Johns Hopkins
Whiting School of Engineering

Model Efficiency

CSCI 601-471/671 (NLP: Self-Supervised Models)

https://self-supervised.cs.jhu.edu/sp2025/

Our models are getting larger!

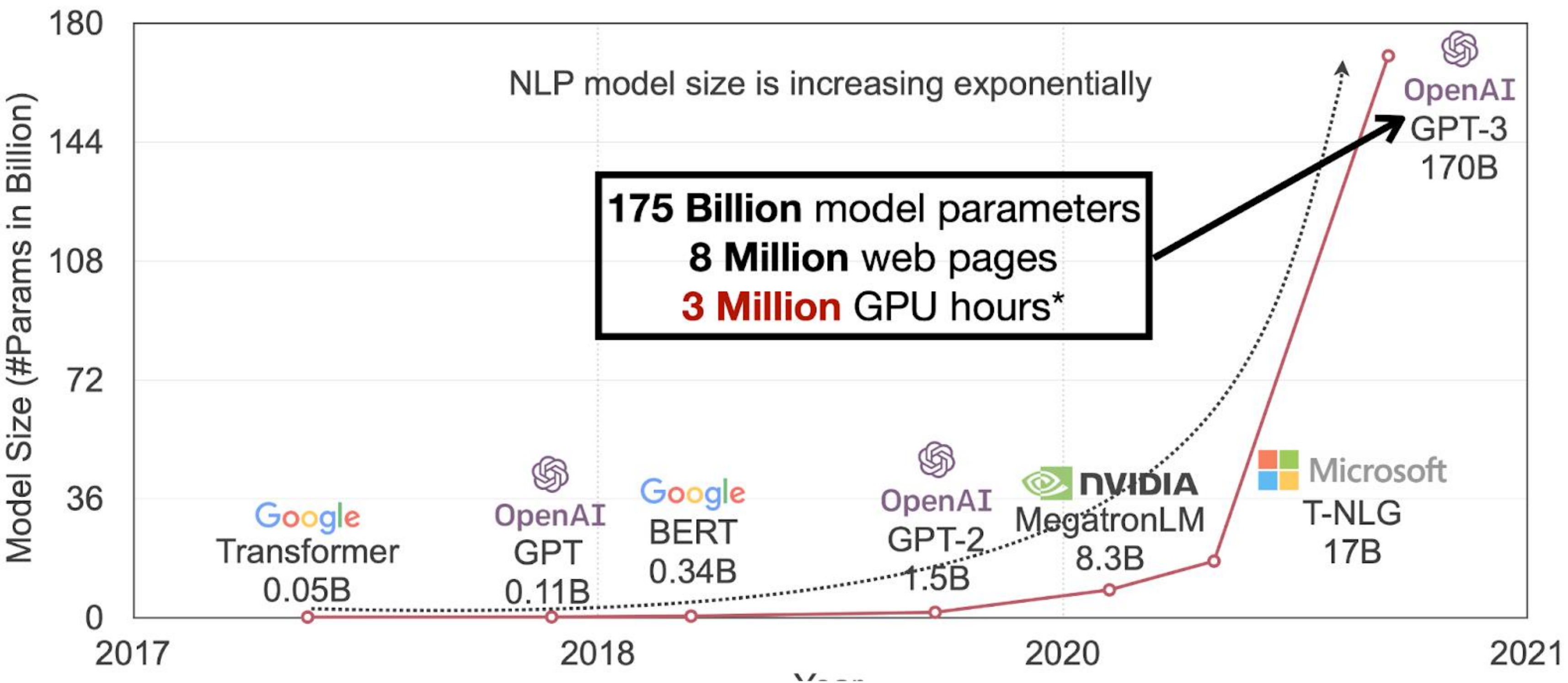
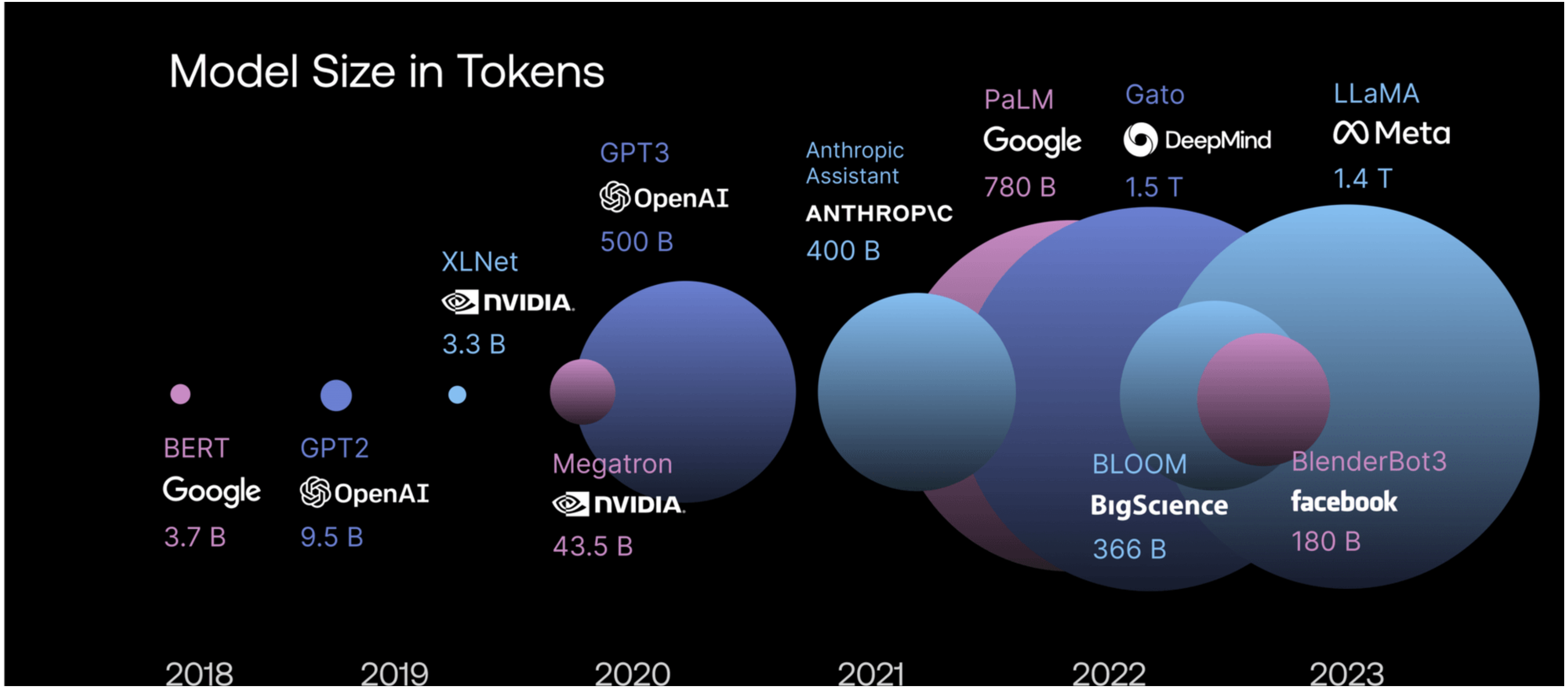


Figure Credit: Song Han (MIT)

And consumes a lot of data!



Motivation

How much GPU memory (at least) do we need to perform inference/training?

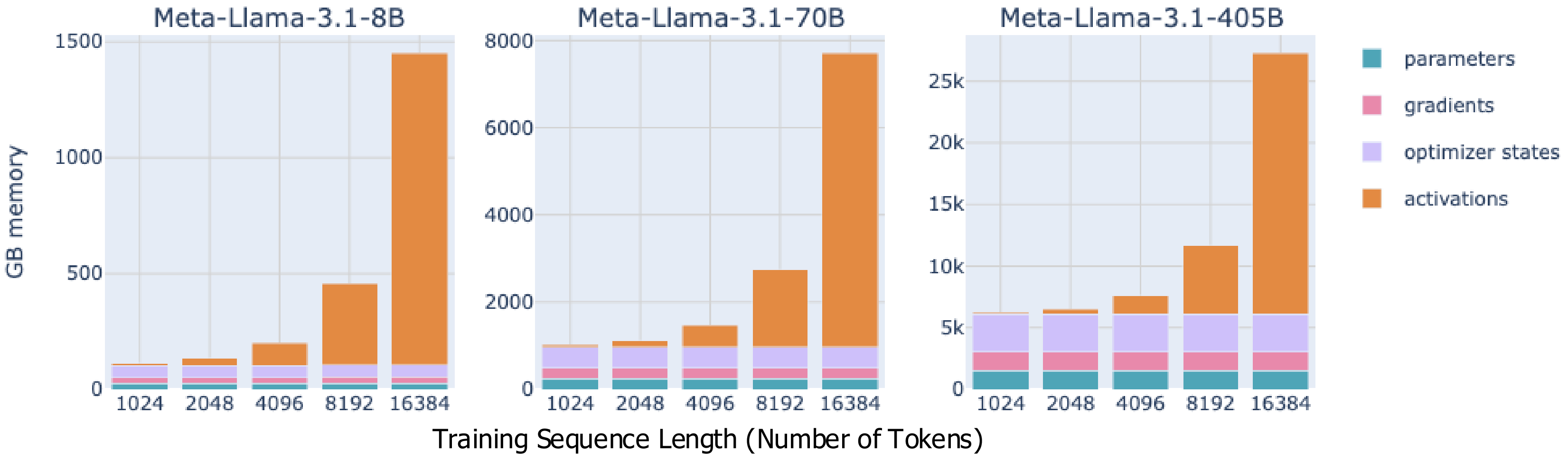
(batch size=1, ignoring the KV cache)

Model Size (Llama 3 Arch)	Inference Memory (~2x model size)	Training Memory (~7x model size)
8B	16GB	60GB
70B	140GB	500GB
405B	810GB	3.25TB

Source: https://huggingface.co/blog/llama31#inference-memory-requirements

Where did all the memory go?

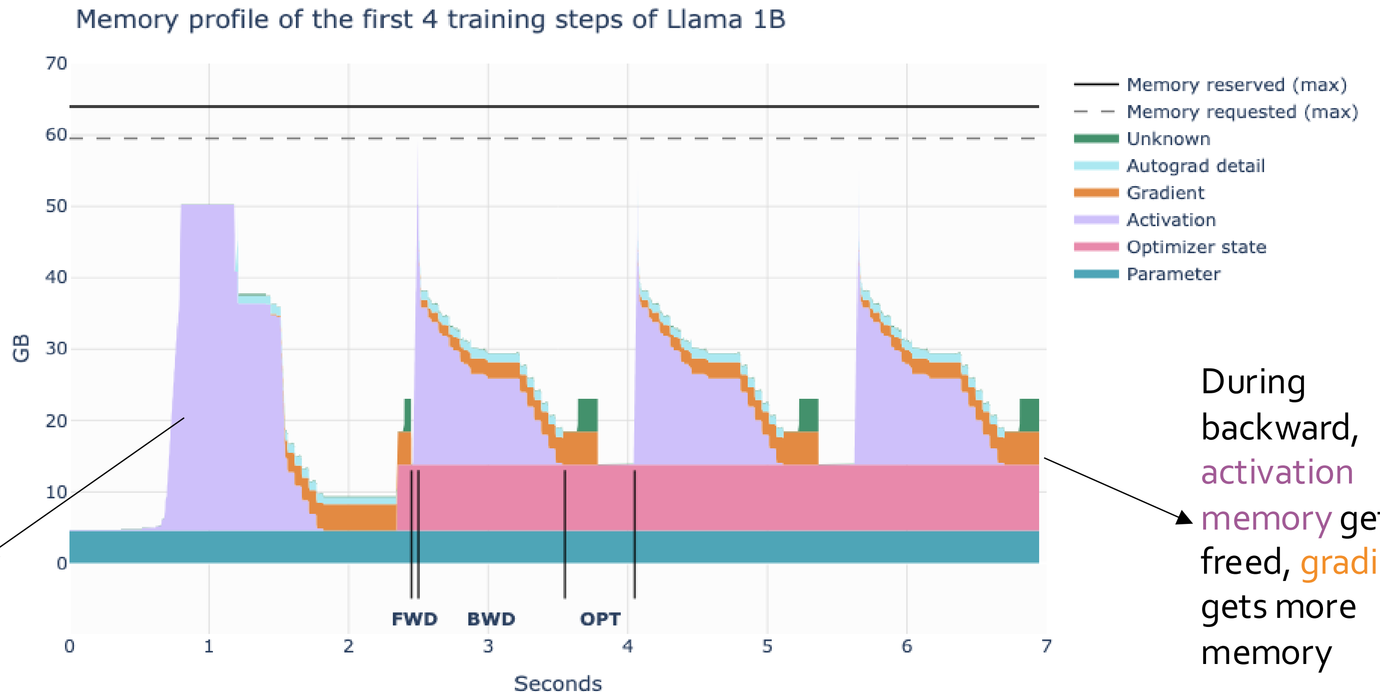
Longer sequences require much more memory in training!



Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Johns Hopkins Whiting School of Engineering

Memory consumption is not static



During the forward step, the activations occupy most of the memory

During backward, activation memory gets freed, gradients gets more memory

Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Model Efficiency: Topics

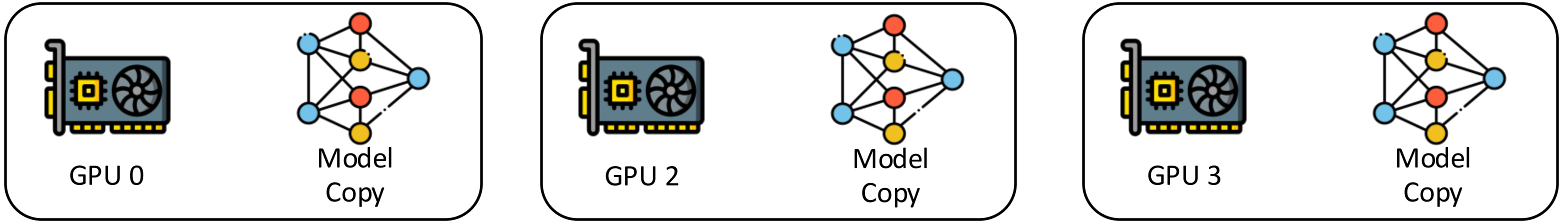
1. Distributed Training
2. Quantization (Post Training Quantization)
3. Distillation

Chapter goal: Getting comfortable with various mathematical and systems foundations for efficient deployment of LLMs.

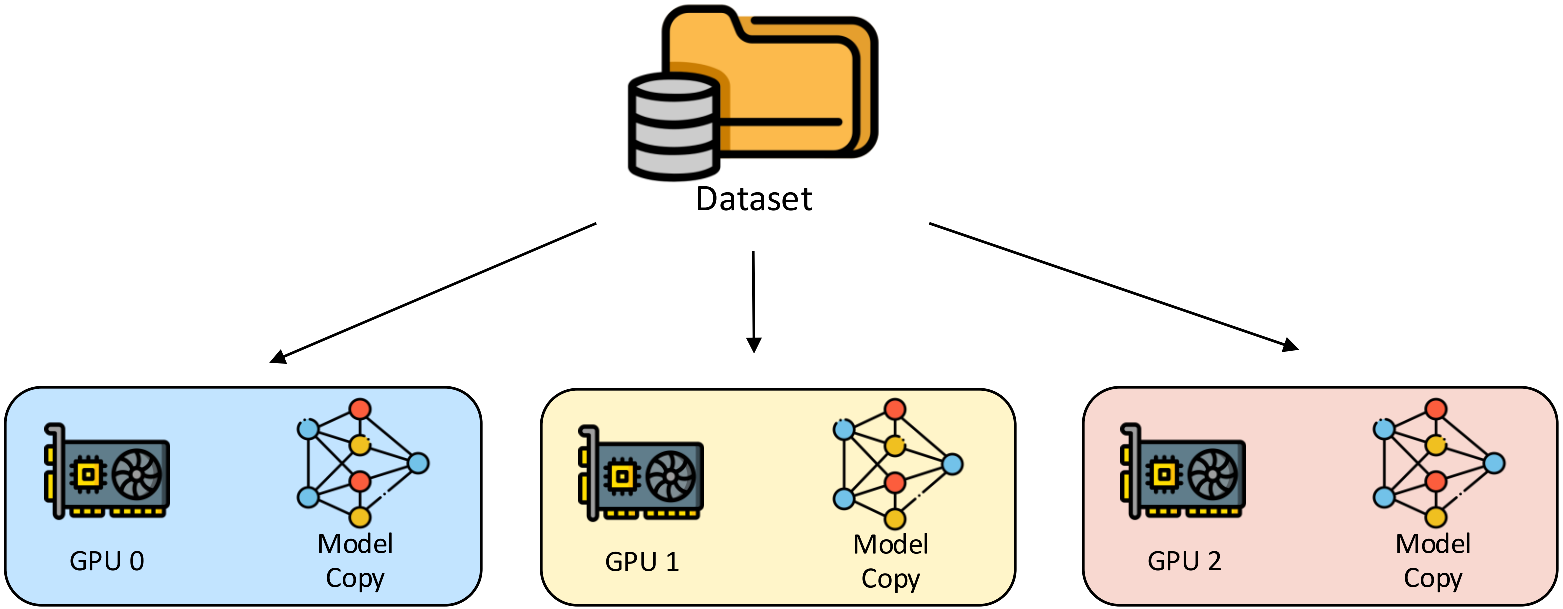
Distributed Training

Distributed Training

1. Naïve Data Parallelism
2. Sharding Optimizer States (ZeRO, FSDP)
3. Model Parallelism (Tensor Parallelism, Pipeline Parallelism)



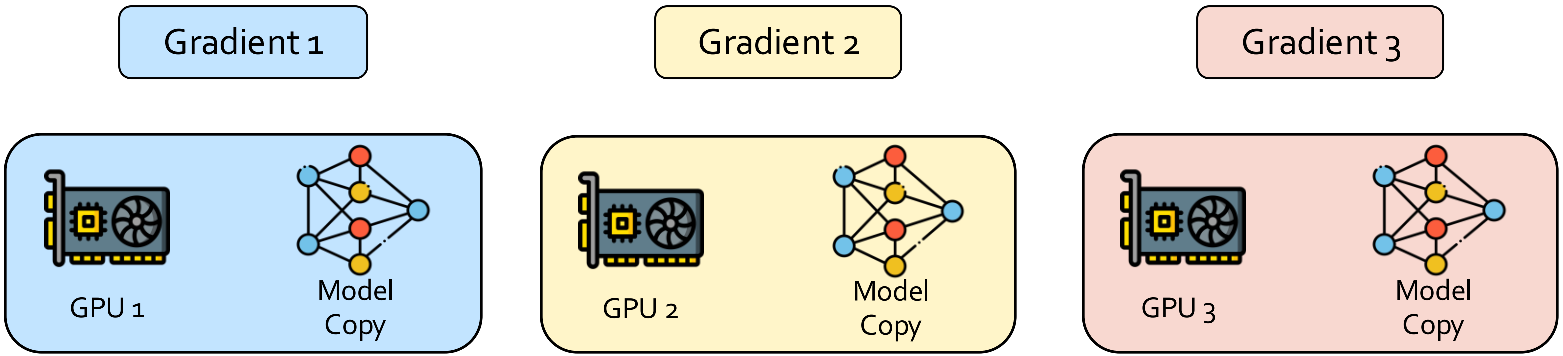
Naïve Data Parallelism



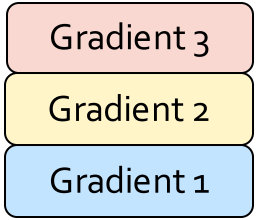
First, we want to shard the dataset and feed them into different GPUs
How do we update the parameters?

Naïve Data Parallelism

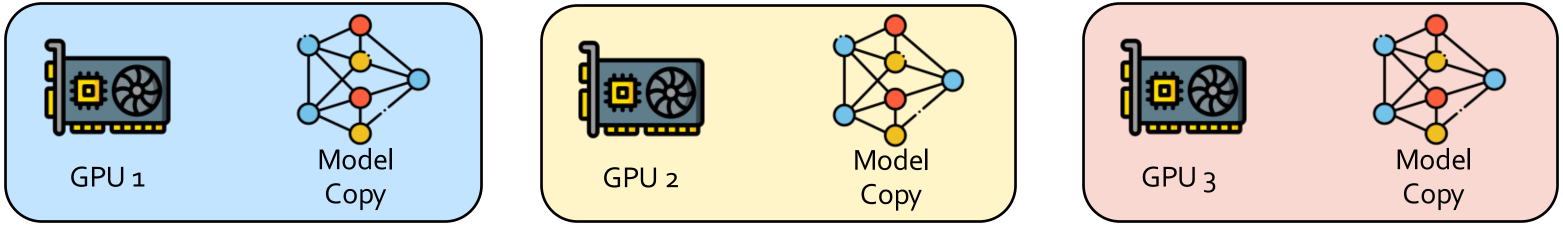
Each GPU compute gradient with a single shard of data



Naïve Data Parallelism

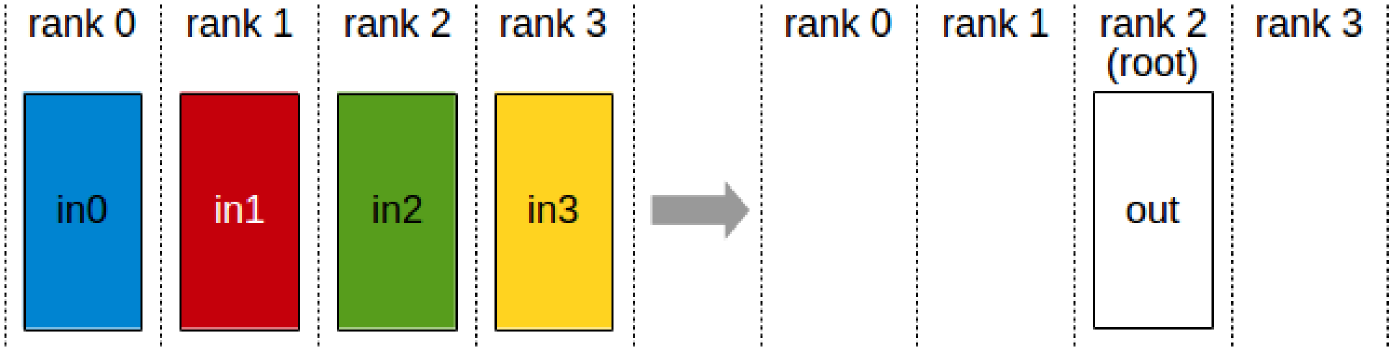


One GPU accumulates the gradients
(reduce in torch.distributed)



NCCL Operations: Reduce

- Nvidia Collective Communications Library (NCCL) - A library developed to provide inter-GPU communications primitives (operations)
- Reduce: *Sums* over all *tensors* and stores it in a root GPU



out[i] = sum(inX[i])

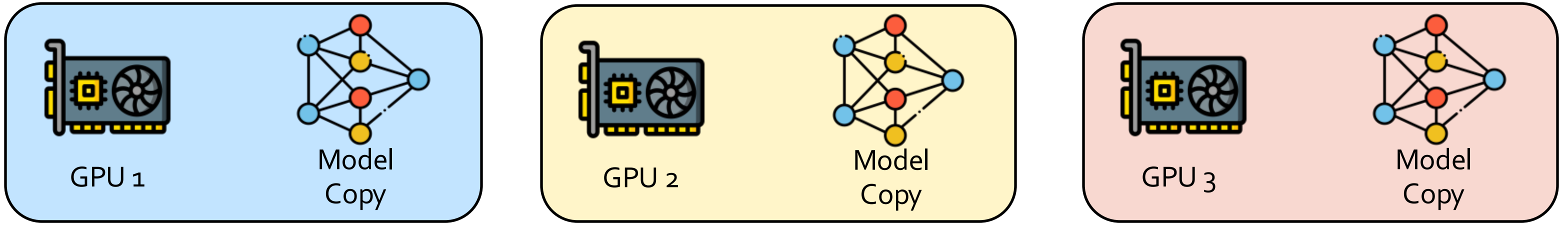
Naïve Data Parallelism

Gradient 3

Gradient 2

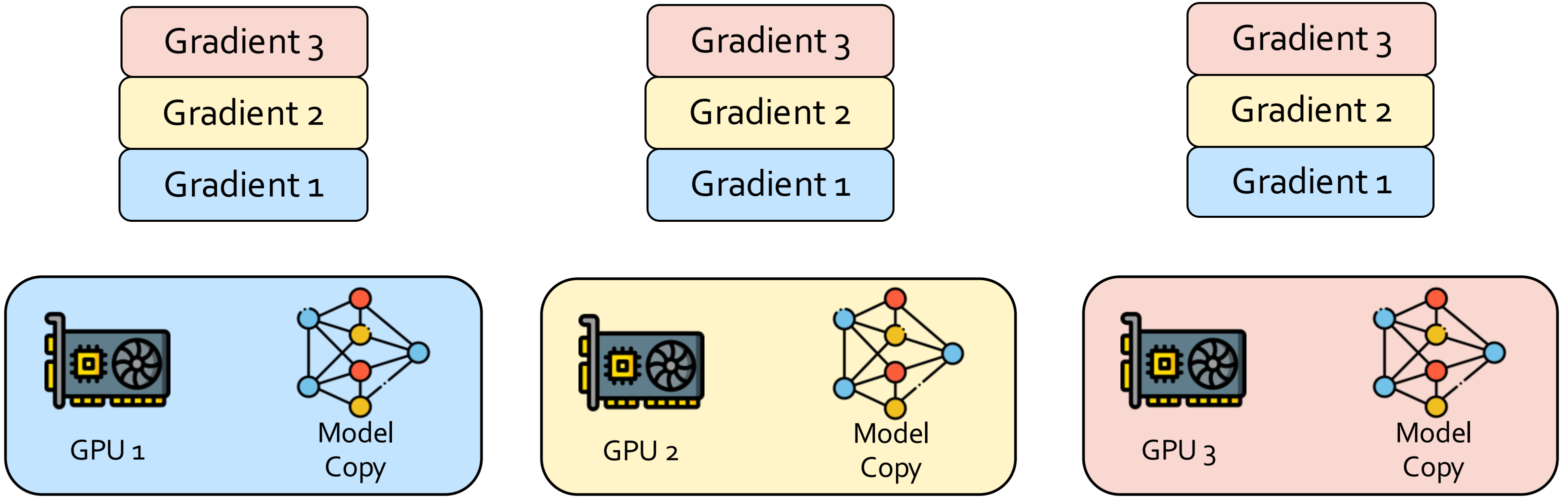
Gradient 1

One GPU accumulates the gradients
(reduce in torch.distributed)



