

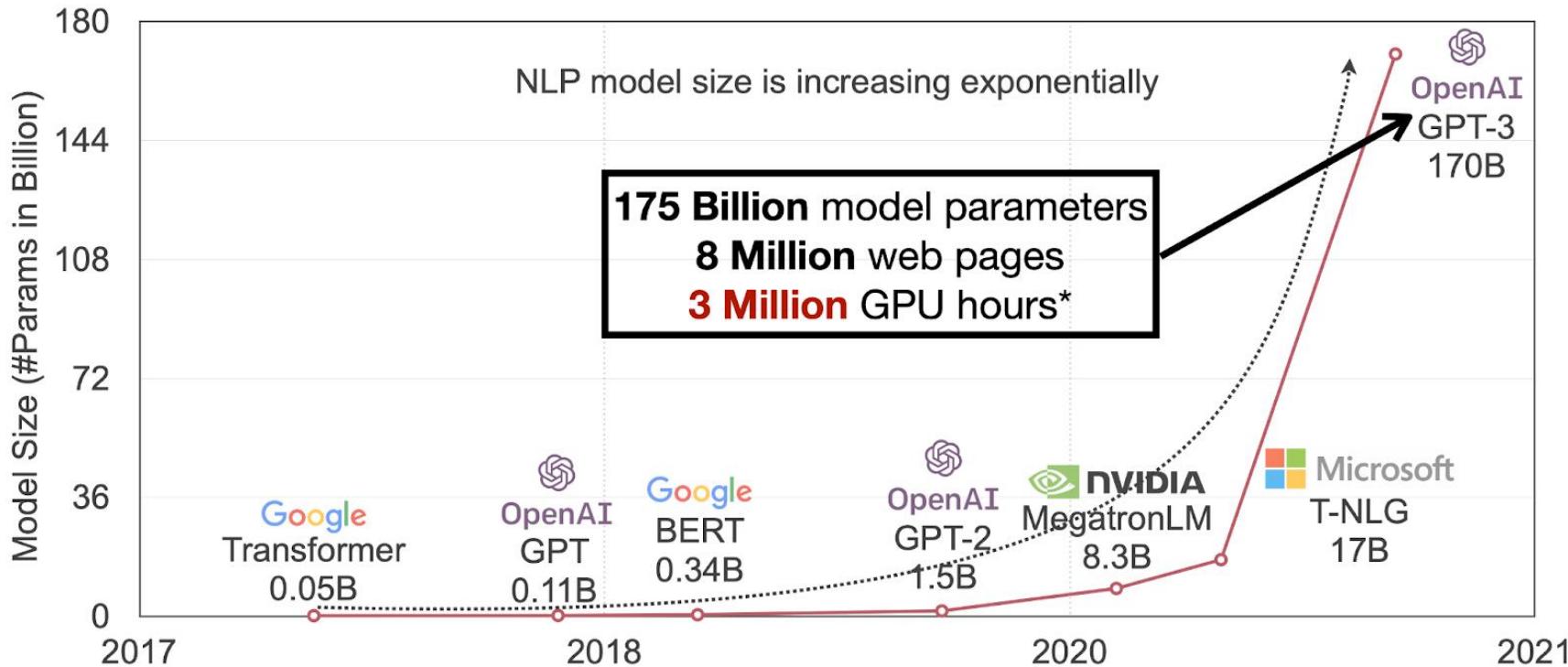


LM Efficiency

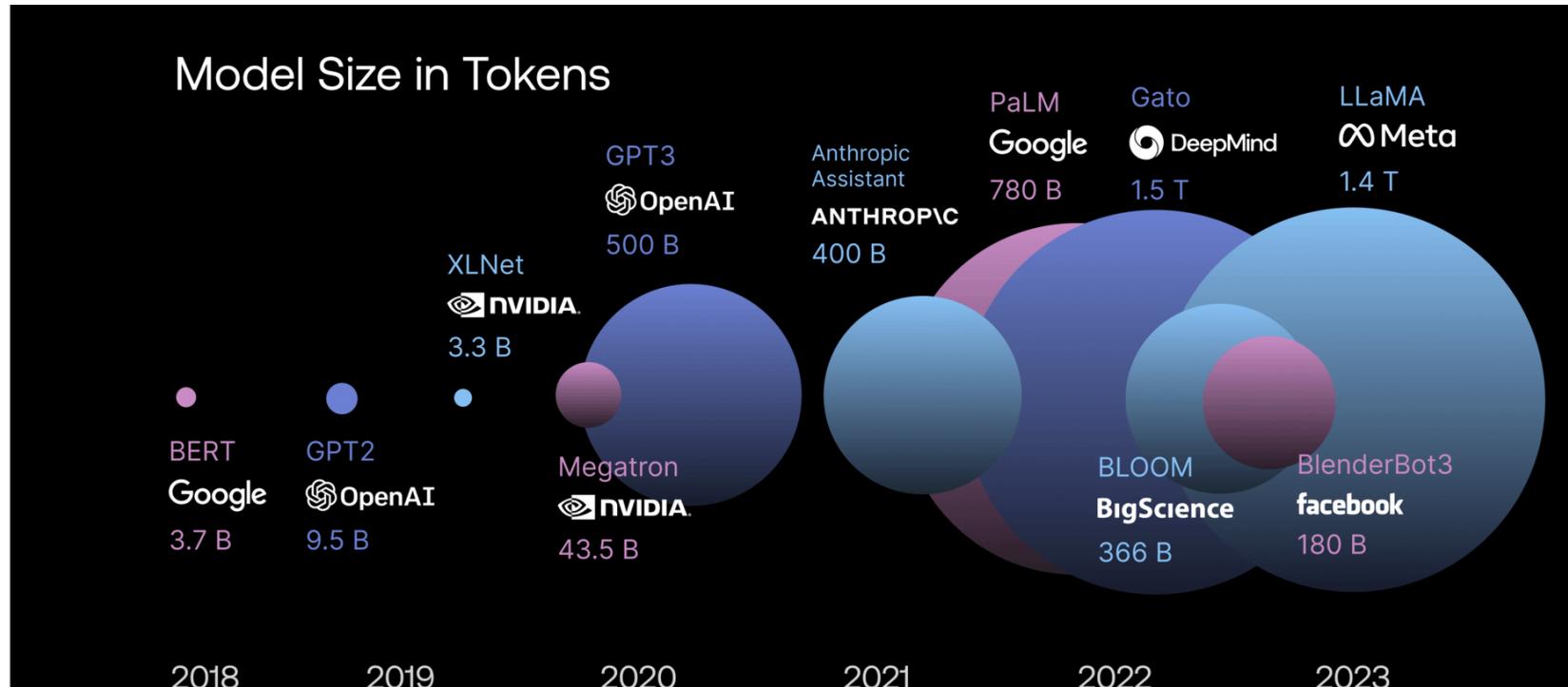
CSCI 601-771 (NLP: Self-Supervised Models)

<https://self-supervised.cs.jhu.edu/fa2025/>

Our models are getting larger!

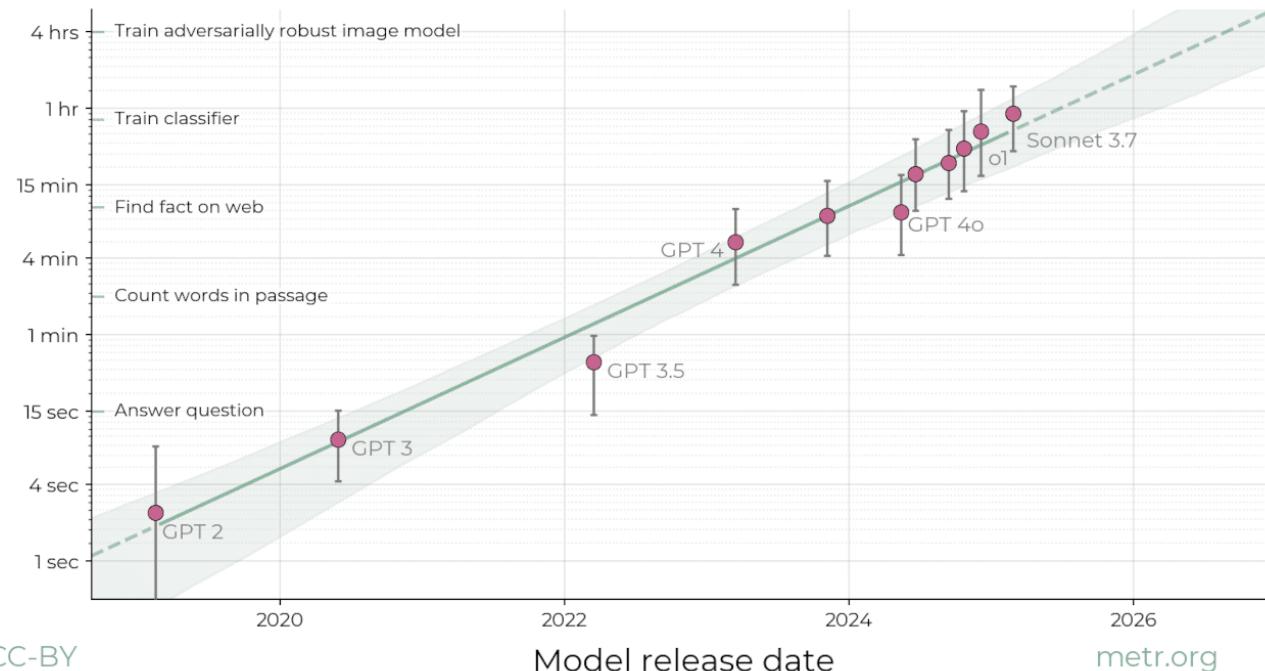


And consumes a lot of data!



Reasoning models solving long tasks

The length of tasks AI can do is doubling every 7 months
Task length (at 50% success rate)



Compute cost of Transformers

Diversion: Floating-point Ops: FLOPS

- Floating point operations per second (FLOPS, flops or flop/s)
- Each FLOP can represent an addition, subtraction, multiplication, or division of floating-point numbers,
- The total FLOP of a model (e.g., Transformer) provides a basic approximation of computational costs associated with that model.

FLOPS of Matrix Multiplication

- Matrix-vector multiplication are common in Self-Attention (e.g., QKV projection)
 - Requires $2mn$ ($2 \times$ matrix size) operations for multiplying $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^n$
 - (2 because 1 for multiplication, 1 for addition)

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n \\ A_{21}x_1 + A_{22}x_2 + \cdots + A_{2n}x_n \\ \vdots \\ A_{m1}x_1 + A_{m2}x_2 + \cdots + A_{mn}x_n \end{bmatrix}$$

Quiz: thinking about computations

- Consider the following matrix multiplication:

$$A B, \quad \text{where } A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times k}$$

- Question 1:** Computing AB involves how many arithmetic operations?

- (also referred to as floating-point operations or FLOPs)

- Answer:** It's $O(n \times m \times k)$.

- (to be a bit more precise, it's $\approx 2n \times m \times k$ since each element in AB requires almost equal num of multiplications and summations.)

- Question 2:** Computing AB involves how many memory/IO access?

- Answer:** It's $n \times m$ (reading A) + $m \times k$ (reading B) + $n \times k$ (writing AB).

Computations in Self-Attention Block

- We are going to count computations and IO access in Transformer computations.
- Note we assume that the full input sequence is given at once (e.g., training time).
- Here is the first step. Given: $\mathbf{x} \in \mathbb{R}^{b \times n \times d}$, $\mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$
- Let's think about the following computation: $\mathbf{x}\mathbf{W}_i^q$
- **Q1:** What is the number of arithmetic operations? $O(b \times n \times d \times \frac{d}{m})$ for each head
- **Q2:** What about the number of IO? $O(b \times n \times d + d \times \frac{d}{m} + b \times n \times \frac{d}{m})$ for each head
- **Q3:** What are these quantities for all heads?
 - Number of arithmetic ops: $O(bnd^2)$
 - Number of IO ops: $O(2bnd + d^2)$

Computations in Self-Att

b : batch size,
 n : sequence length,
 m : number of heads
 d : feature dimension in output of SA
 d/m : feature dimension inside each SA head
 $d_{\text{ff}} = 4d$: feature dimension inside FFN

Dimensions	Operation	Computations	IO
$\mathbf{x} \in \mathbb{R}^{b \times n \times d}, \mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$	$\mathbf{x}\mathbf{W}_i^q, \mathbf{x}\mathbf{W}_i^k, \mathbf{x}\mathbf{W}_i^v$ for m heads	$O(bnd^2)$	$O(d^2 + 2bnd)$
$\mathbf{Q}_i, \mathbf{K}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}$	$P_i \leftarrow \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d/m}}\right)$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\mathbf{V}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}, P_i \in \mathbb{R}^{b \times n \times n}$	$\text{head}_i \leftarrow P_i \mathbf{V}_i$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\mathbf{W}^o \in \mathbb{R}^{d \times d}, \text{head}_i \in \mathbb{R}^{b \times n \times \frac{d}{m}}$	$Y = \text{Concat}(\text{head}_1, \dots, \text{head}_m) \mathbf{W}^o$	$O(bnd^2)$	$O(2bnd + d^2)$
$Y \in \mathbb{R}^{b \times n \times d}, \mathbf{W}_1 \in \mathbb{R}^{d \times d_{\text{ff}}}, \mathbf{W}_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$	$Y = \text{ReLU}(Y \mathbf{W}_1) \mathbf{W}_2$	$O(16bnd^2)$	$O(2bnd + 8d^2)$
---	Total	$O(bnd^2 + bn^2d)$	$O(bnd + bmn^2 + d^2)$

The bottlenecks

b : batch size,
 n : sequence length,
 d : feature dimension in output of SA

- So, in total, we have →
- The quadratic terms are based on n and d
- d is fixed (part of architecture) but n changes with input.
- **Bottlenecks #1:** If n (sequence length) $\gg d$ (feature dimension), the time and space complexity would be dominated by $O(n^2)$.
- **However,** these despite this quadratic dependence these are parallelizable operations which can be computed efficiently in GPUs.
 - In comparison, RNNs perform less arithmetic ops but they're not all parallelizable.
- **Bottlenecks #2:** Another potential bottleneck is how fast we can run IO. (more on this later)

Computations	IO
$O(bnd^2 + bn^2d)$	$O(bnd + bmn^2 + d^2)$

Layer Type	Complexity per Layer	Sequential Operations
Self-Attention	$O(n^2 \cdot d)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$

[[Vaswani et al. 2017](#)]

Recap

- Transformers **computation of a full sequence** is bounded by $O(bnd^2 + bn^2d)$.
 - Generally, the quadratic term that depends on seq len n is more concerning.
- **IO may also impose** other limits on this (coming up).
- Also, the above calculations is for a given sentences (e.g. training time).
 - How bad is the computational complexity during the decoding time where we want to generate text one token at a time?

During decoding time, how slow
is attention computation?

Q: Pulse Check: What is a KV-Cache?

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

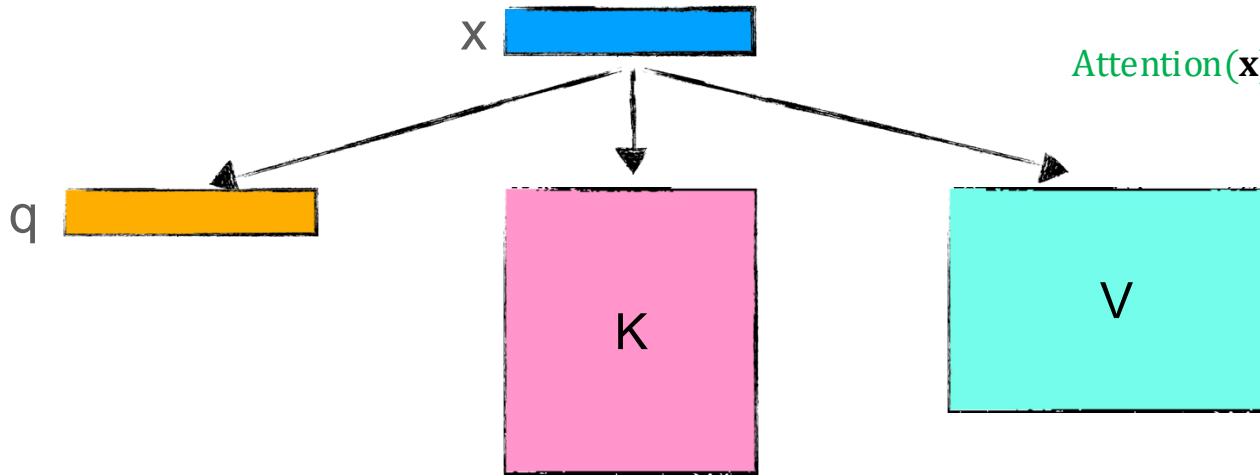
$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Self-Attention During Inference

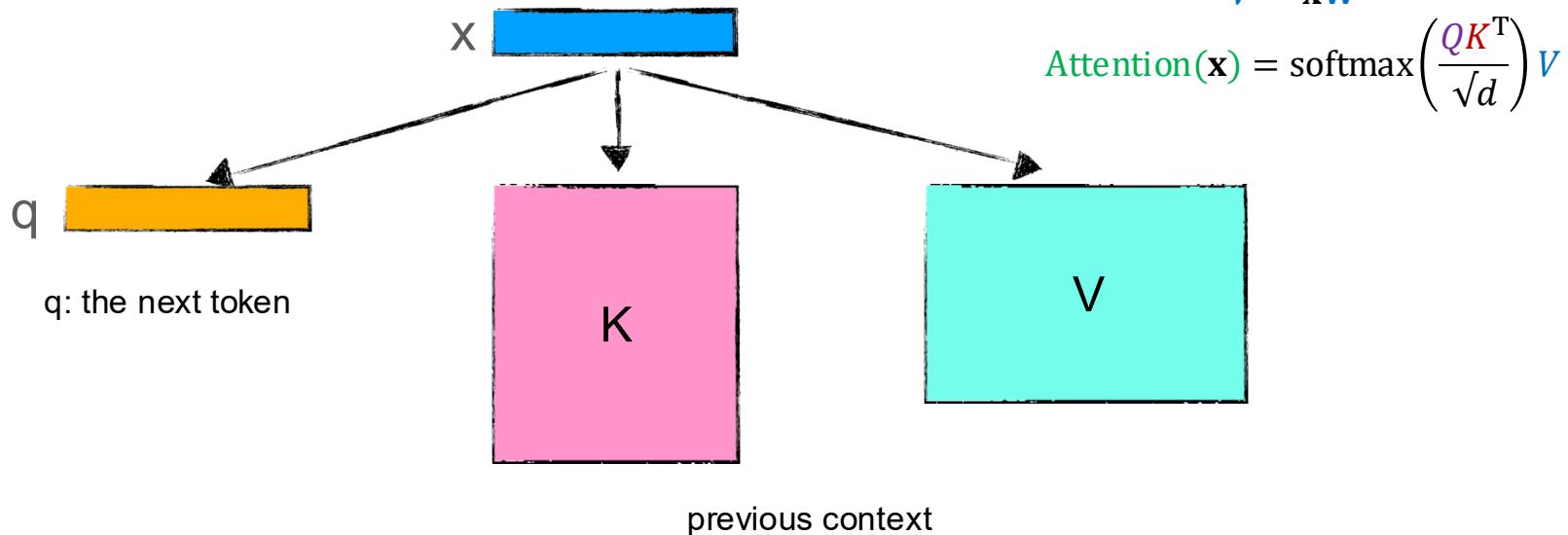
$$\begin{aligned} Q &= \mathbf{x}\mathbf{W}^q \\ K &= \mathbf{x}\mathbf{W}^k \\ V &= \mathbf{x}\mathbf{W}^v \end{aligned}$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}$$



[Slide credit: Arman Cohan]

Self-Attention During Inference



[Slide credit: Arman Cohan]

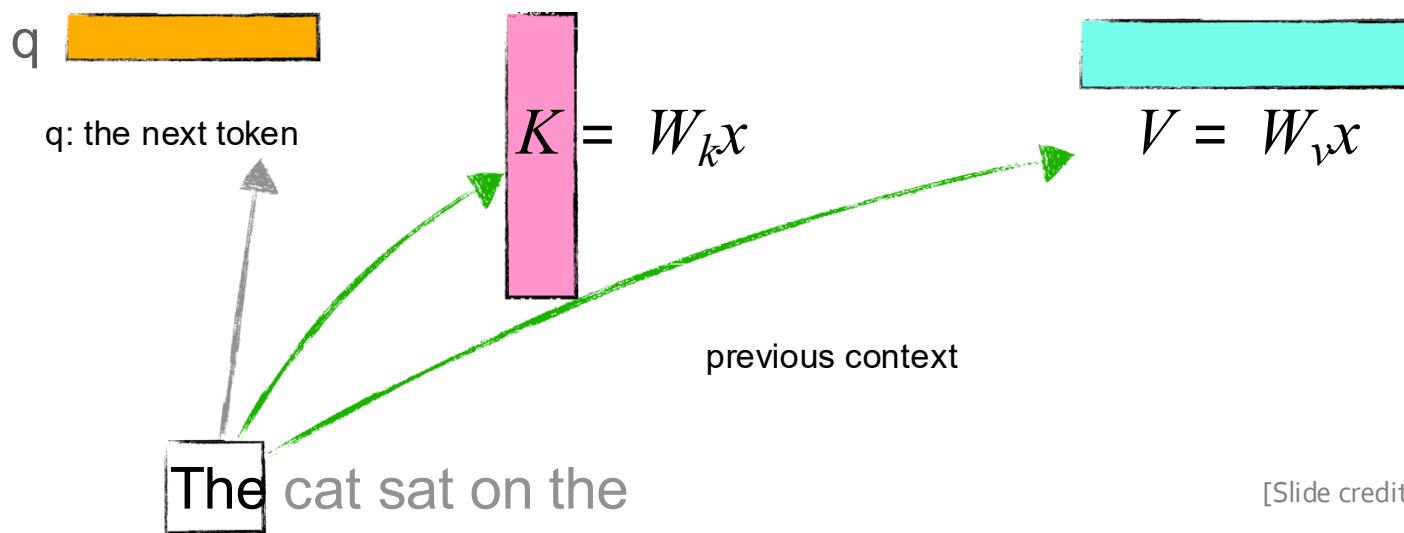
Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$

$$K = \mathbf{x}\mathbf{W}^k$$

$$V = \mathbf{x}\mathbf{W}^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

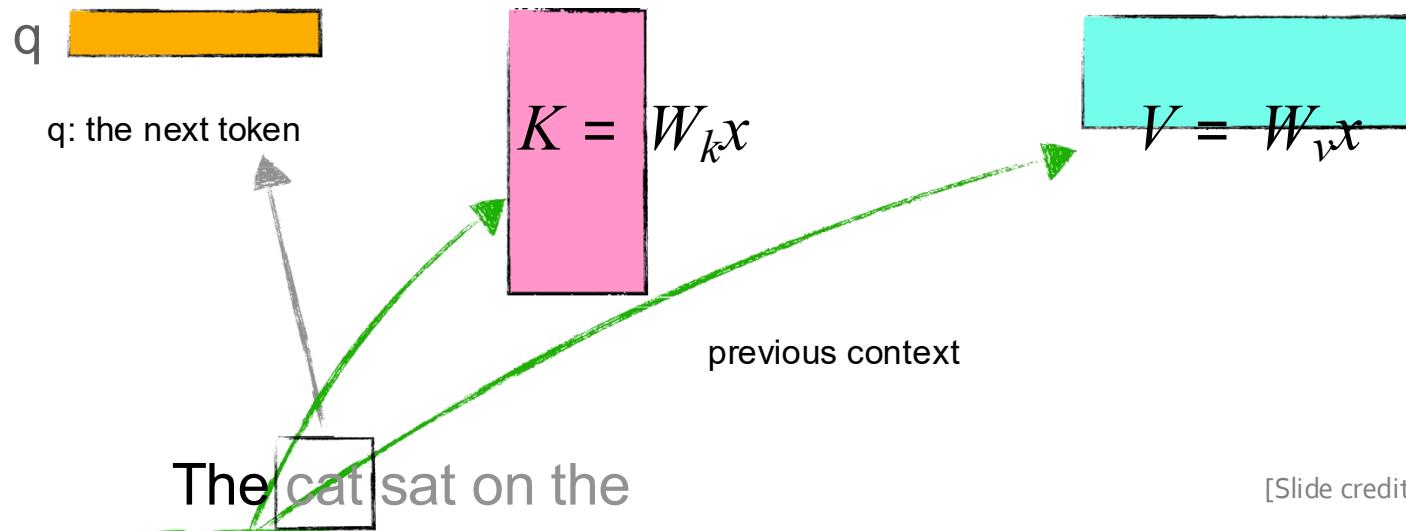
Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$

$$K = \mathbf{x}\mathbf{W}^k$$

$$V = \mathbf{x}\mathbf{W}^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

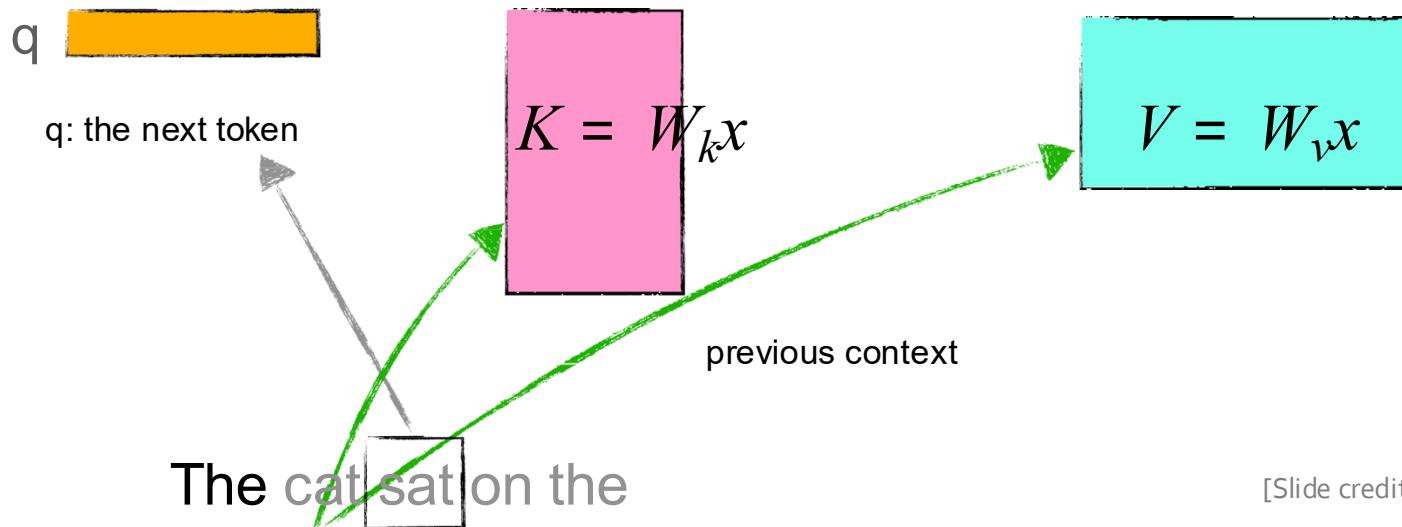
Self-Attention During Inference

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

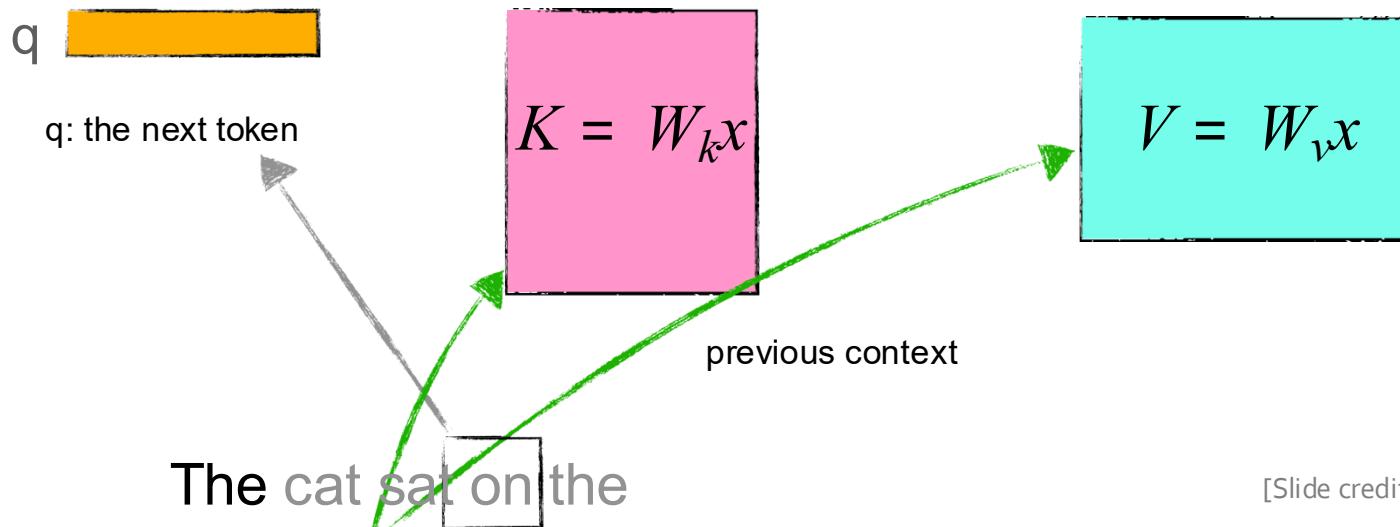
Self-Attention During Inference

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

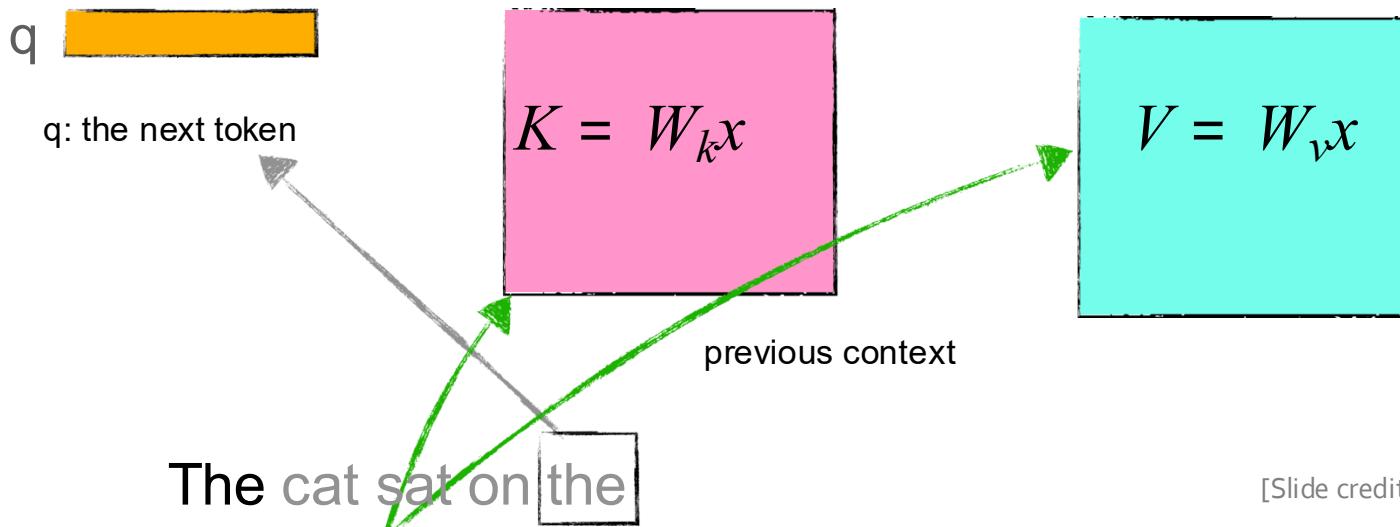
- We are computing the Keys and Values many times!
 - Let's reduce redundancy! 😊

