Distributed Training

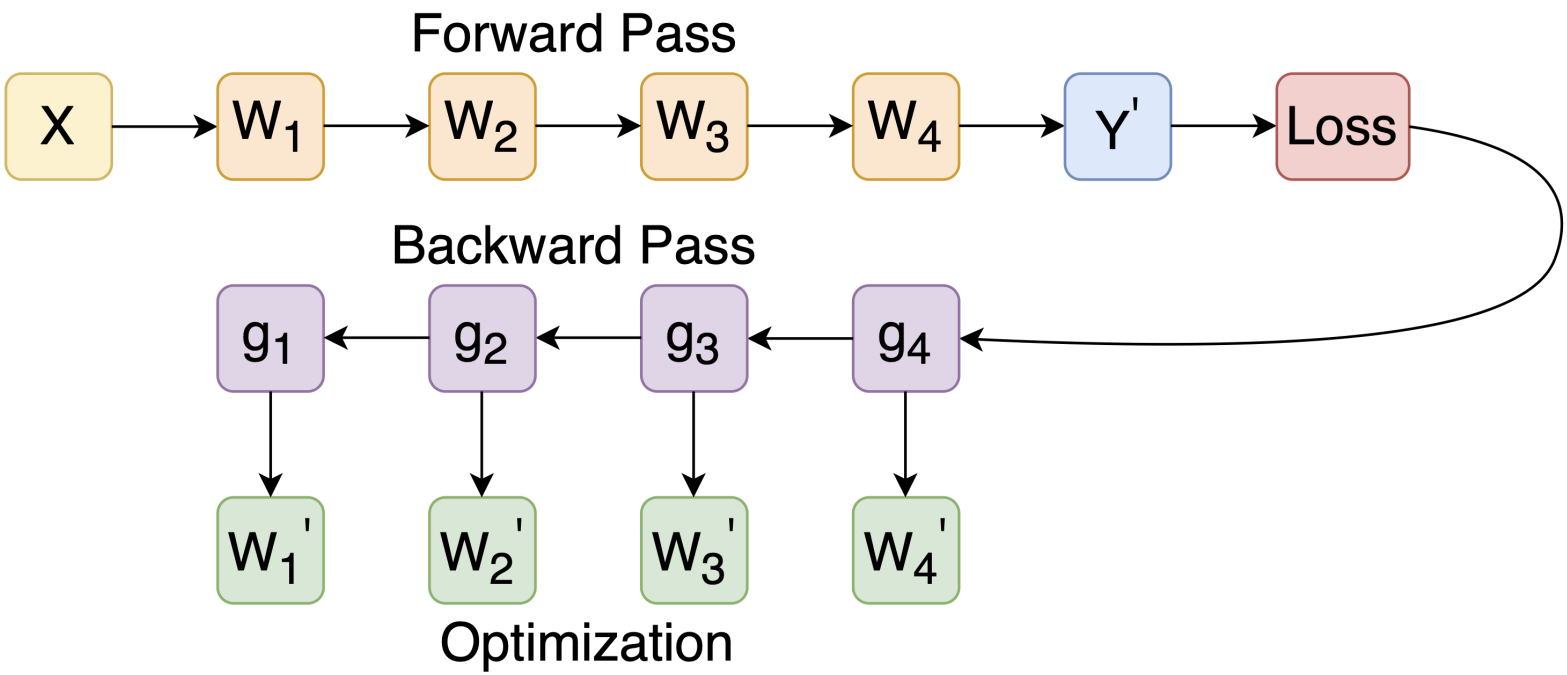
Systems for Machine Learning

Chapter 6

https://www.cse.iitb.ac.in/~mythili/sysml/

Training

- Forward pass, backward pass, optimizer step
- **Activations** kept till backward pass complete
- Memory needed for **optimizer state** also



Batching

- **Batch** of inputs is processed together
  - Gradients are accumulated across the entire batch
- Trade-off between speed of convergence and memory
  - Larger batch size reduces total number of steps, and makes **training faster**
  - Larger batch size also needs **more activations** held in memory

Memory Usage During Training

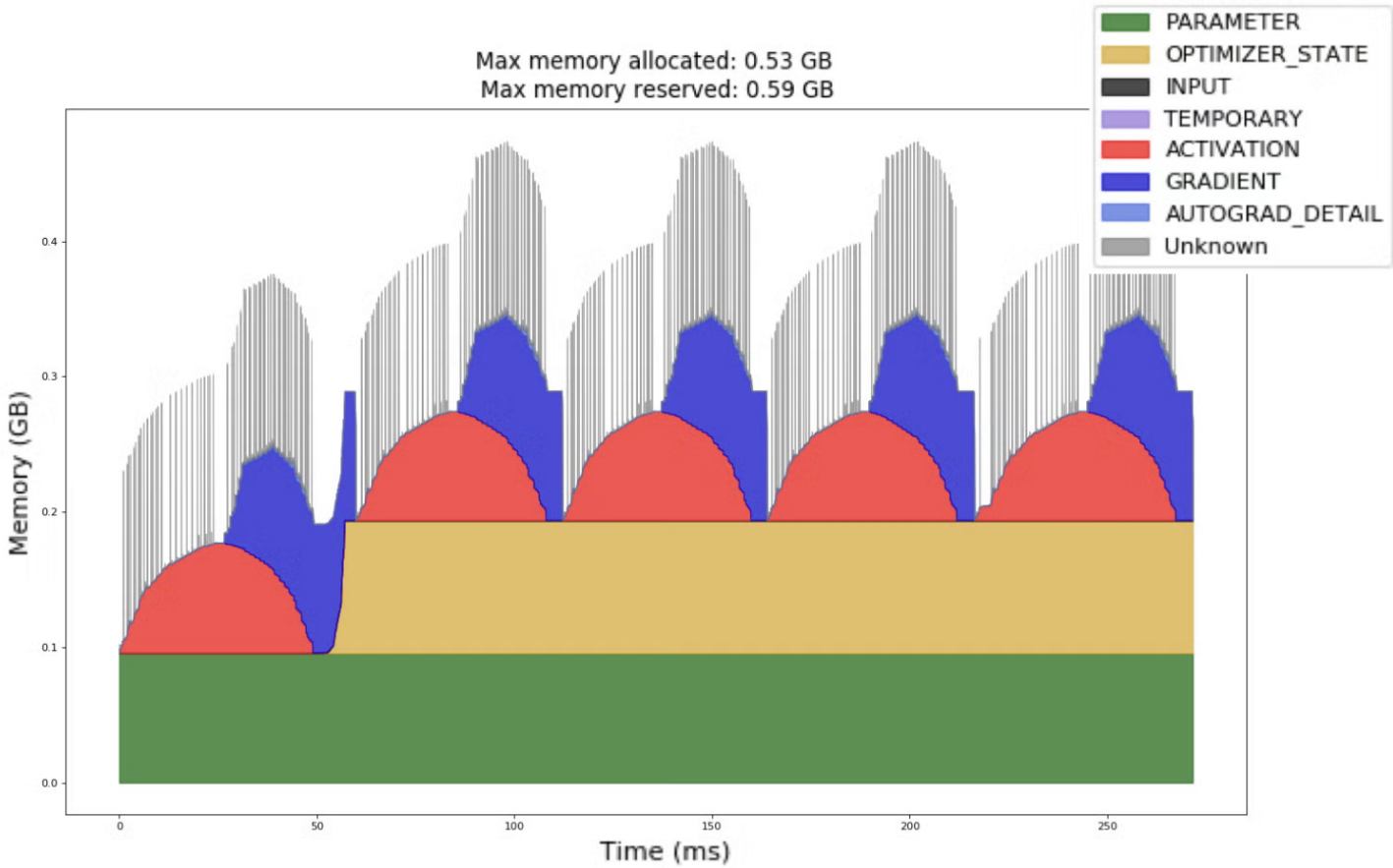


Image Credit: https://arxiv.org/abs/2407.21783

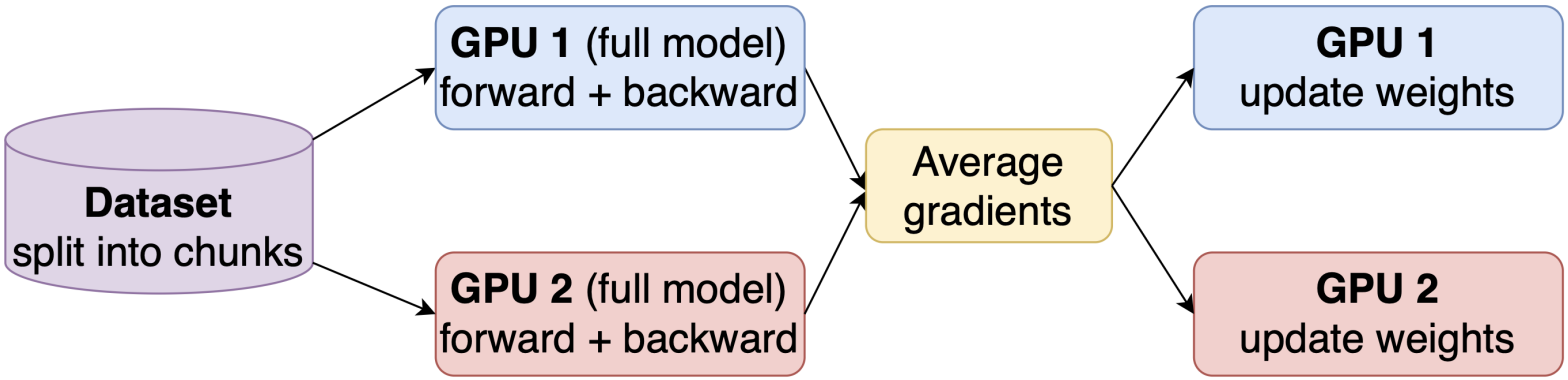
Training Large Models

- Compute and memory of training exceed single machine capacity, must be distributed and parallelized
- Modern LLMs trained over thousands of GPUs across weeks
- Two foundational strategies:
  - **Data parallelism** – training data distributed, each GPU holds full copy of the model
  - **Model parallelism** – model itself distributed across GPUs

Data Parallelism

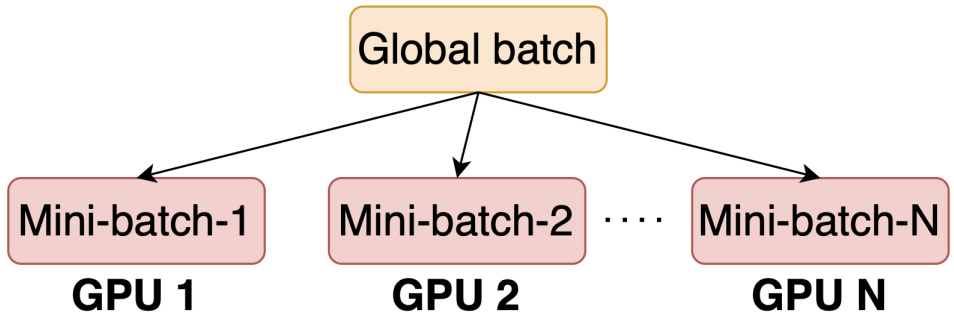
Data Parallelism

- Training data split into chunks, **one chunk per GPU**
- Each GPU is given an **identical** copy of the model
- Before updating model weights, gradients are combined
- **Degree** – number of GPUs involved



Data-parallel Training

- Each device computes partial sum of gradients over its own **mini-batch**
- Final **average** gradient used for the weight update
- Preserves statistical properties of gradient descent
- Amount of data exchanged grows with the number of **parameters** and the **degree** of parallelism

