LLM Distributed Training

Part B

Autumn 2026

Lecturer: Yuedong (Steven) Xu

Shenzhen Loop Area Institute

yuedongxu@slai.edu.cn

Disclaimer

Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blogposts, research talks, tutorial videos, and other materials shared by the research community. Part of the content are also generated by AI.

Distributed LLM Training

- Model Parallelism
  - **Pipeline Parallelism**
  - Tensor Parallelism
  - Sequence Parallelism
- Mixture of Experts
  - MoE Basics
  - Expert Load Balancing

Pipeline parallelism starts with a memory constraint

What must fit on one GPU?

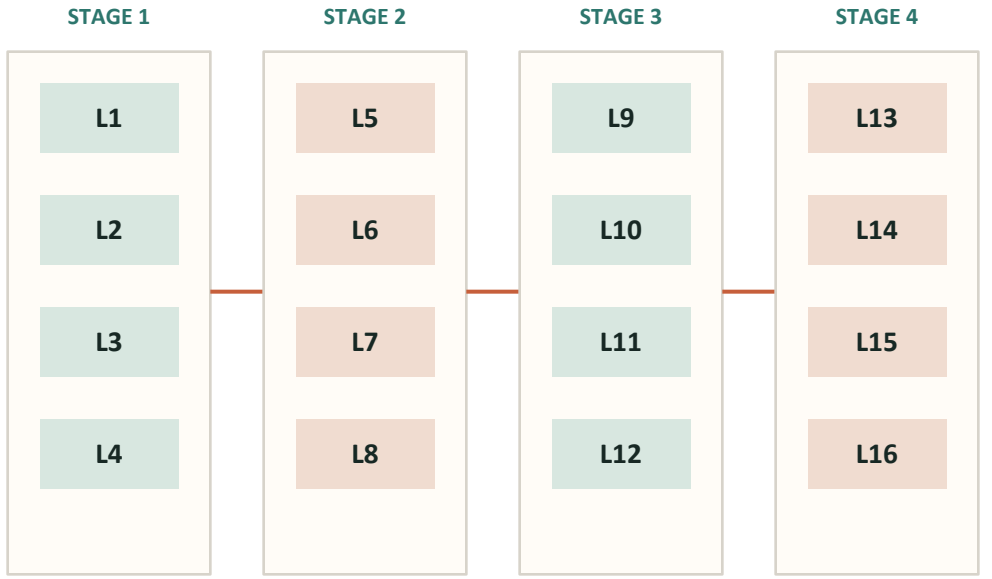
SCHEMATIC TRAINING-STATE FOOTPRINT



- Parameters
- Gradients
- Optimizer states
- Activations + buffers

Pipeline stages partition layer-owned state; activations cross stage boundaries.

Partition the layer sequence

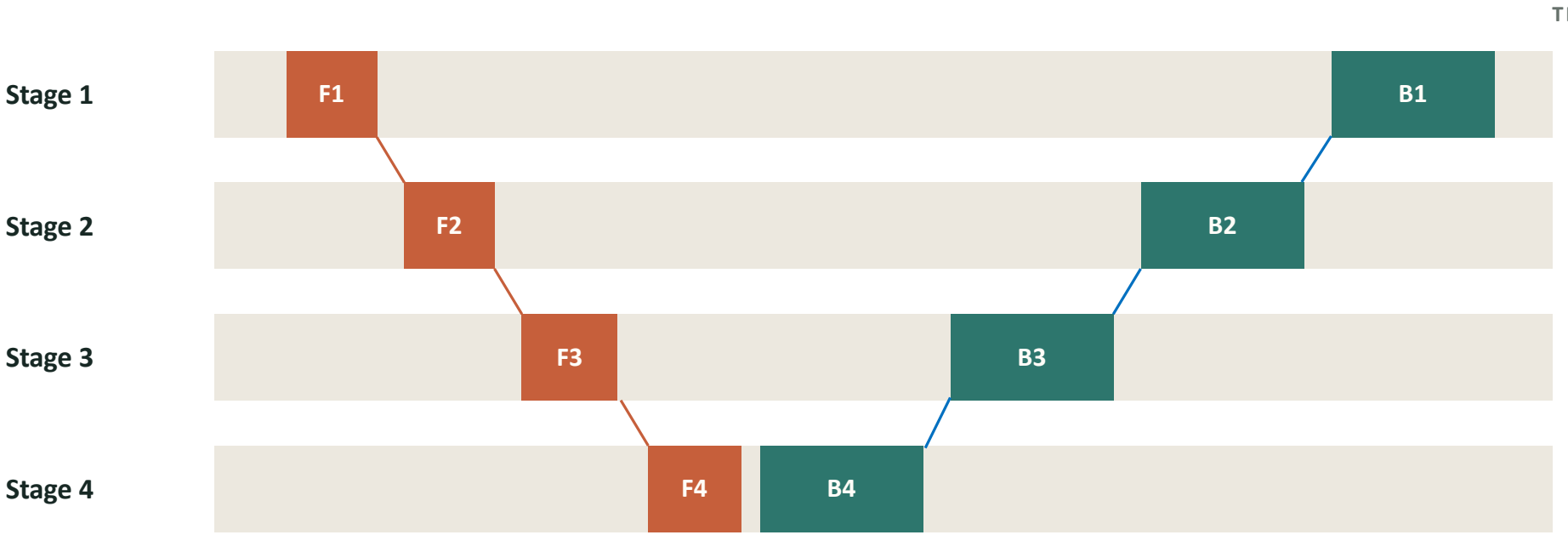


Each device owns a ***contiguous*** stage. A microbatch carries activations to the next stage and input gradients back.

Layer counts are illustrative, not a partitioning recipe.

Without microbatches, the stages mostly wait

One microbatch · four stages · forward then backward



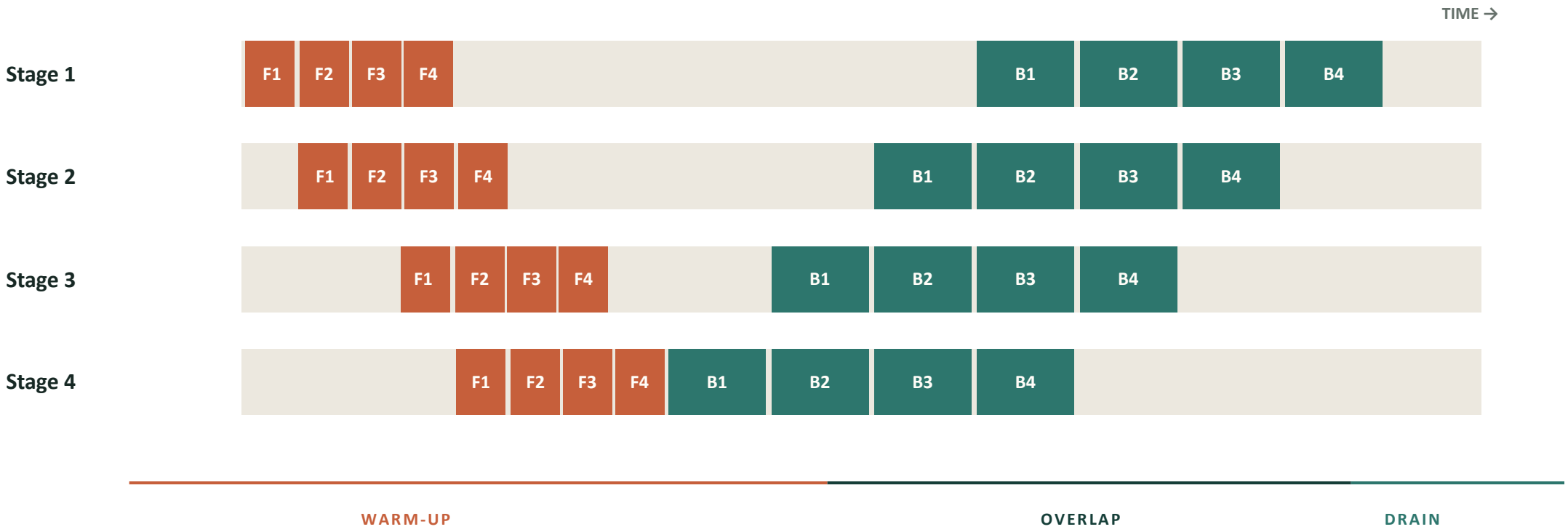
Only one stage computes at a time. The other devices wait for dependencies.

Point-to-Point (P2P) traffic:
activations →
input gradients ←

Source: GPipe · Huang et al., 2018

Microbatches let different stages work simultaneously

Split one global batch into M independent microbatches; each still traverses all K stages.



A vertical slice contains work for different microbatches—but never two operations on one stage.

Source: GPipe · Huang et al., 2018

More microbatches shrink the idealized bubble

Assume **K equal-cost stages and M equal-cost microbatches**; ignore communication and load imbalance.

IDEALIZED F-THEN-B MODEL

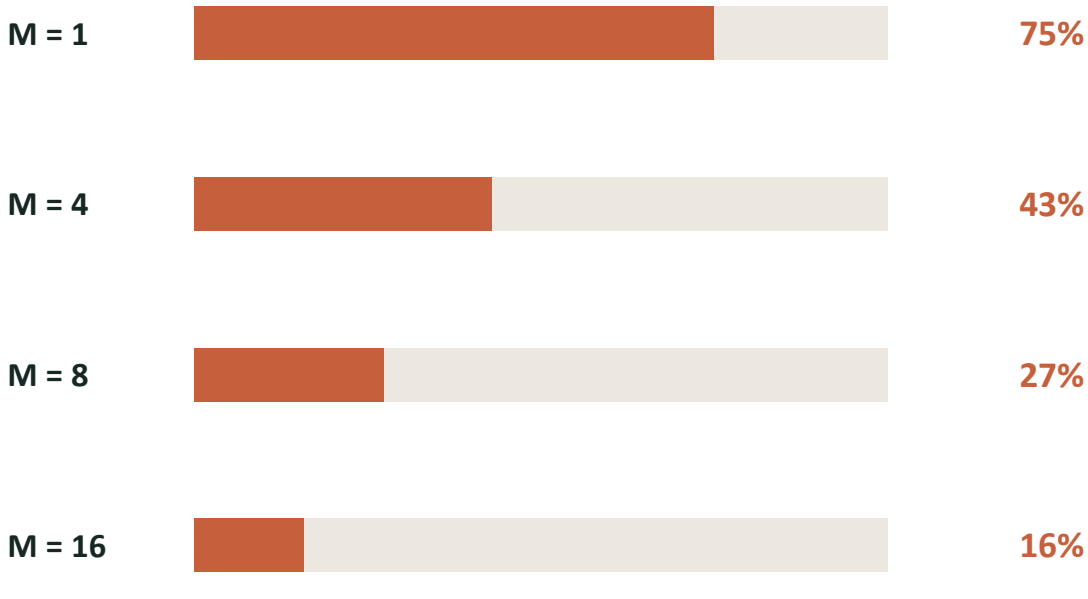
bubble fraction = (K – 1) / (M + K – 1)

For K = 4 and M = 8:

3 / 11 ≈ 27% of the idealized timeline is bubble.

Increasing M improves utilization. Under F-then-B, it also increases the number of activation sets that may remain live.

Bubble share when K = 4



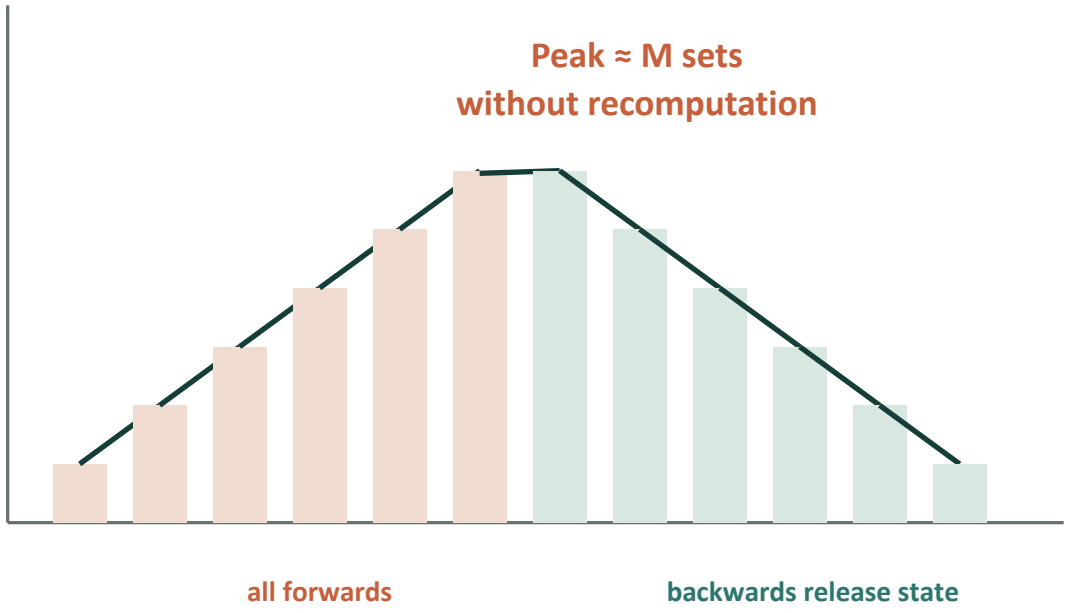
Colored portion = idle fraction

Source: GPipe · Huang et al., 2018; calculation shown is illustrative

F-then-B pays for simplicity with activation memory

A stage must preserve the boundary state needed to compute its backward pass.

In-flight activation sets at one stage



RETAIN

Output activations and backward metadata remain live until their microbatch reaches backward.

RECOMPUTE

Discard selected intermediates, then rerun forward work during backward to reduce peak memory.

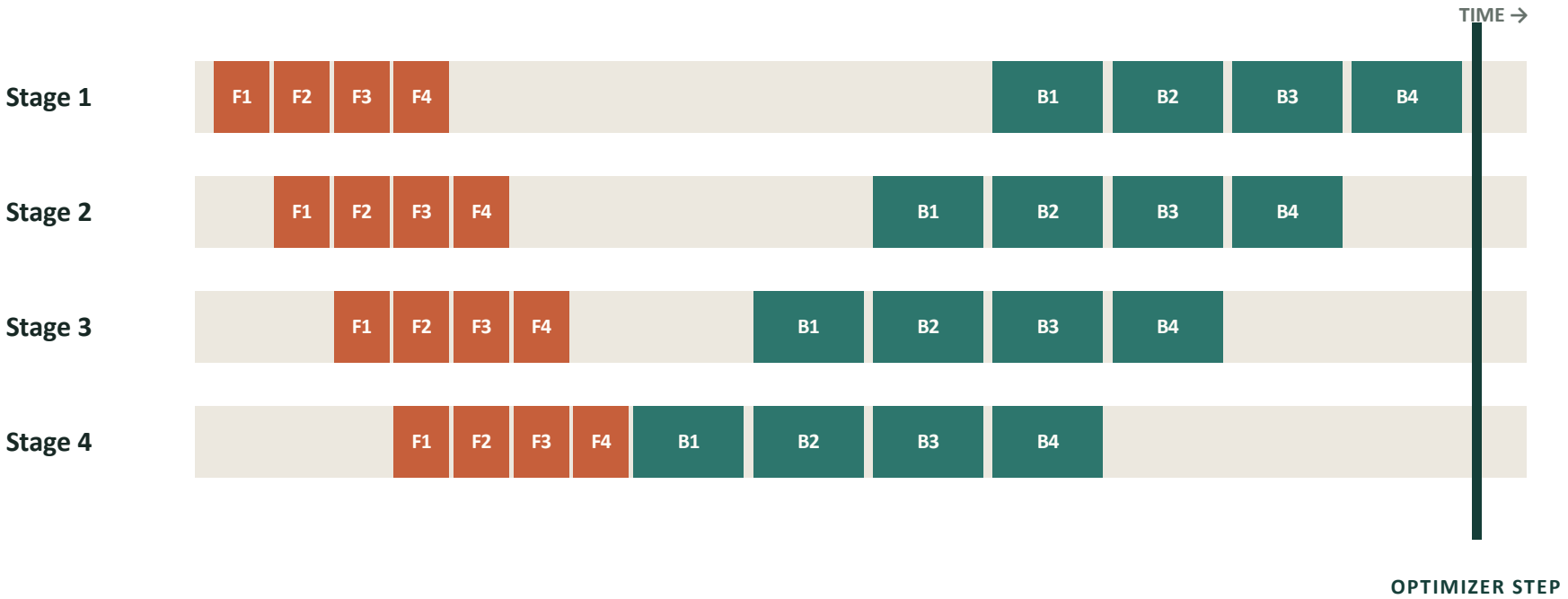
PAY

Recomputation lowers memory by adding compute; it does not remove stage dependencies.

Source: GPipe · Huang et al., 2018

A flush restores consistent weights—and drains the pipe

Run the batch → wait for all stages → update weights → refill the pipeline



CONSISTENCY

All work before the barrier observes the intended synchronous update boundary.

COST

Warm-up and cool-down recur at every optimizer step; no stage can hide the global drain.

A flush is a semantic operation, not just a scheduling inconvenience.

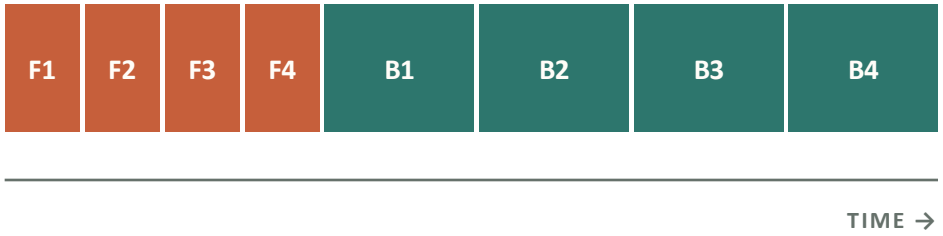
Source: GPipe · Huang et al., 2018

1F1B Changes When Backward Starts

Compare the last stage; block widths use F = 1 and B = 2 time units.

F-THEN-B

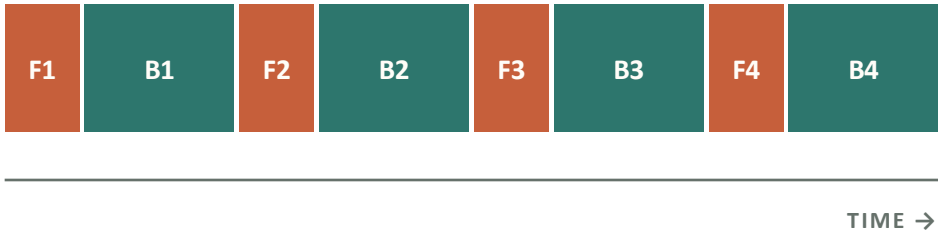
Finish every forward first



B1 waits until F2–F4 have also completed on this stage. Saved activations accumulate.

1F1B

Alternate once warm-up ends



B1 follows F1 immediately. Once B1 finishes, this stage can release microbatch 1’s activations.

CORE DISTINCTION **Earlier backward starts shorten activation lifetime while preserving synchronous updates.**

Source: PipeDream · Harlap et al., 2018

1F1B: Start Backward Earlier

Warm up with forwards, then alternate one forward and one backward on each stage.

FOUR STAGES · EIGHT MICROBATCHES



The microbatches 8 leads to a bubble ratio of 27.3% — warm-up and cool-down cost is fixed per step.

01 Warm-up

Run stage-dependent forward passes until the first backward can start.

02 Steady state

Alternate F and B. Each backward frees the microbatch's activations.

03 Cool-down

Drain the remaining backwards, then synchronize before the optimizer step.

MEMORY BENEFIT

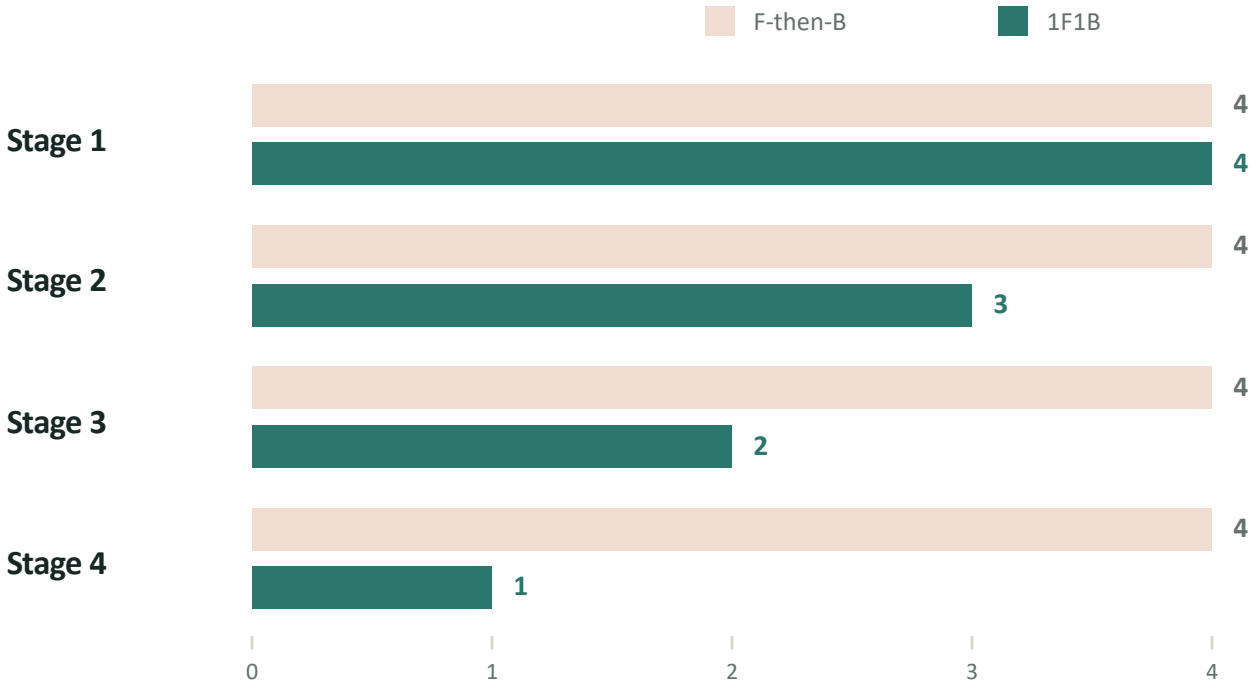
Peak activation storage stays ≤ p in-flight microbatches per stage — more microbatches shrink the bubble, not the memory.

Source: PipeDream · Harlap et al., 2018

Why 1F1B Reduces Peak Activation Memory

Backward consumes the saved activations, allowing stage-local storage to be released.

OUTSTANDING MICROBATCH ACTIVATIONS · K = M = 4



Forward

Save the intermediate activations needed to compute gradients later.

Backward

Use those activations, then free them for that microbatch on this stage.

NON-INTERLEAVED WARM-UP

Stage index i: $K - i - 1$ warm-up forwards

BUBBLE RATIO

$$\frac{K-1}{M+K-1}$$

PEAK MEMORY

per-stage peaks change from 4, 4, 4, 4 to 4, 3, 2, 1.

Source: PipeDream · Harlap et al., 2018

Interleaving Splits Each Stage into Two Chunks

16 layers · 4 physical stages · 2 chunks per stage · 2 layers per chunk

LOGICAL CHUNK ORDER FOR ONE MICROBATCH

C1	C2	C3	C4	C5	C6	C7	C8
L1–L2	L3–L4	L5–L6	L7–L8	L9–L10	L11–12	L13–14	L15–16
STAGE 1	STAGE 2	STAGE 3	STAGE 4	STAGE 1	STAGE 2	STAGE 3	STAGE 4

Forward: C1 → C2 → C3 → C4 → C5 → C6 → C7 → C8

Backward: C8 → C7 → C6 → C5 → C4 → C3 → C2 → C1

C4 → C5 wraps communication from Stage 4 back to Stage 1.

PHYSICAL STAGE OWNERSHIP

Stage 1	Chunk A · C1 · layers 1–2	Chunk B · C5 · layers 9–10
Stage 2	Chunk A · C2 · layers 3–4	Chunk B · C6 · layers 11–12
Stage 3	Chunk A · C3 · layers 5–6	Chunk B · C7 · layers 13–14
Stage 4	Chunk A · C4 · layers 7–8	Chunk B · C8 · layers 15–16

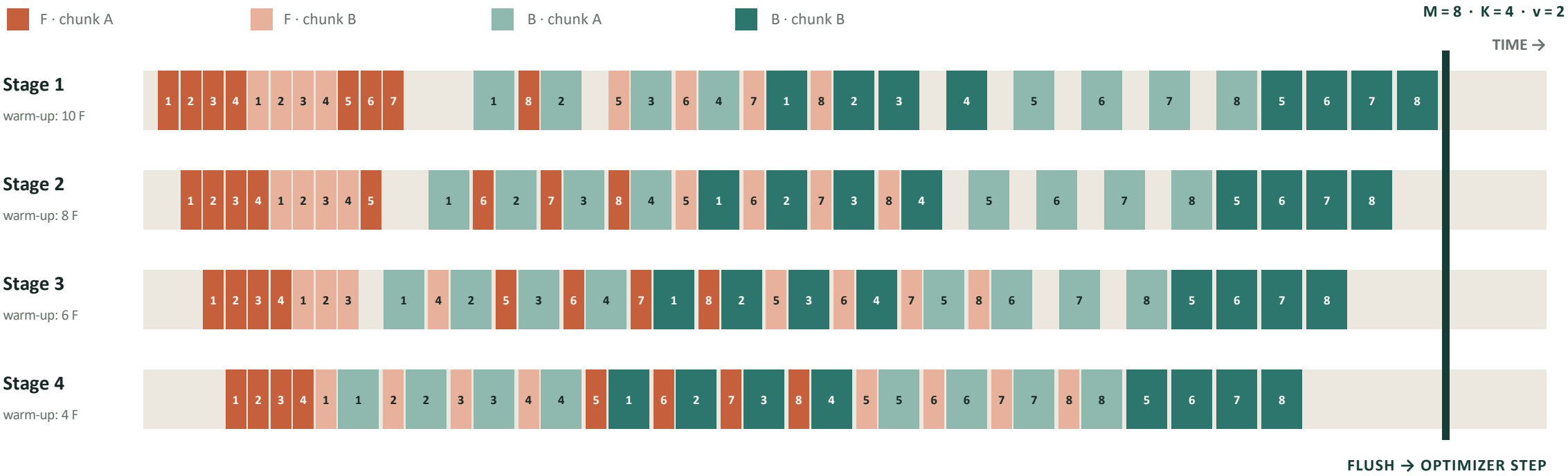
KEY IDEA

Four physical stages hold eight logical chunks; each stage stores four layers in two non-contiguous chunks.

Source: Megatron-LM · Narayanan et al., 2021

Interleaved 1F1B: Eight Microbatches, One Flush

Numbers identify microbatches; colors identify operation and local chunk. Idealized timing: F = 1, B = 2.



Idealized; communication ignored: 16 F + 16 B chunk operations per stage = 48 work units · Makespan 57 · Idle 9

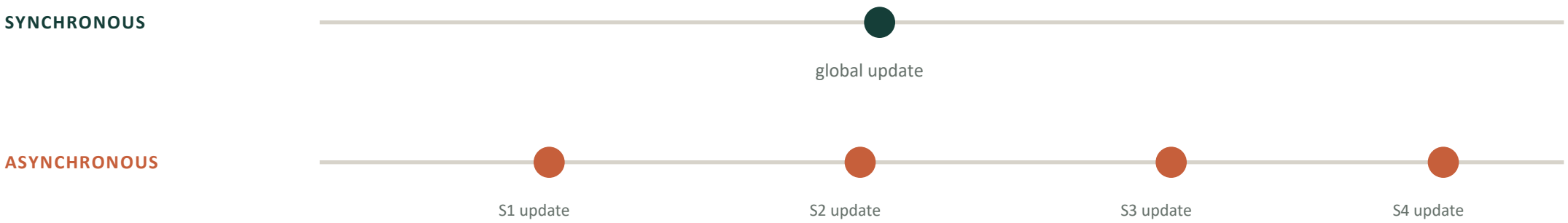
SYNCHRONOUS FLUSH

No weight update occurs until every microbatch finishes backward through both chunks on every stage.