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

KV-Cache for reducing inference redundancy

- We are computing the Keys and Values many times!
 - Let's reduce redundancy! 😊

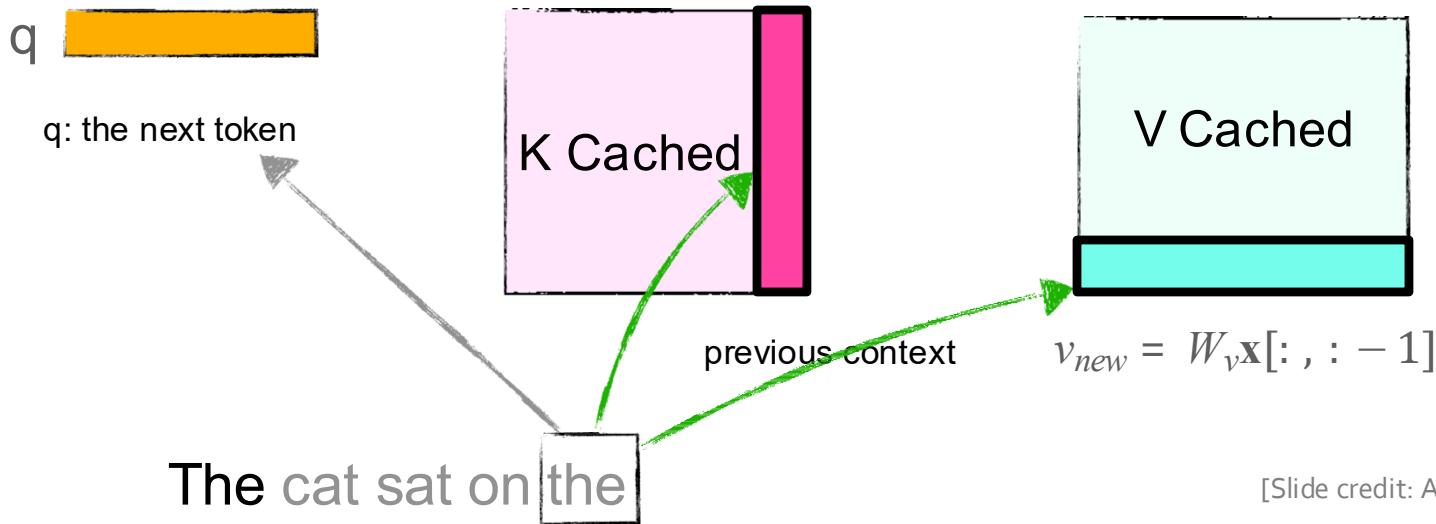
$$k_{new} = W_k \mathbf{x}[:, :, -1]$$

$$Q = \mathbf{x}W^q$$

$$K = \mathbf{x}W^k$$

$$V = \mathbf{x}W^v$$

$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

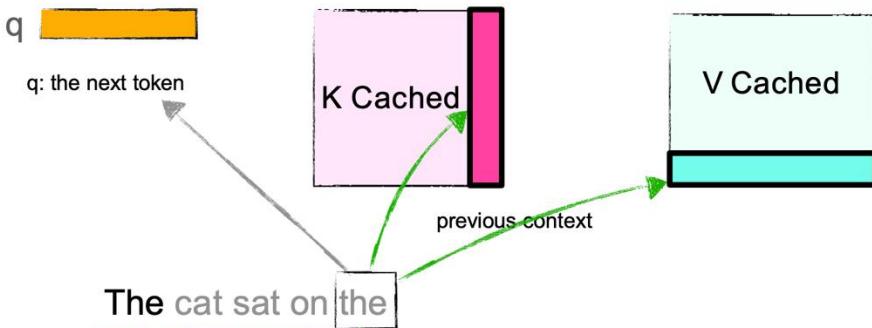


[Slide credit: Arman Cohan]

Quiz: KV-Cache

- How much **memory** does this KV-cache require? Let's assume,
 - batch size b ,
 - embedding dimension is d ,
 - the length of the sequence seen so far is n ,
 - your model has L layers,
 - Each param is stored k bytes (e.g., FP16 takes 2 bytes)

1. $2bnLk$
2. $2bdLk$
3. $bndLk$
4. $2bndLk$



Batch Size:

1

Sequence Length:

1024

KV-Cache size

- For GPT2, this comes out to a modest size of ~36 MB assuming we use the max sequence length of 1024 tokens and a batch size of 1.
- For larger models, however, the KV Cache can take up GBs of memory.
 - [Try this calculator](#)

Model	Parameter Count	KV Cache Size
GPT-3 Small	125M	36.000 MB
GPT-3 Medium	350M	96.000 MB
GPT-3 Large	760M	144.000 MB
GPT-3 XL	1.3B	288.000 MB
GPT-3 2.7B	2.7B	320.000 MB
GPT-3 6.7B	6.7B	512.000 MB
GPT-3 13B	13B	800.000 MB
GPT-3	175B	4.500 GB

Q: Where do we store the KV Cache?

- Depends.
- If you're doing single-GPU inference, it sits on the GPU that computes attention.
- KV cache can be offloaded to CPU (RAM) (if GPU is running out of space) but adds latency.
- Later we will revisit this discussion after seeing distributed training/inference.

Who has played w/ KV-Cache in practice?

KV-Cache in practice

```
model_name = "gpt2" # swap for your decoder-only model (Llama, Mistral, etc.)
tok = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device).eval()

# ----- 1) Shared prefix (system prompt or template) -----
prefix = "System: You are a concise assistant.\n"
prefix_ids = tok(prefix, return_tensors="pt").to(device)

with torch.no_grad():
    # Build the KV cache for the prefix ONCE
    pref_out = model(**prefix_ids, use_cache=True)
    prefix_kv = pref_out.past_key_values # this is the cached prefix KV
    # (No generation here; we just cached the prefix.)
```

KV-Cache in practice

```
# ---- 2) Reuse the cached prefix for multiple queries ----
queries = [
    "User: Define gradient clipping.\nAssistant:",
    "User: Difference between Adam and SGD?\nAssistant:",
]

for q in queries:
    q_ids = tok(q, return_tensors="pt").to(device)
    with torch.no_grad():
        # Reuse the prefix
        out = model(**q_ids, past_key_values=prefix_kv, use_cache=True)
        # Demo: take one token step (for clarity). In practice, use model.generate(...)
        next_token = out.logits[:, -1, :].argmax(dim=-1, keepdim=True)
        print(tok.decode(next_token[0]))
```

Note, here we don't need to re-feed the prefix tokens (prefix_ids) to the model. Why? (q_ids)

KV-Cache in practice

```
# Assume you already built prefix_kv on GPU as above

# ----- Offload to CPU to save VRAM when not in use -----
prefix_kv_cpu = tuple(
    (k.cpu(), v.cpu()) for (k, v) in prefix_kv
)

# Later, when you need to reuse it again:
prefix_kv_gpu = tuple(
    (k.to(device, non_blocking=True), v.to(device, non_blocking=True))
    for (k, v) in prefix_kv_cpu
)
```

Recap

- To avoid **redundant computations** during decoding time, KV-cache is used to keep track of previous calculations of keys and values.
- But how exactly how costly are these computations?

Decoding Computations

Notice we're doing this computations for one token

Compared to the previous table (SA for a seq of length n), all the cells have one less dependence on n (e.g., $n^2 \rightarrow n$ or $n \rightarrow 1$).

Dimensions	Operation	Computations	IO
$\mathbf{x} \in \mathbb{R}^{b \times 1 \times d}, \mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$	Query/key computations get combined with KV-cache	$O(bd^2)$	$O(d^2 + 2bd)$
$\mathbf{Q}_i, \mathbf{K}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}} + \text{KV-cache}$	$P_i \leftarrow \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d/m}}\right)$ for m heads	$O(bnd)$	$O(bnm + bnd + bd)$
$\mathbf{V}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}}, P_i \in \mathbb{R}^{b \times n \times 1}$	$\text{head}_i \leftarrow P_i \mathbf{V}_i$ for m heads	$O(bnd)$	$O(bnm + bnd + bd)$
$\mathbf{W}^o \in \mathbb{R}^{d \times d}, \text{head}_i \in \mathbb{R}^{b \times 1 \times \frac{d}{m}}$	$Y = \text{Concat}(\text{head}_1, \dots, \text{head}_m) \mathbf{W}^o$	$O(bd^2)$	$O(2bd + d^2)$
$Y \in \mathbb{R}^{b \times 1 \times d}, \mathbf{W}_1 \in \mathbb{R}^{d \times d_{ff}}, \mathbf{W}_2 \in \mathbb{R}^{d_{ff} \times d}$	$Y = \text{ReLU}(Y \mathbf{W}_1) \mathbf{W}_2$	$O(16bd^2)$	$O(2bd + 8d^2)$
---	Total	$O(bd^2 + bnd)$	$O(bmn + bnd + d^2)$

Now the computations (of next token) has linear dependence on seq length.

b : batch size,

n : sequence length **thus far**,

m : number of heads

d : feature dimension in output of SA

$d_{ff} = 4d$: feature dimension inside FFN

Summary: Computational Complexity of Transformers

- **Process a sequence at once:** Computation is bounded by $O(n^2)$.
- **Processing one token at a time during inference:**
 - **KV-Cache:** To avoid redundant computations during decoding time, KV-cache is used to keep track of previous calculations of keys and values.
 - The computation is bounded by $O(n)$.
- Though in all cases, the computations are parallelizable (modulo Transformer layers).



Thinking about compute vs IO tension



Diversion: Arithmetic Intensity

- Arithmetic Intensity of a program execution:
 $(\# \text{ of floating-point operations}) / (\# \text{ of data bytes transferred to memory})$
- It helps determine whether a program is *compute-bound* or *memory-bound*:
 - If AI is **high**, performance is limited by how fast the GPU can compute.
 - If AI is **low**, performance is constrained by how fast data can be transferred between global memory and GPU cores.
- A good rule of thumb:
 - Memory-bound: $\text{AI} < 10 \text{ FLOPs/byte}$
 - Balanced: $10 \leq \text{AI} \leq 100 \text{ FLOPs/byte}$
 - Compute-bound: $\text{AI} > 100 \text{ FLOPs/byte}$

Quiz

- If a model has **high** arithmetic intensity, which of the following is true?
 - A) Performance is mostly limited by memory bandwidth
 - B) Performance is mostly limited by compute throughput
 - C) Memory accesses dominate execution time
 - D) The workload is not well-suited for GPUs
- **Answer:** High AI means the GPU spends more time computing per byte of memory fetched, making it **compute-bound** rather than **memory-bound**. Hence, B.

Arithmetic Intensity: An example

- We are going to compute AI for the first operation in Self-Attention.
- Note we assume that the full input sequence is given at once (e.g., training time).
- Given: $\mathbf{x} \in \mathbb{R}^{b \times n \times d}$, $\mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$ we want to compute: $\mathbf{x}\mathbf{W}_i^q$. From last week:

Dimensions	Operation	Computations	IO
$\mathbf{x} \in \mathbb{R}^{b \times n \times d}$, $\mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$	$\mathbf{x}\mathbf{W}_i^q$, $\mathbf{x}\mathbf{W}_i^k$, $\mathbf{x}\mathbf{W}_i^v$ for m heads	$O(bnd^2)$	$O(d^2 + 2bnd)$

$$\text{AI} = O\left(\frac{bnd^2}{d^2 + 2bnd}\right) = O\left(\left(\frac{d^2 + 2bnd}{bnd^2}\right)^{-1}\right) = O\left(\left(\frac{1}{bn} + \frac{2}{d}\right)^{-1}\right)$$

Quiz

- Given: $\mathbf{x} \in \mathbb{R}^{b \times n \times d}$, $\mathbf{W}_i^q \in \mathbb{R}^{d \times \frac{d}{m}}$ we know that the AI for computing $\mathbf{x}\mathbf{W}_i^q$ is:

$$\text{AI} = O\left(\left(\frac{1}{bn} + \frac{1}{d}\right)^{-1}\right)$$

- This process is_____?
 - Memory-bound
 - Balanced
 - Compute-bound
- Answer:** Our AI is **large-ish**. Depending on hyperparams, this is either balanced or compute-bound.
 - If $n = 10$ (sent len), $b = 10$ (batch size), $d = 512$ (hidden dim). Then $\text{AI} = 71$.
 - If $n = 30$ (sent len), $b = 20$ (batch size), $d = 512$ (hidden dim). Then $\text{AI} = 179$.

Arithmetic Intensity of Training Self-Attention

Bonus

Operation	Computations	IO	Arithmetic Intensity
$\mathbf{xW}_i^q, \mathbf{xW}_i^k, \mathbf{xW}_i^v$ for m heads	$O(bnd^2)$	$O(d^2 + 2bnd)$	$O\left(\left(\frac{1}{d} + \frac{1}{bn}\right)^{-1}\right)$
$P_i \leftarrow \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^T}{\sqrt{d/m}}\right)$ for m heads	$O(bn^2 d)$	$O(2bnd + bmn^2)$	$O\left(\left(\frac{m}{d} + \frac{1}{n}\right)^{-1}\right)$
$\text{head}_i \leftarrow P_i \mathbf{V}_i$ for m heads	$O(bn^2 d)$	$O(2bnd + bmn^2)$	$O\left(\left(\frac{m}{d} + \frac{1}{n}\right)^{-1}\right)$
$Y = \text{Concat}(\text{head}_1, \dots, \text{head}_m) \mathbf{W}^o$	$O(bnd^2)$	$O(2bnd + d^2)$	$O\left(\left(\frac{1}{d} + \frac{1}{bn}\right)^{-1}\right)$

b : batch size,

n : sequence length,

m : number of heads

d : feature dimension in output of SA

d/m : feature dimension inside each SA head

$d_{ff} = 4d$: feature dimension inside FFN

16

en

All these AI values are large!
We can continue running our
GPUs during training! 

Self-Attention Cost of Computation During Incremental (Autoregressive) Generation

Bonus

- Note that these numbers involve KV-caching.

Operation	Computations	IO	Arithmetic Intensity
$\mathbf{xW}_i^q, \mathbf{xW}_i^k, \mathbf{xW}_i^v$ for m heads	$O(bd^2)$	$O(d^2 + 2bd)$	$O\left(\left(1/d + 1/b\right)^{-1}\right)$
These two rows have low AI. For example, if $n = 20$ (sent len), $h = 12$ (num heads), $d = 512$ (hidden dim), then AI = 0.93. Hence, our program is memory bound during inference! 🚨			$O\left(\left(1 + m/d + 1/n\right)^{-1}\right)$
Note this is partly due to the memory-bandwidth cost of repeatedly loading the large "keys" and "values" tensors.			$O\left(\left(1 + m/d + 1/n\right)^{-1}\right)$
$Y = \text{concat}(\mathbf{m}_1, \dots, \mathbf{m}_m)$	$O(bd^2)$	$O(2bd + 8d^2)$	$O\left(\left(1/d + 1/b\right)^{-1}\right)$

b : batch size,

n : sequence length **thus far**,

m : number of heads

d : feature dimension in output of SA

$d_{ff} = 4d$: feature dimension inside FFN

KV-Cache drag

- Slowdown of autoregressive decoding.
 - As the sequence length grows, KV cache size increases, making cache lookup slower.
 - As we generate more output tokens (i.e. chatbot responding to user), throughput will slow down.
- **Simple idea:** Retain only the last L tokens of the KV-cache and compute attention using these recent tokens:
 - Inference cost will be **constant** $O(L)$ per token.

Sparse / sliding window attention

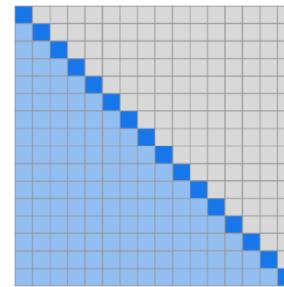
- Left: Just use the main part of the strided pattern – let depth extend effective context (Mistral)
- Right: Build sparse / structured attention that trades off expressiveness vs runtime.

The cat sat on the					
The	1	0	0	0	0
cat	1	1	0	0	0
sat	1	1	1	0	0
on	1	1	1	1	0
the	1	1	1	1	1

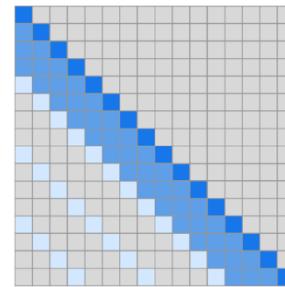
Vanilla Attention

The cat sat on the					
The	1	0	0	0	0
cat	1	1	0	0	0
sat	1	1	1	0	0
on	1	1	1	1	0
the	1	1	1	1	1

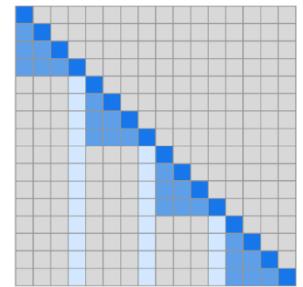
Sliding Window Attention



(a) Transformer



(b) Sparse Transformer (strided)



(c) Sparse Transformer (fixed)

Notable models:
GPT3 and Mistral

Quiz

- What are the drawbacks of sliding window?
 1. If the model was not trained for sliding window, generation will be out-of-distribution and unstable.
 2. If uses few layers, it'll retains local/recent information and cannot see global context.
 3. After a while, it will forget the input text (e.g. the original instruction provided by the user).
 4. All of the above.

Terminology: “Eviction”

- **Removing old K/V pairs** from the cache.
- Here is a minimal example:

```
# ---- Step 1: process a short 5-token sentence ----
text = "The dog runs fast today"
inp = tok(text, return_tensors="pt").to(device)

with torch.no_grad():
    out = model(**inp, use_cache=True)
    kv = out.past_key_values

print("Original KV length (layer 0):", kv[0][0].shape[2])

# ---- Step 2: evict one token inline (drop first token) ----
evicted = tuple(
    (k[:, :, 1:, :].contiguous(), v[:, :, 1:, :].contiguous())
    for (k, v) in kv
)
```

Terminology: “Eviction”

```
# ----- Step 3: continue generation using the evicted cache -----
next_input = tok(" happily", return_tensors="pt").to(device)

with torch.no_grad():
    out2 = model(**next_input, use_cache=True, past_key_values=evicted)

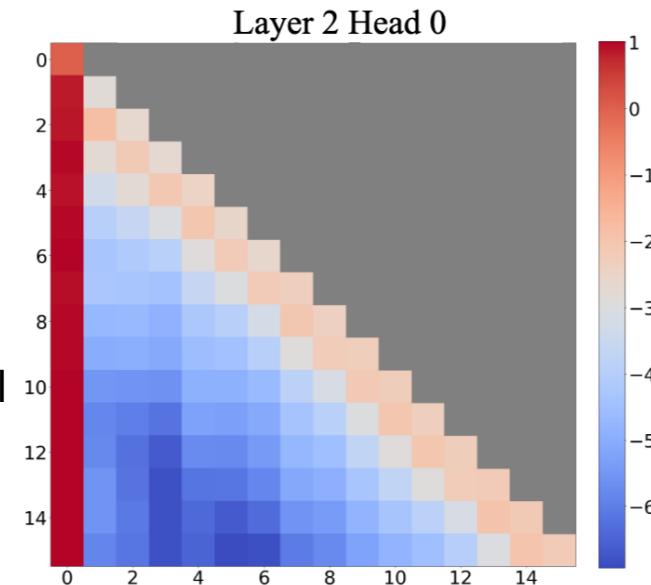
print("KV length after reuse:", out2.past_key_values[0][0].shape[2])

# optional: decode a quick sample
gen = model.generate(**next_input, past_key_values=evicted, max_new_tokens=5)
print(tok.decode(gen[0], skip_special_tokens=True))
```

Examples of Last-only window

Sliding Window Attention with “Sinks”

- Observations: There are few tokens (“sinks”) that the model heavily relies on.
- These sink tokens (e.g., BOS) consistently receive high attention.
- Removing them leads to unstable generation.
- Figure: In most layers, SA heavily attends to the initial token across all heads.
(the bottom two layers don’t always show this).
- Why? Complicated. One explanation is, when no meaningful tokens attract attention, the model must still distribute probability mass somewhere (e.g., uninformative tokens).
(note SA has to be a proper probability)

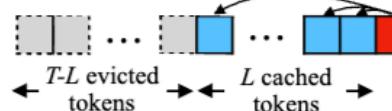
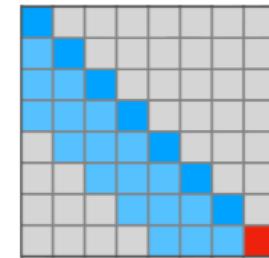


Sliding Window Attention with “Sinks”

- If you use sliding window, without retraining, your attention values will be distorted.

- StreamingLLM** always maintains few initial positions (sinks).

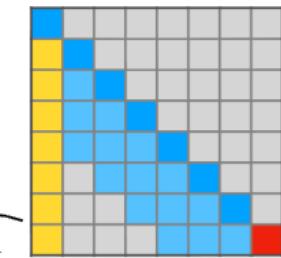
(b) Window Attention



$O(TL)$ ✓ PPL: 5158

Breaks when initial tokens are evicted.

(d) StreamingLLM (ours)



Attention Sink



$O(TL)$ ✓ PPL: 5.40 ✓

Can perform efficient and stable language modeling on long texts.

Examples of Caching the "sinks" (last+first)

- **Prompt:** The kid is wearing a red shirt,
- **Input length:** 24, **Output length:** 49
- KV Cache: unlimited -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is red, and the kid's hat's color is blue. The kid's pants' color is green. The kid is wearing
- KV Cache: first3+last_20 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is red. The kid's color is red. The kid's color. The kid's red is. is
- KV Cache: first3+last_10 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is a white shirt with a black and red and white and orange collar and blue and green with a black on each side of the
- KV Cache: first3+last_5 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is the most important part of the company is the home, so as to be sure that you are on the home and you are



Prefix Caching (or, prompt caching)



Prefix Caching

- Many prompts (e.g., from different users) share the same prefix.

<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>

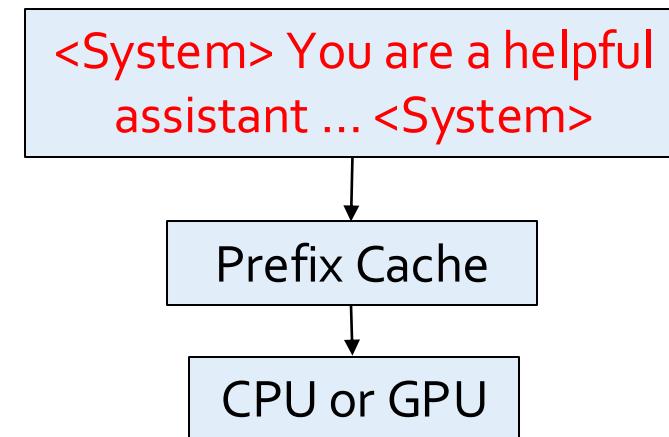
<System> You are a helpful assistant ... <System>

<User> Help me revise my email ... <User>

Prefix Caching

Prefix caching means **reusing the KV cache from a shared prefix of tokens** across multiple inference runs or generations, instead of recomputing it.

Most libraries (e.g., vLLM)
have this feature implemented:
`enable_prefix_caching=True`



But, can we slice the activations to fit them in different GPUs?
- Yes, by Tensor Parallelism

Distributed Training

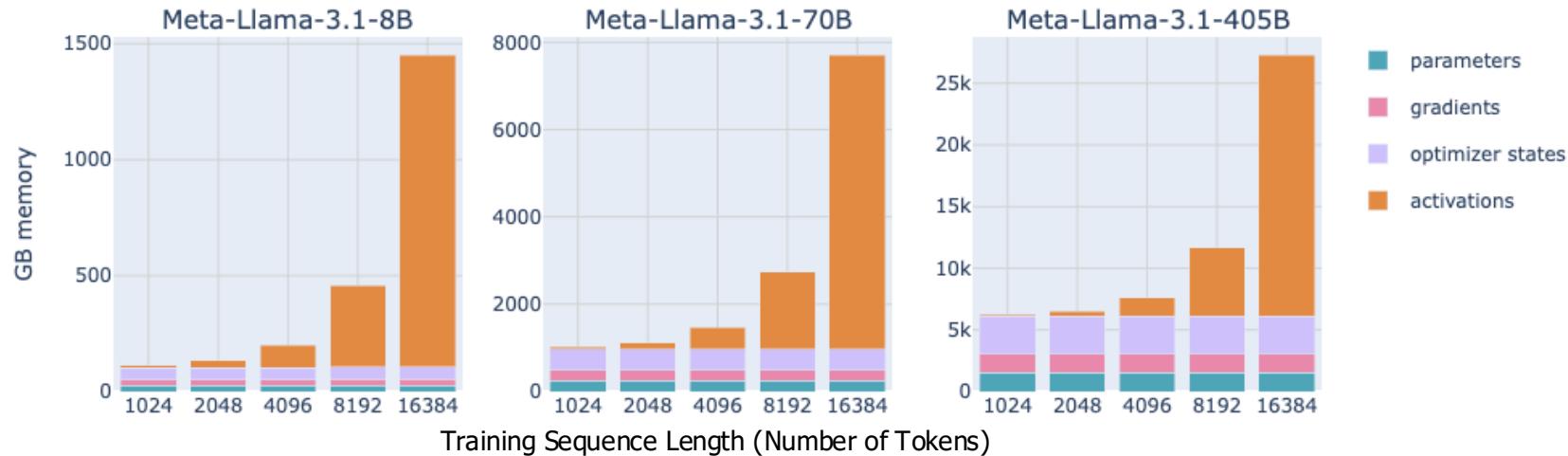
Motivation

How much GPU memory (**at least**) do we need to perform inference/training?
(batch size=1, ignoring the KV cache and optimizer states)

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

Where did all the memory go?

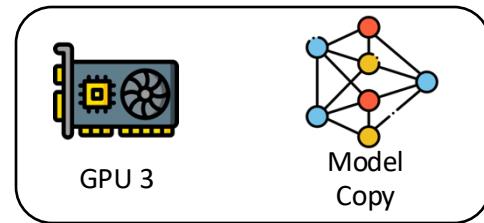
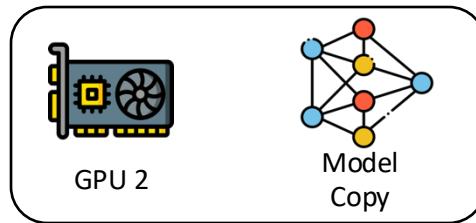
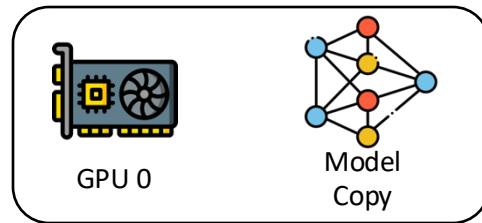
Longer sequences require much more memory in training!



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

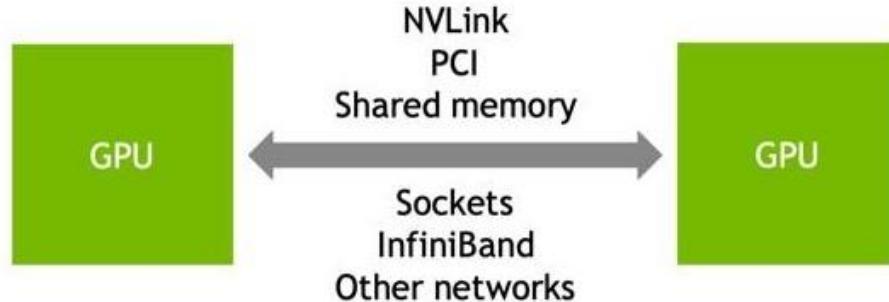
Distributed Training

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



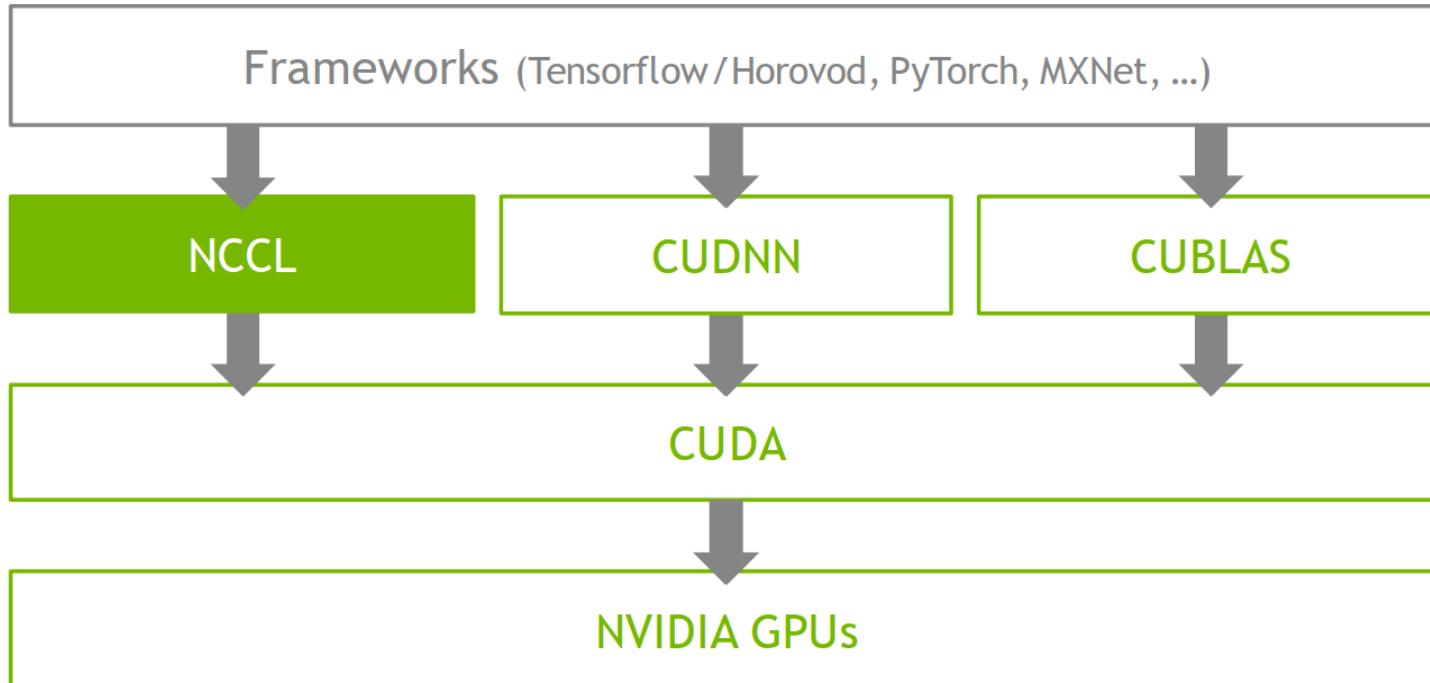
Brief on GPU Operations for Communicating Tensors

NCCL: NVIDIA Collective Communication Library

- Pronounced "Nickel"!!
 - NCCL provides routines to send and receive information within NVIDIA GPUs.
 - NCCL provides the following collective communication primitives :
 - AllReduce
 - Broadcast
 - Reduce
 - AllGather
 - ReduceScatter
 - AlltoAll
 - Gather
 - Scatter
- 
- The diagram illustrates the communication links between two NVIDIA GPUs. Two green rectangular boxes, each labeled 'GPU', are positioned on either side of a double-headed arrow. Above the arrow, the text 'NVLink' is written above 'PCI Shared memory'. Below the arrow, the text 'Sockets' is written above 'InfiniBand', which is further above 'Other networks'.

