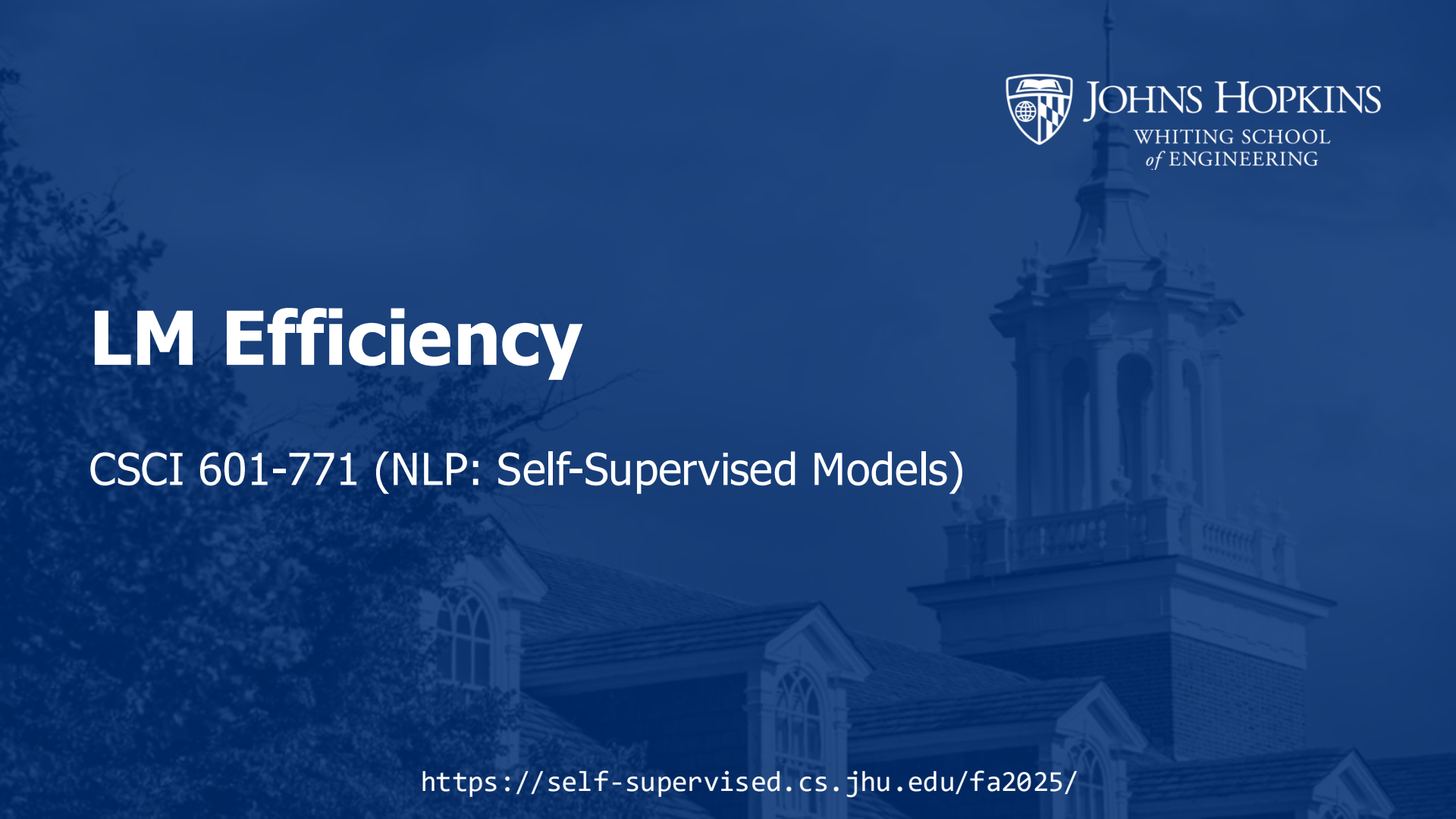


LM Efficiency

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

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

Our models are getting larger!

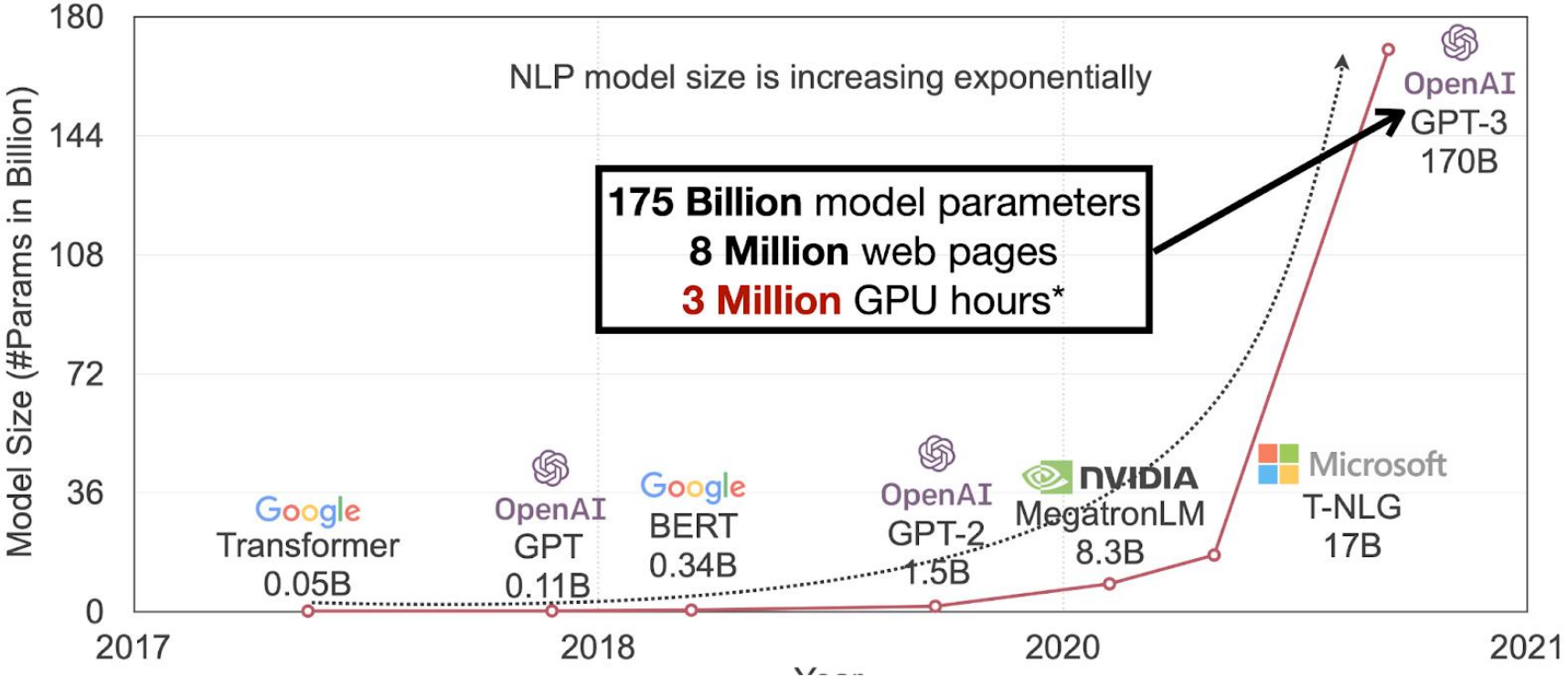
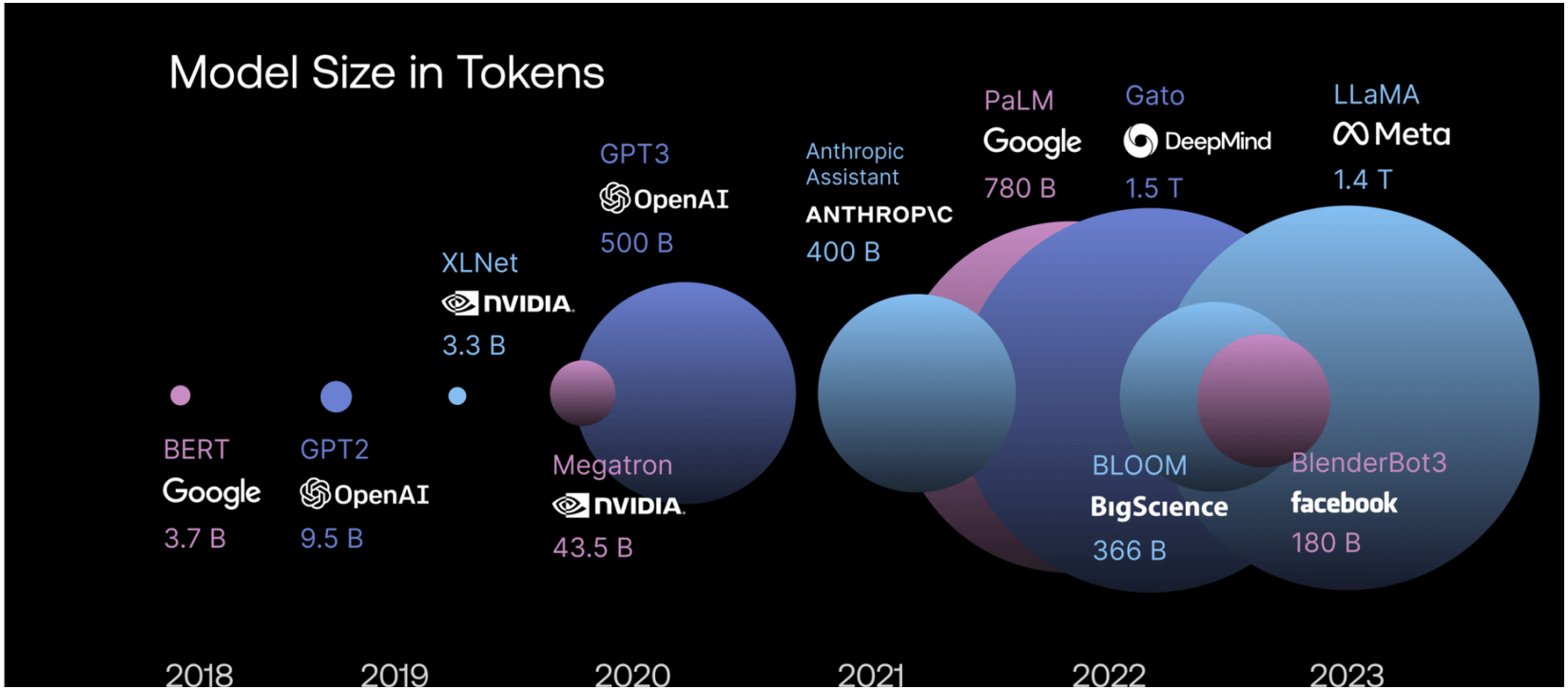
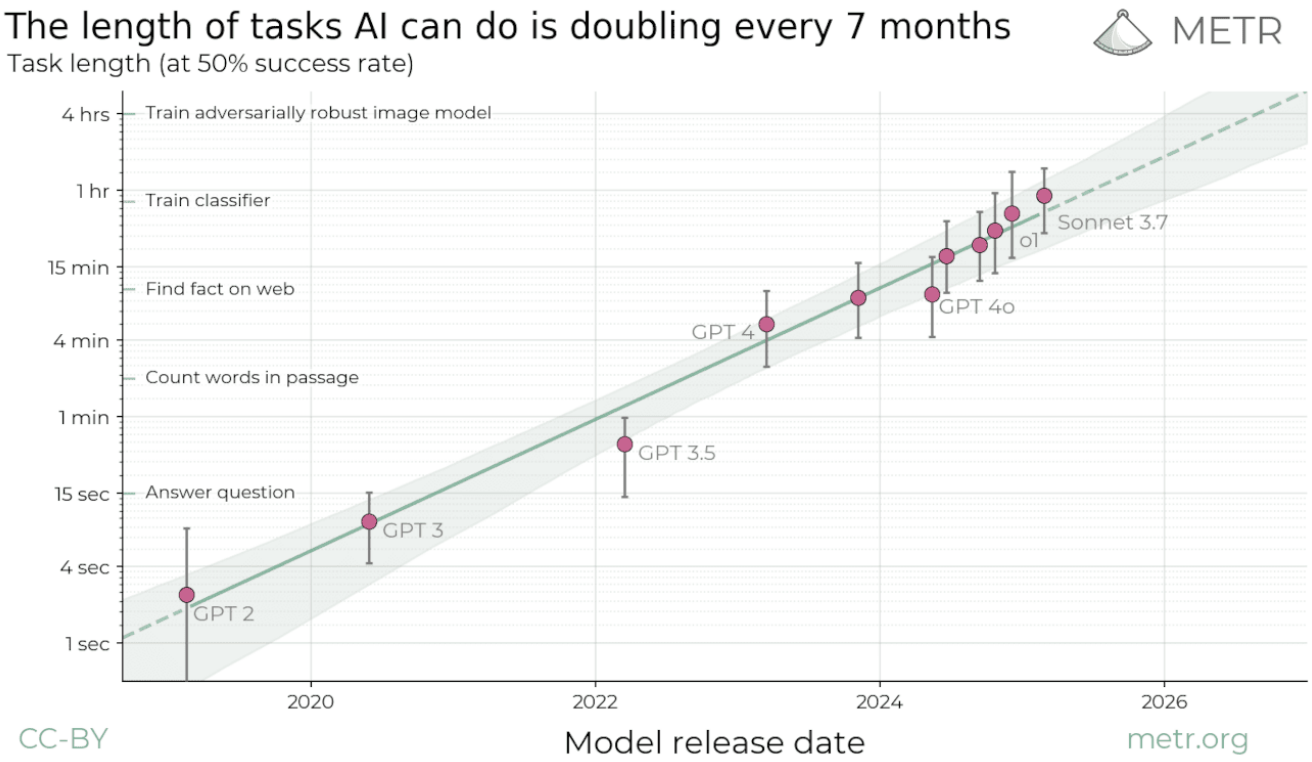


Figure Credit: Song Han (MIT)

And consumes a lot of data!



Reasoning models solving long tasks



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 x 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:

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

- **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 $A\,B$ 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	$O(bnd^2)$	$O(d^2 + 2bnd)$
$\boldsymbol{Q_i}, \boldsymbol{K_i} \in \mathbb{R}^{b\times n\times\frac{d}{m}}$	$P_i \leftarrow \text{softmax}\left(\frac{\boldsymbol{Q_i}\boldsymbol{K_i}^{\text{T}}}{\sqrt{d/m}}\right)$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\boldsymbol{V_i} \in \mathbb{R}^{b\times n\times\frac{d}{m}}, P_i \in \mathbb{R}^{b\times n\times n}$	$\text{head}_i \leftarrow P_i\boldsymbol{V_i}$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$
$\boldsymbol{W^O} \in \mathbb{R}^{d\times d}, \text{head}_i \in \mathbb{R}^{b\times n\times\frac{d}{m}}$	$Y = \text{Concat}(\text{head}_1, \ldots, \text{head}_m)\boldsymbol{W^O}$	$O(bnd^2)$	$O(2bnd + d^2)$
$Y \in \mathbb{R}^{b\times n\times d}, \boldsymbol{W}_1 \in \mathbb{R}^{d\times d_{\text{ff}}}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_{\text{ff}}\times d}$	$Y = \text{ReLU}(Y\boldsymbol{W}_1)\boldsymbol{W}_2$	$O(16bnd^2)$	$O(2bnd + 8d^2)$
---	Total	$O(bnd^2 + bn^2d)$	$O(bnd + bmn^2 + d^2)$

https://le.qun.ch/en/blog/2023/05/13/transformer-batching/

The bottlenecks

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

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

- 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)

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?

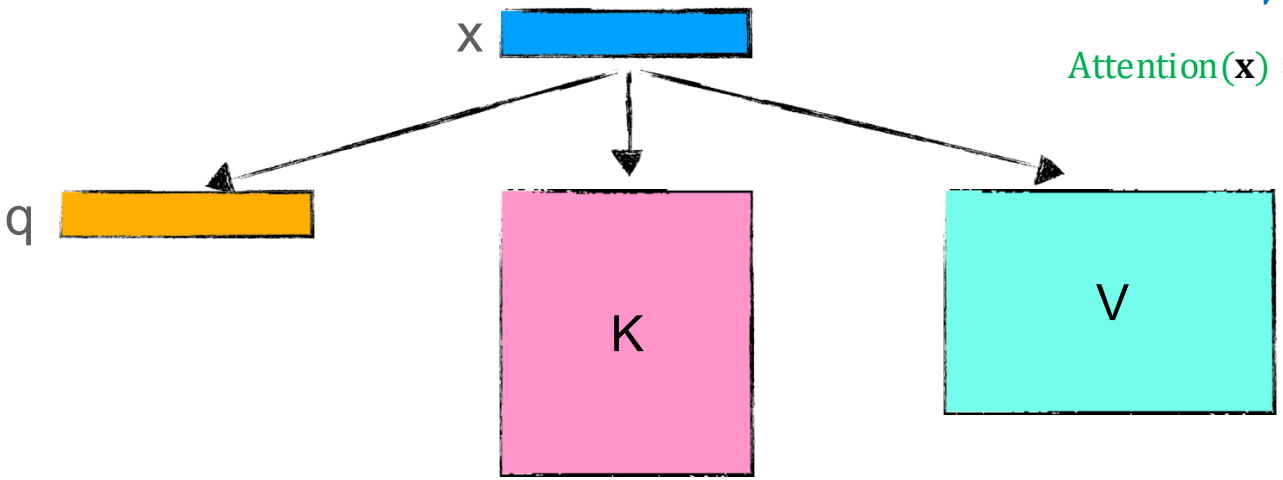
Q: Pulse Check: What is a KV-Cache?

$$Q = \mathbf{x}\mathbf{W}^{q}$$
$$K = \mathbf{x}\mathbf{W}^{k}$$
$$V = \mathbf{x}\mathbf{W}^{v}$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$

Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$

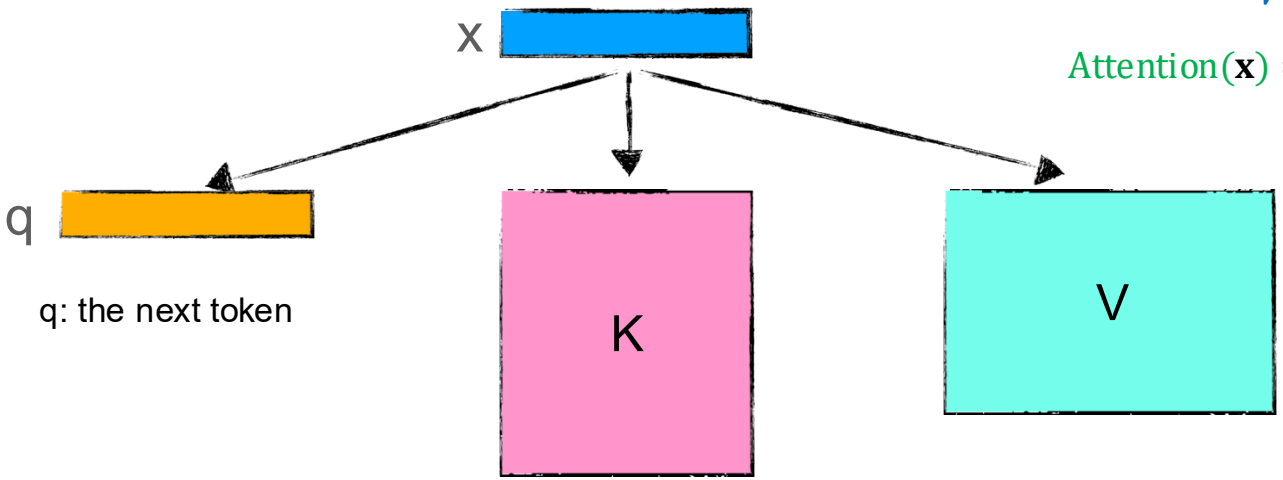
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

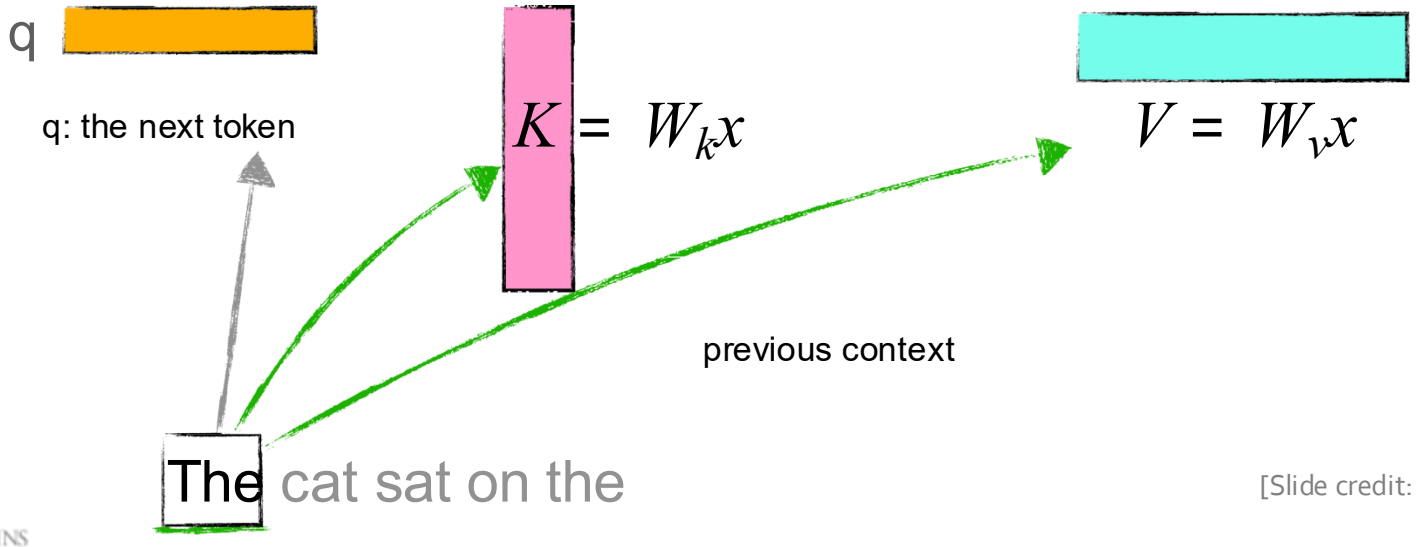
$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

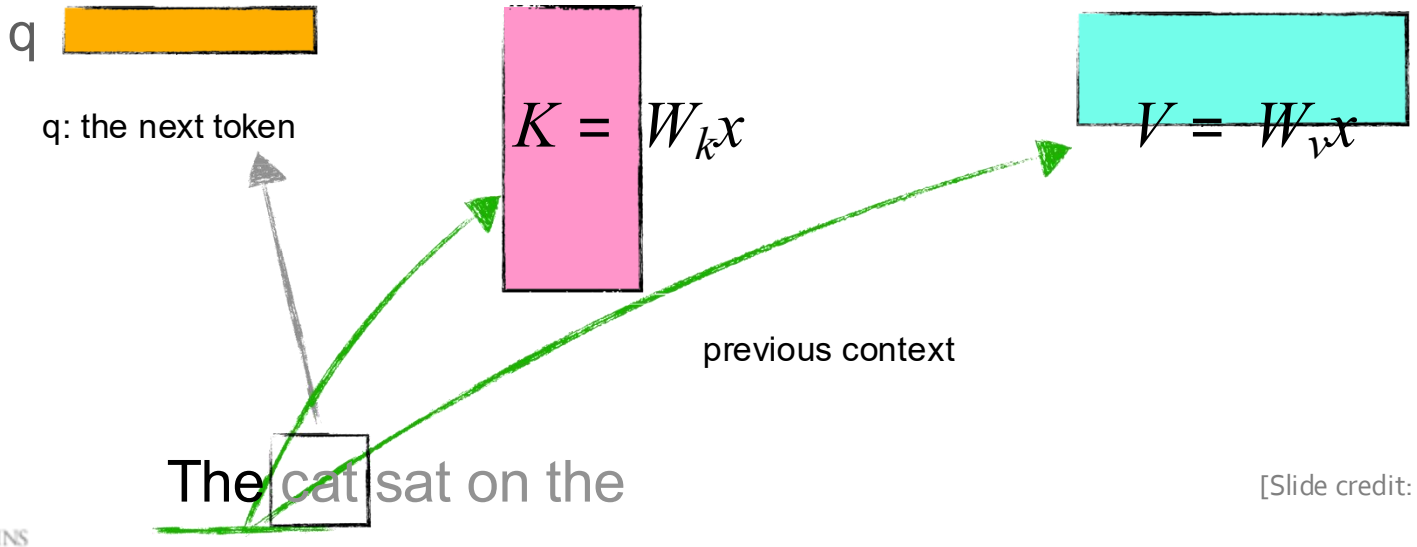
$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

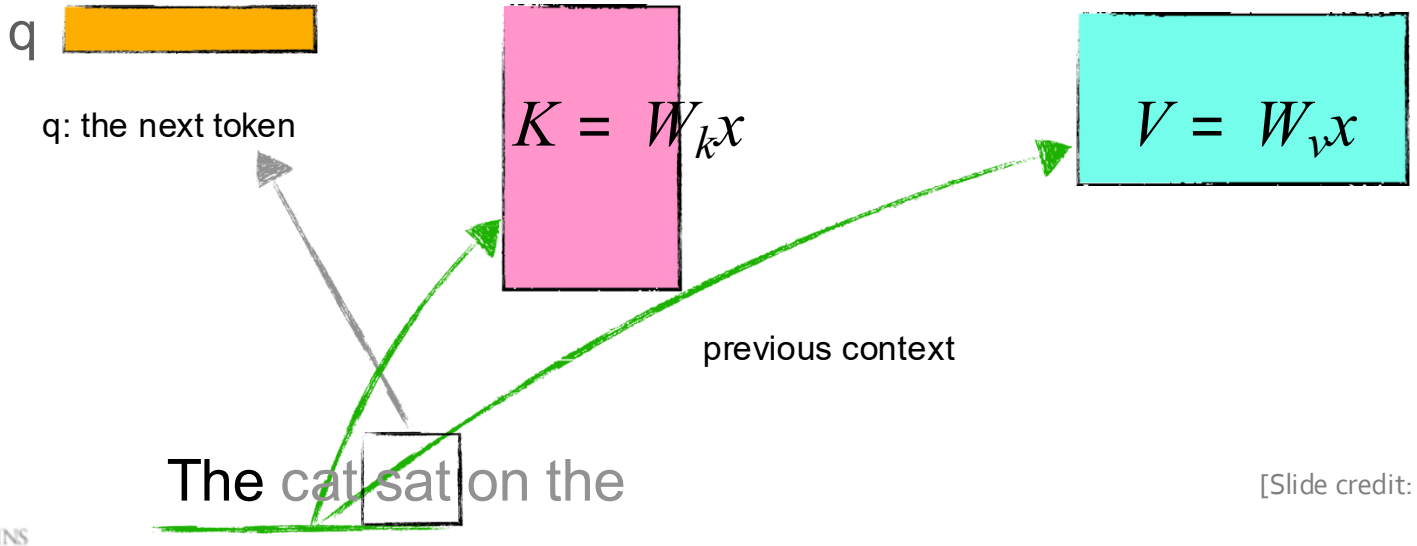
$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

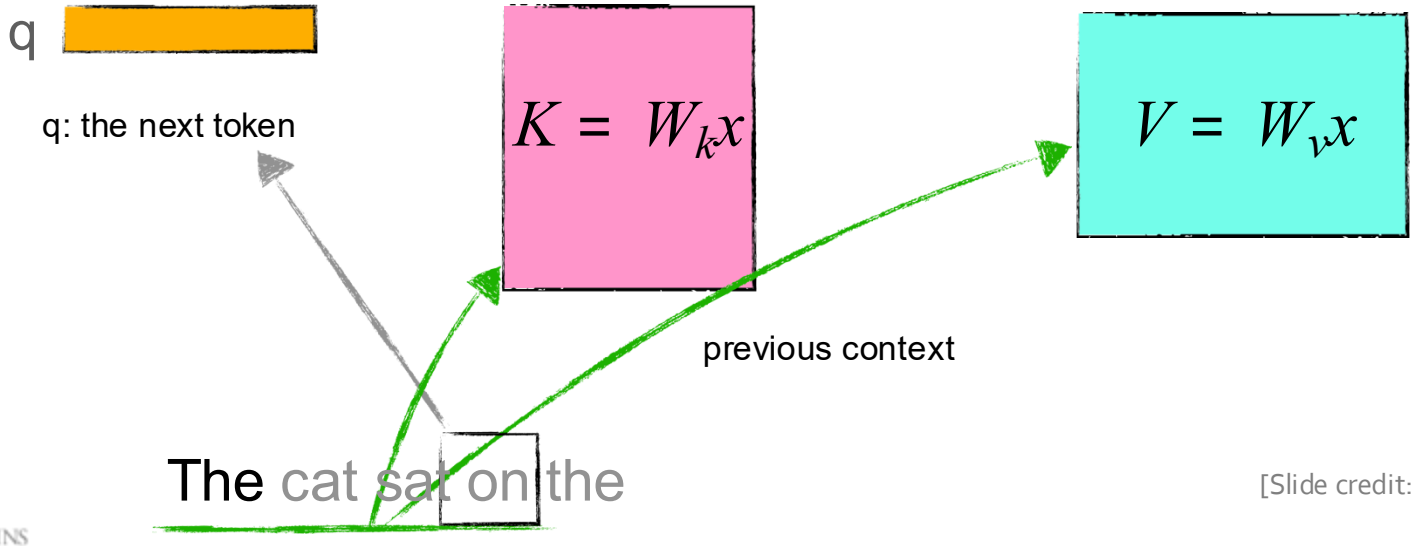
$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



[Slide credit: Arman Cohan]

Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$
$$\text{Attention}(\mathbf{x}) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d}}\right)V$$



