Lecture 7

PARALLELISM BASICS

CS336

Tatsu H

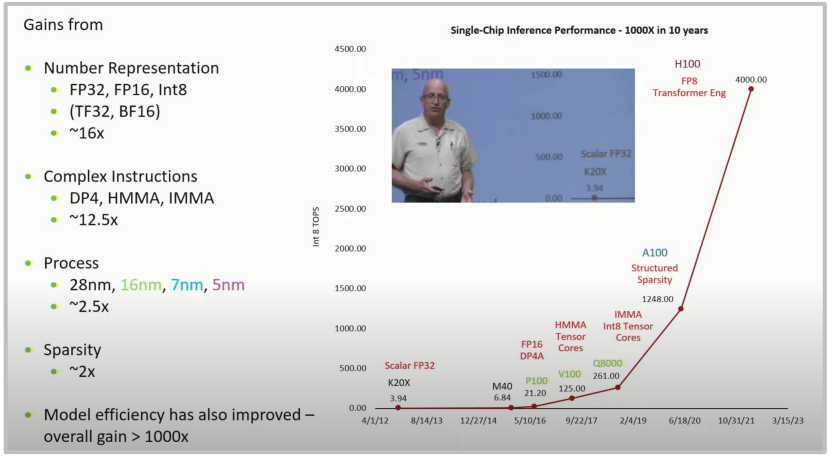
Outline and goals

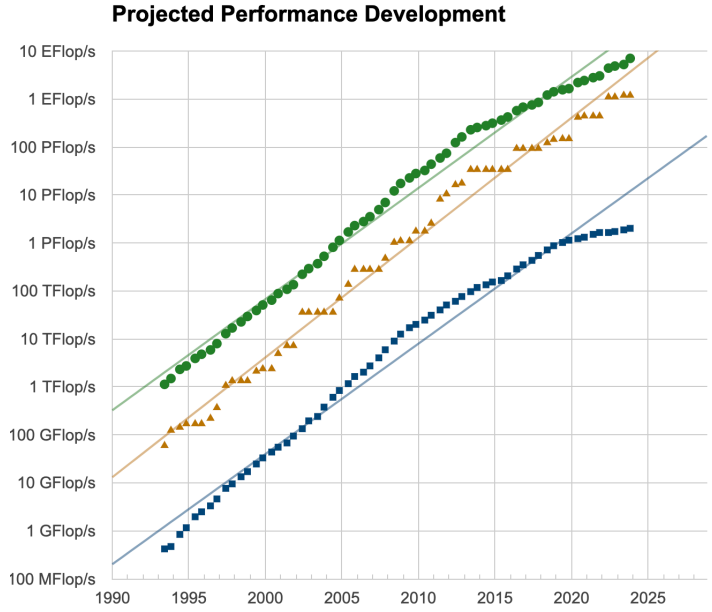
- Understand the systems complexities of training huge models
- Different parallelization paradigms and why people use multiple at once
- What large scale training runs often look like

Organization today:

- **Part 1**: Basics of networking for LLMs
- **Part 2**: Different forms of parallel LLM training
- **Part 3**: Scaling and training big LMs with parallelism

Limits to GPU-based scaling – compute



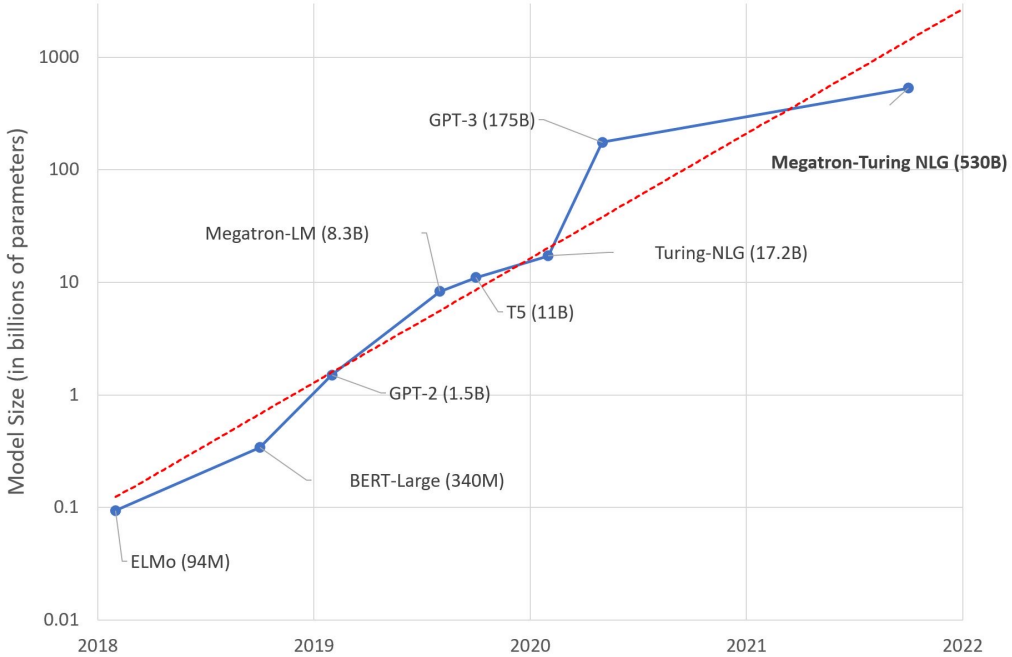


There are limits to single-GPU scaling.

The word's fastest supercomputers have *exaflops* of compute

Limits to GPU-based scaling - memory

Models are getting really big..

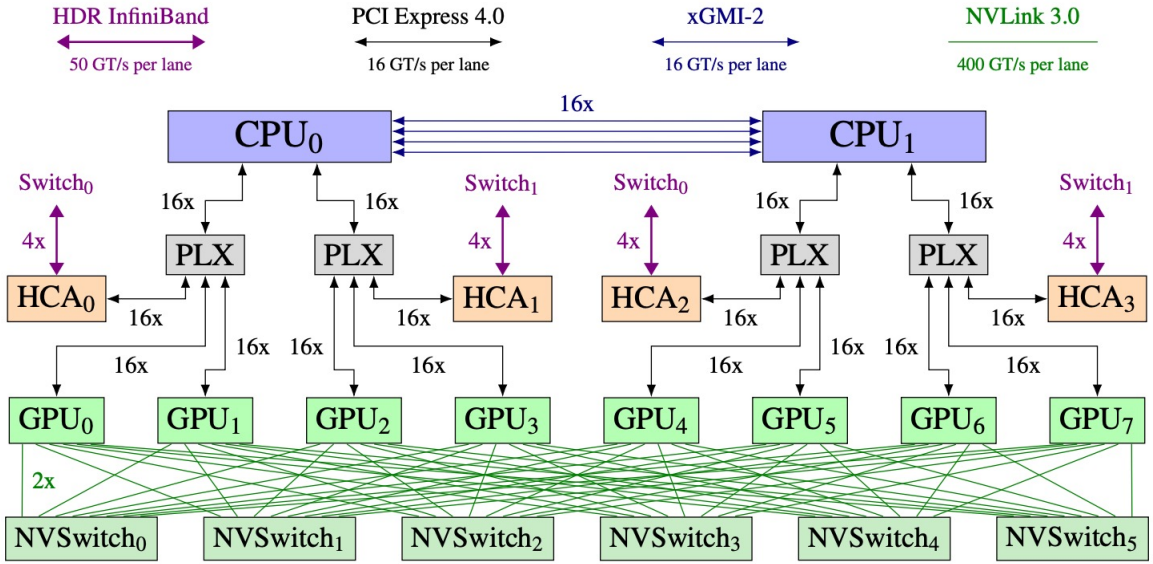


A single GPU can’t fit most of these large models!

What do we do? Multi-GPU, multi-machine parallelism

Intra-node parallelism
via high-speed interconnects

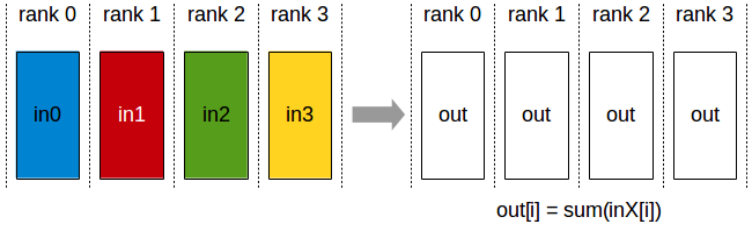
High-speed inter-node parallelism



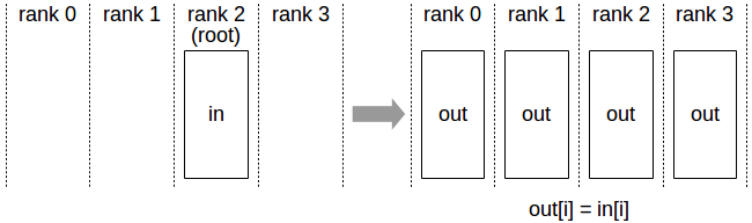
Split up memory and compute requirements across GPUs and machines