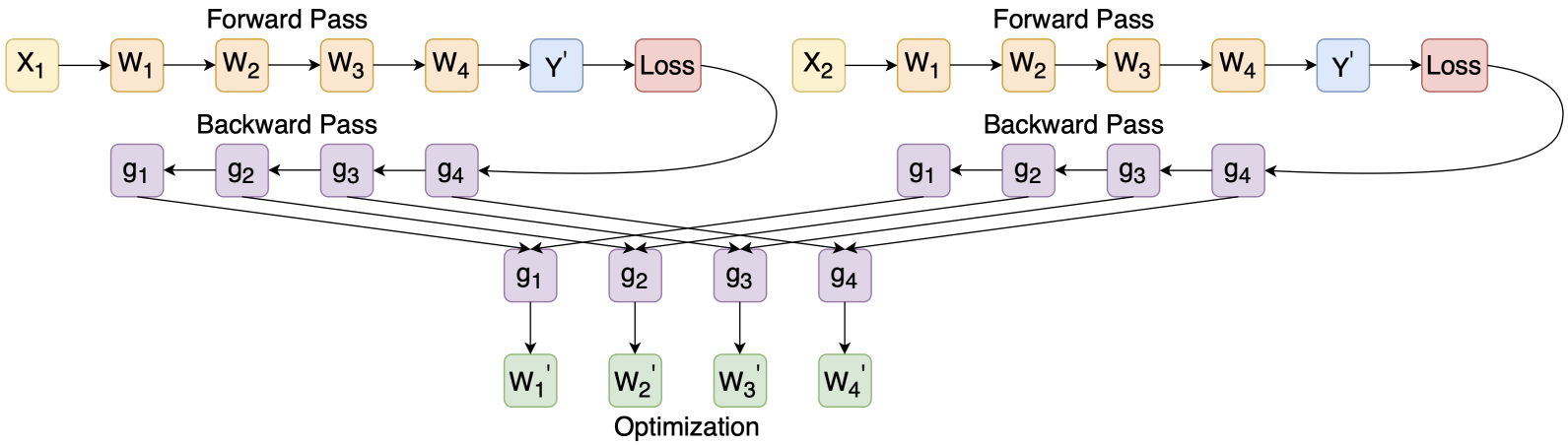


Batch Size Trade-off

- A small mini-batch completes **quickly**, but:
  - Requires **more communication**
  - **Noisier** gradient due to fewer samples
- A large mini-batch increases the **memory** required to store activations
- Choose the largest mini-batch size that **fits** within the memory of a single GPU

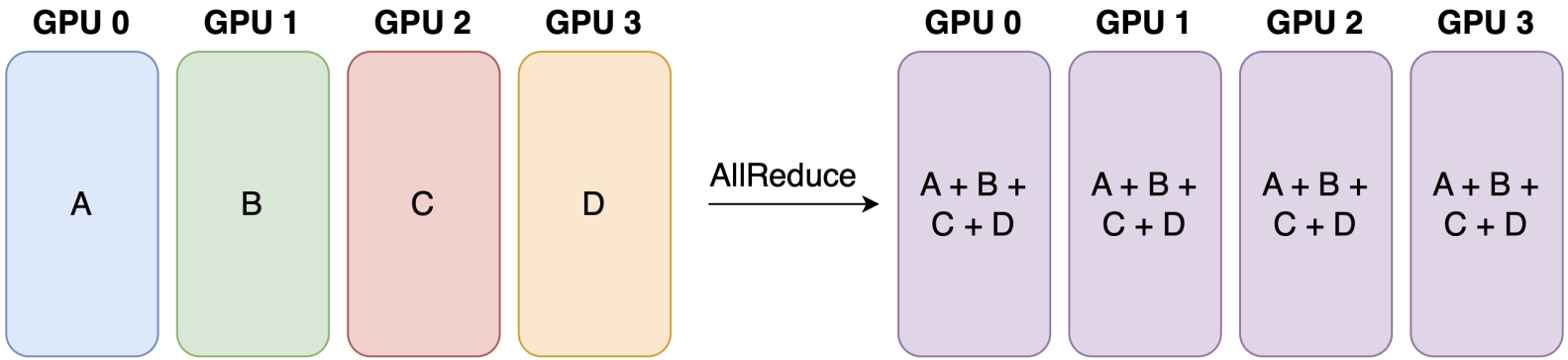
Gradient Accumulation

- Break mini-batch into smaller **micro-batches**, process them one after another on the same GPU
- Only the activations belonging to the current micro-batch held in memory
- Allows us to reach a **larger mini-batch size**



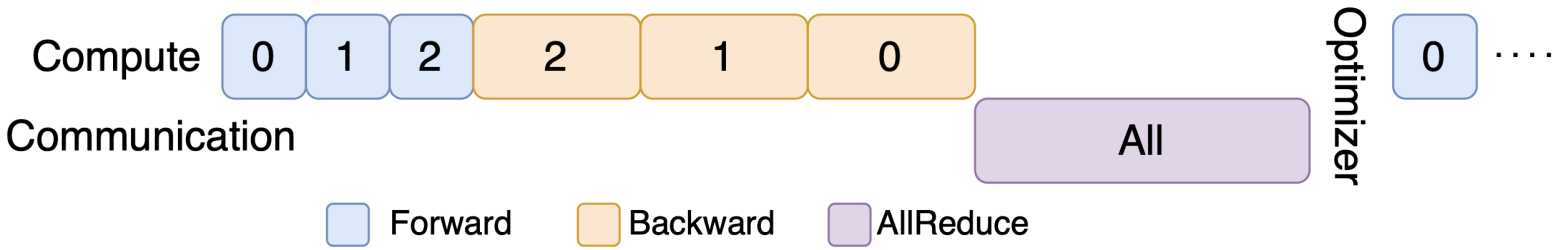
Synchronizing Gradients

- Gradients **combined** across all the GPUs before weight update applied
- AllReduce collective communication primitive – every device ends the operation holding the sum (reduction)

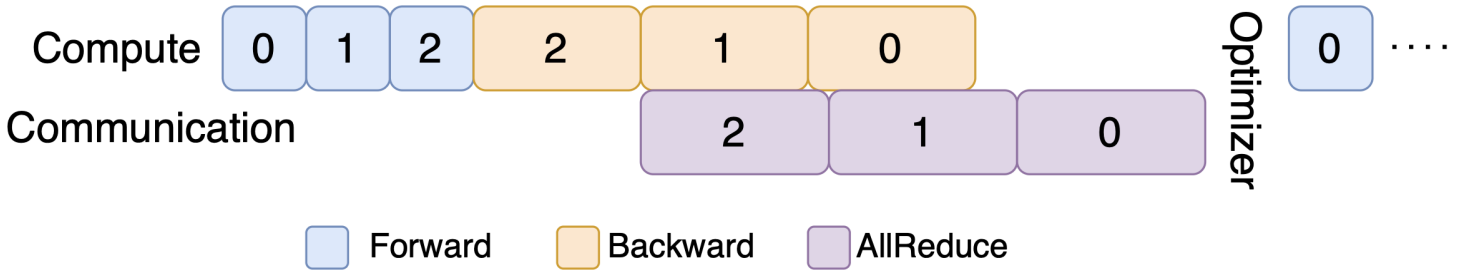


Wait-free backpropagation

- Overlaps communication of gradients during AllReduce with compute



(a) Data-parallel training without wait-free backpropagation



(b) Data-parallel training with wait-free backpropagation

Limits to Scaling DP

- As degree increases, beyond a certain point, **communication cannot be hidden**
- **Bursty** nature of the communication pattern
- Cannot scale to thousands of GPUs

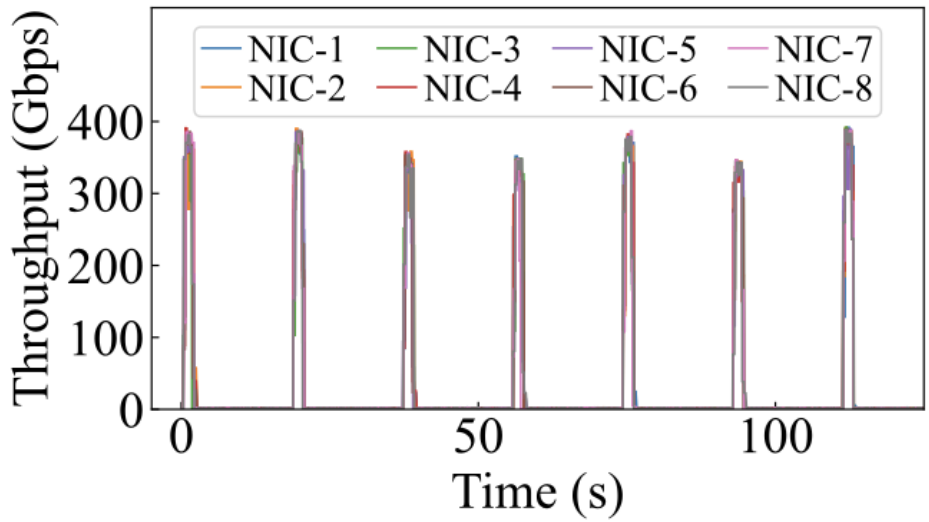


Figure: NIC egress traffic pattern during production model training

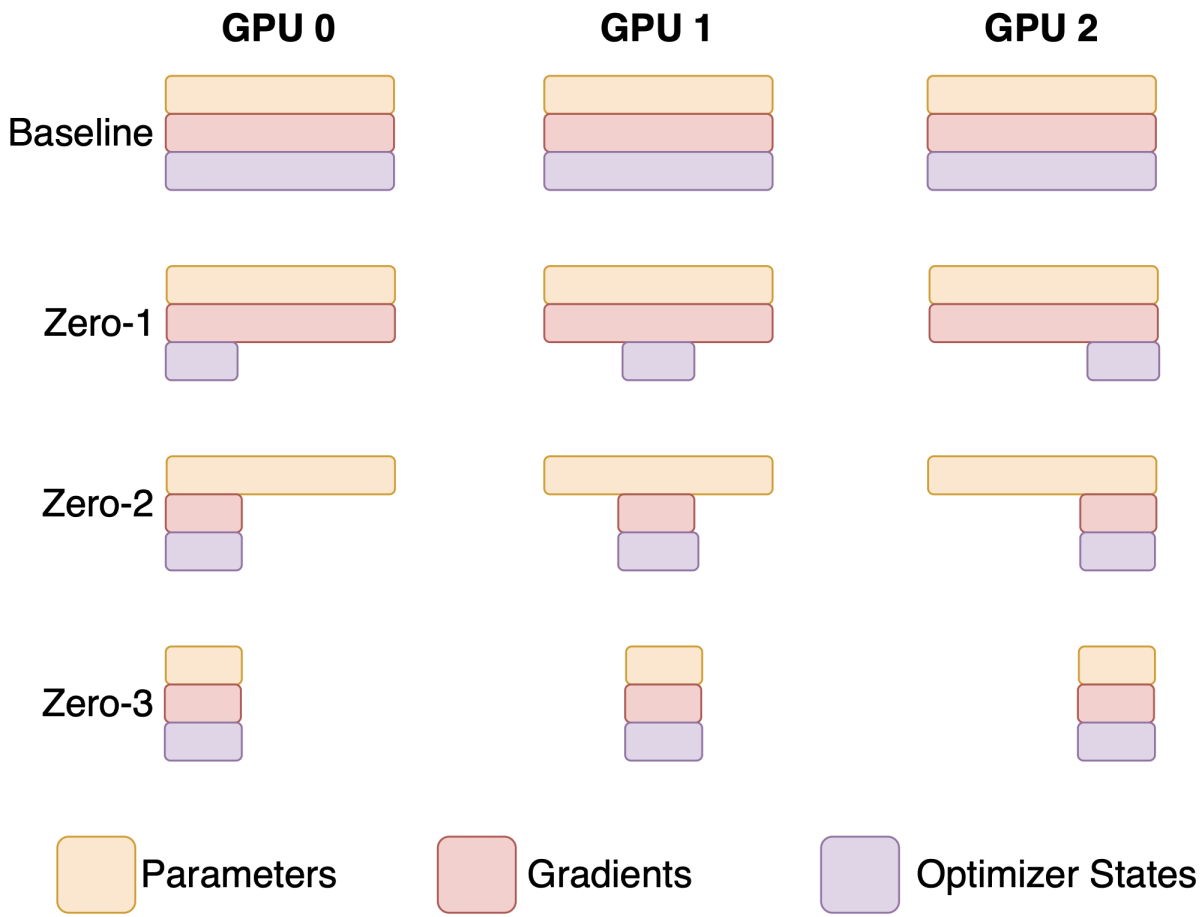
Image Credit: https://doi.org/10.1145/3651890.3672265

