Distributed Training

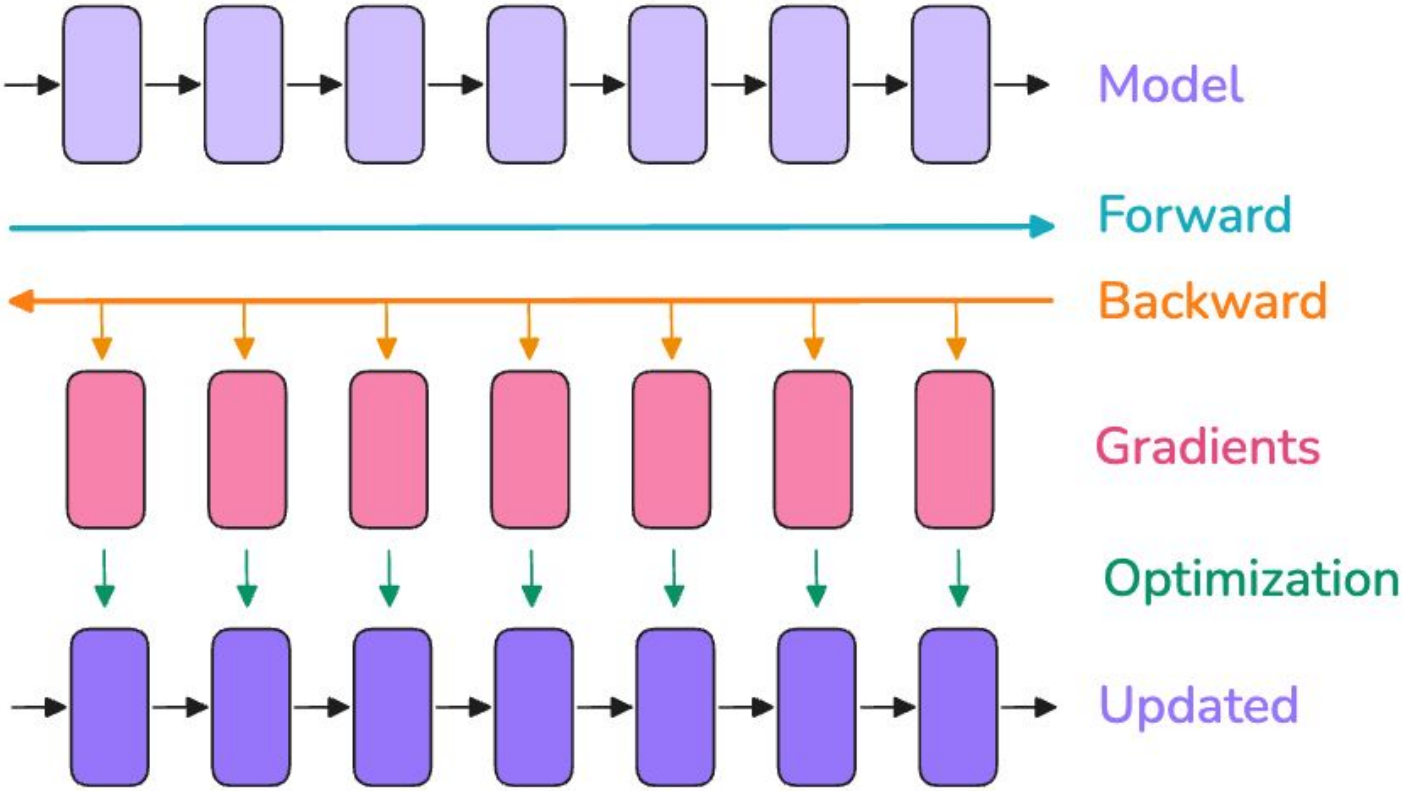
Make training system go brrrrr

Yongye Zhu, Anderson Lee, Ahmed Khaled

Problem Statement

- How to train a 405B model from scratch?
  - How to train a smaller 1B model?
  - How to scale the model size with multiple GPUs? (e.g. 16k)
  - How to let the training run as fast as possible? Run fast means keep the GPU doing matmul

Training Model



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Calculating Memory Usage

Model Parameter: $\lambda \cdot N$

Model Gradient: $\lambda \cdot N$

Optimizer State: $2\lambda \cdot N$

\+ Activation + Data

λ: Bytes per parameter

Peak GPU Memory Usage

Model parameters	FP32 or BF16 w/o FP32 grad acc
1B	16 GB
7B	112 GB
70B	1120 GB
405B	6480 GB

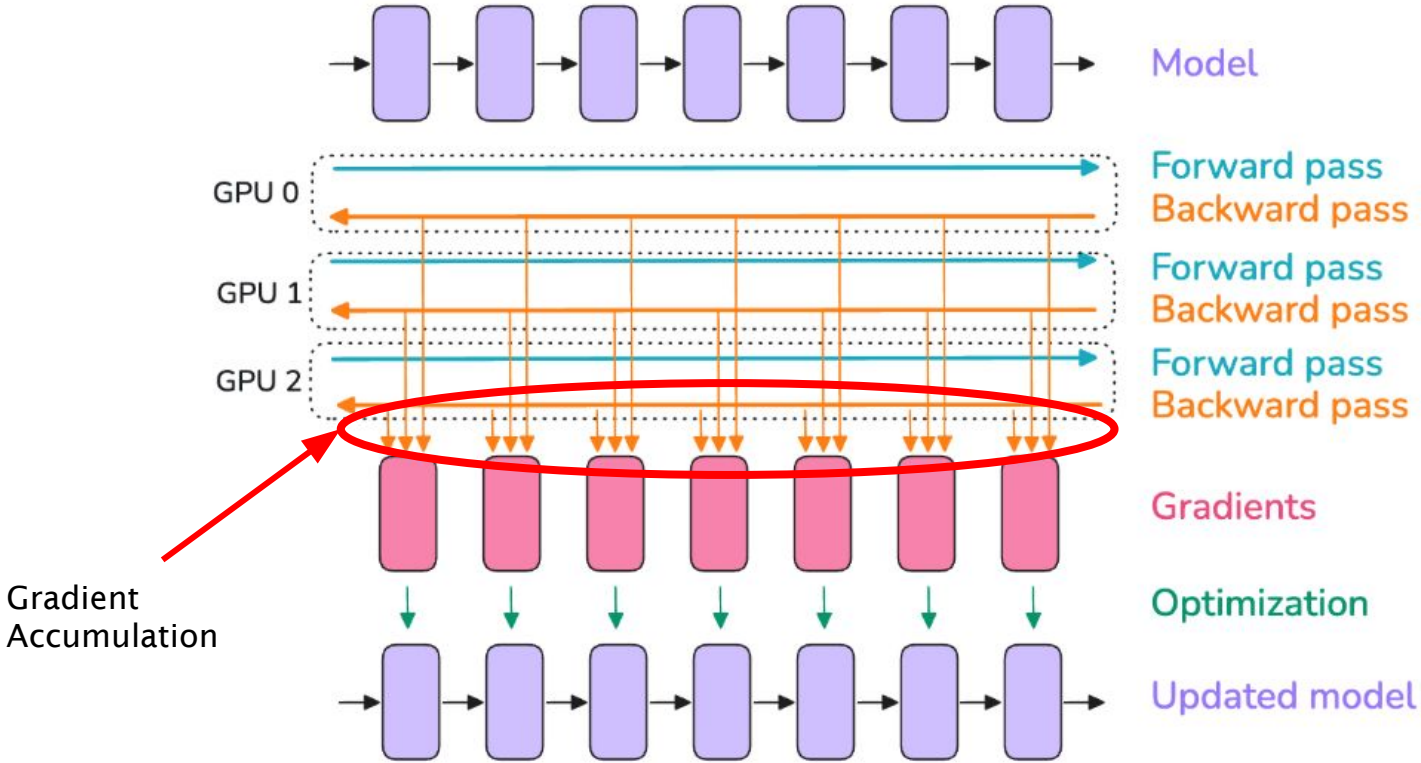
We can only train 1B on single GPU!!

https://huggingface.co/spaces/nanotron/ultrascale-playbook

Gradient Accumulation

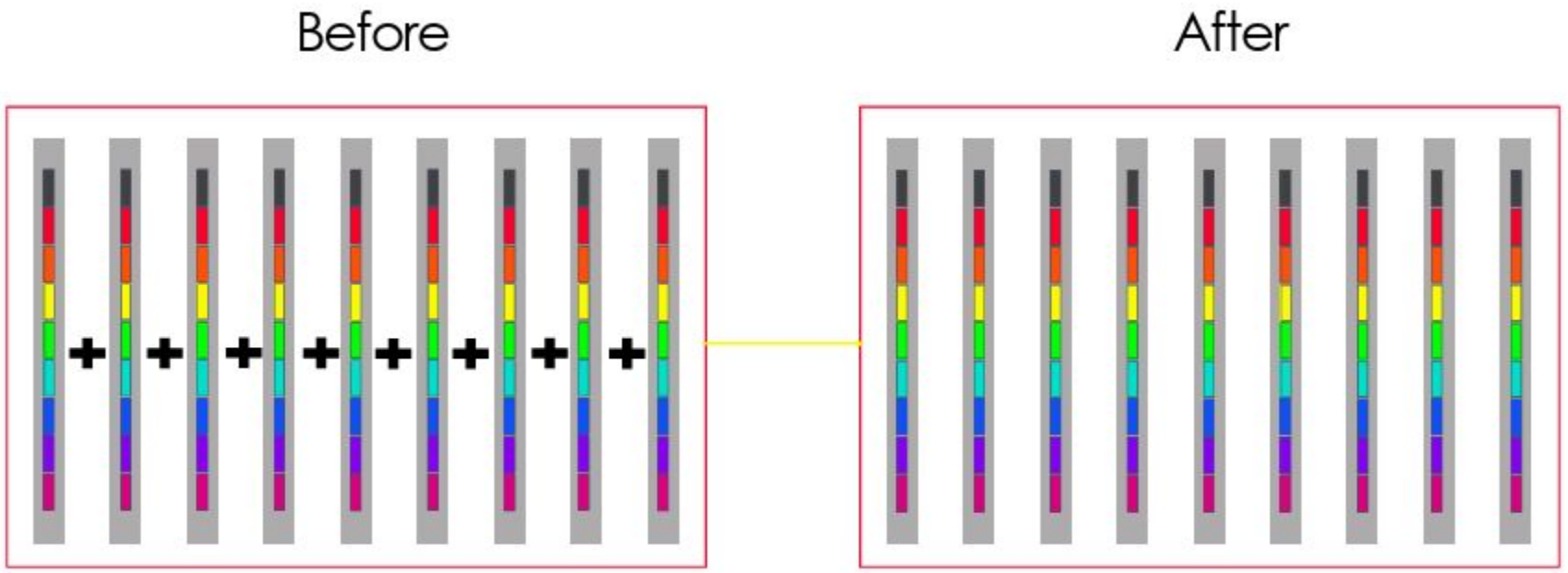
- We training LLM in batches
  - We will update the parameter every ***gbs*** (global batch size)
- During forward pass, we keep activation to calculate gradient.
- We can calculate the mean of gradient to update the parameter. (Parallel-Friendly)

