Scaling LLM Training: Part II

Vedant Nanda
Researcher @ Aleph Alpha Research

Recap from part I

- Learned to calculate FLOPs and Memory
  - Most FLOPs and params in MLP
- Memory and compute needed for LLM Training
- Getting the most out of one GPU
  - Activation Checkpointing
  - Gradient Accumulation

Exercise from part I

- Solution up on GitHub, you should be able to step through the notebook on colab
- Reach out in case of any issues!

Exercise for Today

- Today we get hands dirty getting things to work on more than one GPU!
- Runpod machines with 2xA40s, divide in groups of X(?) — things will work this time, I promise :)
  - Send me one public key per group

Part II — Going Beyond One GPU

All about multi-GPU training!

- Data parallelism recap (covered in part I) — 10 min
- ZeRO redundancy sharding / Fully-sharded data parallel — 25 min
- Tensor parallel — 20 min
- Tensor + sequence parallel — 10 min
- Context parallel— 15 min
- Concluding thoughts — 5 min

Scaling Challenges — Recap

Achieving “strong scaling”, ie, increase the number of chips used for training while achieving a proportional, linear increase in throughput, is hard due to communication overheads

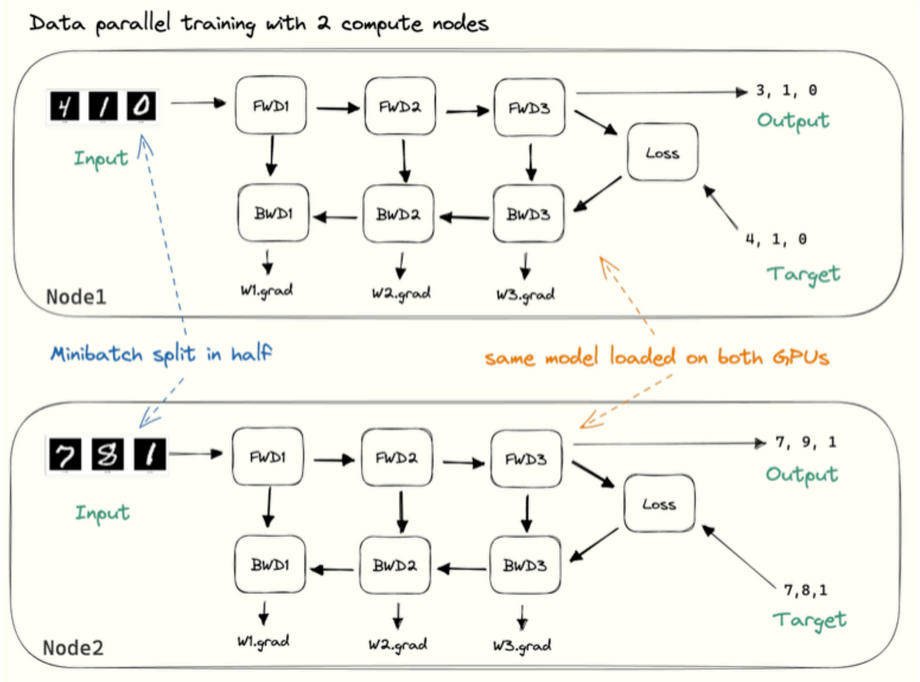
Fitting everything (model, optimizer, gradients, activations) in memory

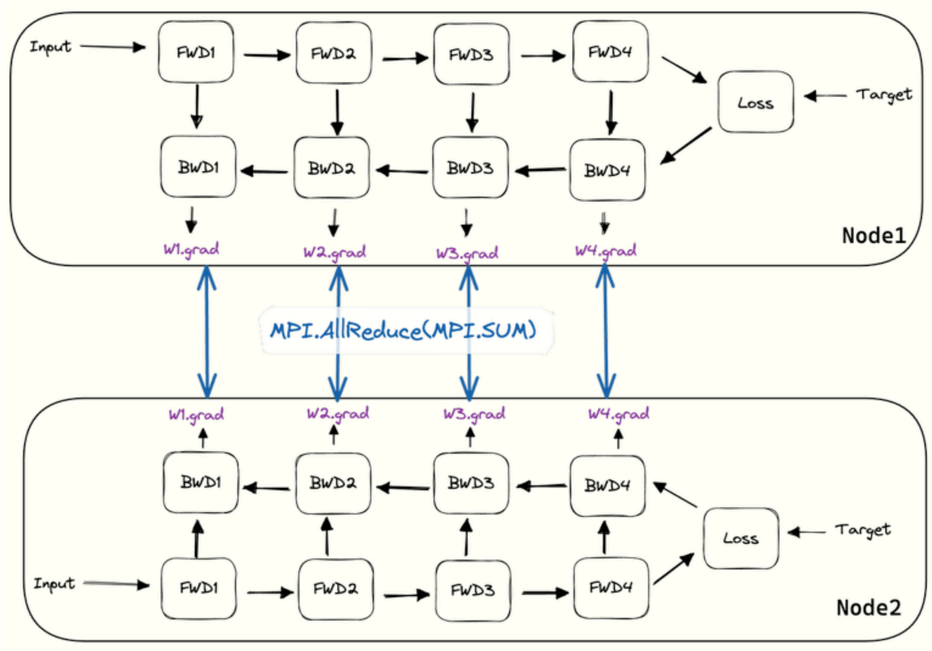
Choosing the right strategy for sharding / parallelizing when things don’t fit in memory

Alleviating communication bottlenecks that arise from parallelisms and sharding

Data Parallelism — Recap

Data parallelism: run different batches of data in parallel on different chips

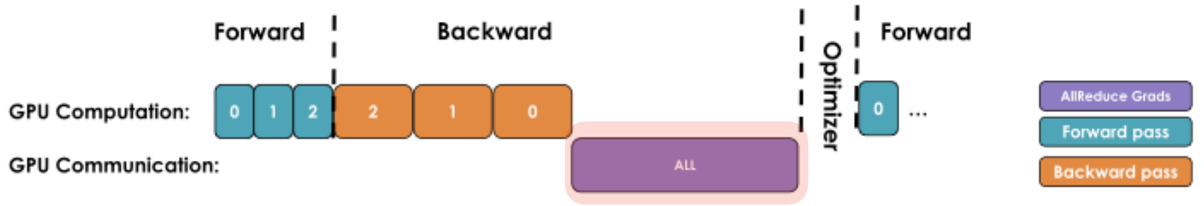




Credits: https://siboehm.com/articles/22/data-parallel-training

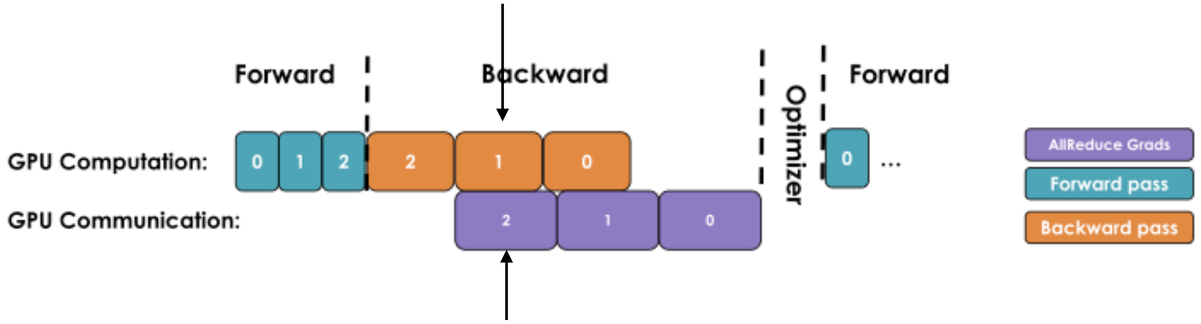
Data Parallelism w Overlapping Comms — Recap

View on each chip:



Every device is waiting for communication

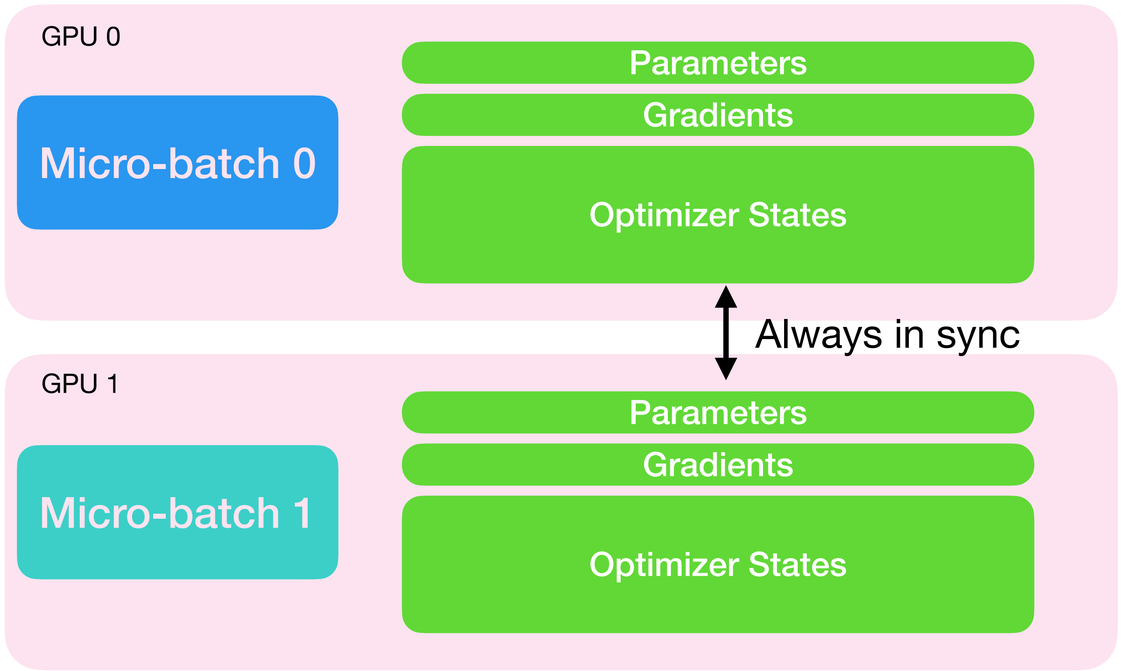
overlapping compute with comm



Start communicating gradient of layer 2 when done

Redundancy in Data Parallel

Model params (+ grads + optimizer states) are duplicated on all GPUs

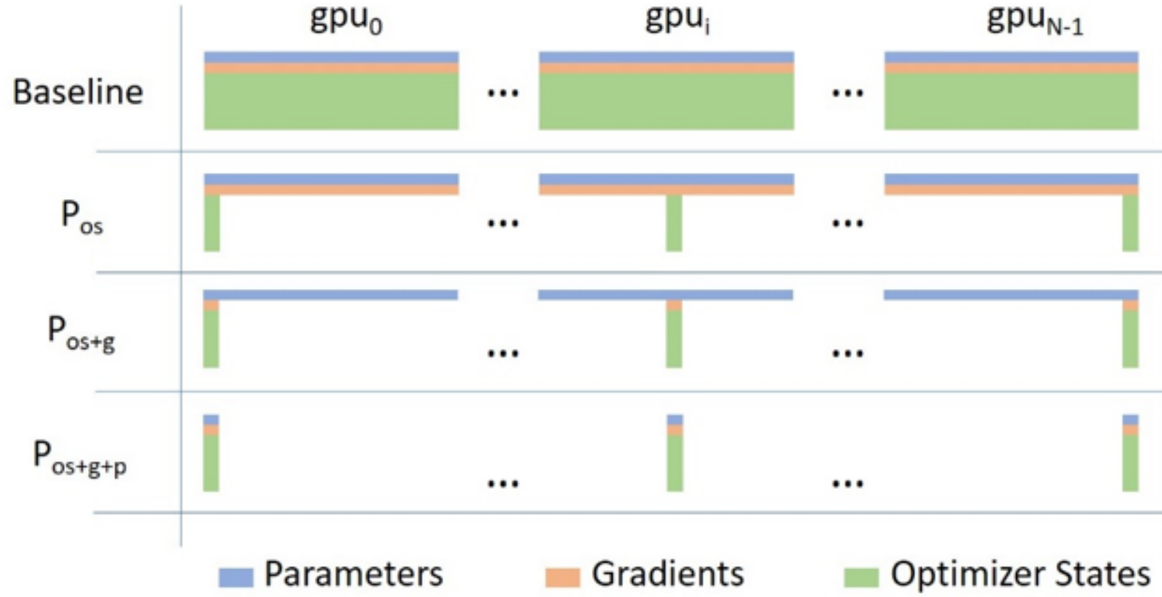


What about the cases where params + grads + optimizer + activations don't fit on a single card?