But first.. Some basics about collective communication

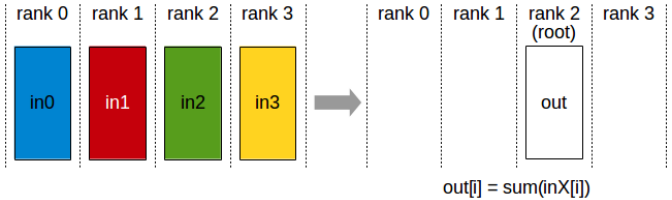
All reduce



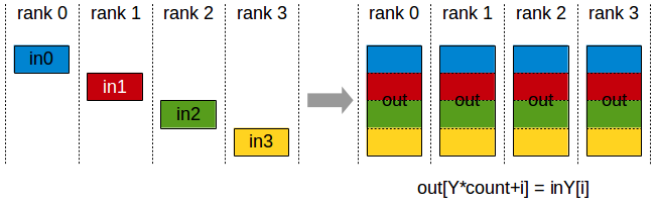
Broadcast



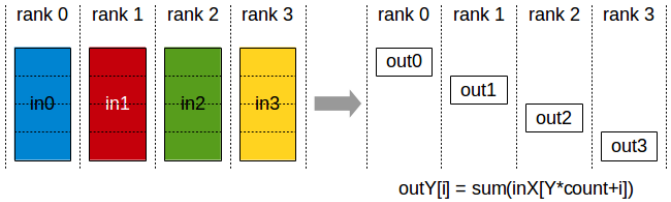
Reduce



All Gather

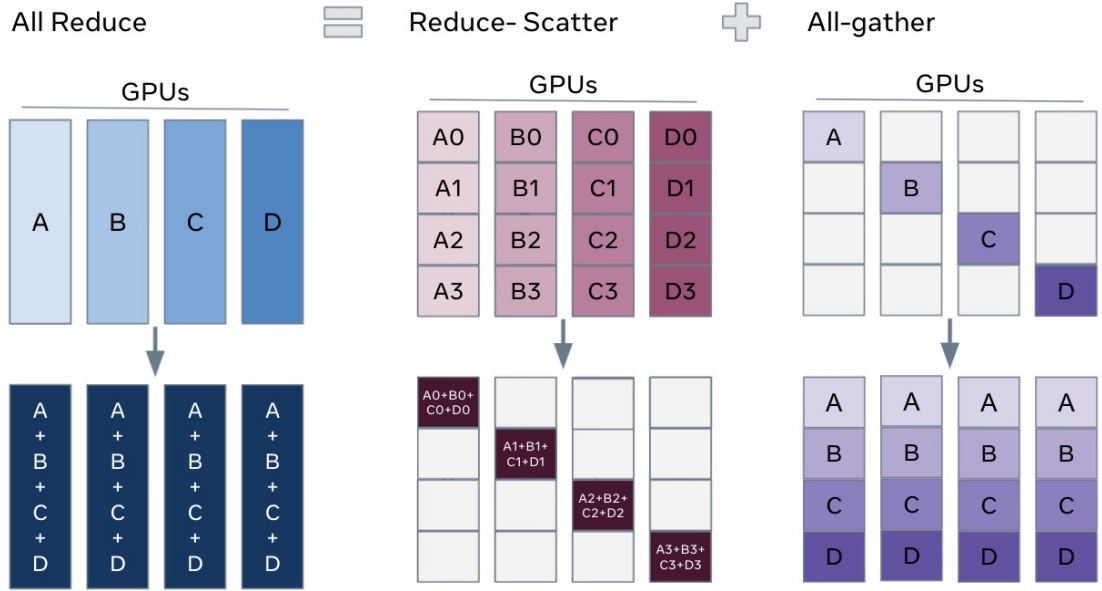


Reduce Scatter



Important detail – all reduce vs reduce-scatter-gather.

Reduce can be implemented as two steps: reduce-scatter and all-gather



Importantly, in the bandwidth-limited regime, this is the best you can do

Part 1 recap

- New unit of compute – the datacenter
- What we want from multi-machine scaling:
  - Linear memory scaling (max model params scales with num gpus)
  - Linear compute scaling (model flops scale linearly with num gpus)
- Simple collective comms primitives

Part 2 – Standard LLM parallelization primitives

How do we parallelize LLMs? 3 important ideas

- Data parallelism
  - Naïve data parallel
  - ZeRO levels 1-3
- Model parallelism
  - Pipeline parallel
  - Tensor parallel
- Activation parallelism
  - Sequence parallel

Naïve data parallelism

Starting point – imagine we are doing naïve SGD

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

Naive parallelism: split the elements of B sized batch across M machines. Exchange gradients to synchronize

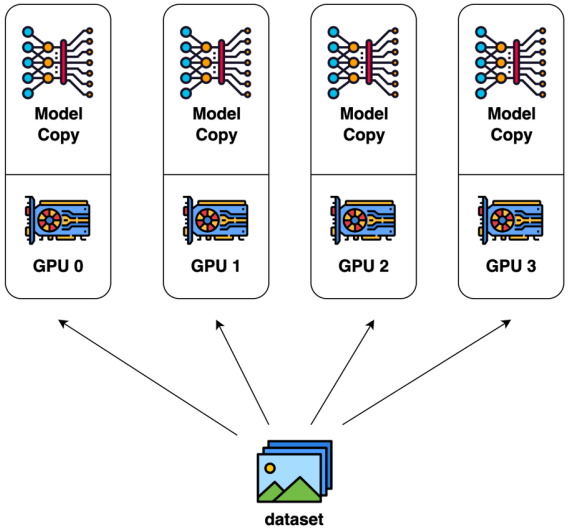
How does this do?

Compute scaling – each GPU gets B/M examples.

Communication overhead – transmits 2x # params every batch. OK if batches are big

Memory scaling – none. Every GPU needs # params at least

What's wrong with naïve data parallel?



Memory seems like it'd be a problem – we copy the model parameters to each GPU.

Let's take a closer look..

What's wrong with naïve data parallelism? - Memory

Our memory situation is actually *terrible*.

Depending on our precision..

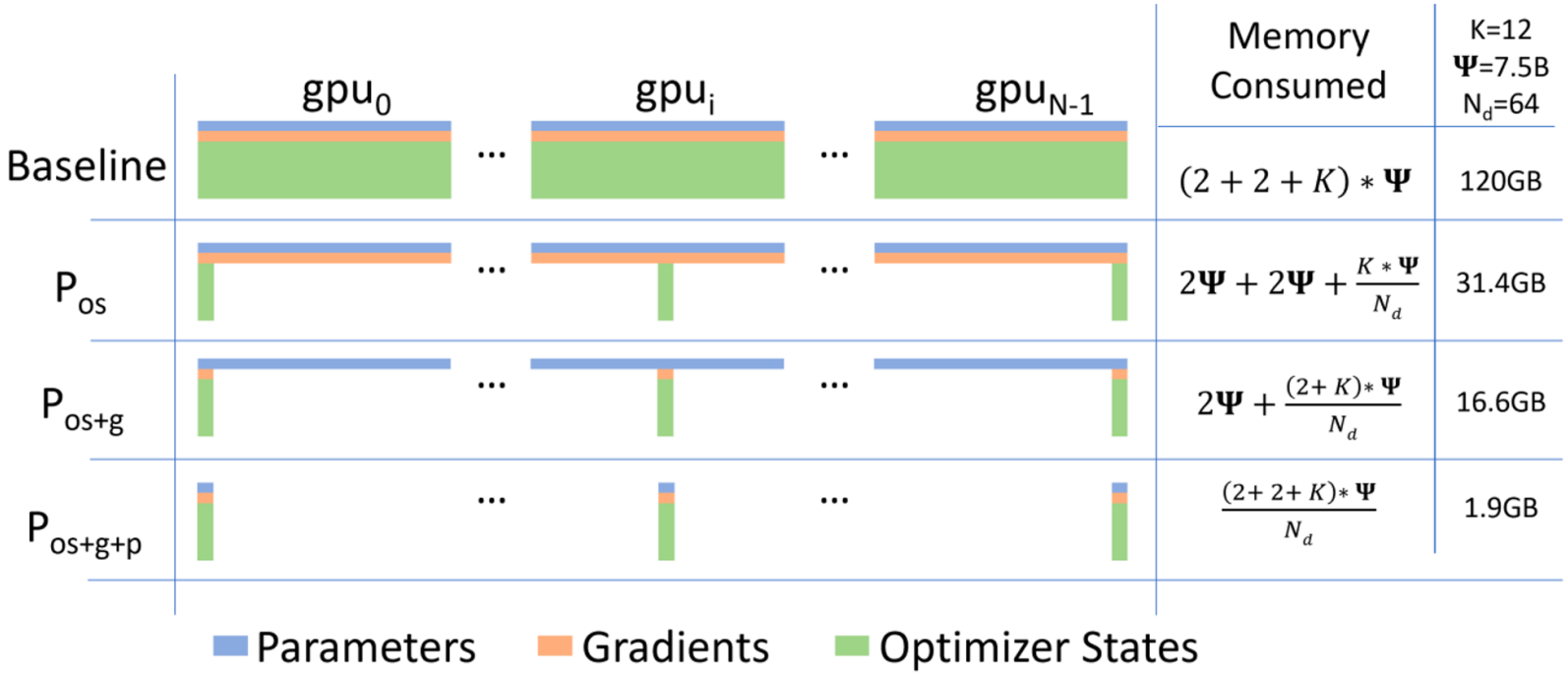
We need 5 copies of weights and 16 bytes per param!

