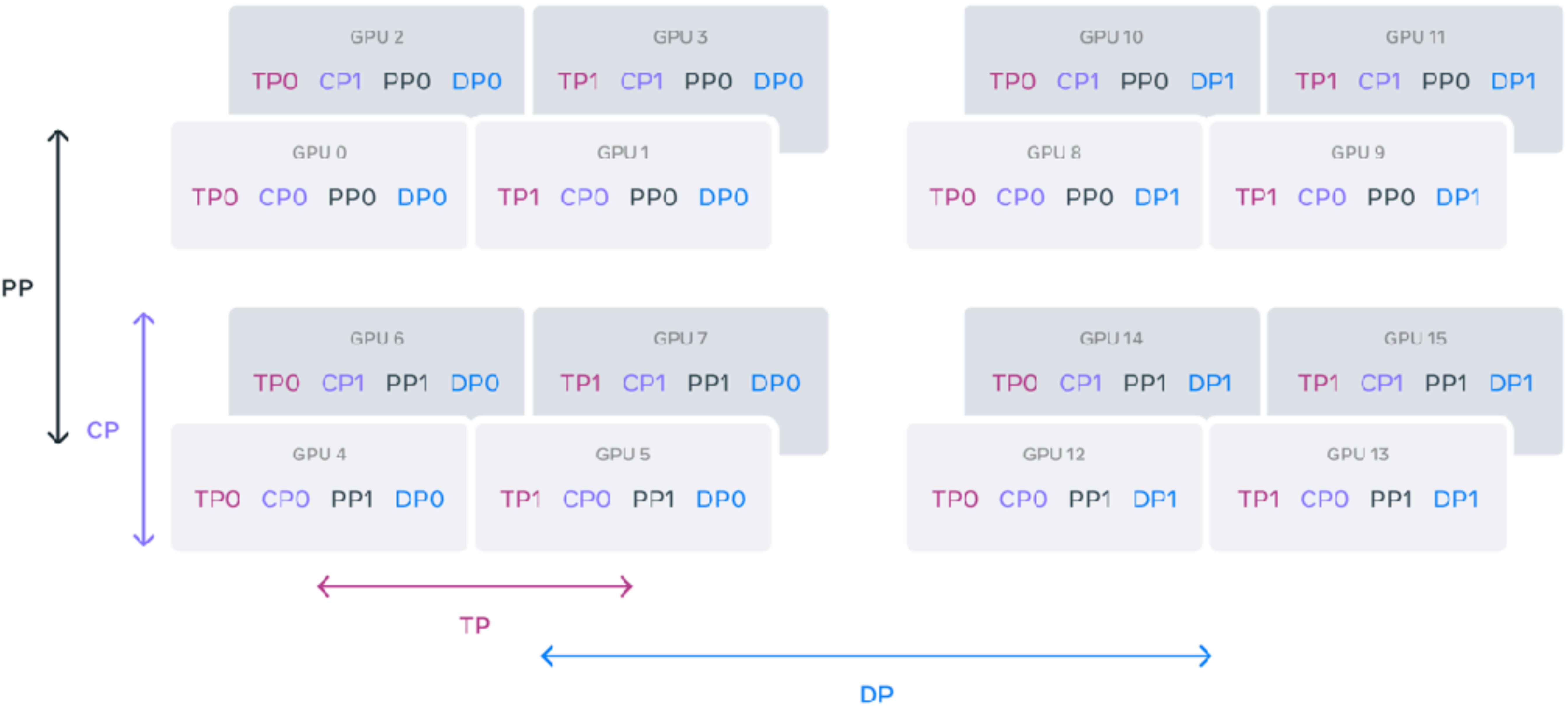
- It often assumes homogeneity in the hardware.

- Each axis implies a different communication pattern. Members in the same k-th groups follow a dimension specific pattern and are given by:

$$\{(i_1, \dots, i_k, \dots, i_d), i_k \in [n_k]\}$$

- **Key trade-off:** more splitting $\rightarrow$ less work/memory per GPU, but **more communication** (bandwidth/latency) and potential **compute bubbles or imbalance**.

Example from Llama3

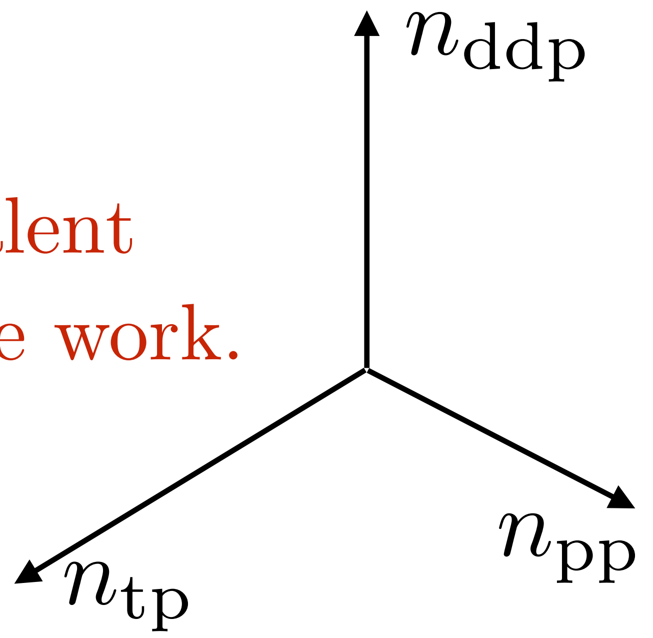


3D-Parallelism

3D-Parallelism

- The probably best known of training LLM is 3D-Parallelism, which splits GPUs into a grid. It mixes:
  - **Data axis (Distributed Data Parallelism):** split the *batch* across GPU groups; each group has a full copy of the model.
  - **Tensor axis (Tensor Parallelism):** split large *matrix* operations / weight tensors inside layers across GPUs.
  - **Pipeline axis (Pipeline Parallelism):** split the model *layers* into stages; GPUs work on different stages.

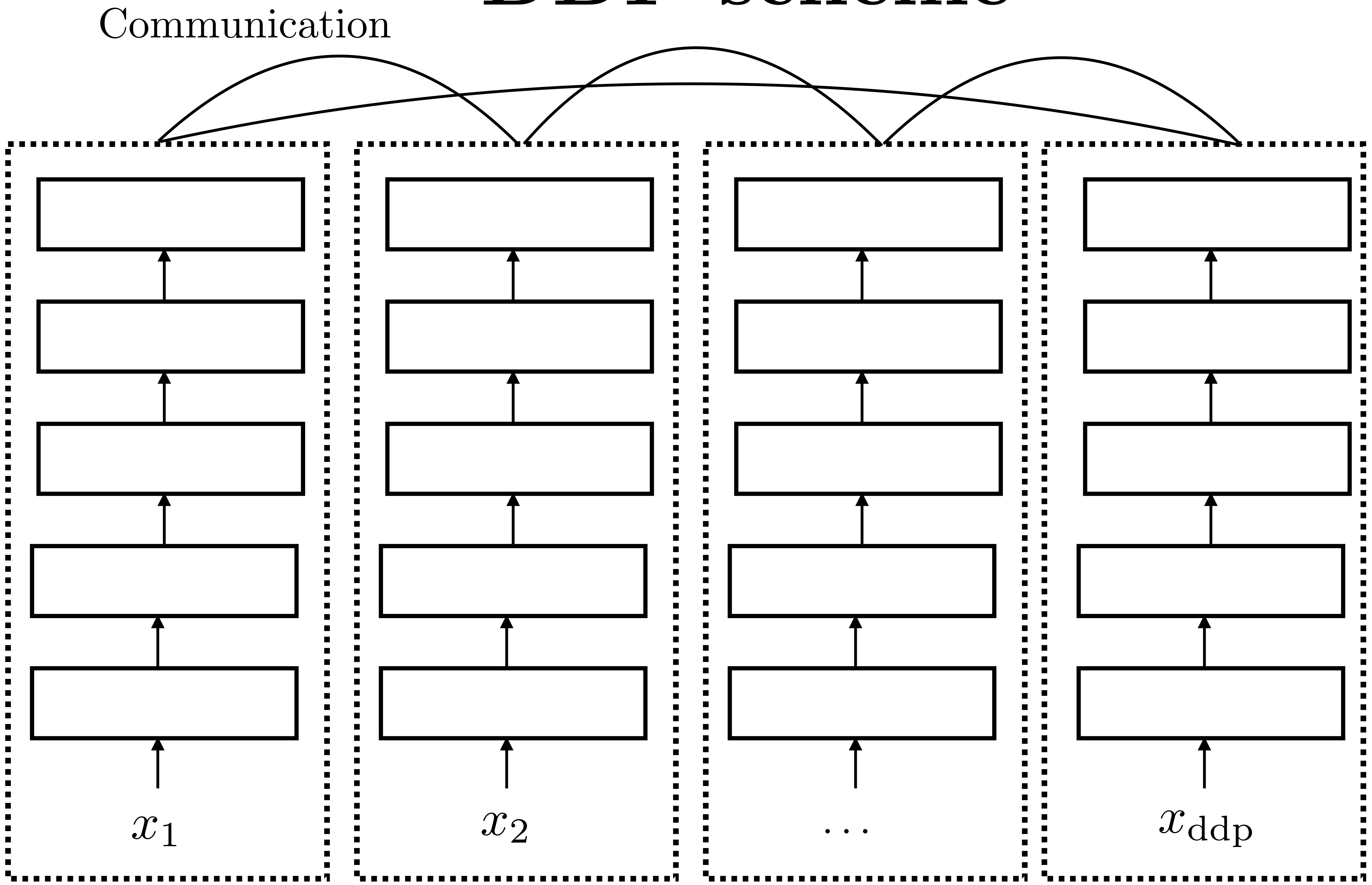
Crucially, 3D-Parallelism is (mathematically) equivalent to the non-parallel training run; it only re-partitions the work.