We communicate anyways after backward pass, do we need to keep all model states on every device?

ZeRO Sharding

Shard model states to reduce redundancy in memory



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
ZeRO-1 (P_{os})	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
ZeRO-2 (P_{os+g})	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
ZeRO-3 / FSDP (P_{os+g+p})	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

ZeRO-1: Each device updates its portion of optimizer states

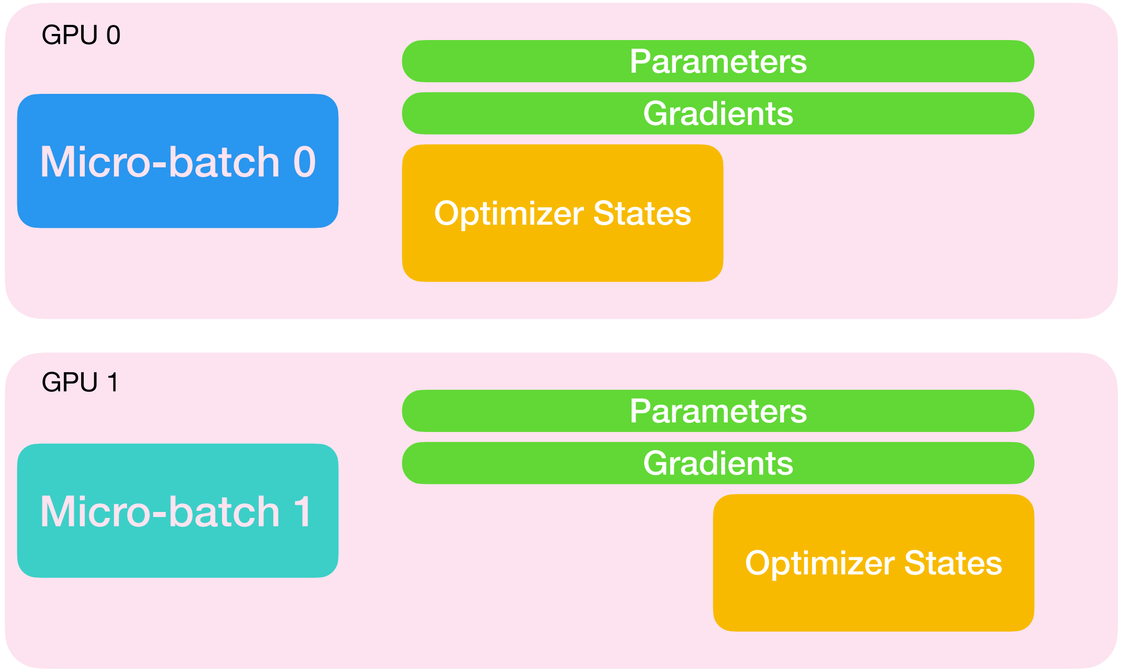
ZeRO-2: Each device updates its portion of grads + optimizer states

ZeRO-3 / FSDP: Each device only holds a portion of params and its corresponding grads + optimizer states

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, SC 2020, Rajbhandari, Rasley et al.

