ECE7115 ~~Multimodal VLM~~ LLM

9. Parallelism

Spring 2026

Namhyuk Ahn, Inha University

Last Week: GPUs (Memory)

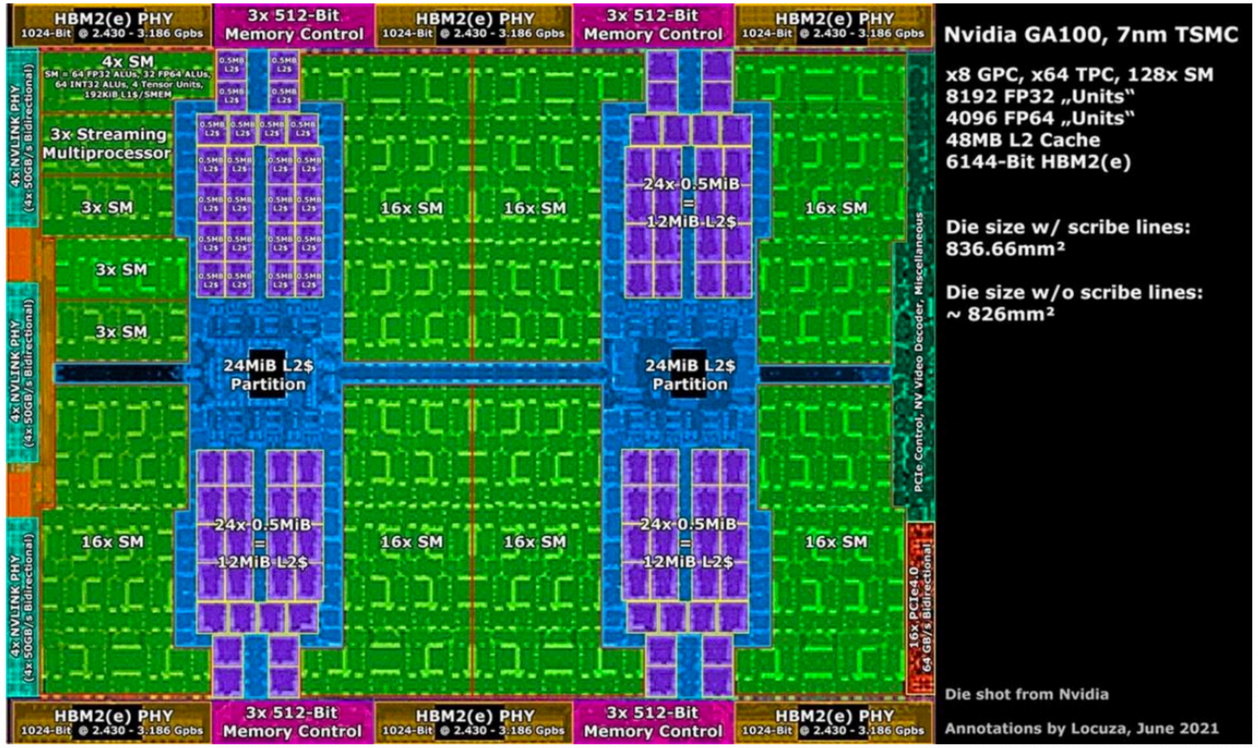
- L1 cache and shared memory is inside the SM
- L2 is on die, and global memory are the memory chips next to the GPU
- ❗❗❗: The closer the memory to the SM, the faster it is

TABLE IV
THE MEMORY ACCESSES LATENCIES

Memory type	*CPI (cycles)*
Global memory	290
L2 cache	200
L1 cache	33
Shared Memory (ld/st)	(23/19)

For example (A100),

- DRAM: 80GB (big, slow)
- L2: 40MB
- L1: 192KB per SM (small, fast)



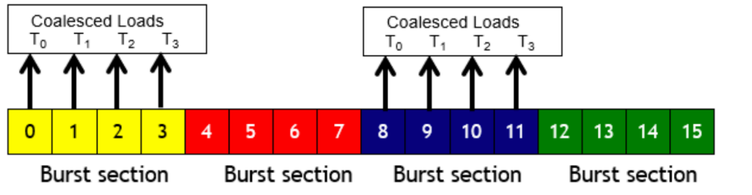
Last Week: Arithmetic Intensity

- Operational efficiency of an algorithm relative to data movement
  - $\mathbf{I}$ = Total Operations (FLOPs) / Total Communication (Bytes)
  - Higher $\mathbf{I}$: Efficient; high volume of computation per data fetch
  - Lower $\mathbf{I}$: Inefficient; performance is bottlenecked by data movement

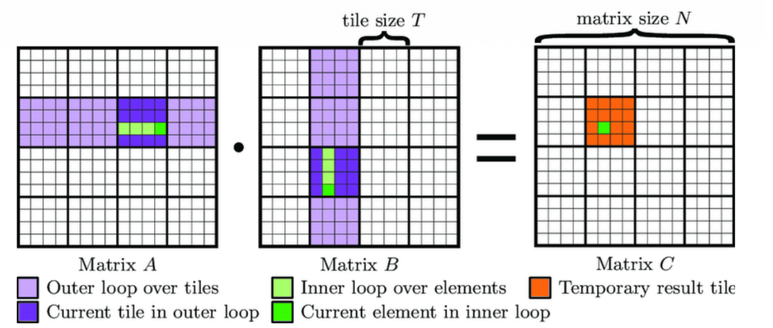
- Peak arithmetic intensity (ridge point)
  - Every hardware architecture has its own optimal threshold $\mathbf{I}_{ridge}$
    - e.g. H100 has 295 FLOPs/Byte
  - $\mathbf{I} < \mathbf{I}_{ridge}$: Comm.-bound (perf. limited by memory bandwidth)
  - $\mathbf{I} > \mathbf{I}_{ridge}$: Compute-bound (fully utilizing compute capacity)

Last Week: GPU Performance

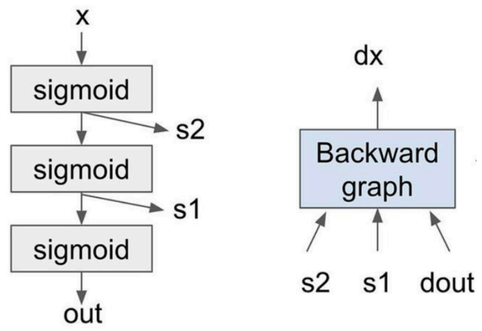
- Reducing memory accesses
  - Coalescing (get many data at once)
  - Kernel fusion (reduce data movement)



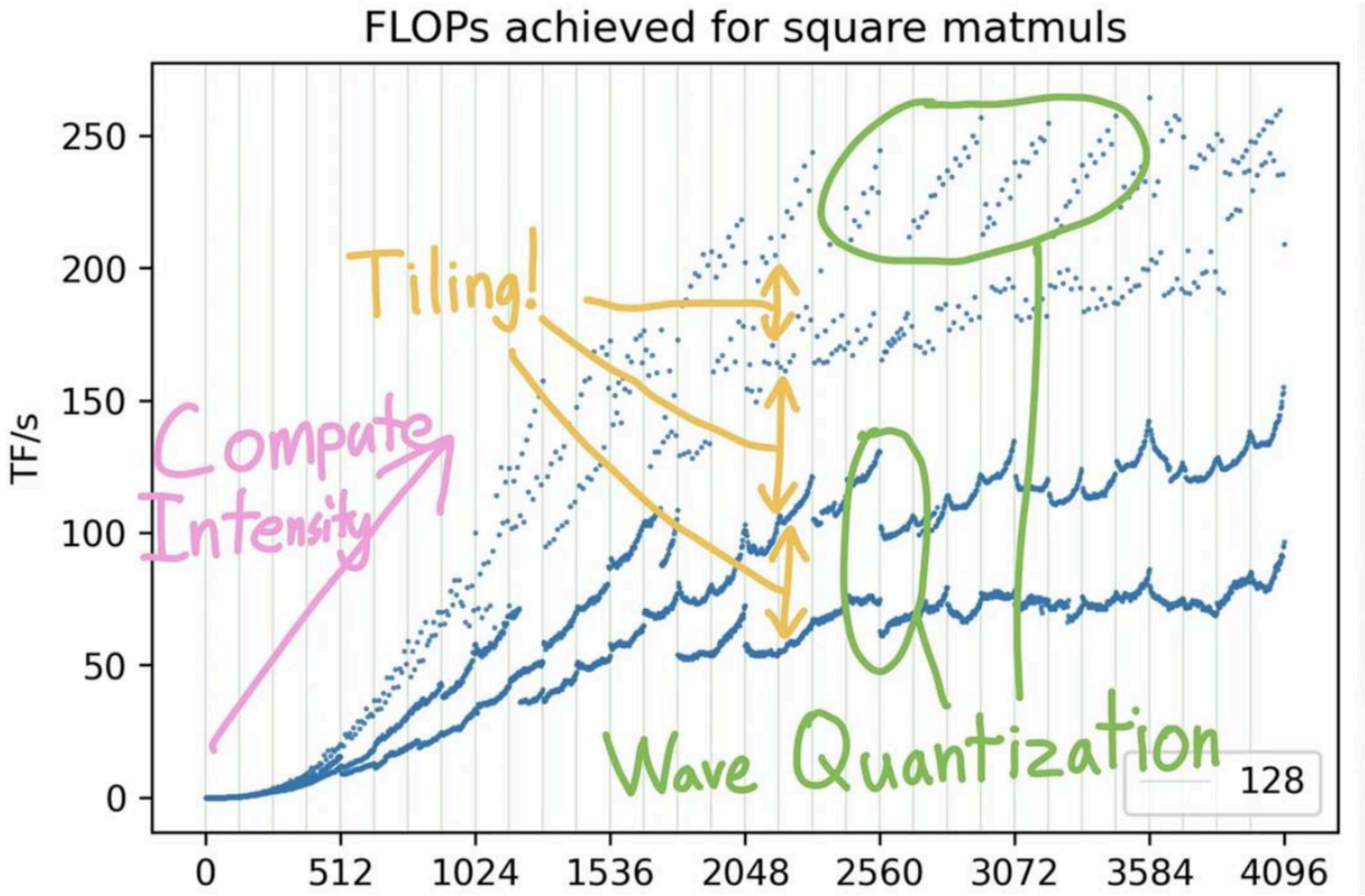
- Move memory to shared memory
  - Tiling (try my best to keep in shared mem)



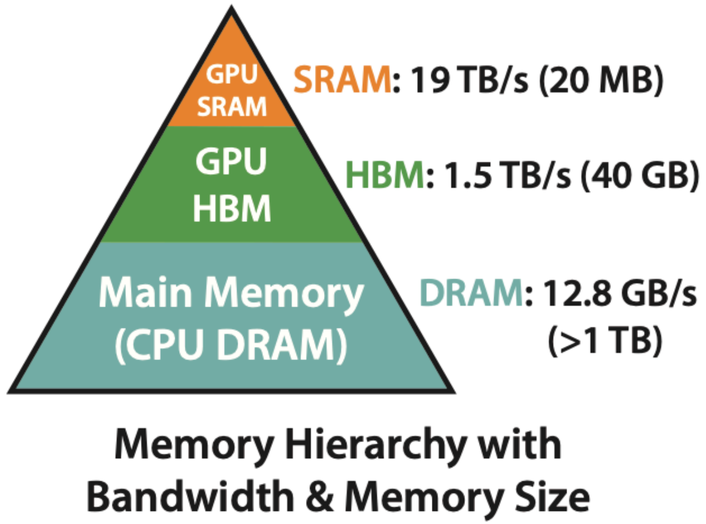
- Trade memory for compute/accuracy
  - Quantization (trade memory and accuracy)
  - Recomputation (trade memory and compute)



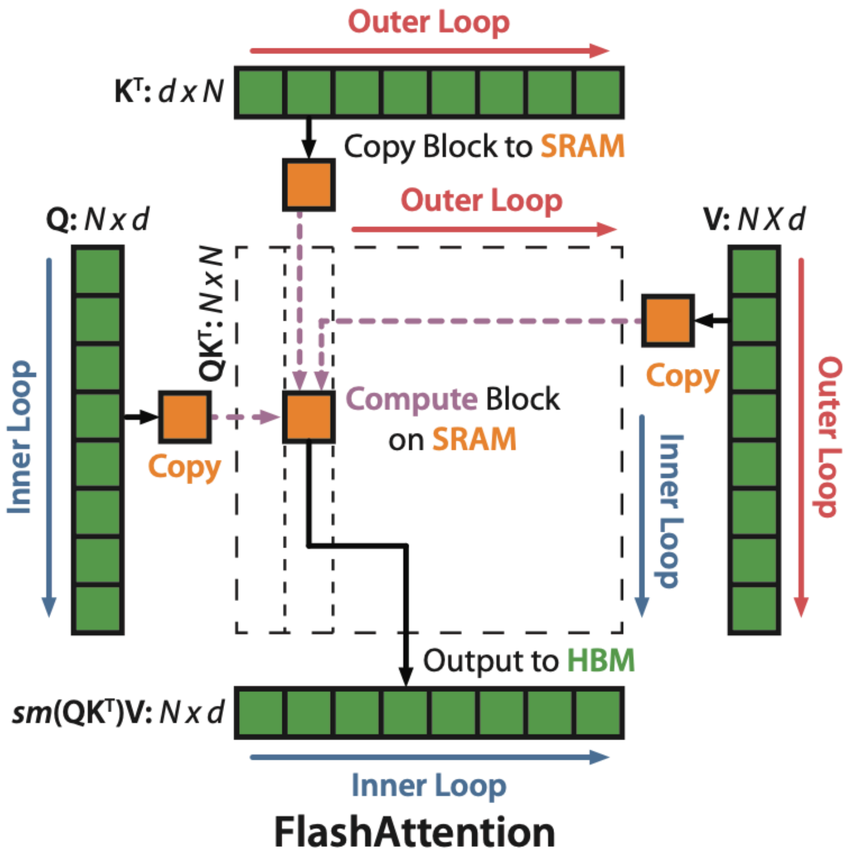
Last Week: GPU Performance



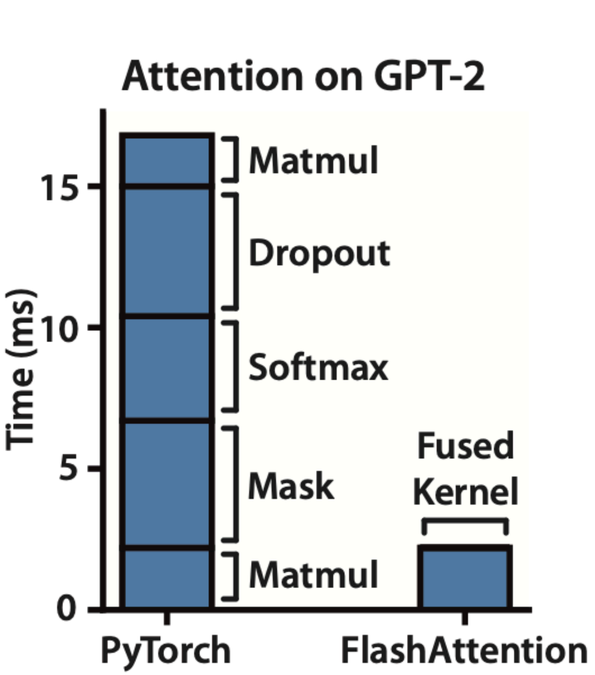
Last Week: FlashAttention



Memory Hierarchy with Bandwidth & Memory Size

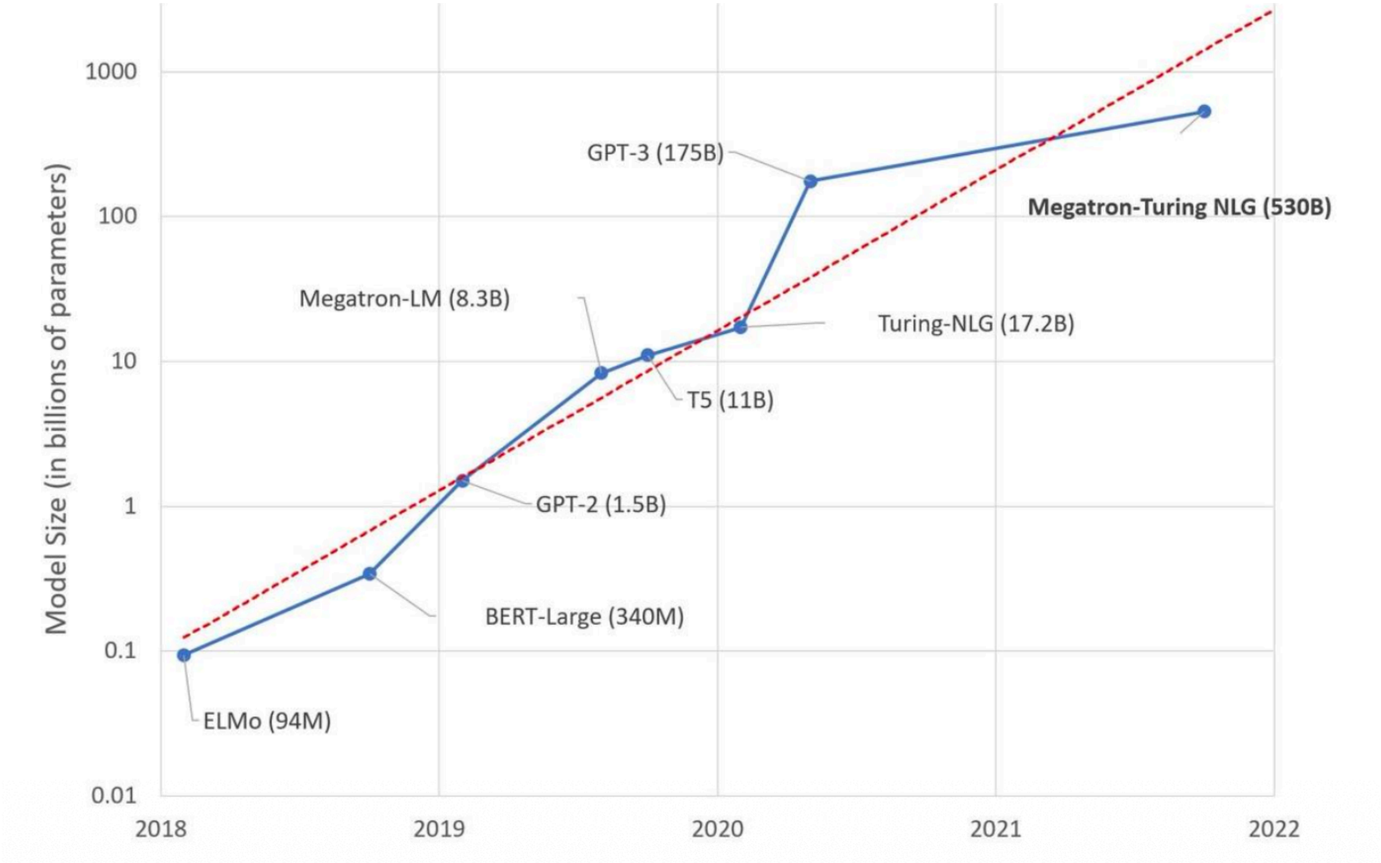


FlashAttention



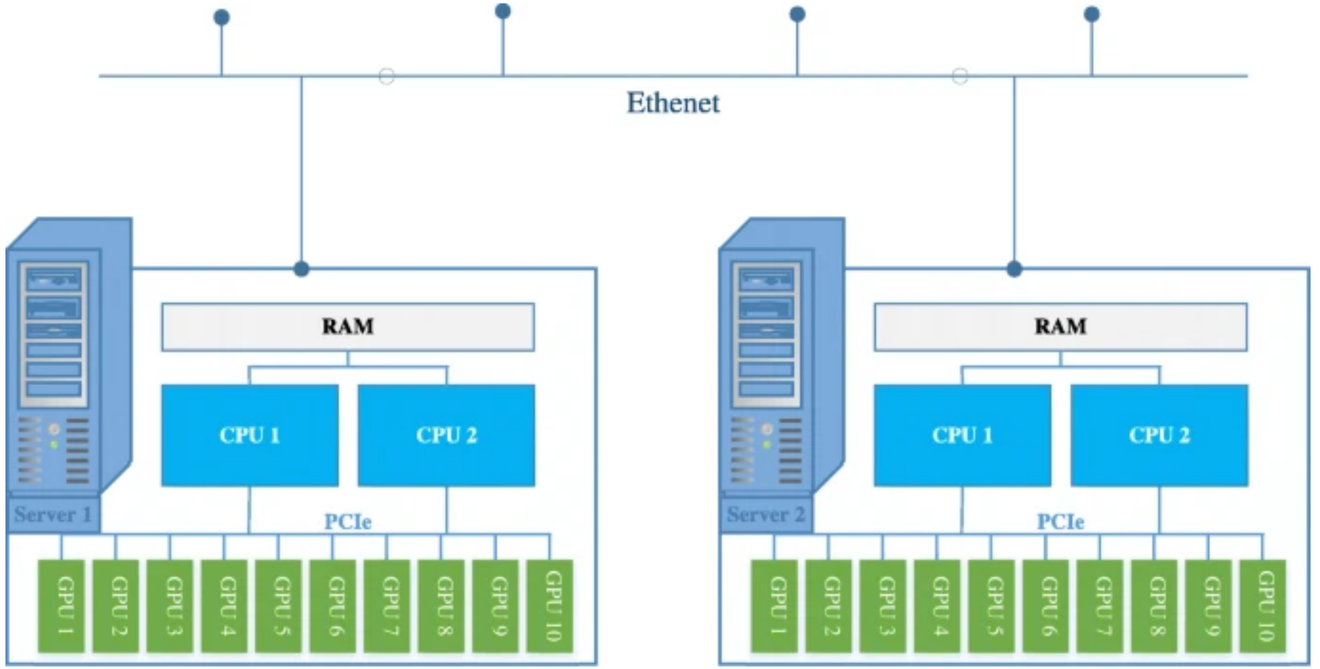
Is Single GPU All We Need?