ZeRO

ZeRO

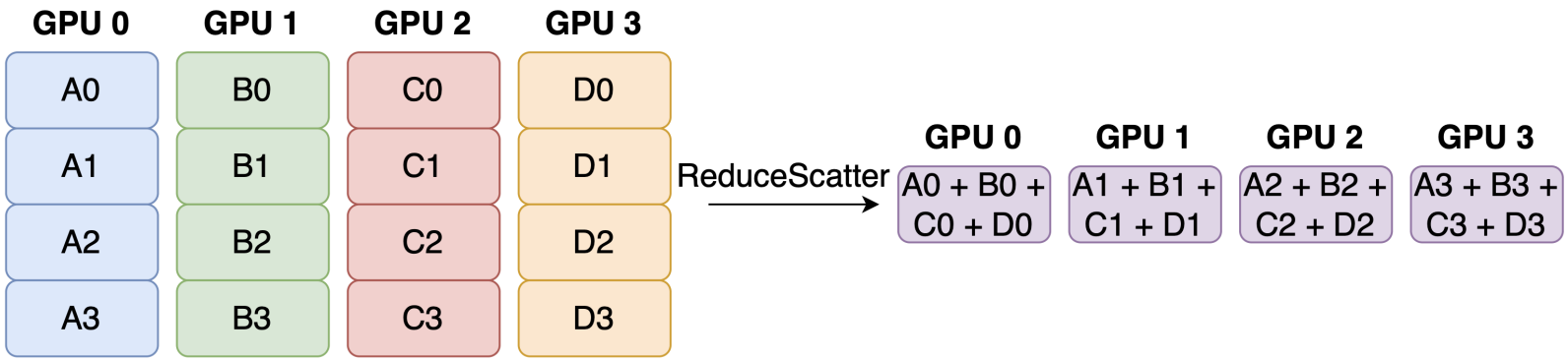
- DP requires full model at every GPU, high memory footprint
- **Zero Redundancy Optimizer (ZeRO)** – shards model states of parameters, gradients, optimizer across GPUs
- Every GPU still runs the forward and backward pass over the entire model
- Whenever full version required, GPUs temporarily **reconstruct** full copy by communicating with each other

ZeRO Stages

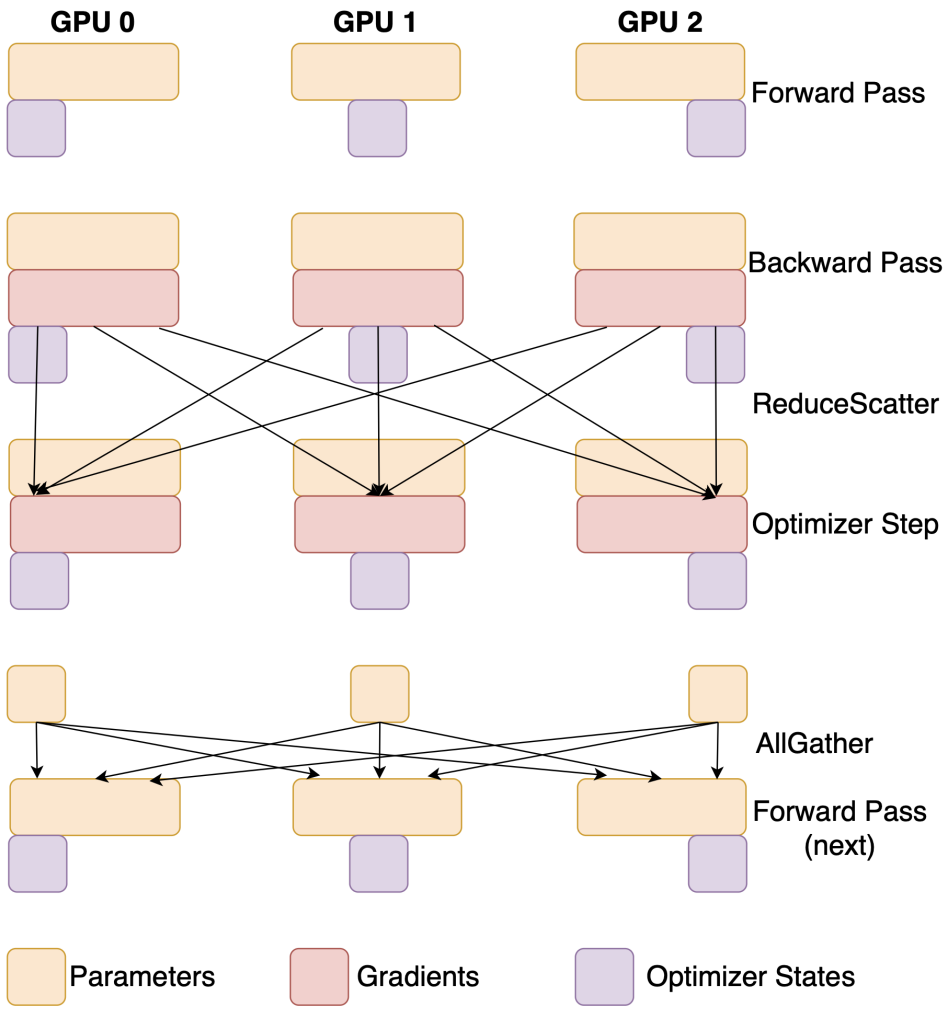

GPU 0
GPU 1
GPU 2
Baseline
Zero-1
Zero-2
Zero-3
Parameters
Gradients
Optimizer States

ZeRO-1

- Shards only the **optimizer states**
- ZeRO-1 performs a ReduceScatter
  - Result of summing gradients is **scattered**
  - Each GPU owns slice of the summed gradient corresponding to the shard it is responsible for

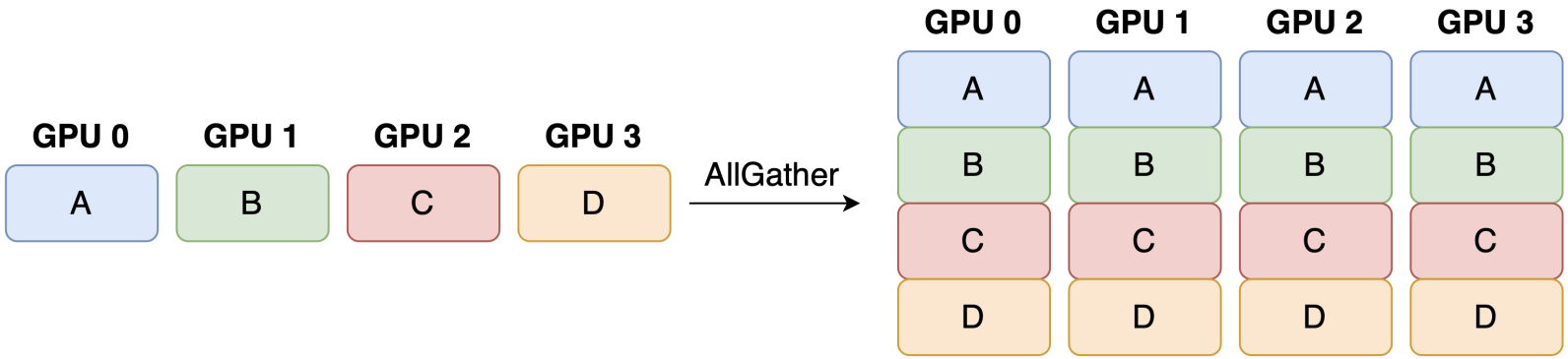


ZeRO-1


GPU 0
GPU 1
GPU 2
Forward Pass
Backward Pass
ReduceScatter
Optimizer Step
AllGather
Forward Pass (next)
Parameters
Gradients
Optimizer States

ZeRO-1

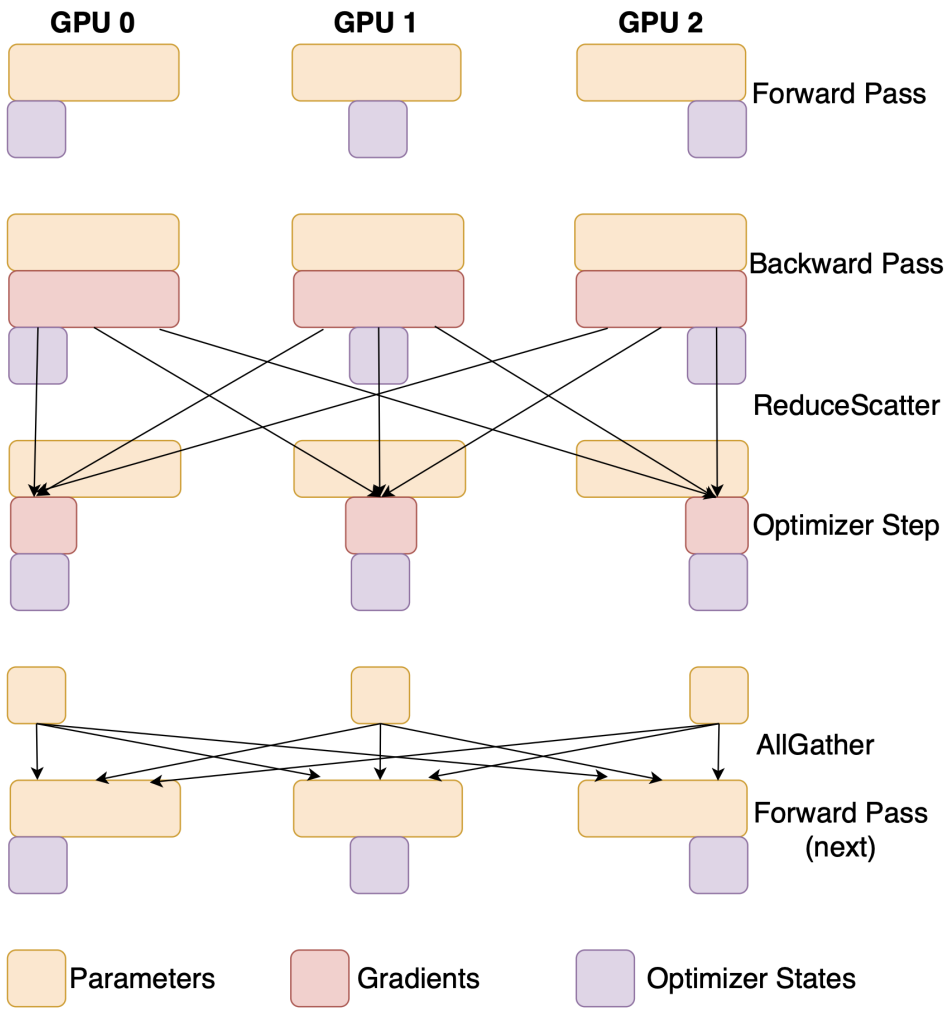
- After optimizer step, every GPU holds a **different**, updated piece of the model
- GPUs perform an AllGather to get complete, unsharded set of parameters before the next forward pass