Fully Sharded Data Parallel: faster AI training with fewer GPUs, Meta AI, 2021

ZeRO-1 Sharding Optimizer States



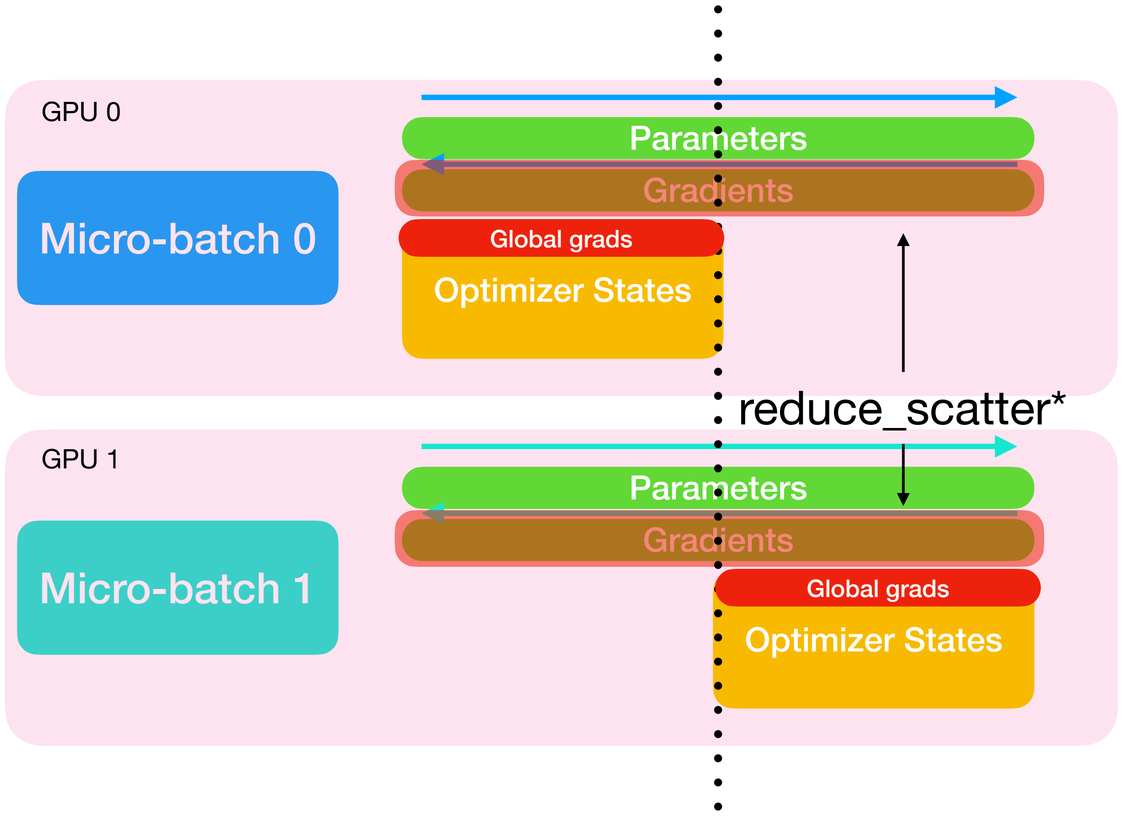
All params and grads on both devices

However, each device only has its portion of optimizer states

Memory (per GPU) =

$$2.P + 2.P + \frac{12.P}{\text{num\_gpus}}$$

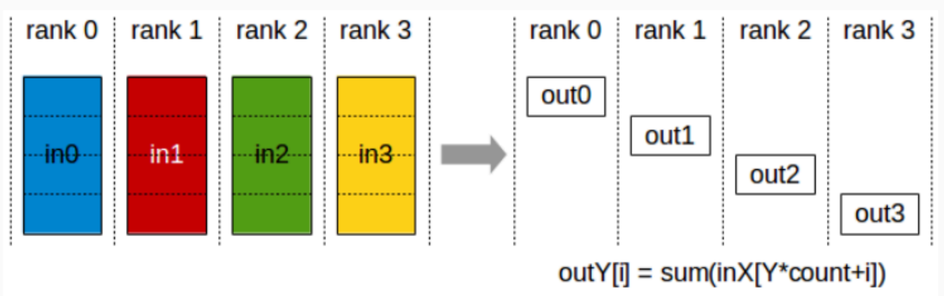
ZeRO-1 Sharding Optimizer States



Memory (per GPU) =

$$2.P + 2.P + \frac{12.P}{\text{num\_gpus}}$$

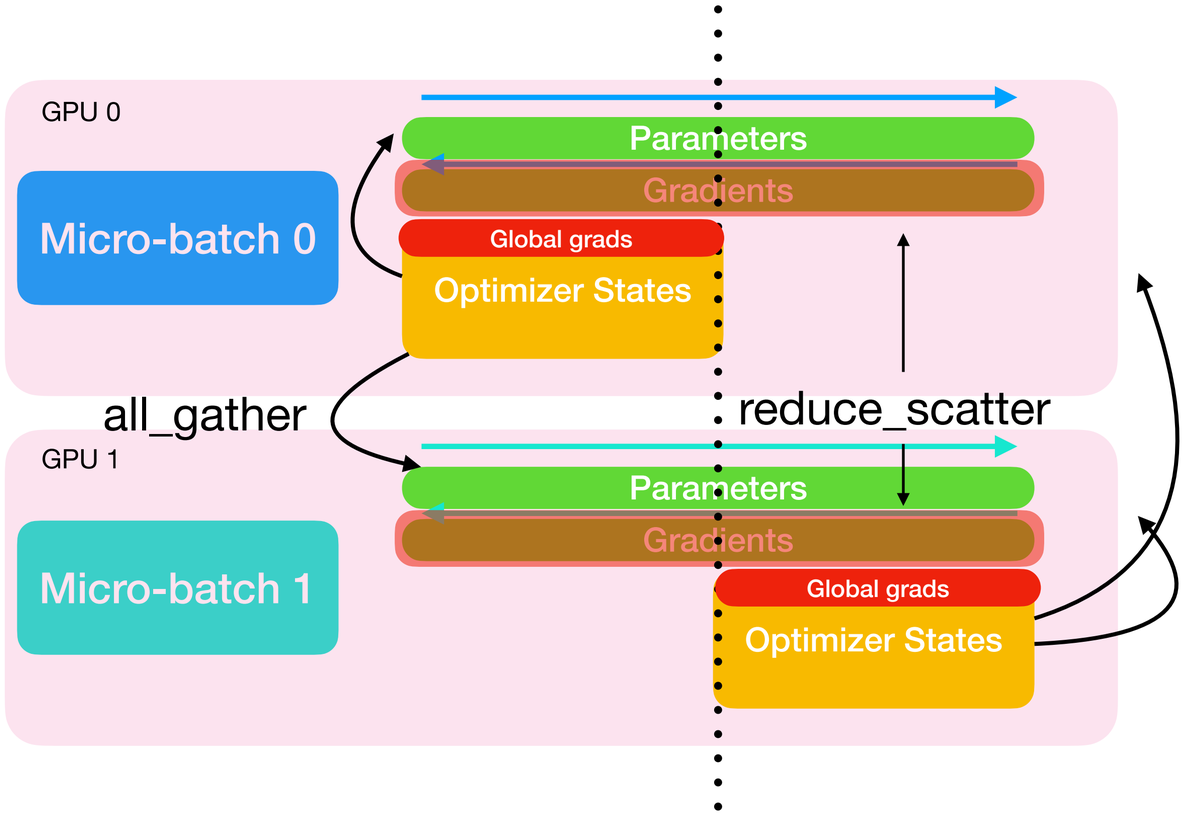
Each device runs its own optimizer step



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

ZeRO-1 Sharding Optimizer States



Memory (per GPU) =

$$2.P + 2.P + \frac{12.P}{\text{num\_gpus}}$$

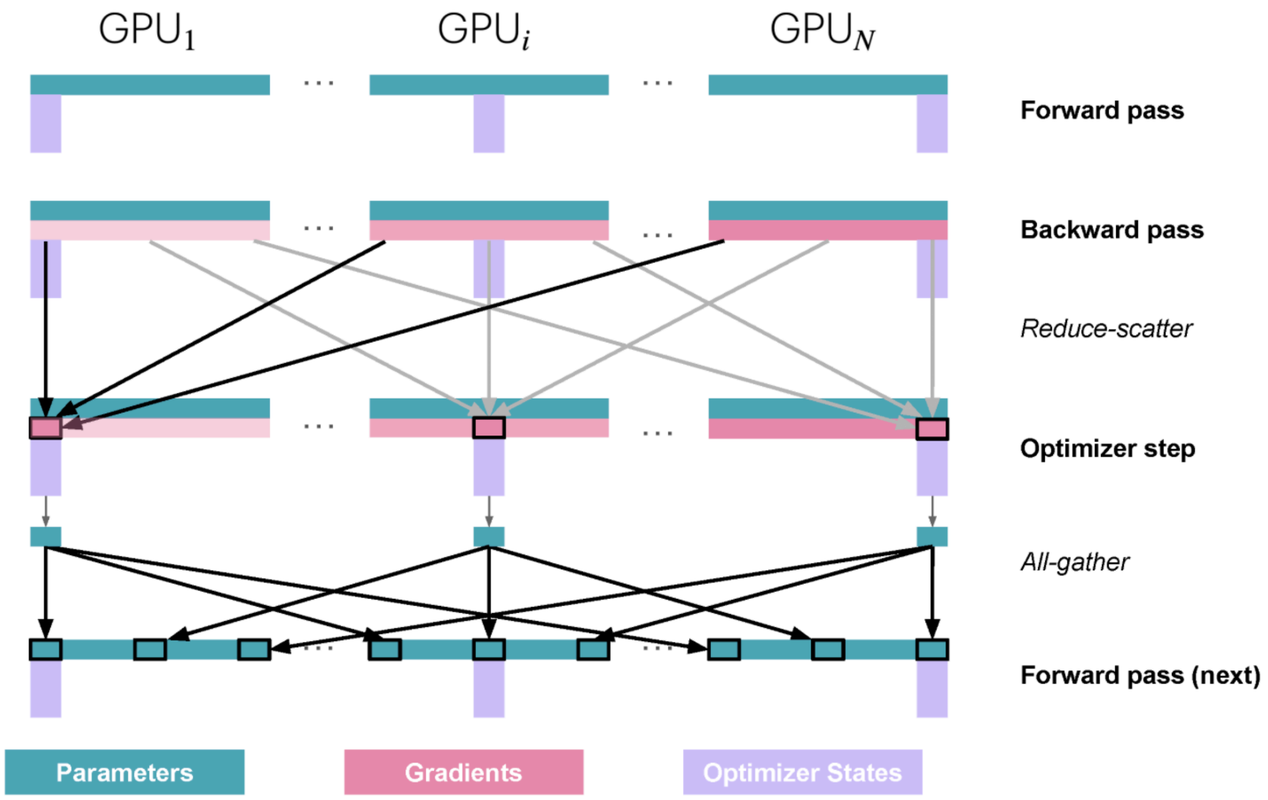
Each device runs its own optimizer step

After optimizer step, communicate new params to each device

[+] reduce_scatter is more efficient than all_reduce

[+] Lesser memory by sharding optimizer states

ZeRO-1 Sharding Optimizer States



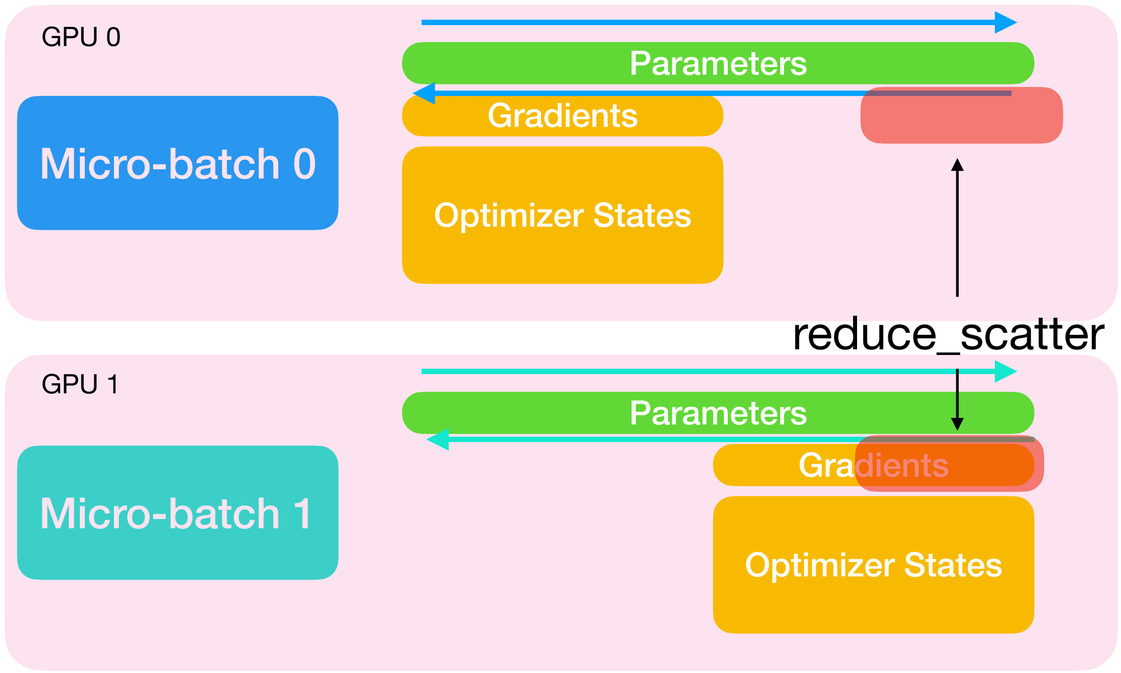
[+] reduce_scatter is more efficient than all_reduce

[+] Lesser memory by sharding optimizer states

[-] An additional all_gather after optimizer step

If we only need gradients of each device's portion of optimizer states, then why do we need to replicate gradients?

ZeRO-2 Optimizer + Gradients



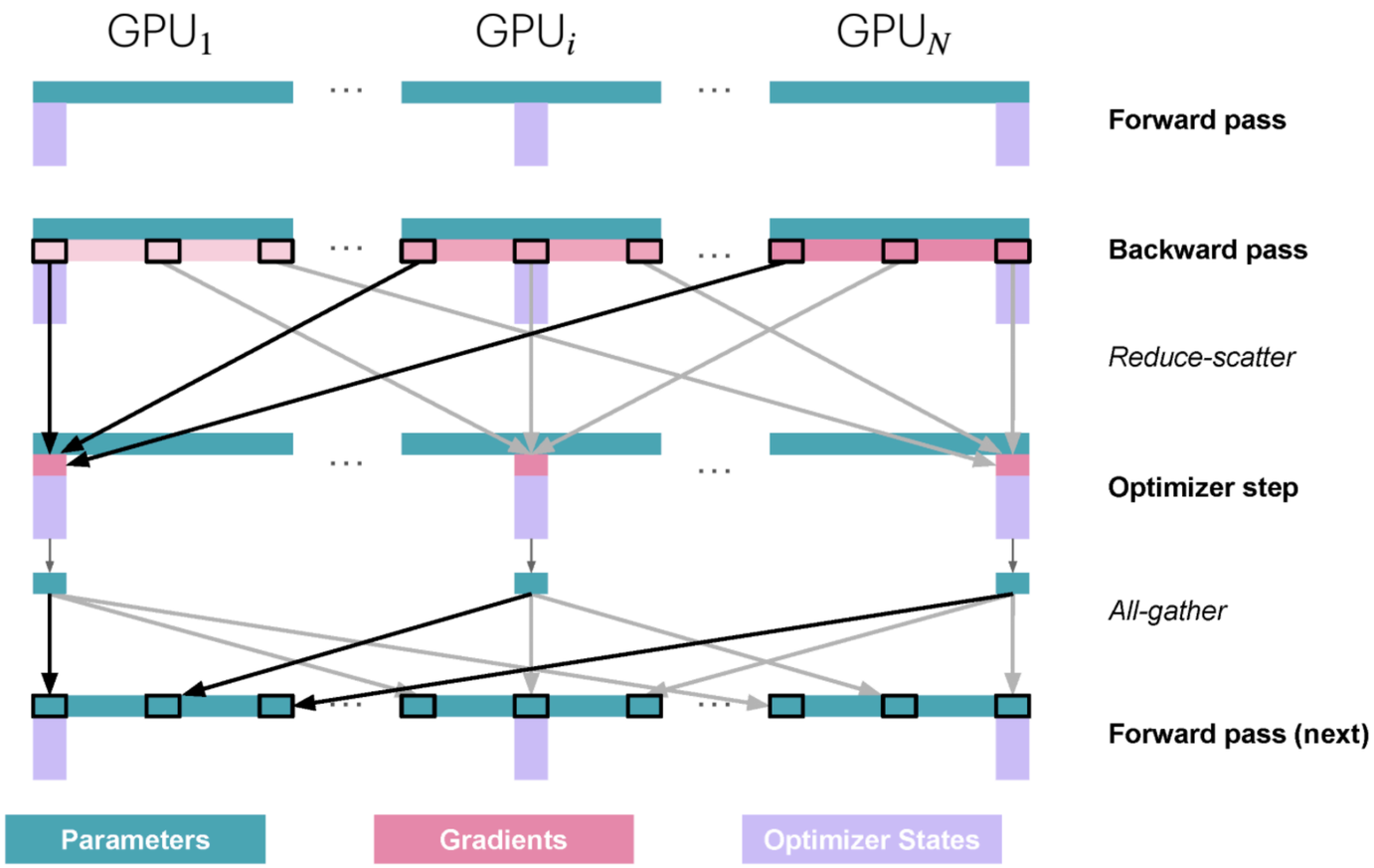
Memory (per GPU) =

$$2.P + \frac{2.P + 12.P}{\text{num\_gpus}}$$

Key difference from ZeRO-1: drop gradients from device where it does not belong right after reduce_scatter

[+] Exactly the same as ZeRO-1, except with lower memory requirement

ZeRO-2 Optimizer + Gradients



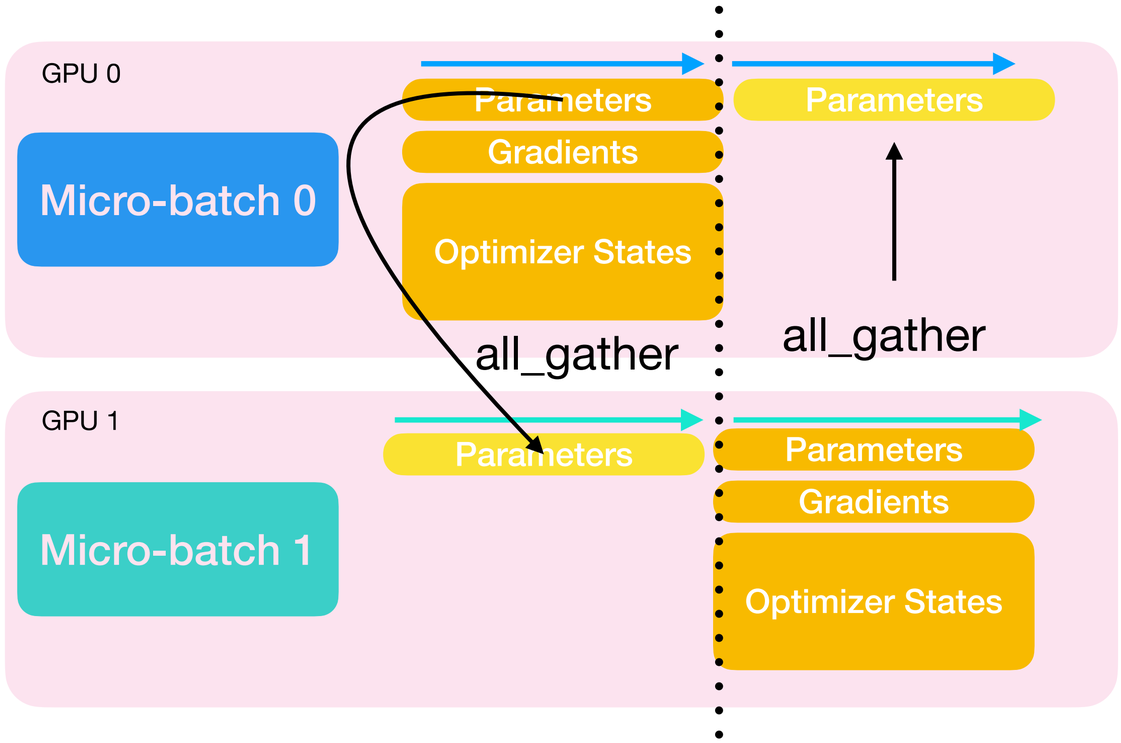
[+] Exactly the same as ZeRO-1, except with lower memory requirement

Available now natively in PyTorch with `torch.distributed.fsdp`

We will try this out in our exercise today!