Source: Megatron-LM · Narayanan et al., 2021

Asynchronous updates remove flushes, not consistency problems

A stage can update its local weights while other stages continue processing microbatches.



Three version questions

STALENESS

How many updates separate the forward weights from the weights used to compute a gradient?

LOCAL MATCH

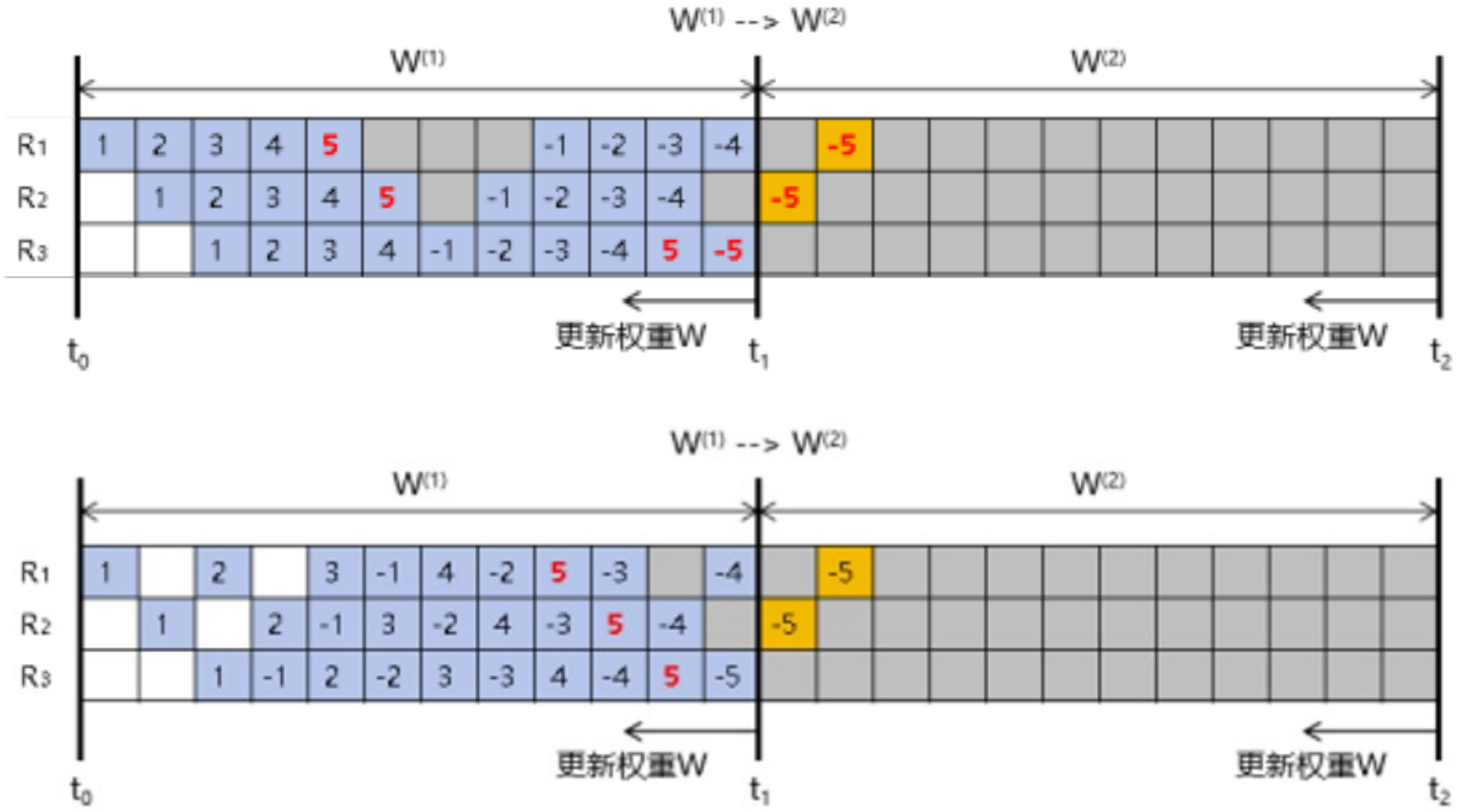
Do forward and backward for one microbatch use the same version within a stage?

GLOBAL MATCH

Do all stages see mutually consistent versions for that microbatch?

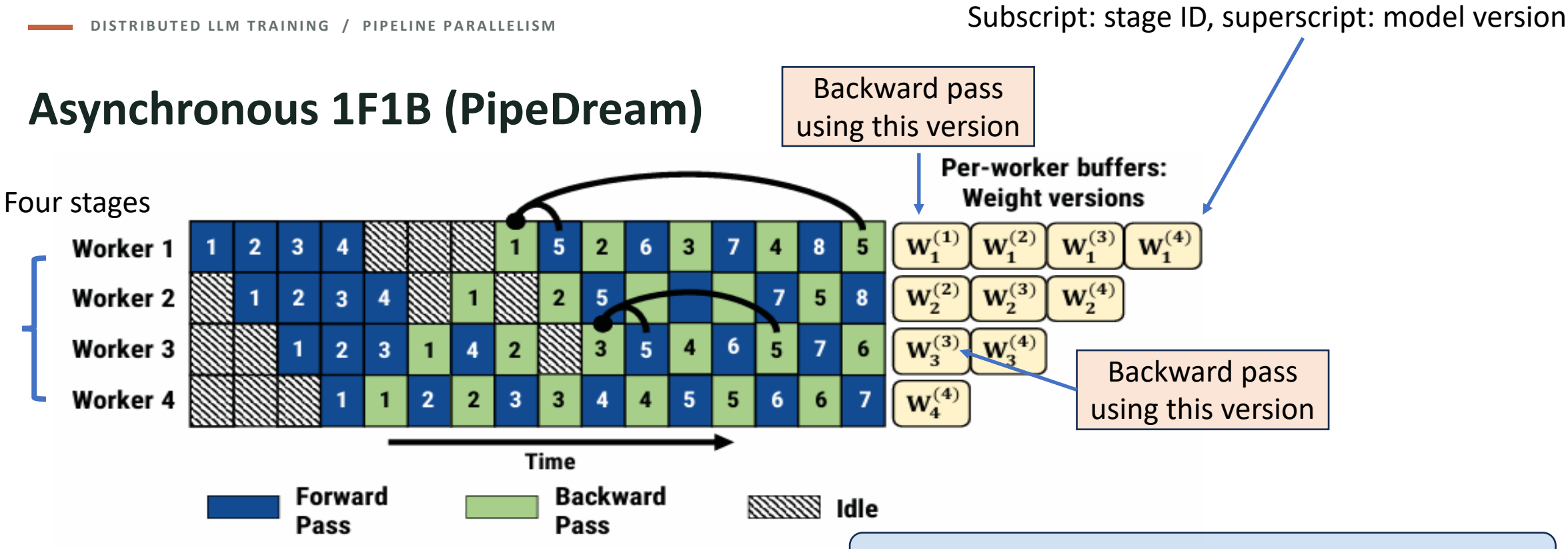
Source: PipeDream · Narayanan et al., 2018

Removing Flush Barrier



Removing flushing barrier causes misaligned model versions

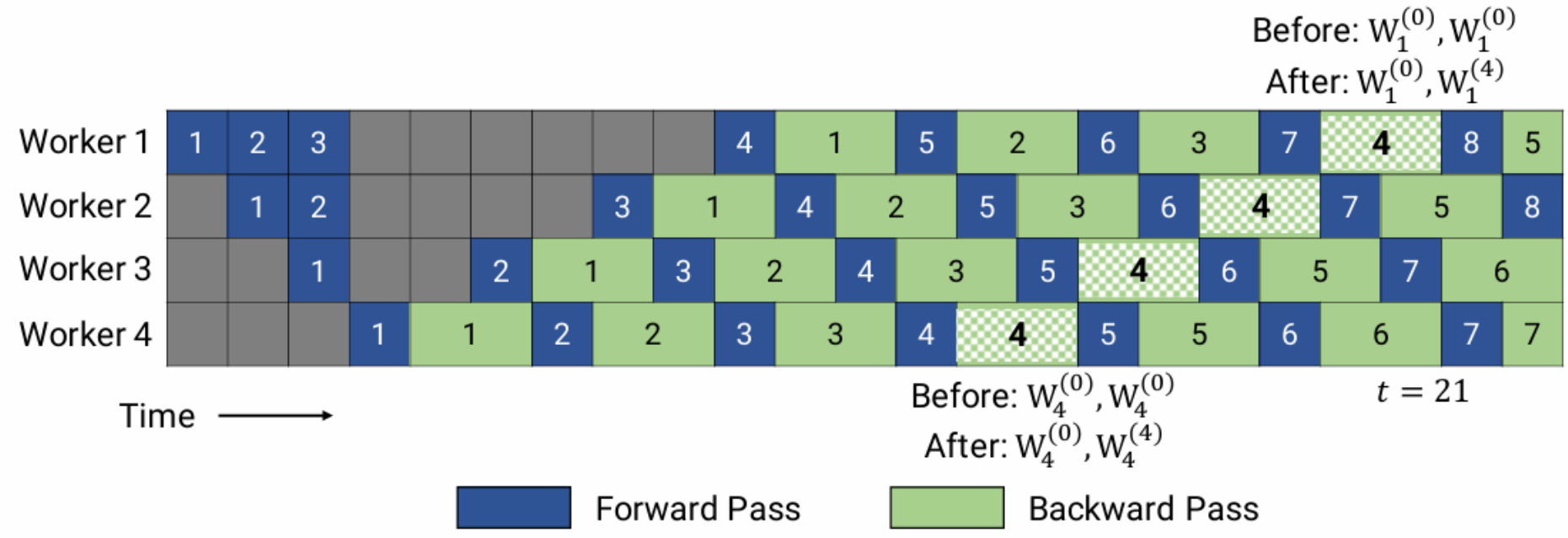
Asynchronous 1F1B (PipeDream)



No guarantees across stages!

- ***Weight stashing*** of PipeDream
  - Keeps multiple versions of model weights—one for each active minibatch.
  - Forward pass: each pipeline stage uses the most recent weight version available. Once the forward pass for a minibatch is completed, the corresponding weight version is stored.
  - Backward pass: the same version is reused to compute gradients and update weights, ensuring that both passes for a given minibatch use identical parameters within a stage.

Asynchronous 1F1B-2BW (PipeDream-2BW)



PipeDream-2BW:

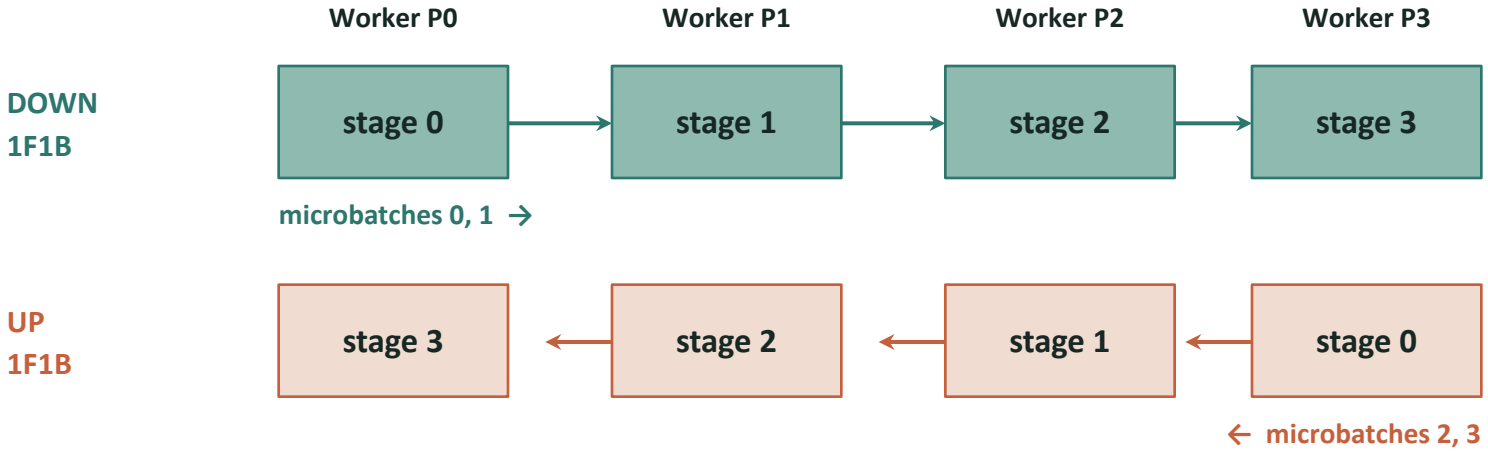
- **less frequent flushing than Gpipe**
- **keeping no more than two model versions.**

Think why? How does it relate to pipeline depth K and number of microbranches M

Chimera Runs Two 1F1B Pipelines in Opposite Directions

Four workers · four logical stages · N = 4 microbatches, split evenly between down and up pipelines

WORKER / STAGE OWNERSHIP



Gradient all-reduce for the same logical stage

S0: P0 + P3 · S1: P1 + P2 · S2: P2 + P1 · S3: P3 + P0

ONE SYNCHRONOUS ITERATION

1. Split N microbatches evenly between directions.
2. Run N/2 microbatches per direction with 1F1B.
3. Merge both streams on the same D workers.
4. All-reduce stage gradients, flush, then update.

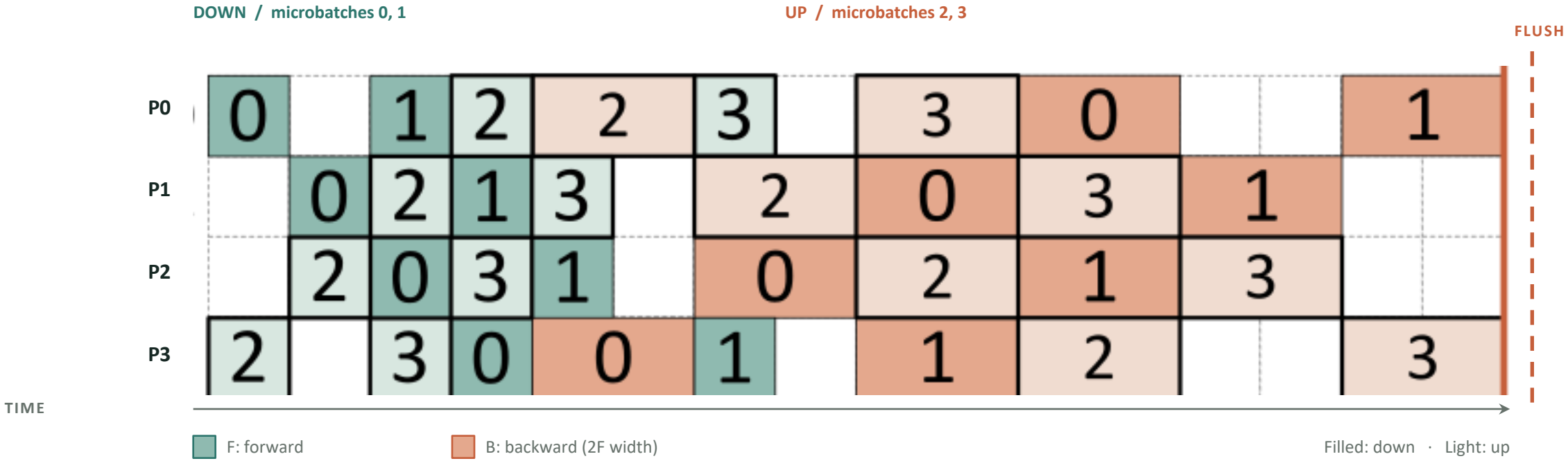
Cost: each worker stores weights for two logical stages; activation peaks are more balanced.

Chimera reuses the same workers and keeps a synchronous flush: no PipeDream-style weight stashing, but two logical stage weights per worker.

Source: Li & Hoefler, Chimera (SC 2021), .

Two Directions Fill Each Other's Idle Slots

D = 4, N = 4 · practical paper example: backward workload = 2 × forward workload · numbers identify microbatches



Equal-time bubble count

GPipe / DAPPLE: 2(D−1) → Chimera: D−2

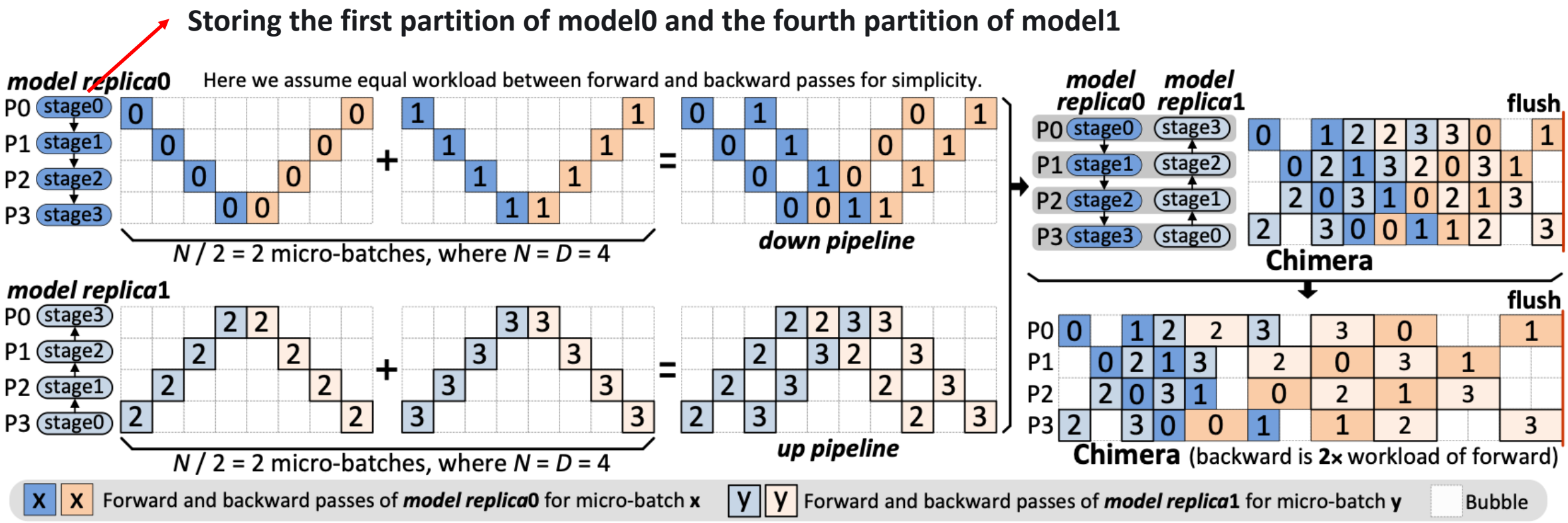
Flush trade-off

Chimera keeps the flush; 2BW flushes less often with ≤2 weight versions.

For D = 4, the idealized forward-plus-backward bubble count falls from six to two; practical gains also depend on communication.

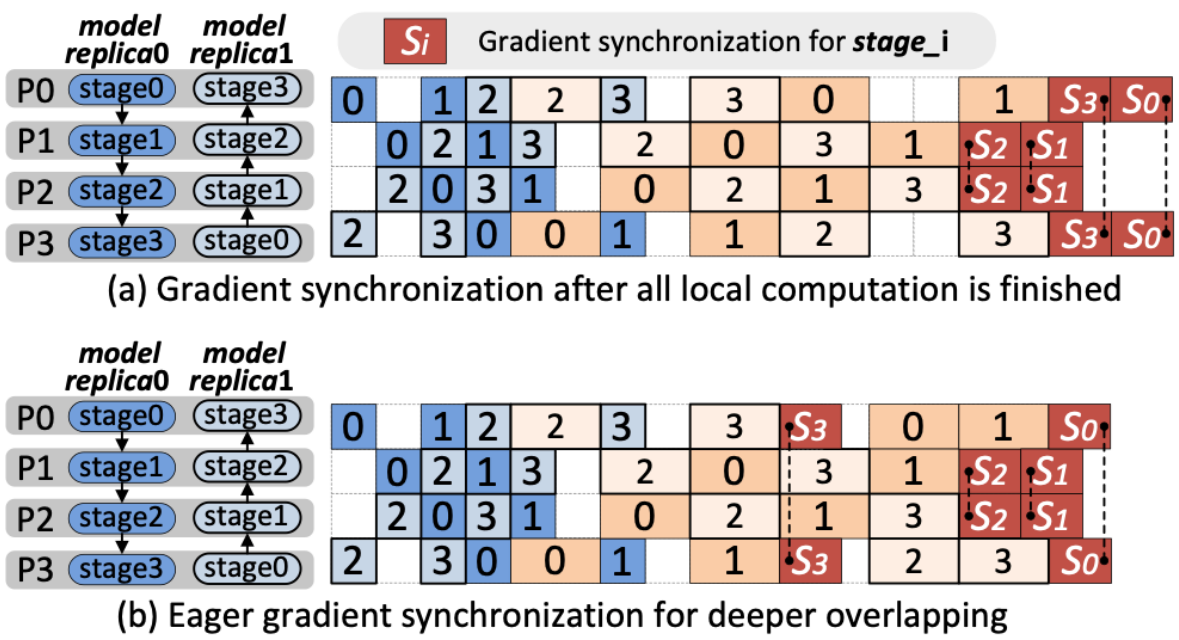
Source: Li & Hoefler, Chimera (SC 2021),

How to Merge Pipelines of Two Directions

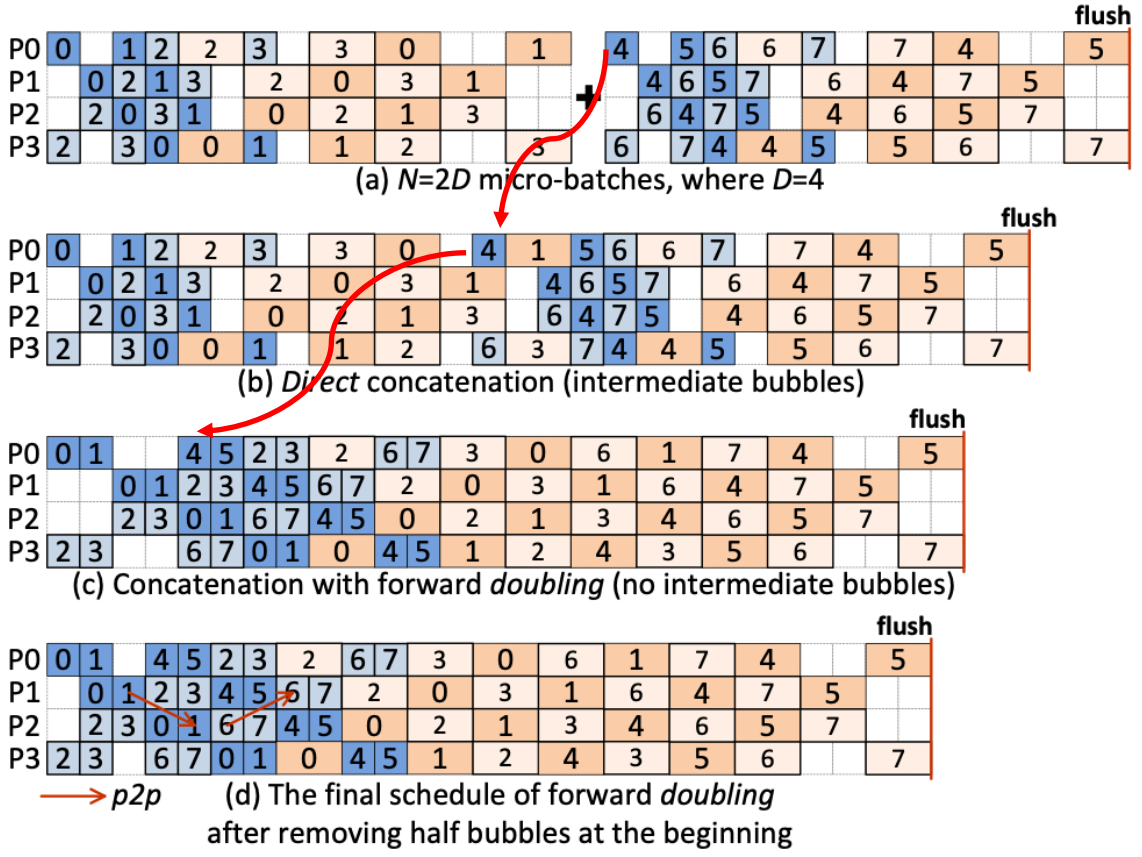


Smaller bubbles yet doubled memory occupation for model replicas and FP/BP activations

How to Merge Pipelines of Two Directions



Hide the gradient synchronization overhead by overlapping them with computations, e.g. shifting S_3 to an empty slot in the left.

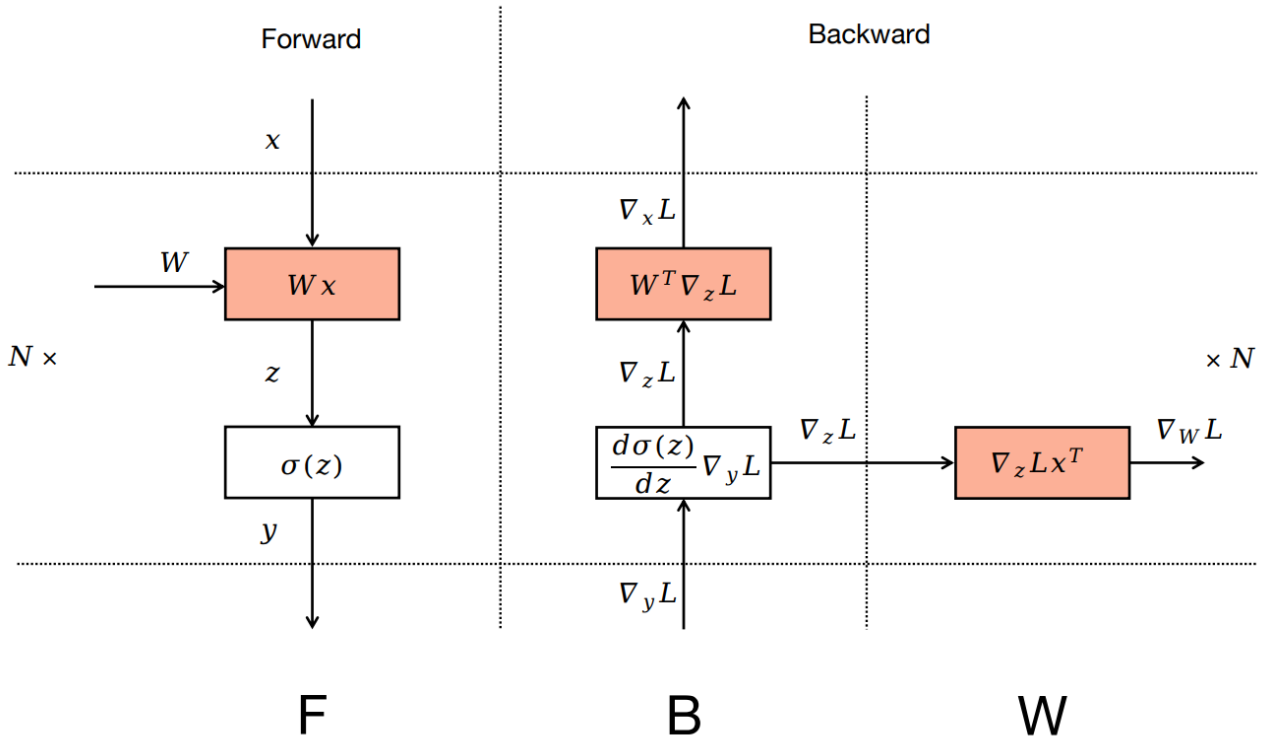


Select # replicas, # stages and Assemble blocks

Chimera with data parallelism

Zero Bubble Splits Backward Into B and W

Separate input-gradient work (B) from weight-gradient work (W)



B: input gradient

Sent to the previous stage; it stays on the inter-stage dependency chain.