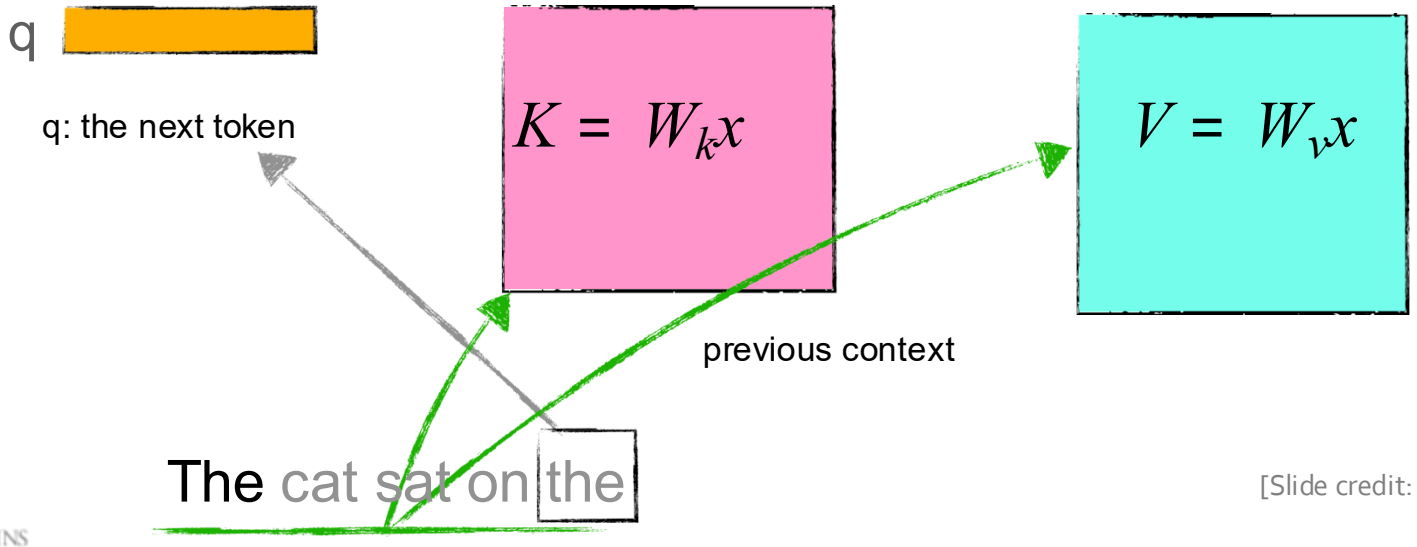
[Slide credit: Arman Cohan]

Self-Attention During Inference

$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$

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

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



[Slide credit: Arman Cohan]

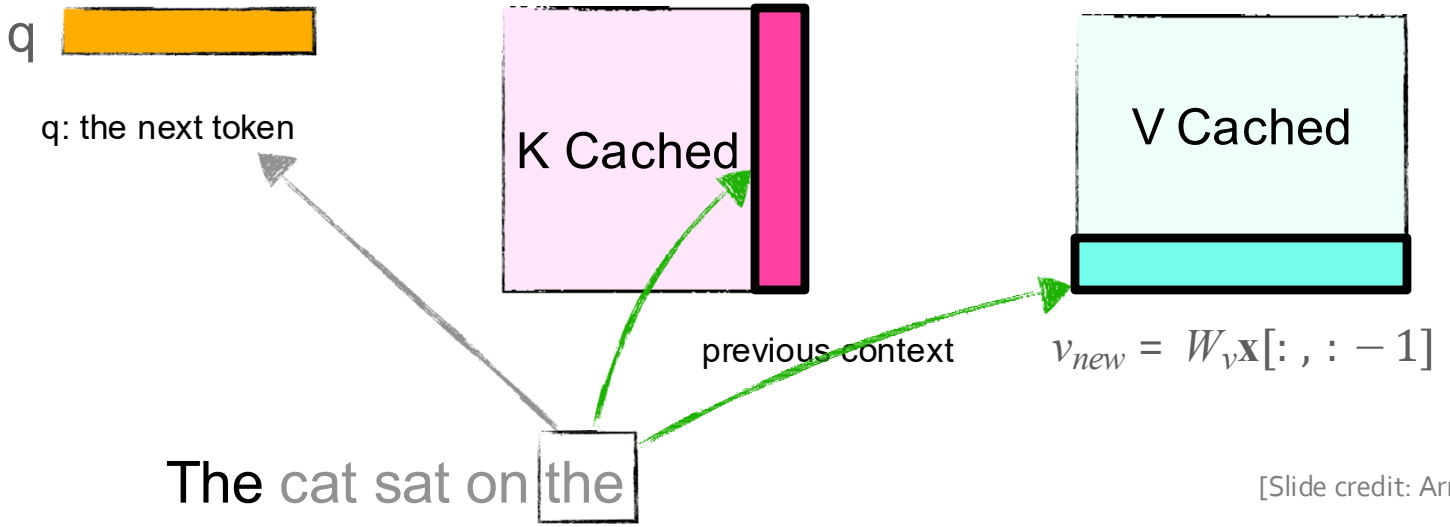
KV-Cache for reducing inference redundancy

$$Q = \mathbf{x}\mathbf{W}^q$$
$$K = \mathbf{x}\mathbf{W}^k$$
$$V = \mathbf{x}\mathbf{W}^v$$

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

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

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

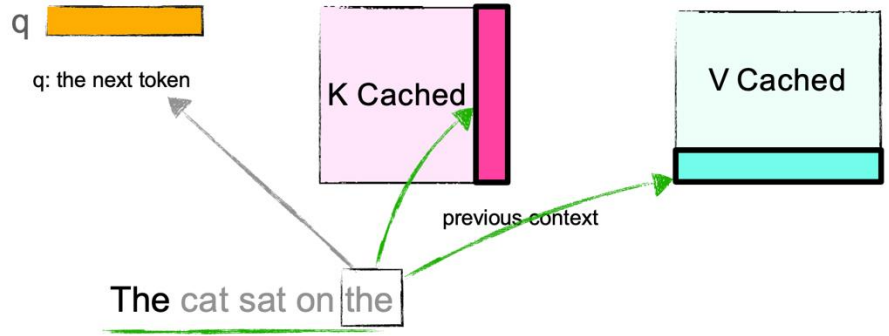


[Slide credit: Arman Cohan]

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. $2bndLk$



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

Batch Size: 1

Sequence Length: 1024

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                                    (q_ids)
        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]))
```

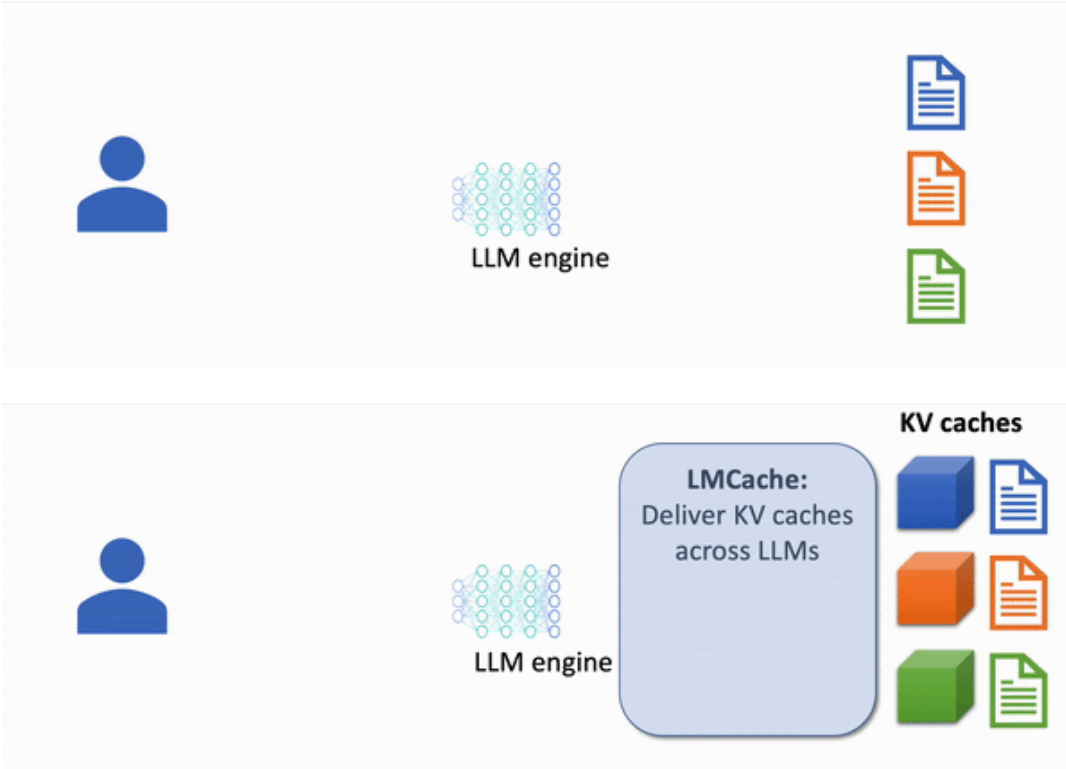
Note, here we don't need to re-feed the prefix tokens (prefix_ids) to the model. Why?

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
)
```

KV-cache for knowledge intensive tasks

- Creative use of KV-cache can help you speed up tasks that requires repeated use of knowledge that is fixed ahead of time and may be repeated across different inputs.
- For example: the task of answering questions based on retrieved documents.



Visualization from: https://lmcache.ai/

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

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

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., $n^2 \to n$ or $n \to 1$).

Query/key computations get combined with KV-cache

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

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

https://lequ7.com/blog/2023/05/fast-transformer-batching/

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:

  (# of floating-point operations) / (# 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: AI < 10 FLOPs/byte
  - Balanced: 10 ≤ AI ≤ 100 FLOPs/byte
  - Compute-bound: AI > 100 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 AI $= 71$.
  - If $n = 30$ (sent len), $b = 20$ (batch size), $d = 512$ (hidden dim). Then AI $= 179$.

Arithmetic Intensity of Training Self-Attention

Bonus

Operation	Computations	IO	Arithmetic Intensity
$\mathbf{x}\mathbf{W}_i^q$, $\mathbf{x}\mathbf{W}_i^k$, $\mathbf{x}\mathbf{W}_i^v$ for m heads	$O(bnd^2)$	$O(d^2 + 2bnd)$	$O\left(\left({}^1/_d + {}^1/_{bn}\right)^{-1}\right)$
$P_i \leftarrow \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d/m}}\right)$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$	$O\left(\left({}^m/_d + {}^1/_n\right)^{-1}\right)$
$\text{head}_i \leftarrow P_i V_i$ for m heads	$O(bn^2d)$	$O(2bnd + bmn^2)$	$O\left(\left({}^m/_d + {}^1/_n\right)^{-1}\right)$
$Y = \text{Concat}(\text{head}_1, \ldots, \text{head}_m)W^O$	$O(bnd^2)$	$O(2bnd + d^2)$	$O\left(\left({}^1/_d + {}^1/_{bn}\right)^{-1}\right)$
			$O\left(\left({}^1/_d + {}^1/_{bn}\right)^{-1}\right)$

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

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
$\mathbf{x}\mathbf{W}_i^q$, $\mathbf{x}\mathbf{W}_i^k$, $\mathbf{x}\mathbf{W}_i^v$ for m heads	$O(bd^2)$	$O(d^2 + 2bd)$	$O\left(\left({}^1/_d + {}^1/_b\right)^{-1}\right)$
			$O\left(\left(1 + {}^m/_d + {}^1/_n\right)^{-1}\right)$
			$O\left(\left(1 + {}^m/_d + {}^1/_n\right)^{-1}\right)$
$Y = \text{concat}(\text{head}_1, \ldots, \text{head}_m)W^O$			$O\left(\left({}^1/_d + {}^1/_b\right)^{-1}\right)$
	$O(16bd^2)$	$O(2bd + 8d^2)$	$O\left(\left({}^1/_d + {}^1/_b\right)^{-1}\right)$

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! Note this is partly due to the memory-bandwidth cost of repeatedly loading the large "keys" and "values" tensors.

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

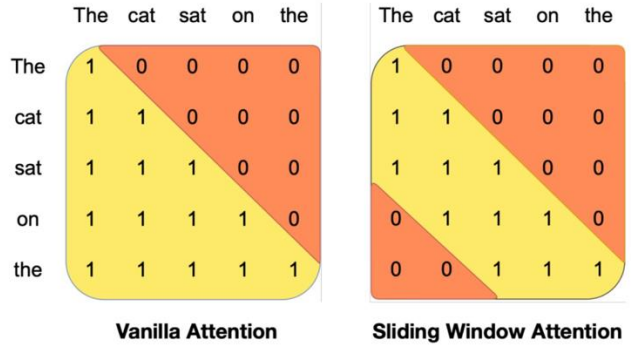
KV-Cache drag

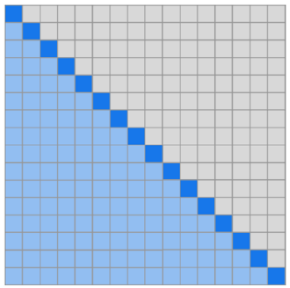
- Slowdown of autoregressive decoding.
  - As the sequence length grows, KV cache size increases, making cache lookup slower.
  - As we generate more output tokens (i.e. chatbot responding to user), throughput will slow down.

- **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.

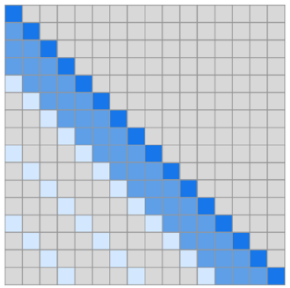
Sparse / sliding window attention

- 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.

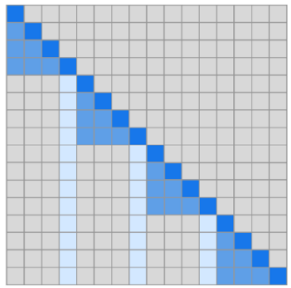




(a) Transformer



(b) Sparse Transformer (strided)



(c) Sparse Transformer (fixed)

Notable models:
GPT3 and Mistral

[Generating Long Sequences with Sparse Transformers, 2019]

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.

[Slide credit: Samet Oymak]

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

- **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, the hat's color is blue, and the pants' color is green.
- KV Cache: last_50 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is red. This is a picture of a kid in a red shirt, a blue hat, and green pants.
- KV Cache: last_20 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is a red shirt's'!!!!!!!!!!!!!!!;!!
- KV Cache: last_10 -> Generated: The kid is wearing a red shirt, a blue hat, and green pants. The kid's shirt's color is is is is is's's's's's's's's's's }s's c, and and and

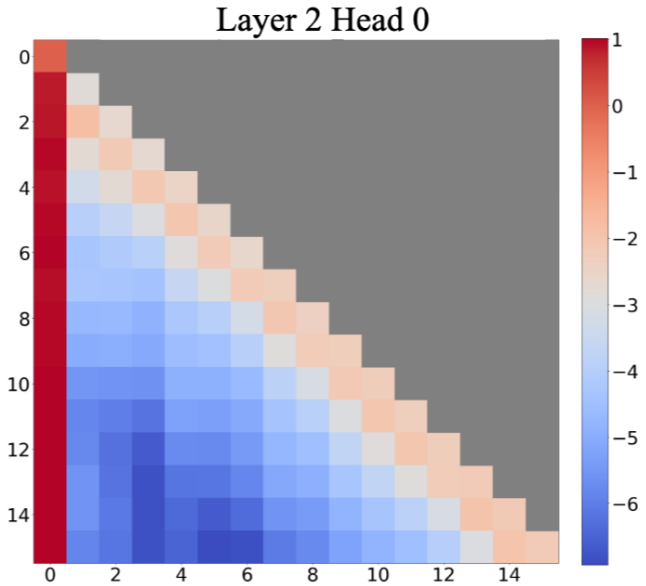
[Example credit: Sungwon Kim]

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)

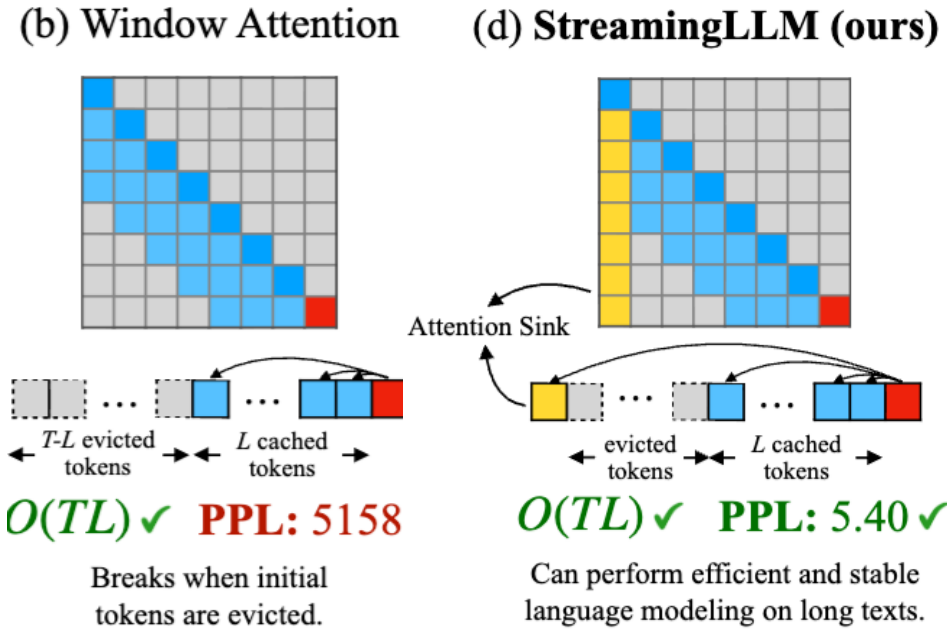


[Efficient Streaming Language Models with Attention Sinks, 2023]

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).



[Efficient Streaming Language Models with Attention Sinks, 2023]

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. 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

[Example credit: Sungwon Kim]

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>

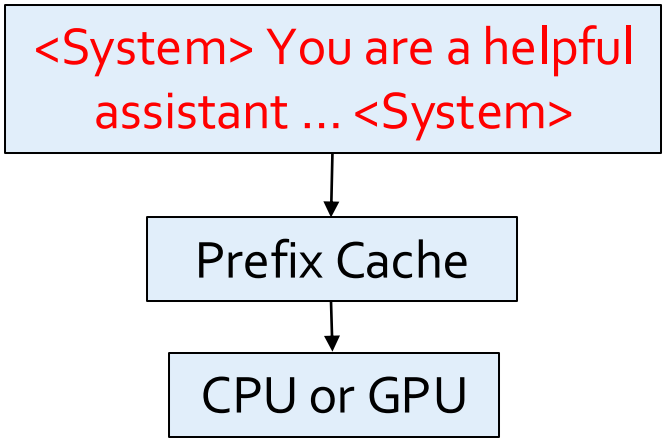
<System> You are a helpful assistant … <System>
<User> Help me revise my email … <User>

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`

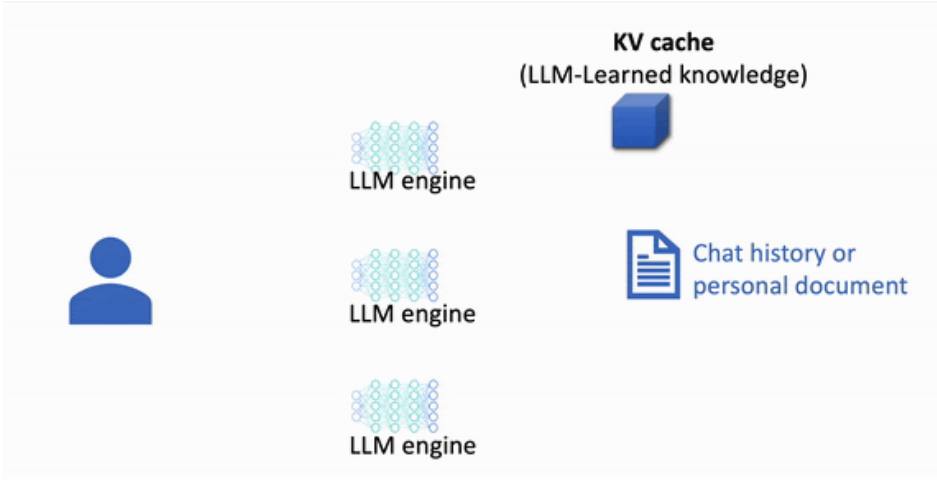


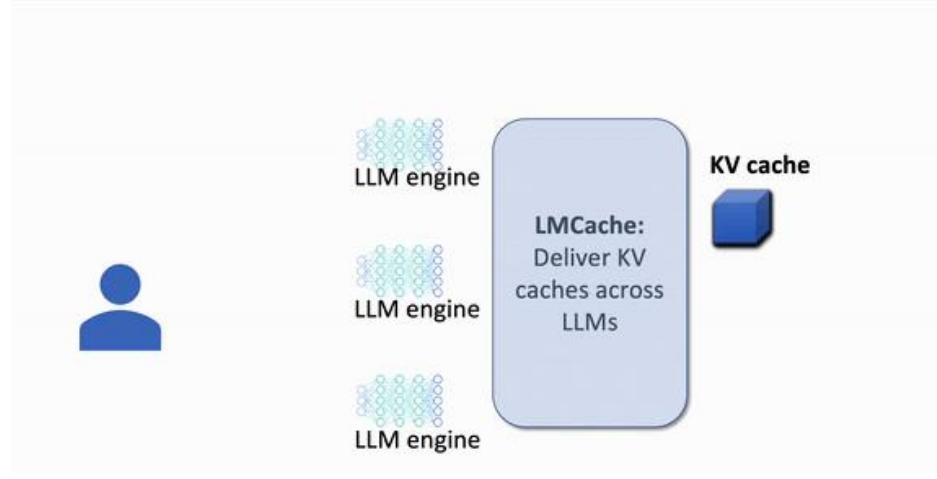
But, can we slice the activations to fit them in different GPUs?

- Yes, by Tensor Parallelism

Prefix Caching

- Can accelerate one user using multiple language models.
- Different users using one language models.



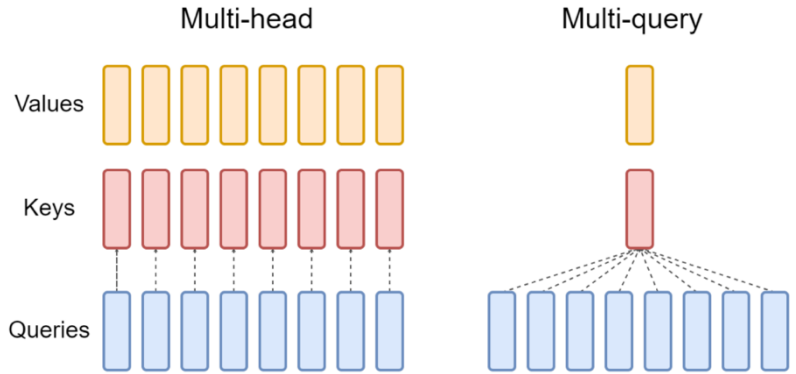


Visualization from: https://lmcache.ai/

Architecture change to reduce IO

Multi-Query Attention (MQA)

- The idea is to reduce the memory-bandwidth cost of repeatedly loading the large "keys" and "values" tensors.
- **Key idea** – have multiple queries, but just one dimension for keys and values.



Small PPL w/ MQA [Shazeer 2019]

Attention	h	d_k, d_v	d_{ff}	dev-PPL
multi-head	8	128	8192	**29.9**
multi-query	8	128	9088	30.2
multi-head	1	128	9984	31.2
multi-head	2	64	9984	31.1
multi-head	4	32	9984	31.0
multi-head	8	16	9984	30.9

Bonus

MQA in practice

```
        # Independent queries, but shared keys and values
        self.W_q = nn.Linear(embed_dim, embed_dim, bias=False)  # Queries
        self.W_kv = nn.Linear(embed_dim, 2 * self.head_dim, bias=False)  # Shared Key and Value

        self.out_proj = nn.Linear(embed_dim, embed_dim)
```

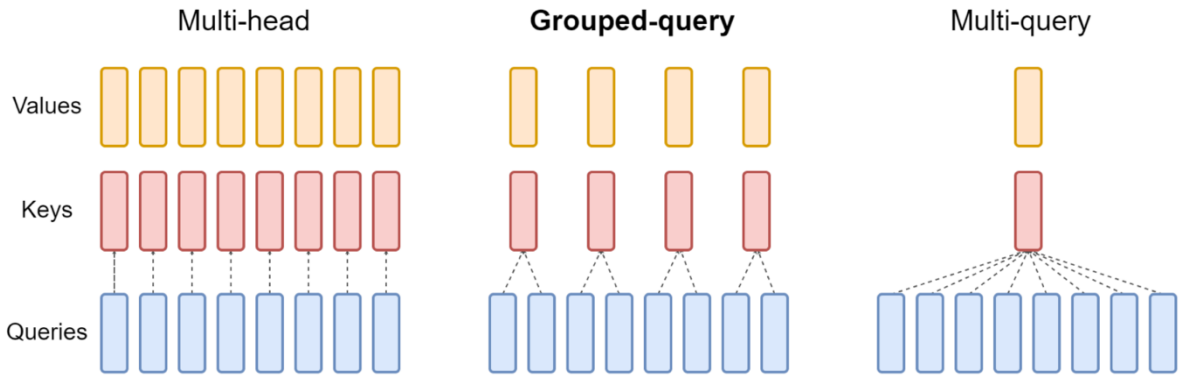
```
def forward(self, x):
    batch_size, seq_len, _ = x.shape

    # Compute Queries (B, L, D) → (B, L, H, D/H) → (B, H, L, D/H)
    Q = self.W_q(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

    # Compute shared Keys and Values (B, L, D) → (B, L, 2 * (D/H)) → (B, 1, L, D/H)
    KV = self.W_kv(x).view(batch_size, seq_len, 2, self.head_dim).permute(2, 0, 1, 3)
    K, V = KV[0].unsqueeze(1), KV[1].unsqueeze(1)  # Shared across all heads

    # Scaled Dot-Product Attention
    attn_weights = torch.einsum("bhqd,bkhd->bhqk", Q, K) / (self.head_dim ** 0.5)
    attn_weights = torch.nn.functional.softmax(attn_weights, dim=-1)
    output = torch.einsum("bhqk,bkhd->bhqd", attn_weights, V)

    # Merge heads and apply output projection
    output = output.transpose(1, 2).reshape(batch_size, seq_len, self.embed_dim)
    return self.out_proj(output)
```

Script

Grouped Query-Attention (GQA)

Notable models:
Llama 2, Mistral, Qwen2

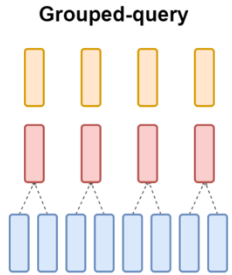
- An interpolation between "multi-head" attention and "multi-query" attention.



- Simple knob to control expressiveness (key-query ratio) and inference efficiency

GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023

Grouped Query-Attention (GQA)

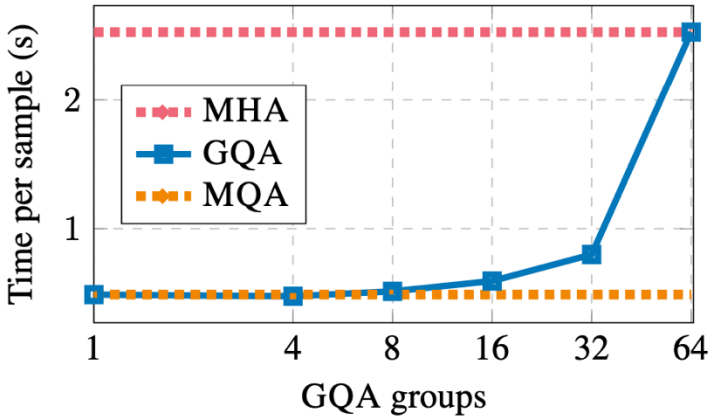


- Does it actually work? Depends.

Output quality of various models; all these SA variants are on-par on quality.

Model	WMT	TriviaQA
	BLEU	F1
MHA-Large	27.7	78.2
MHA-XXL	28.4	81.9
MQA-XXL	28.5	81.3
GQA-8-XXL	28.4	81.6

Inference speed as a function of GQA group size — 8 heads gives you inference speed as good as 1 head!

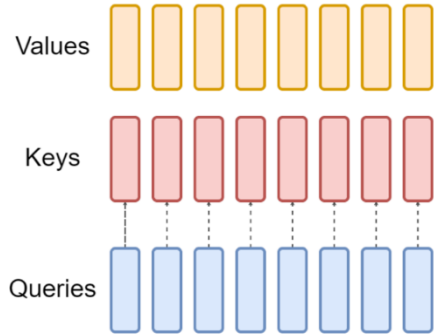


GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023

Recap

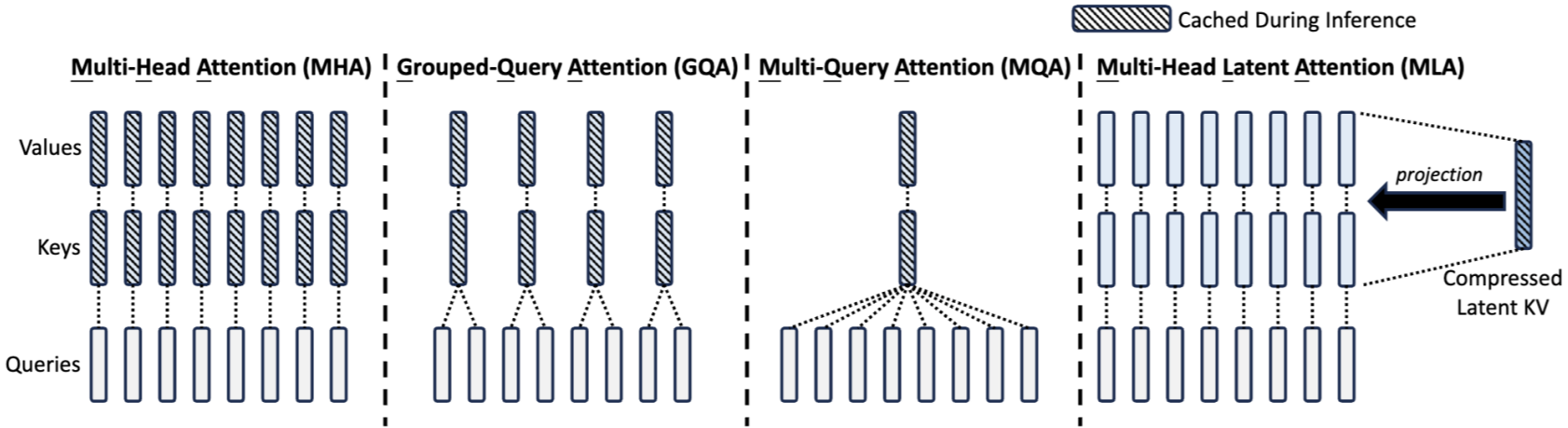
- SA's AI during inference is not good.
  - We're doing a lot of IO relative to computations (KV drag).
- Sliding window attention: sparsifying attention pattern by looking at nearby things.
- MQA and GQA: sharing attention keys and values.

Do you the dimensions of my latent embeddings?



Low-Rank Key-Value Joint Compression

- Joint compress (project) keys and values to reduce KV cache:



Low-Rank Key-Value Joint Compression

- Joint compress (project) keys and values to reduce KV cache:

$$\mathbf{c}_t^{KV} = W^{DKV}\mathbf{h}_t,$$
$$\mathbf{k}_t^{C} = W^{UK}\mathbf{c}_t^{KV},$$
$$\mathbf{v}_t^{C} = W^{UV}\mathbf{c}_t^{KV}$$

where $\mathbf{c}_t^{KV} \in \mathbb{R}^{d_c}$ is the compressed latent vector for keys and values; $d_c (\ll d_h n_h)$ denotes the KV compression dimension; $W^{DKV} \in \mathbb{R}^{d_c \times d}$ is the down-projection matrix; and $W^{UK}, W^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$ are the up-projection matrices for keys and values, respectively. During inference, MLA only needs to cache $\mathbf{c}_t^{KV}$, so its KV cache has only $d_c l$ elements, where l denotes the number of layers. In addition, during inference, since W^{UK} can be absorbed into W^{Q}, and W^{UV} can be absorbed into W^{O}, we even do not need to compute keys and values out for attention. Figure 3 intuitively illustrates how the KV joint compression in MLA reduces the KV cache.