- Models are getting really big, we cannot train them with a single GPU



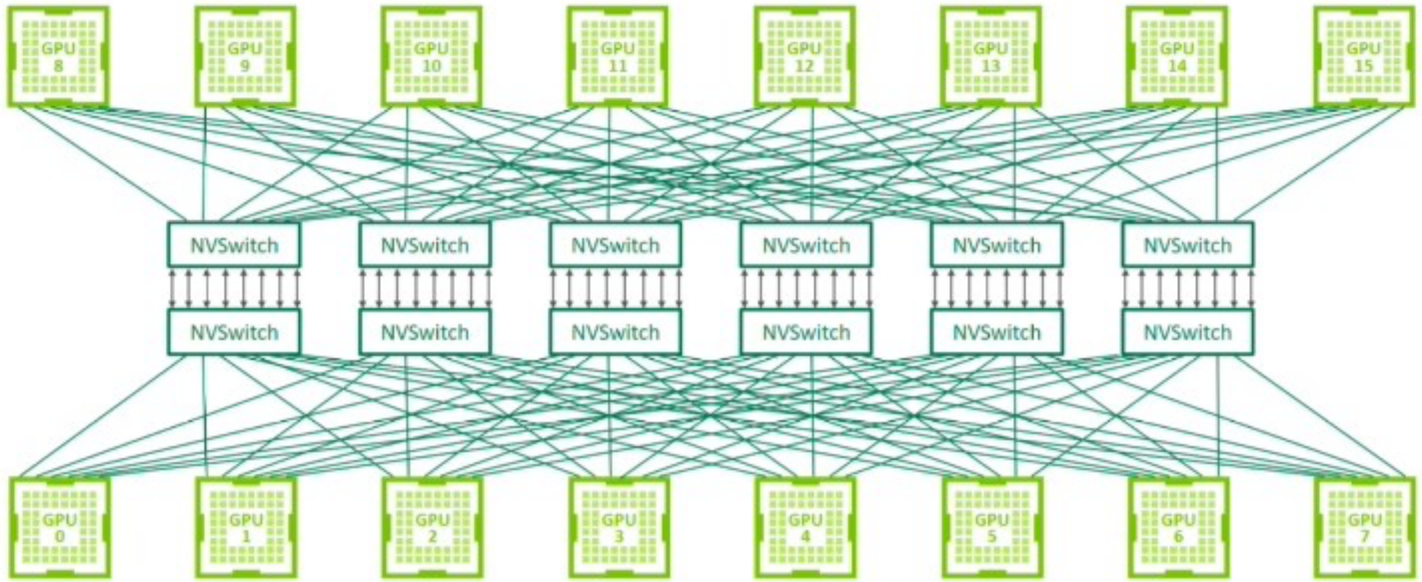
Multi-GPU, Multi-Machine

- Split up memory and compute requirements across GPUs / machines
- Classical setups:
  - GPUs on same node communicate via a PCI(e) (16 lanes => 242 GB/s)
  - GPUs on different nodes communicate via Ethernet (~200 MB/s)



Multi-GPU, Multi-Machine

- Split up memory and compute requirements across GPUs / machines
- Modern setups:
  - Within a node: NVLink connects GPUs directly, bypass CPU
  - Across nodes: NVSwitch connects GPUs directly, bypass Ethernet

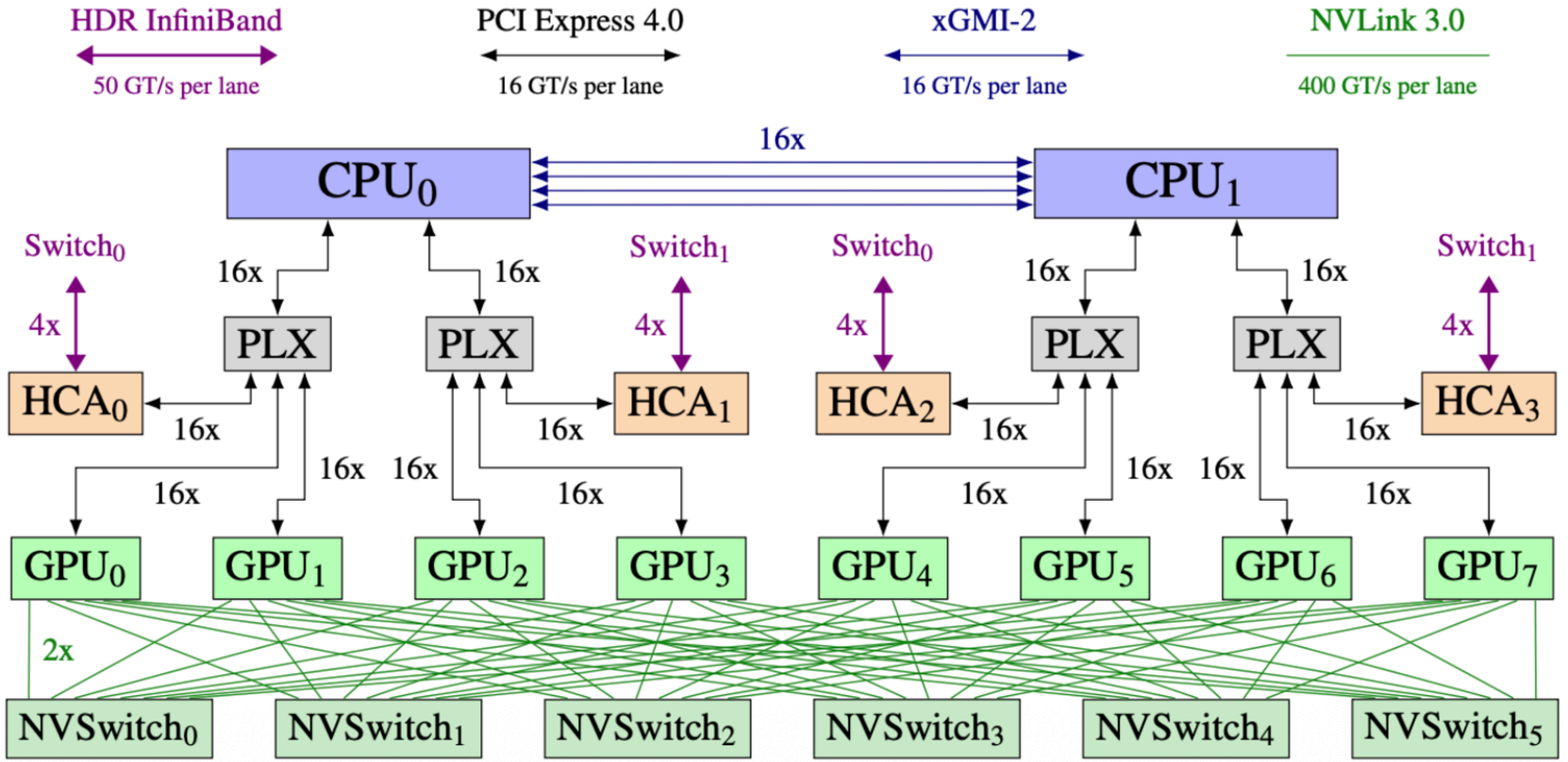


Each H100 has 18 NVLink, totally 900GB/s

Memory bandwidth for HBM is 3.9TB/s

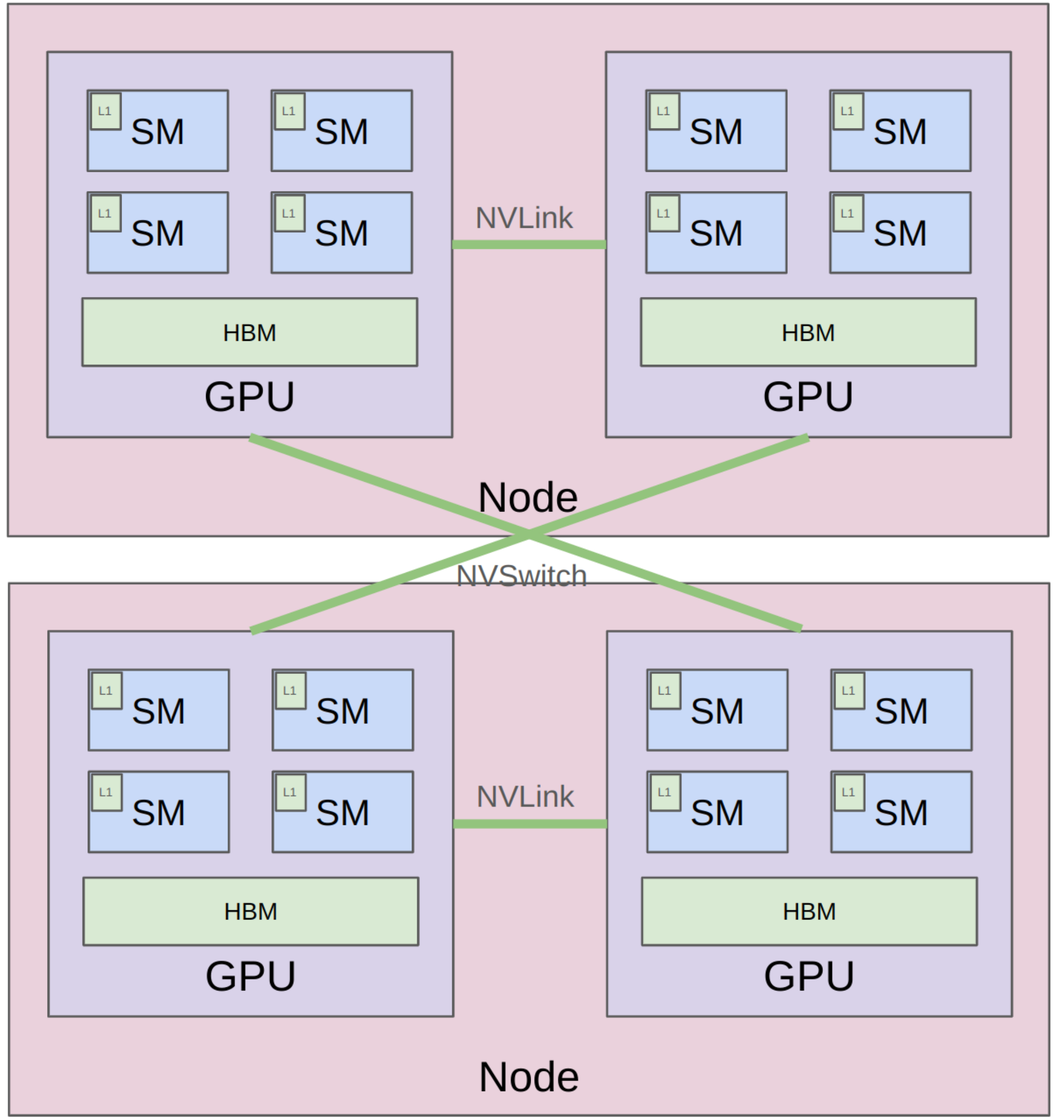
Multi-GPU, Multi-Machine

- Intra-node parallelism via high-speed interconnects
- High-speed inter-node parallelism



Multi-GPU, Multi-Machine

- Summary: communication hierarchy (from small/fast to big/slow)
  - Single node, single GPU: L1 cache / shared memory
  - Single node, single GPU: HBM
  - Single node, multi-GPU: NVLink
  - Multi-node, multi-GPU: NVSwitch



Lecture Overview

- Basic of Collective Communication

- LLM Parallelization
  - Data Parallelism (ZeRO-1 to 3)
  - Model Parallelism (Pipeline, Tensor Parallel)
  - Activation Parallelism (Sequence Parallel)
  - ND Parallelism

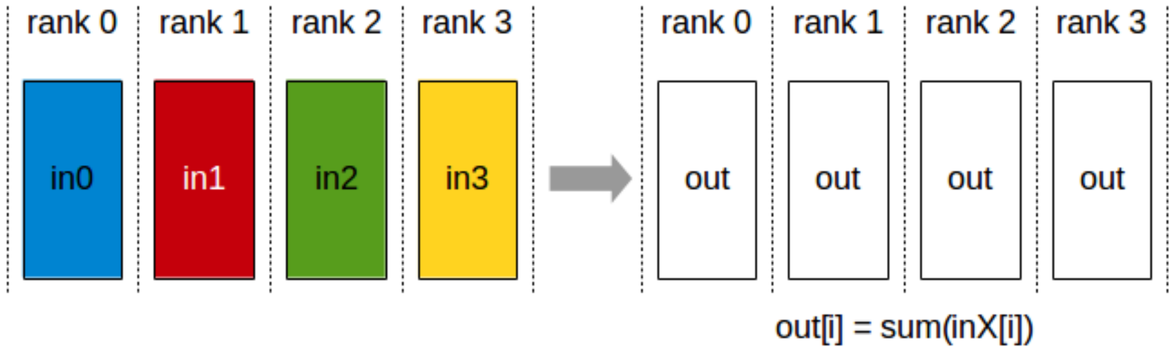
- Case Study

Collective Communication

Collective Communication

- Note: world size (# devices) is 4, and rank means device id

All-Reduce

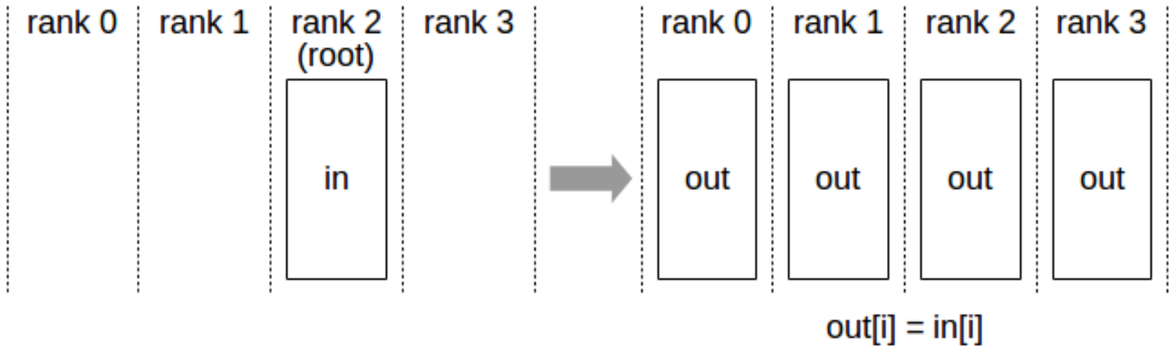


Performs reductions on data (for example, sum, min, max) across devices and stores the result in the receive buffer of every rank

Collective Communication

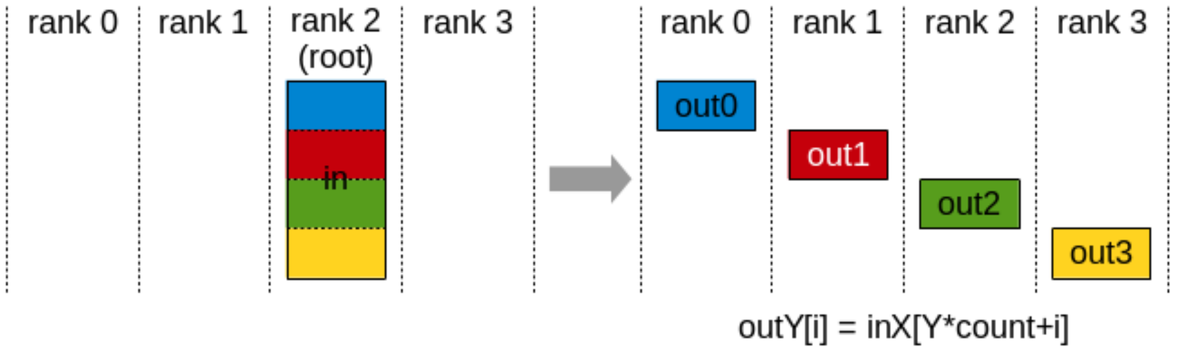
- Note: world size (# devices) is 4, and rank means device id

Broadcast



Copies an N-element buffer from the root rank to all the ranks

Scatter

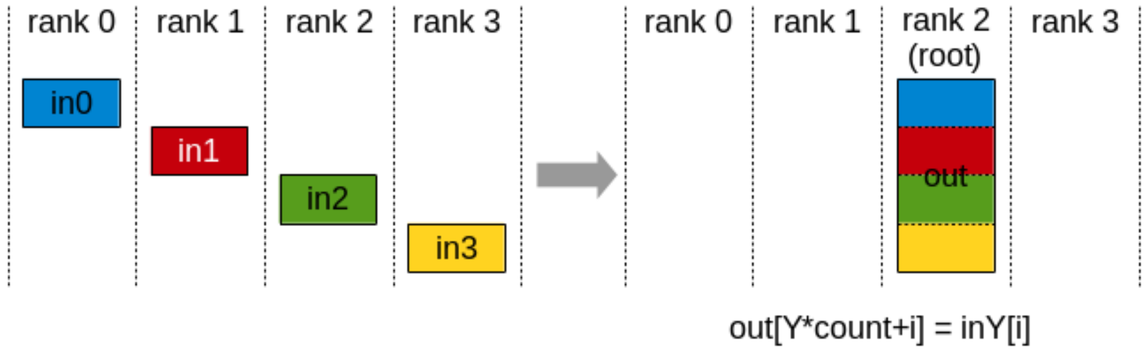


Distributes a total of N*k values from the root rank to k ranks, each rank receiving N values

Collective Communication

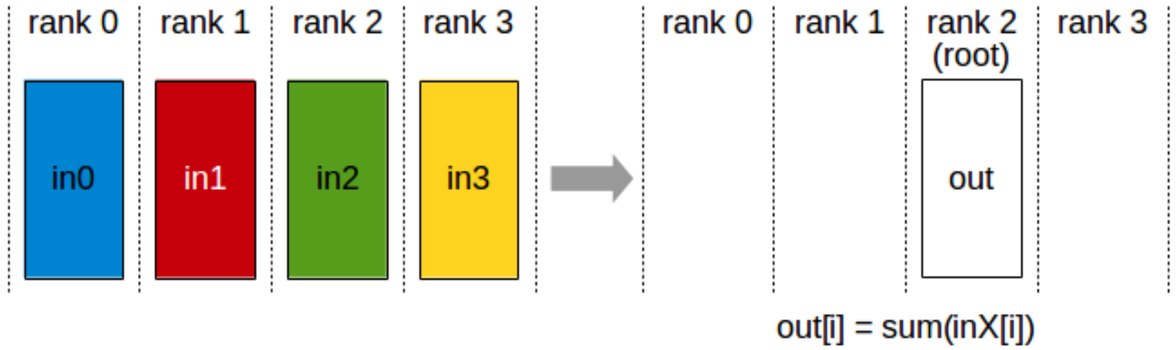
- Note: world size (# devices) is 4, and rank means device id

Gather



Gathers N values from k ranks into an output buffer on the root rank of size k*N

Reduce

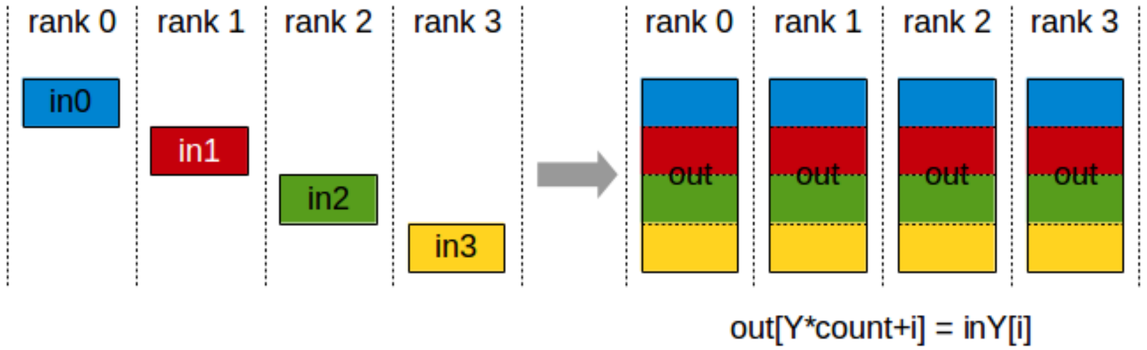


Performs the same operation as AllReduce, but stores the result only in the receive buffer of a specified root rank

Collective Communication

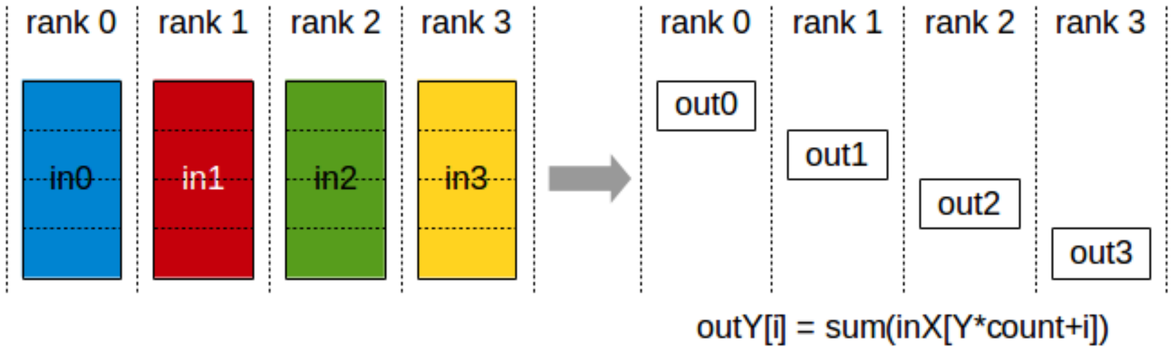
- Note: world size (# devices) is 4, and rank means device id

All-Gather



Gathers N values from k ranks into an output buffer of size k*N, and distributes that result to all ranks

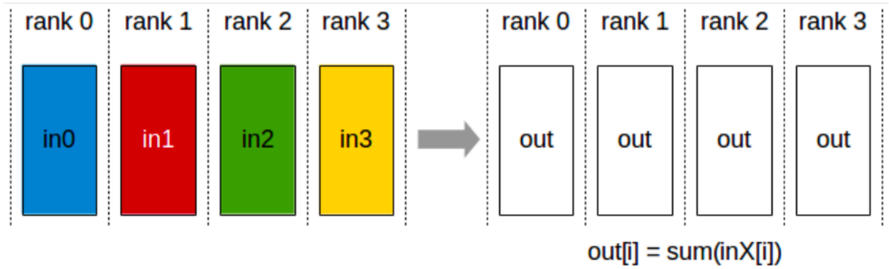
Reduce-Scatter



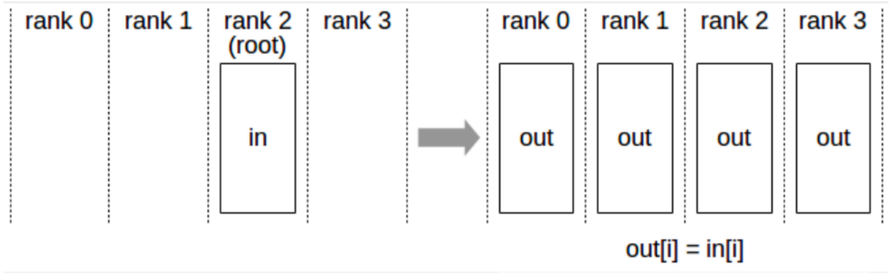
Performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index