Data Parallel



https://huggingface.co/spaces/nanotron/ultrascale-playbook

GPU Communication: All-Reduce



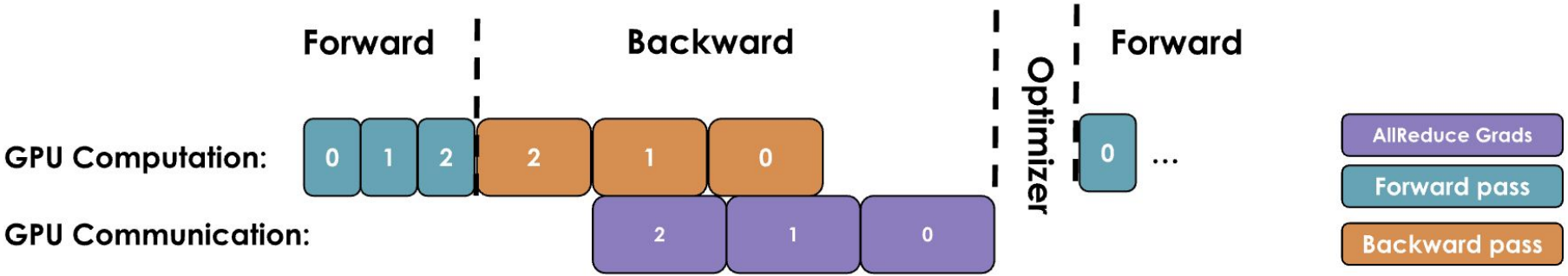
https://hao-ai-lab.github.io/cse234-w25/assets/slides/feb13.pdf

Data Parallel

- Each GPU perform ***mbs = gbs // num_gpus*** batches
- Result gradient are all-reduced (averaged). This can be hidden in computations
- Each GPU update parameter independently.
- Training total batches faster with more GPUs.
- Still can’t train a 7B model since each GPU has full model parameter.

Compute/Communication Overlap

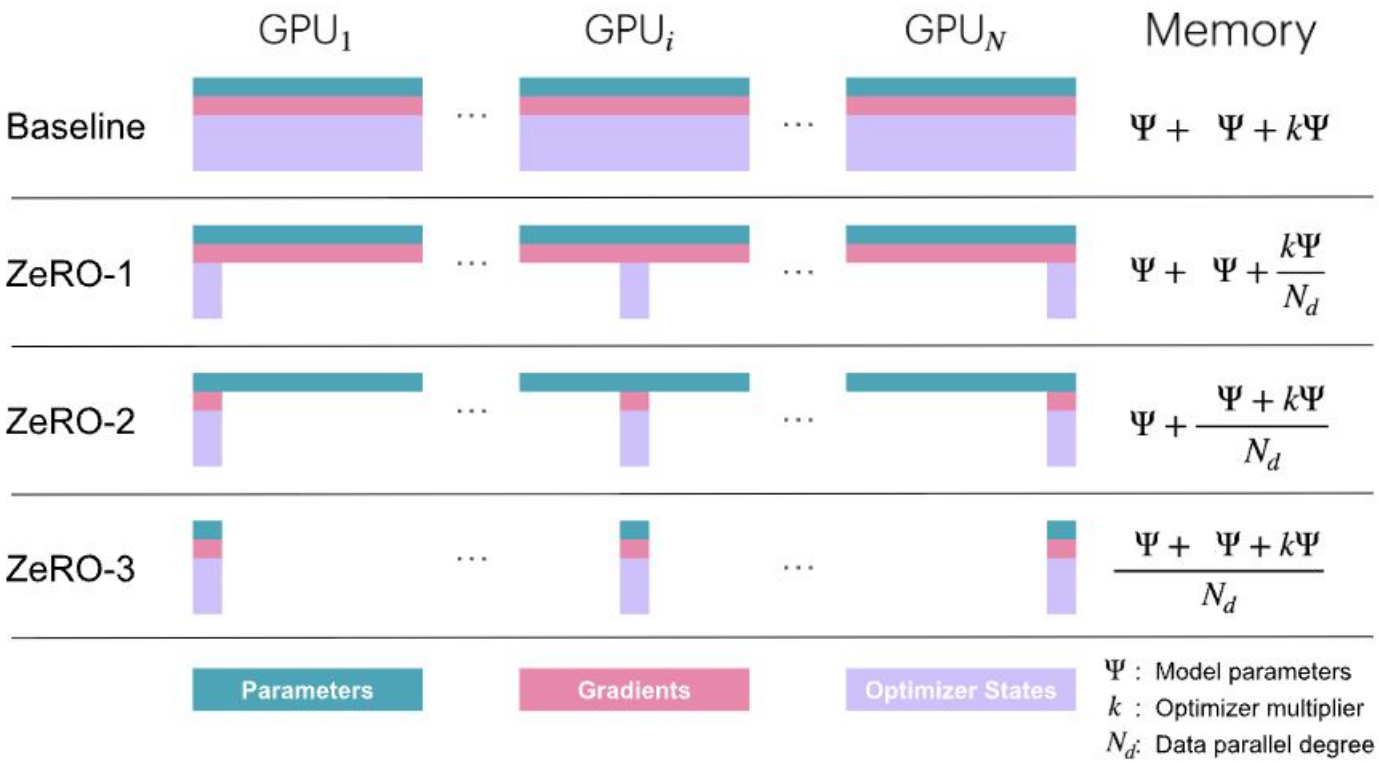
- Gradient accumulation for each layer happens immediately after calculated.
- Communication happens in the *next* (previous) layer gradient calculation.



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Parameter Sharding

Big Picture



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Measuring network efficiency

How to Scale Your Model

A Systems View of LLMs on TPUs

Training LLMs often feels like alchemy, but understanding and optimizing the performance of your models doesn't have to. This book aims to demystify the science of scaling language models: how TPUs (and GPUs) work and how they communicate with each other, how LLMs run on real hardware, and how to parallelize your models during training and inference so they run efficiently at massive scale. If you've ever wondered "how expensive should this LLM be to train" or "how much memory do I need to serve this model myself" or "what's an AllGather", we hope this will be useful to you.

AUTHORS

Jacob Austin
Sholto Douglas
Roy Frostig
Anselm Levskaya
Charlie Chen
Sharad Vikram
Federico Lebron
Peter Choy
Vinay Ramasesh
Albert Webson
Reiner Pope*

AFFILIATION

Google DeepMind

PUBLISHED

Feb. 4, 2025

FLOP utilization

- FLOPs/s of operation ÷ FLOPs/s of accelerator
- Example: If H100 achieves 989 TFLOPS peak for BF16, but your model runs at 400 TFLOPS
  Utilization = 400/989 ≈ 40%
- For LLM training, Model FLOPs Utilization (MFU) we usually aim for is around 50%-60%

Communication intensity

- Communication bytes ÷ (Memory / Network bandwidth bytes)
- Measures fraction of bandwidth used for inter-device communication
- Example: all-reduce gradient
  - Simplest form of parallelism
  - Must communicate $2\times(N-1)/N \times$ gradient_size bytes per iteration

Arithmetic Intensity

- Computation FLOPs ÷ Memory bytes accessed
- Measures compute per byte of memory traffic
- High arithmetic intensity
  = compute-bound (good!)
- Low arithmetic intensity
  = memory-bound
- E.g. for H100:
  - Need >240 FLOPs/byte to be compute-bound with BF16

Matrix multiplication example

- Most computations in a transformer are just mat muls
- Consider multiplying two N×N matrices:
  - FLOPs: $2N^3$ (in reality can be different due to blocking)
  - Memory read/write: $3N^2$ elements
  - Arithmetic intensity (AI) = $2N^3 \div (3N^2 \times$ bytes per element)
  - N=1024, BF16: AI = 341 FLOPs/byte
    - Compute-bound for H100.

Sharded matrix multiplication (I)

- What if we sharded the matrices?
- Consider two NxN matrices distributed across M devices
- Device *i* holds

$A[iN/M : (i+1)N/M, :]$ — N/M rows of A
$B[:, iN/M : (i+1)N/M]$ — N/M columns of B

Sharded matrix multiplication (II)

Each device i needs to compute C[iN/M:(i+1)N/M, :]:

1. Local compute: $A_{local} \times B_{local} \rightarrow C[i,i]$ block
2. Ring communication: Send B_{local}, receive $B_{neighbor}$
3. Repeat M-1 times to compute all blocks of C's rows

Sharded matrix multiplication (III)

- FLOPs per device: $2N^3/M$ (perfectly scaled!)
- Memory per device: $2N^2/M$ (input) + N^2/M (output)
- Communication: $(M-1) \times N^2/M$ elements = $N^2(M-1)/M$
- Arithmetic intensity: $2N^3/M \div N^2(M-1) \approx 2N/(M-1)$
  - **Decreases with more devices**
  - Need larger N to maintain compute-bound operation

Some key numbers

- **H100 BF16 Performance:** 989 TFLOPS
- **Transformer Training FLOPs:** 6×N×T
  - N = number of parameters
  - T = number of tokens per step
- **Intranode bandwidth (NVLink):** 900 GB/s
- **Internode bandwidth:**
  - InfiniBand NDR: 400 Gbps = 50 GB/s
  - Ethernet alternatives: 100-400 Gbps typical

Forms of parallelism

Review: Adam

- Parameters: N
- Gradients: N
- Momentum: N
- 2nd order momentum: N
- **Total:** 4N parameters in memory
- + Activations + Data

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$

$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

$$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$$

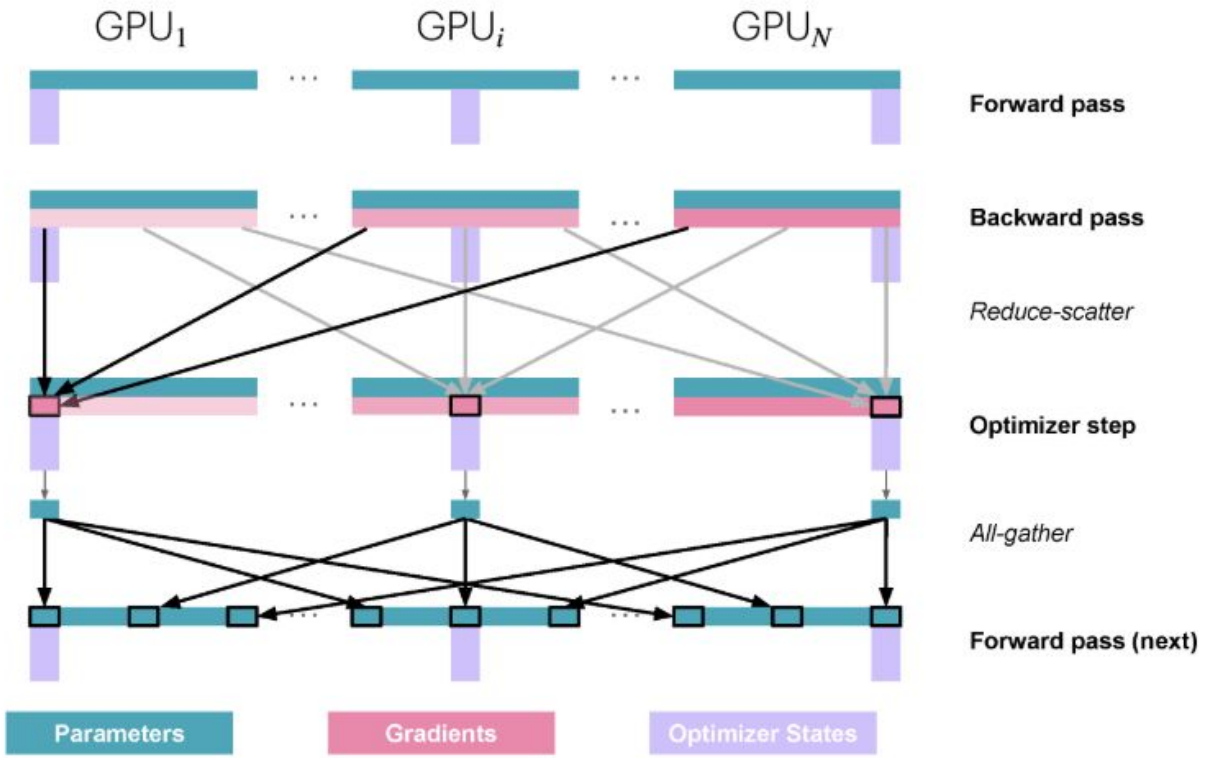
Sharding

- Different slices of a tensor belong to different devices
- Example: Weight matrix $W \in \mathbb{R}^{M \times N}$ with 4 devices
  - Device 0: W[0:M/4, :]
  - Device 1: W[M/4:M/2, :]
  - Device 2: W[M/2:3M/4, :]
  - Device 3: W[3M/4:M, :]
- Key challenge: Managing communication when sharded tensors interact