Distributed Data Parallelism (DDP)

- **Data Parallelism (DDP)** is the standard baseline: replicate the full model on each GPU and split the batch across GPUs.
- After each iteration, GPUs must **synchronize gradients** (typically via an all-reduce); applying the same update keeps **parameters consistent**.
- To use DDP efficiently, choose a **global batch size** so each GPU has enough work to stay busy.
- DDP usually increases **throughput** (tokens/s) by parallelizing compute.
- It introduces **communication overhead** (gradient sync) and some memory/overhead costs (activation/gradient buffers, optimizer state per replica).
- **Step time / latency is not identical**: it often increases slightly due to synchronization/stragglers, even though throughput improves.

DDP scheme



DDP Summary

- **What's sharded: Data / batch** (each GPU gets different samples)
- **Memory footprint:** dominated by **replicated model state** (+ activations); doesn't reduce weight memory per GPU
- **Communications:**
  - **Communication pattern: gradient all-reduce** (typically once per optimizer step / per backward pass)
  - **Comm volume:** scales with #**parameters** and #**GPUs**.
- When it bottlenecks:
  - **Compute-bound** if **per-GPU batch** is large / GPU busy
  - **Comm-bound** as GPU count increases or network bandwidth decreases
- **Primary goal: increase throughput (latency issue due to comm.)**
- **Statistical efficiency:** tied to **global batch size** (bigger global batch may need LR tuning / schedule changes)

Tensor Parallelism

Motivation for Tensor Parallelism (Forward Pass)

- The computations of a 1-hidden layer neural network are separable.

$$\text{Write} \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix}, \mathbf{y} = \mathbf{A}\mathbf{x} \quad \text{then} \quad \mathbf{y} = \mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{A}_1\mathbf{x} \\ \mathbf{A}_2\mathbf{x} \end{bmatrix}.$$

(column parallelism)

$$\text{Write} \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 \end{bmatrix}, \; \mathbf{h} = \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \end{bmatrix}, \text{ then } \mathbf{y} = \mathbf{B}\mathbf{h} = \mathbf{B}_1\mathbf{h}_1 + \mathbf{B}_2\mathbf{h}_2.$$

(row parallelism)

- For a 1-hidden layer neural network, as done in transformers, we want to compute

$$\mathbf{y} = \mathbf{B}\sigma(\mathbf{A}\mathbf{x}) \qquad \text{or} \qquad \mathbf{h} = \sigma(\mathbf{A}\mathbf{x}), \qquad \mathbf{y} = \mathbf{B}\mathbf{h}.$$

Or, equivalently, which makes parallelisation explicit:

$$\begin{aligned} \mathbf{h}_1 &= \sigma(\mathbf{A}_1\mathbf{x}) \\ \mathbf{h}_2 &= \sigma(\mathbf{A}_2\mathbf{x}) \\ \mathbf{y} &= \mathbf{B}_1\mathbf{h}_1 + \mathbf{B}_2\mathbf{h}_2. \end{aligned}$$

Motivation for Tensor Parallelism (Backward Pass)

Again:
$$\mathbf{h}_1 = \sigma(\mathbf{A}_1\mathbf{x})$$
$$\mathbf{h}_2 = \sigma(\mathbf{A}_2\mathbf{x})$$
$$\mathbf{y} = \mathbf{B}_1\mathbf{h}_1 + \mathbf{B}_2\mathbf{h}_2.$$

Then if $\delta^y = \dfrac{\partial \ell}{\partial y}$

then $\delta^{h_i} = B_i^\top \delta^y$ *(column parallelism)*

and if $\delta^{z_i} = \delta^{h_i} \odot \sigma'(A_i x)$,

then $\dfrac{\partial \ell}{\partial x} = A_1^\top \delta^{z_1} + A_2^\top \delta^{z_2}$ *(row parallelism)*

TP: Tensor Parallelism

$$\begin{tikzcd}
 & & \text{GPU1 holds } (\mathbf{A}_1, \mathbf{B}_1) & & \\
 & \mathbf{A}_1\mathbf{x} \arrow[r] & \mathbf{h}_1 = \sigma(\mathbf{A}_1\mathbf{x}) \arrow[r] & \mathbf{y}_1 = \mathbf{B}_1\mathbf{h}_1 \arrow[dr] & \\
x \arrow[ur] \arrow[dr] \arrow[rrr, dotted, no head] & & & & \mathbf{y} = \mathbf{y}_1 + \mathbf{y}_2. \\
 & \mathbf{A}_2\mathbf{x} \arrow[r] & \mathbf{h}_2 = \sigma(\mathbf{A}_2\mathbf{x}) \arrow[r] & \mathbf{y}_2 = \mathbf{B}_2\mathbf{h}_2 \arrow[ur] & \\
 & & \text{GPU2 holds } (\mathbf{A}_2, \mathbf{B}_2) & &
\end{tikzcd}$$

- **Intra-layer model parallelism:** split large linear/attention ops across GPUs.
- **TP shards weights** across GPUs $\rightarrow$ each GPU stores a fraction $\dfrac{1}{n_{\mathbf{tp}}}$ of the layer weights.
- Each GPU does a partial matmul/attention projection on its shard.
- It requires **collectives inside the layer** (e.g., all-reduce / all-gather) to combine partial results; model states are localised on GPUs.
- TP aims at throughput and model fit; latency depends on communications.

Column vs Row Parallelism

- Tensor parallelism can be applied typically for linear layers in attention or MLP layers. With PyTorch notations, one typically want to perform

$$Y = XW$$

- **Column-parallel linear** (split output features): each GPU computes a slice of the output → may need **all-gather** later.

$$\text{Split: } W = \left[\, W^{(1)} \; W^{(2)} \; \cdots \; W^{(p)} \,\right]$$

$$\text{then } Y^{(r)} = X\,W^{(r)}, Y = \text{Concat}\left(Y^{(1)}, \dots, Y^{(p)}\right).$$

- **Row-parallel linear** (split input features): each GPU consumes a slice and produces partial sums → combine with **all-reduce** / **reduce-scatter**.

$$\text{Split: } X = \left[\, X^{(1)} \; X^{(2)} \; \cdots \; X^{(p)} \,\right], \qquad W = \begin{bmatrix} W^{(1)} \\ W^{(2)} \\ \vdots \\ W^{(p)} \end{bmatrix}$$

$$\text{then } \tilde{Y}^{(r)} = X^{(r)}\,W^{(r)}, Y = \sum_{r=1}^{p} \tilde{Y}^{(r)}.$$

TP Implementation in PyTorch

- In PyTorch `DTensor` TP, you apply tensor parallelism by passing a `parallelize_plan` (per-submodule sharding style such as colwise/rowwise/sequence) to `parallelize_module(model, device_mesh, ...)`.

- PyTorch shards the targeted weights accordingly and automatically inserts the required collectives (e.g., all-gather, reduce-scatter, all-reduce) to make the sharded layers compose correctly.

- These TP APIs are experimental and can change: so I would treat this procedure as a black-box.

TP Summary

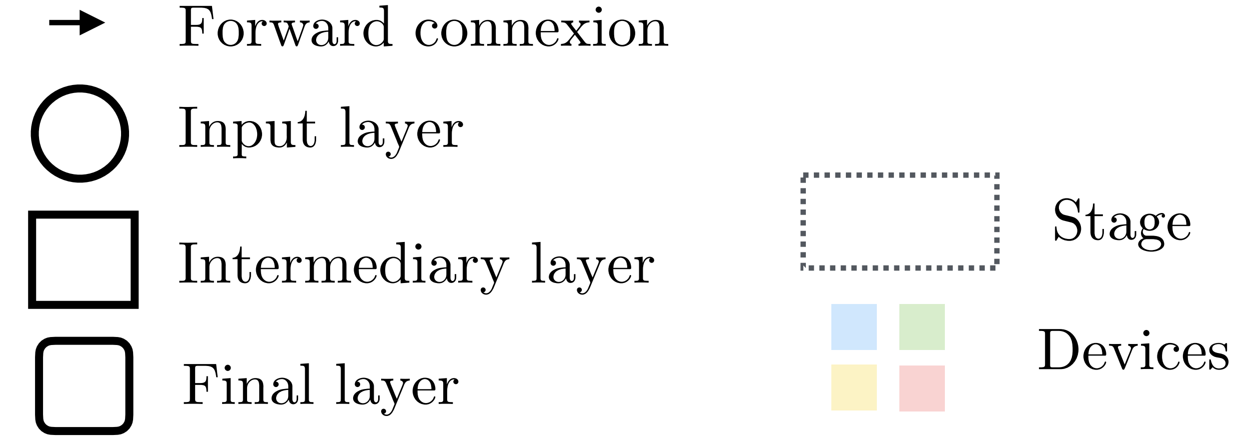
- **What's sharded:** Layers states.
- **Memory footprint**: Per-GPU model states drops by $\frac{1}{n_{\text{tp}}}$, but some activations memory are replicated.
- Communications:
  - **Communication pattern:** collectives inside layers between activations (all-reduce / reduce-scatter / all-gather, depending on row/col/sequence parallel style).
  - **Comm volume**: scales with activation size per layer and with how many TP collectives you do per layer.
- When it bottlenecks:
  - **Compute-bound** when GEMMs are large and GPUs stay busy.
  - **Comm-bound** as TP degree increases (more/larger collectives) or interconnect bandwidth/latency is weak.
- **Primary goal**: make large layers fit + increase throughput (and sometimes enable larger batch/seq); latency may not improve much if comm dominates.
- **Statistical efficiency: unchanged**.