MLA: KV-Cache size comparison

- MLA's KV cache size is equal to GQA with only 2.25 groups. How about its performance? (vs. MHA and GQA)

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2} d_h l$	Stronger

Table 1 | Comparison of the KV cache per token among different attention mechanisms. n_h denotes the number of attention heads, d_h denotes the dimension per attention head, l denotes the number of layers, n_g denotes the number of groups in GQA, and d_c and d_h^R denote the KV compression dimension and the per-head dimension of the decoupled queries and key in MLA, respectively. The amount of KV cache is measured by the number of elements, regardless of the storage precision. For DeepSeek-V2, d_c is set to $4d_h$ and d_h^R is set to $\frac{d_h}{2}$. So, its KV cache is equal to GQA with only 2.25 groups, but its performance is stronger than MHA.

DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Mode, 2024

MLA: KV-Cache size comparison

- MLA (DeepSeek-V2) shows better performance than MHA, but requires a significantly smaller amount of KV cache.

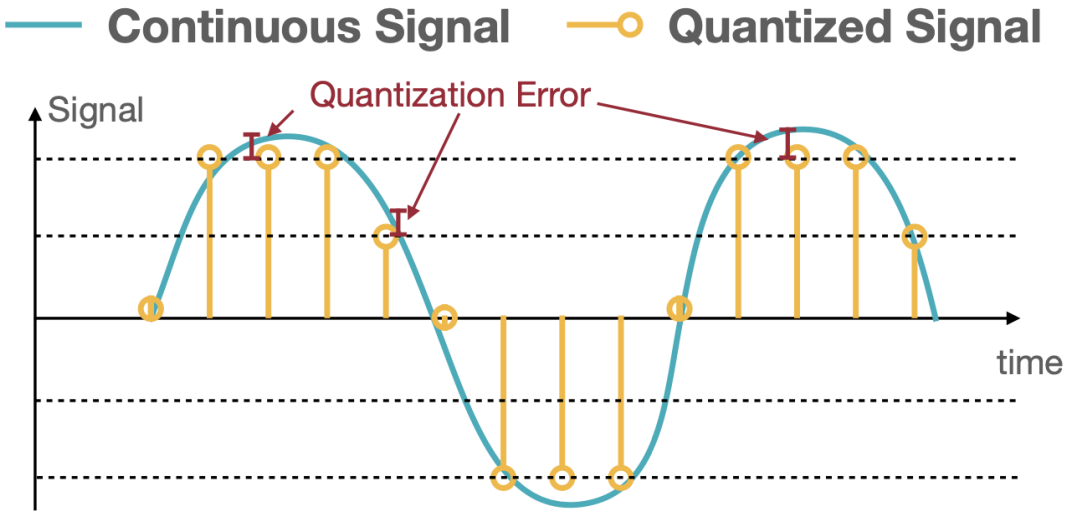
Benchmark (Metric)	# Shots	Small MoE w/ MHA	Small MoE w/ MLA	Large MoE w/ MHA	Large MoE w/ MLA
# Activated Params	-	2.5B	2.4B	25.0B	21.5B
# Total Params	-	15.8B	15.7B	250.8B	247.4B
KV Cache per Token (# Element)	-	110.6K	15.6K	860.2K	34.6K
BBH (EM)	3-shot	37.9	**39.0**	46.6	**50.7**
MMLU (Acc.)	5-shot	48.7	**50.0**	57.5	**59.0**
C-Eval (Acc.)	5-shot	**51.6**	50.9	57.9	**59.2**
CMMLU (Acc.)	5-shot	52.3	**53.4**	60.7	**62.5**

Table 9 | Comparison between MLA and MHA on hard benchmarks. DeepSeek-V2 shows better performance than MHA, but requires a significantly smaller amount of KV cache.

Quantization

Quantization: Mapping from high to low precision

- In deep learning, this process lets models run faster with less memory by storing weights/activations in lower precision (e.g., FP16 or INT8).



Numeric Data Types: FP32

- 32-bit **floating point** number or FP32 (IEEE 754)

Sign: 1 bit

Exponent: 8 bits

Range

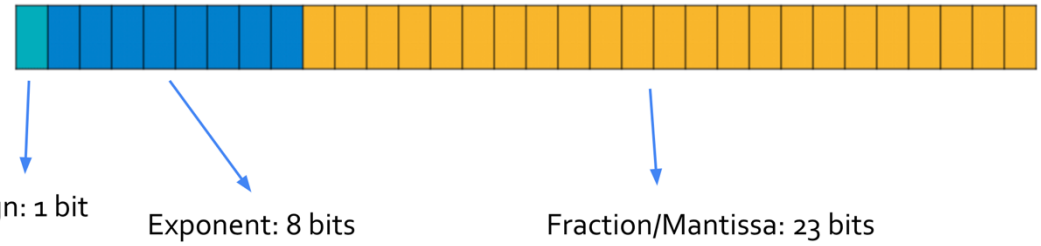
Fraction/Mantissa: 23 bits

Precision

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

Numeric Data Types: FP32 (example)

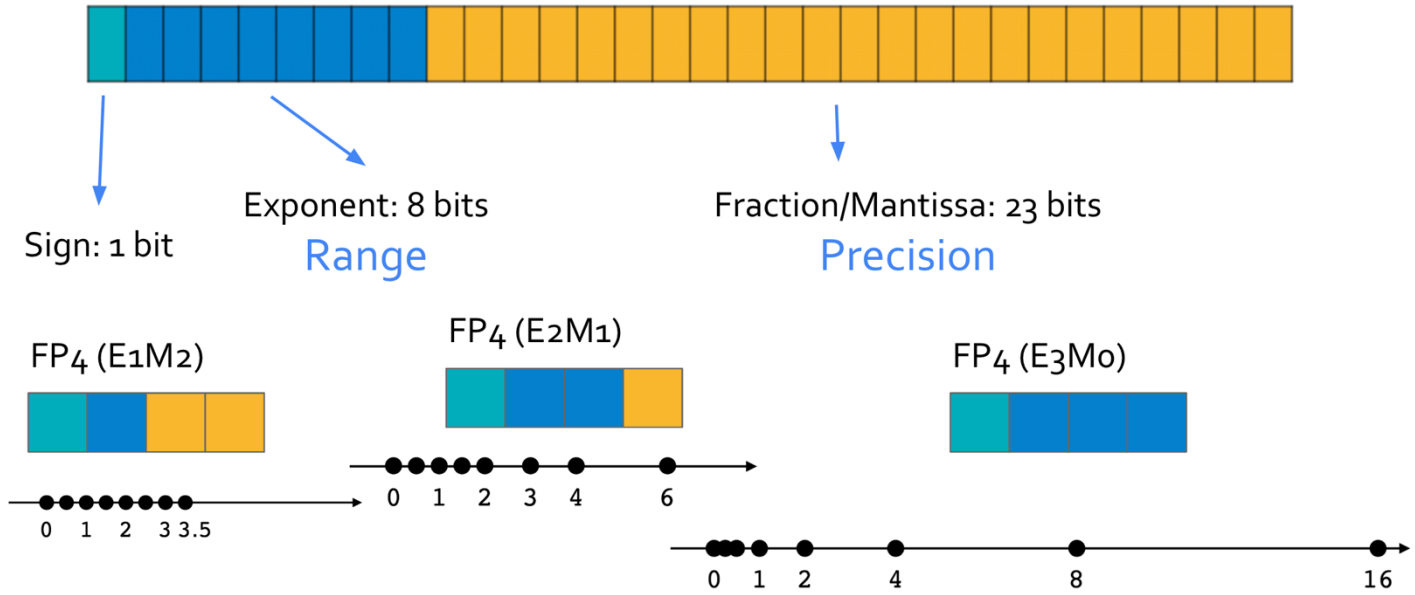
- This bit string corresponds to what number? 11000000110110000000000000000000
- Split first:
  - Sign = 1
  - Exponent = $(10000001)_2 = (129)_{10}$
  - 1+ Fraction: $(1.10110000000000000000000)_2 = (1.1011)_2 = 1 + 0.5 + 0.125 = 0.0625 = 1.6875$
- So, in total: $(-1)^1 \times 1.6875 \times 2^{(129-127)} = -0.6.75$



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

Floating Point Numbers: Range vs Precision

- Floating-point design is always **a trade-off between range and precision**
  - and this trade-off is central to quantization and mixed-precision computing.

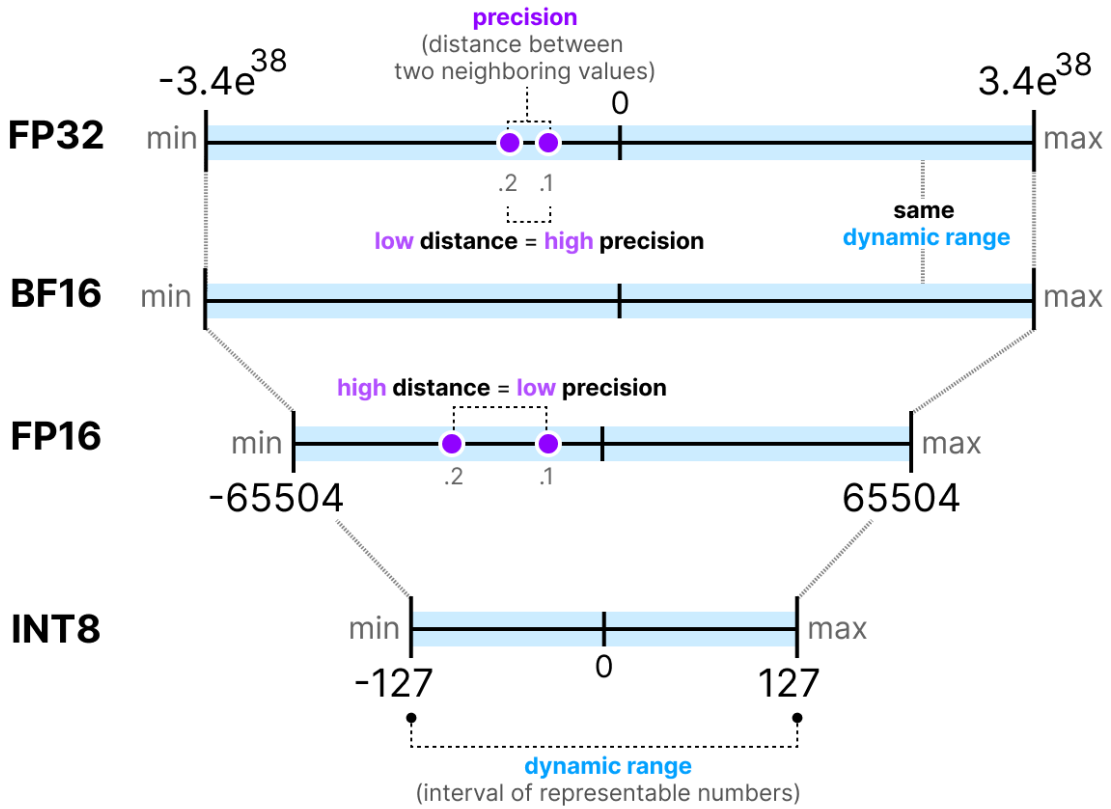


Floating Point Numbers

Different floating-point formats used in machine learning, showing how they trade **range** and **precision** based on how many bits are allocated to the **exponent** and **fraction (mantissa)**.

	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 (E_4M_3) 	4	3

Range vs Precision



https://newsletter.maartengrootendorst.com/p/a-visual-guide-to-quantization

During training LLMs, what numeric type is used where?

Component	Typical Precision	Purpose / Reason	Notes
Model Parameters	BF16 / FP16	Reduce memory use and training bandwidth while maintaining adequate range	BF16 preferred (same exponent range as FP32, avoids loss scaling)
Activations	BF16 / FP16	Efficient representation of large tensors during forward/backward passes	Some ops (e.g., softmax, attention) temporarily use FP32 for stability
Gradients	FP32	Prevent precision loss when summing many small updates	Down-cast to BF16/FP16 when applying updates
Optimizer States (e.g., Adam m/v)	FP32	Maintain stable parameter updates across steps	Typically 2–3× the memory of model weights
Checkpoint Storage	BF16 / FP32	Preserve accuracy and reproducibility	Mixed-precision checkpoints often store FP32 master weights

Example: LLama3 paper

GPU utilization. Through careful tuning of the parallelism configuration, hardware, and software, we achieve an overall BF16 Model FLOPs Utilization (MFU; Chowdhery et al. (2023)) of 38-43% for the configurations shown in Table 4. The slight drop in MFU to 41% on 16K GPUs with DP=128 compared to 43% on 8K GPUs with DP=64 is due to the lower batch size per DP group needed to keep the global tokens per batch constant during training.

Numerical stability. By comparing training loss between different parallelism setups, we fixed several numerical issues that impact training stability. To ensure training convergence, we use FP32 gradient accumulation during backward computation over multiple micro-batches and also `reduce-scatter` gradients in FP32 across data parallel workers in FSDP. For intermediate tensors, *e.g.*, vision encoder outputs, that are used multiple times in the forward computation, the backward gradients are also accumulated in FP32.

The Llama 3 Herd of Models

Example: LLama3 paper

- **Positional embedding**. We have chosen RoPE (Rotary Positional Embedding) (Su et al. 2021) as our preferred option for incorporating positional information into our model. RoPE has been widely adopted and has demonstrated success in contemporary large language models, notably PaLM (Chowdhery et al., 2022; Anil et al., 2023) and LLaMA (Touvron et al., 2023a;b). In particular, we have opted to use FP32 precision for the inverse frequency matrix, rather than BF16 or FP16, in order to prioritize model performance and achieve higher accuracy.

Qwen tech report, 2023

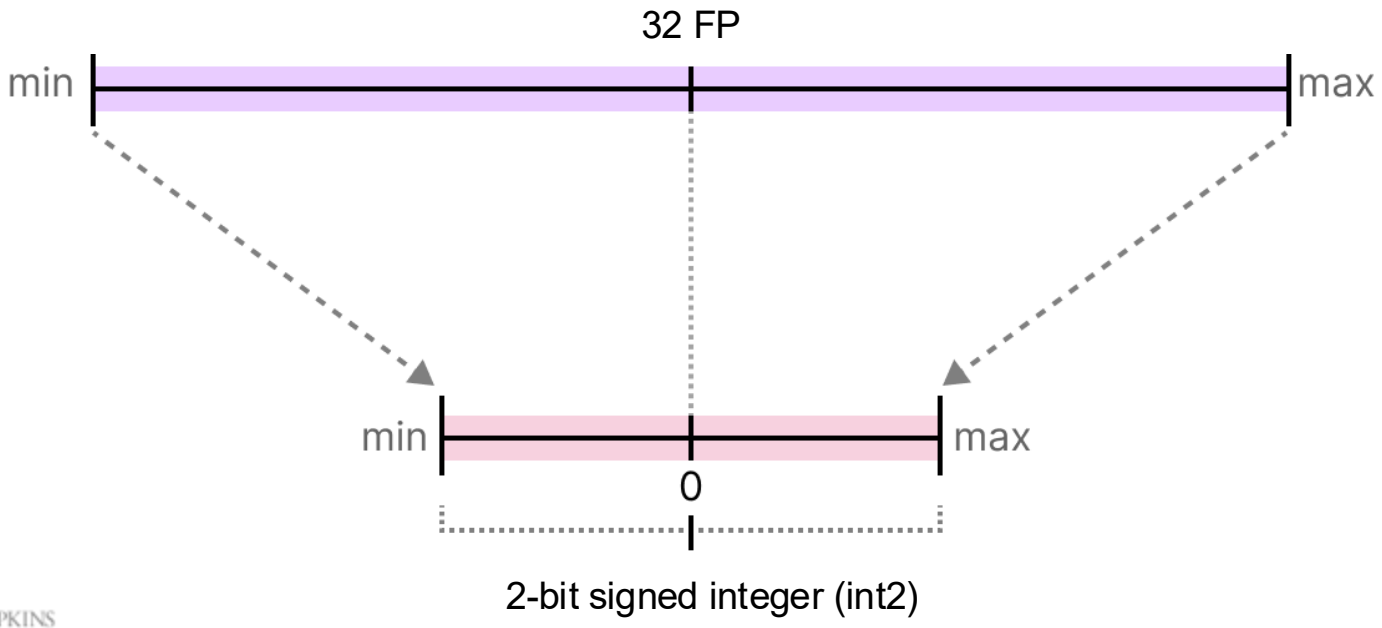
Terminology: “Full Precision"

- “Floating point” refers to **how** numbers are stored
- “Full precision” usually means **32-bit floating point (FP32)** — the traditional standard for numerical accuracy in deep learning and scientific computing.

Term	Meaning	Example
Floating point	Numeric format that represents real numbers using sign, exponent, mantissa	FP32, FP16, BF16
Full precision	Highest-precision floating-point format commonly used in training	FP32

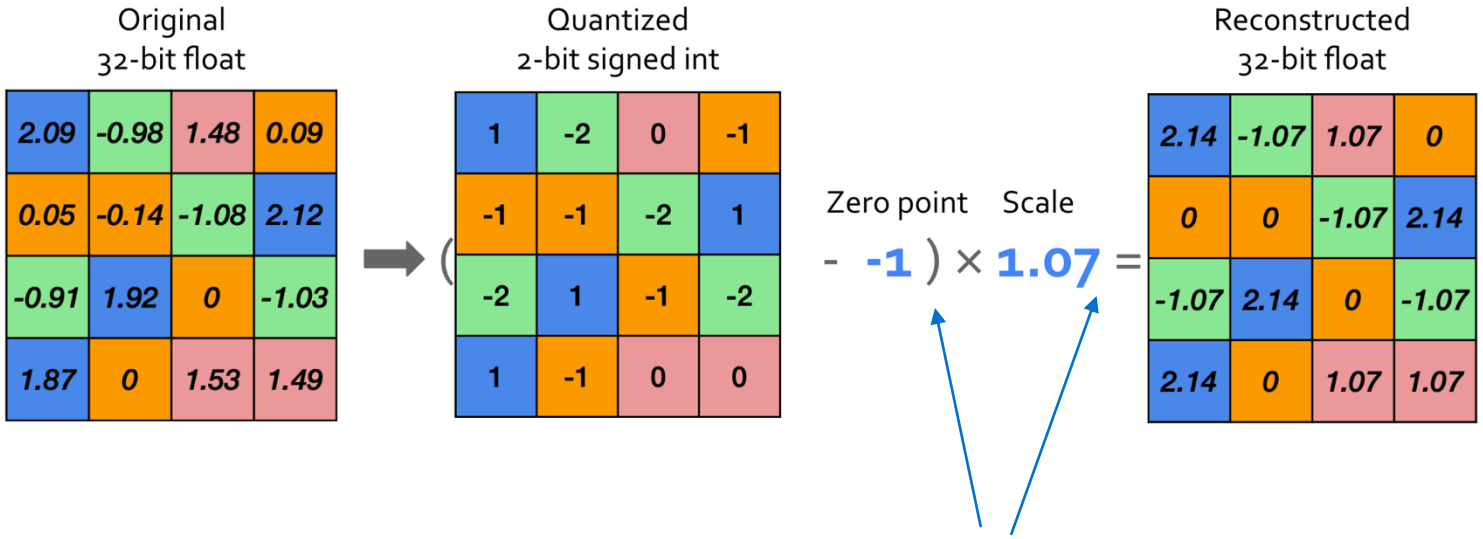
Linear Quantization

- **Linear quantization** maps continuous floating-point values to discrete integer levels using a uniform step size.



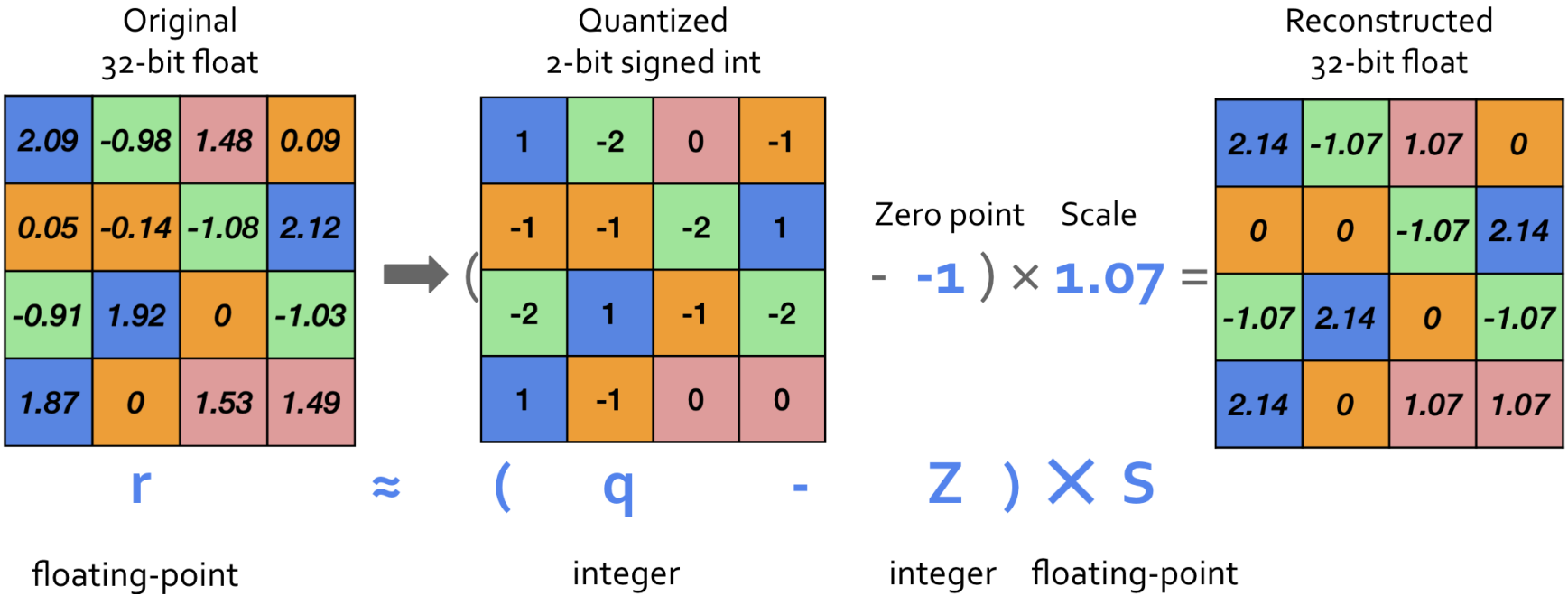
Linear Quantization

- **Linear quantization** maps continuous floating-point values to discrete integer levels using a uniform step size.
- The uniform step size is defined by a *scale* and *zero-point.*



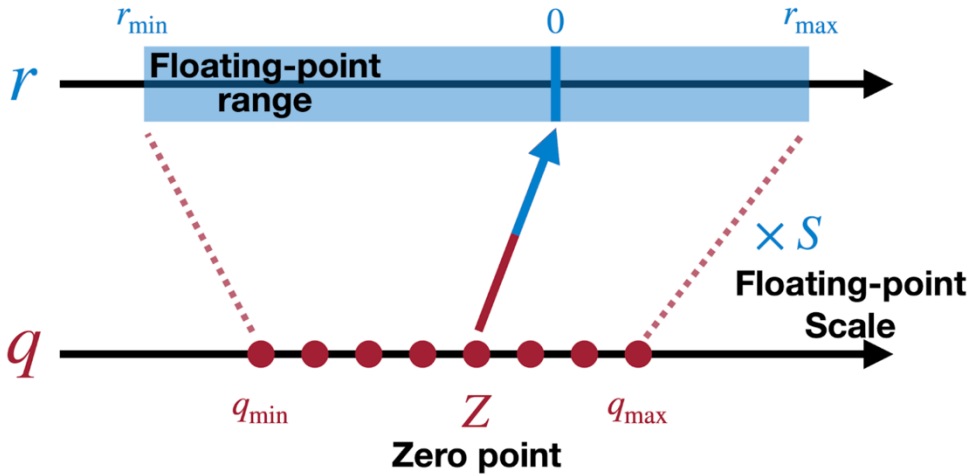
How to find these numbers?

Linear Quantization



Linear Quantization: Scale

- The **top line (r)** represents the original *floating-point range.*
- The **bottom line (q)** represents the *quantized integer rang*

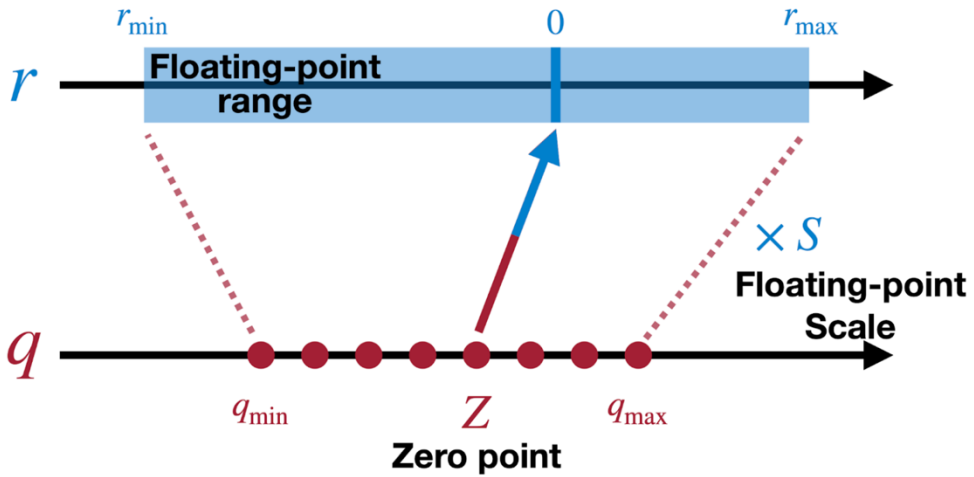


$$S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$$

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

Linear Quantization: Zero Point

- The **top line (r)** represents the original *floating-point range.*
- The **bottom line (q)** represents the *quantized integer rang*



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

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

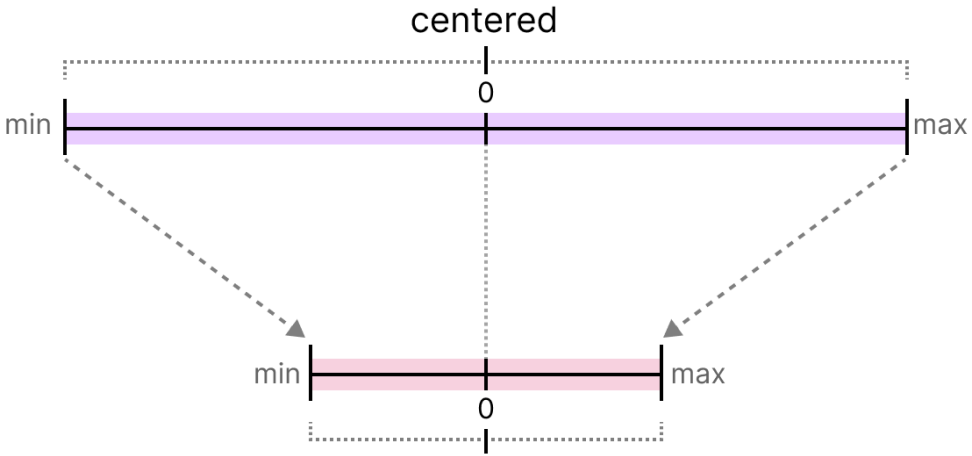
$$Z = \text{round}\left(q_{\min} - \frac{r_{\min}}{S}\right)$$

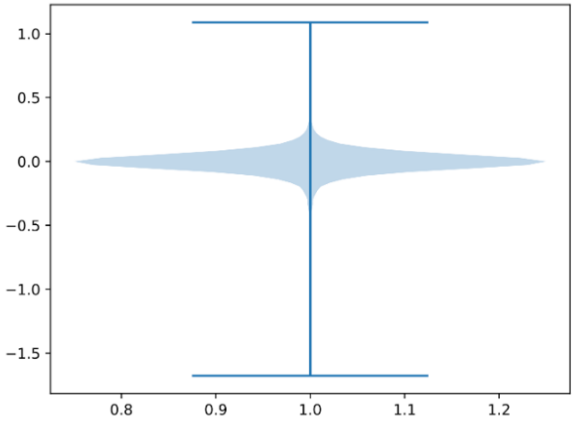
"Round" because Z must be an integer

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

Linear Quantization: Zero Point

- In practice, weights are usually centered around zero.
- Hence, we can set Z=0.





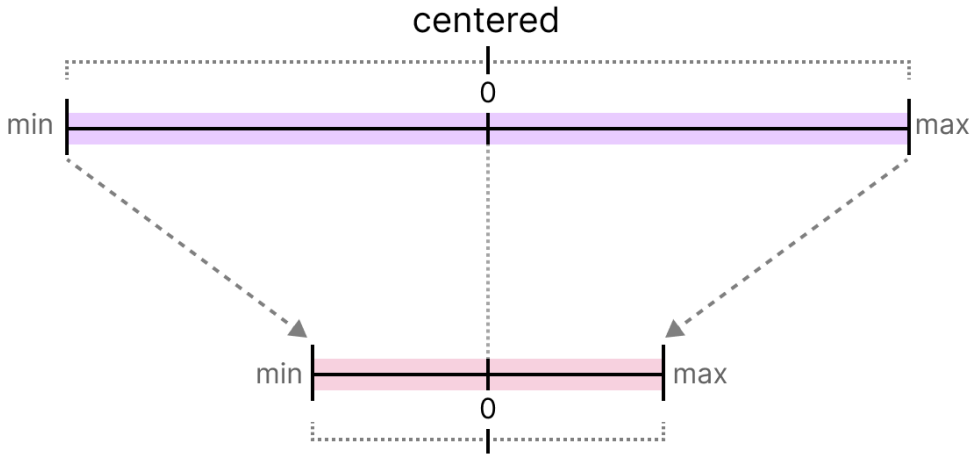
Weight distribution of first conv layer of ResNet-50.