ZeRO-1 Optimizer State Sharding

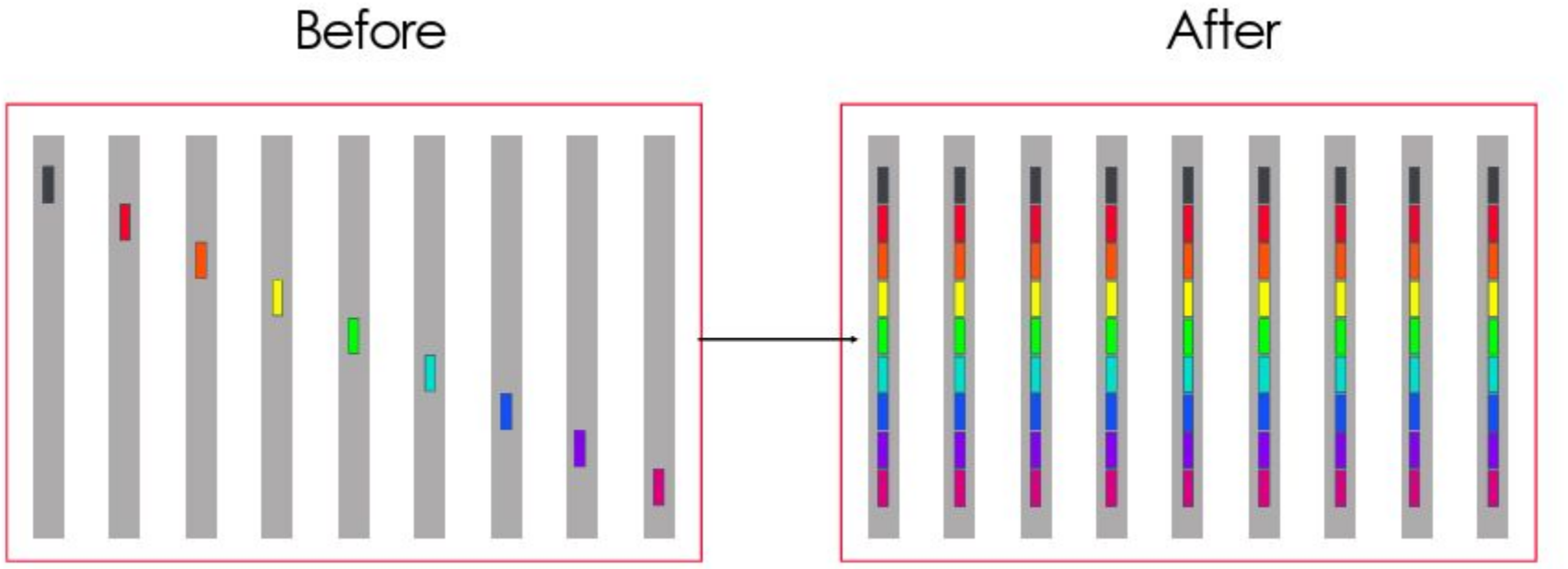
- Optimizer states (m, v from Adam) divided layerwise among devices
- Each device only maintains optimizer states for its assigned layers
- After computing gradients:
  - Device updates only its owned optimizer states
  - Broadcasts updated parameters to all devices

ZeRO-1 Optimizer State Sharding



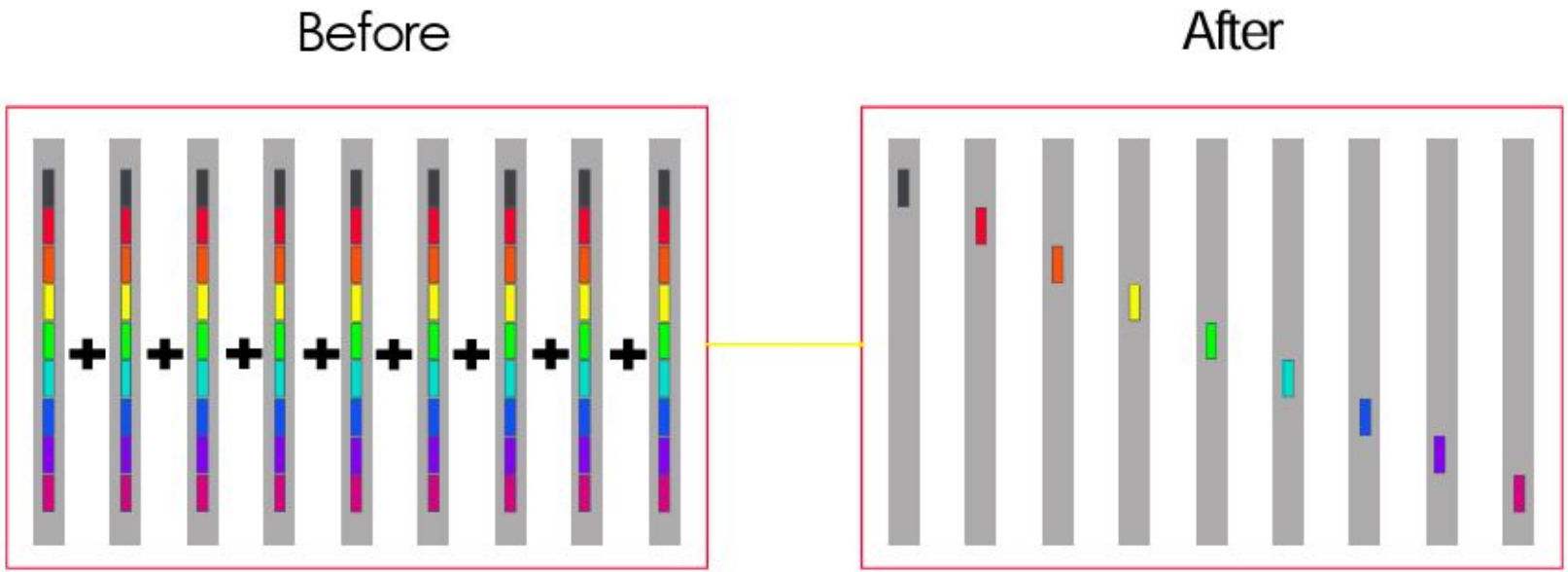
https://huggingface.co/spaces/nanotron/ultrascale-playbook

All-Gather



https://hao-ai-lab.github.io/cse234-w25/assets/slides/feb13.pdf

Reduce-Scatter



https://hao-ai-lab.github.io/cse234-w25/assets/slides/feb13.pdf

ZeRO-1: Memory & Communication

- **Memory:** $k\Psi/N+2\Psi$ parameters per device
- **Communication:**
  - Same as data parallelism
  - No additional communication overhead (q for class: why?)
  - Again we can hide this overhead in forward and backward computation

Note on ZeRO

- This was a bit confusing to me personally
- We need to differentiate between memory usage for all layers vs transient memory usage (i.e. for a single layer)
- (see whiteboard)

ZeRO-2: Optimizer+Gradient Sharding

- Gradients also sharded with same layout as optimizer states
- During backward:
  - Compute full gradients locally
  - Reduce-scatter to get sharded gradients
  - Each device keeps only gradients for its layers

ZeRO-2: Memory & Communication

- **Memory:** $(k + 1)\Psi/N+\Psi$ parameters per device
- **Communication:**
  - Same as data parallelism
  - No additional communication overhead
  - We don't need to get the gradients for other layers
  - Still communicate updated params
- Supported by DeepSpeed, Megatron-LM, PyTorch FSDP

ZeRO-3: Full Parameter Sharding

- Parameters also sharded layerwise across devices
- At layer L:
  - Before forward pass: All-gather parameters for this layer -> do forward computation (replicated)
  - After forward: Discard non-owned parameters
  - Before backward: All-gather parameters again -> do backward computation (replicated)
  - After backward: Do optimizer step + discard non-owned parameters

ZeRO-3 Tradeoffs

- **Memory**: (k+2)Ψ/N bytes per device
- **Communication:** 1.5X of DP
  - Prev: one all-gather in bwd pass
  - Now: one all-gather in fwd, one all-gather in bwd
- “Sharded parameters, **replicated computation**”

FSDP2

- Like ZeRO-3, but instead of sharding layerwise, shard across dim-0 of every param/grad/optimizer state tensor
- Easier conceptually to implement
- Natively integrated into PyTorch
- Can be integrated with smarter torch.compile.

Where we are now?