- 2 bytes for FP/BF 16 model parameters
- 2 bytes for FP/BF 16 gradients
- 4 bytes for FP32 master weights (the 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

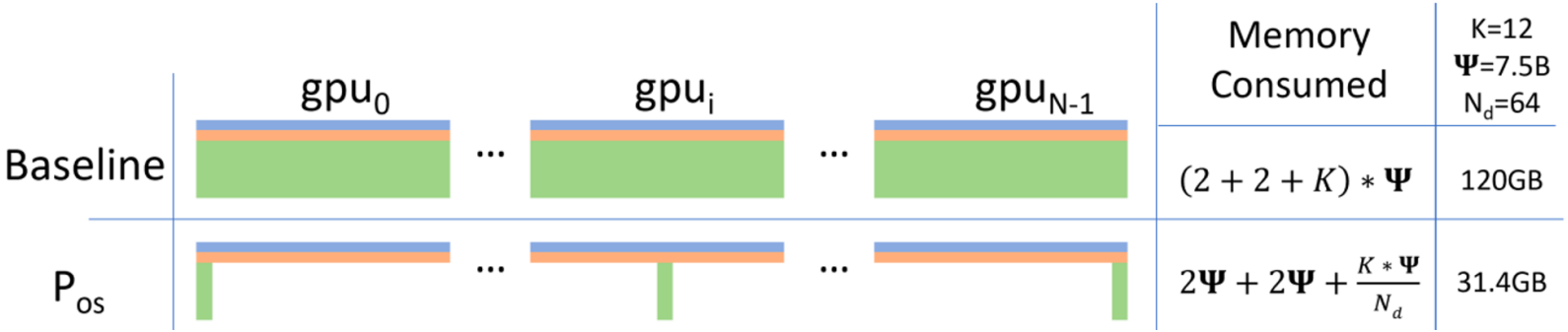
"Optimizer state"

ZeRO – solving the memory overhead issue of DP

Core idea: split up the expensive parts (state) and use the reduce-scatter equivalence.



ZeRO stage 1. optimizer state sharding



High level idea:

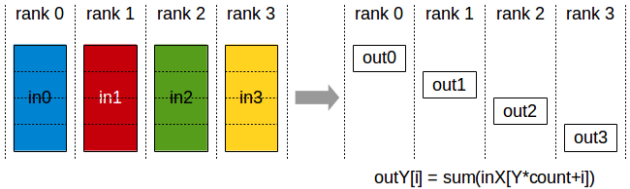
- Split up the optimizer state (first + second moments) across GPUs
- Everyone has the parameters + gradients

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

ZeRO stage 1. how it works

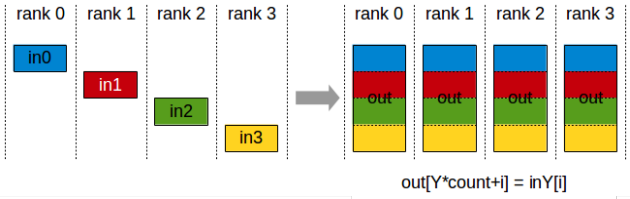
Step 1. Everyone computes a full gradient on their subset of the batch

Step 2. ReduceScatter the gradients – incur #params communication cost



Step 3. Each machine updates their param using their gradient + state.

Step 4. All Gather the parameters – incur #params communication cost



Comparing ZeRO stage 1 and naïve data parallel

	Naïve DDP	ZeRO stage 1
Communication primitive	One all-reduce (gradients)	One reduce scatter (send gradients) + all gather (collect params)
Communication cost	2* # params	2* # params
Memory	(4+K) * #params	(4+K/Ngpu) * #params

ZeRO stage 1 is *free* (in the bandwidth limited regime) memory wins

ZeRO stage 2. the simple extension to gradient sharding

Emboldened by our success, let's shard even more stuff

High level idea

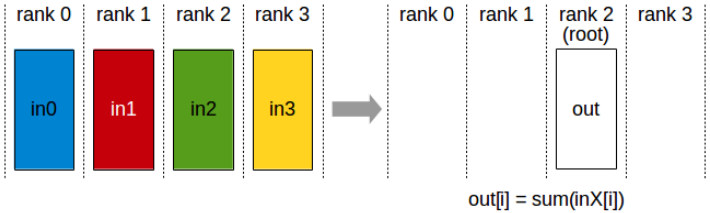
- Also keep the gradients (pink slices) sharded across the machines.
- Use the same (rough) tricks as stage 1.

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

ZeRO stage 2. how it works

Step 1. Everyone incrementally goes backward on the computation graph

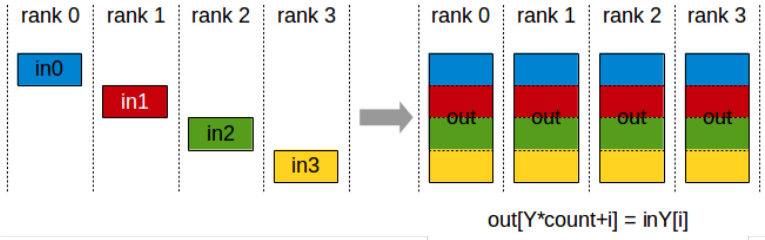
Step 1a. After computing a layer's gradients, immediately reduce to send this to the right worker



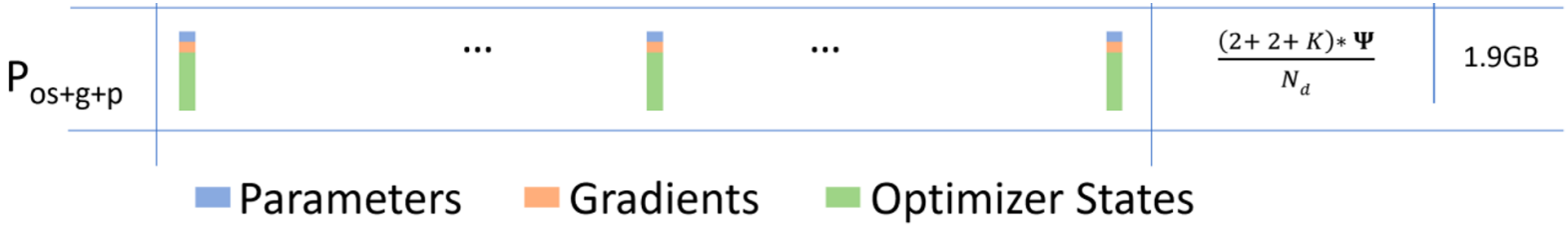
Step 1b. Once gradients are not needed in the backward graph, immediately free it.

Step 2. Each machine updates their param using their gradient + state.

Step 3. All Gather the parameters.



ZeRO stage 3 (aka FSDP) shard *everything*



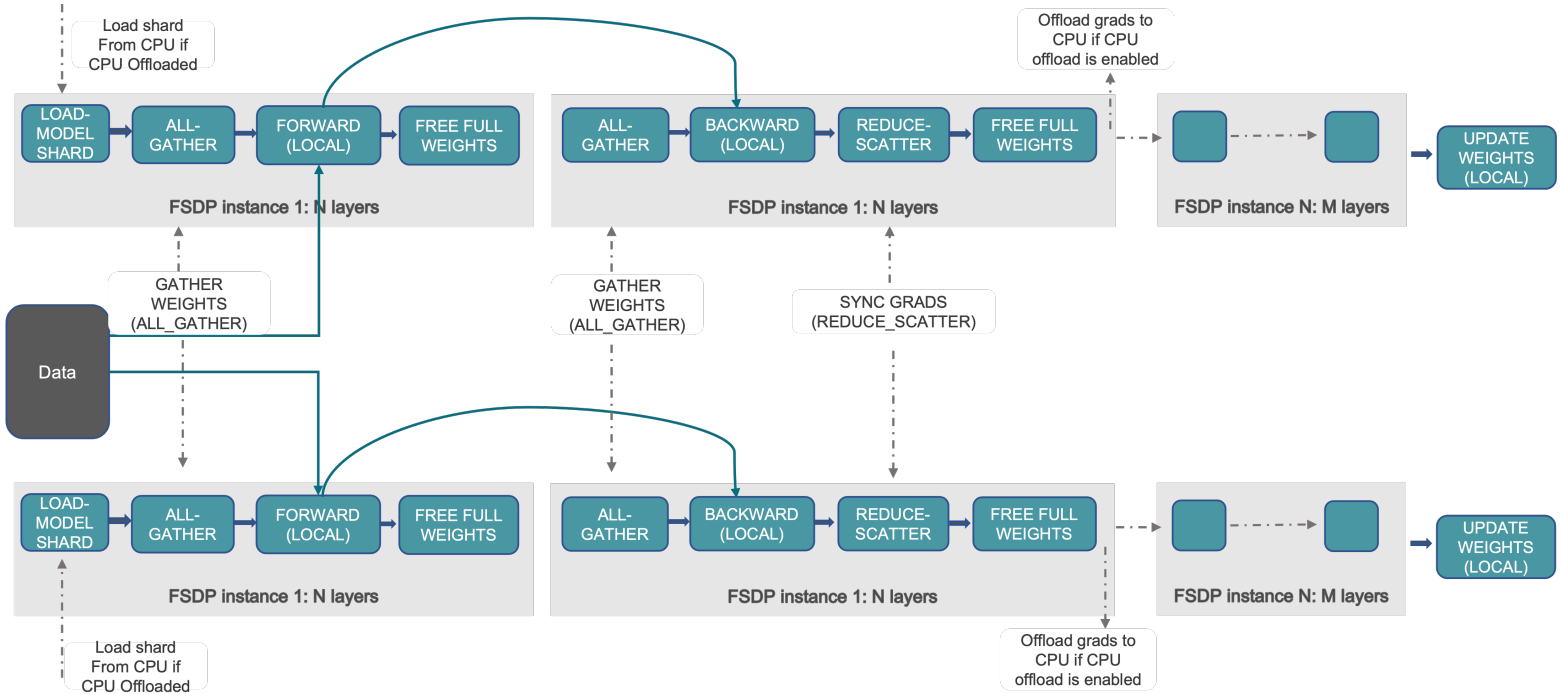
We've gotten almost everything for free so far.. lets try to solve *all* our memory issues

High level idea

- Shard everything – incl parameters!
- Use the same 'incremental communication / computation' ideas
- Send and request parameters on demand while stepping through the compute graph.