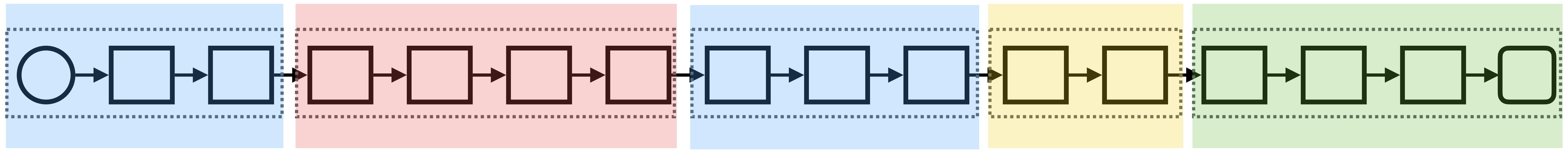
Pipeline Parallelism

PP: Pipeline Parallelism

- Even with TP, **layers are executed sequentially** in forward/backward, and a full model may still **not fit on one GPU**.
- **Core idea: partition the network into stages** (contiguous blocks of layers) and place each stage on a different GPU (or group of GPUs).
- PP splits the global batch into **microbatches** and run them through the stages with a **pipelined schedule** (overlap compute across stages): different stages work on different microbatches at the same time, increasing device utilization.
- **Communication: send/recv activations** between adjacent stages (per microbatch) + gradients on the backward path.
- Main trade-off (which depends on the PP schedule):
  - more microbatches $\rightarrow$ **less pipeline bubble** (better utilization)
  - more microbatches $\rightarrow$ **larger activation stash** (more memory)

Deploying Large Models across multiple GPUs via Pipelining





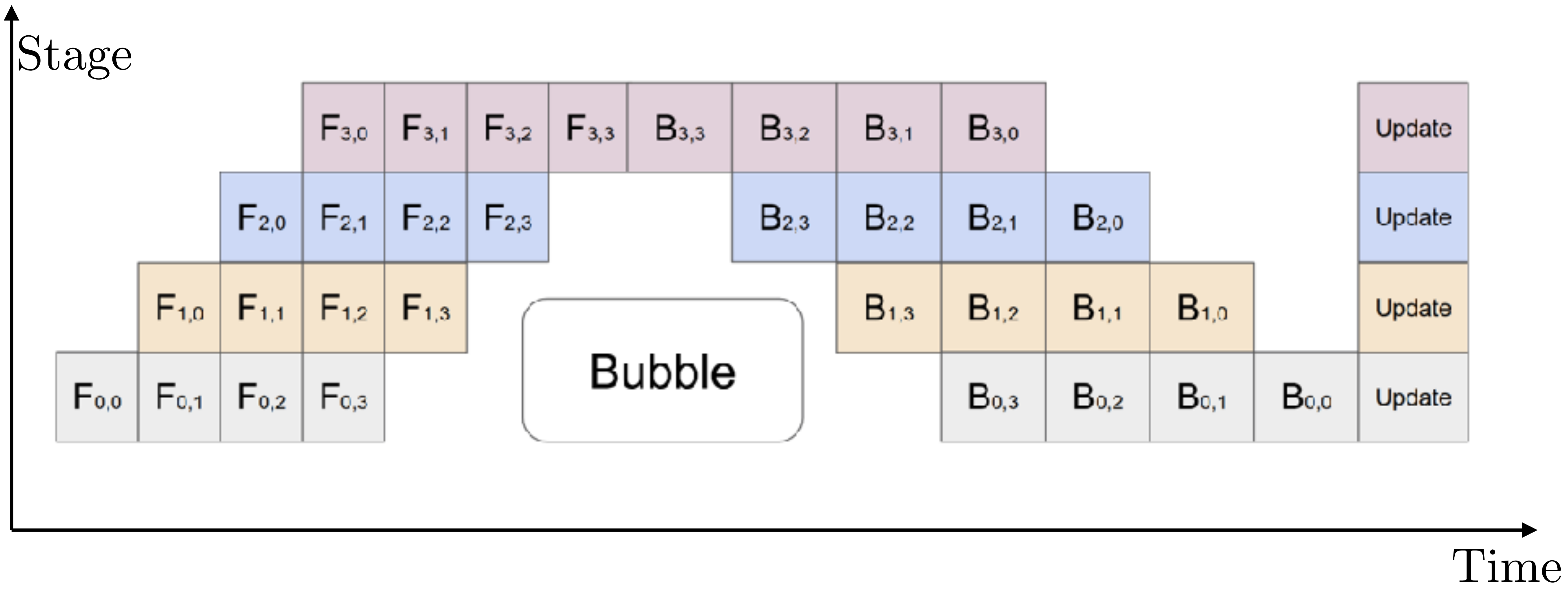
- **Typical setting**: Feedforward model.
- **Stages**: Disjoint segments of consecutive layers.
- **Device assignment:** Each stage resides entirely on one device; multiple stages can share a single device.
- **Local management:** Each stage independently handles its memory, computations, and communication.

Micro-batches for GPipe

$$\underset{\text{Stage index}}{A_{i,j}} \qquad A \in \{F, B\}$$

Stage index

Micro-batch index

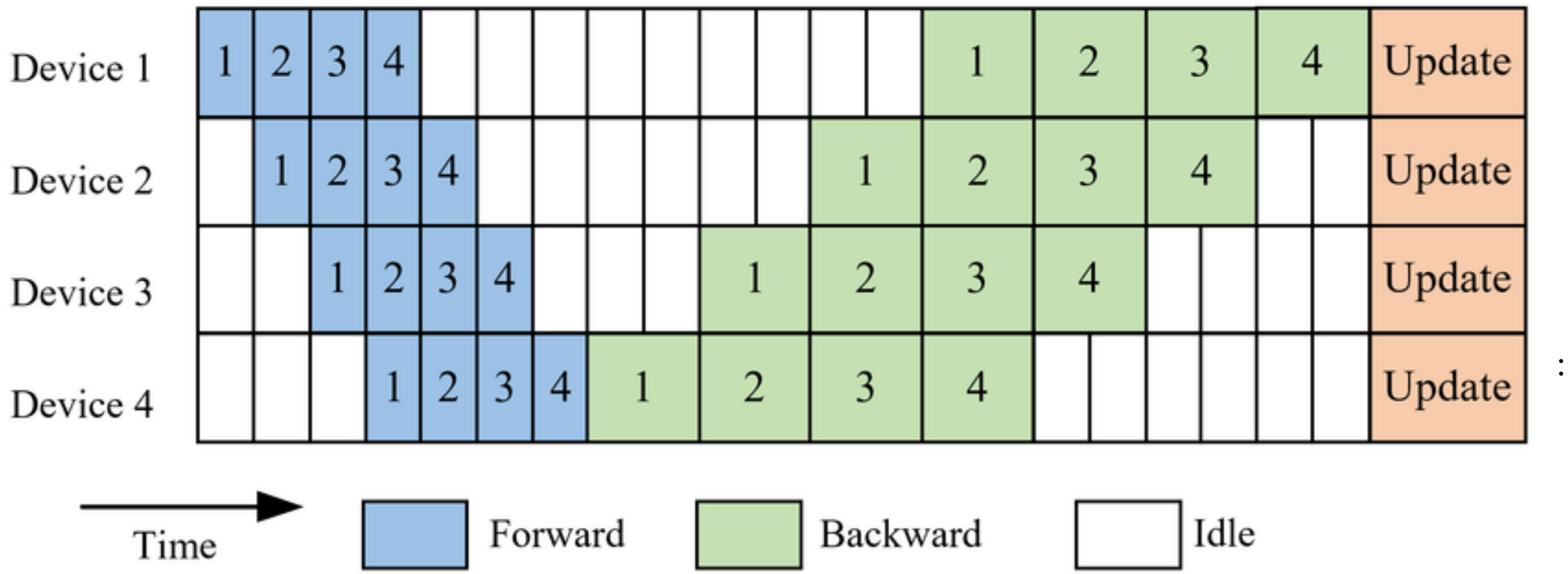


Stage
F3,0 F3,1 F3,2 F3,3 B3,3 B3,2 B3,1 B3,0 Update
F2,0 F2,1 F2,2 F2,3 B2,3 B2,2 B2,1 B2,0 Update
F1,0 F1,1 F1,2 F1,3 B1,3 B1,2 B1,1 B1,0 Update
F0,0 F0,1 F0,2 F0,3 B0,3 B0,2 B0,1 B0,0 Update
Bubble
Time

Main overhead: pipeline “bubble” (idle time)

GPipe

- Assume we have p stages and n micro-batches, each on a diff GPU.



Per stage:

Forward total time: $(p+n-1)T_f$

Forward total idling time: $(p-1)T_f$

Bubble time (per stage): $(p-1)(T_f+T_b)$

T_f : forward time

T_b : backward time

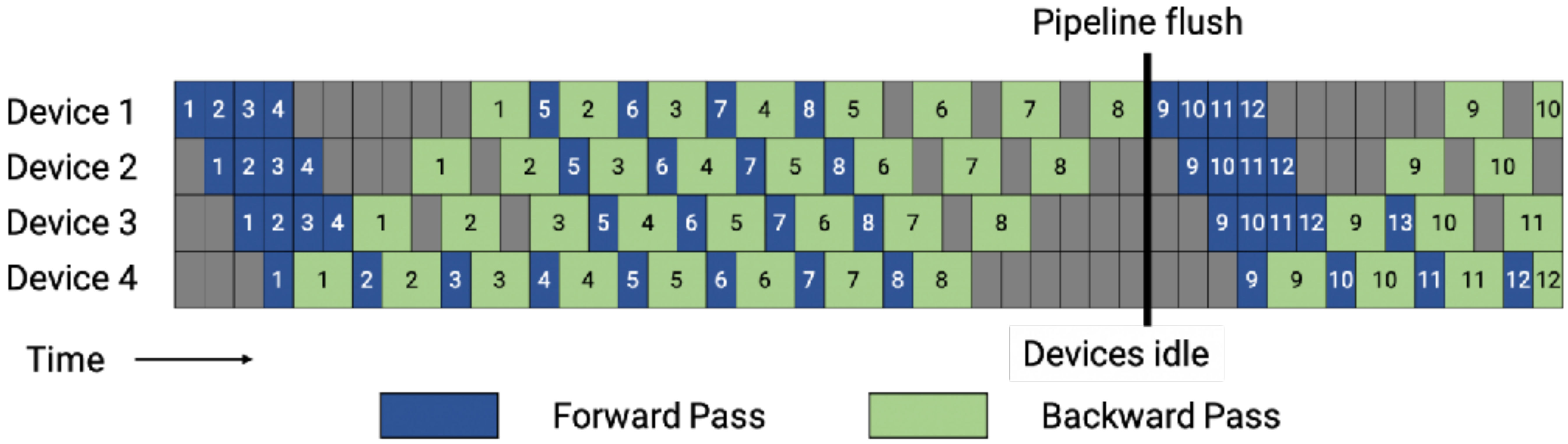
Bubble ratio (over all GPUs):

$$\frac{p-1}{p+n-1}$$

What is the activation memory per layer?