W: weight gradient

Needed by the local optimizer, but not immediately by another stage.

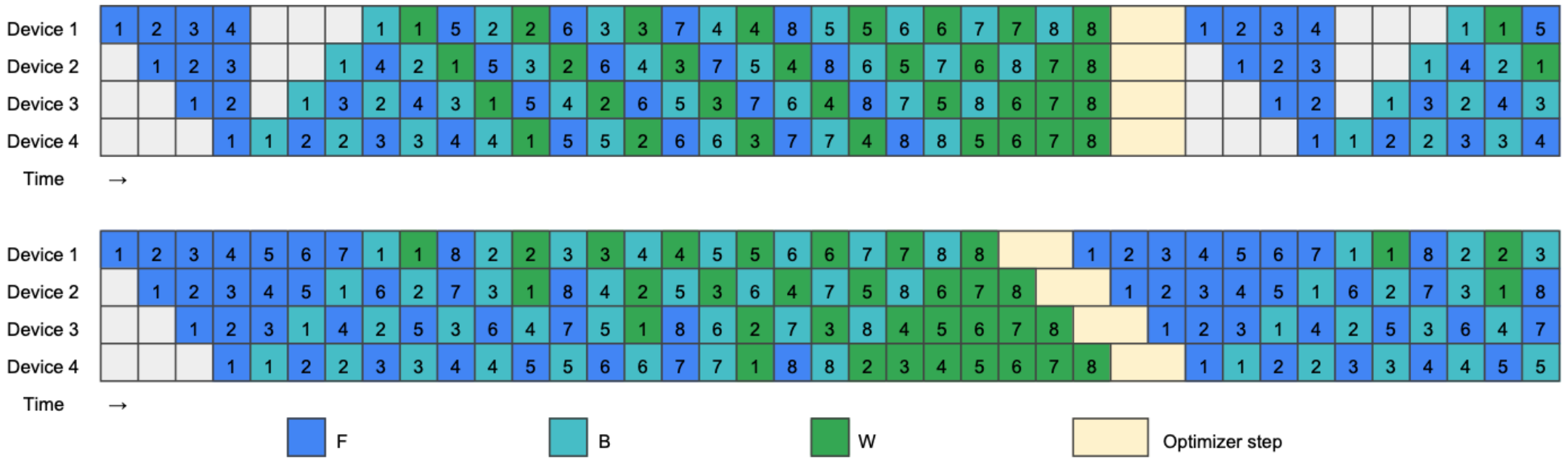
Prioritize B. Defer W into idle slots.
F creates saved state; B releases part; W releases the rest.

Splitting does not remove weight-gradient work—it makes that work schedulable.

Source: Qi et al., Zero Bubble Pipeline Parallelism (ICLR 2024),

Zero Bubble Illustration

Extra warm-up F work fills the head; delayed W fills the tail · p = 4, m = 8 · equal F, B, W costs



ZB-H1 · memory-efficient

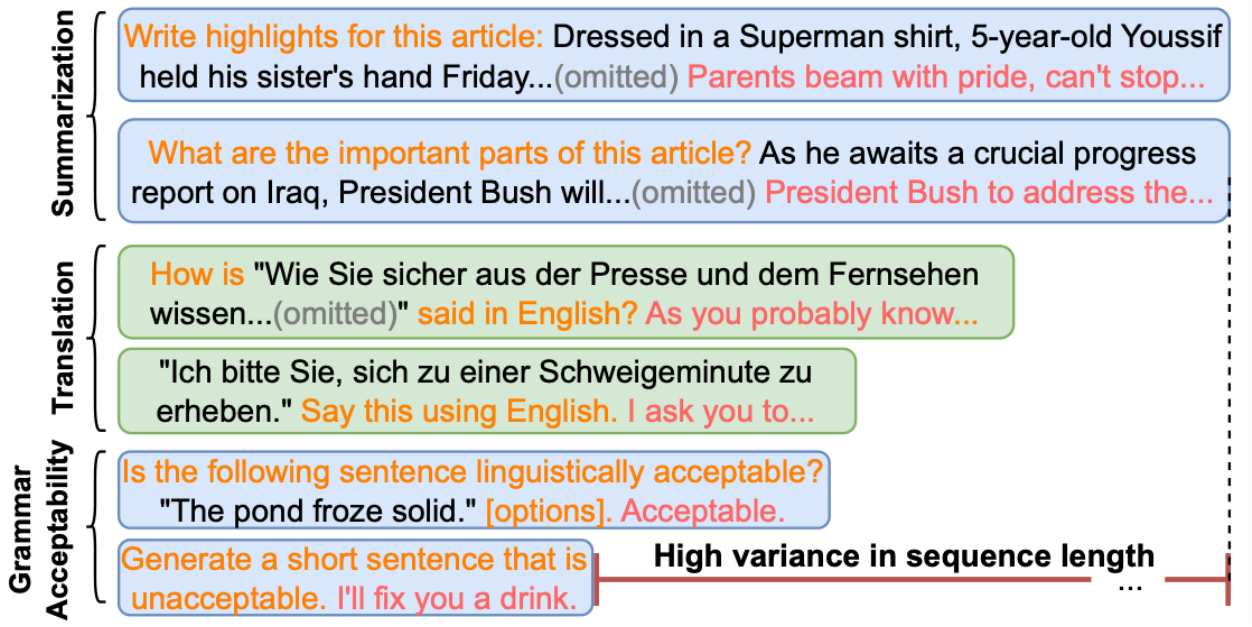
Peak activation memory pM_B; idealized bubble size is about one third of 1F1B.

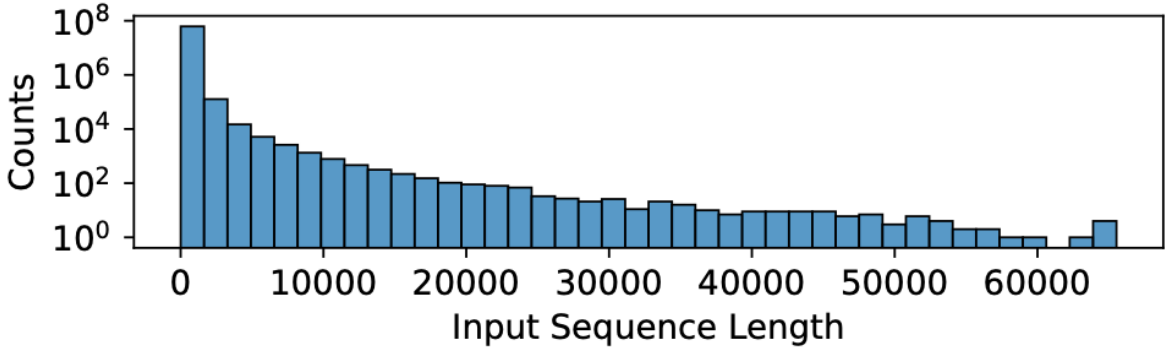
ZB-H2 · zero-bubble target

Peak activation memory $(2p-1)M_B$; needs enough microbatches and optimizer-bypass support.

Source: [Qi et al., Zero Bubble Pipeline Parallelism (ICLR 2024)](#).

Dynamic Sequence Length for Multi-task Training

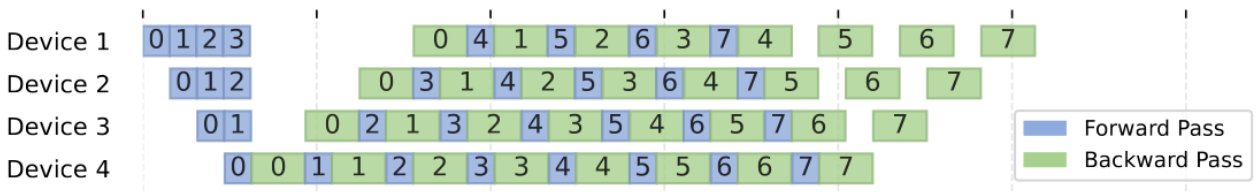




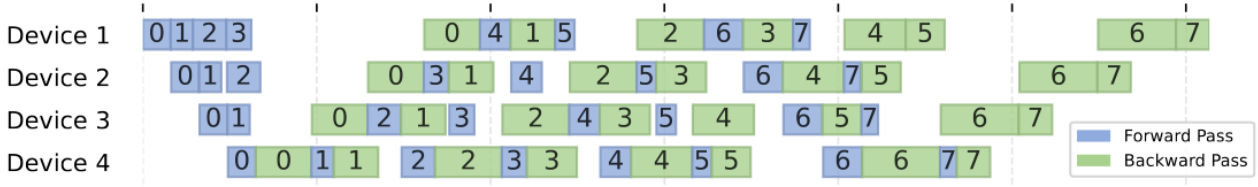
- (left) Orange texts are instructions to the model. Inputs to process are colored black. Expected responses are in red
- (upper) The sequence length distribution in dataset

Randomly packing sequences into regular microbatches may cause inefficient GPU utilization or out-of-memory

Dynamic Sequence Length for Multi-task Training

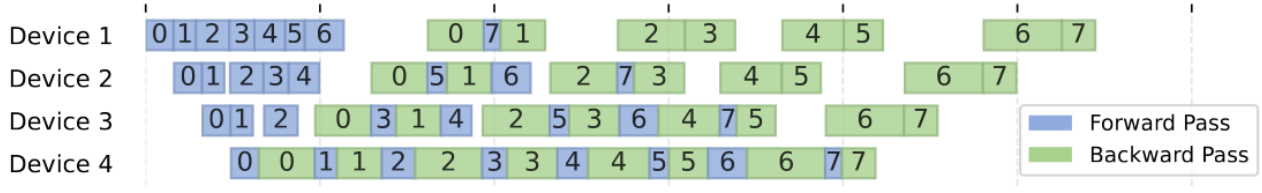


(a) 1F1B (uniform micro-batches)

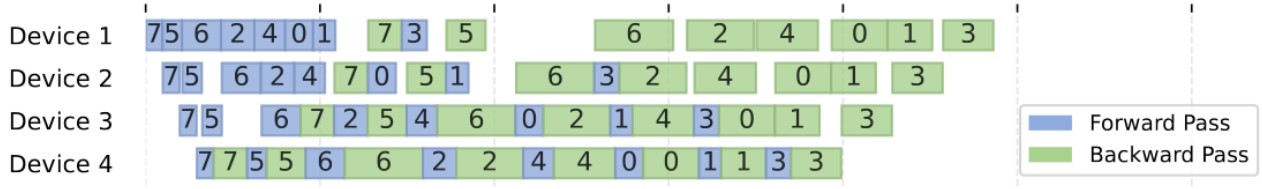


(b) 1F1B (dynamic micro-batches)

Misaligned pipeline with irregular bubbles



(c) Adaptive Schedule



(d) Adaptive Schedule & Micro-batch Reordering

Adaptive scheduling controls the injection time of micro-batches into the pipeline; reordering adjusts the sequence of computations for different microbatches

Defend the schedule choice, not just the bubble number

METHOD	MAIN LEVER	UPDATE SEMANTICS	COST TO CHECK
GPipe	F-then-B microbatching	Synchronous flush	Peak activation memory
1F1B	Earlier backward	Synchronous step	Warm-up + retained state
PipeDream family	Async / stashing / 2BW	Async or periodic flush	Versions + weight buffers
Interleaved	Smaller model chunks	Synchronous step	More P2P messages
Chimera / Hanayo	Directions / waves	Synchronous	Replicas or communication
Zero Bubble	Separate B from W	Synchronous	W placement + state
DynaPipe	Variable-length microbatches	Synchronous training	/

REVIEW TASK

K = 8, M = 16, HBM is tight, and strict synchronous updates are required. Which baseline and which alternative would you test first?

LLM Parallel Training

- Model Parallelism
  - Pipeline Parallelism
  - **Tensor Parallelism**
  - Sequence Parallelism
- Mixture of Experts
  - MoE Basics
  - Expert Load Balancing

Tensor Parallelism Splits Work Inside Each Layer

The same layer runs on every TP device; only its tensors and computation are partitioned

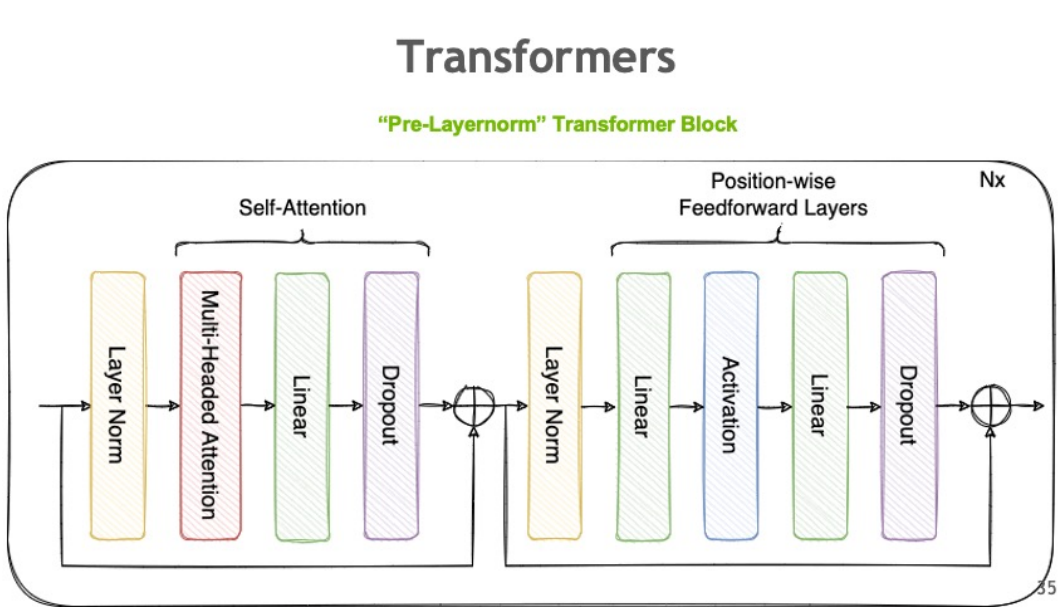
DATA PARALLELISM	**Split the batch**	Each device keeps the full model and processes different samples.	Synchronize weight gradients.
PIPELINE PARALLELISM	**Split the layer**	Each device or stage owns different layers.	Pass activations at stage boundaries.
TENSOR PARALLELISM	**Split matrix operations**	Each device owns shards of weights inside the same layer.	**Combine activation-sized partial results.**

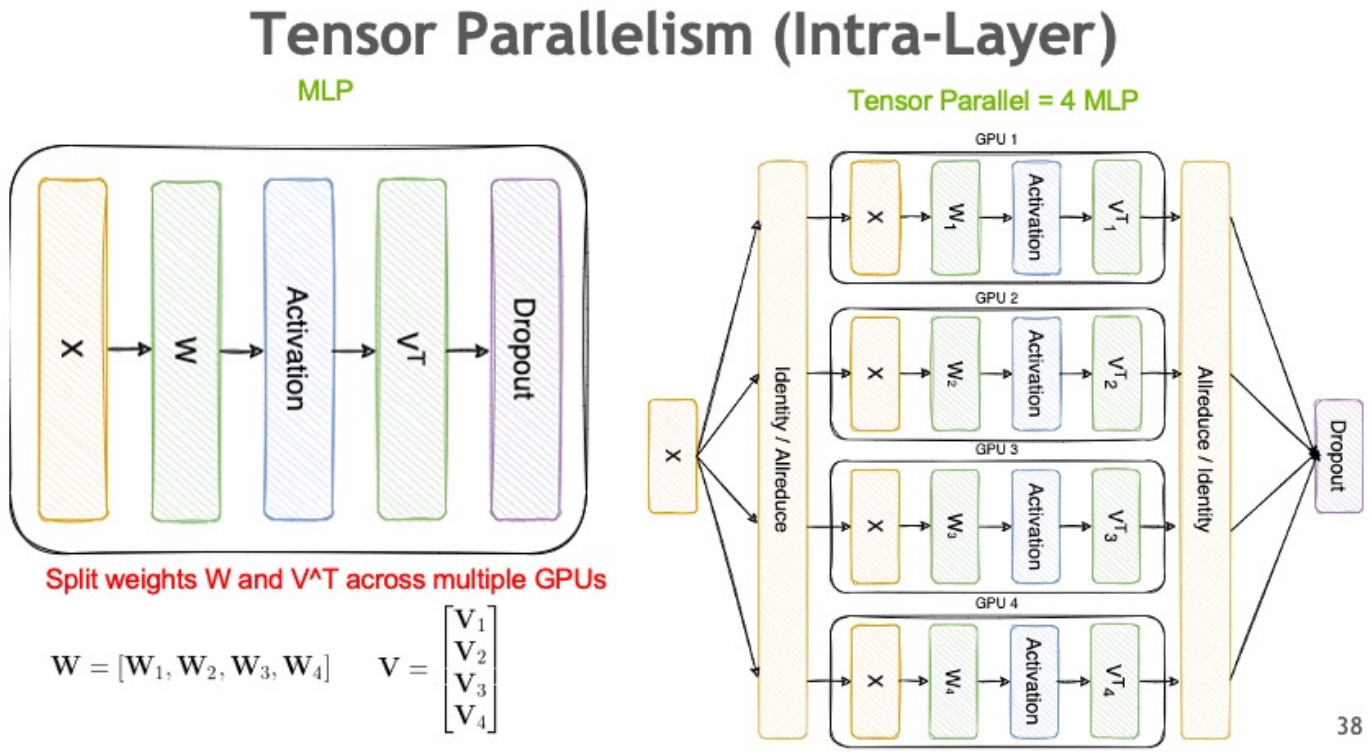
TP handles wide layers; pipeline parallelism handles depth; data parallelism handles batch scale.

REMEMBER **Tensor parallelism is intra-layer model parallelism: every TP device participates in every partitioned layer.**

Source: Shoeybi et al., Megatron-LM (2019), .

Tensor Parallelism – A Visual Guide





What data is splitted?

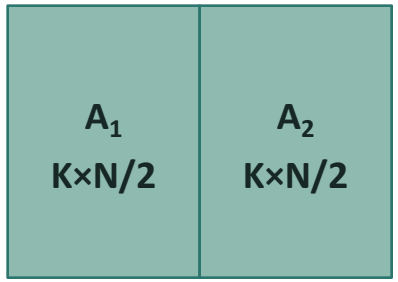
Column Splits Output Features; Row Splits Input Features

For Y = XA, use X: S×K and A: K×N; therefore Y: S×N

X · S×K **A · K×N** **Y · S×N**

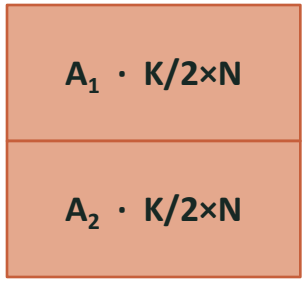
S collects batch × sequence token positions. K is contracted; N indexes output features.

COLUMN PARTITION · A = [A_1 A_2]



Splits N, the output-feature dimension.
Each shard produces different output columns.

ROW PARTITION · A = [A_1 ; A_2]



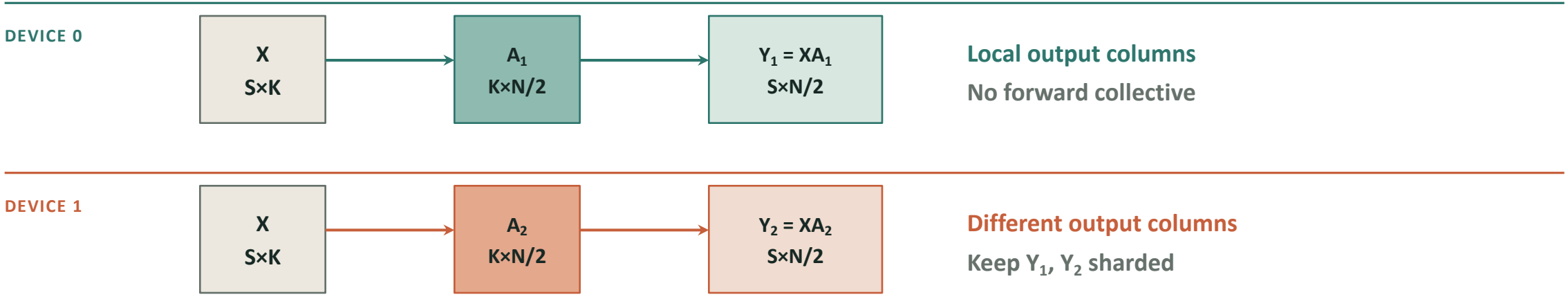
Splits K, the contracted input dimension.
Each shard contributes a partial output sum.

REMEMBER **Do not choose a split by appearance: identify whether it partitions output features or the summed input dimension.**

Column-Parallel GEMM Produces Independent Output Shards

Both devices see the same X; device i stores A_i and computes $Y_i = XA_i$

$$A = [\, A_1 \;\; A_2 \,] \quad \Rightarrow \quad [\, Y_1 \;\; Y_2 \,] = [\, XA_1 \;\; XA_2 \,]$$



BACKWARD CHECK

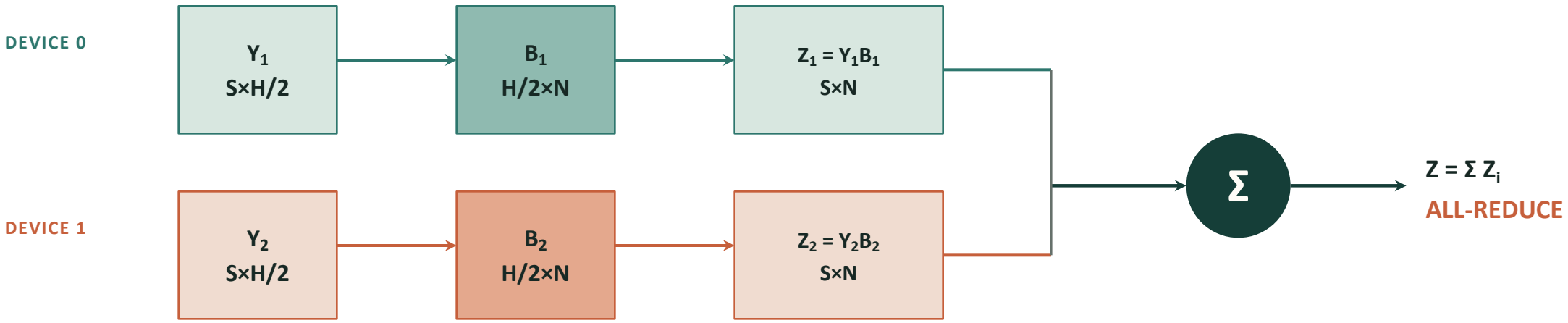
$$dA_i = X^T dY_i \quad \cdot \quad dX_i = dY_i A_i^T \quad \cdot \quad \mathbf{dX = \Sigma_i dX_i}$$

REMEMBER **Column parallelism needs no forward reduction, but a replicated downstream dX requires summing the local input gradients.**

Row-Parallel GEMM Produces Partial Sums

Partition Y by columns and B by matching rows; every device produces a full-shape partial Z

$$Y = [\, Y_1 \;\; Y_2 \,] \quad \cdot \quad B = [\, B_1 \,;\, B_2 \,] \quad \Rightarrow \quad Z = Y_1B_1 + Y_2B_2$$



Each Z_i has shape S×N; the shards contract different rows of B.

BACKWARD CHECK

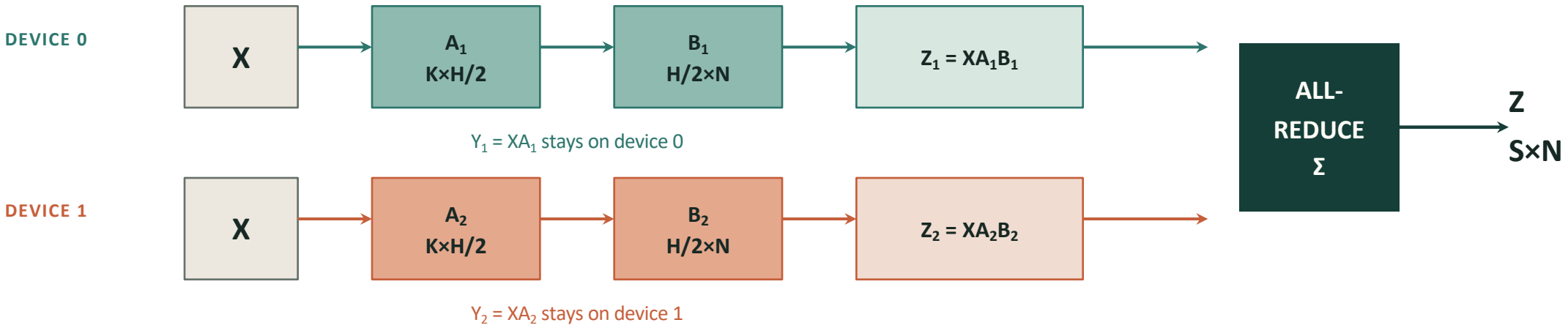
$$dB_i = Y_i^T dZ \quad \cdot \quad dY_i = dZB_i^T \quad \cdot \quad \text{input gradients remain column-sharded}$$

REMEMBER **Row parallelism reduces partial outputs; do not gather Y first—the column shards already match the row shards of B.**

Column Then Row Makes the Intermediate Shard Reusable

A is column-partitioned; B is row-partitioned along exactly the same intermediate dimension H

$$Z = XAB = X[A_1 \ A_2][B_1;B_2] = XA_1B_1 + XA_2B_2$$



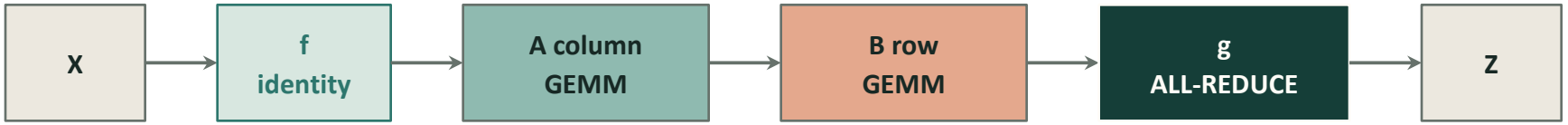
Shape chain: (S×K)(K×H/2)(H/2×N) → (S×N) on each device; sum the two partial outputs.