Is it possible to do this with low overhead?

ZeRO stage 3 (aka FSDP) how it works (baby version)



Communication cost – 2 all gather (#param), 1 reduce-scatter (#param).

https://pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Actual picture of how FDSP / ZeRO stage 3 works

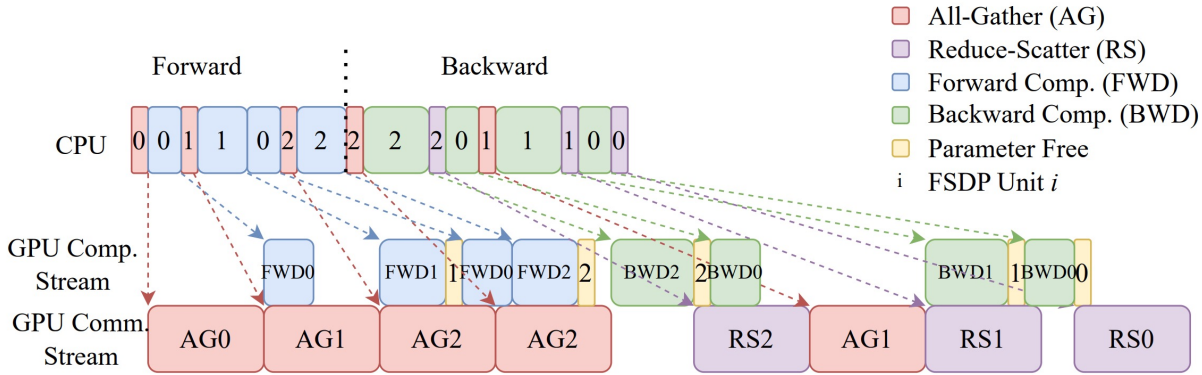
Let's walk through a FSDP example to see some important ideas

Incremental computation / communication

- Parameters / gradients are requested / sent and then immediately freed

Overlapping communication and computation

- The all-gathers happen all at once while forward happens, masking the comm cost.



https://arxiv.org/pdf/2304.11277.pdf

What's the point?

Distributed data parallel costs 2*# param communication

What about ZeRO?

- **Zero stage 1** is 2*# param – it's free! – you might as well always do it
- **Zero stage 2** is 2*# param – it's (almost) free (ignoring overhead)
- **Zero stage 3** is 3*# param – 1.5x comm cost, but that's not bad! (ignoring latency..)

This is also conceptually very simple – write a FSDP block wrapper.

ZeRO in practice: will it fit?

In 2024 – pure BF16 training (with Kahan summation), is viable optimizer states are less beefy. Let's say BF16 for everything but the master weights – 12 bytes per param

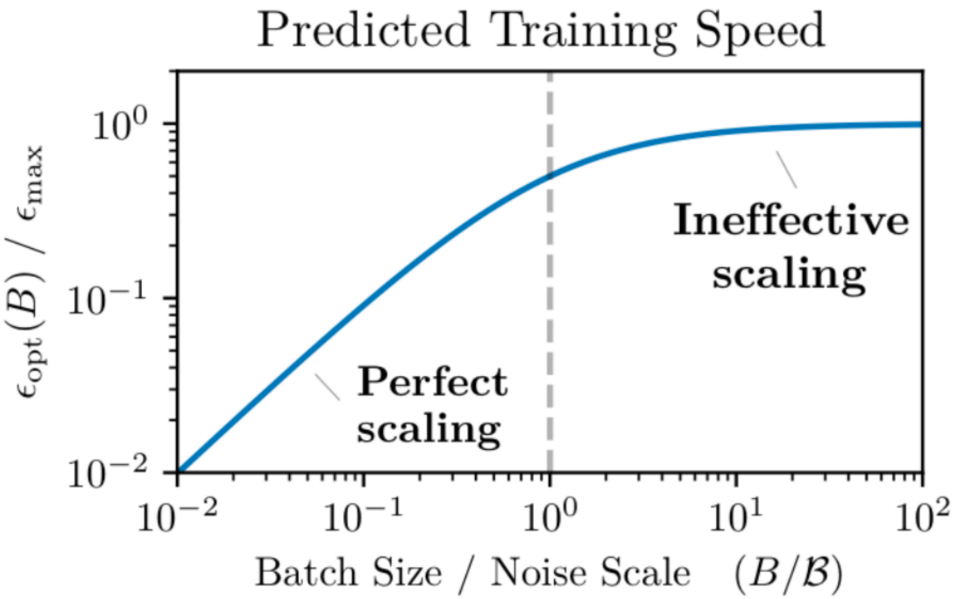
On a 8X A100 80G..

	Max size (params)	Formula for B/param
Baseline	6.66.. B	12
Zero stage 1	16 B	5
Zero stage 2	24.62 B	2 (param) + (10 (grad+state))/8)
Zero stage 3	53.33 B	12/8

Issues remain with data parallel – compute scaling

With data parallel**, #machines < batch size** (and near this, comm overhead is high)

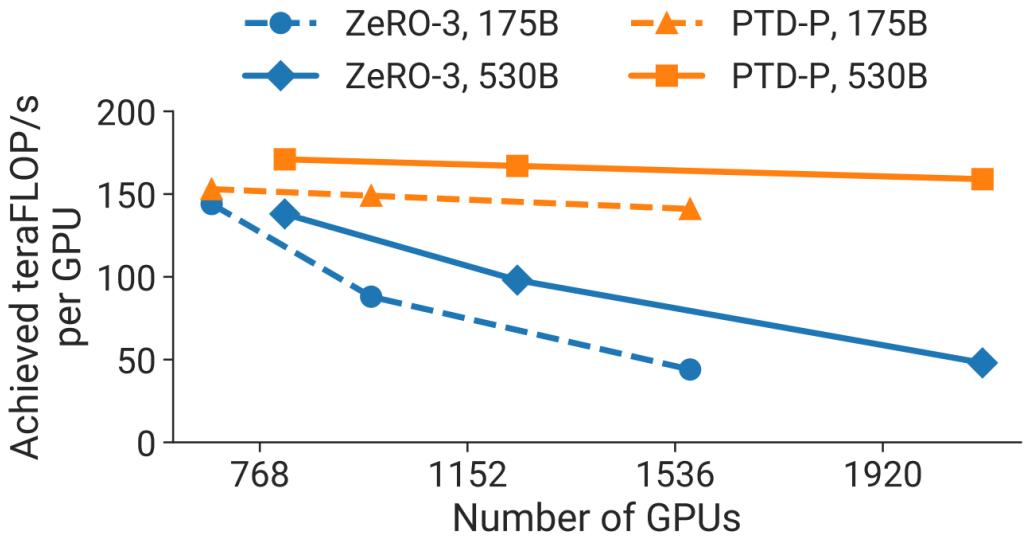
And there's diminishing returns to batch sizes



Issues remain with data parallel – models don't fit

Zero stages 1 and 2 don't let you scale memory

Zero stage 3 is nice in principle, but slow in practice



Better ways to split up the model is needed…

Beyond data parallel – model parallelism

Scaling up in memory (without changing batch size) with model parallelism

What model parallelism is..

