Lect 8: Parallelization

Sham Kakade and Nikhil Anand

CS 2281: How to Train Your Foundation Model

Fall 2024

Today

- Recap++:
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

Recap++

Today

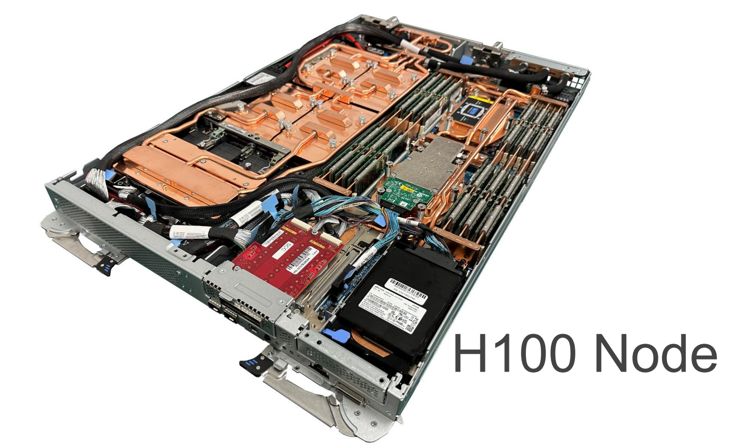
- Recap++:

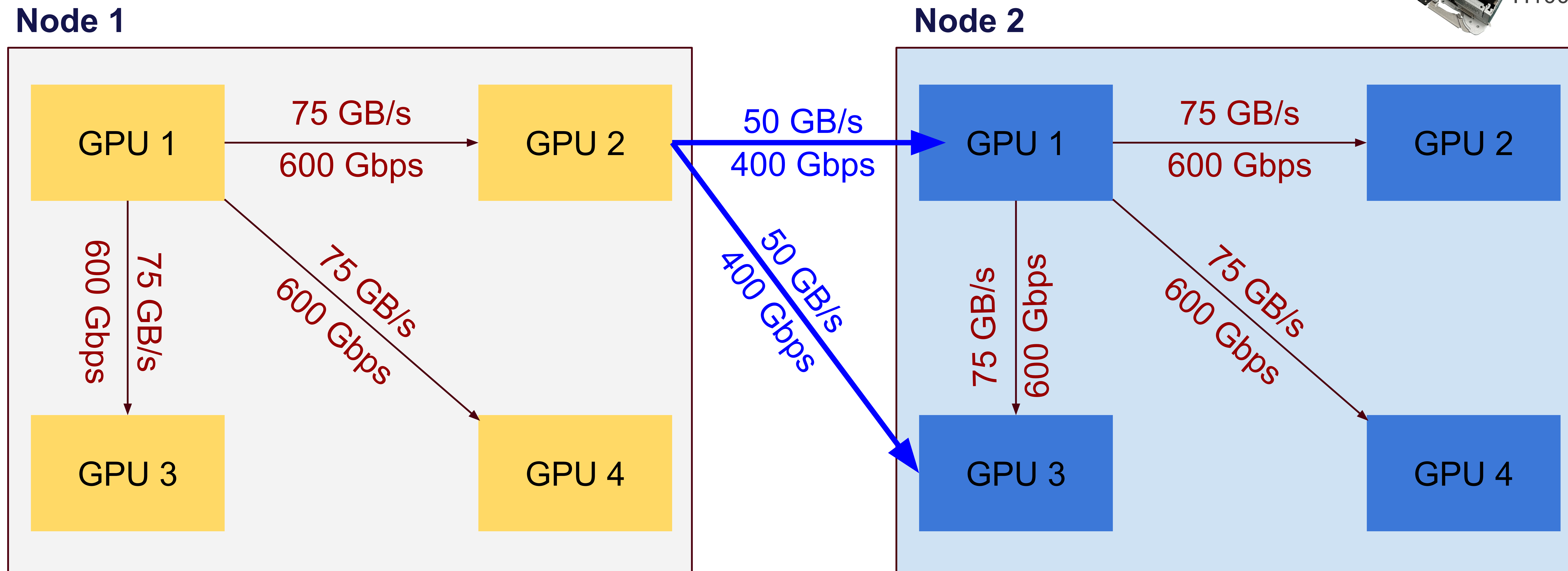
  - hardware
  - communication primitives
  - Distributed Data Parallelism (DDP)
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

Scales of Some Foundation Models
note: 1b, 8b, 70B, 405B, “big”

- A100/H100: let’s say 80gb memory on each chip.
- LLMs (??):
  - GPT4.0: 1.6T (8x200B MoE model), ≈10T tokens, (flop equiv) 30K for several months
  - GPT4o: 16(or 32)x30B MoE model, ≈30T text tokens,
  - Gemini: 2T param model (also MoE?), ≈10T tokens (trained on TPUs)
  - **Llama 3.1**: 8B, 70B, 405B (dense), ≈10T tokens
- Code: Copilot ≈10-20B (?),
- Images/Video: MidJourney/Sora ≈10-20B (?), 10K gpu for 1 month (?)
- Bio: AlphaFold

GPU-to-GPU Communication





Inside Node (**NVLINK**): Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed. 75 GB/s * 6 * 2 = 900 GB/s

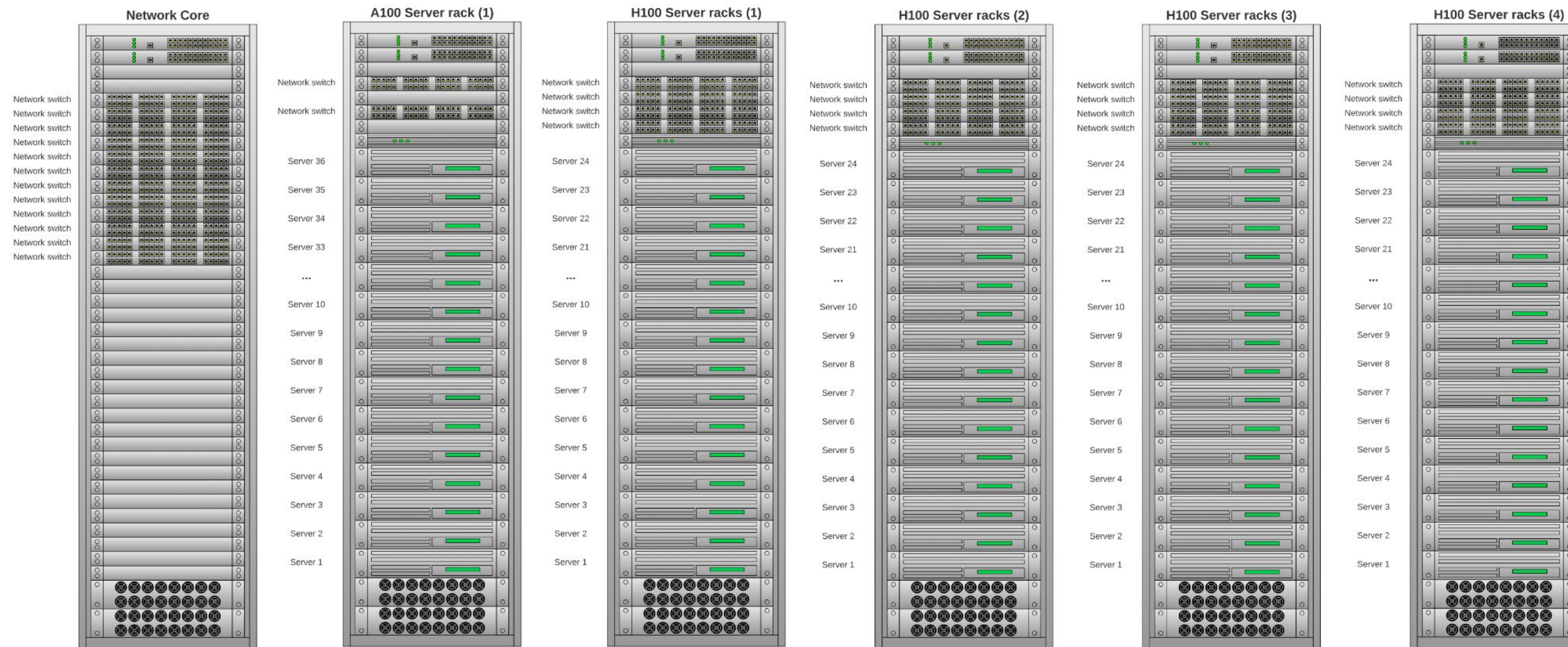
Outside Node (**InfiniBand Network NDR**): Each GPU communicates to other GPUs in another node at 400 Gbps (50 GB/s).

Kempner INSTITUTE | HARVARD UNIVERSITY

Modern nodes: usually **8 gpus per node.**
Intra node communication (**FAST**): nvlink, basically **1tb/sec (total)**
Inter node communication (**SLOW**): infiniband, **50GB/sec**
Nodes connected in a “tree” structure.

[Figure credit: Yasin Mazloumi]

HPC Cluster



In Production (144 x A100 40 GB, 384 x H100 80 GB)

Kempner INSTITUTE | HARVARD UNIVERSITY

Nodes sit in racks.
Intra node communication (**FAST**): nvlink, basically **1tb/sec (total)**
Inter node communication (**SLOW**): infiniband, **50GB/sec**
Nodes connected in a “tree” structure.

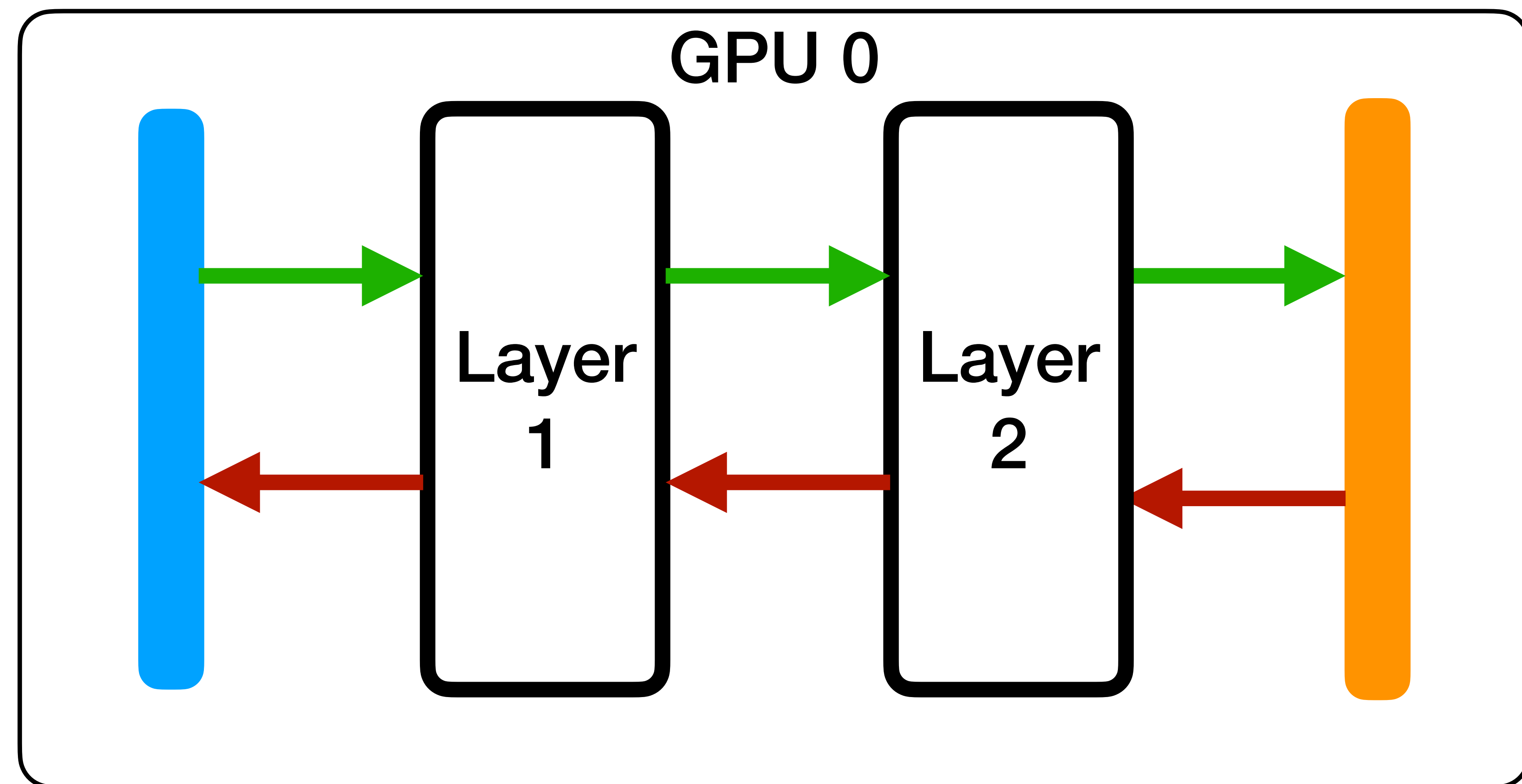
[Figure credit: Yasin Mazloumi]

Kempner Racks

TPUs

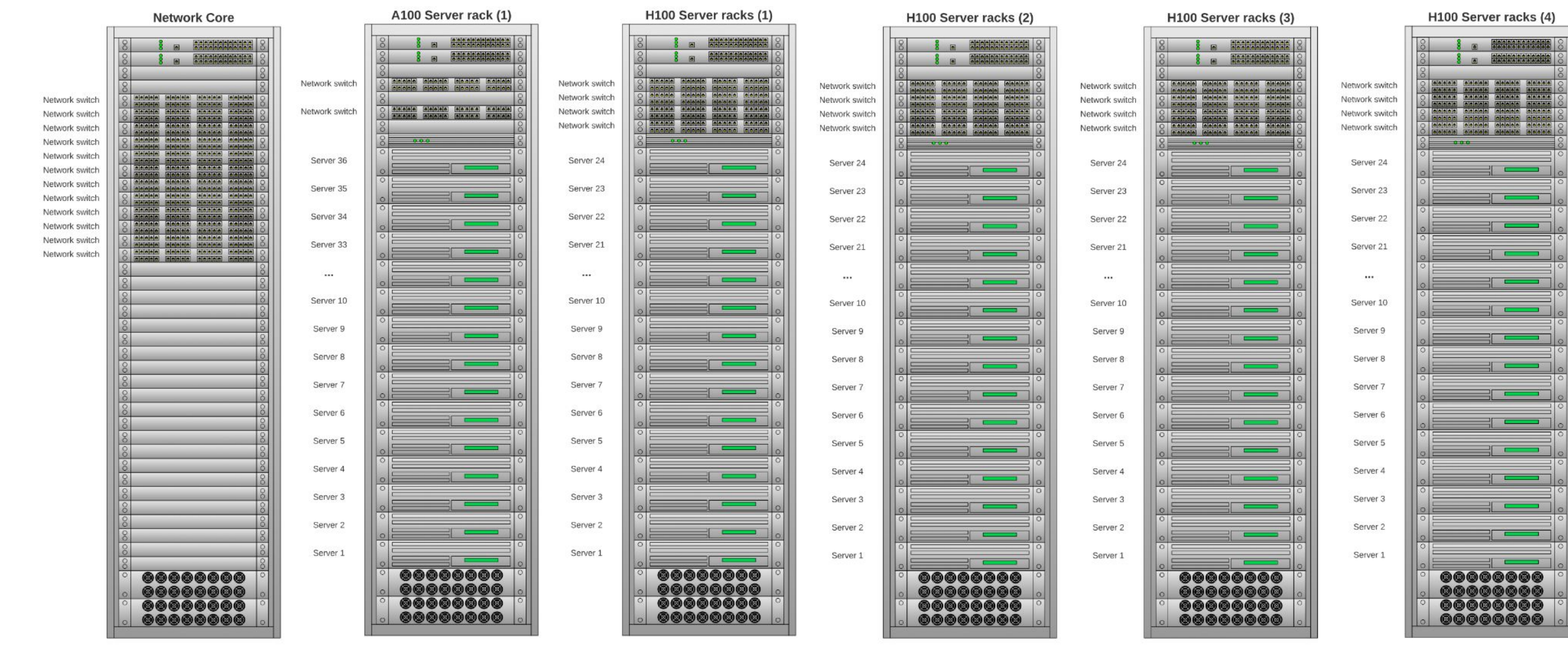
These are very very nice, because all the “Processing Elements” (basically GPUS) are connected point-to-point (with very fast communication).

Today: Single GPU -> Distributed Training



$$M_{\text{train}} = P_{\text{transformer}} + M_{\text{optimizer}} + M_{\text{activations}}$$

HPC Cluster



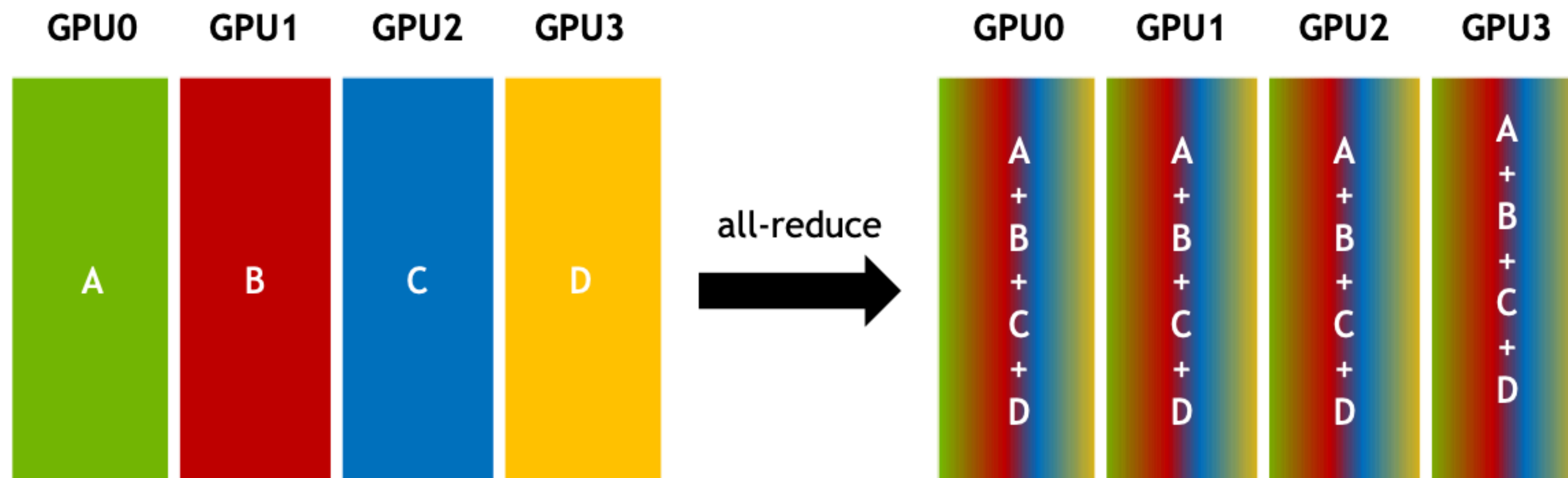
In Production (144 x A100 40 GB, 384 x H100 80 GB)

Today

- Recap++:
  - hardware
  - ✓ communication primitives
  - Distributed Data Parallelism (DDP)
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

NCCL communication primitives

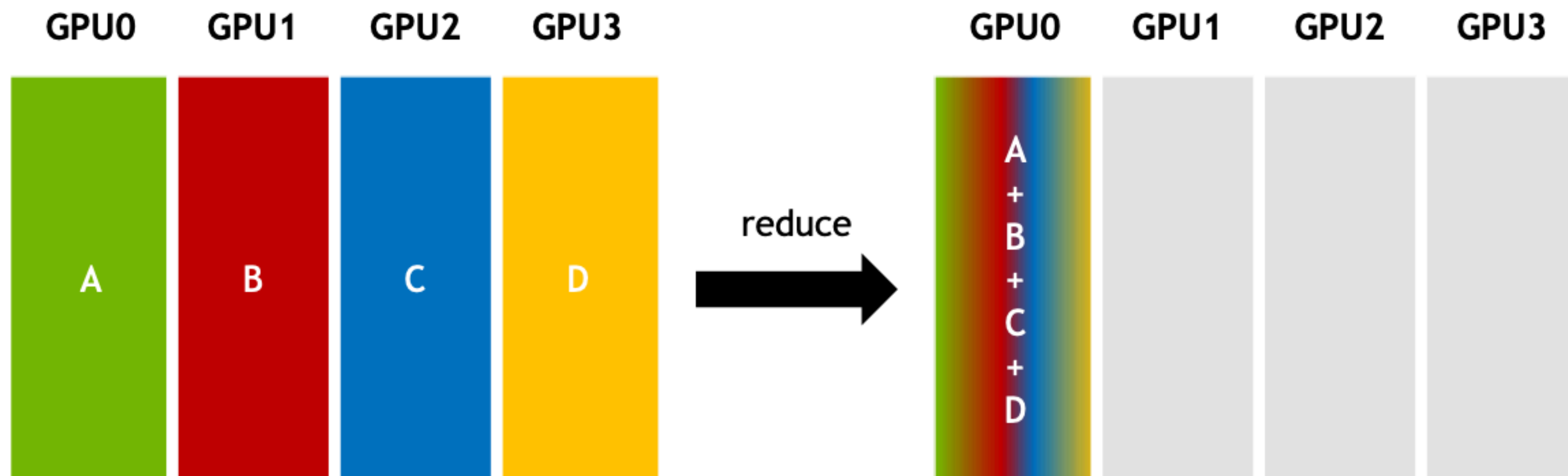
AllReduce



There are many possible implementations! E.g., compute in a ring

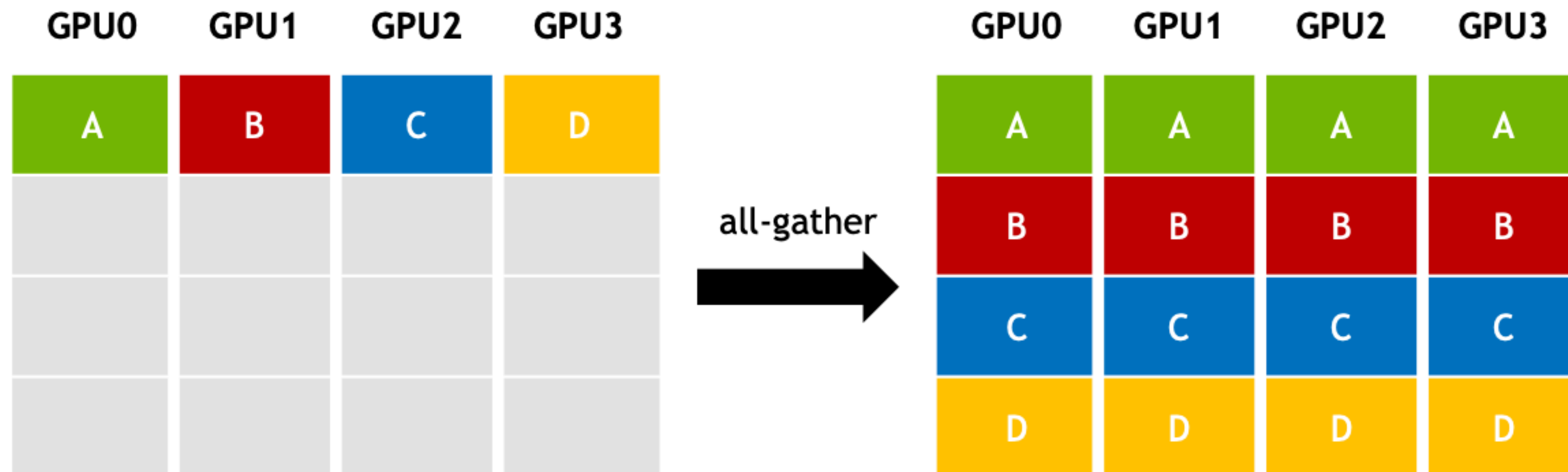
NCCL communication primitives

Reduce



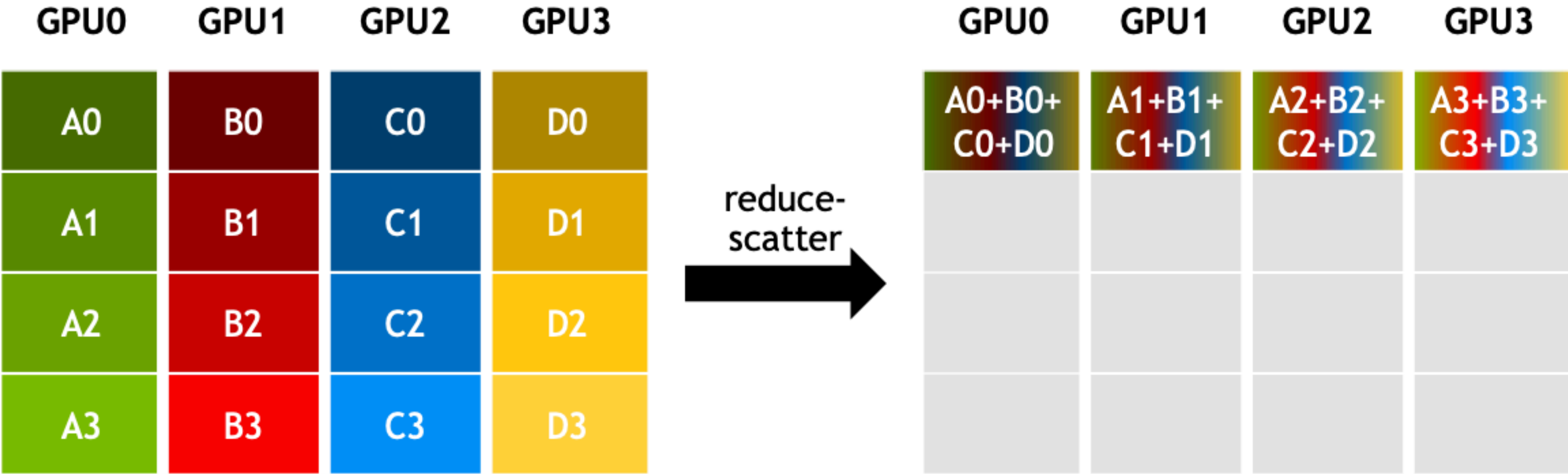
NCCL communication primitives

AllGather



NCCL communication primitives

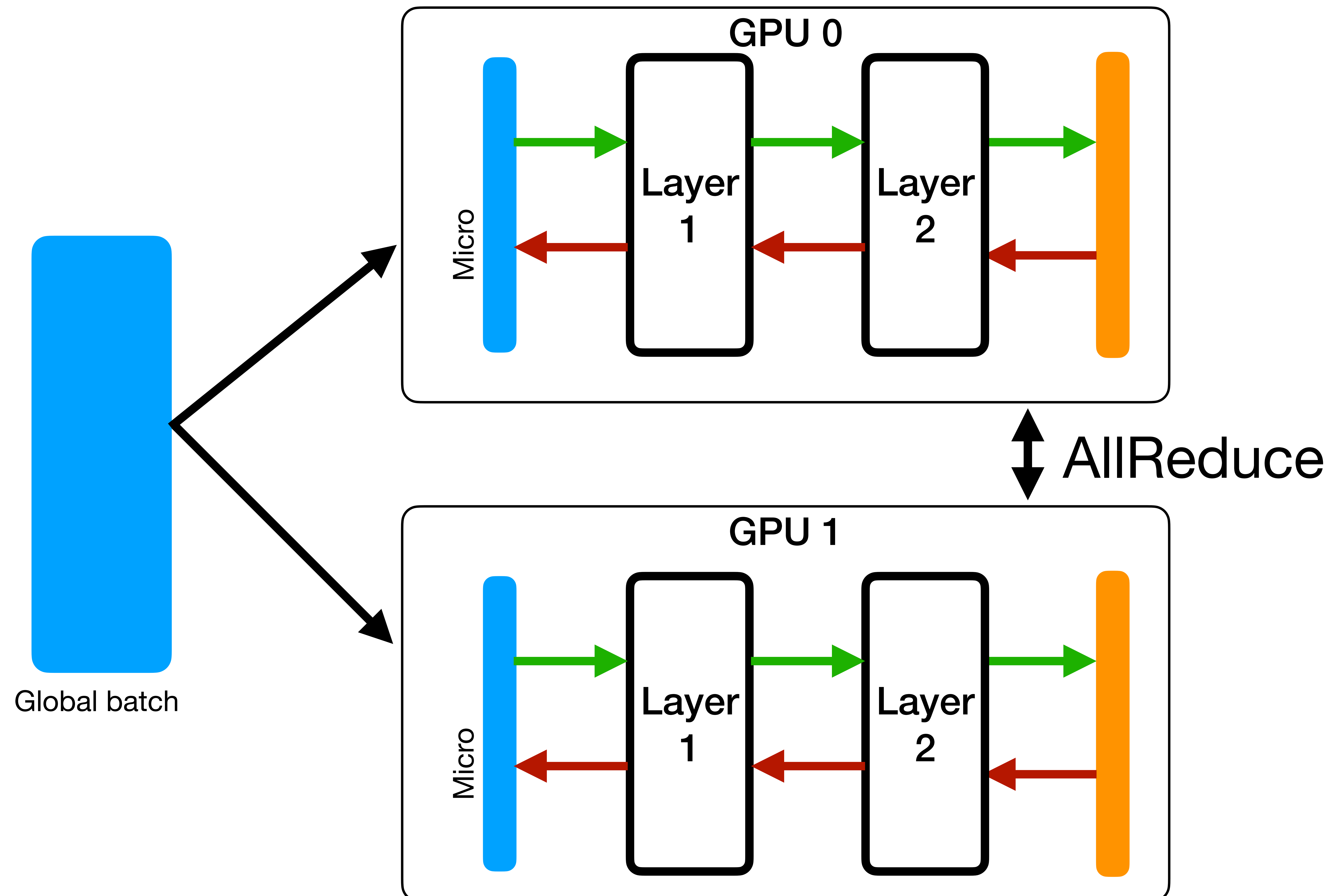
ReduceScatter



Today

- Recap++:
    - hardware
    - communication primitives
    - ✓ Distributed Data Parallelism (DDP)
- Pipeline Parallelism
- FSDP
- Tensor Parallelism

Multi GPU Training: DDP



When is DDP useful?