Still bottlenecked by activation memory (especially with long seq len)

https://huggingface.co/spaces/nanotron/ultrascale-playbook

Activation Sharding

Tensor Parallelism

- Like ZeRO-3, but computation is also sharded
- Parameters stay on assigned devices
- Activations are all-reduced/all-gathered between devices as needed for computation
- How we choose to do the sharding strongly influences the computation/communication

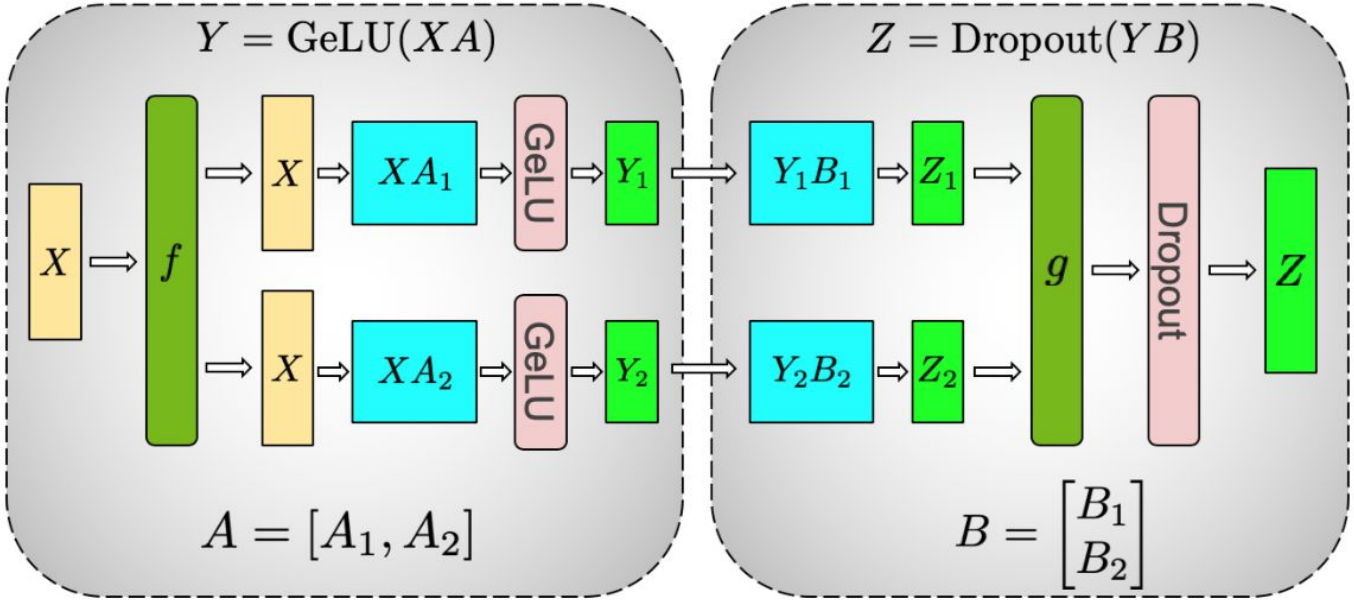
Tensor Parallelism

- Consider MLP: $Y = GeLU(XA)$ followed by $Z = YB$
- **Row-wise split of A:**
  - Partition: $X = [X_1, X_2]$, $A = [A_1; A_2]$
  - Result: $Y = GeLU(X_1A_1 + X_2A_2)$
  - Problem: $GeLU(X_1A_1 + X_2A_2) \neq GeLU(X_1A_1) + GeLU(X_2A_2)$
  - Requires synchronization before GeLU, beating the point

Tensor Parallelism

- Consider MLP: $Y = GeLU(XA)$ followed by $Z = YB$
- **Column-wise split of A:**
  - Partition: $A = [A_1, A_2]$
  - Result: $[Y_1, Y_2] = [GeLU(XA_1), GeLU(XA_2)]$
  - GeLU applied independently - no sync needed!
  - Second GEMM: Split B row-wise to match
  - Only one all-reduce after second GEMM

Tensor Parallelism



(a) MLP

From Megatron-LM paper

Tensor Parallelism

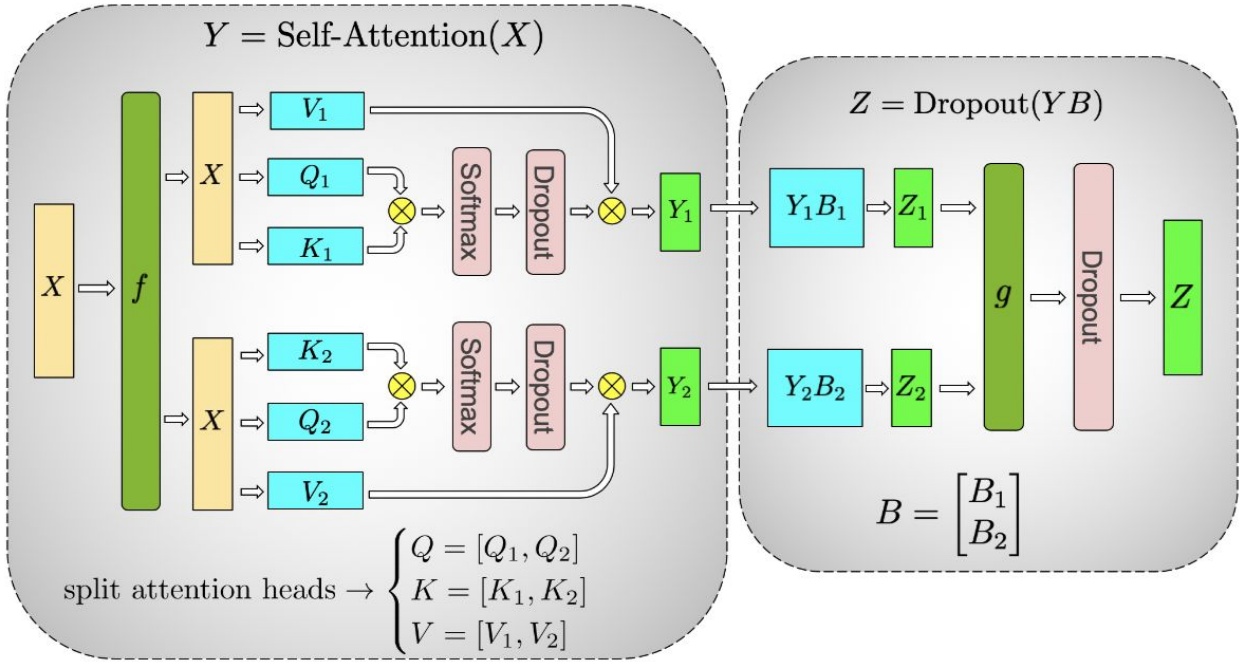
```
class f(torch.autograd.Function):
    def forward(ctx, x):
        return x
    def backward(ctx, gradient):
        all_reduce(gradient)
        return gradient
```

Code 1. Implementation of f operator. g is similar to f with identity in the backward and all-reduce in the forward functions.

From Megatron-LM paper

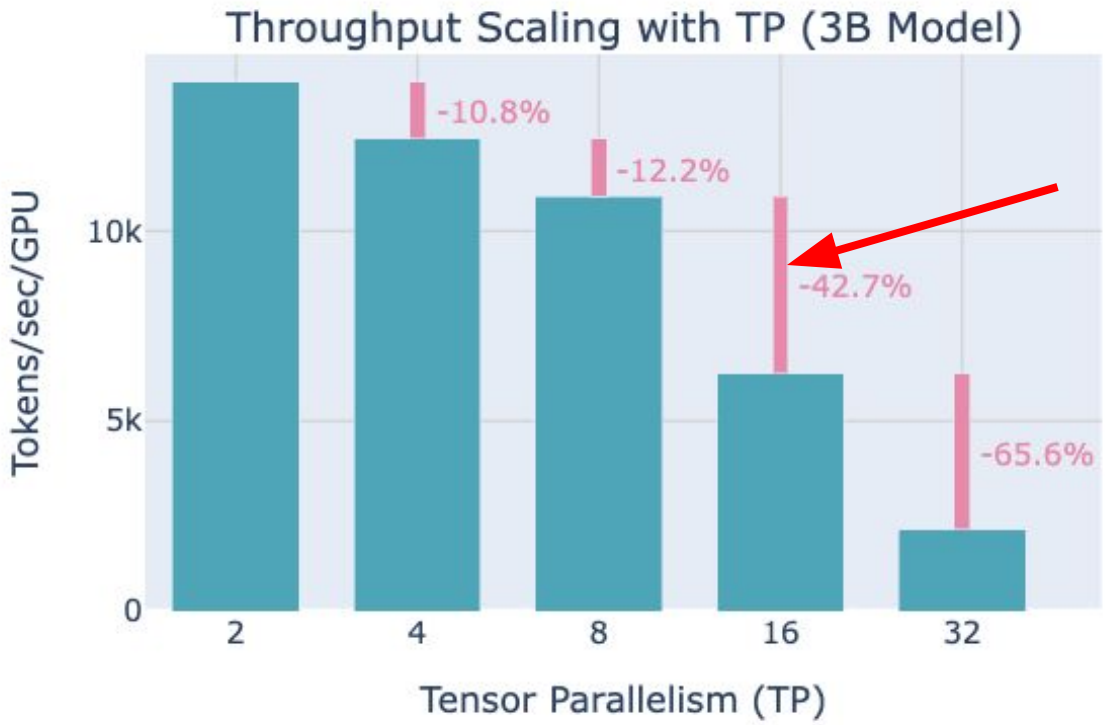
Tensor Parallelism

From Megatron-LM paper



(b) Self-Attention

Tensor Parallel Performance

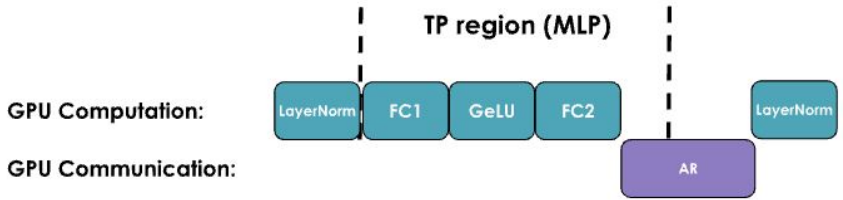


Huge Drop

https://huggingface.co/spaces/nanotron/ultrascale-playbook

Tensor Parallel Performance

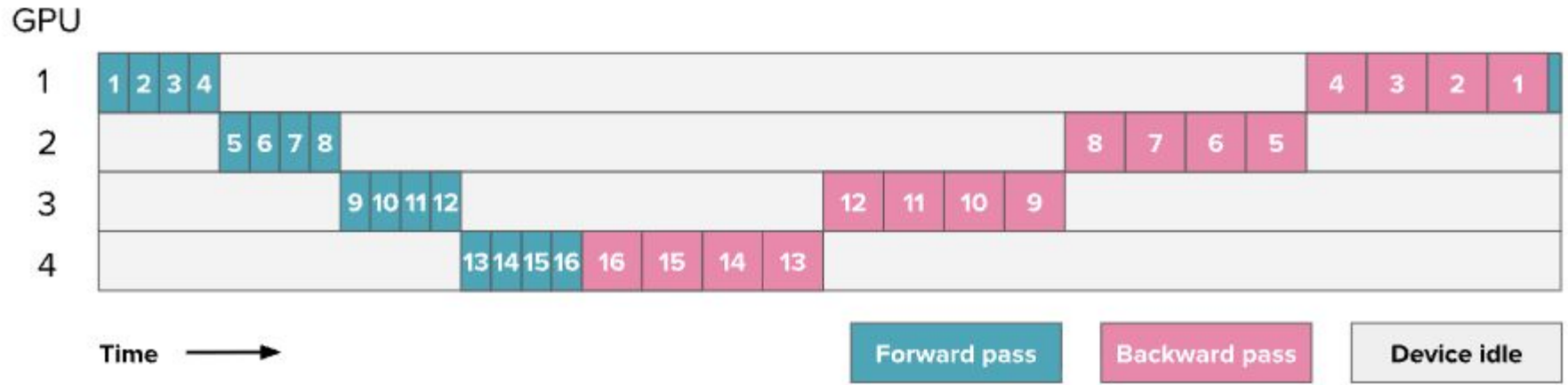
- All-Reduce sits on the critical path during forward.
- One node has 8 GPUs.
- Cross-node comm is much slower than inter-node comm. (NvLink vs InfiniBand)