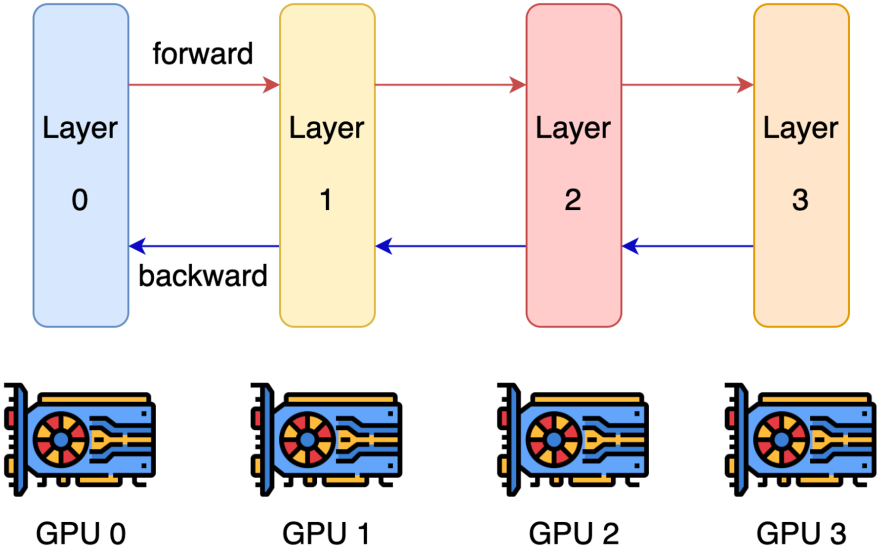
- It splits up the parameters across GPUs (like zero3)..
- But communicate activations (while zero3 sends params).

We cover two different types of parallelism

1. Pipeline parallel
2. Tensor parallel (+ Sequence parallel)

These correspond to two different ways of cutting up the model.

Layer-wise parallel



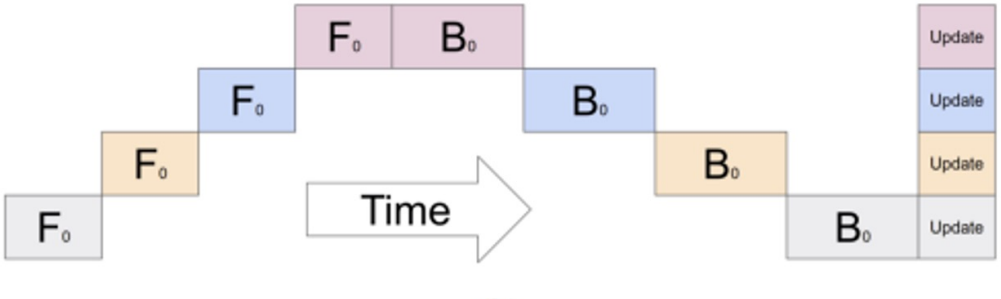
Layer-wise parallel cuts up layers, assigns some subset to GPUS.

Activations and partial gradients are passed back and forth

What's wrong with layer-wise parallel

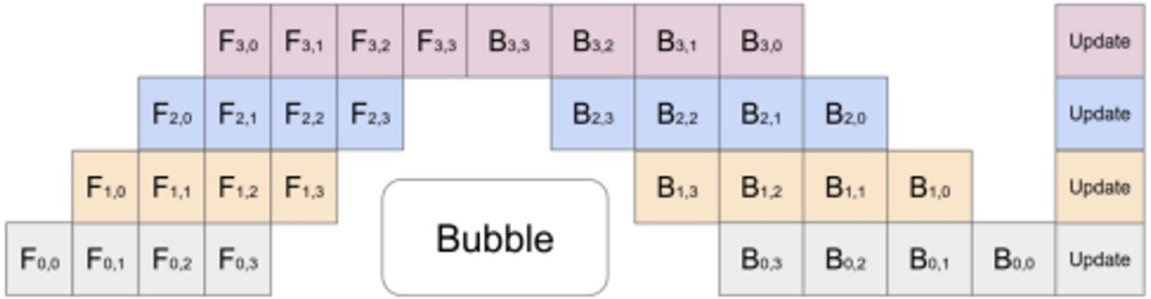
Utilization of layer-wise parallelism is *terrible*..

With n gpus, each gpu is active $\frac{1}{n}$ of the time.



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

A solution: pipeline parallel



Solution: Pipeline-parallel.

Process ‘micro-batches’ (in this case, 4).

Send off the first microbatch and start computing the second.

The ratio of bubble time to useful compute is .. $\frac{n_{stages}-1}{n_{micro}}$ so we need a big batch size!

Why pipeline parallel?

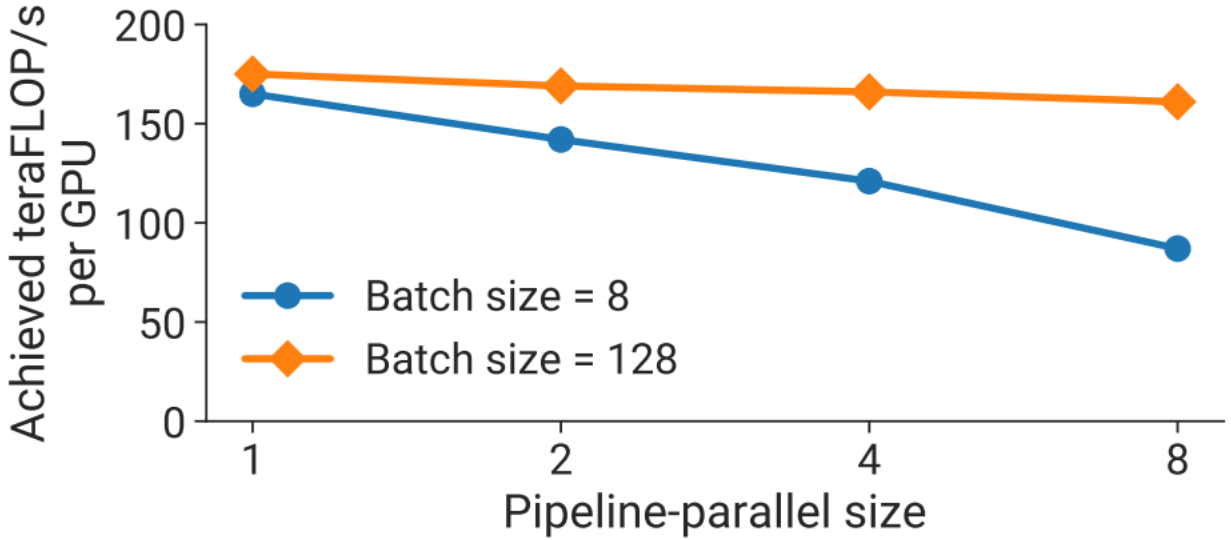
Pipelines seem terrible. Why do we do it?

1. Pipelines save memory (compared to DDP)

2. Pipelines can have good communication properties (compared to FDSP) – it depends only on activations ($b \times s \times h$) and is *point to point*

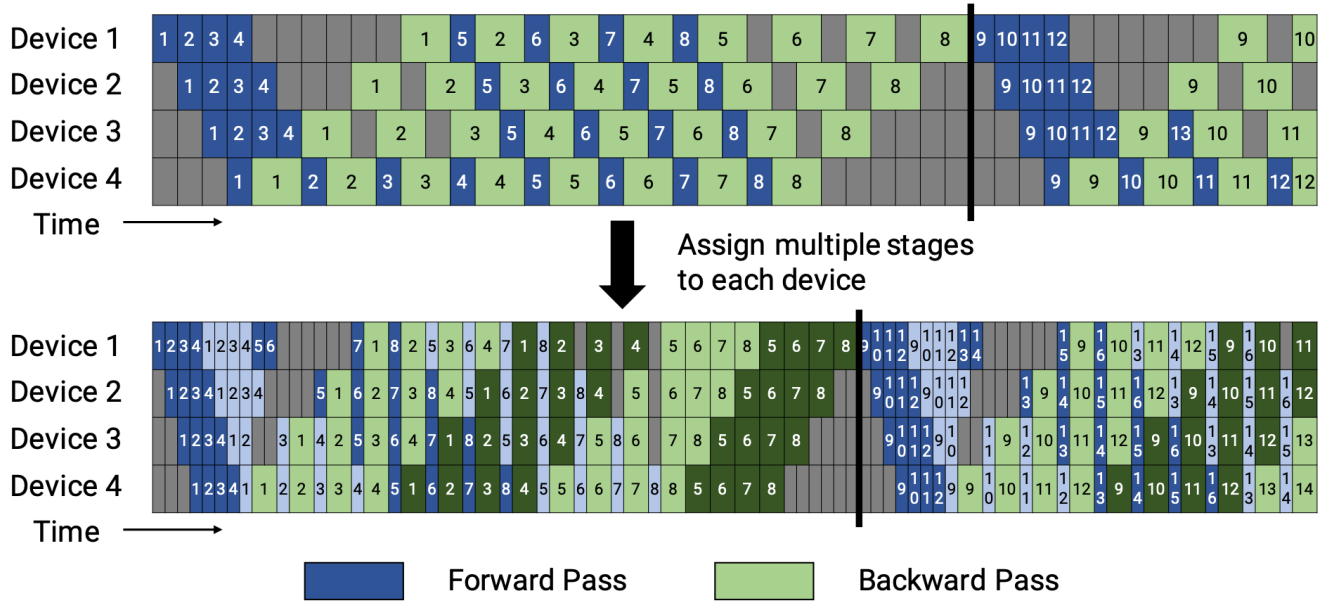
Generally, we will use pipelines on slower network links (i.e. inter-node) as a way to get better memory-wise scaling.