REMEMBER **No communication occurs between the two GEMMs: the output-column shard of A is exactly the input-column shard expected by B.**

The Same Pair Needs One All-Reduce in Each Direction

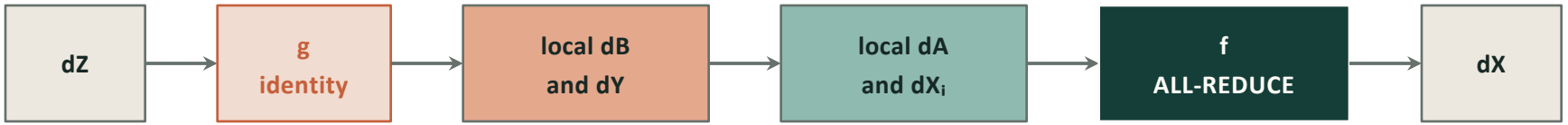
Megatron's f and g operators place communication at the block boundaries, not between the local GEMMs

FORWARD



X replicated → local Y shard → partial Z → replicated Z

BACKWARD



dZ replicated → local parameter gradients → sum partial dX across devices

COLLECTIVE COUNT

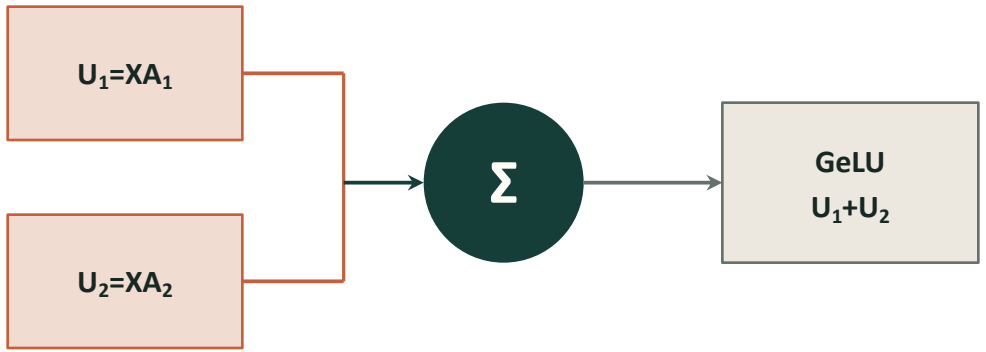
1 forward all-reduce on Z + **1 backward all-reduce** on dX = **2 total per two-GEMM block**

REMEMBER **Weight gradients stay with their weight shards; the block communicates activations in forward and input gradients in backward.**

Column Parallelism Keeps GeLU Local

A nonlinearity prevents independently computed partial sums from being activated separately

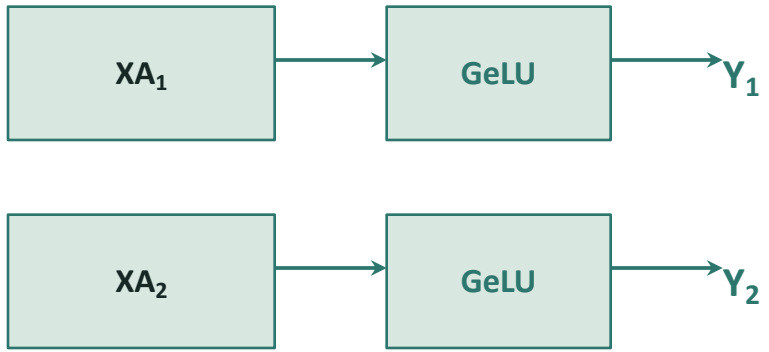
ROW-FIRST: PARTIAL SUMS BEFORE GeLU



Requires a reduction before GeLU.

$GeLU(U_1+U_2)$ differs from $GeLU(U_1)+GeLU(U_2)$

COLUMN-FIRST: COMPLETE LOCAL CHANNELS



Each device owns complete intermediate channels.

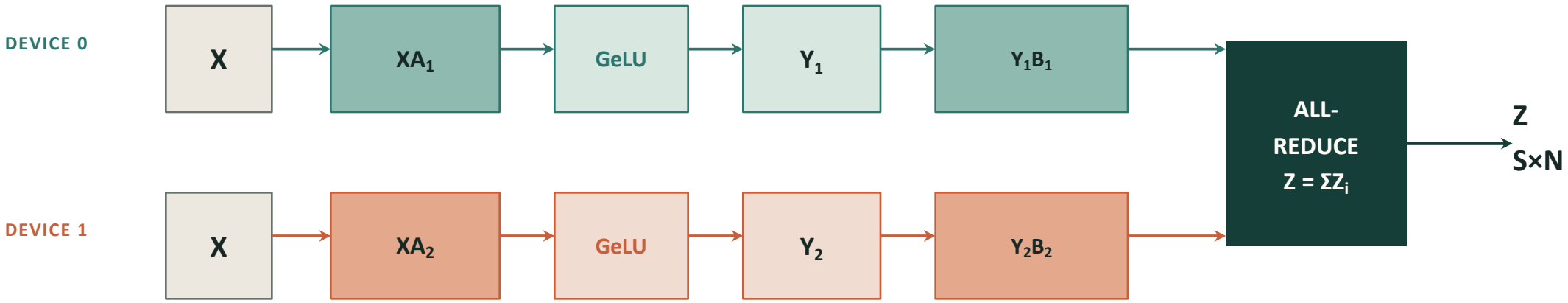
$Y_i = GeLU(XA_i)$ with no intervening collective.

REMEMBER **Use column parallelism before an elementwise nonlinearity; delay reduction until after the following row-parallel GEMM.**

The MLP Uses the Column–Row Pair Around GeLU

Y = GeLU(XA), Z = YB: keep Y sharded and reduce only the final output

A: K×H column-split · B: H×N row-split · $Z = \Sigma_i \, GeLU(XA_i)B_i$ · $GELU(Y_1+Y_2) \neq GELU(Y_1)+GELU(Y_2)$ · $GELU(Y_1; Y_2) = [GELU(Y_1)\ GELU(Y_2)]$

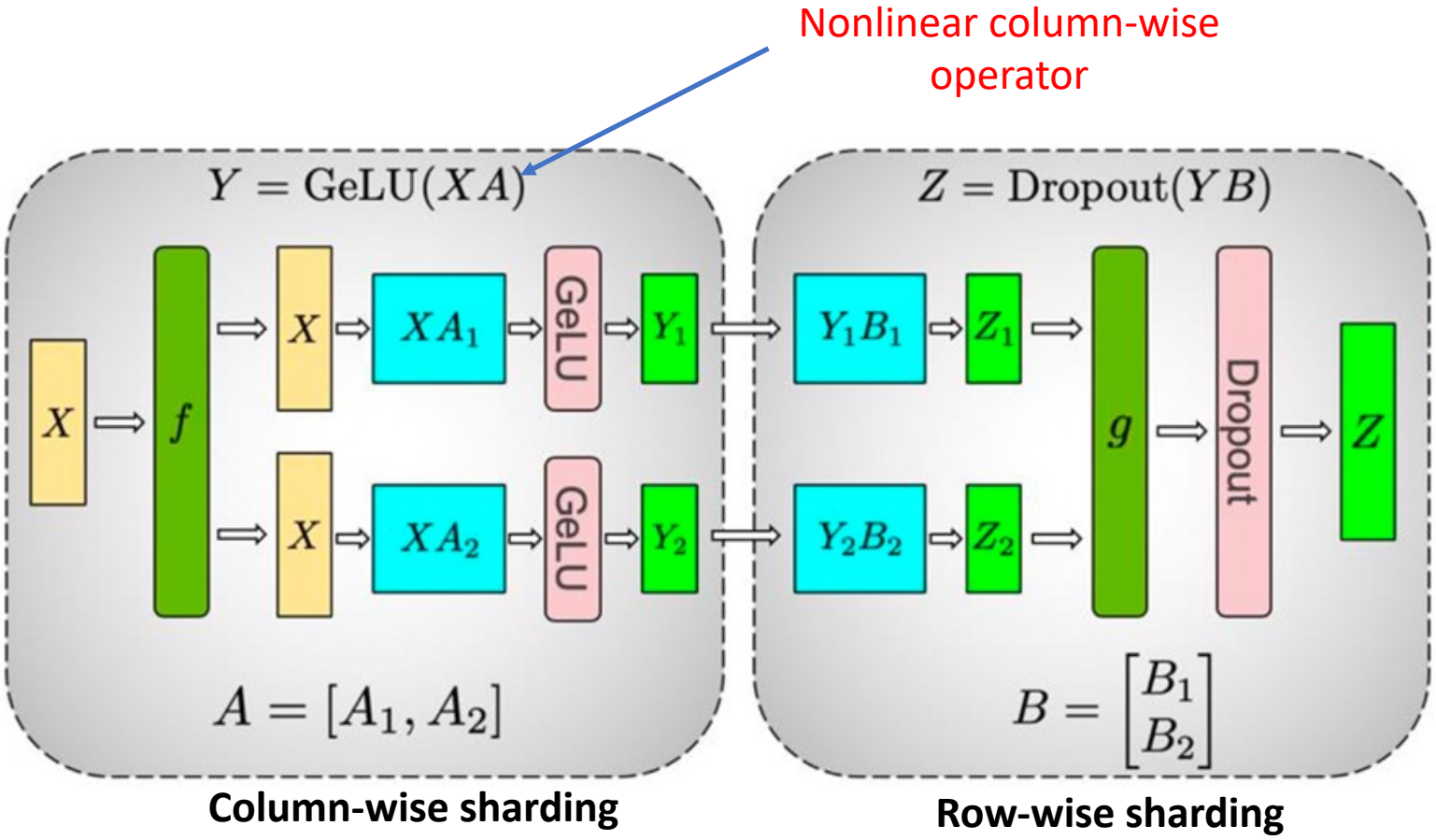


No all-gather or all-reduce between GeLU and the second GEMM

Local work: A_i, Y_i, B_i, dA_i, dB_i · **Shared result:** Z after summation

REMEMBER **The MLP preserves the original result while storing only half of each weight matrix per device when P = 2.**

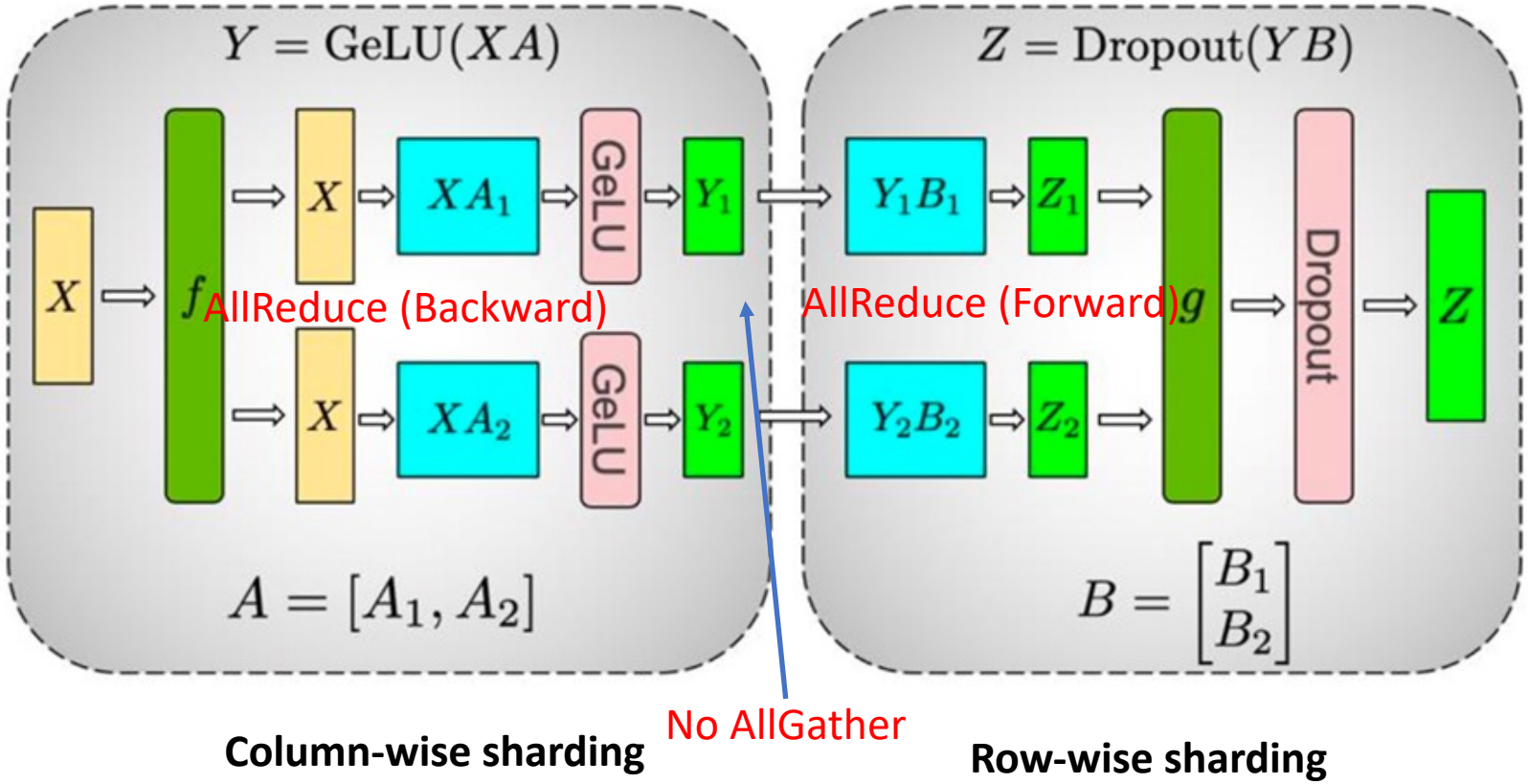
The MLP Uses the Column–Row Pair



- Partition A by columns and B by rows
- Forward pass
  - "Copy" X to both GPUs via ***f***
  - Execute math operations
  - "Add" Z_1 and Z_2 for a complete output Z via ***g***
- Backward pass
  - "copy" $\partial L/\partial Z$ to both GPUs via ***g***
  - Execute math operations
  - "Add" partial derivatives of X on two paths via ***f***

f and ***g*** are **conjugate**, ***f*** is identity operator in the forward pass and all-reducein the backward pass while ***g*** is all-reduce in forward and identity in backward

MLP Communiation Analysis



f and ***g*** are **conjugate**, ***f*** is identity operator in the forward pass and all-reduce in the backward pass while ***g*** is all-reduce in forward and identity in backward

A Two-Device MLP Example Makes Every Shape Explicit

Illustrative dimensions: K = 4096, H = 16384, N = 4096, P = 2; S sequence length

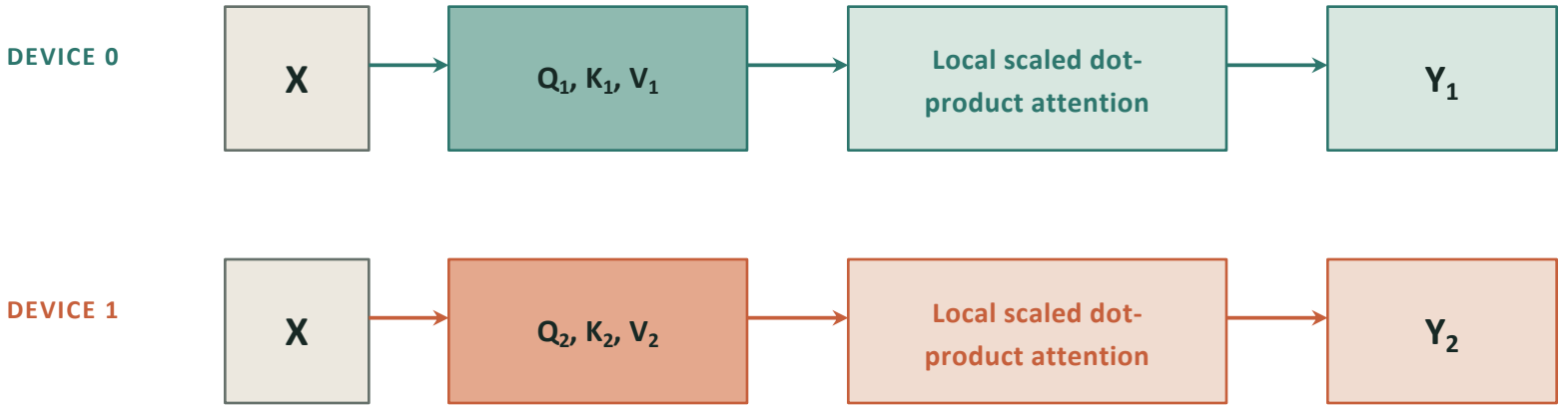
TENSOR	FULL LOGICAL SHAPE	EACH DEVICE HOLDS	COMMUNICATION
A	4096 × 16384	4096 × 8192 (A_i)	None
Y = GeLU(XA)	S × 16384	S × 8192 (Y_i)	No gather
B	16384 × 4096	8192 × 4096 (B_i)	None
Z = YB	S × 4096	Partial S × 4096 (Z_i)	**All-reduce Z**
WEIGHT FOOTPRINT	Each device stores 4096×8192 + 8192×4096 = 67,108,864 weights—half of the two full matrices.		

REMEMBER **The intermediate Y is half as wide per device; Z is full-size but partial until the output all-reduce.**

Self-Attention Splits QKV Projections by Heads

For h heads and P devices, each device owns h/P heads in standard multi-head attention

$d = h \cdot d_h$ · W^Q, W^K, W^V are column-partitioned along complete heads



heads 1 … h/2
No communication inside attention

heads h/2+1 … h
No communication inside attention

For each local head j: $\text{Attn}_j = \text{softmax}(Q_j K_j^T / \sqrt{d_h})\, V_j$

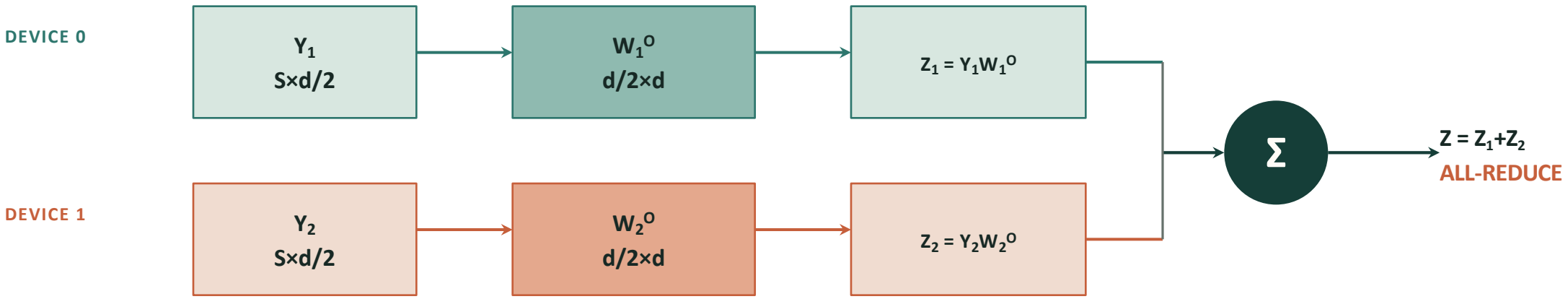
Softmax is **independent** per head, so complete local heads can be processed without a collective.

REMEMBER **Column-parallel QKV projections produce complete local heads; the first output collective waits until after the output projection.**

The Output Projection Turns Head Shards Into One Sum

Concatenating heads is logical only: each device sends its local attention output directly to a row shard of W^O

$$Y = [\, Y_1 \;\; Y_2 \,] \quad \cdot \quad W^O = [\, W_1^O \,;\, W_2^O \,] \quad \cdot \quad Z = YW^O$$

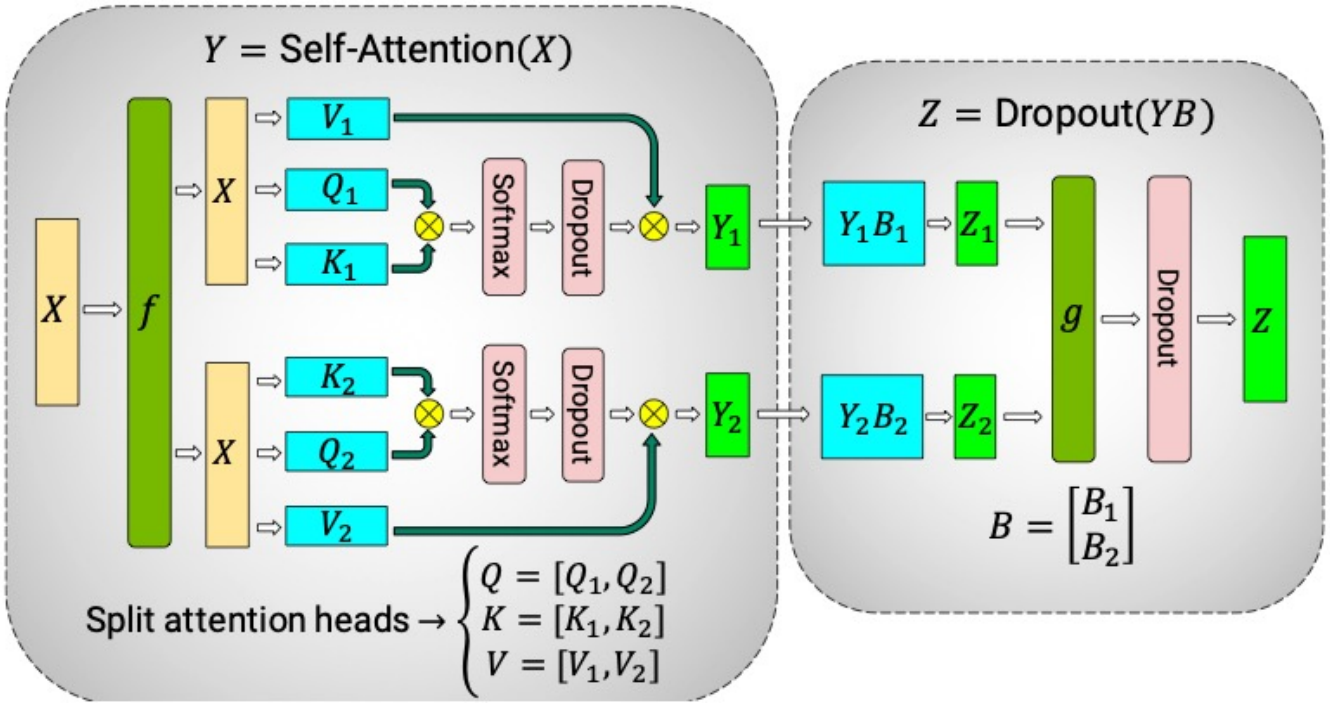


Each Z_i has S×d outputs; only the head contributions differ.

Final algebra: $Z = Y_1W_1^O + Y_2W_2^O = [Y_1\ Y_2]W^O$

REMEMBER **Attention reuses the same column–row pattern: QKV by heads, local attention, row-parallel output projection, then summation.**

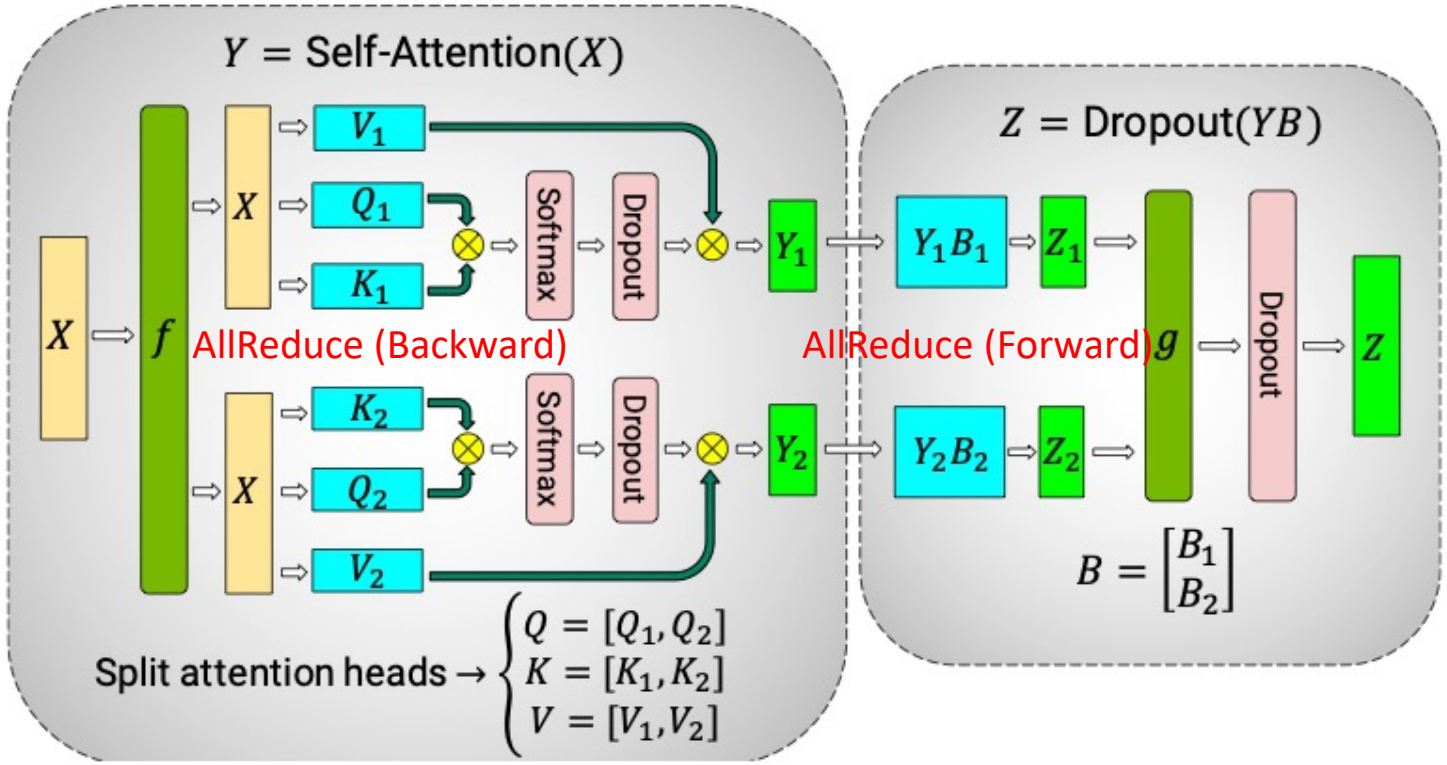
Self-Attention: computation analysis



Column-wise sharding **Row-wise sharding**

- Partition three parameter matrices W_Q, W_K and W_V by columns and partition the linear layer B by rows
- Forward pass
  - "Copy" X to both GPUs via ***f***
  - Execute math operations such as computing Q, K, V
  - "Add" Z_1 and Z_2 for a complete output Z via ***g***
- Backward pass
  - "copy" $\partial L/\partial Z$ to both GPUs via ***g***
  - Execute math operations
  - "Add" partial derivatives of X on two paths via ***f***