ZeRO-2

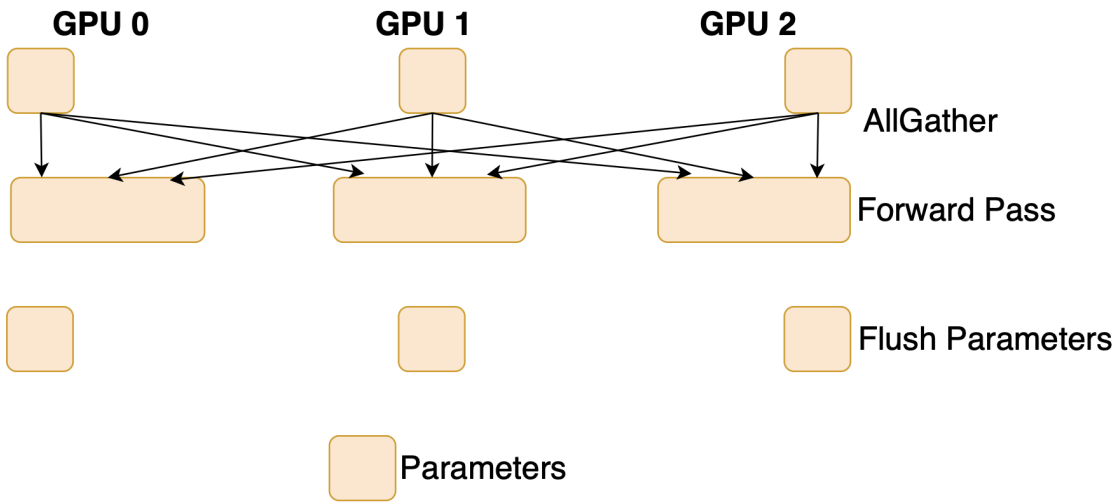
- Shards **gradients** in addition to optimizer states
- Full unsharded gradient tensor discarded after ReduceScatter
- ZeRO-2 achieves a **strictly larger memory saving** than ZeRO-1 at essentially no extra communication cost

ZeRO-2


GPU 0
GPU 1
GPU 2
Forward Pass
Backward Pass
ReduceScatter
Optimizer Step
AllGather
Forward Pass (next)
Parameters
Gradients
Optimizer States

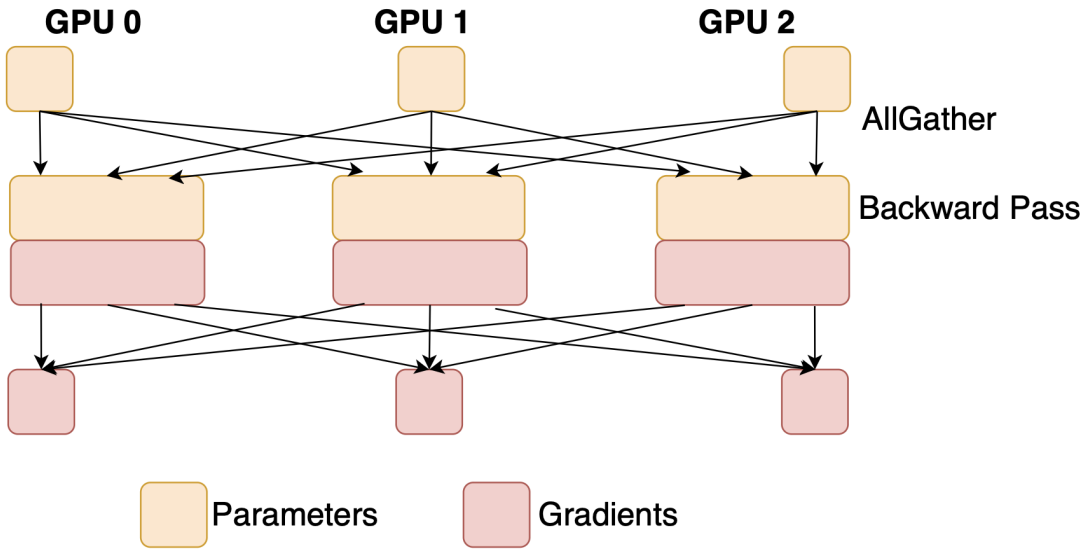
ZeRO-3

- Shards **parameters** too
- Performs an AllGather to **reconstruct the full layer** immediately before it is used in the forward pass



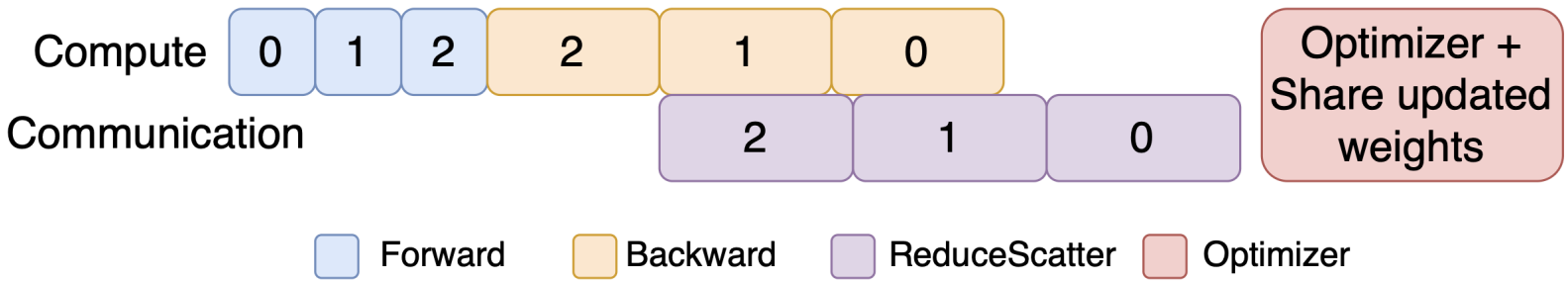
ZeRO-3

- Layer's parameters assembled again before backward pass using AllGather
- Significantly higher overhead of collective communication operations at every layer



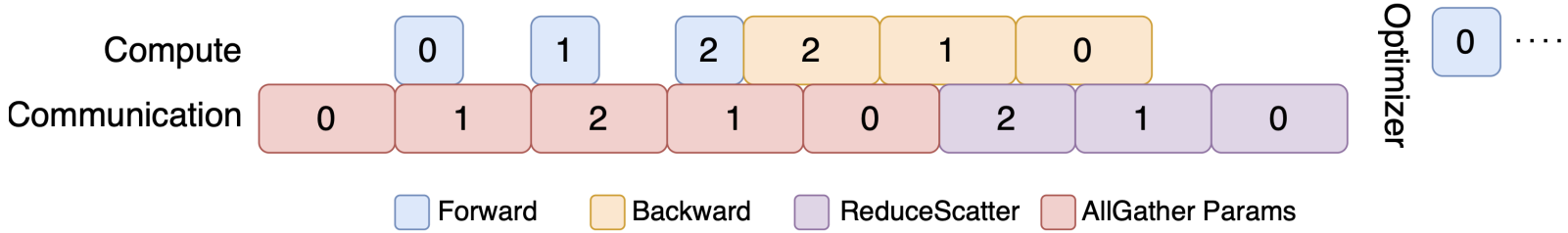
Overlapping Communication in ZeRO

- For ZeRO-1 and ZeRO-2, ReduceScatter can proceed **concurrently** with the computation of gradients for earlier layers



Overlapping Communication in ZeRO

- For ZeRO-3, AllGather before its forward and backward passes can be prefetched **one layer ahead of time**
- Avoid releasing a layer's reconstructed parameters immediately after the forward pass and retain till the backward pass



Pros and Cons of ZeRO

- Reduces memory footprint:
  - Makes large models feasible to train
  - Increases per-GPU batch size
- Increases GPU communication overhead significantly
- If a model and its gradients comfortably fit in memory and only the optimizer state is the limiting factor, ZeRO-2 is usually sufficient and preferable
- If number of parameters too large, go for ZeRO-3

PyTorch FSDP

- PyTorch's Fully Sharded Data Parallel implements ZeRO-3

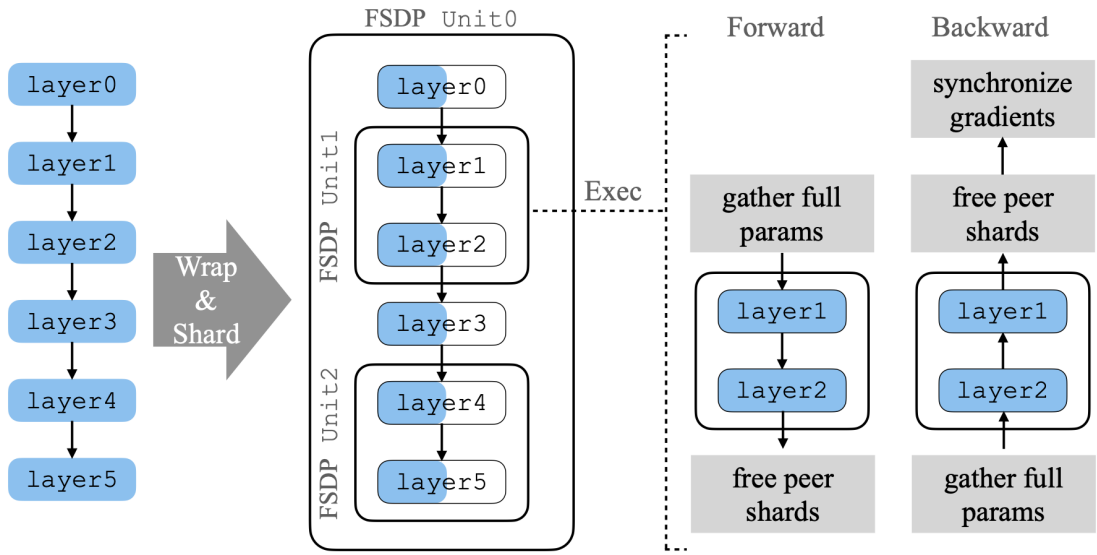
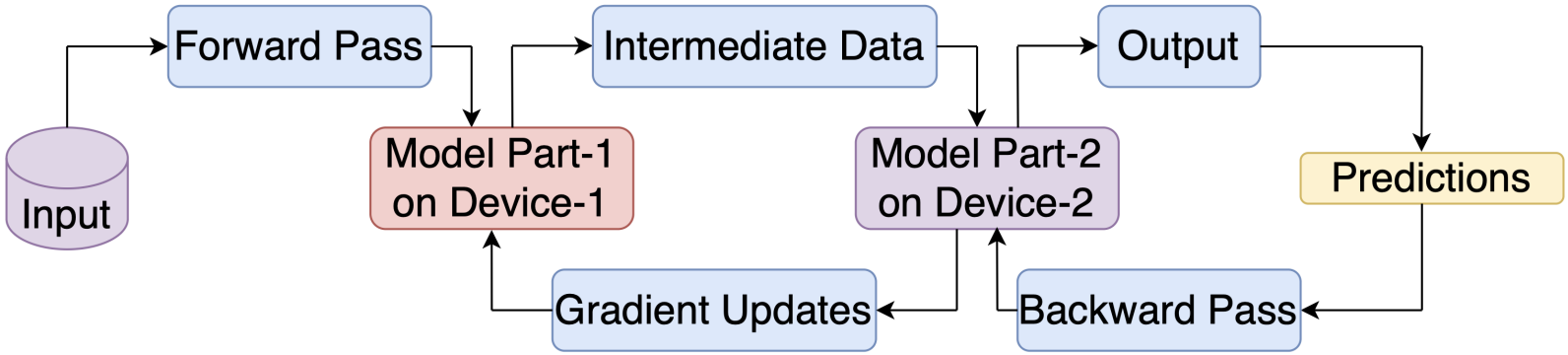