- When a model fits on a single GPU, and we want to increase data throughput i.e. train faster
- When it makes sense to keep inter-GPU communication as simple as possible (e.g., smaller scale experiments)
- Models that are large enough that cannot be fit on a single GPU are trained with other distributed frameworks (FSDP, etc.)

Today

- Recap++:
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

Before this, let's consider linear nets as our model (i.e. matrix multiplies)

Our Computational Model

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,
    layers have input/output dimension d, batch size is m

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^{0} X$
  - $W^{\ell} \in R^{d \times d}, X \in R^{d \times m}$,

    layers have input/output dimension d, batch size is m
- Written with activations:

  $A^{0} = X$

  $A^{\ell+1} = W^{\ell} A^{\ell}$

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,

    layers have input/output dimension d, batch size is m
- Written with activations:

  $A^0 = X$

  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:

  depth/number of layers: L, width/hidden dim: d, batch size: m

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,

    layers have input/output dimension d, batch size is m
- Written with activations:

  $A^0 = X$

  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:

  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,
    layers have input/output dimension d, batch size is m
- Written with activations:
  $A^0 = X$
  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:
  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD
  - Parameters: Ld^2

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,

    layers have input/output dimension d, batch size is m
- Written with activations:

  $A^0 = X$

  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:

  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD
  - Parameters: Ld^2
  - Flops for forward pass: $Ld^2 m$

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,
    layers have input/output dimension d, batch size is m
- Written with activations:
  $A^0 = X$
  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:
  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD
  - Parameters: Ld^2
  - Flops for forward pass: Ld^2m
  - Activation memory: Ldm

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,
    layers have input/output dimension d, batch size is m
- Written with activations:
  $A^0 = X$
  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:
  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD
  - Parameters: Ld^2
  - Flops for forward pass: Ld^2m
  - Activation memory: Ldm
- Important: note that param memory $\gg$ activation memory, when $d \gg m$

Our Computational Model

- (batch) Linear Network: $Y = W^{L-1} \dots W^0 X$
  - $W^\ell \in R^{d \times d}, X \in R^{d \times m}$,

    layers have input/output dimension d, batch size is m
- Written with activations:

  $A^0 = X$

  $A^{\ell+1} = W^\ell A^\ell$
- Scalings:

  depth/number of layers: L, width/hidden dim: d, batch size: m
- Compute/Memory for reverse Mode AD
  - Parameters: Ld^2
  - Flops for forward pass: Ld^2m
  - Activation memory: Ldm
- Important: note that param memory $\gg$ activation memory, when $d \gg m$
- Other considerations:
  - Total flops: # gpus * time gpus run for * flops/gpu
  - Serial runtime: we want our job to finish soon (in a few months?)

Now lets look at LLama 3.1

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function		SwiGLU	
Vocabulary Size		128,000	
Positional Embeddings		RoPE ($\theta = 500,000$)	

Table 3 Overview of the key hyperparameters of Llama 3. We display settings for 8B, 70B, and 405B language models.

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function		SwiGLU	
Vocabulary Size		128,000	
Positional Embeddings		RoPE ($\theta = 500,000$)	

Table 3 Overview of the key hyperparameters of Llama 3. We display settings for 8B, 70B, and 405B language models.

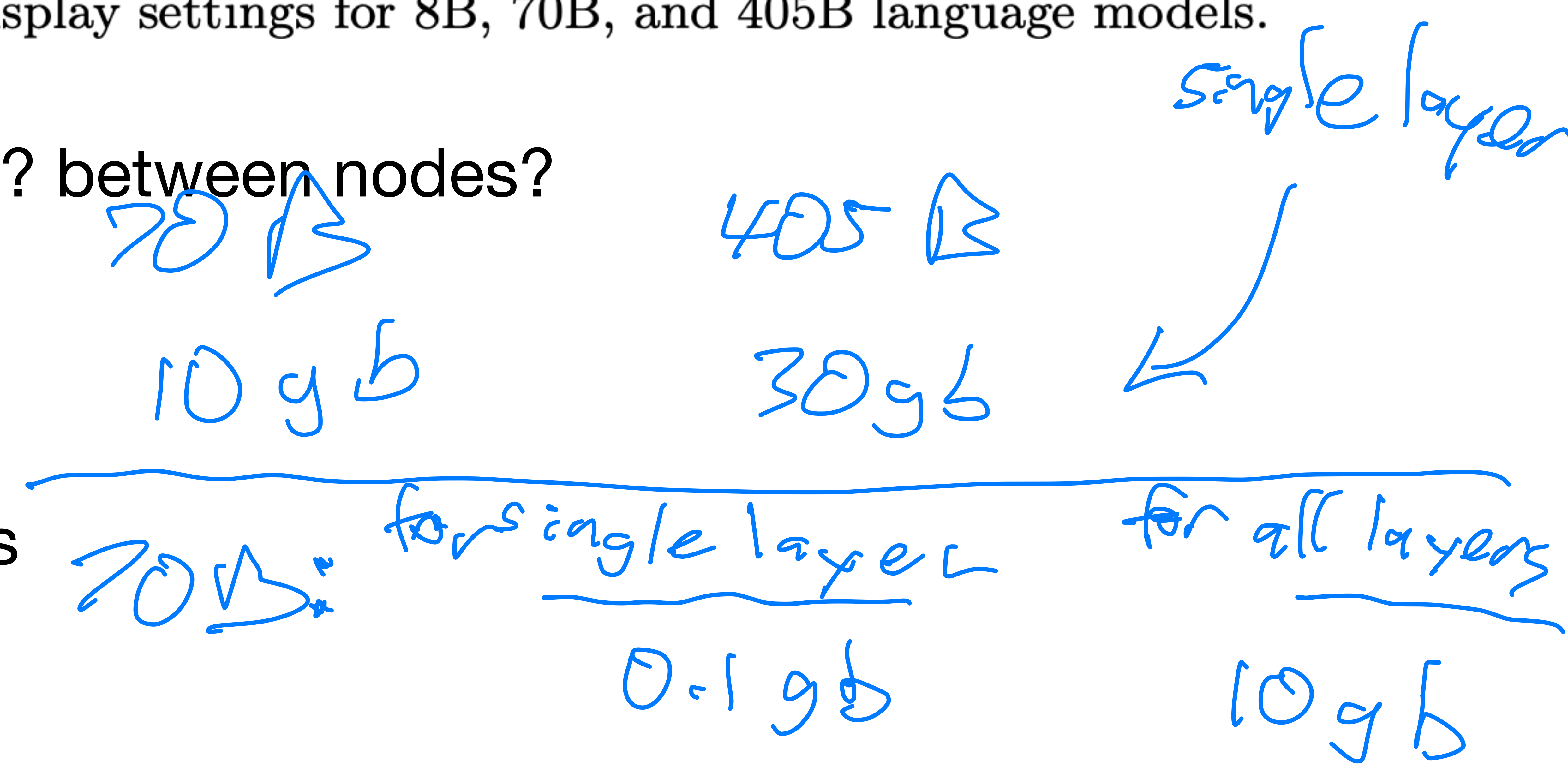
- Concepts: what fits on a gpu? on a node? between nodes?

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function		SwiGLU	
Vocabulary Size		128,000	
Positional Embeddings		RoPE ($\theta = 500,000$)	

Table 3 Overview of the key hyperparameters of Llama 3. We display settings for 8B, 70B, and 405B language models.

- Concepts: what fits on a gpu? on a node? between nodes?
- Params/transformer block: $10d^2$
  - Bytes/block: $2 \cdot 10d^2$
- Activation Memory for batch size 1:

  $2 \cdot \text{seq_length} \cdot d \cdot \text{num_layers}$ bytes



GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	4	131,072	16	16M	380	38%

Table 4 Scaling configurations and MFU for each stage of Llama 3 405B pre-training. See text and Figure 5 for descriptions of each type of parallelism.

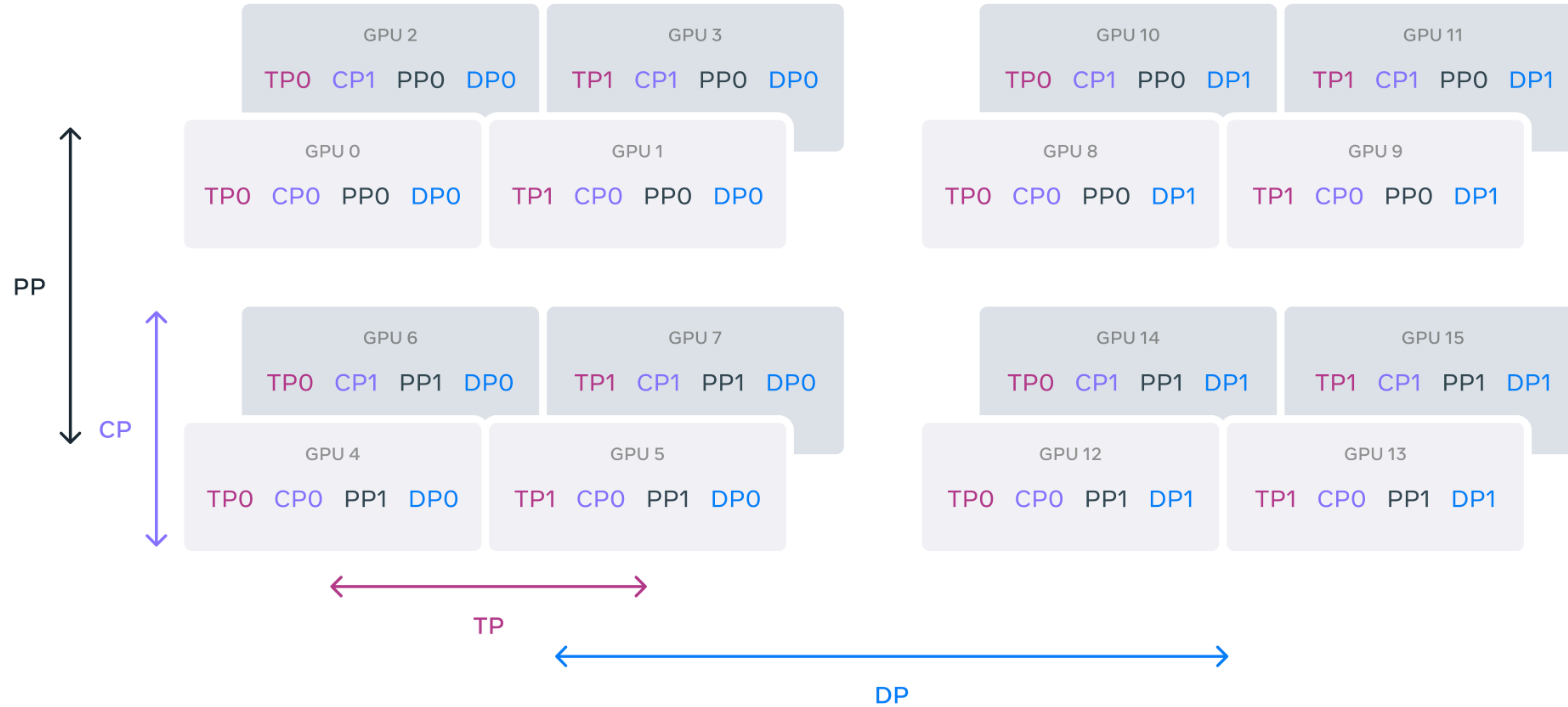
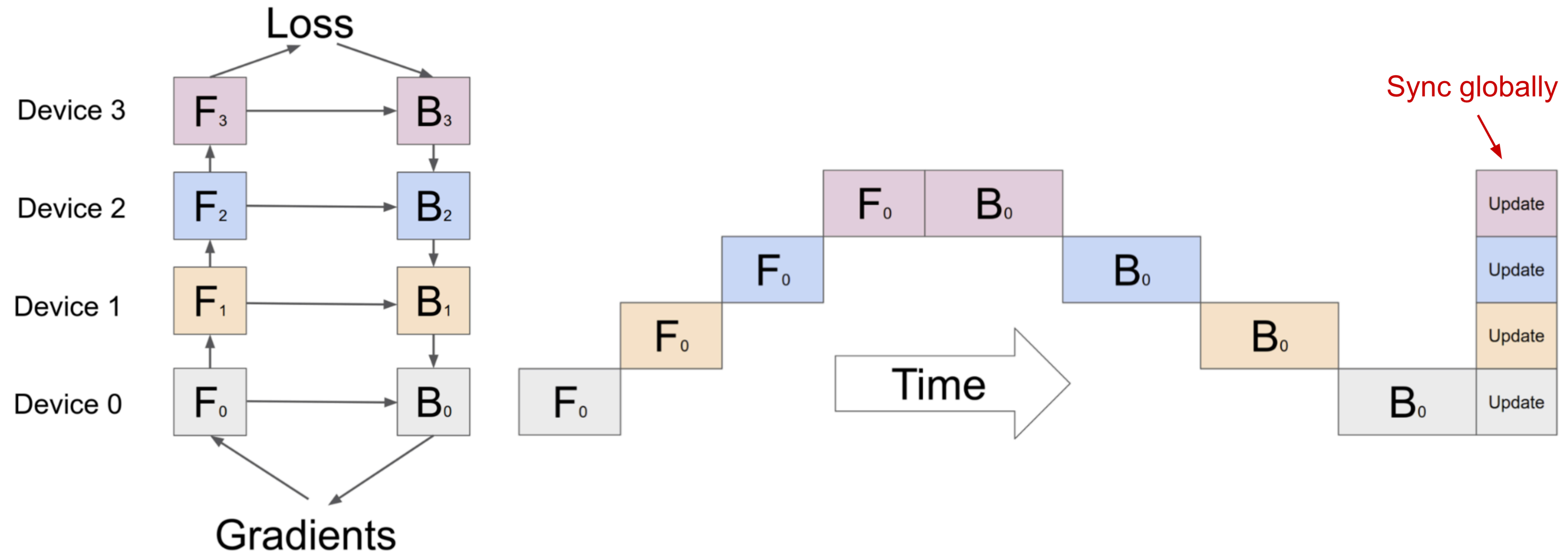


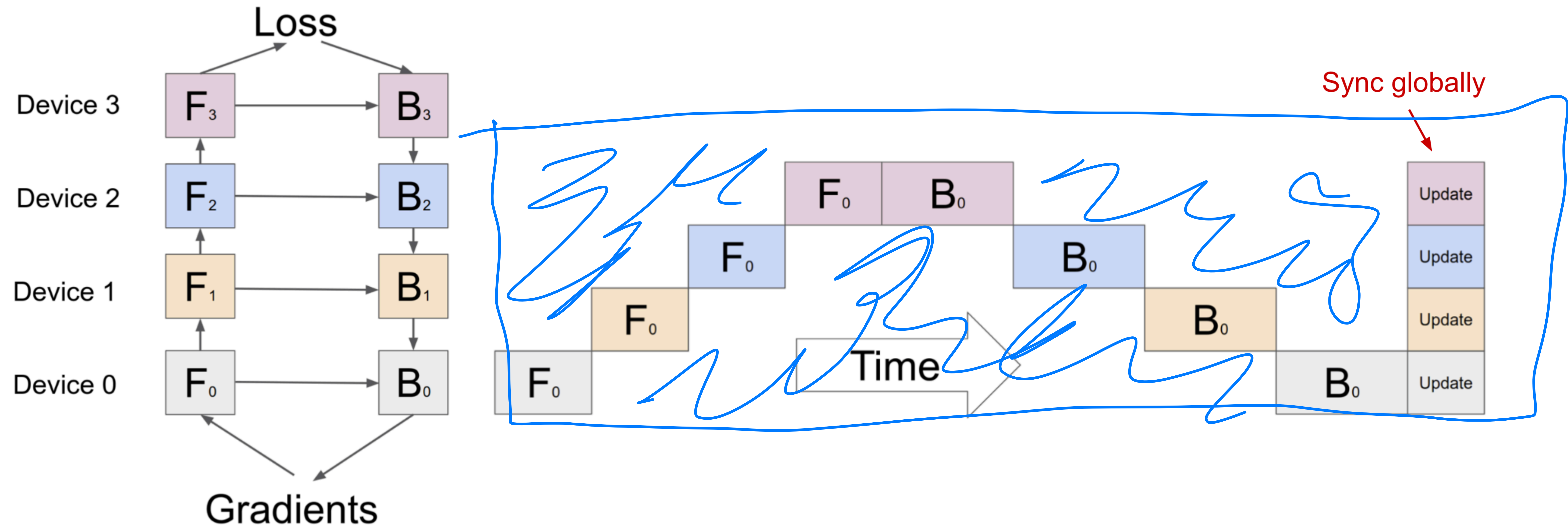
Figure 5 Illustration of 4D parallelism. GPUs are divided into parallelism groups in the order of [TP, CP, PP, DP], where DP stands for FSDP. In this example, 16 GPUs are configured with a group size of |TP|=2, |CP|=2, |PP|=2, and |DP|=2. A GPU's position in 4D parallelism is represented as a vector, $[D_1, D_2, D_3, D_4]$, where D_i is the index on the i-th parallelism dimension. In this example, GPU0[TP0, CP0, PP0, DP0] and GPU1[TP1, CP0, PP0, DP0] are in the same TP group, GPU0 and GPU2 are in the same CP group, GPU0 and GPU4 are in the same PP group, and GPU0 and GPU8 are in the same DP group.

Pipeline Parallelism

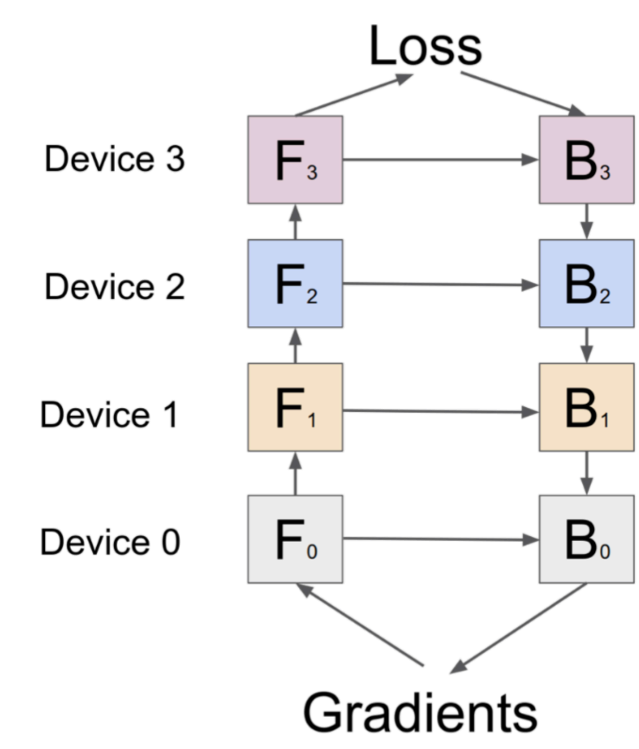
Naive Model Parallelization



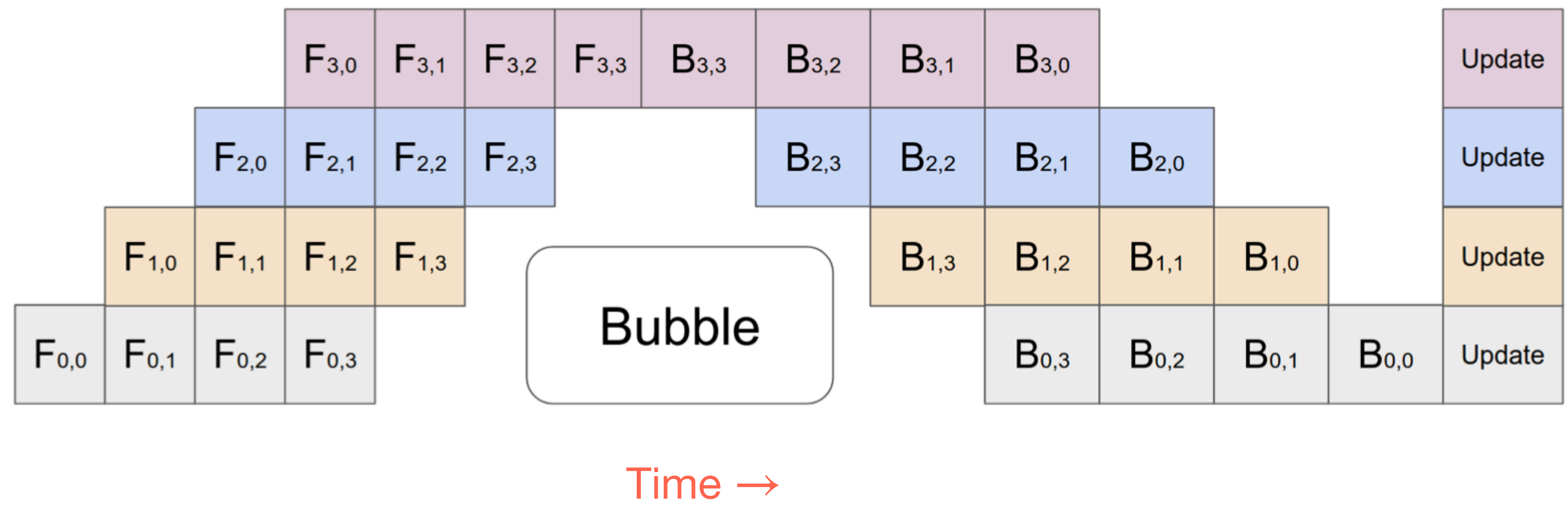
Naive Model Parallelization

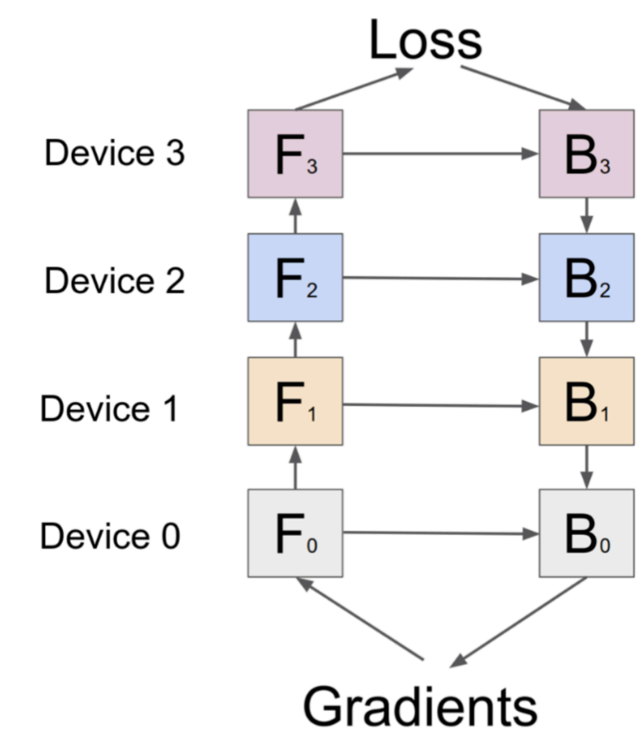


- Shard model into k partitions, e.g. $k = 4$, $W^{0:L/4}$, $W^{L/4+1:L/2}$, $W^{L/2:3L/4}$, $W^{3L/4:L-1}$
- Assume: each device can hold a model partition, which is of size $Ld^2/4$
- **What is the problem?**

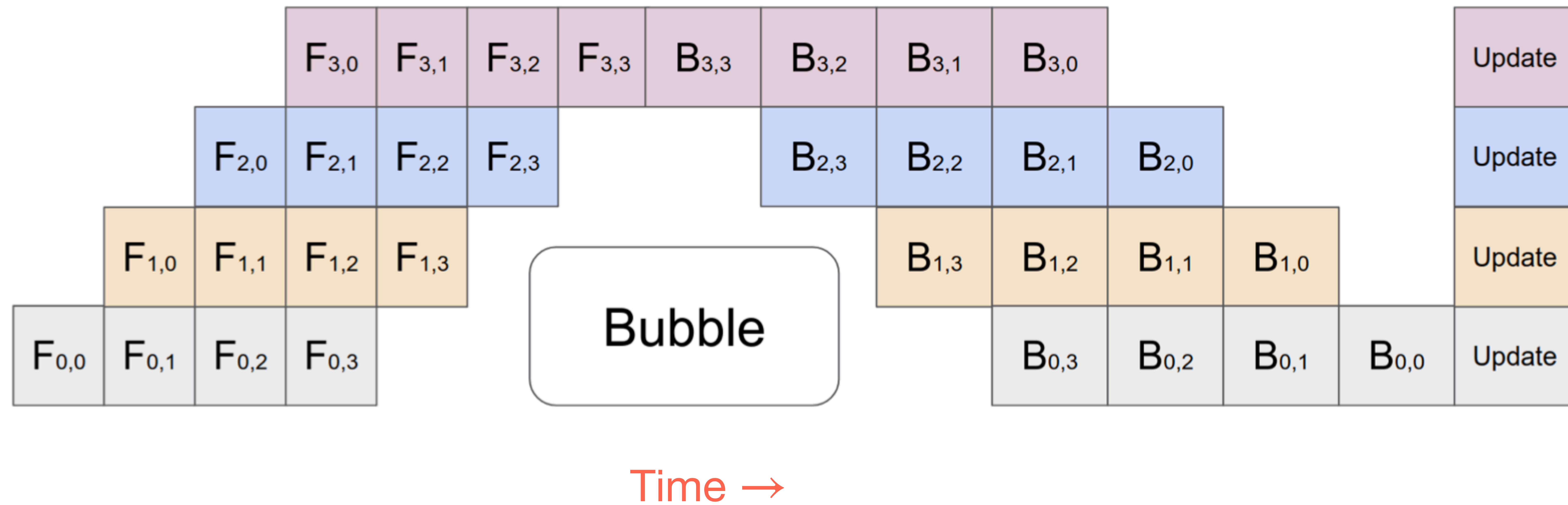


The pipeline approach:



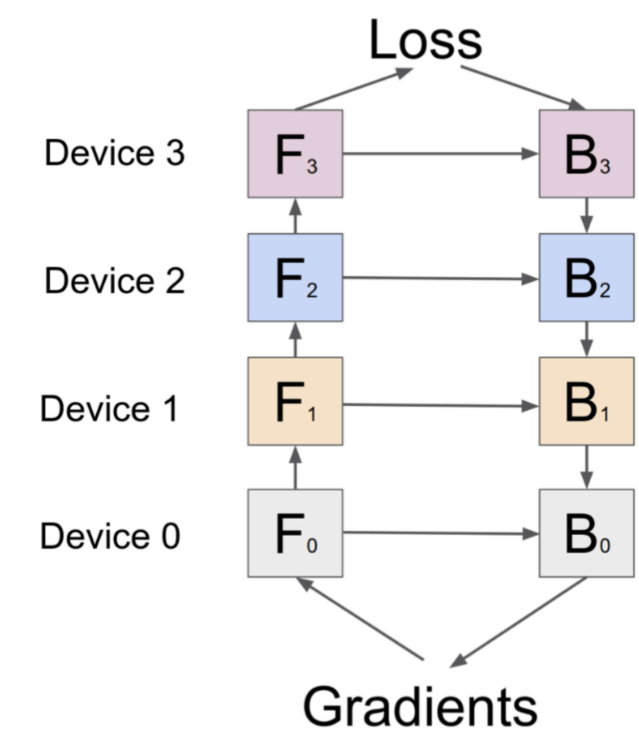


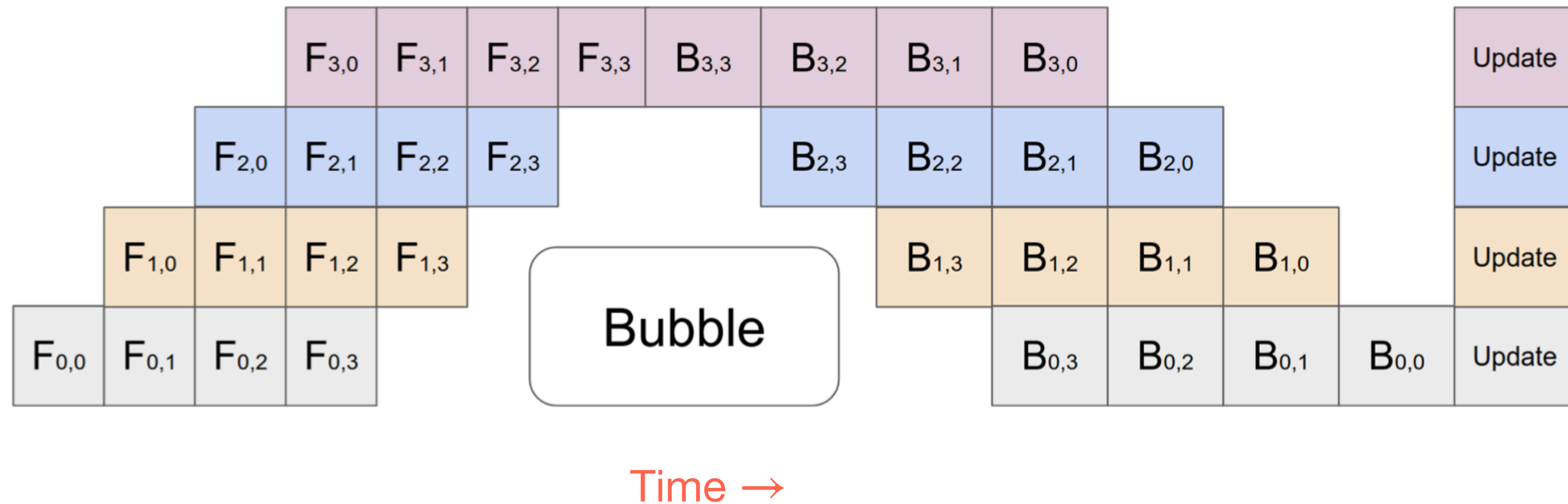
The pipeline approach:



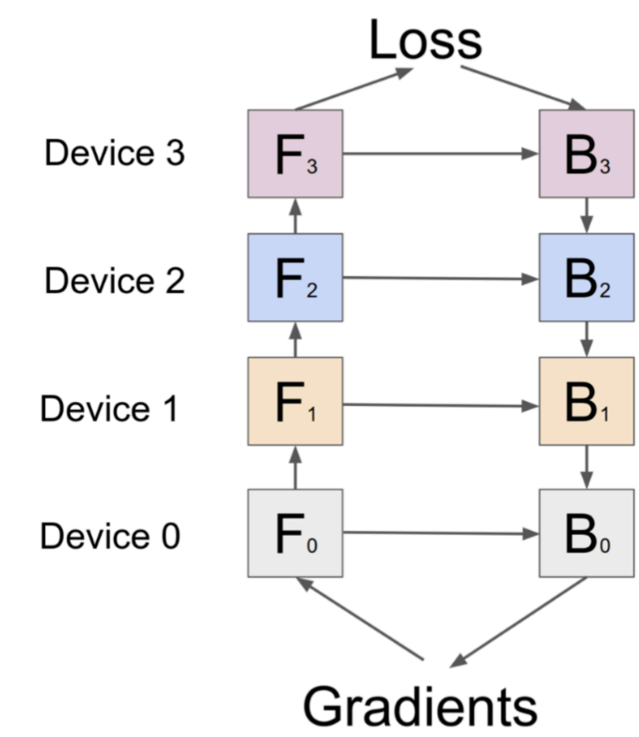
- Shard model into p partitions and place these on p devices, i.e. $p = 4$ example, where device 0 contains $W^{0:L/4}$, device 1 contains $W^{L/4+1:L/2}$, …
- Split mini-batch T, of size m, into k “micro-batches”, $\{T_0, \ldots, T_{k-1}\}$, each of size m/k

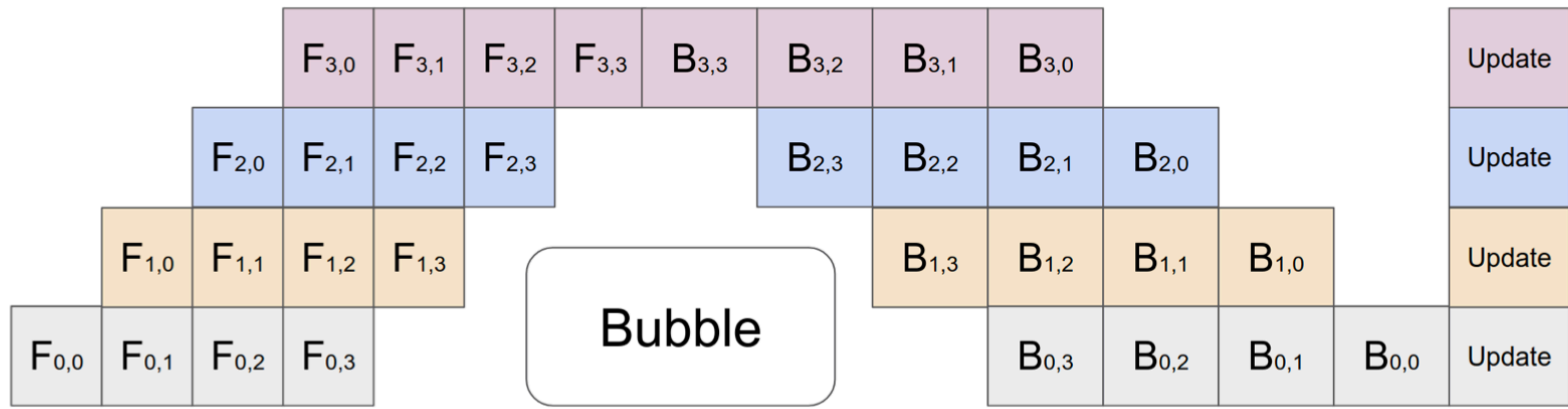
The pipeline approach:

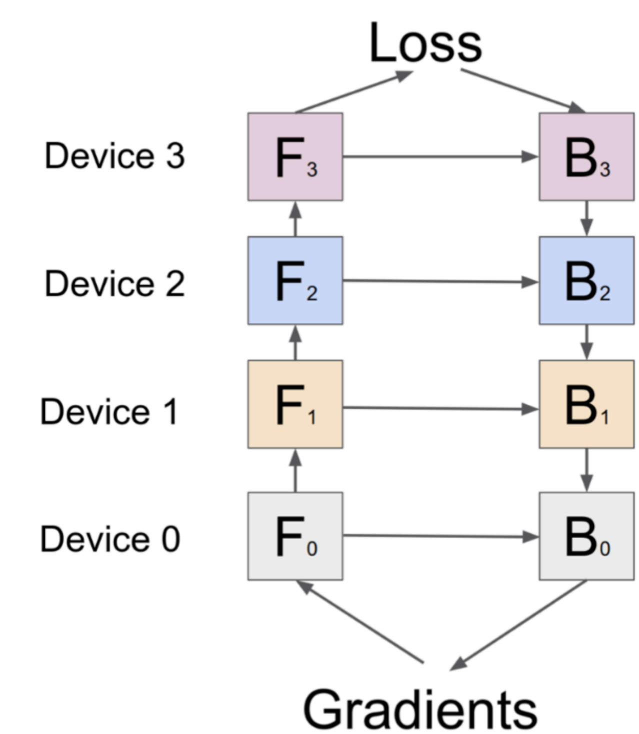


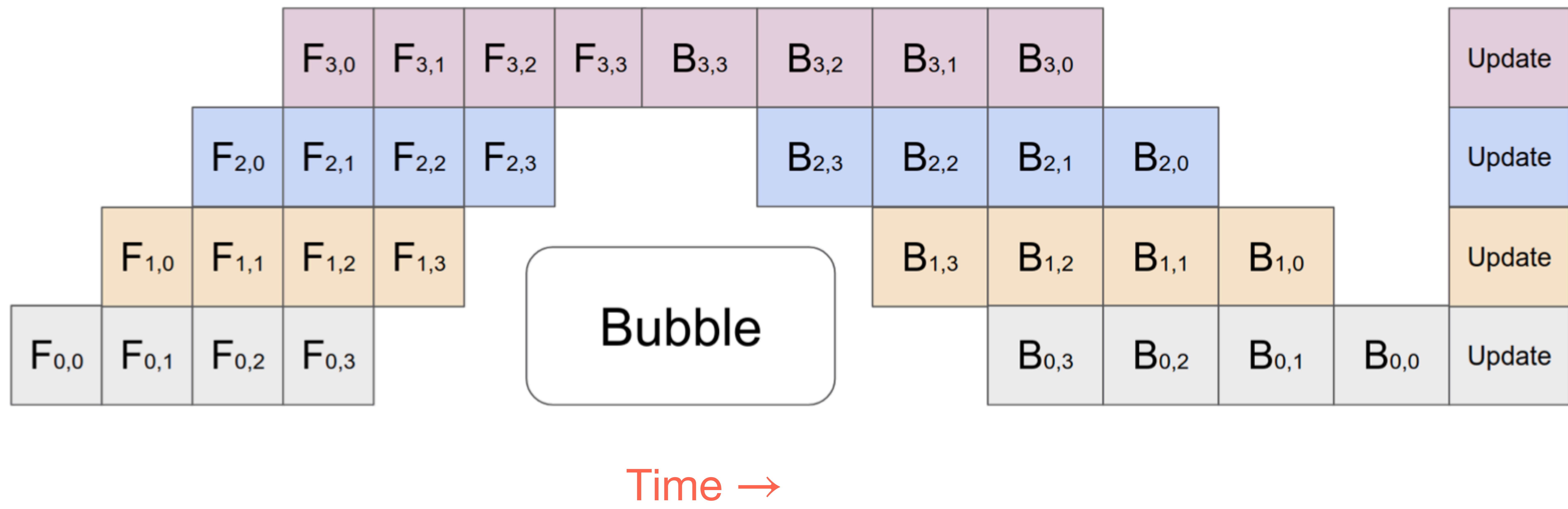


- Shard model into p partitions and place these on p devices, i.e. $p = 4$ example, where device 0 contains $W^{0:L/4}$, device 1 contains $W^{L/4+1:L/2}$, …
- Split mini-batch T, of size m, into k "micro-batches", $\{T_0, \ldots, T_{k-1}\}$, each of size m/k
- The pipeline approach tries to avoid the idle time, by sending the next micro-batch when it can.
- Assume devices can hold: a model shard of size $Ld^2/4$ and the activation checkpoints, of total size $Ldm/4$

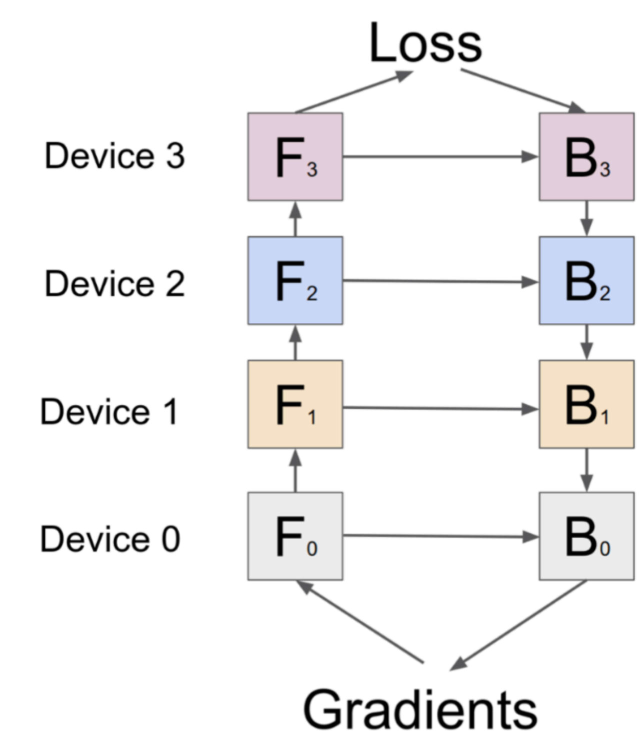

Loss
Device 3
Device 2
Device 1
Device 0
F3
F2
F1
F0
B3
B2
B1
B0
Gradients

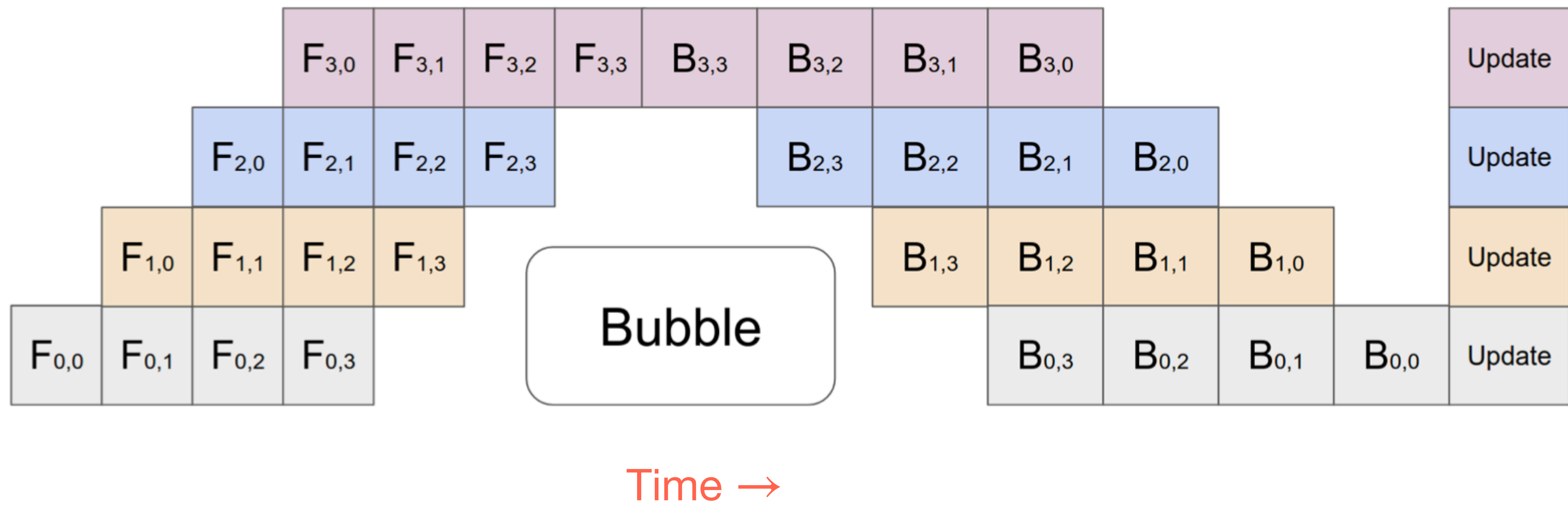

Device 3
Device 2
Device 1
Device 0
F3,0 F3,1 F3,2 F3,3 B3,3 B3,2 B3,1 B3,0 Update
F2,0 F2,1 F2,2 F2,3 B2,3 B2,2 B2,1 B2,0 Update
F1,0 F1,1 F1,2 F1,3 B1,3 B1,2 B1,1 B1,0 Update
F0,0 F0,1 F0,2 F0,3 B0,3 B0,2 B0,1 B0,0 Update
Bubble
Time →





- Notation: $F_{i,j}$ denotes the forward pass computed by device i on micro-batch T_j (analogous def for$B_{i,j}$) recall that device i contains a model shard, containing some of the layers.

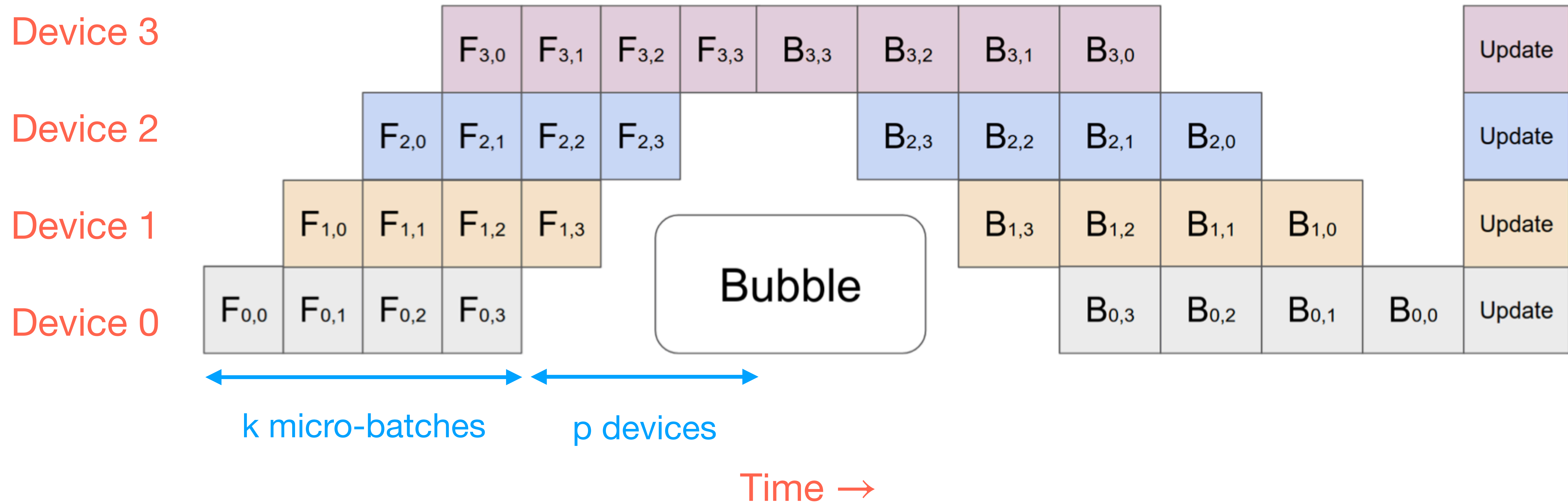




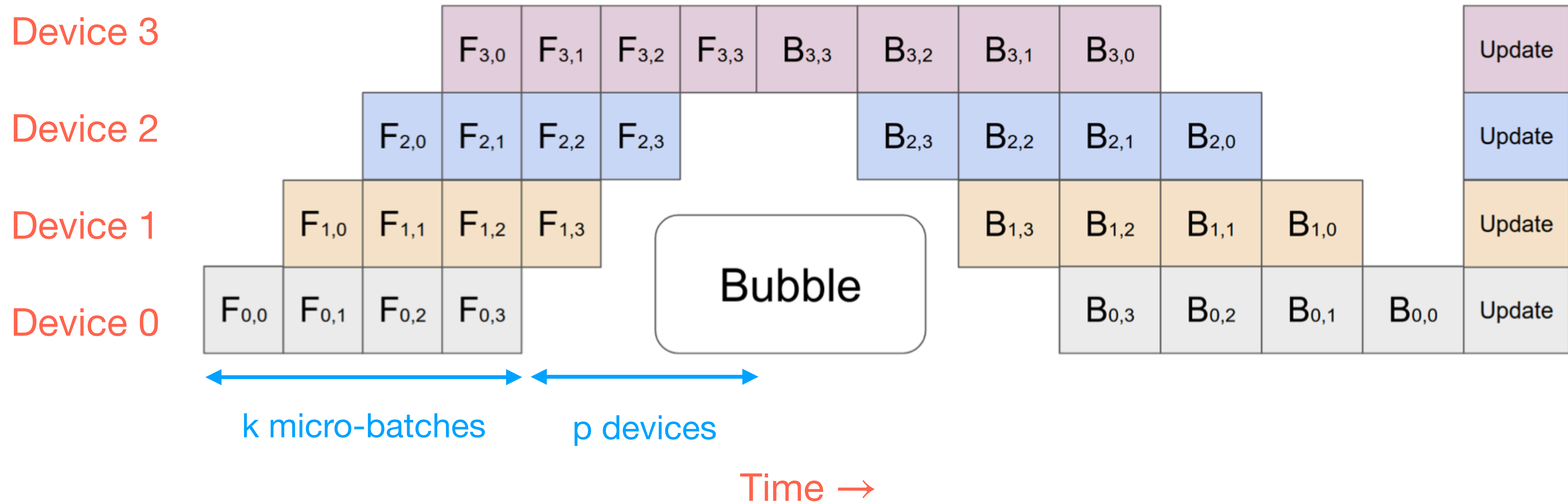
- Notation: $F_{i,j}$ denotes the forward pass computed by device i on micro-batch T_j (analogous def for$B_{i,j}$) recall that device i contains a model shard, containing some of the layers.
- **Forward:**
  - Device i computes the forward pass on data T_j, after receiving the input activations from device $i-1$.
  - All activations are checkpointed on their corresponding device.
- After all the forward passes are done, we can begin the backward pass. One approach is to start with $B_{3,3}$

- Notation: $F_{i,j}$ denotes the forward pass computed by device i on micro-batch T_j (analogous def for$B_{i,j}$) recall that device i contains a model shard, containing some of the layers.
- **Forward:**
  - Device i computes the forward pass on data T_j, after receiving the input activations from device $i - 1$.
  - All activations are checkpointed on their corresponding device.
- After all the forward passes are done, we can begin the backward pass. One approach is to start with $B_{3,3}$
- **Backward:**
  - Device i computes the backward pass on data T_j, after receiving the activation grads from device $i + 1$.
  - Each gradient can be all-reduced.

Idle Time & MFU

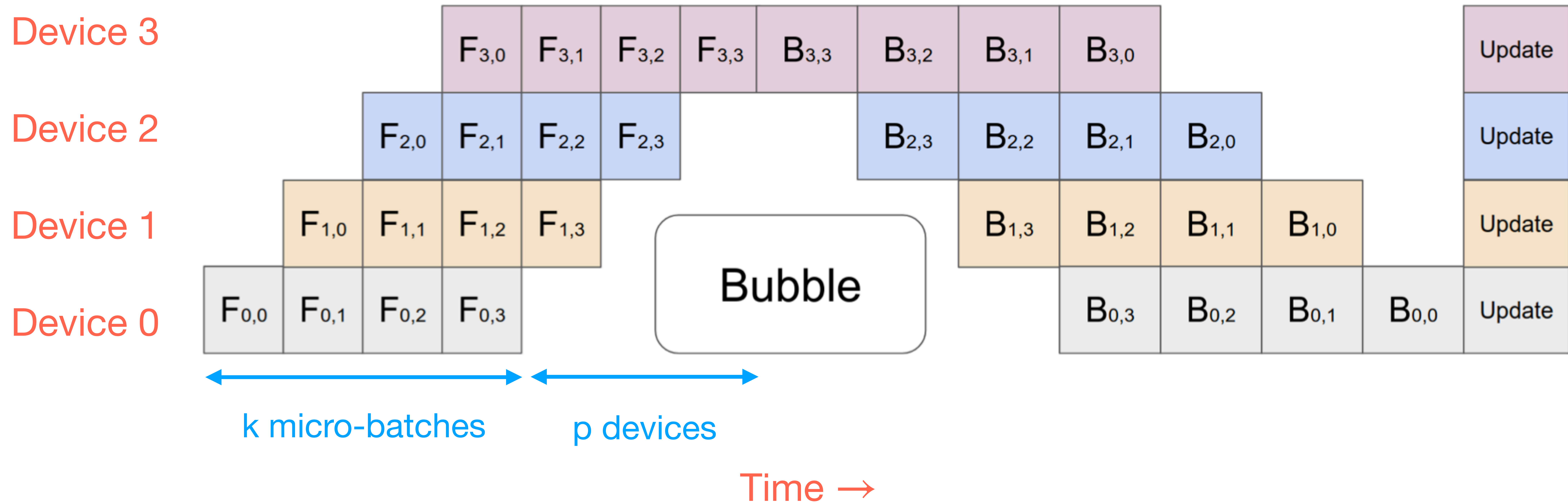


Idle Time & MFU



What fraction of time is IDLE?

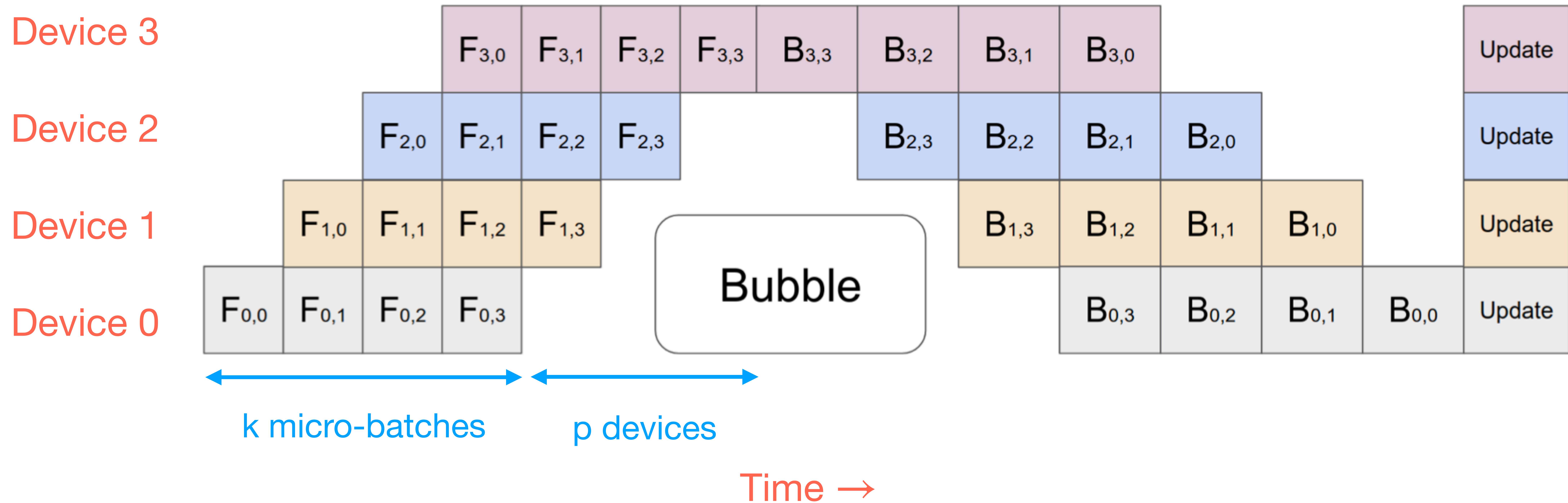
Idle Time & MFU



What fraction of time is IDLE?

$$\frac{p}{p+k}$$

Idle Time & MFU



What fraction of time is IDLE?

$$\frac{p}{p+k}$$

So if $k \approx 3p$, then … → 1/4 idle time

Scaling Considerations: how does it scale?

Scaling Considerations: how does it scale?

- Suppose we want a big model. How can grow the model?

Scaling Considerations: how does it scale?

- Suppose we want a big model. How can grow the model?
  - Suppose V layers can fit on a device and their activations, so we have $Vd^2 + Vdm$ free memory.
  - With p devices, we can get a model with $L = pV$ and width d.

pVd^2

$V^2dm + 2dm$

Scaling Considerations: how does it scale?

- Suppose we want a big model. How can grow the model?
  - Suppose V layers can fit on a device and their activations, so we have $Vd^2 + Vdm$ free memory.
  - With p devices, we can get a model with $L = pV$ and width d.
- We need to **ensure communication and compute overlap** between the nodes/devices
  - Activations get passed between devices: need to communicate dm (per step in the pipe)
  - Need to all-reduce model params: need to communicate Vd^2 (per backard step in the pipe)

Scaling Considerations: how does it scale?

- Suppose we want a big model. How can grow the model?
  - Suppose V layers can fit on a device and their activations, so we have $Vd^2 + Vdm$ free memory.
  - With p devices, we can get a model with $L = pV$ and width d.
- We need to **ensure communication and compute overlap** between the nodes/devices
  - Activations get passed between devices: need to communicate dm (per step in the pipe)
  - Need to all-reduce model params: need to communicate Vd^2 (per backard step in the pipe)
- **Take aways:** (assume Chinchilla where $\text{tokens} \approx 20 \cdot \text{model size}$)
  - Depth scaling: **serial processing time of pipeline grows linearly** with model size, fixing m.
  - This means that serial run time grows quadratically (serial runtime is O(tokens*time_per_iter))
  - We pay a serial factor of 3 (the amortization factor) over to pure DDP (i.e. using batch size one).

Scaling Considerations: how does it scale?

- Suppose we want a big model. How can grow the model?
  - Suppose V layers can fit on a device and their activations, so we have $Vd^2 + Vdm$ free memory.
  - With p devices, we can get a model with $L = pV$ and width d.
- We need to **ensure communication and compute overlap** between the nodes/devices
  - Activations get passed between devices: need to communicate dm (per step in the pipe)
  - Need to all-reduce model params: need to communicate Vd^2 (per backard step in the pipe)
- **Take aways:** (assume Chinchilla where $\text{tokens} \approx 20 \cdot \text{model size}$)
  - Depth scaling: **serial processing time of pipeline grows linearly** with model size, fixing m.
  - This means that serial run time grows quadratically (serial runtime is O(tokens*time_per_iter))
  - We pay a serial factor of 3 (the amortization factor) over to pure DDP (i.e. using batch size one).