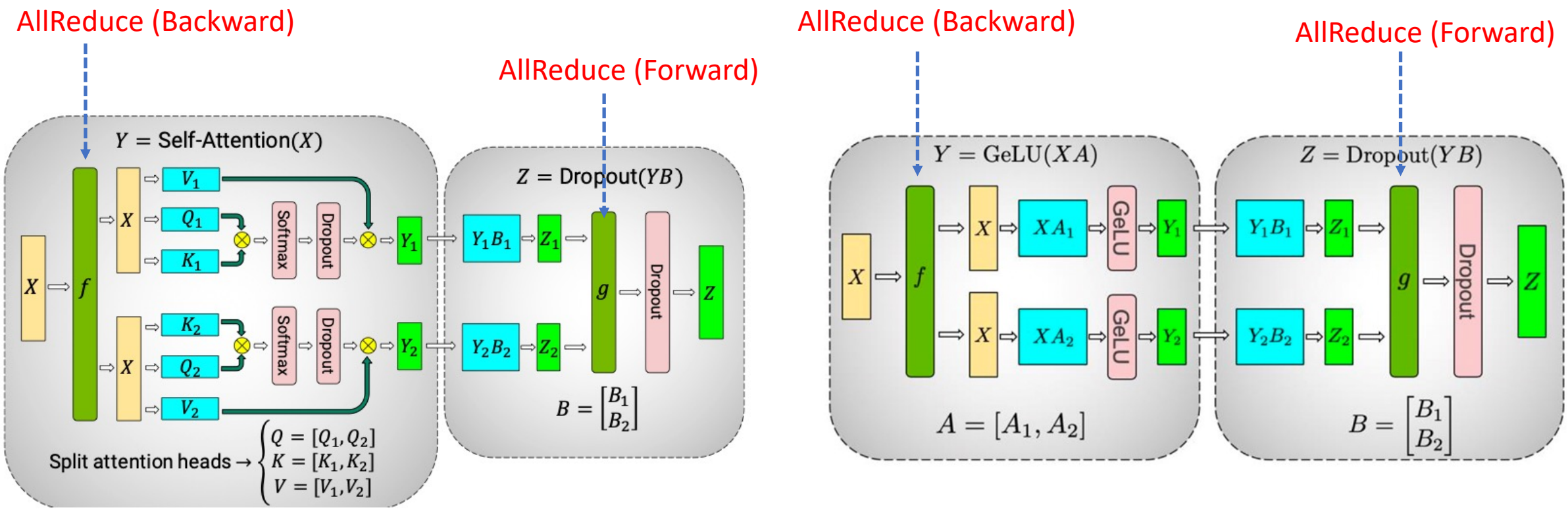
Self-Attention: comunication analysis



Column-wise sharding **Row-wise sharding**

f and ***g*** are **conjugate**, ***f*** is identity operator in the forward pass and all-reduce in the backward pass while ***g*** is all-reduce in forward and identity operator in backward

Self-Attention: comunication analysis



REMEMBER **Two AllReduce in the forward pass, and two AllReduce in the backward pass**

Tensor Parallelism Communication Analysis

One output all-reduce in forward and one input-gradient all-reduce in backward per attention block

BLOCK	FORWARD	BACKWARD	TOTAL	TENSOR PER ALL-REDUCE
Self-attention	1 · output Z	1 · input dX	**2**	S×d
MLP, K = N = d	1 · output Z	1 · input dX	**2**	S×d
Whole Transformer layer	2	2	**4**	four S×d tensors

RING BYTE ESTIMATE

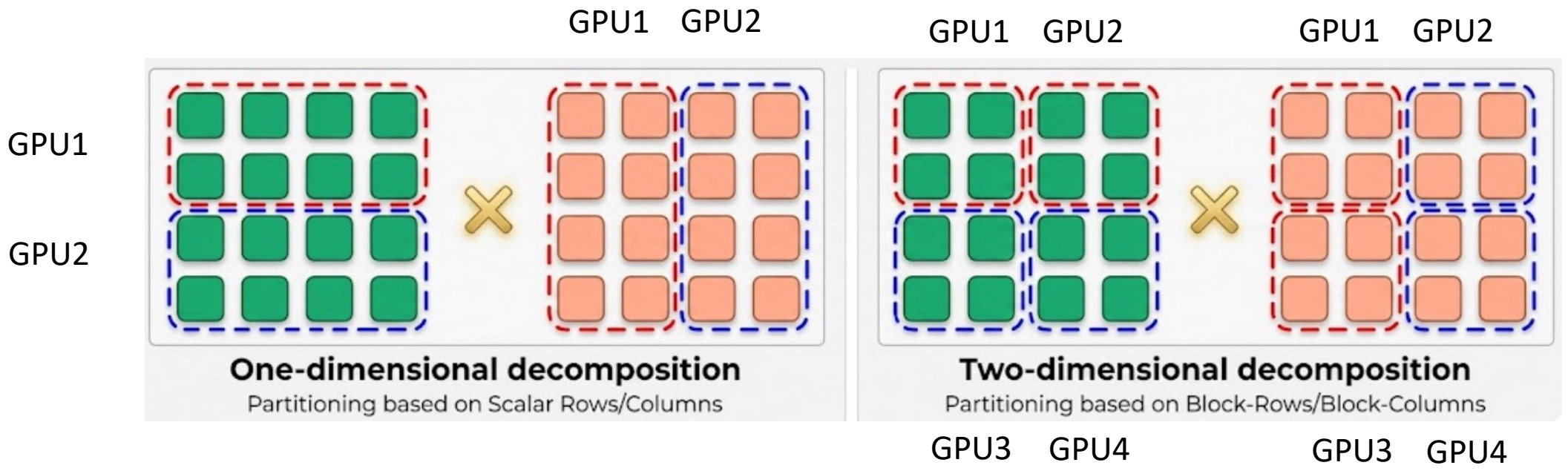
Attention per device ≈ $4(P-1)/P \cdot b \cdot S \cdot d$ bytes

Whole layer ≈ $8(P-1)/P \cdot b \cdot S \cdot d$ bytes

REMEMBER **Attention and MLP each contribute 2 all-reduces; together, one Transformer layer has 4 across forward + backward.**

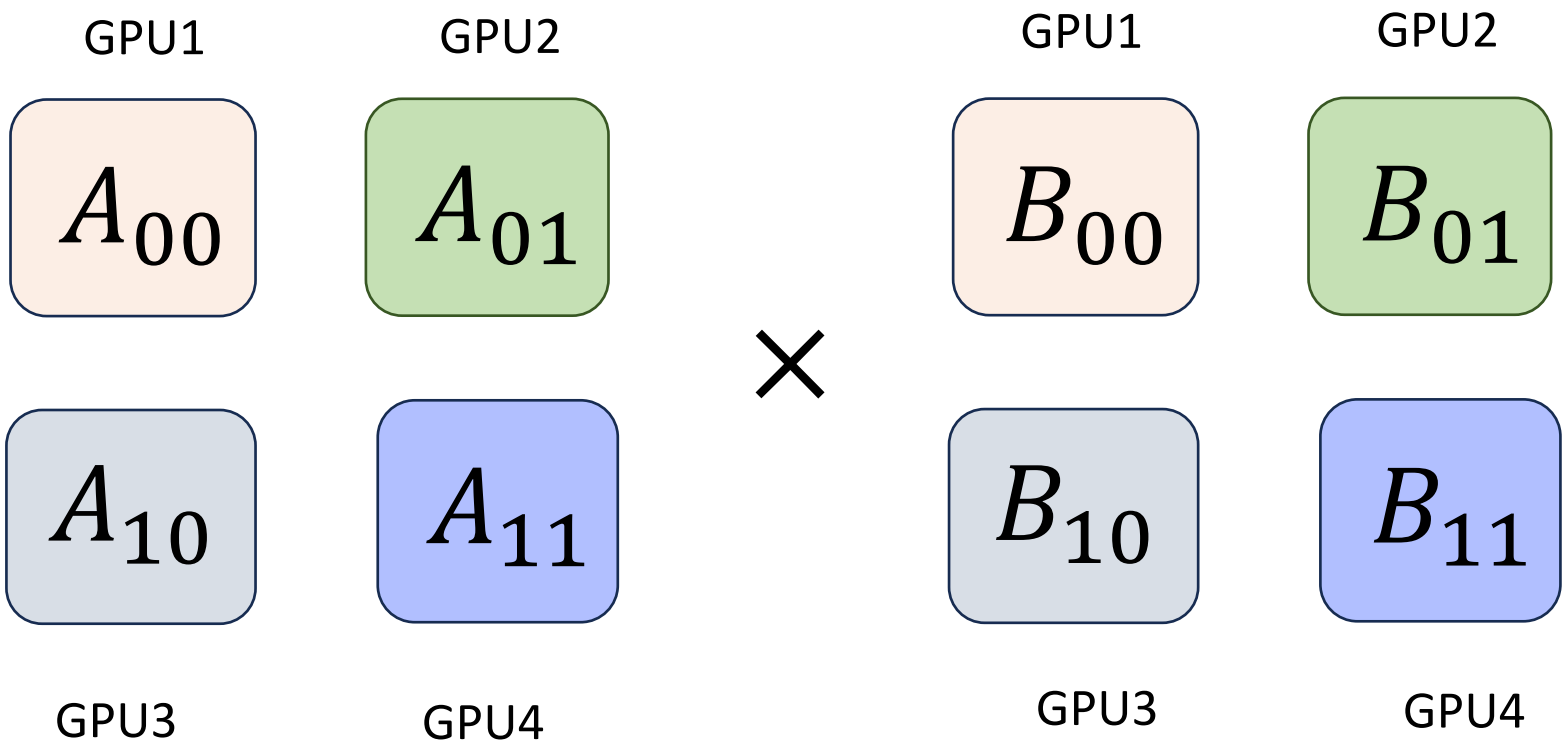
Two-Dimensional Tensor Parallelism

- Submatrix Multiplication: more GPUs working simultaneously

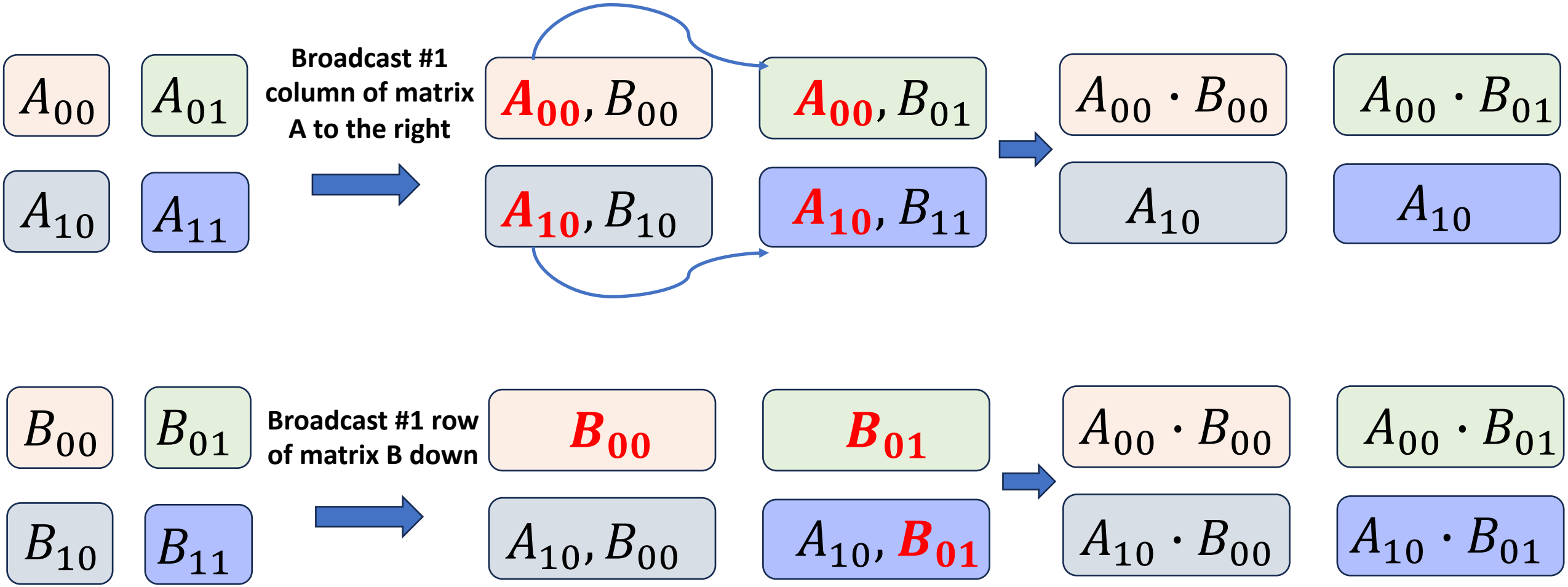


2D Tensor Parallelism splits matrices

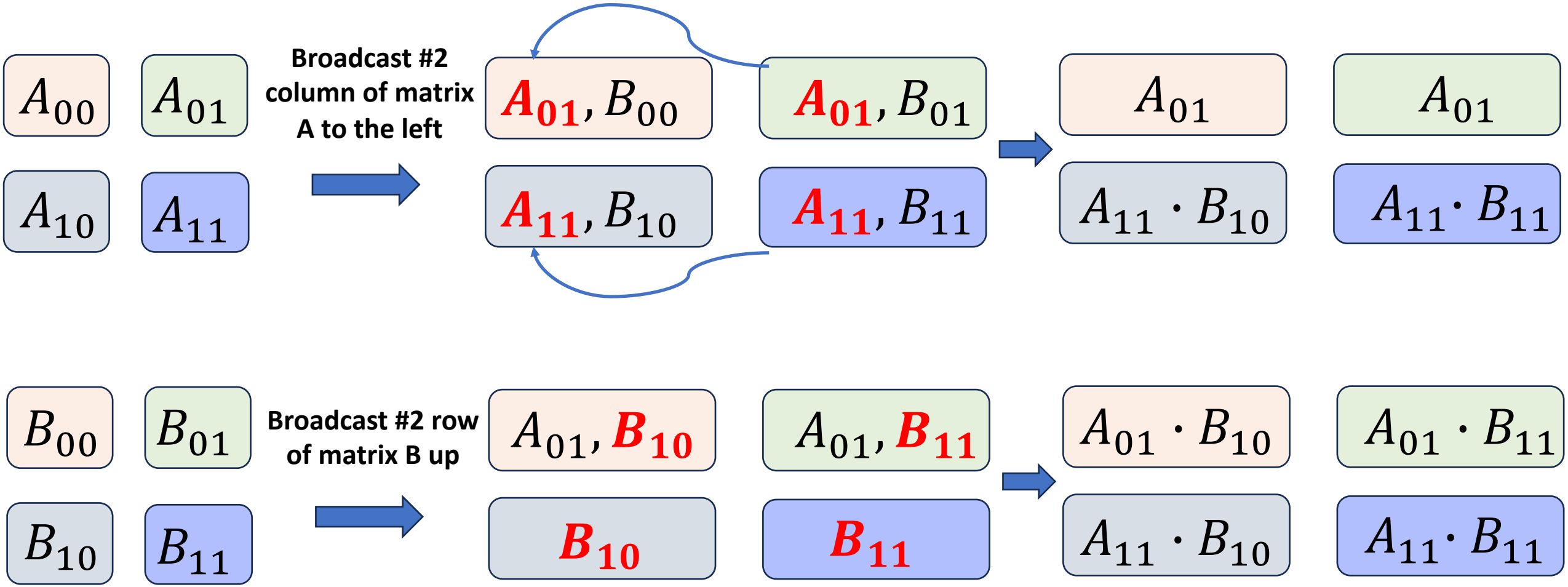
- each color represents a different GPU



2D Tensor Parallelism: a conceptual analysis



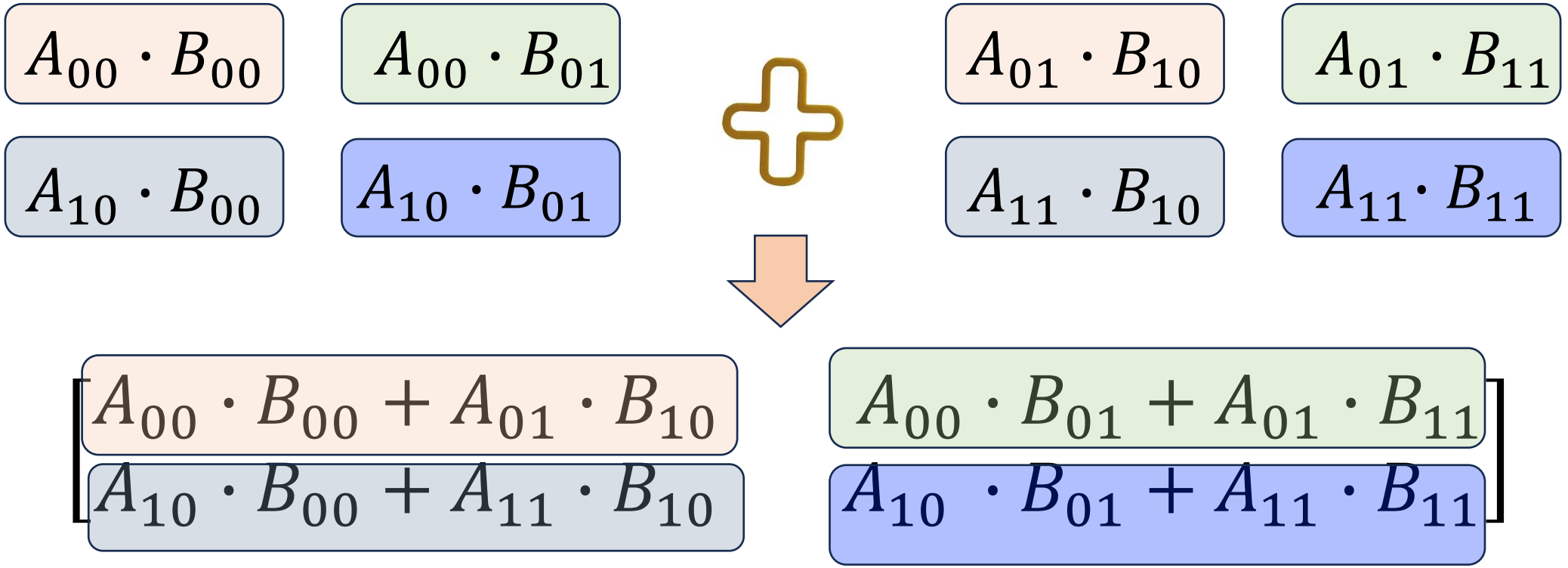
2D Tensor Parallelism: a conceptual analysis



2D Tensor Parallelism: a conceptual analysis

- 2D Tensor Parallelism

Sequential computation



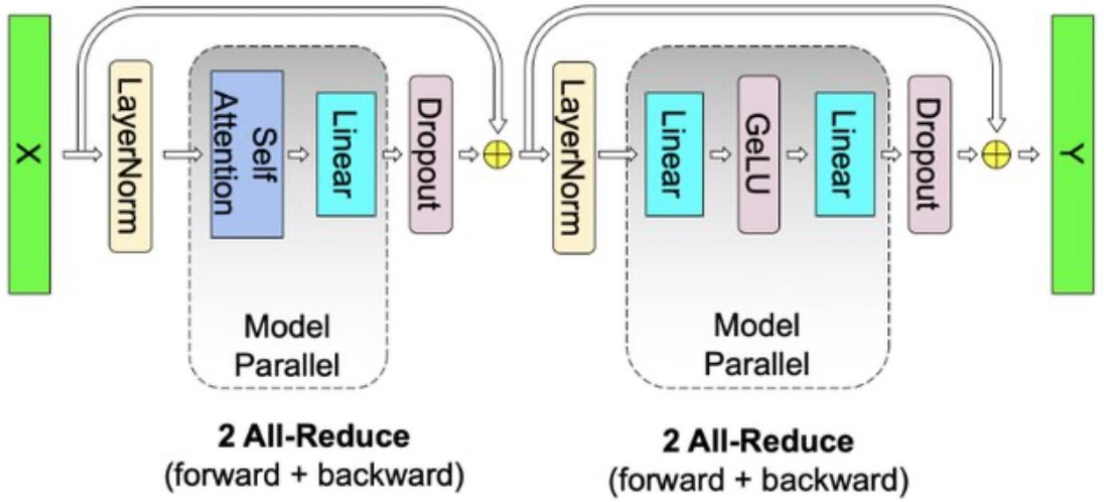
- 2D Tensor Parallelism with more rows and columns?

2D Tensor Parallelism: a conceptual analysis

- 2D Tensor Parallelism: general procedure on $Y = AB$
  - Demand $P = q^2$ GPUs, and partition both input and parameters on P GPUs
  - At step t, broadcast A_{it} in its row, and broadcast B_{tj} in its column
  - Repeat the above step by q rounds, and perform normal sub-matrix multiplications and additions
- Computation, memory and communication complexity
  - Computation: $O\left(\frac{1}{P}\right)$
  - Memory (parameter and activation): $O\left(\frac{1}{P}\right)$
  - Communication: $O\left(\frac{1}{q}\right)$ (note that broadcast is replaced by All-Gather)

Tensor Parallelism: take-home messages

- Tensor parallelism segments matrix on feature dimensions,
- Tensor parallelism introduces **multiple AllReduce operations,** thus constrained by the high bandwidth domain
- Only for Attention and MLP, leaving Keeping the entire input and output to the Attention and MLP layers



LLM Parallel Training

- Model Parallelism
  - Pipeline Parallelism
  - Tensor Parallelism
  - **Sequence Parallelism**
- Mixture of Experts
  - MoE Basics
  - Expert Load Balancing

Context windows grew ~250× in five years; training must follow

CONTEXT LENGTH OF FRONTIER MODELS

Year	Model	Max context
2020	GPT-3	2K tokens
2022	MT-NLG / PaLM	2K tokens
2023	GPT-4 · Llama 2	4K – 32K tokens
2024	Llama 3.1 · Claude · Gemini 1.5	128K – 1M+ tokens
2025–	frontier LLMs, video models	256K – 1M+ routine

WHY IT REACHES THE TRAINER

· **Long documents and codebases** — full-context attention over 100K+ tokens during finetuning.

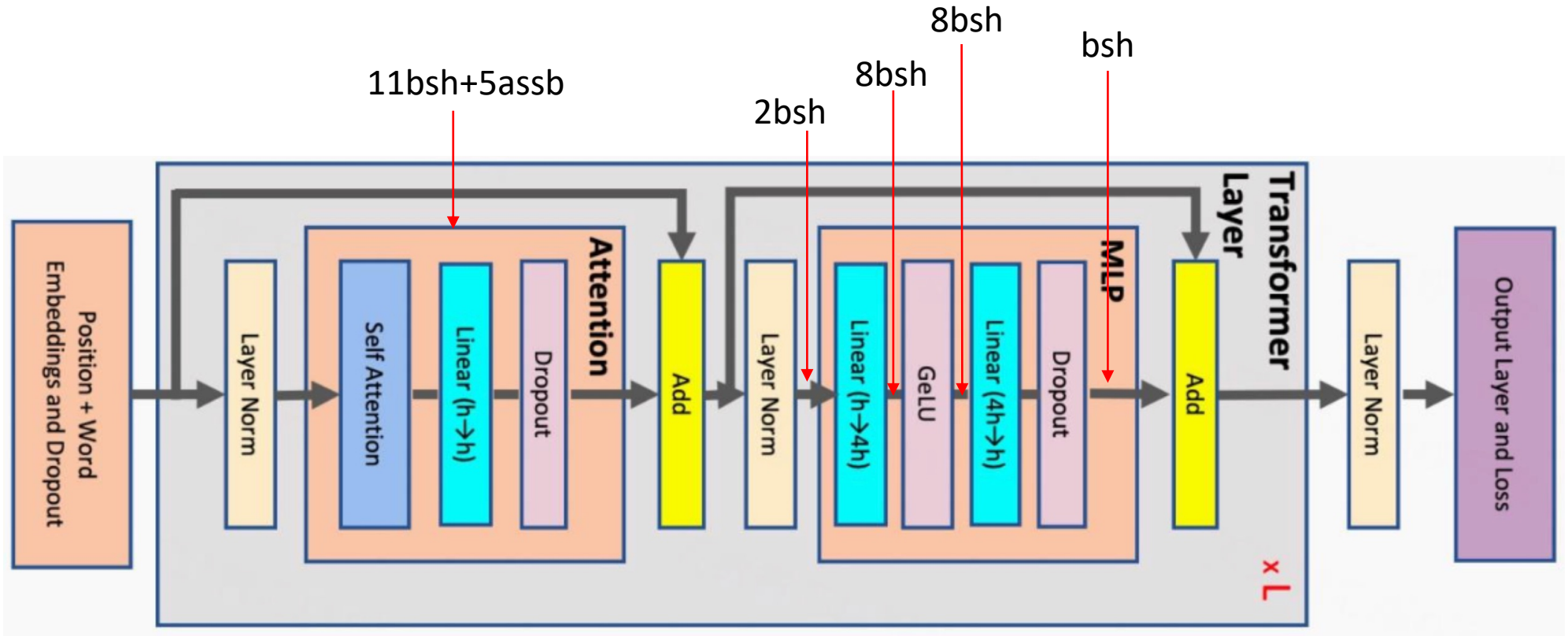
· **Agent workloads** — tool-call traces and multi-turn histories quickly exceed 100K tokens.

· **Video and multimodal** — minutes of video tokenize to 10^5–10^6 patch tokens; attention runs over space and time.

· **Pretraining itself** — long-context capability is trained, not only inferred: curricula extend s well beyond 4K–8K.

> Parameters are handled by the classic 3-D stack (DP · TP · PP). At 128K–1M tokens, the binding constraint shifts to the **sequence dimension**: attention FLOPs per token grow with s, and activation memory grows with s. The sequence needs its own parallelism axis.

Activation Memory Footprint



In-total: 5abss + **34sbh**

Removed by FlashAttention

Per-layer activation memory grows with s — and it dominates

activations / layer ≈ **$34 \cdot s \cdot b \cdot h$** + **$5 \cdot a \cdot s^2 \cdot b$** bytes (bf16, no recompute)

$34 \cdot s \cdot b \cdot h$ — QKV, attention out-proj, MLP, LayerNorm, dropout: linear in s, stored even with FlashAttention

$5 \cdot a \cdot s^2 \cdot b$ — materialized attention scores (QK^T, softmax, dropout): quadratic; **removed by FlashAttention kernels**

70B-SCALE MODEL (h = 8192, a = 64, L = 80, b = 1, FLASHATTENTION)

Sequence s	Per layer (34·s·b·h)	All 80 layers
8K tokens	≈ 4.6 GB	≈ 0.37 TB
128K tokens	≈ 73 GB	≈ 5.8 TB
1M tokens	≈ 584 GB	≈ 47 TB

Even with FlashAttention, the **linear term alone** exceeds any single GPU's memory once s passes ~100K.

Doubling s doubles the bill — memory, not FLOPs, binds first.

OBSERVATION **Long context activations dominate the GPU memory during training.**

What TP misses: LayerNorm and dropout replicate on every rank

ONE TRANSFORMER LAYER UNDER TP (t = 8)

LayerNorm — replicated, full [b, s, h] × t ranks

QKV proj + attention + out proj — h/t sharded

Dropout + residual — replicated, full [b, s, h] × t

LayerNorm — replicated, full [b, s, h] × t

MLP h → 4h → h — column / row parallel (h/t)

Dropout + residual — replicated, full [b, s, h] × t

replicated on all t ranks — memory NOT divided by t

sharded along h — memory divided by t

THE ACCOUNTING (KORTHIKANTI ET AL. 2022)

TP activations / layer = **10·s·b·h** + (24 + 5·a·s/h)·s·b·h/t

The **10·s·b·h term** — LayerNorm inputs, dropout outputs and masks — is **never divided by t**, because these ops are replicated across the TP group.