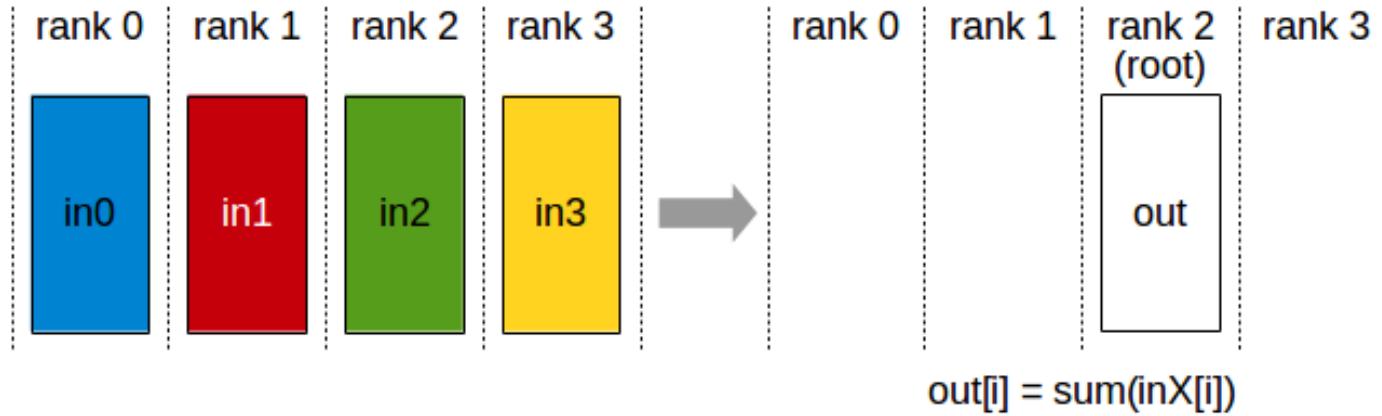
Documentation: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html#>
Github: <https://github.com/NVIDIA/nccl>

NCCL: NVIDIA Collective Communication Library



NCCL Operations: Reduce

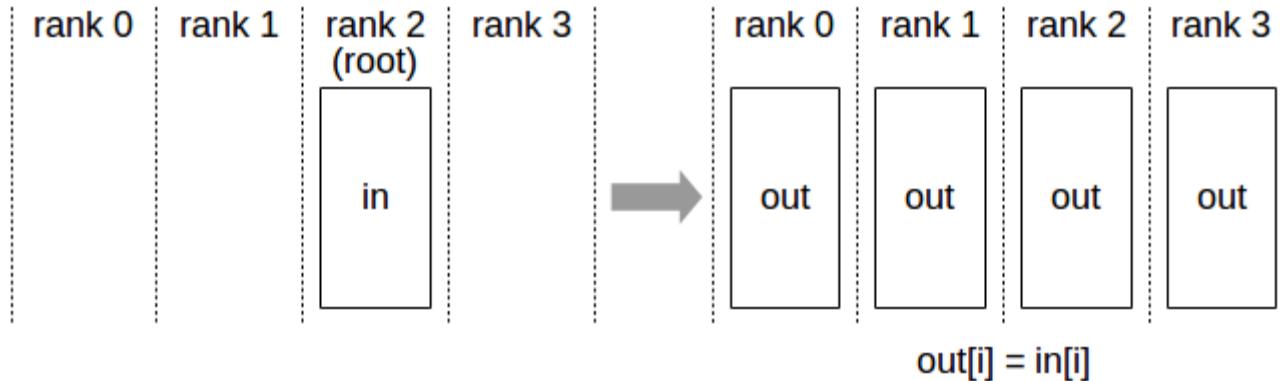
- Nvidia Collective Communications Library (NCCL) - A library developed to provide inter-GPU communications primitives (operations)
- Reduce: *Sums* over all *tensors* and stores it in a root GPU



NCCL (NVIDIA Collective Communication Library): NVIDIA's high-performance library for multi-GPU and multi-node communication: <https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html>

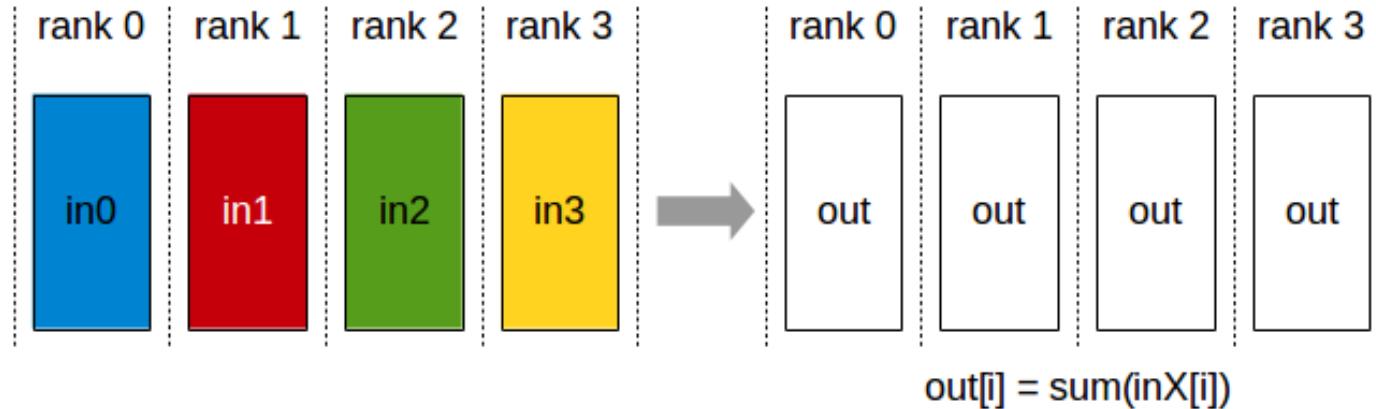
NCCL Operations: Broadcast

- Broadcast: Duplicates one tensor to all GPUs



NCCL Operations: All Reduce

- All Reduce = Reduce + Broadcast
 - = Sum over input tensors, then duplicate it to all GPUs

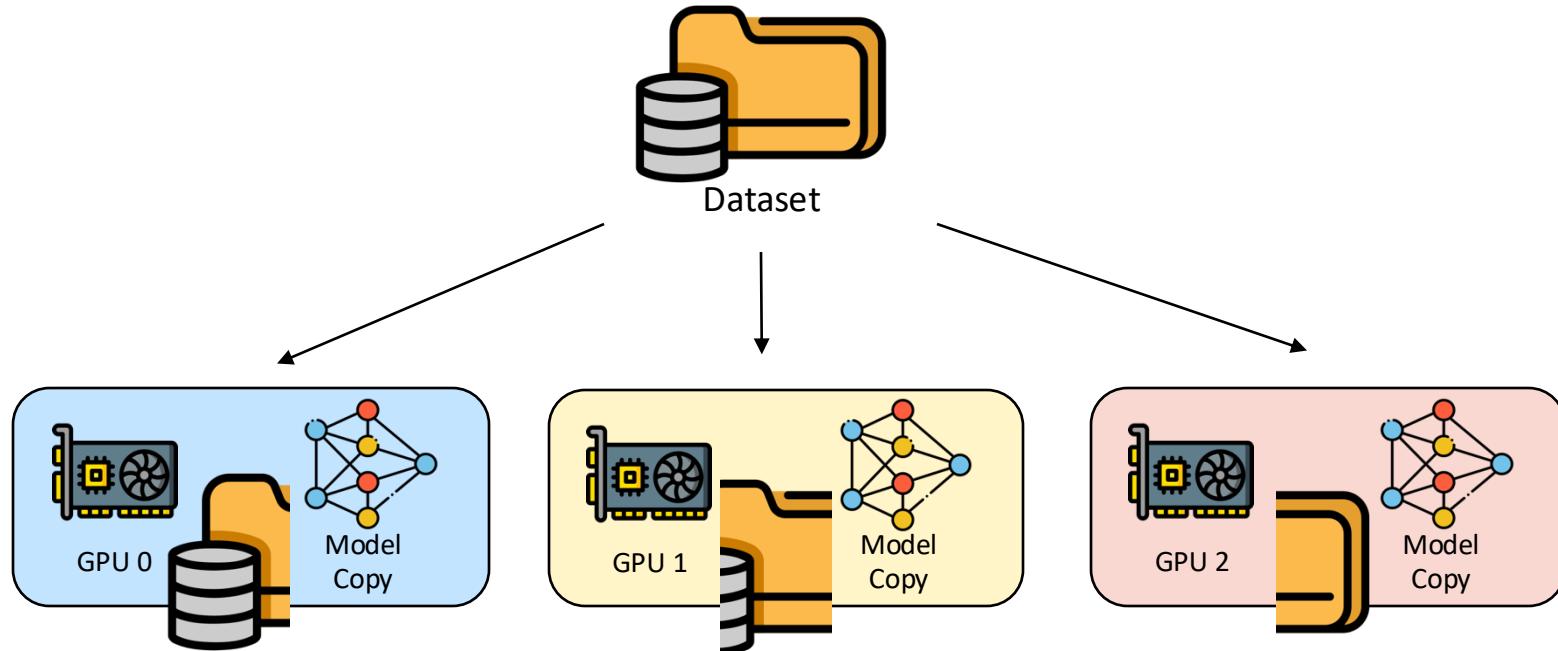




Back to distributed training!



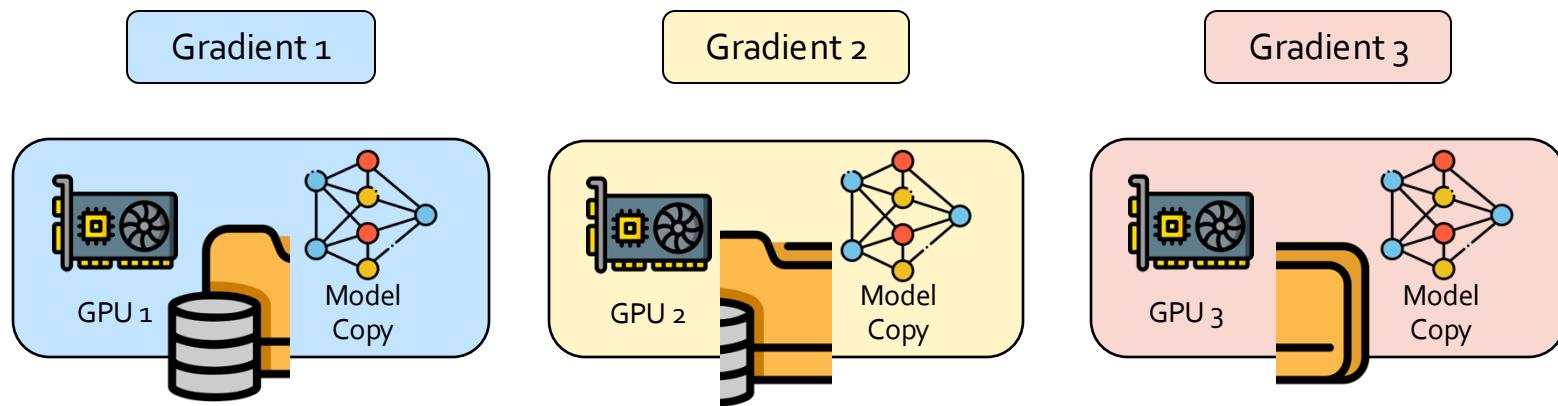
Naïve Data Parallelism



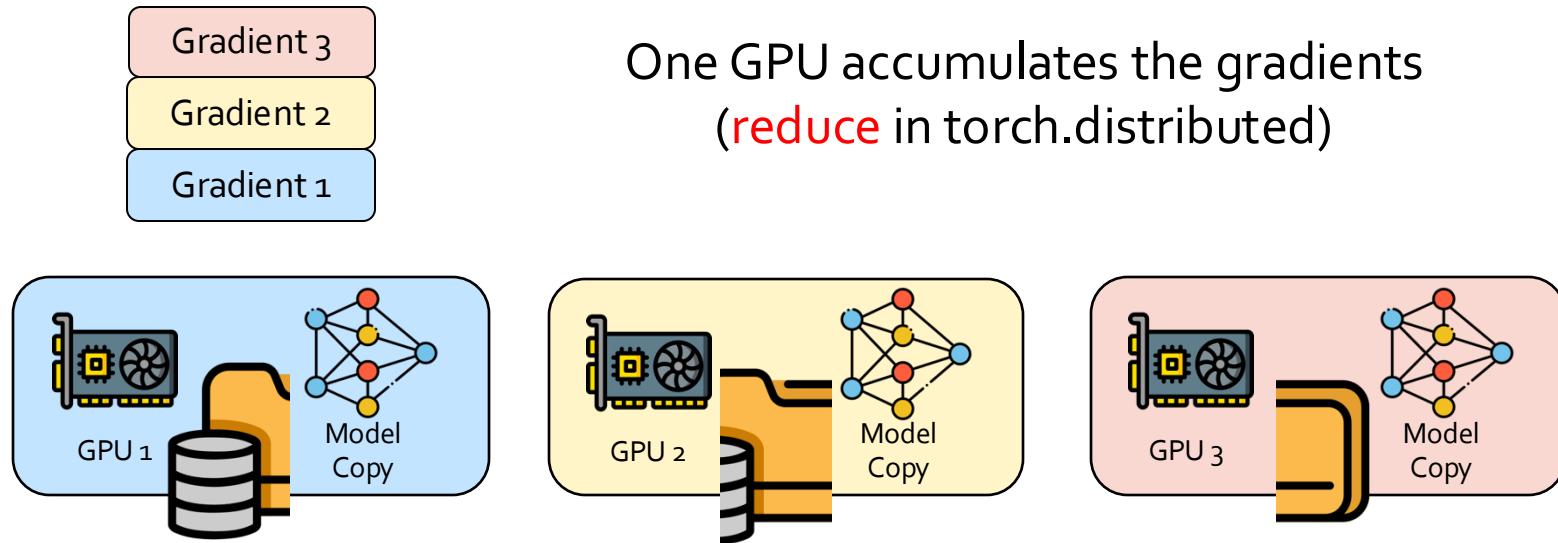
First, we want to shard the dataset and feed them into different GPUs
How do we update the parameters?

Naïve Data Parallelism

Each GPU compute gradient with a single shard of data

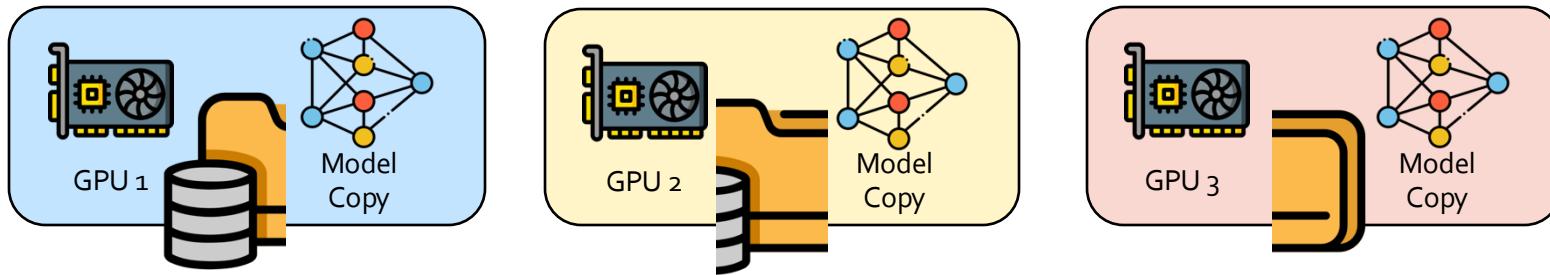
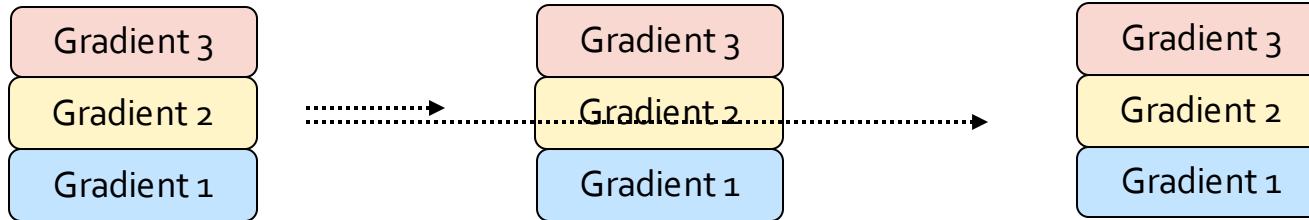


Naïve Data Parallelism



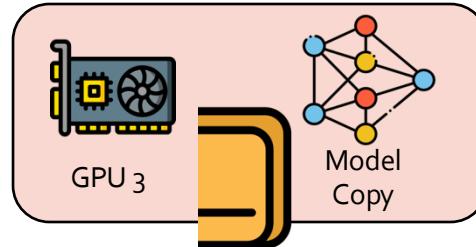
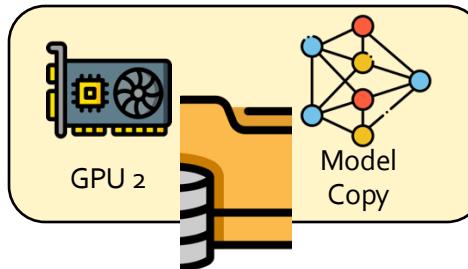
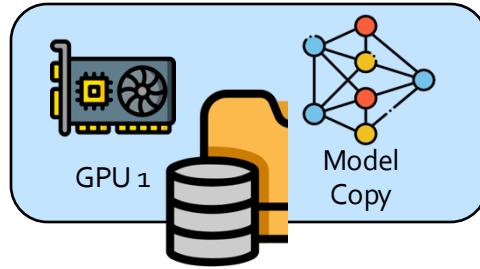
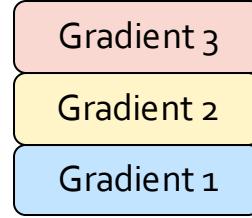
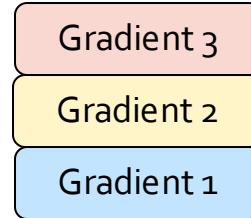
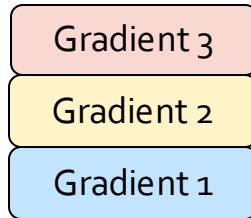
Naïve Data Parallelism

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



Naïve Data Parallelism

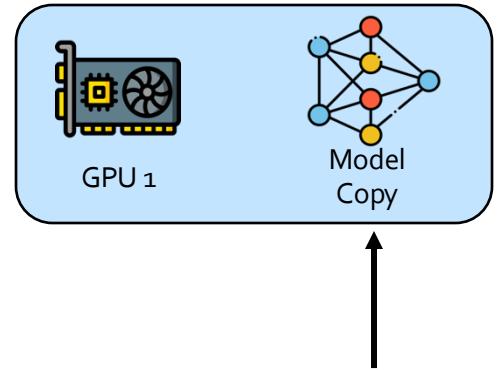
Accumulate gradients across all GPUs and perform gradient updates
(`all_reduce` in `torch.distributed`)



What is wrong with Naïve DP

- In *naïve DP*, each GPU holds a *full copy* of everything related to training — not just the model parameters.

Component	Precision	Bytes per param	Purpose
Model parameters	FP16/BF16	2	The half-precision weights used in forward/backward passes
Gradients	FP16/BF16	2	The half-precision gradients computed during backprop
Master weights	FP32	4	Full-precision version used for stable updates in mixed-precision training
Adam first moment (m)	FP32	4	Tracks running average of gradients
Adam second moment (v)	FP32	4	Tracks running average of squared gradients



For each parameter, you actually need **five versions**, which together take **16 bytes per parameter!**

That's why naïve DP quickly runs out of memory as model size grows

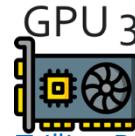
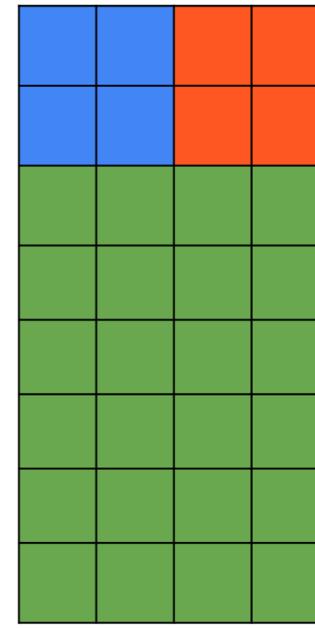
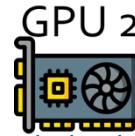
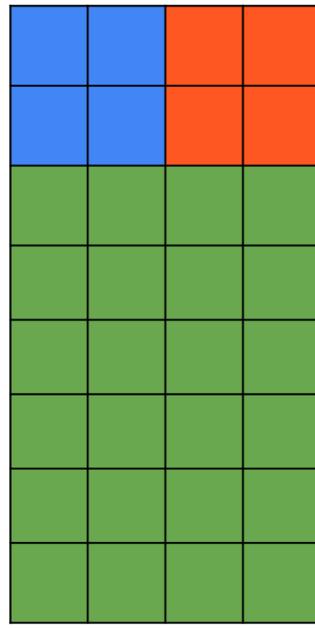
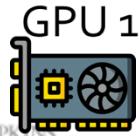
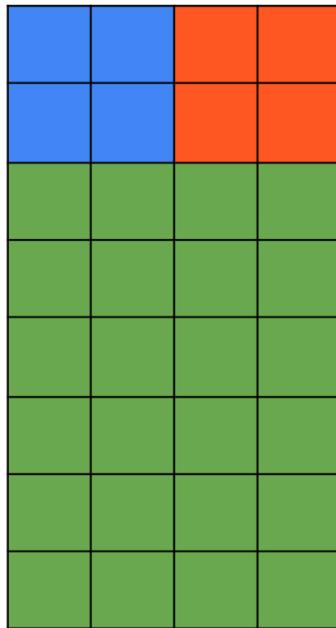
Memory/GPU for a 7.5B model:
7.5B * 16 bytes = **120 GB!**

Naïve DP: Summary

- Each GPU holds a *full copy* of the model. The training data is split into batches distributed across GPUs; each computes gradients locally, then gradients are averaged (e.g., via all-reduce) before updating model parameters.
- **Pros:** Simple, widely supported.
- **Cons:** Model parameters and optimizer states are fully replicated → high memory cost, limiting scalability.

Stage	What's Sharded	Approx bytes / param	Memory / GPU (7.5 B params)
Naïve DP	None	16 B	120 GB

Naïve DP – Requires too much memory!



Parameters
 Gradients
 Optimizer States

ZeRO: Sharding Optimizer States

- **ZeRO (Zero Redundancy Optimizer)** reduces memory use in distributed training by **sharding** model states across GPUs instead of replicating them.
 - **Stage 1:** shard optimizer states
 - **Stage 2:** + shard gradients
 - **Stage 3:** + shard model parameters
- Result: Dramatically reduced memory use. Integrated to **DeepSpeed** and **PyTorch FSDP**.

Stage	What's Sharded	Approx bytes / param	Memory / GPU (7.5 B params)
Naïve DP	None	16 B	120 GB
ZeRO-1	Optimizer states	≈ 6–7 B	50 GB
ZeRO-2	+ Gradients	≈ 5 B	40 GB
ZeRO-3	+ Parameters	≈ 2 B	15 GB

More on GPU Operations for Communicating Tensors

Quick pulse check

Four GPUs (GPU_0 – GPU_3) each start with values [2, 4, 6, 8].

After a **Reduce (sum) to GPU₀**, what are the final values on all GPUs?

GPU_0 : 20

GPU_1 – GPU_3 : unchanged / discarded (they don't receive the result).

Quick pulse check (2)

GPU₀ starts with **5**, and GPUs _{1–3} hold [1, 2, 3].

After a **Broadcast** from GPU₀, what are the final values on all GPUs?

GPU₀–GPU₃: **[5, 5, 5, 5]**

Quick pulse check (3)

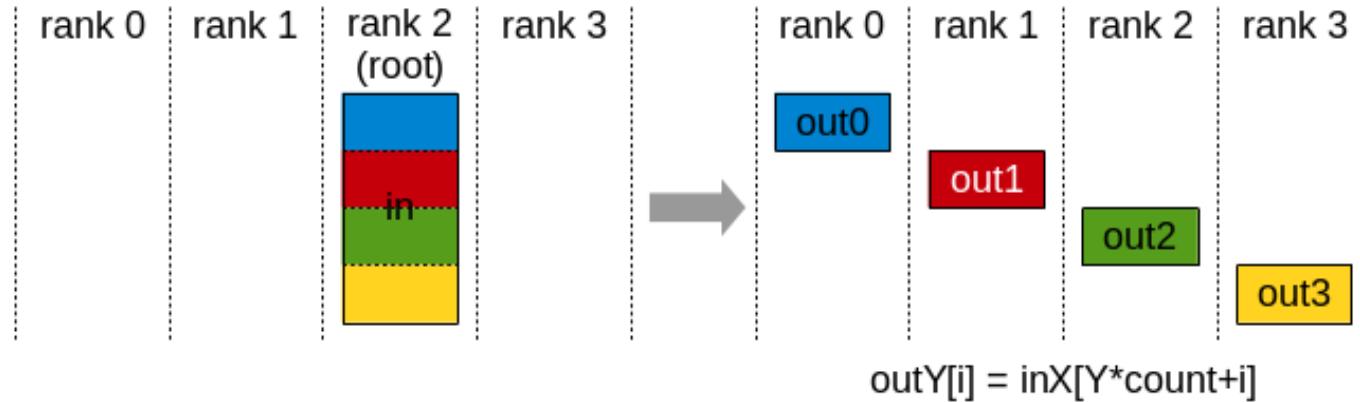
Four GPUs (GPU_0 – GPU_3) each start with $[2, 4, 6, 8]$.

After an **All-Reduce (sum)**, what are the final values on all GPUs?

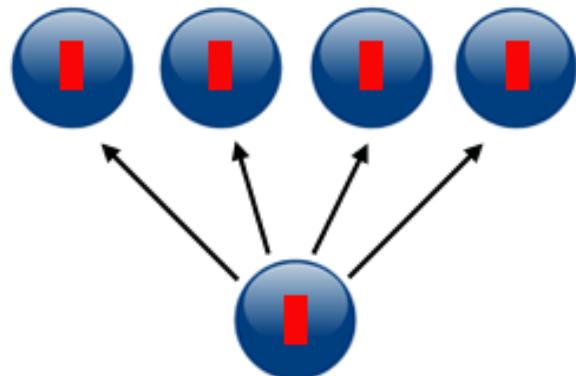
GPU_0 – GPU_3 : **[20, 20, 20, 20]**

NCCL Operations: Scatter

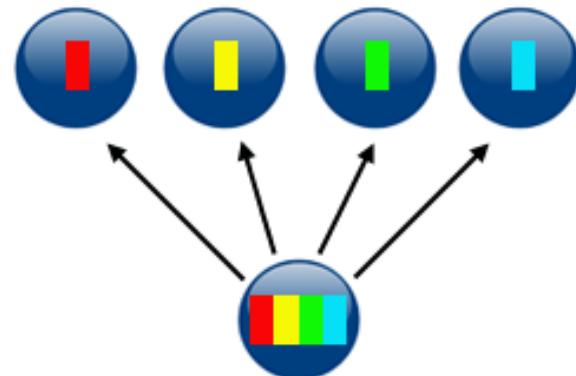
- Split a tensor into chunks and send *one chunk to each GPU*.



Broadcast vs Scatter



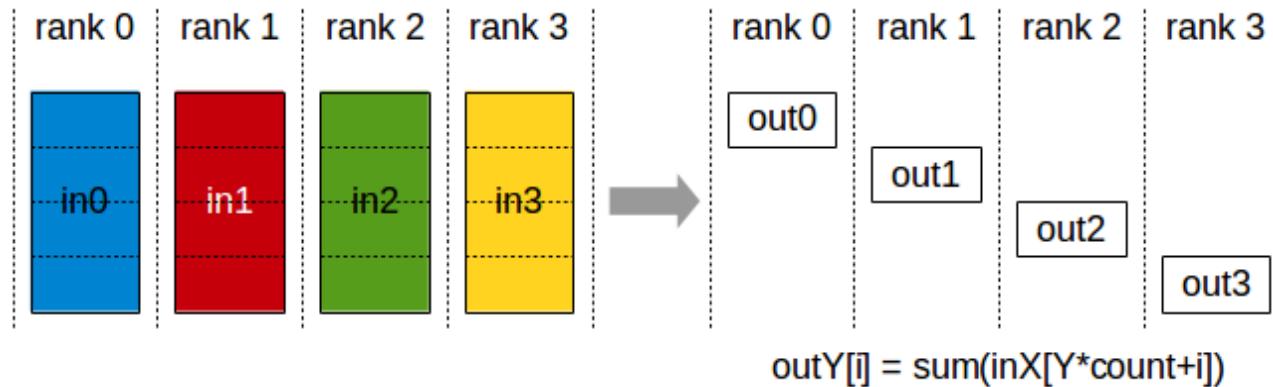
broadcast



scatter

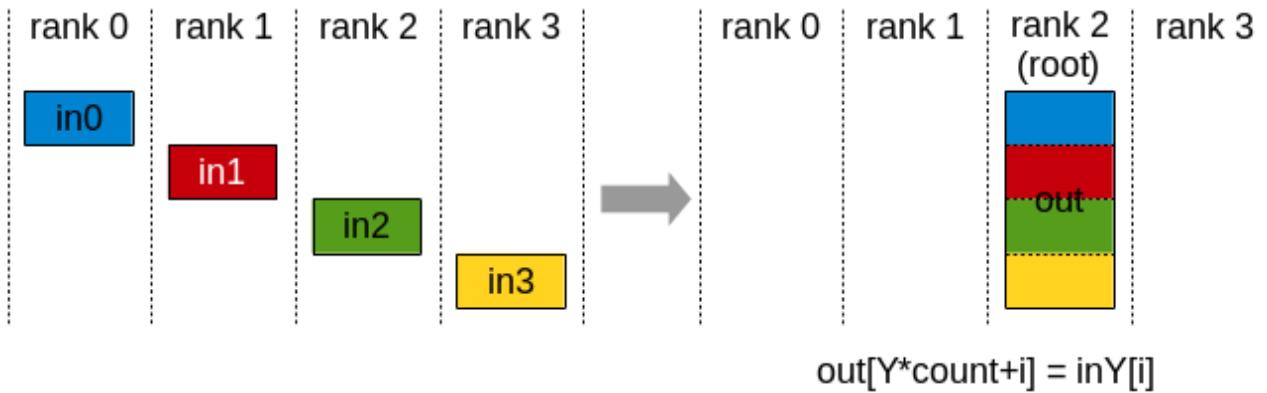
NCCL Operations: Reduce Scatter

- Each GPU stores the sum of a shard of the input.