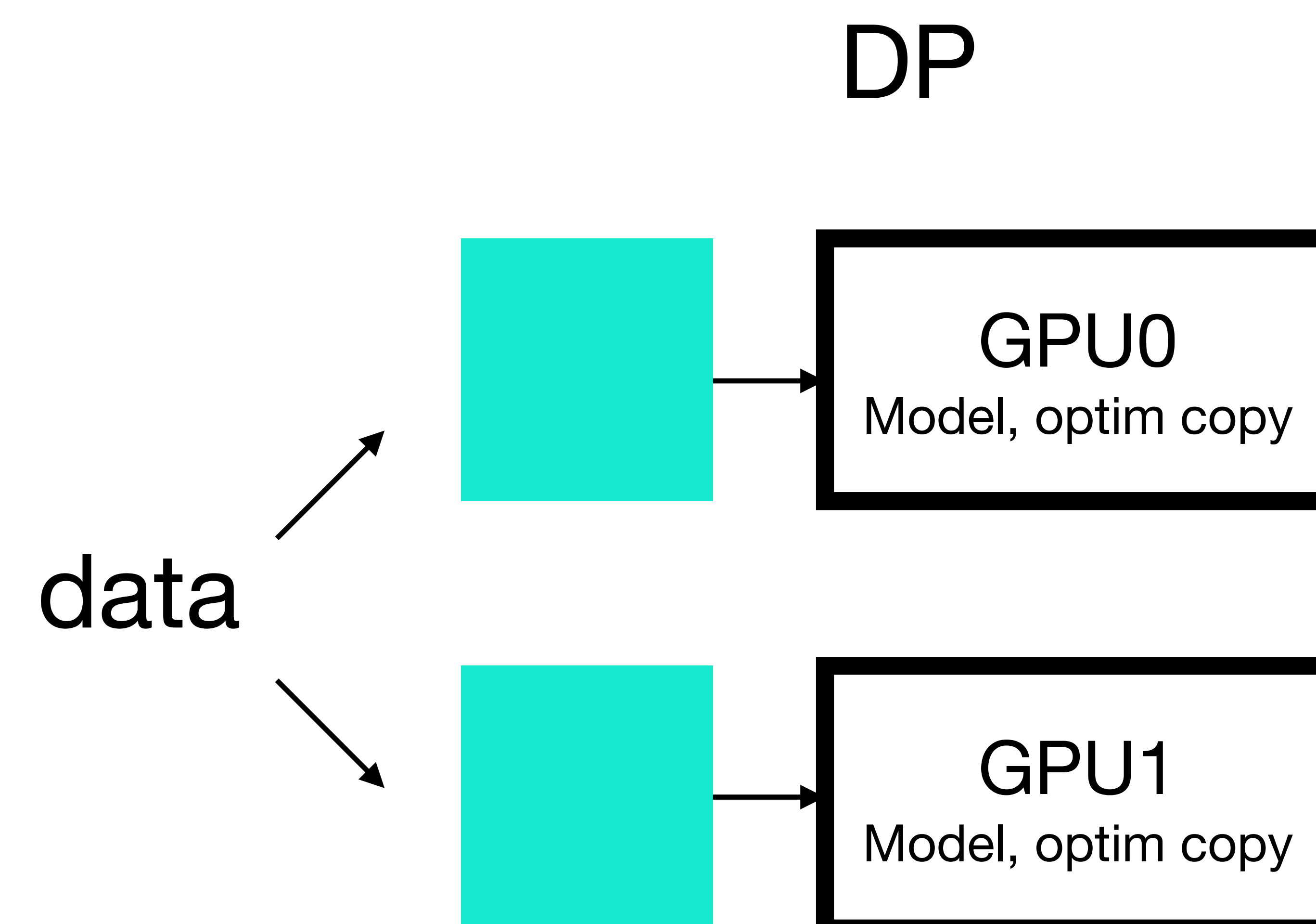
- Even if we make the per-device-batchsize-m smaller (+DDP over the pipelines), we still have a communication bottleneck due to the model params on the backward pass.

Today

- Recap++:
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

Fully Sharded Data Parallel (FSDP)

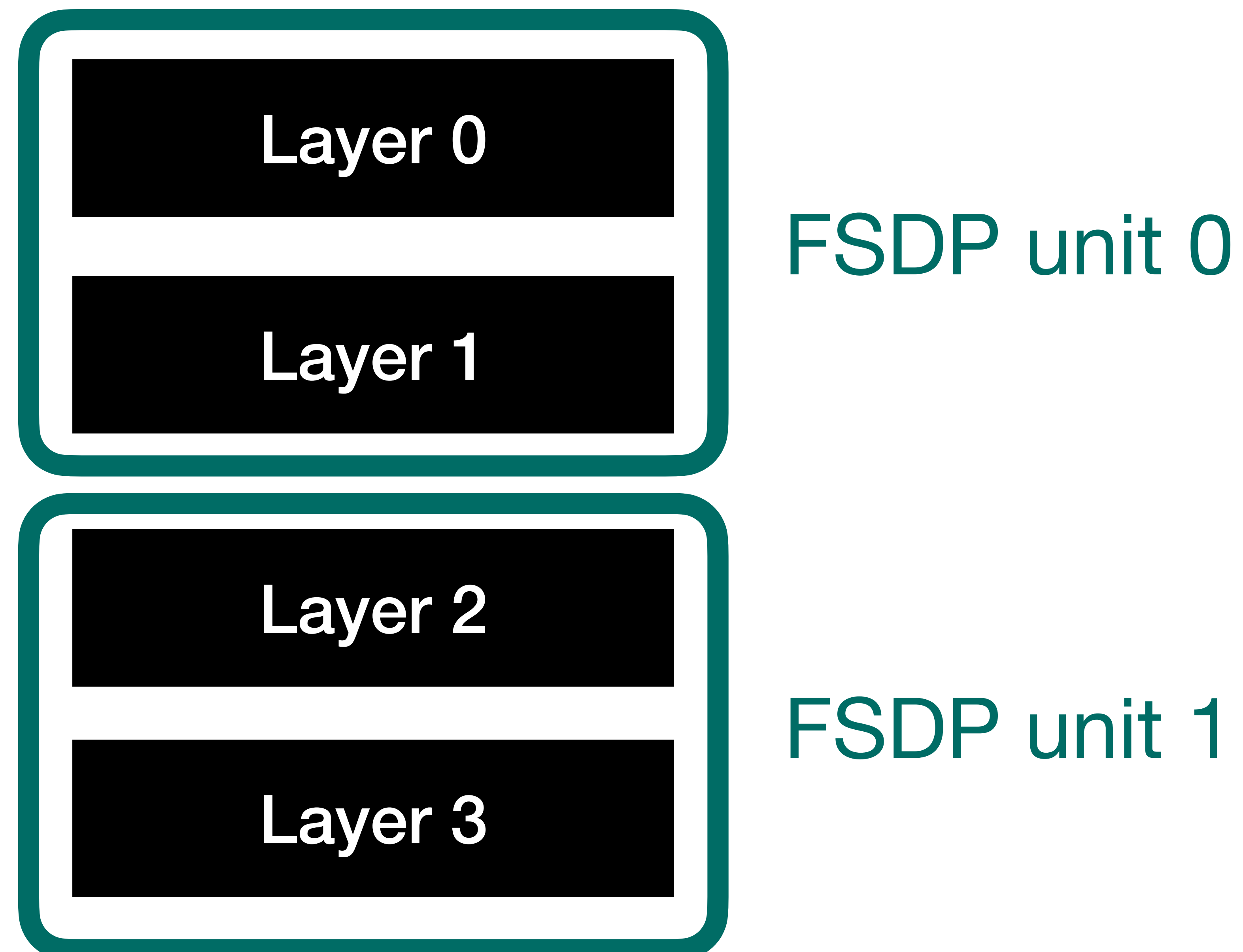
- Motivation: we want to do data distributed training *without* a pipeline
- **Fully** sharded = shard model, optimizer, and data across GPUs (can be seen as a kind of successor to DDP.)
- Inspired by “ZeRO” algorithm (particularly ZeRO-3), but some specifics are different



How is the model stored? Terminology: shards and units

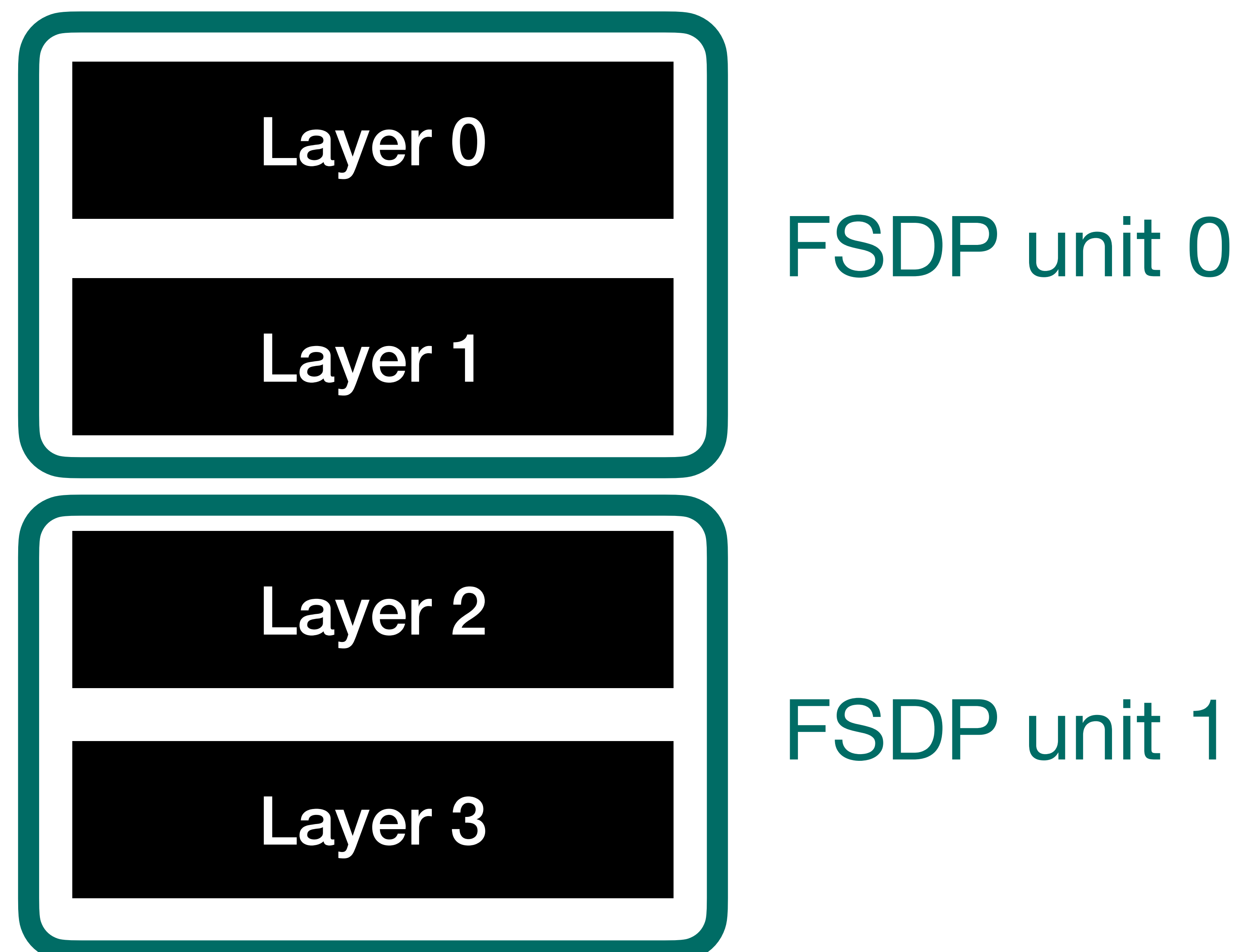
- Let's forget about optimizer states for a second, how is the model even stored?
- An **FSDP unit** is an abstraction that determines how the model will be split. We have flexibility in how to define it!
- For example, an FSDP unit could be a single layer (or a series of layers). Practically, it is a `torch.nn.Module` or collection of layers
- Units are broken down further into **shards,** which is how the units will be stored across the GPUs, i.e. each FSDP unit is **sharded** across GPUs.

How is the model stored? Terminology: shards and units



How is the model stored? Terminology: shards and units

- **Note**: “unit” is some piece of the model that we want to load. (e.g. loading unit 0 onto gpu 3, means layers 0/1 are on gpu 3)

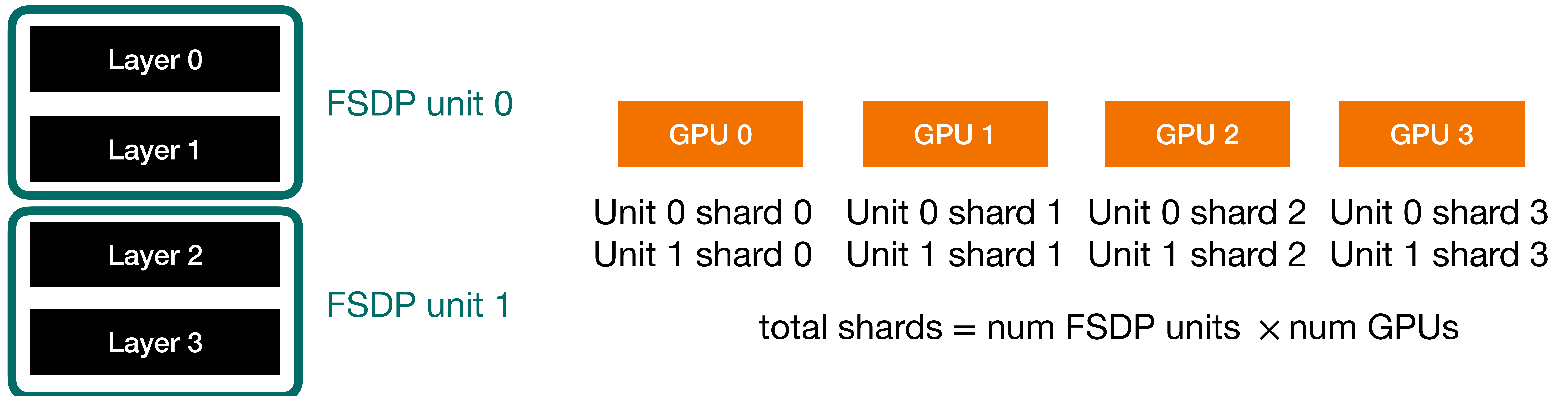


How is the model stored? Terminology: shards and units

- **Note**: “unit” is some piece of the model that we want to load. (e.g. loading unit 0 onto gpu 3, means layers 0/1 are on gpu 3)

- A fundamental operation will be to load a unit onto multiple GPUs

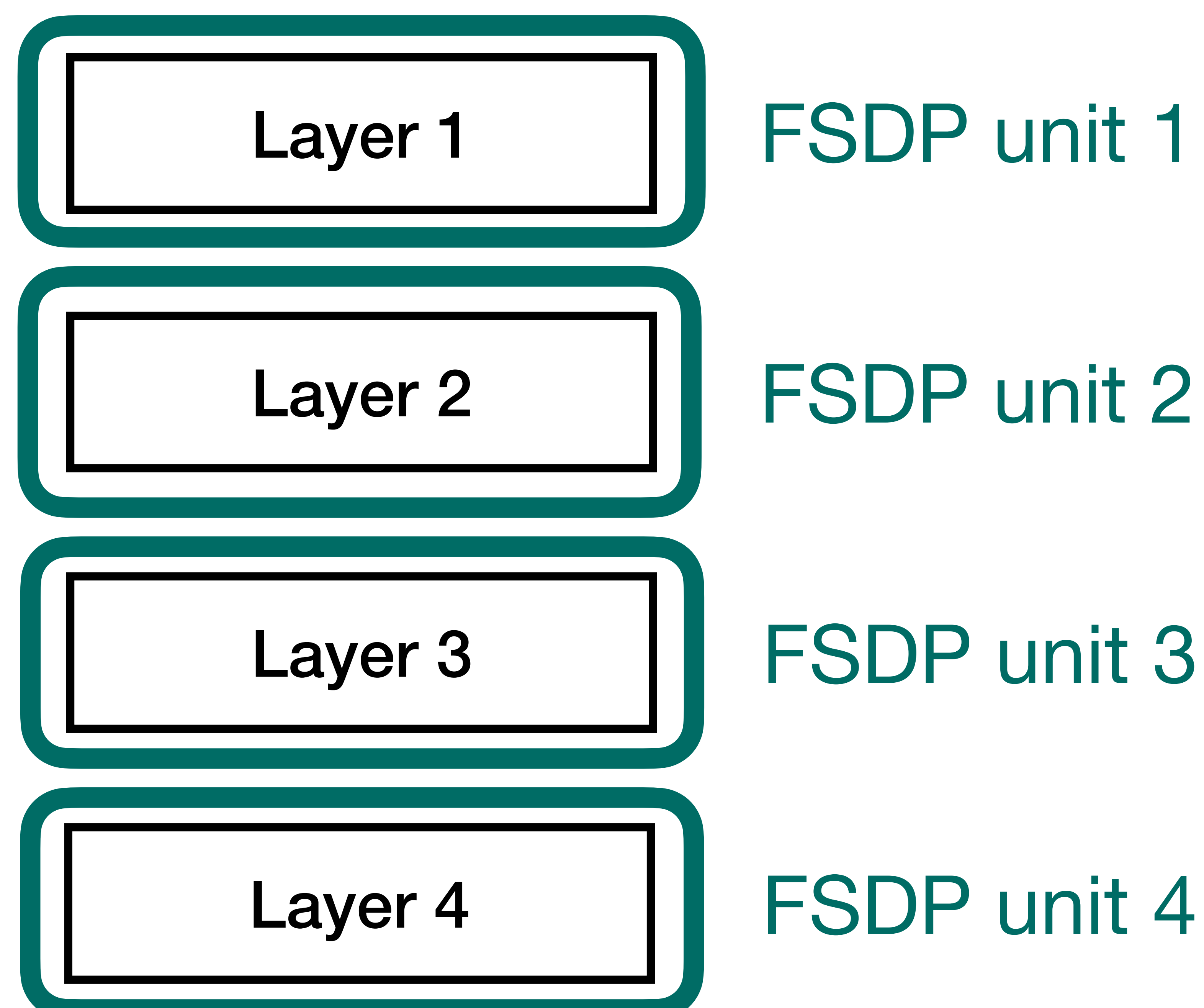
  - Conceptually, **we will only load units** so we can ignore the shards.



How does the computation work?

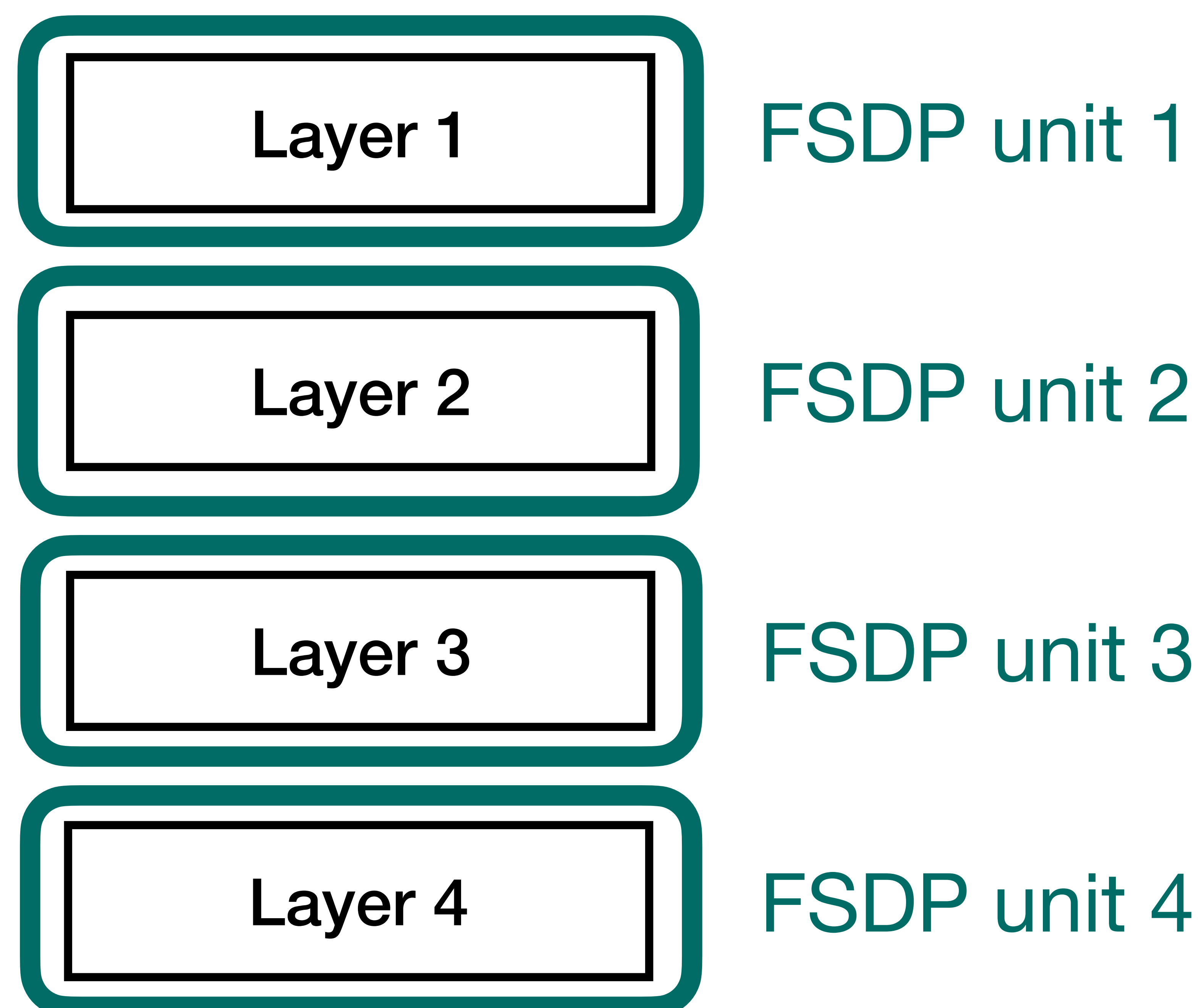
- FSDP tries to simulate DDP, but since each GPU only has access to a shard of the units, we have to **all-gather** a unit that we need for current part of the forward/backward pass

- Let's look at an example:

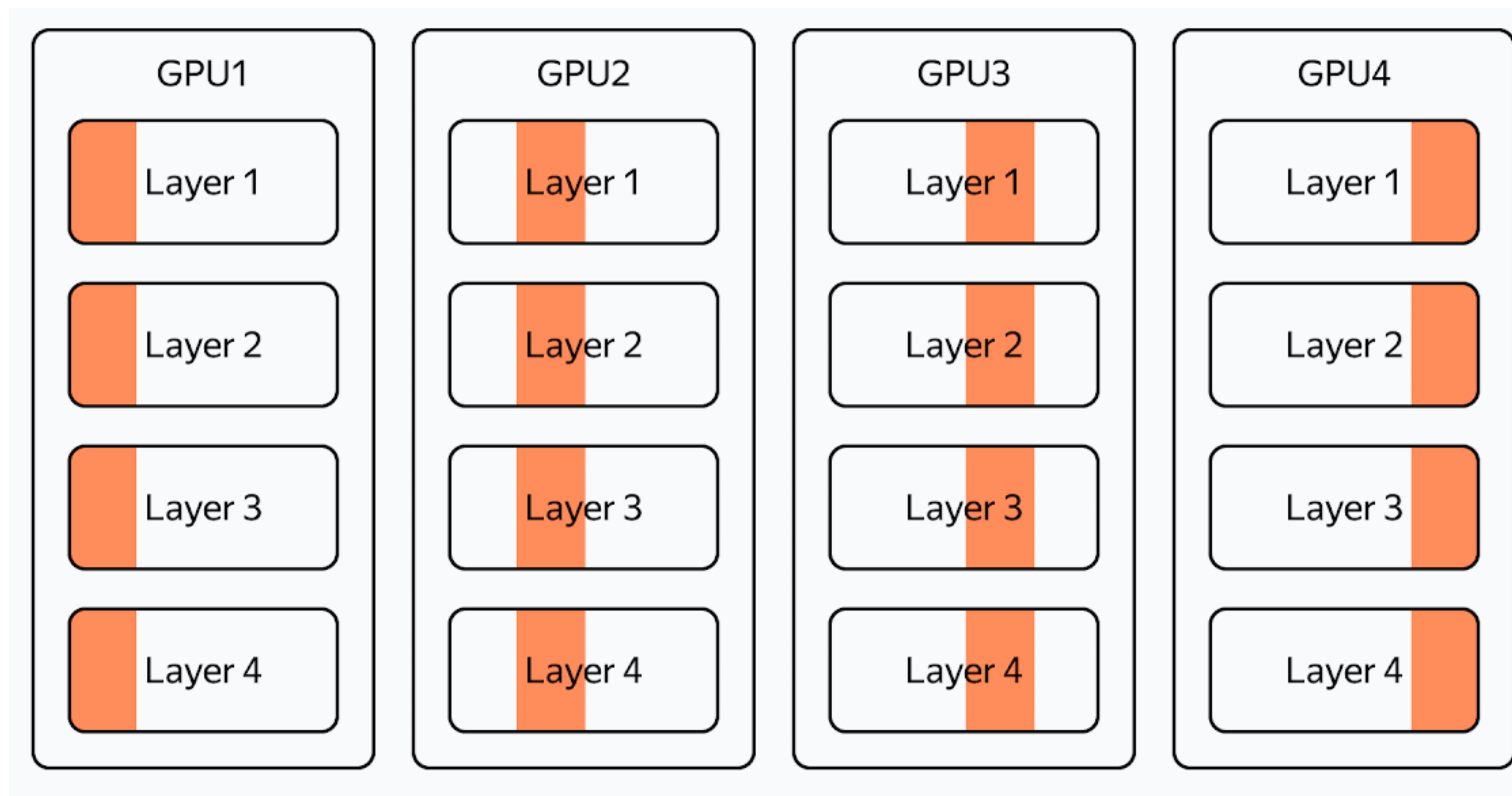


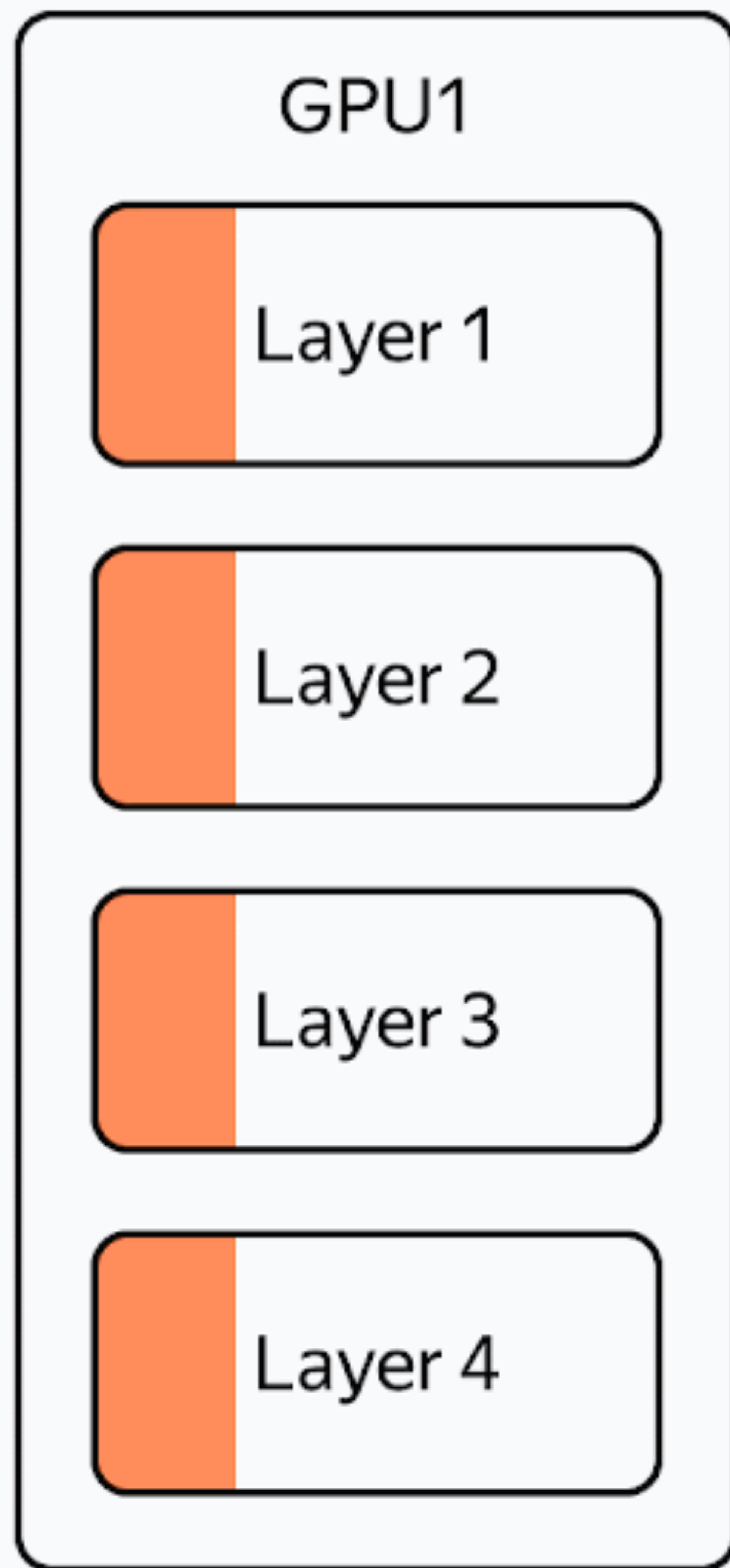
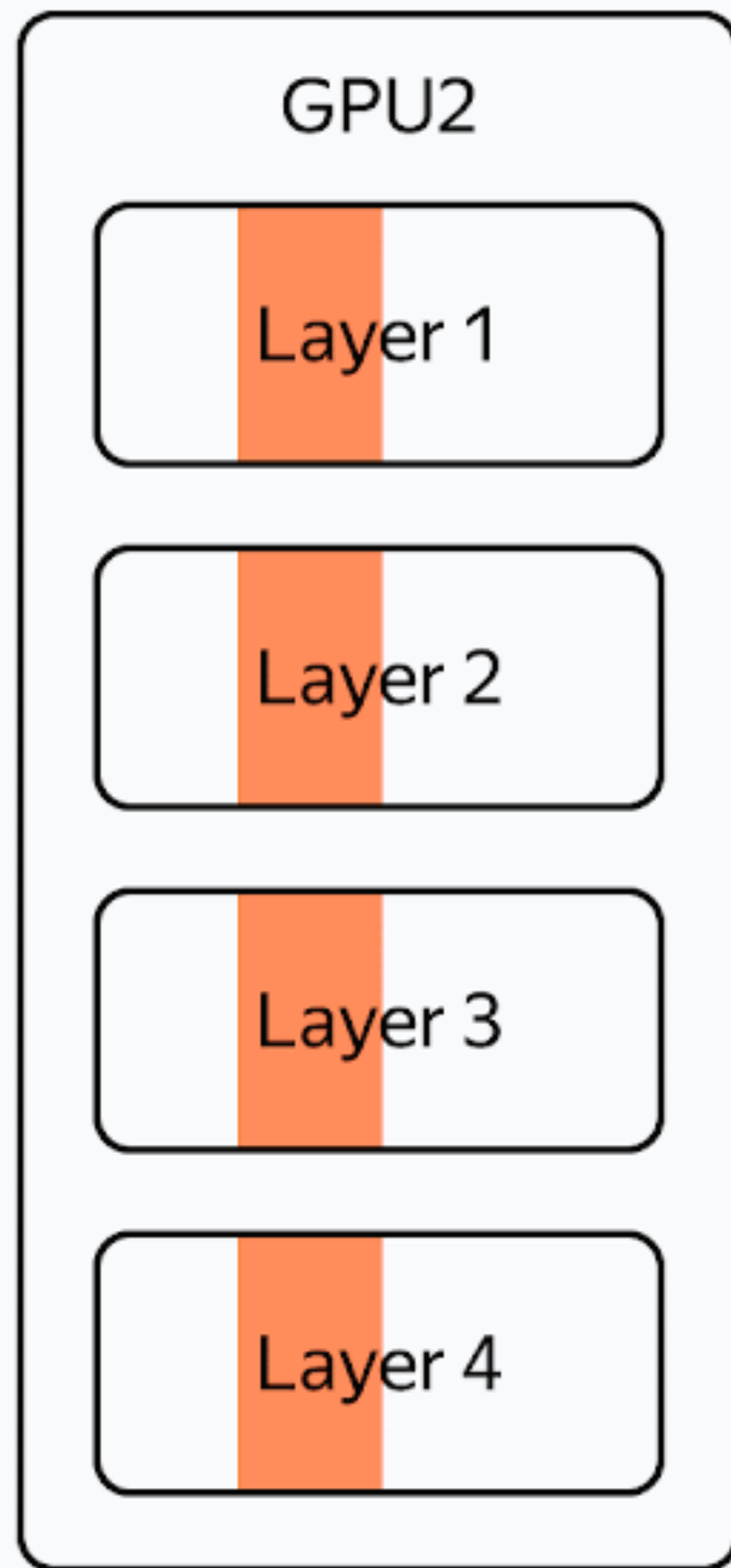
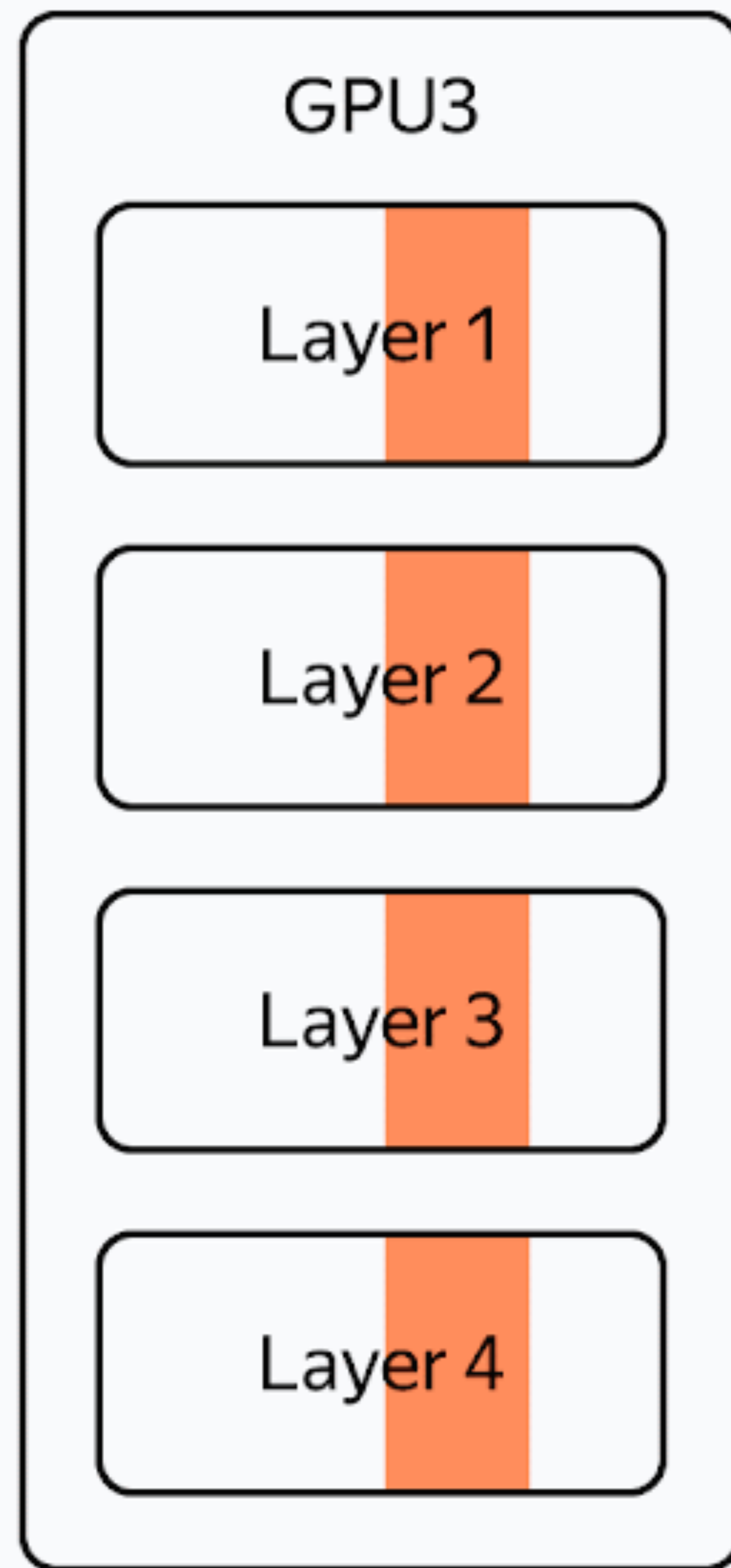
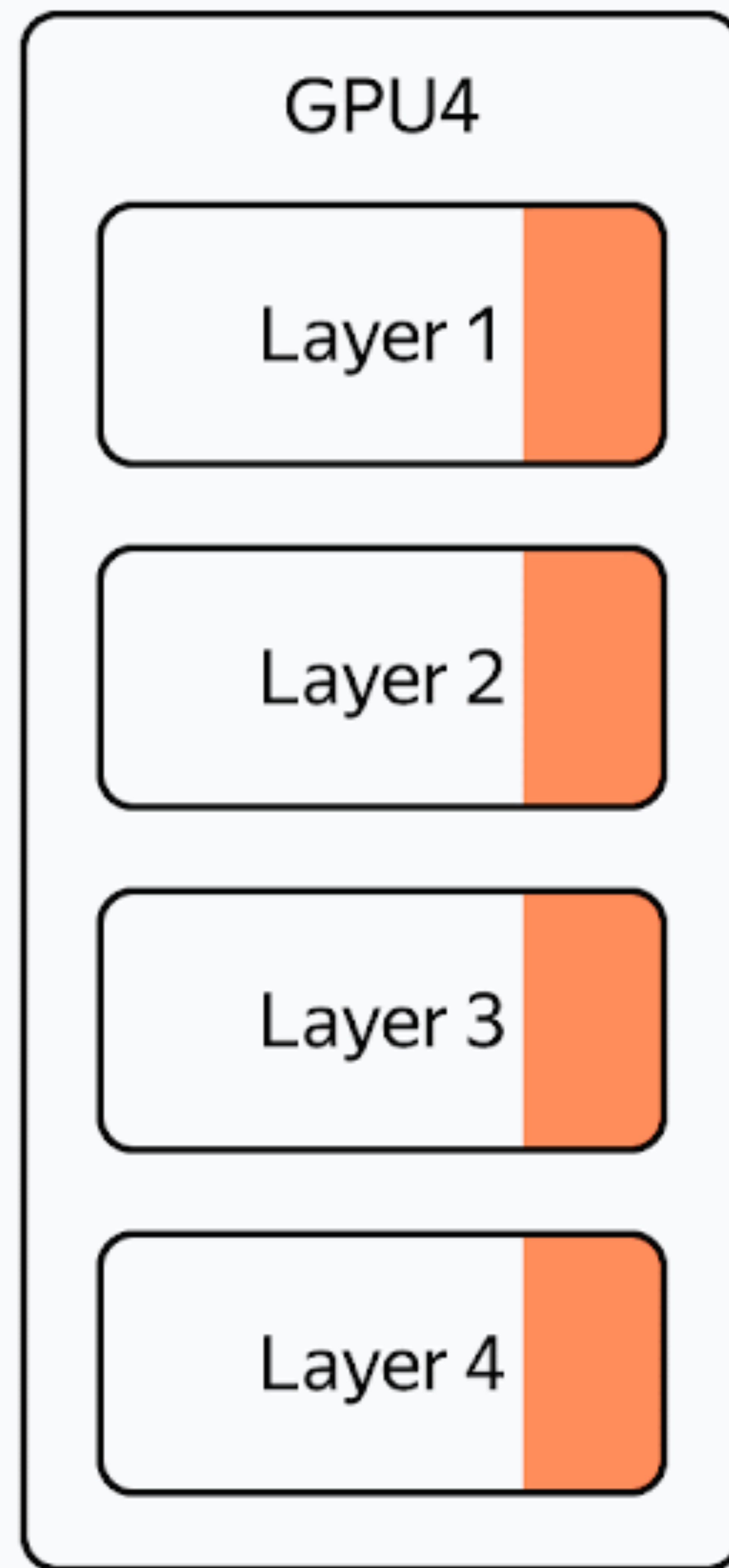
How does the computation work?

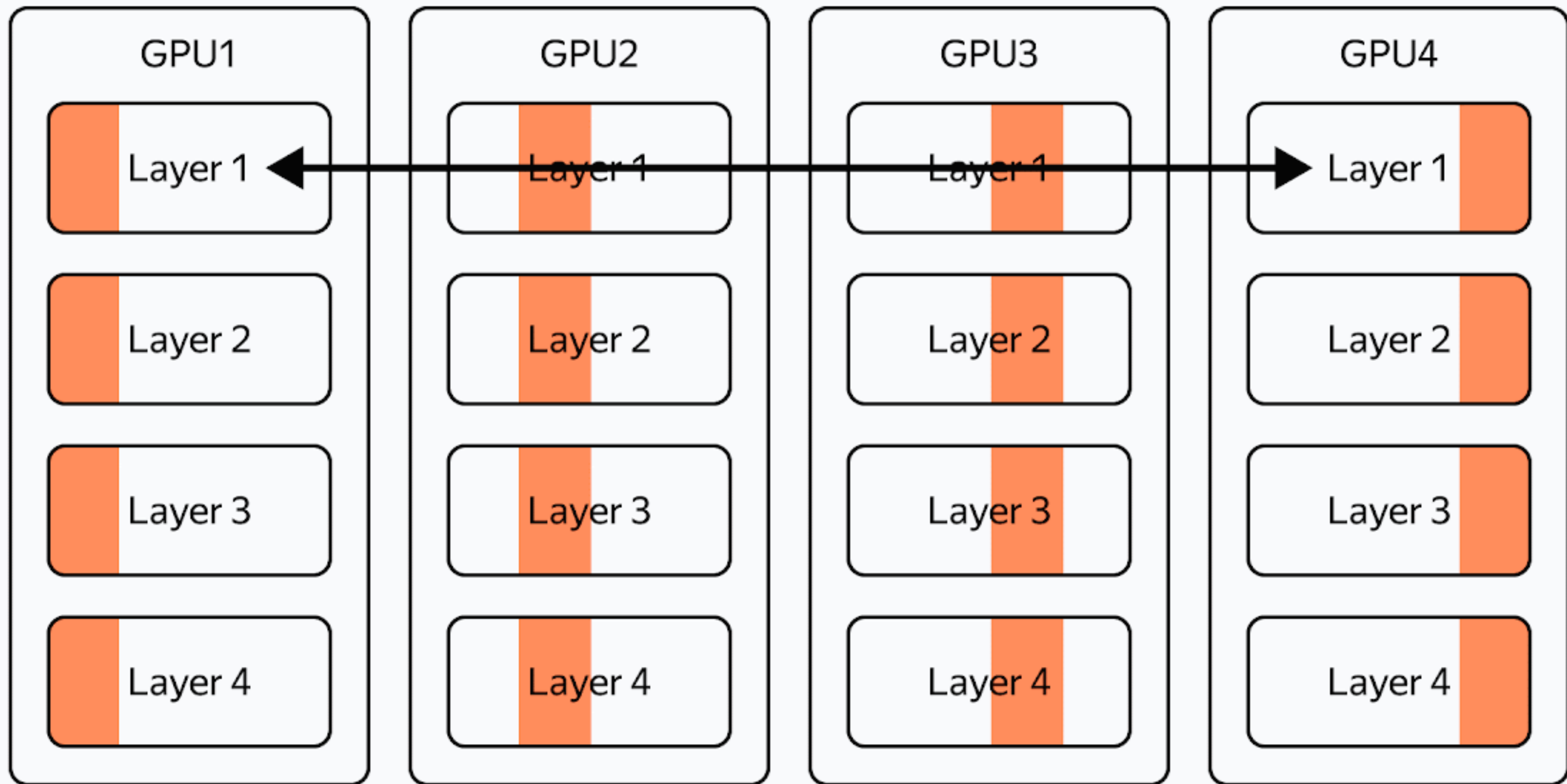
- FSDP tries to simulate DDP, but since each GPU only has access to a shard of the units, we have to **all-gather** a unit that we need for current part of the forward/backward pass
- Let's look at an example:



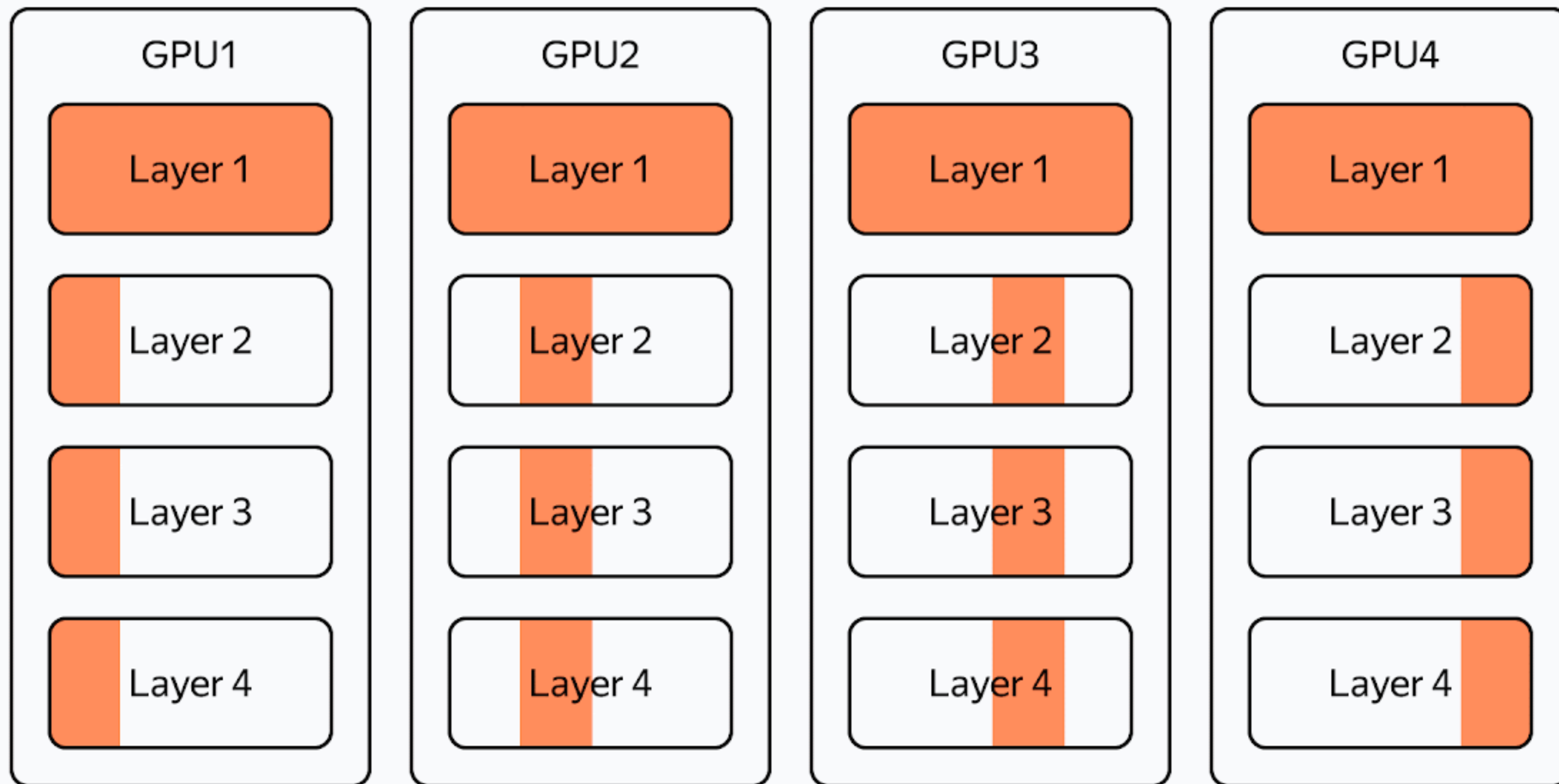
FSDP unit 1
FSDP unit 2
FSDP unit 3
FSDP unit 4

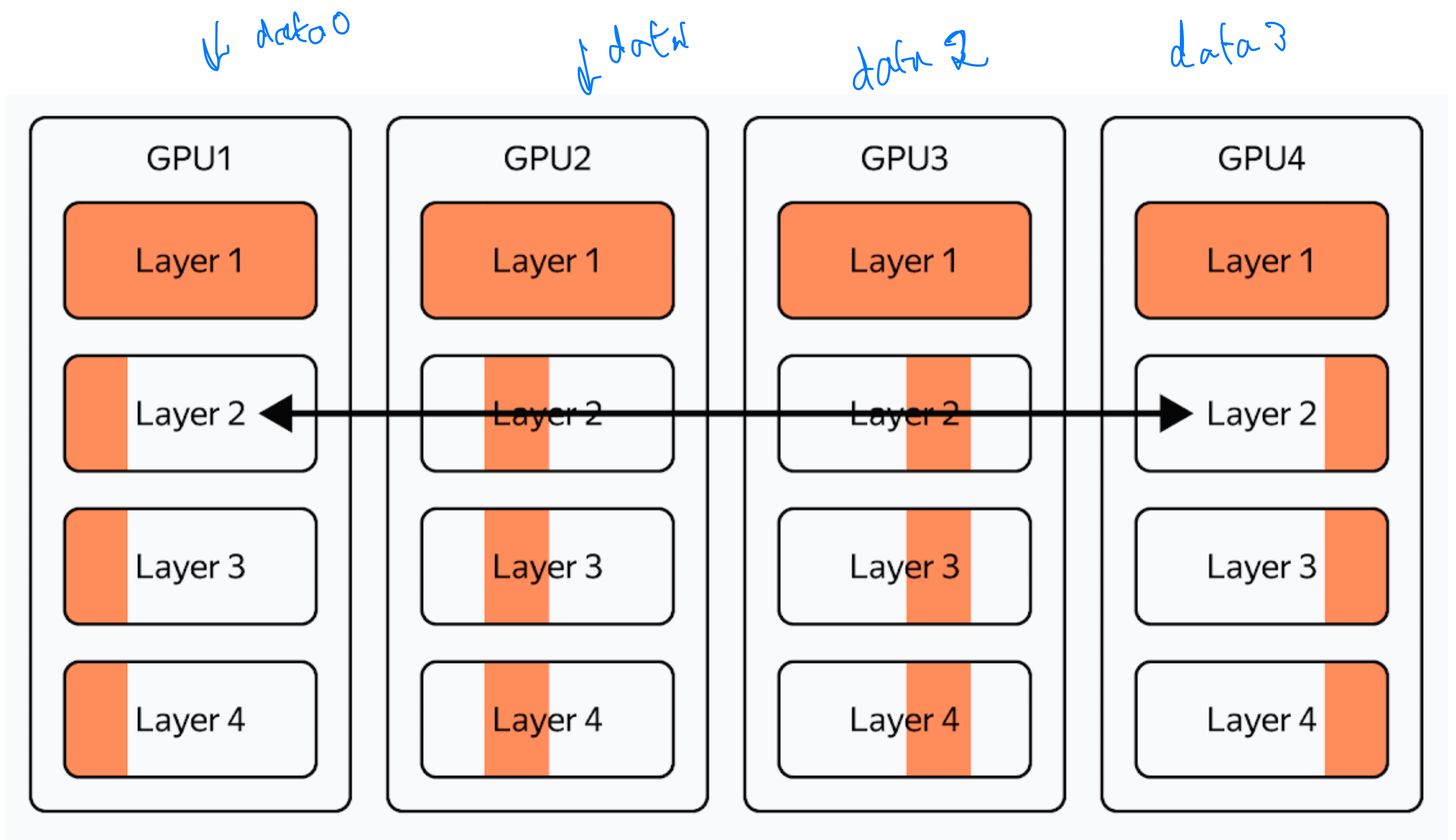



GPU1
Layer 1
Layer 2
Layer 3
Layer 4

GPU2
Layer 1
Layer 2
Layer 3
Layer 4

GPU3
Layer 1
Layer 2
Layer 3
Layer 4

GPU4
Layer 1
Layer 2
Layer 3
Layer 4

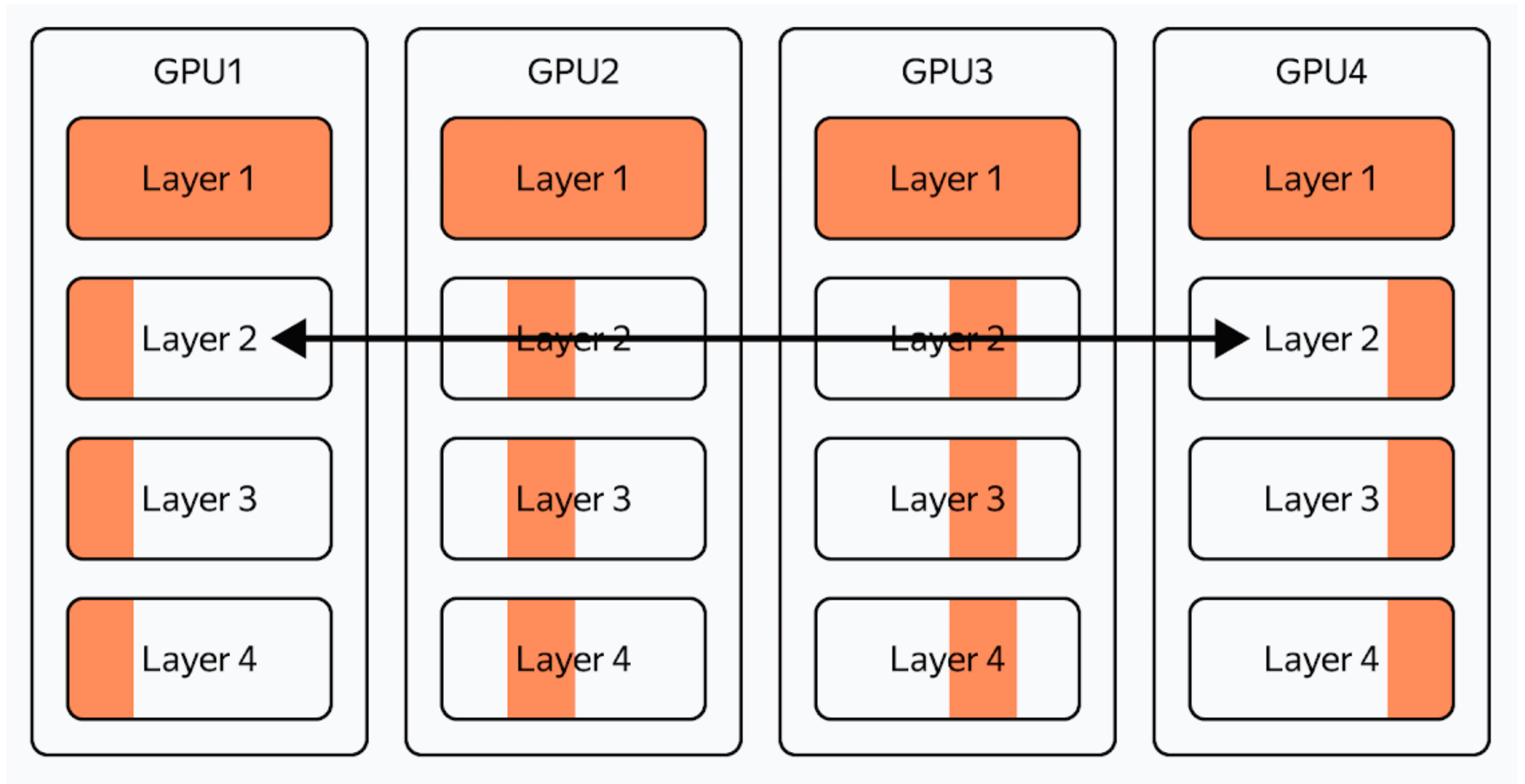


↔ **= All-Gather**

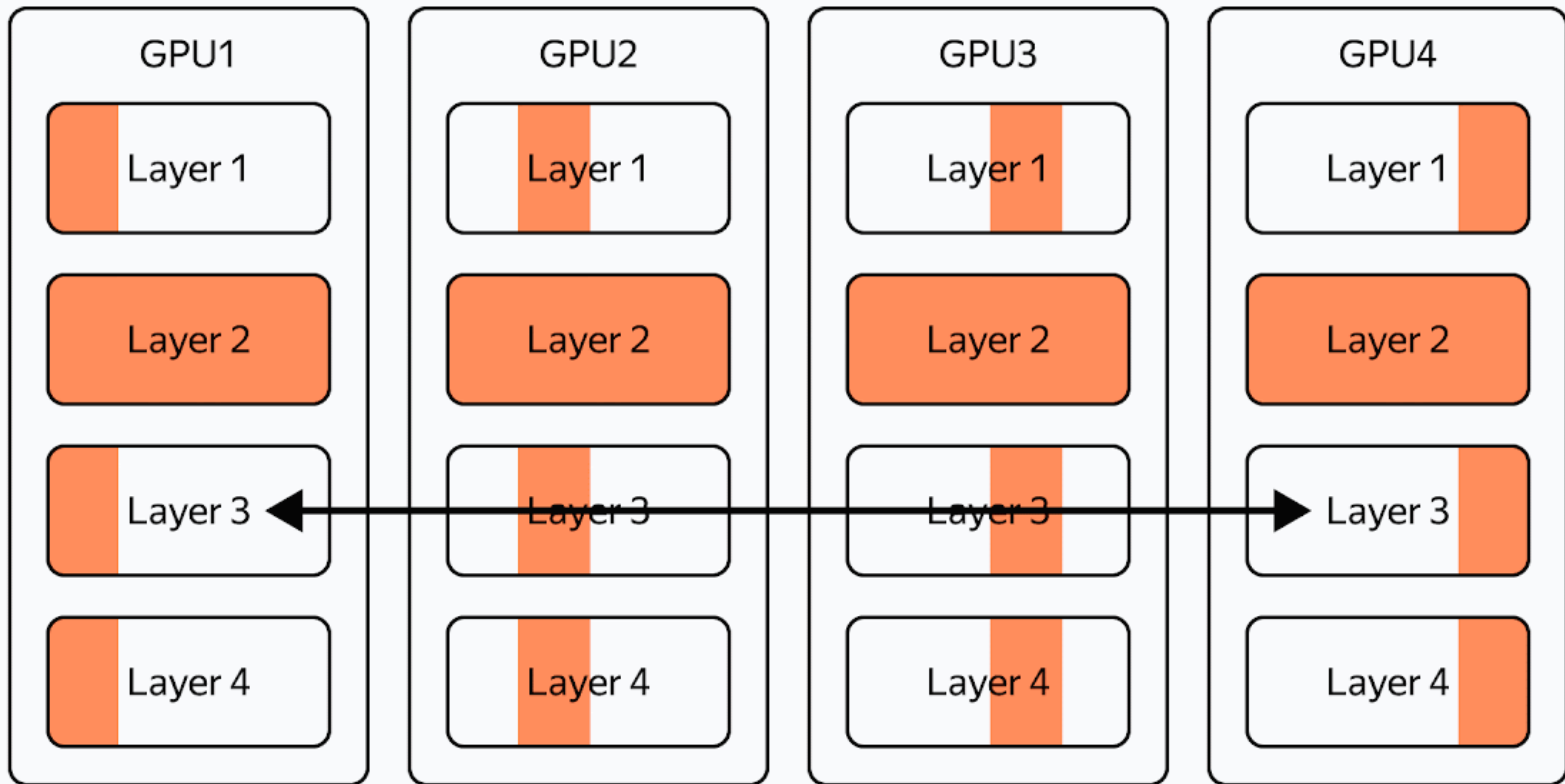

GPU1
Layer 1
Layer 2
Layer 3
Layer 4
GPU2
Layer 1
Layer 2
Layer 3
Layer 4
GPU3
Layer 1
Layer 2
Layer 3
Layer 4
GPU4
Layer 1
Layer 2
Layer 3
Layer 4