Non-interleaved 1F1B

- One can do much more complex Pipelining strategies:



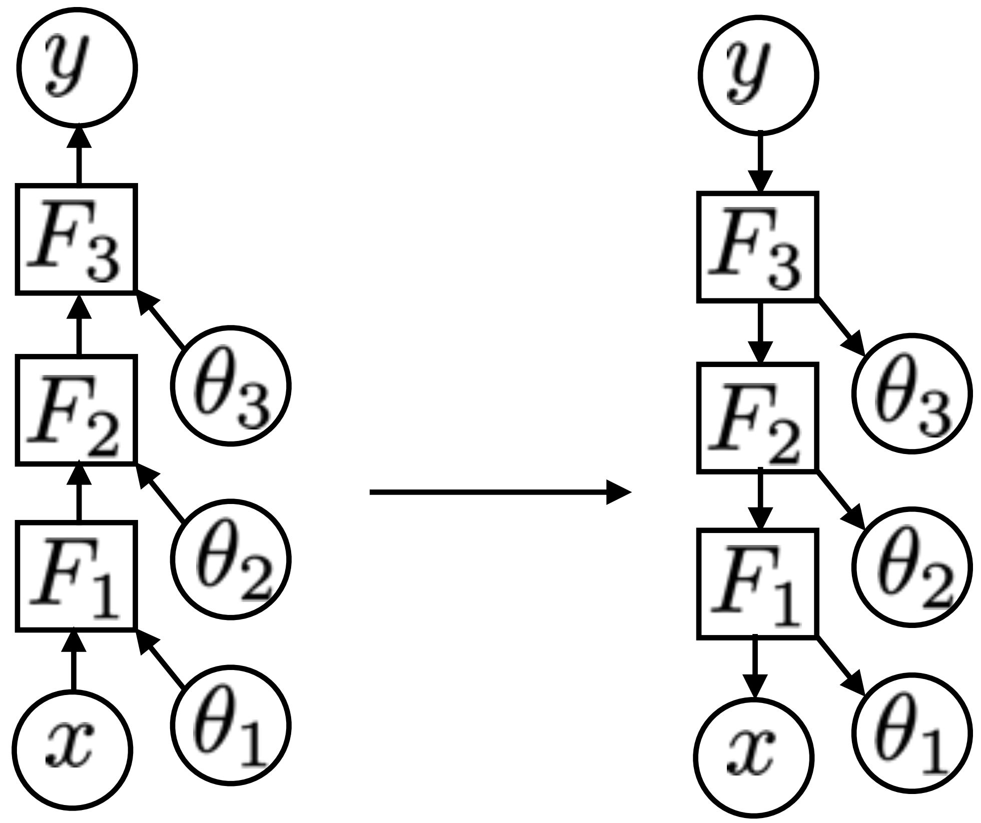
- After a short warmup, each stage alternates exactly one Forward (F) and one Backward (B) pass per micro batch

- Here, the bubble is again given by: $\dfrac{p-1}{p+n-1}$

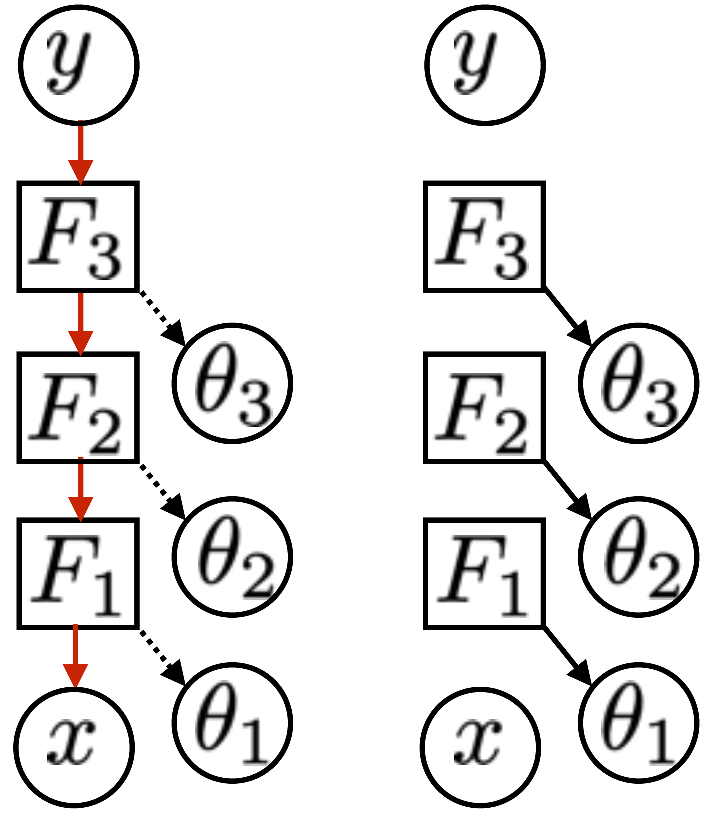
- More memory efficient than GPipe because we immediately perform backward, dropping stored activations.

Decomposing Backprop: Gradients w.r.t. Weights vs Activations

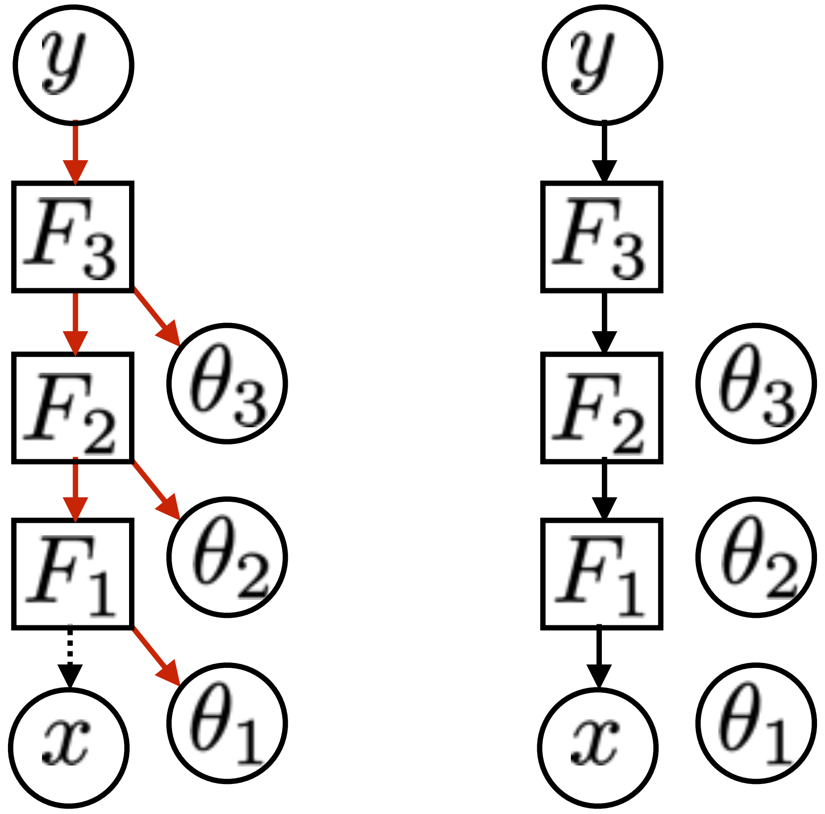
$\nabla_{x,\theta}$



$\nabla_\theta \nabla_x$



$\nabla_x \nabla_\theta$



Splitting

- Observe that the backward pass can be either done jointly, either done activation, then weights

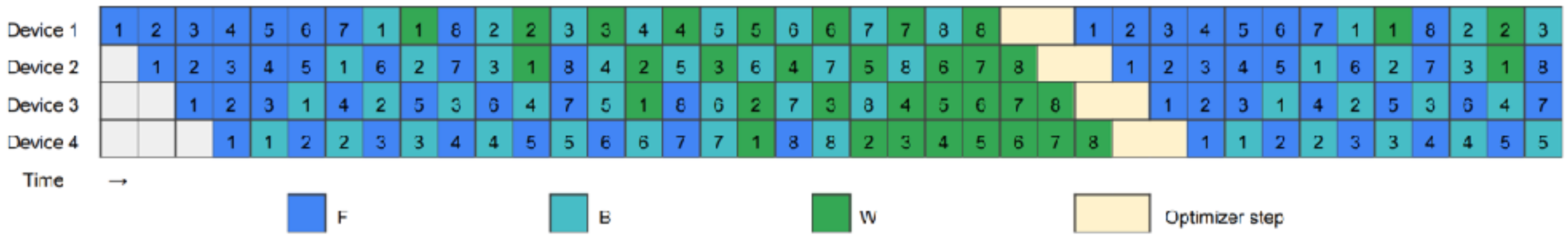
$$\begin{aligned} a_1 &= W_1 x \\ h_1 &= \phi_1(a_1) \\ a_2 &= W_2 h_1 \\ h_2 &= \phi_2(a_2) \\ y &= W_3 h_2 \end{aligned}$$

$$\begin{aligned} g &= \nabla_y L \\ \delta_2 &= (W_3^\top g) \odot \phi_2'(a_2) \\ \delta_1 &= (W_2^\top \delta_2) \odot \phi_1'(a_1) \\ \nabla_x L &= W_1^\top \delta_1 \\ \nabla_{W_3} L &= g\, h_2^\top \\ \nabla_{W_2} L &= \delta_2\, h_1^\top \\ \nabla_{W_1} L &= \delta_1\, x^\top \end{aligned}$$

Starting from ∇_W, we would need to compute ∇_x twice

- Typically: $T_f = T_a = T_w, T_b = T_a + T_w$

ZB-H2



- **Key trick:** split the backward pass into activation-gradient and weight-gradient computations.