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

Linear Quantization: Zero Point

- In practice, weights are usually centered around zero.
- Hence, we can set Z=0.
- This means that our formula for scale will be much simpler:

Called *absmax* or absolute maximum quantization



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

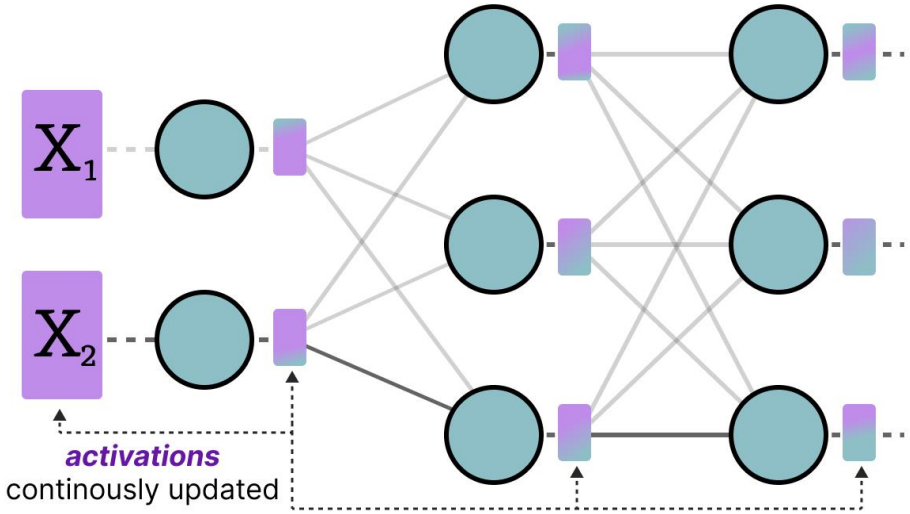
$$S = \frac{r_{\min}}{q_{\min} - Z} = \frac{-|r|_{\max}}{q_{\min}}$$

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

Quantization of LLMs

- What is an LLM?
  - **Weights**
  - Input
  - Their confluence result in **activations**, and ultimately outputs.

- **Weights** are static and known in-advance.
- **Activations** change based on input.

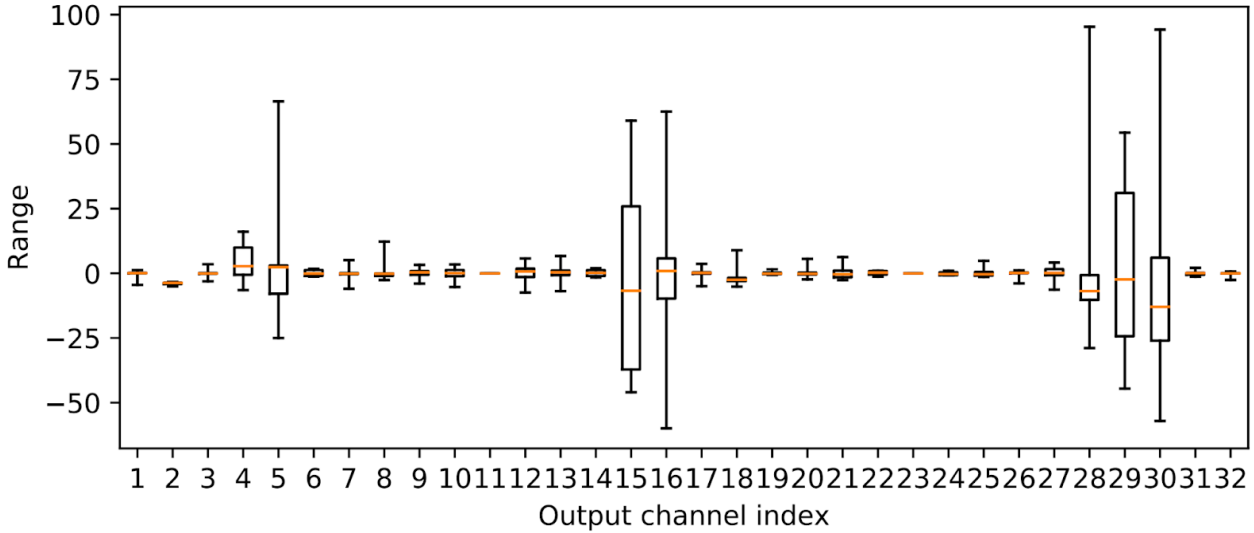


Quantization of LLMs: Weight vs Activations

Aspect	Weights	Activations
Nature	**Static**, known before inference	**Dynamic**, varies with input and layer outputs
Timing	Quantized **offline** (PTQ or QAT)	Quantized **on-the-fly** during inference
Distribution	Stable, near-zero symmetric	Variable, often skewed with large outliers
Precision	**Lower (4–8 bit)**	**Higher (8–16 bit)** to preserve range
Zero Point	Often symmetric (Z = 0)	Often asymmetric (non-zero mean)
Common Methods	GPTQ, AWQ	SmoothQuant, ZeroQuant

Quantization of LLM Activations: Outliers

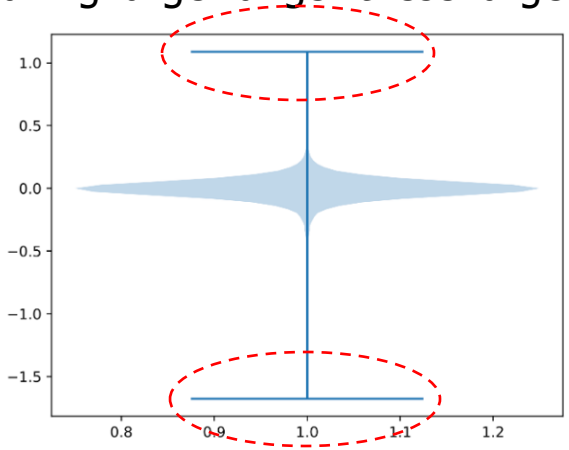
- The plot shows **activation ranges** (i.e., min–max spread) **for each output channel (dimension)** in the first layer of *MobileNetV2*.
- There exists many outliers in activations!



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

Quantization of LLM Activations : Outliers

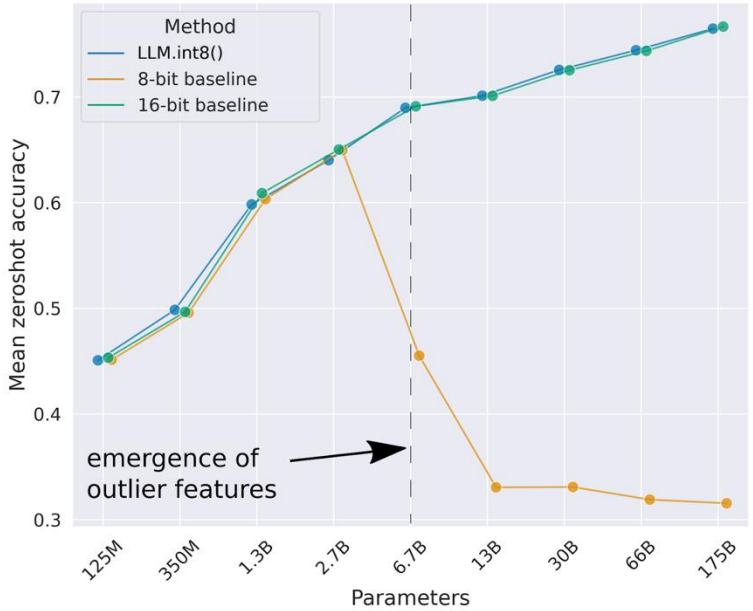
- The plot shows **activation ranges** (i.e., min–max spread) **for each output channel (dimension)** in the first layer of *MobileNetV2*.
- There exists many outliers in activations!
- It's not just about having large range: these large numbers are rare but important.



Weight distribution of first conv layer of ResNet-50.

Quantization of LLM Activations: Ignoring Outliers?

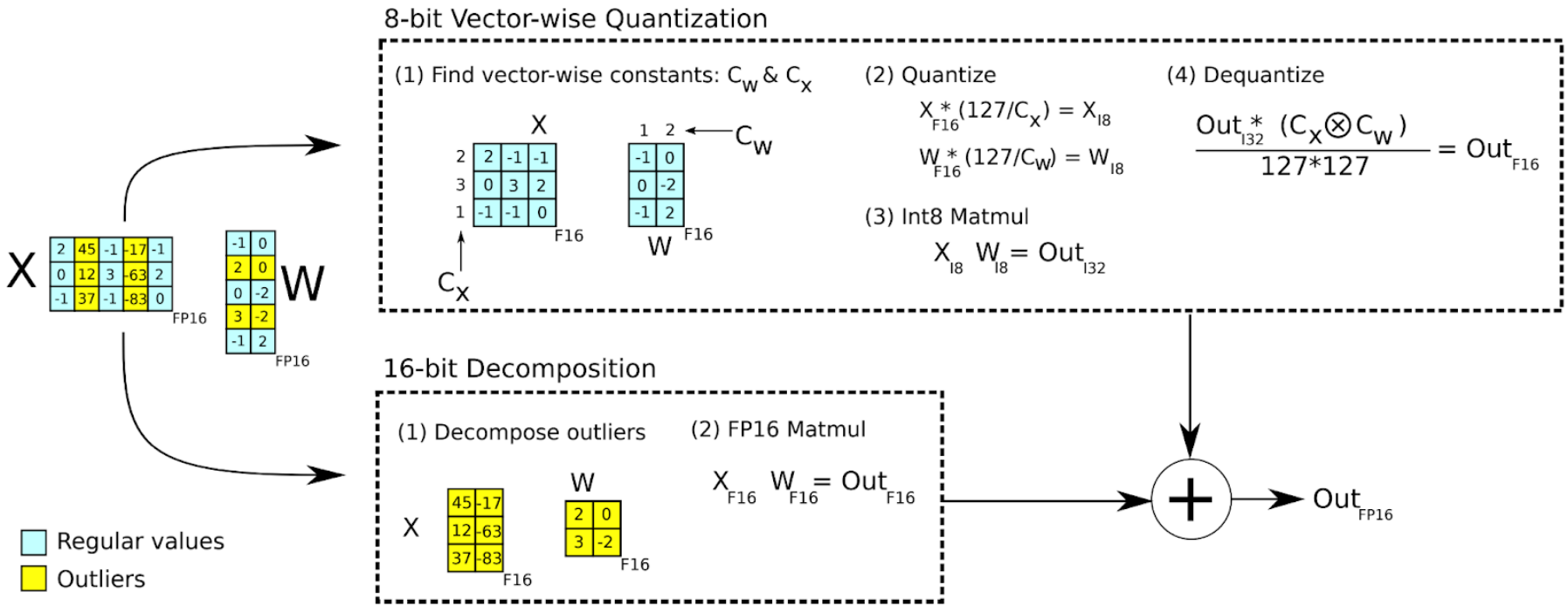
Ignoring the outliers (skipping them in quantization) significantly harms performance after quantization in LMs.



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

Quantization of LLM Activations: LLM.int8()

- Keep outlier channels in 16-bit, quantize the remaining channels.



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

Quantization of LLM Activations: LLM.int8()

- Under the hood, this calls `bitsandbytes`' implementation of `LLM.int8()` which automatically detects and handles outlier channels,

```
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-13b-hf",
    load_in_8bit=True,              # activates LLM.int8()
    device_map="auto"
)
```

Quantization of LLMs: Recap

- Quantization maps high-precision numbers (FP32, FP16, BF16) to lower-precision formats (FP8, INT8) to **save memory and speed up computation**.
- Used in both **training and inference** to reduce memory footprint and bandwidth.
- Main challenge: handling **outliers** in activations that can distort the quantization range.
- Modern methods (e.g., **SmoothQuant**, **LLM.int8()**) mitigate outlier effects to maintain accuracy.

Distributed Training & Inference

Motivation

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

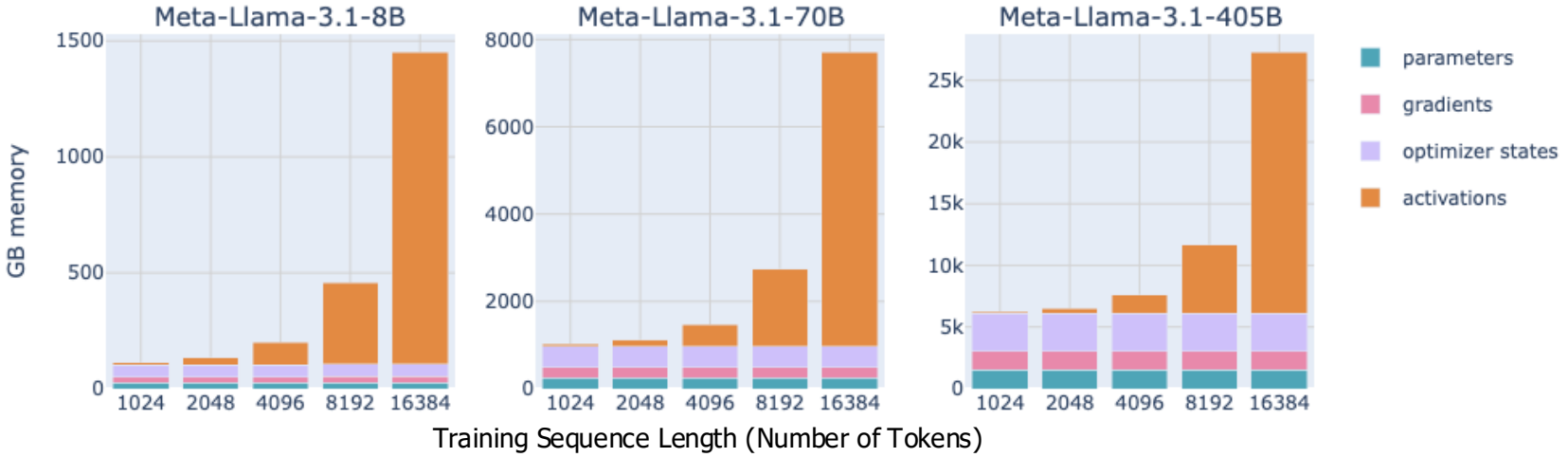
(batch size=1, ignoring the KV cache and optimizer states)

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

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

Where did all the memory go?

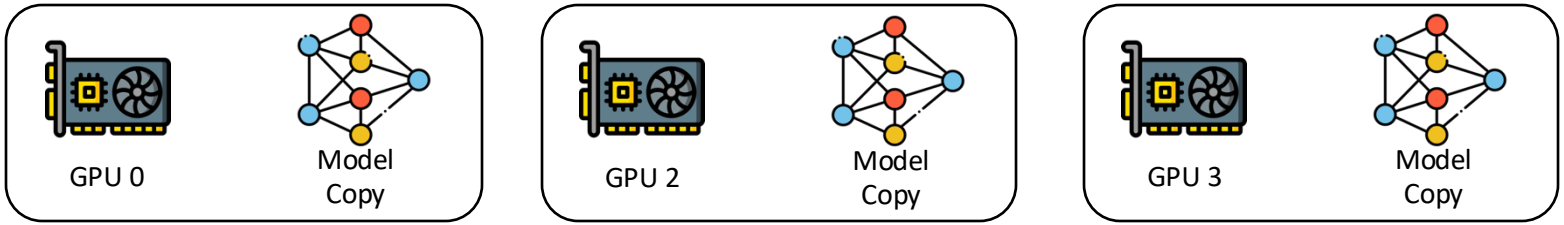
Longer sequences require much more memory in training!



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

Distributed Training and Inference

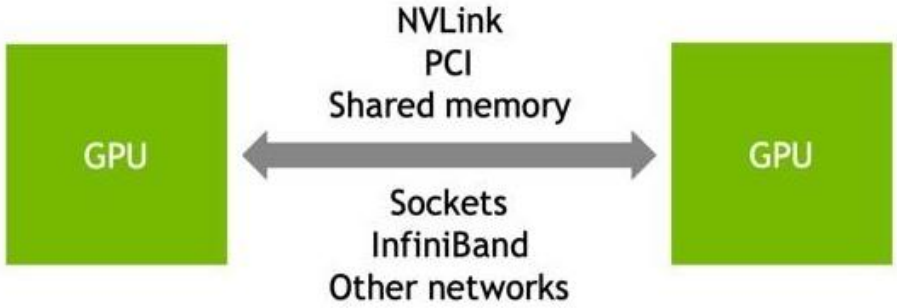
1. Naïve Data Parallelism
2. Sharding Optimizer States (ZeRO, FSDP)
3. Model Parallelism (Tensor Parallelism, Pipeline Parallelism)



Brief on GPU Operations for Communicating Tensors

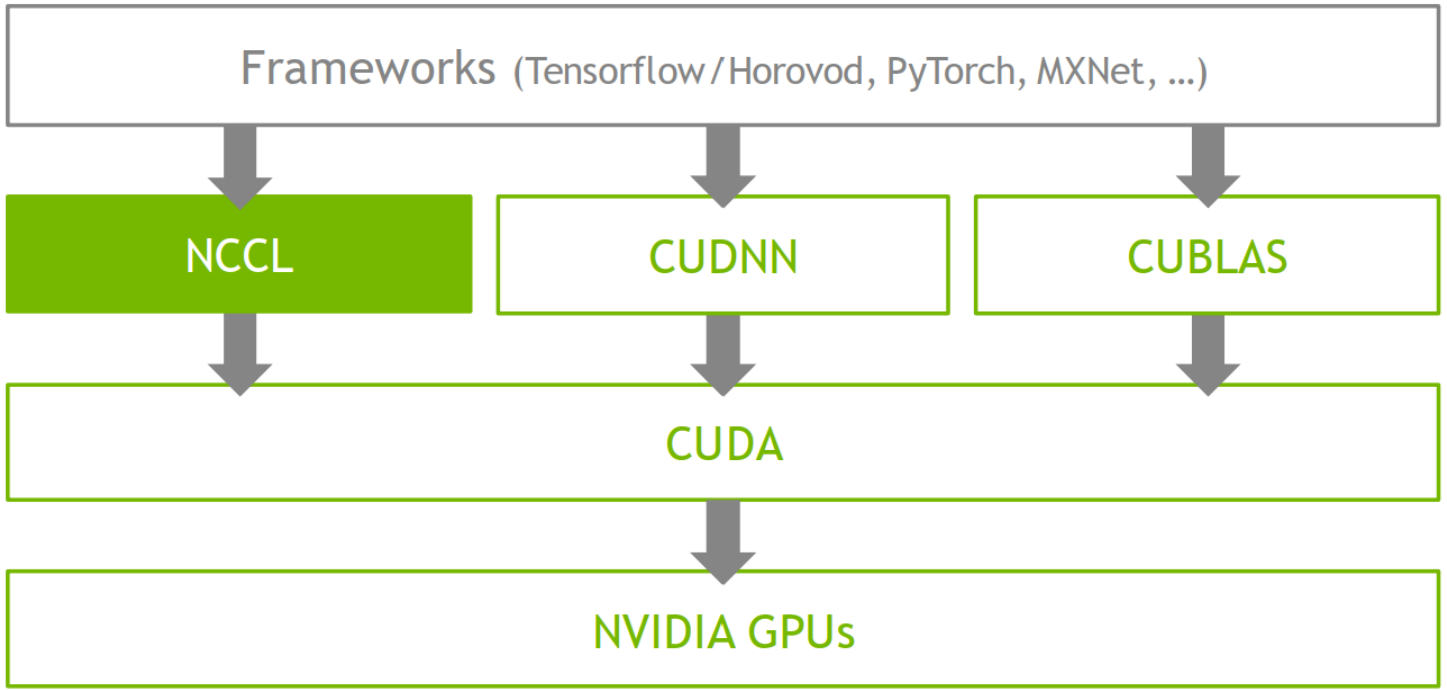
NCCL: NVIDIA Collective Communication Library

- 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



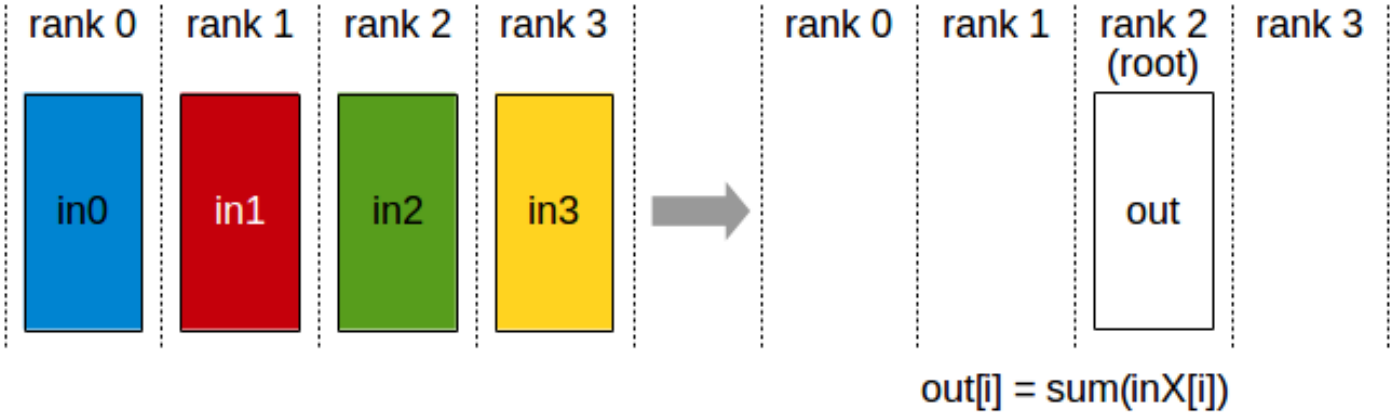
Documentation: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html#
Github: https://github.com/NVIDIA/nccl

NCCL: NVIDIA Collective Communication Library



NCCL Operations: Reduce

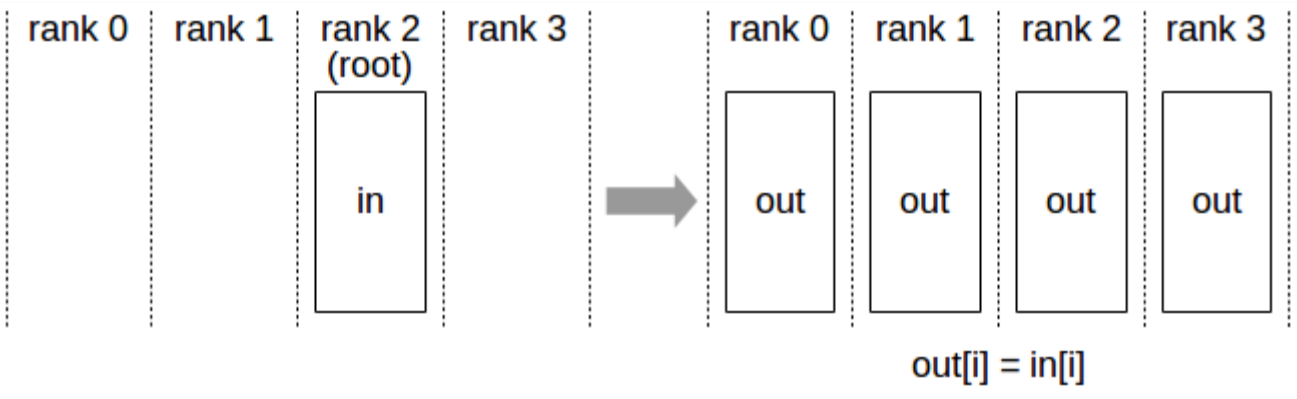
- Nvidia Collective Communications Library (NCCL) - A library developed to provide inter-GPU communications primitives (operations)
- Reduce: *Sums* over all *tensors* and stores it in a root GPU



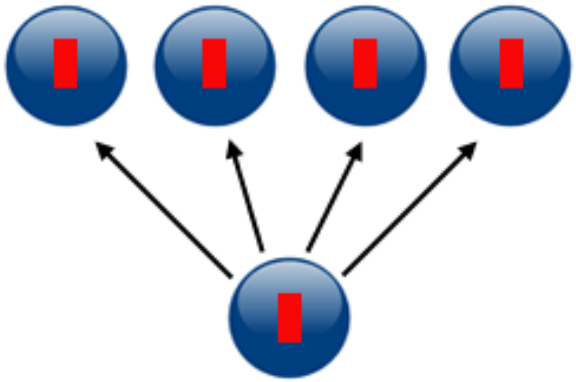
NCCL (NVIDIA Collective Communication Library): NVIDIA's high-performance library for multi-GPU and multi-node communication: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html

NCCL Operations: Broadcast

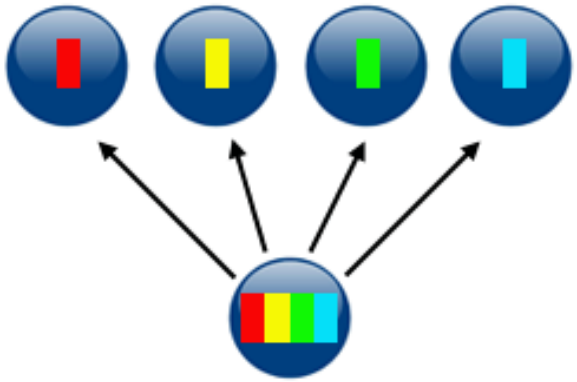
- Broadcast: Duplicates one tensor to all GPUs



Broadcast vs Scatter

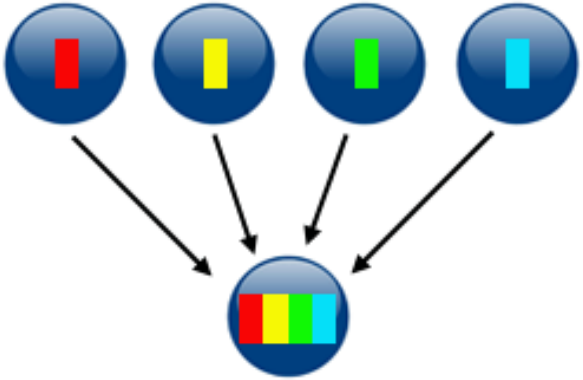


broadcast

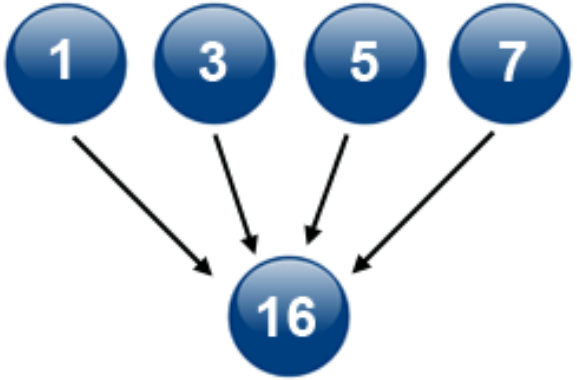


scatter

Gather vs Reduce



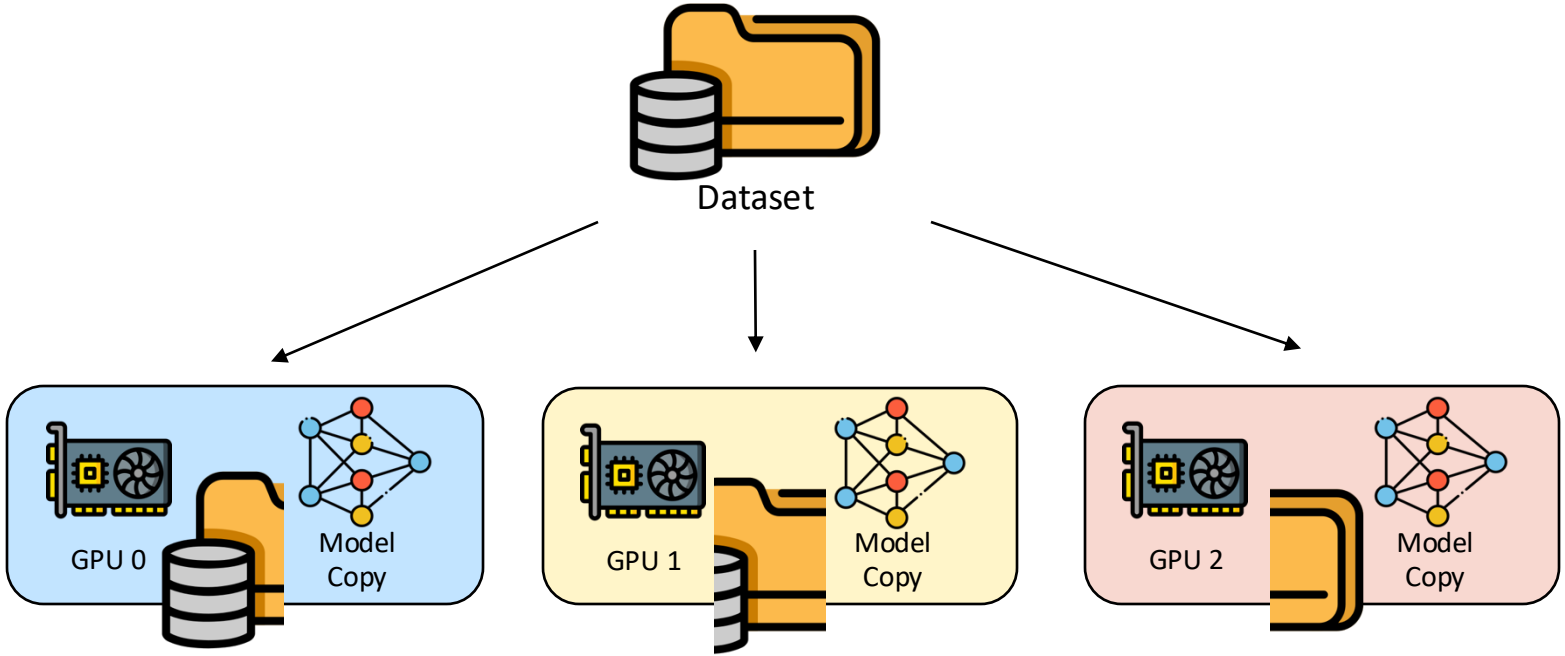
gather



reduction

Back to distributed training!