ZeRO-3 Optimizer + Gradients + Params

Also known as FSDP (fully-sharded data parallel)



No redundancy, everything is sharded

Memory (per GPU) =

$$\frac{2.P + 2.P + 12.P}{\text{num\_gpus}}$$

However, now we need additional comms during the forward + backward pass

reduce_scatter on gradients is same as ZeRO-2

Additionally: no need for all_gather on params, since each device has only its own copy of params

ZeRO-3 Optimizer + Gradients + Params

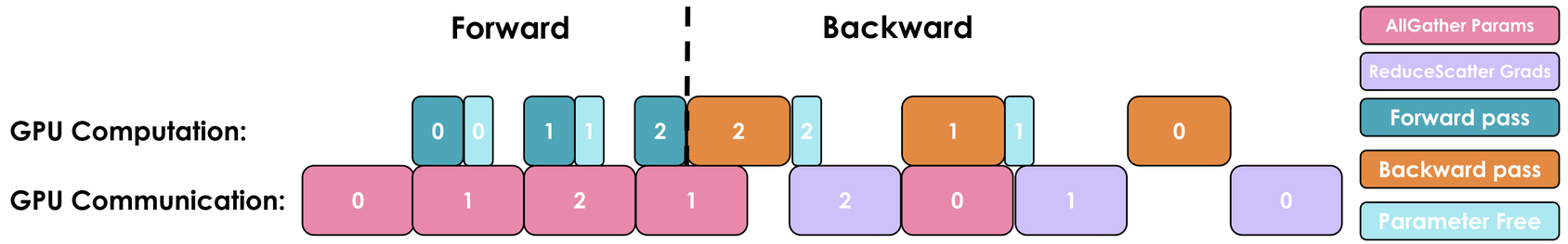
Aren’t these a lot of comms?

Indeed, 1.5x more than in ZeRO-2/1

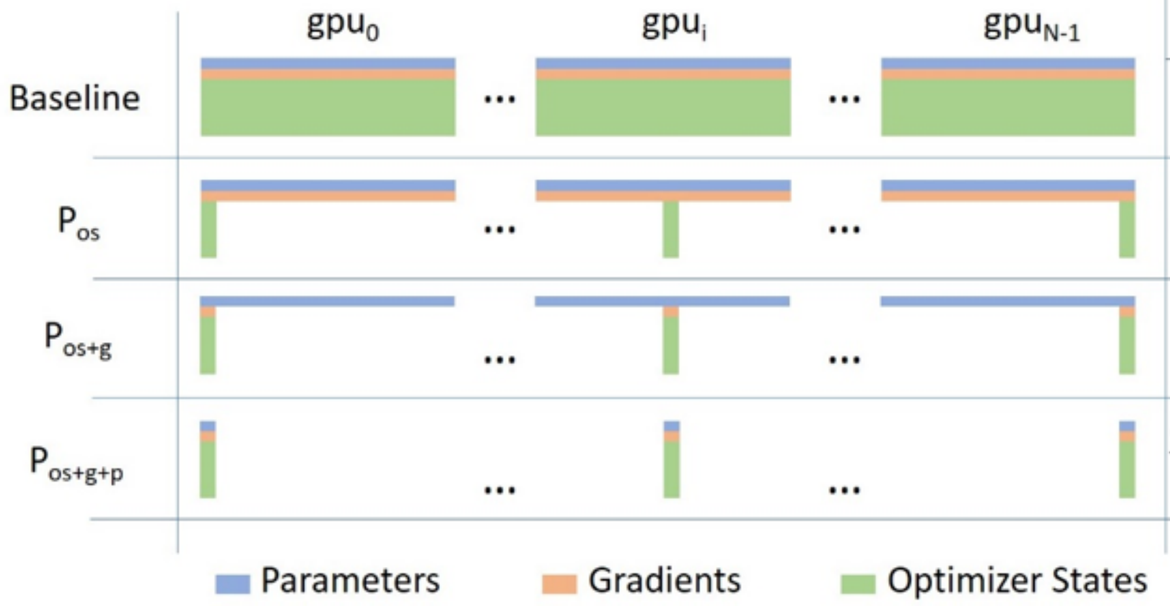
ZeRO-3 — we do P in fwd, P in bwd and P during gradient reduce = 3P

ZeRO-2/1 — P during gradient reduce, P in all_gather = 2P

However, most of it can be hidden in practice.



ZeRO Recap



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models, SC 2020, Rajbhandari, Rasley et al.

Exercise

Run a toy-GPT model from nanoGPT

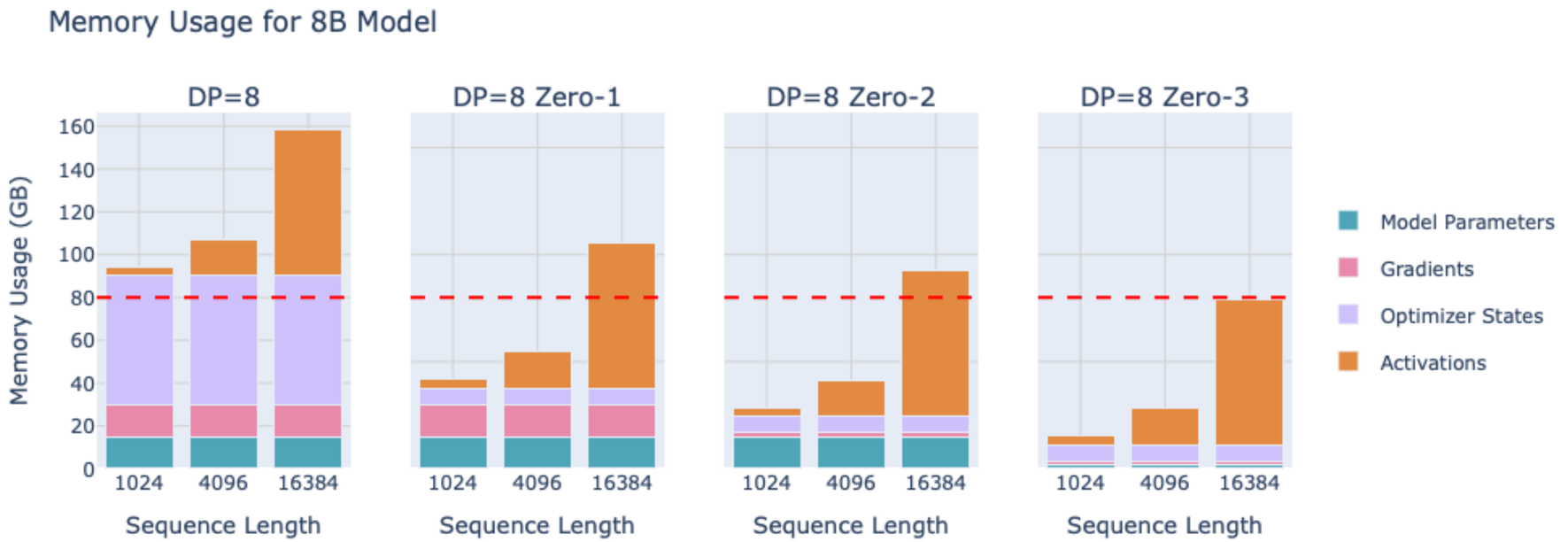
1. on one GPU

2. On 2 GPUs using vanilla data parallelism

3. On 2 GPUs using fully sharded data parallel (refer to PyTorch docs)

Is ZeRO enough?

What about activations?

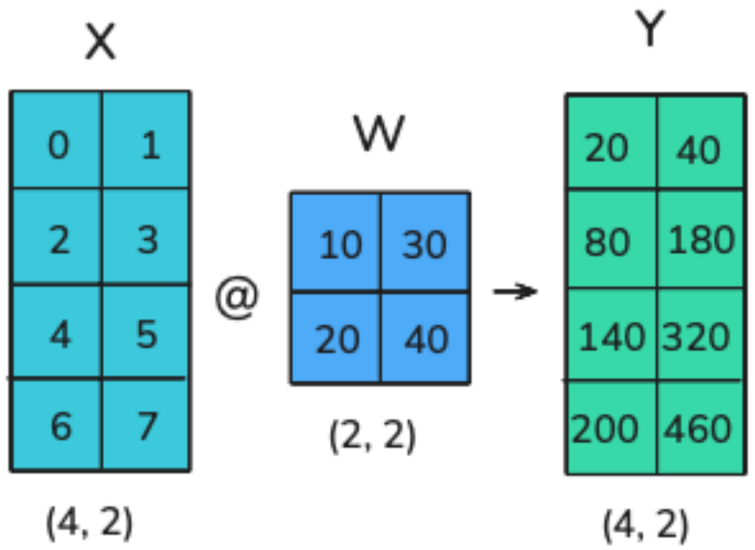


ZeRO requires heavy param comms and does not shard activations. Can we do more?