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Pipeline parallelism

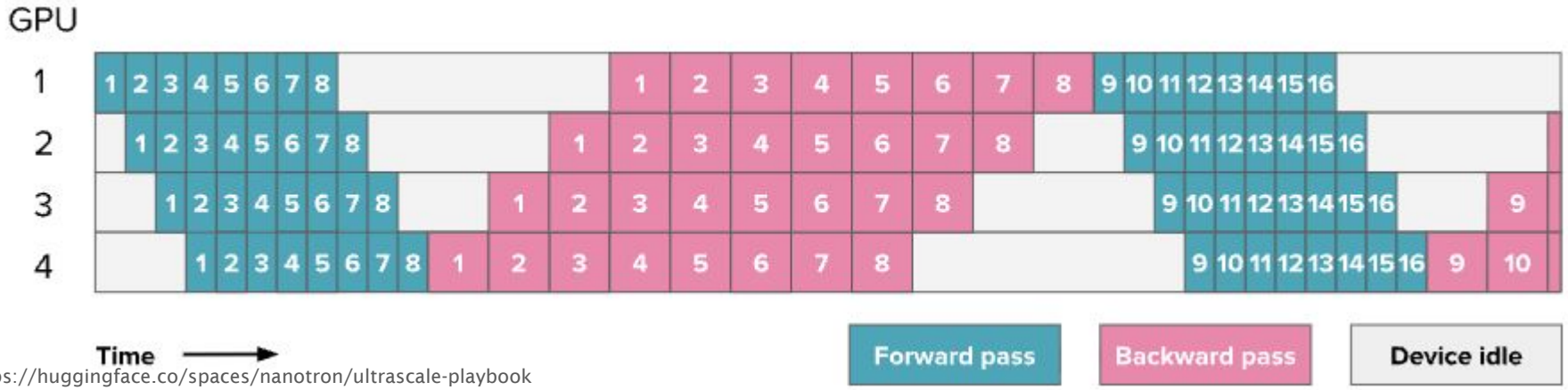
- Model divided into stages (groups of layers)
- Each device owns one pipeline stage
- Data flows through devices sequentially
- Limitation: Device Idle



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Microbatch All Forward All Backward

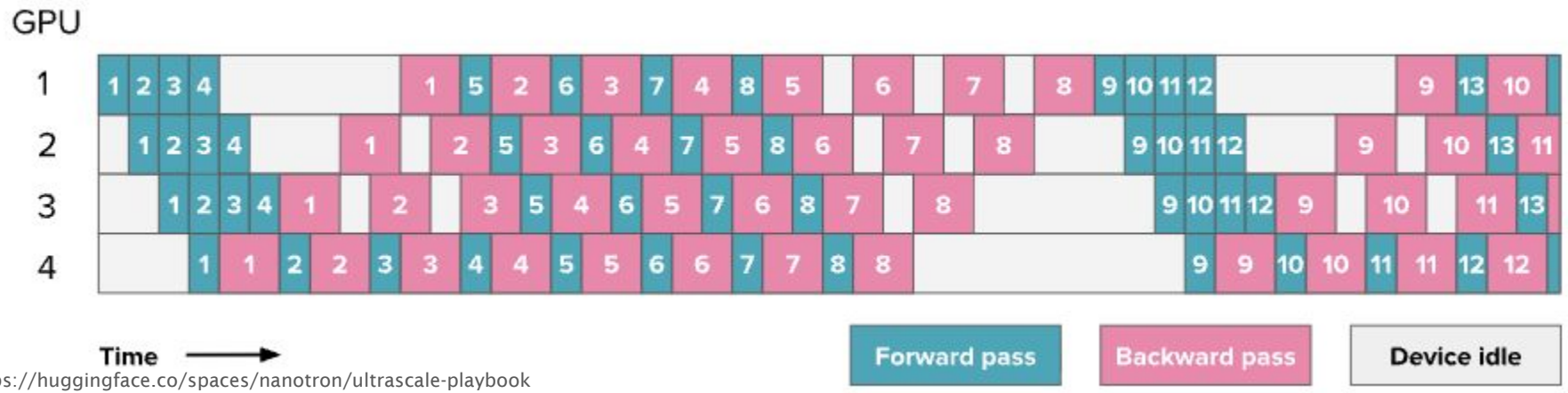
- Micro-batching to reduce pipeline bubbles
- Reduce total computation time by ***m***, number of microbatches
- Problem: Need to store all microbatch activations



https://huggingface.co/spaces/nanotron/ultrascale-playbook

1Forward 1Backward (Depth-First Scheduling)

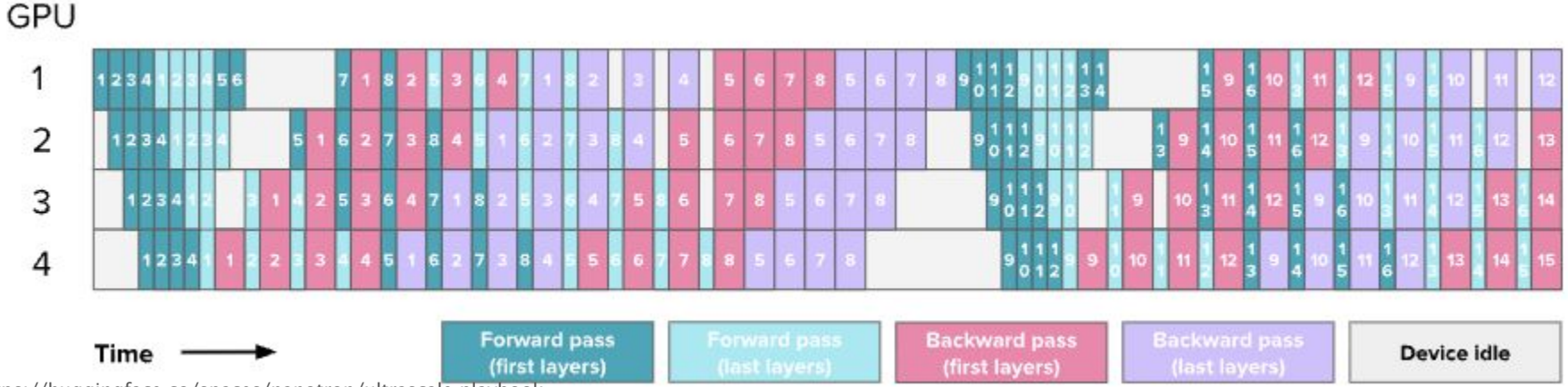
- Prioritize backward pass over forward pass
- Immediate release activation memory after backward pass
- Still same time as AFAB, but more memory efficient



https://huggingface.co/spaces/nanotron/ultrascale-playbook

Interleaved 1F1B

- Each GPU has more than one layer
- Computation circulating around GPUs
- Further reduce overall training time by ***n***, number of layer each GPU has.



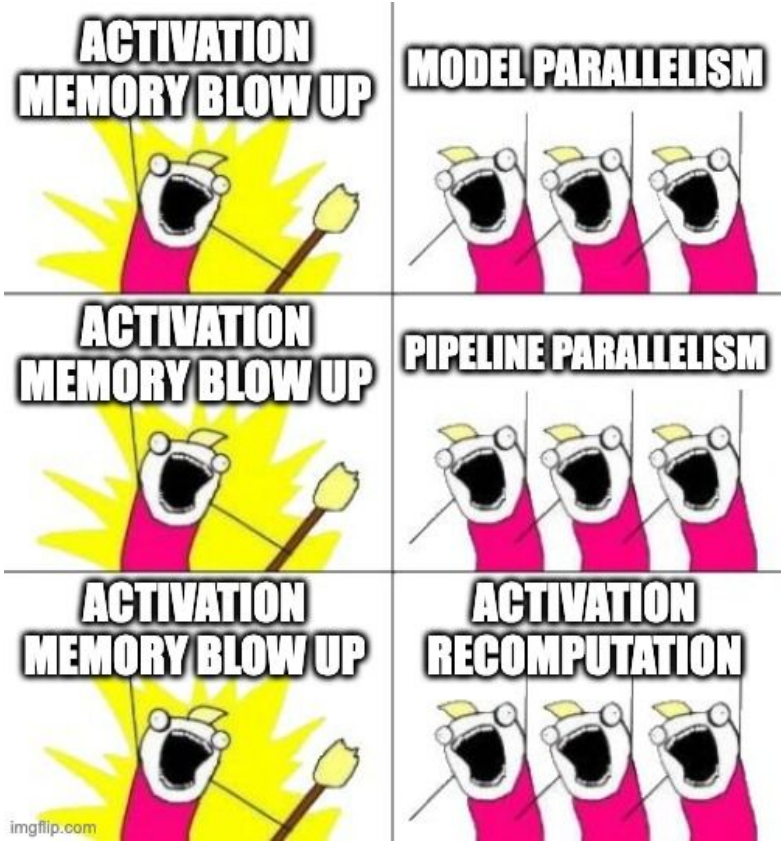
https://huggingface.co/spaces/nanotron/ultrascale-playbook

Zero Bubble Pipeline/DualPipe

DeepSeek-V3 Technical Report: arxiv:2412.19437

Memory problem in training models

- Tensor parallelism
  - Help but introduces overhead
- Pipeline Parallelism
  - Doesn't help if maintaining device utility to avoid bubble
- Activation Recomputation



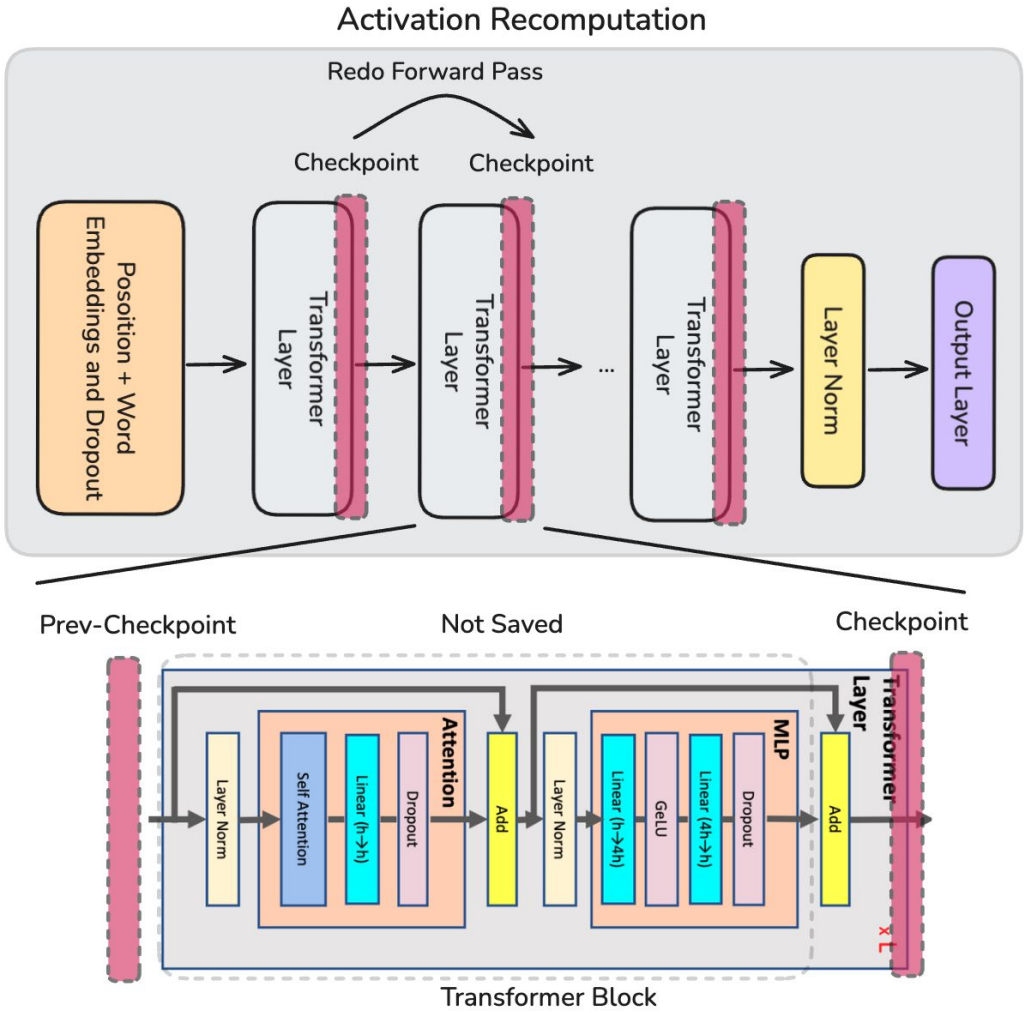
Activation Recomputation

- A.k.a. Gradient Checkpointing
- Select all/some layers' output to save, then re-forward pass from closest "checkpoint"
- Trading off compute for saving memory