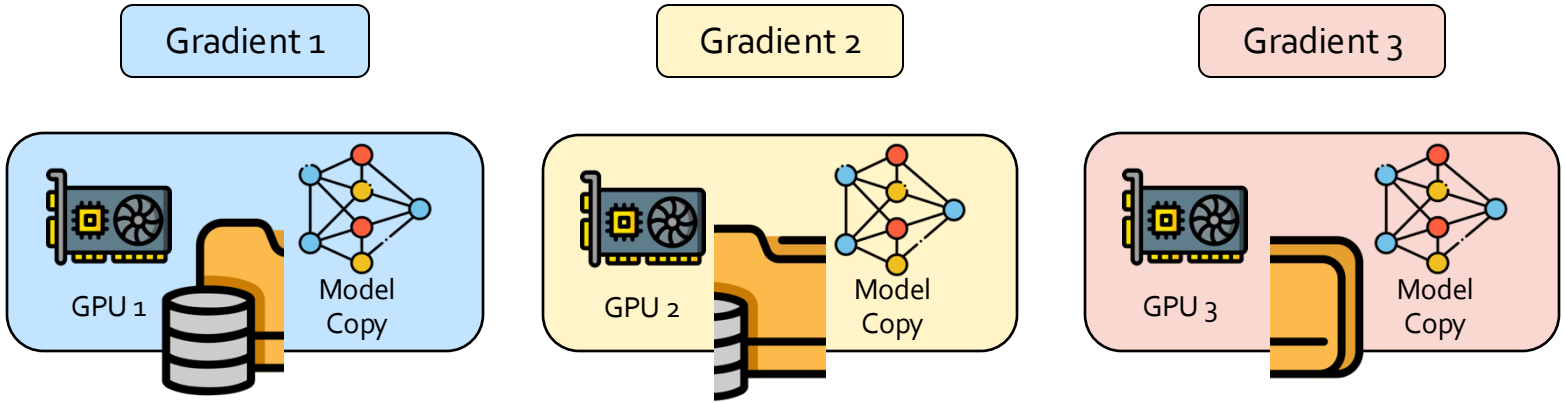
Naïve Data Parallelism



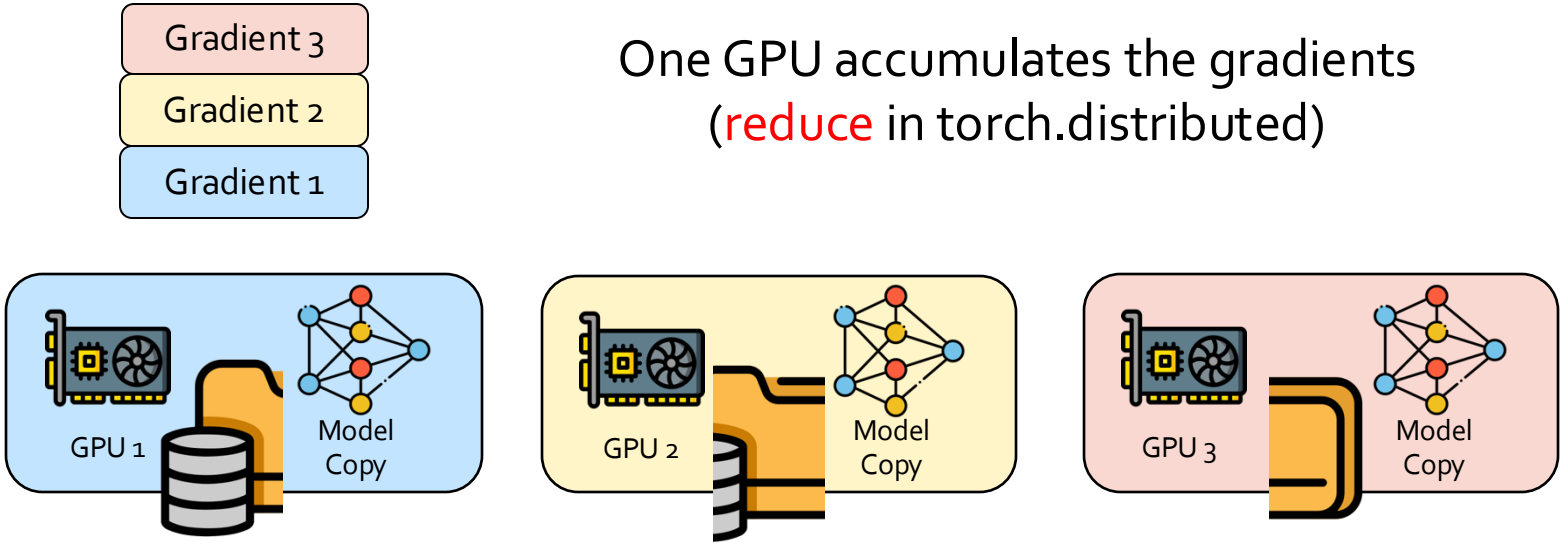
First, we want to shard the dataset and feed them into different GPUs
How do we update the parameters?

Naïve Data Parallelism

Each GPU compute gradient with a single shard of data



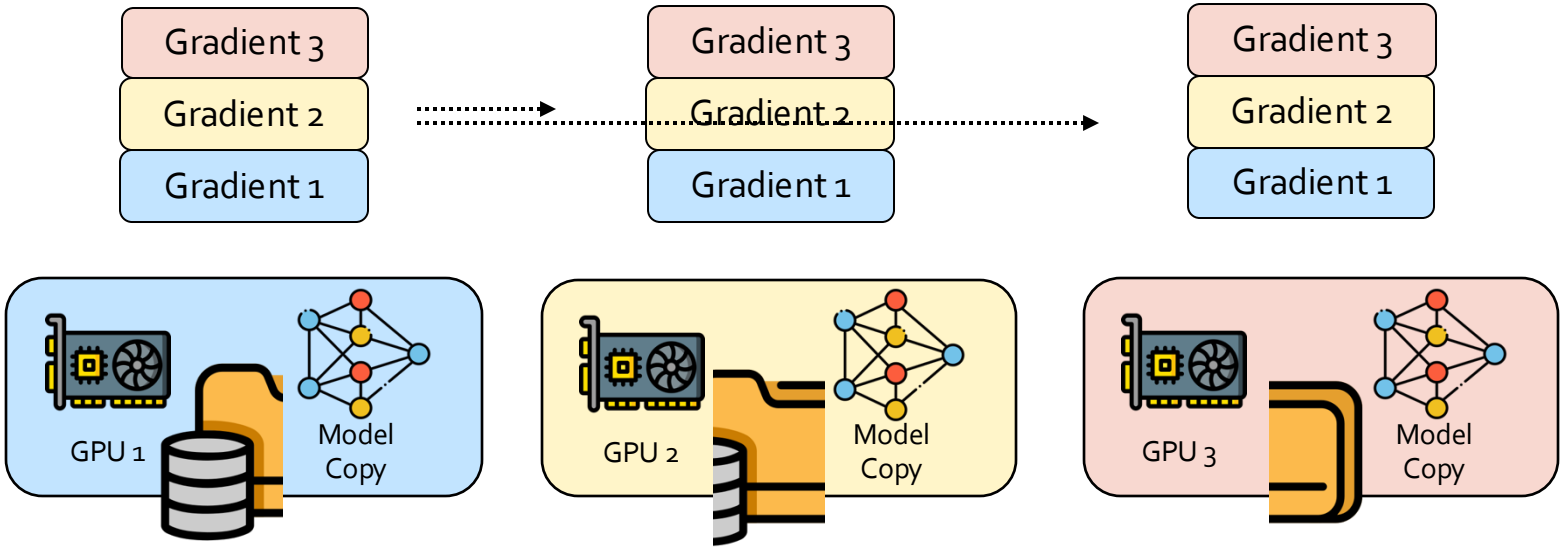
Naïve Data Parallelism



https://docs.pytorch.org/docs/stable/distributed.html

Naïve Data Parallelism

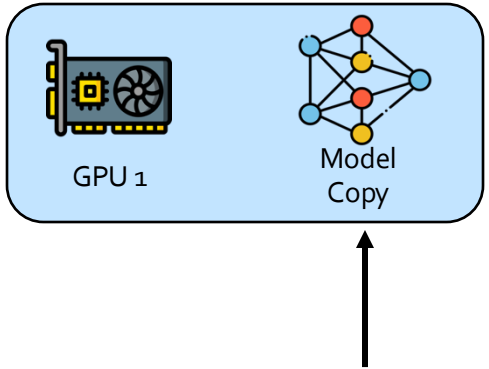
And send the accumulated gradient to all other GPUs (broadcast in torch.distributed)



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!

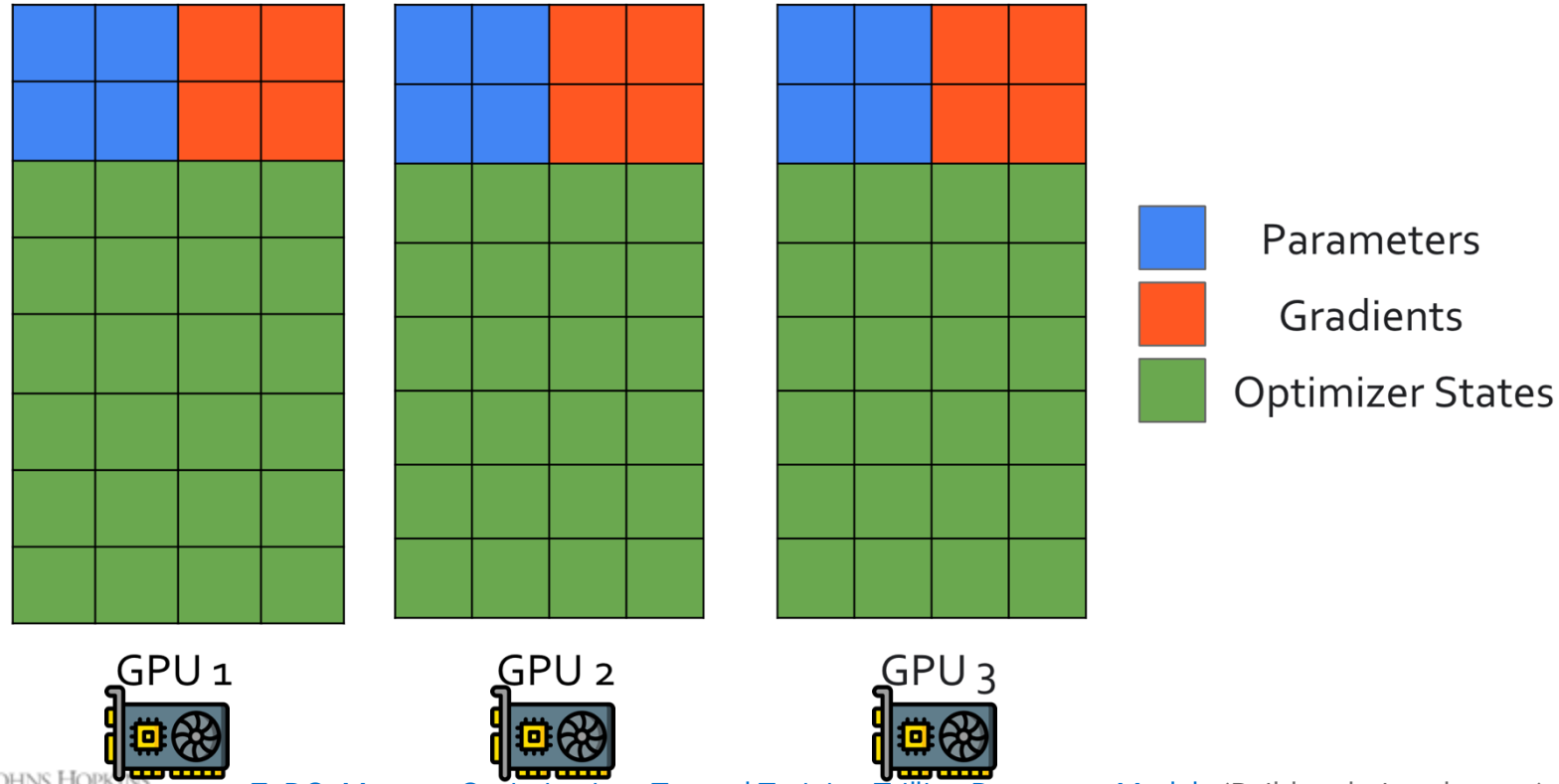
Naïve DP: Summary

- 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!



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

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

Note: "Stage x" means a **variant**, not a step!

- Result: Dramatically reduced memory use. Integrated to **DeepSpeed** and **PyTorch FSDP.**

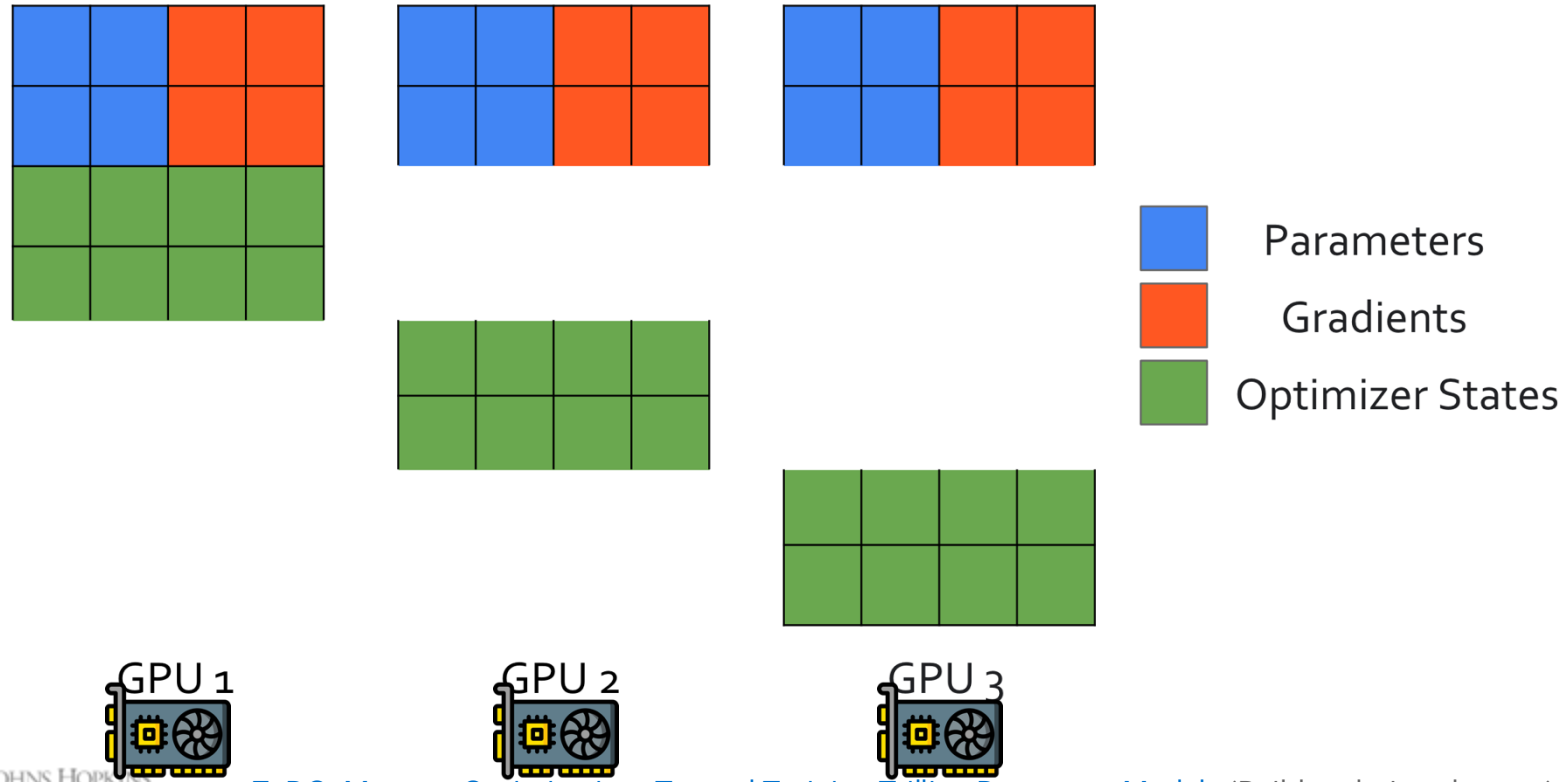
Stage	What's Sharded	Approx bytes / param	Memory / GPU (7.5 B params)
Naïve DP	None	16 B	**120 GB**
ZeRO-1	Optimizer states	≈ 6–7 B	**50 GB**
ZeRO-2	+ Gradients	≈ 5 B	**40 GB**
ZeRO-3	+ Parameters	≈ 2 B	**15 GB**

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

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 **not sharded** (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

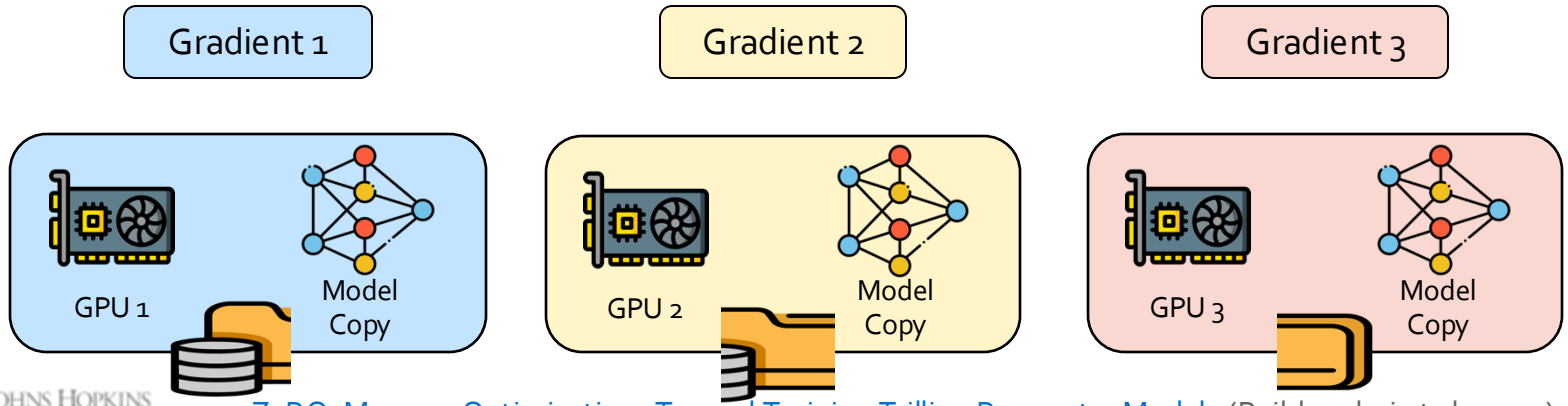


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

ZeRO Stage 1: How it works

Gradient computation

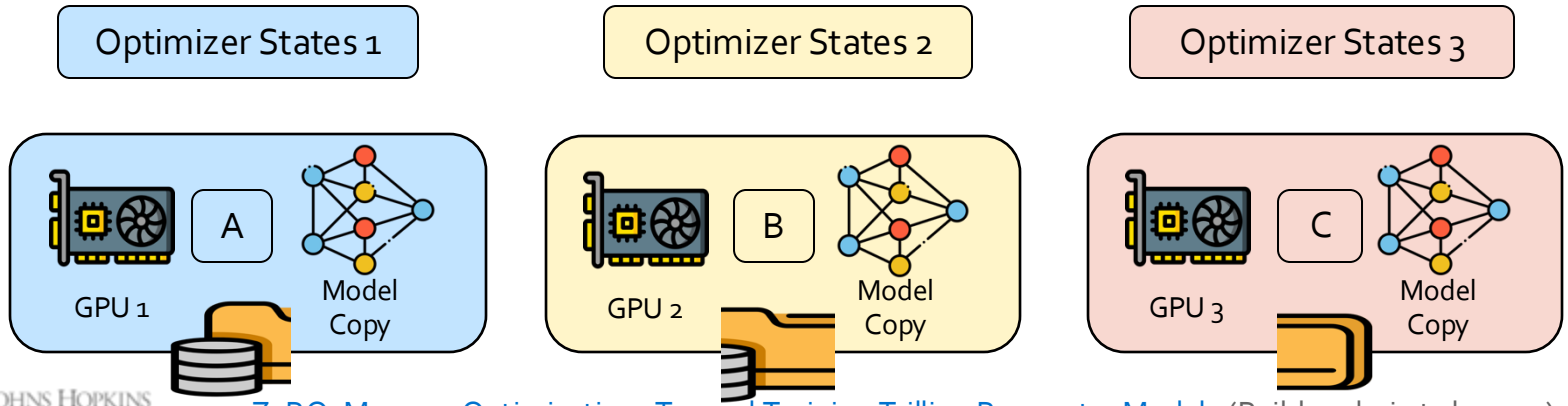
Each GPU compute gradient with a single shard of data.
They're later communicated to other GPUs to accumulate.
(The same as naïve DP)



ZeRO Stage 1: How it works

Upon one round of update, we'll have optimizer states.
Assuming that

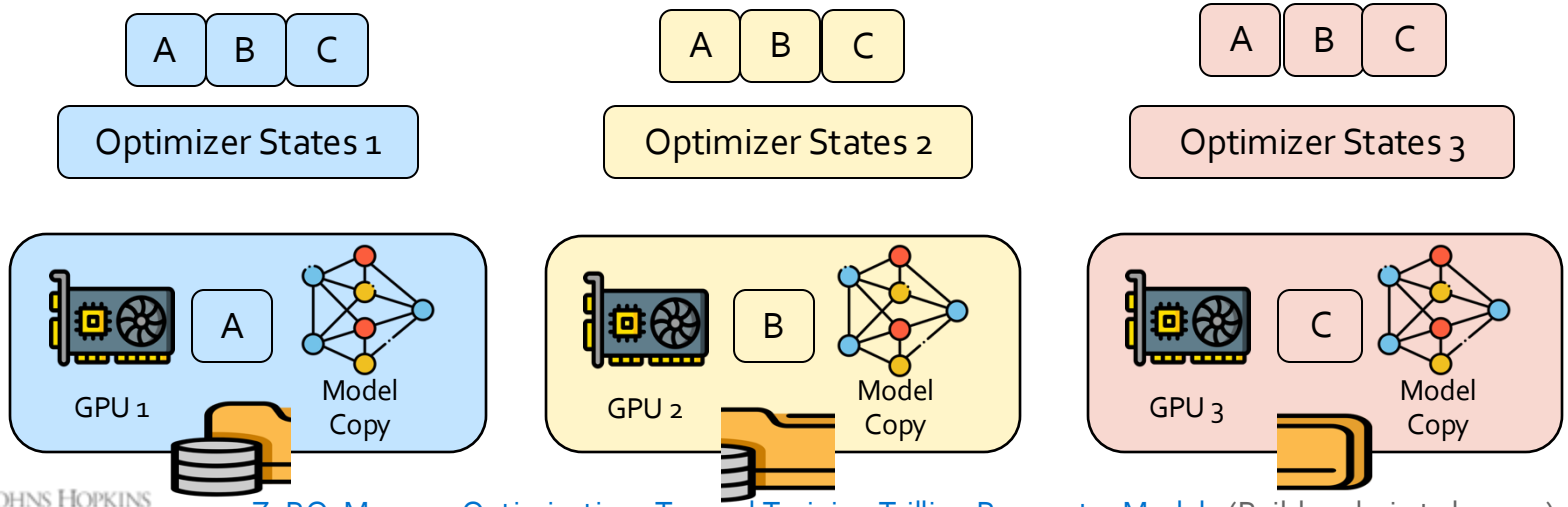
- GPU1 stores optimizer states for parameters A,
- GPU2 stores optimizer states for params B,
- GPU3 stores optimizer states for params C



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

ZeRO Stage 1: How it works

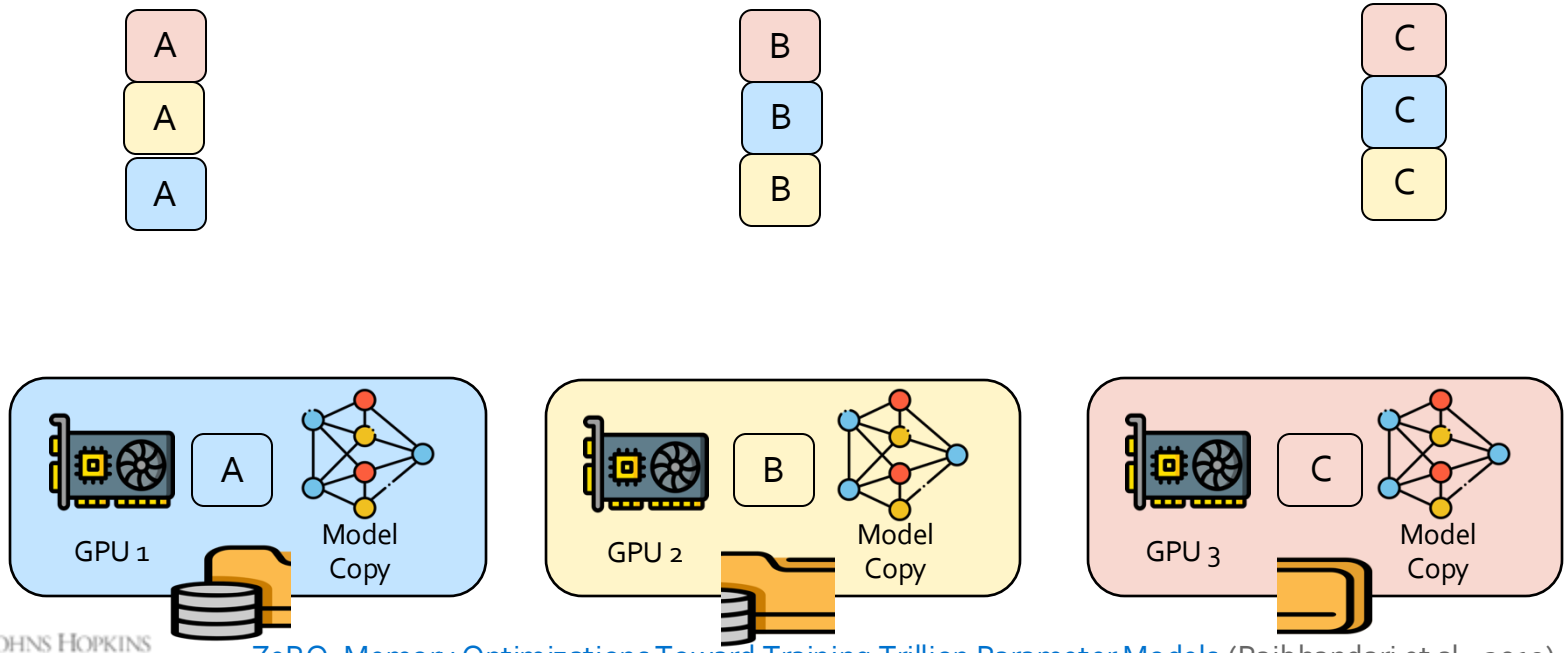
Split / shard the optimizer states into 3 parts!



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)



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

ZeRO Stage 1: How it works

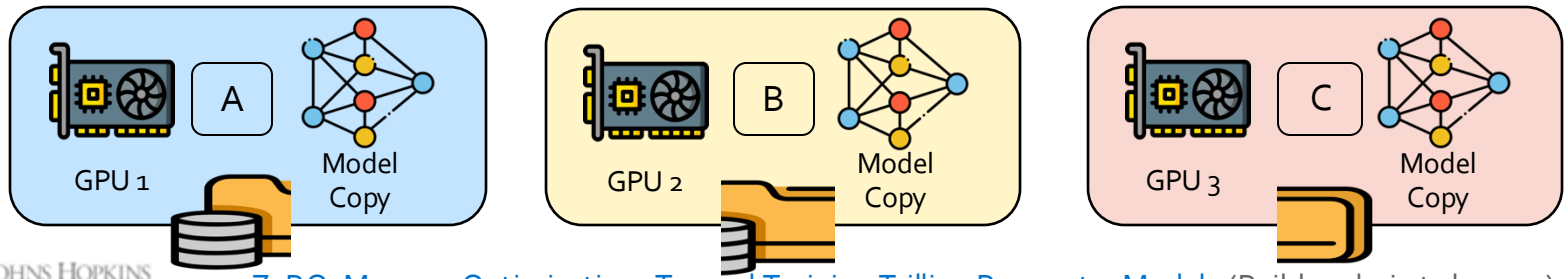
GPU1 : update params A;
GPU2: Updates Params B;
GPU3: updates params C.

GPU1 can only update params A since it only stores optimizer states of params A.

Updated A

Updated B

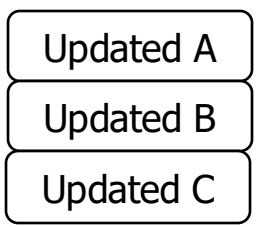
Updated C

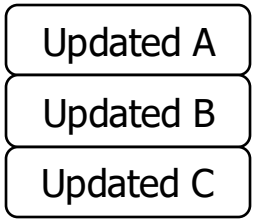


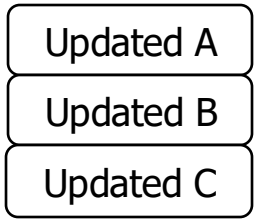
ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

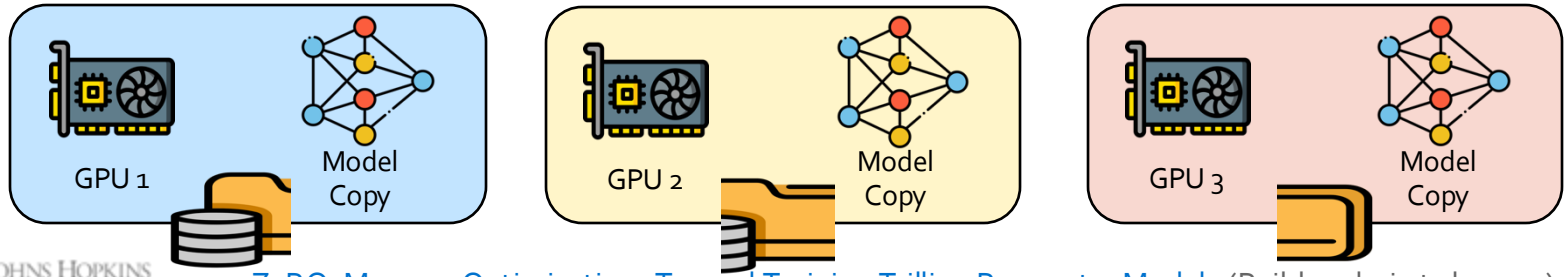
ZeRO Stage 1: How it works

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







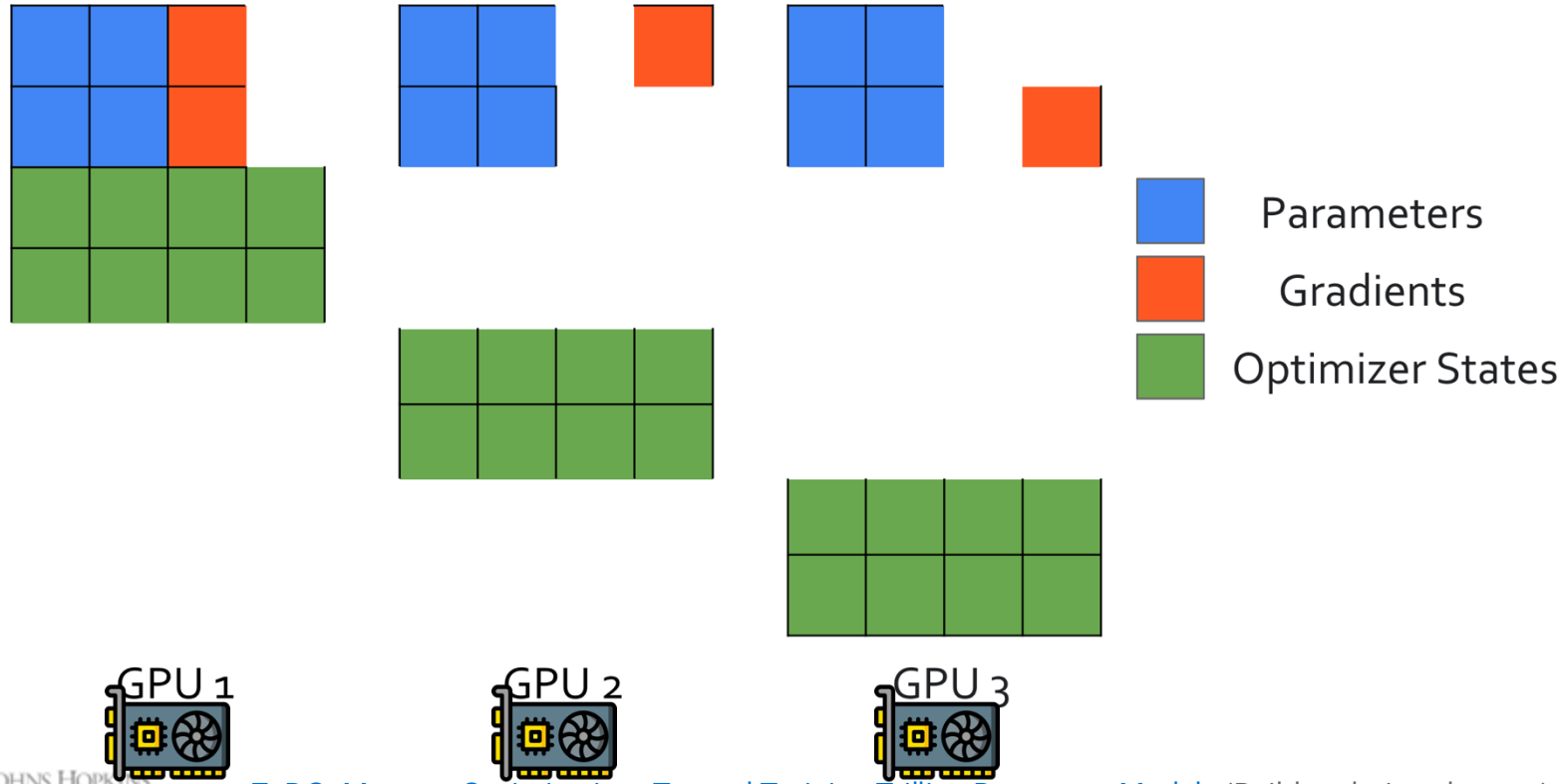


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

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)

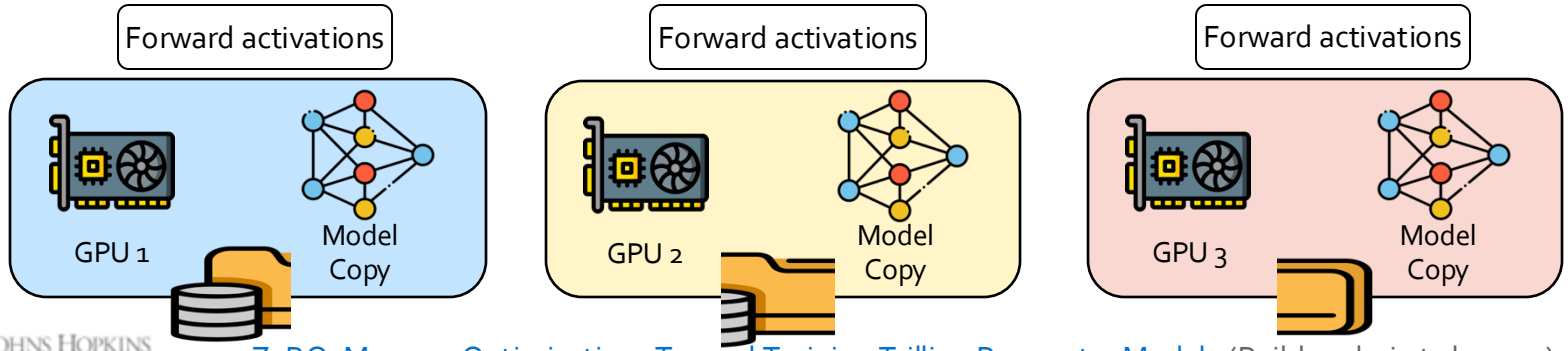
ZeRO Stage 2: Sharding Gradients



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

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.