Pipeline performance is highly dependent on batch size



Batch sizes are key to hiding the bubble – otherwise pipeline rapidly degrades perf

Trading communication bandwidth for utilization

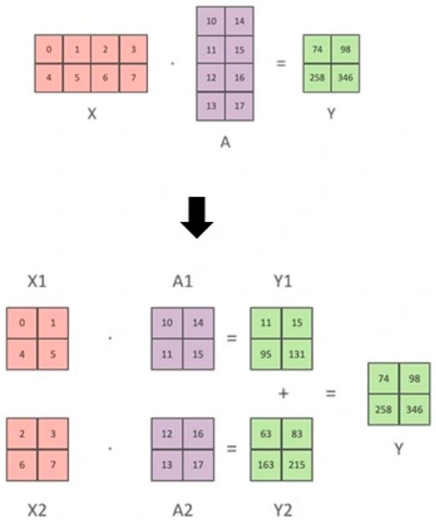


Some more crazy pipeline patterns can improve utilization, but at the cost of bandwidth

Model parallel along the width axes

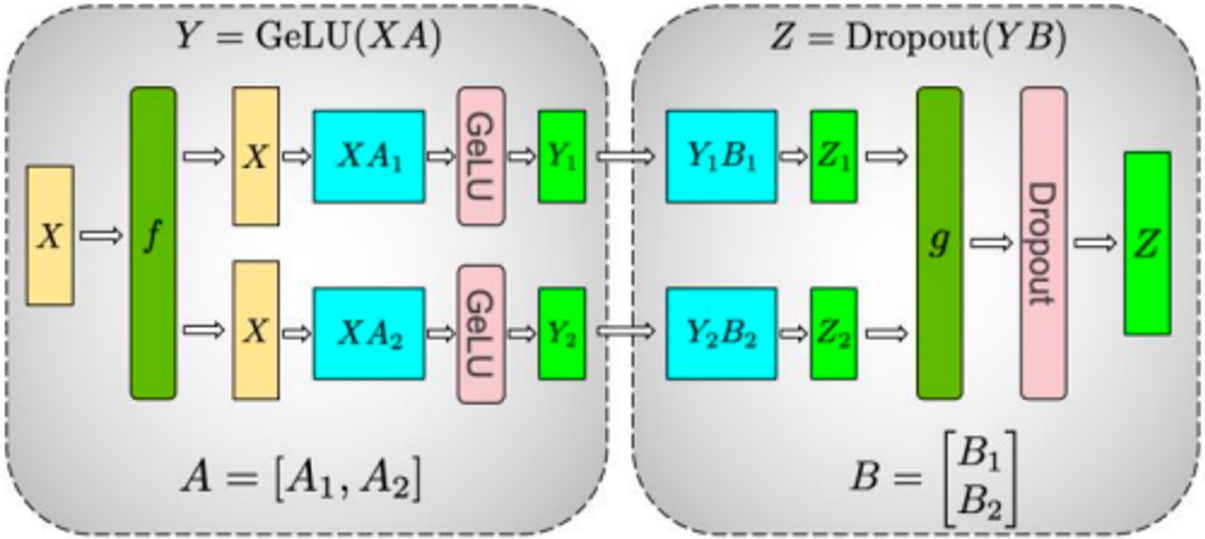
Are there model parallel schemes with better utilization?

We can think of pipeline parallel as cutting up along depth. What about width?



Simple matrix multiplication observation: decompose into submatrices, add partial sums

Tensor parallel – 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 – pros and cons vs pipeline parallel

How do things compare to pipeline parallel?

Pros – no bubble. If your network is fast enough, there's no waiting for others.
– doesn't need large batch sizes to work well

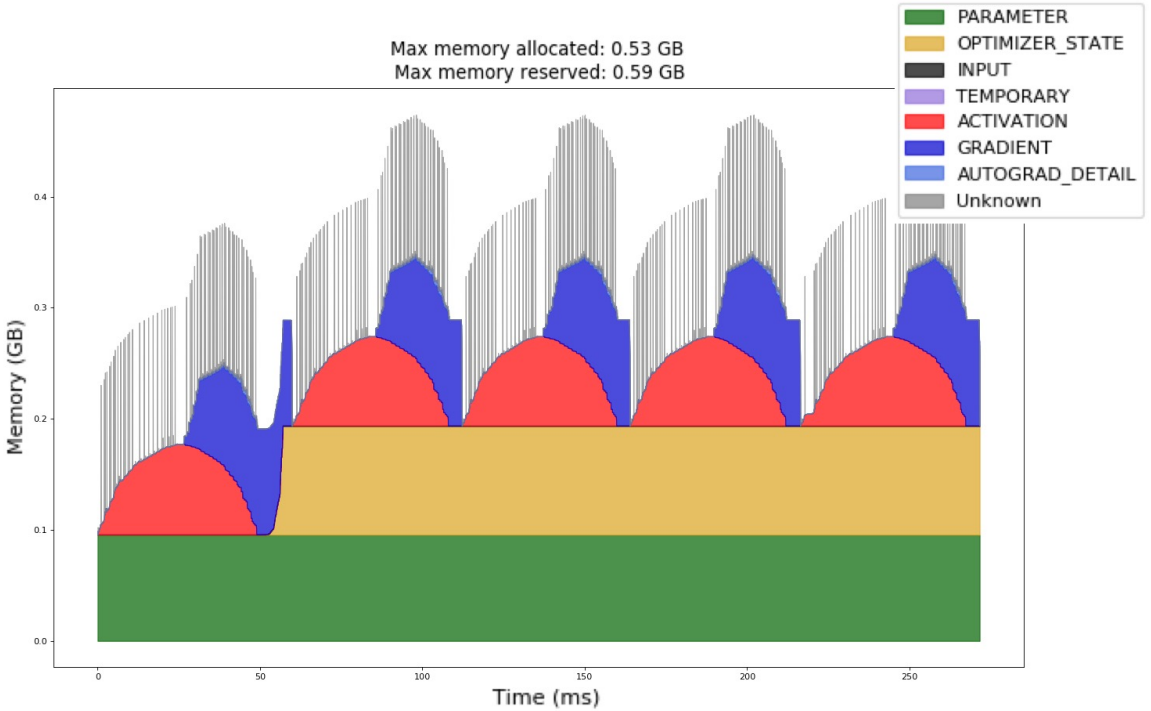
Cons – *much* larger communication than pipeline parallel.

- Pipeline: bsh point-to-point communication per microbatch
- Tensor: $8bsh\left(\frac{n_{devices}-1}{n_{devices}}\right)$ *per layer* and *all-reduce* communication.

Use tensor parallel whenever we have low-latency, high-bandwidth interconnects

A final complexity – memory is dynamic!

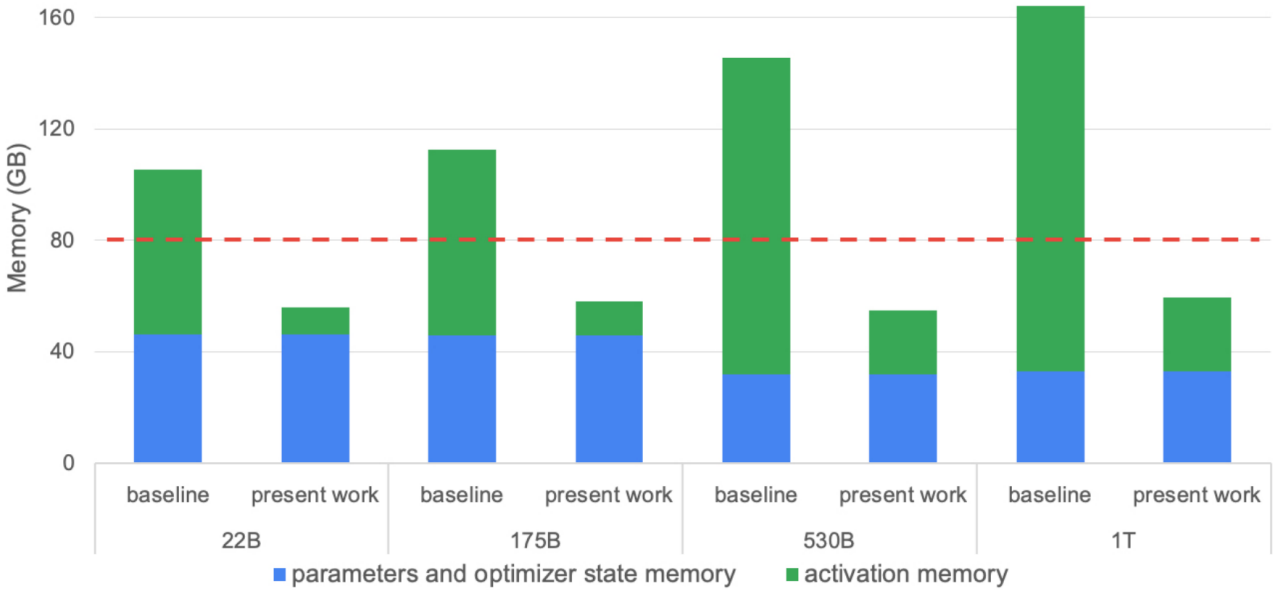
Memory isn't just the static bits, but also activations! This can be big



A final complexity – activation memory

Thus far, we have only really discussed parameter memory.

Tensor and pipeline parallel can linearly reduce those..**but what about activations?**



[Korthikanti et al 2022]

What's the activation memory per layer?