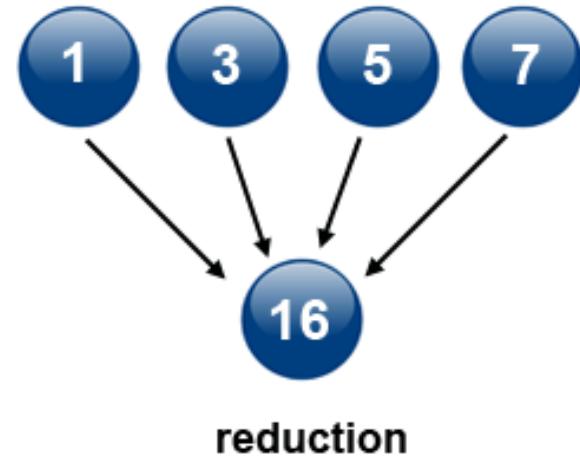
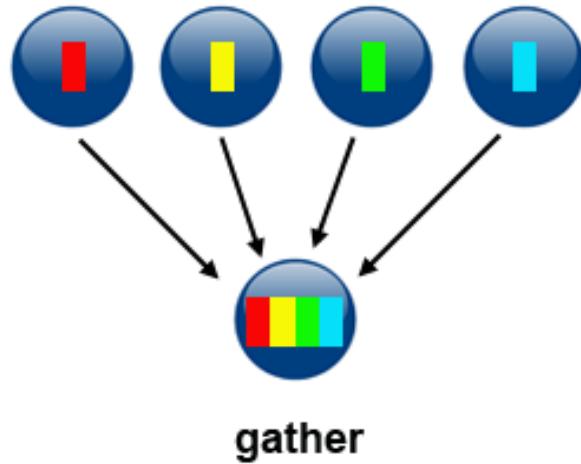


NCCL Operations: Gather

- **Gather** is the opposite of **Scatter**.
- Collect chunks of data from multiple GPUs and **assemble them into one larger tensor** on a single GPU (or all GPUs, depending on the variant).

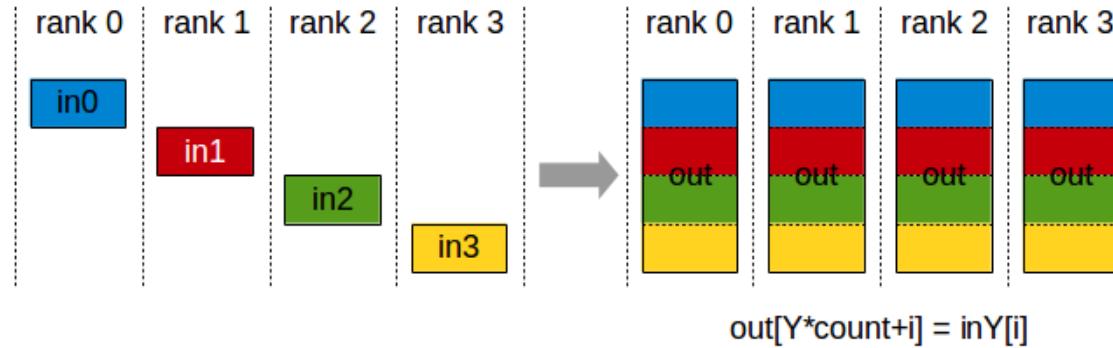


Diversion: Gather vs Reduce



Quiz: NCCL Operations: All Gather

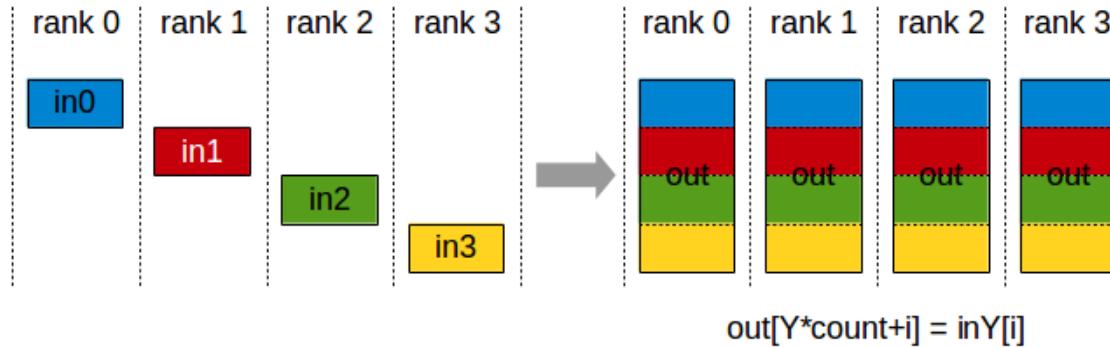
- Everyone sends their piece → everyone receives the full set.
- Every GPU performs a ___?___ operation in parallel.



- A. Reduce B. Broadcast C. Reduce_scatter

NCCL Operations: All Gather

- Everyone sends their piece → everyone receives the full set.
- Every GPU performs a broadcast operation in parallel.



- A.Reduce B. Broadcast C. Reduce_scatter



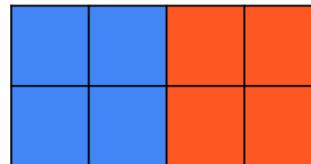
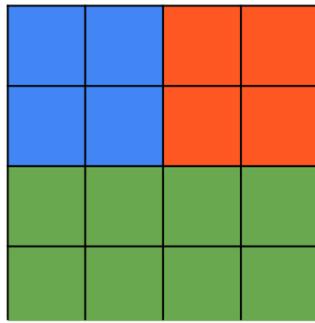
Back to ZeRO!



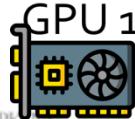
ZeRO Stage 1: Sharding Optimizer States

- Only **optimizer states** are **partitioned (sharded)** among GPUs:
 - Each GPU keeps just **1/N of the optimizer states**, where N is the number of GPUs.
- Model parameters and gradients are still fully replicated (as in naïve DP).
- When it's time to update parameters, GPUs **communicate** to access the parts of the optimizer state they need, apply updates, and synchronize the results.

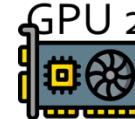
ZeRO Stage 1: Sharding Optimizer States



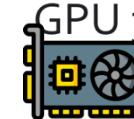
Parameters
Gradients
Optimizer States



GPU 1



GPU 2

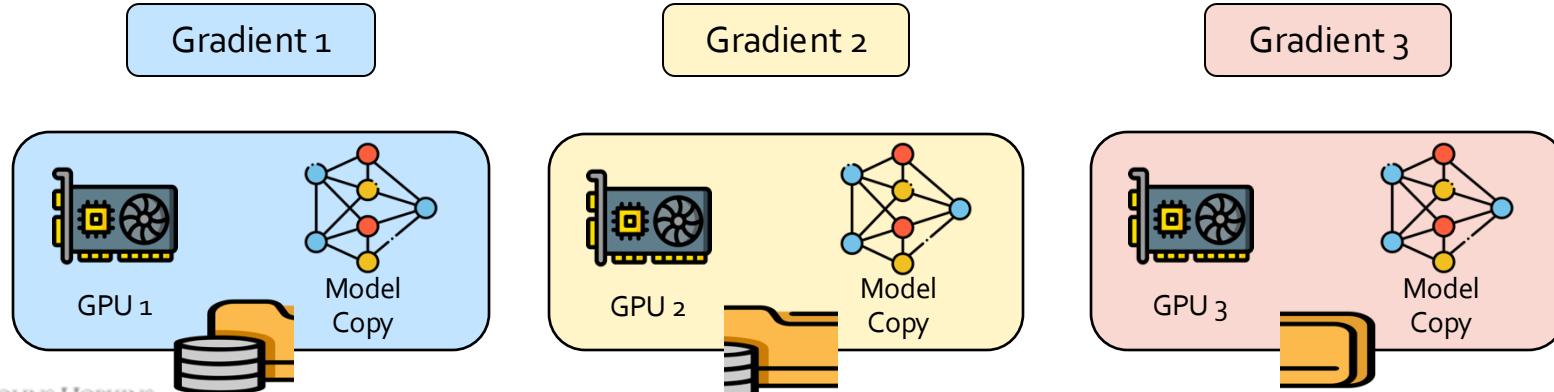


GPU 3

ZeRO Stage 1: How it works

Update Parameters

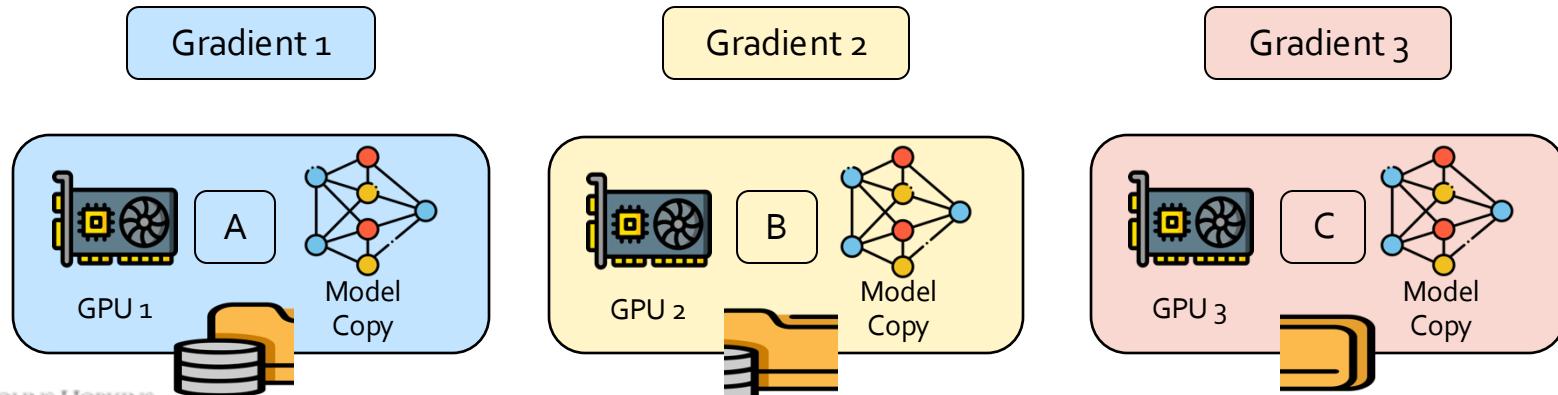
Each GPU compute gradient with a single shard of data
(The same as naïve DP)



ZeRO Stage 1: How it works

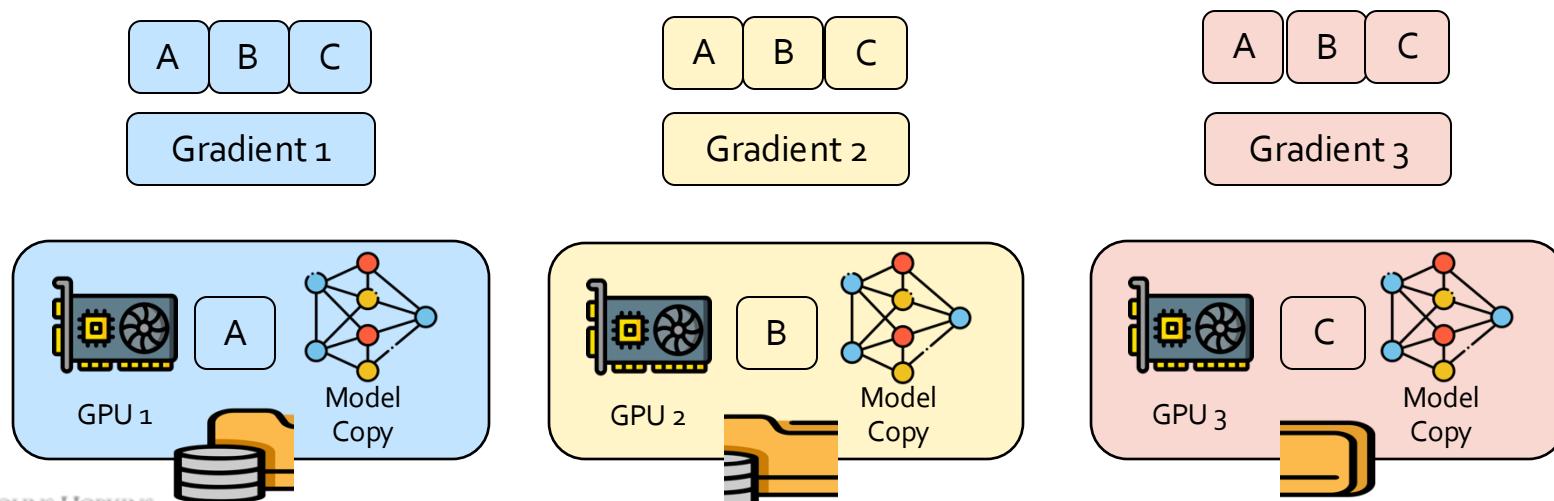
Assuming that

- GPU₁ stores optimizer states for parameters A,
- GPU₂ stores optimizer states for params B,
- GPU₃ stores optimizer states for params C



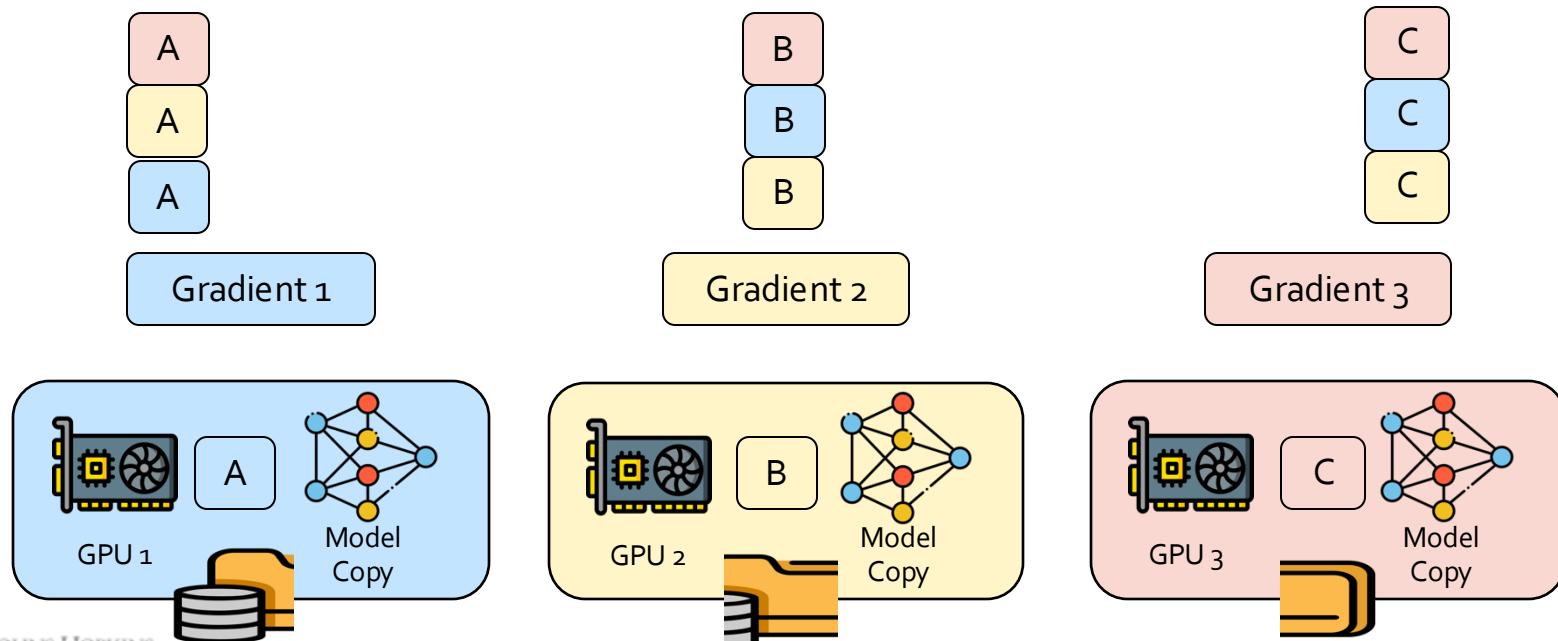
ZeRO Stage 1: How it works

Split / shard the optimizer states into 3 parts!



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 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.

Updated A

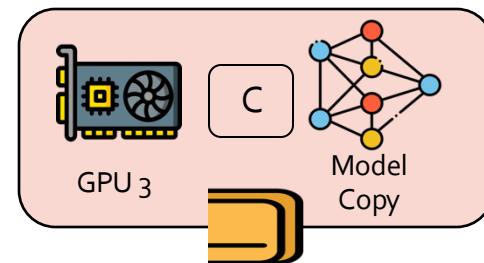
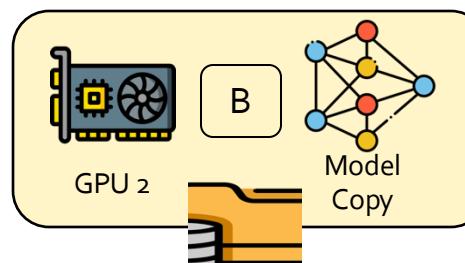
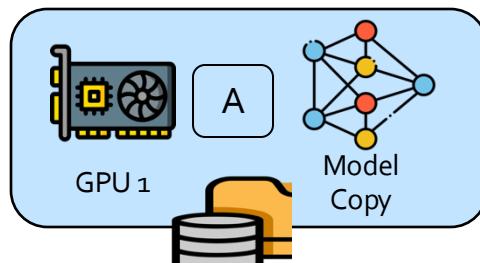
Updated B

Updated C

Gradient 1

Gradient 2

Gradient 3

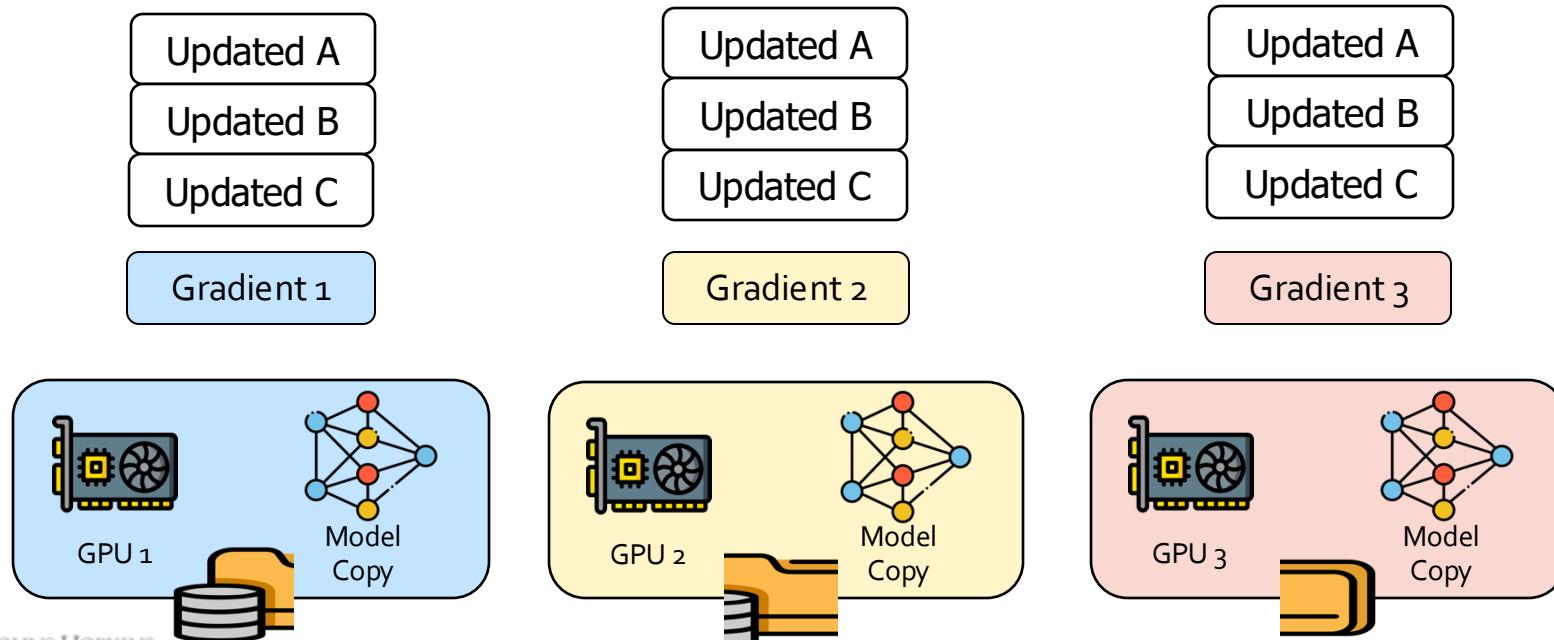


ZeRO Stage 1: How it works

Each GPU sends updated params to every other GPU.

Finishing `optimizer.step()`. (`all_gather` in `torch.distributed`)

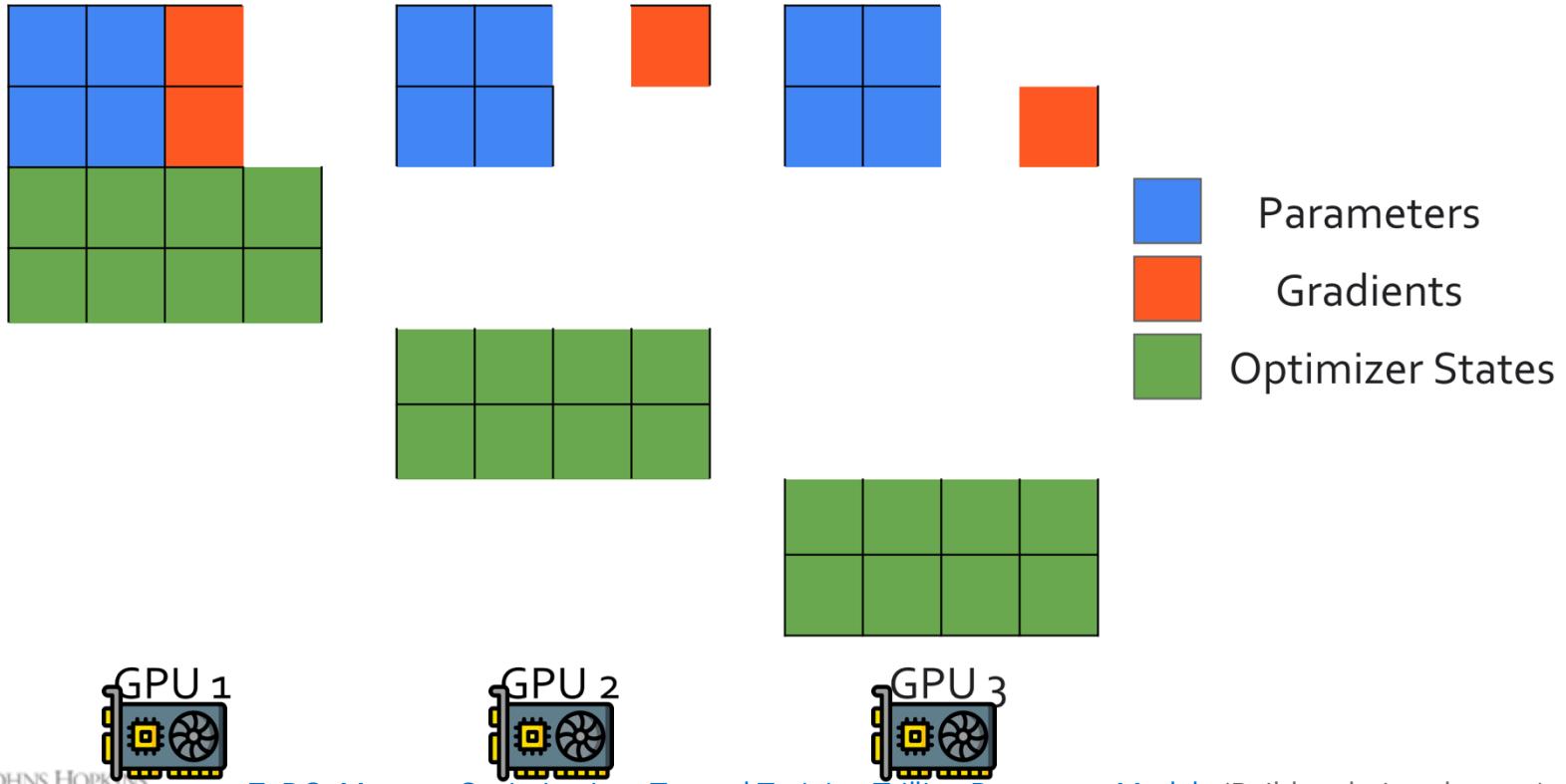
After all_gather, every GPU has an updated copy of the model



Summary: ZeRO 1

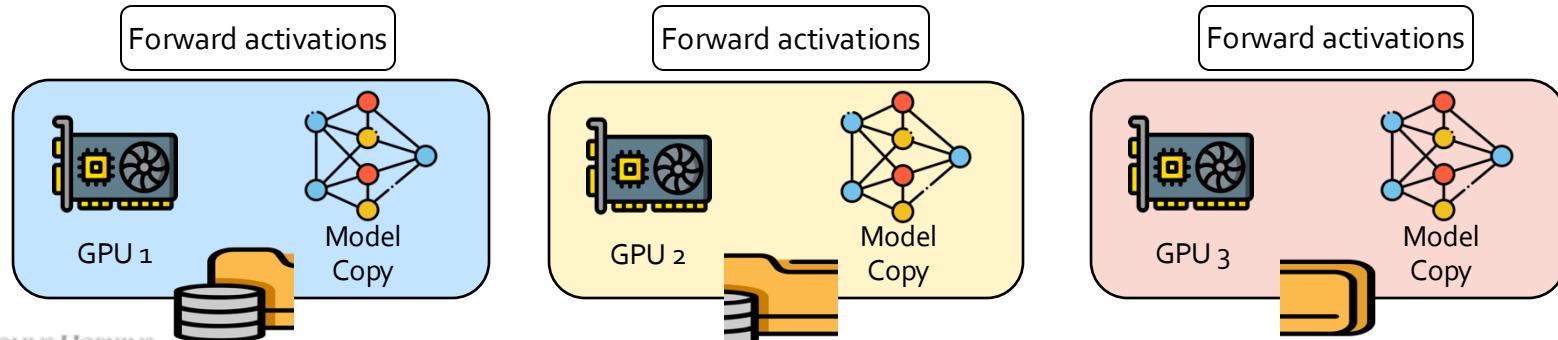
- It shards only the **optimizer states** (e.g., Adam's m, v) across GPUs.
- While **model parameters** and **gradients** remain fully replicated.
- Each GPU individually perform gradient updates
- This removes redundant copies of the heaviest memory component, giving roughly a **1 / N reduction** in optimizer-state memory (for N GPUs) with minimal-ish communication overhead.
- Basically free! (Compared to Naïve Data Parallelism)

ZeRO Stage 2: Sharding Gradients



ZeRO Stage 2: How it works

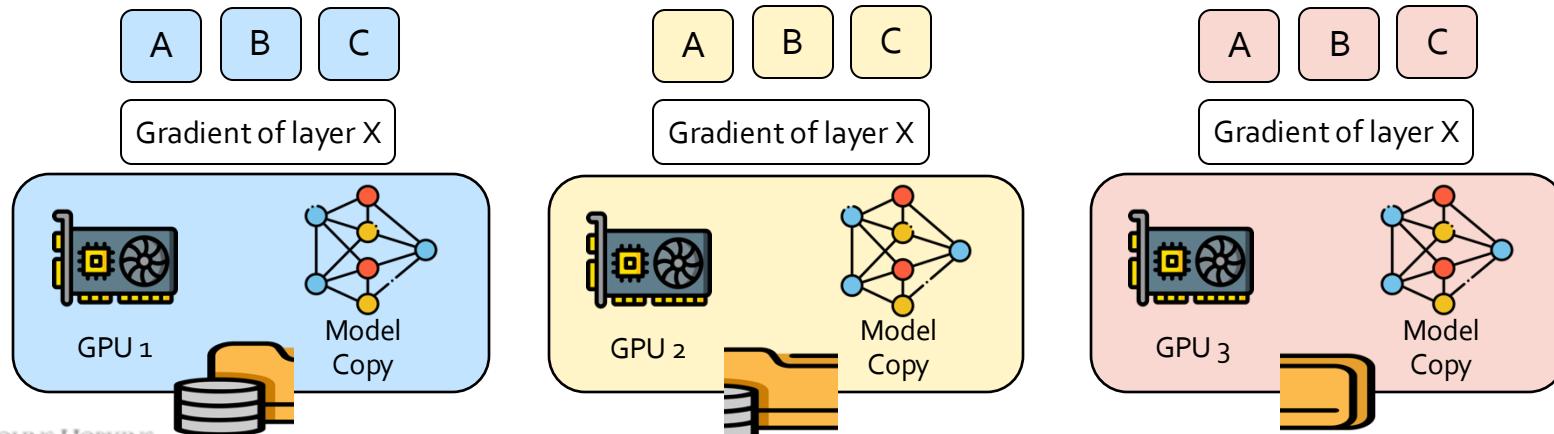
- Model params are replicated. But we're going to shard gradients + optimizer states.
- **Forward pass:** Each GPU holds a full copy of the model parameters (as in Stage 1). No extra communication happens here.