Tensor Parallelism

So far our axis of parallelism was data, and forward and backward passes for each operation remained the same

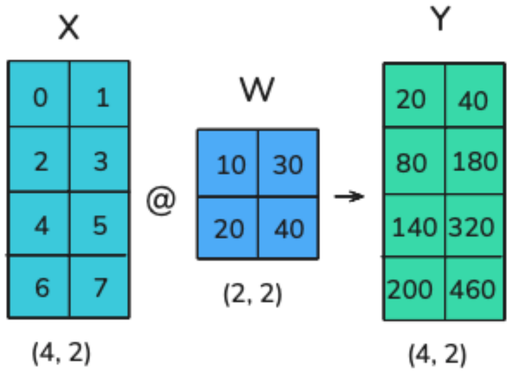
In Tensor Parallelism, we shard the weight tensors such that we also automatically shard activations

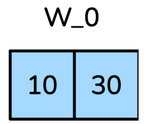


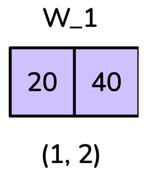
Think of this as a linear layer in a transformer, X is the activation, W is the weight, now let's split this across two devices

Row Parallel Linear

Let’s shard the W matrix by row, i.e., GPU0 gets row 0 and so on



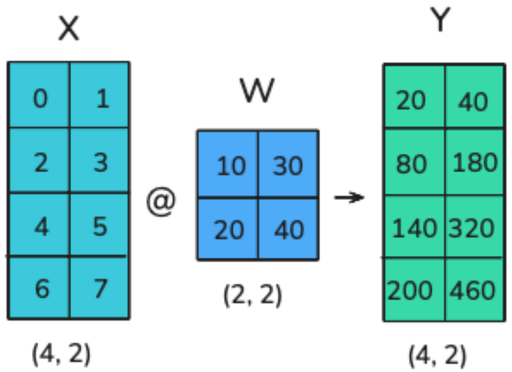


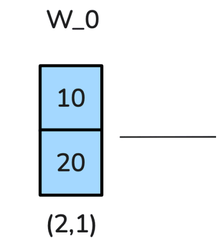


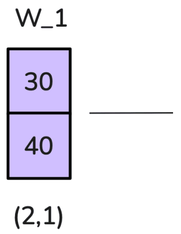
Row linear

Column Parallel Linear

Let’s shard the W matrix by column, i.e., GPU0 gets column 0 and so on



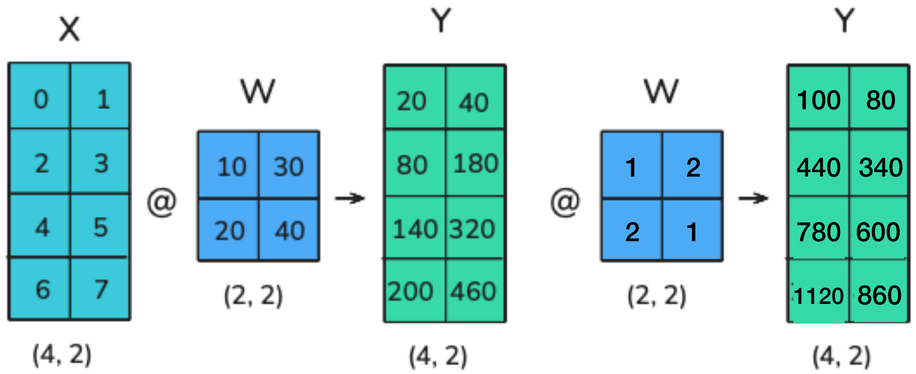




Column linear

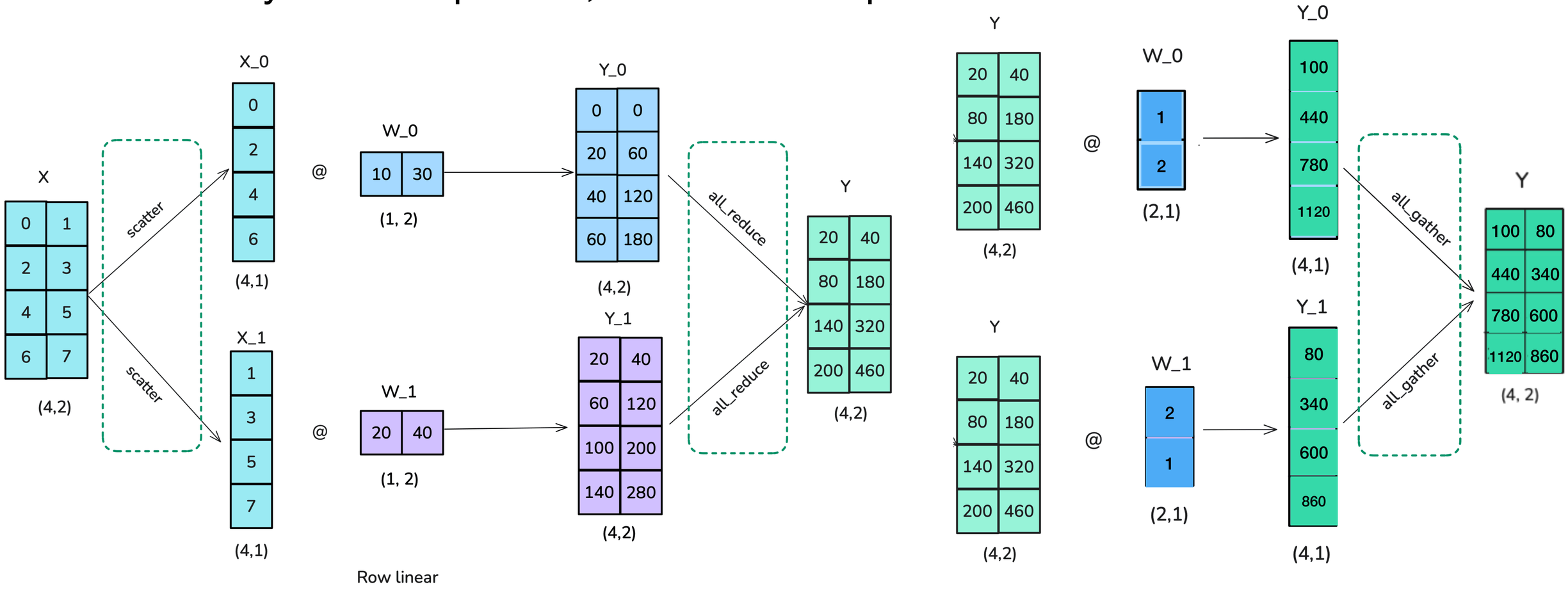
Making the MLP Tensor Parallel

Let’s apply this to the MLP layers — two successive linear layers



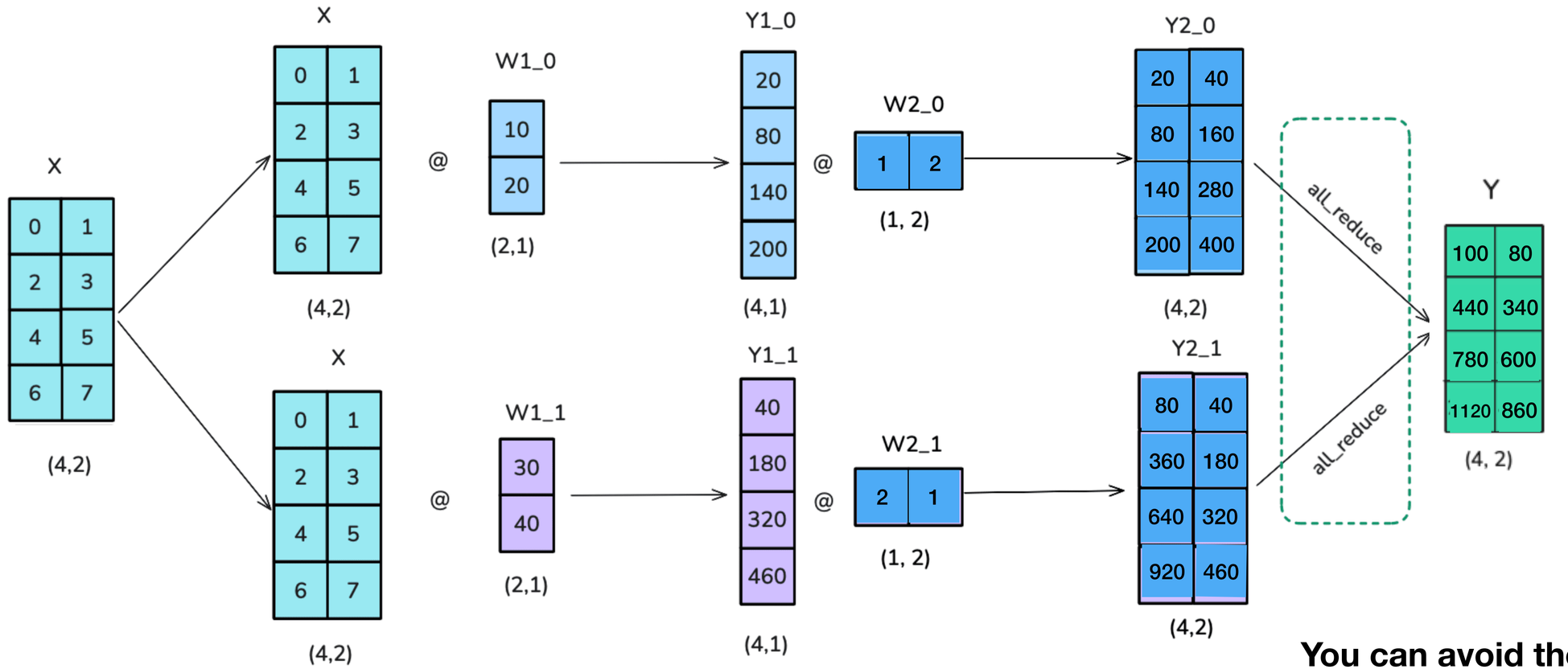
Making the MLP Tensor Parallel

Run first layer in row parallel, second in col parallel



Making the MLP Tensor Parallel

Run first layer in col parallel, second in row parallel

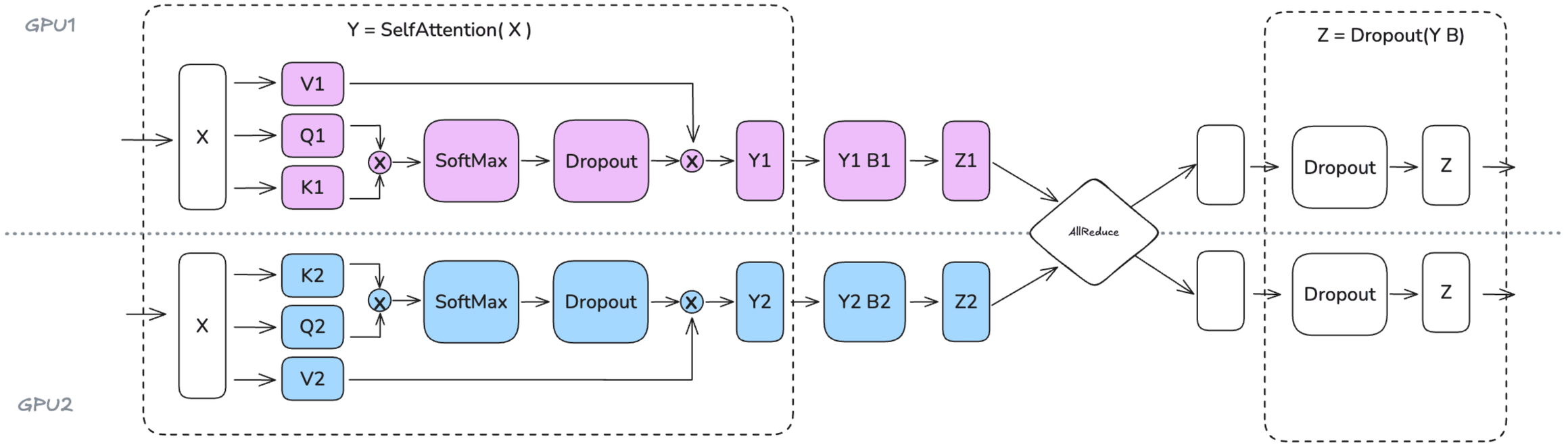


Tensor parallelism with column linear + row Linear

You can avoid the intermediate comms!