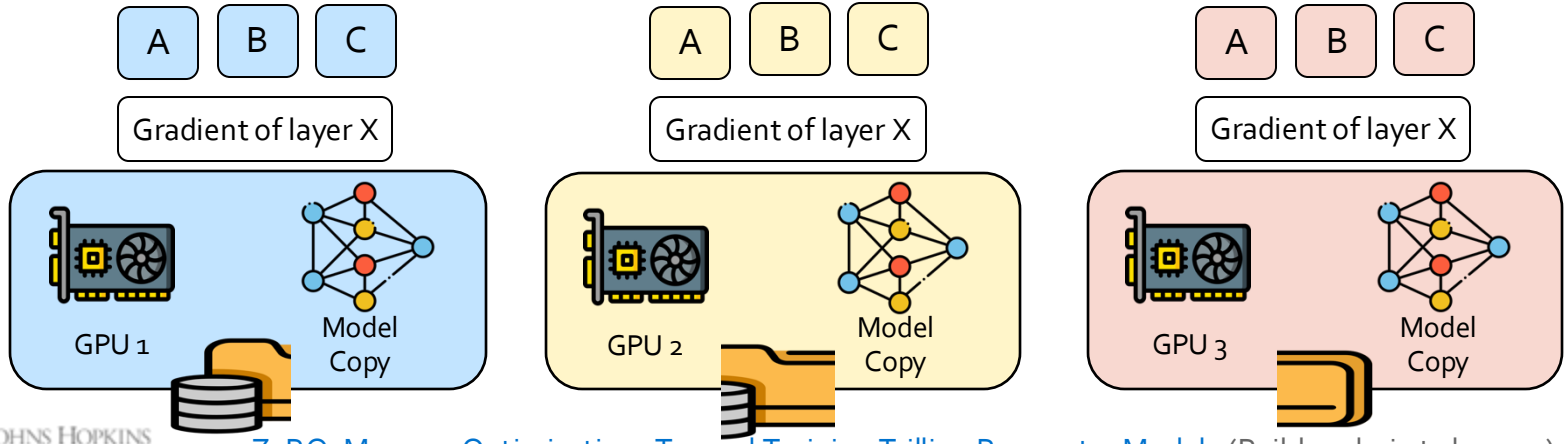


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

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.

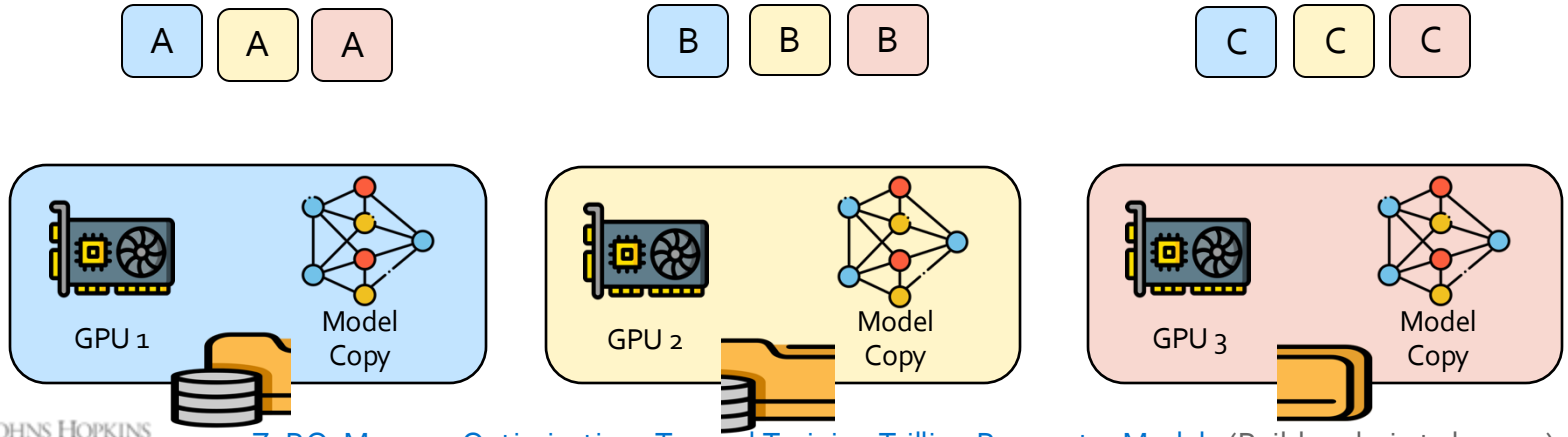


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

ZeRO Stage 2: How it works

Backward pass:

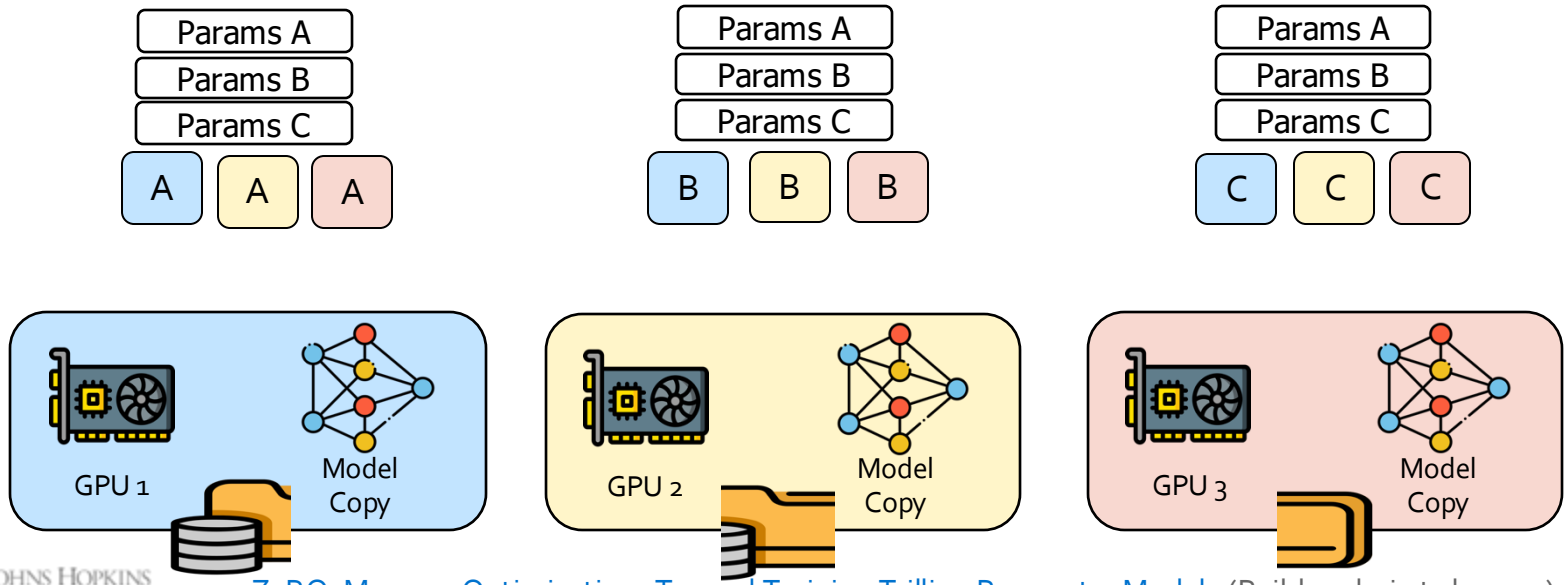
- Each GPU computes gradients for its local batch.
- Instead of storing *all* gradients, each GPU keeps **only its shard** of the gradient.
- As soon as the gradients of layer X are computed during backprop, shard them across GPUs.
- This uses reduce-scatter operation **incrementally during backpropagation**,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

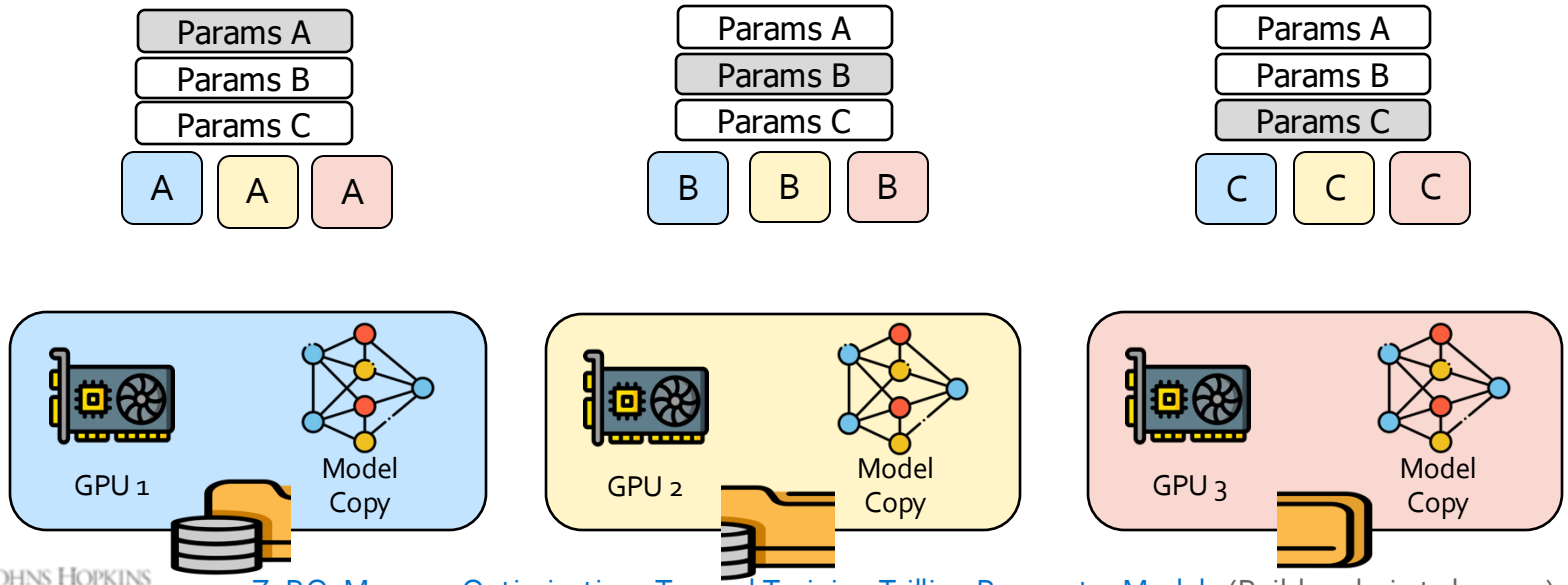


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

ZeRO Stage 2: How it works

Optimizer step:

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

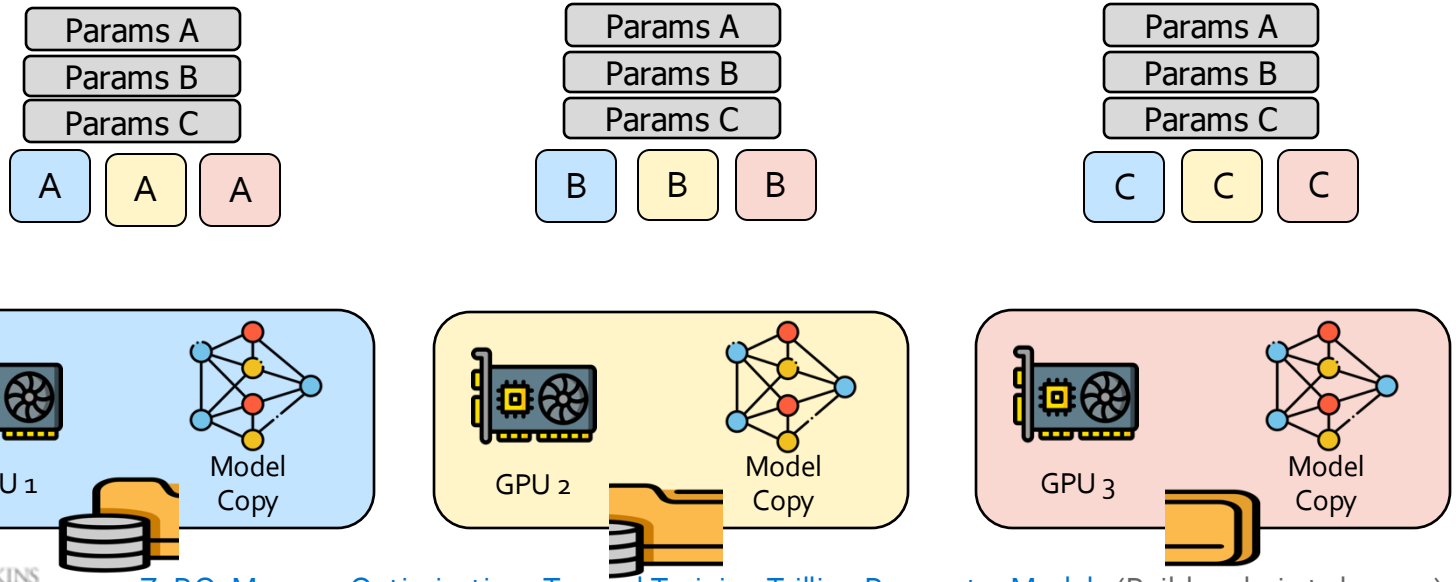


ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

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)**



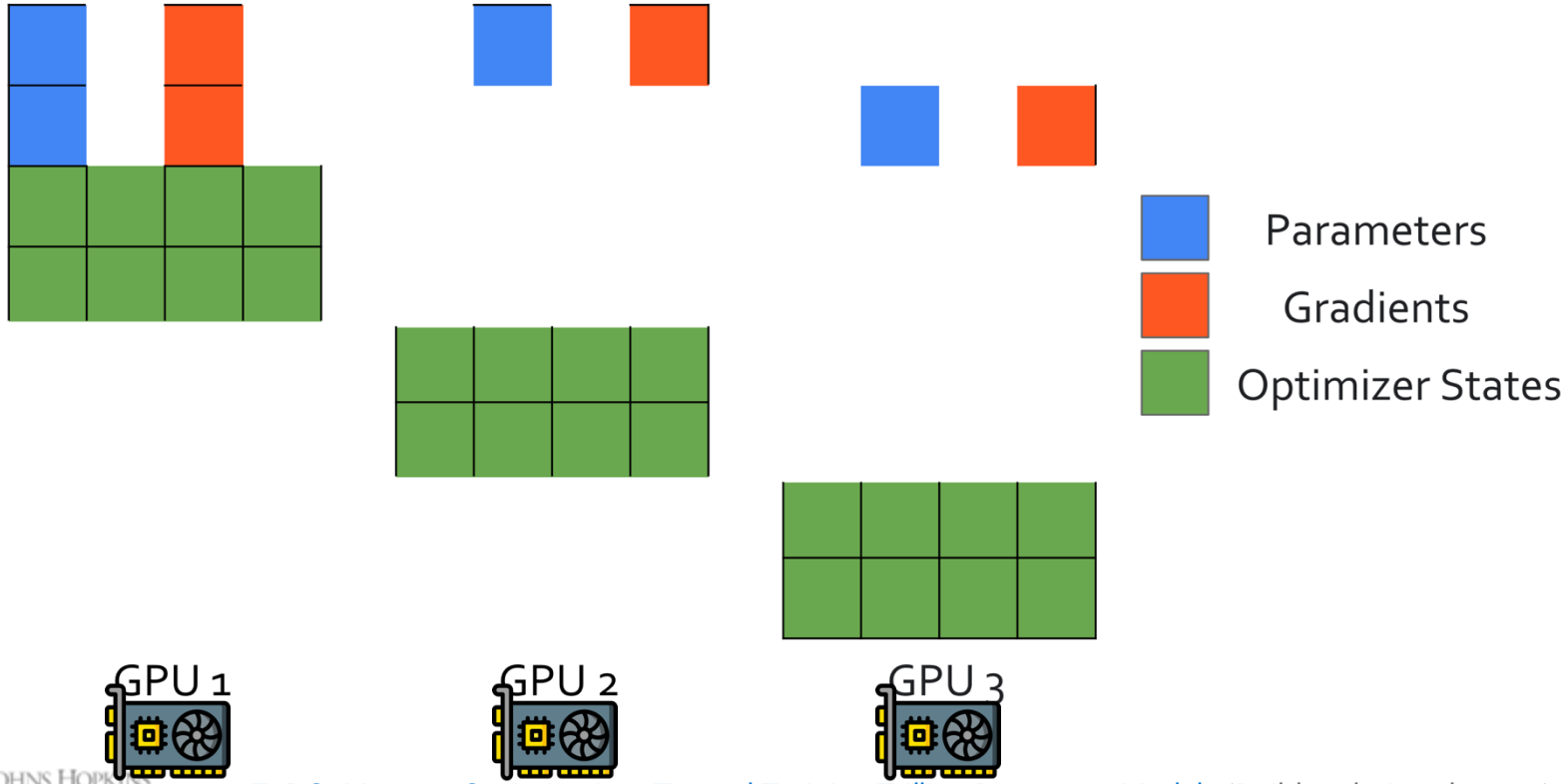
ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

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	≈ 6–7 B	**50 GB**
ZeRO-2	+ Gradients	≈ 5 B	**40 GB**

ZeRO-3 (aka FSDP): Shard Everything!



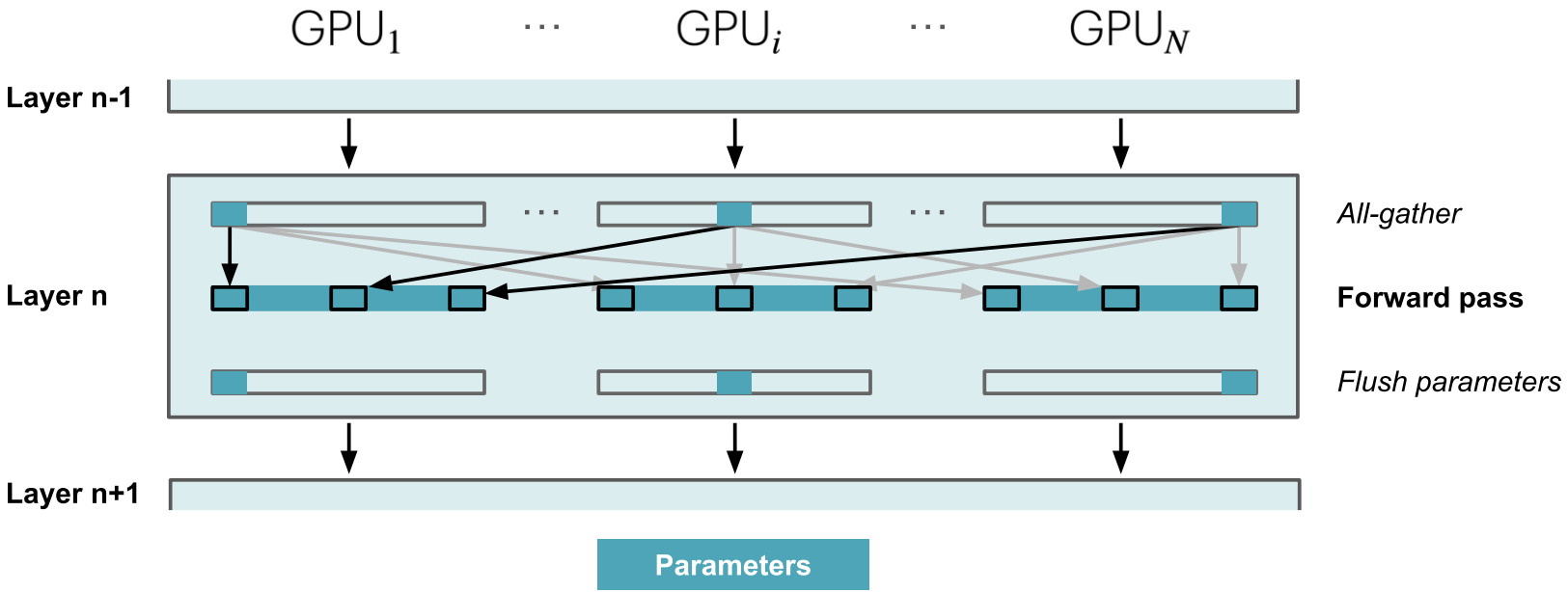
ZeRO: Memory Optimizations Toward Training Trillion Parameter Models (Rajbhandari et al., 2019)

ZeRO Stage 3

- Shards **everything** — parameters, gradients, and optimizer states — across GPUs.
- Provides **maximum memory savings (~1/N per GPU)**, enabling extremely large models.

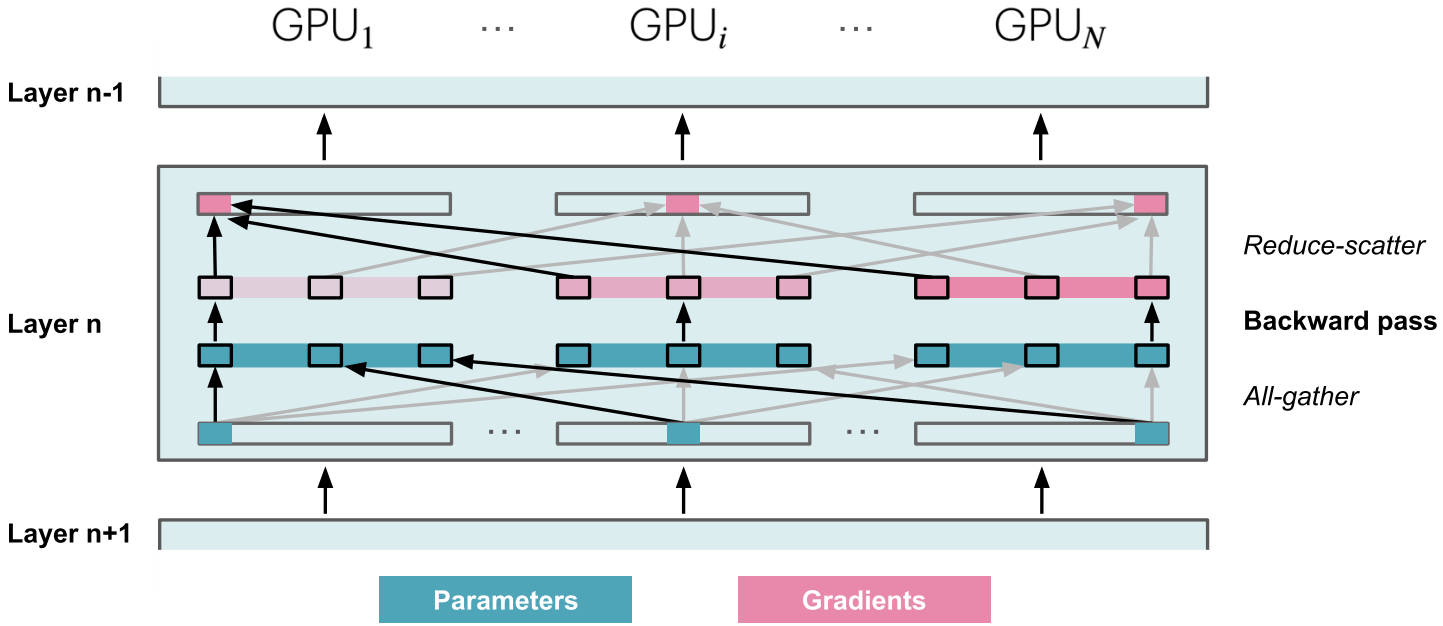
ZeRO Stage 3: How it works (simplified)

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.



ZeRO Stage 3

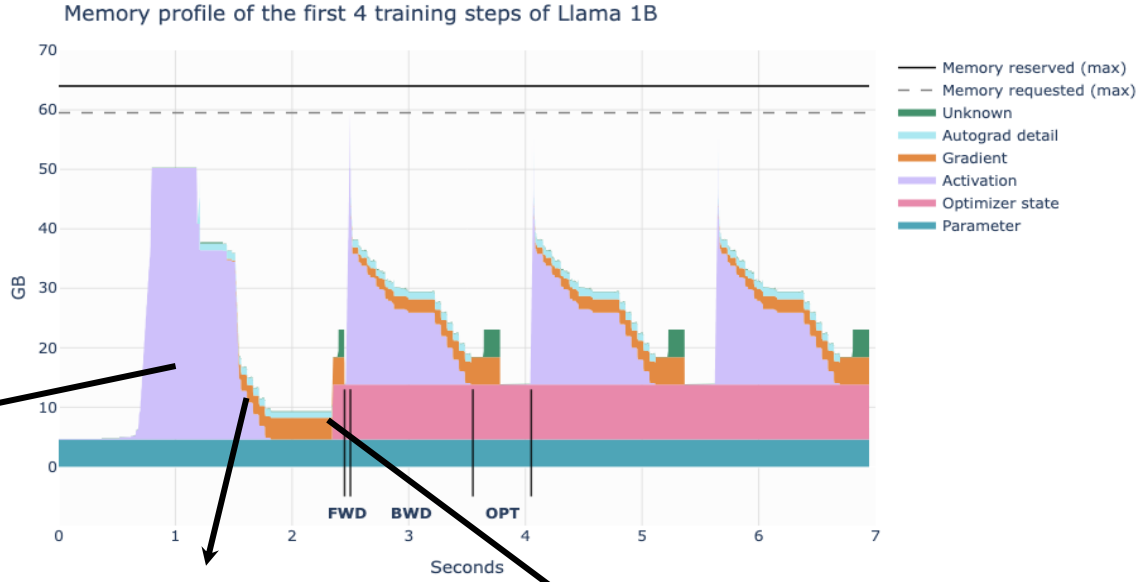
- Shards **everything** — parameters, gradients, and optimizer states — across GPUs.
- Provides **maximum memory savings (~1/N per GPU)**, enabling extremely large models.
- Comes with **higher communication cost**, since parameters must be gathered repeatedly during training.

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.

Memory consumption is not static

- You can use **PyTorch profiler** helps visualize GPU memory usage across a training.



(1) During the forward step, the activations occupy most of the memory

(2) As the backward pass propagates, the stored activations used to compute the gradients are progressively cleared. But gradients gets more memory.

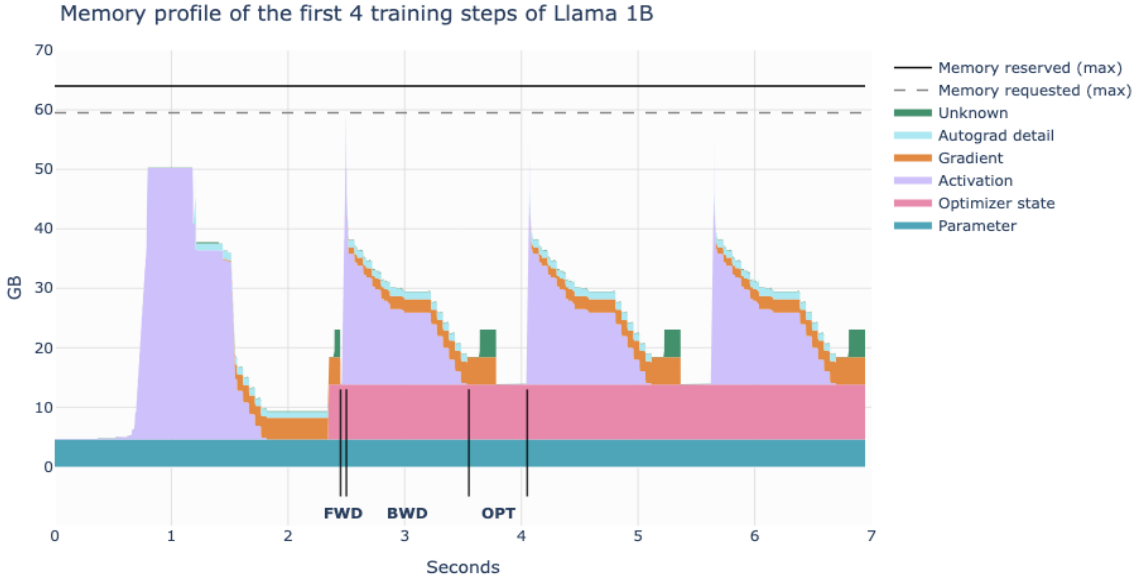
(3) We perform optimization, during which we need all the gradients, and then update the optimizer states before we start the next forward pass.