ZeRO Stage 2: How it works

Backward pass:

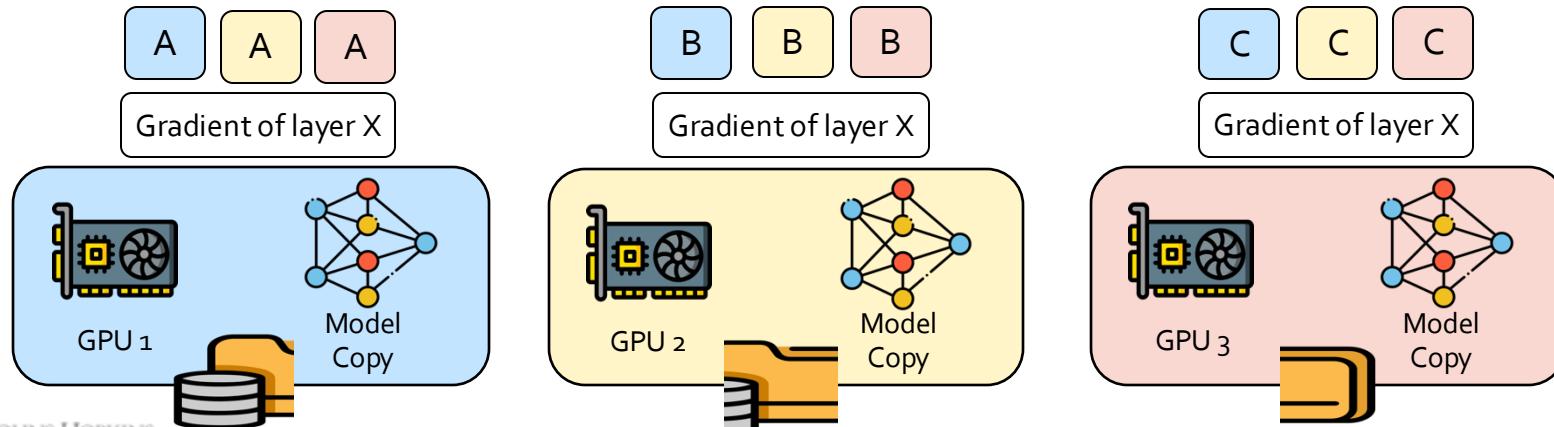
- Each GPU computes gradients for its local batch.
- Instead of storing *all* gradients, each GPU keeps **only its shard** of the gradient.
- As soon as the gradients of layer X are computed during backprop, shard them across GPUs.
- This uses reduce-scatter operation.



ZeRO Stage 2: How it works

Backward pass:

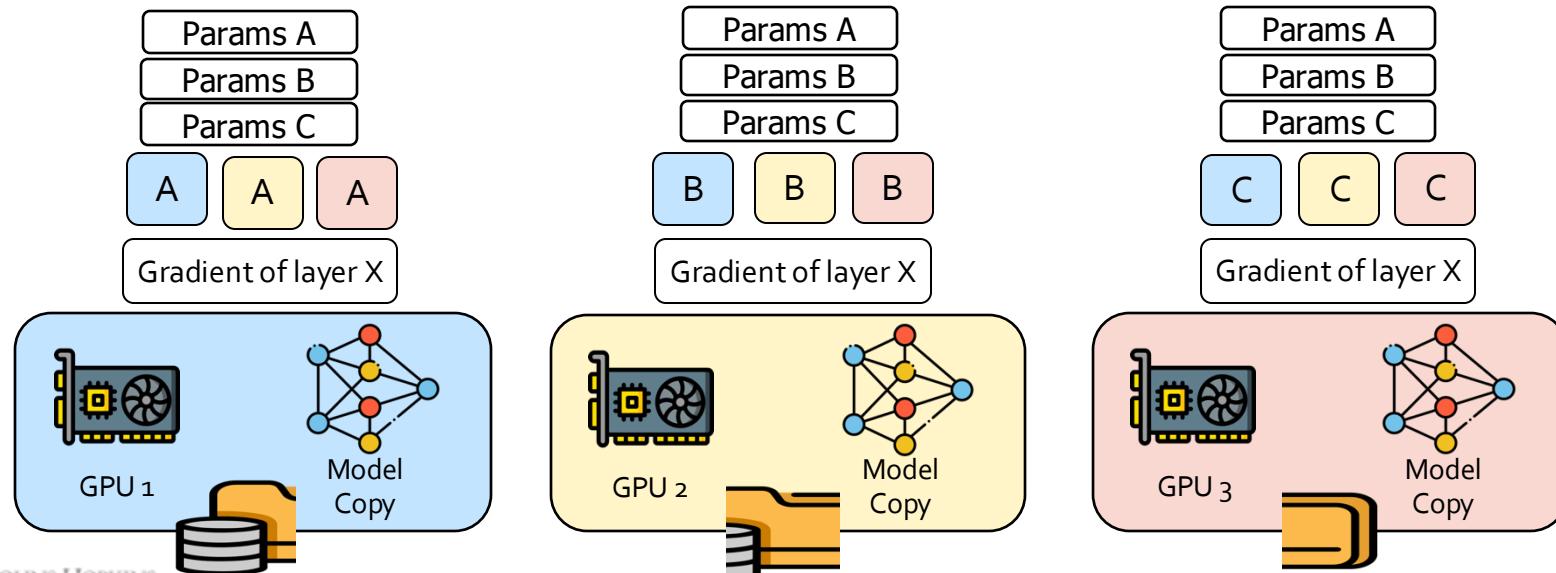
- Each GPU computes gradients for its local batch.
- Instead of storing *all* gradients, each GPU keeps **only its shard** of the gradient.
- As soon as the gradients of layer X are computed during backprop, shard them across GPUs.
- This uses reduce-scatter operation **incrementally during backpropagation**, layer by layer, rather than all at once.



ZeRO Stage 2: How it works

Optimizer step:

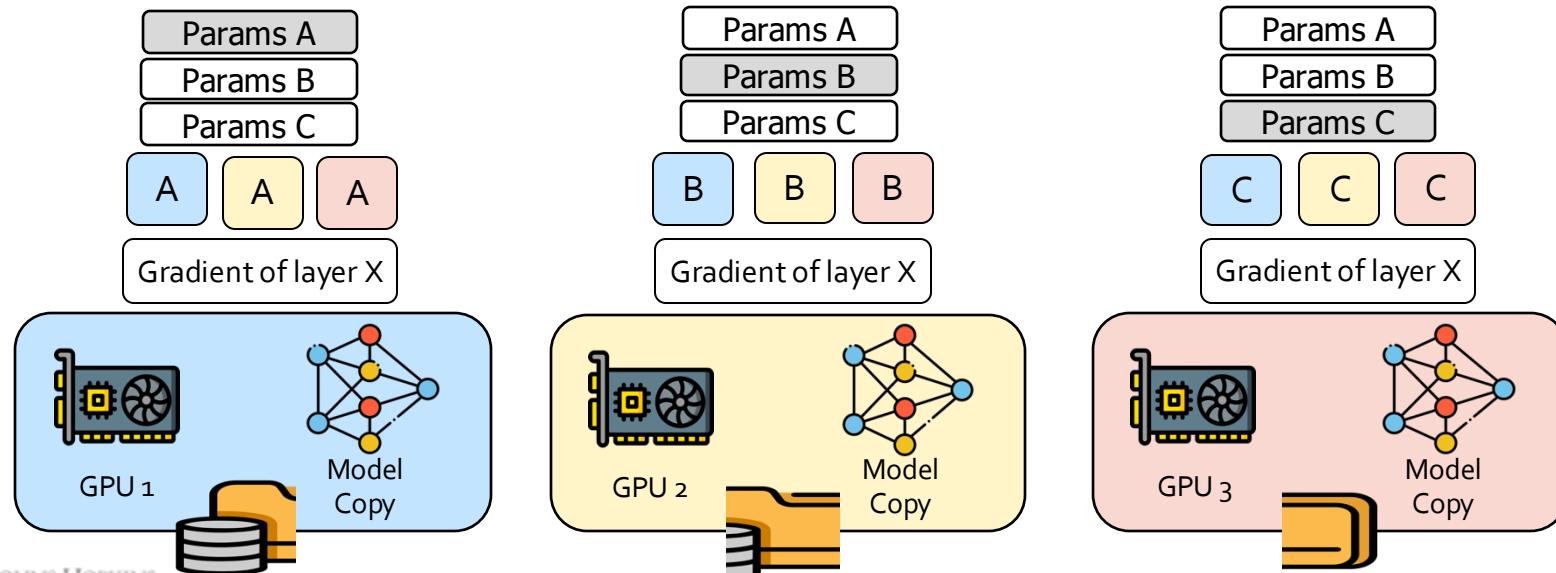
- It updates only its own parameter using its shard of grads + optimizer states.



ZeRO Stage 2: How it works

Optimizer step:

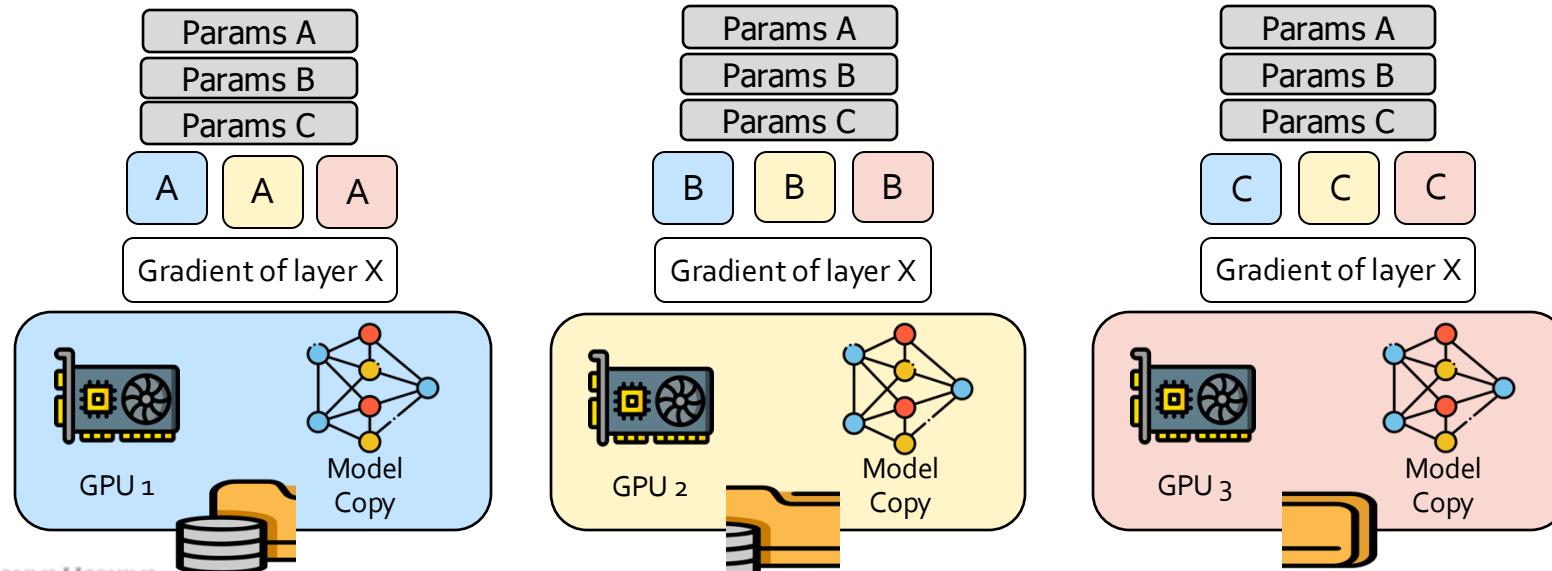
- It updates only its own parameter using its shard of grads + optimizer states.



ZeRO Stage 2: How it works

Optimizer step:

- It updates only its own parameter using its shard of grads + optimizer states.
- Parameters are still replicated, so updates must be broadcast after each step to keep all GPUs in sync. (done with **all-gather**)

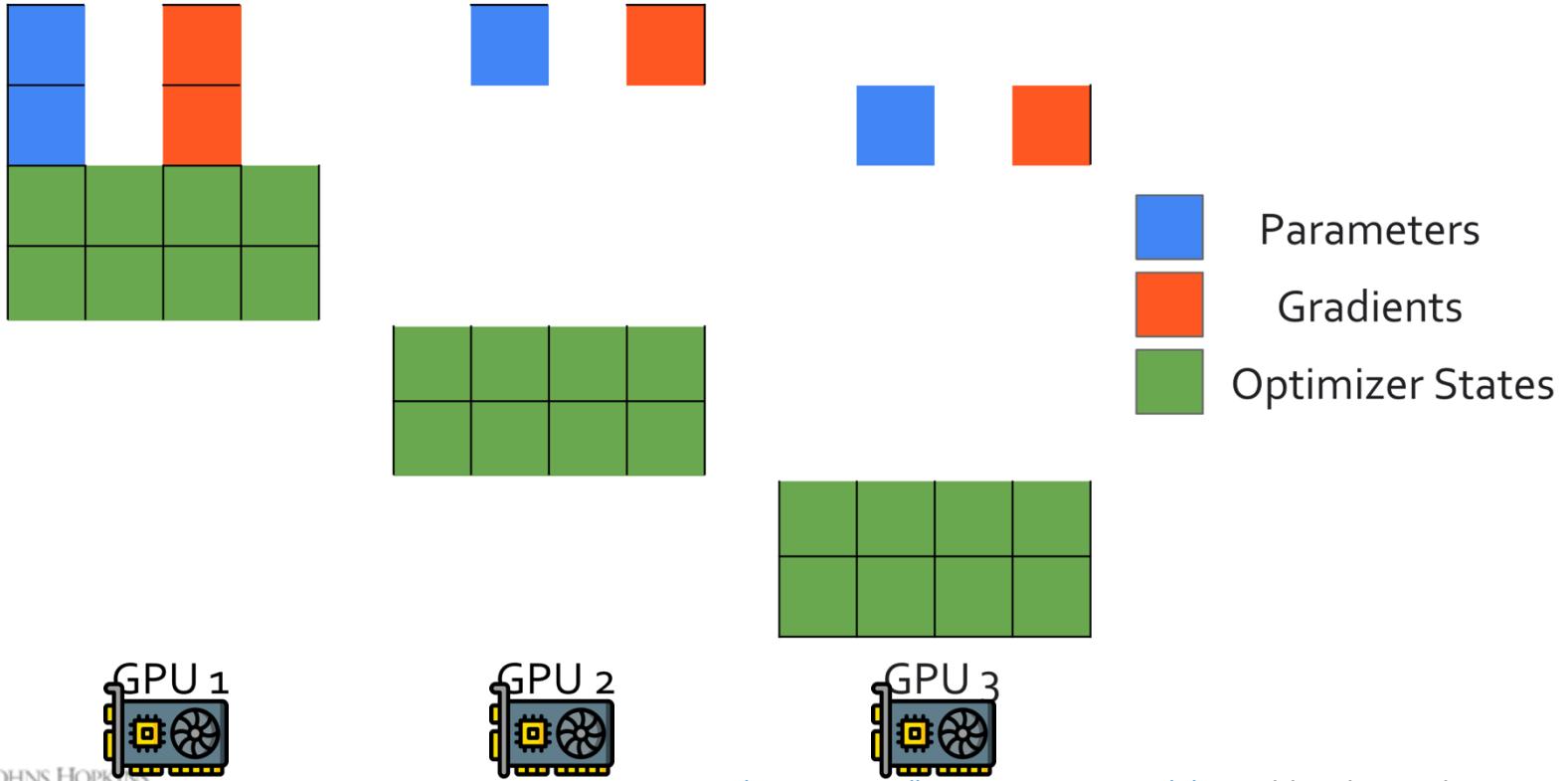


Summary: ZeRO Stage 2

- Builds on Stage 1 by sharding gradients in addition to optimizer states.
- Memory savings: no duplicate gradients or optimizer states.
- Trade-off: slightly higher communication overhead each step.
 - Communication is not bad since it's amortized!!

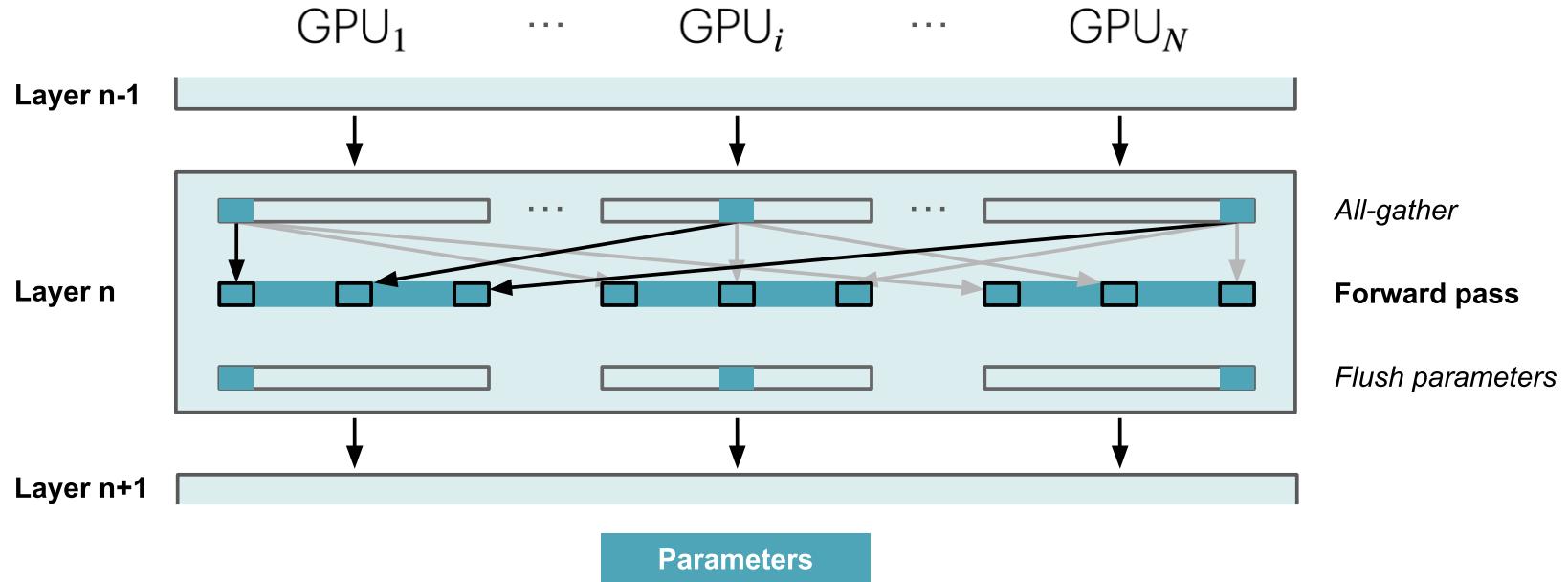
Stage	What's Sharded	Approx bytes / param	Memory / GPU (7.5 B params)
Naïve DP	None	16 B	120 GB
ZeRO-1	Optimizer states	\approx 6–7 B	50 GB
ZeRO-2	+ Gradients	\approx 5 B	40 GB

ZeRO-3 (aka FSDP): Shard Everything!



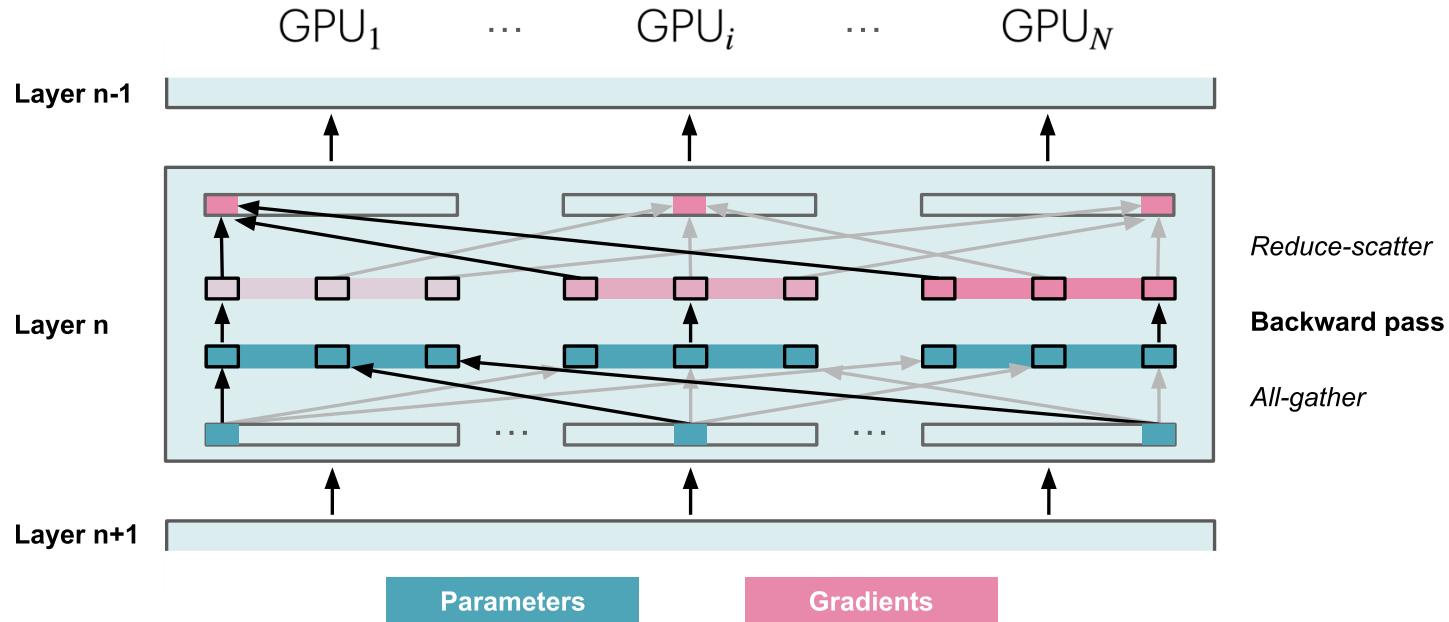
ZeRO Stage 3: How it works (simplified)

When a layer is needed for computation, its parameters are All-Gathered from all GPUs just in time.



ZeRO Stage 3: How it works (simplified)

After computing gradients, those gradients are Reduce-Scattered back to distribute and free memory.



Zero3 Recap

- Shards **everything** — parameters, gradients, and optimizer states — across GPUs.
- Provides **maximum memory savings ($\sim 1/N$ per GPU)**, enabling extremely large models.
- Comes with **higher communication cost**, since parameters must be gathered repeatedly during training.

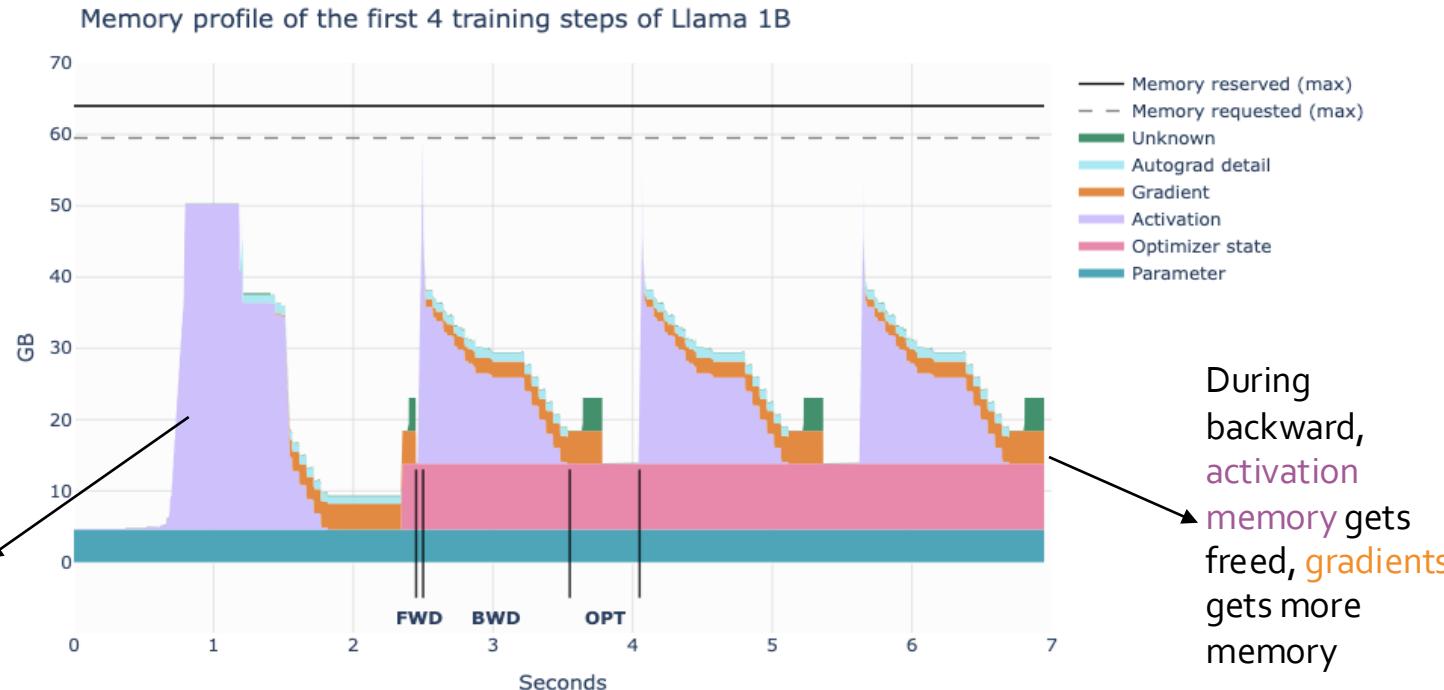
Communication Costs

- Naïve Data Parallel: 2x parameter (all_reduce)
- ZeRO-1: ~1x 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! (the communication is amortized)
- ZeRO-3: ~3x parameter – which can be quite slow.

Q: Where do we store the KV Cache?

- Depends.
 - If you're doing single-GPU inference, it sits on the GPU that computes attention.
 - KV cache can be offloaded to CPU (RAM) (if GPU is running out of space) but adds latency.
- **Data Parallel (incl. ZeRO):** Each rank keeps the **full KV** for the sequences it's serving (belongs to activations, not optimizer/params). Activation checkpointing can trade KV m for recompute; ZeRO doesn't shard KV.
- **Tensor Parallel (TP):** KV is **sharded across TP ranks** (each rank holds the slice of KV that corresponds to its tensor partition). Collective reads happen implicitly during attention matmuls.
- **Pipeline Parallel (PP):** Each pipeline stage stores KV **only for its own layers**.

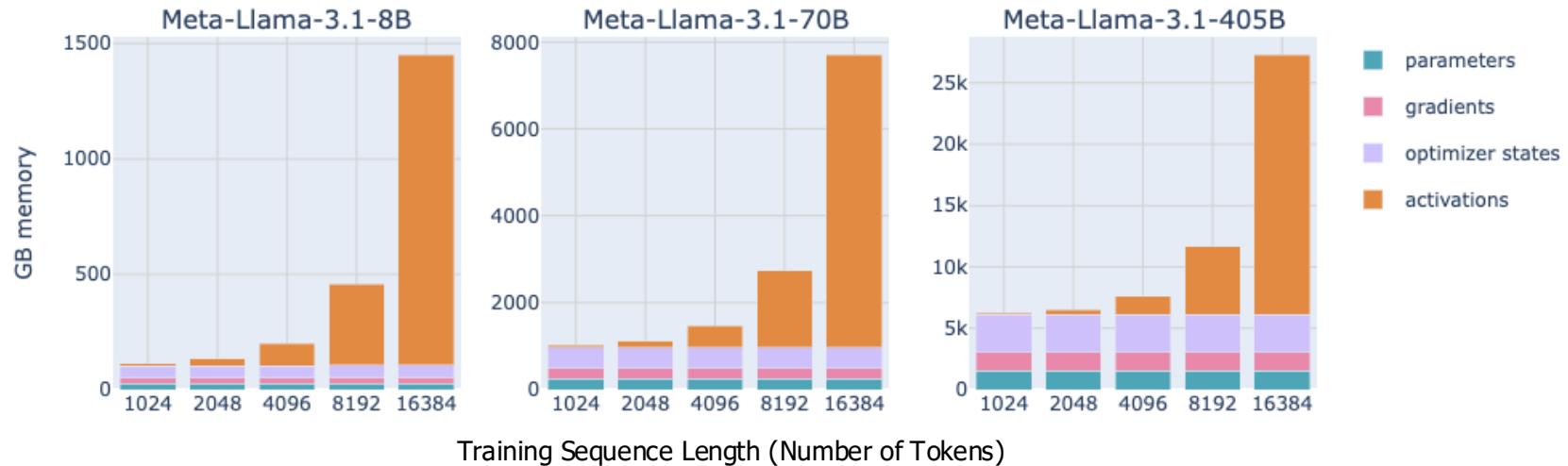
Memory consumption is not static



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

Where did all the memory go?

So far, we dealt with the **optimizer states**
but what about the **activations**?