Activation Recomputation

Within-block activation not saved

Redo forward pass from last block's output



Bottom figure from Megatron-LM

Reducing Activation Recomputation

- Three main components
  - Tensor Parallelism (Megatron-LM)
  - Sequence Parallelism
  - *Selective* Activation Recomputation
- where 2. and 3. Are orthogonal

Computing Activation Memory

- Setup/Notation

Table from Korthikanti et. al

a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

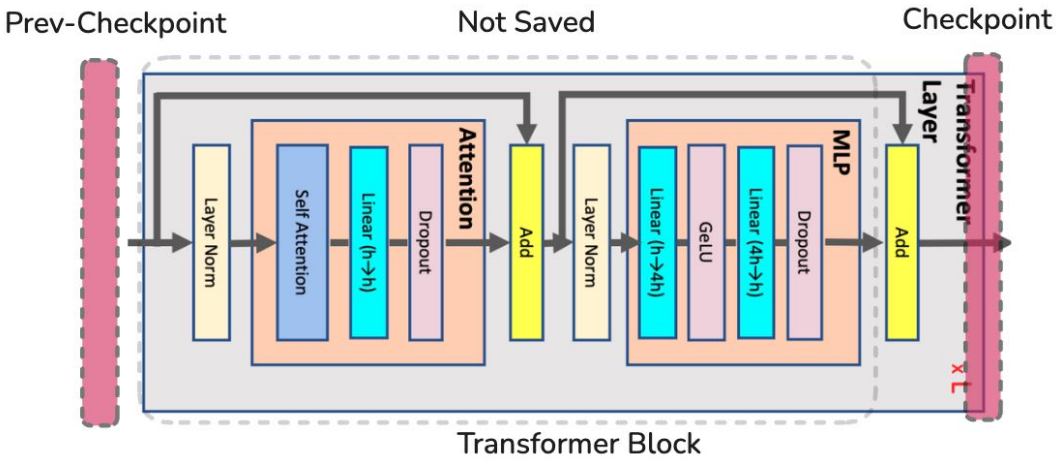
- Fp16 $\rightarrow$ each element takes 2 bytes, except dropout mask takes 1 byte

Agenda: Computing Activation Memory

- Full Activation Recomputation (Lowest)
- Full Activation (Highest)
- w/ Tensor Parallelism
- w/ Sequence Parallelism
- w/ Selective Activation Recomputation

Full Activation Recomputation

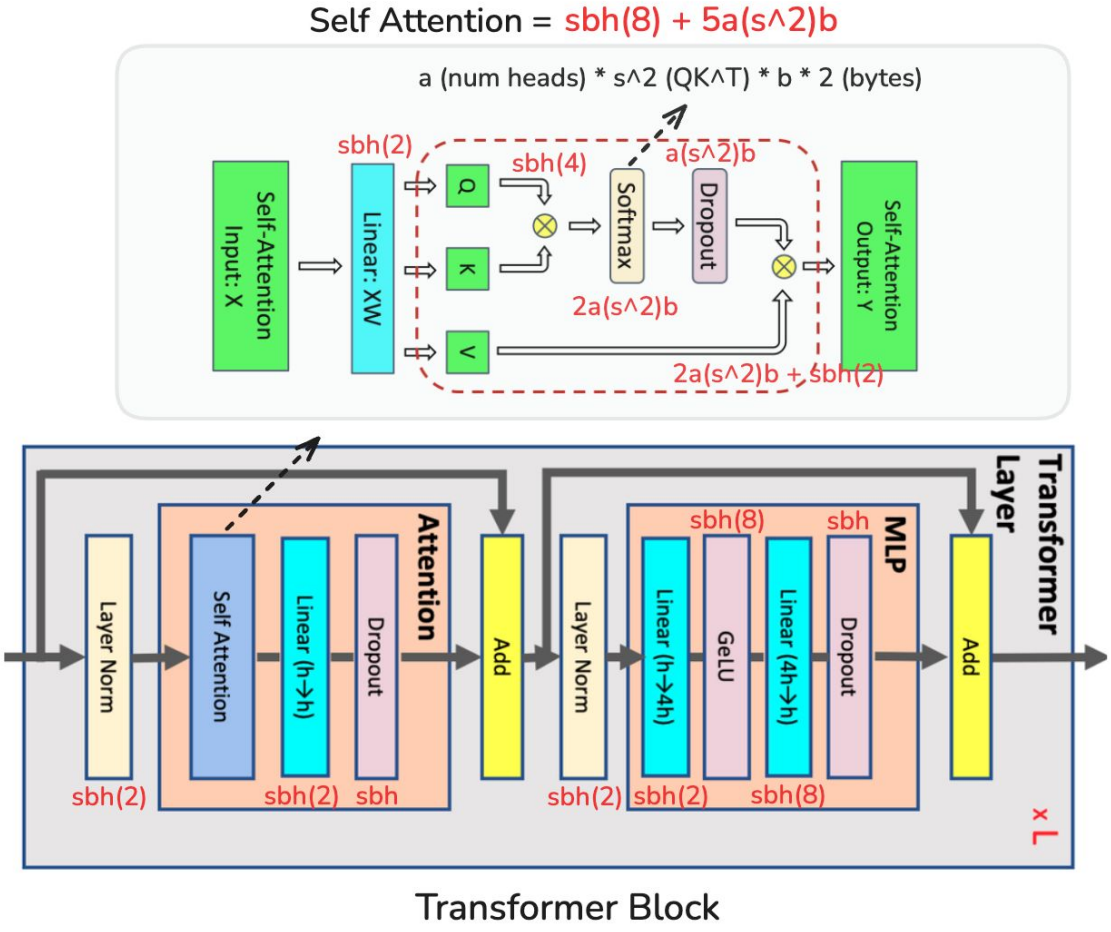
- Memory (per layer): sbh(2)



Full Activation

Memory (per layer)

$$sbh\left(34+5\frac{as}{h}\right)$$

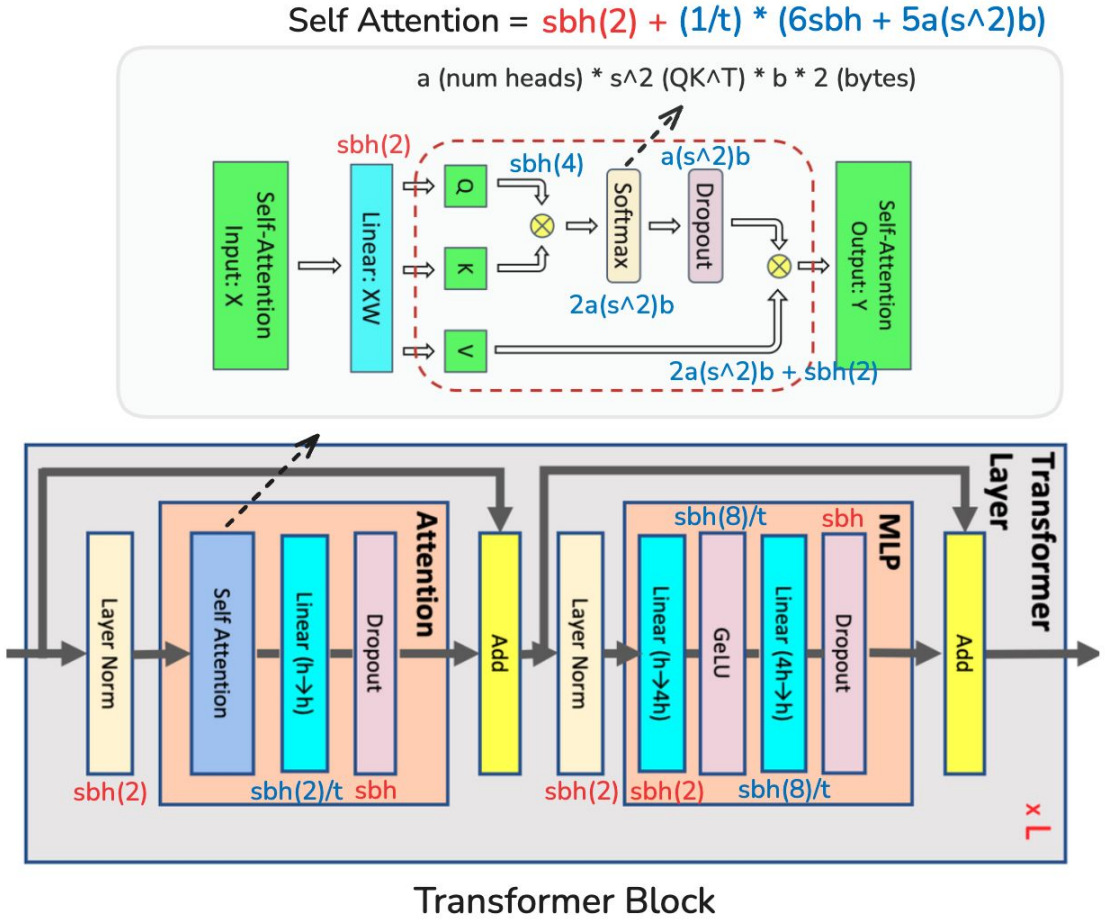


Tensor Parallelism

Memory (per layer)