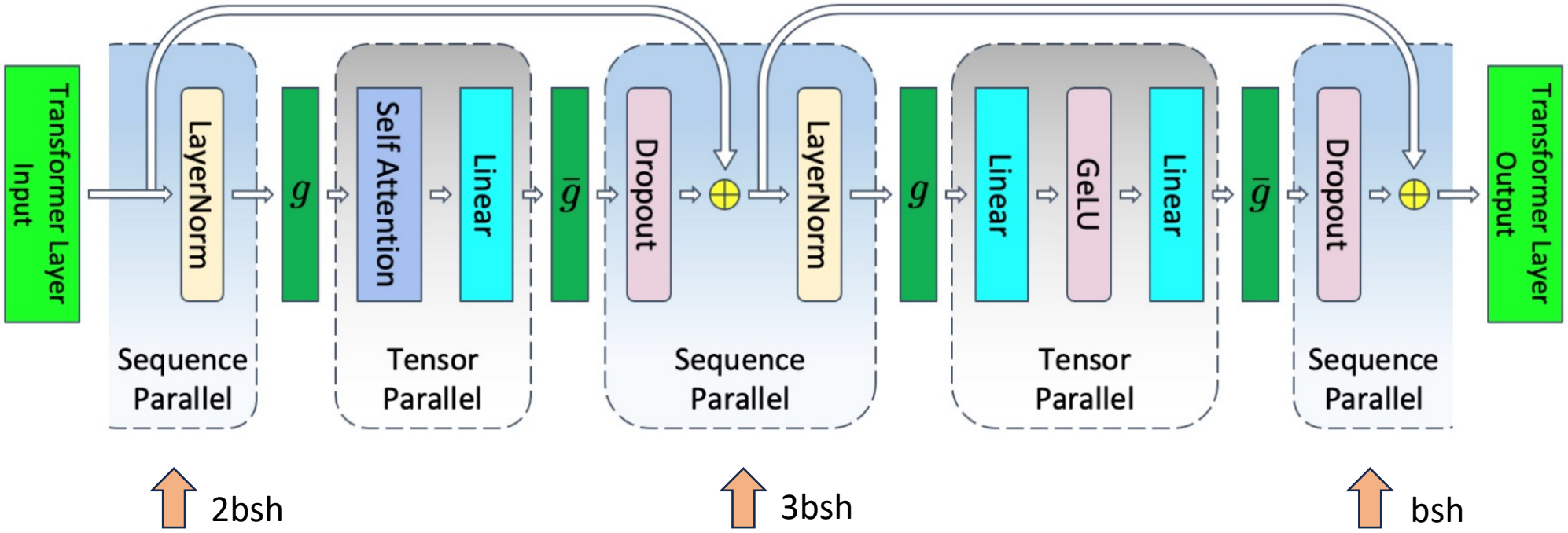
At t = 8, s = 128K, h = 8192 (b = 1): **≈ 21.5 GB per layer, per rank** — over 80 layers that is ~1.7 TB of redundant storage.

The enabling observation: LayerNorm, dropout, and residual are **elementwise along the token axis**. Tokens have no cross-dependencies here, so this region can be sharded along s with zero extra communication.

- Representative SP frameworks
  - Megatron-LM SP
    - 《Reducing Activation Recomputation in Large Transformer Models》
    - Deemed as a further optimization of Tensor Parallelism
  - DeepSpeed Ulysses
    - 《DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models》
  - Colossol AI SP
    - 《Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training》

Sequence Parallelism

- Megatron-LM sequence parallelism



Tensor parallelism only segments inputs and outputs of ***self-attention*** and ***MLP***

Sequence Parallelism

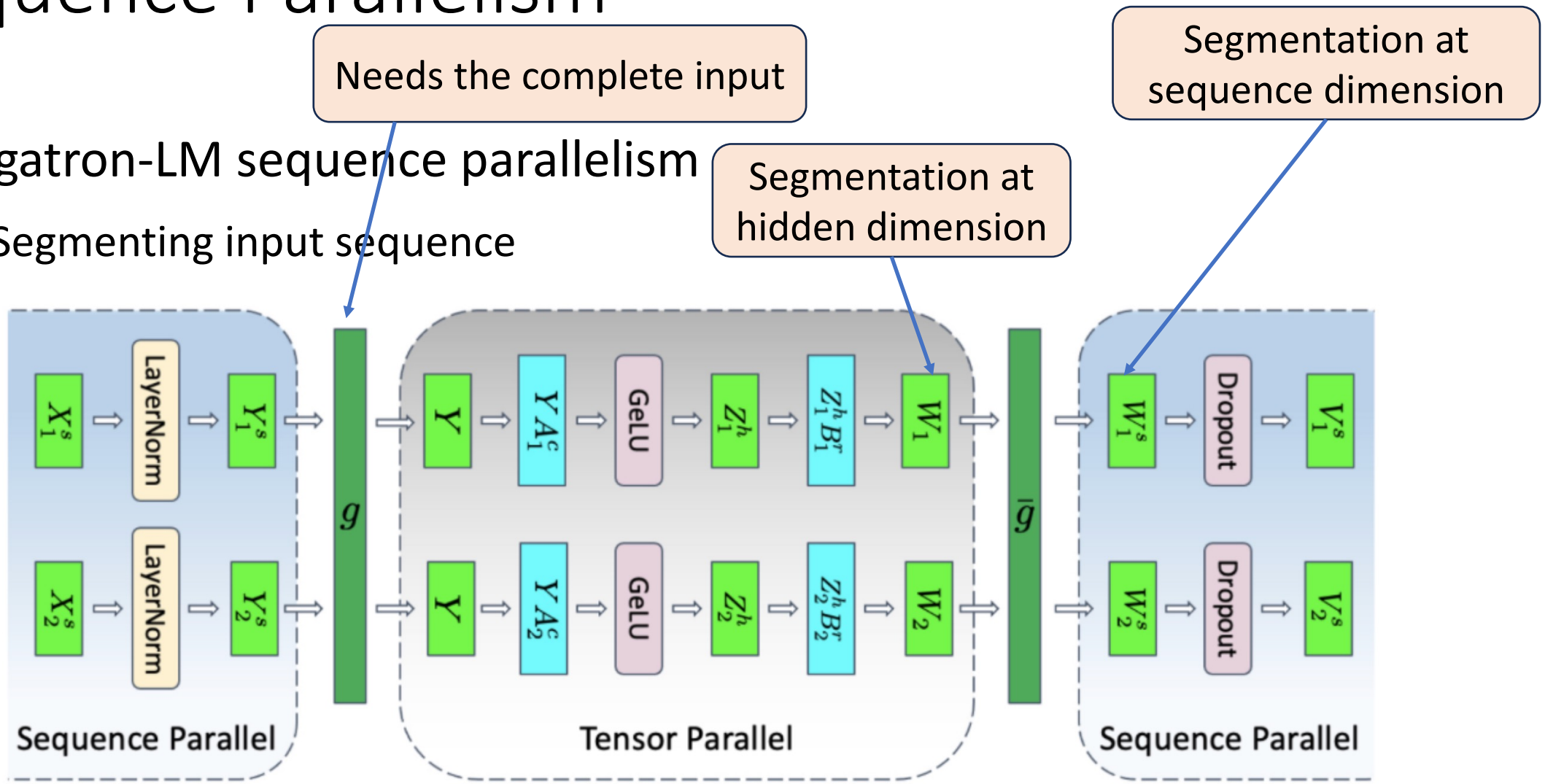
- Megatron-LM sequence parallelism
  - *layernorm* operation
    - The shape of an input matrix: [batch_size, seq_len, hidden_dim]
    - Compute mean, variance and a learnable affine transformation for each ***TOKEN***
    - Can be naturally partitioned by tokens
  - *dropout* operation
    - Each GPU compute dropout independently

Activations (input/output of layernorm) on each GPU: $\frac{2bsh}{t}$

Activations (dropout) on each GPU: $\frac{bsh}{t}$

Sequence Parallelism

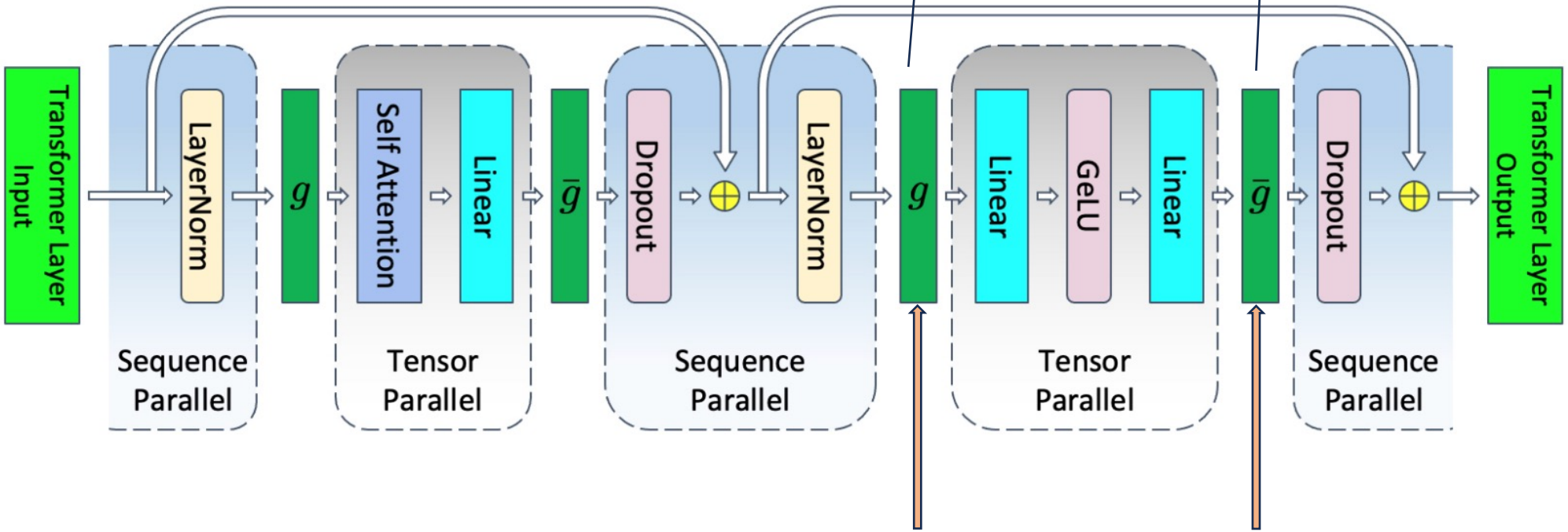
- Megatron-LM sequence parallelism
  - Segmenting input sequence



MLP layer with tensor and sequence parallelism. $\boldsymbol{g}$ and $\bar{\boldsymbol{g}}$ are conjugate. $\boldsymbol{g}$ is all-gather in forward pass and reduce-scatter in backward pass. $\bar{\boldsymbol{g}}$ is reduce-scatter in forward pass and all-gather in backward pass.

Sequence Parallelism

- Megatron-LM sequence parallelism
  - MLP layer (forward pass)

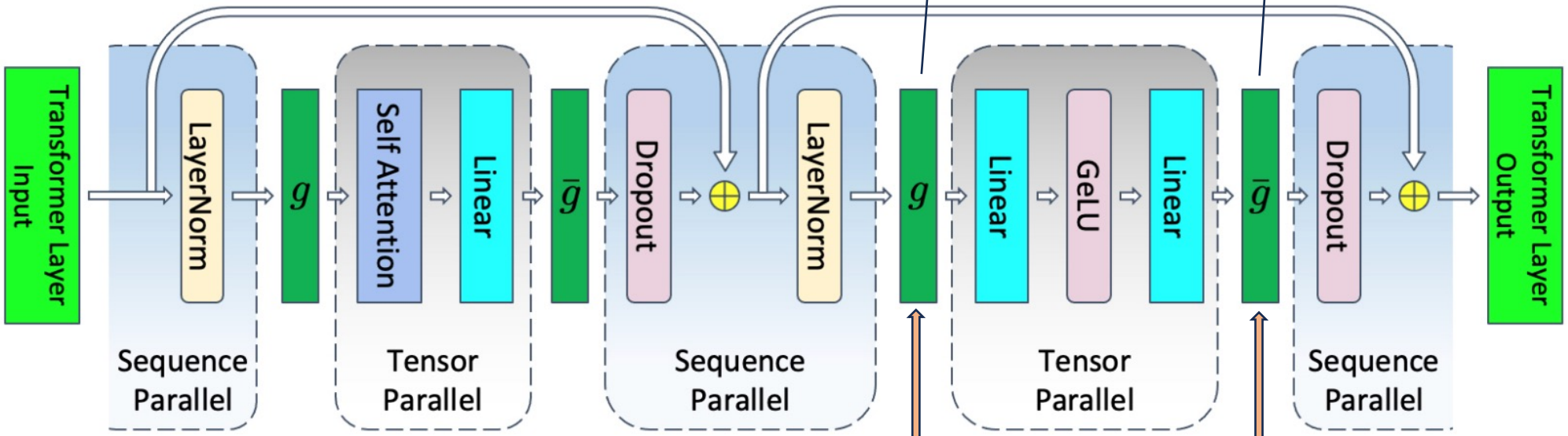


Sequence Parallelism

- Megatron-LM sequence parallelism
  - MLP layer (backward pass)

The currently buffered output is not what you need to proceed to Dropout

Matrix multiplication needs the complete input



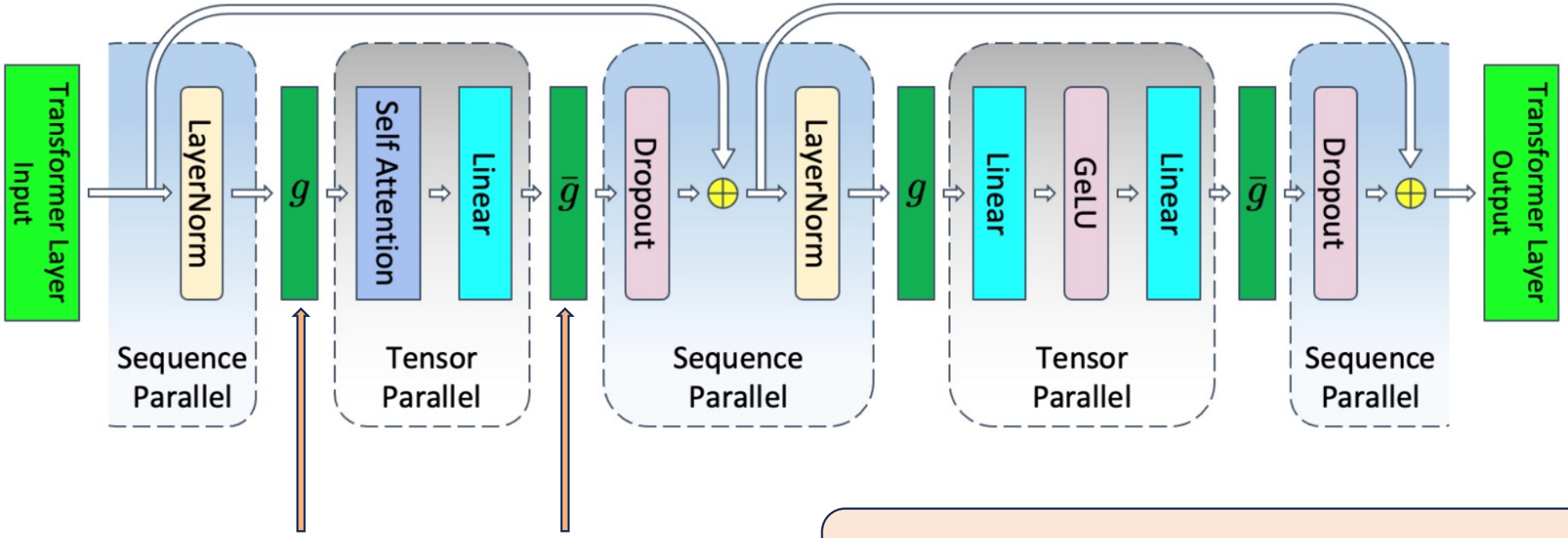
Finally: $5\text{bsh} + \frac{16bsh}{t}$ becomes $\frac{21bsh}{t}$

ReduceScatter to obtain the output fraction

AllGather to obtain the complete input

Sequence Parallelism

- Megatron-LM sequence parallelism
  - Attention (forward pass, backward pass omitted)



AllGather to obtain the complete input

ReduceScatter to obtain the output fraction

Finally: $5\text{bsh} + \frac{5bass+8bsh}{t}$ becomes $\frac{5bass+13bsh}{t}$

MEGATRON-LM SP · MEMORY WIN

Changing AllReduce into ReduceScatter and AllGather

without SP (t = 1)

$$10 \cdot b \cdot s \cdot h$$

full sequence replicated on every rank

with SP (t ranks)

$$10 \cdot b \cdot s \cdot h / t$$

each rank keeps only its sequence shard

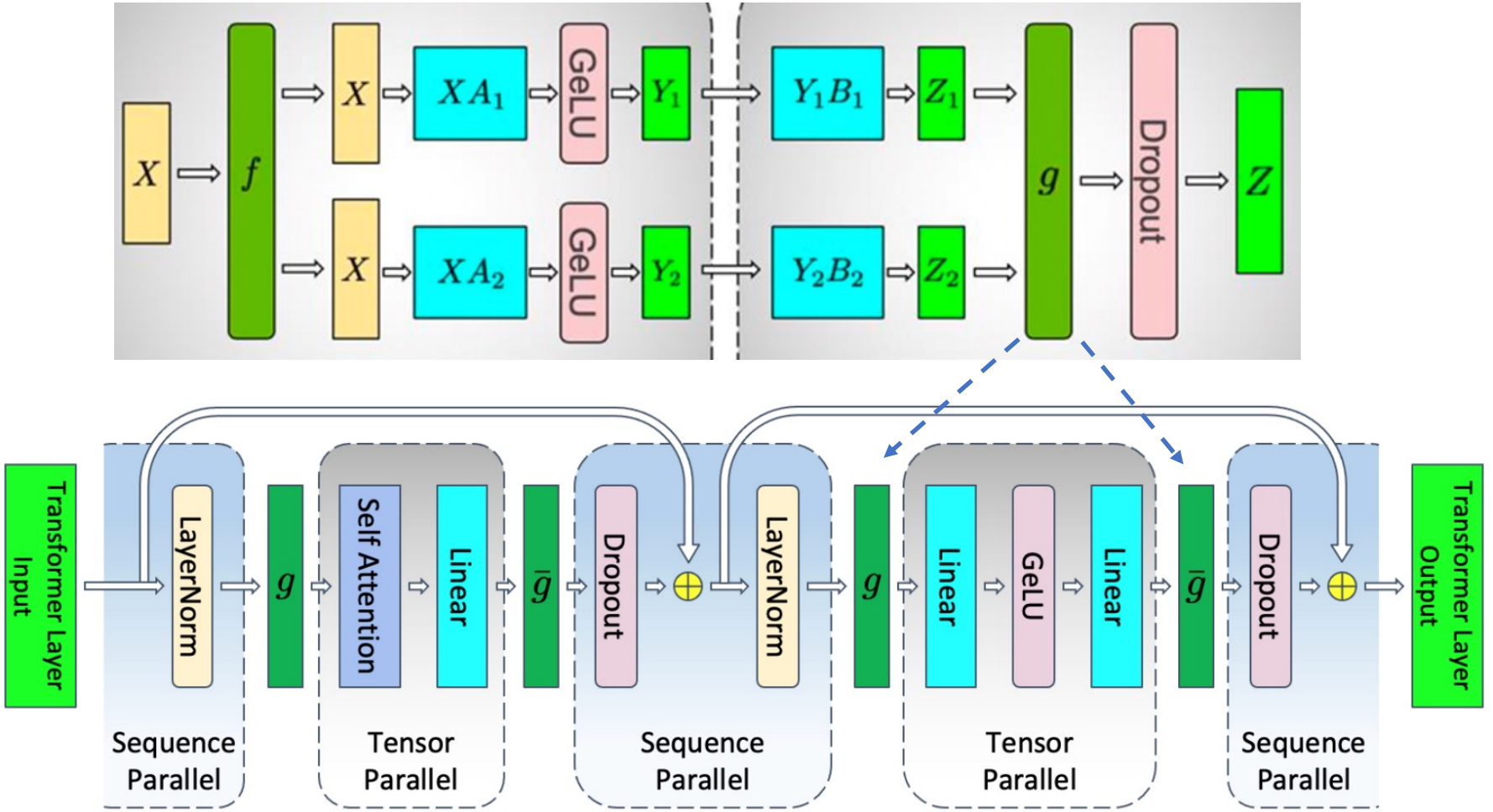
Worked example from the source lecture — batch_size = 64, hidden_size = 4096, seq_len = 32768, SP degree t = 8:

quantity	without SP	with SP (t = 8)
activation memory per block	86 GB	10.7 GB
scaling	10bsh	10bsh / 8

Takeaway: an 8× memory reduction on the residual activations, no extra communication load.

Sequence Parallelism

- **ZERO** extra communication overhead



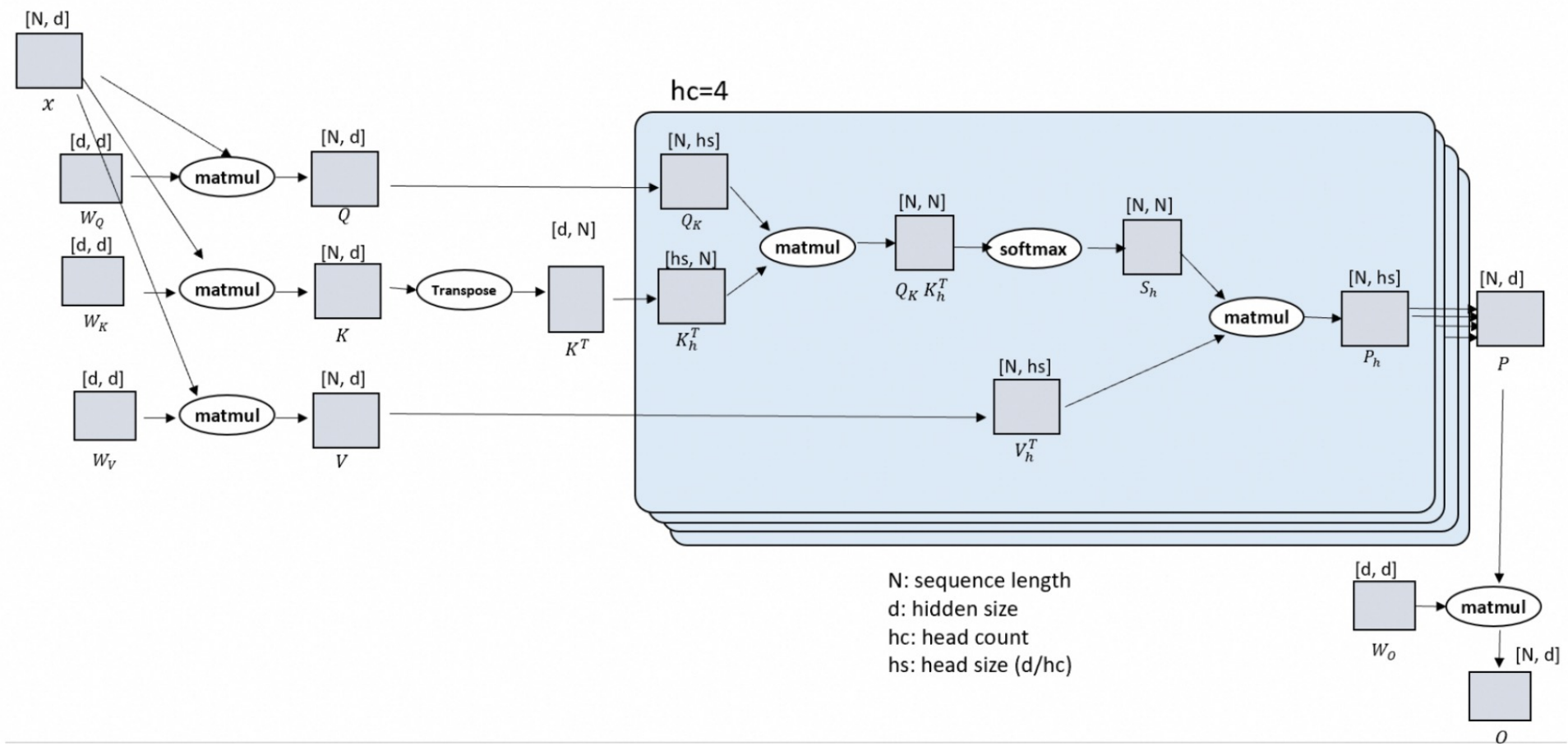
Sequence Parallelism

- DeepSpeed Sequence Parallelism
  - Megatron SP in fact slice activations at the token (sequence) dimension

How about slicing sequences in the very beginning?

Sequence Parallelism

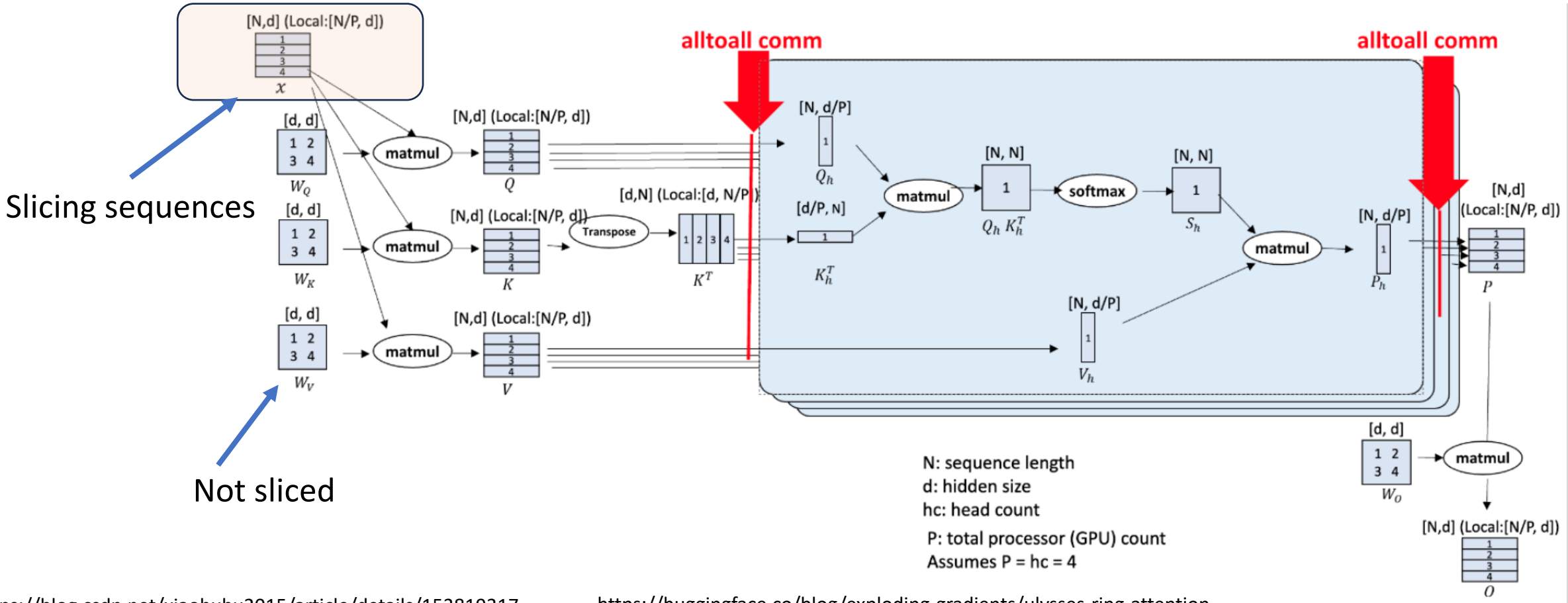
- Before Ulysses



Sequence Parallelism

Model not partitioned!