Starting point: activation memory needed if storing everything

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

- The $5\frac{as}{h}$ terms come from the quadratic attention terms incl dropout
- As with flash attention, we can drop this term via recomputation

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

Activation under tensor parallel

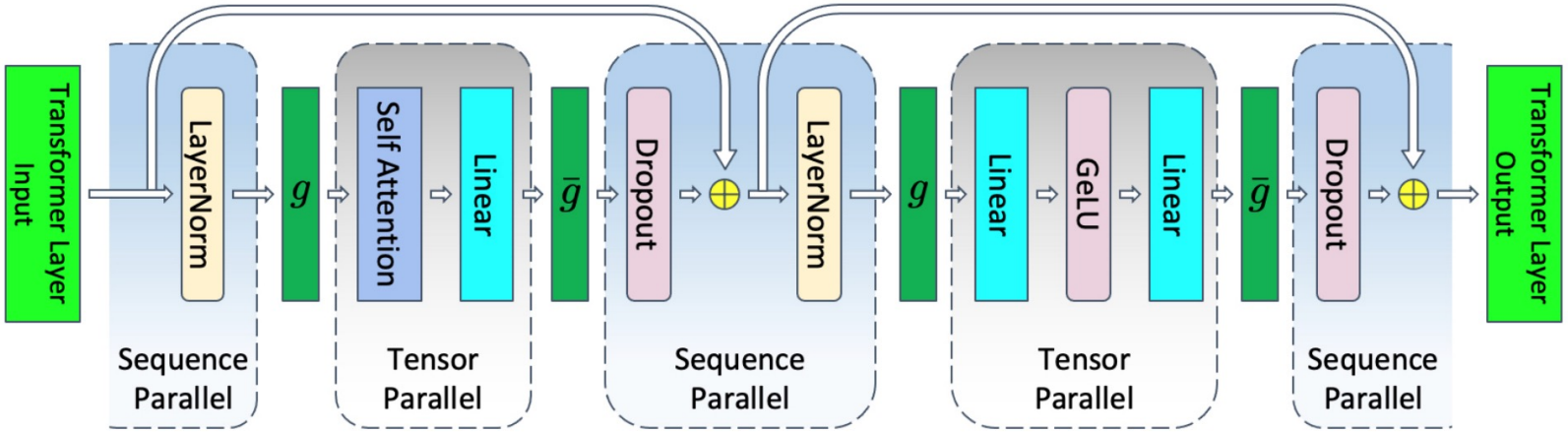
$$\text{Activations memory per layer} = sbh\left(10 + \frac{24}{t} + 5\frac{as}{ht}\right)$$

Tensor parallel splits out the matrix multiplies in attention + MLP

The remaining **10** term is for the LayerNorm (4sbh), Dropout (2sbh), and inputs to the attention and MLP (4sbh). These terms alone will continue to grow with size

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

Making memory truly linear – sequence parallel



Observation: all the 10sbh terms are pointwise ops over the sequence

… so split up the layer norm/dropout layers along the sequence axis.

- In the forward pass, 'g' is an all gather, '$\bar{g}$' is reduce-scatter
- In the backward pass, the two are reversed.

Making activation memory fully scale with more machines

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: LLM parallelism table..

What are each of the parallelism primitives good for?

	Sync overhead	Memory	Bandwidth	Batch size	Easy to use?
DDP/ZeRO1	Per-batch	No scaling	2* # param	Linear	Very
FSDP (ZeRO3)	3x Per-FSDP block	Linear	3 * # param	Linear	Very
Pipeline	Per-pipeline	Linear	Activations	Linear	No
Tensor+seq	2x transformer block	Linear	8*activations per-layer all-reduce	No impact	No

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

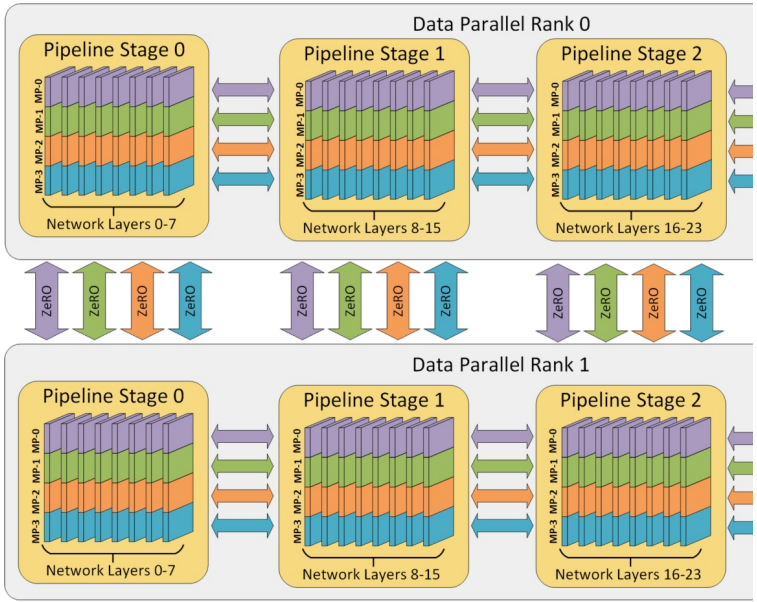
‘3D parallelism’ – putting it all together

Simple rules of thumb from the literature.

1. Until your model fits in memory..
   - Tensor parallel up to GPUs / machine
   - Pipeline parallel across machines

2. Then until you run out of GPUs
   - Scale the rest of the way with data parallel

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



Recent LMs – what do they do?

3.1 Distributed Training Framework

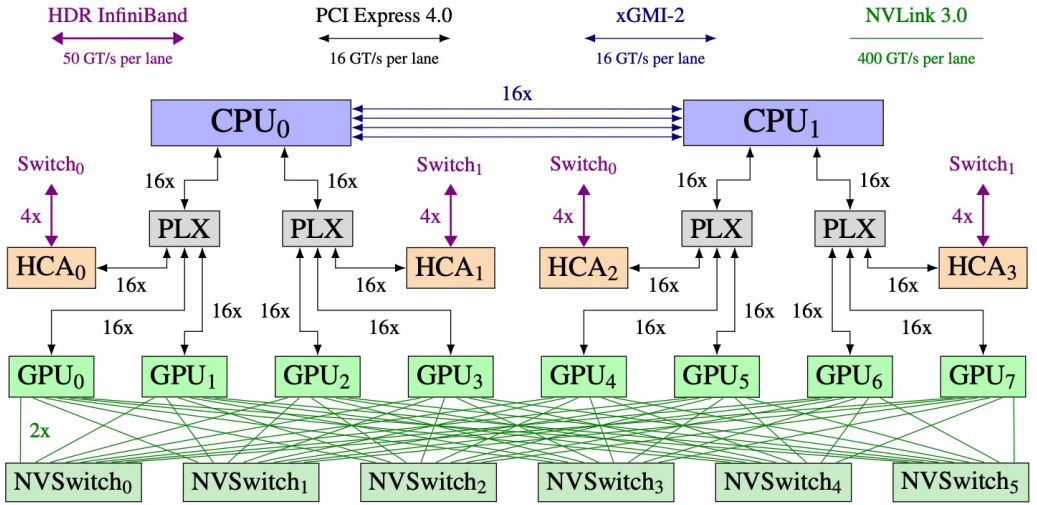
We train our models using the *ZeRO* optimizer strategy (Rajbhandari et al., 2019) via PyTorch's FSDP framework (Zhao et al., 2023), which reduces memory consumption by sharding the model weights and their corresponding optimizer state across GPUs. At the 7B scale, this enables training with a micro-batch size of 4096 tokens per GPU on our hardware (see Section 3.4). For OLMo-1B and -7B models, we use a constant global batch size of approximately 4M tokens (2048 instances, each with a sequence length of 2048 tokens). For OLMo-65B model (currently training), we use a batch size warmup that starts at approximately 2M tokens (1024 instances), then doubles every 100B tokens until reaching approximately 16M tokens (8192 instances).

To improve throughput, we employ mixed-precision training (Micikevicius et al., 2017) through FSDP's built-in settings and PyTorch's `amp` module. The latter ensures that certain operations like the softmax always run in full precision to improve stability, while all other operations run in half-precision with the `bfloat16` format. Under our specific settings, the sharded model weights and optimizer state local to each GPU are kept in full precision. The weights within each transformer block are only cast to `bfloat16` when the full-sized parameters are materialized on each GPU during the forward and backward passes. Gradients are reduced across GPUs in full precision.

Dolma – 7B model, FDSP (probably fits intra-node)

GPT-NeoX

We use the AdamW (Loshchilov and Hutter, 2019) optimizer, with beta values of 0.9 and 0.95 respectively, and an epsilon of 1.0E−8. We extend AdamW with the *ZeRO* optimizer (Rajbhandari et al., 2020) to reduce memory consumption by distributing optimizer states across ranks. Since the weights and optimizer states of a model at this scale do not fit on a single GPU, we use the tensor parallelism scheme introduced in Shoeybi et al. (2020) in combination with pipeline parallelism (Harlap et al., 2018) to distribute the model across GPUs. To train GPT-NeoX-20B, we found that the most efficient way to distribute the model given our hardware setup was to set a tensor parallel size of 2, and a pipeline parallel size of 4. This allows for the most communication intensive processes, tensor and pipeline parallelism, to occur within a node, and data parallel communication to occur across node boundaries. In this fashion, we were



GPT-NeoX – ZeRO stage 1 + Tensor Parallel (4) + Pipeline (2).