Collective Communication

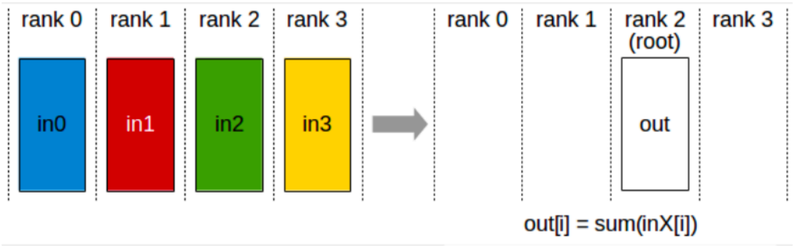
All reduce



Broadcast

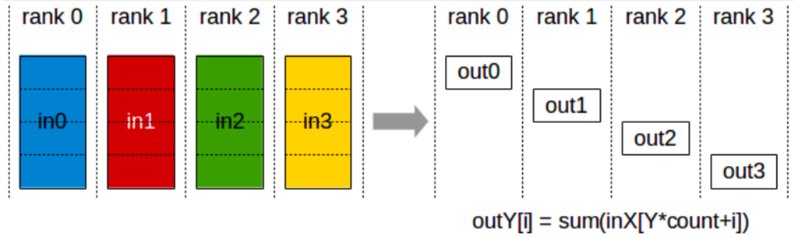


Reduce



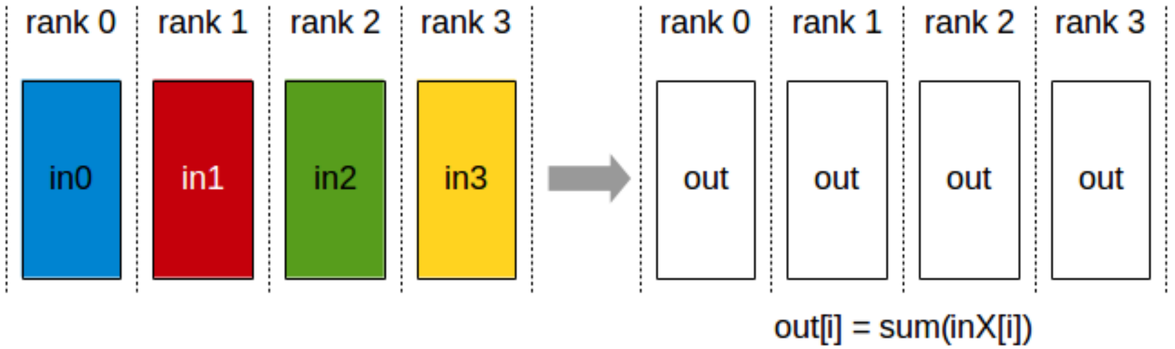
All Gather

Reduce Scatter

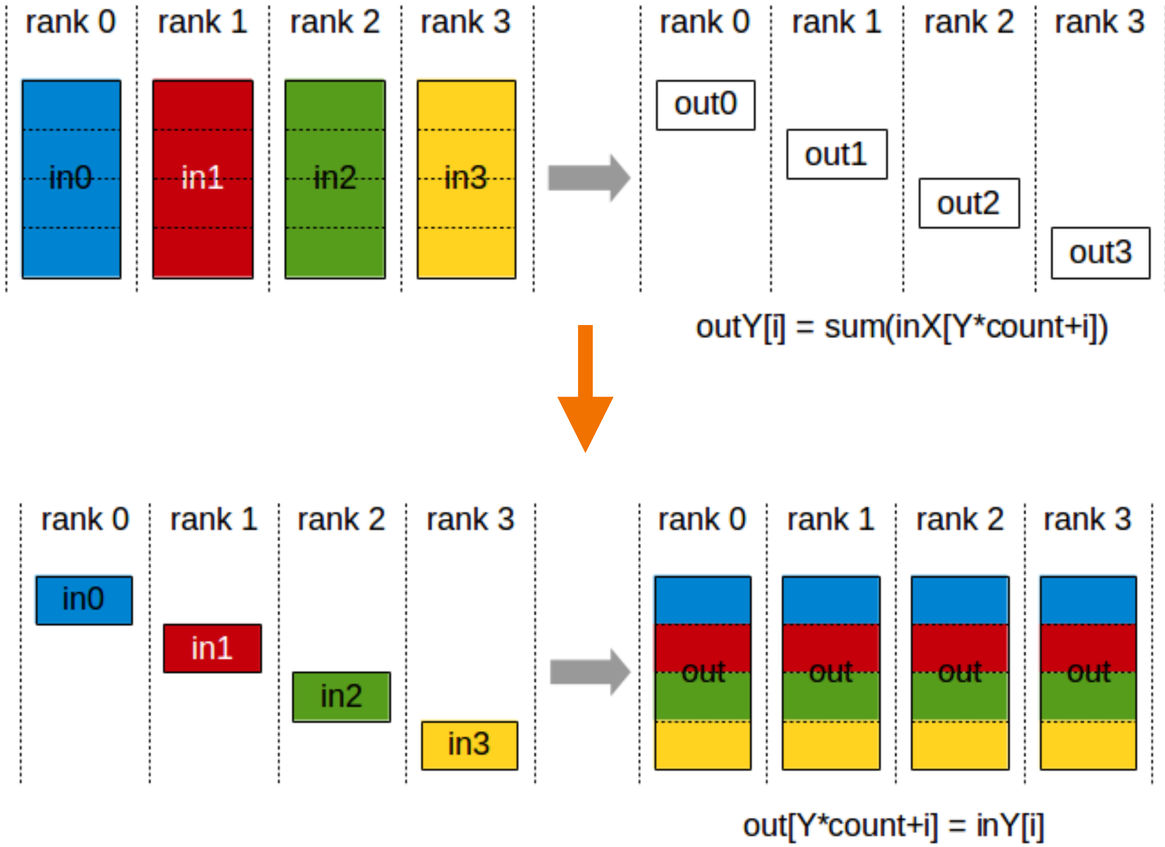


All-Reduce = Reduce-Scatter + All-Gather

All-reduce

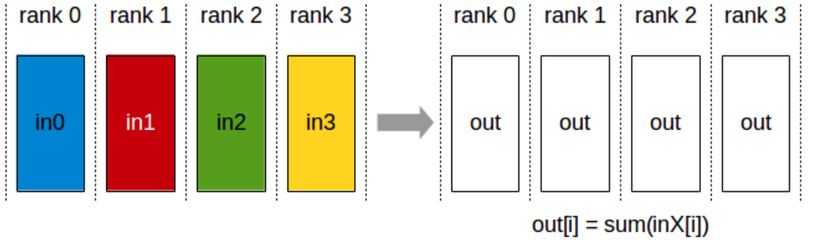


Reduce-scatter + All-gather

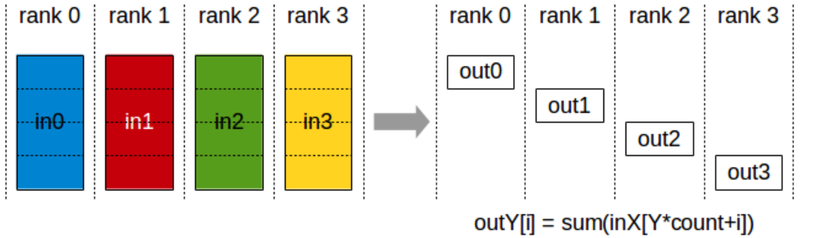


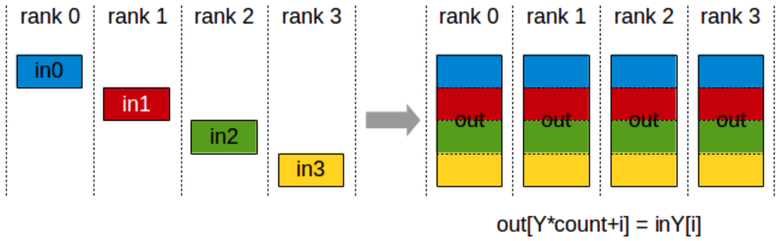
All-Reduce = Reduce-Scatter + All-Gather

- Communication cost? (D = # bytes, P = # GPUs)
- Naive all-reduce
  - Each GPU sends/receives D data to P-1 GPUs
  - Comm. cost: 2 x D x (P-1) ≈ 2DP



- Reduce-scatter + all-gather
  - Reduce-scatter: Each GPU sends D/P blocks to next GPU → (P-1) x D / P
  - All-gather: Same way; (P-1) x D / P
  - Comm. cost: 2 x D x (P-1) / P ≈ 2D
  - In the bandwidth-limited regime, this is the best you can do

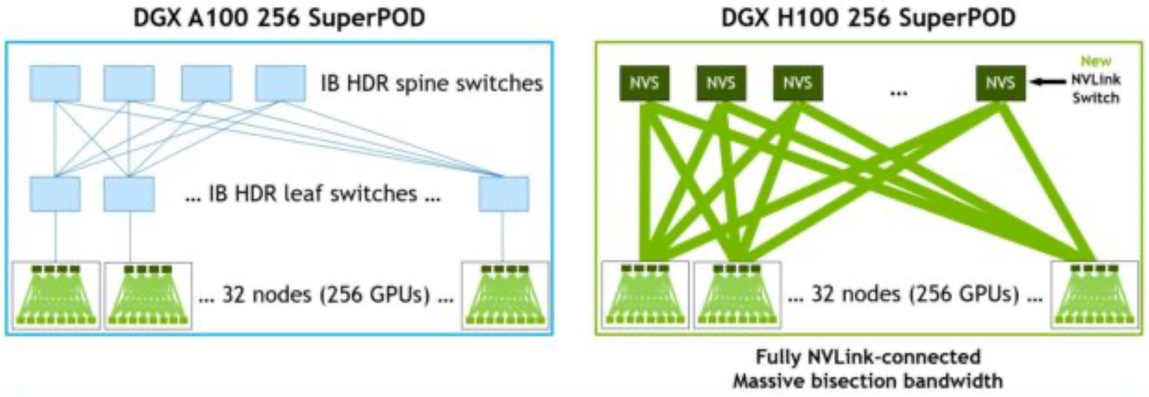




TPUs vs. GPUs

GPU networking

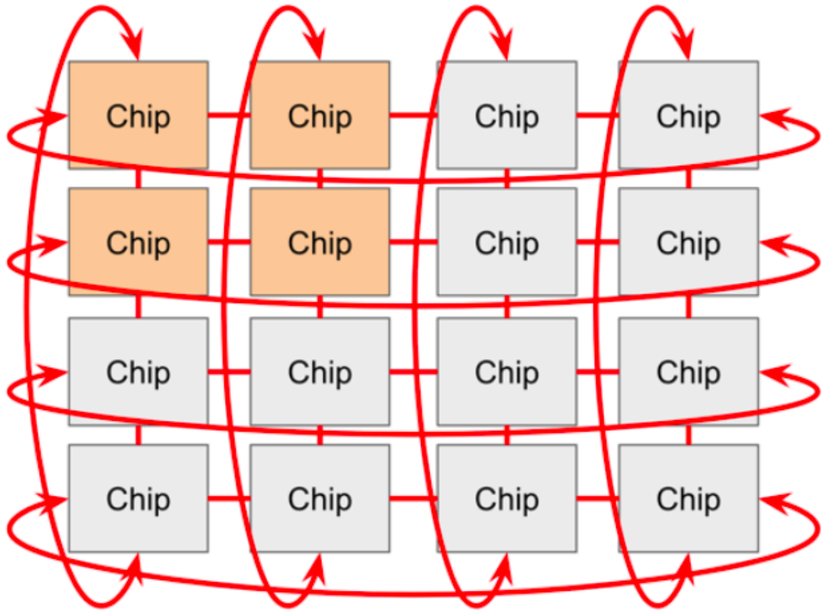
- All-to-all (up to 256)



	A100 SuperPod			H100 SuperPod			Speedup	
	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Dense PFLOP/s	Bisection [GB/s]	Reduce [GB/s]	Bisection	Reduce
1 DGX / 8 GPUs	2.5	2,400	150	16	3,600	450	**1.5x**	**3x**
32 DGXs / 256 GPUs	80	6,400	100	512	57,600	450	**9x**	**4.5x**

TPU networking

- Toroidal mesh

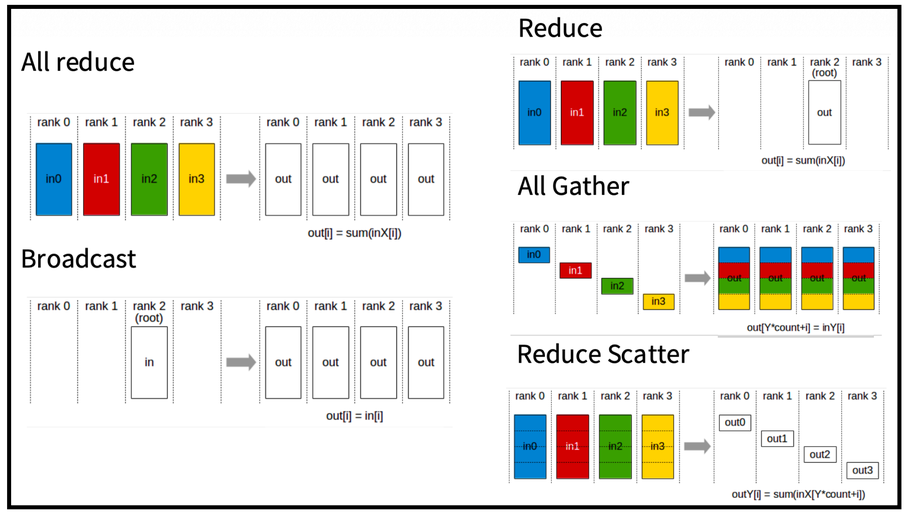


Recap

- Beyond a single GPU, we introduce new unit of compute, "datacenter"

- What we want from multi-machine scaling?
  - Linear memory scaling (max model params scale with # GPUs)
  - Linear compute scaling (model flops scale linearly with # GPUs)

- Basic collective communications



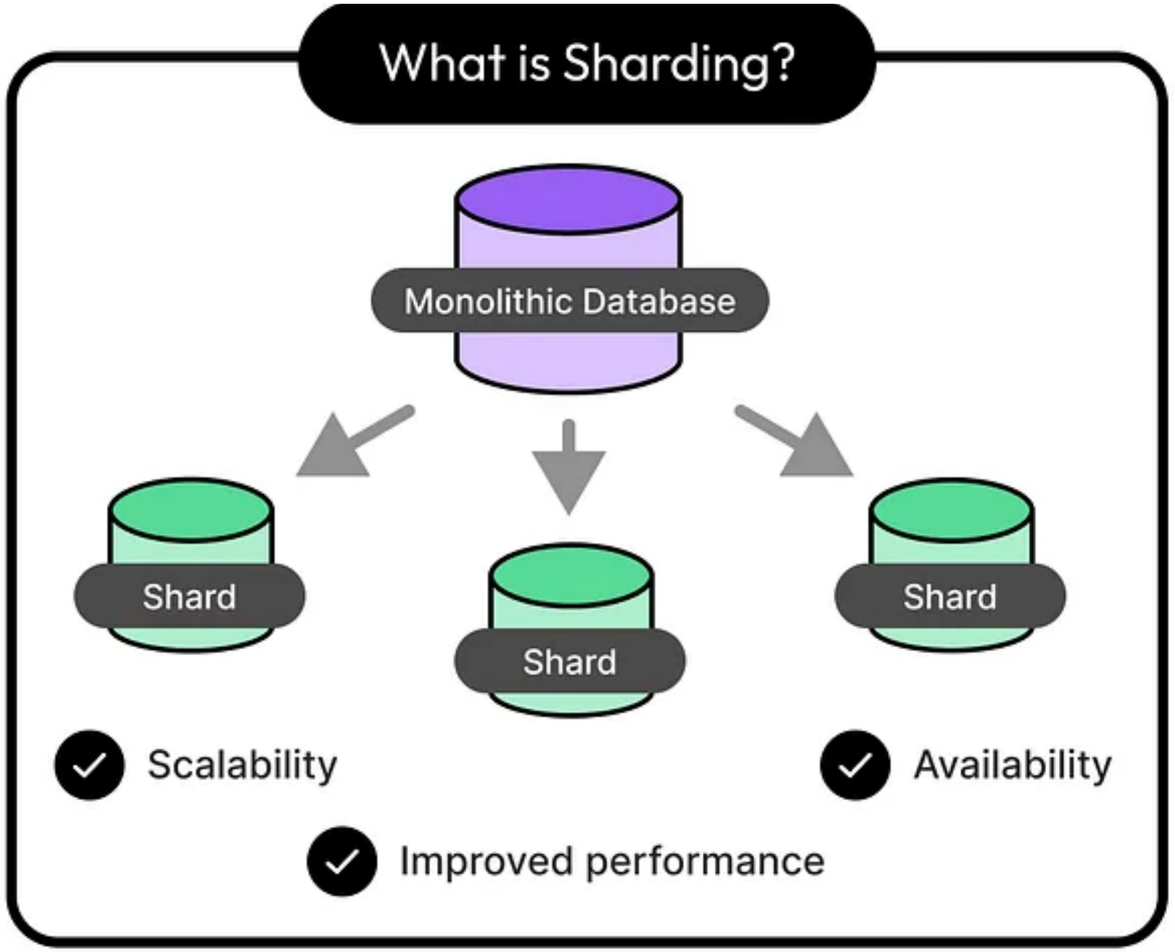
Standard LLM Parallelization

- How do we parallelize LLMs? Three important ideas
- Data parallelism
  - Naive data parallel
  - ZeRO-1, 2, 3
- Model parallelism
  - Pipeline parallel
  - Tensor parallel
- Activation parallelism
  - Sequence parallel

Data Parallelism

Image credit: https://blog.bytebytego.com/p/a-guide-to-database-sharding-key

Sharding

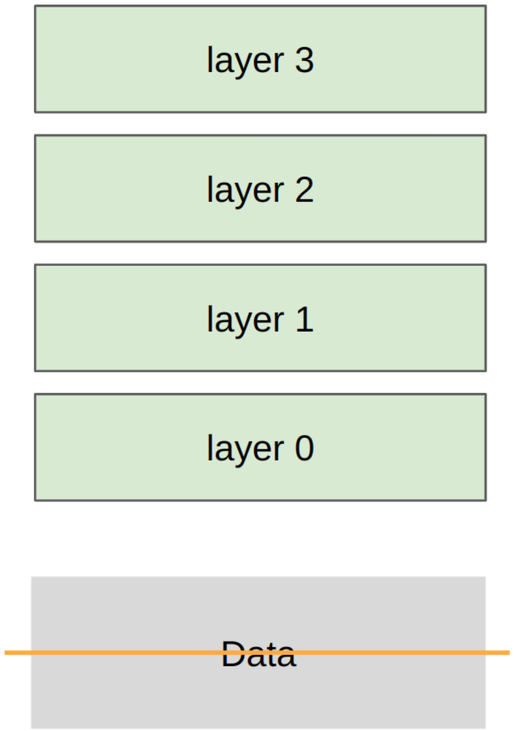


Naive Data Parallelism

- Sharding strategy: each rank gets a slice of the data
  - Split the elements of `B` sized batch across `M` GPUs
  - Exchange gradients to synchronize

- Abstracted workflow
  - Losses are different across ranks (on local data)
  - Gradients are all-reduced to be the same across ranks
  - Therefore, parameters remain the same across ranks

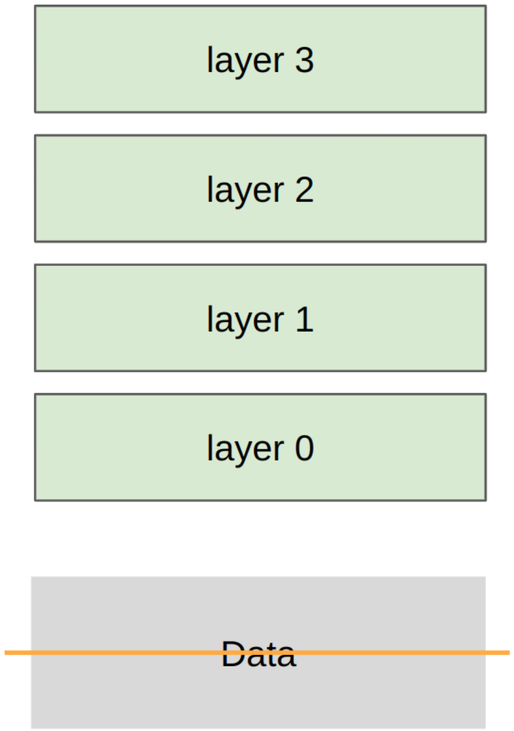
- This is the distributed data parallel (DDP) in PyTorch



Naive Data Parallelism

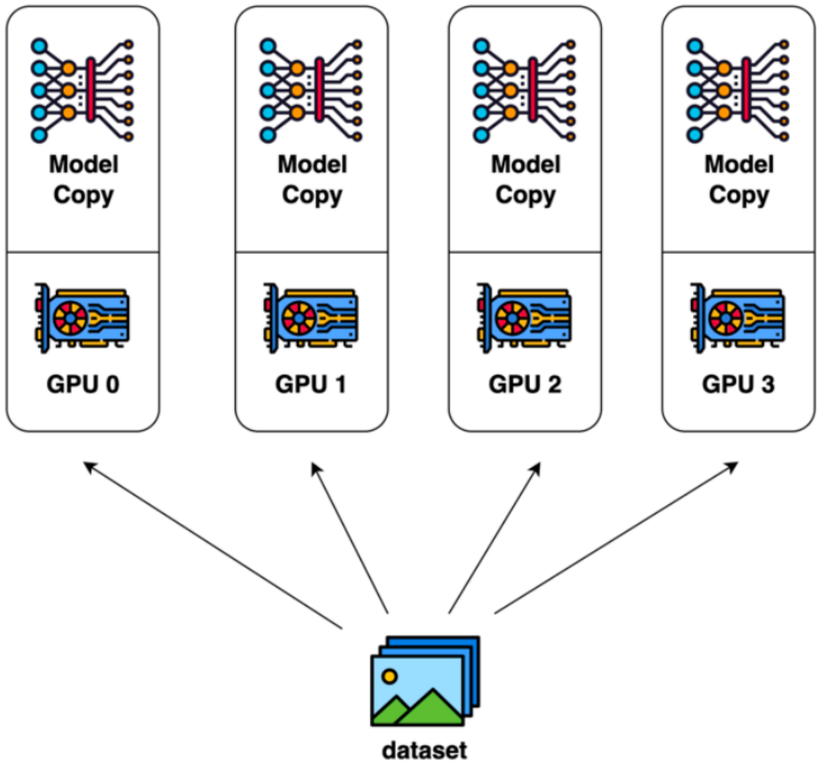
- Sharding strategy: each rank gets a slice of the data
  - Split the elements of `B` sized batch across `M` GPUs
  - Exchange gradients to synchronize

- How does this do?
  - Compute scaling: each GPU gets `B/M` examples
  - Communication overhead: transmits 2x # params every batch
    - OK if batches are big enough
  - Memory scaling: none. Every GPU needs # params at least



Naive Data Parallelism

- Memory seems like it'd be a problem
  - We copy the model parameters to each GPU



Naive Data Parallelism

- What is wrong with naive data parallelism?
  - Our memory situation is actually terrible