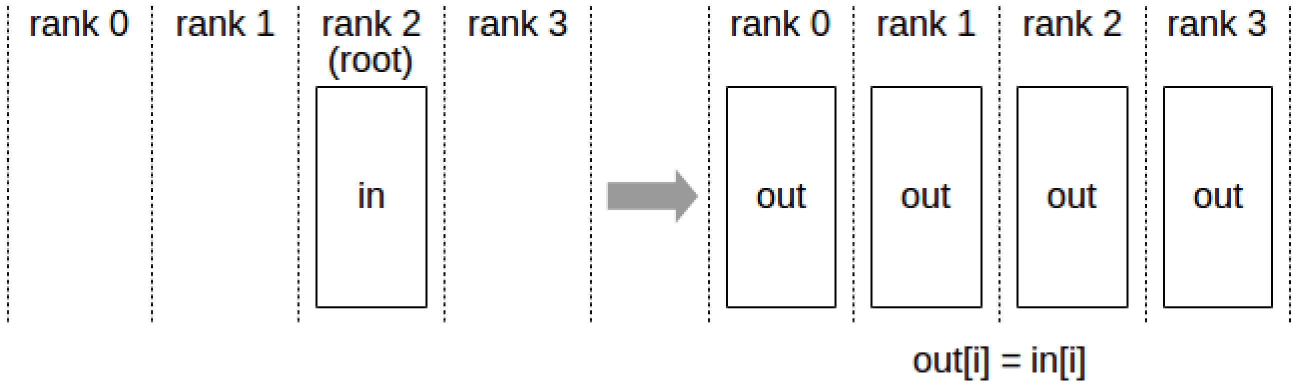
Naïve Data Parallelism

And send the accumulated gradient to all other GPUs (broadcast in torch.distributed)



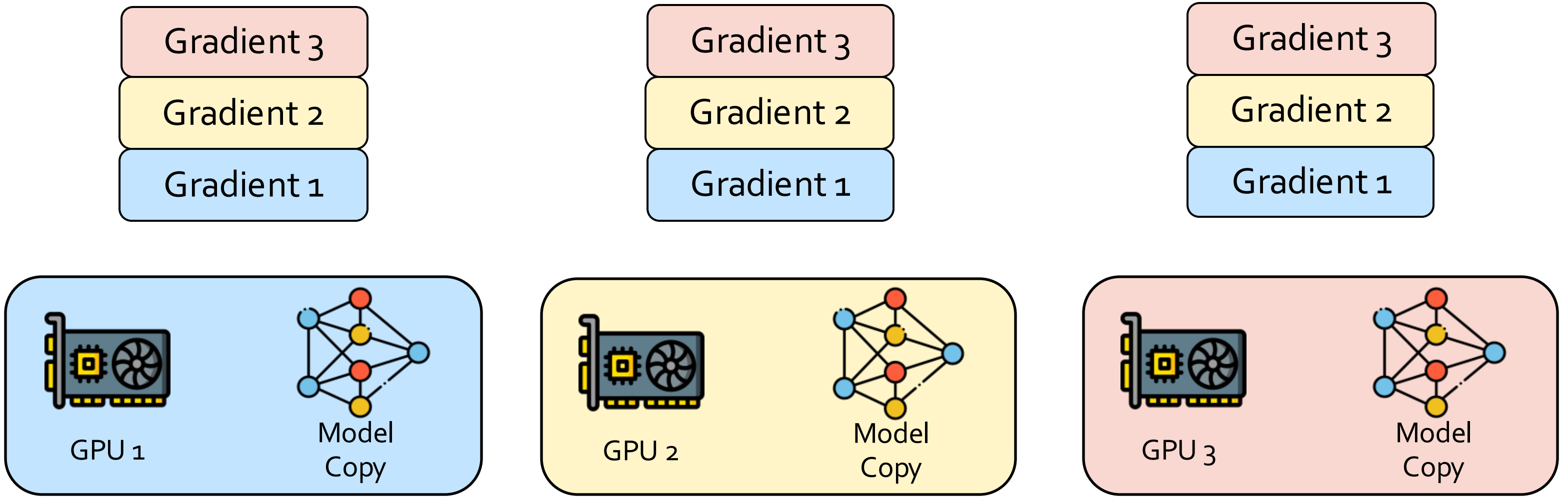
NCCL Operations: Broadcast

- Broadcast: Duplicates one tensor to all GPUs



Naïve Data Parallelism

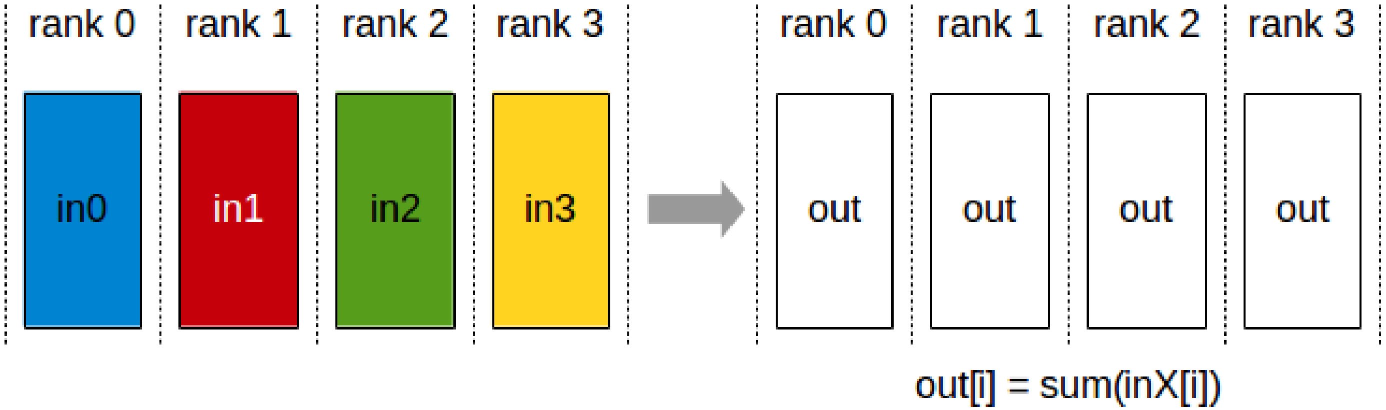
Accumulate gradients across all GPUs and perform gradient updates
(all_reduce in torch.distributed)



NCCL Operations: All Reduce

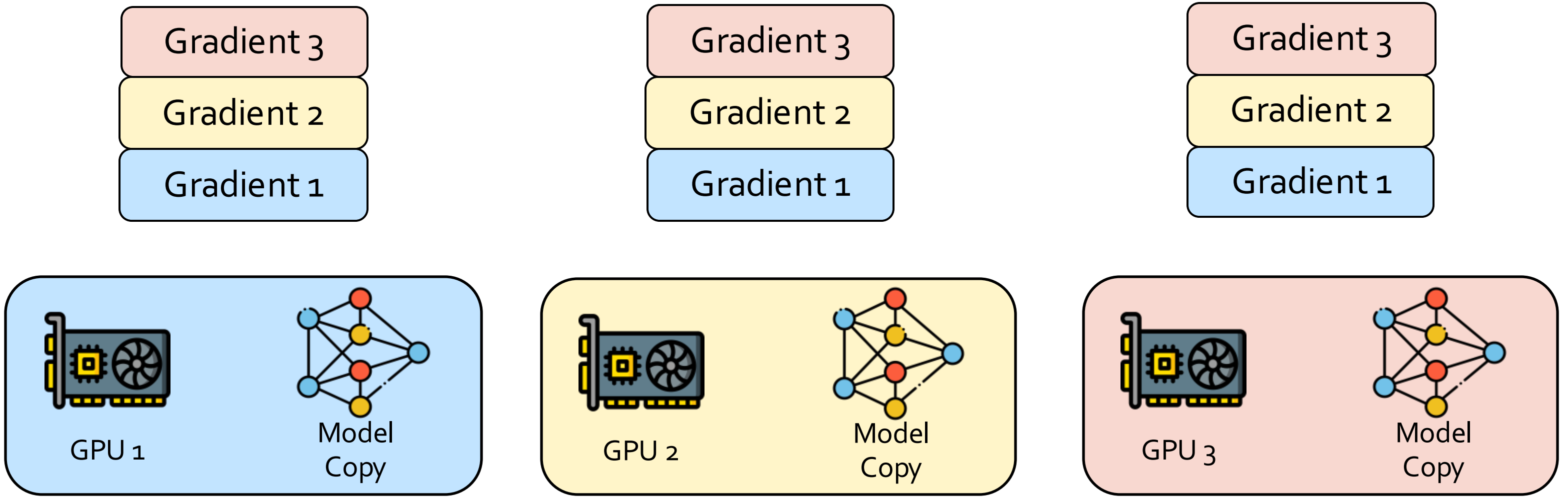
- All Reduce = Reduce + Broadcast

  = Sum over input tensors, then duplicate it to all GPUs



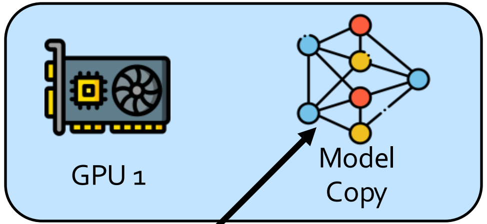
Naïve Data Parallelism

Accumulate gradients across all GPUs and perform gradient updates
(all_reduce in torch.distributed)



What is wrong with Naïve DP

- Consumes too much memory in each GPU!

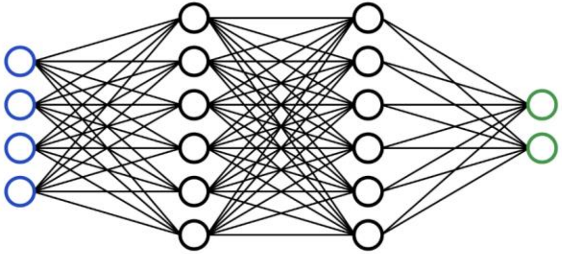


We need to store 5 copies of weights,
which occupies 16 bytes per param

- 2 bytes for FP/BF16 model params
- 2 bytes for FP/BF16 gradients
- 4 bytes for FP32 master weights

  (the thing you accumulate into in SGD, used in mixed precision training)
- 4 bytes for FP32 Adam first order estimates
- 4 bytes for FP32 Adam second order estimates

What is wrong with Naïve DP



GPT-2
1.5B params

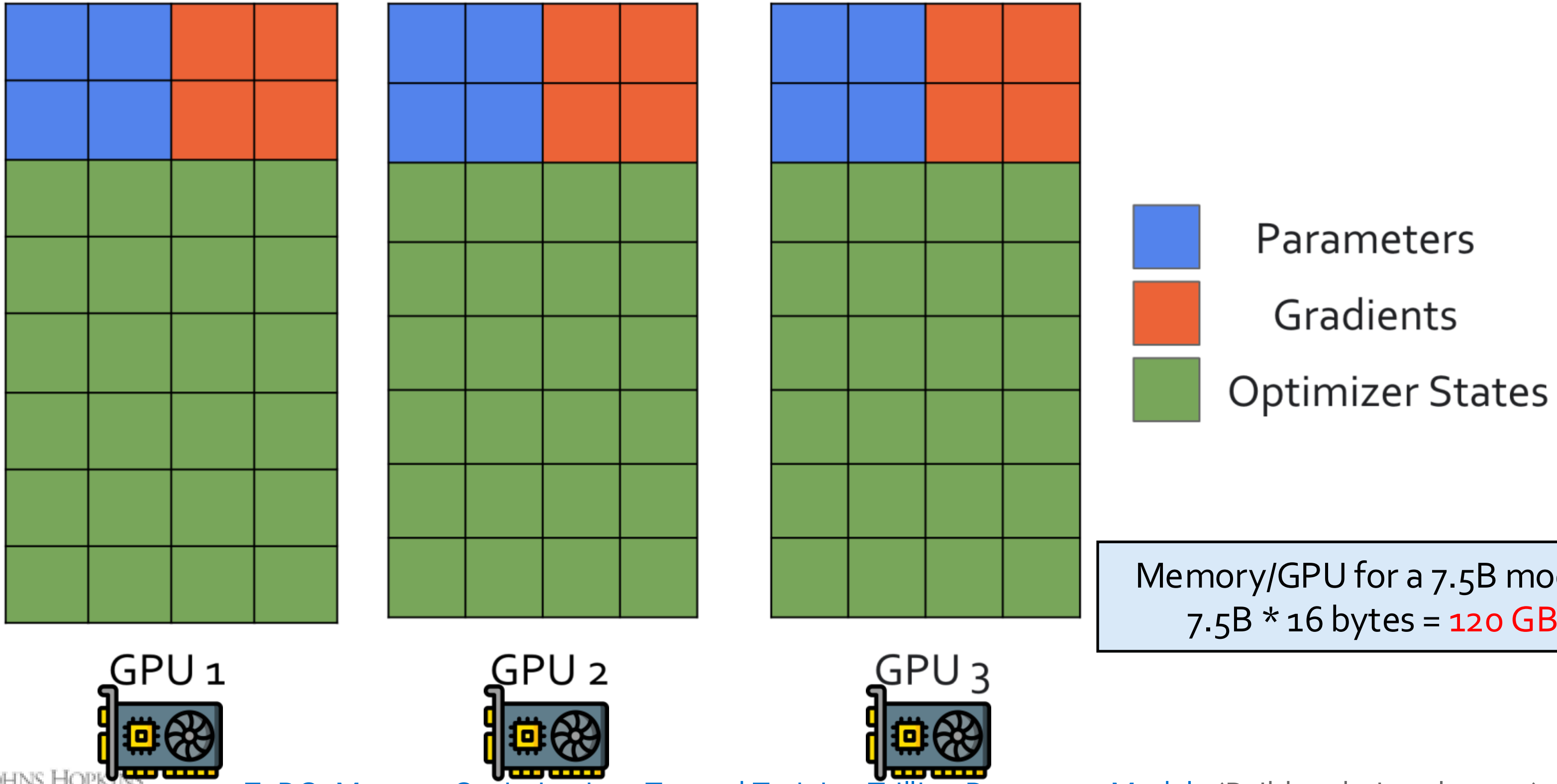
FP16
3GB memory

Training
> 32 GB memory

Most of the memory are occupied by optimizer states.

Some are also occupied by *residual states*: activations, buffers and fragmented memory

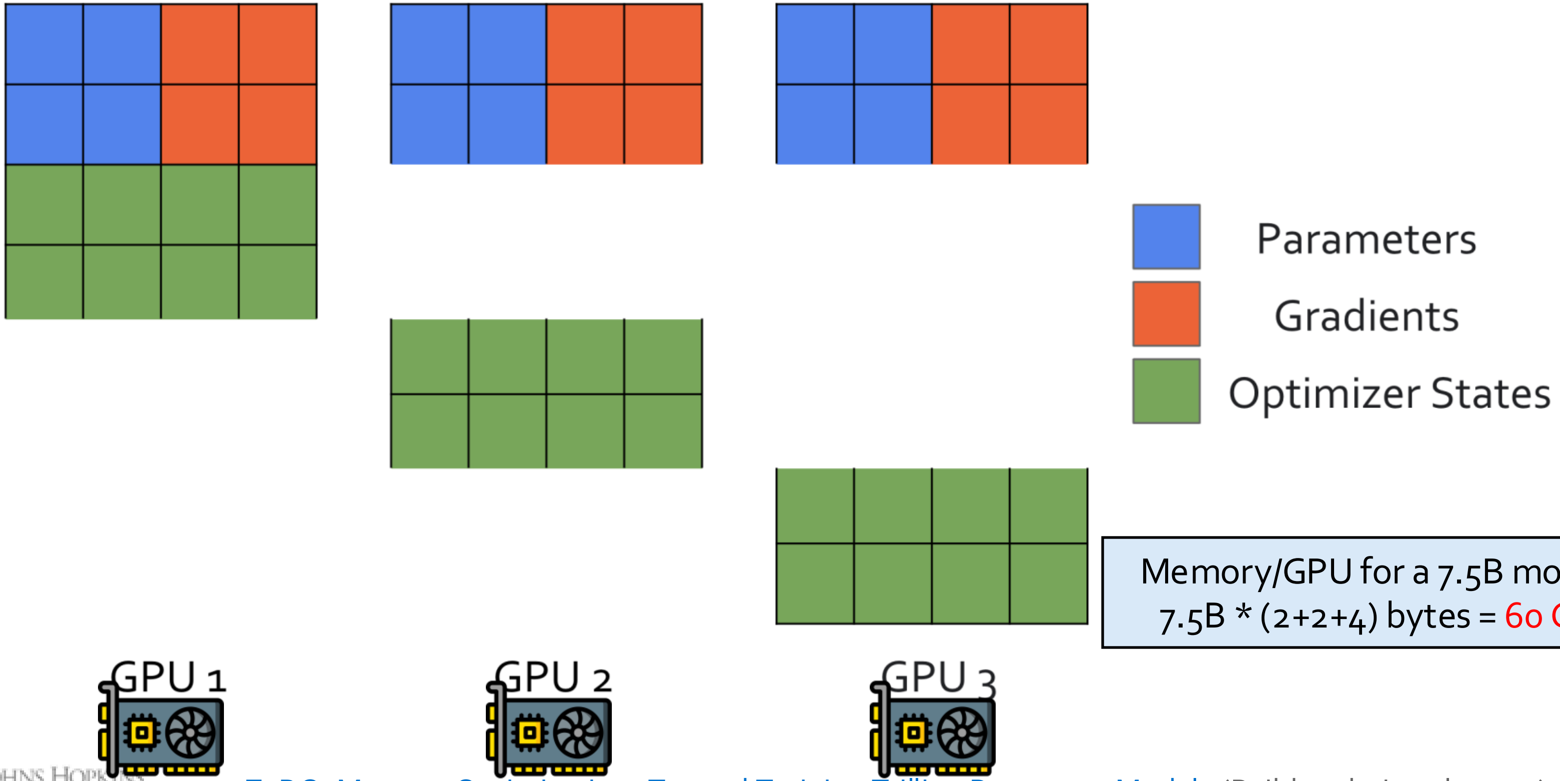
Naïve DP – Requires too much memory!



Memory/GPU for a 7.5B model:
7.5B * 16 bytes = 120 GB!

[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

ZeRO Stage 1: Sharding Optimizer States



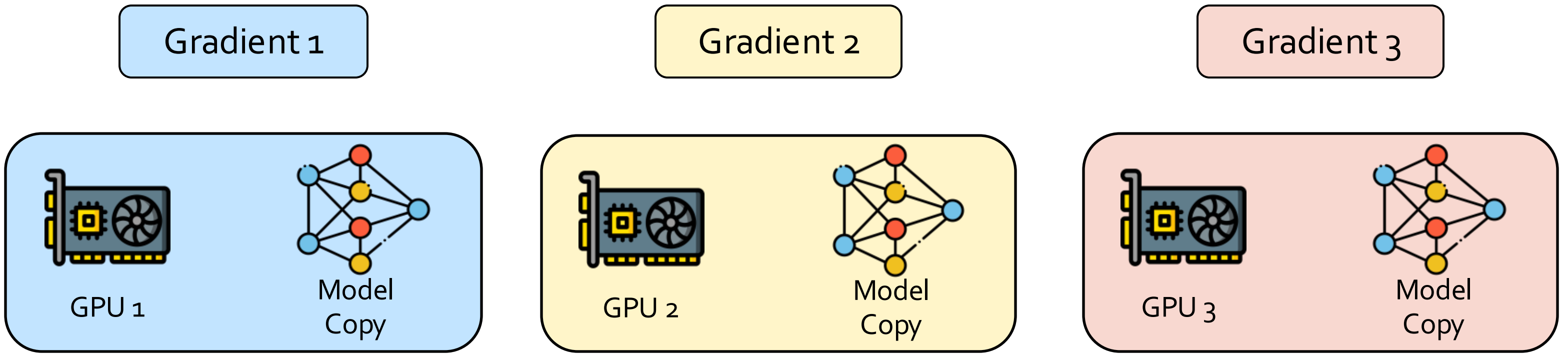
Memory/GPU for a 7.5B model:
7.5B * (2+2+4) bytes = 60 GB!

[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

ZeRO Stage 1: How it works

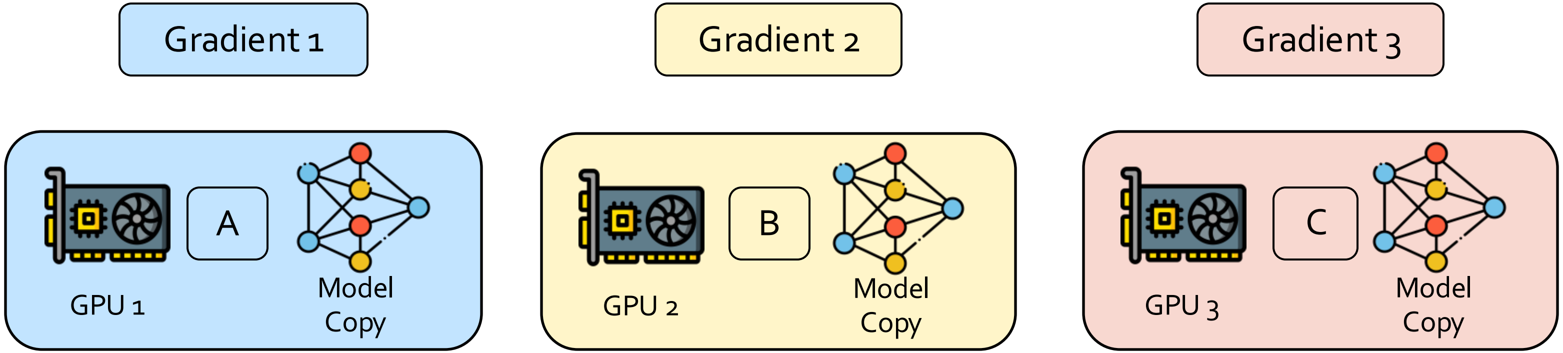
Update Parameters

Each GPU compute gradient with a single shard of data
(The same as naïve DP)



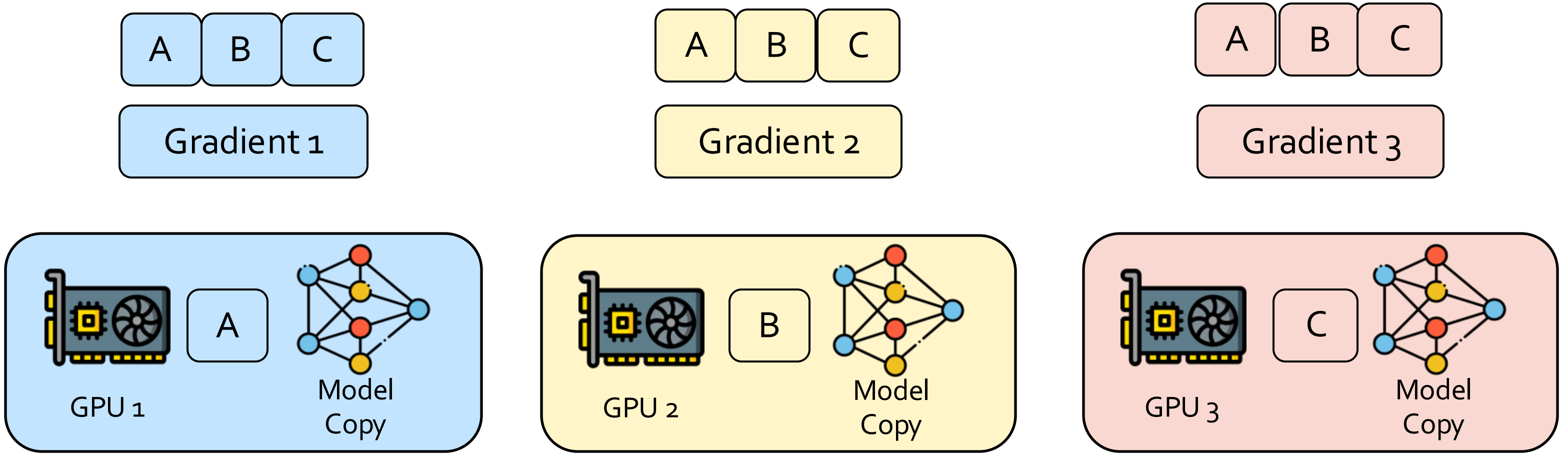
ZeRO Stage 1: How it works

Assuming that
GPU1 stores parameter states for parameters A,
GPU2 stores states for params B,
GPU3 stores states for params C



ZeRO Stage 1: How it works

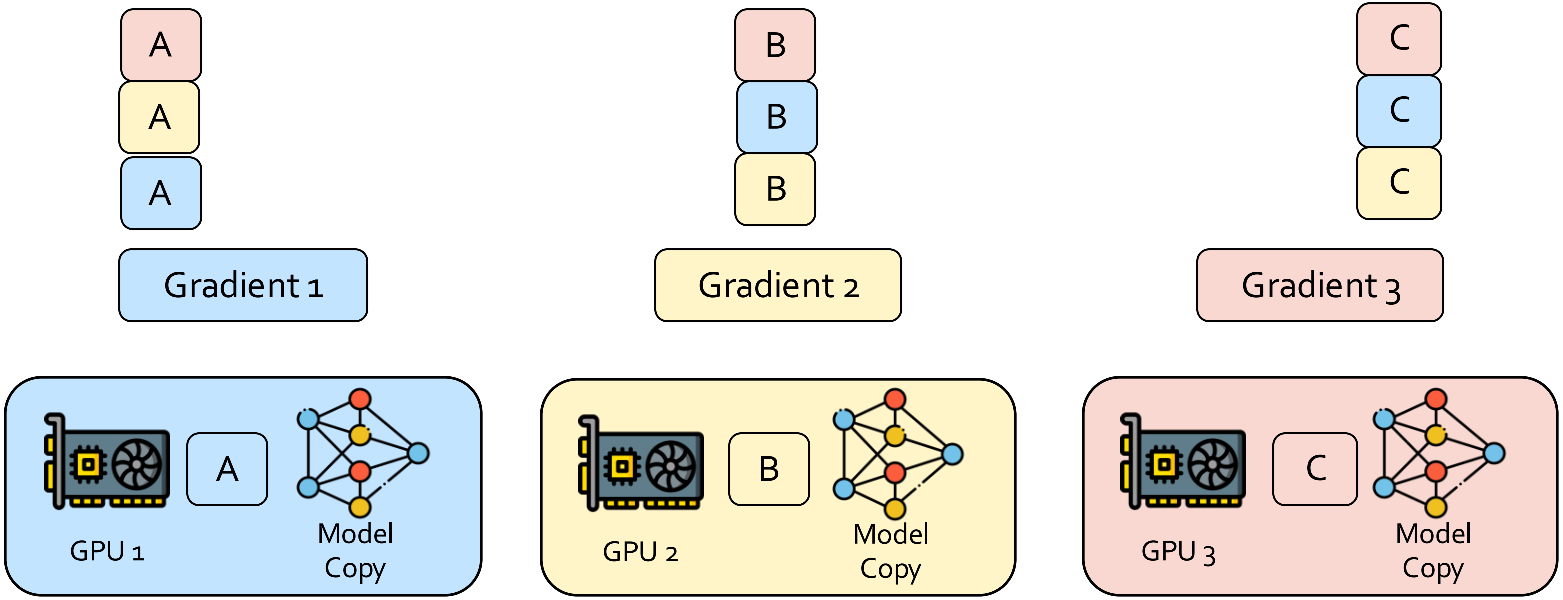
Split / shard the gradients into 3 parts!