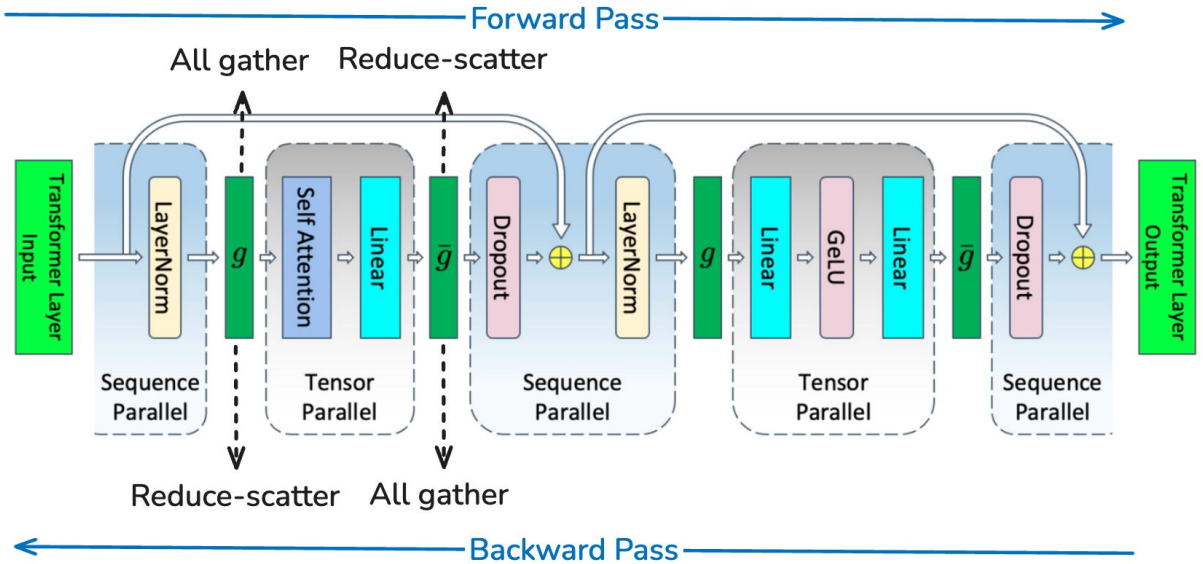
$$sbh\left(10+\frac{24}{t}+5\frac{as}{ht}\right)$$

Can we do better?



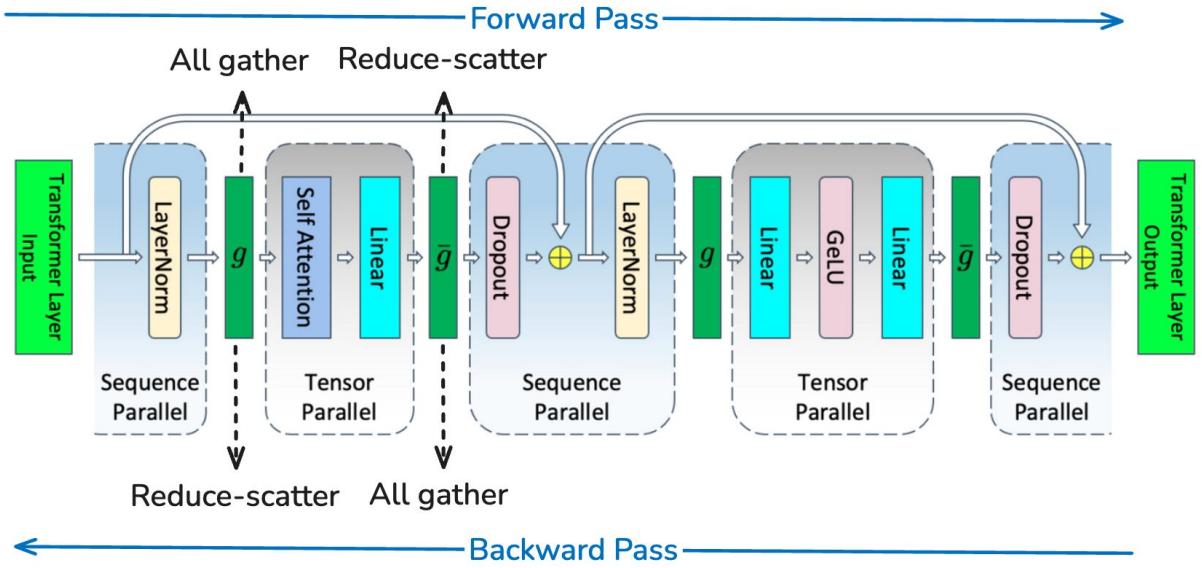
+ Sequence Parallelism

Main Observation – Those *tensor-parallelism-excluded* components are not *sequence-dependent*



+ Sequence Parallelism

Does it increase extra communication by all-gather + reduce-scatter? No, because ring all reduce = all-gather + reduce-scatter

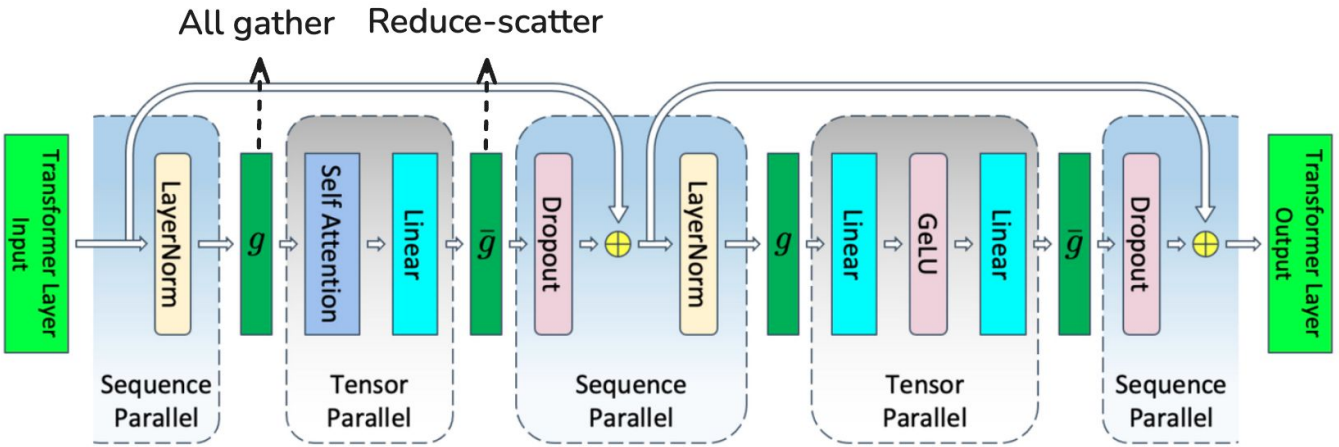


+ Sequence Parallelism

We are not yet achieving 10 / t

Solution: save only the i-th (sequence-wise) proportion on device. Then, introduce an additional all-gather to gather all sequence-wise proportions in backward pass. (overlap the communication with gradient calculation)

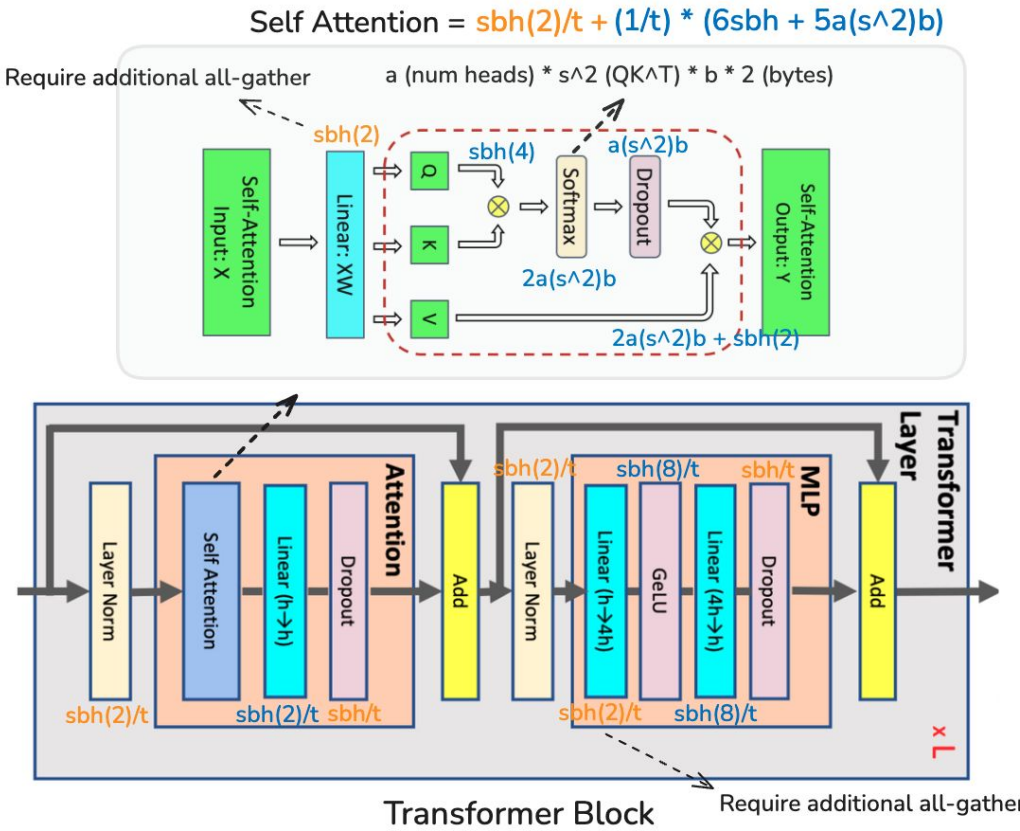
Each device is still getting Full Input (sequence portion)!



+ Sequence Parallelism

Memory (per layer)

$$sbh\left(\frac{10}{t}+\frac{24}{t}+5\frac{as}{ht}\right)=\frac{sbh}{t}\left(34+5\frac{as}{h}\right)$$

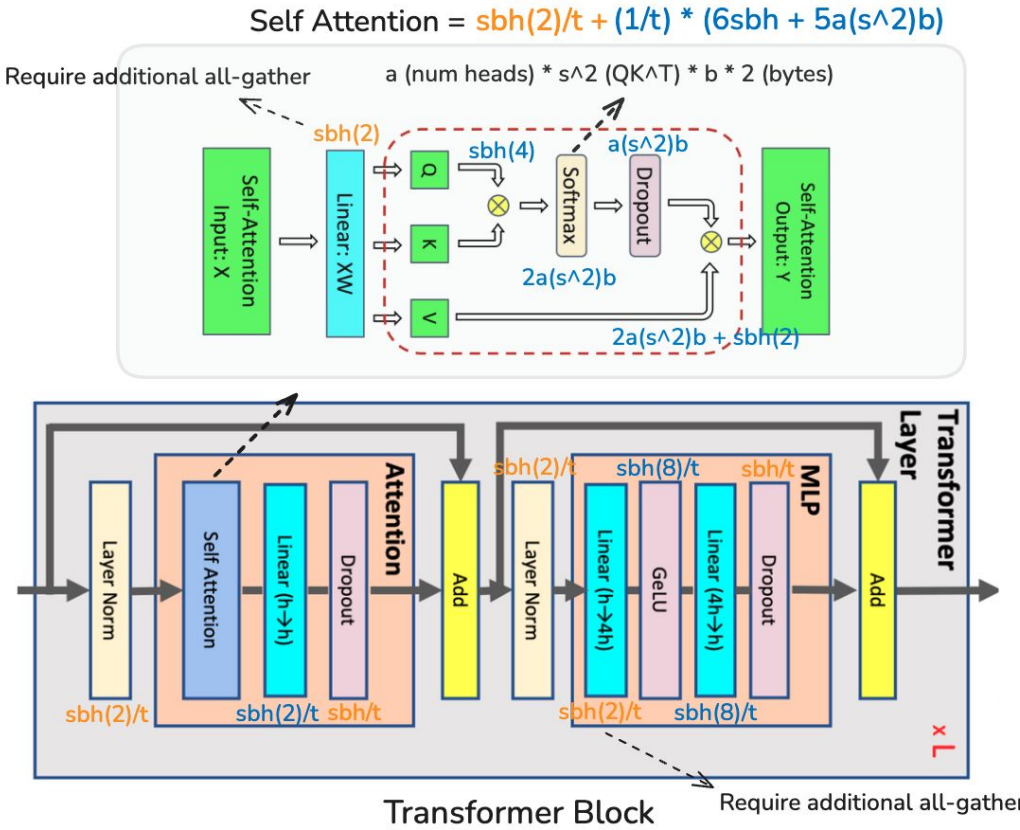


+ Sequence Parallelism

Memory (per layer)

$$sbh\left(\frac{10}{t}+\frac{24}{t}+5\frac{as}{ht}\right)=\frac{sbh}{t}\left(34+5\frac{as}{h}\right)$$