- **Tradeoff: zero bubble costs memory** ZB-H2 increases peak activation memory to about $2p - 1$ compared to p for 1F1B.

- Bubble size: for ZB-H2, we get

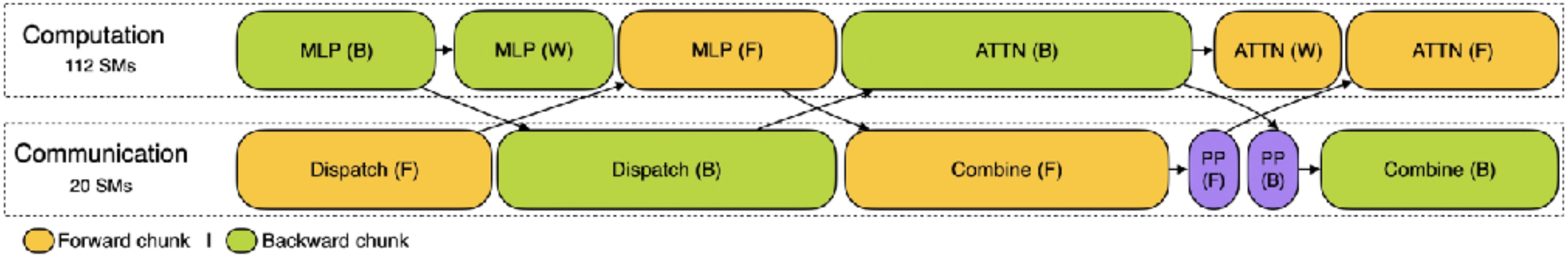
$$(p-1)(T_f + T_a - 2T_w)$$

and for 1F1B

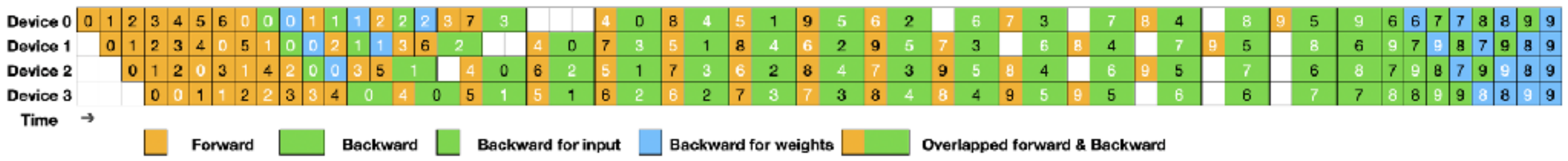
$$(p-1)(T_f + T_a + T_w)$$

Beyond 1 stage per Device: Overlapping Stage Computations & Communications

- **PP already enables overlap:** while one stage computes, other stages can send/recv activations/gradients..
- **DeepSeek (DualPipe) pushes this further:** each micro-batch "chunk" is split into components (attention / all-to-all dispatch / MLP / all-to-all combine), then reordered so PP computations hide communications (could be TP, but actually it's EP): *backward comm/comp* and *forward comp/comm* in parallel.
- **Bidirectional pipeline scheduling:** micro-batches are injected from both ends of the pipeline to reduce bubbles and increase overlap opportunities.



DualPipe-V



- DualPipeV is a **V-shaped pipeline schedule**, obtained from DualPipe via a **"cut-in-half"** transformation.

- Two stages per GPU: a pipeline with p stages runs on $p/2$ devices (each device hosts 2 stages).

- Memory profile: it stores roughly the same number of micro-batches as 1F1B.

$$(p/2-1)(T_f+2(T_a+T_w)-3T_w)$$

- **Bubble size:** DualPipeV bubble size is:

$$(p-1)(T_f+T_a+T_w)$$

PP Summary

- **What's sharded:** Stages across GPUs
- **Memory footprint:** but activation memory can increase due to microbatch activation stash.
- Communications:
  - **Communication pattern:** point-to-point send/recv of boundary activations between adjacent stages (per microbatch) + corresponding backward boundary grads.
  - **Communication volume**: scales with activation sizes and micro batches
- When it bottlenecks:
  - **Compute-bound** if stages are well balanced and each stage has enough work (no pipeline bubble; bubble is often reduced as micro-batches increase).
  - **Communication-bound:** if interconnect is slow, or micro-batch count is small (more overhead per unit work).
- **Primary goal: fit very large models** and **increase throughput via pipelining**; **end-to-end latency** often doesn't improve and can worsen with more microbatching.
- **Statistical efficiency: unchanged**.

Composability of 3D Parallelism

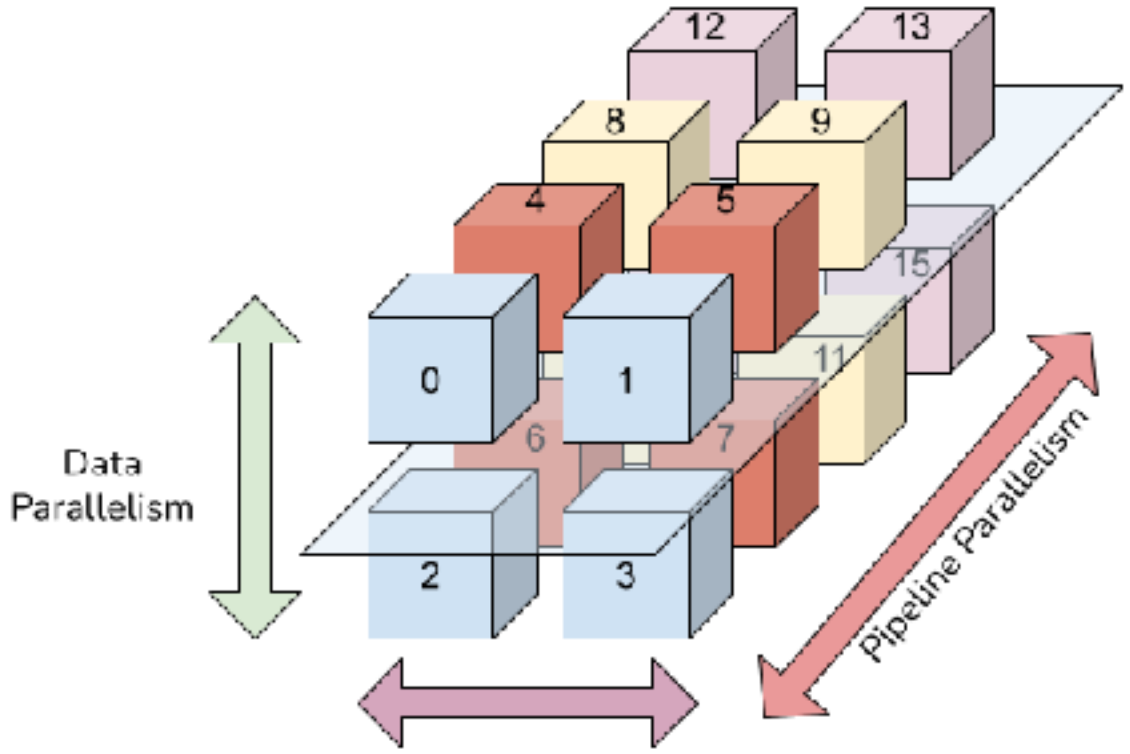
- **3D parallelism is composable:** define orthogonal process groups for DDP, TP and PP, i.e.

$$n = n_{\text{ddp}} \cdot n_{\text{tp}} \cdot n_{\text{pp}}$$

- **Homogeneous assumption:** the same parallel degrees apply everywhere assuming identical GPUs

- **Result:** scaling is "lego-like", you can modify $n_{\text{ddp}}, n_{\text{tp}}, n_{\text{pp}}$ independently…

- … or at least guided by throughput!



Tensor Parallelism

Each cube represents a GPU, and each color represents the node they belong to

What is the recipe to find the right degree of parallelism?

- Pick the **smallest** $(n_{\text{tp}}, n_{\text{pp}})$ that makes the model **fit in memory** while keeping **good per-GPU efficiency:**
    - **TP** helps with **wide layers** (intra-layer sharding)
    - **PP** helps with **depth** (shard layers across stages)
- Topology rule of thumb: keep TP within a node (fast interconnect); PP can span nodes more comfortably.
- Set n_{ddp} with the remaining GPUs:
- And then **extensively** try your potential configurations on few iterations to verify MFU.

Optimizing Memory

Training memory: What dominates and how to reduce it?

- Main contributors:
  - Activations saved for backward
  - Parameters (weights)
  - Gradients
  - Optimizer states
- Common memory reduction levers, that we already discussed:
  - Activation checkpointing (store fewer activations, recompute during backward)
  - Activation and optimizer offloading (move states to host memory when needed)
  - Lower-precision training (BF16 or FP16; FP8 where supported) with appropriate handling of master weights

Optimising the Optimizer

- For large models, **optimizer states dominate GPU memory** (often more than activations at scale).

- Example: Adam-family optimizers keep extra per-parameter tensors (e.g., momentum + variance), which can add **parameter-size** *just for states* (and even more in mixed precision when states are stored in FP32).

- Two high-level strategies

  - Change the optimizer (reduce state size per parameter), with a new optimizer or with **low-precision optimizer states**: store states in FP16 while keeping stability tricks.

  - **Shard optimizer states** (and optionally gradients/params) so each GPU holds only **a fraction** of the state, yet this increases communications.