[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

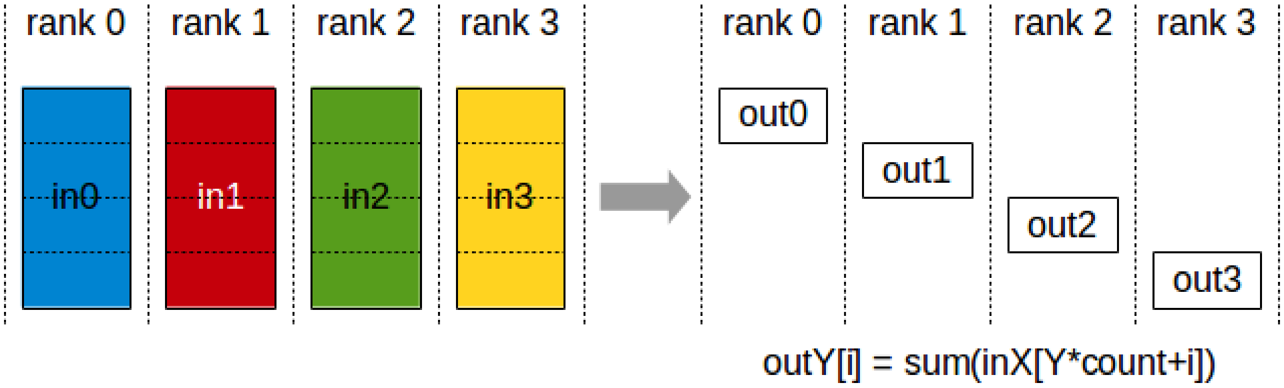
ZeRO Stage 1: How it works

Each GPU accumulates gradients of the params whose optimizer states the GPU is storing (reduce_scatter in torch.distributed)



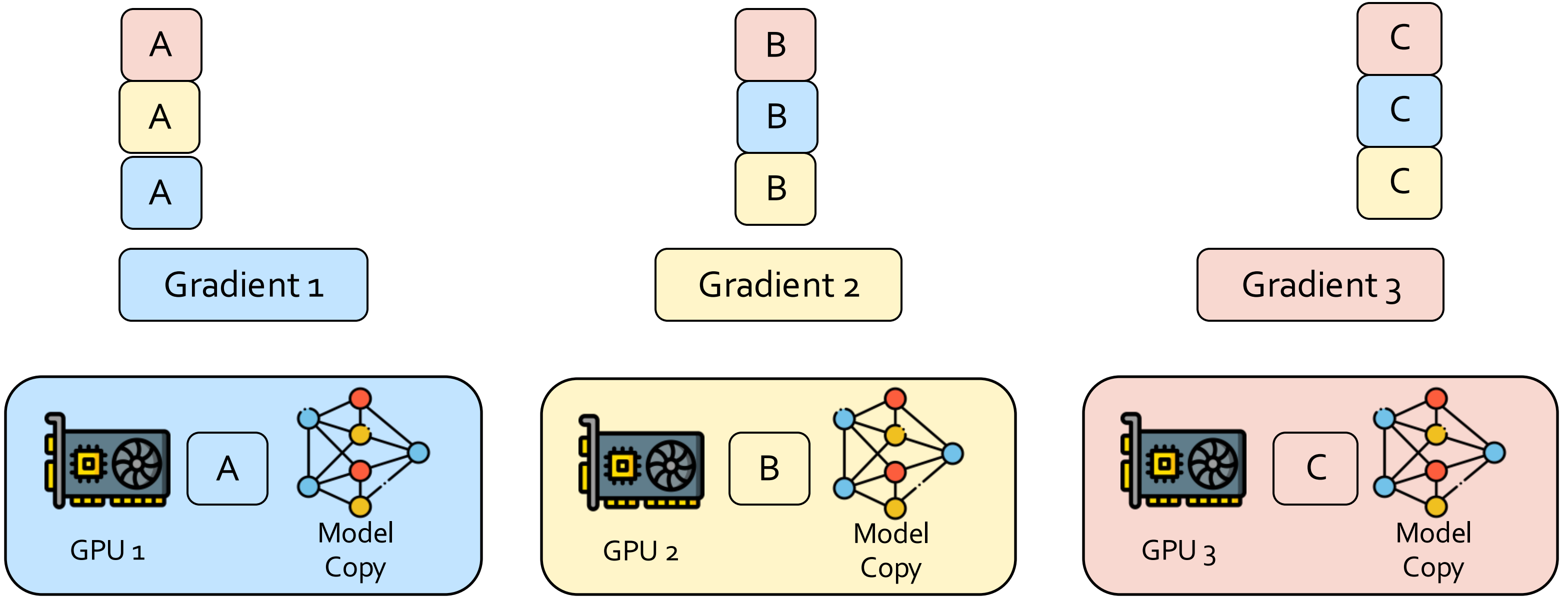
NCCL Operations: Reduce Scatter

- reduce_scatter: each GPU stores the sum of <u>a shard</u> of the input.
- all_reduce: one GPU stores the sum over <u>all</u> the input.



ZeRO Stage 1: How it works

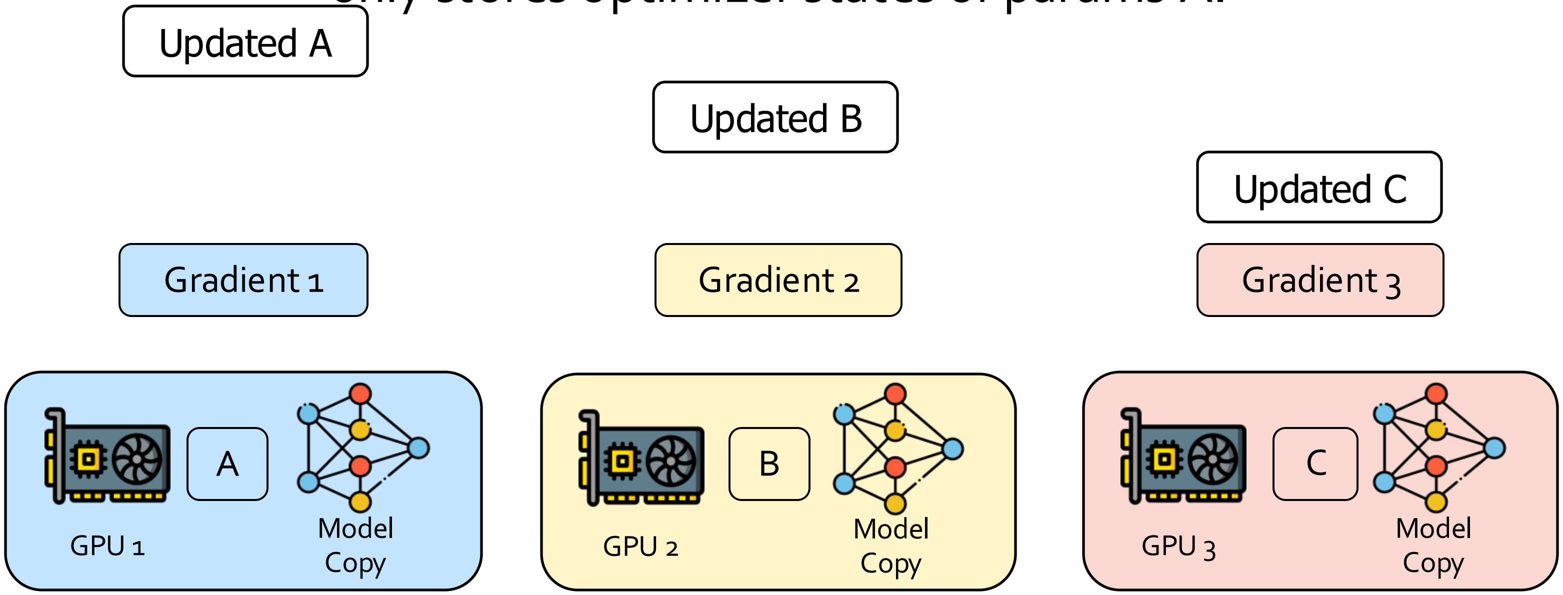
Each GPU accumulates gradients of the params whose optimizer states the GPU is storing (reduce_scatter in torch.distributed)



[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

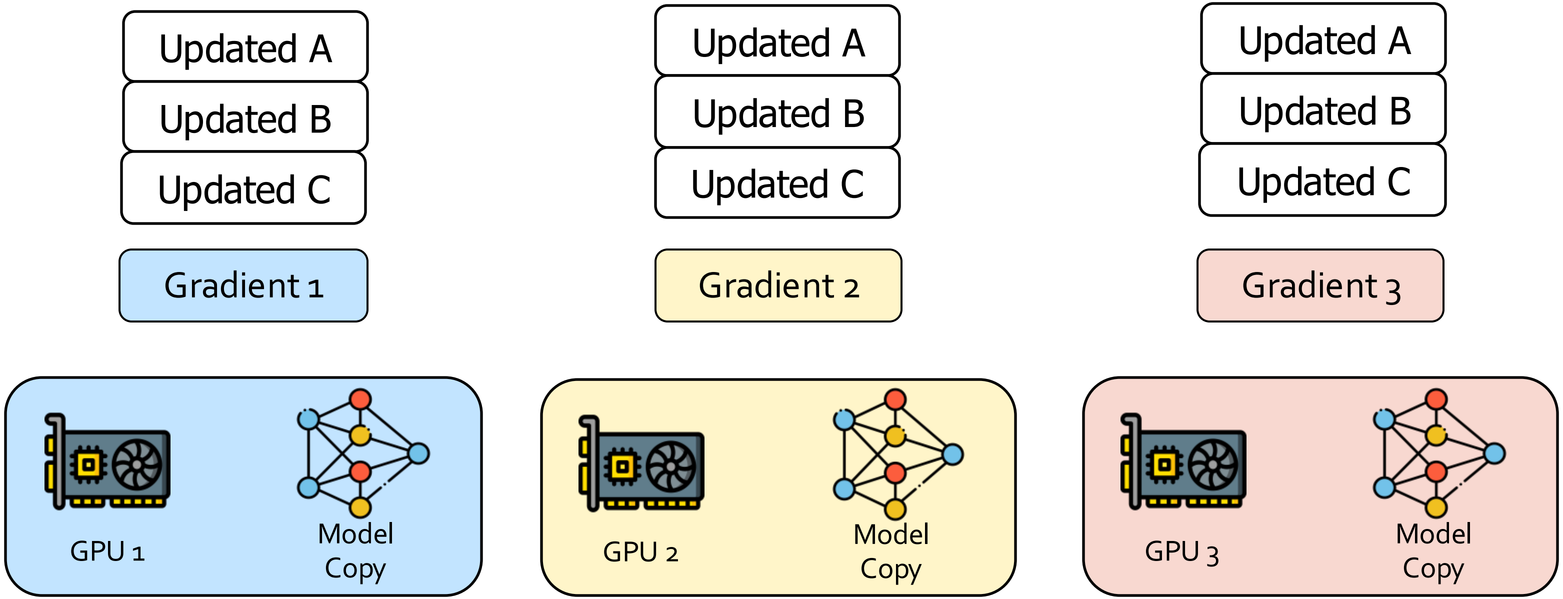
ZeRO Stage 1: How it works

GPU1 : update params A; GPU2: Updates Params B; GPU3: updates params C. GPU1 can only update params A since it only stores optimizer states of params A.



ZeRO Stage 1: How it works

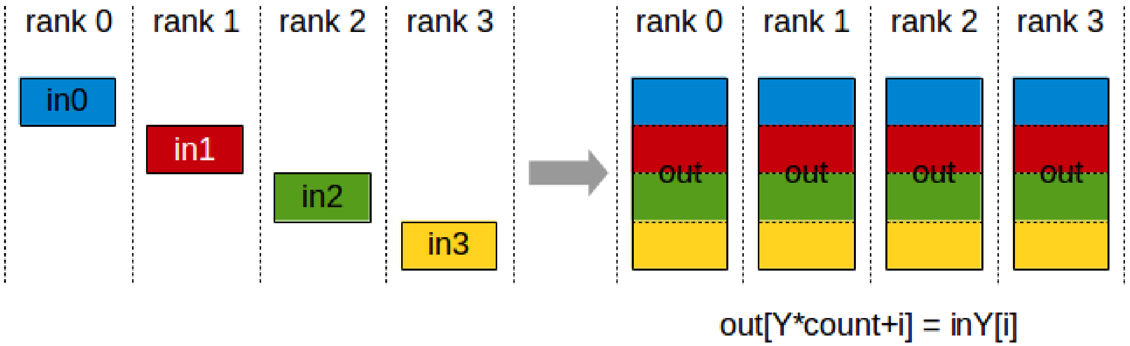
Each GPU sends updated params to every other GPU.
Finishing optimizer.step(). (all_gather in torch.distributed)



[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

Quiz: NCCL Operations: All Gather

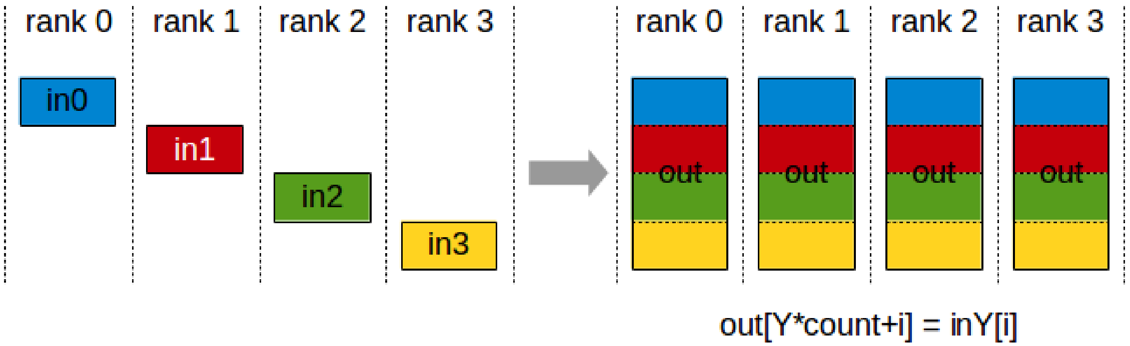
- all_gather: every GPU performs a ___?___ operation in parallel.



A.Reduce B. Broadcast C. Reduce_scatter

NCCL Operations: All Gather

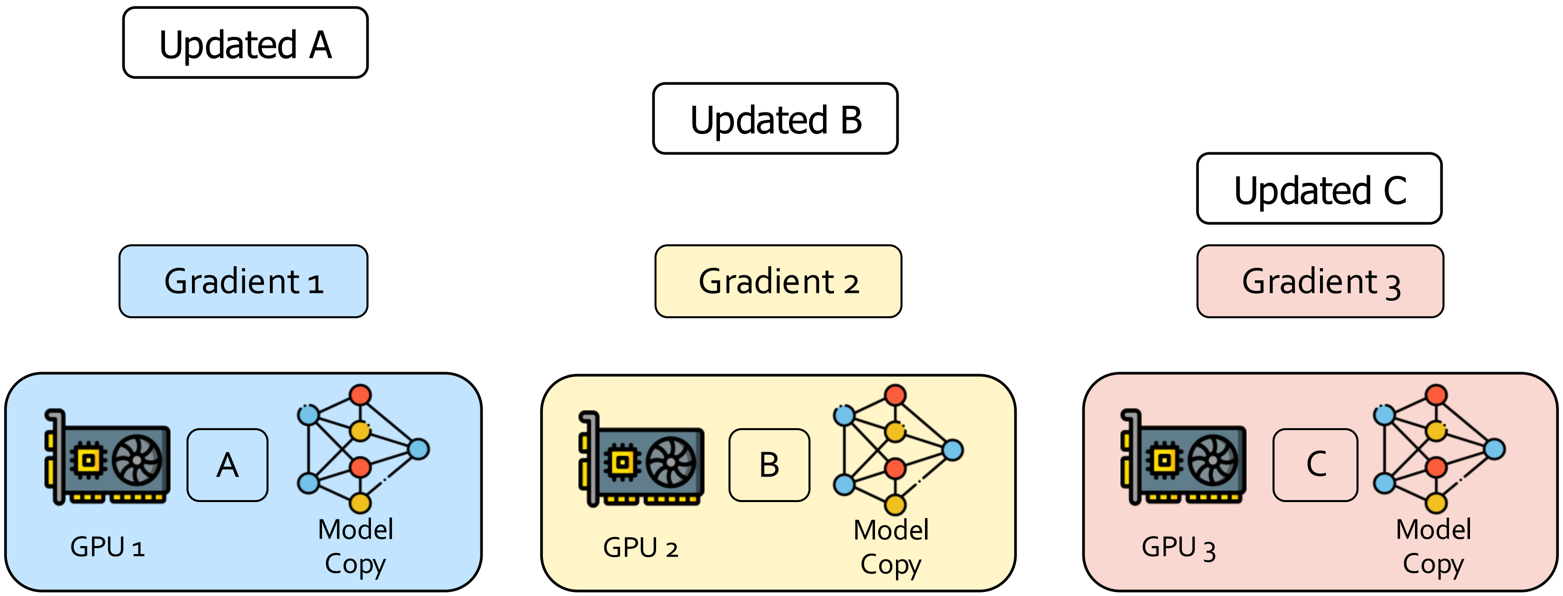
- all_gather: every GPU performs a ___broadcast___ operation in parallel.



A.Reduce B. Broadcast C. Reduce_scatter

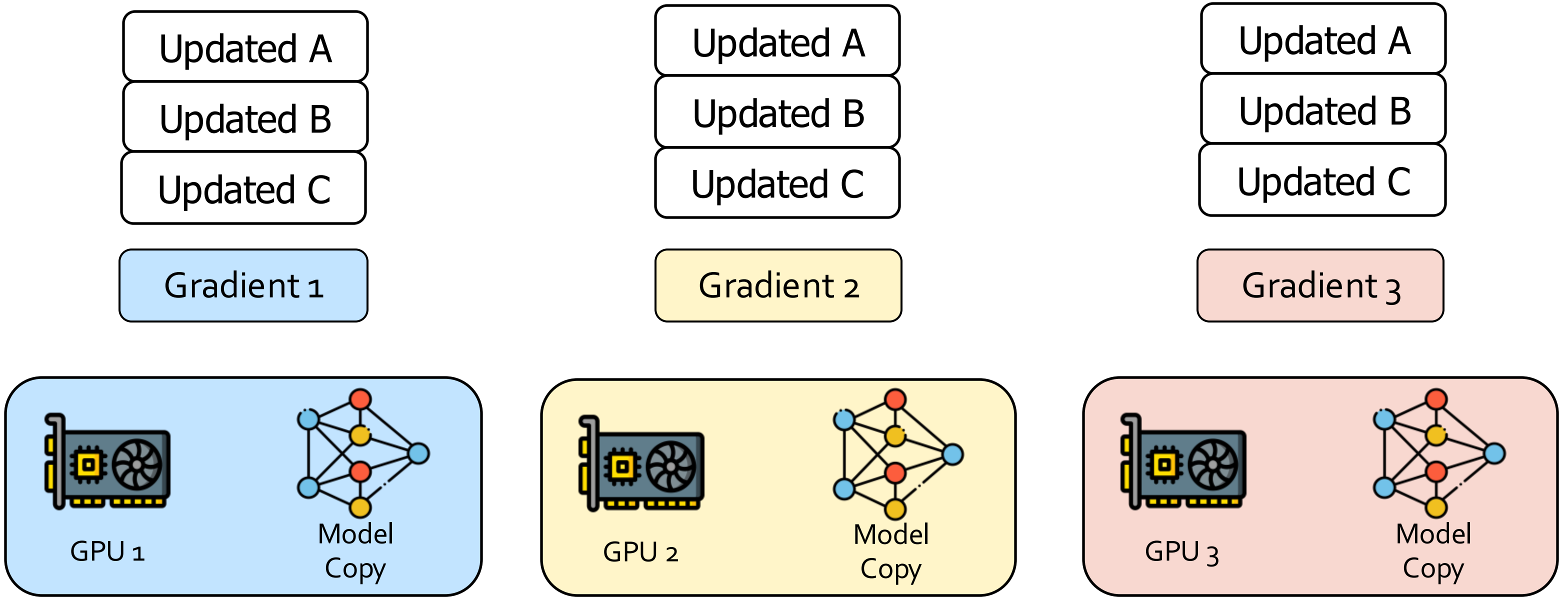
ZeRO Stage 1: How it works

Before all_gather



ZeRO Stage 1: How it works

After all_gather, every GPU has a updated copy of the model



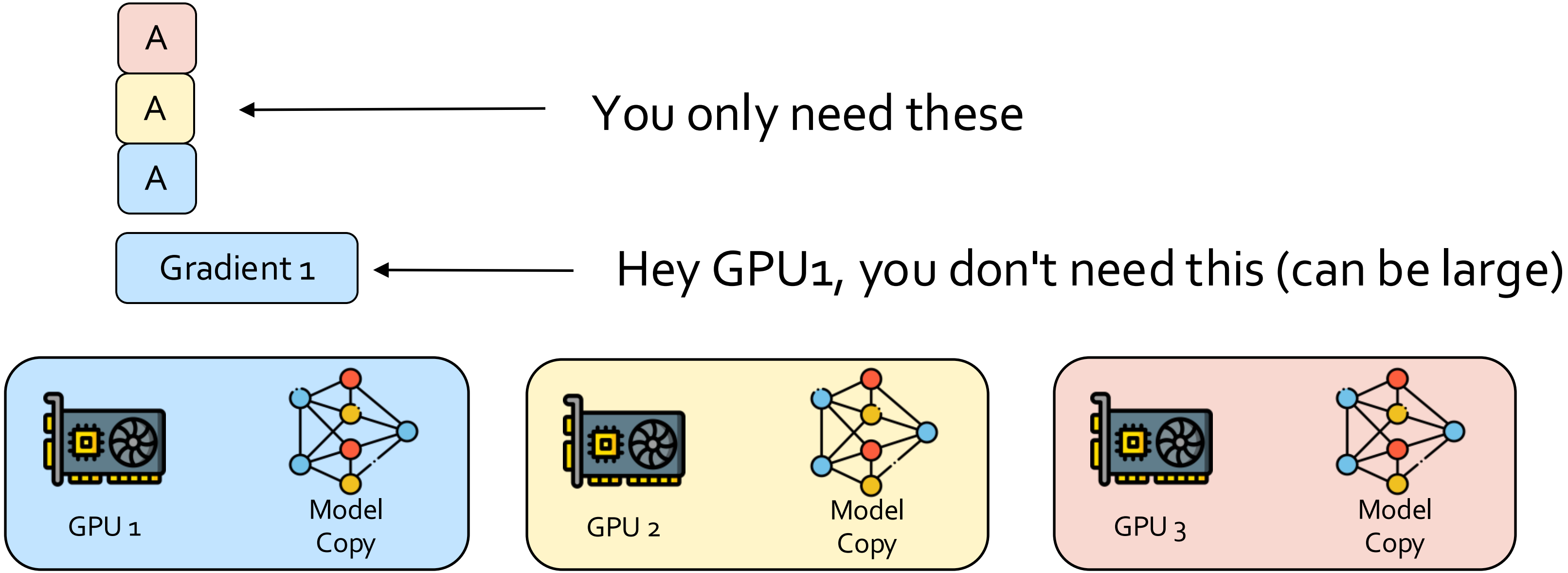
[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