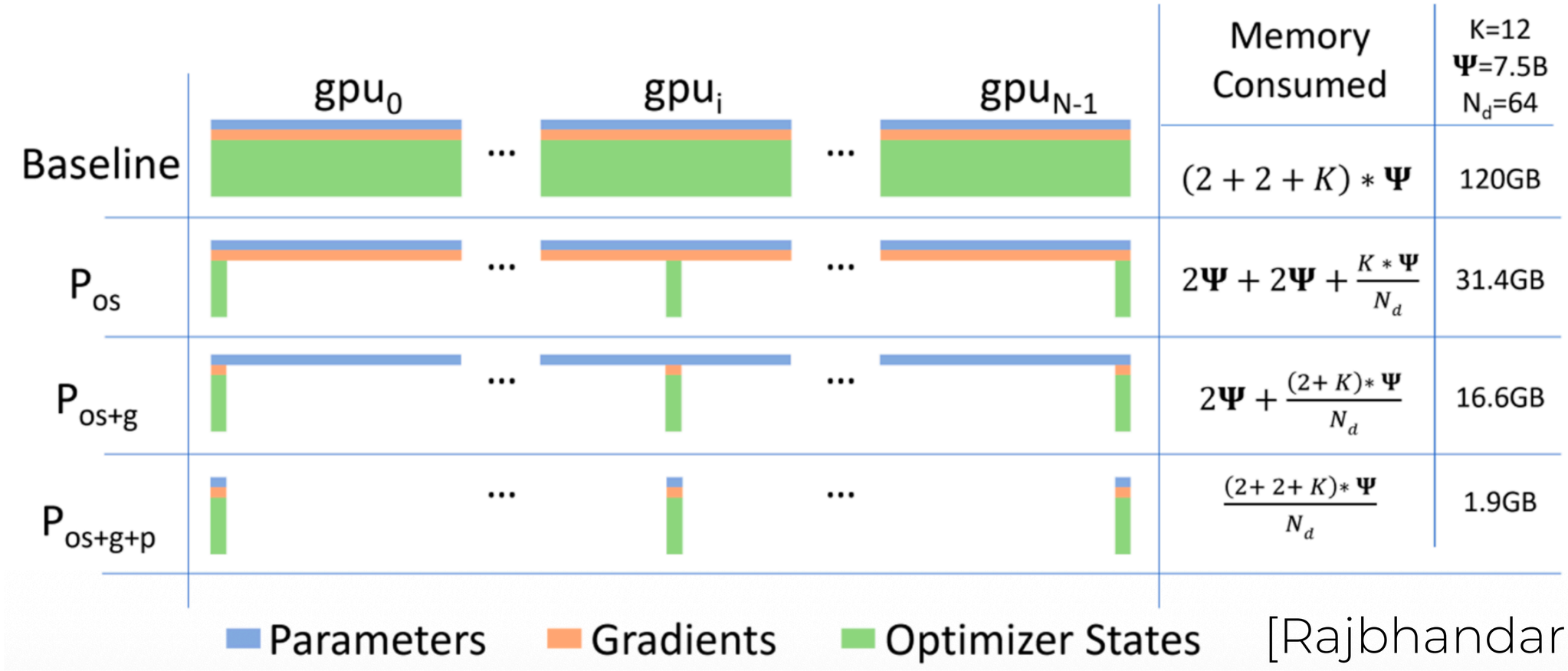
- Assume we use fp16 mixed precision
  - We need 5 copies of weights and 16 bytes per param!
    - 2 bytes for FP/BF16 model parameters
    - 2 bytes for FP/BF16 gradients

optimizer state

- 4 bytes for FP32 master weights (thing you accumulate into in SGD)
- 4 (or 2) bytes for FP32/BF16 Adam first moment estimates
- 4 (or 2) bytes for FP32/BF16 Adam second moment estimates

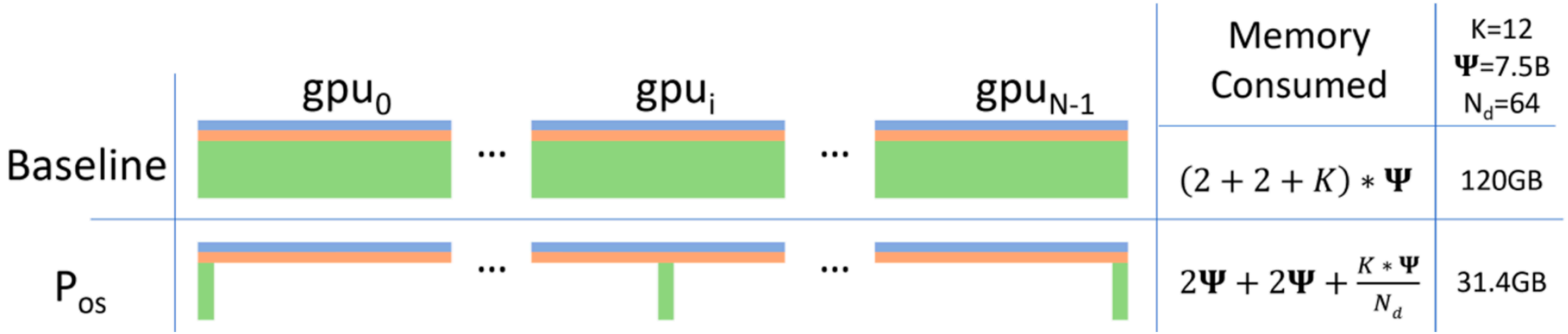
ZeRO: Solving the Memory Overhead

- Let's split up the expensive parts and use the reduce-scatter equivalence



[Rajbhandari+ 2019]

ZeRO-1: Sharding Optimizer State [Rajbhandari+ 2019]



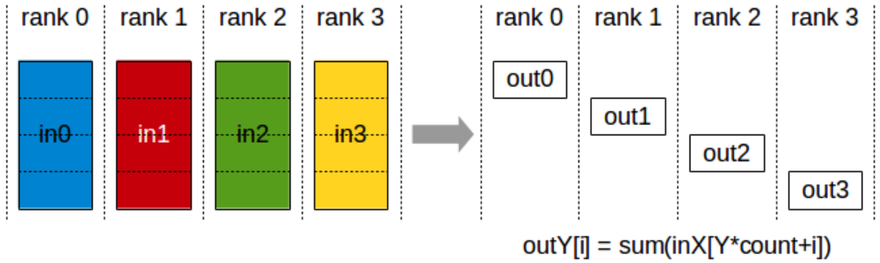
- High-level idea
  - Split up the optimizer state (first + second moments) across GPUs
  - Everyone has the parameters + gradients

  - Each worker is responsible for updating a subset of params (corresponding to their slice)

ZeRO-1: How it Works? [Rajbhandari+ 2019]

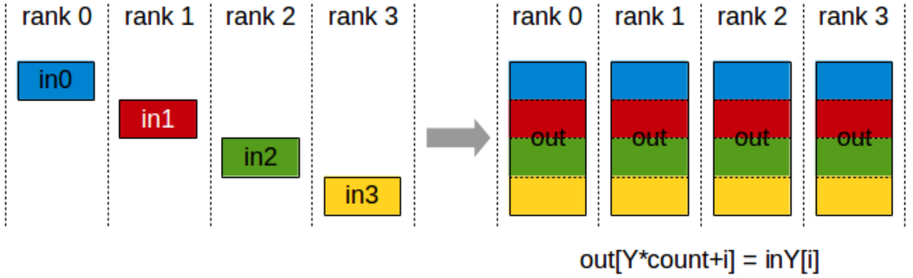
- Step 1. Everyone computes a full gradient on their subset of the batch

- Step 2. Reduce-scatter the gradients
  - This incurs # params communication cost
  - Recap: (P-1) x D / P ≈ D comm. cost



- Step 3. Each GPUs updates their params using their gradient + state

- Step 4. All-gather the parameters
  - This incur # params communication cost
  - Recap: (P-1) x D / P ≈ D comm. cost



ZeRO-1 vs. Naive DP

	Naive DP	ZeRO-1
Communication primitive	One all-reduce (grads)	reduce-scatter (send grads) + all-gather (collect params)
Communication cost	2 x # params	2 x # params
Memory	(4+K) x #params	(4+K/N_{gpu}) x #params

- ZeRO-1 is a free lunch (at least in the bandwidth-limited regime)

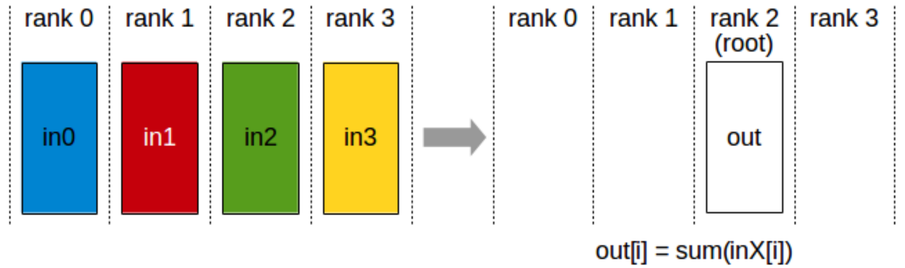
ZeRO-2: Sharding Gradient [Rajbhandari+ 2019]

- High-level idea
  - Also keep the gradients sharded across GPUs
  - Use the same (rough) tricks as ZeRO-1

- Complexity
  - We can never instantiate a full gradient vector, but each worker must compute a full gradient (since we're data parallel)

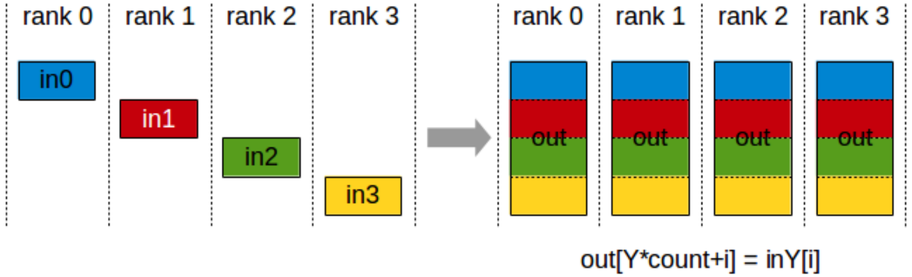
ZeRO-2: How it Works? [Rajbhandari+ 2019]

- Step 1. Every GPU incrementally backwards on computation graph
  - Step 1a. After computing a layer's gradients, immediately reduce to send this to the right worker
  - Step 1b. Once gradients are not needed in the backward graph, immediately free it
  - This incurs # params communication cost

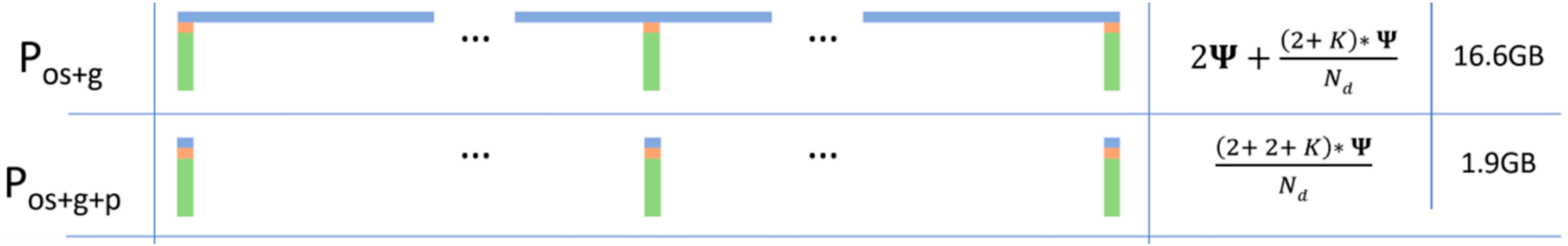


- Step 2. Each GPU updates their params using their gradient + state

- Step 4. All-gather the parameters
  - This incur # params communication cost



ZeRO-3 (FSDP): Sharding All [Rajbhandari+ 2019]



- High-level idea
  - Shard everything, including parameters!
  - Use the same *incremental communication / computation* ideas
  - Send and request parameters on demand while stepping through the computational graph; materialize and free just-in-time

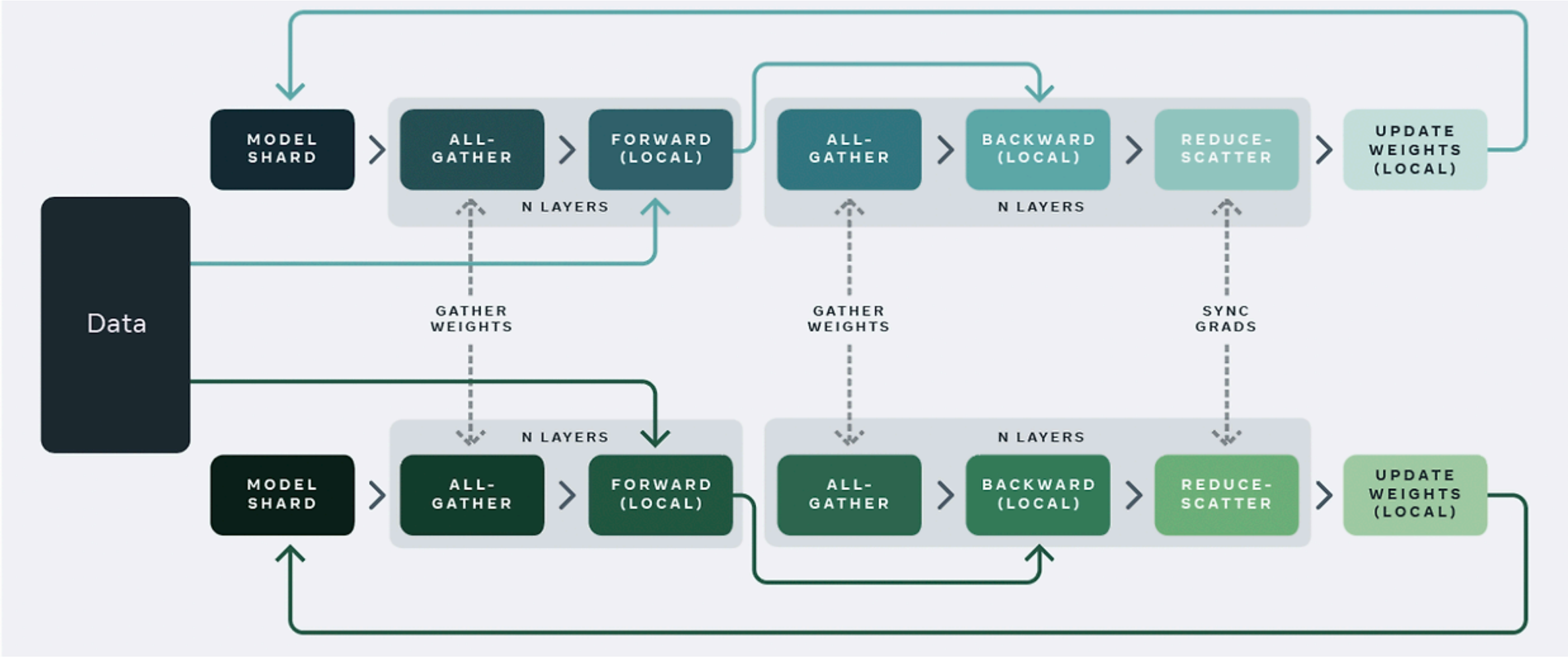
- This is the fully-sharded data parallel (FSDP) in PyTorch

ZeRO-3: How it Works? [Rajbhandari+ 2019]

- > Forward pass
- Step 1. Before compute layer i's forward pass, every GPU all-gather params to materialize the full params of layer i
- Step 2. Once layer i's is done, immediately free params of layer i

- > Backward pass
- Step 3. To compute layer i's backward pass, every GPUs all-gather params to materialize the full params of layer i
- Step 4. Each GPU calculates grads and immediately reduce-scatter
- Step 5. Each GPU updates their params using their gradient + state
- Step 6. Once params update is done, immediately free grads of layer i

ZeRO-3: How it Works? [Rajbhandari+ 2019]



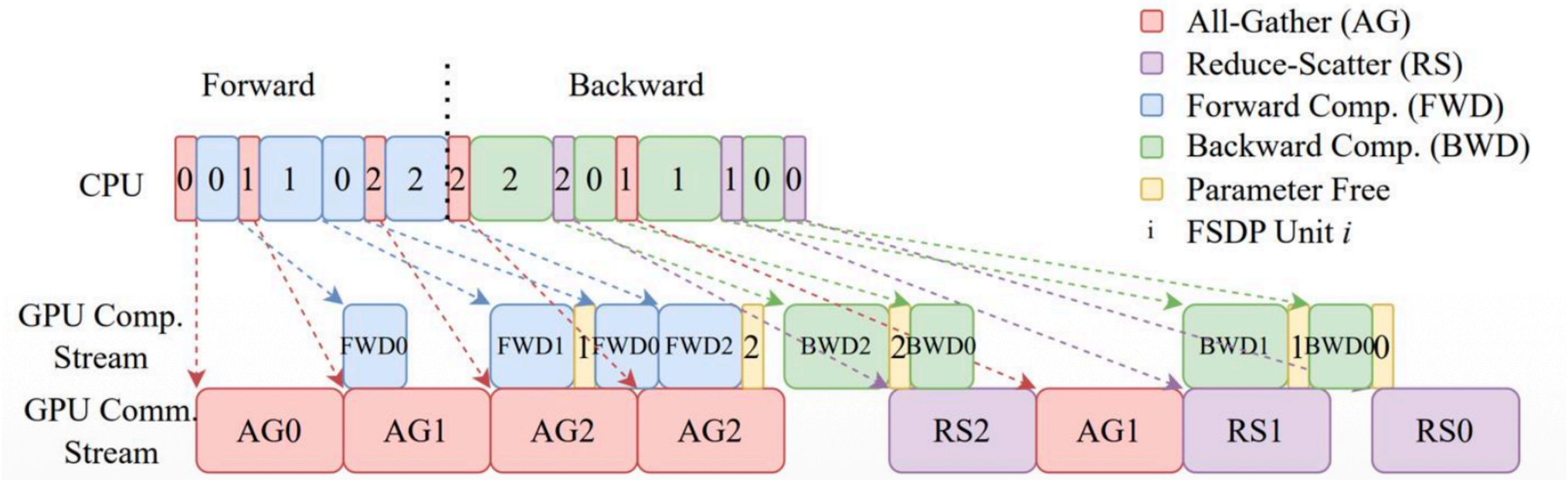
ZeRO-1, 2, 3 Communication Cost

- DDP: 2 x # params
- Zero-1: 2 x # params (it's entirely free!)
- Zero-2: 2 x # params (it's almost free!; when we ignore overhead)
- Zero-3: 3 x # params (1.5x cost, but not bad considering memory)
  - We have 1 all-gather (# params) and 2 reduce-scatter (# params)

ZeRO-3: Communication Overlapping

- All-gathers happen all at once while forward happens
  - It can mask out (overlap) the communication cost

- Example: $(W_1W_0 + W_2W_0)x = y$



ZeRO-3 in Practice

- Pure bf16 training (bf16 for optimizer states as well)
  - But keep master weight to fp32

- On a 8X A100 80G

	Max size (params)	Bytes per Params
DDP	6.66B	12
Zero-1	16B	5
Zero-2	24.62B	2 + (10/8)
Zero-3 (FSDP)	53.33B	12 / 8

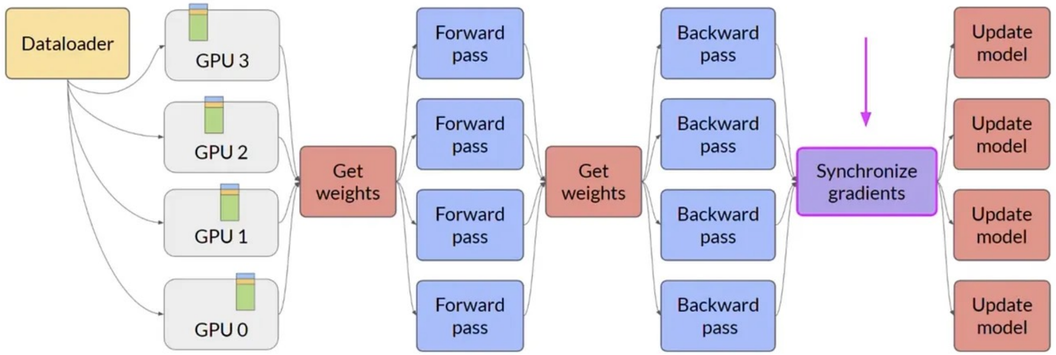
Recap

- Data parallel: each rank gets a slice of the data

- Zero-1: sharding optimizer states
  - Reduce-scatter grads → calc local params → all-gather params

- Zero-2: + sharding grads
  - Immediately reduce grads → free grads (if possible, immediately) → calc local params → all-gather params

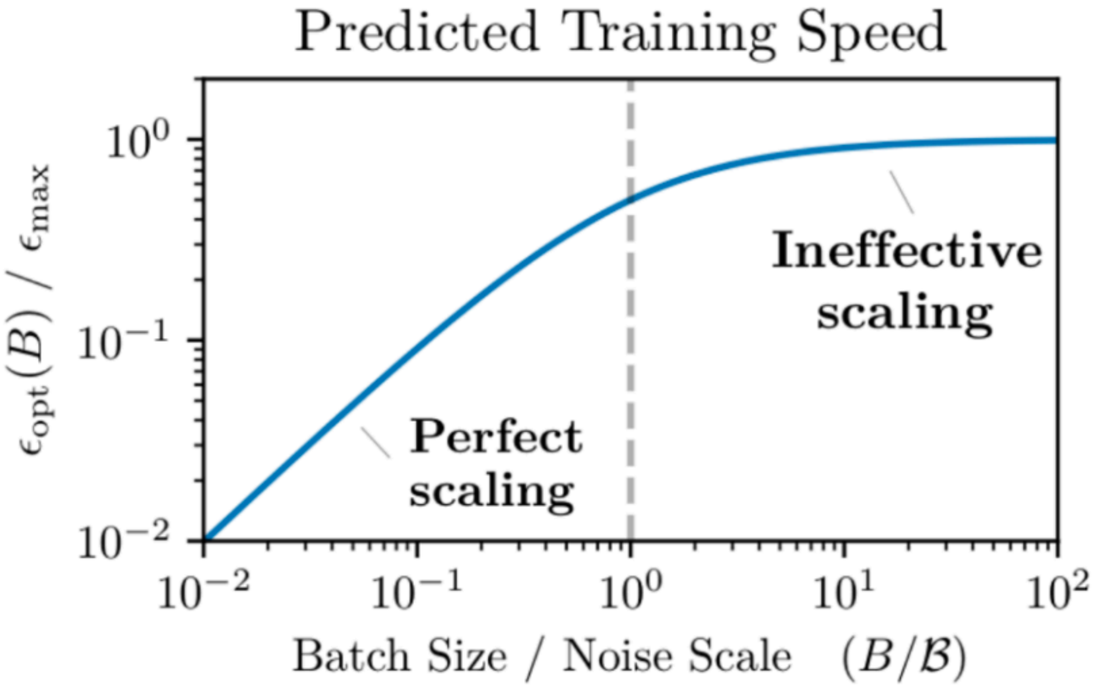
- Zero-3: + sharding params

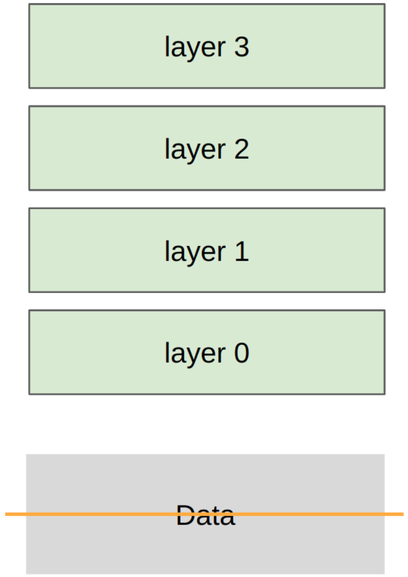


Model Parallelism

Issues of Data Parallel

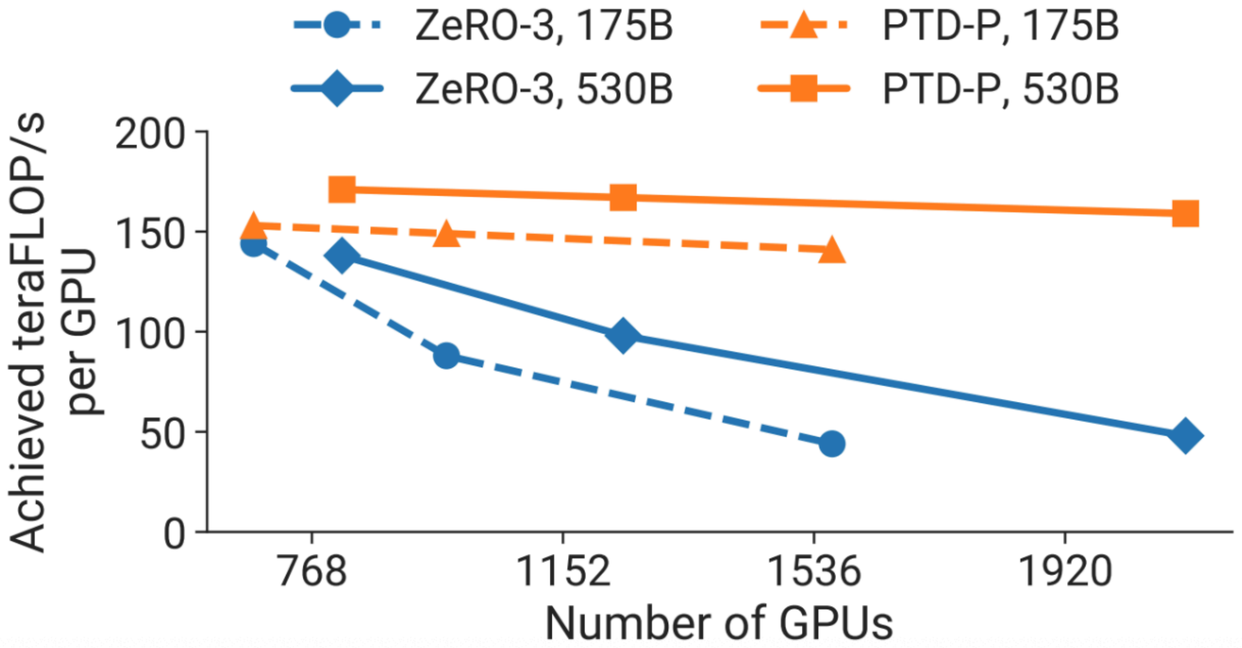
- With data parallel, batch size should be really big
  - To increase # GPUs, batch size is proportionally increased
  - And there's diminishing returns to batch sizes





Issues of Data Parallel

- ZeRO-1 and 2 don't let you scale memory
- ZeRO-3 is nice, but can be slow and does not reduce activation memory



[Narayanan+ 2021]

- Any other options we have?