Issues with Adam

$$\begin{cases} m_t = \beta_1 m_{t-1} + (1-\beta_1)\,\nabla f_t(\theta_{t-1})\,, \\ v_t = \beta_2 v_{t-1} + (1-\beta_2)\,\nabla f_t(\theta_{t-1}) \odot \nabla f_t(\theta_{t-1})\,, \\ \hat{m}_t = \dfrac{m_t}{1-\beta_1^t}, \\ \hat{v}_t = \dfrac{v_t}{1-\beta_2^t}, \\ \theta_t = \theta_{t-1} - \eta\,\dfrac{\hat{m}_t}{\sqrt{\hat{v}_t}+\varepsilon}, \end{cases}$$

- **Adam's memory cost:** per parameter you store the weight plus **two optimizer-state tensors**: the **first and second moments.** Let's count the **bytes per parameter**.

- Typical mixed-precision setup: weights and gradients are often BF16 $\rightarrow$ 2 bytes (weights) + 2 bytes (grads). Many implementations also keep FP32 master weights $\rightarrow$ +4 bytes.

- Optimizer states: moments are typically stored in FP32 for stability $\rightarrow$ 4 + 4 bytes. Total = 2 + 2 + 4 + 4 + 4 = 16 bytes per parameters).

SGD. . . ?

$$\begin{cases} b_t = \mu\, b_{t-1} + (1-\tau)\,\nabla f_t(\theta_{t-1})\,, \\ \theta_t = \theta_{t-1} - \gamma\, b_t, \end{cases}$$

- **SGD's memory cost:** per parameter you store the weight, and (optionally) **one optimizer-state tensor**: the **momentum buffer**. Let's count the **bytes per parameter** again.

- Typical mixed-precision setup: weights and gradients are often BF16 $\rightarrow$ 2 bytes (weights) + 2 bytes (grads). Many implementations also keep FP32 master weights $\rightarrow$ +4 bytes.

- **Optimizer states:** with **SGD + momentum**, you store the momentum buffer, typically in **FP32** $\rightarrow$ **+4 bytes**. **Total = 2 + 2 + 4 + 4 = 12 bytes per parameter**.

- **Note:** plain SGD/momentum typically has **poor convergence** for training LLMs compared to Adam-type optimizers.

Optimizer choice: MuOn

$$\begin{cases} b_t = \mu\, b_{t-1} + (1-\tau)\, \nabla_W \ell_t(W_{t-1})\,, \\ X_0 = \dfrac{b_t}{\|b_t\|_F + \varepsilon}, \\ X_{k+1} = a\, X_k + \big(b\, X_k X_k^\top + c\, (X_k X_k^\top)^2\big) X_k, \quad k = 0, \dots, s-1, \\ W_t = W_{t-1} - \eta\, X_s. \end{cases}$$

orthogonalization

- **MuOn's memory cost:** per parameter you store the weight, and (optionally) **one optimizer-state tensor**: the **momentum buffer**. Let's count the **bytes per parameter** again.

- Typical mixed-precision setup: weights and gradients are often BF16 $\rightarrow$ 2 bytes (weights) + 2 bytes (grads). Many implementations also keep FP32 master weights $\rightarrow$ +4 bytes.

- **Optimizer states:** you store the momentum buffer, typically in **FP32** $\rightarrow$ **+4 bytes**. **Total = 2 + 2 + 4 + 4 = 12 bytes per parameter**.

- **Note:** contrary to SGD, MuOn seems to have extremely favorable convergence properties in many settings! (with minor overhead due to orthogonalization)

Beyond 3D Parallelism

Toward d-D Parallelism, $d \geq 4$

- Modern LLM training often adds a **4th axis (and sometimes more)** to scale further, and in particular to reduce memory overhead:
  - **Expert Parallelism (EP)** using Mixture of Experts layers (experts sharded across GPUs). EP changes convergence properties.
  - **FSDP/ZeRO-style sharding** for optimizer + parameter states
  - **Context / Sequence parallelism** to handle very long sequences, for inference, that we will review in a future lecture.
- Typically, in TorchTitan the axis are so that

$$n = n_{\mathrm{dsp}} \cdot n_{\mathrm{ddp}} \cdot n_{\mathrm{pp}} \cdot n_{\mathrm{tp}} \cdot n_{\mathrm{ep}}$$

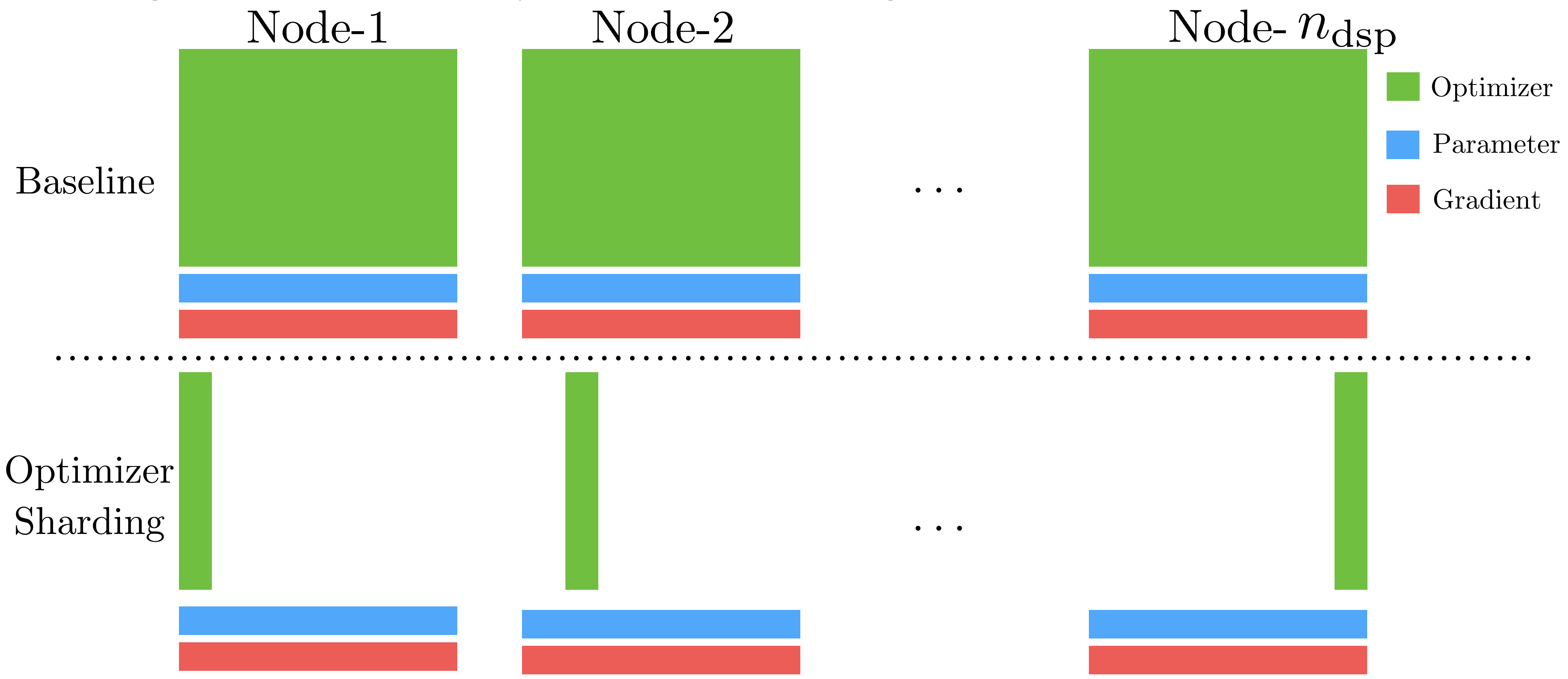
Sharding Parallelism

Optimizer Sharding (ZerO)

- **Shard tensors across GPUs** so each device holds only a fraction instead of a full replica. It's also referred to as ZerO-1.

- GPUs **gather** shards when a layer is used (forward/backward) and **reduce-scatter / all-reduce** to combine updates—so correctness matches the non-sharded model.

- Sharding **saves memory but increases communication**, which can reduce throughput unless carefully overlapped and engineered.