Summary: ZeRO 1

- reduce_scatter on the gradients: splitting the gradients into different GPUs
- Each GPU individually perform gradient updates
- all_gather on updated parameters
- Basically free! (Compared to Naïve Data Parallelism)

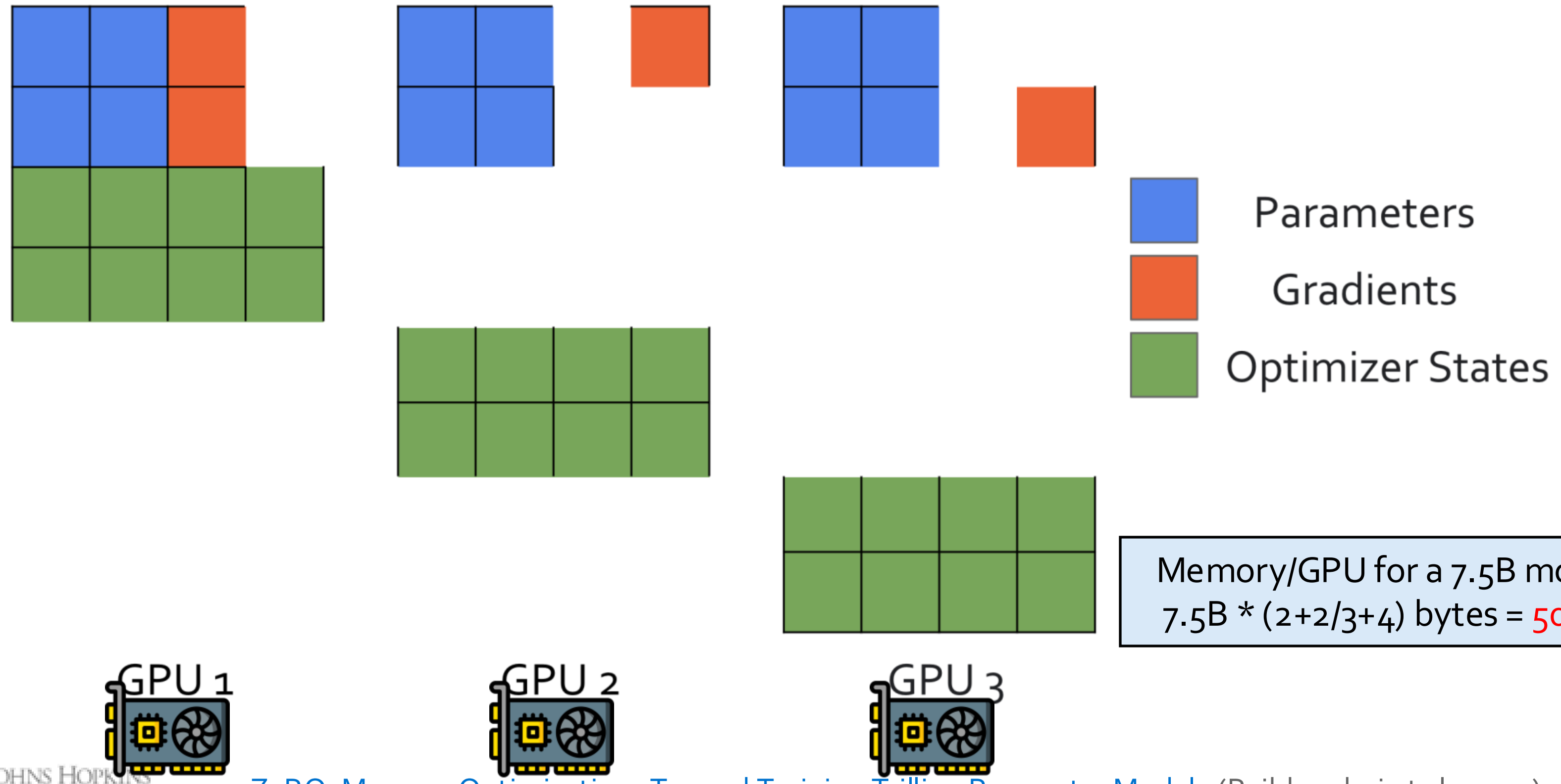
ZeRO Stage 1: How it works

Notice: Aside from the forward pass, GPU 1 only needs gradients A, but in fact it stores A and B and C



[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

ZeRO Stage 2: Sharding Gradients

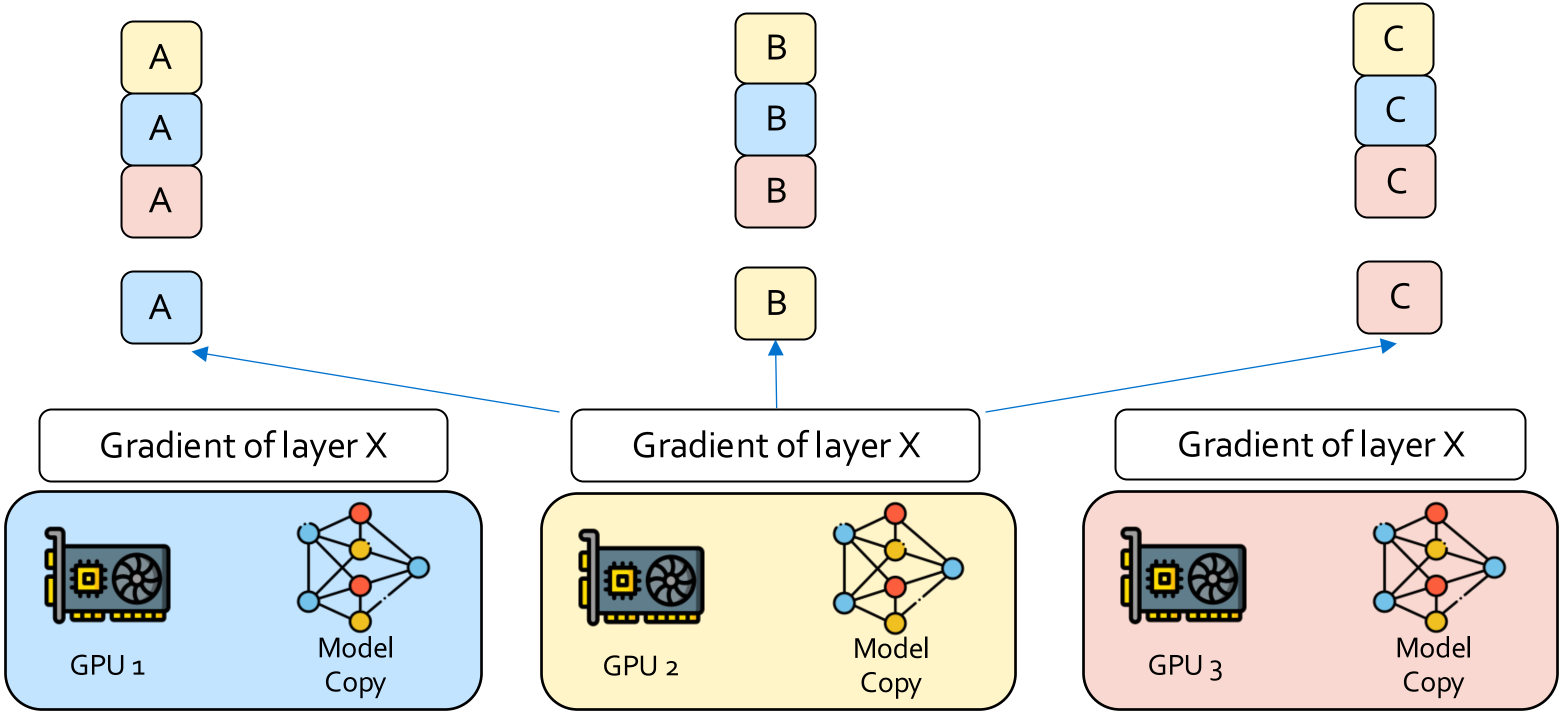


Memory/GPU for a 7.5B model:
7.5B * (2+2/3+4) bytes = 50 GB!

[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

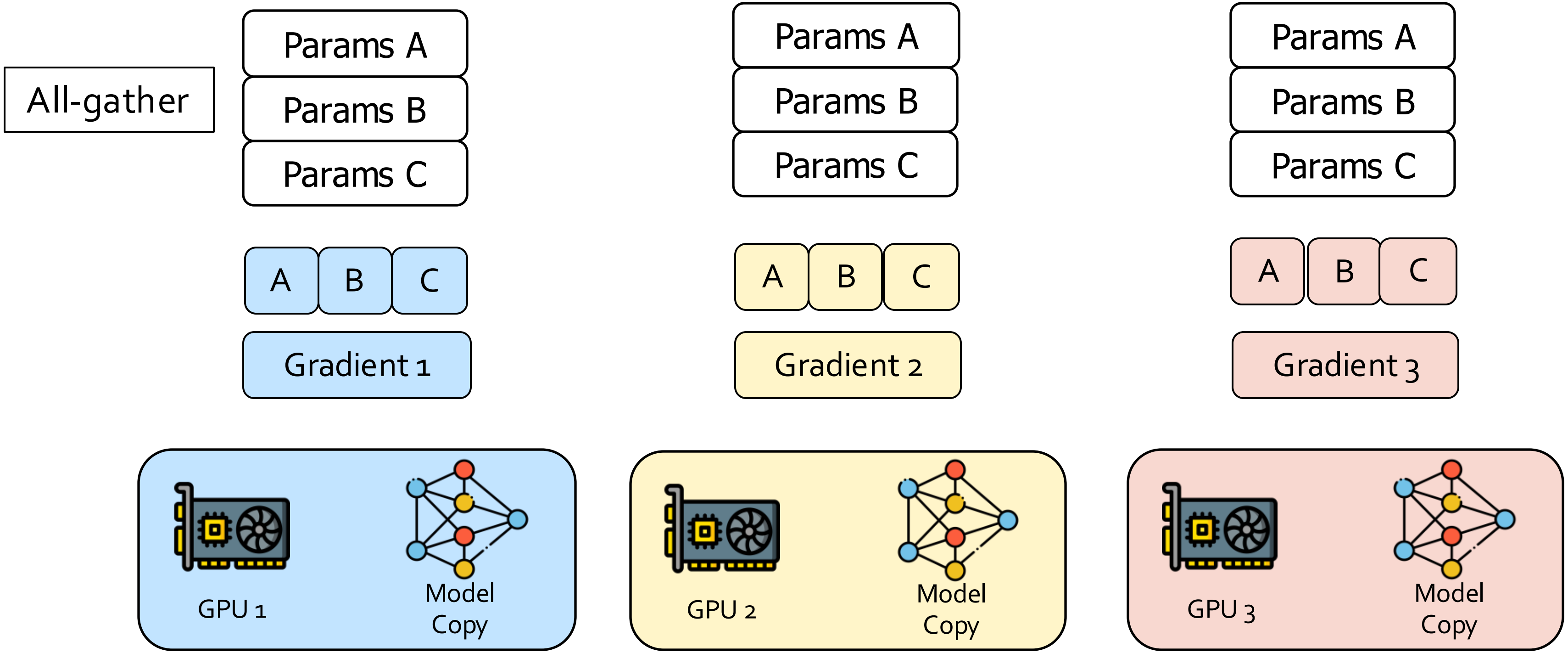
ZeRO Stage 2: How it works

Splitting the gradient of a **single layer** during backprop, then immediately shard it!



[ZeRO: Memory Optimizations Toward Training Trillion Parameter Models](#) (Rajbhandari et al., 2019)

ZeRO Stage 2: How it works



Summary: ZeRO ~~1~~2

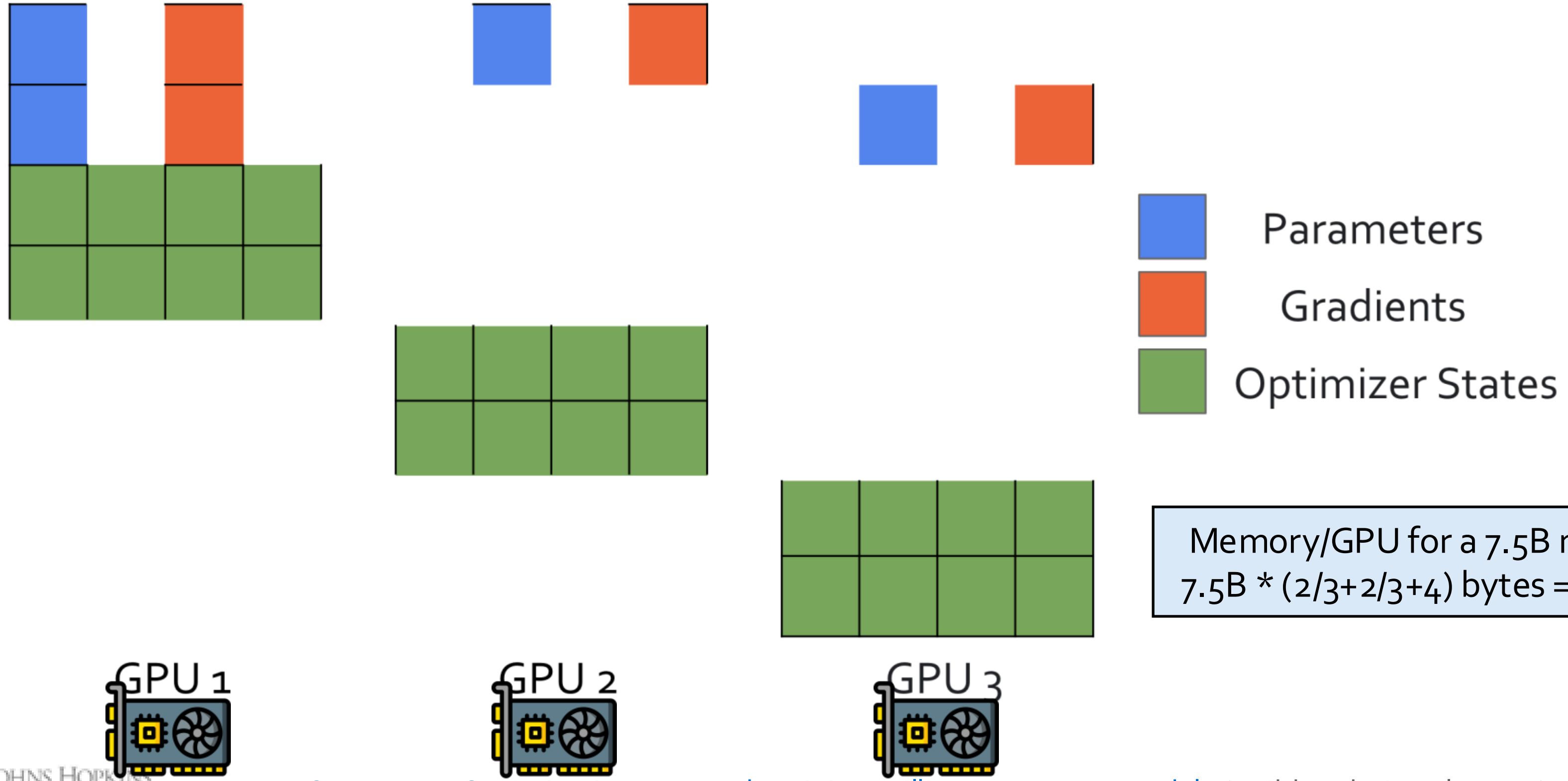
- ~~reduce_scatter on the gradients: splitting the gradients into different GPUs~~
- Calculate gradients layer by layer and perform reduce_scatter, once layer is done, free the gradient

- Each GPU individually perform gradient updates

- all_gather on updated parameters

- Almost free!

ZeRO-3 (aka FSDP): Shard Everything!

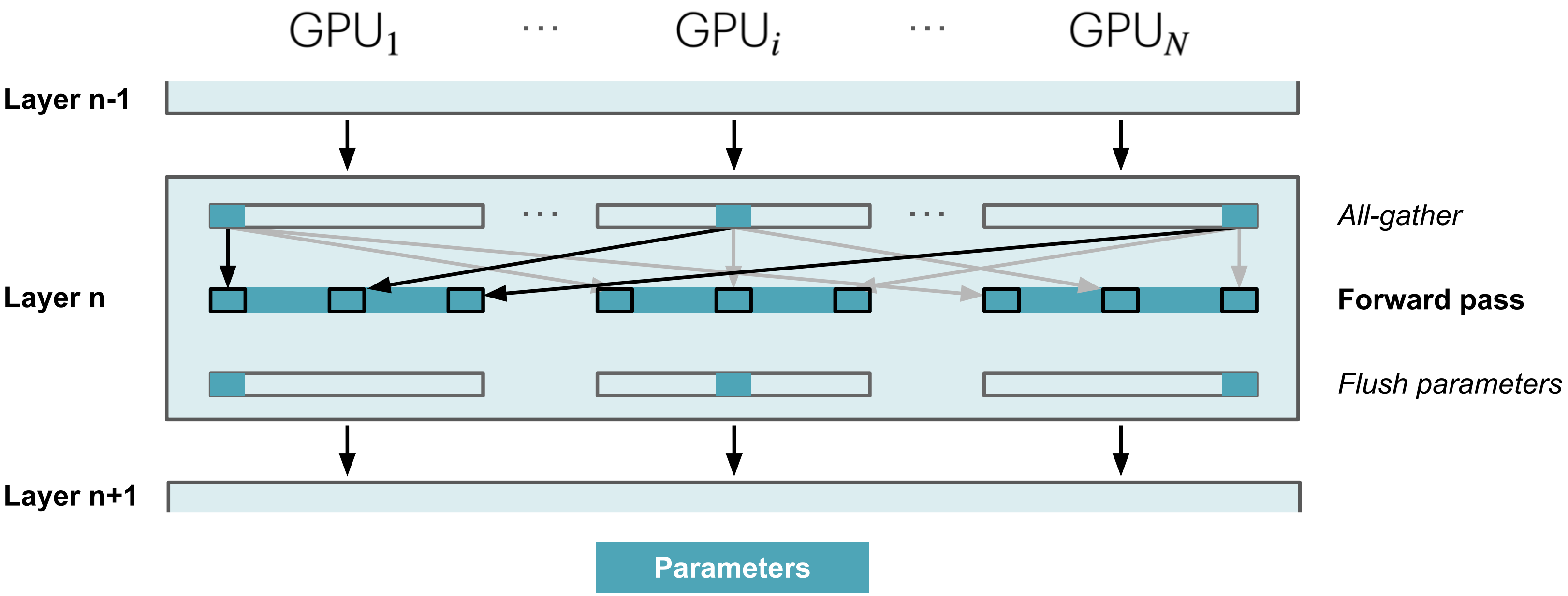


Memory/GPU for a 7.5B model:
7.5B * (2/3+2/3+4) bytes = 40 GB!

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

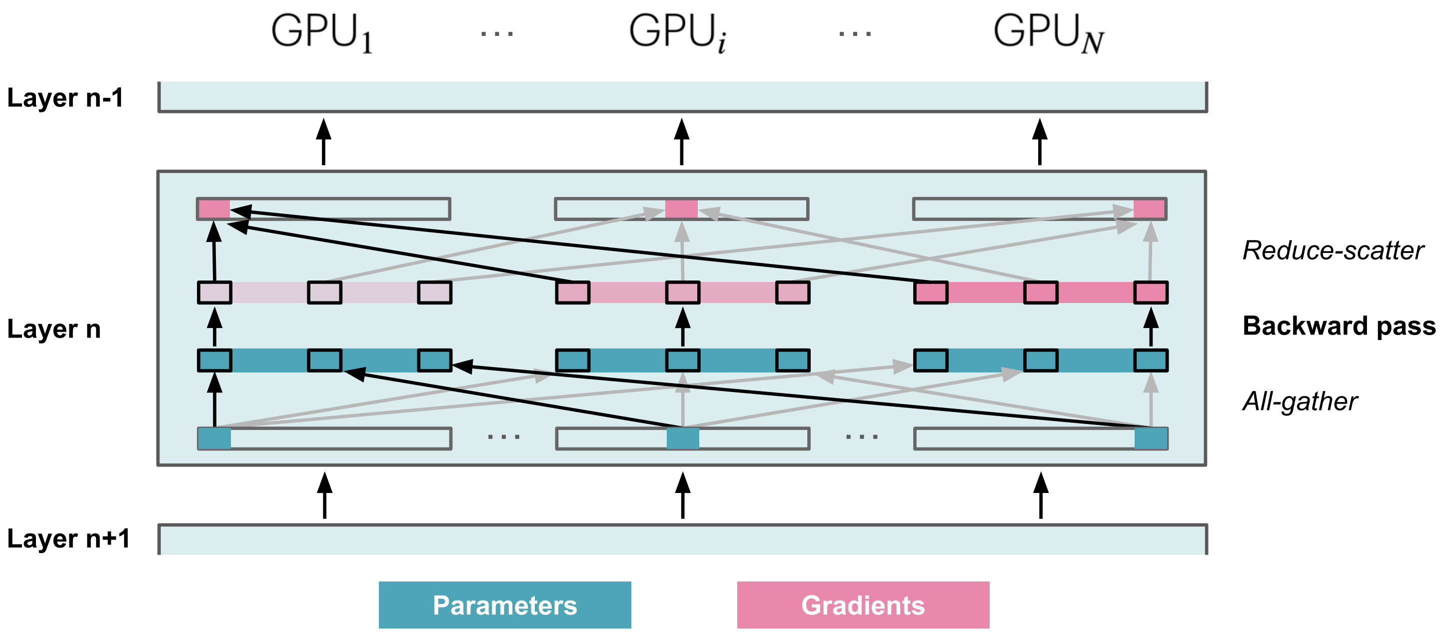
ZeRO Stage 3: How it works (simplified)

During forward pass, the parameters are gathered on-demand



ZeRO Stage 3: How it works (simplified)

During backward pass, the gradients are scattered (Reduce_Scatter)



Communication Costs

- Naïve Data Parallel: 2x parameter (all_reduce)

- ZeRO-1: 2x parameter (reduce_scatter + all_gather) - this is free! Might as well always use it.

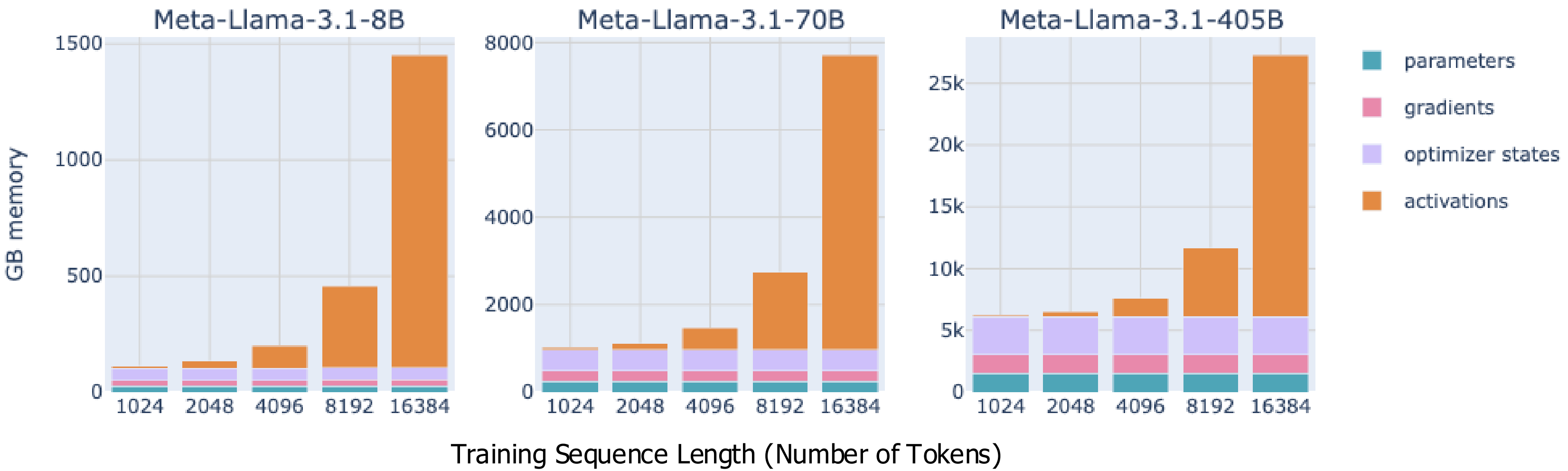
- ZeRO-2: 2x parameter (reduce_scatter + all_gather + overhead) - this is (almost) free!

- ZeRO-3: 3x parameter – which can be quite slow.

Where did all the memory go?

So far, we dealt with the optimizer states

but what about the activations?



Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Prefix Caching

but what about the activations?