- Ulysses Sequence Splitting

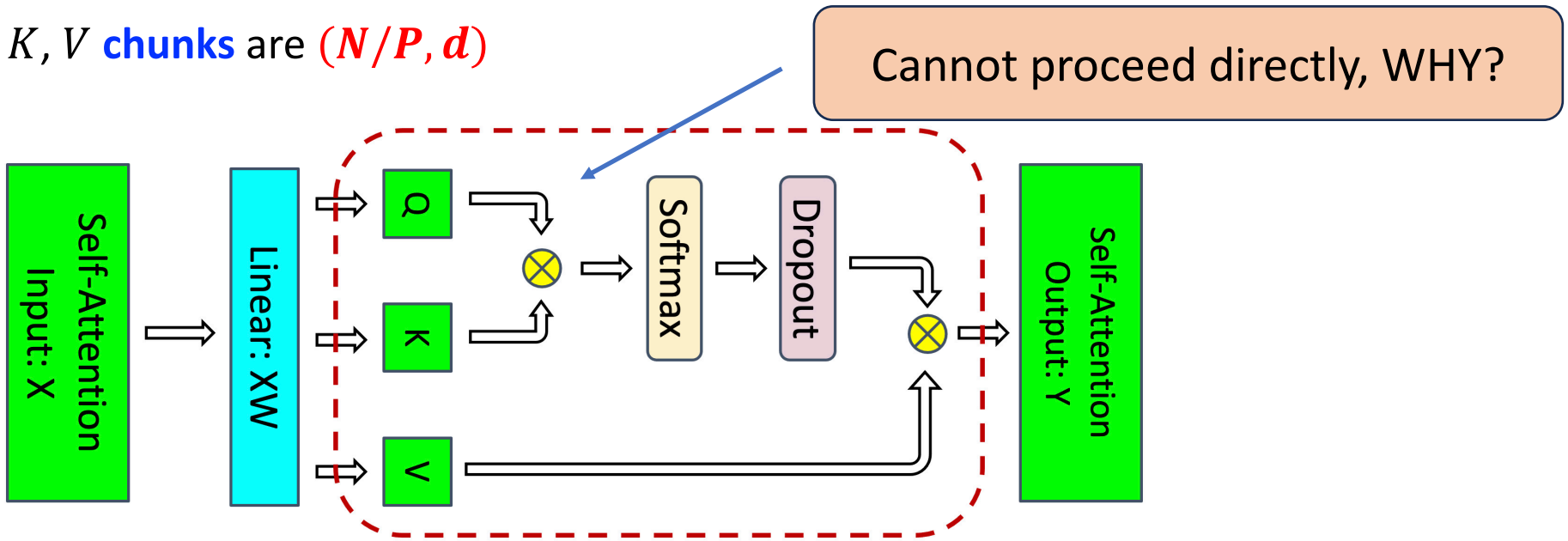


https://blog.csdn.net/xiaobubu2015/article/details/153819317

https://huggingface.co/blog/exploding-gradients/ulysses-ring-attention

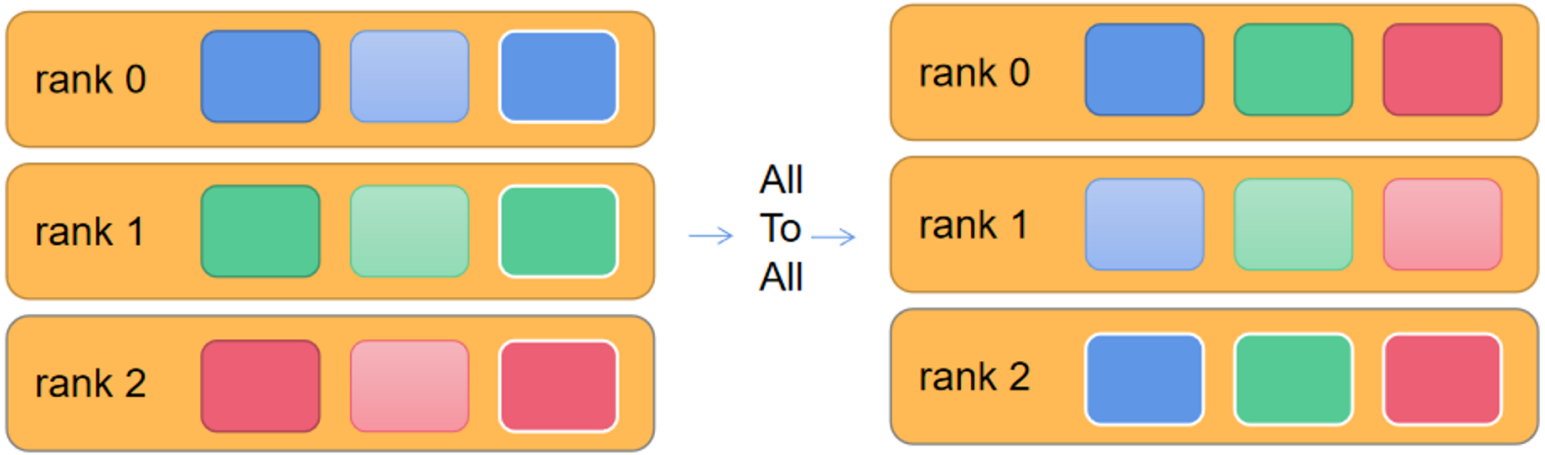
Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism
  - Slicing input *X=(N, d)*, with *N* for sequence length and *d* for hidden dimension
    - Each GPU has an input ***(N/P, d)***, with *d* for # of GPUs
  - Attention weight matrices W_Q, W_K and $W_V \subseteq \mathbb{R}^{d \times d}$ not partitioned
    - Sizes of Q, K, V **chunks** are $(N/P, d)$



Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism
  - All-to-All communication
    - Before all-to-all, each GPU holds the SEQUENCE CHUNK of **ALL HEADs** (i.e. $(N/P, d)$)
    - After all-to-all, each GPU holds ALL SEQUENCES on a specific HEAD (i.e. $(N, d/P)$)

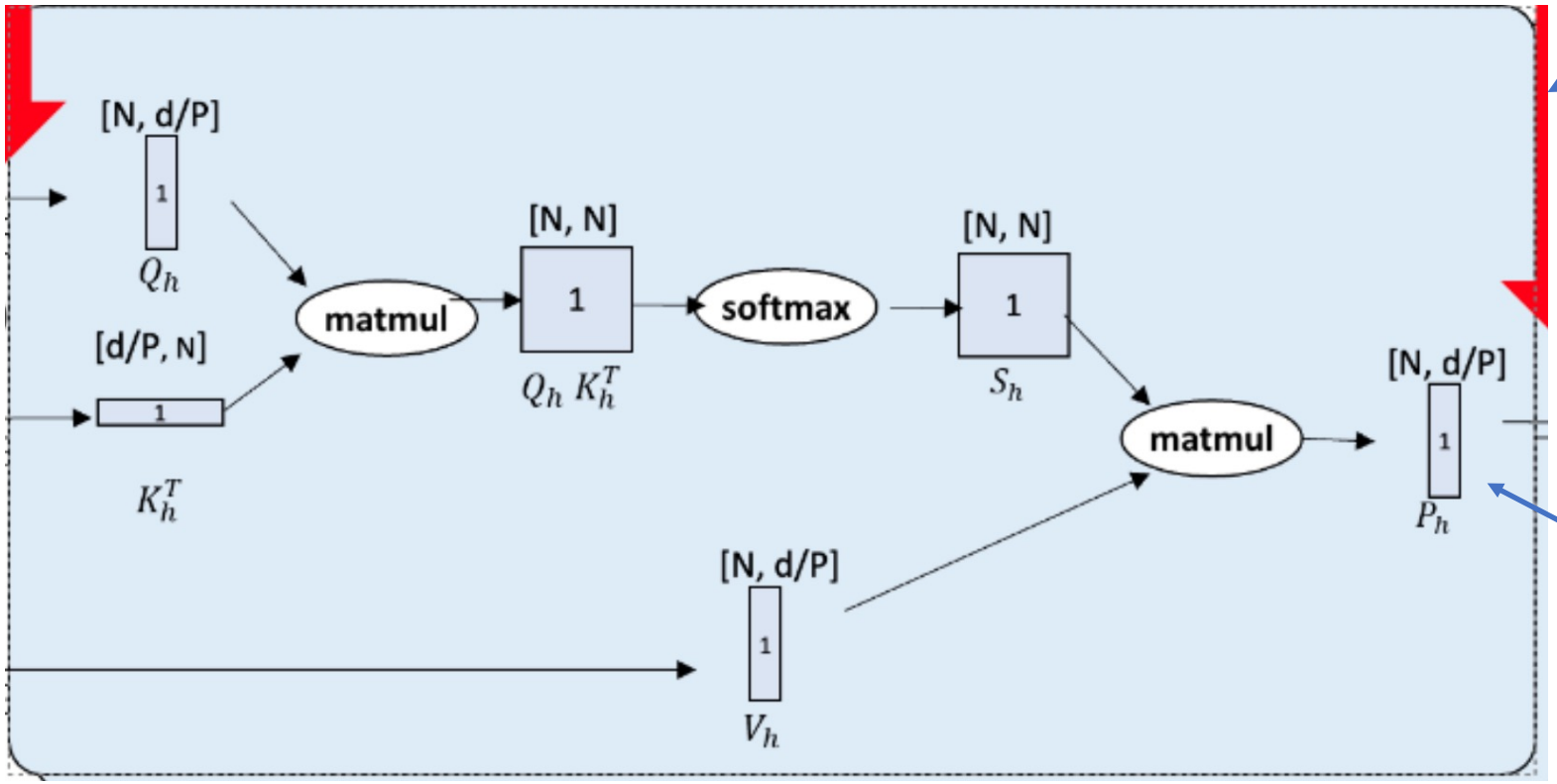


Looks like a transpose operation!

https://mdnice.com/writing/974038f2cc3d46239377b0928e8b9eea

Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism
  - All-to-All communication

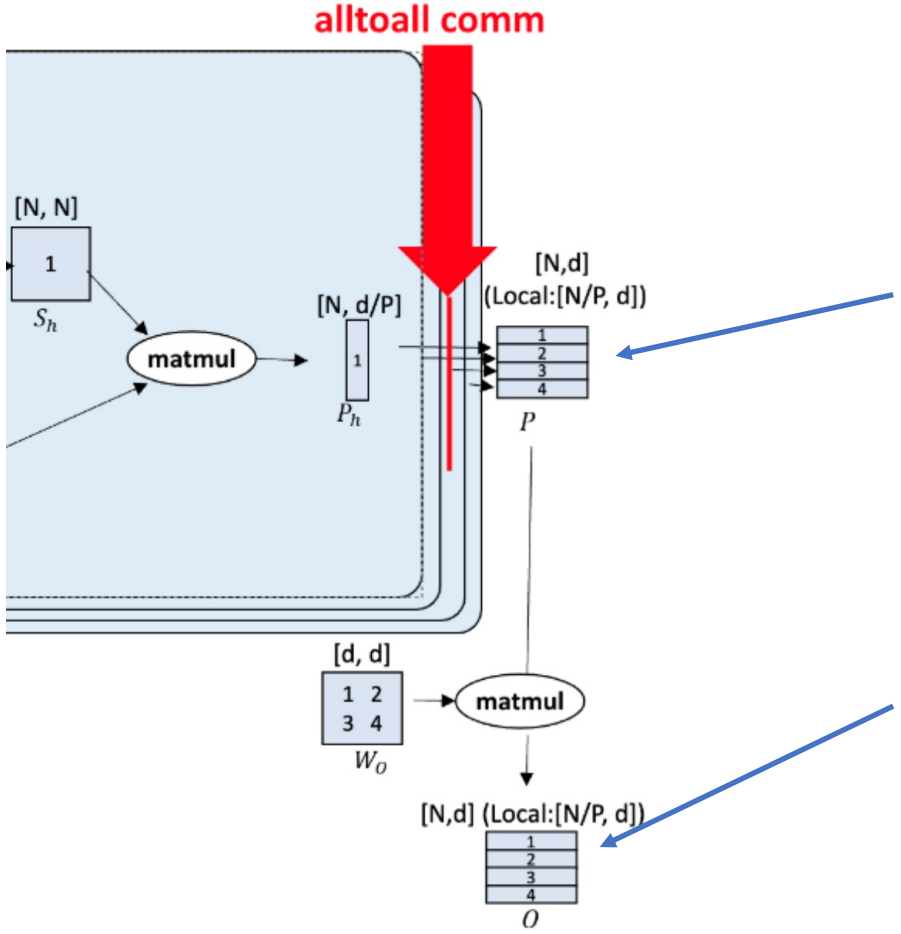


Multi-head Attention Computation

Output dimension: $(\boldsymbol{N}, \boldsymbol{d/P})$

Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism
- MLP can compute at each sequence chunk granularity
- DeepSpeed Ulysses backward pass



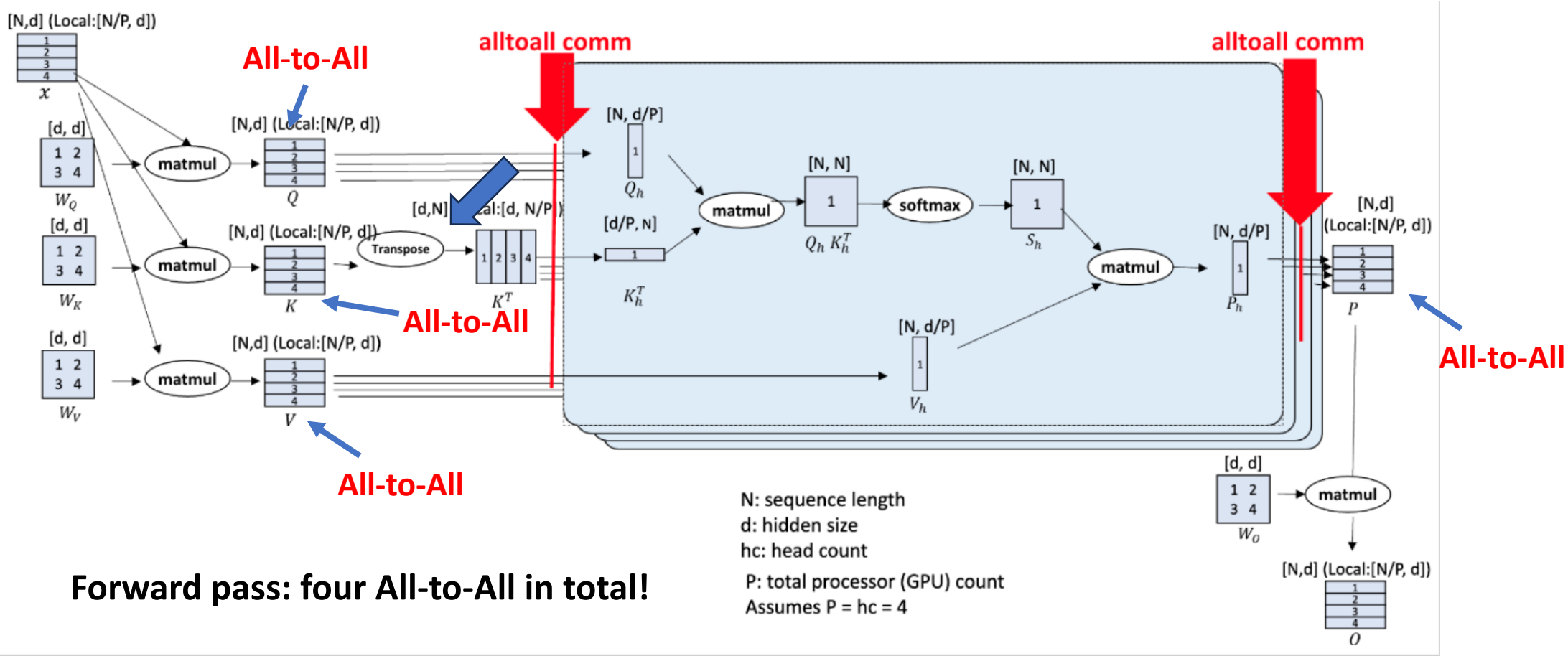
Output dimension: $\left(\frac{N}{P}, d\right)$

after ALL-to-ALL

W_O dimension: $\left(\frac{N}{P}, d\right)$

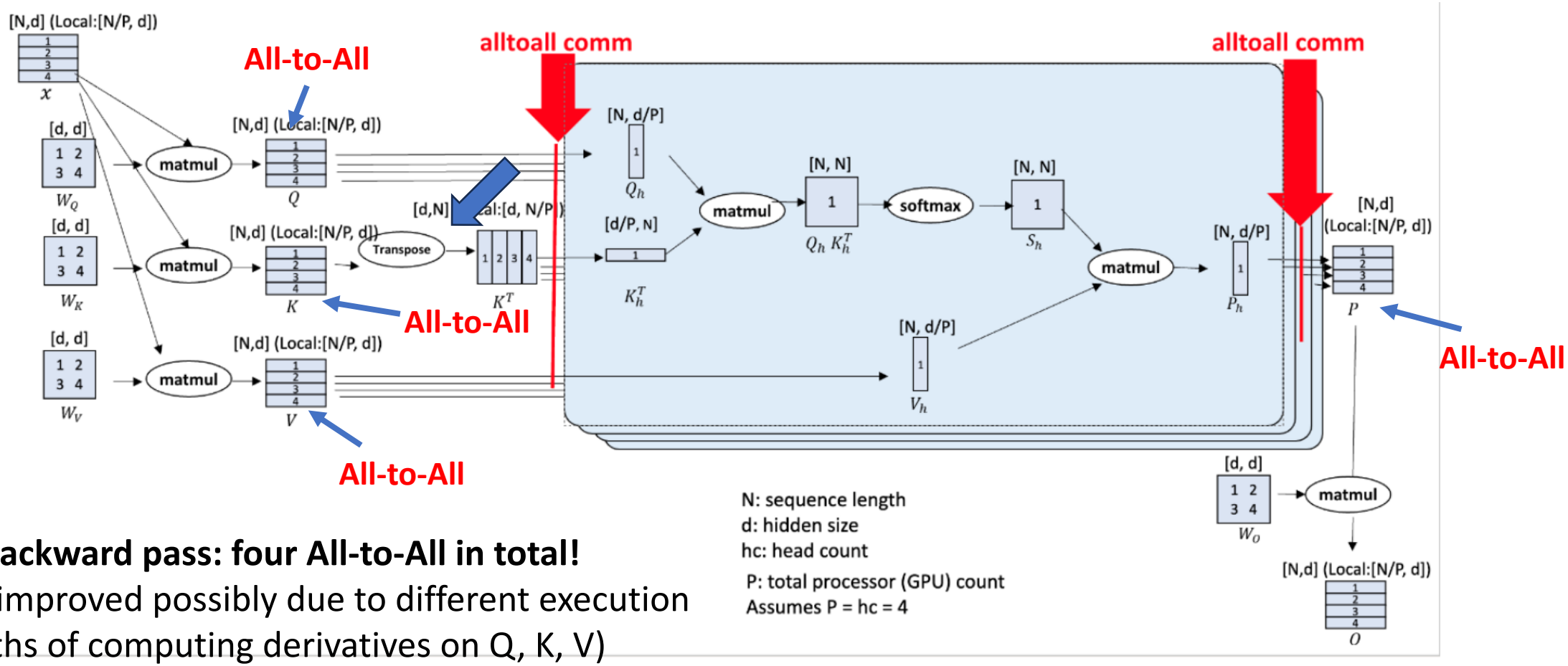
Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism: Communication Analysis



Sequence Parallelism

- DeepSpeed Ulysses Sequence Parallelism: Communication Analysis



Backward pass: four All-to-All in total!

(Can be improved possibly due to different execution paths of computing derivatives on Q, K, V)

Sequence Parallelism

- DeepSpeed Ulysses vs NVIDIA Megatron
  - Megatron SP **splits attention and MLP layer parameters** by using tensor parallelism, and splits **layernorm** and **dropout** inputs accordingly. The computations are carried out on the heads at different GPUs.

  - Ulysses splits attention and MLP by segmenting the input sequences, and the computations are carried out on the heads at different GPUs. **No segmentation** on the attention layer parameters.

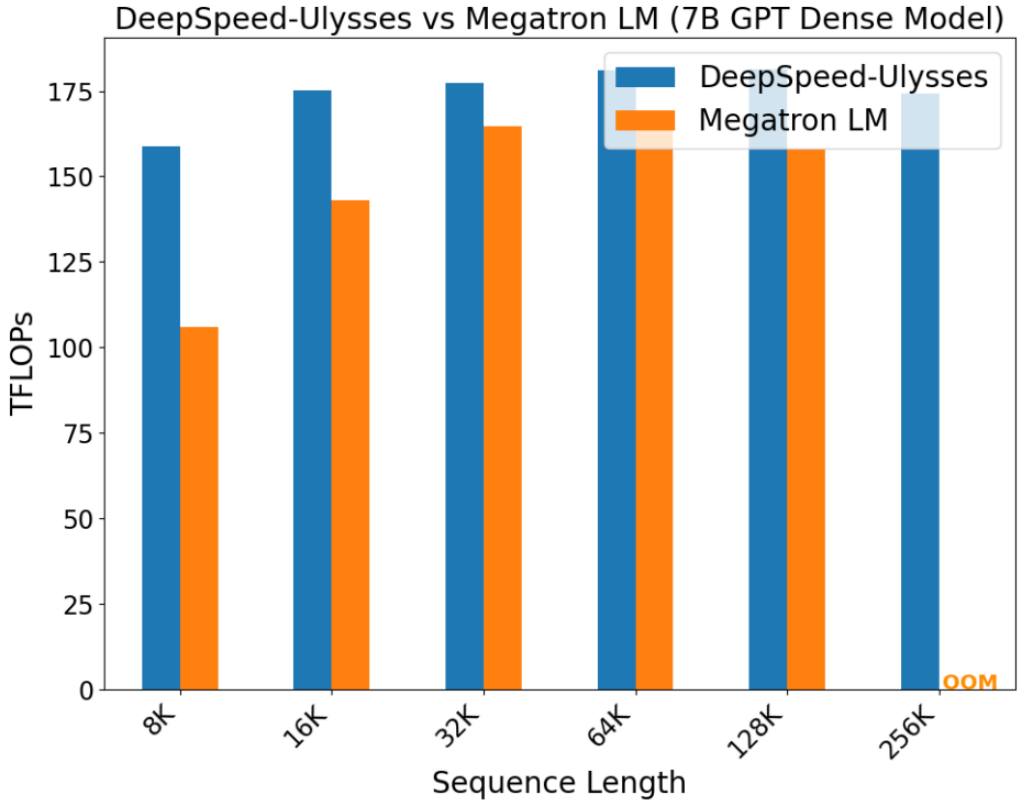
Sequence Parallelism

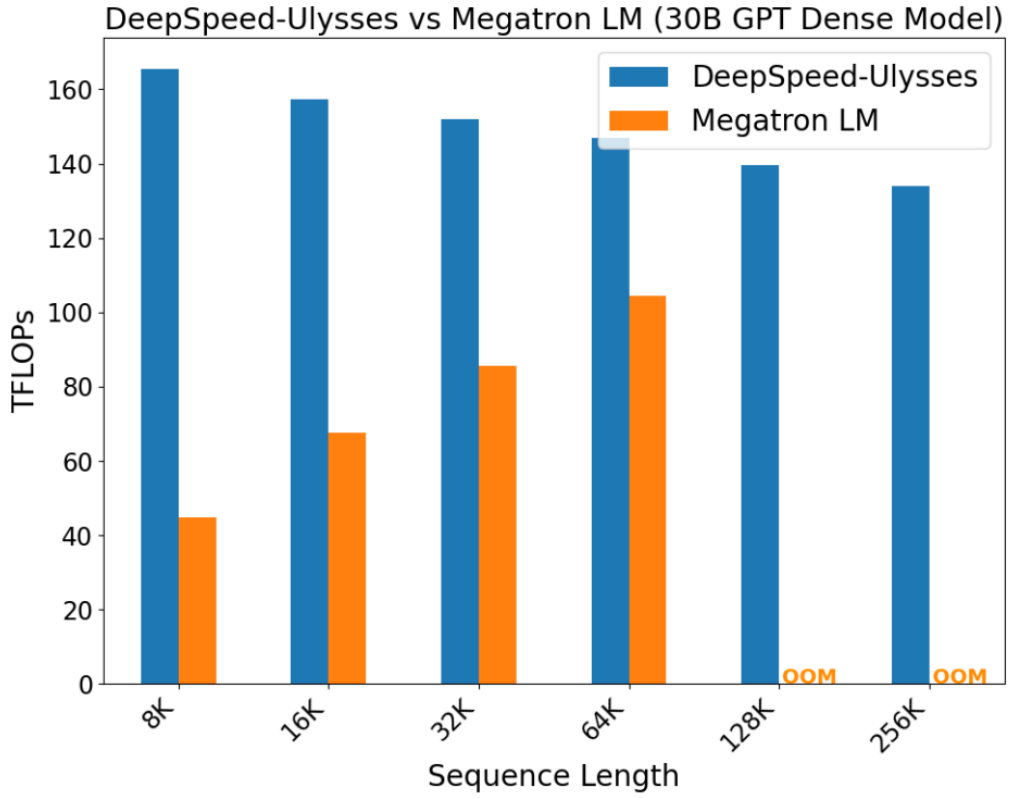
- Ulysses versus Megatron-LM SP

	Attention	**MLP**
Megatron-LM (combined with TP)	***4 N*B*H*(P-1)/P*Bytes*** Forward: ReduceScatter + AllGather, Backward: ReduceScatter + AllGather	***4 N*B*H*(P-1)/P*Bytes*** Forward: ReduceScatter + AllGather, Backward: ReduceScatter + AllGather
DeepSpeed Ulysses	***8 N*B*H*(P-1)/P^2*** Forward: 4 All-to-All, Backward: 4 All-to-All	***0*** MLP is element-wise, naturally fitting SP

Sequence Parallelism

- Ulysses vs Megatron-LM SP





Sequence Parallelism

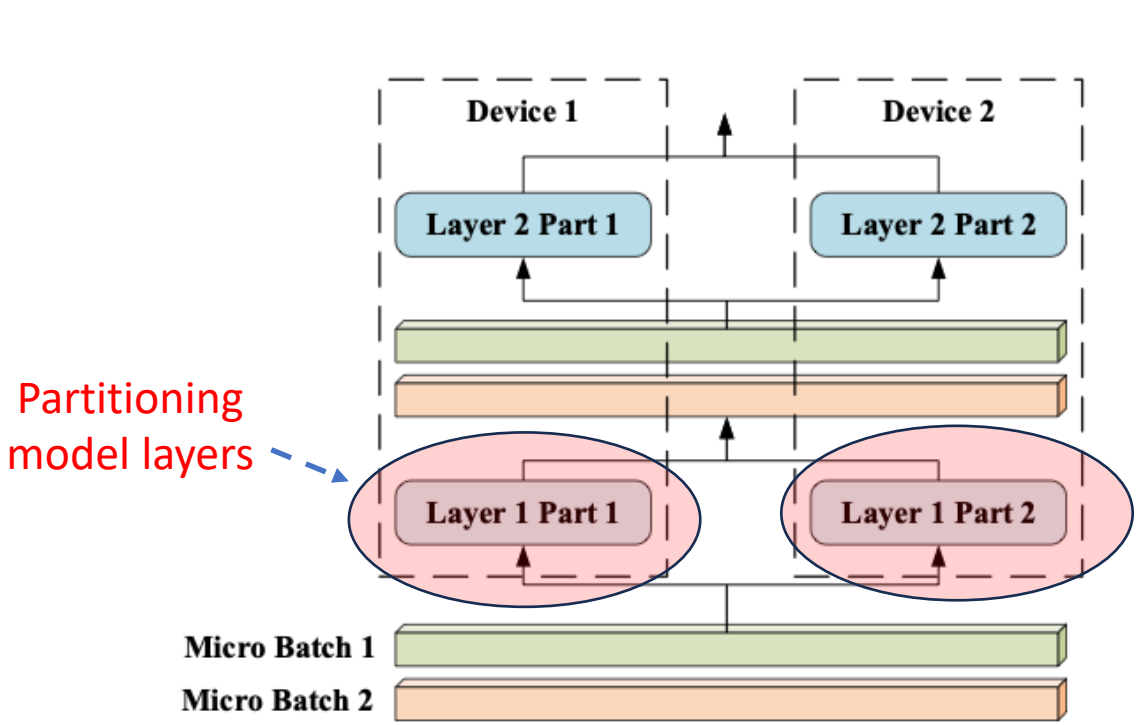
- Ulysses versus Megatron-LM SP
  - N = 1M tokens, B = 1, H = 4096, P = 8, FP16
  - Megatron-LM: $8 \times N \times B \times H \times (P-1)/P = 8 \times 1e6 \times 4096 \times 2 \times 7/8 = 57.344$ GB
  - Ulysses: $8 \times N \times B \times H \times (P-1)/P/P = 57.344$ GB / 8 = 7.168 GB

"Ulysses reduces communication volume by eliminating All-Gather in MLP and using All-to-All only in attention, achieving near-linear scaling for sequence lengths up to 1M."