Image Credit: https://dl.acm.org/doi/10.14778/3611540.3611569

Pipeline Parallelism

Model Parallelism

- Large models **exceed the capacity** of a single GPU
- **Split** the model and let each worker hold only a part

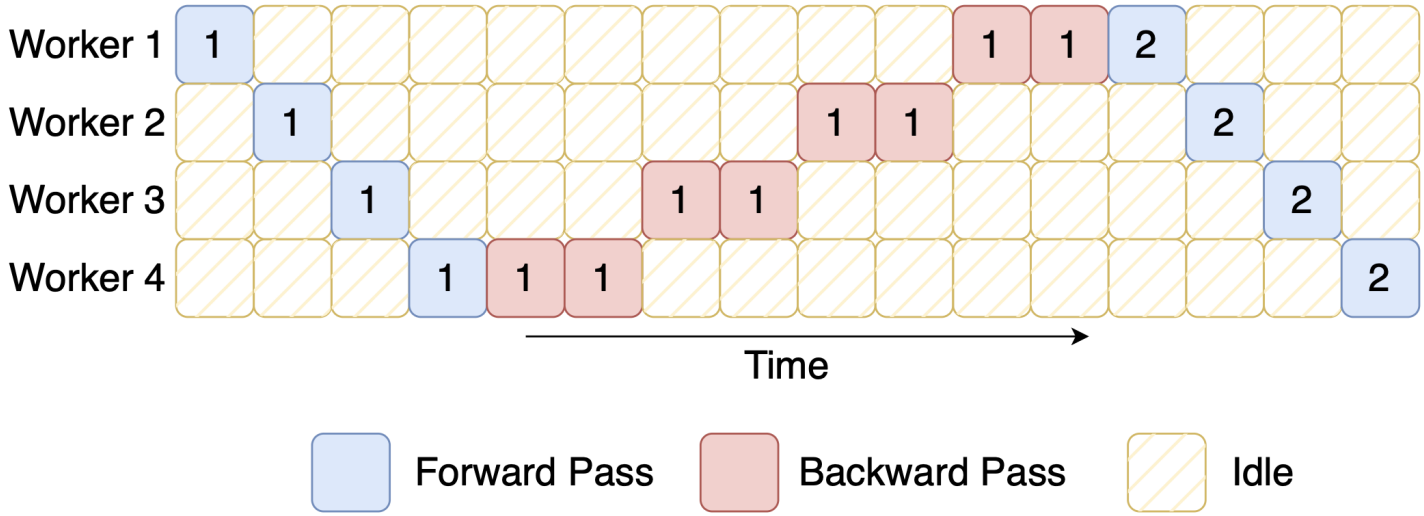


Simplest Form of Model Parallelism

- Partition the model into contiguous groups of layers called **stages**
- During forward pass, input travels through stages, forwarding intermediate **activations** to the next device
- During backward pass, **gradients** flow in reverse order
- Directly addresses the memory capacity problem

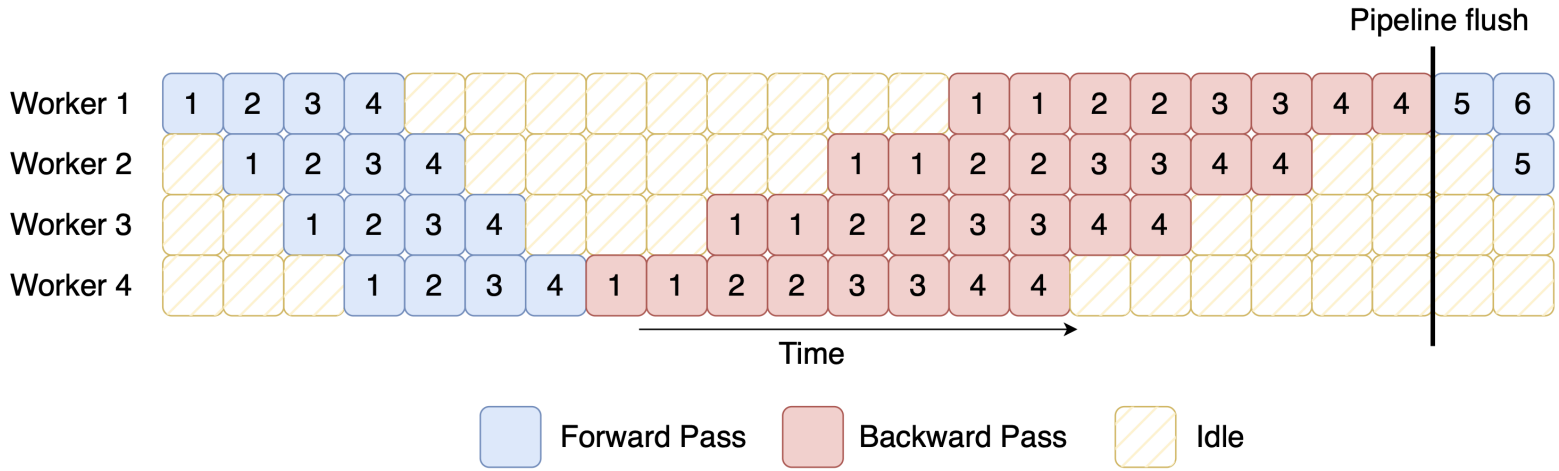
Limitations of Naive Model Parallelism

- At any instant **only one device** is actually doing useful work, others sitting idle
- Idle time referred to as **bubble**



Pipeline Parallelism: GPipe

- Split the mini-batch into smaller **micro-batches** and inject them into the pipeline one after another
- Devices simultaneously processing different micro-batches at different stages

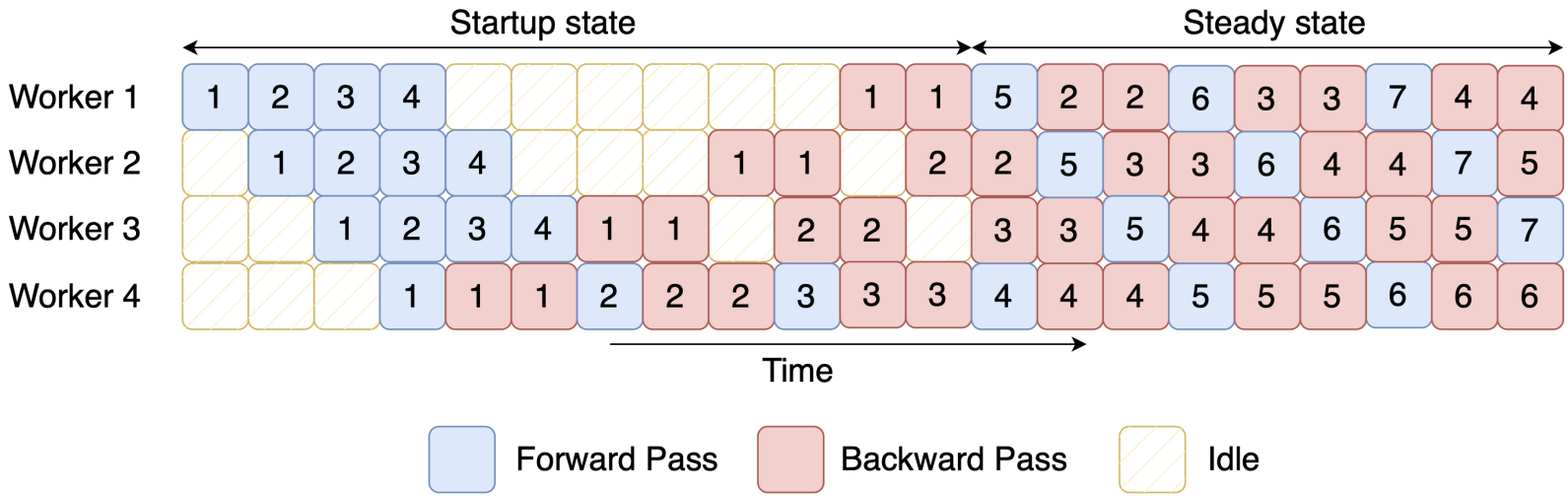


Pipeline Parallelism: GPipe

- **Pipeline flush** – all in-flight micro-batches drained from the pipeline and weights updated
- Pipeline with P stages, M micro-batches per mini-batch, time taken for forward pass F and backward pass B
  - Bubble size $= (P-1)(F+B)$
  - Ideal time $= M(F+B)$
  - **Fraction of time wasted** $= \frac{(P-1)(F+B)}{M(F+B)} = \frac{P-1}{M}$
- Increasing M reduces bubble but involves holding more activations in GPU memory

PipeDream

- Each worker begins the backward pass of an already processed micro-batch as soon as it is available
- **Updates weights asynchronously** and locally at each worker as soon as the relevant gradient is ready

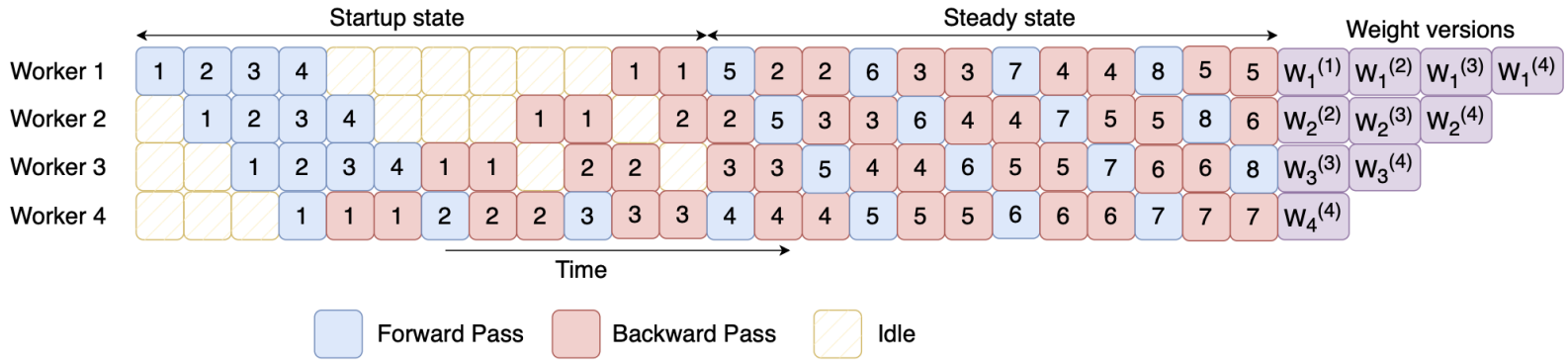


PipeDream

- Steady-state pattern alternates between performing one forward and one backward pass (**1F1B**)
- No global flush, so no large bubble
- Number of micro-batches in-flight needed is approx **number of workers**
- Lower memory footprint for in-flight activations
- 1F1B achieves full utilization only when each stage takes roughly the same amount of complete