Model Parallelism

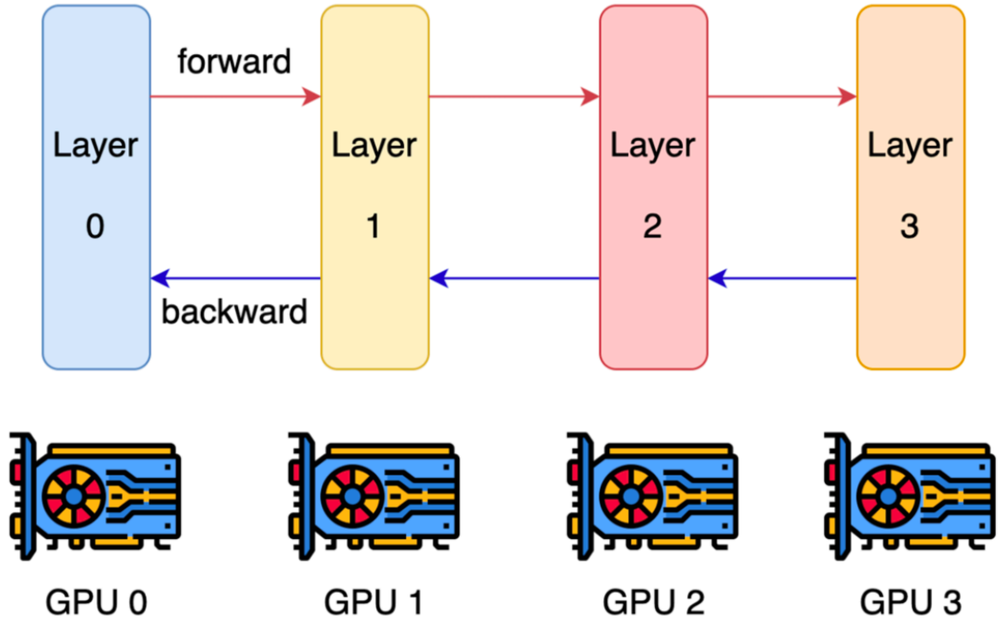
- Scaling up in memory without changing batch size

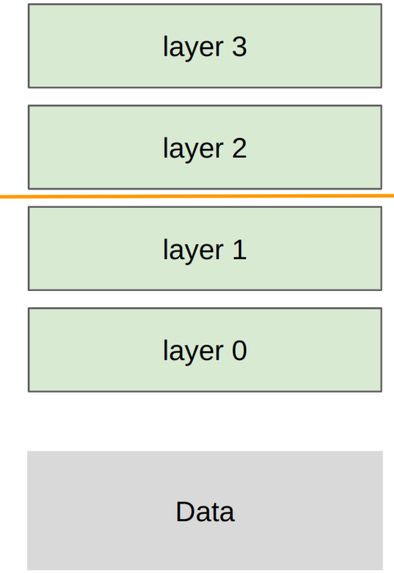
- What model parallelism is
  - It splits up the parameters across GPUs (like ZeRO-3)
  - But communicate activations (while ZeRO-3 sends params)

- Two types of model parallelism
  - Pipeline parallel (parallelize along depth)
  - Tensor parallel (parallelize along width)

Layer-Wise Parallel

- Very straightforward model parallel

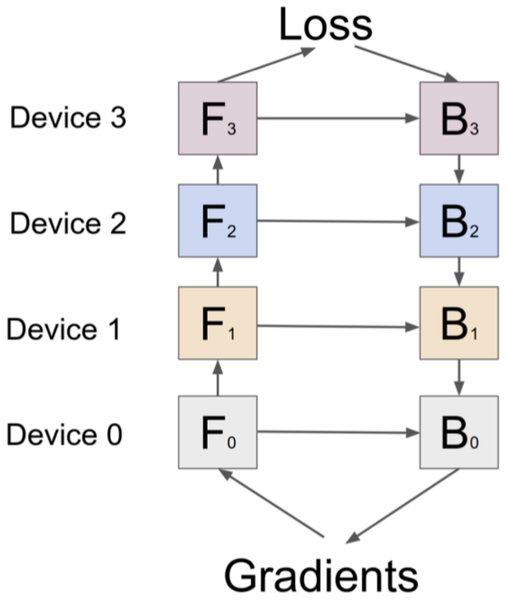


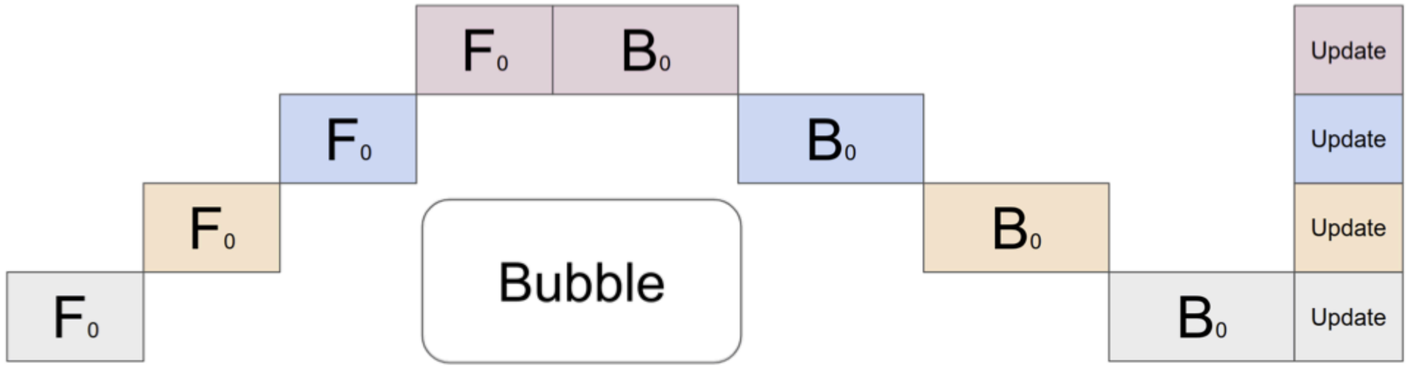


- Layer-wise parallel cuts up layers, assigns some subsets to GPUs
- Activations and partial grads are passed back and forth

Layer-Wise Parallel

- Looks good, but utilization of layer-wise parallelism is terrible…



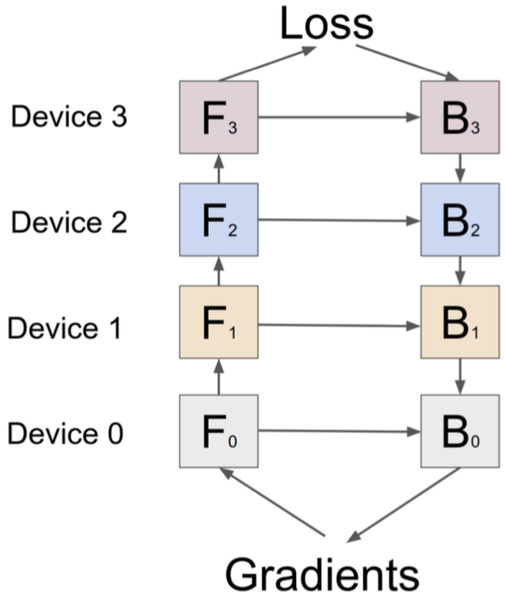


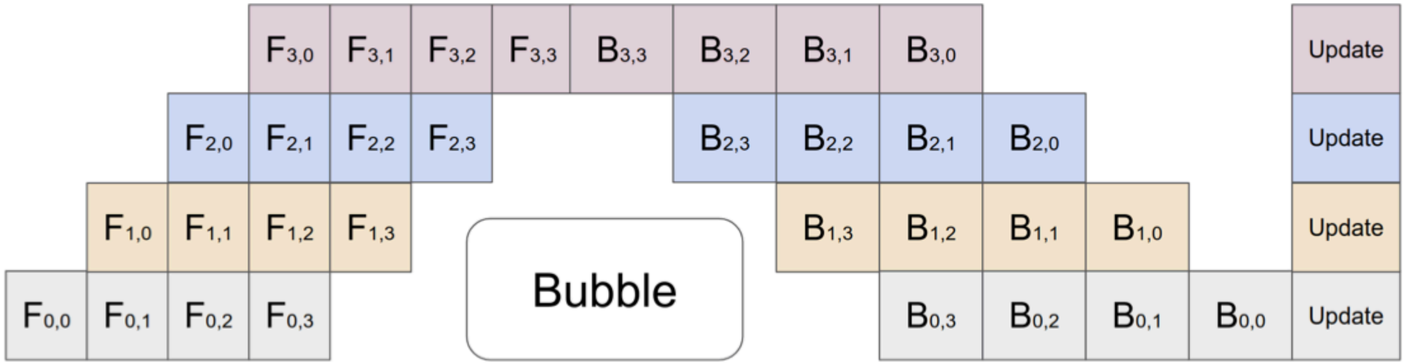
With N GPUs, each GPU is active 1/N of the time

- Each GPU is idling most of the time, waiting for the backward pass to propagate back

GPipe: Pipeline Parallel [Huang+ 2018]

- GPipe splits batch to micro-batch and process micro-batches
  - Send off the first micro-batch and start computing the second





- The ratio of bubble time is $\frac{P-1}{M}$ so we need a big batch size

P= # GPUs, M = # micro-batch

Why Pipeline Parallel?

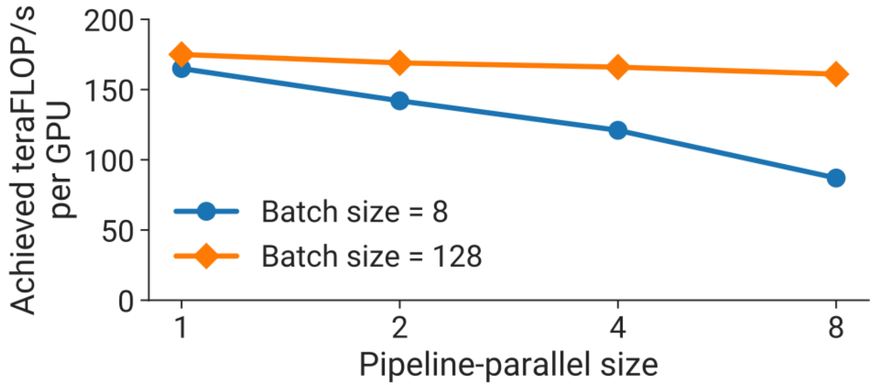
- Pipelines seem not good (e.g. bubble time, etc)

- Why do we do it?
  - Pipelines save memory (compared to DDP)
  - Pipelines can have good comm. properties (compared to FDSP)
    - It depends on activations (**B x S x H**) and is point-to-point comm.

- Generally, we use pipelines on slower network links (e.g. inter-node) as a way to get better memory-wise scaling

Issues of GPipe

- To reduce bubble, 1) increase batch size and **M** 2) increase **M** only
  - Very large batch size can cause diminishing returns
  - Small batch size rapidly degrades GPU performance
    - Micro-batch goes too small, so compute-bound → memory-bound

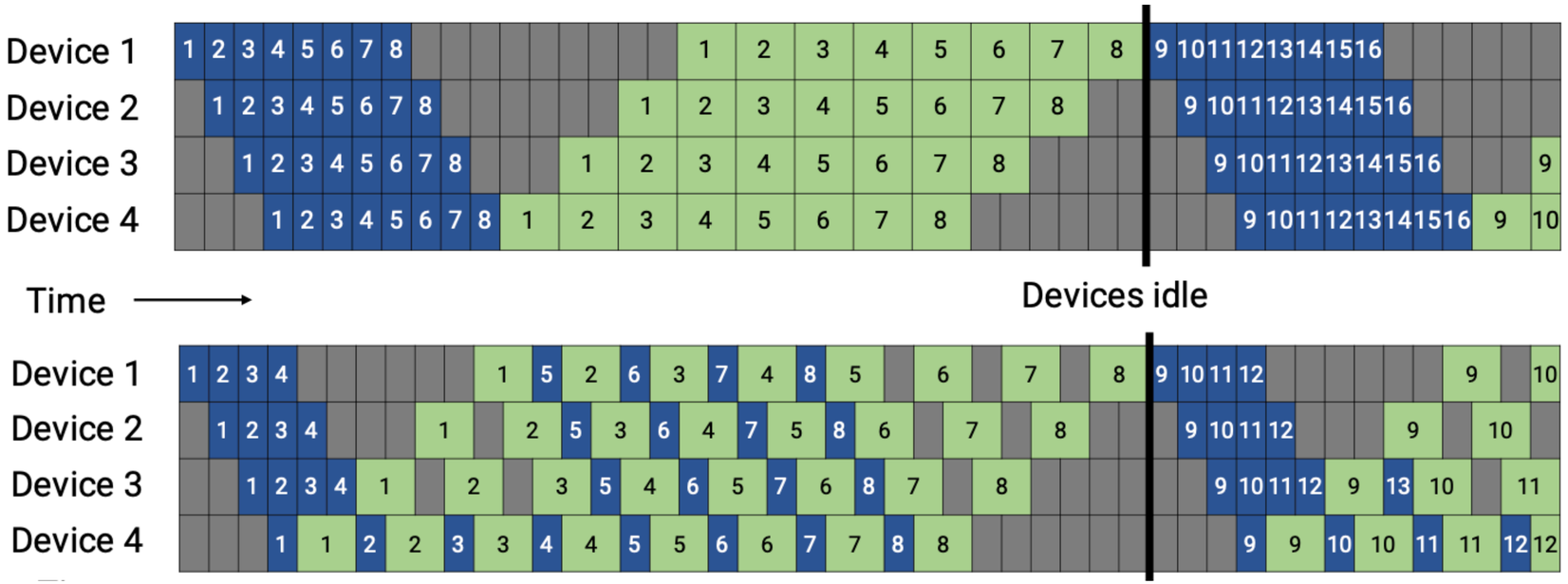


[Narayanan+ 2021]

- Another issue: explosion of activation memory
  - First GPU should store all activations until **M** micro-batches are done
  - Because of the memory limit, we cannot set very high **M**

Solution: 1F1B (PipeDream) [Harlap+ 2018]

- One forward, one backward scheduling
  - Step 1. Forward until each GPU reaches the end of the pipeline
  - Step 2. Do 1F1B: 1F for new micro-batch, 1B for oldest micro-batch
  - Step 3. Immediately free if backward pass is done

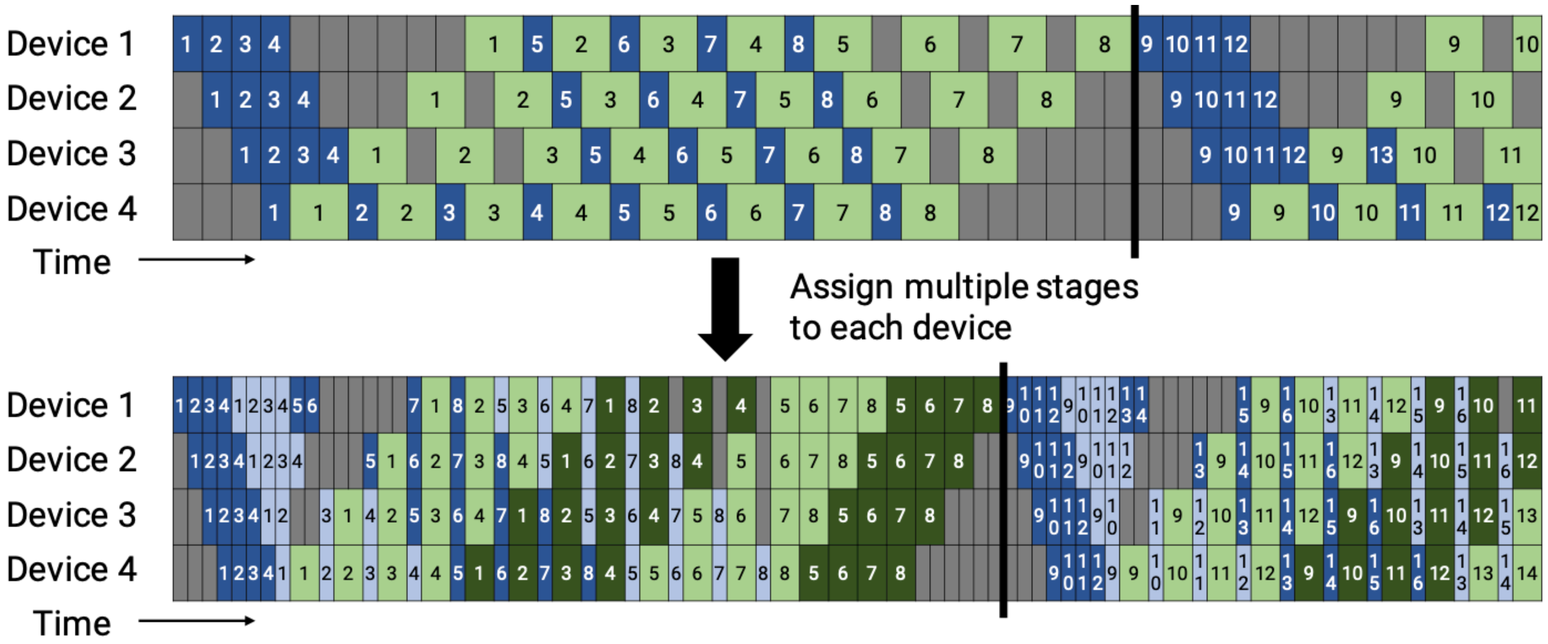


GPipe

PipeDream (1F1B)

Solution: Interleaved 1F1B [Narayanan+ 2021]

- 1F1B addresses memory issue, but still has bubble (at pipeline flush)
- In Megatron-LM v2, they propose interleaved 1F1B
  - Each GPU is assigned fragmented layers (chunk)
    - e.g. GPU 0: layer 0, 1, 4, 5 / GPU 1: layer 2, 3, 6, 7



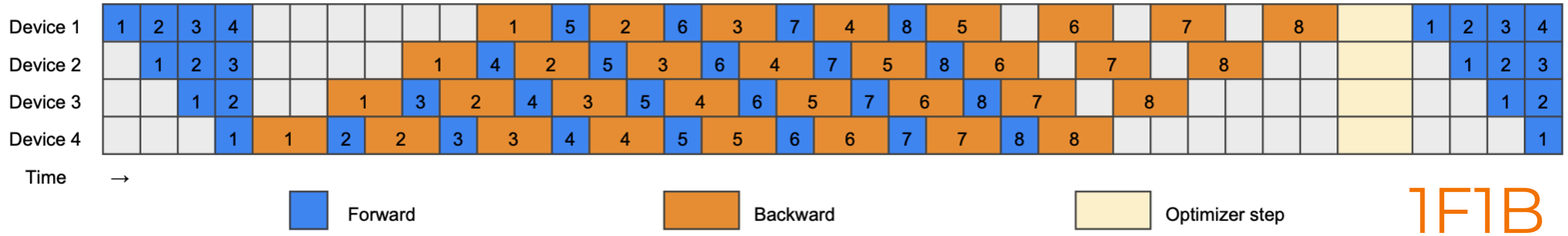
$$Bubble = \frac{P-1}{M}$$

$$Bubble = \frac{P-1}{VM}$$

V = # chunks

Go Beyond? Zero-Bubble PP [Qi+ 2024]

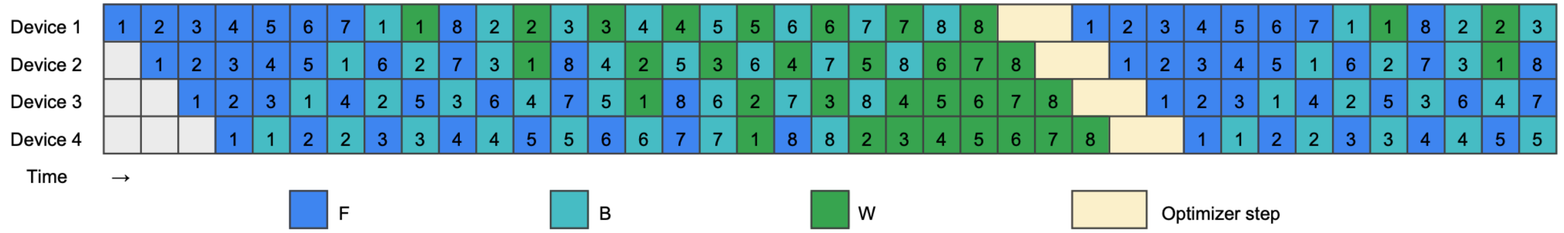
- Actually, backward pass is grad prop. + weight update
  - Weight update doesn't depend on the pipeline
  - So we can do weight update at any time!