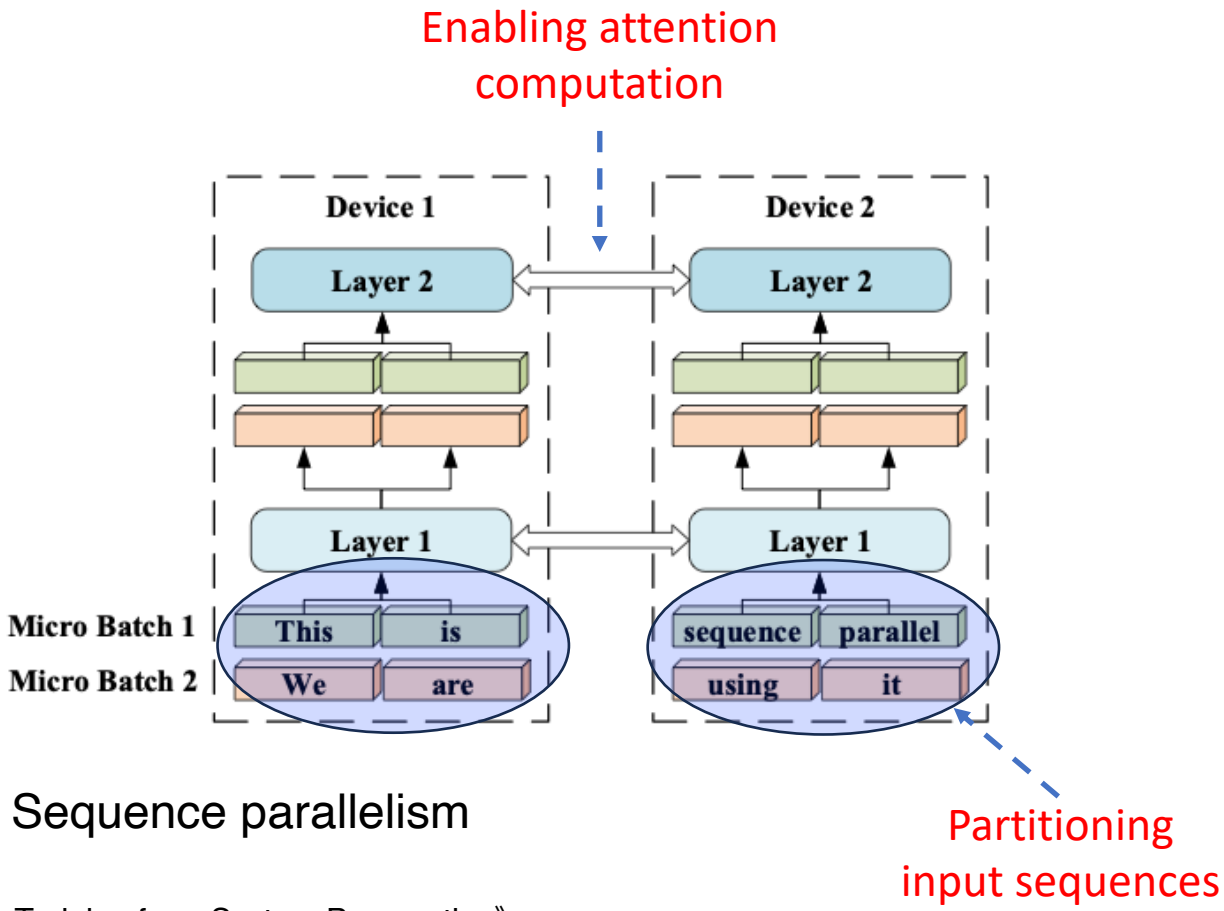
"Megatron-LM sequence parallelism incurs a communication volume P times larger than that for DeepSpeed sequence parallelism."

Sequence Parallelism

- Colossal AI Sequence Parallelism



Tensor parallelism



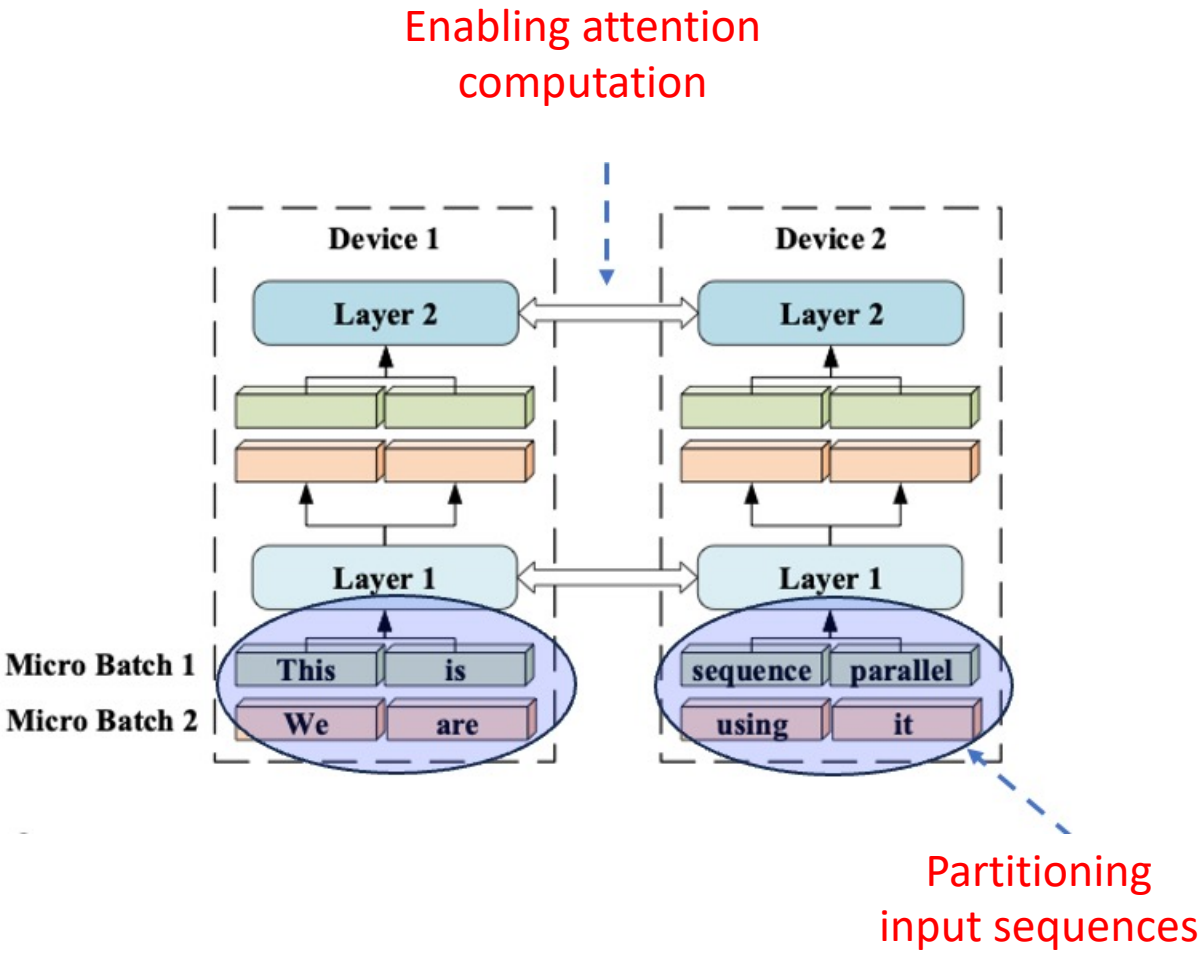
Sequence parallelism

《Sequence Parallelism: Long Sequence Training from System Perspective》

Sequence Parallelism

- Colossal AI Sequence Parallelism
  - Q, K, V partitioned on the **sequence**
  - Computing attention scores need full Q, K, V on all tokens, in which communication is inevitable

- **Ring Attention** adopted in Colossal-AI



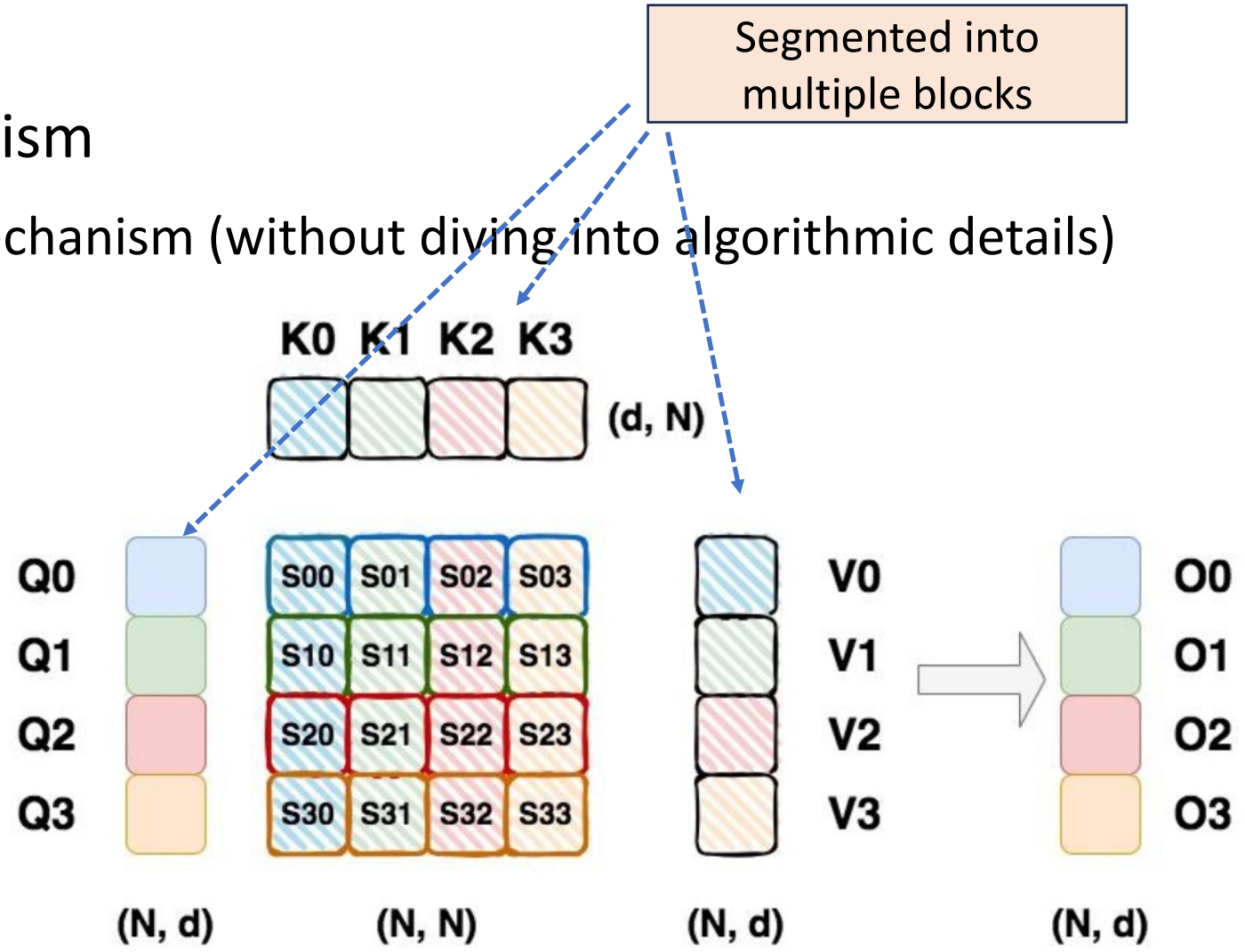
Sequence Parallelism

- Colossal AI Sequence Parallelism
  - Convoluted with Attention mechanism (without diving into algorithmic details)

Step 1: for each Q_i, compute attention score by feeding K_j and V_j, and obtain O_{ij}

Step 2: for each K_j and V_j, repeat Step 1 and update O_{ij}, and eventually obtain O_i

Step 3: change Q_i to Q_{i+1}, repeat Step 1 and Step 2 until Q_C

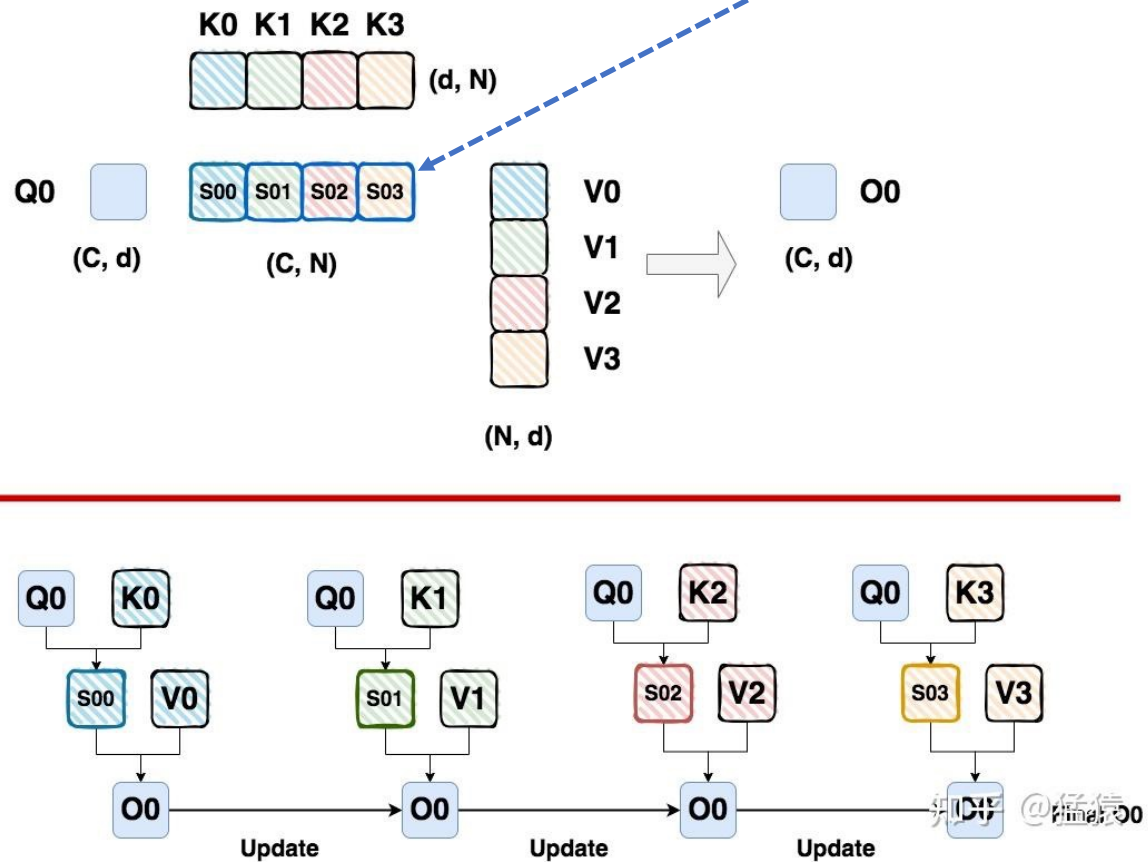


From https://zhuanlan.zhihu.com/p/4963530231

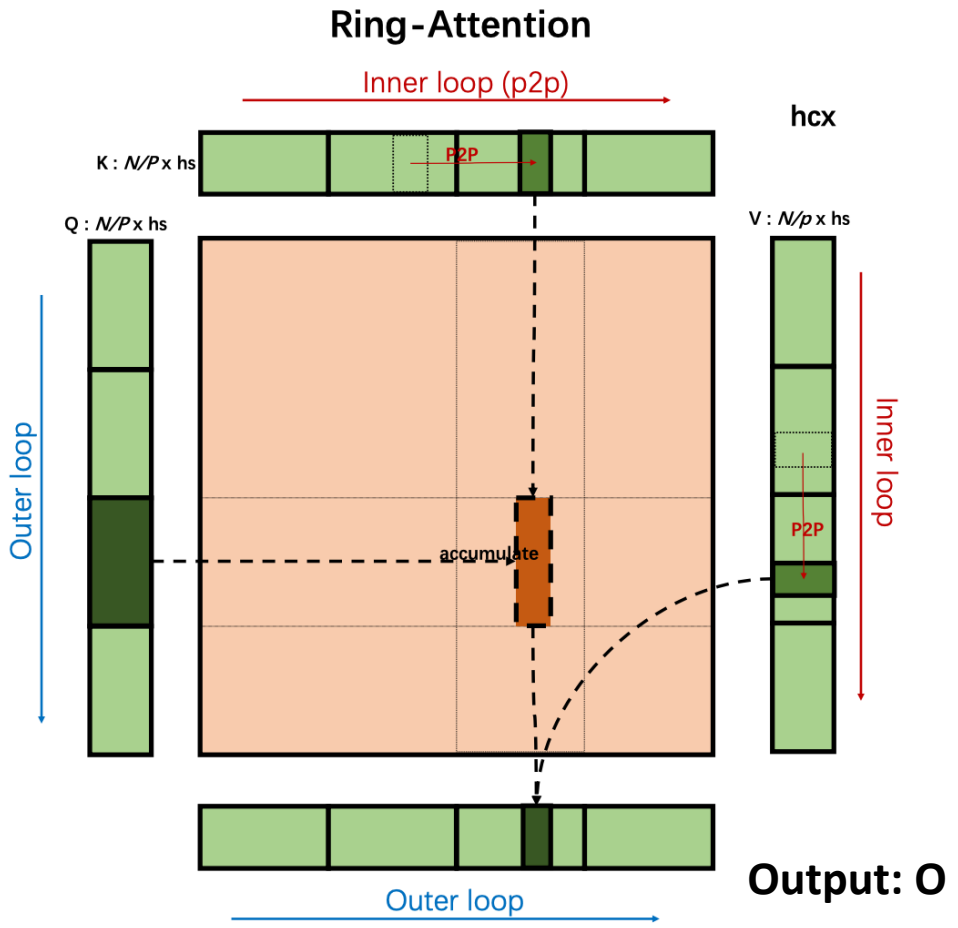
Sequence Parallelism

Mathematical deduction shows that S_{00}, S_{01}, S_{02}, and S_{03} can be computed in an arbitrary order!

- Colossal AI Sequence Parallelism



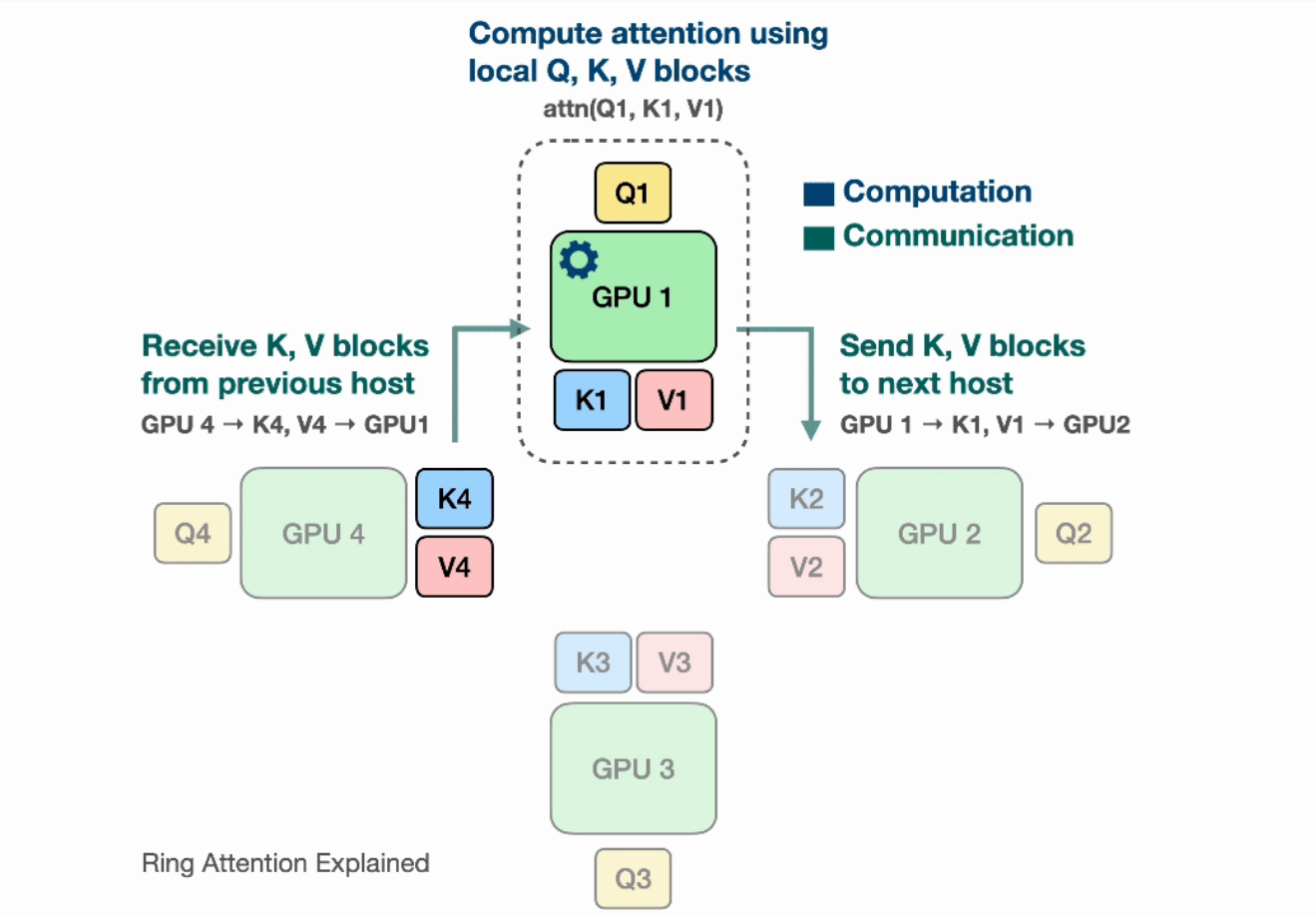
https://www.themoonlight.io/en/review/usp-a-unified-sequence-parallelism-approach-for-long-context-generative-ai



From https://zhuanlan.zhihu.com/p/496530231

Sequence Parallelism

- Colossal AI Sequence Parallelism



Step 1: *GPU1 computes O1 using Q1, K1 and V1 (other GPUs do the same thing)*

Step 2: *GPU1 sends K1 and V1 to GPU2, and receives K4 and V4 from GPU4; compute and update O1 using Q1, K4 and V4*

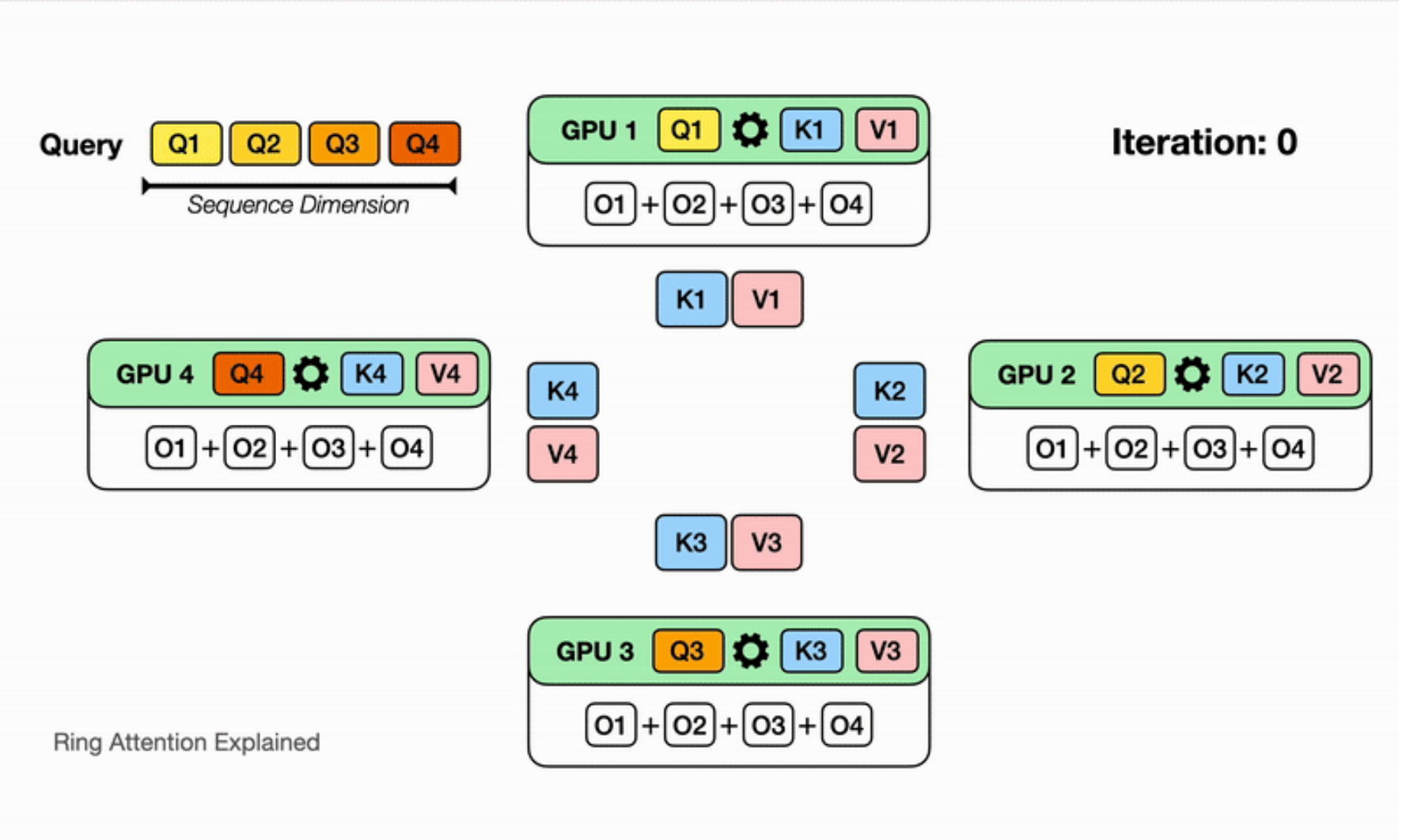
Step 3: *GPU1 sends K4 and V4 to GPU2, and receives K3 and V3 from GPU4; compute and update O1 using Q1, K3 and V3*

Step 4: *GPU1 sends K3 and V3 to GPU2, and receives K2 and V2 from GPU4; compute and update O1 using Q1, K2 and V2*

The network communications are illustrated by the blinking arrows.

From https://coconut-mode.com/posts/ring-attention/

Sequence Parallelism



Illustrate for N devices, it will take N iterations to finish the whole output comptutation. Watch for each iteration on each device, different partial sum of the output is computed with the K, V block it currently has, and it eventually sees all the K, V blocks and has all the partial sum for the output!

From https://coconut-mode.com/posts/ring-attention/

Sequence Parallelism

- Recap
  - Megatron CP: sharding weights and some inputs on the hidden dimension
  - Ulysses CP: sharding input token sequence
  - Ring Attention CP: letting KV blocks flow on the ring

Thanks!