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

Memory consumption is not static

- You can use **PyTorch profiler** helps visualize GPU memory usage across a training.

This also explains why your training in step 1 may success and get OOM only in later steps!!



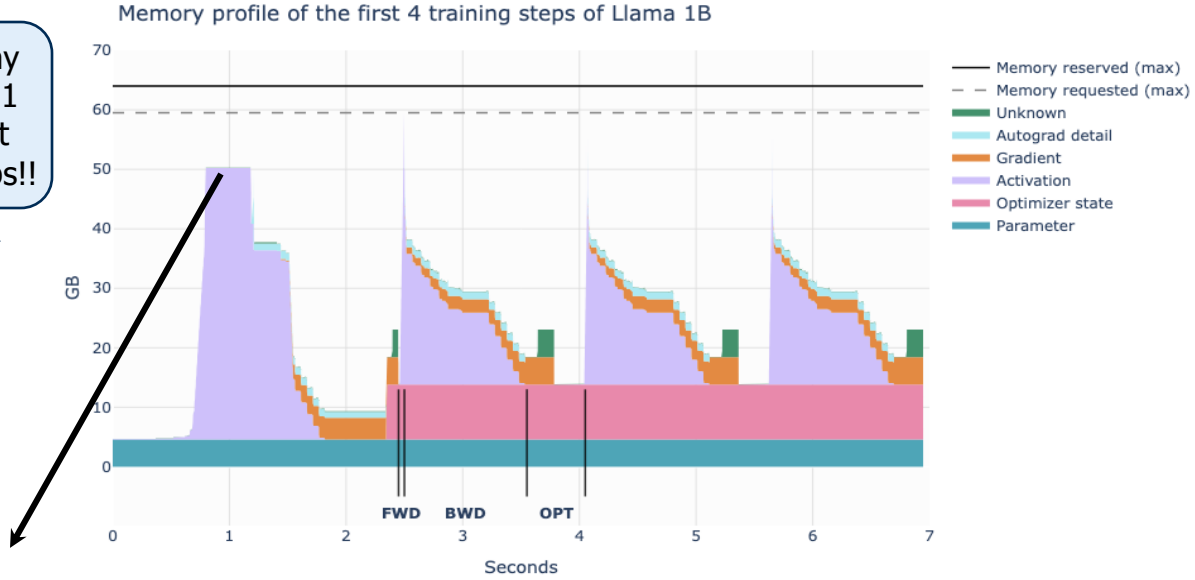
Because of the build-up of the optimizer states, the peak memory for step 2 onwards is higher!

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

Memory consumption is not static

- You can use **PyTorch profiler** helps visualize GPU memory usage across a training.

This also explains why your training in step 1 may success and get OOM only in later steps!!



the first step looks different: the activations increase quickly and then plateau for a while. Why?

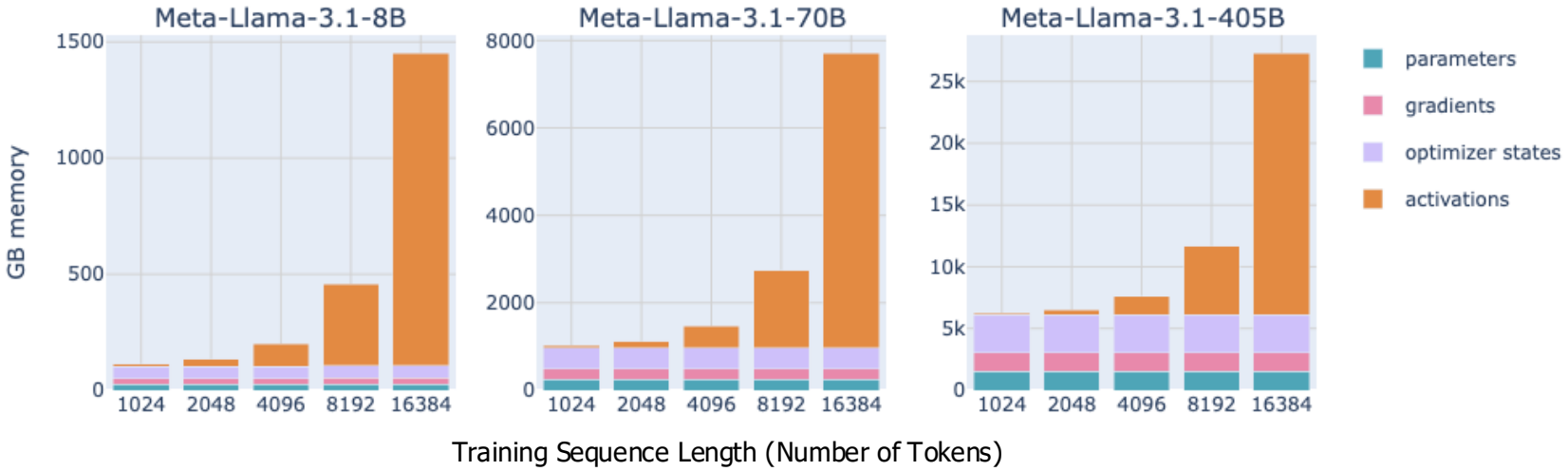
In this first step, the PyTorch caching allocator does a lot of prep work, preparing memory allocations so that the subsequent steps don't have to search for free memory blocks, which speeds them up.

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

Where did all the memory go?

So far, we dealt with the optimizer states

but what about the activations?

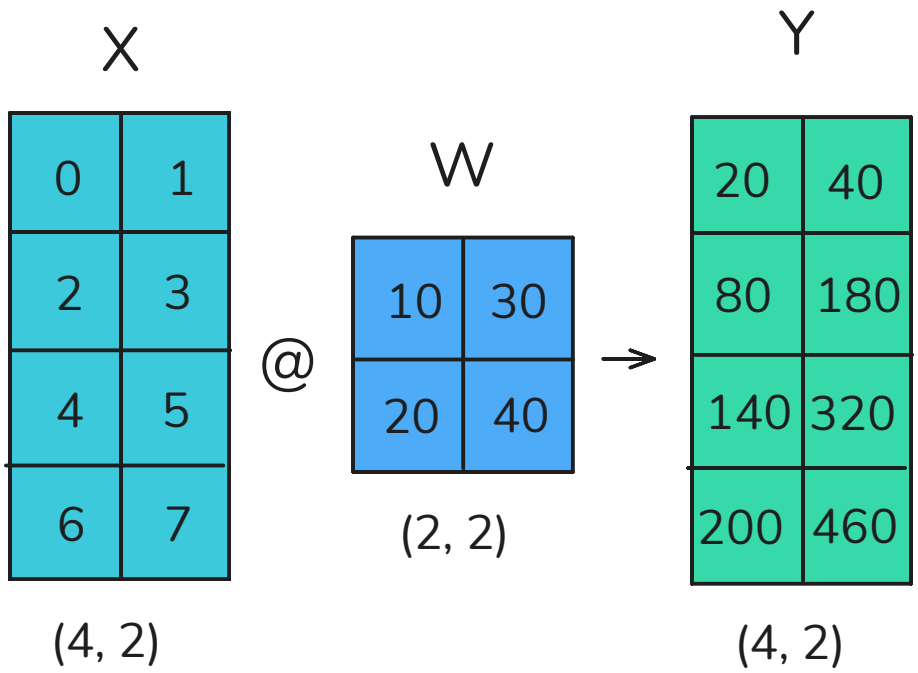


Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

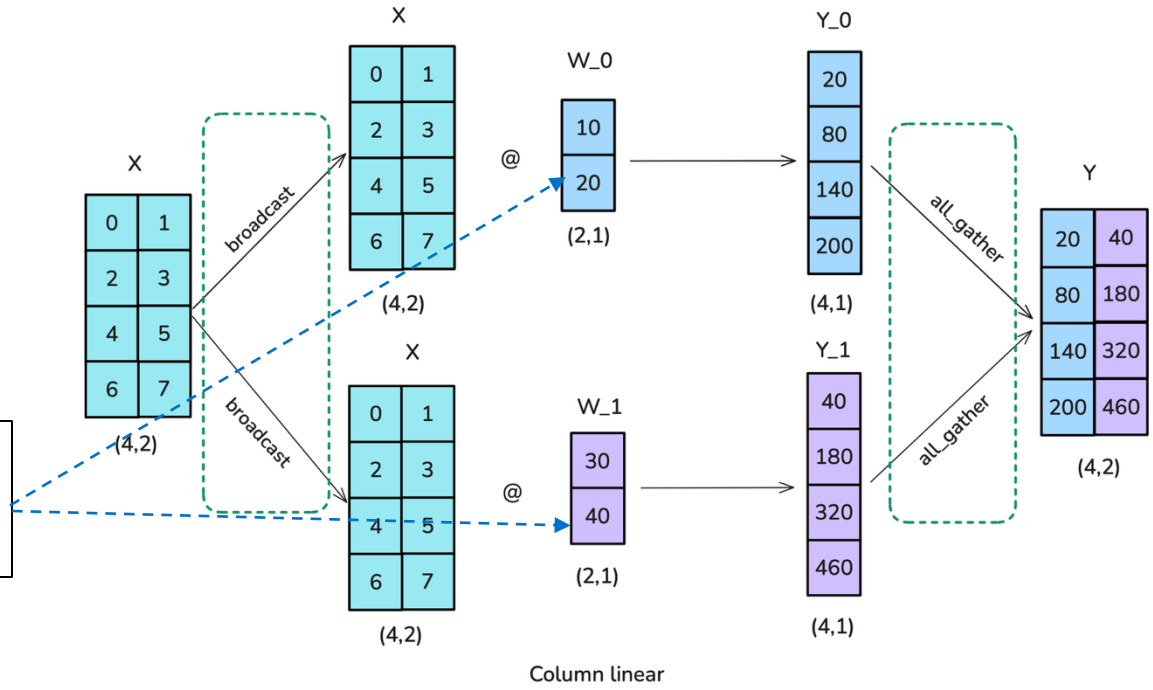
Tensor Parallelism

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

or into two rows (Row Parallelism)



Column-wise Tensor Parallelism

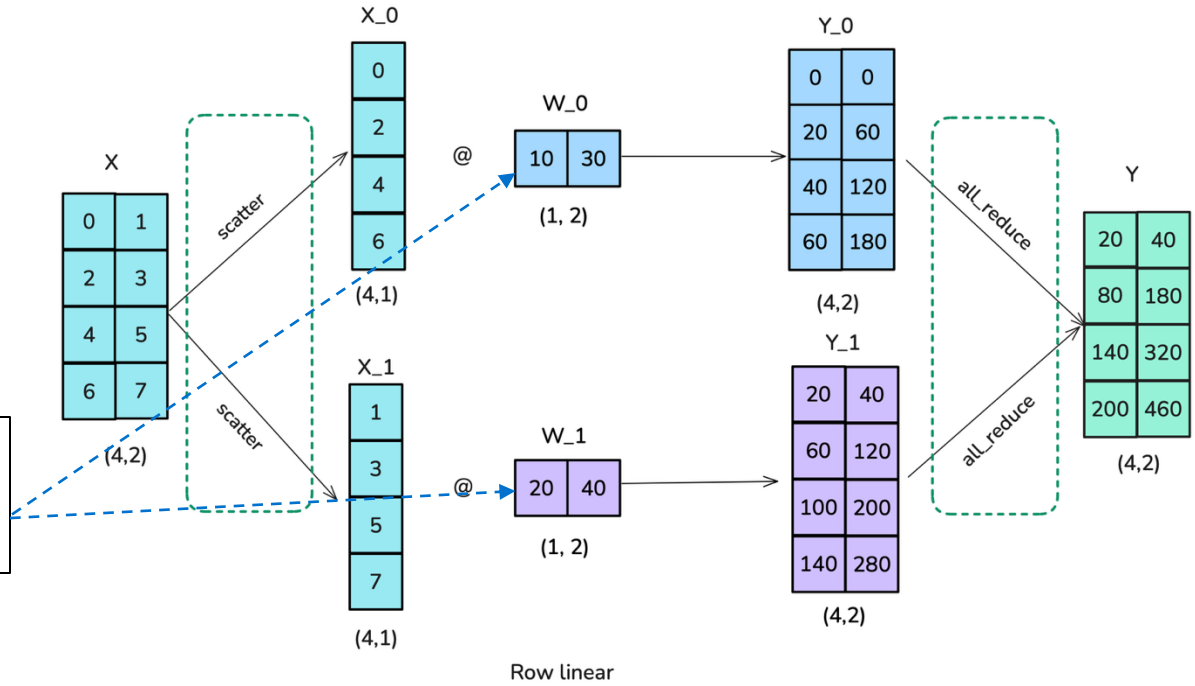


Cuts the weight matrix W into 2 columns

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

Row-wise Tensor Parallelism

Cuts the weight matrix W into 2 rows

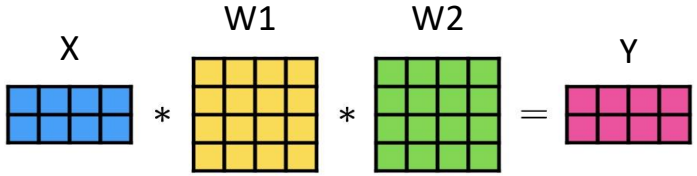


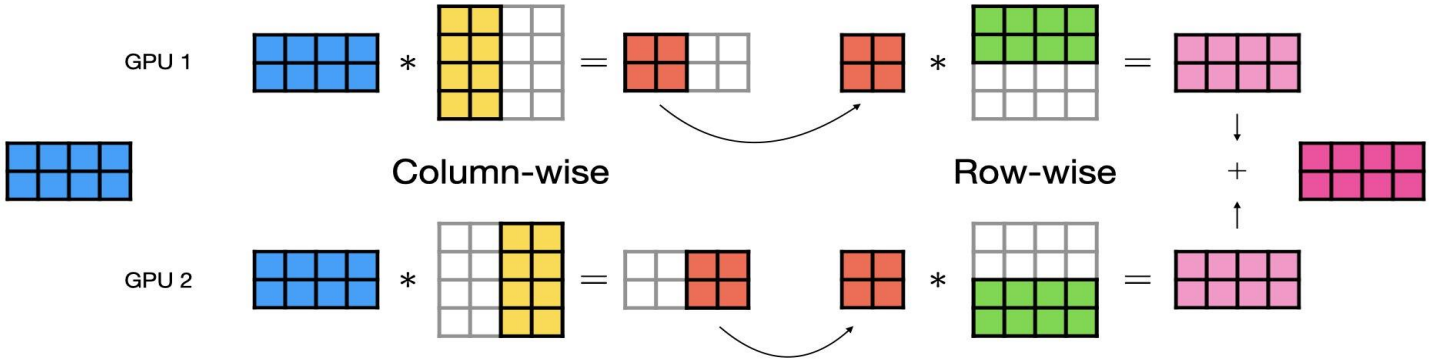
Source: https://nanotron-ultrascale-playbook.static.hf.space/dist/index.html

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

```
        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,
)
```

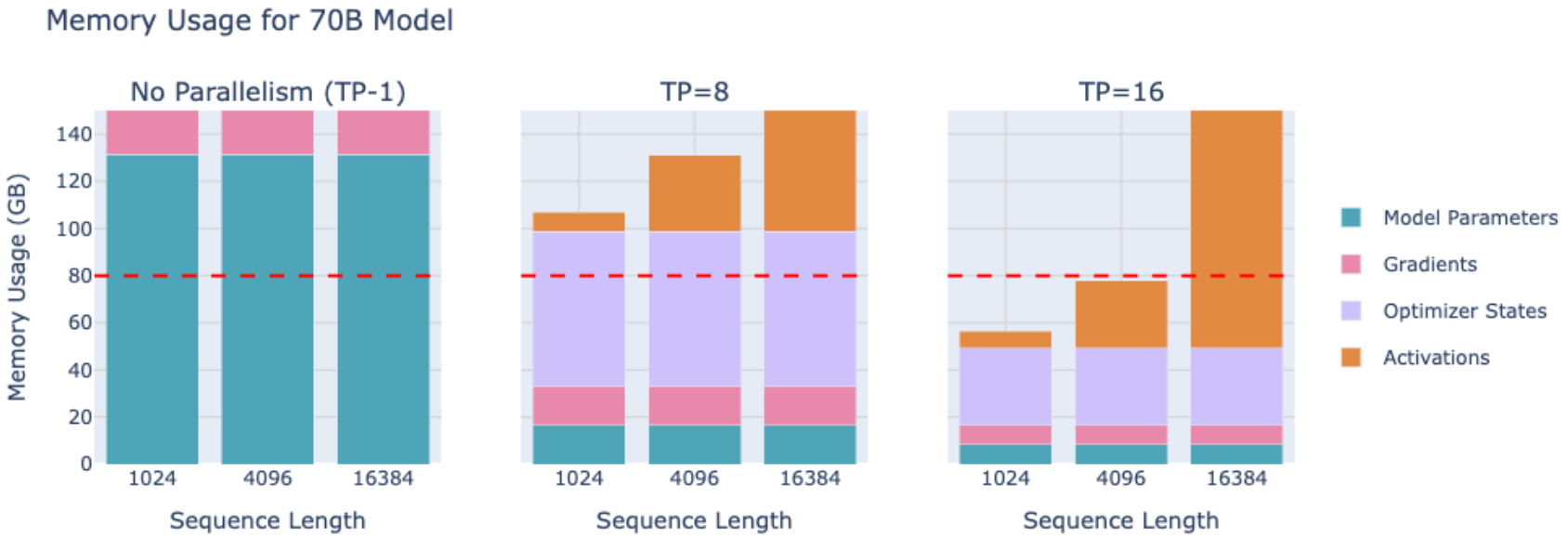
```
self.wo = RowParallelLinear(
    args.n_heads * self.head_dim,
    args.dim,
    bias=False,
    input_is_parallel=True,
    init_method=lambda x: x,
)
```

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

Summary so far

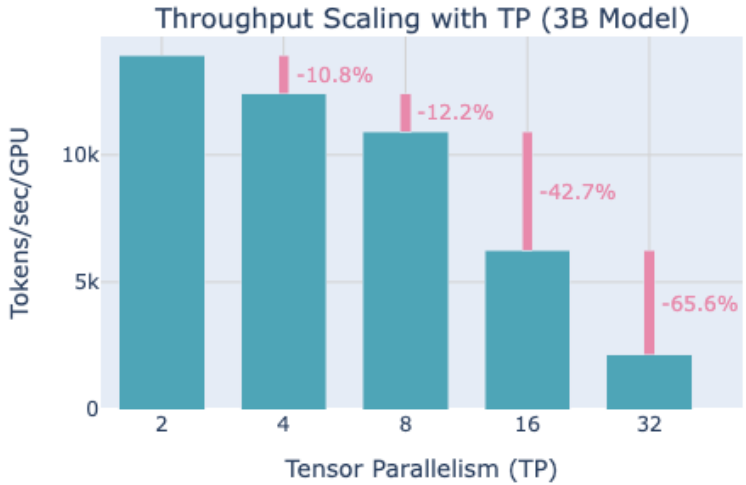
- Data Parallelism
  - Naïve Data Parallelism
  - NCCL Operations
    (reduce, all_reduce, reduce_scatter, broadcast, all_gather)
  - ZeRO-1, ZeRO-2, ZeRO-3
- Tensor Parallelism
  - Row-wise Tensor Parallelism
  - Column-wise Tensor Parallelism

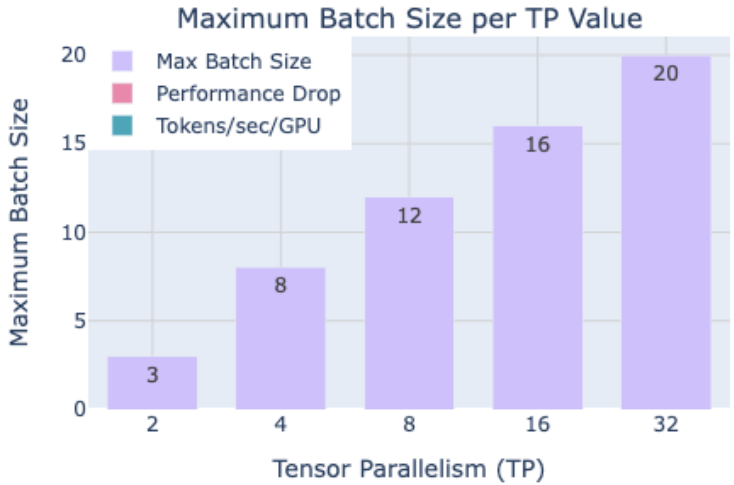
Tensor Parallelism



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

Throughput Scaling of Tensor Parallelism

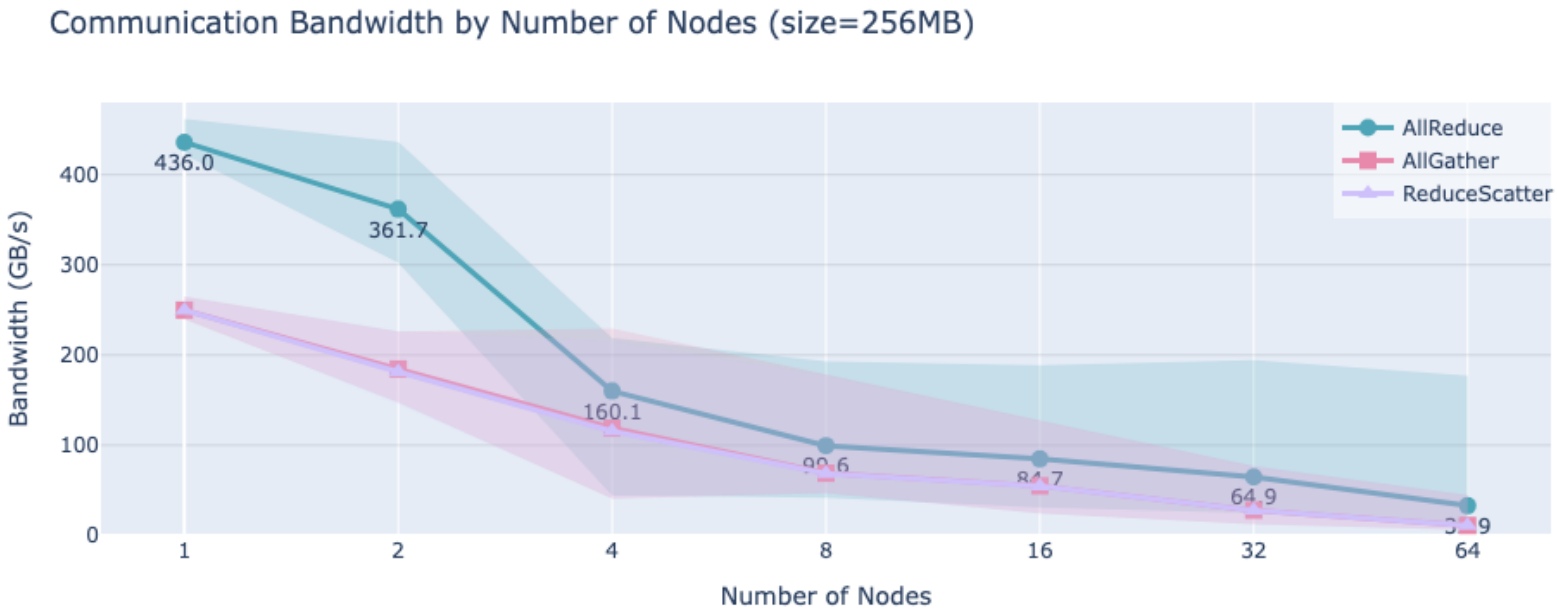




A large drop in throughput when scaling beyond 8 GPUs (one node)

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

Throughput Scaling of Tensor Parallelism



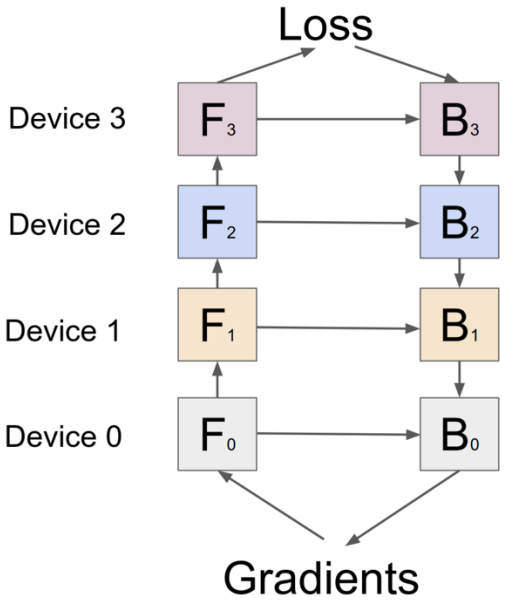
Throughput drops significantly once we go beyond one node!

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

Pipeline Parallelism: The idea

- Splitting the model's **layers** across multiple GPUs.
- Then pipelining the forward and backward passes so all GPUs stay busy.

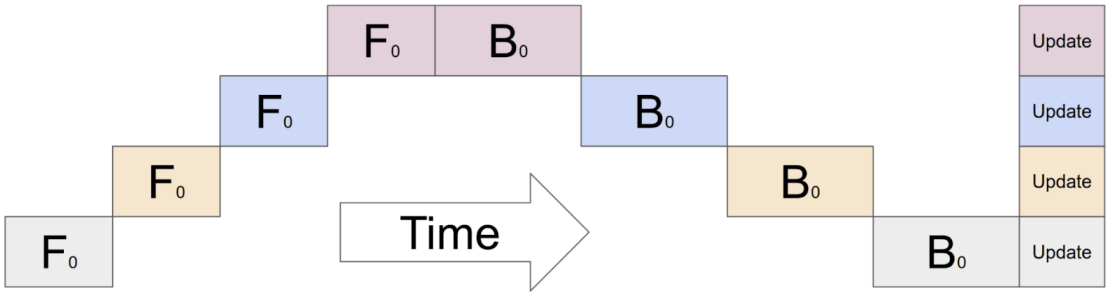
- For example:
  - [GPU0] layers 1–6
  - [GPU1] layers 7–12
  - [GPU2] layers 13–18
  - [GPU3] layers 19–24

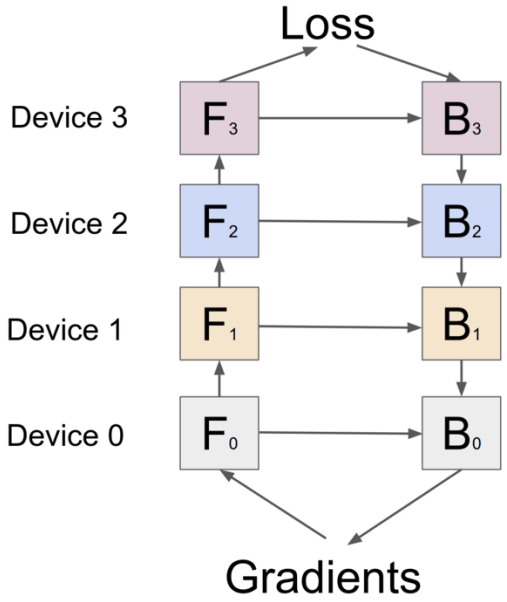


- **Execution flow:** outputs from one device's chunk are passed sequentially.
- **Benefit:** reduces per-device **memory load** by distributing model weights.

Pipeline Parallelism

- **Limitation:** sequential dependency causes **idle periods ("pipeline bubbles")** where some devices wait for others' results.



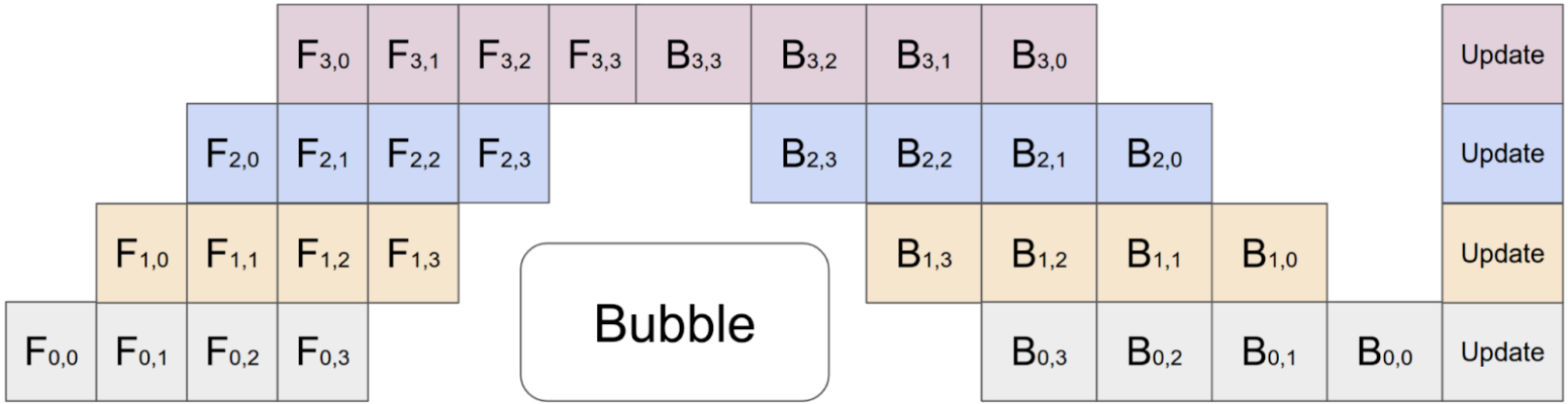


Each GPU is only working for 1/PP = ¼ of the time!

Idle/Work ratio = pp – 1 = 3

Pipeline Parallelism: Improvement

- Solution: **Microbatching** splits the global batch into smaller sub-batches processed in a staggered manner.
- **Gradients** from all microbatches are accumulated afterward.
- **Result:** better GPU utilization (smaller bubbles), though not fully eliminated.