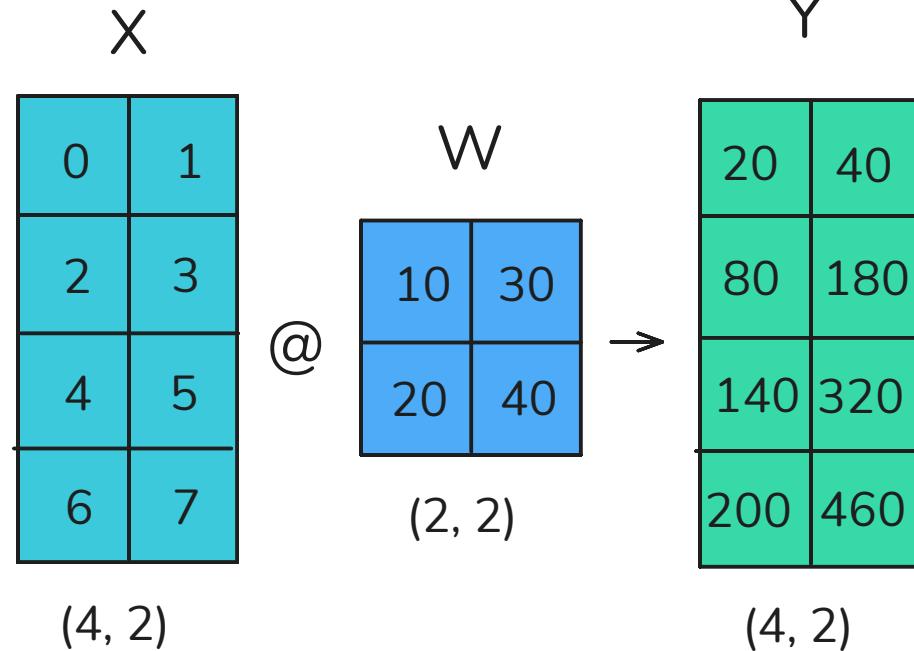


Source: <https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html>

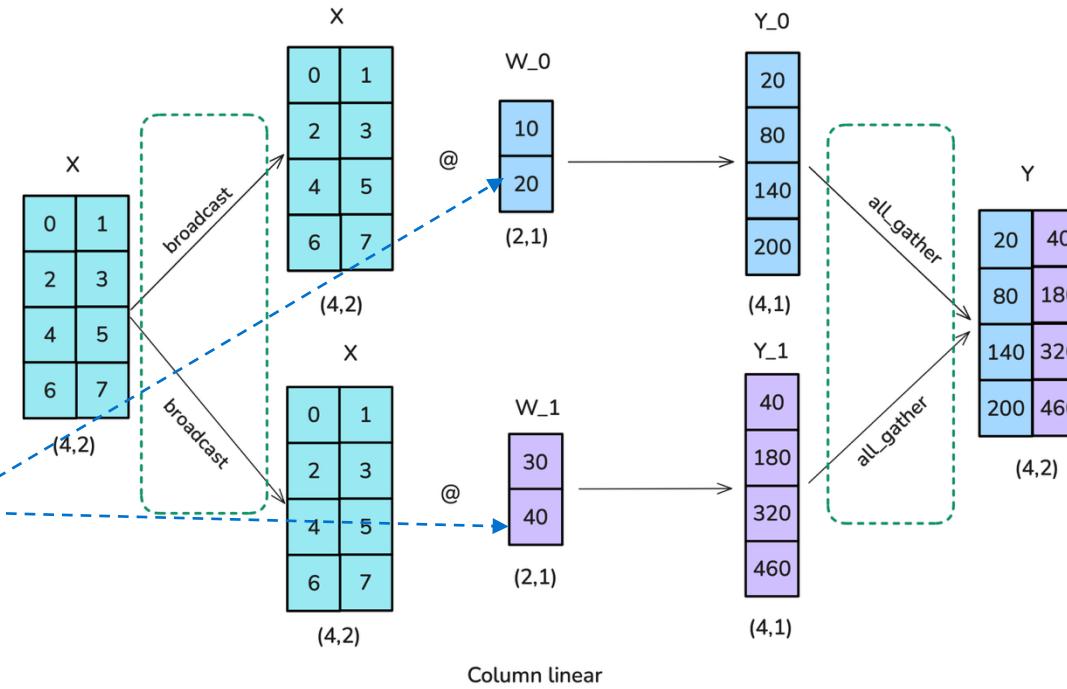
Tensor Parallelism

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

or into two rows
(Row Parallelism)

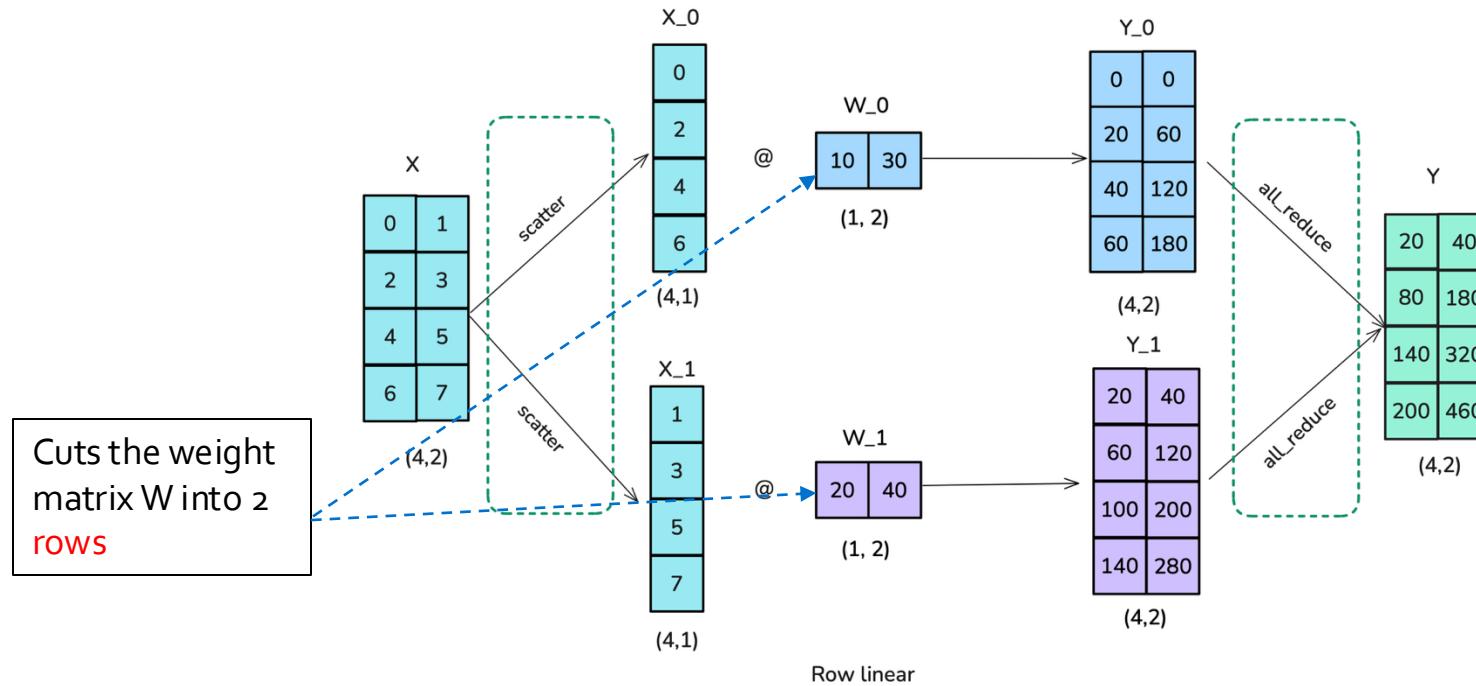


Column-wise Tensor Parallelism



Source: <https://nanotron-ultrascaling-playbook.static.hf.space/dist/index.html>

Row-wise Tensor Parallelism



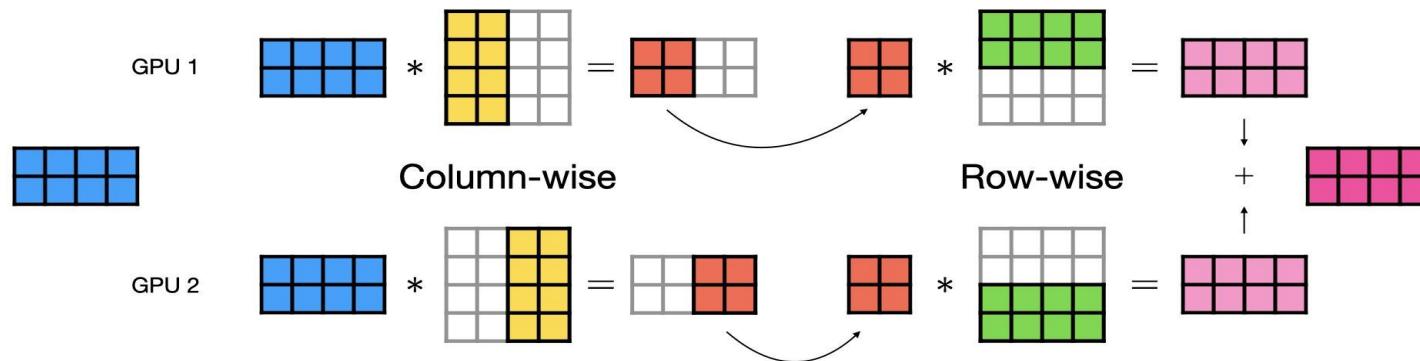
Source: <https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html>

Tensor Parallelism

Computing matrix multiplications without storing internal activations (e.g. xW_1)

$$X \quad \quad W_1 \quad \quad W_2 \quad \quad Y$$
$$\begin{matrix} \text{blue grid} \\ * \end{matrix} \quad \quad \begin{matrix} \text{yellow grid} \\ * \end{matrix} \quad \quad \begin{matrix} \text{green grid} \\ = \end{matrix} \quad \quad \begin{matrix} \text{pink grid} \end{matrix}$$

In Feed-Forward Networks,
The dimension of
 W_1 is usually 4x the hidden
dimension.



Tensor Parallelism: Llama Feed-Forward

```
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,  
)
```

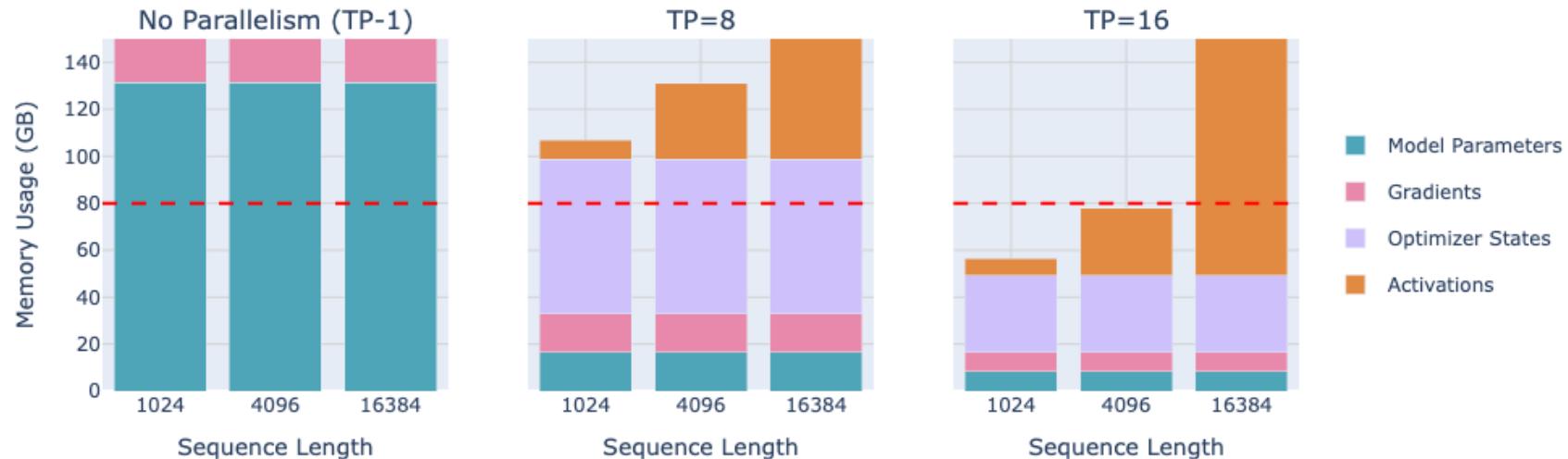
Source: <https://github.com/meta-llama/llama/blob/main/llama/model.py>

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

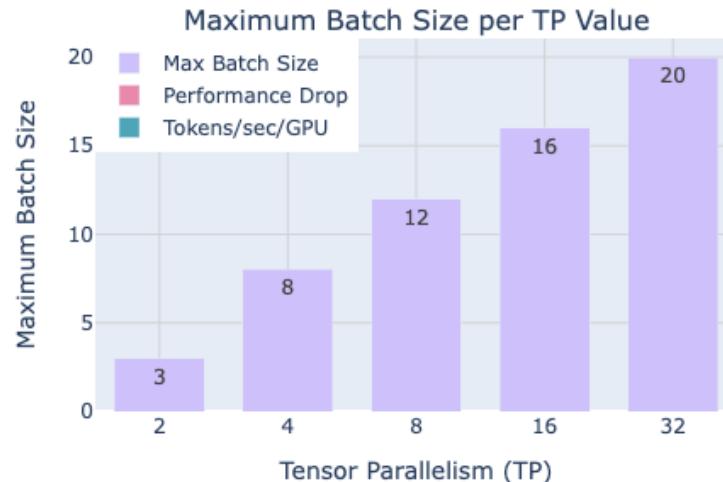
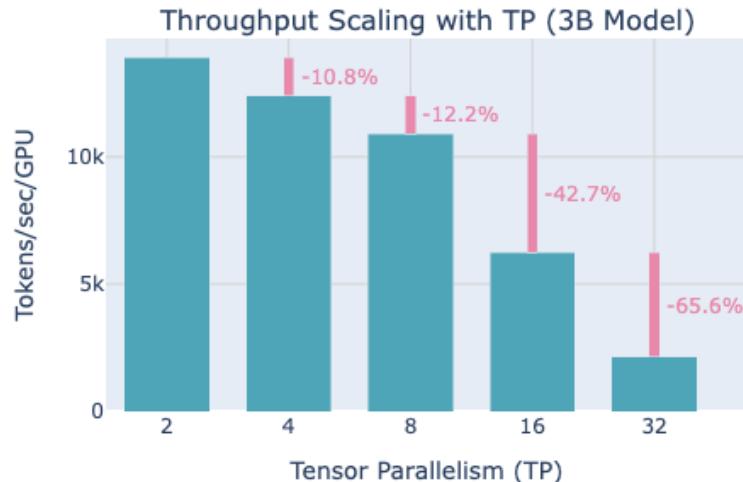
Tensor Parallelism

Memory Usage for 70B Model



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

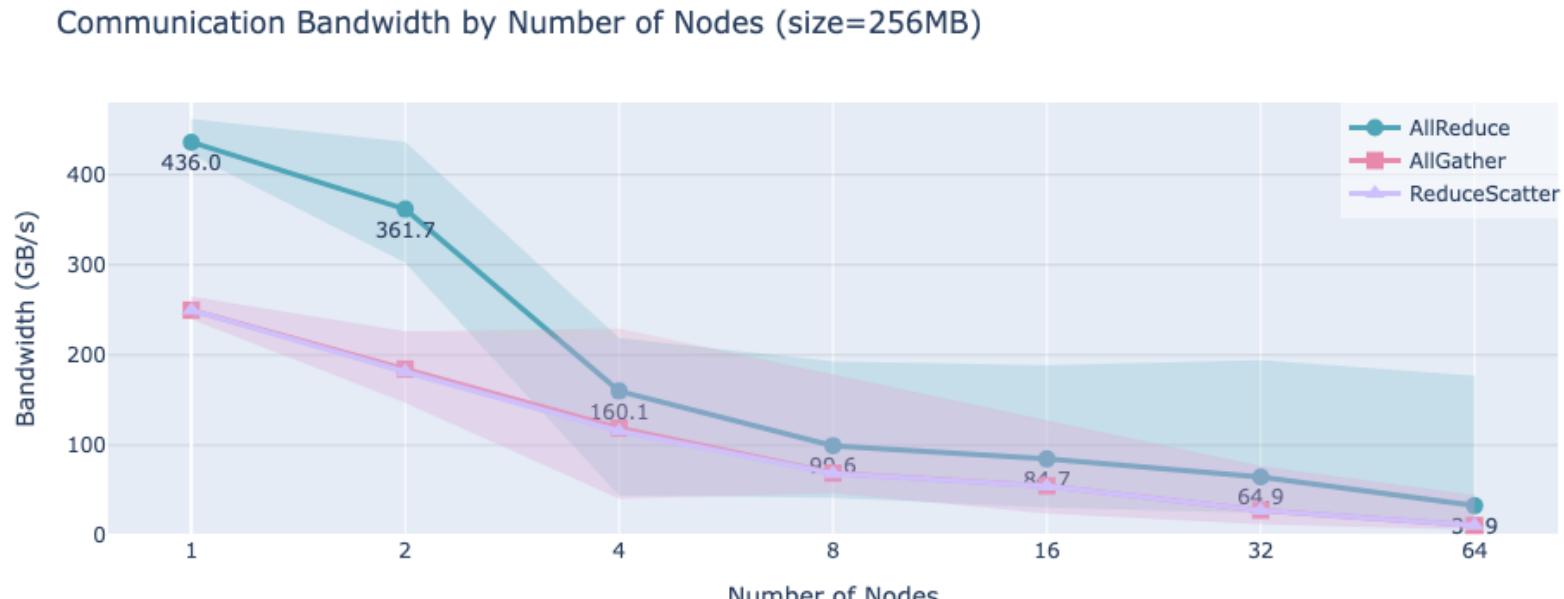
Throughput Scaling of Tensor Parallelism



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



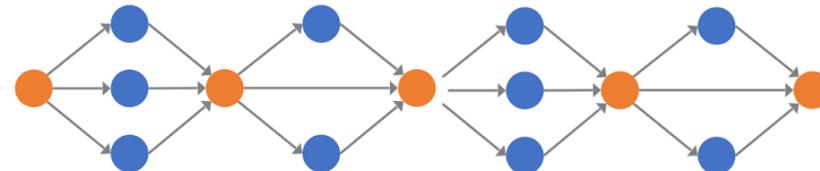
Throughput drops significantly once we go beyond one node!

Source: <https://nanotron-ultrascaling-playbook.static.hf.space/dist/index.html>

Pipeline Parallelism

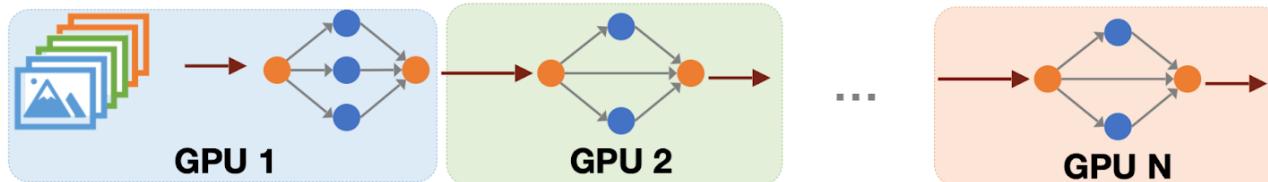


Training Dataset



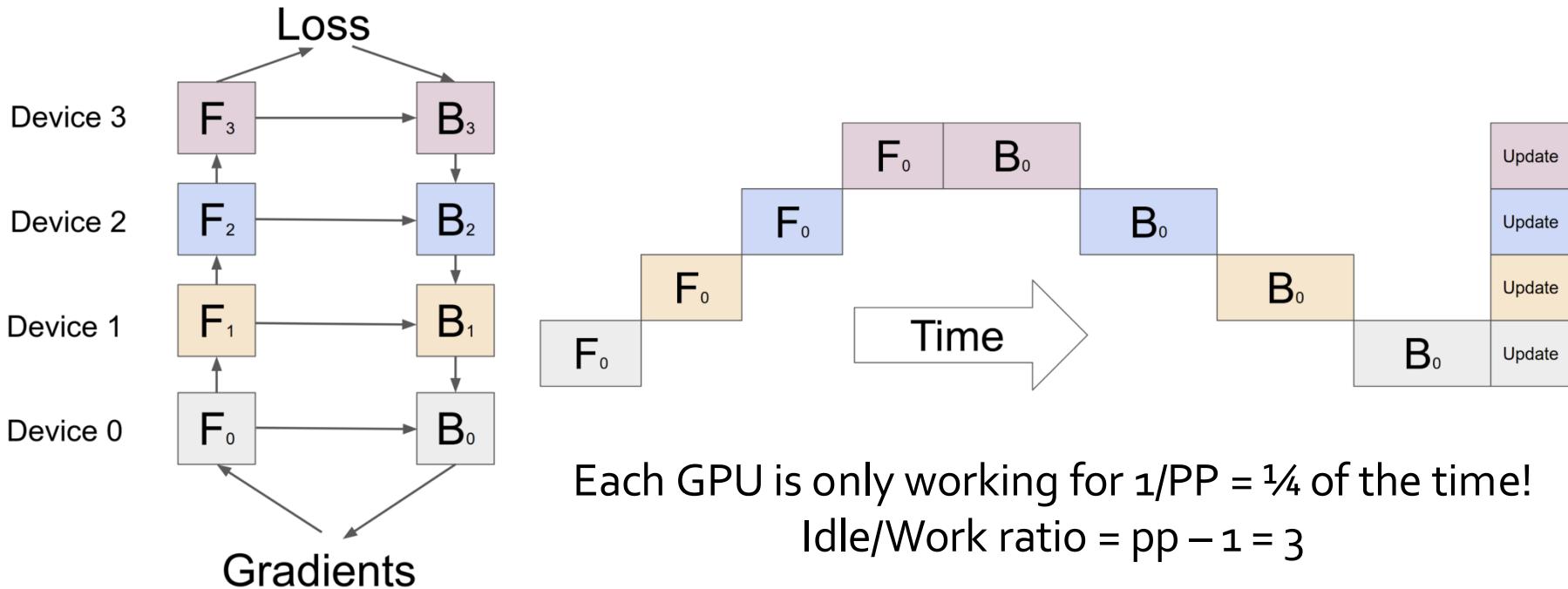
ML Model

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



Credit: Song Han (MIT)

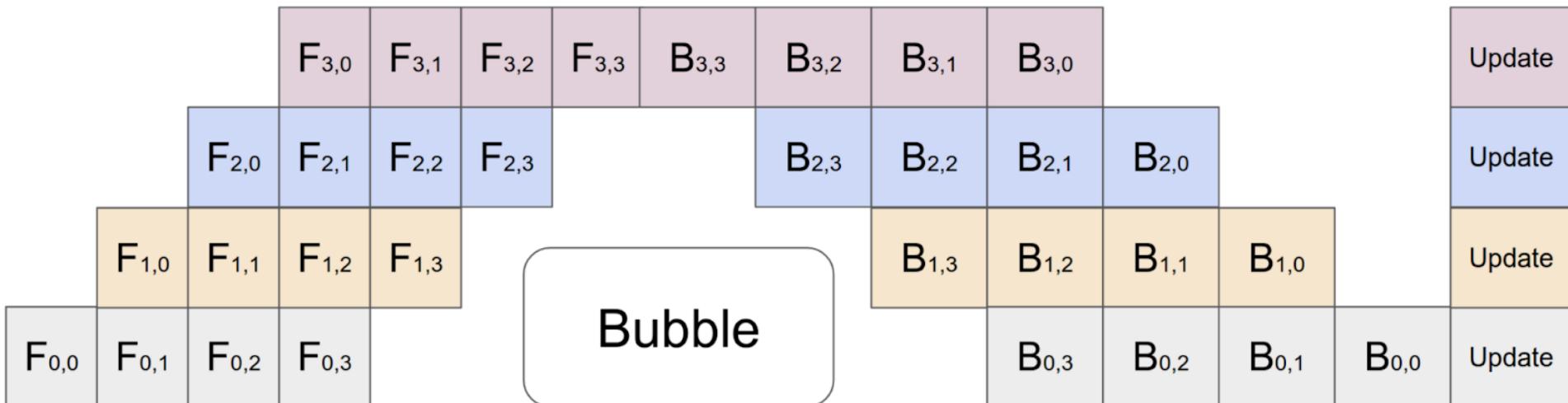
Pipeline Parallelism



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

Pipeline Parallelism: Improvement

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



$$\text{Idle / Work Ratio} = \text{PP-1} / M = 3 / 4$$

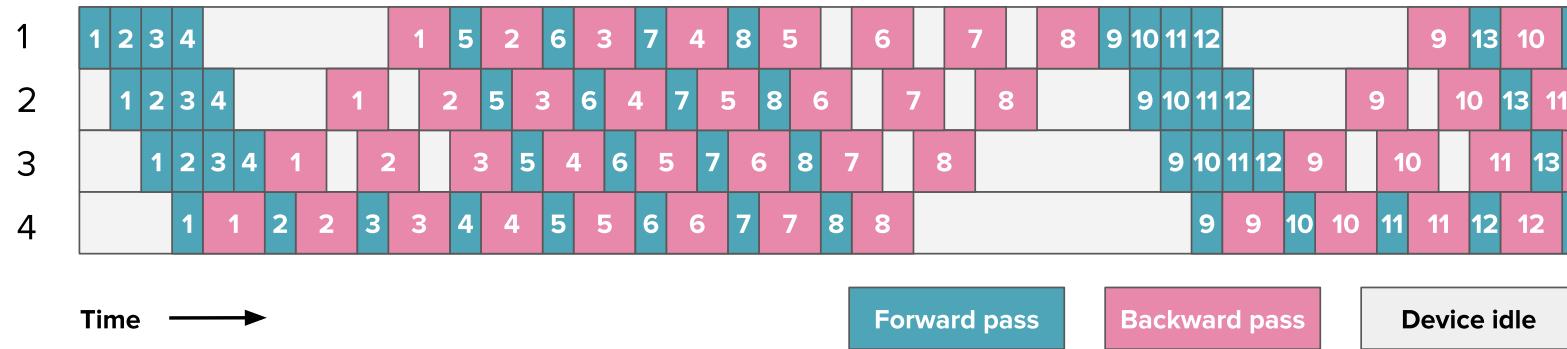
[GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism](#) (Huang et al., NeurIPS 2019)

Pipeline Parallelism

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

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

GPU

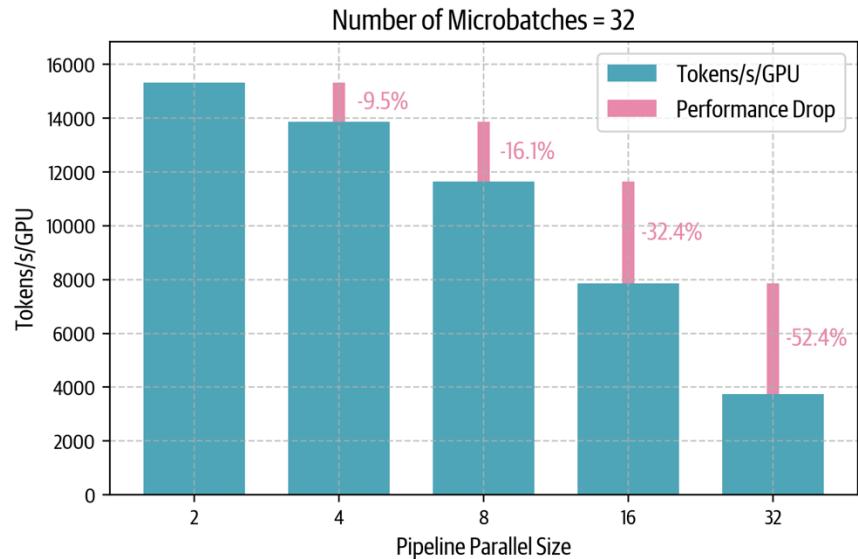
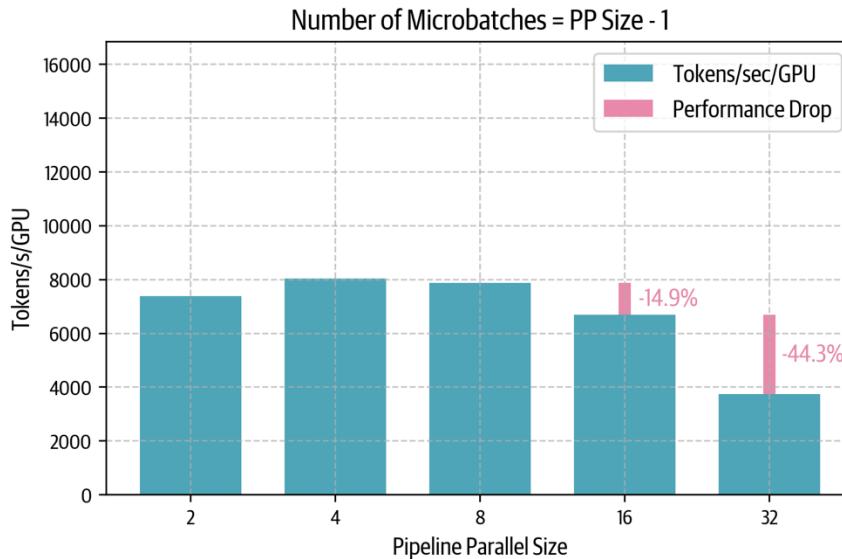


Roughly the same Idle/Work Ratio but less memory
(as you only need to store p=4 activations rather than m=8)

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

Pipeline Parallelism Throughput

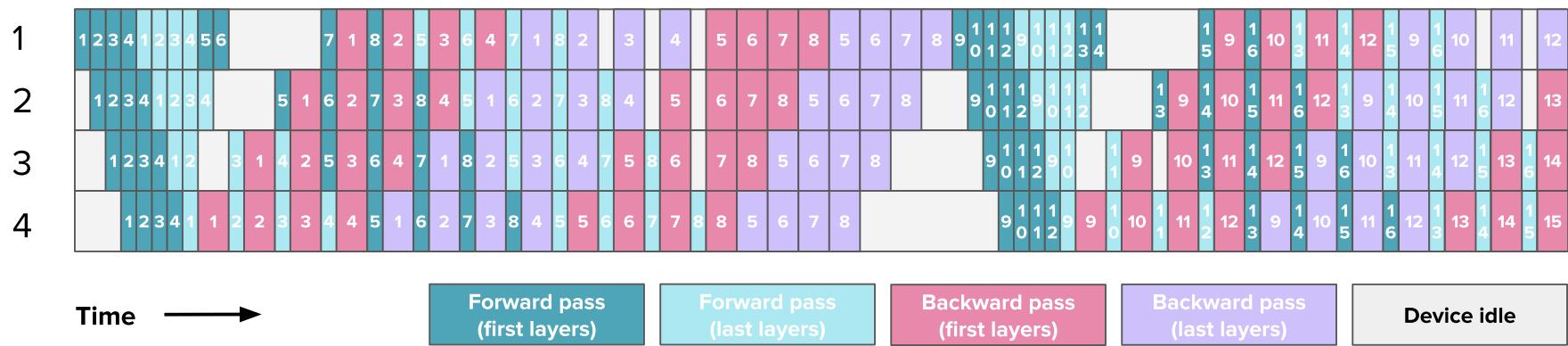
Throughput Scaling with Pipeline Parallelism (1F1B schedule)



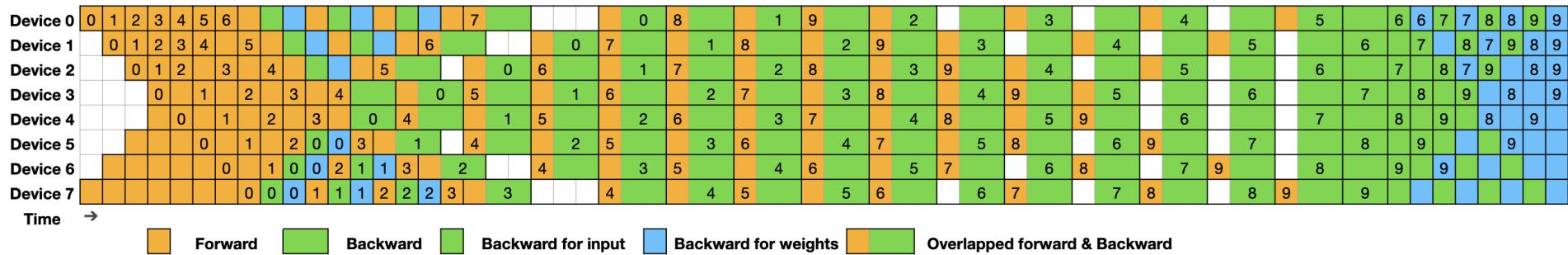
A **small** drop in throughput when scaling beyond 8 GPUs (one node)
but a large drop as we increase the microbatch number

Interleaving Pipeline Parallelism (LLama3)

GPU



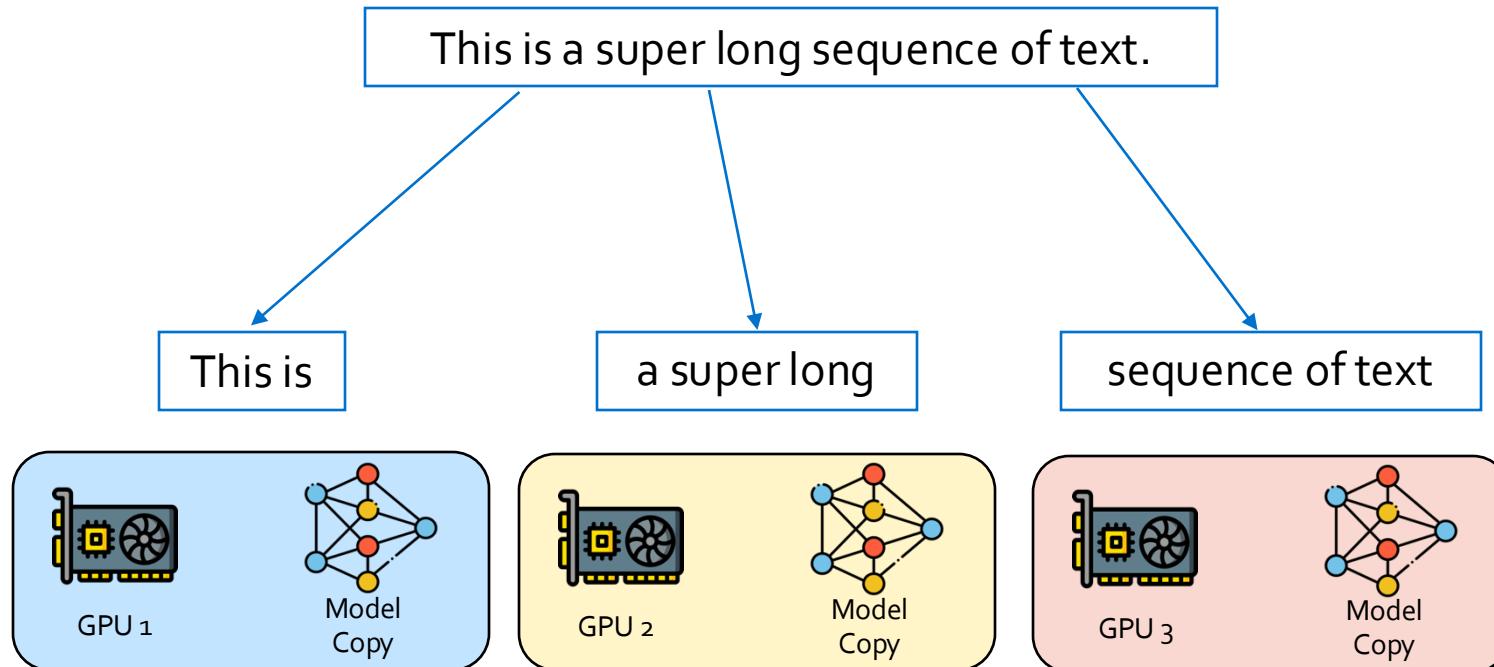
Interleaved Pipeline Parallelism (DeepSeek)



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

What about (super) long sequences?