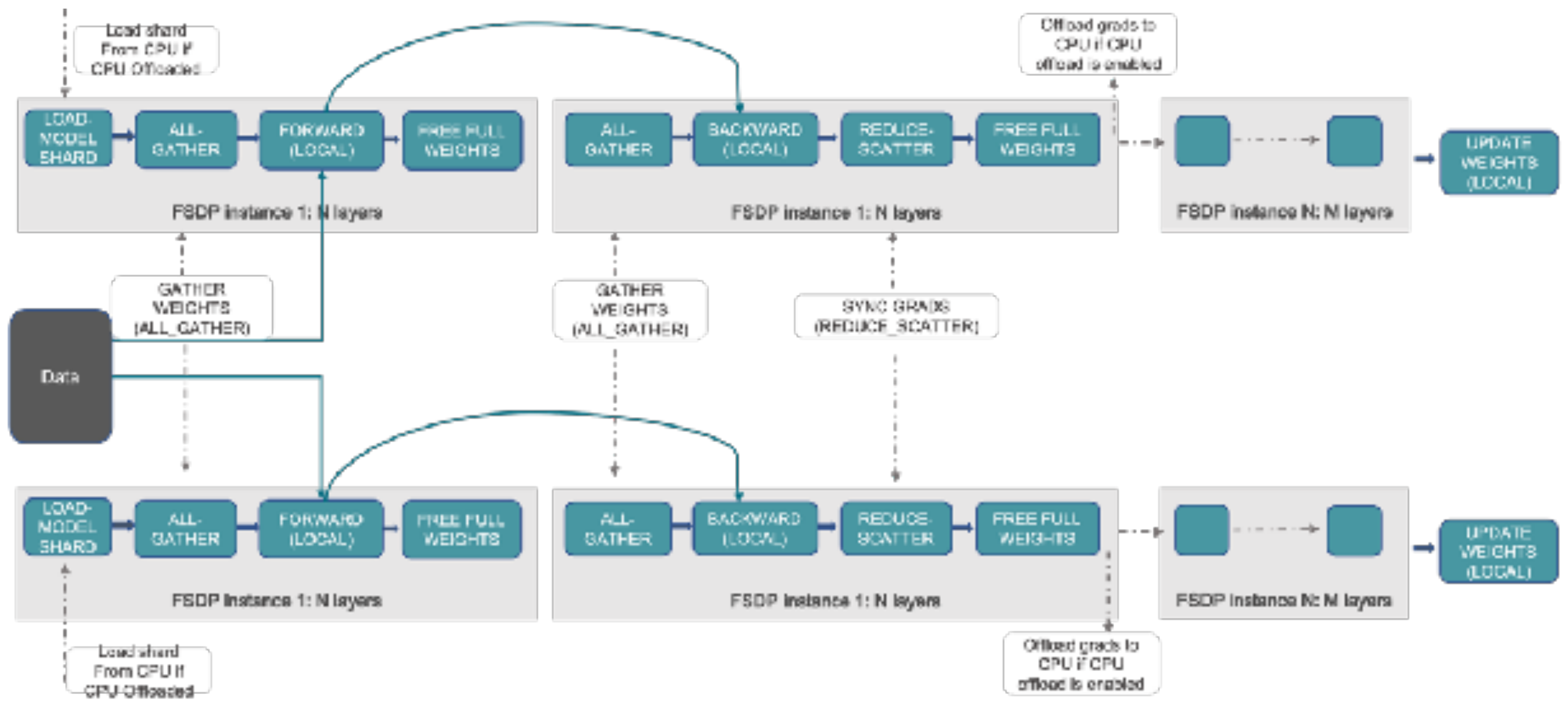
Hybrid Sharded Data Parallel

- *Hybrid Sharded Data Parallel* adds a sharding layer by **partitioning tensors across ranks** instead of replicating them. It extends 3D Parallelism.

- HSDP can shard **optimizer states, gradients, and parameters** (ZeRO-3-style savings), so each GPU holds only a **fraction** of the model "payload" at any time

- Sharding increases **collective communication**, so it's best to keep the hottest traffic on **fast intra-node links** (e.g., NVLink) and carefully manage cross-node bandwidth.

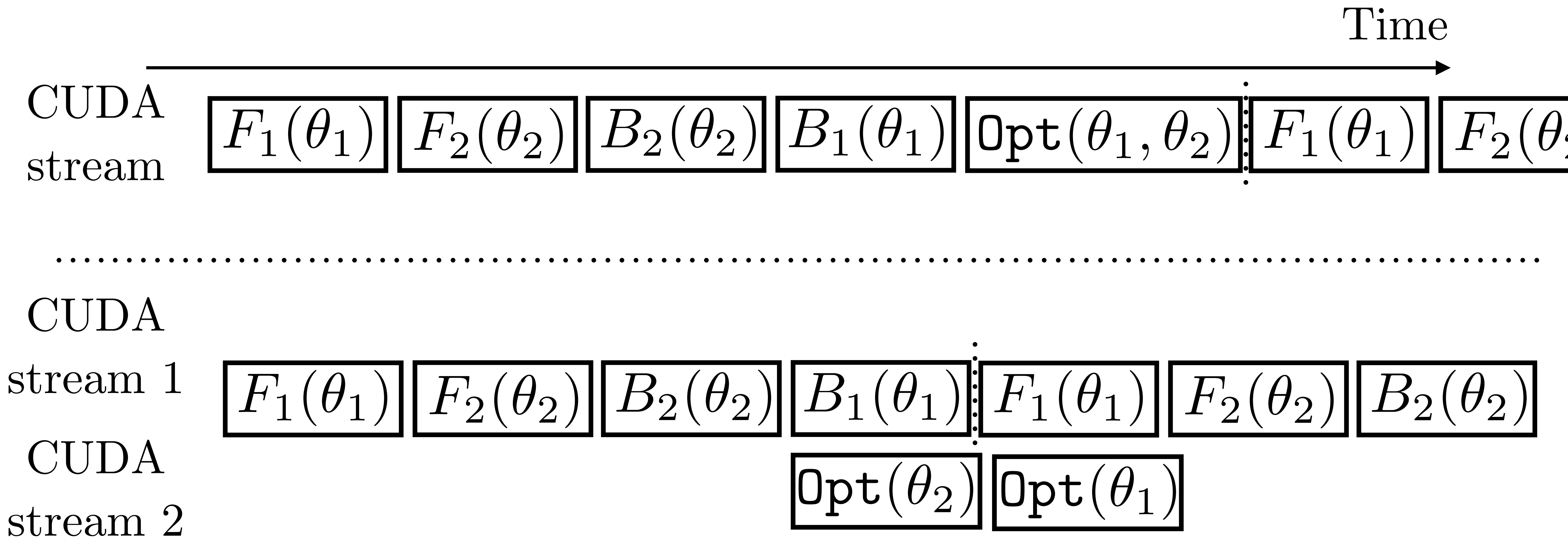
- HSDP includes routines to **overlap comm/compute**, choose **wrapping granularity**, and tune **bucketing/prefetch**, attempting to recover maximum throughput.



Load shard From CPU if CPU Offloaded
Offload grads to CPU if CPU offload is enabled
LOAD MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
FSDP instance N: N layers
UPDATE WEIGHTS (LOCAL)
GATHER WEIGHTS (ALL_GATHER)
GATHER WEIGHTS (ALL_GATHER)
SYNC GRADS (REDUCE_SCATTER)
Data
LOAD MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
FREE FULL WEIGHTS
FSDP instance 1: N layers
ALL-GATHER
BACKWARD (LOCAL)
REDUCE-SCATTER
FREE FULL WEIGHTS
FSDP instance 1: N layers
FSDP instance N: N layers
UPDATE WEIGHTS (LOCAL)
Load shard From CPU if CPU Offloaded
Offload grads to CPU if CPU offload is enabled

No Algorithmic Changes: Implementation-Level Comms/Compute Overlap

- Let a network perform a forward pass via F_1, F_2, a backward pass B_1, B_2 and an optimizer step via `Opt`.



- **Overlap via hooks.** Using backward hooks (e.g., `prepare_for_backward)`, a standard technique is to launch communication for a layer's gradients immediately after that layer's backward computation completes, overlapping communication with remaining compute.

Expert Parallelism

Mixture of Experts

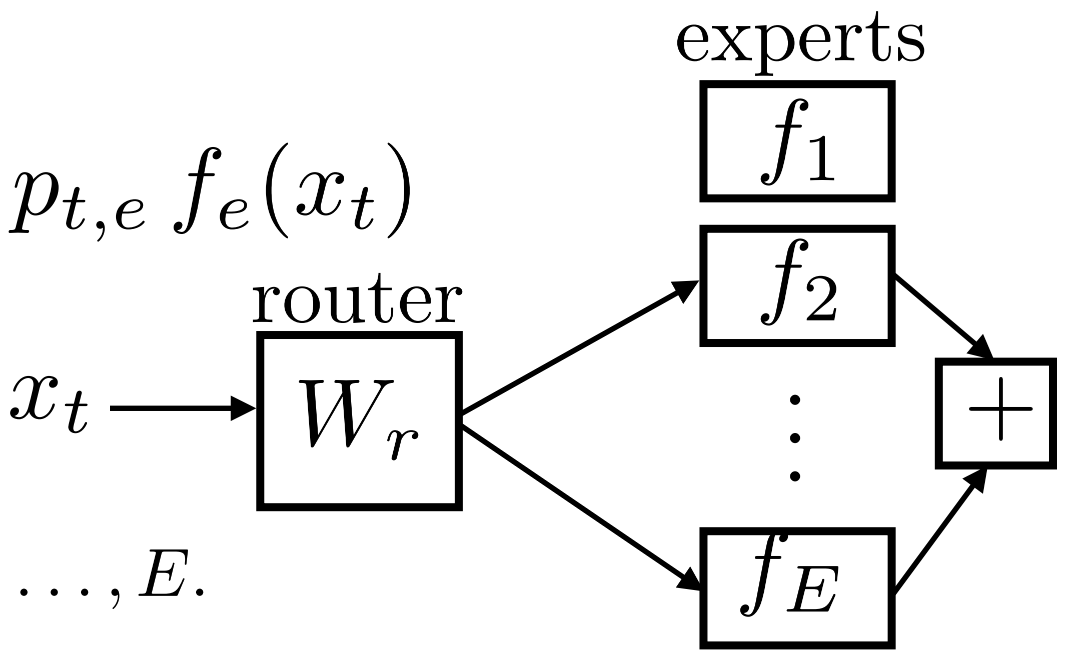
- In Transformer blocks, the FFN/MLP accounts for most FLOPs per token. We want to increase parameter count while keeping per-token FLOPs constant (i.e., reducing FLOPs per parameters).

- Idea: **Mixture-of-Experts (MoE)** = many FFN "experts", but **activate only k-times per token**

- FLOPs/token $\approx k \cdot \mathrm{FLOPs(FFN.)}$ (k is small, often 1–2)

$$p_t = \mathrm{softmax}(\mathrm{Top}_k(W_r x_t)) \quad y_t = \sum_{e=1}^{E} p_{t,e}\, f_e(x_t)$$

$$[\mathrm{Top}_k(v)]_i = \begin{cases} v_i, & \text{if } |\{j:\ v_j > v_i\}| < k, \\ -\infty, & \text{otherwise}, \end{cases} \qquad i = 1, \ldots, E.$$



- "Expert" is just naming: **it's an FFN branch**, not a model specialist by default: it's unclear if specialisation occurs.