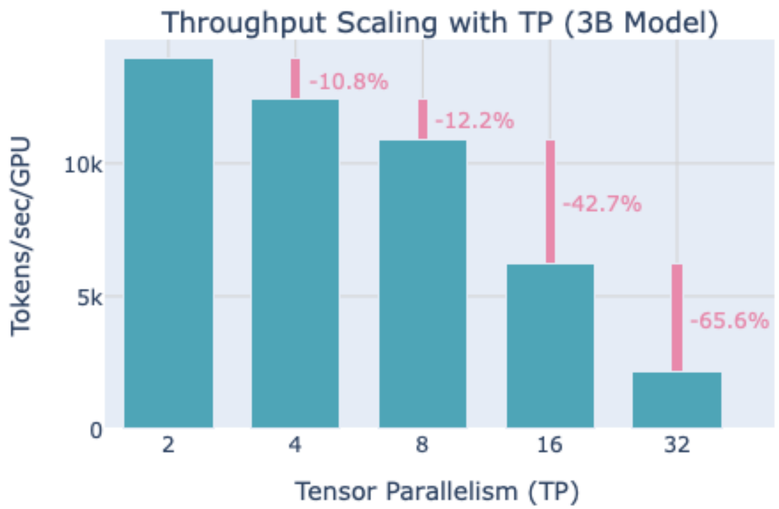
Making Self Attention Tensor Parallel

Much simpler, shard Q, K, V and output projections along the head dimension

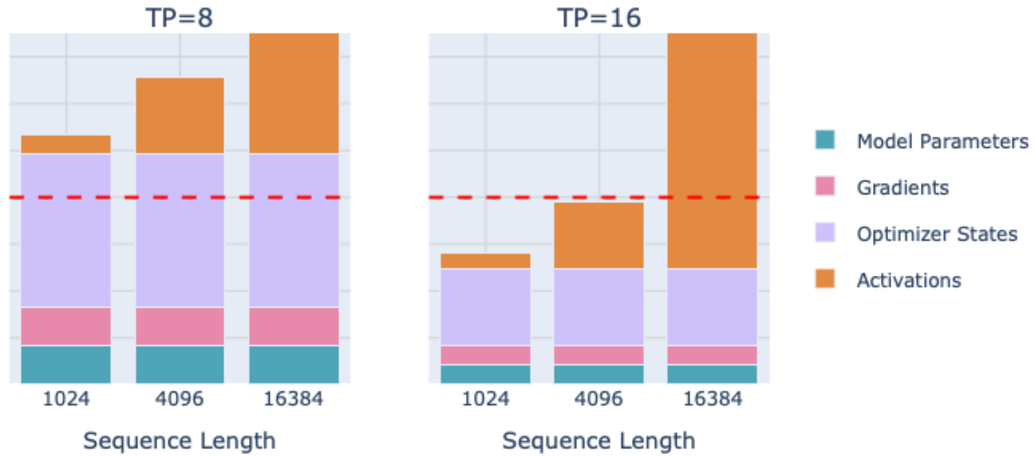


However

We have now introduced many comms in the forward (and backward) of our layers, this quickly becomes an issue when going beyond one node



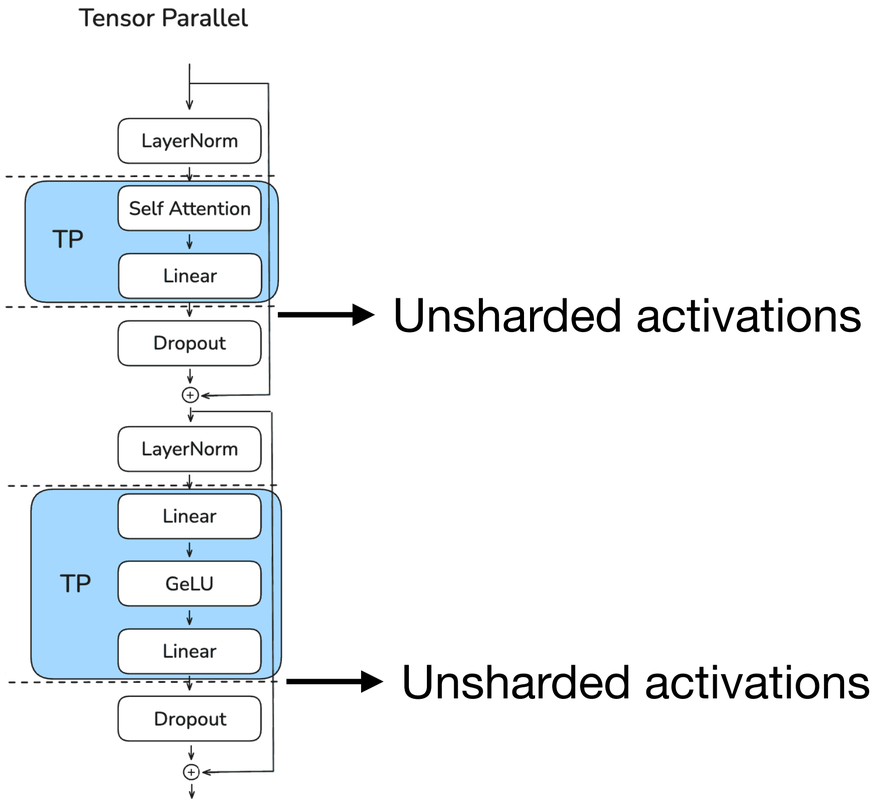
Quick fall in throughput as we go from 8 to 16 GPUs



We do successfully reduce memory

Further Reducing Activation Footprint

After self attention and MLP blocks we have ops like dropout and layernorm, these require unsharded activations



Unsharded activations

Unsharded activations

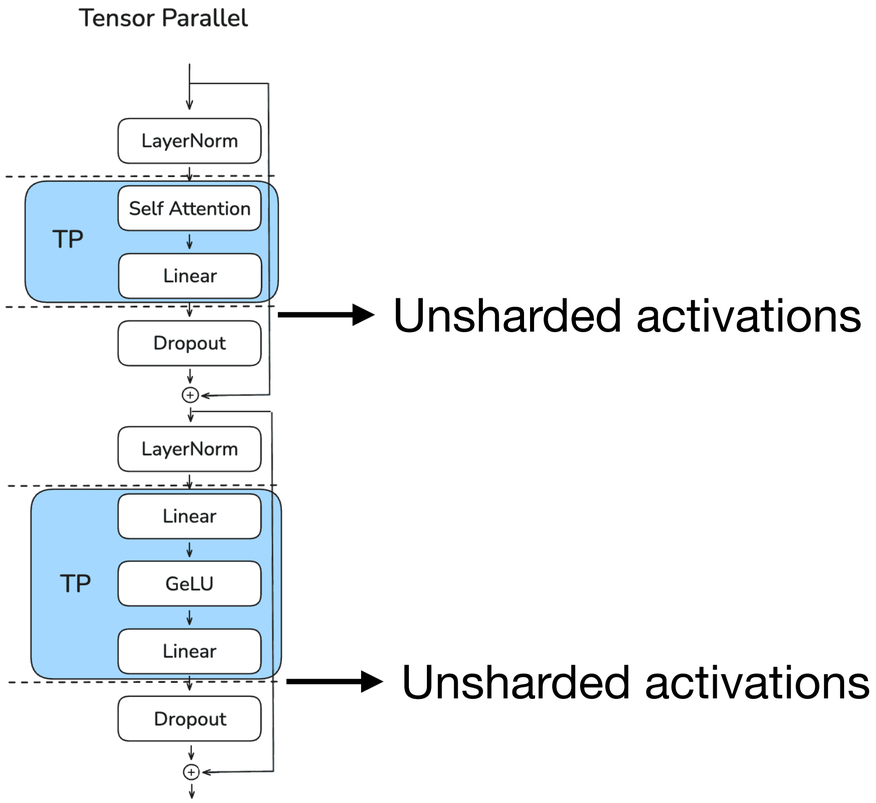
$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Require the entire hidden dimension

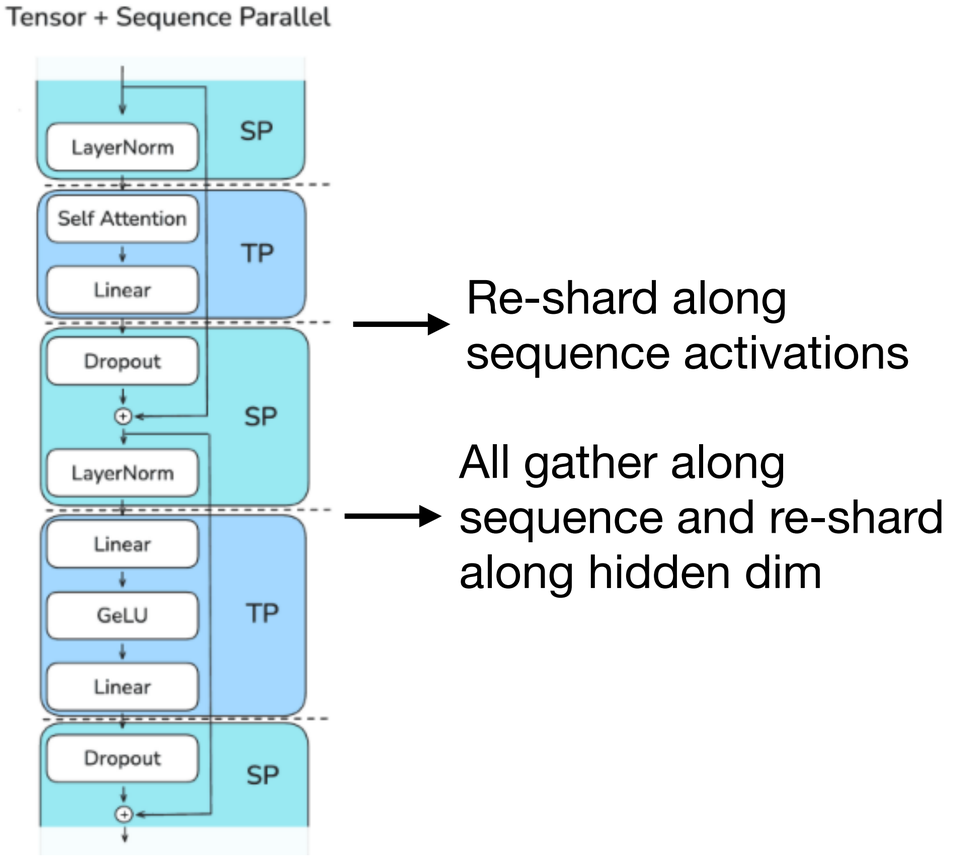
Tensor + Sequence Parallel

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

Require the entire hidden dimension

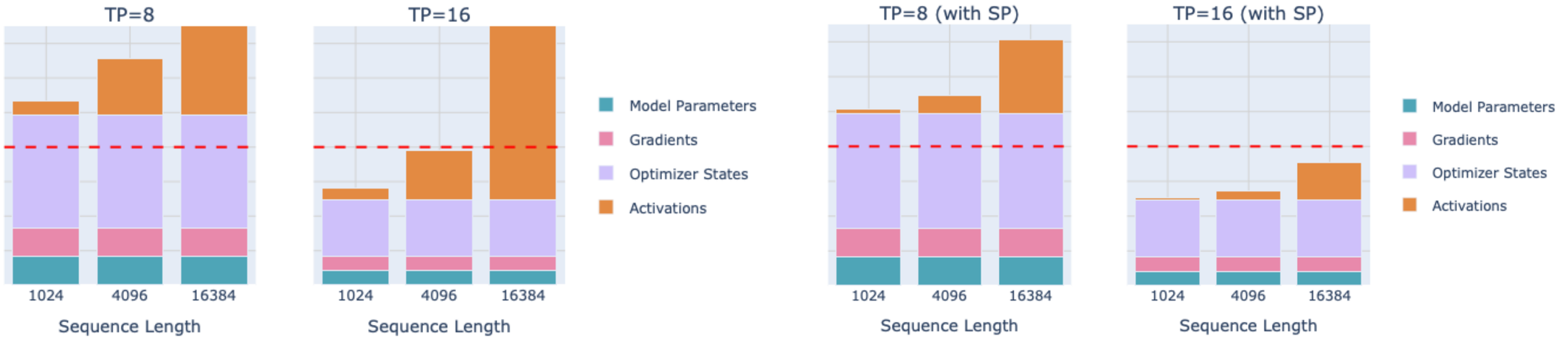


However, LayerNorm and Dropout both don't require the full sequence, so right after the TP region, we can easily shard across the sequence dimension



Tensor + Sequence Parallel

At the cost of more comms, this reduces activation memory since we never keep full set of activation on a single device



Key issue with both DP and TP

For very large sequence lengths, computing attention can become the bottleneck and activations can very large in the sequence dimension ==> **context parallelism**

Across nodes, both DP and TP become very inefficient due to slower comms across nodes ==> **pipeline parallelism**

Pipeline parallel

Deserves a full lecture — needed mostly for massive models, on massive clusters

See references at the end of the lecture for some reading material

Was it wise to pipeline? As we now know pipelining is not wise.