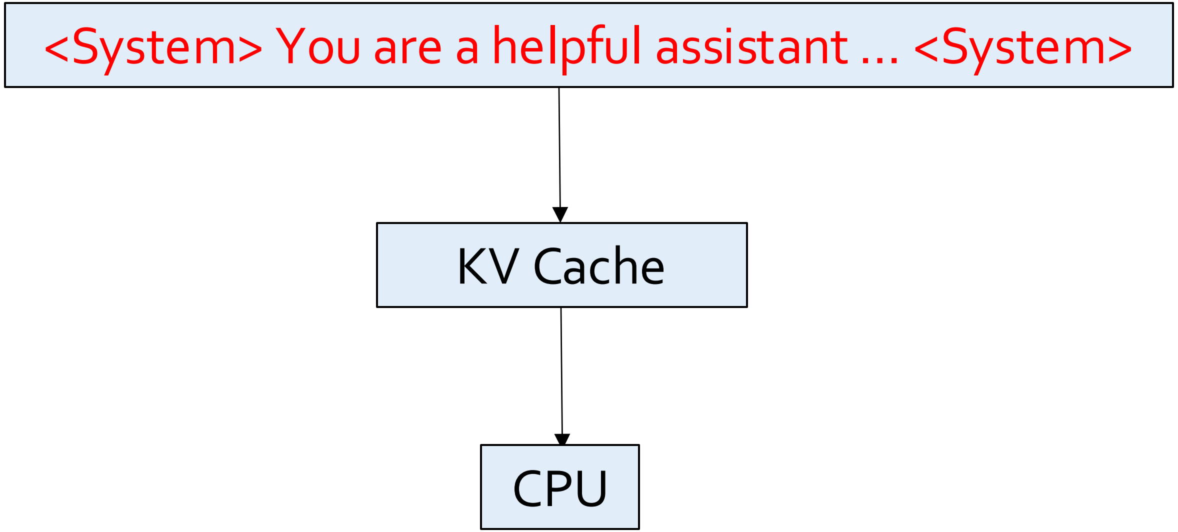
<System> You are a helpful assistant … <System>
<User> I want to know how can I use the coffee machine <User>

<System> You are a helpful assistant … <System>
<User> Write the code for training my language model. <User>

<System> You are a helpful assistant … <System>
<User> Help me revise my email … <User>

Prefix Caching

Storing the activations in CPU and retrieve it when needed.



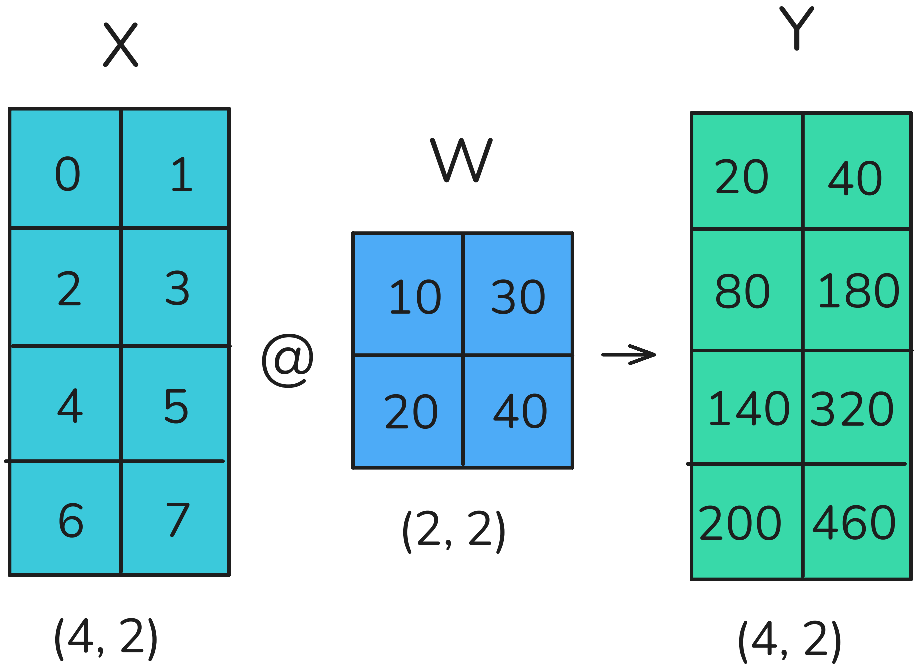
But, can we slice the activations to fit them in different GPUs?

- Yes, by Tensor Parallelism

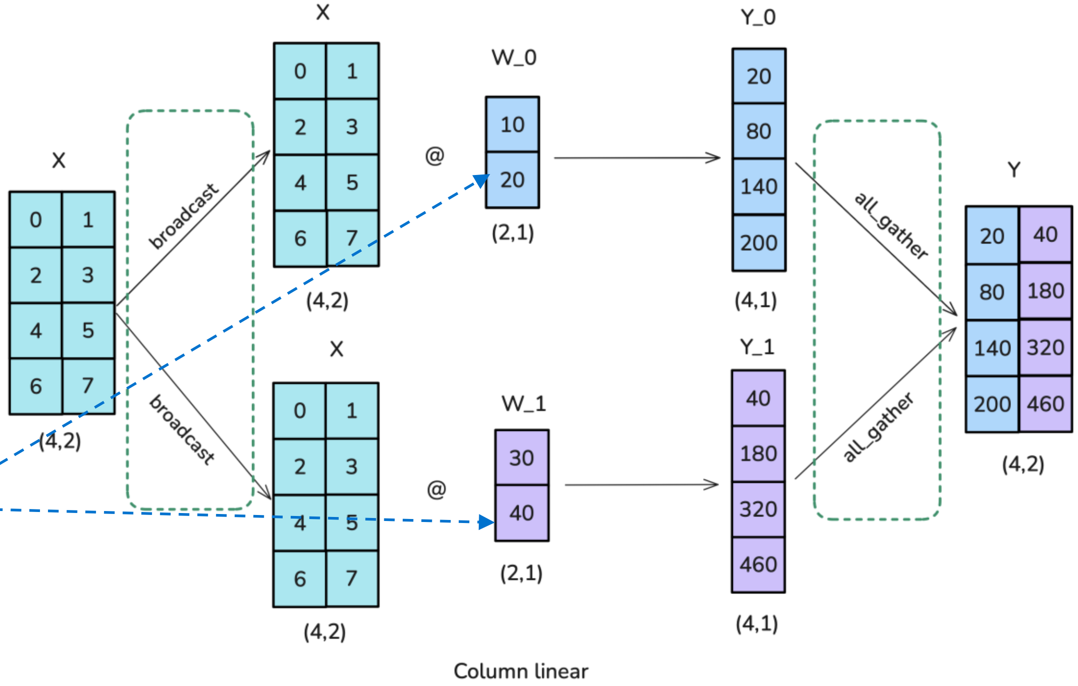
Tensor Parallelism

We can either cut the weights W into two columns (Column Parallelism)

or into two rows (Row Parallelism)



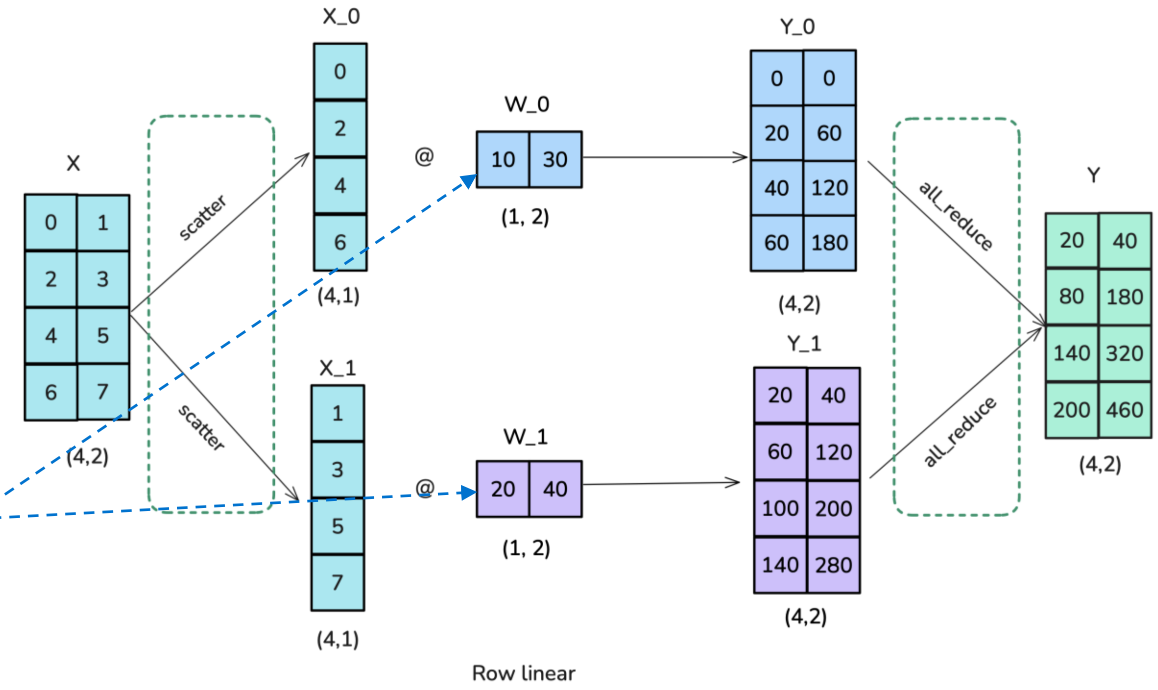
Column-wise Tensor Parallelism



Cuts the weight matrix W into 2 columns

Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

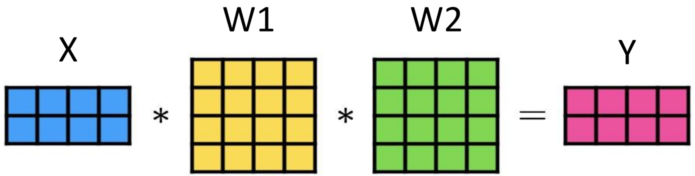
Row-wise Tensor Parallelism



Cuts the weight matrix W into 2 rows

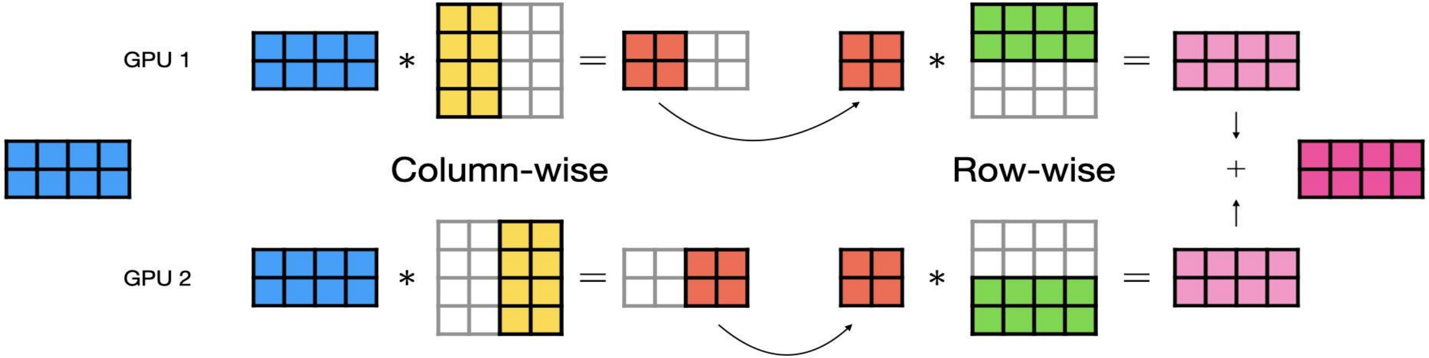
Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Tensor Parallelism



Computing matrix multiplications without storing internal activations (e.g. xW1)

In Feed-Forward Networks, The dimension of W1 is usually 4x the hidden dimension.



Tensor Parallelism: Llama Feed-Forward

```python
    self.w1 = ColumnParallelLinear(
        dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
    )
    self.w2 = RowParallelLinear(
        hidden_dim, dim, bias=False, input_is_parallel=True, init_method=lambda x: x
    )
    self.w3 = ColumnParallelLinear(
        dim, hidden_dim, bias=False, gather_output=False, init_method=lambda x: x
    )

def forward(self, x):
    return self.w2(F.silu(self.w1(x)) * self.w3(x))
```

activations are element-wise operations, can be parallelized

Source: https://github.com/meta-llama/llama/blob/main/llama/model.py

Tensor Parallelism: Llama Attention

Column Parallel for Query, Key and Vector and Row Parallel for attention output

```
self.wq = ColumnParallelLinear(
    args.dim,
    args.n_heads * self.head_dim,
    bias=False,
    gather_output=False,
    init_method=lambda x: x,
)
self.wk = ColumnParallelLinear(
    args.dim,
    self.n_kv_heads * self.head_dim,
    bias=False,
    gather_output=False,
    init_method=lambda x: x,
)
self.wv = ColumnParallelLinear(
    args.dim,
    self.n_kv_heads * self.head_dim,
    bias=False,
    gather_output=False,
    init_method=lambda x: x,
)
```

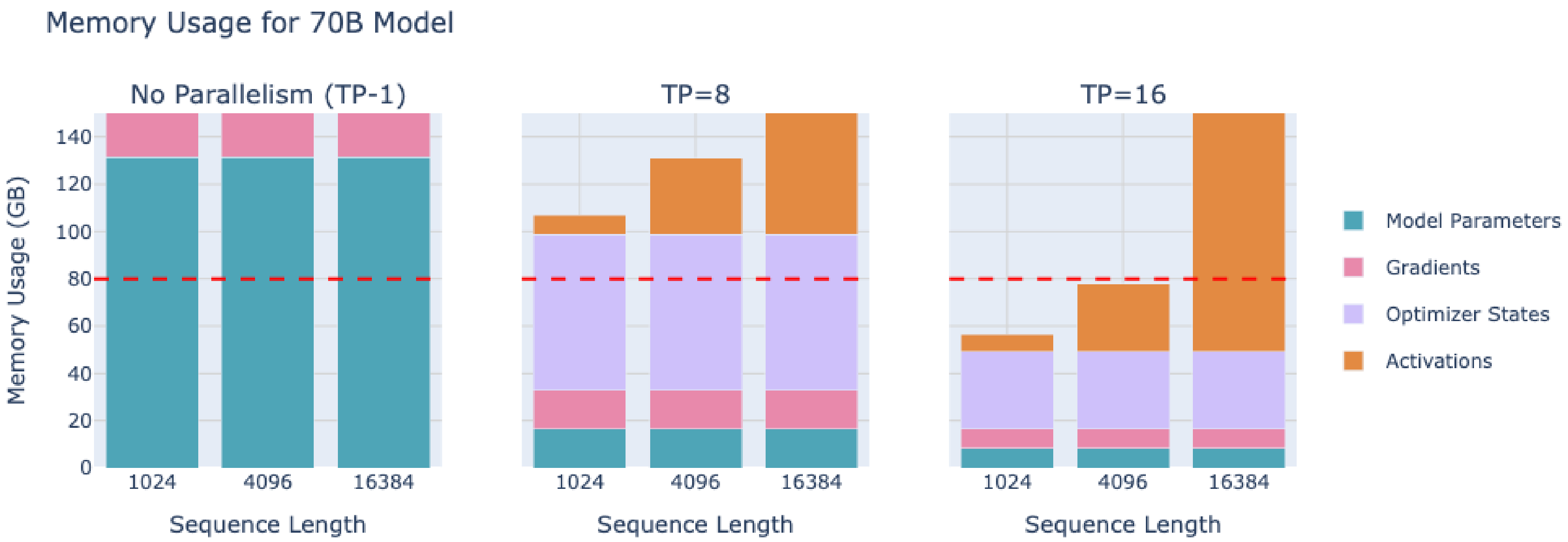
```
self.wo = RowParallelLinear(
    args.n_heads * self.head_dim,
    args.dim,
    bias=False,
    input_is_parallel=True,
    init_method=lambda x: x,
)
```

Source: https://github.com/meta-llama/llama/blob/main/llama/model.py

Summary so far

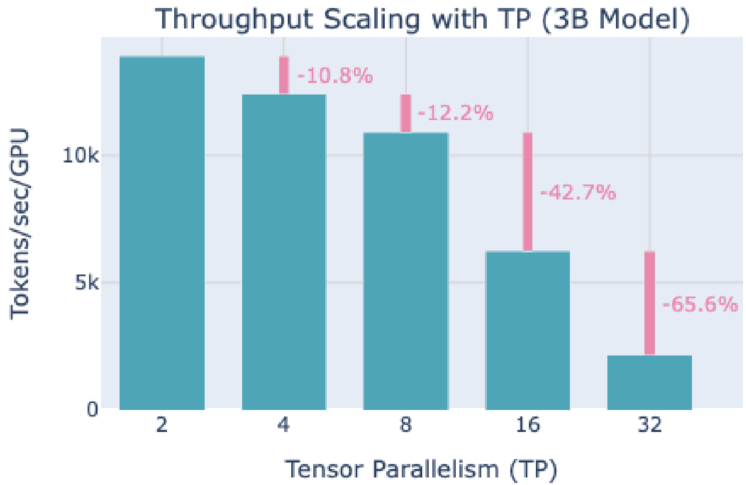
- Data Parallelism
  - Naïve Data Parallelism
  - NCCL Operations
    (reduce, all_reduce, reduce_scatter, broadcast, all_gather)
  - ZeRO-1, ZeRO-2, ZeRO-3
- Prefix Caching
- Tensor Parallelism
  - Row-wise Tensor Parallelism
  - Column-wise Tensor Parallelism

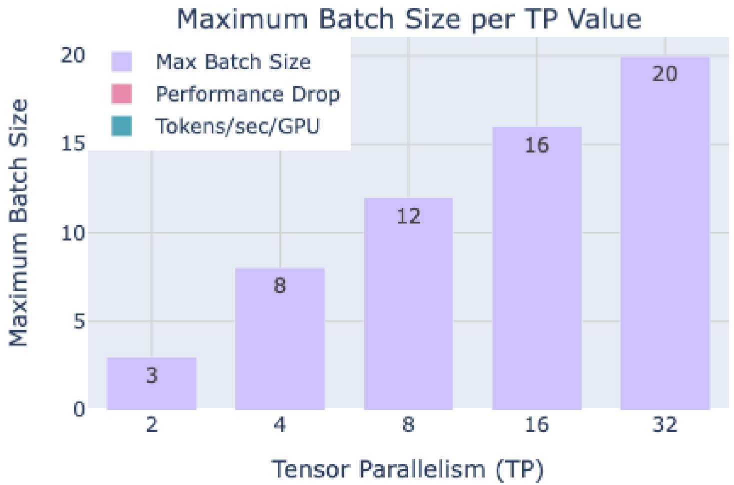
Tensor Parallelism



Source: https://github.com/meta-llama/llama/blob/main/llama/model.py

Throughput Scaling of Tensor Parallelism

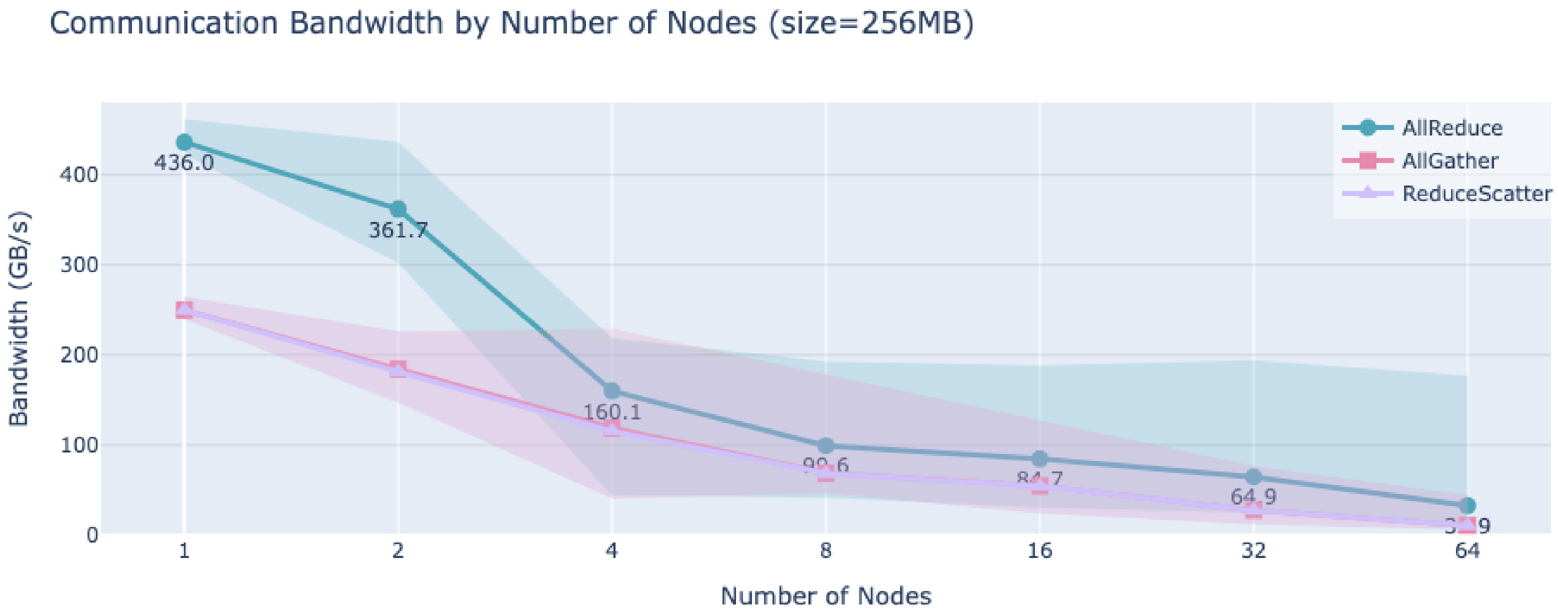




A large drop in throughput when scaling beyond 8 GPUs (one node)

Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

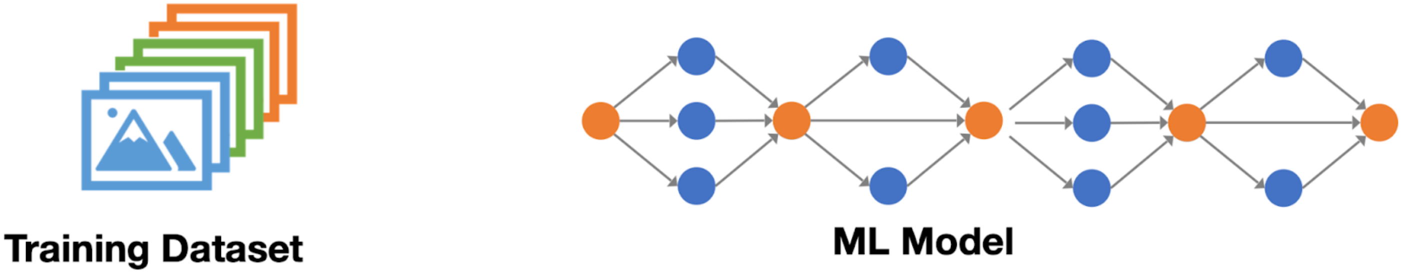
Throughput Scaling of Tensor Parallelism



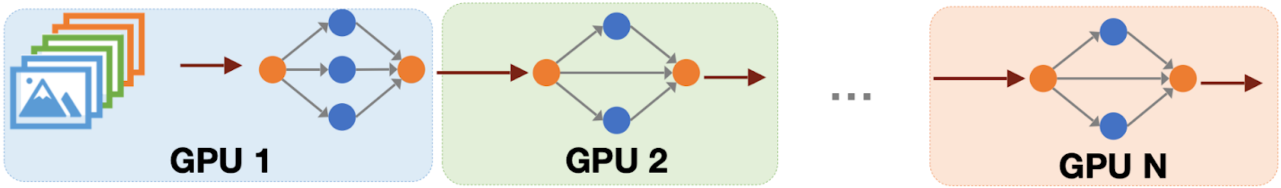
Throughput drops significantly once we go beyond one node!

Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Pipeline Parallelism



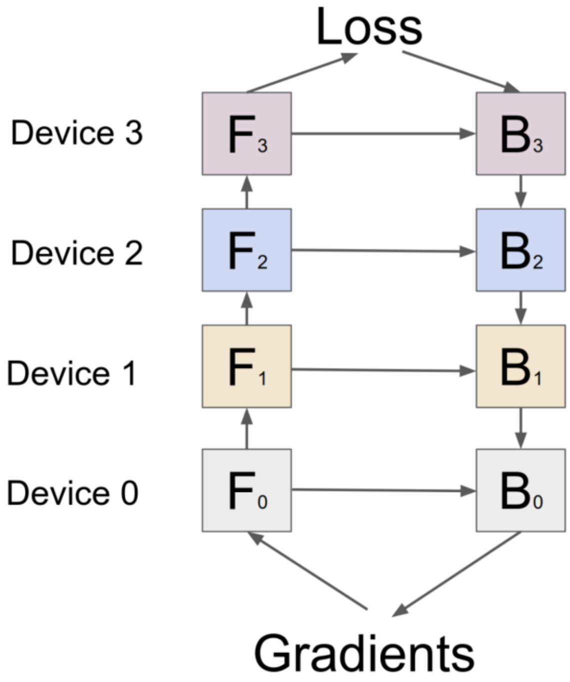
Shard each layer of the model into individual GPUs:
Prevents the cost of syncing params

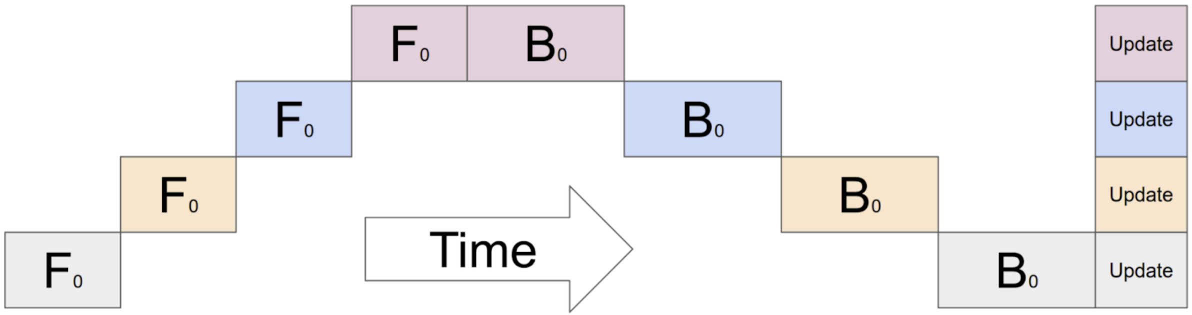


Credit: Song Han (MIT)

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Pipeline Parallelism



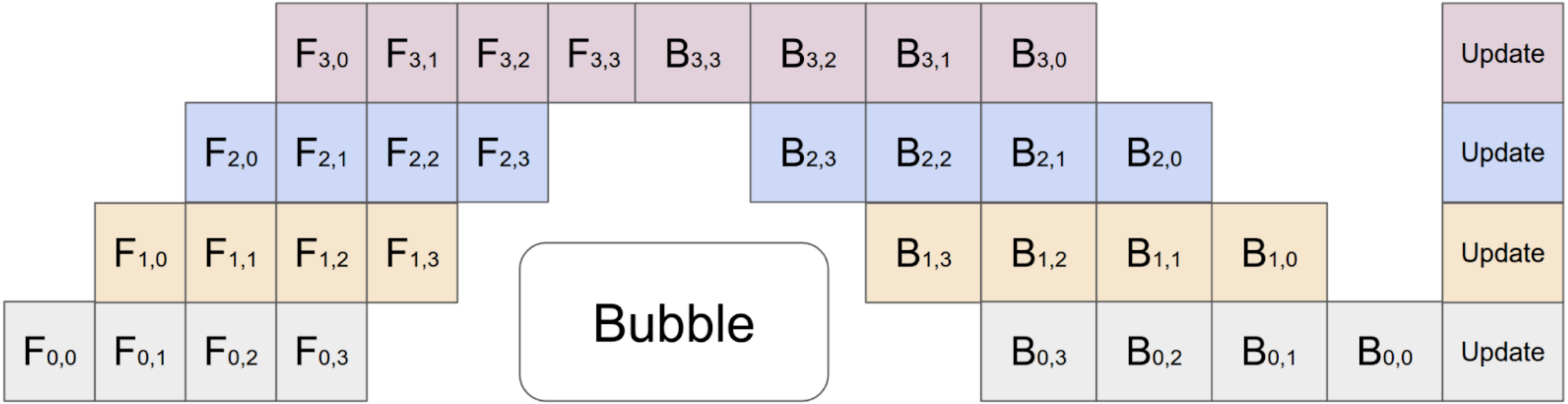


Each GPU is only working for 1/PP = ¼ of the time!
Idle/Work ratio = pp – 1 = 3

[GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism](#) (Huang et al., NeurIPS 2019)

Pipeline Parallelism: Improvement

Solution: Splitting the data into mini-batches! (AFAB)



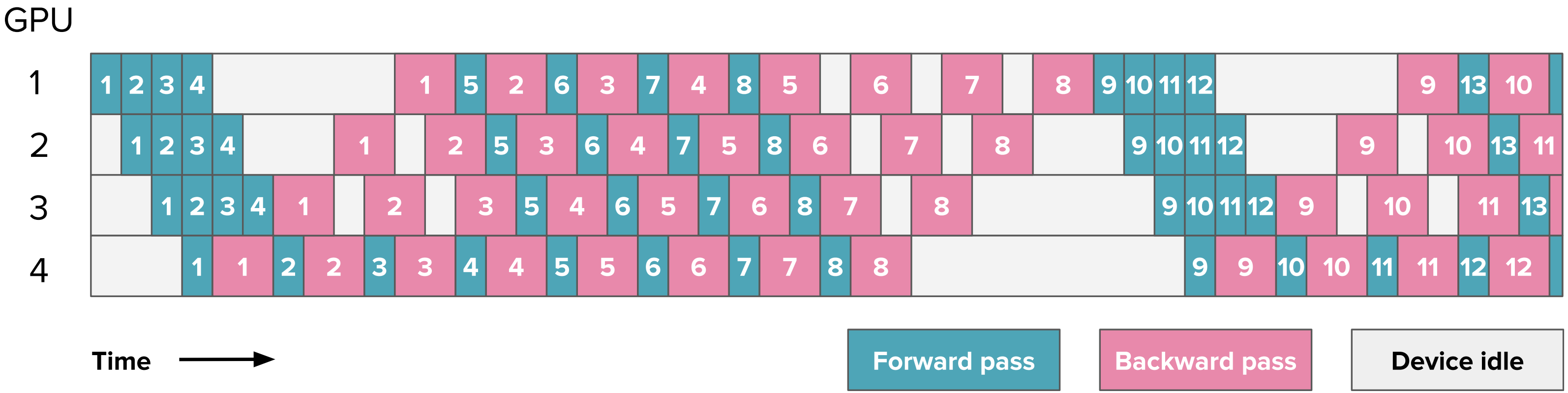
Idle / Work Ratio = PP-1 / M= 3 / 4

GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism (Huang et al., NeurIPS 2019)

Pipeline Parallelism

A cleverer version of AFAB: 1 Forward 1 Backward (1F1B)

Idea: Do backward as early as possible, releasing activations on the fly



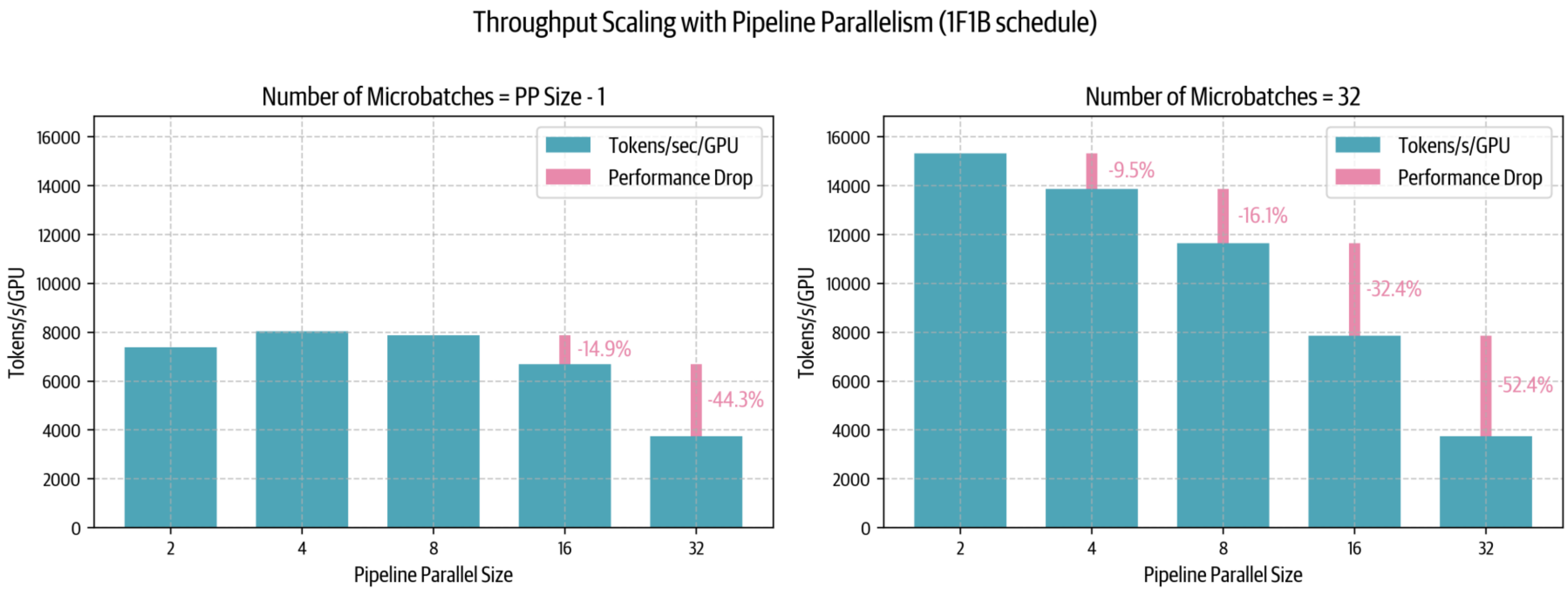
Roughly the same Idle/Work Ratio but less memory

(as you only need to store p=4 activations rather than m=8)

GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism (Huang et al., NeurIPS 2019)

JOHNS HOPKINS WHITING SCHOOL of ENGINEERING

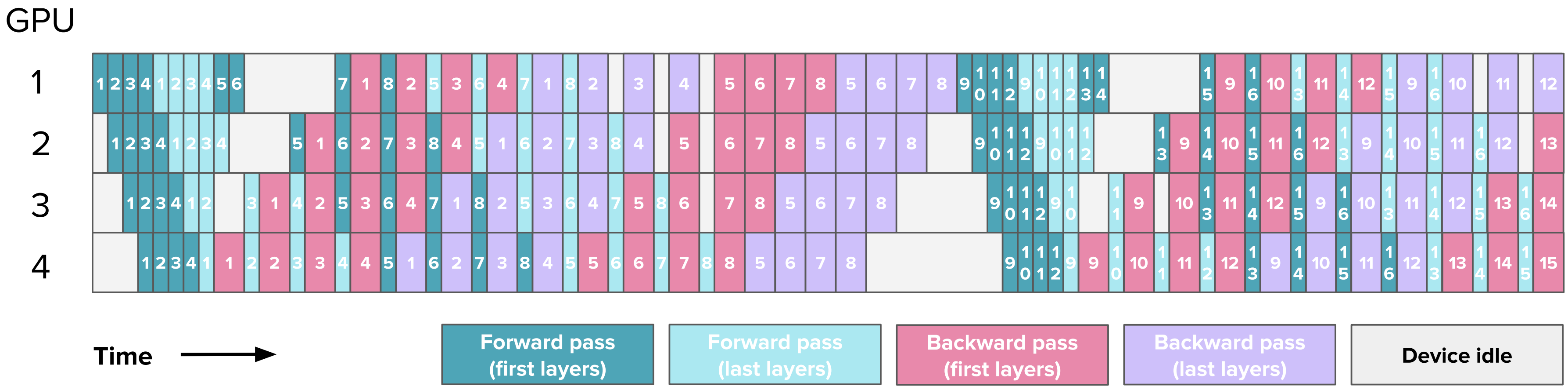
Pipeline Parallelism Throughput



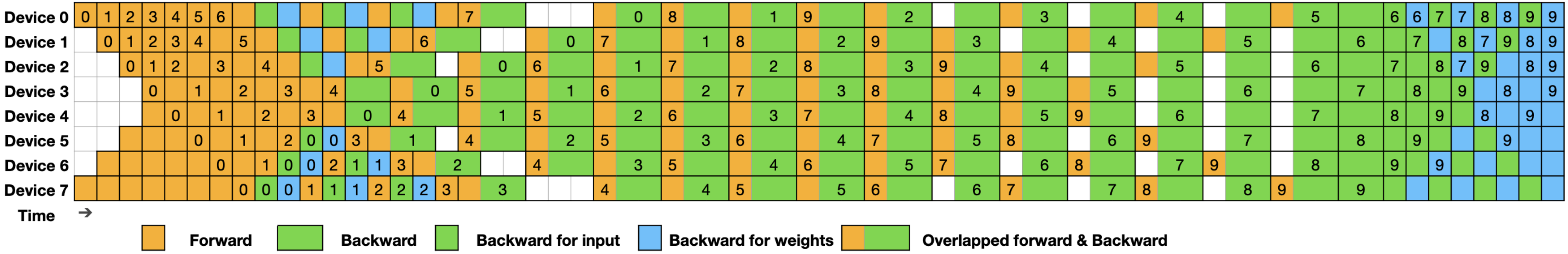
A small drop in throughput when scaling beyond 8 GPUs (one node) but a large drop as we increase the microbatch number

Johns Hopkins Whiting School of Engineering

Interleaving Pipeline Parallelism (LLama3)



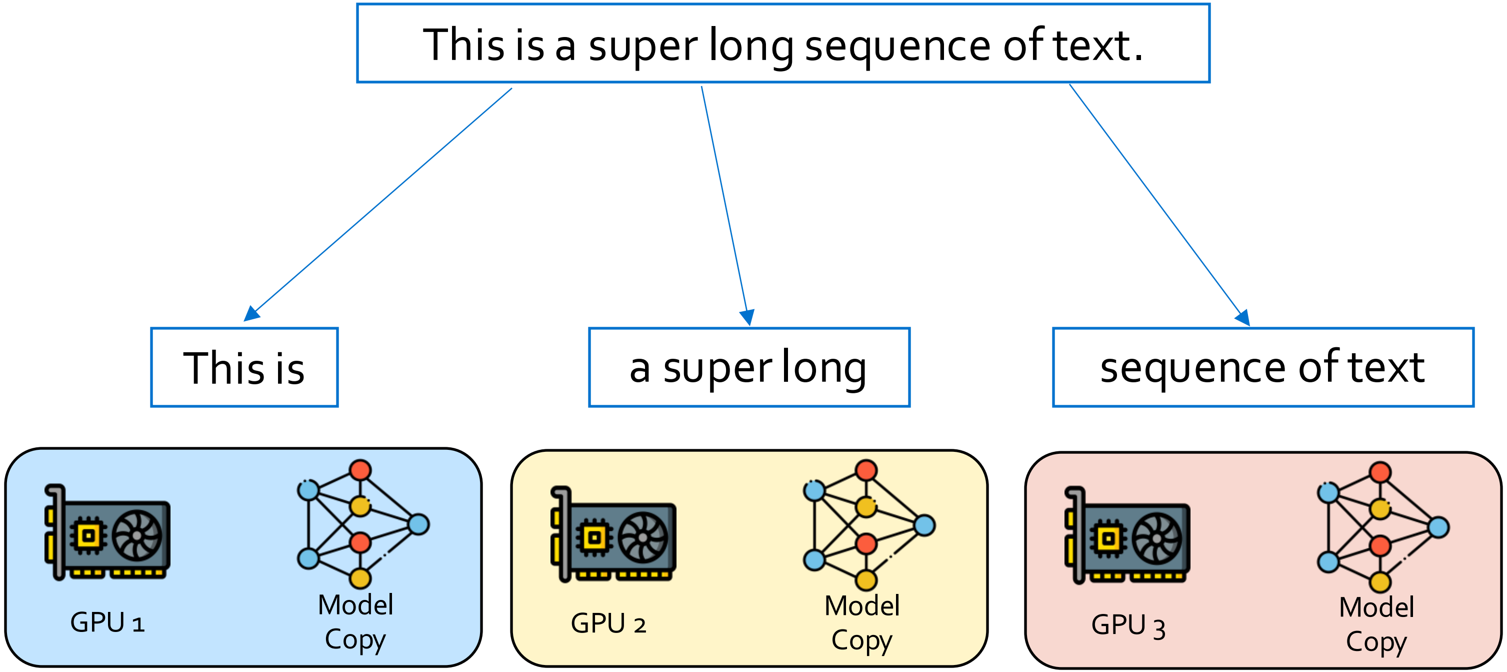
Interleaved Pipeline Parallelism (DeepSeek)



backprop for weights (blue) can be computed at any time!
We fill in the bubble with weight back propagation.

What about (super) long sequences?

Suppose we want to split the sequence into different GPUs

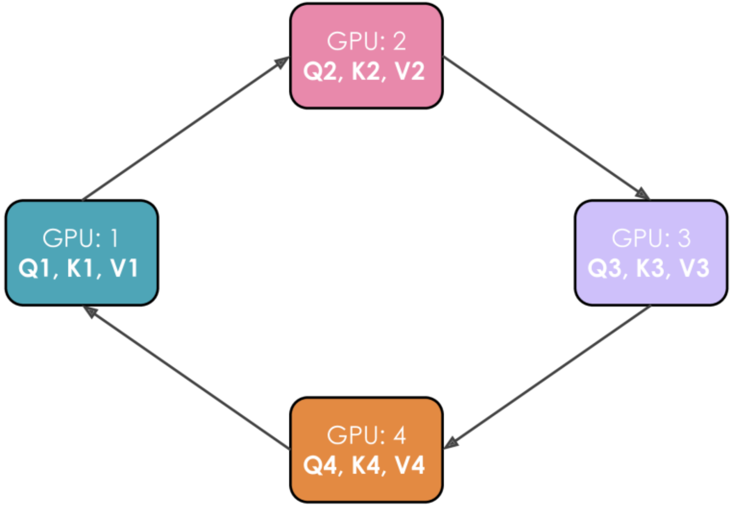


What about (super) long sequences?

- Feed Forward Network / LayerNorm is not affected by splitting the sequence, each token is processed individually

- But what about attention? Each token needs to compute dot product with every other token.

Context Parallelism (Ring Attention)

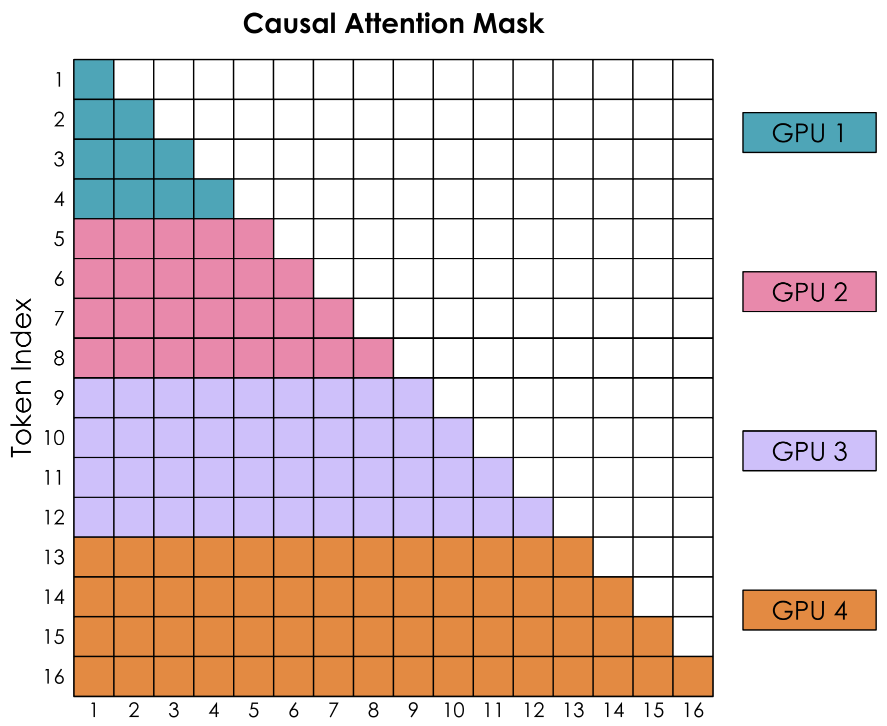


Just pass the Key, Value pairs around!

However, attention mask is usually causal – Q1 does not need K2, V2, ...

Source: Ring Attention with Blockwise Transformers For Near-Infinite Context (Liu et al., 2023)

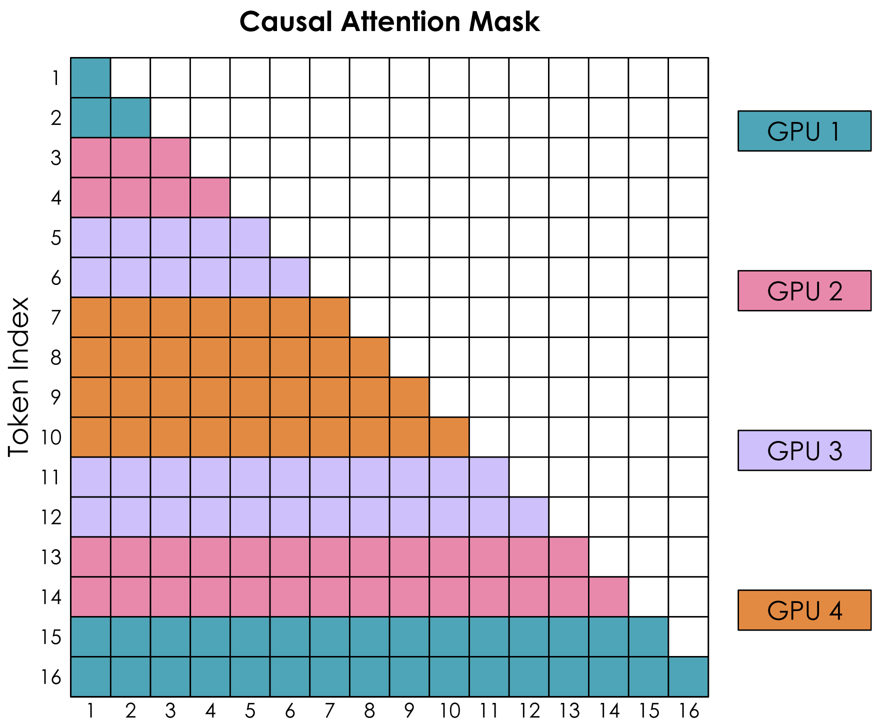
Context Parallelism (Ring Attention)



GPU 1 computes the pre-softmax-ed scores for Q1, Q2, Q3, Q4.. then becomes idle.

Source: Ring Attention with Blockwise Transformers For Near-Infinite Context (Liu et al., 2023)

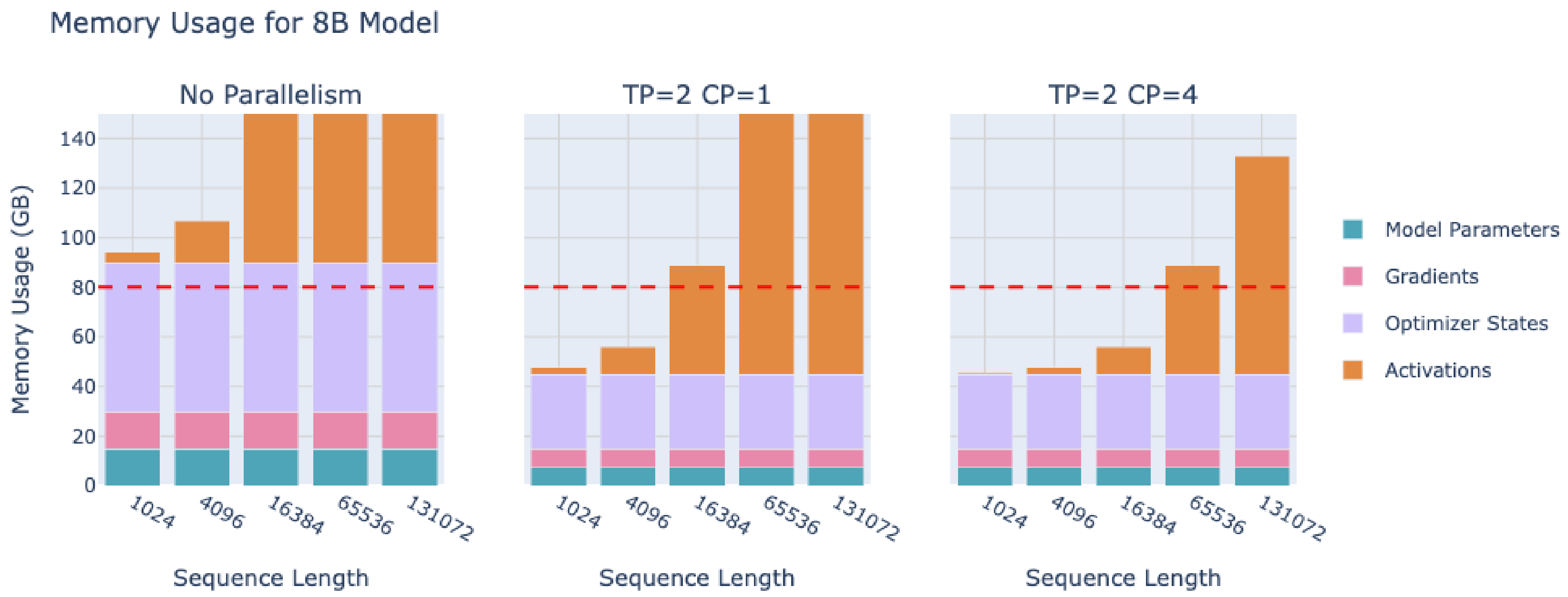
Context Parallelism (Ring Attention)



Balancing the workload for each individual GPUs.

Source: Striped Attention: Faster Ring Attention for Causal Transformers (Brandon et al., 2023)

Context Parallelism (Ring Attention)



Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

Summarizing

	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

Source: Tatsunori Hashimoto (Stanford)

Solutions

- DeepSeek V3: DP=1, PP=16, EP (Expert Parallelism) = 8

3.2 Training Framework

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).