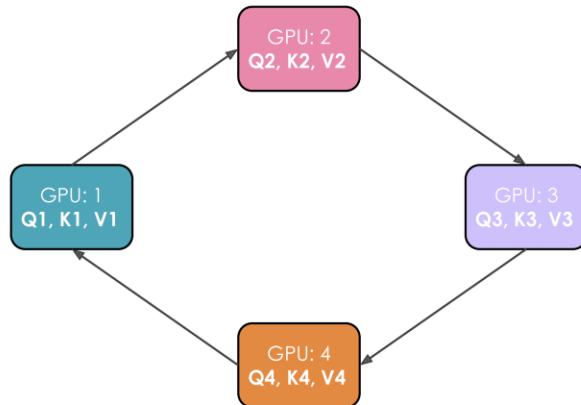
Suppose we want to split the sequence into different GPUs



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)

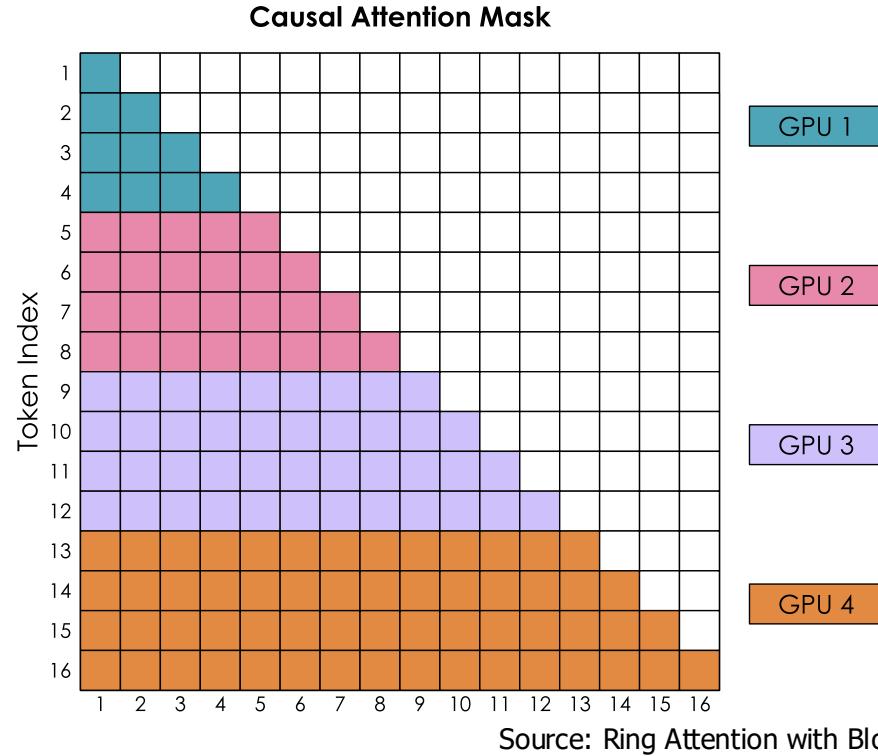


Just pass the Key, Value pairs around!

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

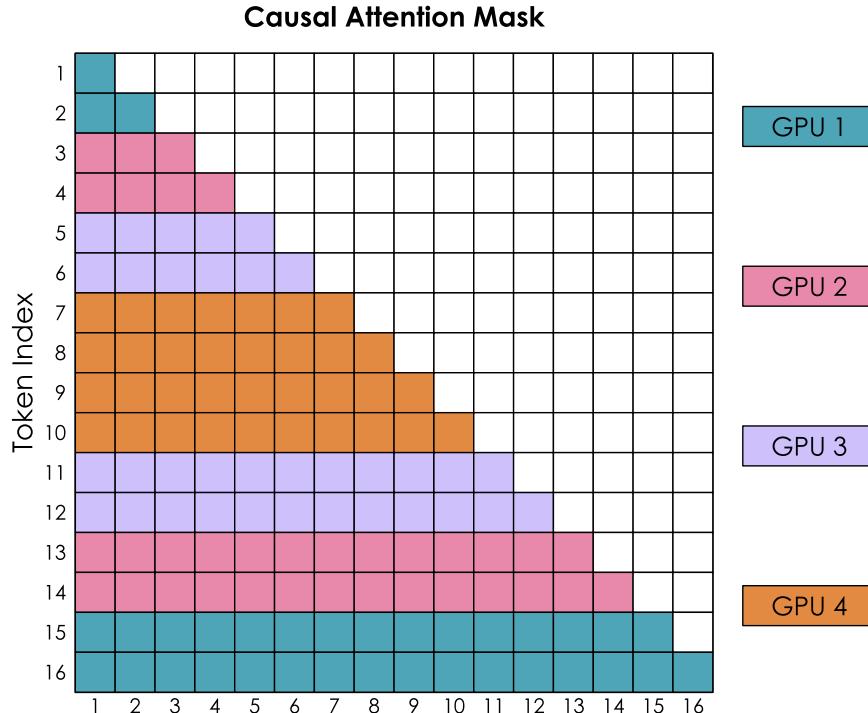
Source: Ring Attention with Blockwise Transformers For Near-Infinite Context (Liu et al., 2023)

Context Parallelism (Ring Attention)



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

Context Parallelism (Ring Attention)

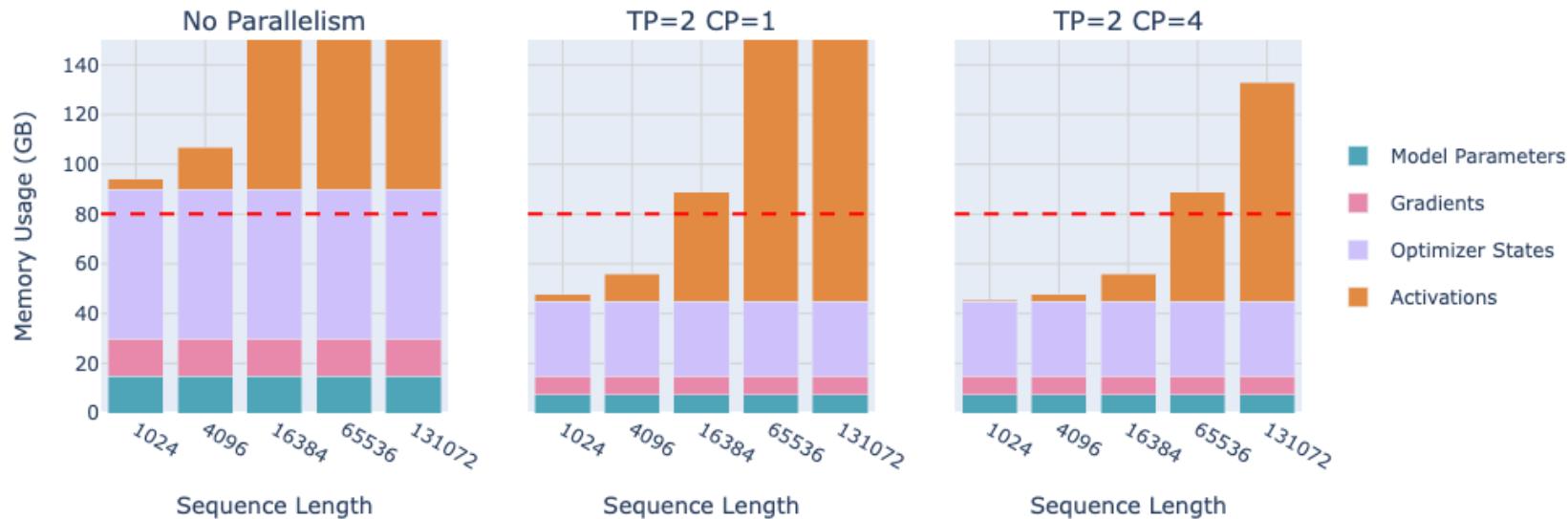


Balancing the workload
for each individual GPUs.

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

Context Parallelism (Ring Attention)

Memory Usage for 8B Model



Source: <https://nанотрон-ultrascale-playbook.static.hf.space/dist/index.html>

Summarizing

	Sync overhead	Memory	Bandwidth	Batch size	Easy to use?
DDP/ZeRO1	Per-batch	No scaling	$2^* \# \text{ param}$	Linear	Very
FSDP (ZeRO3)	3x Per-FSDP block	Linear	$3 * \# \text{ param}$	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*8, EP (Expert Parallelism) = 64

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

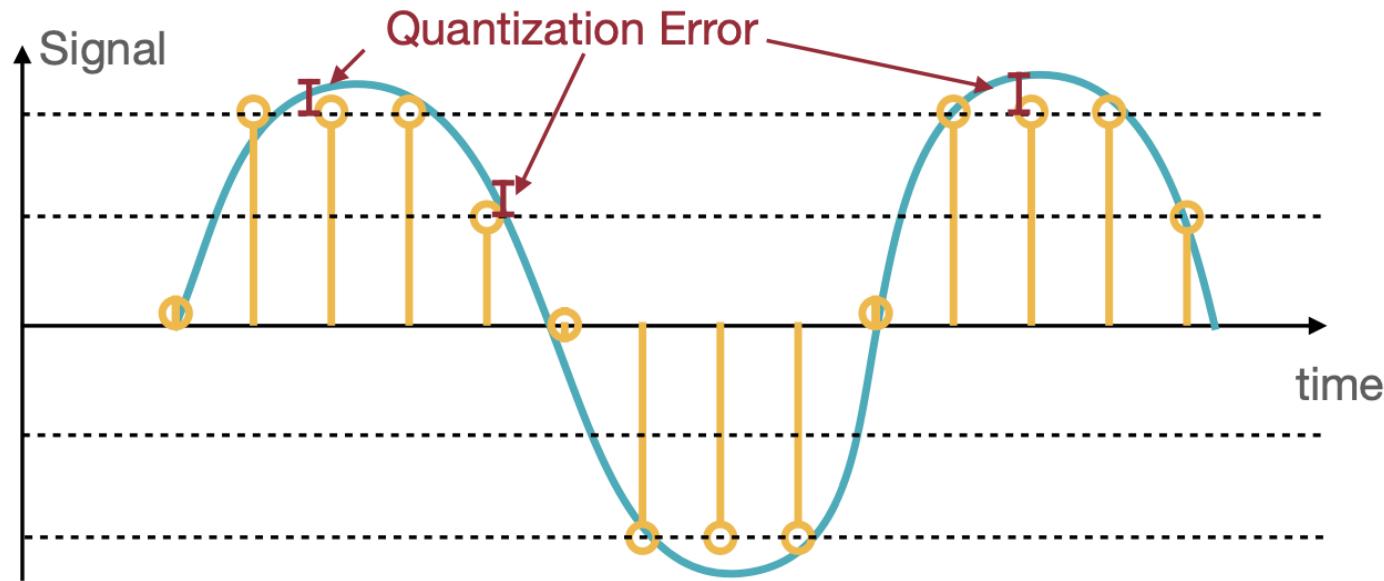
GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	8	131,072	16	16M	380	38%

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

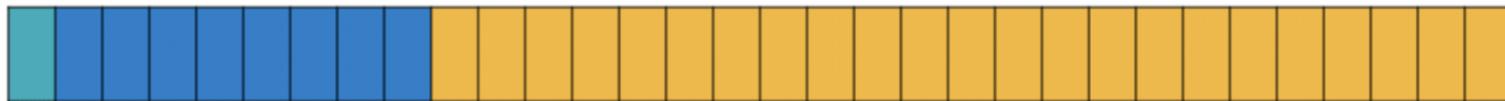
Quantization: Mapping from high to low precision

— Continuous Signal —○ Quantized Signal



Numeric Data Types

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



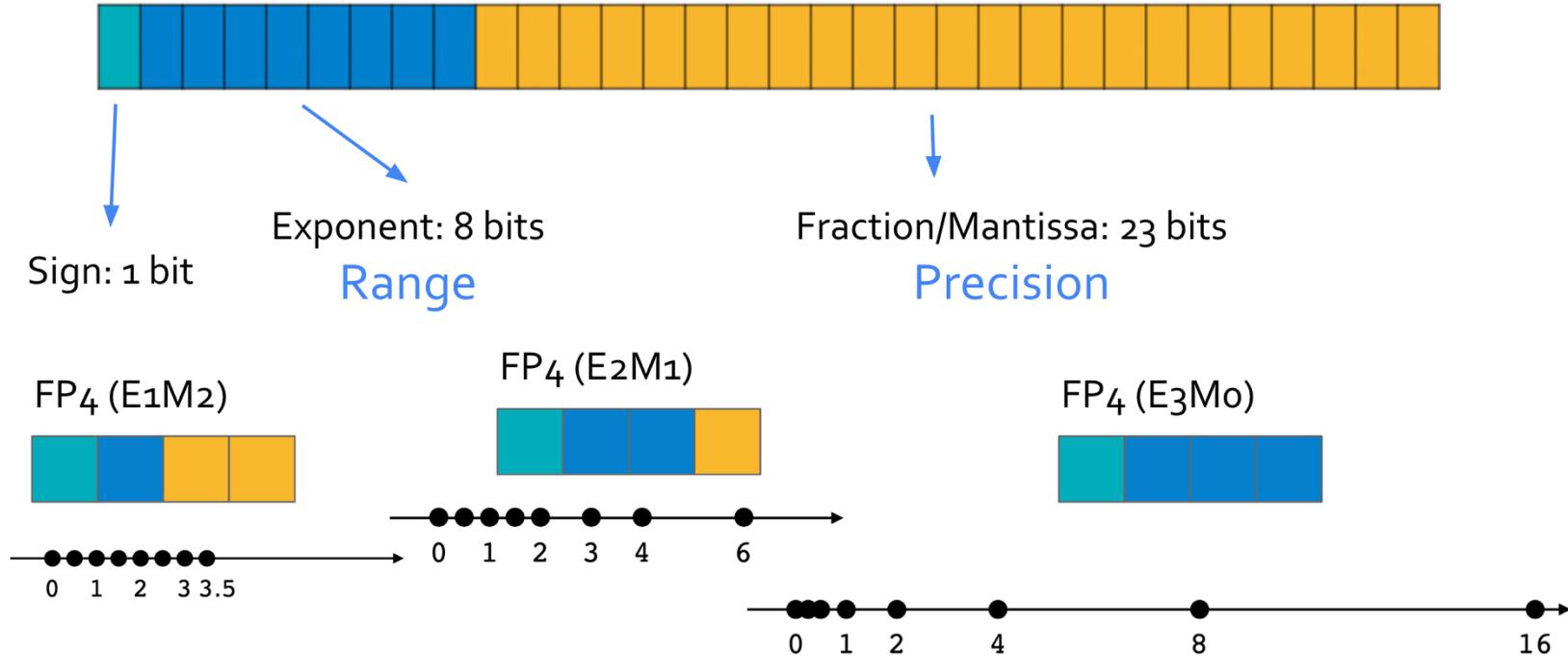
Sign: 1 bit

Exponent: 8 bits

Fraction/Mantissa: 23 bits

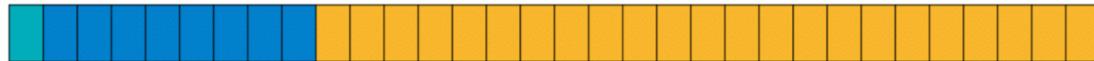
$$\text{Number} = (-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent} - 127}$$

Floating Point Numbers



Floating Point Numbers

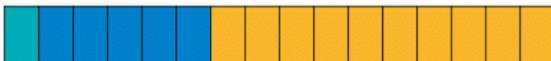
IEEE 754 Single Precision 32-bit Float (FP32)



Exponent Fraction

8 23

IEEE 754 Half Precision 16-bit Float (FP16)



5 10

Google Brain Float (BF 16)



More range, less precision

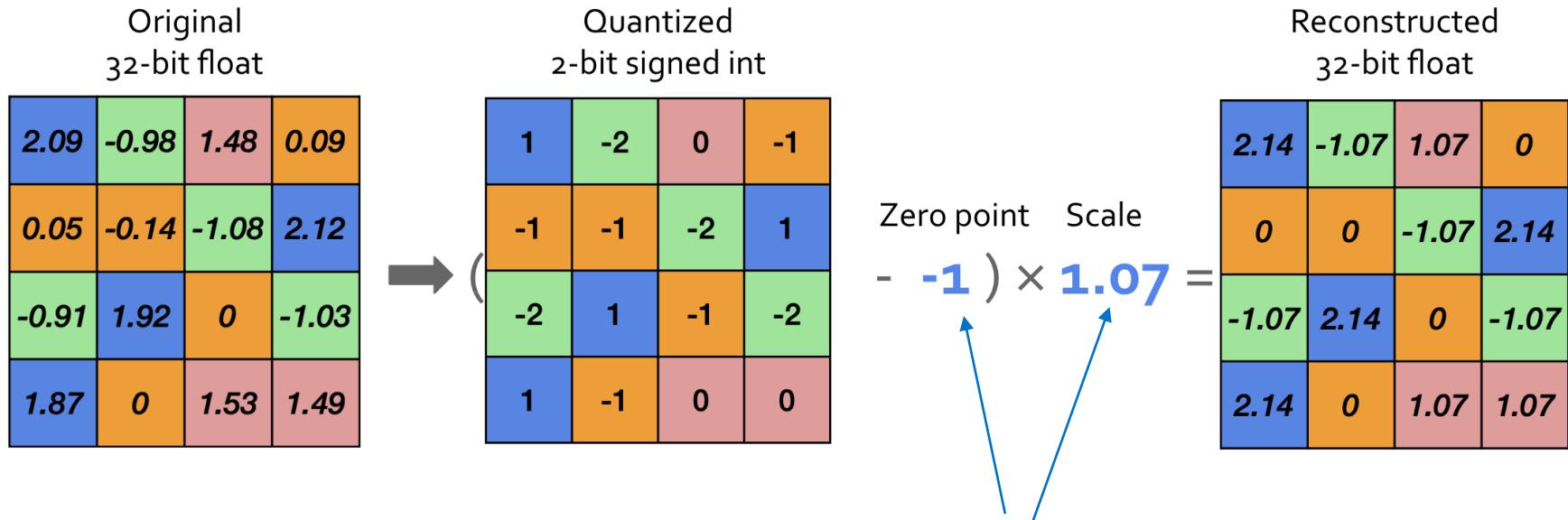
8 7

Nvidia FP8 (E4M3)



4 3

Linear Quantization



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

r

≈

Quantized 2-bit signed int			
1	-2	0	-1
-1	-1	-2	1
-2	1	-1	-2
1	-1	0	0

integer

$$\text{Zero point } - \text{-1} \times 1.07 =$$

$$Z \times S$$

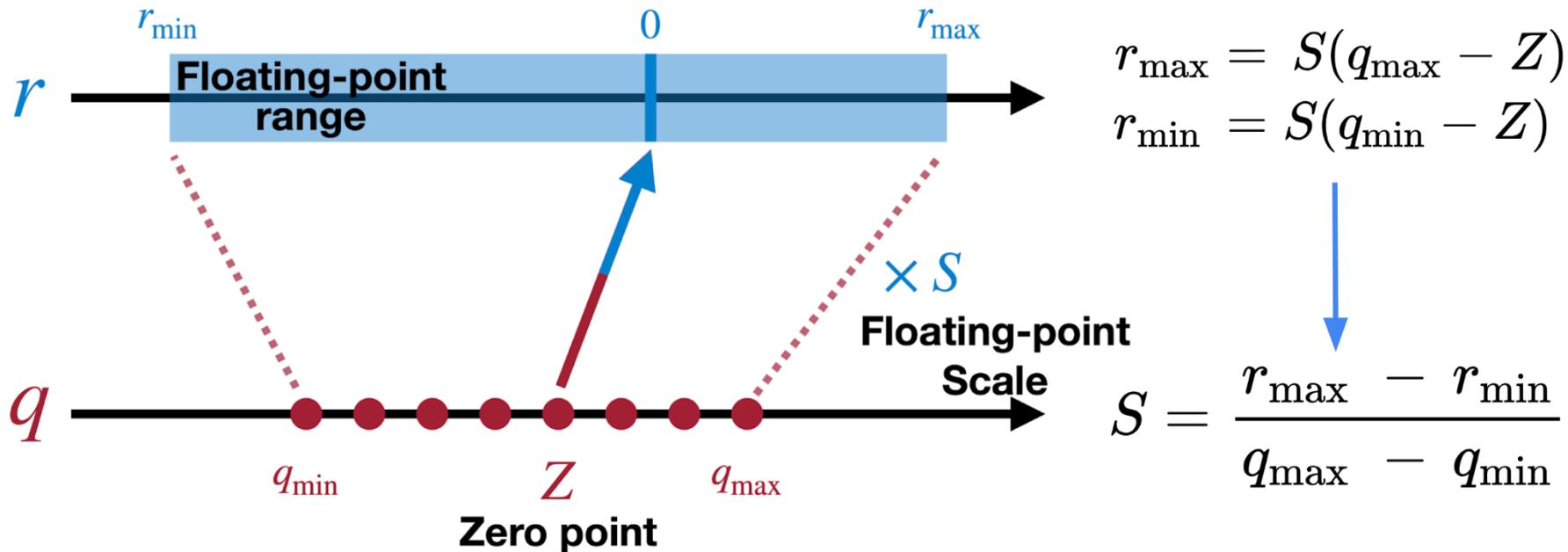
floating-point

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

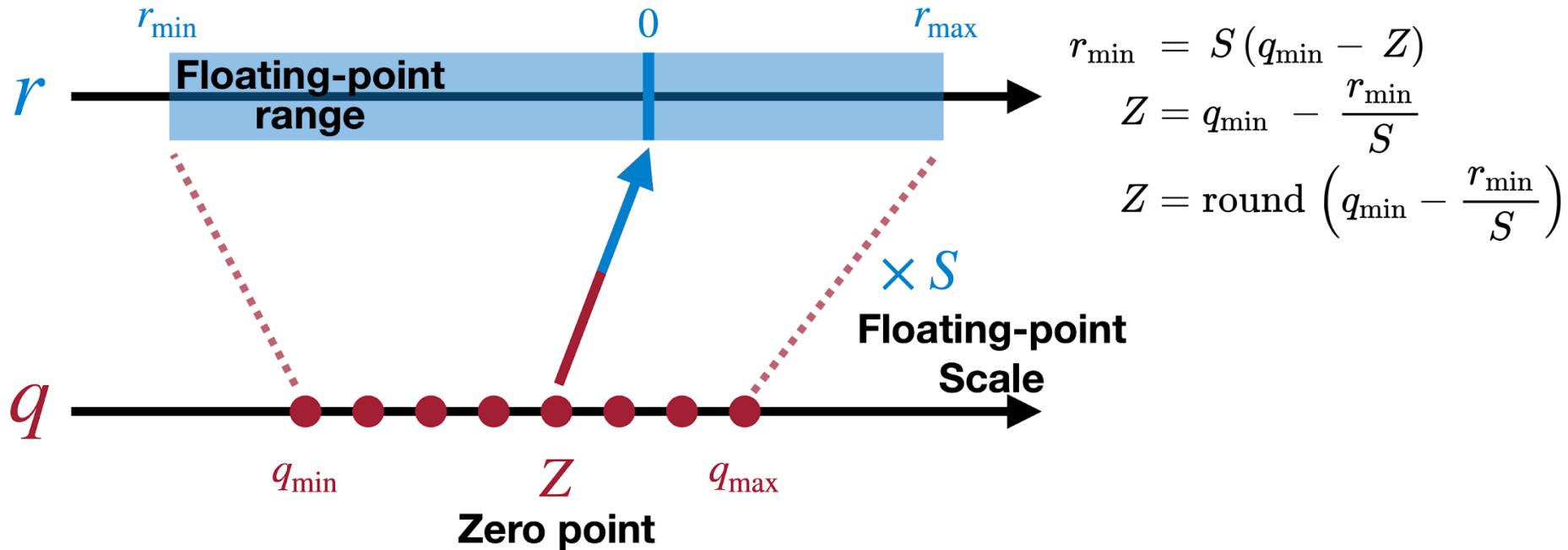
integer floating-point

Linear Quantization: Scale



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

Linear Quantization: Zero Point



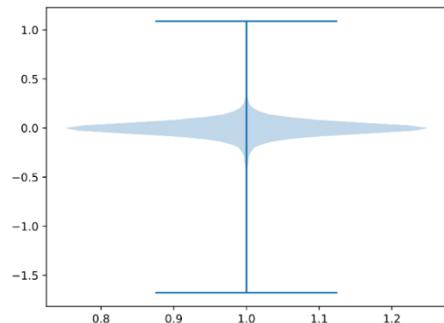
[Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference](#) (Jacob et al., CVPR 2018)

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.



Weight distribution of first conv
layer of ResNet-50.