Weight Stashing

- Asynchronous weight updates implies forward and backward passes see **different weights**
- **Weight stashing** – each worker keeps multiple versions of its own weights in a small buffer, one for every micro-batch that is currently in flight
  - Forward and backward passes use same weights
  - Version discarded if stale or if backward pass completed

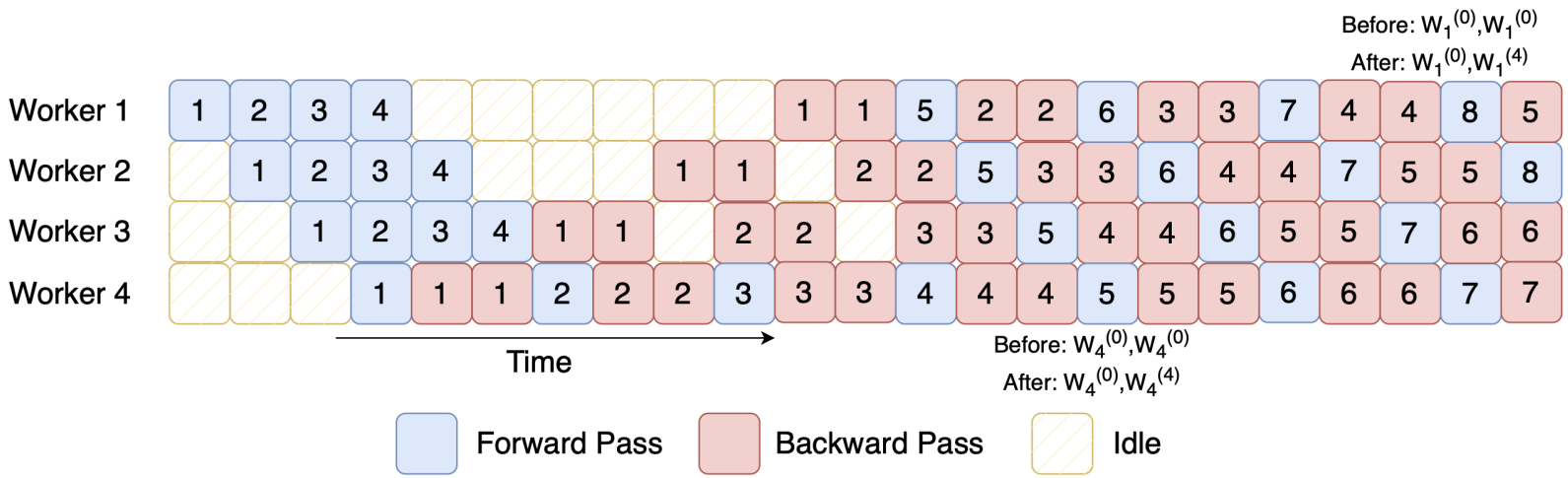


Vertical Sync

- Weight stashing does not guarantee **consistency across workers**
  - A micro-batch may end up seeing different weight versions at different stages
- In **vertical sync**, each micro-batch tagged with a **version identifier**, used by the first batch and propagated through the pipeline
  - Each micro-batch sees a coherent view of the model

PipeDream-2BW

- Each worker maintains only **two weight versions**, one stable and one running
- All micro-batches in a group perform forward and backward passes using the same stable version, after which an accumulated update is applied

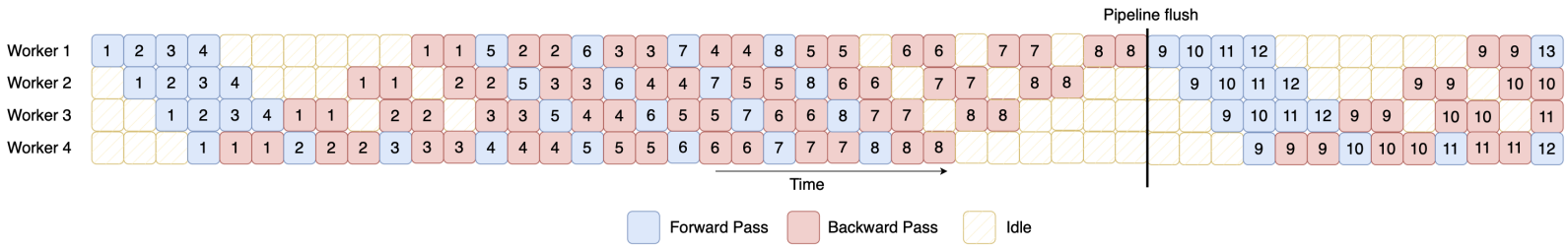


Correctness of Gradient Descent

- Ordinary update, $w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t)}, w_2^{(t)}, \ldots, w_n^{(t)}\right)$
- With weight stashing, $w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-n+1)}, w_2^{(t-n+2)}, \ldots, w_n^{(t)}\right)$
- With vertical sync, $w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-n+1)}, w_2^{(t-n+1)}, \ldots, w_n^{(t-n+1)}\right)$
- With PipeDream-2BW, $w^{(t+1)} = w^{(t)} - \nu \cdot \nabla f\left(w_1^{(t-s)}, w_2^{(t-s)}, \ldots, w_n^{(t-s)}\right)$

PipeDream-Flush

- Waits until the end of a mini-batch, **flushes** the pipeline, and only then performs a single synchronous weight update
- Number of in-flight micro-batches bounded by depth of the pipeline due to 1F1B
- Correctness of gradient descent + lower memory footprint



Pipeline Parallelism with Interleaved Stages

- Cut the model into a larger number of smaller chunks and give each device several **non-contiguous** chunks
- Batch traverses pipeline multiple times, lower bubble size

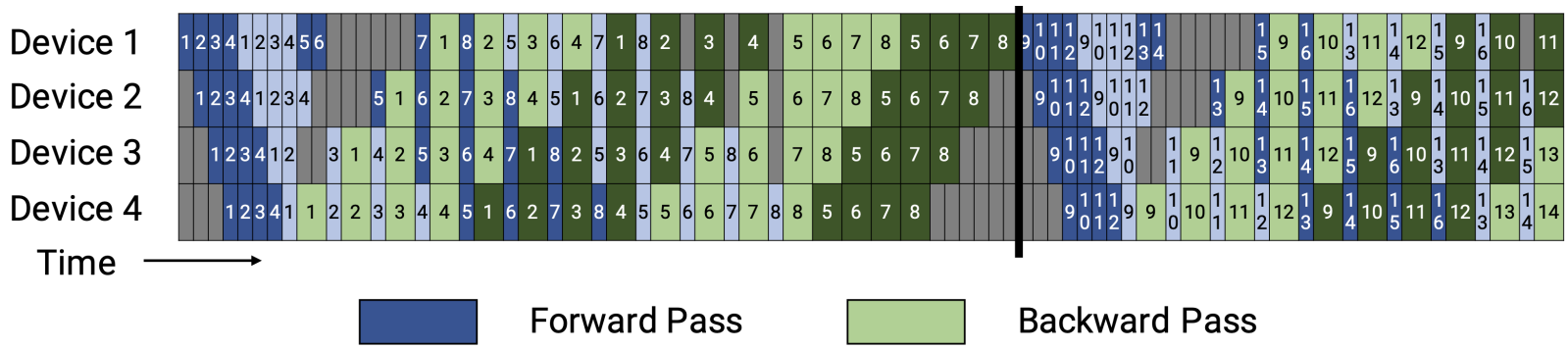


Image Credit: https://dl.acm.org/doi/10.1145/3458817.3476209

Zero Bubble Pipeline Parallelism

- For layer $z = Wx$, backward pass produces:
  - $\nabla_x L = W^\top \nabla_z L$ (input gradient B)
  - $\nabla_W L = \nabla_z L \, x^\top$ (weight gradient W)
- Stage B on critical path, W has no downstream dependency

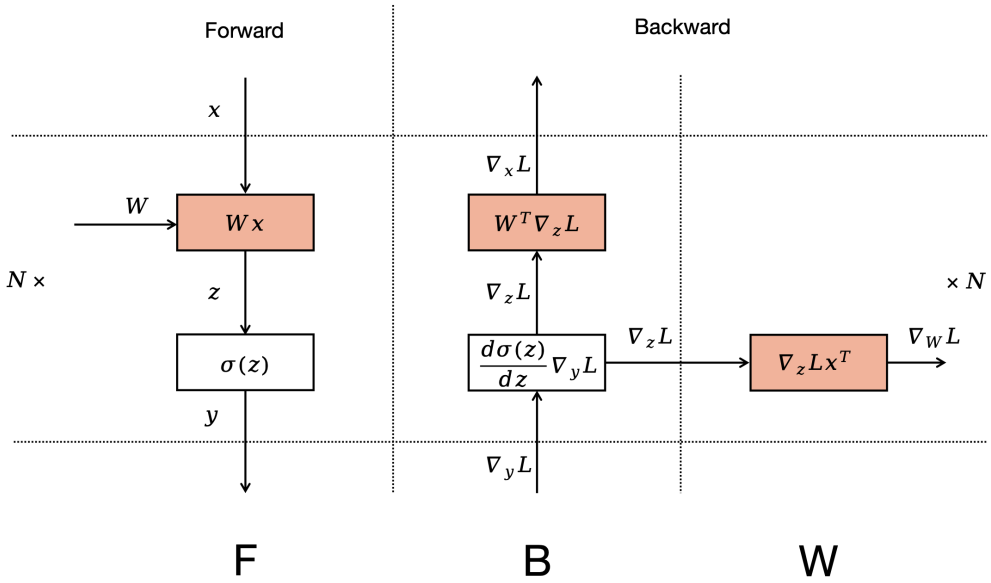


Image Credit: https://arxiv.org/abs/2401.10241

Zero Bubble Pipeline Parallelism

- Separate B and W, find a schedule that interleaves F, B, and W

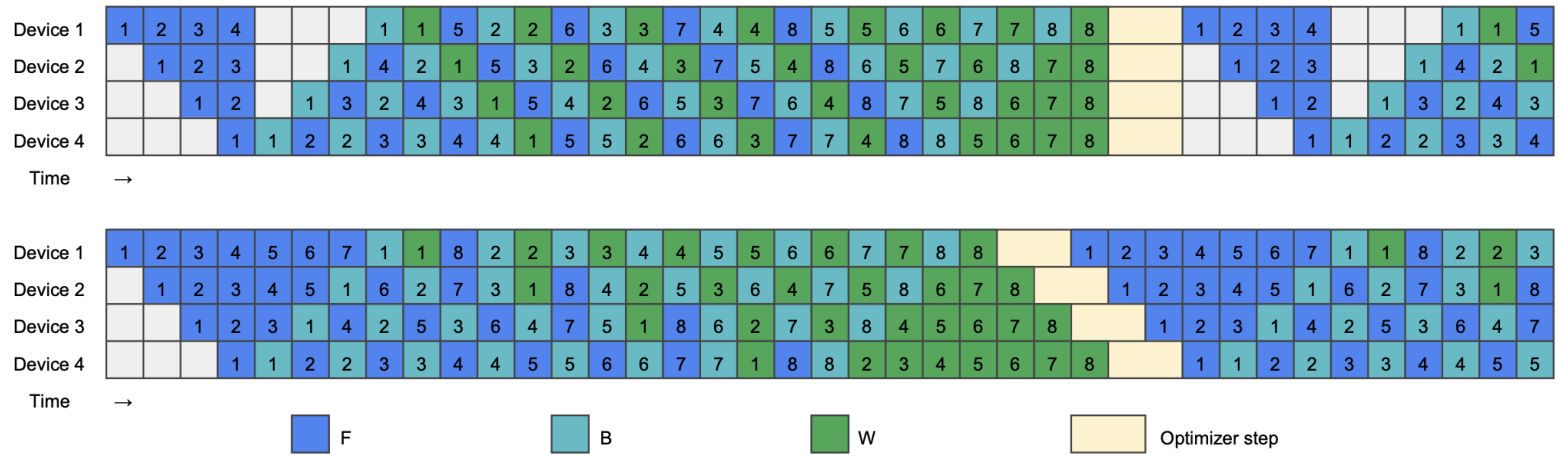


Image Credit: https://arxiv.org/abs/2401.10241

DualPipe: Bidirectional Pipelining

- Feed micro-batches into the pipeline from **both ends** simultaneously
- Also uses the idea of splitting backward pass into two parts, overlaps compute and communication