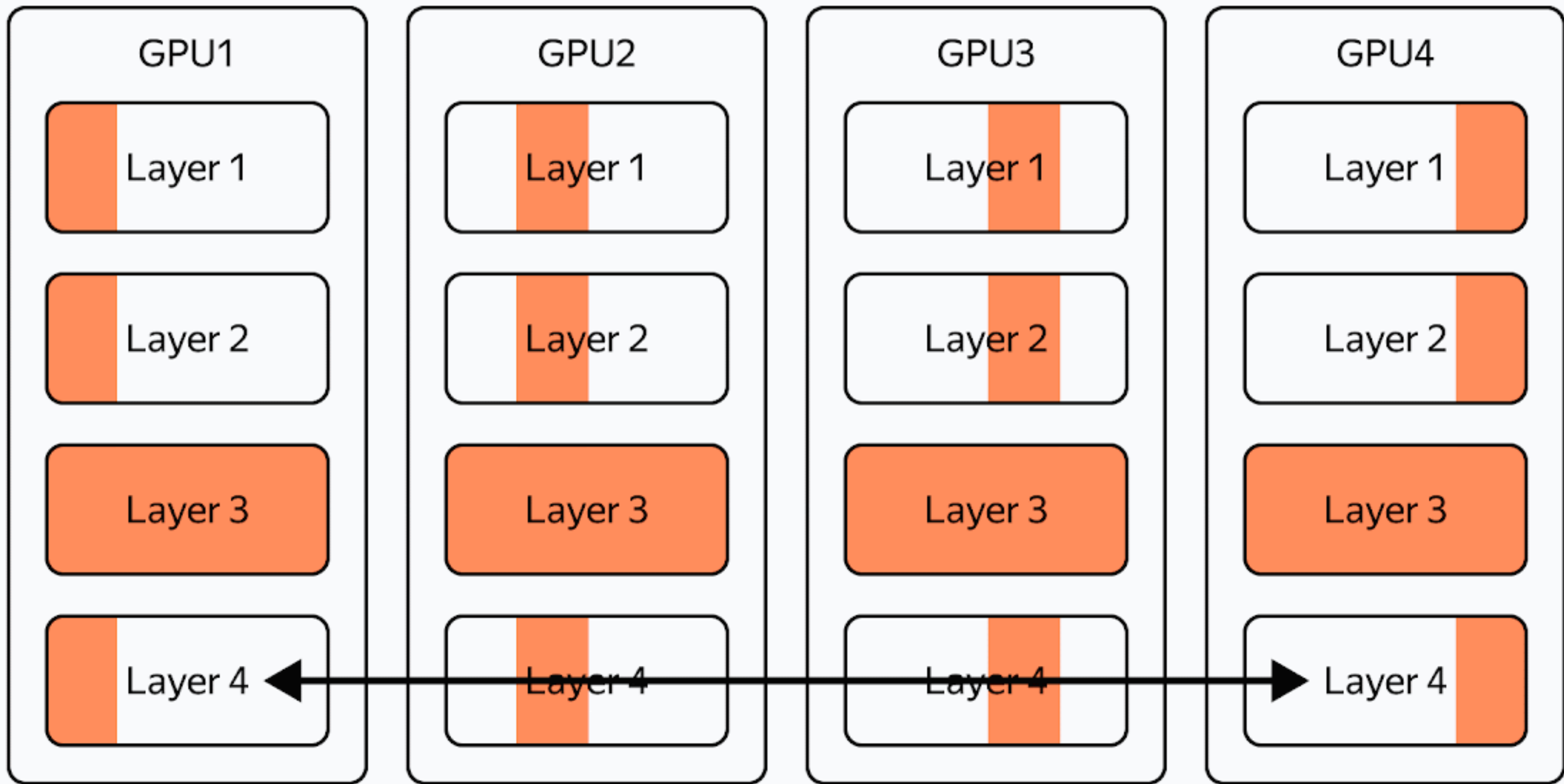


All-Gather later 2 while doing forward for Layer 1 **(communication/compute overlap)**

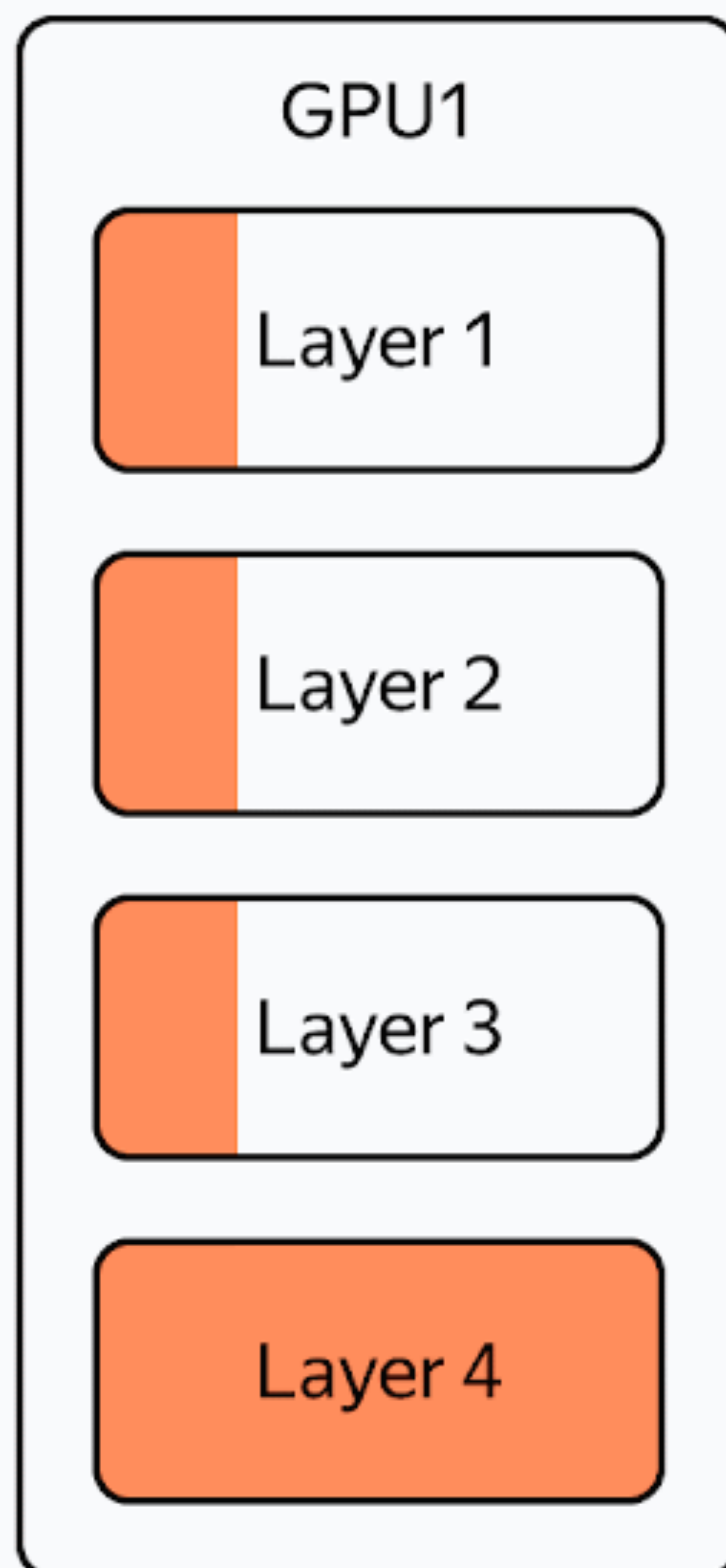
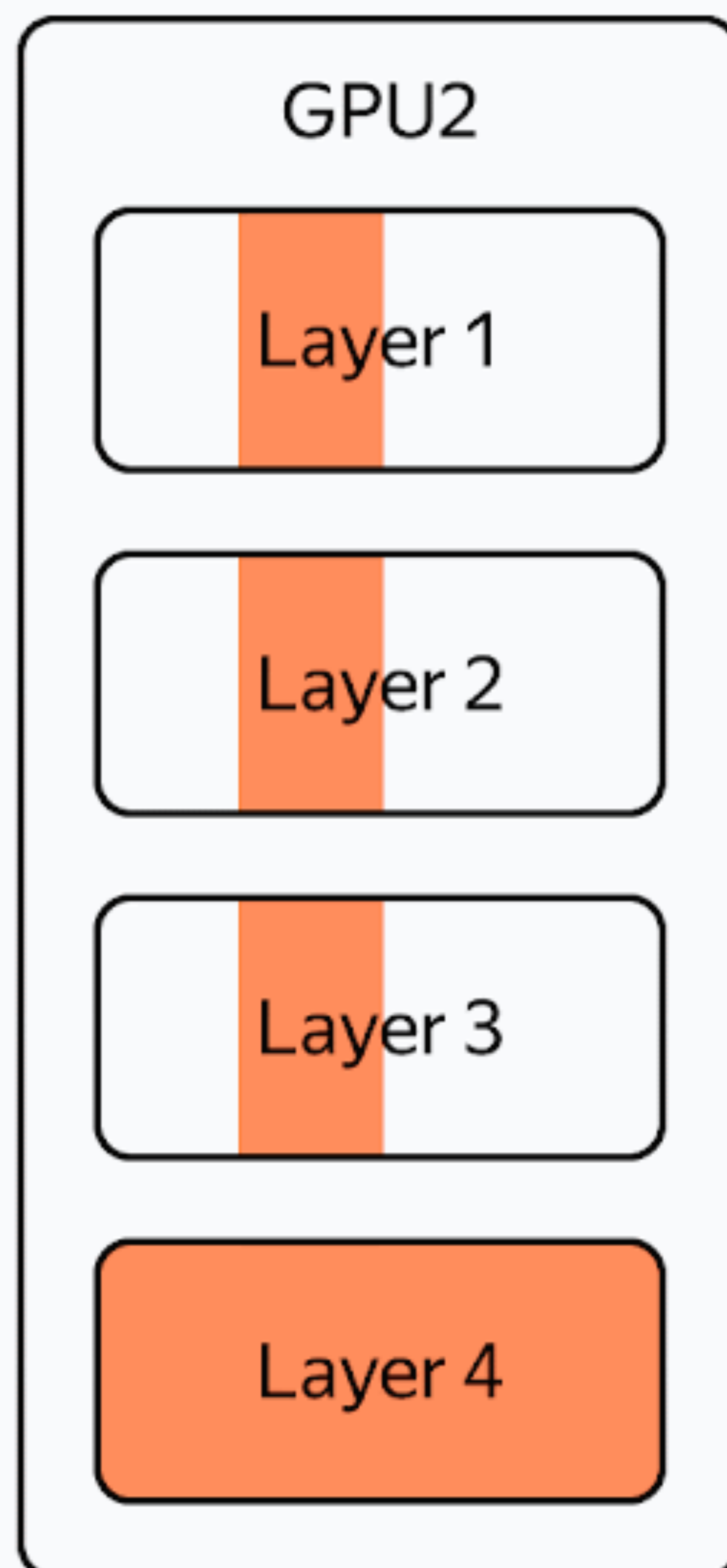
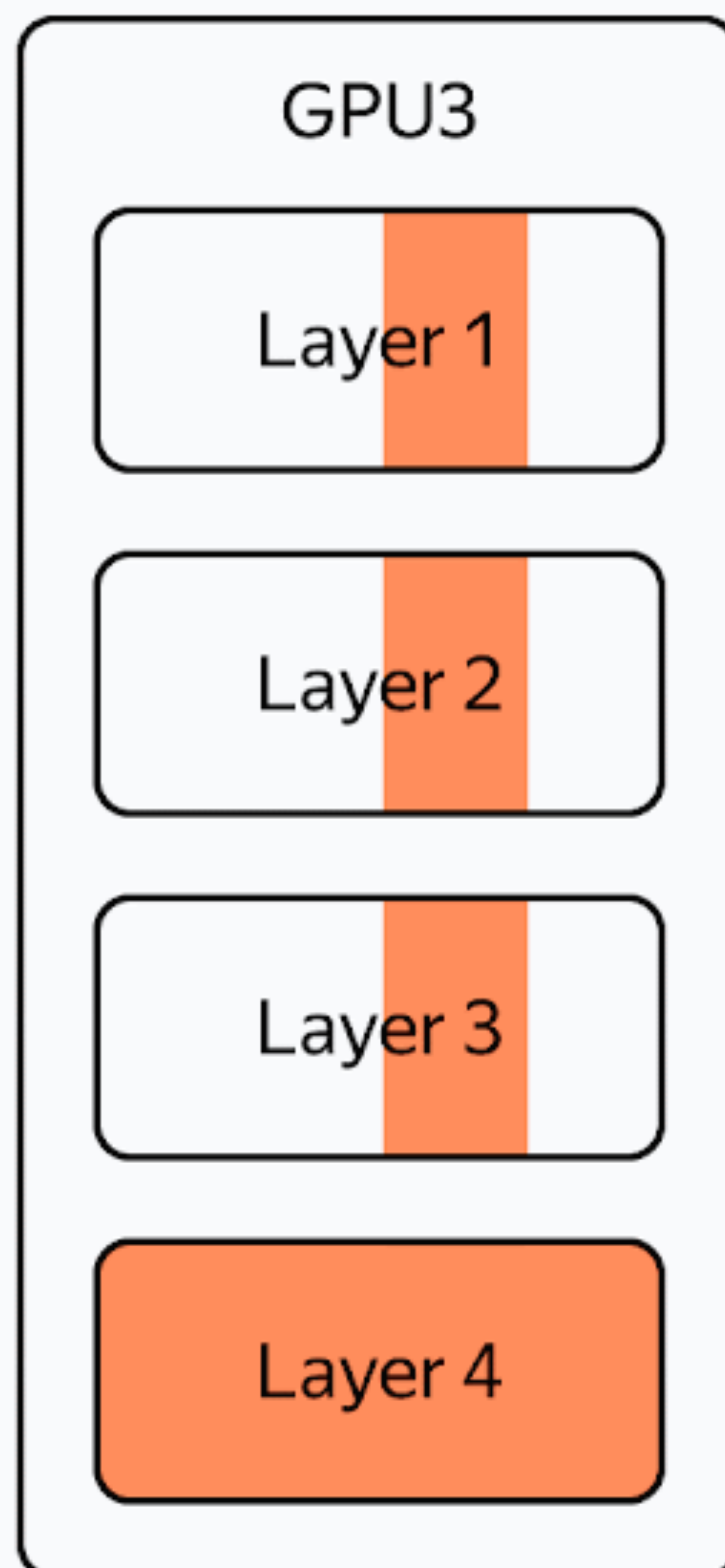
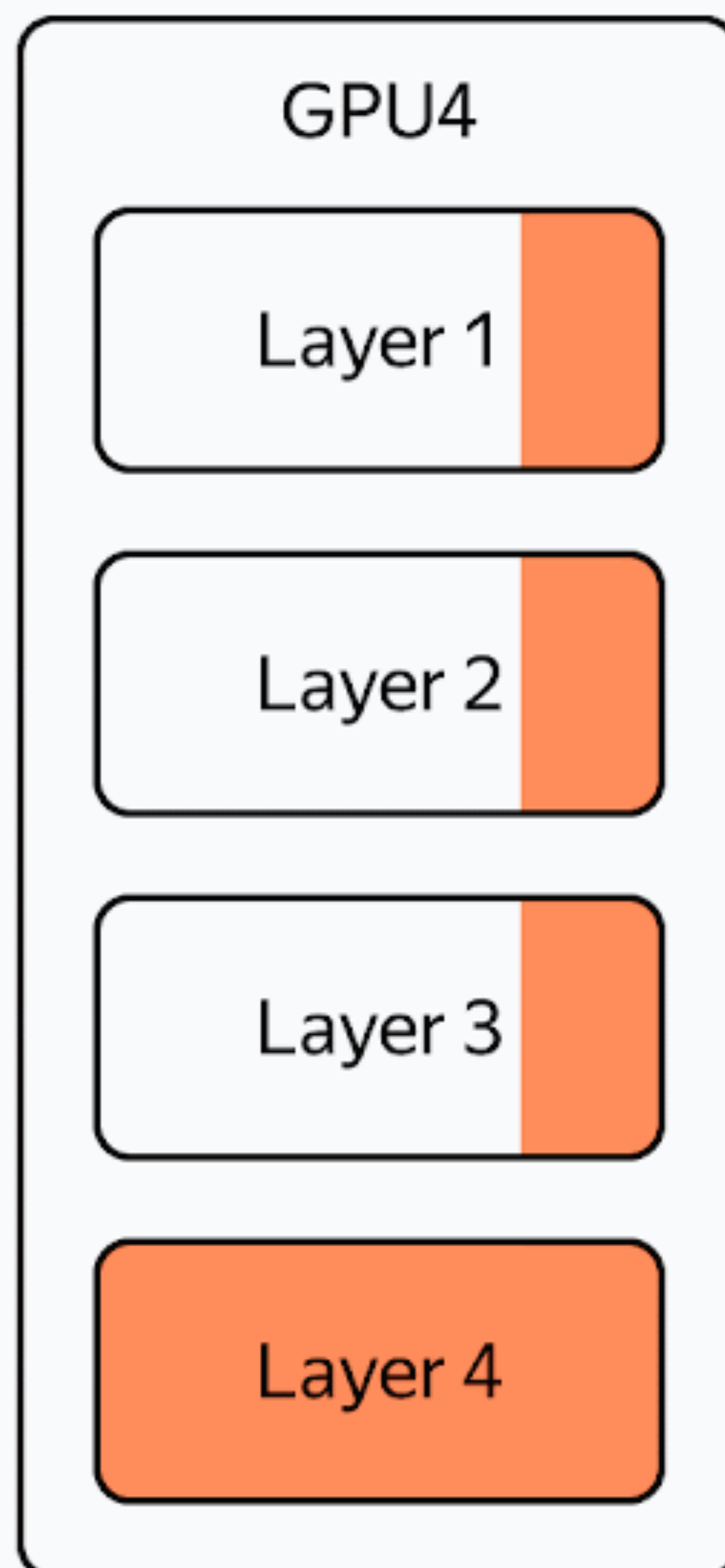


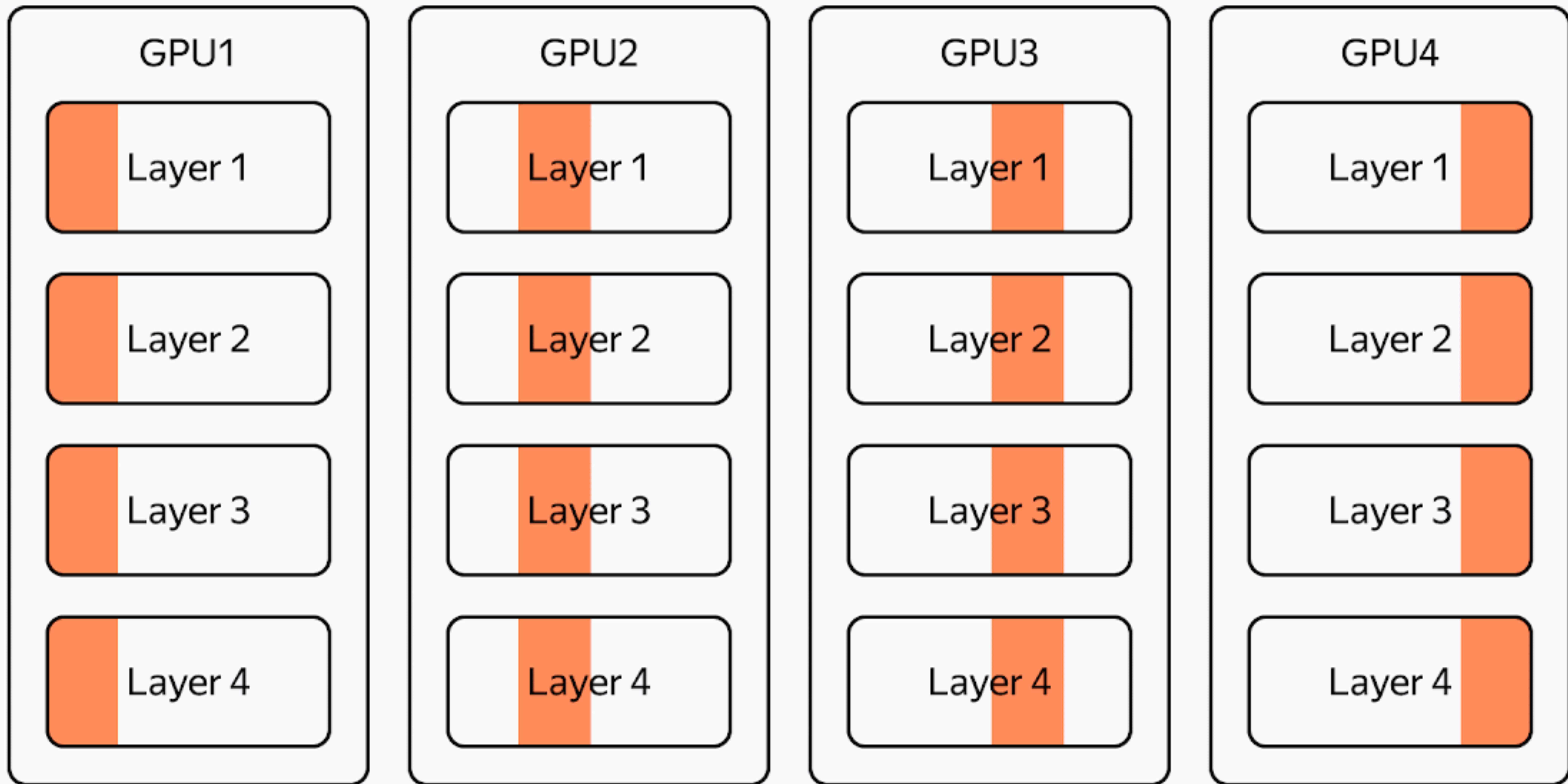
Clear layer 1 when done


GPU1
Layer 1
Layer 2
Layer 3
Layer 4
GPU2
Layer 1
Layer 2
Layer 3
Layer 4
GPU3
Layer 1
Layer 2
Layer 3
Layer 4
GPU4
Layer 1
Layer 2
Layer 3
Layer 4



GPU1
Layer 1
Layer 2
Layer 3
Layer 4
GPU2
Layer 1
Layer 2
Layer 3
Layer 4
GPU3
Layer 1
Layer 2
Layer 3
Layer 4
GPU4
Layer 1
Layer 2
Layer 3
Layer 4


GPU1
Layer 1
Layer 2
Layer 3
Layer 4

GPU2
Layer 1
Layer 2
Layer 3
Layer 4

GPU3
Layer 1
Layer 2
Layer 3
Layer 4

GPU4
Layer 1
Layer 2
Layer 3
Layer 4



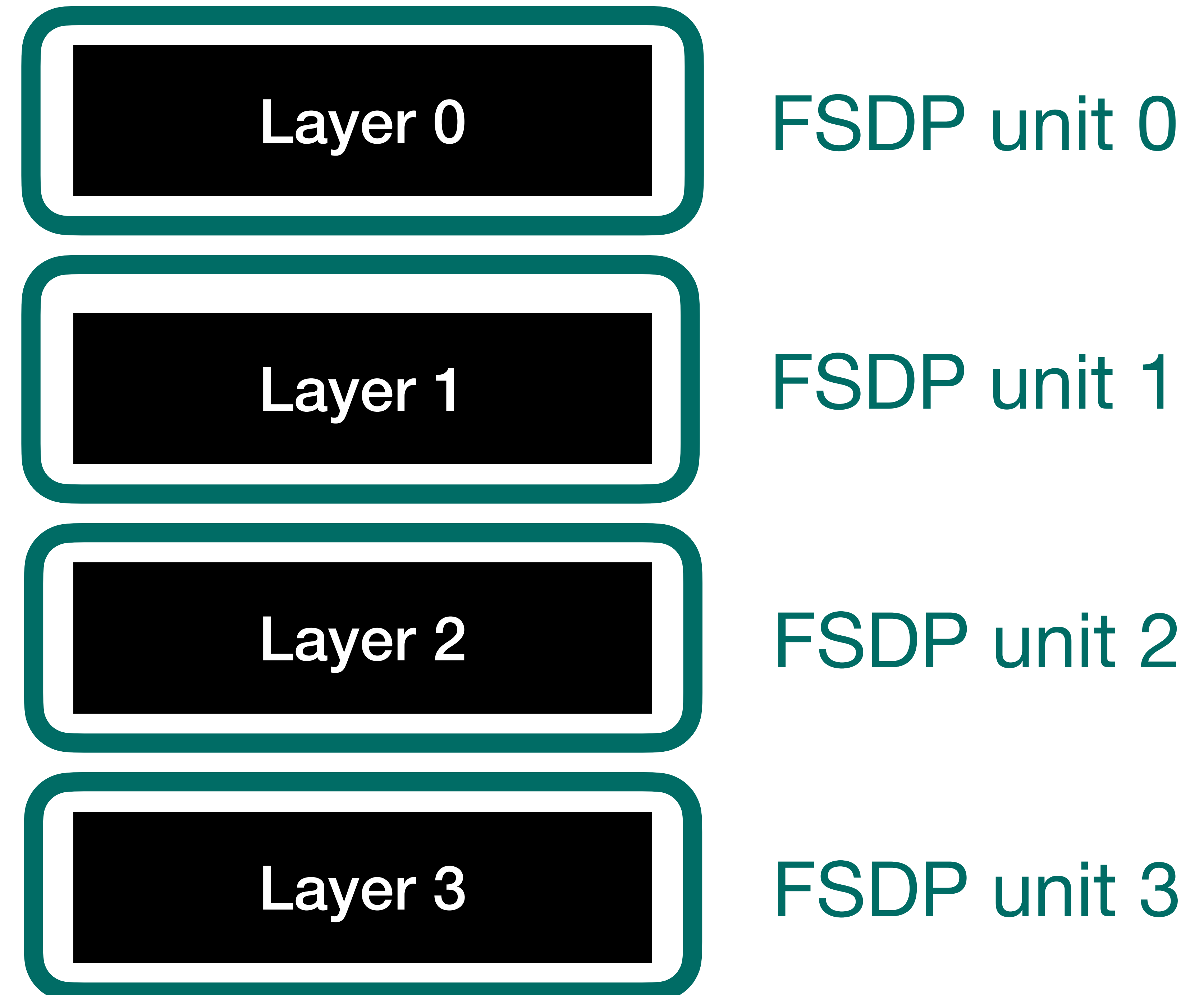
And then do backwards pass!