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

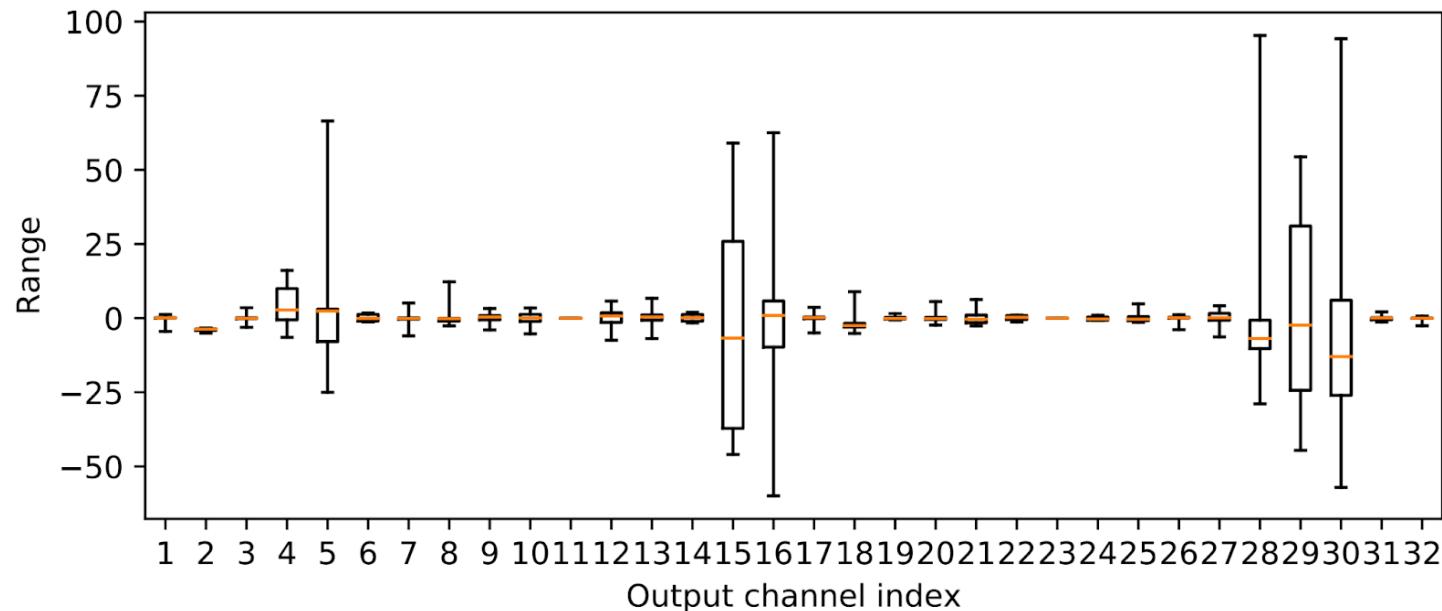


$$S = \frac{r_{\min}}{q_{\min} - Z} = \frac{-|r|_{\max}}{q_{\min}}$$

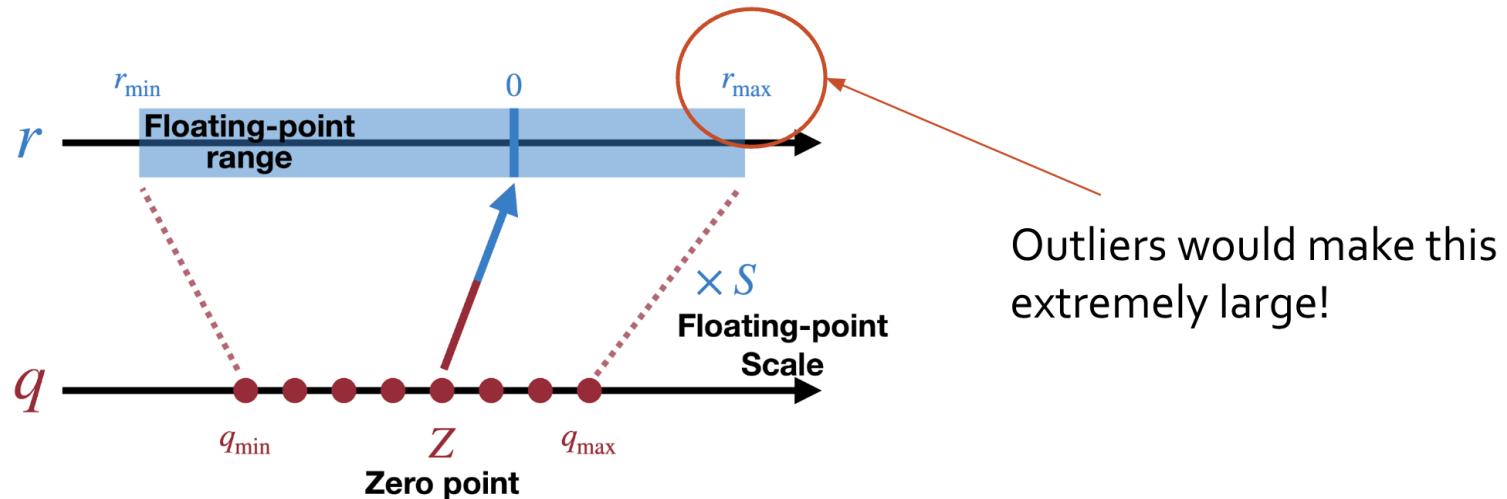
Used in Pytorch, ONNX

Quantization of Language Models

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

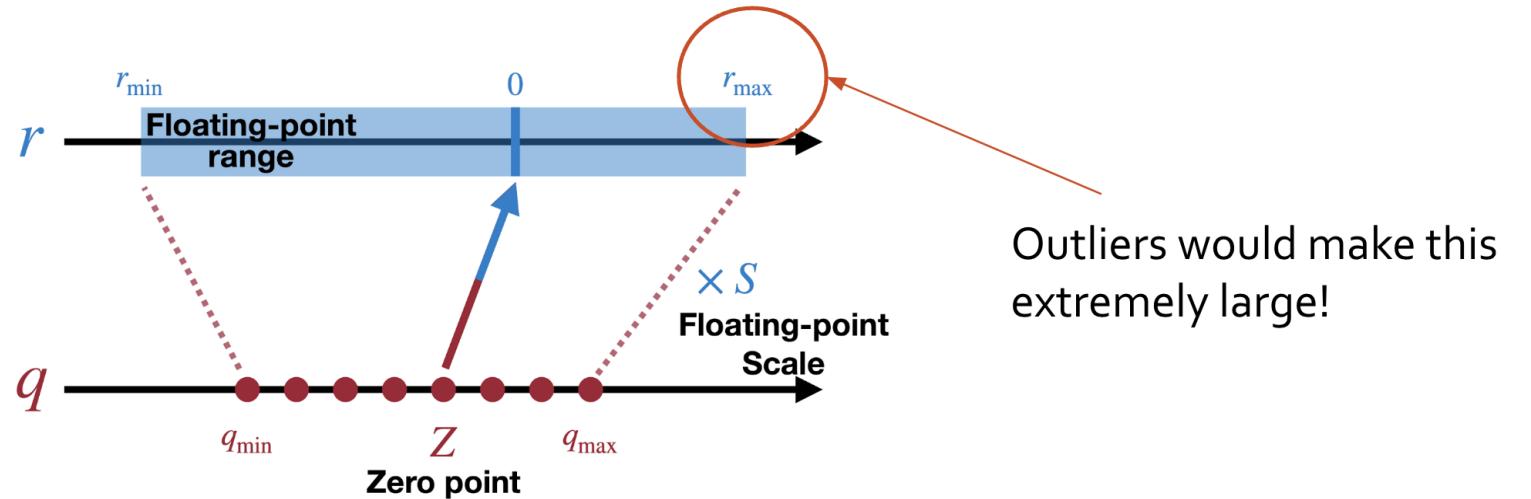


Quantization of Language Models



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

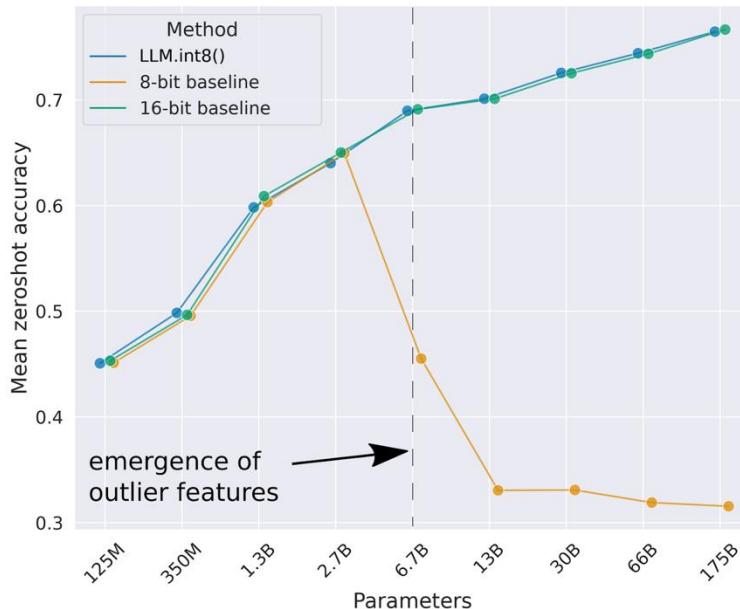


Example: 15, 0.1, 0.02, 1.0, 0.01 -> 127, 1, 0, 8, 0
(Everything under 0.05 gets mapped to 0)

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

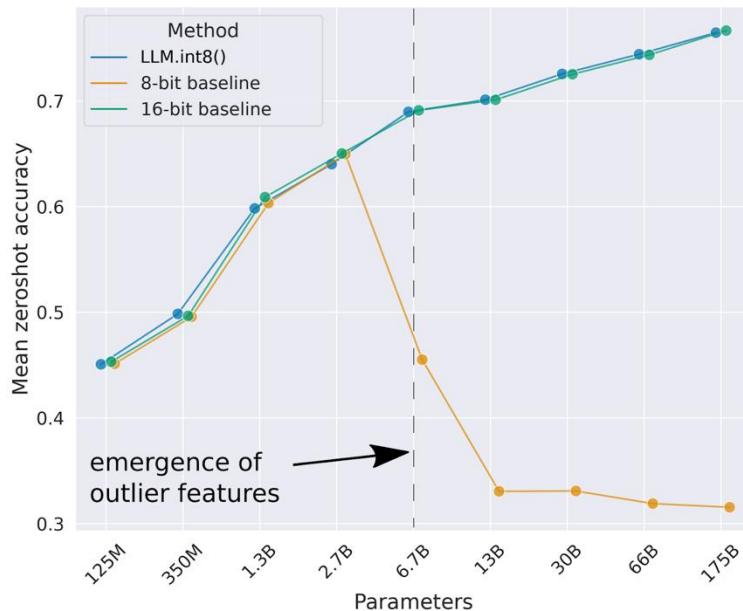
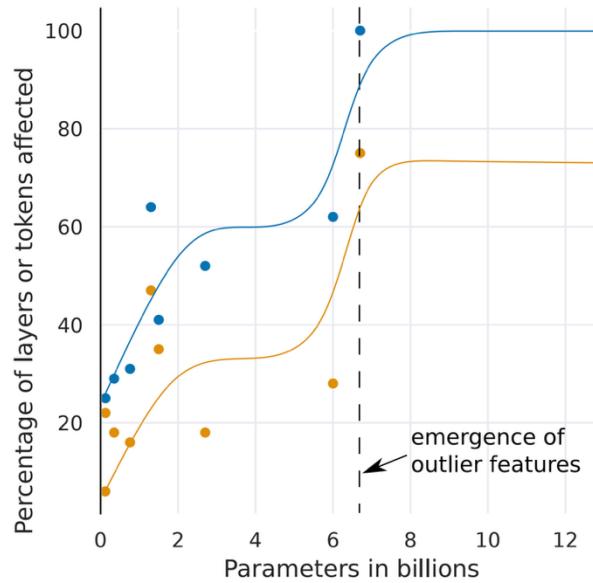
Quantization of Language Models

Outlier features significantly harms performance after quantization in LMs.



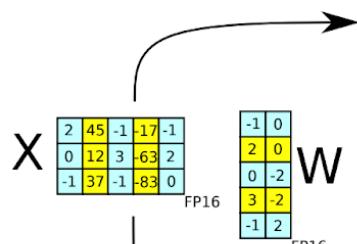
Quantization of Language Models

Outlier features significantly harms performance after quantization in LMs.



Quantization of Language Models

LLM.int8()



■ Regular values
■ Outliers

8-bit Vector-wise Quantization

(1) Find vector-wise constants: C_W & C_X

$$\begin{array}{c} X \\ \uparrow C_X \\ \begin{bmatrix} 2 & 2 & -1 & -1 \\ 3 & 0 & 3 & 2 \\ 1 & -1 & -1 & 0 \end{bmatrix}_{\text{FP16}} \end{array} \quad \begin{array}{c} W \\ \leftarrow C_W \\ \begin{bmatrix} 1 & 2 \\ -1 & 0 \\ 0 & -2 \\ -1 & 2 \end{bmatrix}_{\text{FP16}} \end{array}$$

(2) Quantize

$$X_{\text{FP16}} * (127/C_X) = X_{\text{I8}}$$
$$W_{\text{FP16}} * (127/C_W) = W_{\text{I8}}$$

(3) Int8 Matmul

$$X_{\text{I8}} W_{\text{I8}} = \text{Out}_{\text{I32}}$$

(4) Dequantize

$$\frac{\text{Out}_{\text{I32}} * (C_X \otimes C_W)}{127 * 127} = \text{Out}_{\text{FP16}}$$

16-bit Decomposition

(1) Decompose outliers

$$X_{\text{FP16}} = \begin{bmatrix} 45 & -17 \\ 12 & -63 \\ 37 & -83 \end{bmatrix}_{\text{FP16}}$$

(2) FP16 Matmul

$$X_{\text{FP16}} W_{\text{FP16}} = \text{Out}_{\text{FP16}}$$

$$\text{Out}_{\text{FP16}} = \text{Out}_{\text{I32}} + \text{Out}_{\text{FP16}}$$

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

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

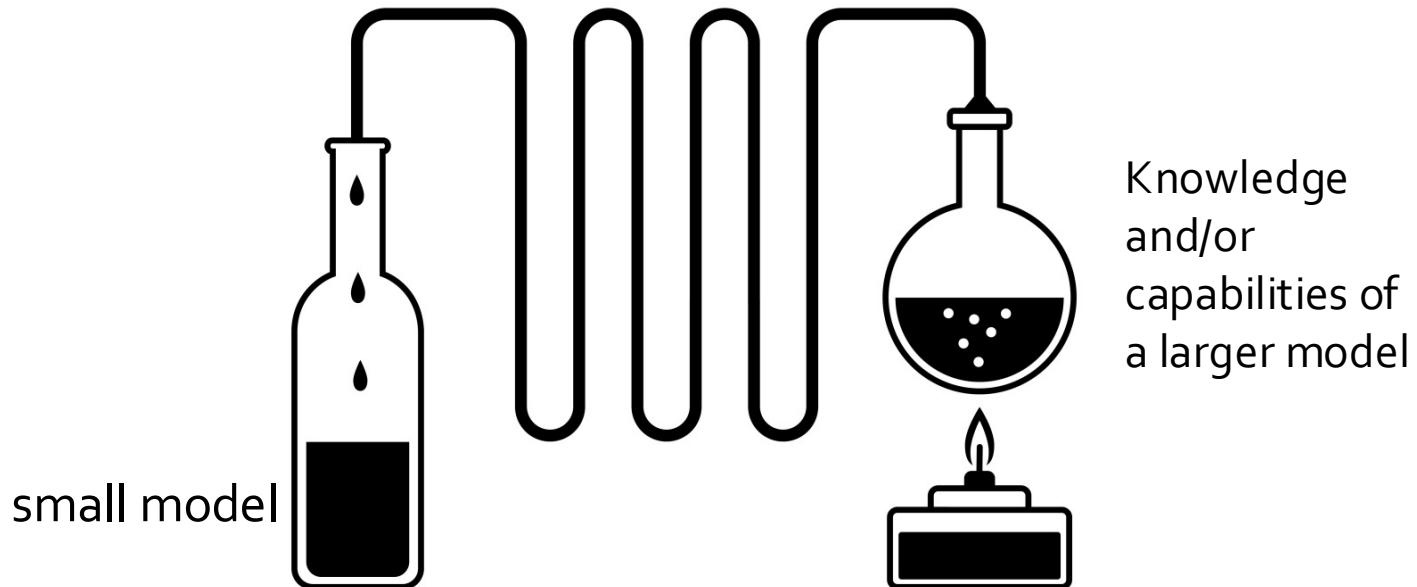
Zeropoint > absmax because outliers non-symmetric (either very large or very small, but not both)

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

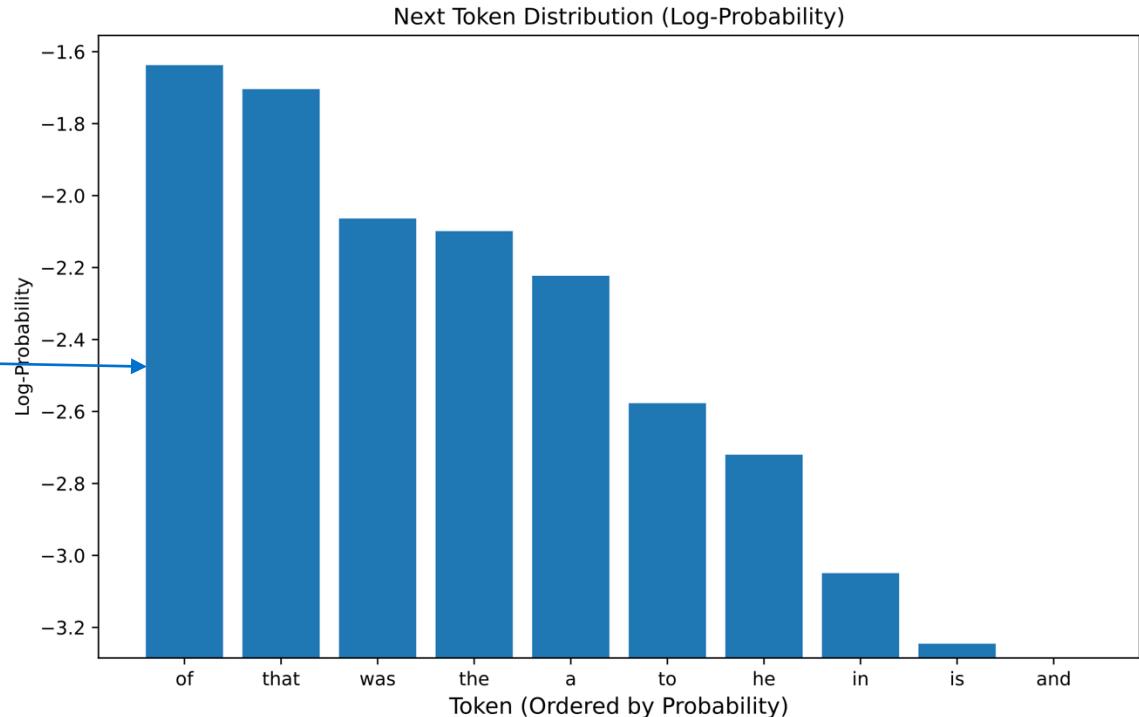
Distillation



Revisit: Standard Training (NLLoss)

prefix: The strange case __
groundtruth: of

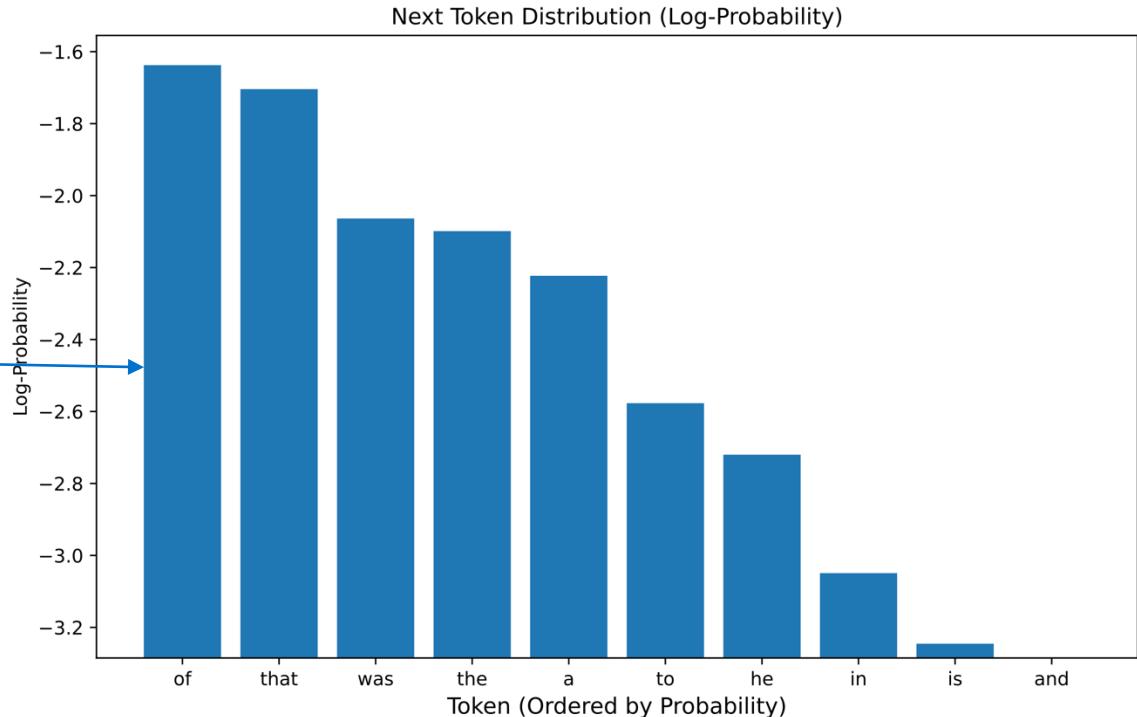
Loss = $-\log p(\text{of})$
= Cross Entropy(y_{pred} ,
groundtruth)



Revisit: Standard Training (NLLloss)

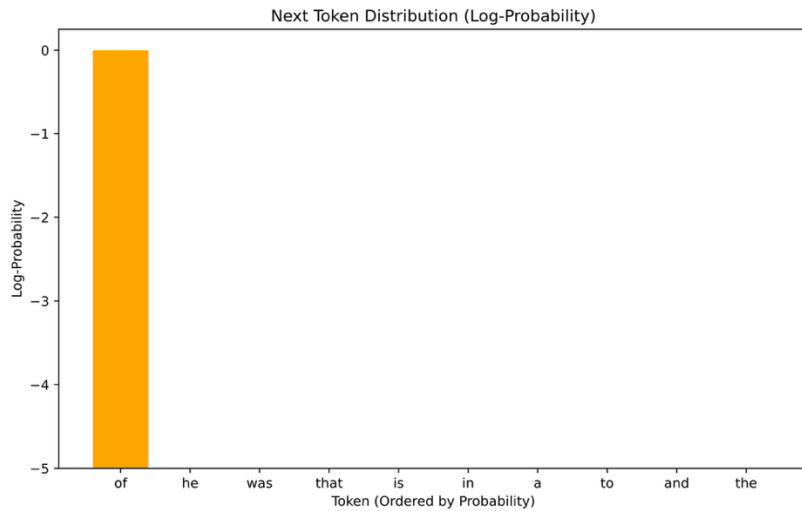
prefix: The strange case __
groundtruth: of

$$\text{loss} = -\log p(\text{of})$$

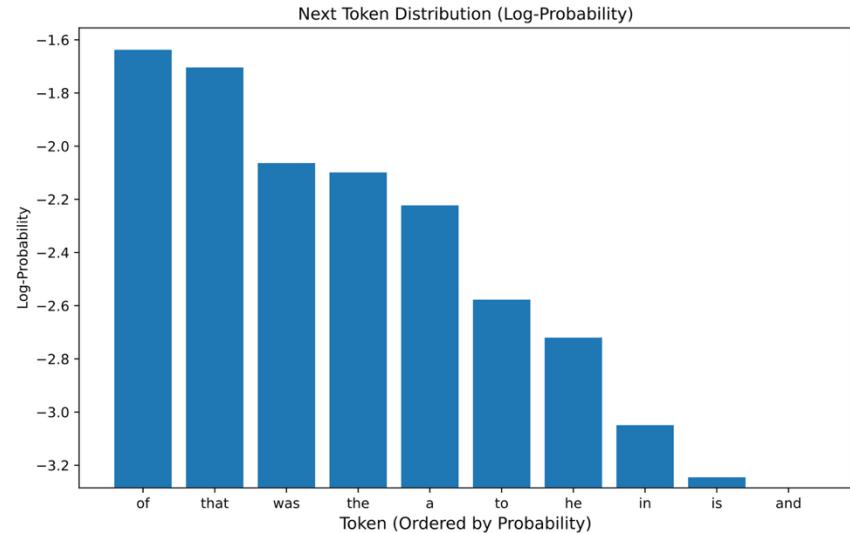


Revisit: Standard Training (NLLloss)

$$\text{loss} = -\log p(\text{of}) = \text{Cross Entropy}(\text{groundtruth}, \text{y_pred})$$



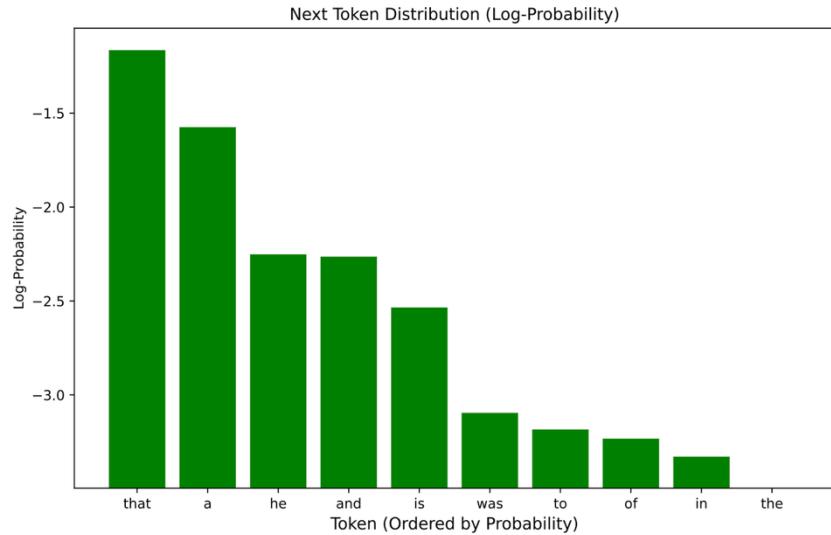
Groundtruth
(one-hot)



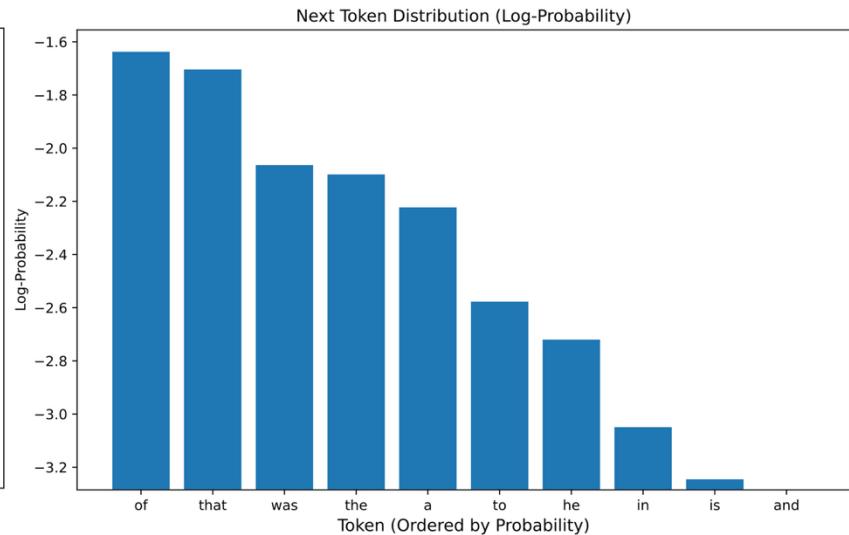
y_pred

Knowledge Distillation

$$\text{KD loss} = \text{Cross Entropy}(y_{\text{large}}, y_{\text{pred}})$$



Large model next token
probs (y_{large})

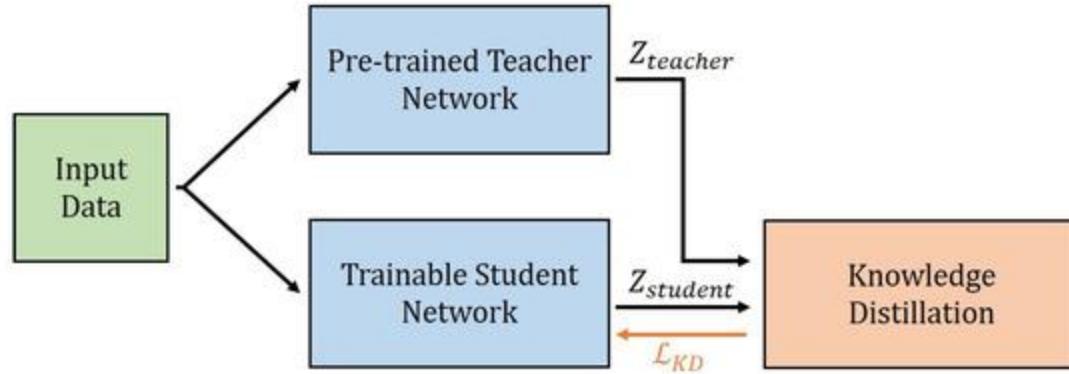


small model next token probs
(y_{pred})

Knowledge Distillation

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

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

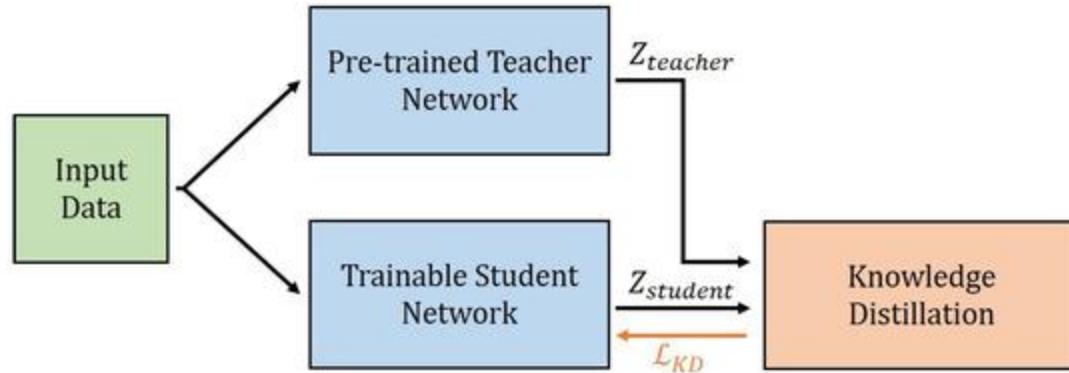


Step 3: Use teacher outputs to train student (Cross Entropy)

What if the teacher is Proprietary (GPT)?

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

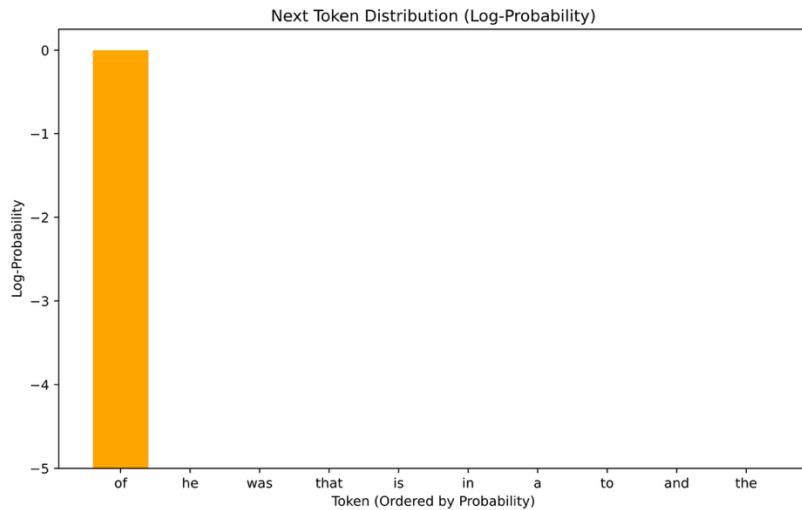
Step 2: Feed input data to both student and teacher (freezed)



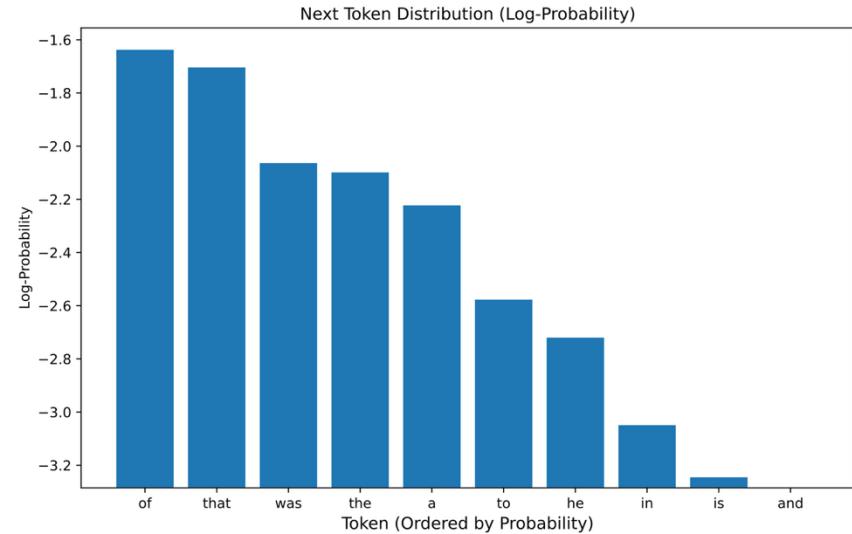
Step 3: Use teacher **generations** (instead of outputs) to train student!

Revisit: Standard Training (NLLloss)

$$\text{loss} = -\log p(\text{of}) = \text{Cross Entropy}(\text{sampled text}, \text{y_pred})$$

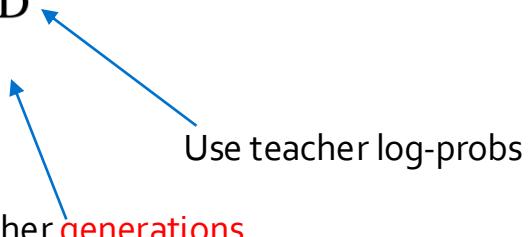


Sampled output
(one-hot)



y_pred

What works better (a study in 2016)

Model	BLEU _{K=1}	Δ _{K=1}	BLEU _{K=5}	Δ _{K=5}
<i>English → German WMT 2014</i>				
Teacher Baseline 4×1000 (Params: 221m)	17.7	—	19.5	—
Baseline + Seq-Inter	19.6	+1.9	19.8	+0.3
<hr/>				
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
				

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

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.

https://www.tensoreconomics.com/p/llm-inference-economics-from-first?utm_source=substack&utm_medium=email

VERY GOOD: Incorporate

<https://arxiv.org/pdf/2502.17129>

Here are some topics:

- KV cache optimization
- memory management
-