Data parallel inter-node.

Yi

Performance and Cost Efficiency

Memory and *communication* restrictions are the two major technical challenges of large scale model training requiring integrated solutions beyond adding more GPUs. We use and improve upon the following techniques to tackle the memory and communication restrictions: (1) ZeRO-1 [60] to remove the memory consumption by partitioning optimizer states cross data-parallel processes; (2) tensor parallel combined with pipeline parallel [70] within each compute node to avoid inter-node communication bottleneck, and the 3D parallel strategy is well designed and optimized to avoid using activation checkpointing and minimize the pipeline bubbles; (3) kernel fusion techniques like flash attention [15] [14] and JIT kernels to reduce redundant global memory access and consumption; (4) topology-aware resource allocation (ranking strategy) to minimize the communication across different layers of switches, which is the limitation of a typical fat-tree-topology.

Yi - ZeRO stage 1 + Tensor + Pipeline parallel

DeepSeek

2.4. Infrastructures

We use an efficient and light-weight training framework named HAI-LLM (High-flyer, 2023) to train and evaluate large language models. Data parallelism, tensor parallelism, sequence parallelism, and 1F1B pipeline parallelism are integrated into this framework as done in Megatron (Korthikanti et al., 2023; Narayanan et al., 2021; Shoeybi et al., 2019). We also leverage the flash attention (Dao, 2023; Dao et al., 2022) technique to improve hardware utilization. ZeRO-1 (Rajbhandari et al., 2020) is exploited to partition optimizer states over data parallel ranks. Efforts are also made to overlap computation and communication to minimize additional waiting overhead, including the backward procedure of the last micro-batch and reduce-scatter operation in ZeRO-1, and GEMM computation and all-gather/reduce-scatter in sequence parallel. Some layers/operators are fused to speed up training, including LayerNorm, GEMM whenever possible, and Adam updates. To improve model training stability, we train the model in bf16 precision but accumulate gradients in fp32 precision. In-place cross-entropy is performed to reduce GPU memory consumption, i.e.: we convert bf16 logits to fp32 precision on the fly in the cross-entropy CUDA kernel (instead of converting it beforehand in HBM), calculate the corresponding bf16 gradient, and overwrite logits with its gradient.

DeepSeek – ZeRO stage 1 with Tensor, Sequence, and Pipeline parallel

Falcon

Training Procedure

Falcon-40B was trained on 384 A100 40GB GPUs, using a 3D parallelism strategy (TP=8, PP=4, DP=12) combined with ZeRO.

Falcon 40B - Zero (probably stage 1?) with Tensor (8), Pipeline (4) parallel
Likely Tensor intra-node.

Scaling strategies from Narayanan 2021

Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM

Deepak Narayanan[‡*], Mohammad Shoeybi[†], Jared Casper[†], Patrick LeGresley[†], Mostofa Patwary[†], Vijay Korthikanti[†], Dmitri Vainbrand[†], Prethvi Kashinkunti[†], Julie Bernauer[†], Bryan Catanzaro[†], Amar Phanishayee[*], Matei Zaharia[‡]
[†]*NVIDIA* [‡]*Stanford University* [*]*Microsoft Research*

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	32
7.5	32	4096	36	4	1	128	512	142	46%	18.2	32
18.4	48	6144	40	8	1	256	1024	135	43%	34.6	32
39.1	64	8192	48	8	2	512	1536	138	44%	70.8	32
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8	32
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1	24
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

Notes

- Tensor parallel first up to 8, then caps out at 8.
- Pipeline parallel goes up to make the model fit.
- Dara parallel gradually decreases with scale, with the largest model having DP=6

Careful ‘3D’ parallelism gives linear gains

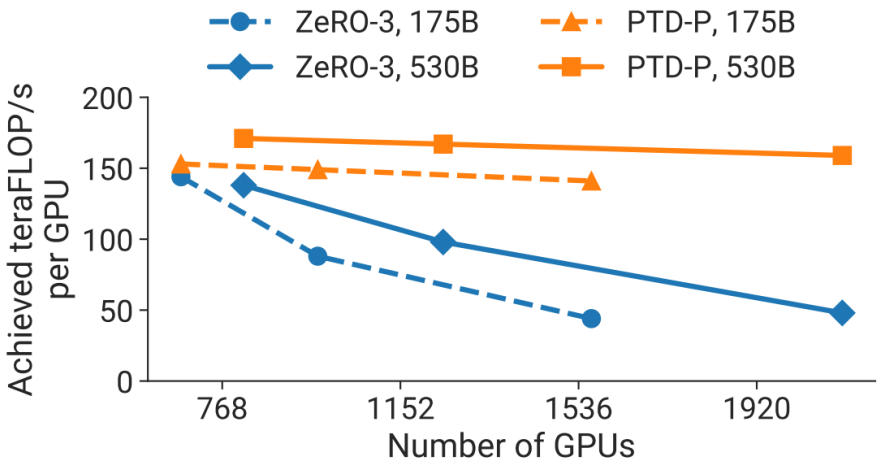


Figure 10: Throughput per GPU of PTD-P and ZeRO-3 for two different GPT models (the 175B GPT-3 model is shown with dotted lines, and the 530B model is shown with solid lines). Global batch sizes are fixed and ZeRO-3 is used without any model parallelism.

More GPUS, same, flat utilization!

Tensor parallel = 8 is often optimal

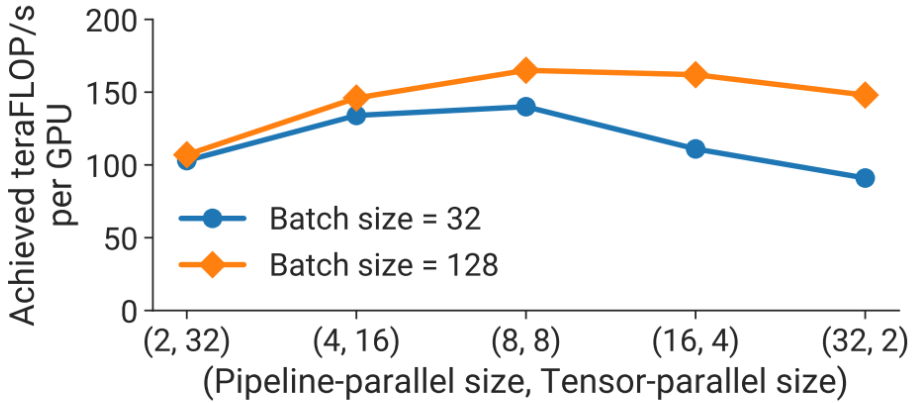
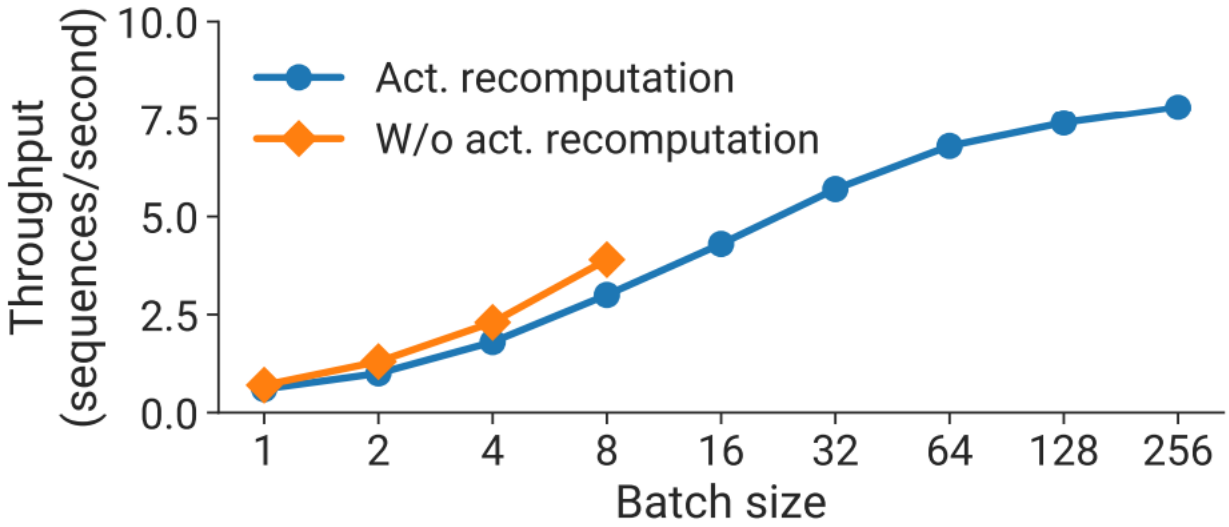


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

When parallelizing across 64 machines – it's best to use a 8 x 8 configuration.

Activation recomputation can pay for itself (via memory)



Activation recomputation enables larger batches, improving throughput (t=8, p=16)

Recap for the whole lecture

- Scaling beyond a certain point requires multi-gpu, multi-node parallelism
- No single solution to the parallelism problem (probably want all 3 approaches)
- Simple, interpretable rules of thumb for combining different forms of parallelism