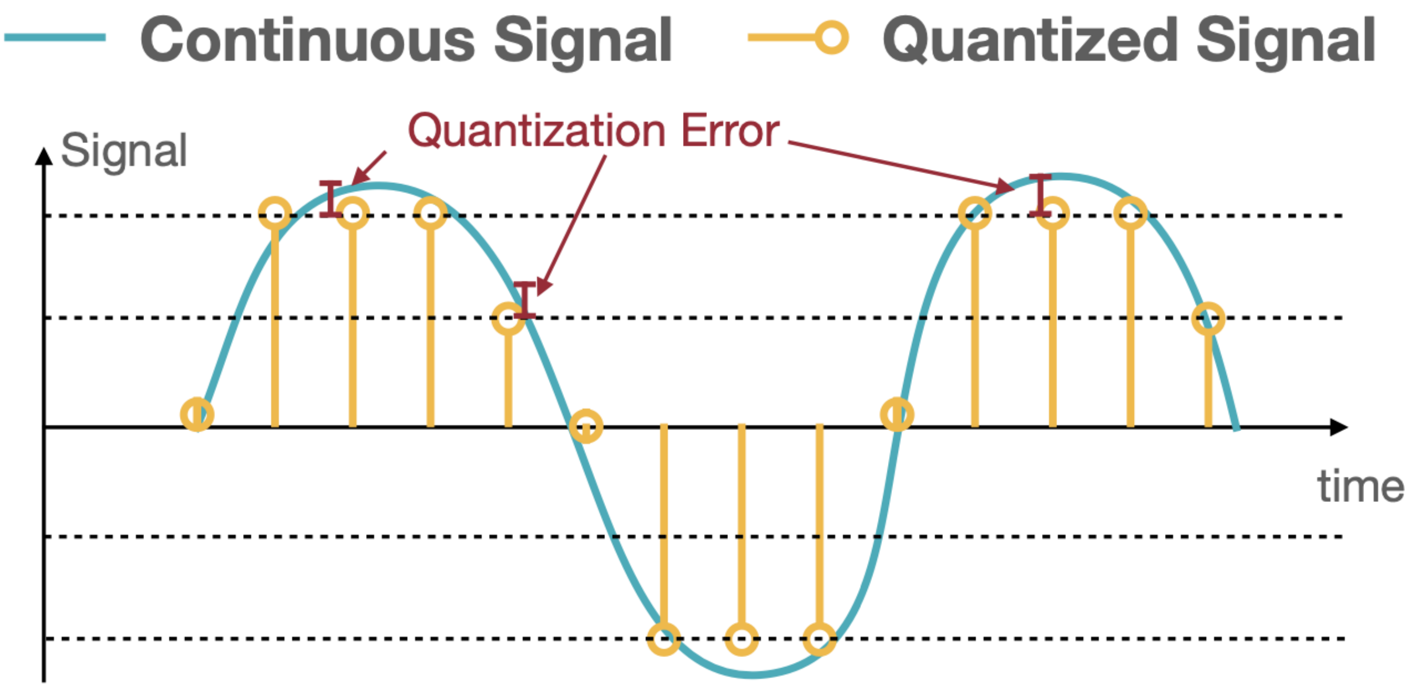
- Llama 3: Staged Training

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%

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

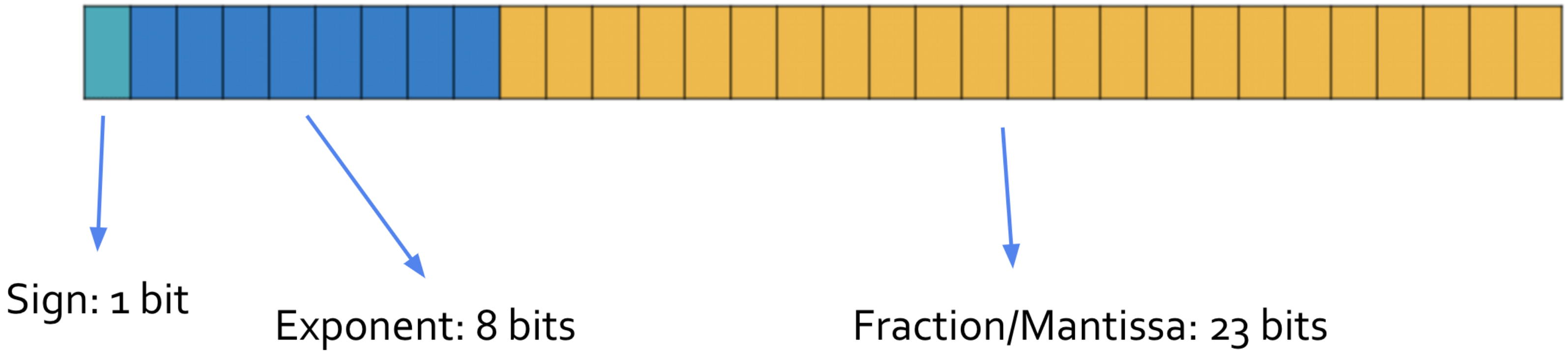
Quantization

Quantization: Mapping from high to low precision



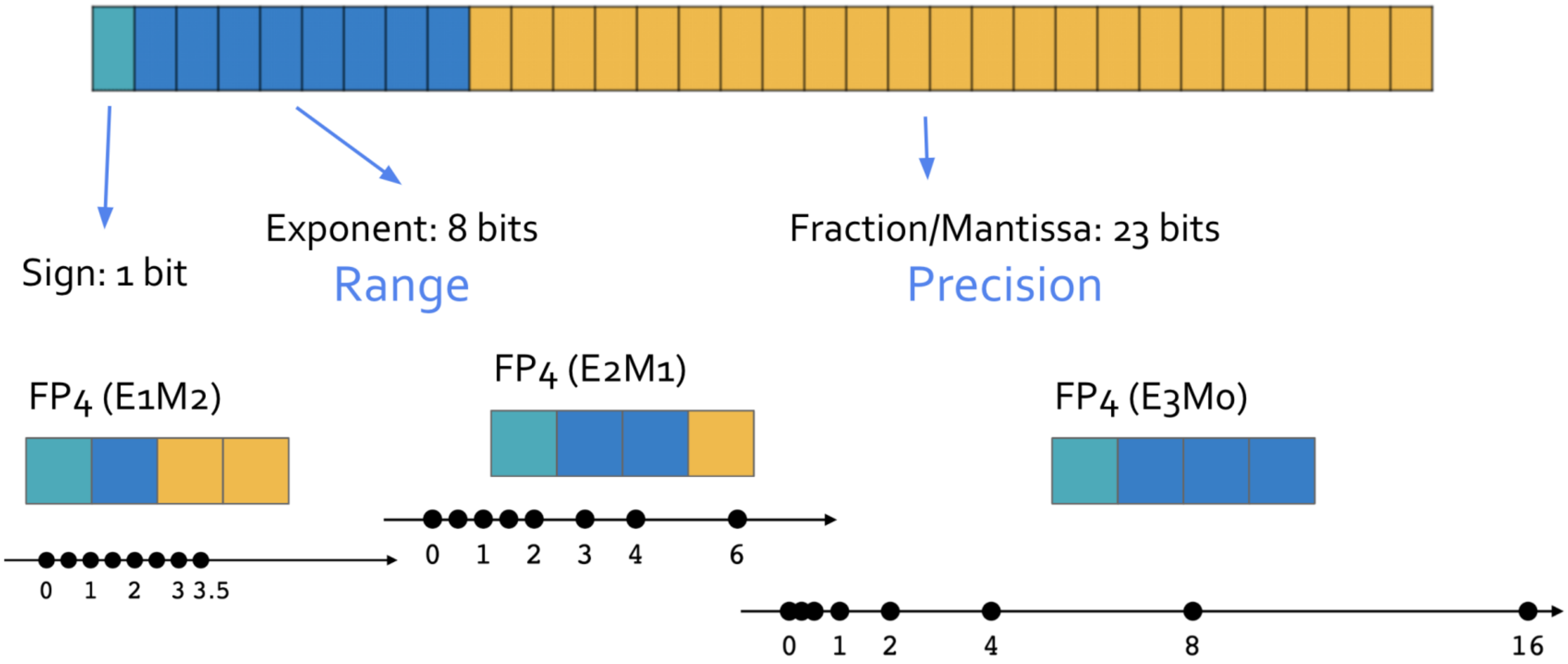
Numeric Data Types

- Example: 32-bit floating point number in IEEE 754 (FP32)



$$\text{Number} = (-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent - 127}}$$

Floating Point Numbers



Floating Point Numbers

	Exponent	Fraction
IEEE 754 Single Precision 32-bit Float (FP32) 	8	23
IEEE 754 Half Precision 16-bit Float (FP16) 	5	10
Google Brain Float (BF 16)  More range, less precision	8	7
Nvidia FP8 (E4M3) 	4	3

Linear Quantization

Original
32-bit float

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Quantized
2-bit signed int

1	-2	0	-1
-1	-1	-2	1
-2	1	-1	-2
1	-1	0	0

Zero point Scale

$(\text{Quantized} - (-1)) \times 1.07 = \text{Reconstructed}$

Reconstructed
32-bit float

2.14	-1.07	1.07	0
0	0	-1.07	2.14
-1.07	2.14	0	-1.07
2.14	0	1.07	1.07

How to find these numbers?

Linear Quantization

Original 32-bit float

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Quantized 2-bit signed int

1	-2	0	-1
-1	-1	-2	1
-2	1	-1	-2
1	-1	0	0

Zero point: -1 Scale: 1.07

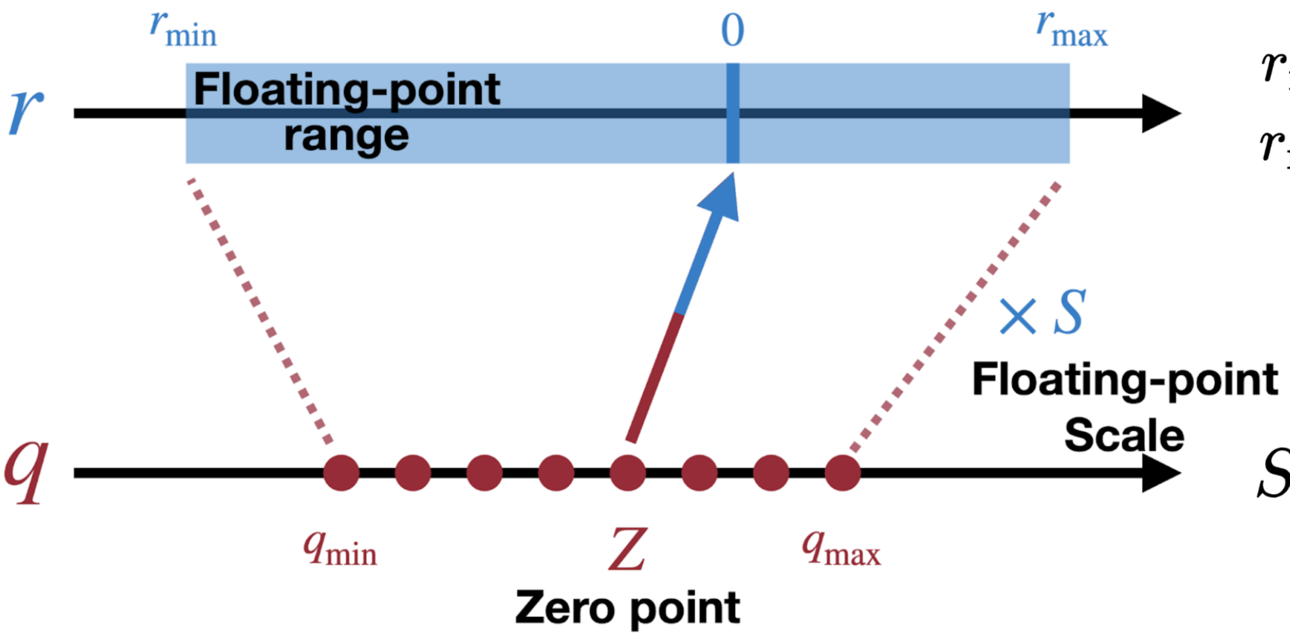
Reconstructed 32-bit float

2.14	-1.07	1.07	0
0	0	-1.07	2.14
-1.07	2.14	0	-1.07
2.14	0	1.07	1.07

$$r \approx (q - Z) \times S$$

r: floating-point; q: integer; Z: integer; S: floating-point

Linear Quantization: Scale

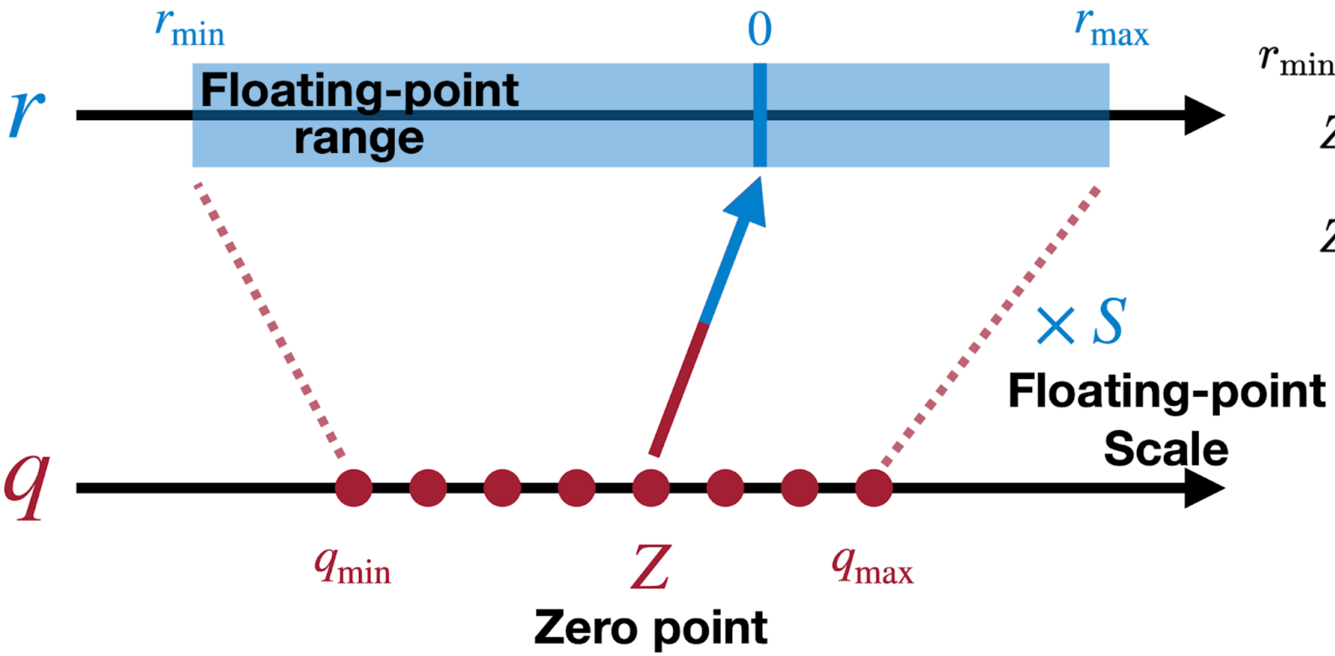


$$r_{\max} = S(q_{\max} - Z)$$
$$r_{\min} = S(q_{\min} - Z)$$

$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$

Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., CVPR 2018)

Linear Quantization: Zero Point



$$r_{\min} = S(q_{\min} - Z)$$

$$Z = q_{\min} - \frac{r_{\min}}{S}$$

$$Z = \text{round}\left(q_{\min} - \frac{r_{\min}}{S}\right)$$

[Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference](#) (Jacob et al., CVPR 2018)

Linear Quantization: Zero Point

"Absmax" Implementation

In practice, the weights are usually centered around zero (Z = 0):

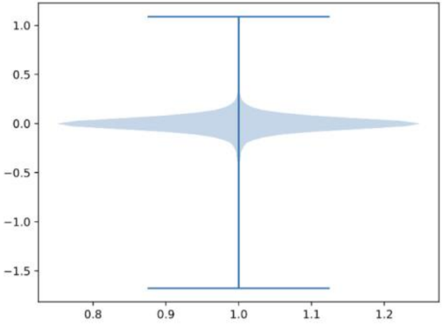
Therefore, we can find scale by using only the max.

$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$

$$\downarrow$$

$$S = \frac{r_{\min}}{q_{\min} - Z} = \frac{-|r|_{\max}}{q_{\min}}$$

Used in Pytorch, ONNX

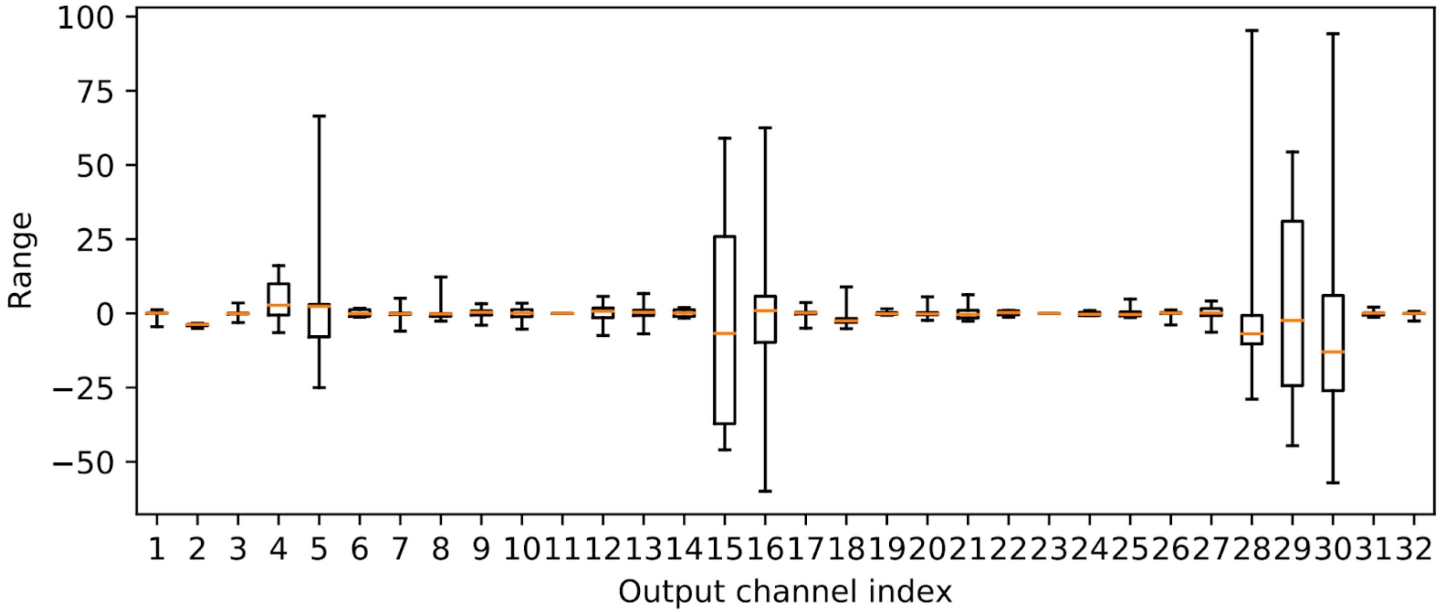


Weight distribution of first conv layer of ResNet-50.

[Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference]() (Jacob et al., CVPR 2018)

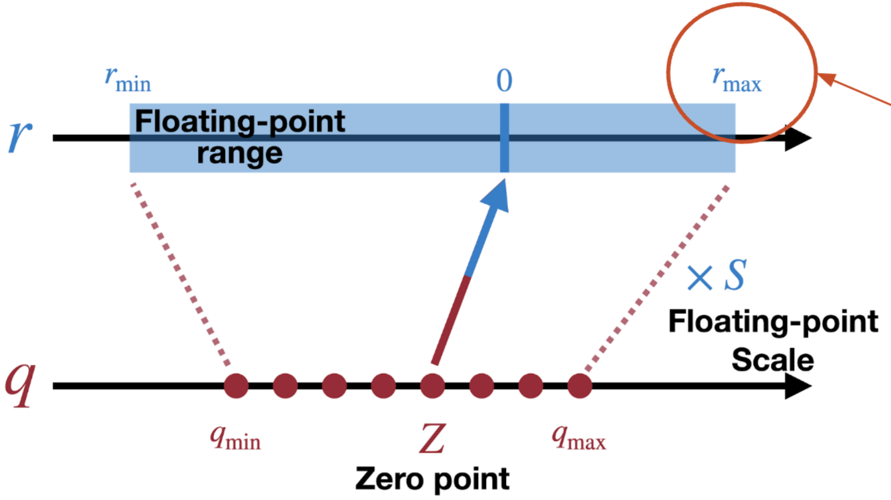
Quantization of Language Models

There exists many outliers in activations (activations of the first layer MobileNetV2):



[Data-Free Quantization Through Weight Equalization and Bias Correction](#) (Kagel et al., ICCV 2019)

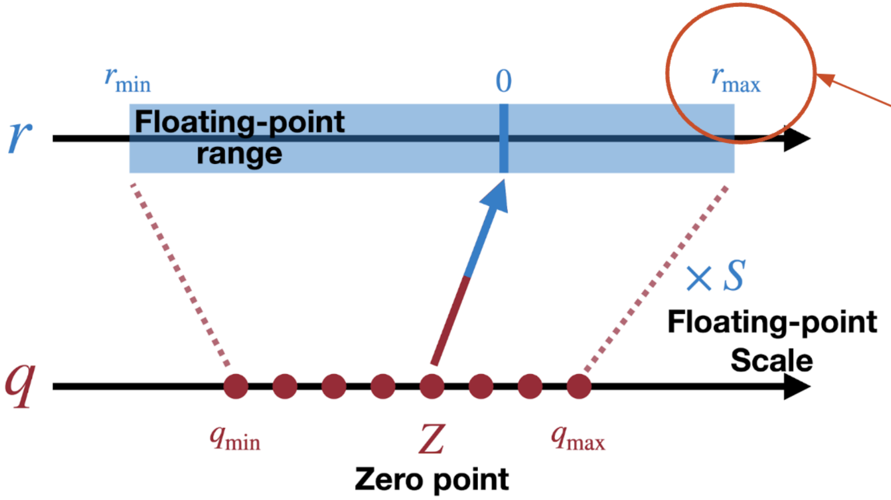
Quantization of Language Models



Outliers would make this extremely large!

Example: 15, 0.1, 0.02, 1.0, 0.01 -> 127, 1, 0, 8, 0
(Everything under 0.05 gets mapped to 0)

Quantization of Language Models



Outliers would make this extremely large!

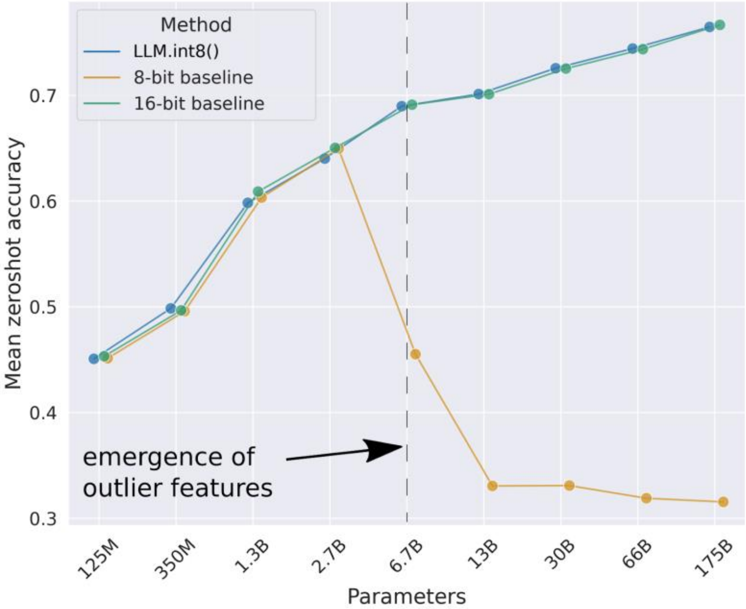
Example: 15, 0.1, 0.02, 1.0, 0.01 -> 127, 1, 0, 8, 0
(Everything under 0.05 gets mapped to 0)

Quantize each channel individually, each channel gets its own scale and Zero-point!

[Data-Free Quantization Through Weight Equalization and Bias Correction](#) (Kagel et al., ICCV 2019)

Quantization of Language Models

Outlier features significantly harms performance after quantization in LMs.