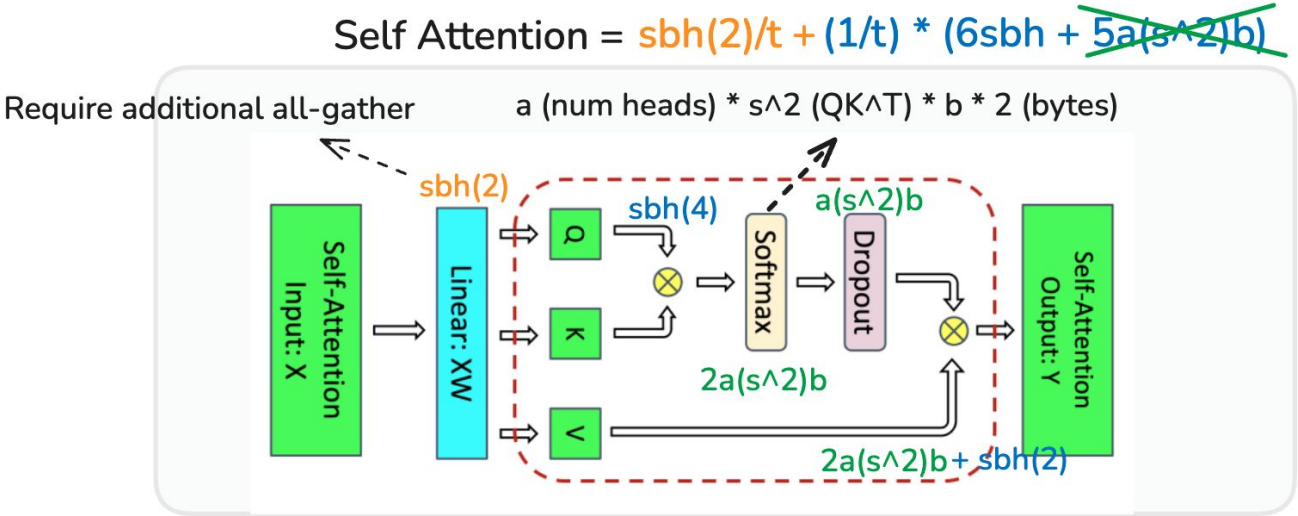
Can we do **even** better?



+ Selective Activation Recomputation

- Recompute the portion isn't too expensive
- Trade-off those recomputation for better memory saving

$$sbh(\frac{34}{t})$$

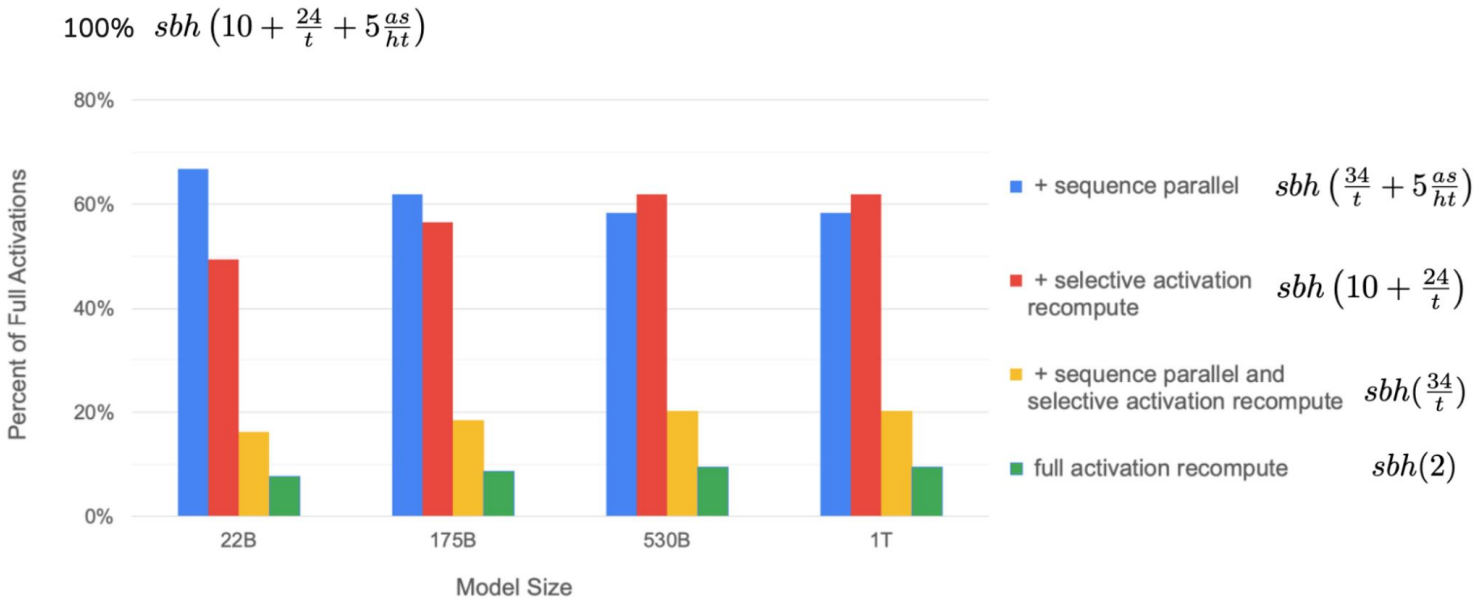


Summary

- Tensor parallelism saves parallelized parts intuitively
- Sequence parallelism saves the rest (some trade-off)
- Selective Recomputation trades off compute for large memory reduction (s^2)
  - Pointer → FlashAttention solves this problem nowadays!

Evaluation

- The terms reduced matter! (linearly reducing the 10 with number of devices)



Evaluation

- Select the right thing to recompute, introducing much less overhead (while saving much more memory)

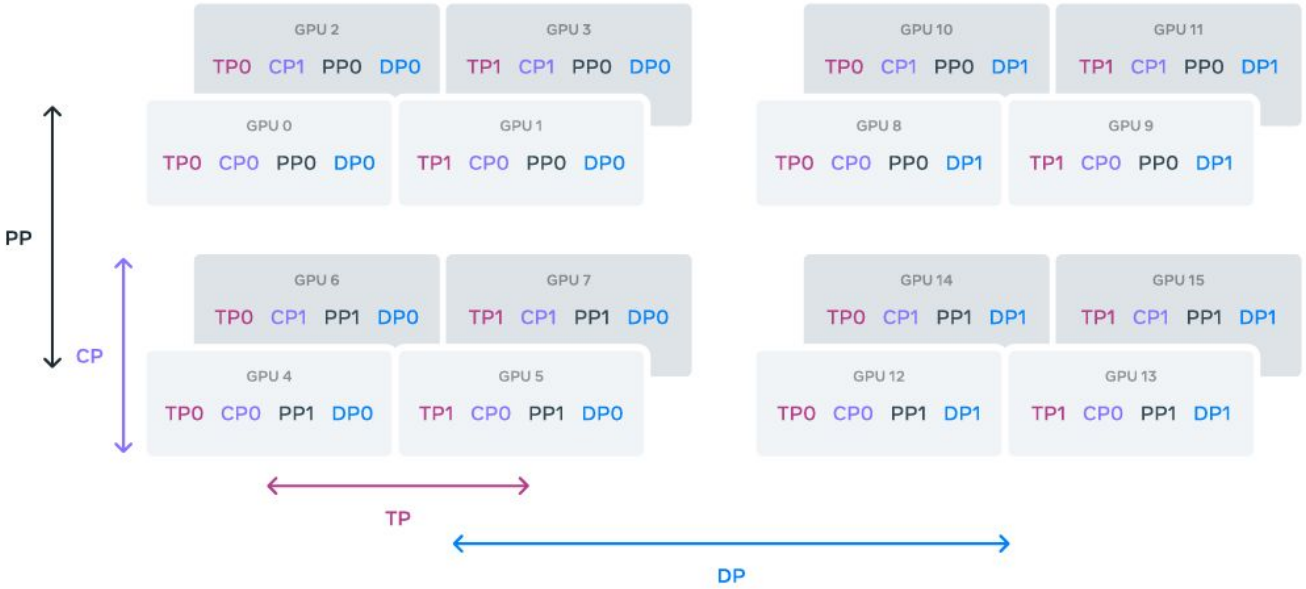
Experiment	Forward (ms)	Backward (ms)	Combined (ms)	Overhead (%)
Baseline no recompute	7.7	11.9	19.6	–
Sequence Parallelism	7.2	11.8	19.0	−3%
Baseline with recompute	7.7	19.5	27.2	39%
Selective Recompute	7.7	13.2	20.9	7%
Selective + Sequence	7.2	13.1	20.3	4%

Table 4: Time to complete the forward and backward pass of a single transformer layer of the 22B model.

Question

- Can we do even even better? What components discussed were not optimized yet?
- Can we do an analysis on overhead of recomputation (over components) and memory reduction to *fit even smaller device*?

Combined Parallelism in real world training



Training LLama3 in 16k GPUs

The Llama 3 Herd of Models, arxiv:2407.21783

Context Parallel

- In Data Parallelism, we shard activation (for attention and FFN) in alone batch dimension, how about long context training? Where we are unable to shard efficiently since batch size is small
- In Sequence Parallel, only the non-attention parts are sharded across devices. The attention part still remains the full sequence
- *What about extremely long context (1M tokens)* -> Context Parallel!!! Next Lecture!!

Expert Parallelism

- MoE layer contains multiple expert networks
- Example: 32 experts distributed across 8 GPUs (4 experts/GPU)
- Tokens routed to different GPUs based on expert selection
- Each token typically activates k experts (e.g., k=2)

Expert Parallelism

- Conceptually still a form of model parallelism
- Differences
  - Requires specialized routing implementation
  - Dynamic load balancing challenges
  - Token dropping and auxiliary losses
  - Different communication patterns (all-to-all vs all-reduce)

Reference

[1] Ultra-Scale Playbook: https://huggingface.co/spaces/nanotron/ultrascale-playbook

[2] UCSD CSE 234: https://hao-ai-lab.github.io/cse234-w25/

[3] The Llama 3 Herd of Models, arXiv:2407.21783