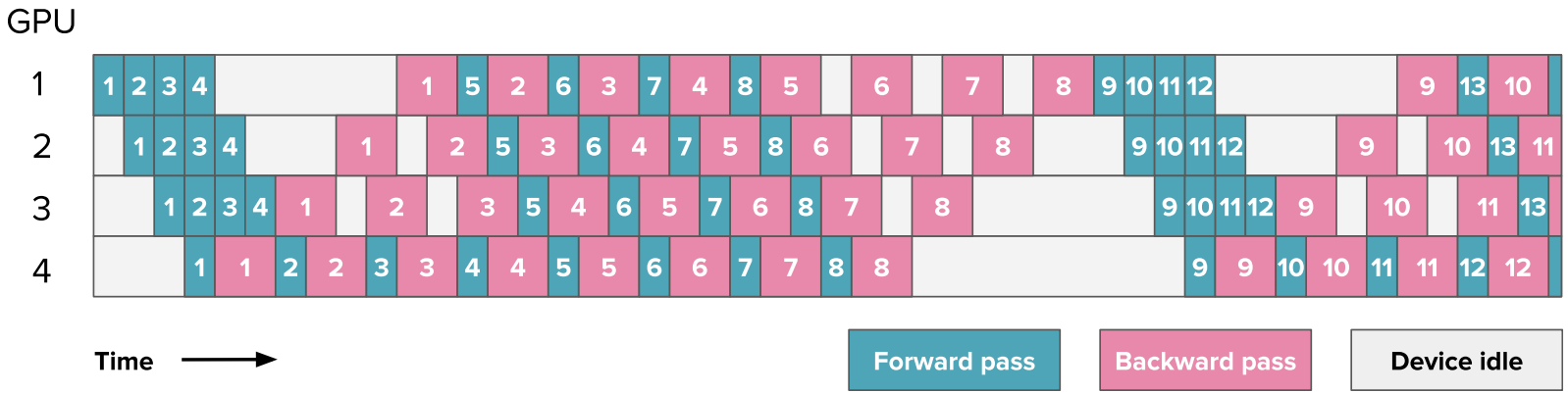


Idle / Work Ratio = PP-1 / M= 3 / 4

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

Pipeline Parallelism

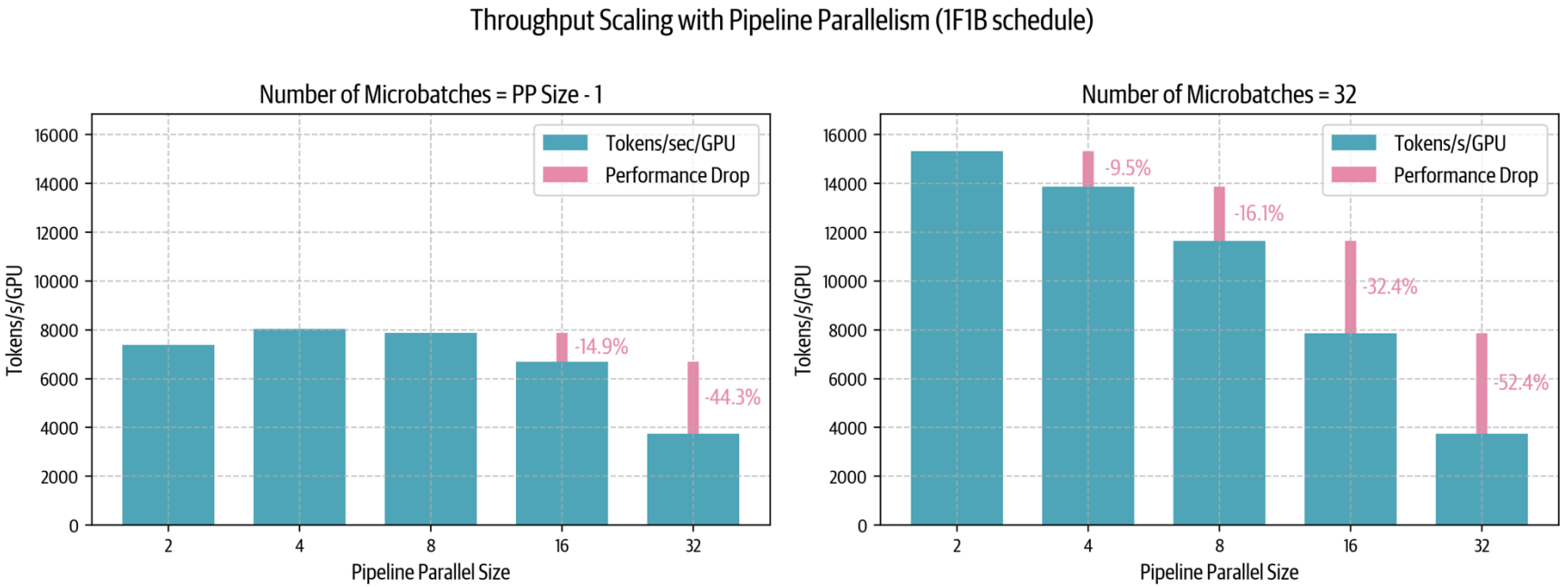
A cleverer version of AFAB: 1 Forward 1 Backward (1F1B)
Idea: Do backward as early as possible, releasing activations on the fly



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

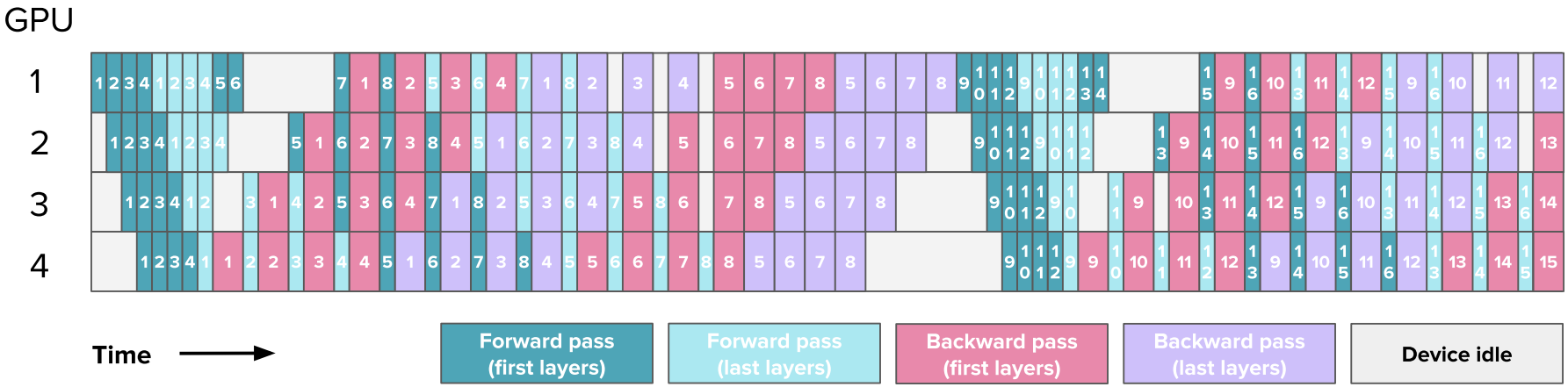
GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism (Huang et al., NeurIPS 2019)

Pipeline Parallelism Throughput

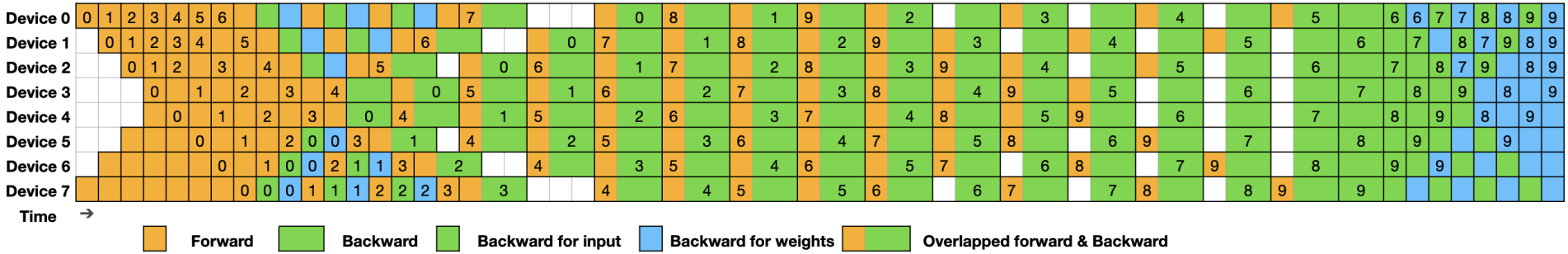


A small drop in throughput when scaling beyond 8 GPUs (one node)
but a large drop as we increase the microbatch number

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.

Pipeline Parallelism, in practice

- In torch, you can define your model as a sequence of layers and wrap with `torch.distributed.pipeline.sync.Pipe`

```
from torch.distributed.pipeline.sync import Pipe
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(1024, 4096),
    nn.ReLU(),
    nn.Linear(4096, 1024)
)
# Split into 2 pipeline stages across 2 GPUs, with 4 microbatches
model = Pipe(model, chunks=4, devices=['cuda:0', 'cuda:1'])
```

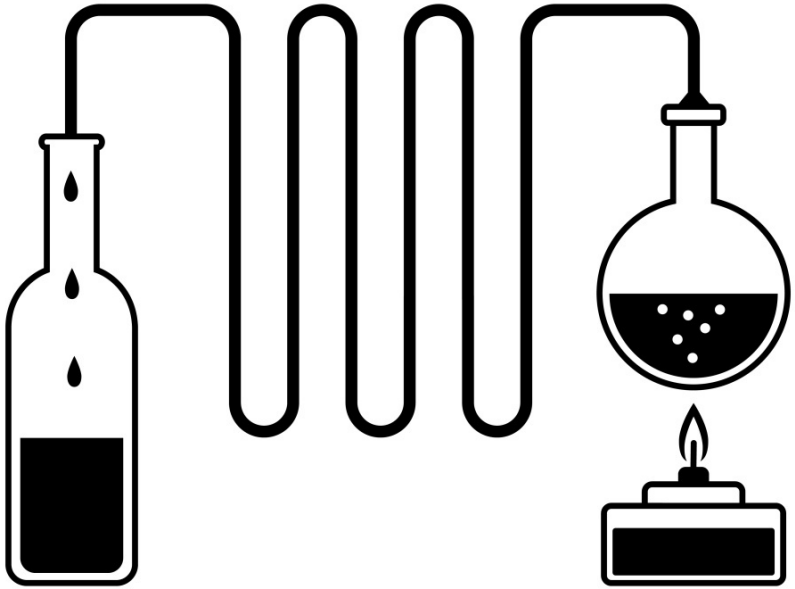

https://docs.pytorch.org/docs/stable/distributed.pipelining.html

Summary

Aspect	Naive Data Parallel (DP)	ZeRO-1	ZeRO-2	ZeRO-3	Tensor Parallel (TP)	Pipeline Parallel (PP)
Core idea	Replicate full model; split data.	Shard optimizer states.	Shard optimizer states + gradients.	Shard optimizer states + gradients + parameters.	Split tensors *within layers* across GPUs.	Split *layers* of the model* across GPUs sequentially.
Memory footprint	Very high	Lower (no optimizer redundancy)	Lower (no optimizer/grad redundancy)	Minimal/no redundancy	Low–moderate	Moderate
Ease of use	Very easy	Easy	Moderate	More complex	Complex (custom kernels)	Moderate (microbatch tuning)
Used for	Training only	Training only	Training only	Training only	Training + inf	Training + inf
Typical combo	Often + ZeRO	Paired with DP	Often w/ TP or PP	Combined with TP + PP (3D)	Paired with PP	Paired with TP + ZeRO

Distilling the knowledge of larger models

Distillation



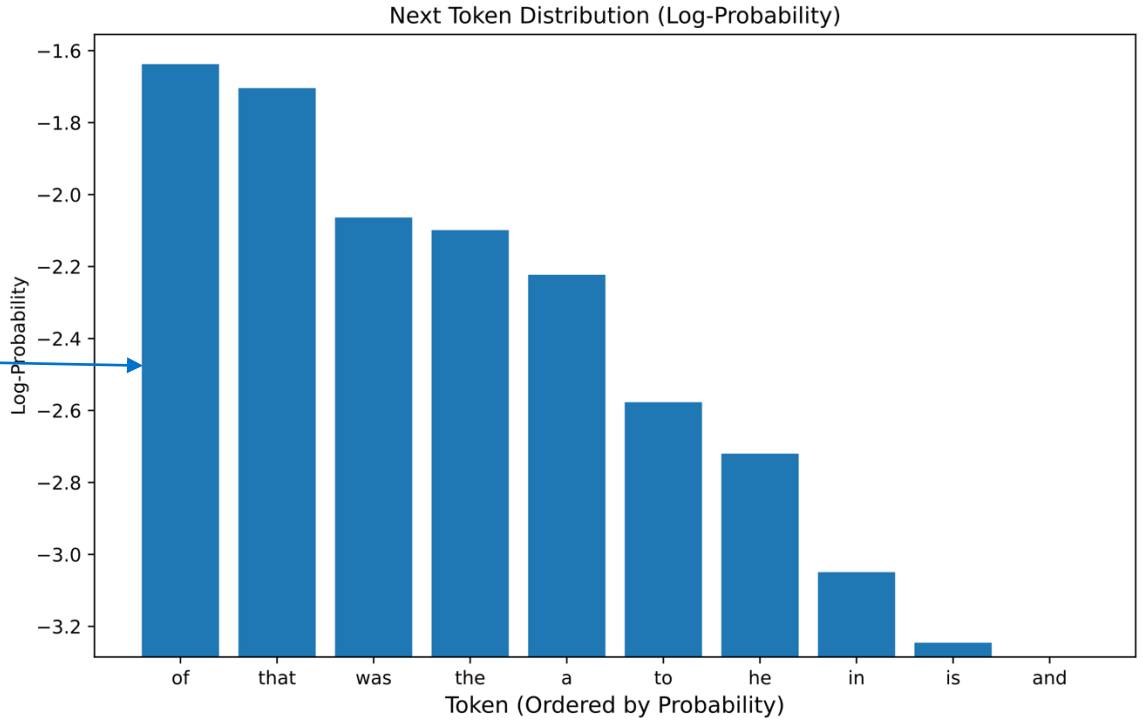
small model

Knowledge and/or capabilities of a larger model

Revisit: Standard Training (NLLloss)

prefix: The strange case ___
groundtruth: of

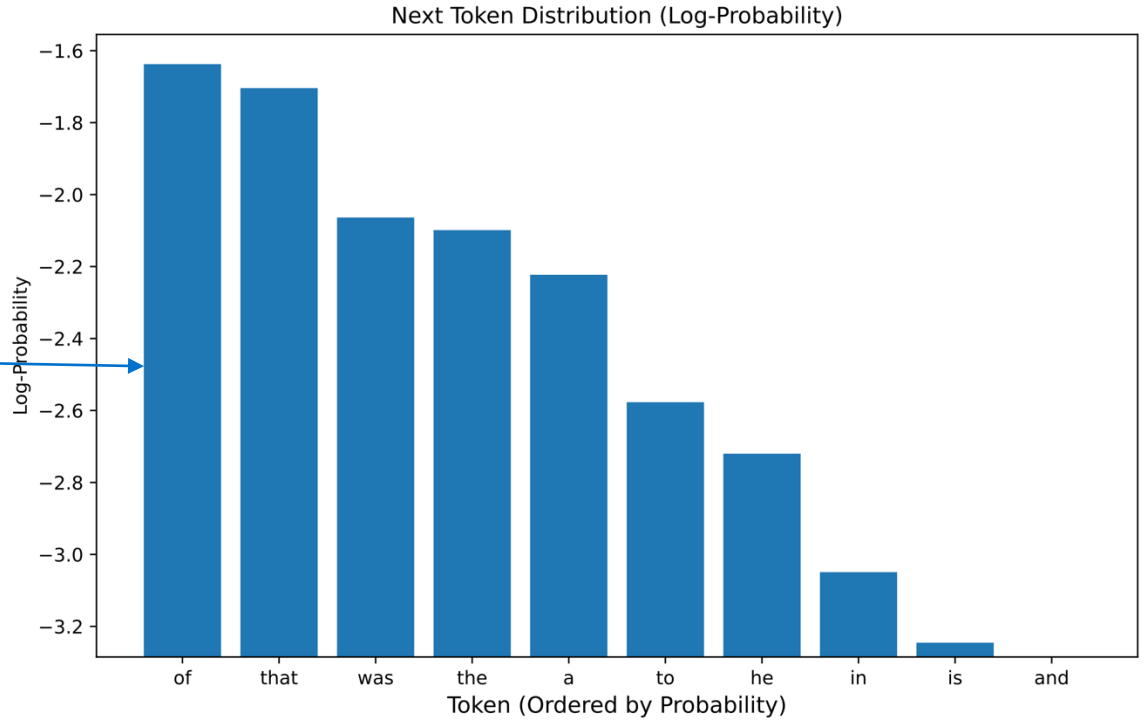
Loss = -log p(of)
= Cross Entropy(y_pred, groundtruth)



Revisit: Standard Training (NLLloss)

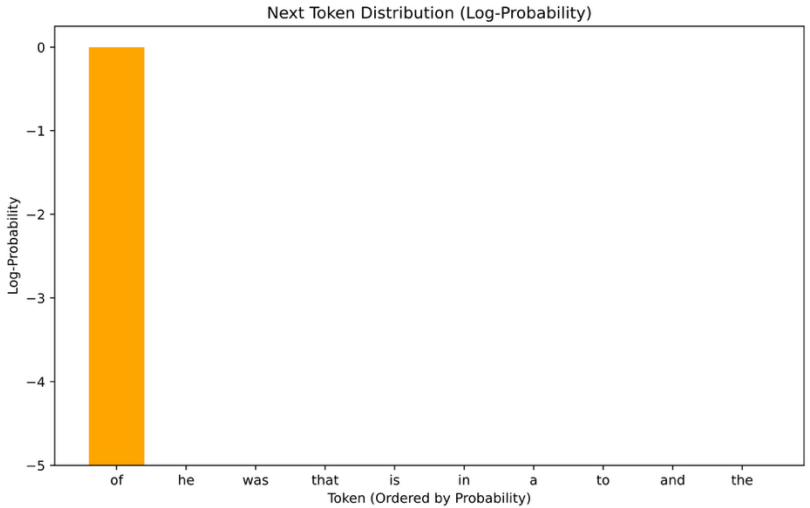
prefix: The strange case ___
groundtruth: of

loss = -log p(of)

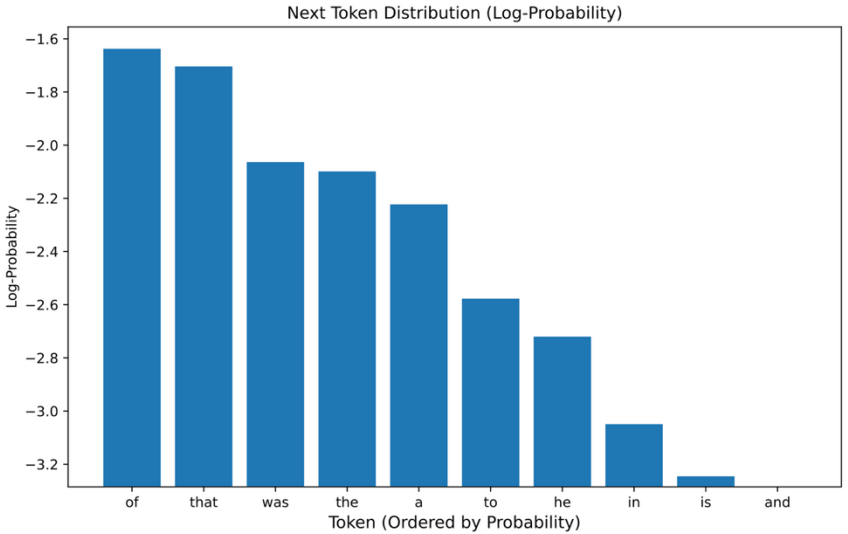


Revisit: Standard Training (NLLloss)

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



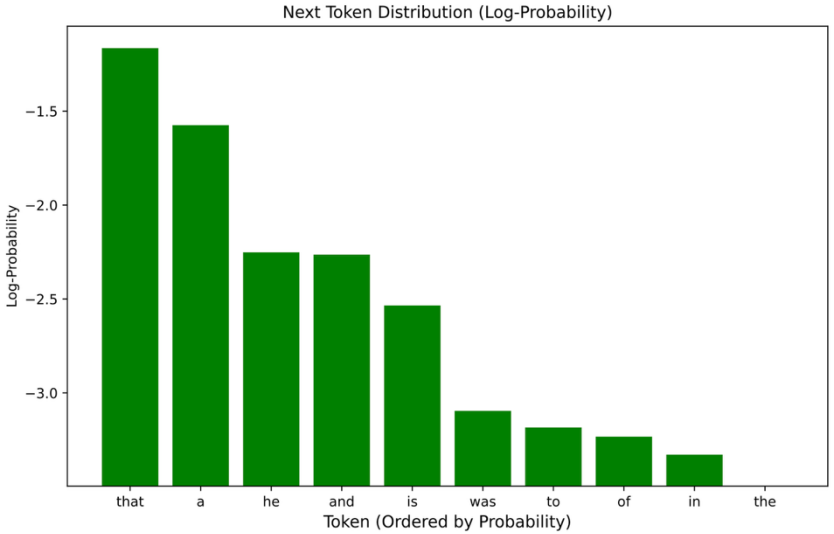
Groundtruth
(one-hot)

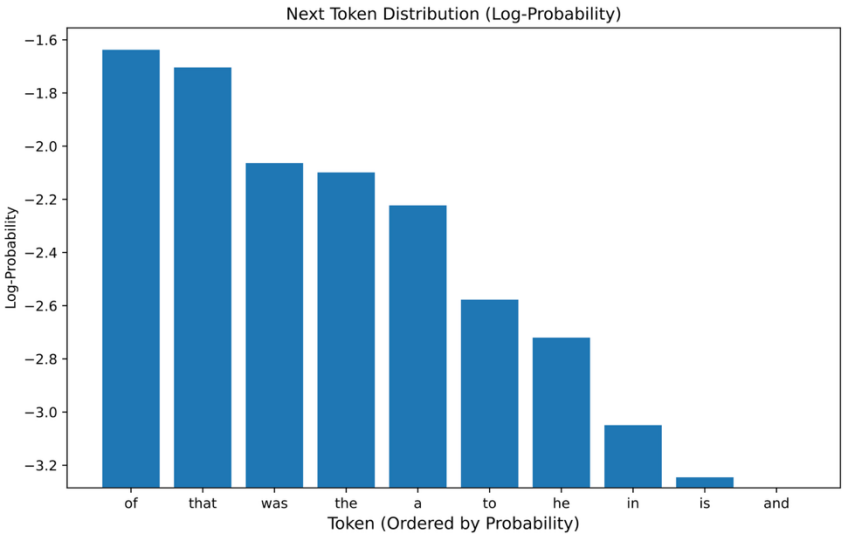


y_pred

Knowledge Distillation

KD loss = Cross Entropy(y_large, y_pred)





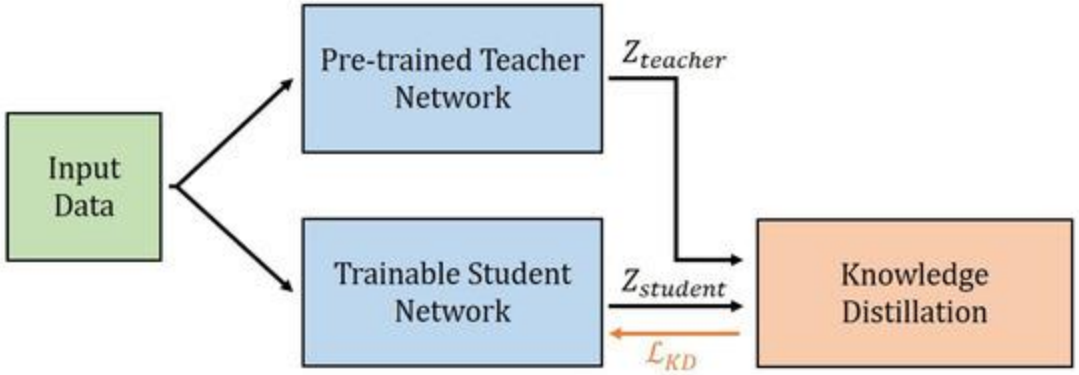
Large model next token probs (y_large)

small model next token probs (y_pred)

Knowledge Distillation

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

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

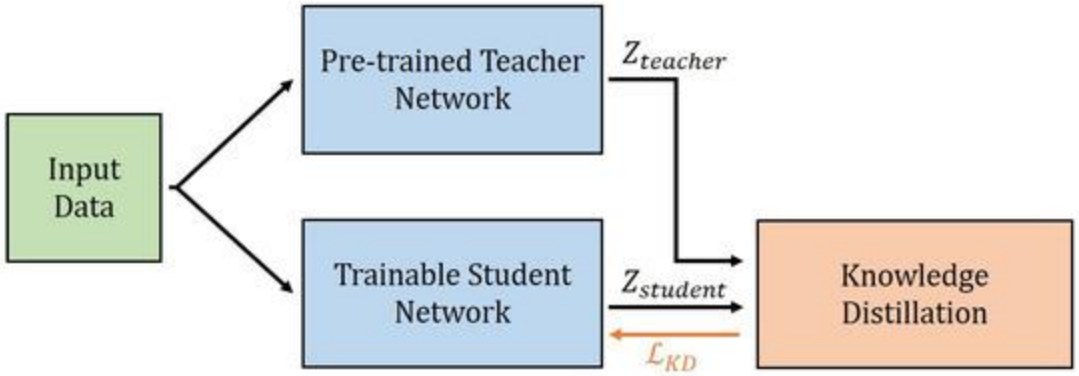


Step 3: Use teacher outputs to train student (Cross Entropy)

What if the teacher is Proprietary (GPT)?

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

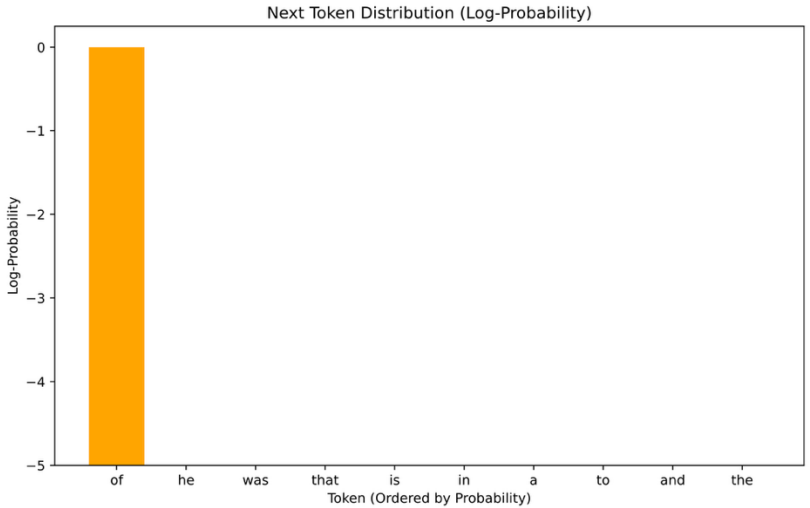
Step 2: Feed input data to both student and teacher (freezed)



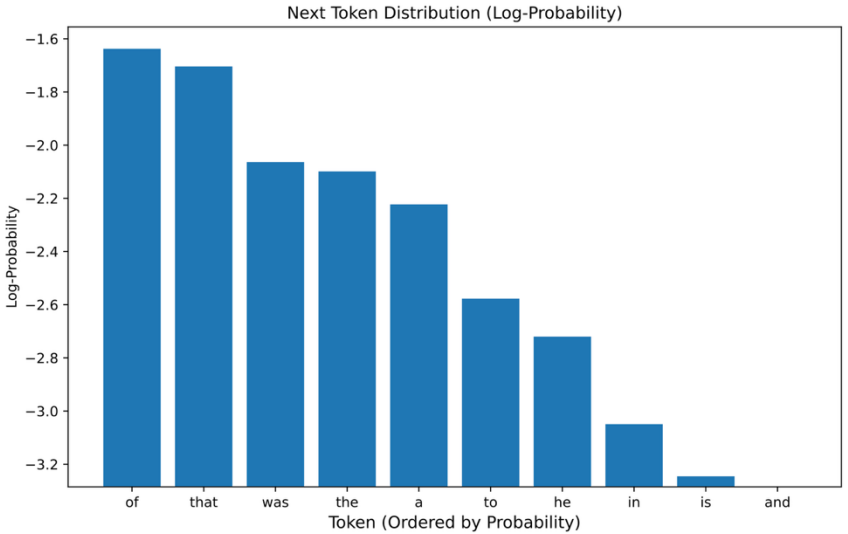
Step 3: Use teacher generations (instead of outputs) to train student!

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)

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.

Model Serving

In-flight batching

- LLMs handle diverse workloads (chat, summarization, code generation, etc.) with highly variable output lengths, making batching difficult.
- Traditional batching struggles because some requests finish much earlier, leaving GPU resources idle.
- In-flight (continuous) batching addresses this by dynamically managing active requests.
- During generation, completed sequences are immediately evicted from the batch.
- New requests are inserted into available slots without waiting for others to finish.
- This allows continuous execution across iterations of model inference.

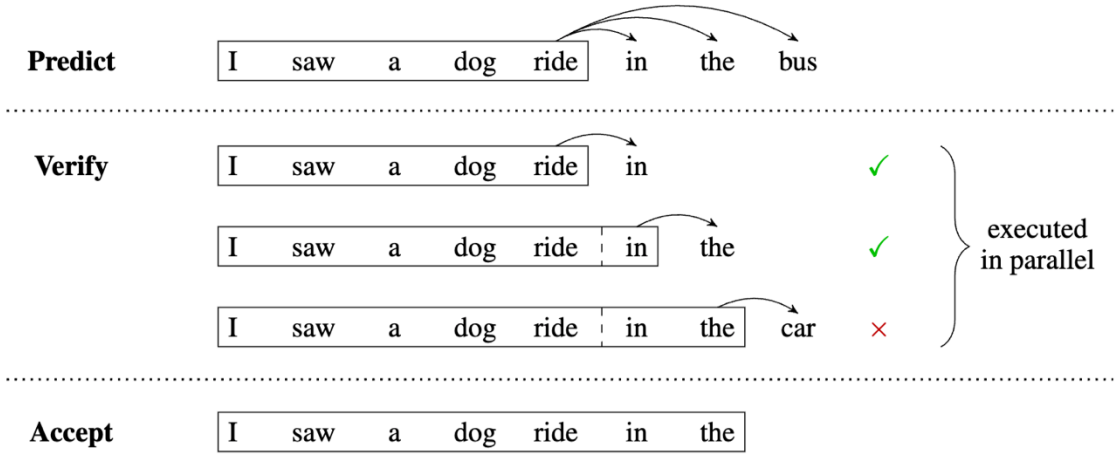
Speculative Inference

- Speculative inference (aka speculative sampling, assisted generation, or blockwise parallel decoding) accelerates LLM generation by partially parallelizing token prediction.
- Standard autoregressive decoding is sequential—each token depends on all prior ones.
- Core idea: use a smaller, faster "draft" model to predict several future tokens ahead of time.

Blockwise Parallel Decoding for Deep Autoregressive Models, 2018

Speculative Inference

- The larger "verification" model then checks these draft tokens in parallel.
- If the verifier's outputs match the draft's predictions, those tokens are accepted immediately.
- If a mismatch occurs, tokens after the first disagreement are discarded, and a new draft is generated
- This process reduces the number of sequential verification steps.



Blockwise Parallel Decoding for Deep Autoregressive Models, 2018

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

-