FSDP Implementation

```
FSDP forward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        forward pass for unit_i
        save activations for unit_i
        discard full weights for unit_i

FSDP backward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        backward pass for unit_i using saved activations
        discard full weights for unit_i
        reduce-scatter gradients for unit_i
```

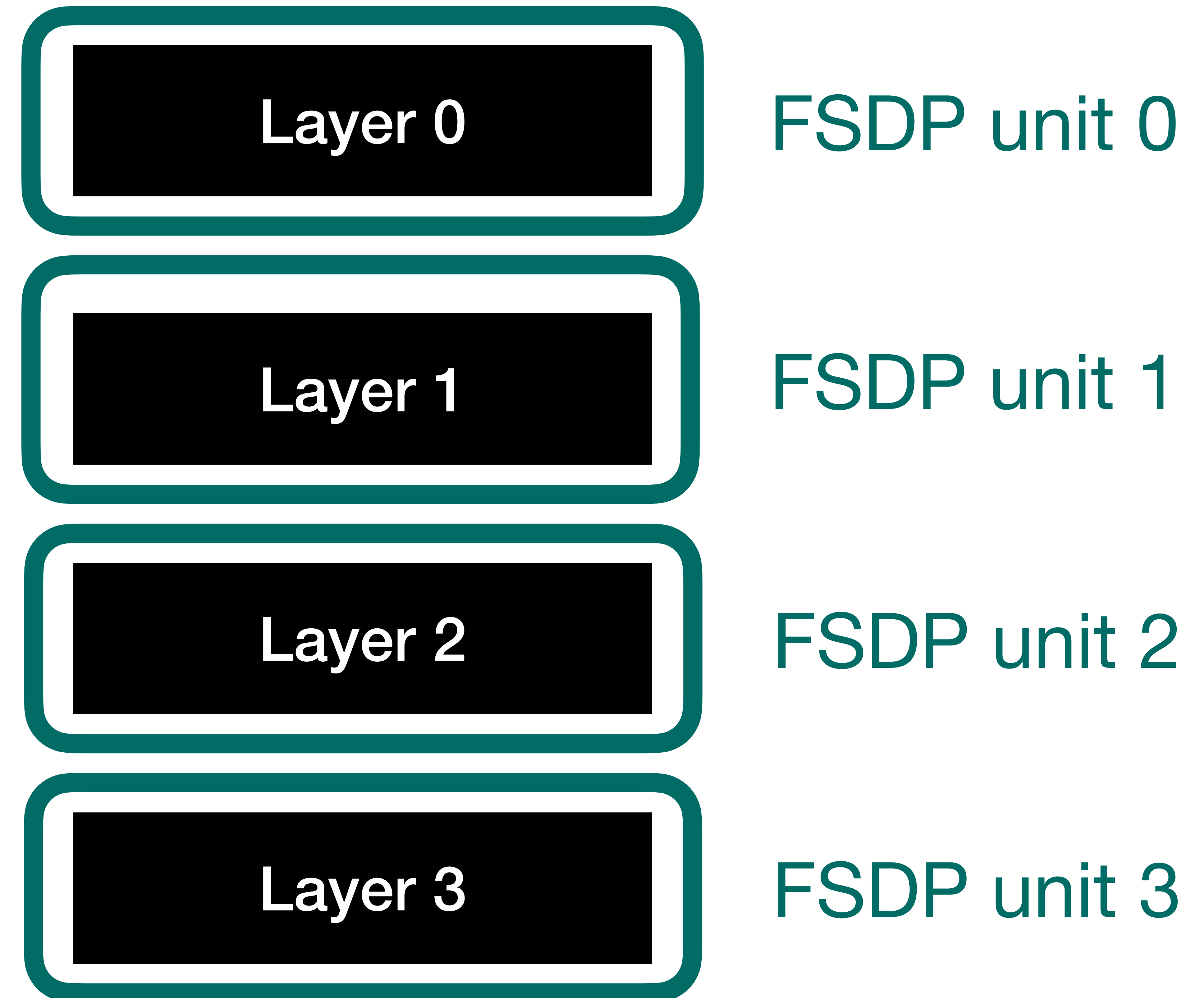
FSDP Implementation

```
FSDP forward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        forward pass for unit_i
        save activations for unit_i
        discard full weights for unit_i

FSDP backward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        backward pass for unit_i using saved activations
        discard full weights for unit_i
        reduce-scatter gradients for unit_i
```

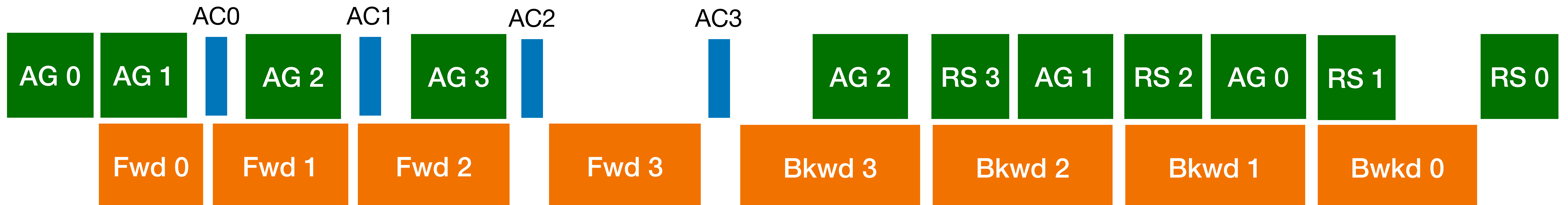


FSDP Implementation

```
FSDP forward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        forward pass for unit_i
        save activations for unit_i
        discard full weights for unit_i

FSDP backward pass:
    for unit_i in units:
        all-gather full weights for unit_i
        backward pass for unit_i using saved activations
        discard full weights for unit_i
        reduce-scatter gradients for unit_i
```



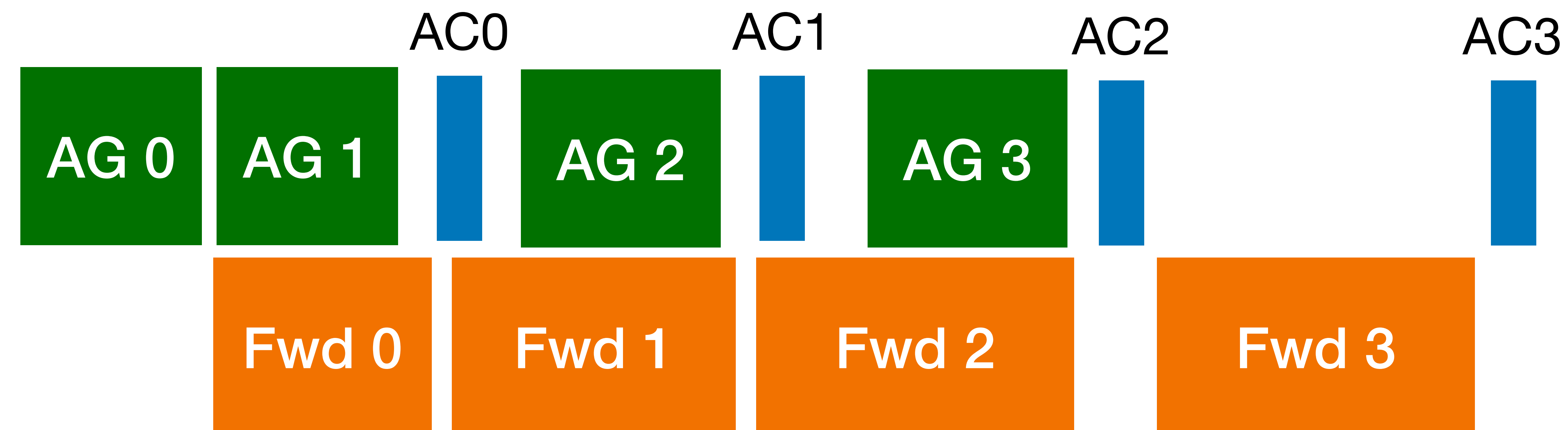
AG = all-gather, AC = activation checkpoint, RS = reduce-scatter



Computation vs communication overlaps

Cartoon is idealized setup. How much can we actually overlap?

Let's assume FSDP units are layers and we're looking at the forward pass for now



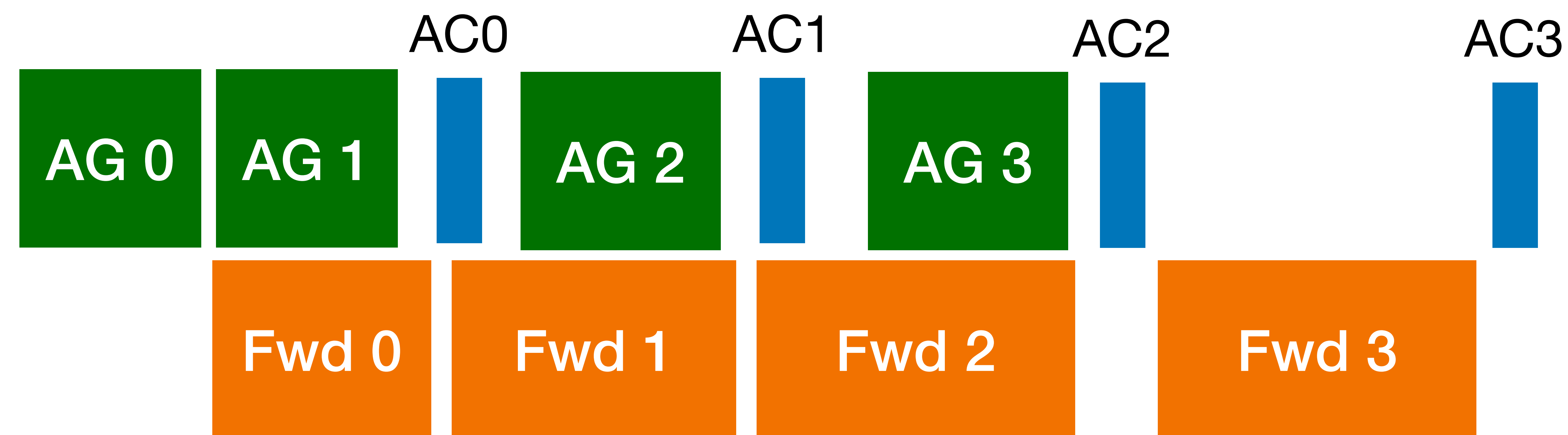
In theory, we are bottlenecked only when it takes longer to AG the next layer than computing the forward pass on the layer before it. When does this happen?

Today

- Recap++:
- Pipeline Parallelism
- FSDP
- ✓ Theoretical Considerations
- Tensor Parallelism

Computation vs communication overlaps

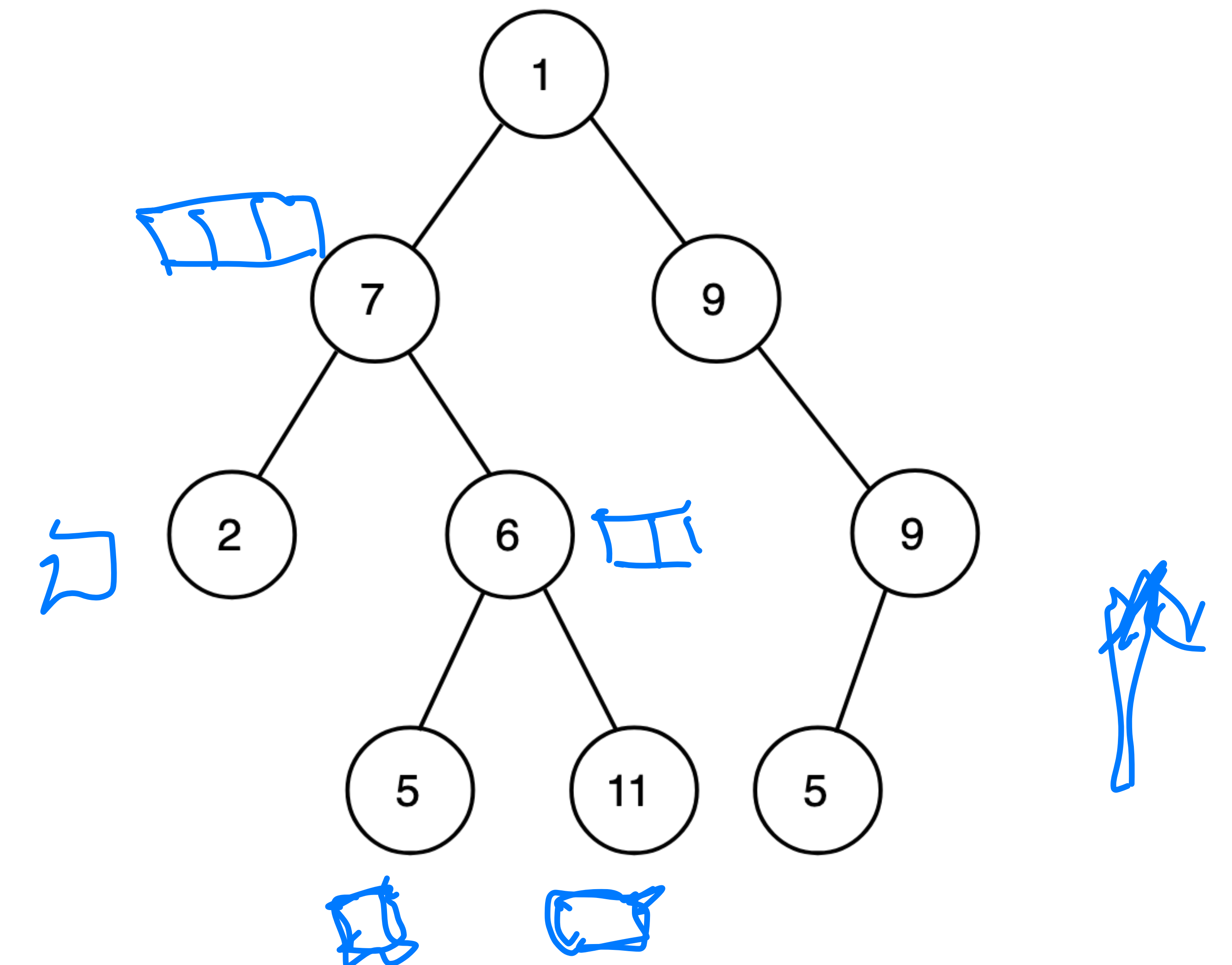
In theory, we are bottlenecked only when it takes longer to all-gather the next layer than computing the forward pass on the layer before it



$$T^{\ell+1}_{\text{all-gather}} \geq T^{\ell}_{\text{forward}}$$

All-gather Time

$$T_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} p N_{\text{message}}$$



α is startup latency to send a message, β is bandwidth,

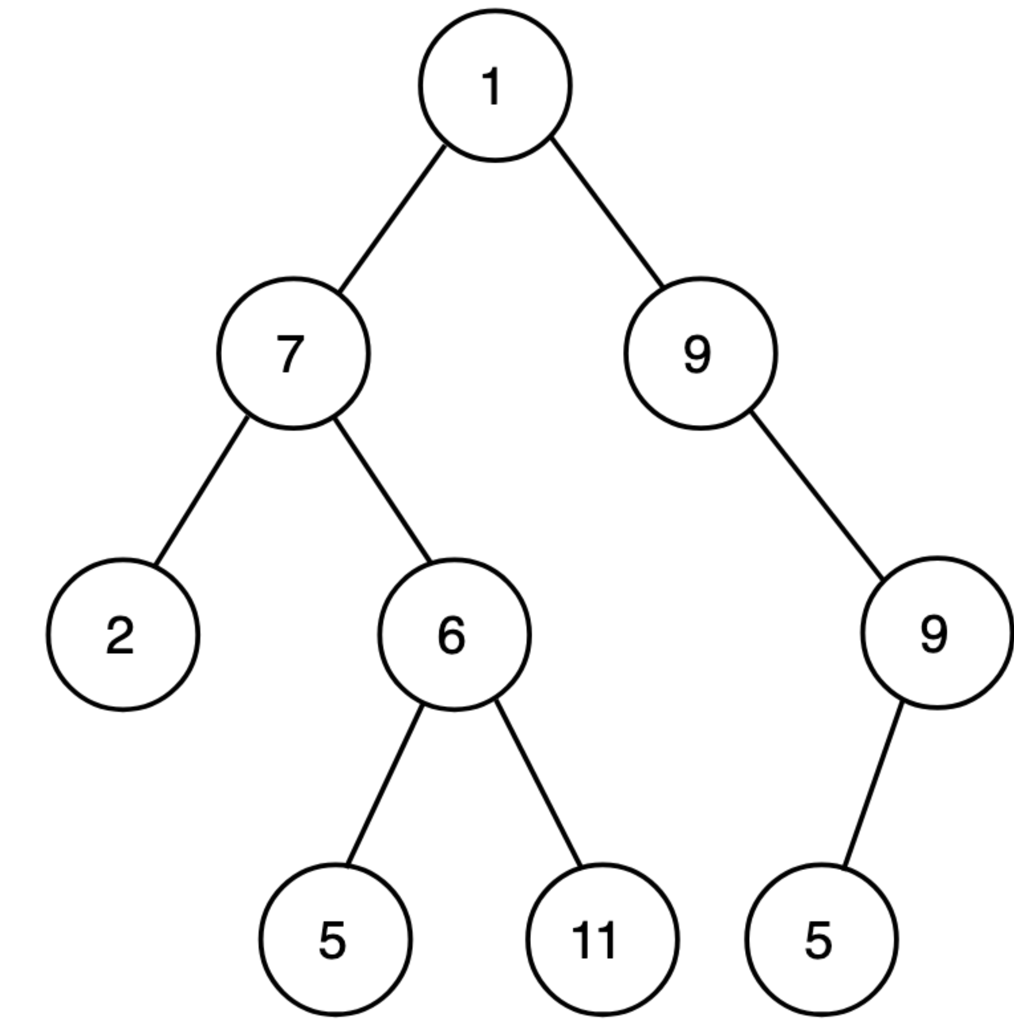
p **is number GPUs/nodes** (11 in picture),

N_{message} **is the size of the message/tensor** to send (stored on nodes)

(in our setting, **there are two different β's (for intra/inter, i.e. gpu/nodes)**

All-gather Time

$$T_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} p N_{\text{message}}$$



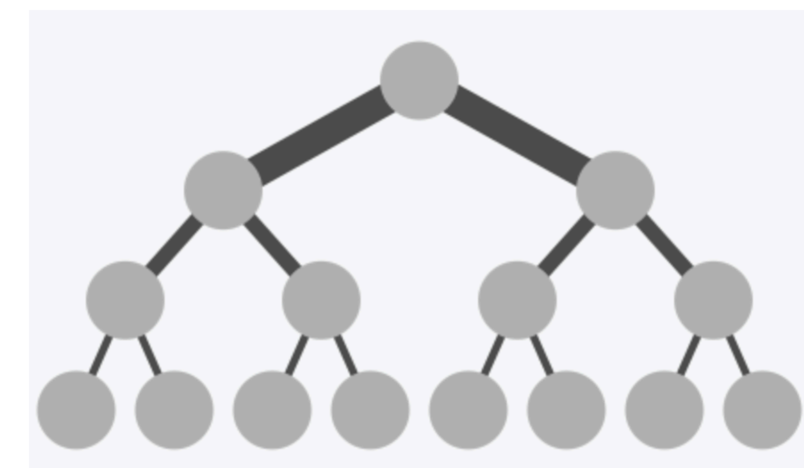
α is startup latency to send a message, β is bandwidth,

p **is number GPUs/nodes** (11 in picture),

N_{message} **is the size of the message/tensor** to send (stored on nodes)

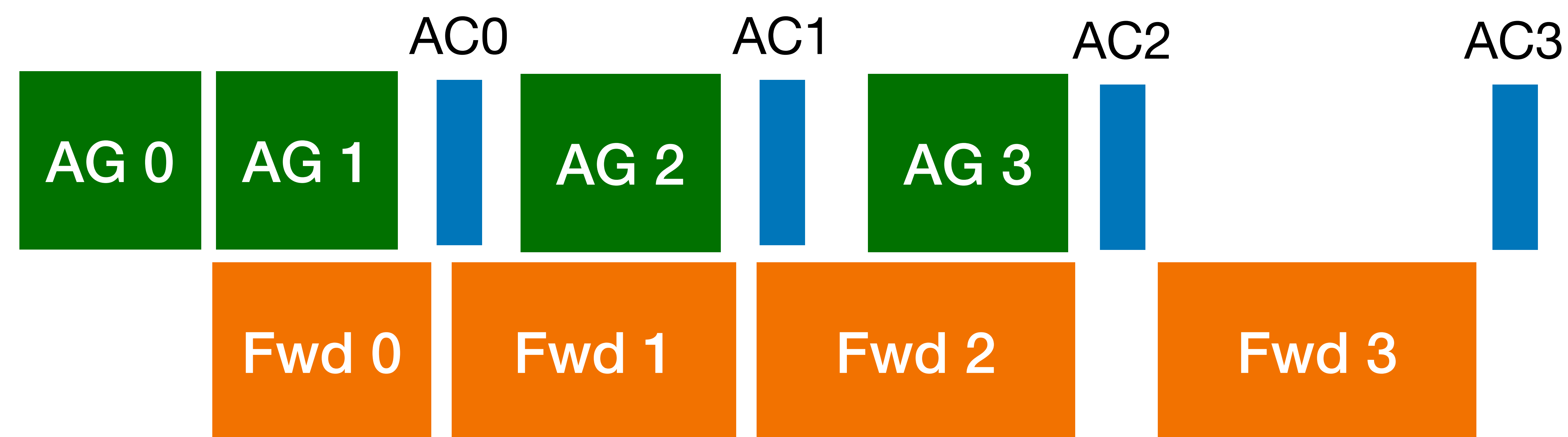
(in our setting, **there are two different β's (for intra/inter, i.e. gpu/nodes)**

Note: Nvidia uses “fat tree”

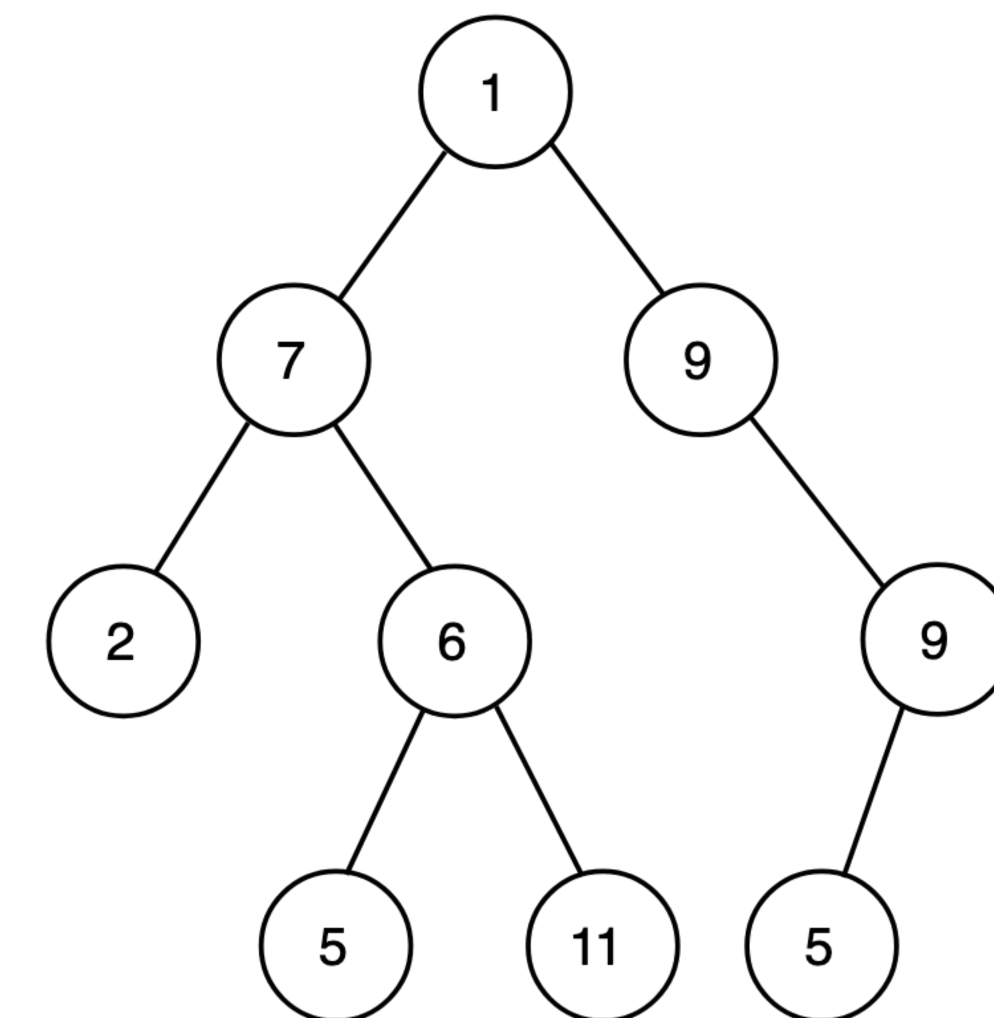


Computation vs communication overlaps

In theory, we are bottlenecked only when it takes longer to AG the next layer than computing the forward pass on the layer before it



$$T^{\ell+1}_{\text{all-gather}} \geq T^{\ell}_{\text{forward}}$$



$$T^{\ell+1}_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} p N_{\text{message}}$$

α is startup latency to send a message, β is bandwidth, p is number GPUs/nodes, N_{message} is the size of the message/tensor to send

Computation vs communication overlaps

Comm bound when $T^{\ell+1}_{\text{all-gather}} \geq T^{\ell}_{\text{forward}}$

$$T^{\ell+1}_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} p N_{\text{message}}$$

$$T^{\ell+1}_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} n$$

We set things up so that:
$pN_{\text{message}} = n$

α is startup latency to send a message, β is bandwidth, p is number GPUs, N_{message} **is the size of the message/tensor** to send, and

n **is num params per layer**

Computation vs communication overlaps

Comm bound when $T^{\ell+1}_{\text{all-gather}} \geq T^{\ell}_{\text{forward}}$

$$T^{\ell+1}_{\text{all-gather}} = \alpha \log p + \frac{1}{\beta} n$$

$$T^{\ell}_{\text{forward}} = \frac{BSn}{C}$$

Batch size B, seq length S,
FLOPs/GPU C

α is startup latency to send a message, β is bandwidth, p is number GPUs,
n is num params per layer

Computation vs communication overlaps

$$\alpha \log p \leq \left(\frac{BS}{C} - \frac{1}{\beta} \right) n$$

Batch size B, seq length S,
FLOPs/GPU C

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

Effectively balancing compute per parameter with how quickly those parameters can be communicated!

Note: if the model needs to fit across nodes, β is **internode** bandwidth which is >> bandwidth within a node

Numbers for inter and intranode

$$\alpha \log p \leq \left(\frac{BS}{C} - \frac{1}{\beta} \right) n$$

Inter (between nodes)

$\alpha = 10^{-6}$ s

$\beta = 50$ GB/s

$C = 1500$ TFLOP/s

$S = 8 \times 10^3$

$\frac{1}{\beta} = 2 \times 10^{-11}$ s/byte

$B \approx 10$

Intra (within a node)

$\alpha = 10^{-6}$ s

$\beta = 1$ TB/s

$C = 1500$ TFLOP/s

$S = 8 \times 10^3$

$\frac{1}{\beta} = 1 \times 10^{-12}$ s/byte

Scaling FSDP: where does it break?

$$\alpha \log p \leq \left(\frac{BS}{C} - \frac{1}{\beta}\right) n$$

Batch size B, seq length S,
FLOPs/GPU C

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

$p \leq 2^{\alpha(\cdots)n}$ > 0

Also: if the right hand side is “non-trivially” positive, then we can make p very large!
(so where does FSDP break?)

For internode: batch size O(10) is needed but note that we have to have enough memory to keep the activations around (which also grows with batch size)!

What about TPUs?

$$\alpha \log p \leq \left(\frac{BS}{C} - \frac{1}{\beta}\right) n$$

For TPUs, β is absurdly large

The following table shows the key chip specifications and their values for v5e.