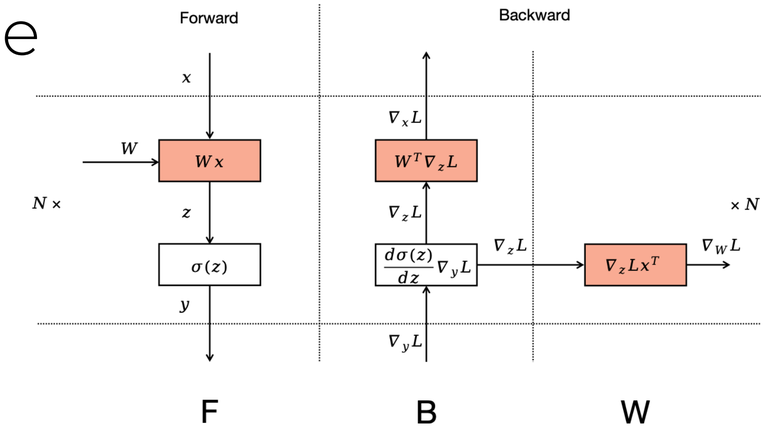
1F1B

ZB-H1

Handcrafted rules

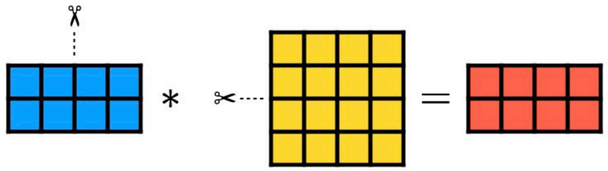


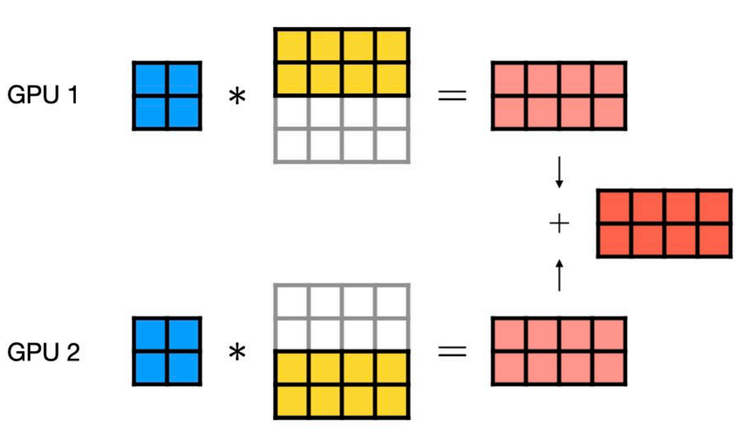
ZB-H2

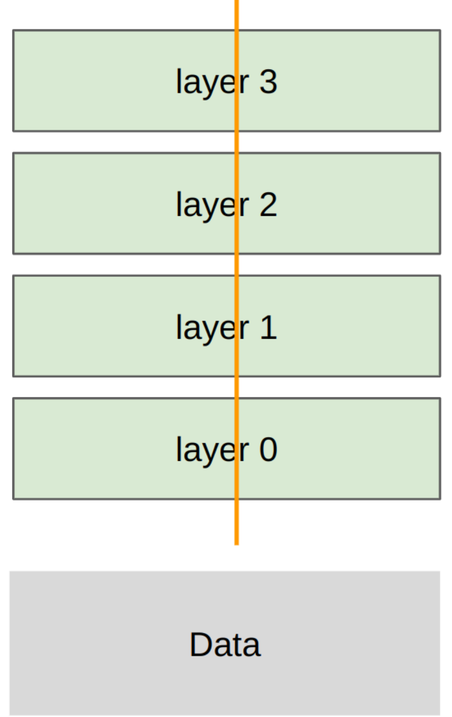


Tensor Parallel

- We can do model parallel along the width axes
  - We can think of PP as cutting up along depth



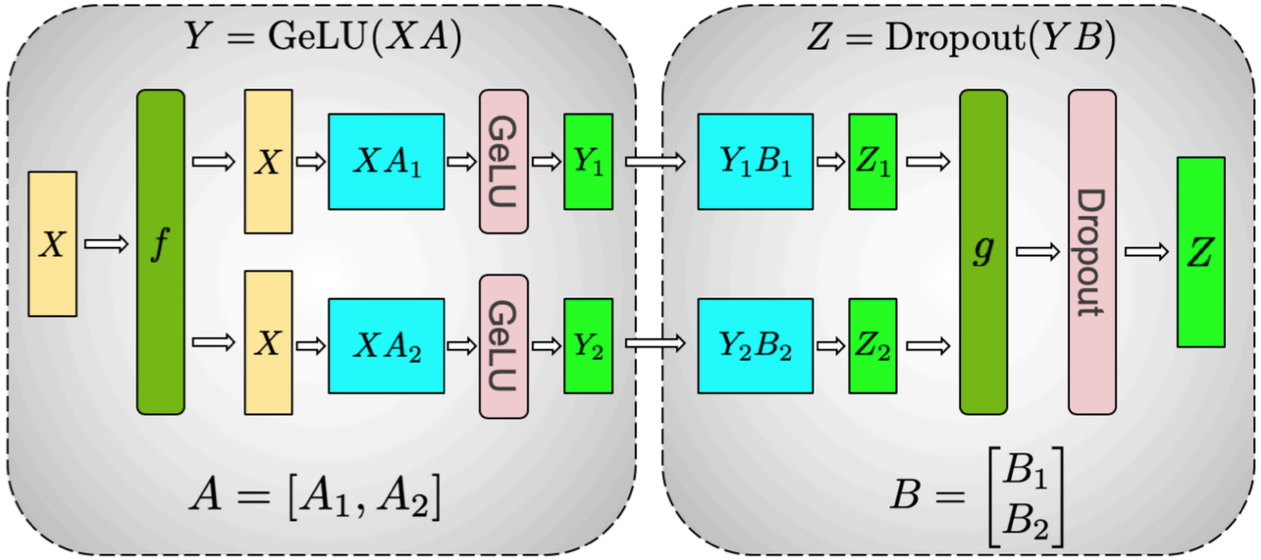


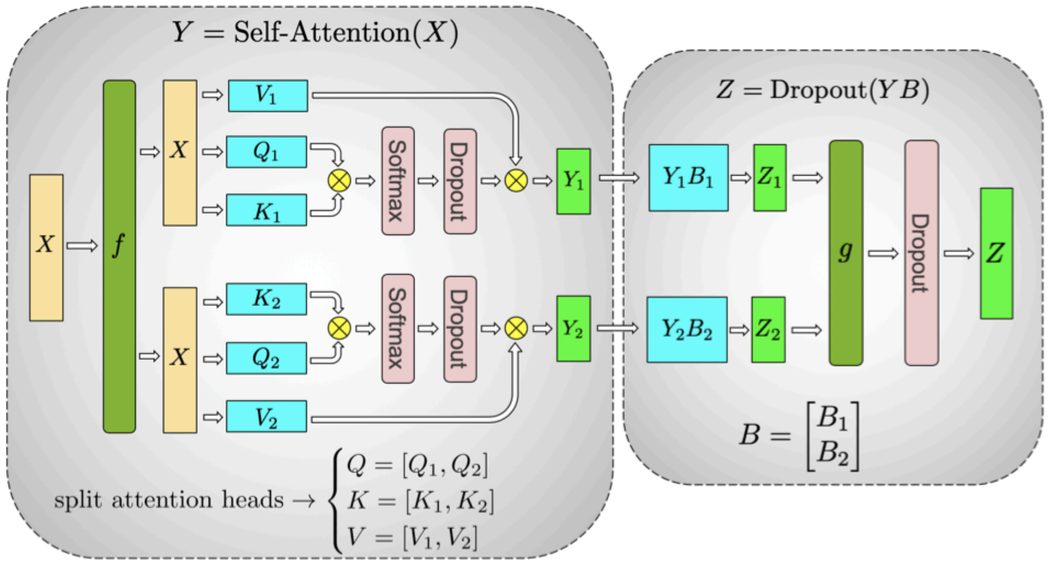


- Decompose a matmul: matmul submatrices, add partial sums

Tensor Parallel [Shoeybi+ 2019]

- Now GPUs have submatrices
- Assign columns (A1, A2) and rows (B1, B2) to separate GPUs
  - In the forward pass, f is the identity, and g is an all-reduce
  - In the backward pass, f is an all-reduce, g is the identity





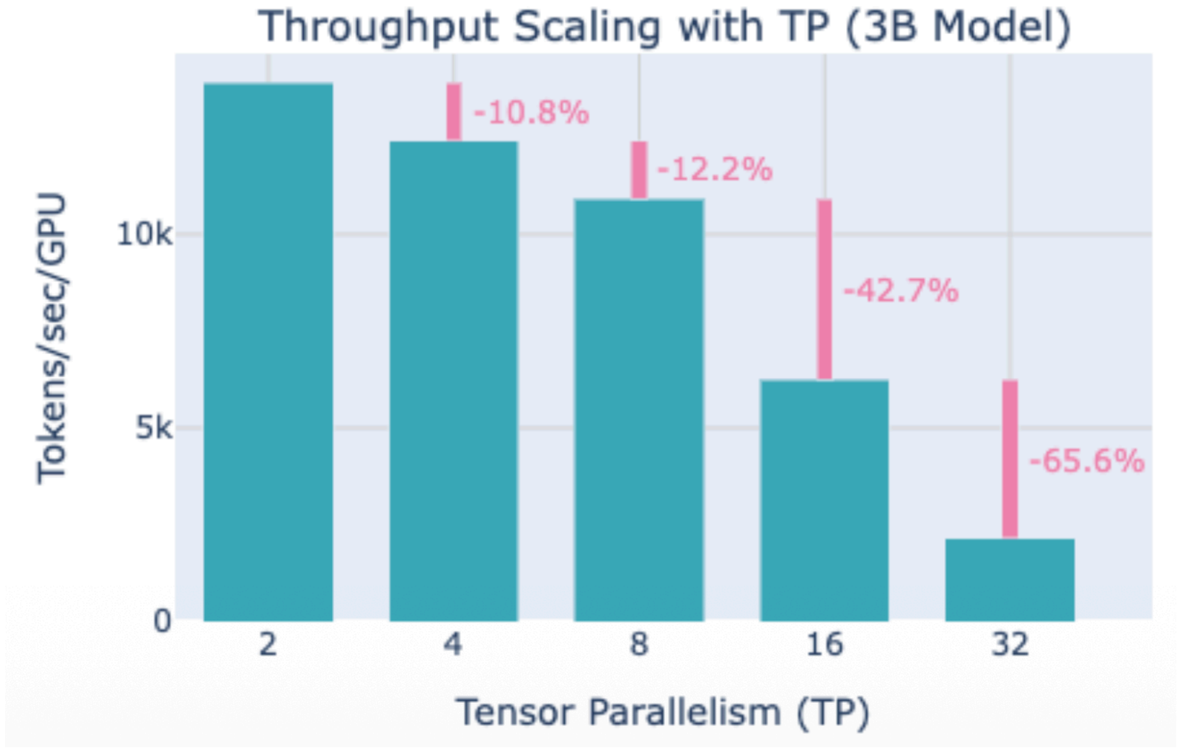
Tensor Parallel [Shoeybi+ 2019]

- TP vs. PP
  - 👍 No bubble; If network is fast enough, there's no waiting for others
  - 👍 Low complexity: just warp models without major infra changes
  - 👍 No need of large batch size

  - 👎 Much larger communication than PP
    - PP: **B x S x H** (and point-to-point comm. per micro-batch)
    - TP: **8 x B x S x H x (P-1) / P** (per layer and all-reduce comm.)
      - Why 8? 2 all-reduce (forward/backward) of FFN, Attention

Tensor Parallel [Shoeybi+ 2019]

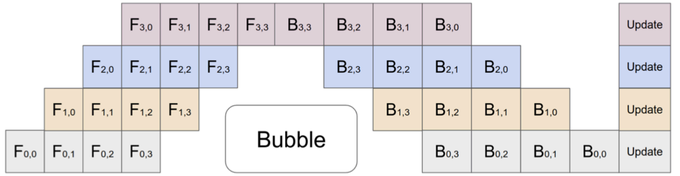
- When do we tensor parallel?
  - Use TP when we have low-latency, high-bandwidth interconnects
  - On GPUs, within a node (up to 8 GPUs) due to high speed interconnects



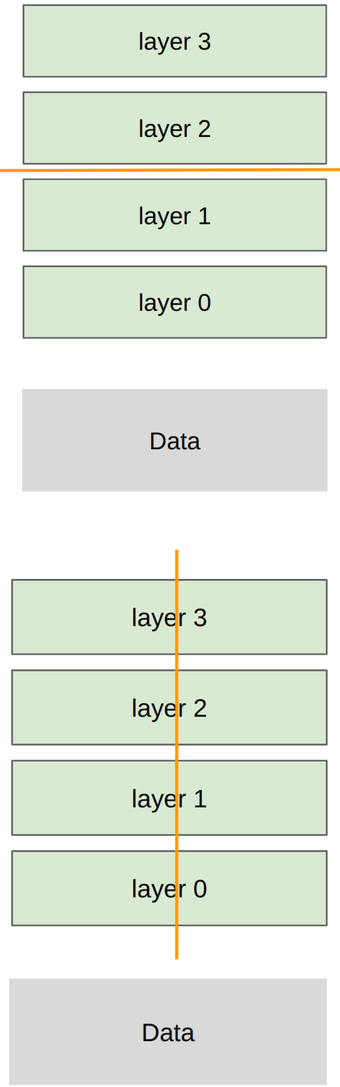
Recap

- Two types of model parallel:

- Pipeline parallel
  - Cut layers and each GPUs process their layers
  - PP split batch into micro-batch
  - Key challenge: how to reduce bubble?



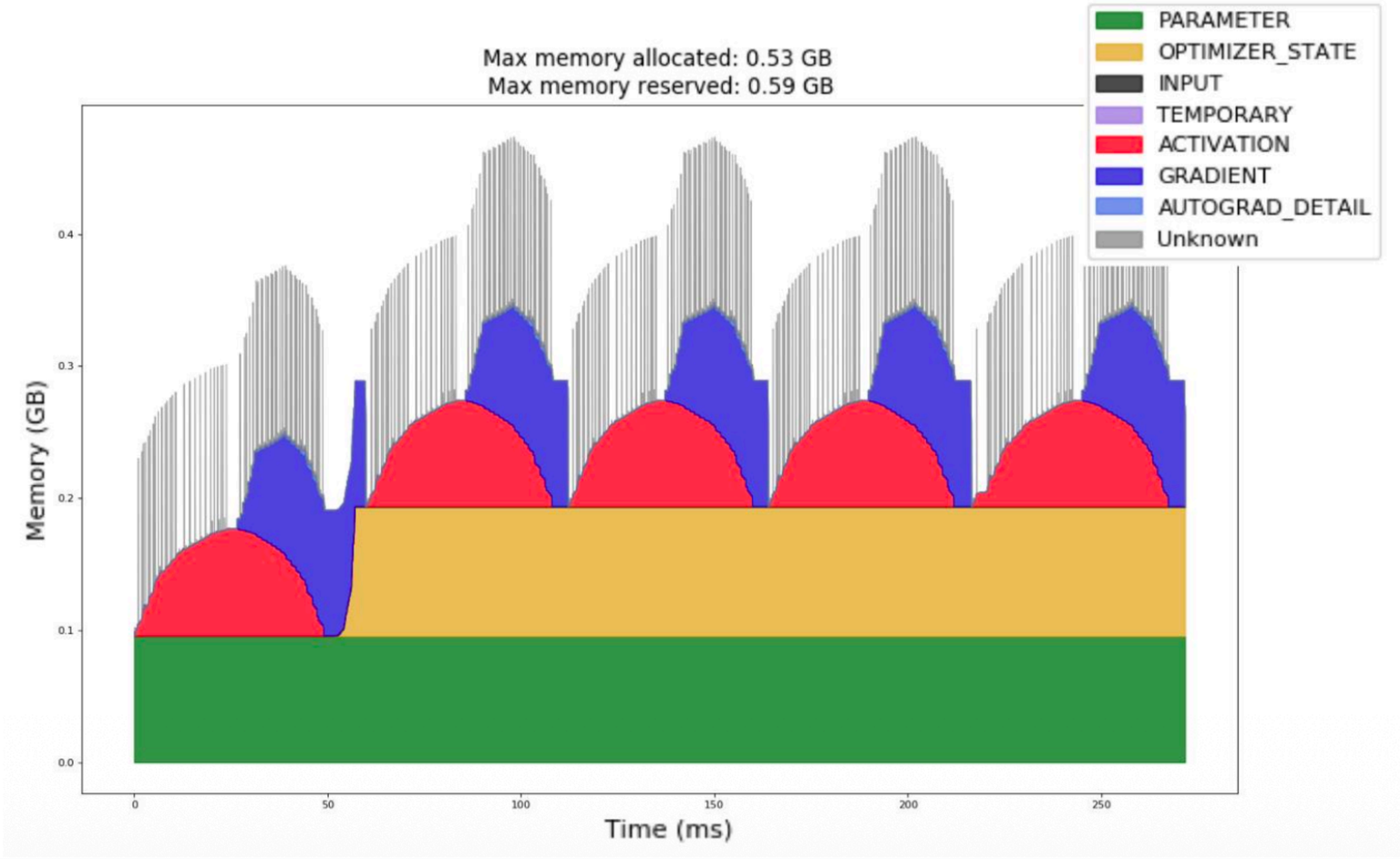
- Tensor parallel
  - Parallelize model width, not depth
  - Cut giant matrix into submatrices and distribute to GPUs
  - No bubble, but very high comm. cost



Activation Parallelism

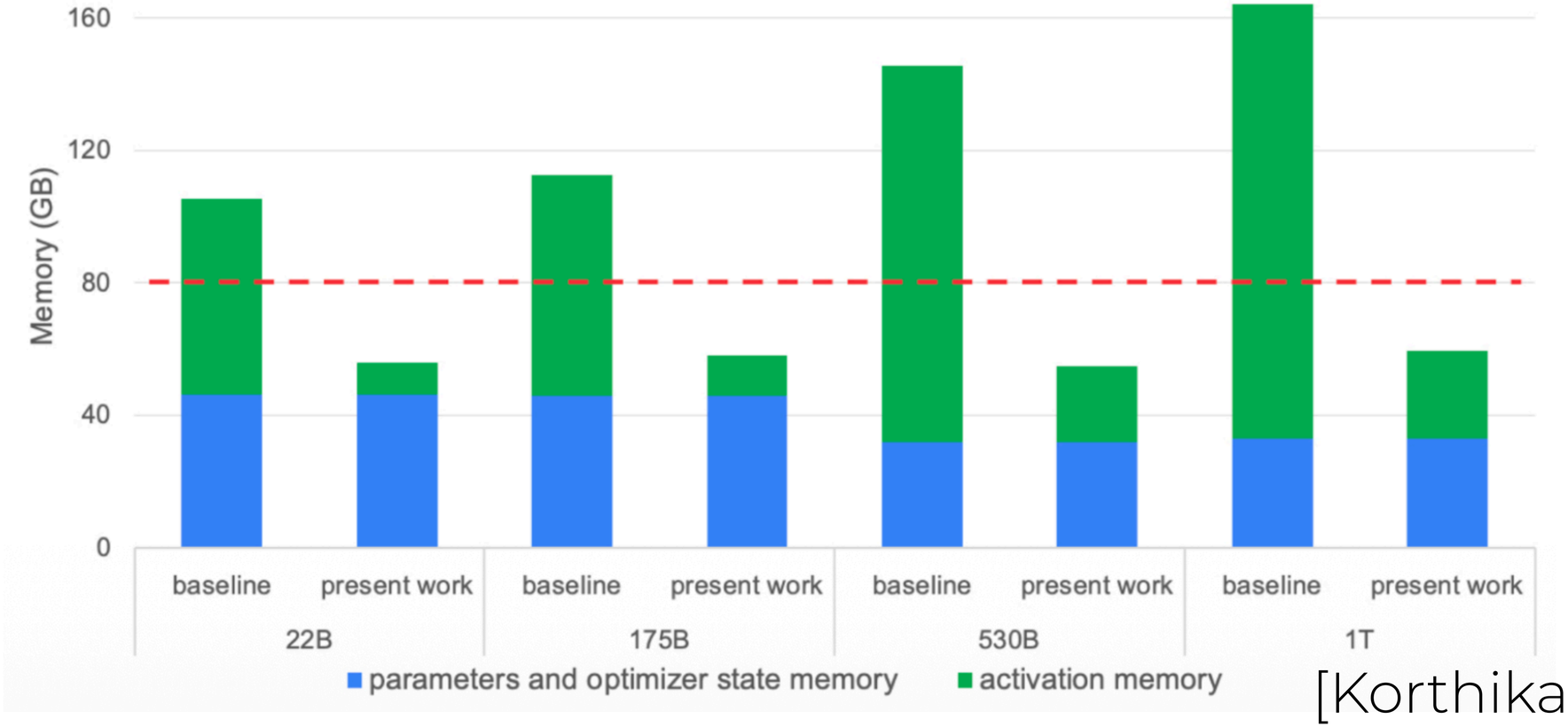
Memory is Dynamic

- Memory isn't just the static memory, but there is also activations



Activation Memory

- Thus far, we have only really discussed parameter memory
- TP and PP can linearly reduce those.. but what about activations?