- Ilya Sutskever, NeurIPS 2024

Context Parallelism

Why can’t we simply run parallelism along the sequence dimension like data parallel, i.e., each device gets a portion of the sequence?

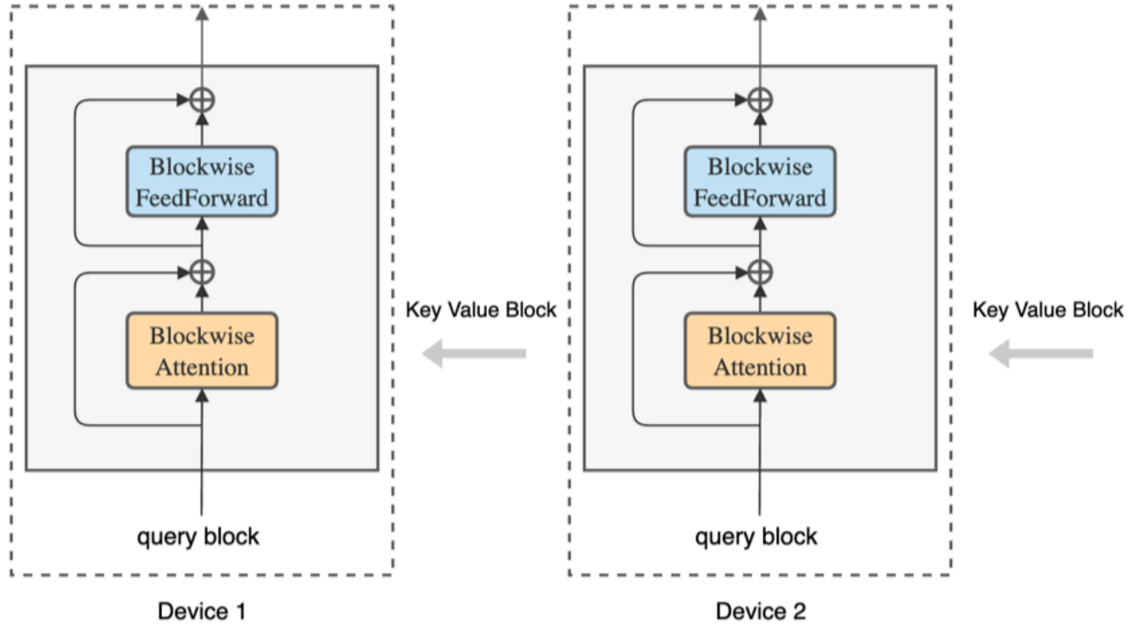
There is one key layer in a transformer that requires all tokens to interact, i.e., self attention

MLP can run on every token in parallel without any interaction

We can do this with a short modification to the attention algorithm!

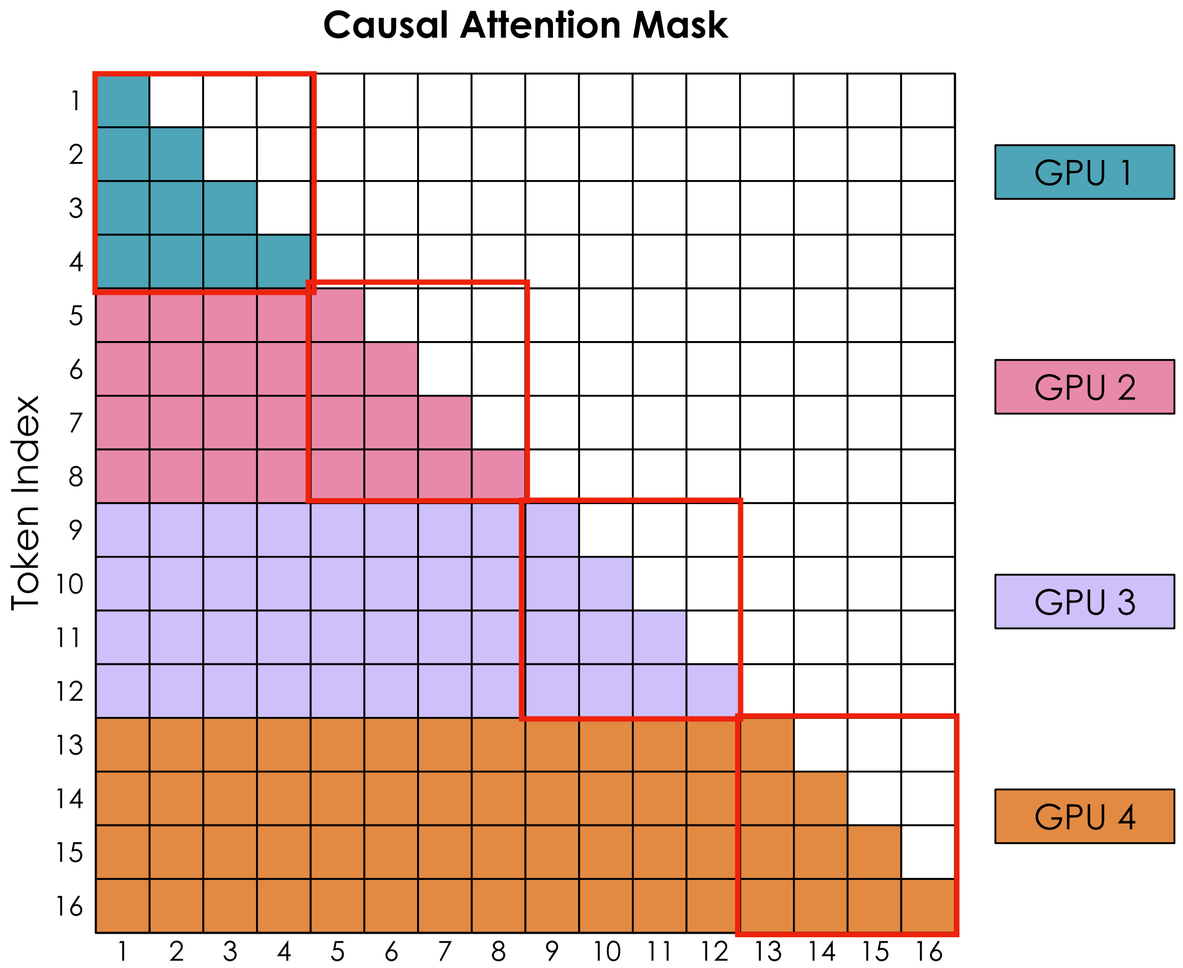
Context Parallelism — Ring Attention

To run parallelism along the sequence dimension like data parallel, i.e., each device gets a portion of the sequence, we must make sure the attention layer communicates to compute the correct attention scores



Ring Attention with Blockwise Transformers for Near-Infinite Context, Liu et al. 2023

Context Parallelism — Ring Attention

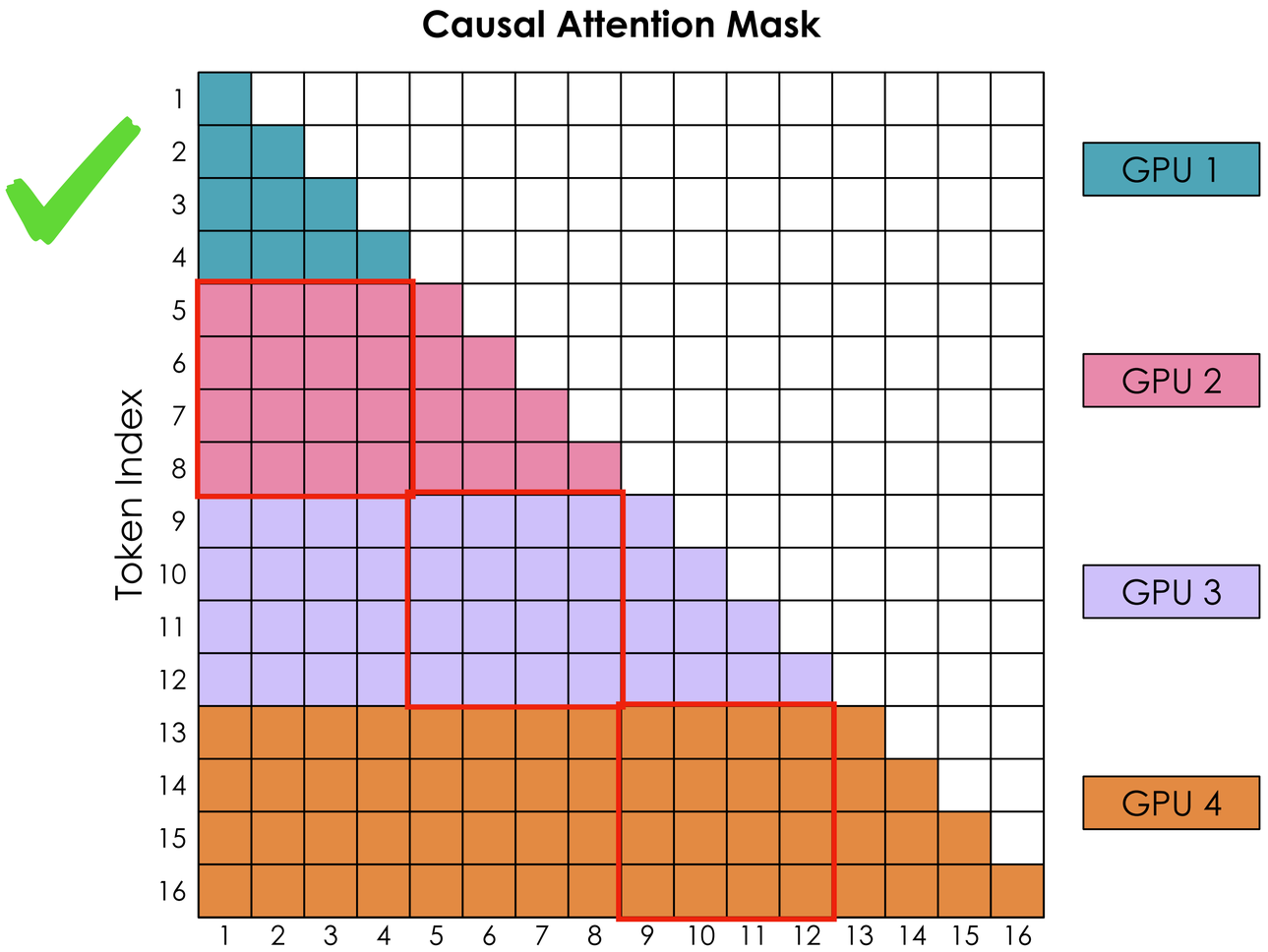


Run attention block in a “ring”