Image Credit: `https://arxiv.org/abs/2412.19437`

Tensor Parallelism

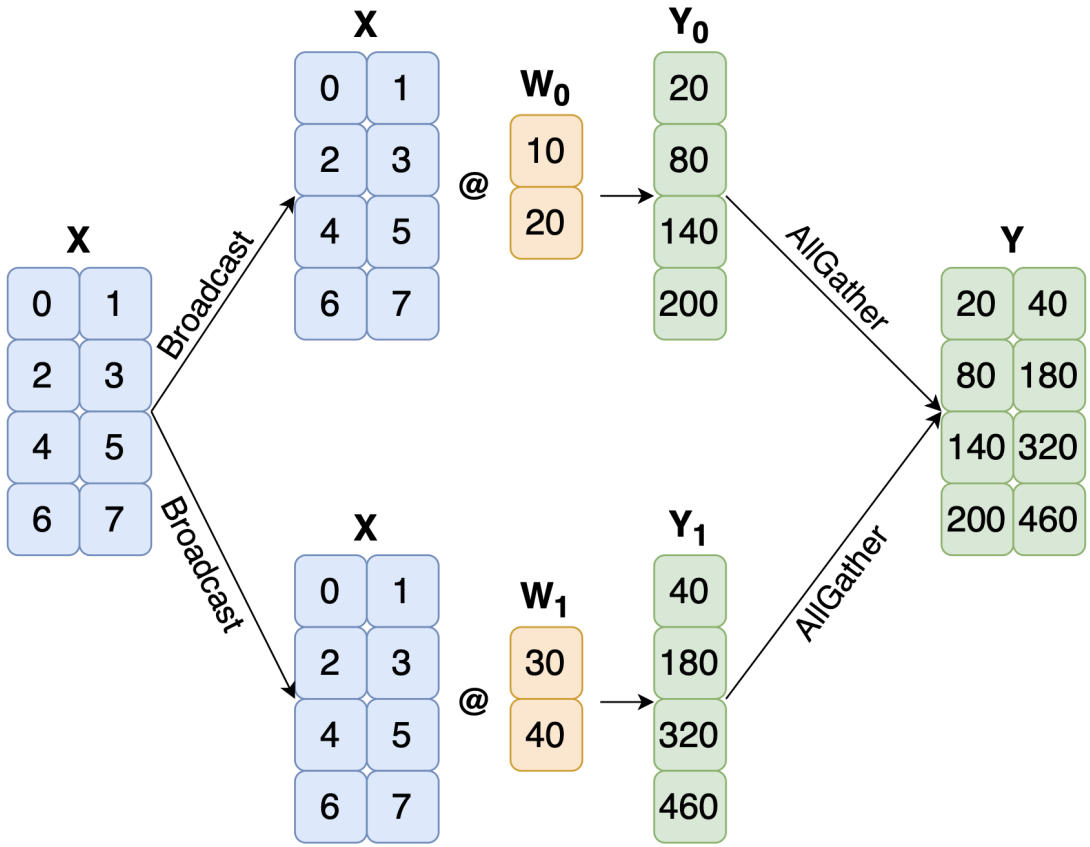
Tensor Parallelism

- Pipeline parallelism does not help if even a single layer too large to fit on one GPU
- Tensor parallelism partitions a model **horizontally**
  - Splits the weights and activations across several devices
- Devices cooperate through communication to jointly compute the output of that layer

Column-Wise Partitioning

- If we partition W **column-wise** into blocks $W = [\, W_1 \; W_2 \; \cdots \; W_n \,]$, then $Y = XW = X[\, W_1 \; W_2 \; \cdots \; W_n \,] = [\, XW_1 \; XW_2 \; \cdots \; XW_n \,]$
- Give one slice to each device
- Input activation must first be Broadcast to every participating device
- Exchange local results with AllGather

Column-Wise Partitioning



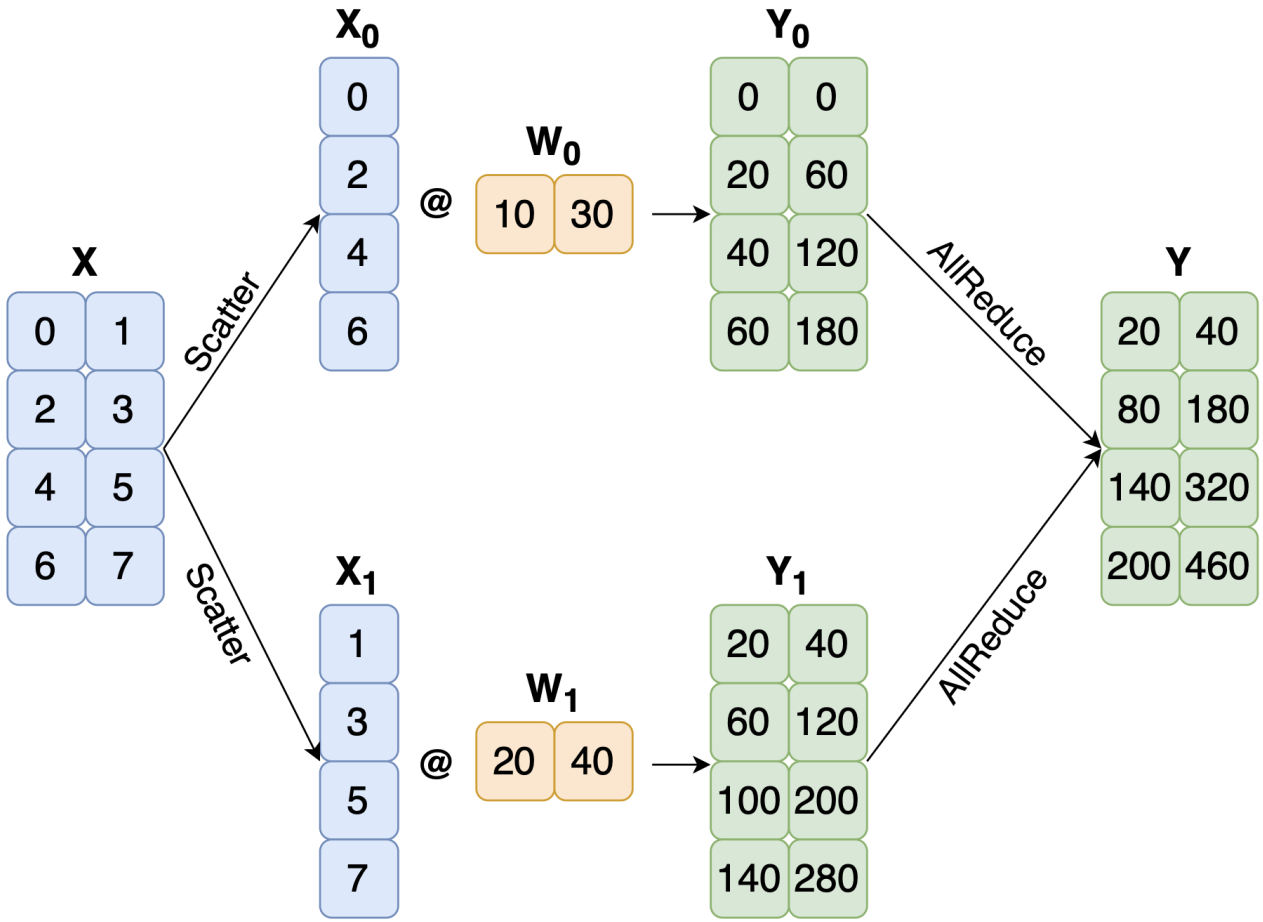
Row-Wise Tensor Partitioning

- If we partition W row-wise, and correspondingly partition X column-wise, writing $X = [\, X_1 \; X_2 \; \cdots \; X_n \,]$ and $W = [\, W_1 \; W_2 \; \cdots \; W_n \,]^\top$, then

$$Y = XW = [\, X_1 \; X_2 \; \cdots \; X_n \,] \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix} = \sum_{i=1}^{n} X_i W_i$$

- Scatter operation distributes the appropriate column slice of X to each device
- Exchange local results with AllReduce

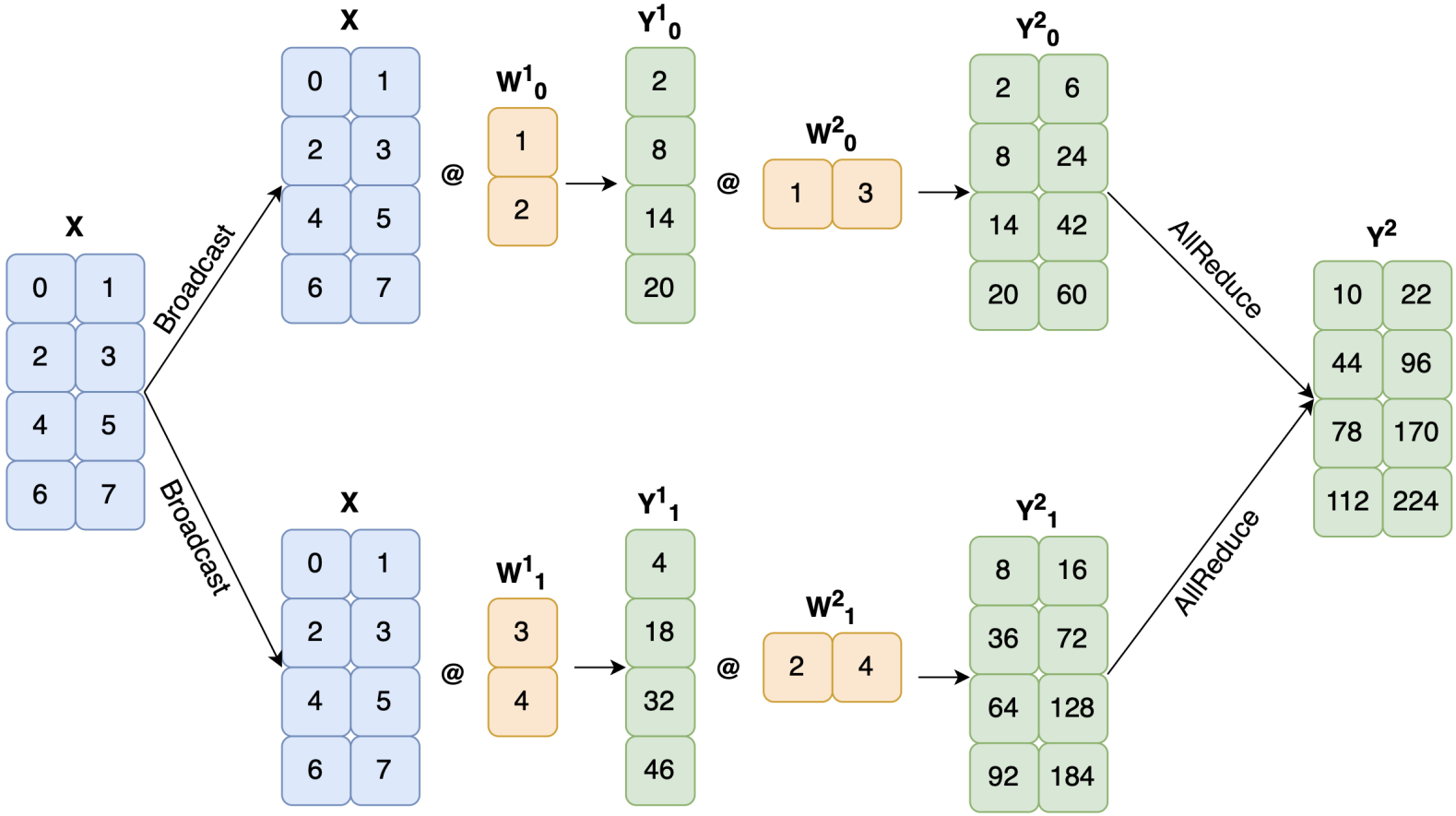
Row-Wise Tensor Partitioning



Tensor Parallelism with MLP

- Consider $Y^2 = f(XW^1)W^2$, f is an activation function
- First layer's weights are split column-wise, and the second layer's weights are split row-wise
- f can be applied **independently** to each column slice of the first layer's output without any communication at all
- Each device only needs to communicate once, at the end