[Korthikanti+ 2022]

Activation Memory

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

- Amount of activation memory of a Transformer layer

$$24sbh + 10sbh + 5abs^2$$

- $24sbh$ (matmul)
  - QKV input: s x b x h tensors (fp16) x 3 = **3 x 2sbh**
  - Attention output: **2sbh**
  - FFN input/output: Transformer 4x input: **2 x 4 x 2sbh**

- $10sbh$ (pointwise ops)
  - LayerNorm: two LNs: **2 x 2sbh**
  - Residual: 2 residual (attention, FFN): **2 x 2sbh**
  - Dropout: 2 dropout (attention, FFN): **2 x 1sbh** (boolean)

Activation Memory

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

- Amount of activation memory of a Transformer layer

$$24sbh + 10sbh + 5abs^2$$

- $5abs^2$ (attention)
  - Attention logits (QK^T): [b, a, s, s] = **2abs²**
  - Softmax prob. (Softmax of logits): [b, a, s, s] = **2abs²**
  - Attention dropout: [b, a, s, s] = **1abs²**

- In summary, we get

$$\text{Activations memory per layer} = sbh\left(34 + 5\frac{as}{h}\right)$$

Activation Memory

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

- Amount of activation memory of a Transformer layer

$$24sbh + 10sbh + 5abs^2$$

- $5abs^2$ (attention)
  - Attention logits (QK^T): [b, a, s, s] = **2abs²**
  - Softmax prob. (Softmax of logits): [b, a, s, s] = **2abs²**
  - Attention dropout: [b, a, s, s] = **1abs²**

we can drop this with recomputation

- In summary, we get

$$\text{Activations memory per layer} = sbh\left(34 + 5\frac{as}{h}\right)$$

+ Tensor Parallel

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

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

- TP splits out matmuls in attention and FFN
- **10sbh**: cannot do linear scaling with TP
  - No matmul in LN, residual, dropout

Sequence Parallel [Korthikanti+ 2022]

- **10sbh**: cannot do TP (no matmul in LN, residual, dropout)
  - How can we parallelize this term?
  - Split pointwise ops in sequence axis! [Korthikanti+ 2022]

  - These ops are pointwise, meaning no dependency in seq. dim
  - We can safely split in sequence axis

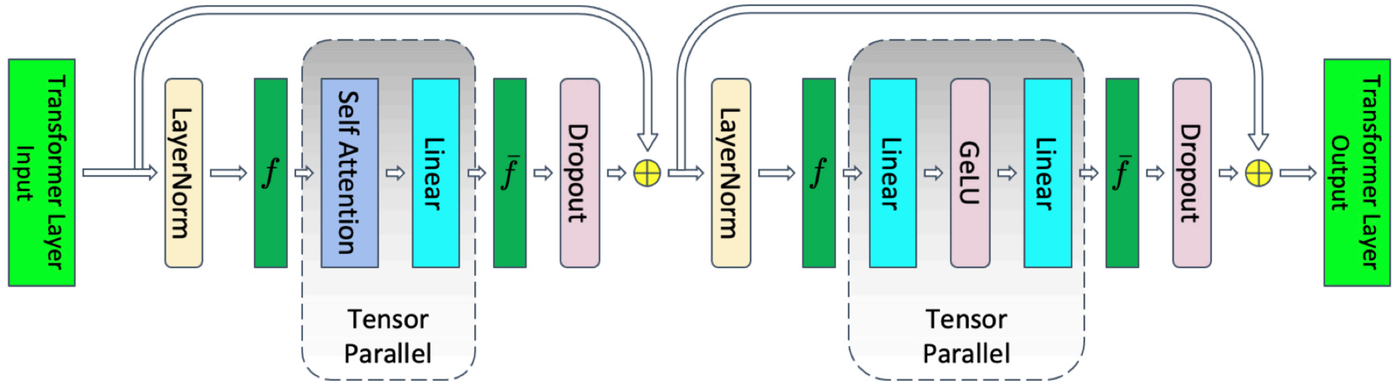
$$\text{Activations memory per layer} = sbh\left(\frac{10}{t} + \frac{24}{t} + 5\frac{as}{ht}\right) = \frac{sbh}{t}\left(34 + 5\frac{as}{h}\right).$$

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

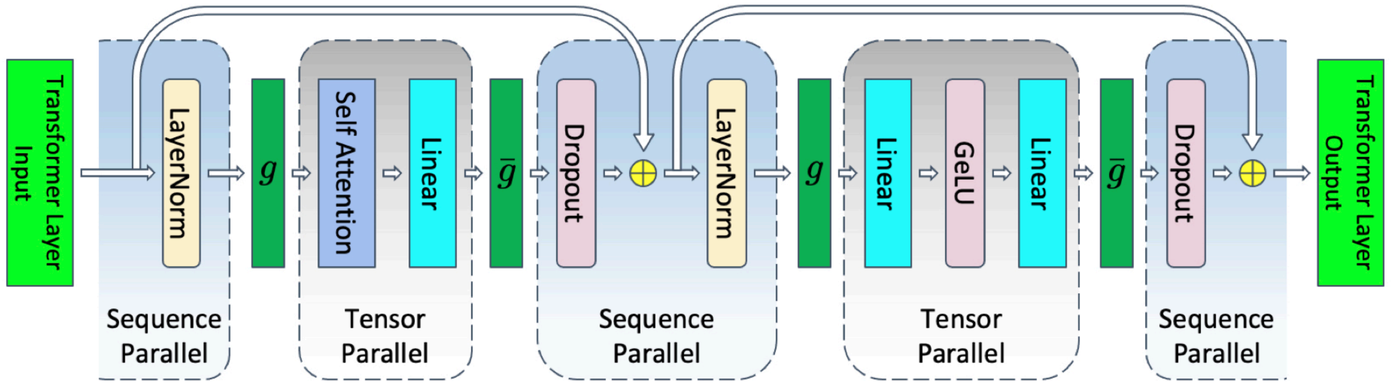
TP and SP [Korthikanti+ 2022]

$\bar{f}$ is all-reduce (forward)

TP



TP+SP



split all-reduce into reduce-scatter ($\bar{g}$) and all-gather (g); no additional comm. cost

TP + SP + Recomputation [Korthikanti+ 2022]

- Putting it together to get full linear scaling for memory

Configuration	Activations Memory Per Transformer Layer
no parallelism	$sbh\left(34+5\frac{as}{h}\right)$
tensor parallel (baseline)	$sbh\left(10+\frac{24}{t}+5\frac{as}{ht}\right)$
tensor + sequence parallel	$sbh\left(\frac{34}{t}+5\frac{as}{ht}\right)$
tensor parallel + selective activation recomputation	$sbh\left(10+\frac{24}{t}\right)$
tensor parallel + sequence parallel + selective activation recomputation	$sbh(\frac{34}{t})$

Recap

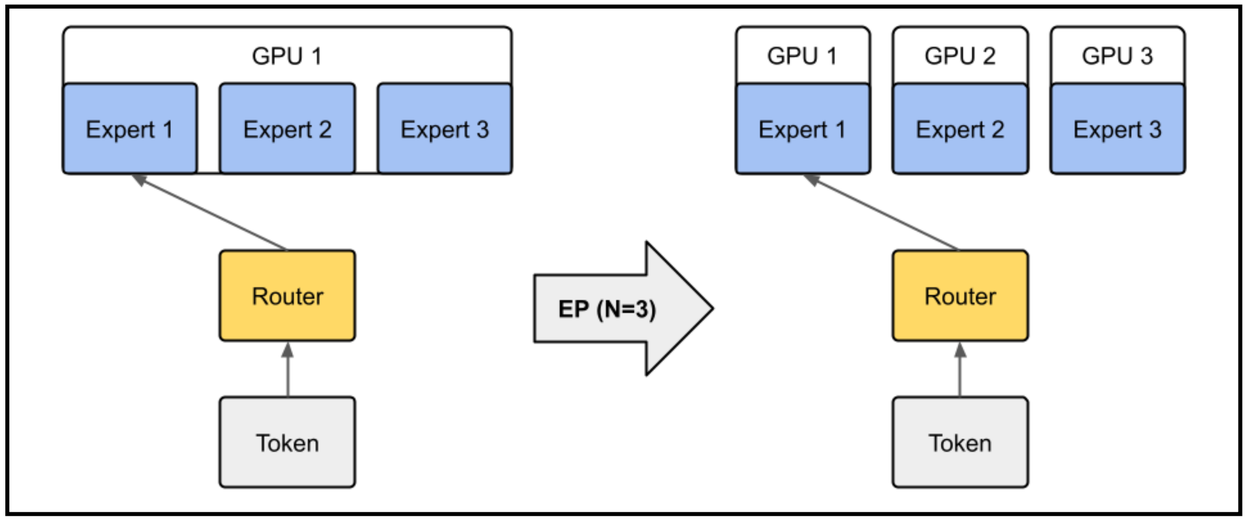
- What are each of the parallelism primitives good for?

	Sync. Overhead	Memory	Bandwidth	Batch size
DDP/ZeRO-1	Per-batch	No scaling	2x # param.	Linear
FSDP (ZeRO-3)	3x per-FSDP block	Linear	3x # param.	Linear
PP	Per-pipeline	Linear	bsh	Linear
TP + SP	2x Transformer block	Linear	8bsh per-layer (all-reduce)	No impact

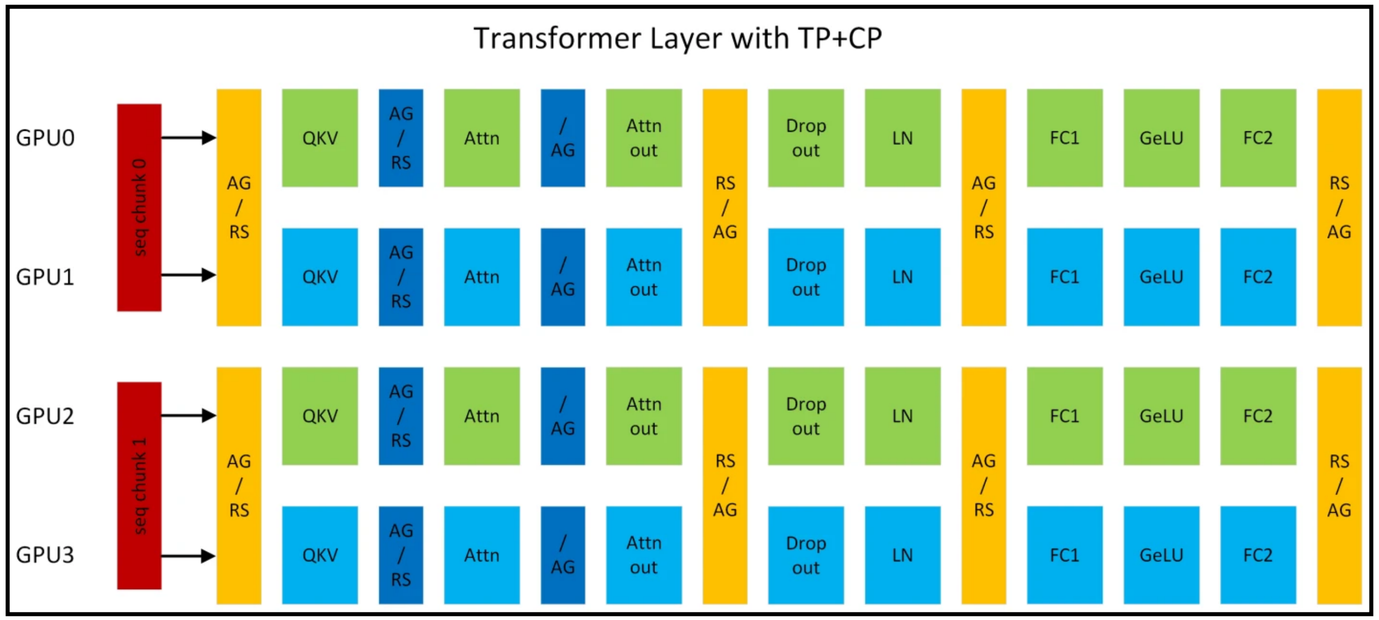
- Have to balance limited resource – memory, bandwidth, batch size

Other Parallelisms

- Expert parallelism (EP)
  - Splits experts in MoE models

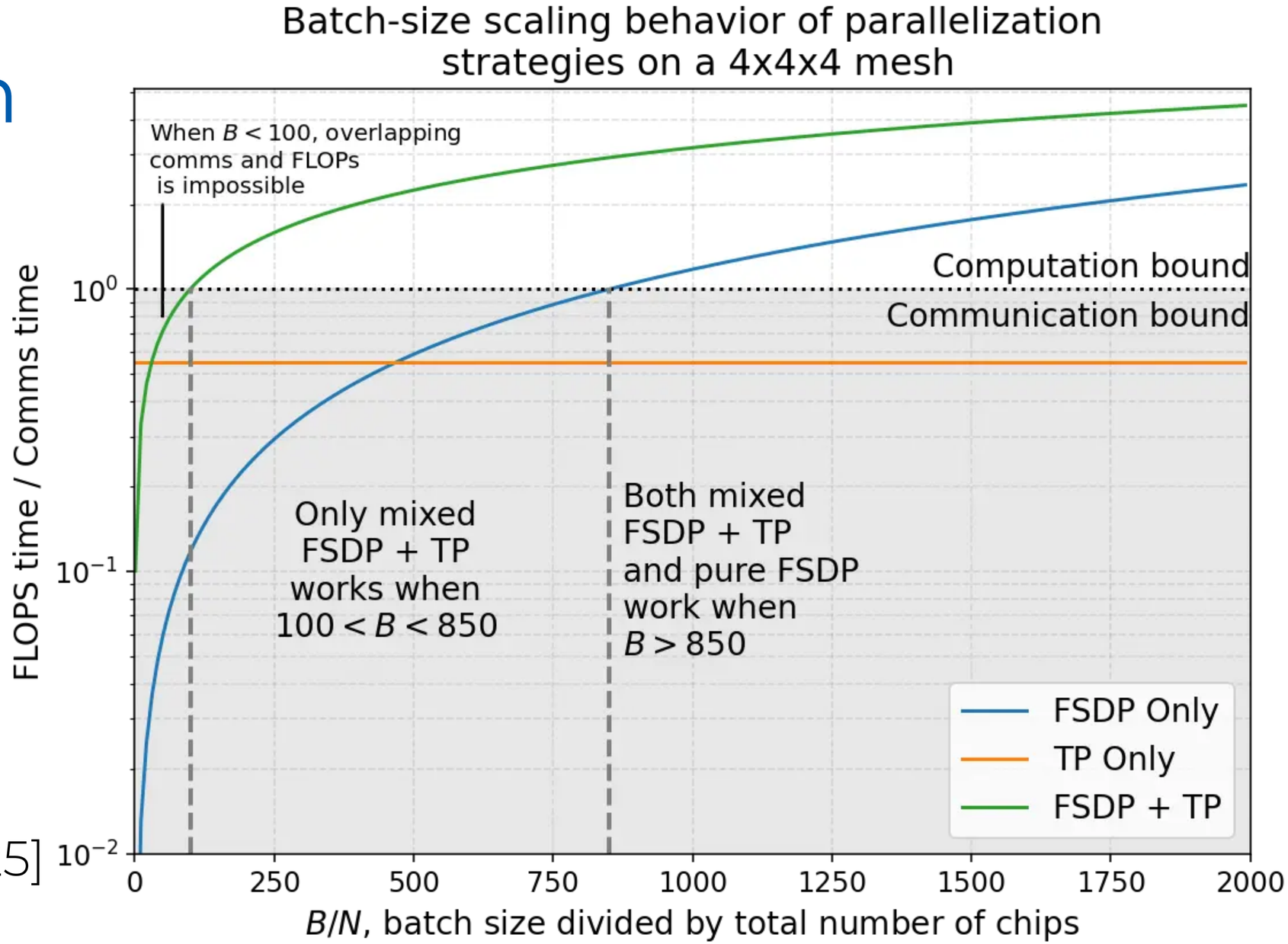


- Context parallelism (CP)
  - Split a long sequence into multiple GPUs



2D Parallelism

- DP + TP

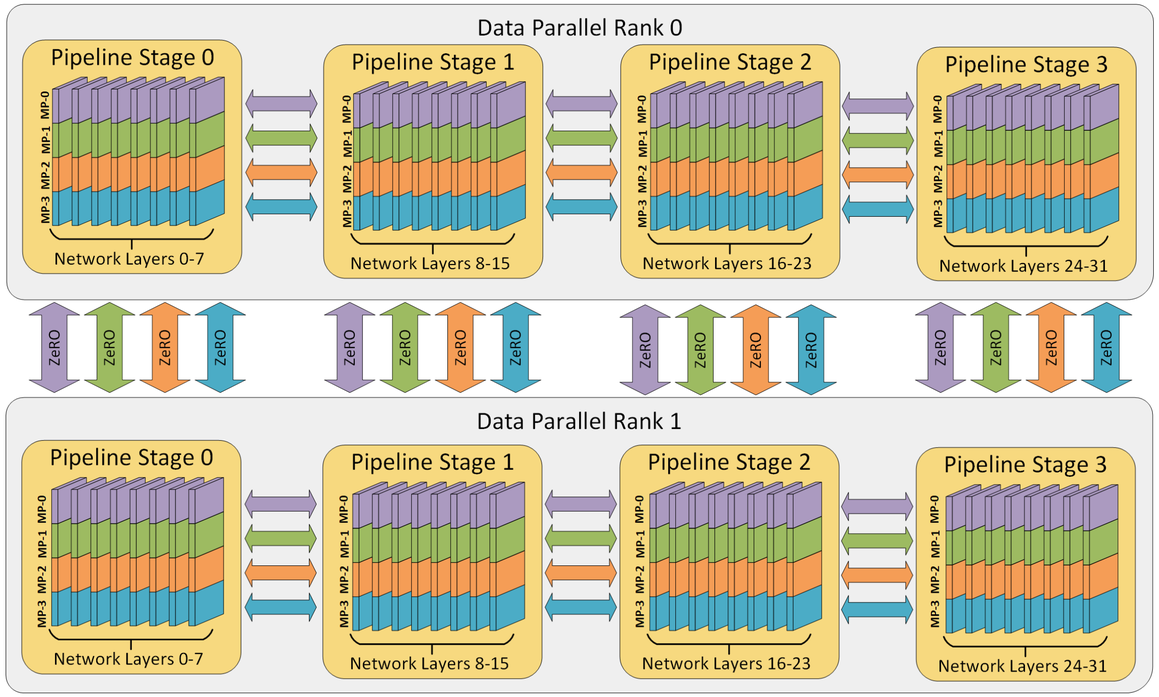


[Austin+ 2025]

3D Parallelism

- Simple rules of thumb from the literature
- 1. Until your model fits in memory
  - TP up to GPUs / machine
  - PP across machines
  - (Or use ZeRO-3, depending on BW)
- 2. Then until you run out of GPUs
  - Scale the rest of the way with DP

- If your batch size is small…
  gradient accumulate to trade higher
  batch sizes for better communication efficiency

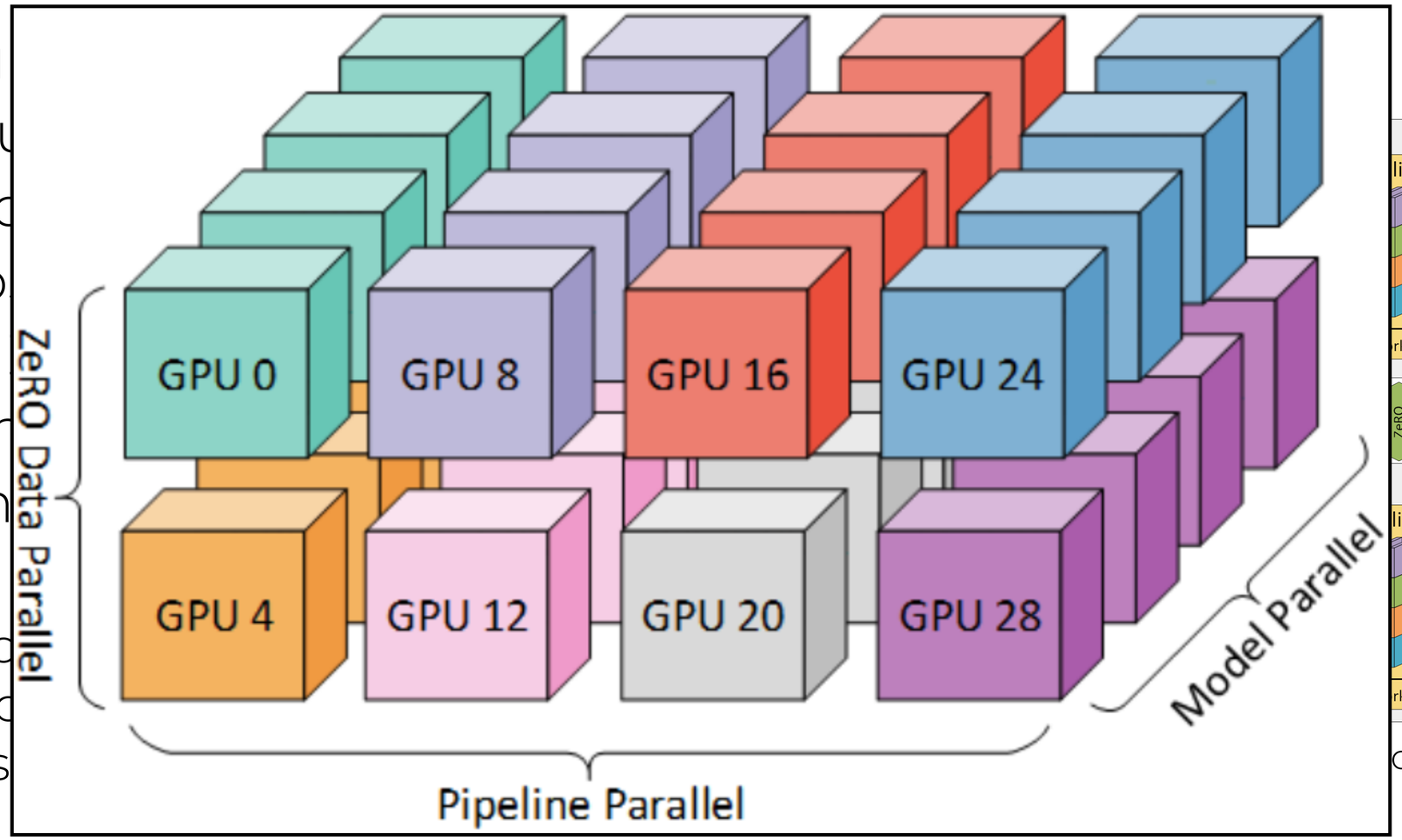


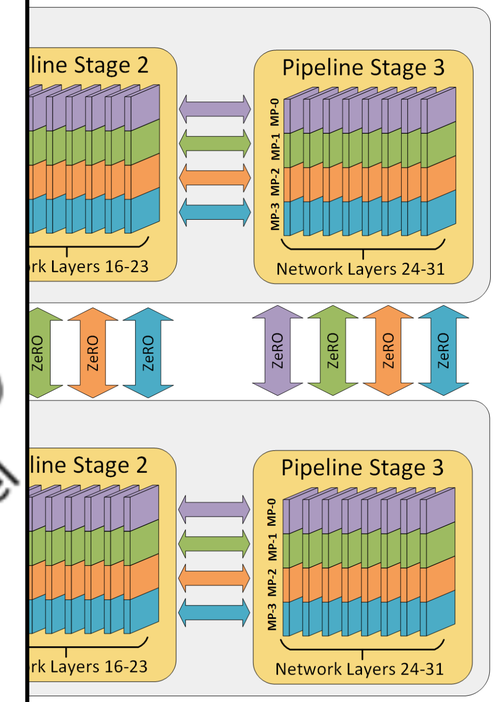
https://www.deepspeed.ai/tutorials/pipeline/