Formula to compute gradients

- We begin with

$$p_t = \mathrm{softmax}(\mathrm{Top}_k(W_r x_t)) \quad \text{and} \quad y_t = \sum_{e=1}^{E} p_{t,e}\, f_e(x_t)$$

- From $g_t \triangleq \dfrac{\partial L}{\partial y_t}$ we derive

This part here can be optimised assuming the expert is a FFN. (SONIC by Tri Dao)

$$\frac{\partial L}{\partial \theta_e} = \sum_t p_{t,e} \left(\frac{\partial f_e(x_t)}{\partial \theta_e}\right)^{\top} g_t.$$

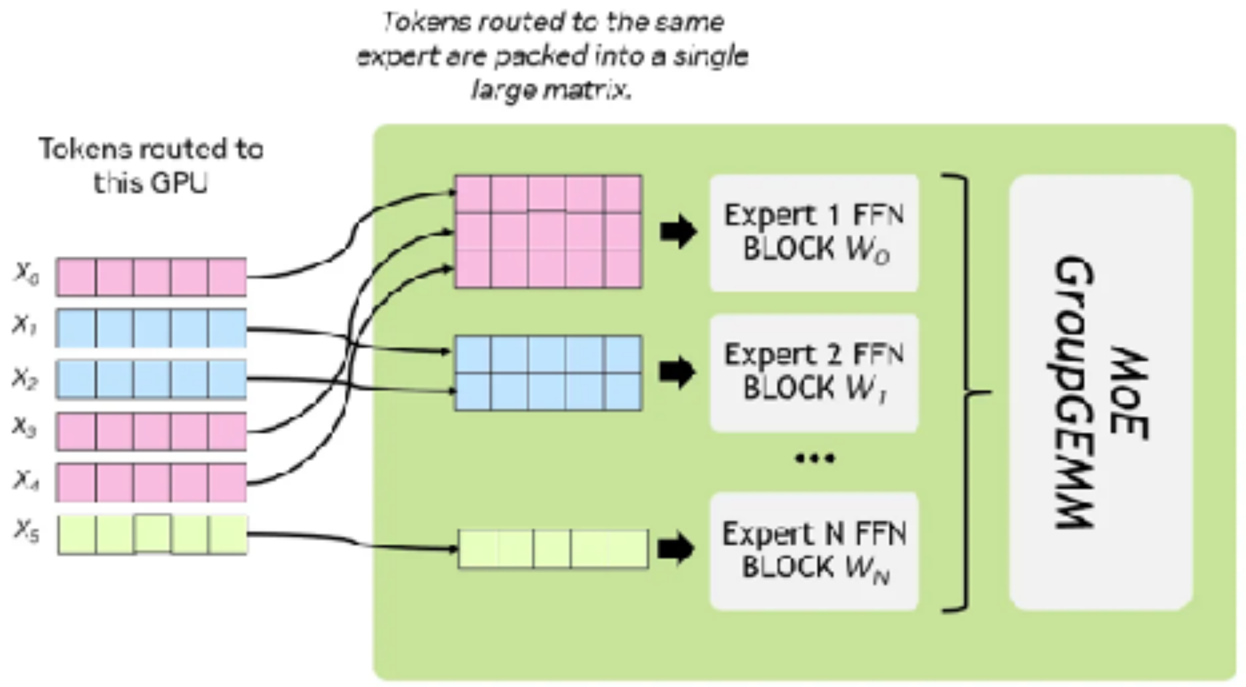
$$\frac{\partial L}{\partial W_r} = \sum_t \left(p_{t,e}\left(a_{t,e} - \sum_{j=1}^{E} p_{t,j}\, a_{t,j}\right)\right) x_t^{\top} \text{with}$$

$$\boxed{\begin{aligned}\frac{\partial L}{\partial p_{t,e}} &= g_t^{\top} f_e(x_t) \\ &\triangleq \quad a_{t,e}.\end{aligned}}$$

$$\frac{\partial L}{\partial x_t} = \underbrace{\sum_{e=1}^{E} p_{t,e} \left(\frac{\partial f_e(x_t)}{\partial x_t}\right)^{\top} g_t}_{\text{expert path}} + \underbrace{W_r^{\top} \frac{\partial L}{\partial z_t}}_{\text{router path}},$$

$$\text{with } \frac{\partial L}{\partial z_{t,e}} = p_{t,e} \left(a_{t,e} - \sum_{j=1}^{E} p_{t,j}\, a_{t,j}\right)$$

MoE Communications



- Experts are sharded across devices: each device stores only a subset of experts (weights are not replicated), justifying the name **Expert Parallelism**.
- **Tokens are routed to experts**: each device starts with local hidden state then **dispatches** token embeddings to the devices hosting the selected experts.
- **Communication-heavy step**: dispatch/return requires **all-to-all exchange of activations** (tokens move to experts, then outputs move back).
- **Compute happens where the expert lives**: each device runs its local expert FFNs on the received tokens, then results are **combined** and restored to original token order.

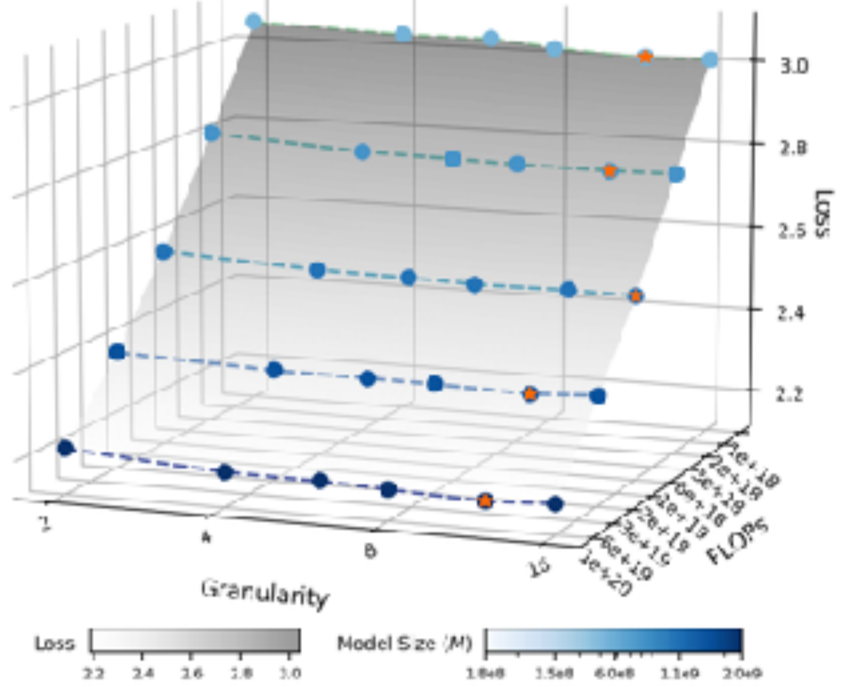
Balancing Expert Utilization

- Without explicit balancing, the router can **collapse**: a few experts get most tokens while others are under-used ("dead experts").

- This creates **stragglers** (slow experts) and hurts both **quality** and **throughput**. There exists strategies to mitigate this:
  - **Thresholding:** limit tokens per expert; overflow tokens are **dropped, rerouted, or sent to a fallback/shared expert**.
  - **Auxiliary load-balancing loss:** encourage more uniform expert usage, typically using $$\mathcal{L}_{\text{aux}} \propto \sum_{e=1}^{E} f_e p_e$$
    - **importance** = average routing probability mass per expert $$p_e = \sum_{t=1}^{T} s_{e,t}$$
    - **load** = fraction/number of tokens actually assigned to each expert $$f_e = \sum_{t=1}^{T} \mathbf{1}[\,p_{t,e} > 0\,].$$

Scaling laws of experts

- Fix the total number of MoE parameters P, the number of experts n_{ep}, and the number of activated parameters per token P_A .
- Those parameters do not determine expert size; expert size is set by the **granularity**, defined as
$$G \triangleq \frac{n_{\mathrm{EP}} \cdot P}{P_A}$$
- Equivalently, G can be define via: the FFN hidden size h_{FFN} of a reference dense model versus the FFN hidden size $h_{\mathrm{FFN}}^{(\mathrm{expert})}$ of an expert, yielding
$$G = \frac{h_{\mathrm{FFN}}}{h_{\mathrm{FFN}}^{(\mathrm{expert})}}.$$
- **Scaling-law lens:** MoE behavior differs from dense models; analyze performance as a function of P, D (training tokens), and G, e.g.



$$\min_{P,D,G} \; L(P,D,G) \quad \text{s.t.} \quad \mathrm{FLOPs}(P,D,G) = C.$$

EP Summary

- **What's sharded:** Experts (FFNs) are partitioned across GPUs; each GPU stores a subset of experts.
- **Memory footprint**: Per-GPU expert parameters + optimizer state.
- Communications:
  - **Pattern**: all-to-all to dispatch token activations to the GPUs hosting selected experts, then all-to-all to return/combine expert outputs (and similarly in backward).
  - **Volume**: scales with the number of routed of tokens, activated experts and size.
- When it bottlenecks:
  - **Comm-bound**: all-to-all dominates (large number of tokens, hidden dimension, number of MoE will be restricted by a poor interconnect).
  - **Imbalance-bound**: routing is uneven (some experts/GPUs get many more tokens) and per-expert token batches are irregular.
- **Primary goal**: MoE scale up by increasing total parameters without replicating experts everywhere; enables larger models at similar FLOPs/token as a dense counter-part.
- **Statistical efficiency**: can be affected by routing (load-balancing), but EP is mainly a systems choice; most training dynamics changes come from the MoE/router design.

Back to TorchTitan

Interaction with `torchtitan`

- To guarantee composability, frameworks like `torchtitan` enforce applying parallelism/optimizations in a specific sequence (each step assumes the previous transforms are already in place).
- Where this is encoded: in TorchTitan, `infra/parallelize.py` is organized around this recommended application order:
  - TP (and EP for MoE models)
  - Activation checkpointing
  - torch.compile
  - HSDP
- **Intuition**: partition the model first (TP/EP), reduce activation memory next (checkpointing), compile once the graph is stable, then add sharding (HSDP) after module boundaries are finalized.

Conclusion

- Today's lab will study PP!