Ring iter 0, each GPU computes its own portion of the causal mask

Once done, communicate keys and values to the next GPU

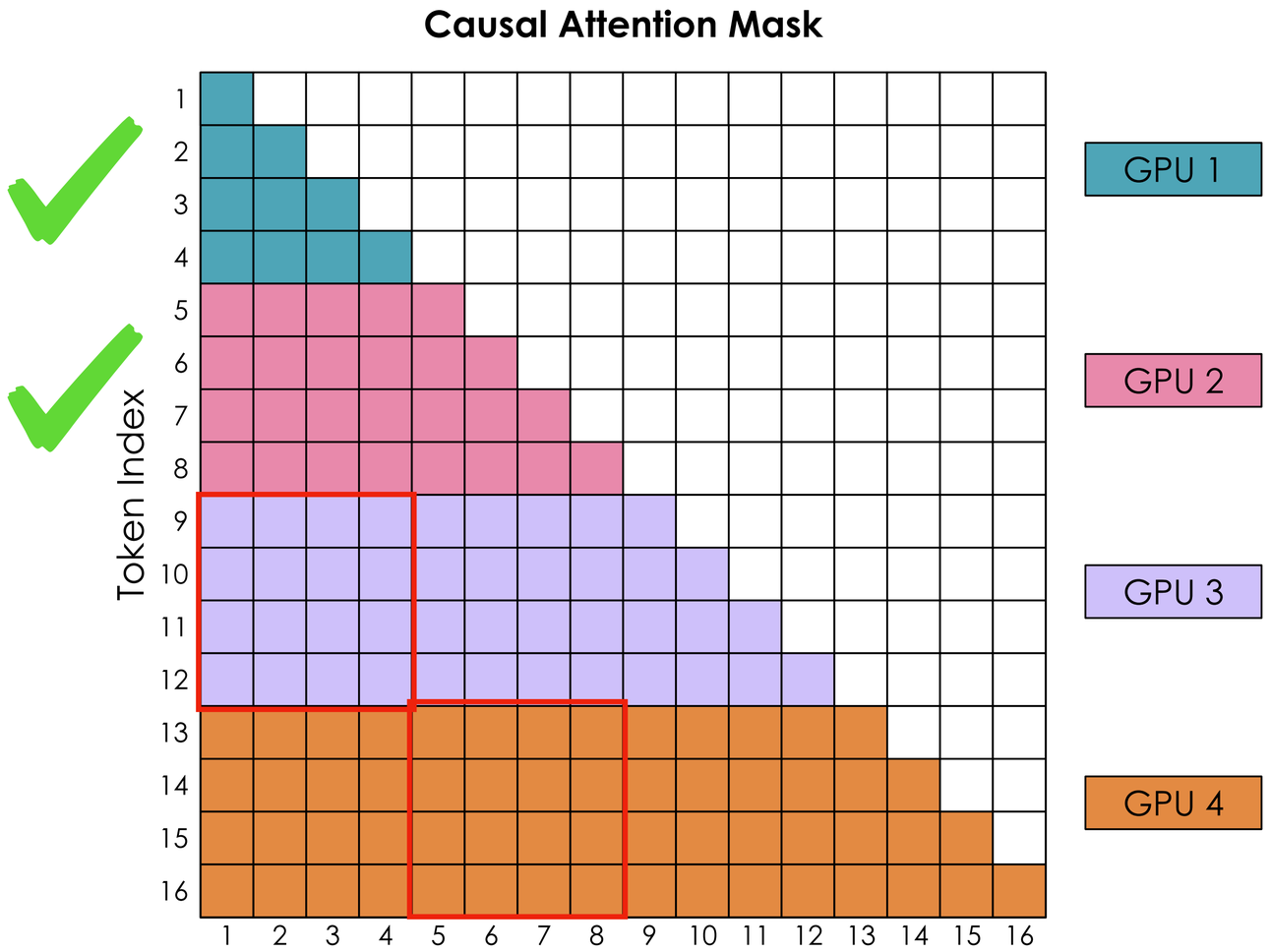
Context Parallelism — Ring Attention



Ring iter 1, each GPU gets keys and values from the previous GPU, and computes attention with it's portion of queries

Once done, communicate keys and values to the next GPU

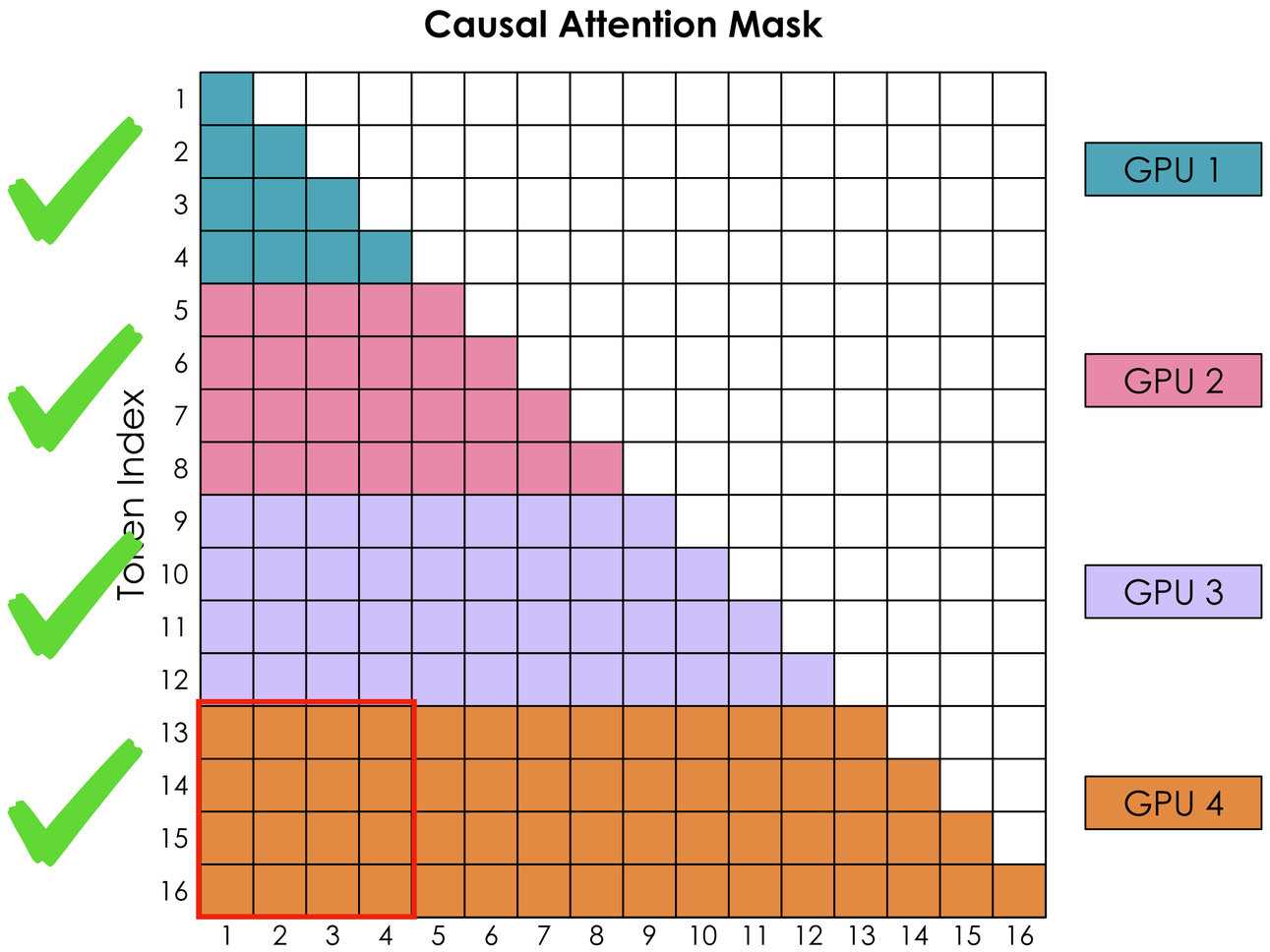
Context Parallelism — Ring Attention



Ring iter 3, each GPU gets keys and values from the previous GPU, and computes attention with it's portion of queries

Once done, communicate keys and values to the next GPU

Context Parallelism — Ring Attention



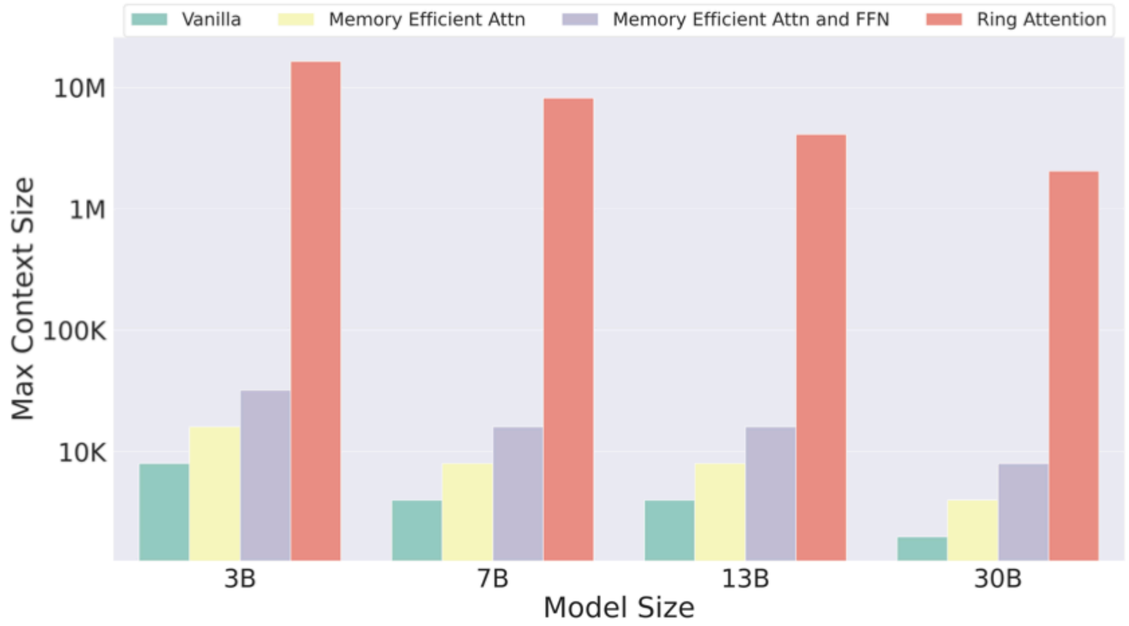
Ring iter 4, each GPU gets keys and values from the previous GPU, and computes attention with it's portion of queries

Now all GPUs are done, ring ends

Context Parallelism — Ring Attention

Introduces comms in self attention, however, MLPs can run without any comms

Allows scaling to much longer sequence lengths than previously possible



Recap

- Multi-GPU Training
  - Data parallelism
  - ZeRO redundancy sharding / Fully-sharded data parallel
  - Tensor Parallel
  - Tensor + Sequence Parallel
  - Context Parallel

Concluding Thoughts

- No shortcut to learn this except trying stuff out
- Sounds easier in theory than in practice
- Many labs including DeepSeek heavily invested their time and resources in getting maximum training throughput, also invented DualPipe to increase their throughput
- Many more interesting optimizations to think about in modern post training with online RL (requires also inference)

Exercise

Implement a Tensor Parallel Linear layer (could be either ColumnParallelLinear or RowParallelLinear) and replace the MLP nn.Linear layers with your implementation

Make your change in model.py of nanoGPT

Take inspiration from fairscale: https://github.com/facebookresearch/fairscale/blob/main/fairscale/nn/model_parallel/layers.py

References

1. *Ultrascale Playbook, HuggingFace, https://huggingface.co/spaces/nanotron/ultrascale-playbook*
2. *How to Scale Your Model, https://jax-ml.github.io/scaling-book*
3. *ZeRO paper: https://arxiv.org/abs/1910.02054*
4. *FSDP Meta Blog: https://engineering.fb.com/2021/07/15/open-source/fsdp/*
5. *Ring Attention: https://arxiv.org/pdf/2310.01889*
6. *[Pipe Parallel] Gpipe: https://arxiv.org/abs/1811.06965*
7. *[Pipe Parallel] PipeDream: https://arxiv.org/abs/1806.03377*
8. *[Pipe Parallel] Zero Bubble Pipeline Parallel https://arxiv.org/abs/2401.10241*
9. *[Pipe Parallel] DeepSeekV3's DualPipe https://github.com/deepseek-ai/DualPipe*