3D Parallelism

- Simple ru
- 1. Until you
  - TP up to
  - PP acro
  - (Or use
- 2. Then ur
  - Scale th

- If your batc
  gradient ac
  batch sizes





d.ai/tutorials/pipeline/

3D Parallelism in Megatron-LM [Narayanan+ 2021]

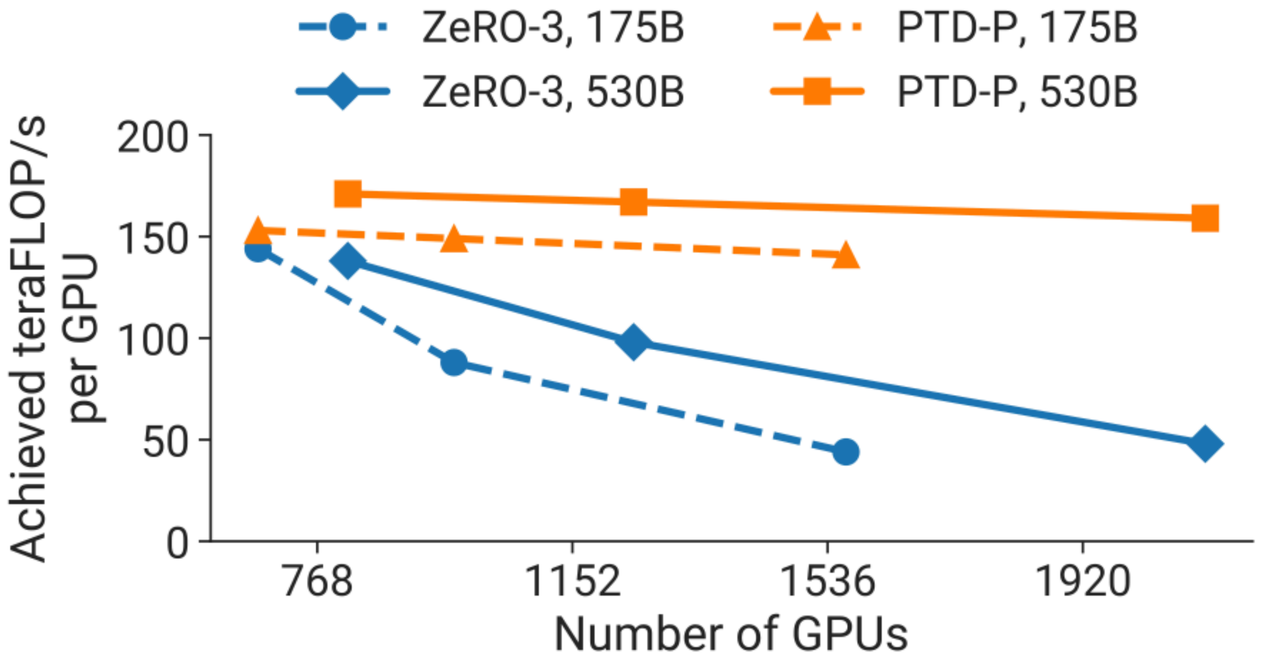
Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s	DP size
1.7	24	2304	24	1	1	32	512	137	44%	4.4	32
3.6	32	3072	30	2	1	64	512	138	44%	8.8	
7.5	32	4096	36	4	1	128	512	142	46%	18.2	
18.4	48	6144	40	8	1	256	1024	135	43%	34.6	
39.1	64	8192	48	8	2	512	1536	138	44%	70.8	
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8	
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1	
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4	15
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2	9
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0	6

Table 1: Weak-scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

- TP first up to 8, then caps out at 8
- PP goes up to make the model fit
- DP gradually decreases with scale, with the largest model having 6

3D Parallelism in Megatron-LM [Narayanan+ 2021]

- PTD-P: pipeline + tensor + data parallel



3D Parallelism in Megatron-LM [Narayanan+ 2021]

- Activation recomputaion helps?
  - For small batch sizes, it leads up to 33% lower throughput
  - But without recomputation, we cannot increase the batch size

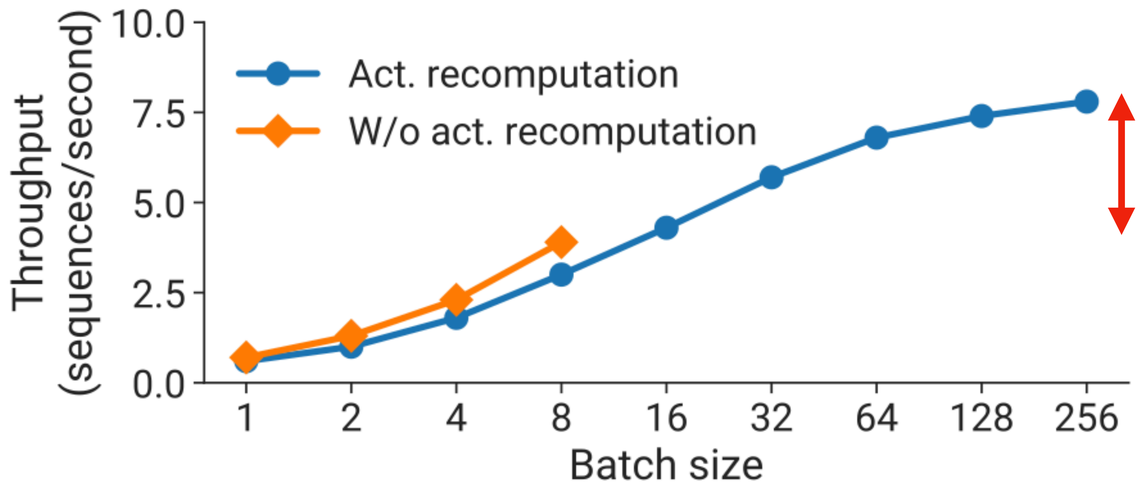
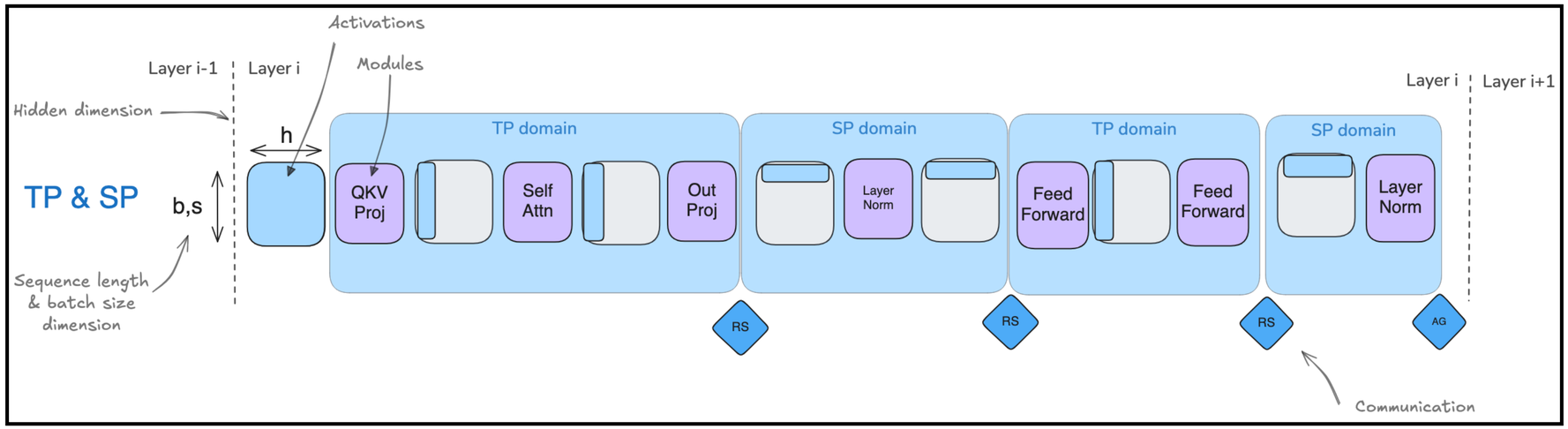


Figure 17: Throughput (in sequences per second) with and without activation recomputation for a GPT model with 145 billion parameters using 128 A100 GPUs ($(t, p) = (8, 16)$).

Image credit: https://gdymind.com/2026/02/07/5D-parallelism-in-LLM-training/

Summary: ND Parallelism



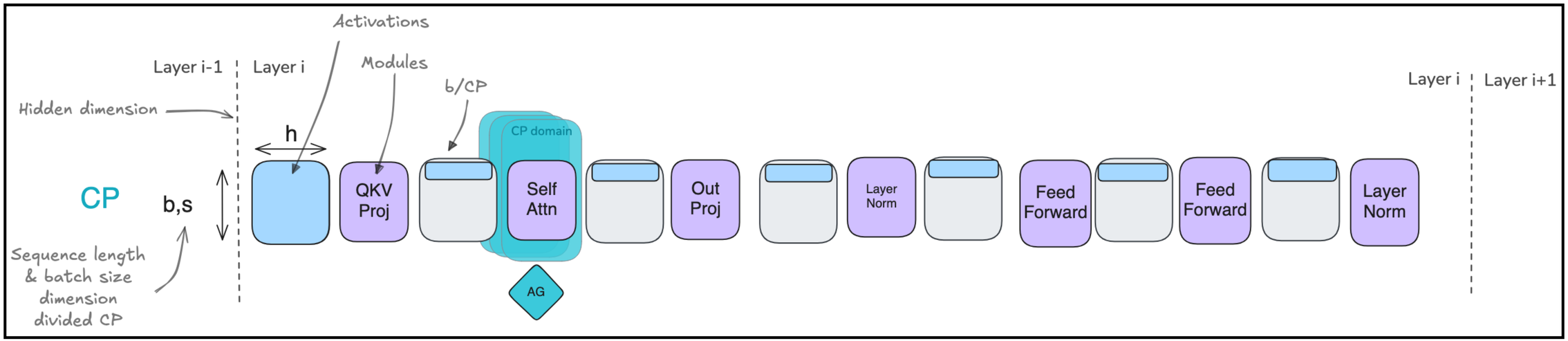
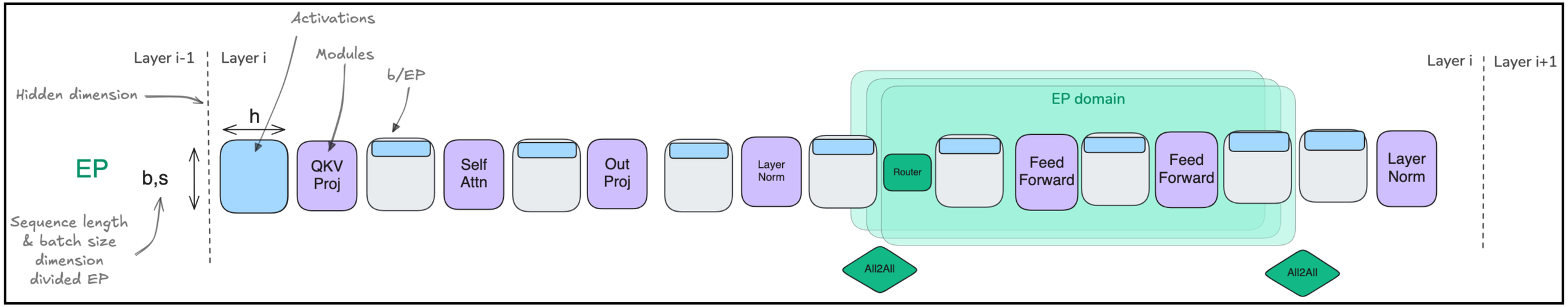
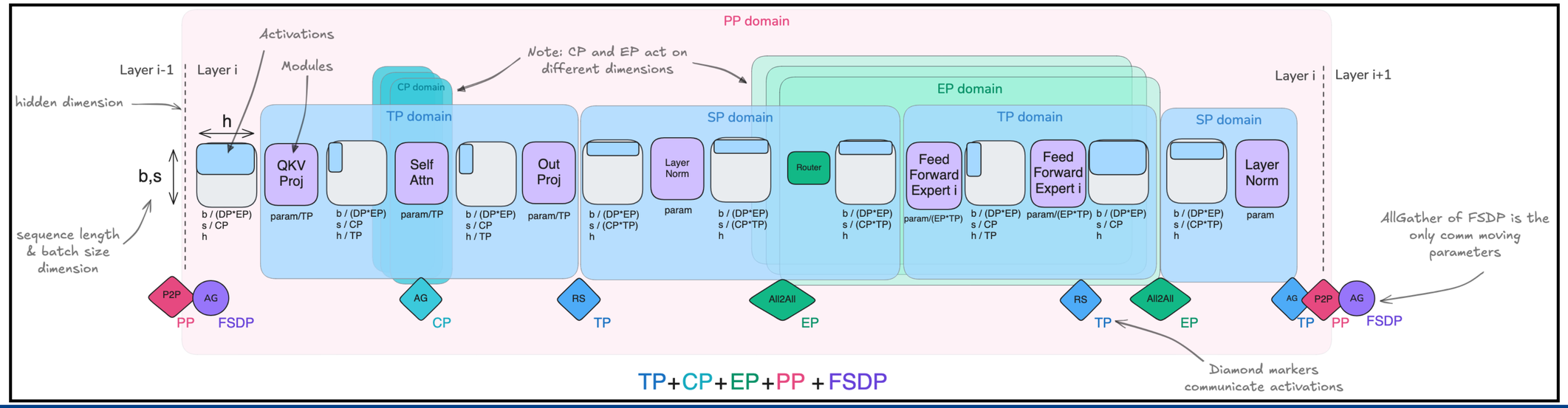


Image credit: https://gdymind.com/2026/02/07/5D-parallelism-in-LLM-training/

Summary: ND Parallelism





Case Study

LLaMA 3

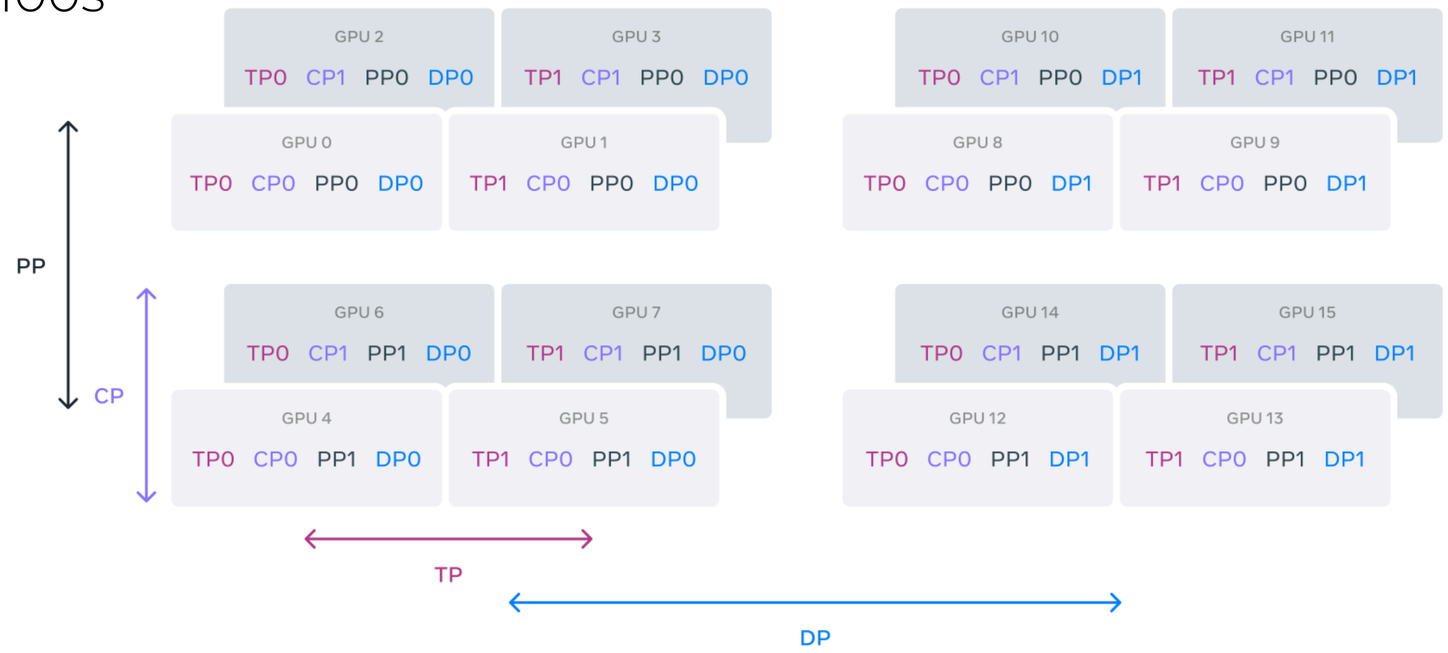
- 4D parallelism: TP + CP + PP + FSDP

LLaMA 3-405B
3-stage training

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	8	131,072	16	16M	380	38%

small bsize
large bsize
long context

H100s



LLaMA 3

Component	Category	Interruption Count	% of Interruptions
Faulty GPU	GPU	148	30.1%
GPU HBM3 Memory	GPU	72	17.2%
Software Bug	Dependency	54	12.9%
Network Switch/Cable	Network	35	8.4%
Host Maintenance	Unplanned Maintenance	32	7.6%
GPU SRAM Memory	GPU	19	4.5%
GPU System Processor	GPU	17	4.1%
NIC	Host	7	1.7%
NCCL Watchdog Timeouts	Unknown	7	1.7%
Silent Data Corruption	GPU	6	1.4%
GPU Thermal Interface + Sensor	GPU	6	1.4%
SSD	Host	3	0.7%
Power Supply	Host	3	0.7%
Server Chassis	Host	2	0.5%
IO Expansion Board	Host	2	0.5%
Dependency	Dependency	2	0.5%
CPU	Host	2	0.5%
System Memory	Host	2	0.5%

Table 5 Root-cause categorization of unexpected interruptions during a 54-day period of Llama 3 405B pre-training. About 78% of unexpected interruptions were attributed to confirmed or suspected hardware issues.

DeepSeek-V3

- 3D parallelism: PP (DualPipe) + EP + ZeRO-1 DP
  - PP = 16, EP = 64
  - 2048 x H800

DualPipe

The training of DeepSeek-V3 is supported by the HAI-LLM framework, an efficient and lightweight training framework crafted by our engineers from the ground up. On the whole, DeepSeek-V3 applies 16-way Pipeline Parallelism (PP) (Qi et al., 2023a), 64-way Expert Parallelism (EP) (Lepikhin et al., 2021) spanning 8 nodes, and ZeRO-1 Data Parallelism (DP) (Rajbhandari et al., 2020).

Kimi K2

- 3D parallelism: PP (interleaved 1F1B) + EP + ZeRO-1 DP
  - PP = 16, EP = 16

2.4.2 Parallelism for Model Scaling

Training of large language models often progresses under dynamic resource availability. Instead of optimizing one parallelism strategy that's only applicable under specific amount of resources, we pursue a flexible strategy that allows Kimi K2 to be trained on any number of nodes that is a multiple of 32. Our strategy leverages a combination of 16-way Pipeline Parallelism (PP) with virtual stages [29, 54, 39, 58, 48, 22], 16-way Expert Parallelism (EP) [40], and ZeRO-1 Data Parallelism [61].

Solar Open

- Pure FSDP
  - 480 x B200

DeepSpeed. Our evaluation reveals two key insights: (1) for 100-billion-parameter scale models, larger batch sizes enabled by full activation checkpointing outweigh memory savings from selective checkpointing strategies; (2) expert parallelism and tensor parallelism provide no benefit over pure FSDP for our configuration, suggesting that architectural simplicity can outperform complex parallelization schemes at this scale.

3.3.2 Multi-Node Scaling Challenge

Standard FSDP2 (Zhao et al., 2023) performance degrades significantly when scaling from 16 to 60 nodes, with TPS dropping from 5,500 to 4,267. This degradation stems from all-reduce operations for gradient synchronization, which scale sub-linearly with node count due to inter-node communication volume. We address this by adopting Hybrid Sharding Data Parallel (HSDP), an extension of FSDP available in TorchTitan that divides the global device pool into smaller sharding groups. HSDP runs FSDP within each sharding group (10 nodes in our configuration) and synchronizes gradients across groups (6 replicas). This hierarchical structure confines most communication to intra-group operations while maintaining global gradient consistency through periodic inter-group all-reduce, achieving 26.5% throughput improvement and reaching 5,400 TPS at 60 nodes. This result demonstrates that exploiting the bandwidth differential between intra-node and inter-node communication effectively mitigates the saturation point of standard FSDP at large scale.

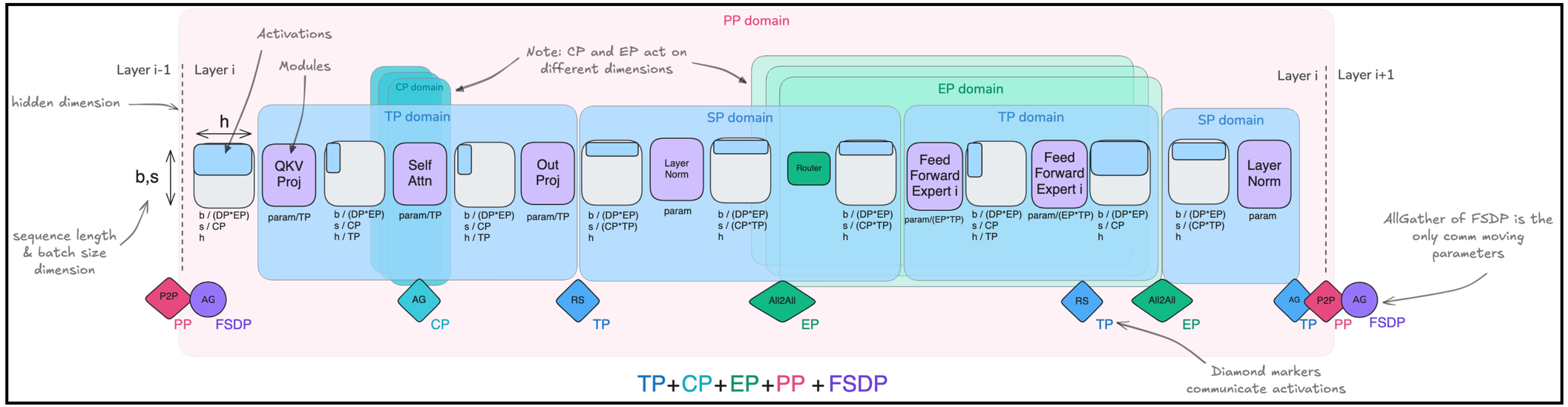
GLM-5

- 3D parallelism: PP (interleaved) + ZeRO-2 + CP

Summary

- Scaling beyond a certain point requires multi-gpu/-node parallelism

- No single solution to the parallelism problem
  - Recent models adopt mixed ND strategy



Further Reading

- How to scale your model? (from Deepmind)
  - https://jax-ml.github.io/scaling-book/
  - https://jh-michael-shin.github.io/scaling-book/ (translated version)