Key chip specifications	v5e values
Peak compute per chip (bf16)	197 TFLOPs
HBM2 capacity and bandwidth	16 GB, 819 GBps
Interchip Interconnect BW	1600 Gbps

The following table shows Pod specifications and their values for v5e.

Key Pod specifications	v5e values
TPU Pod size	256 chips
Interconnect topology	2D Torus
Peak compute per Pod	100 PetaOps(Int8)
All-reduce bandwidth per Pod	51.2 TB/s
Bisection bandwidth per Pod	1.6 TB/s
Data center network bandwidth per Pod	6.4 Tbps

More generally,

$$\alpha \log p \leq \left(\frac{BS}{C} - \frac{1}{\beta}\right) n$$

Batch size B, seq length S, FLOPs/GPU C

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta}\right) n$$

What is the "unit": here, we basically want the unit to be "small", which gives us more flexibility (in TP, we will subdivide the layer)

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

We really need the res to have a “non-trivial” positive gap. So the constants matter.

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

We really need the res to have a “non-trivial” positive gap. So the constants matter.

Pipeline approach:

- We are communicating both the activations and the params (in the backward pass).
- Exactly the same issue as in FSDP. Bottlenecks inter-node communication.

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

We really need the res to have a “non-trivial” positive gap. So the constants matter.

Pipeline approach:

- We are communicating both the activations and the params (in the backward pass).
- Exactly the same issue as in FSDP. Bottlenecks inter-node communication.

FSDP:

- Here, we can effectively make d bigger due to sharding the layer within a node (larger β)
- But, we can’t keep increasing the depth (or width) due to both model and activation memory in the node

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

We really need the res to have a “non-trivial” positive gap. So the constants matter.

Pipeline approach:
- We are communicating both the activations and the params (in the backward pass).
- Exactly the same issue as in FSDP. Bottlenecks inter-node communication.

FSDP:
- Here, we can effectively make d bigger due to sharding the layer within a node (larger β)
- But, we can’t keep increasing the depth (or width) due to both model and activation memory in the node

FSDP+pipeline:
- Use FSDP to make the intra-node model larger
- Pipeline handles between node.

More generally,

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

We really need the res to have a “non-trivial” positive gap. So the constants matter.

Pipeline approach:
- We are communicating both the activations and the params (in the backward pass).
- Exactly the same issue as in FSDP. Bottlenecks inter-node communication.

FSDP:
- Here, we can effectively make d bigger due to sharding the layer within a node (larger β)
- But, we can’t keep increasing the depth (or width) due to both model and activation memory in the node

FSDP+pipeline:
- Use FSDP to make the intra-node model larger
- Pipeline handles between node.

Many mix-match approaches

Today

- Recap++:
- Pipeline Parallelism
- FSDP
- Theoretical Considerations
- Tensor Parallelism

Tensor Parallelism, simplified

Tensor Parallelism, simplified

- Now let us consider width scaling.
  (similar idea helps with context scaling, attention, ring attention).

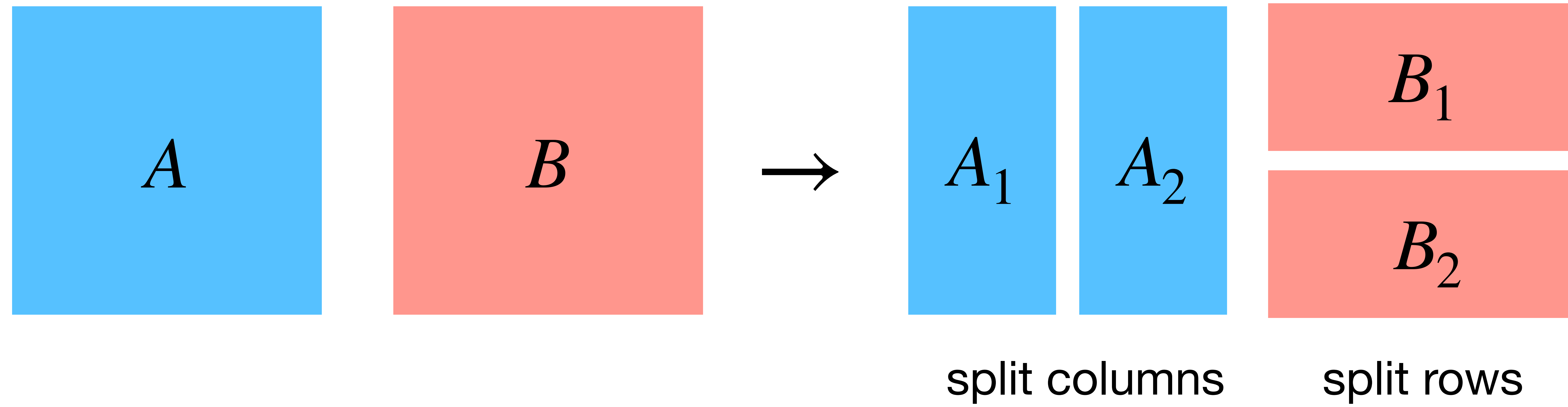
Tensor Parallelism, simplified

- Now let us consider width scaling.
  (similar idea helps with context scaling, attention, ring attention).
- Here it let's consider a simpler matrix multiplication problem of:
  Linear Network: $Y = ABX$
  - Where A, B are both $d \times d$ and X is $d \times m$.

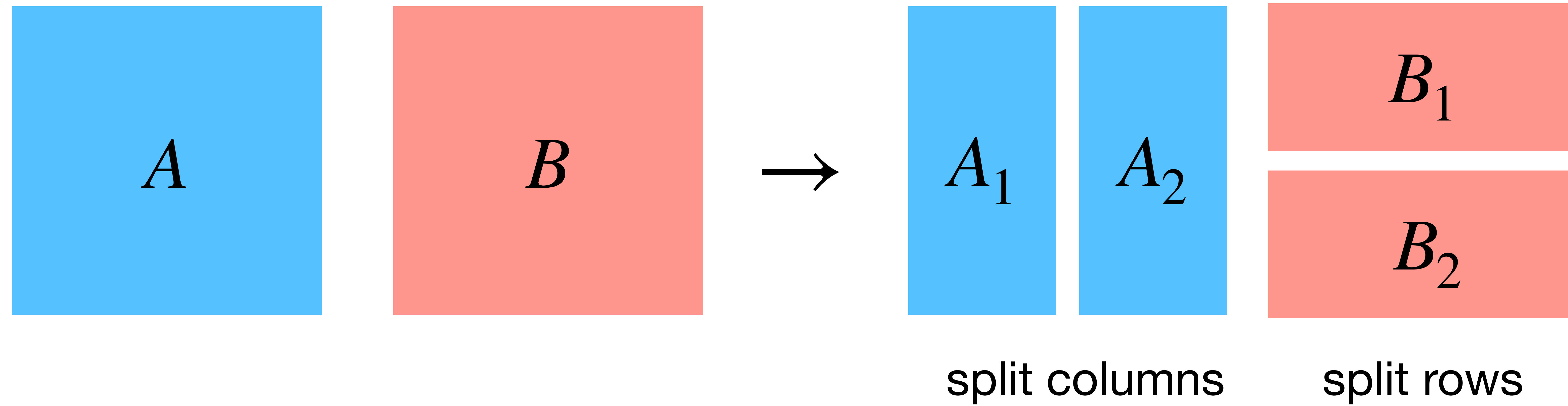
Tensor Parallelism, simplified

- Now let us consider width scaling.
  (similar idea helps with context scaling, attention, ring attention).
- Here it let's consider a simpler matrix multiplication problem of:
  Linear Network: $Y = ABX$
  - Where A, B are both $d \times d$ and X is $d \times m$.
- We will consider d to be large so that we want to break up A, B matrices.

The Basic Idea

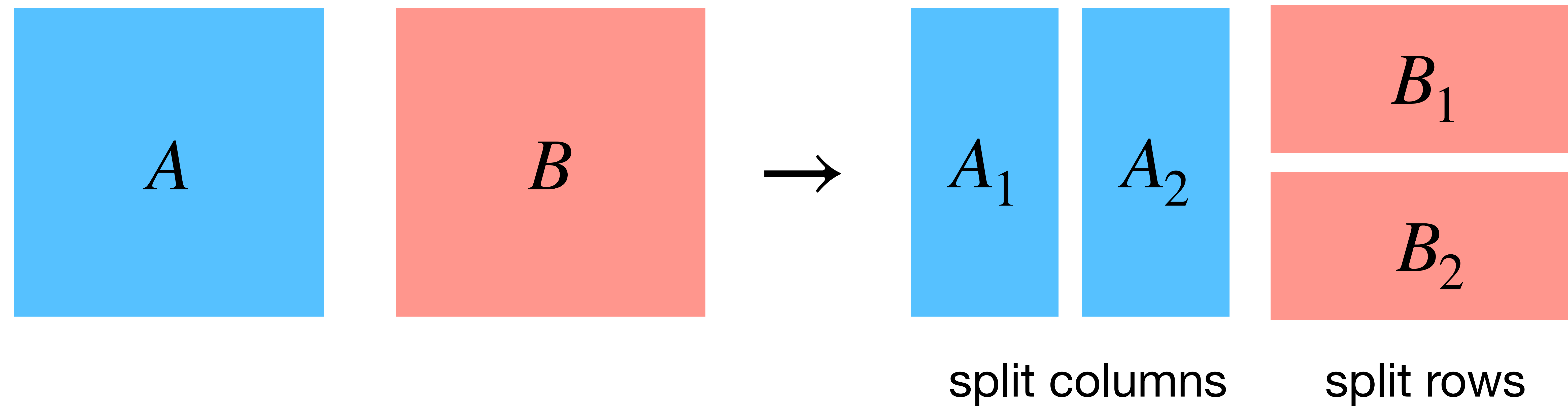


The Basic Idea



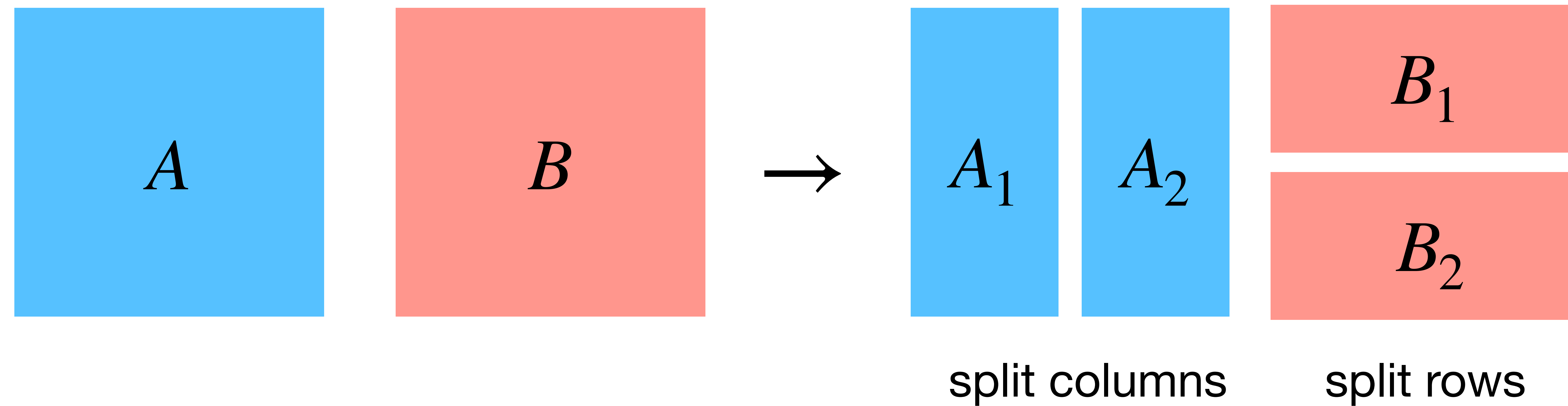
- Split A by columns and B into rows. (in general, we can shard them into more splits)

The Basic Idea



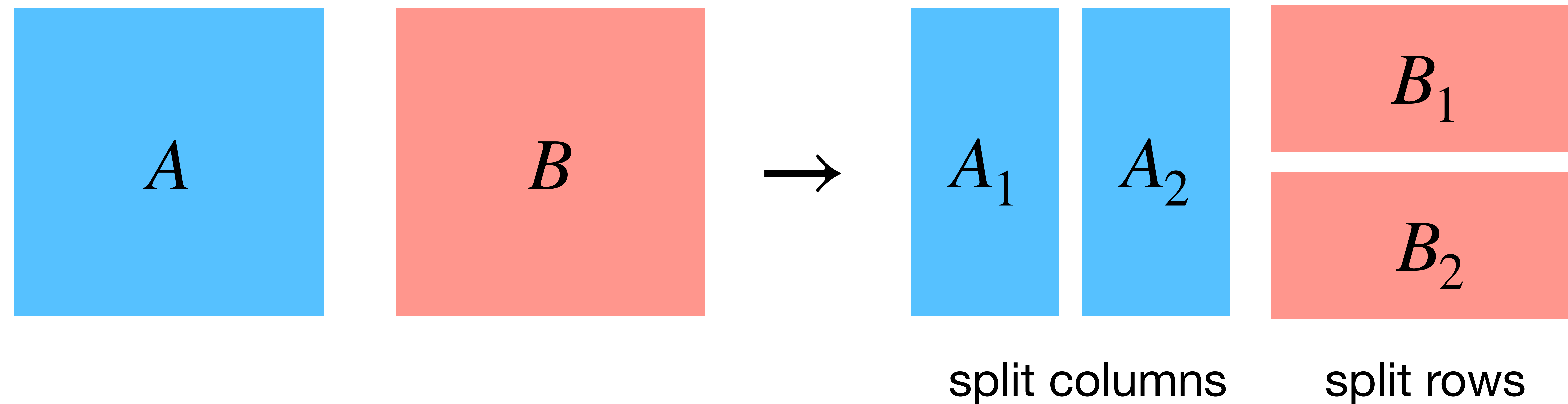
- Split A by columns and B into rows. (in general, we can shard them into more splits)
- It is easy to verify that:
  $AB = A_1B_1 + A_2B_2$
  (e.g. consider multiply by vector x)

The Basic Idea



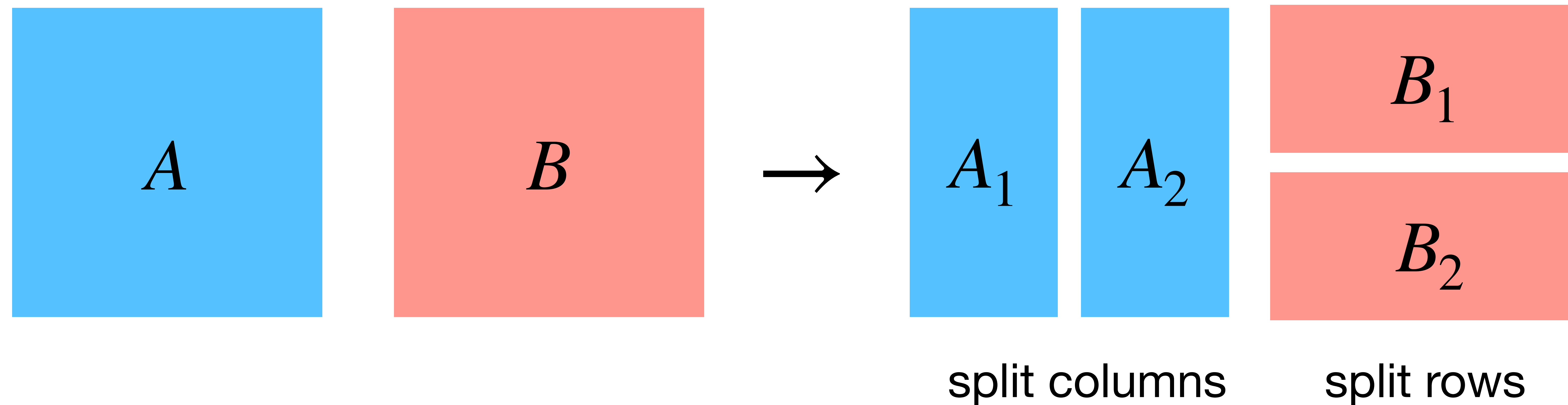
- Split A by columns and B into rows. (in general, we can shard them into more splits)
- It is easy to verify that:
  $AB = A_1B_1 + A_2B_2$
  (e.g. consider multiply by vector x)
- The “tensor” parallelism approach on two devices:

The Basic Idea



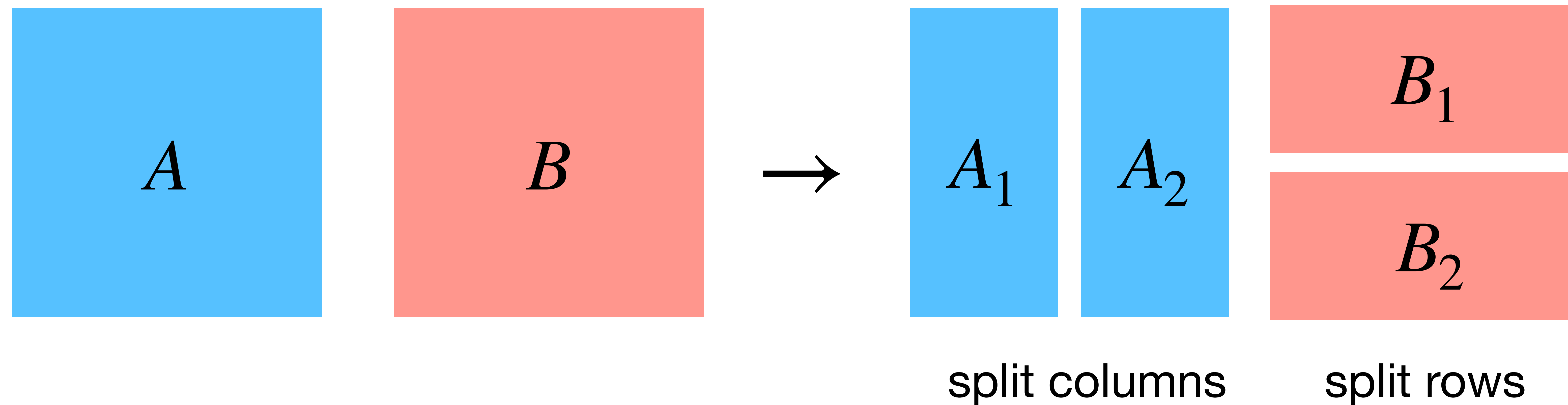
- Split A by columns and B into rows. (in general, we can shard them into more splits)
- It is easy to verify that:
  $AB = A_1B_1 + A_2B_2$
  (e.g. consider multiply by vector x)
- The "tensor" parallelism approach on two devices:
  - Shard A, B so that:
    gpu 1: holds $A_1, B_1, X,$ gpu 2: holds A_2, B_2, X

The Basic Idea



- Split A by columns and B into rows. (in general, we can shard them into more splits)
- It is easy to verify that:
  $AB = A_1B_1 + A_2B_2$
  (e.g. consider multiply by vector x)
- The "tensor" parallelism approach on two devices:
  - Shard A, B so that:
    gpu 1: holds $A_1, B_1, X,$ gpu 2: holds A_2, B_2, X
  - Each device computes:
    $y_1 = A_1B_1X$ $y_2 = A_2B_2X$

The Basic Idea



- Split A by columns and B into rows. (in general, we can shard them into more splits)
- It is easy to verify that:
  $AB = A_1B_1 + A_2B_2$
  (e.g. consider multiply by vector x)
- The "tensor" parallelism approach on two devices:
  - Shard A, B so that:
    gpu 1: holds $A_1, B_1, X,$ gpu 2: holds A_2, B_2, X
  - Each device computes:
    $y_1 = A_1B_1X$ $y_2 = A_2B_2X$
  - **All-reduce** the y's:
    $Y = y_1 + y_2$

Discussion

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.
- We can handle larger width by many row/column splits with TP.

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.
- We can handle larger width by many row/column splits with TP.
  - But we have to all-reduce the answers

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.
- We can handle larger width by many row/column splits with TP.
  - But we have to all-reduce the answers
- **As we grow d, then we can make the serial runtime smaller.**

More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.
- We can handle larger width by many row/column splits with TP.
  - But we have to all-reduce the answers
- **As we grow d, then we can make the serial runtime smaller.**
  - Again, we can do this provided our RHS gap holds.

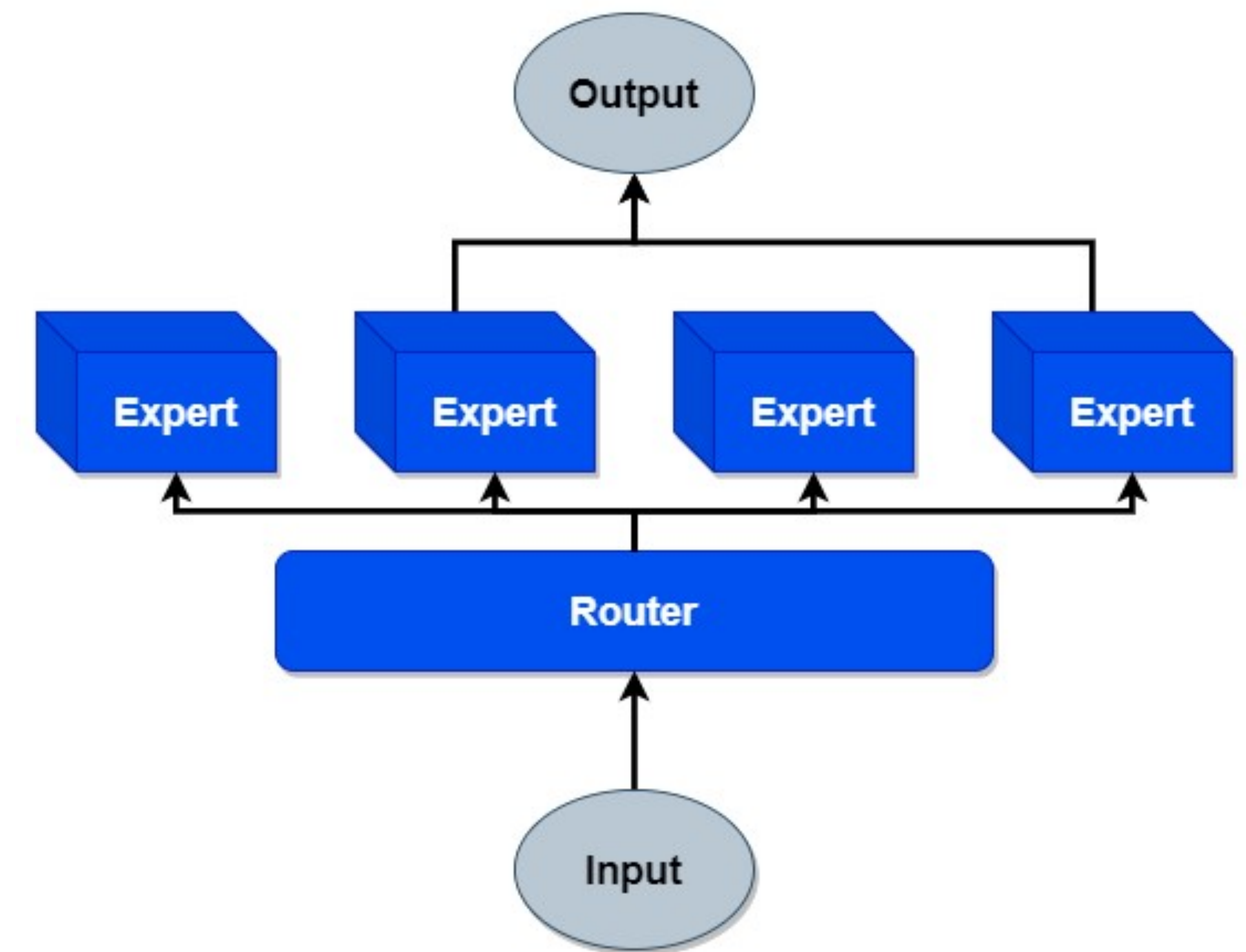
More general considerations and scalings:

$$\alpha \log p \leq \left(\frac{B \cdot \text{compute_per_param}}{C} - \frac{1}{\beta} \right) n$$

α is startup latency to send a message, β is bandwidth, p is number GPUs, n is num params per layer

- The approach is very flexible.
- We can handle larger width by many row/column splits with TP.
  - But we have to all-reduce the answers
- **As we grow d, then we can make the serial runtime smaller.**
  - Again, we can do this provided our RHS gap holds.

- **Mixture of Experts:** Different compute_par_param scaling



GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	4	131,072	16	16M	380	38%

Table 4 Scaling configurations and MFU for each stage of Llama 3 405B pre-training. See text and Figure 5 for descriptions of each type of parallelism.

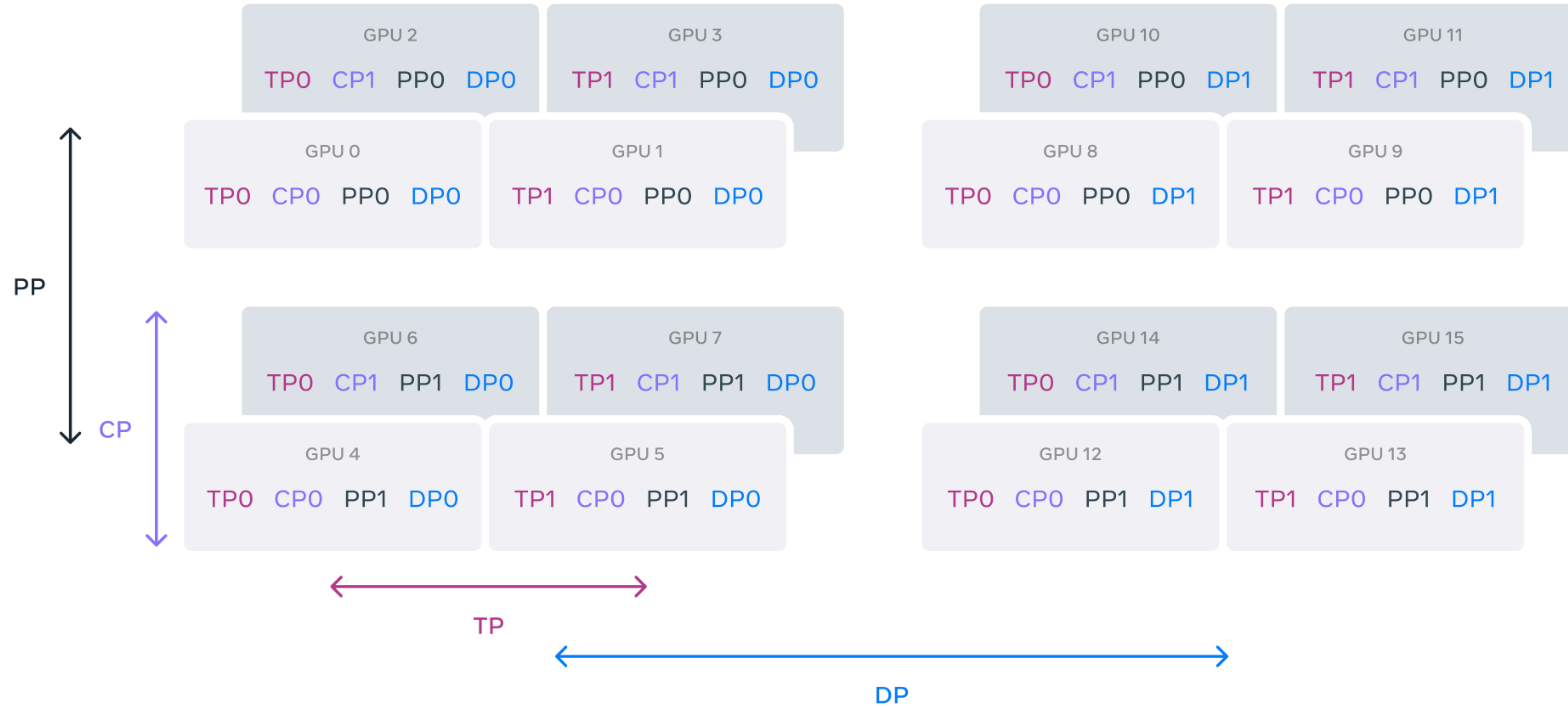


Figure 5 Illustration of 4D parallelism. GPUs are divided into parallelism groups in the order of [TP, CP, PP, DP], where DP stands for FSDP. In this example, 16 GPUs are configured with a group size of |TP|=2, |CP|=2, |PP|=2, and |DP|=2. A GPU's position in 4D parallelism is represented as a vector, $[D_1, D_2, D_3, D_4]$, where D_i is the index on the i-th parallelism dimension. In this example, GPU0[TP0, CP0, PP0, DP0] and GPU1[TP1, CP0, PP0, DP0] are in the same TP group, GPU0 and GPU2 are in the same CP group, GPU0 and GPU4 are in the same PP group, and GPU0 and GPU8 are in the same DP group.

Thanks!

1. Lots of parallelization approaches.
2. communication to compute ratio is important!