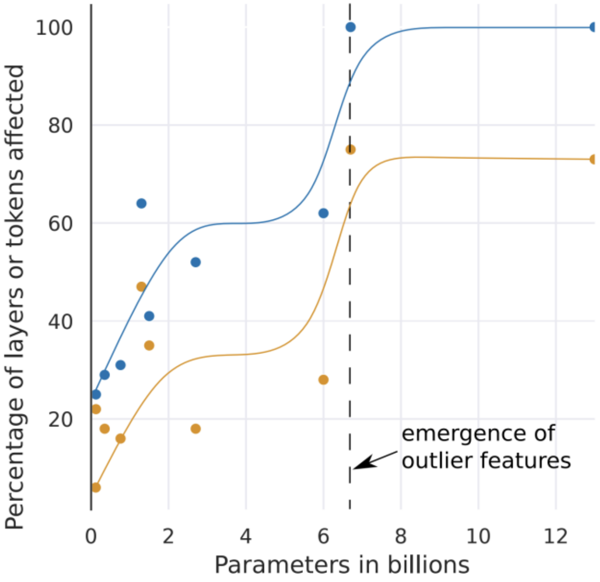


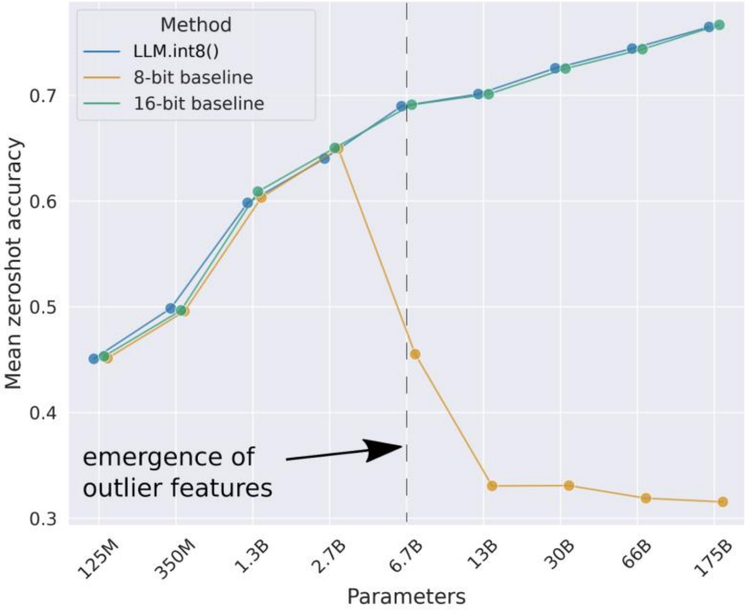
JOHNS HOPKINS WHITING SCHOOL of ENGINEERING

[LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale]() (Dettmers et al., NeurIPS 2022)

Quantization of Language Models

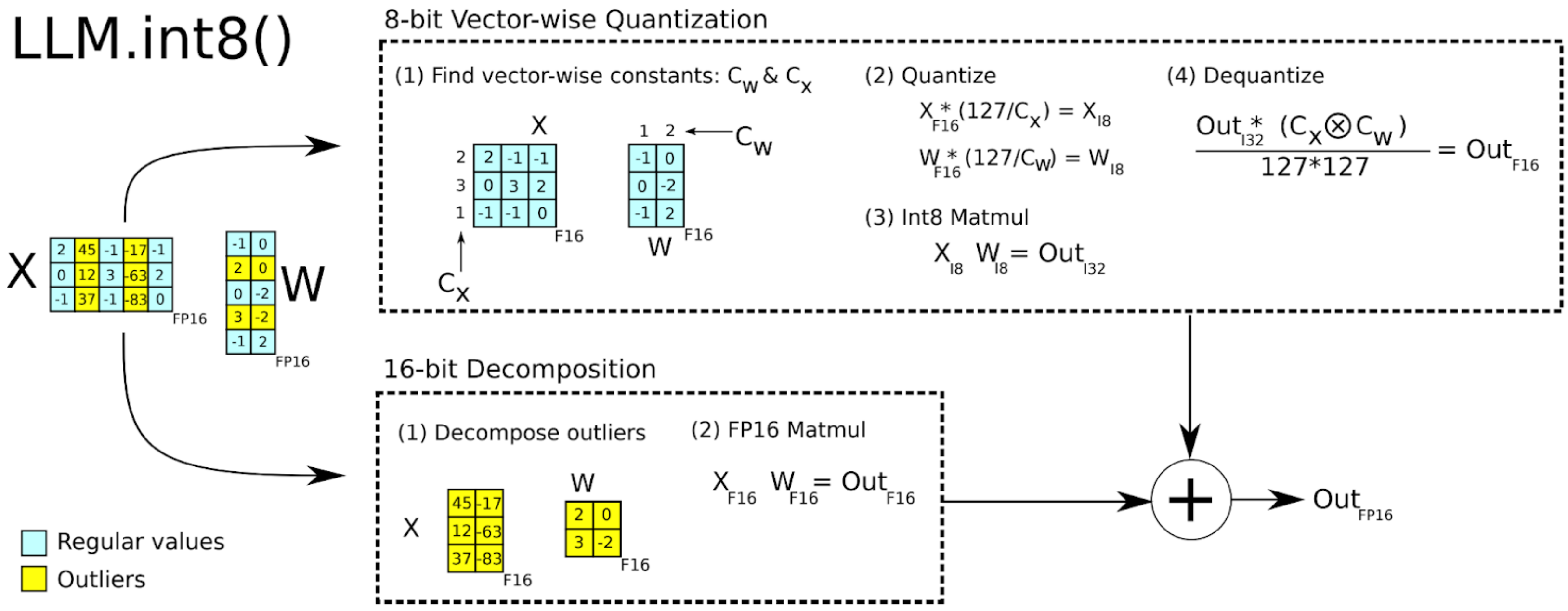
Outlier features significantly harms performance after quantization in LMs.





[LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale](#) (Dettmers et al., NeurIPS 2022)

Quantization of Language Models



Keep outlier channels / features in 16-bit, quantize the rest.

LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., NeurIPS 2022)

Quantization of Language Models

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

Zeropoint > absmax because outliers non-symmetric (either very large or very small, but not both)

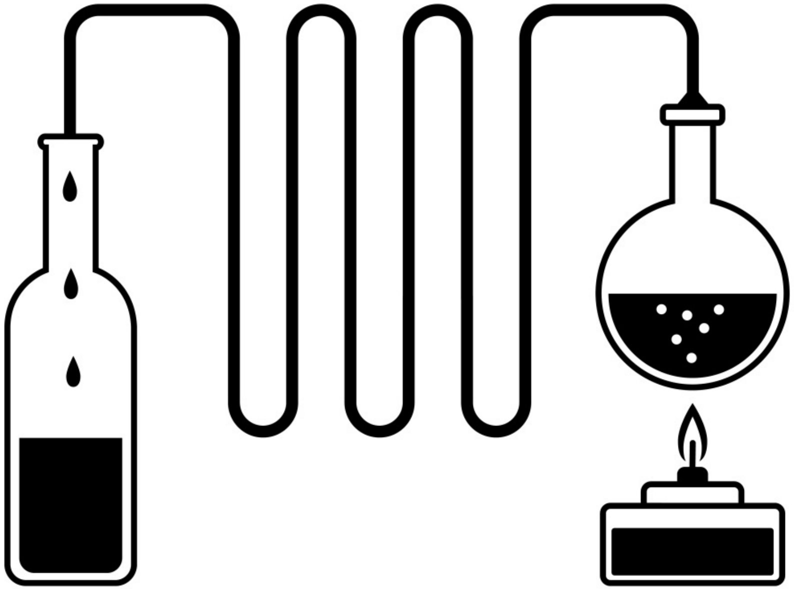
LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale (Dettmers et al., NeurIPS 2022)

Quantization of Language Models

- Maps floating point numbers (fp32, fp16, bf16) to low precision numbers (fp8, int8) to save memory.
- Is effective in reducing the memory required for both training / inference.
- 8-bit quantization loses minimal performance, while 4-bit quantization is hard, can be harmful to model performance.

Distilling the knowledge of larger models

Distillation



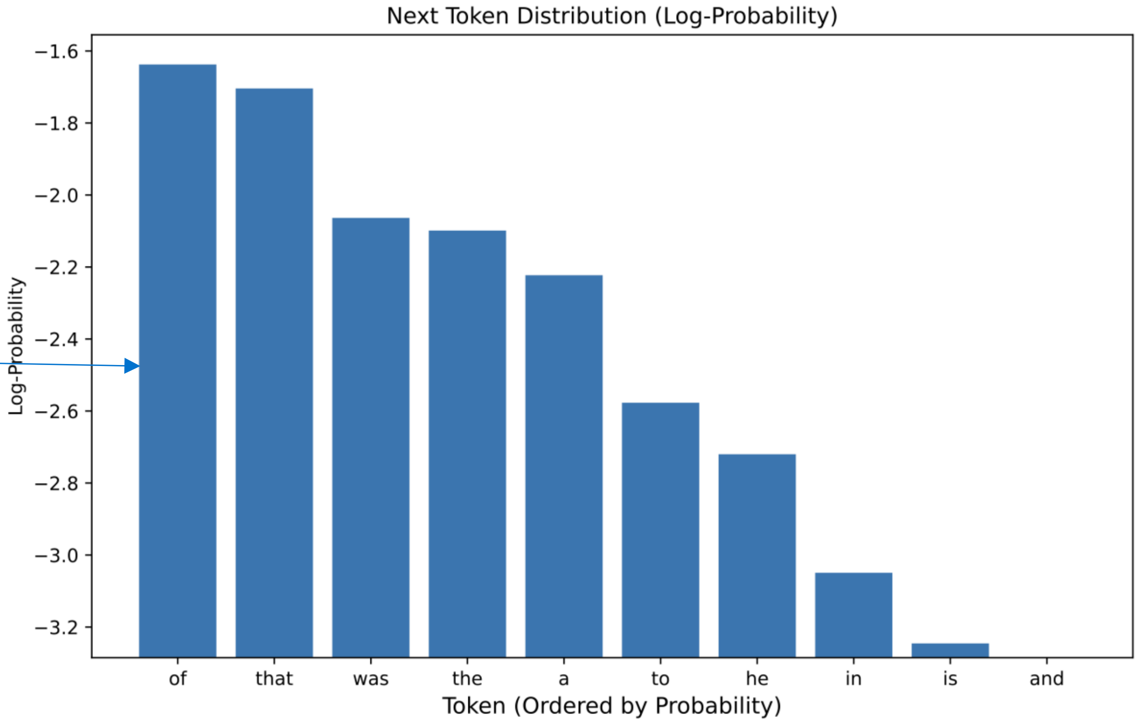
small model

Knowledge
and/or
capabilities of
a larger model

Revisit: Standard Training (NLLloss)

prefix: The strange case ___
groundtruth: of

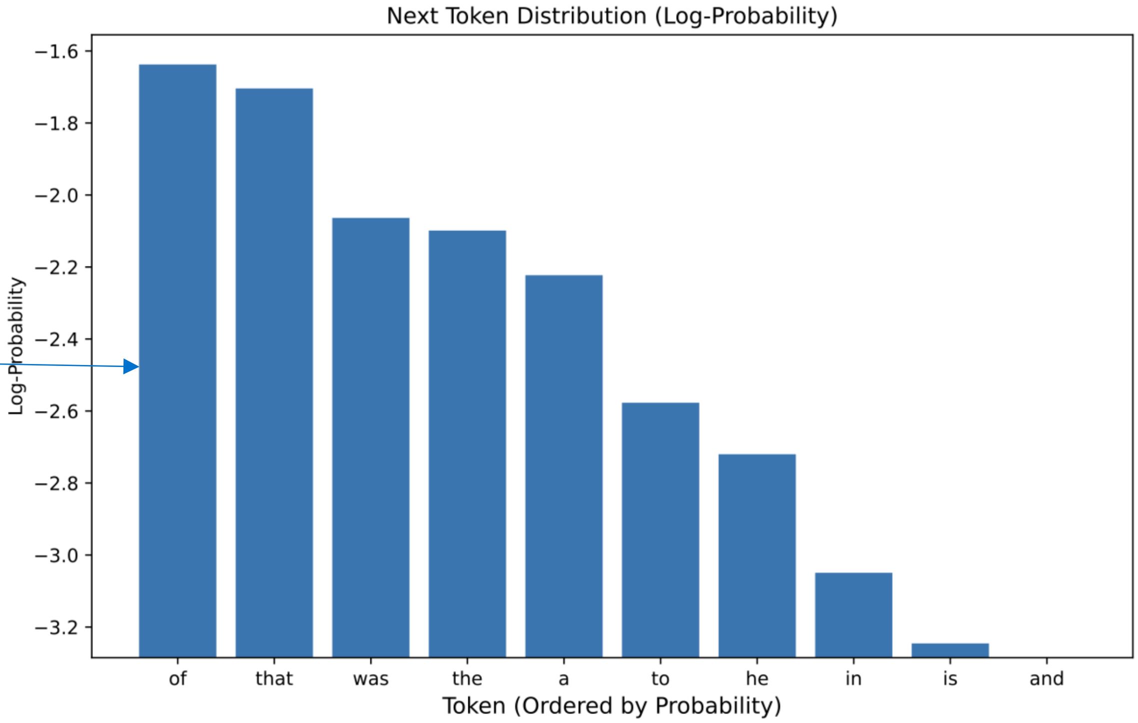
Loss = -log p(of)
= Cross Entropy(y_pred, groundtruth)



Revisit: Standard Training (NLLoss)

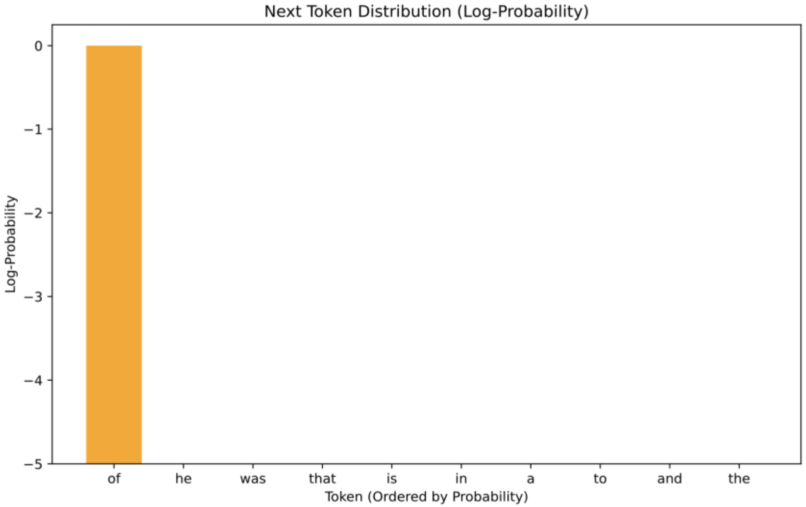
prefix: The strange case ___
groundtruth: of

loss = -log p(of)

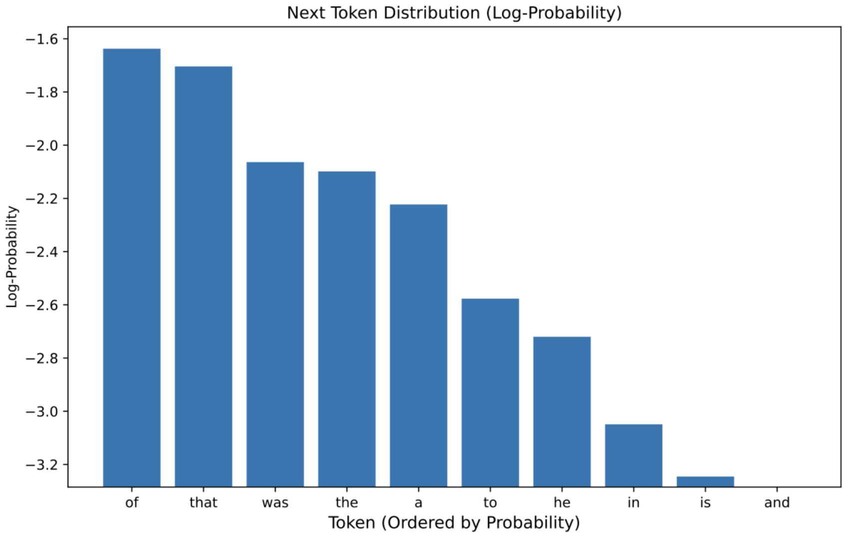


Revisit: Standard Training (NLLloss)

loss = -log p(of) = Cross Entropy(groundtruth, y_pred)



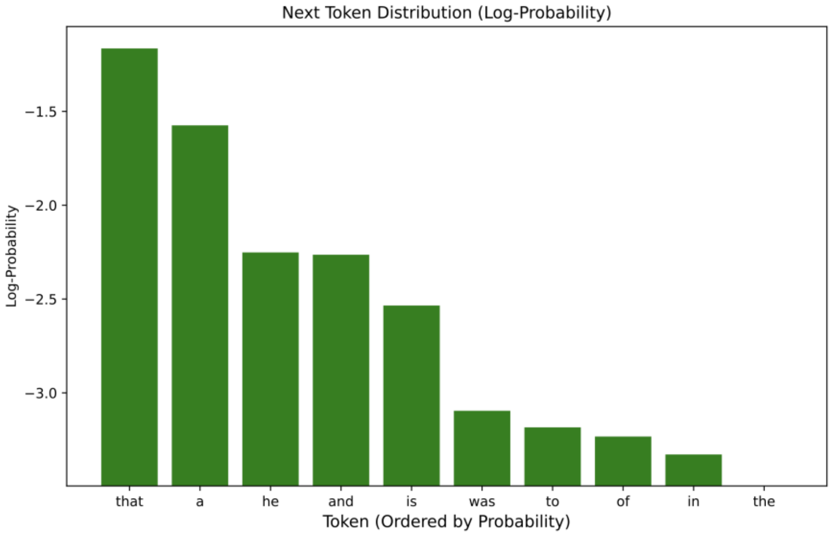
Groundtruth
(one-hot)

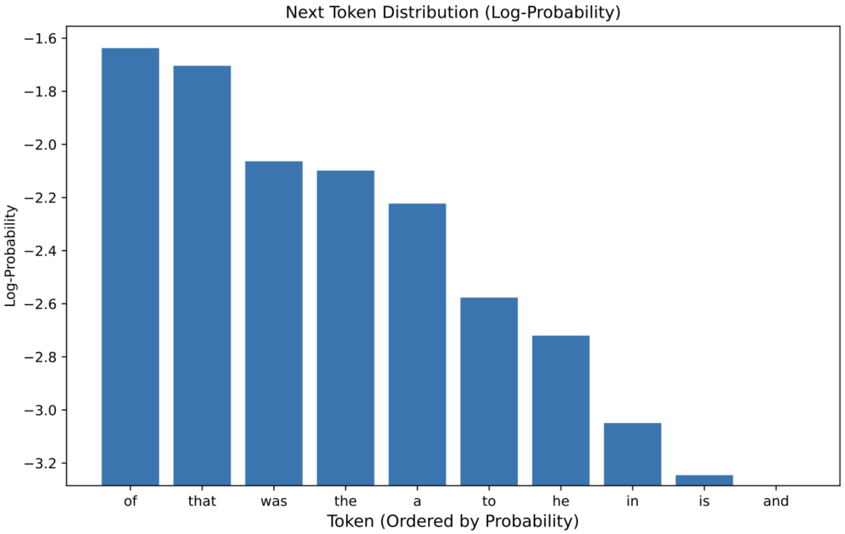


y_pred

Knowledge Distillation

KD loss = Cross Entropy(y_large, y_pred)





Large model next token probs (y_large)

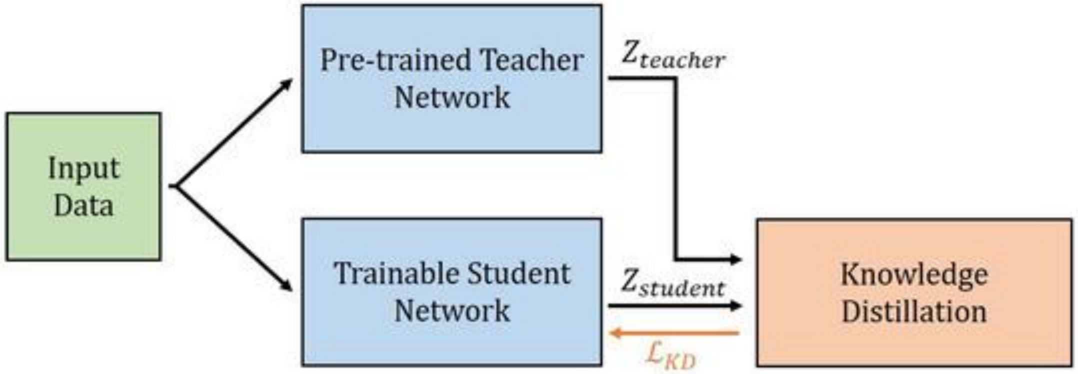
small model next token probs (y_pred)

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Knowledge Distillation

Step 1: Initialize teacher model with a large and capable model

Step 2: Feed input data to both student and teacher (freezed)

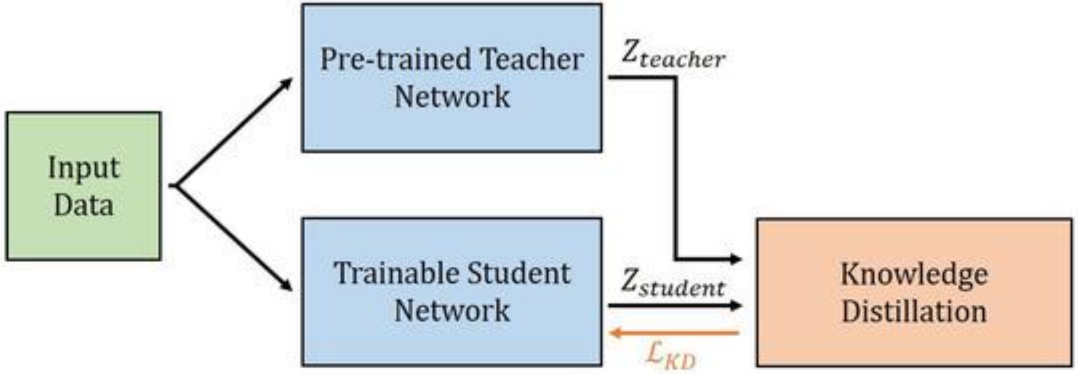


Step 3: Use teacher outputs to train student (Cross Entropy)

What if the teacher is Proprietary (GPT)?

Step 1: Initialize teacher model with a large and capable model

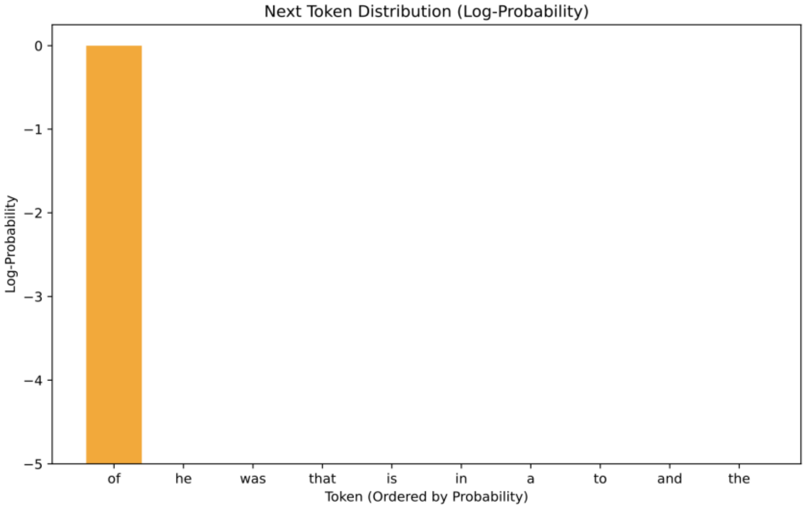
Step 2: Feed input data to both student and teacher (freezed)



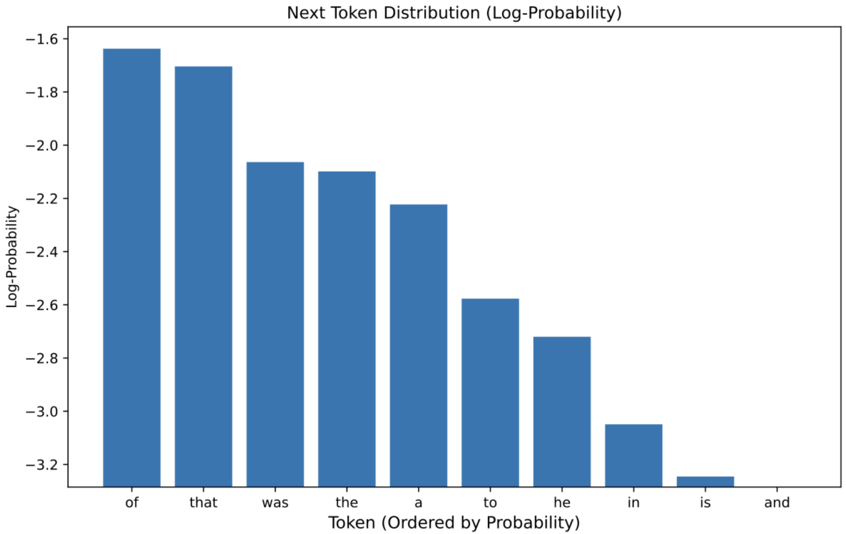
Step 3: Use teacher generations (instead of outputs) to train student!

Revisit: Standard Training (NLLloss)

loss = -log p(of) = Cross Entropy(sampled text, y_pred)



Sampled output (one-hot)



y_pred

What works better (a study in 2016)

Model	$\text{BLEU}_{K=1}$	$\Delta_{K=1}$	$\text{BLEU}_{K=5}$	$\Delta_{K=5}$
English → German WMT 2014				
Teacher Baseline 4 × 1000 (Params: 221m)	17.7	−	19.5	−
Baseline + Seq-Inter	19.6	+1.9	19.8	+0.3
Student Baseline 2 × 500 (Params: 84m)	14.7	−	17.6	−
Word-KD	15.4	+0.7	17.7	+0.1
Seq-KD	18.9	**+4.2**	19.0	+1.4

Use teacher log-probs

Use teacher generations

[Sequence-Level Knowledge Distillation](#) (Kim & Rush, EMNLP 2016)

Johns Hopkins Whiting School of Engineering

Knowledge Distillation

- Train student (usually smaller model) on the output of a teacher (usually a larger model)
- The output can be log-probabilities or sampled outputs
- Effective in "distilling" the knowledge of large models to smaller ones.