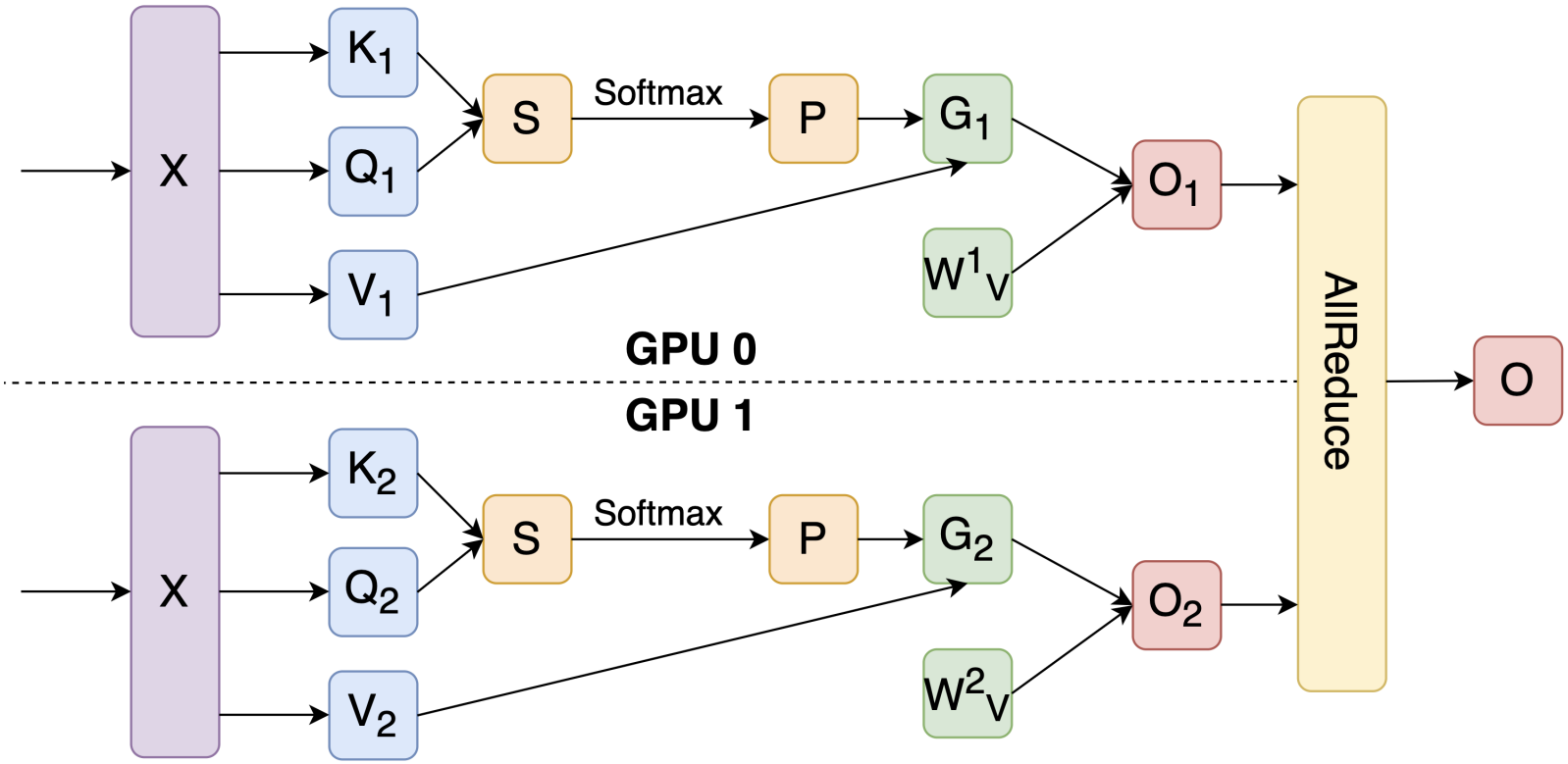
Tensor Parallelism with MLP



Tensor Parallelism in Multi-Head Self-Attention

- Query, key, and value projection matrices are split **column-wise**, with each device receiving the columns corresponding to one or more complete attention heads
- Every device carries out the self-attention computation for its own heads
- Output projection matrix is split **row-wise**
- AllReduce performed at the end of the attention block

Tensor Parallelism in Multi-Head Self-Attention



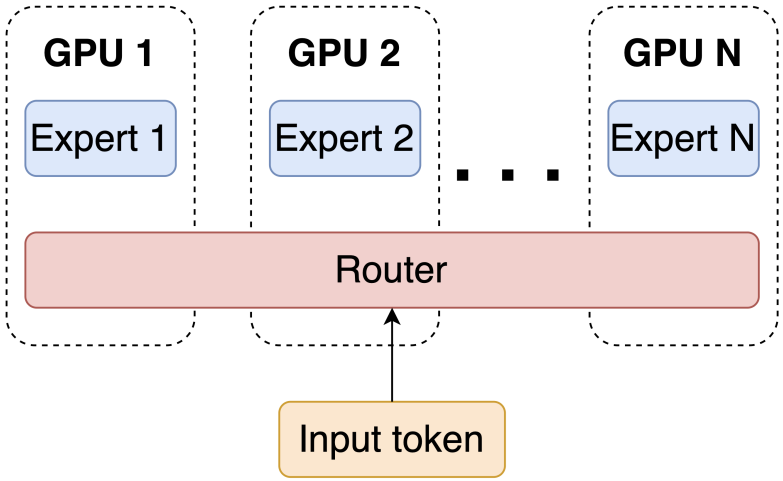
Communication Overhead

- Communication **more frequent** and larger in volume
- AllReduce sits directly on the **critical path**
  - Communication difficult to hide behind computation
- Tensor parallelism is often applied to a confined group of GPUs connected by NVLink
- Why used at all?
  - Shrinks the **memory footprint** of an individual layer

Other Types of Parallelism

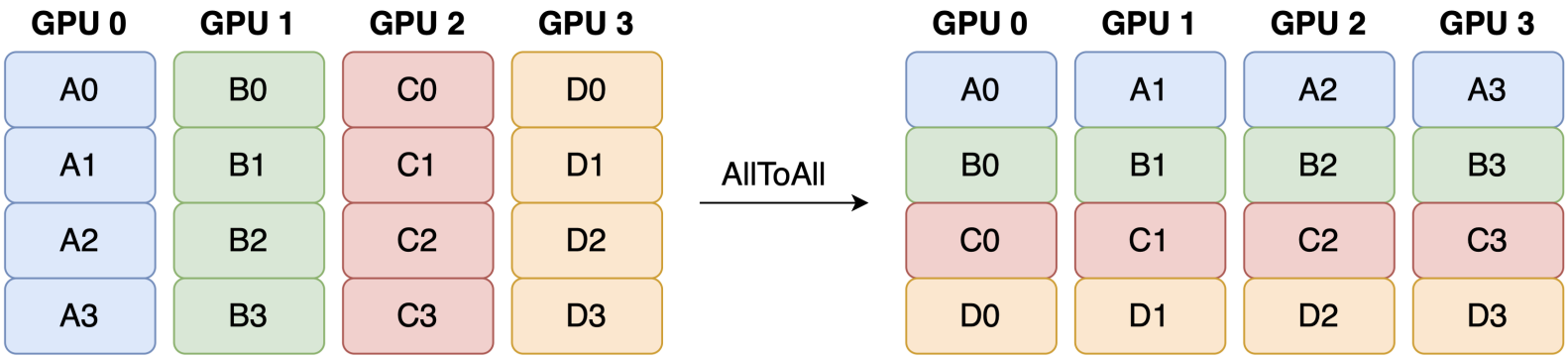
Expert Parallelism

- In an MoE model, the single large MLP replaced by a collection of many smaller MLPs, called **experts**
- Different experts placed on **different GPUs**
- **Router** determines which GPU a given token's computation should be sent to



Expert Parallelism

- Tokens assigned by the router must be delivered to the GPUs that host the **selected experts**
- AllToAll – every GPU simultaneously exchanges distinct data with every other GPU
- **Load balancing** is a challenge



Context Parallelism

- A long sequence is split into chunks, and each chunk placed on a different GPU
- Useful for models with **large context windows**
- For MLP, operations on each token independent
- For self-attention, access to keys and values of every token in the sequence needed

Block-Wise Attention

- Query block $Q_i \in \mathbb{R}^{B \times d}$ fixed on some device
- Full key/value sequence be split into blocks $K_1, \ldots, K_M$ and $V_1, \ldots, V_M$, each of size $B \times d$
- Attention is incrementally computed as key and value blocks arrive
- Online softmax rescaling much like in FlashAttention

Block-Wise Attention

- For each incoming block j, $S_{ij} = \frac{Q_i K_j^\top}{\sqrt{d}} \in \mathbb{R}^{B \times B}$, $m_{ij} = \text{rowmax}(S_{ij})$, $P_{ij} = \exp(S_{ij} - m_{ij})$
- $m_i^{(j+1)} = \max\left(m_i^{(j)}, m_{i,j+1}\right)$,
  $\ell_i^{(j+1)} = e^{m_i^{(j)} - m_i^{(j+1)}} \ell_i^{(j)} + e^{m_{i,j+1} - m_i^{(j+1)}} \sum_k P_{i,j+1}[:, k]$,
  $O_i^{(j+1)} = e^{m_i^{(j)} - m_i^{(j+1)}} O_i^{(j)} + e^{m_{i,j+1} - m_i^{(j+1)}} P_{i,j+1} V_{j+1}$
- After the final block M, the correctly normalized output is recovered as $O_i^{(M)} / \ell_i^{(M)}$

Ring Attention

- One possible implementation of block-wise attention
- GPUs arranged in a **logical ring**; every GPU sends its local chunk of key and value vectors to its neighbour, accumulates partial attention output
- After one full loop around the ring, every node has seen every K, V block once

Ring Attention

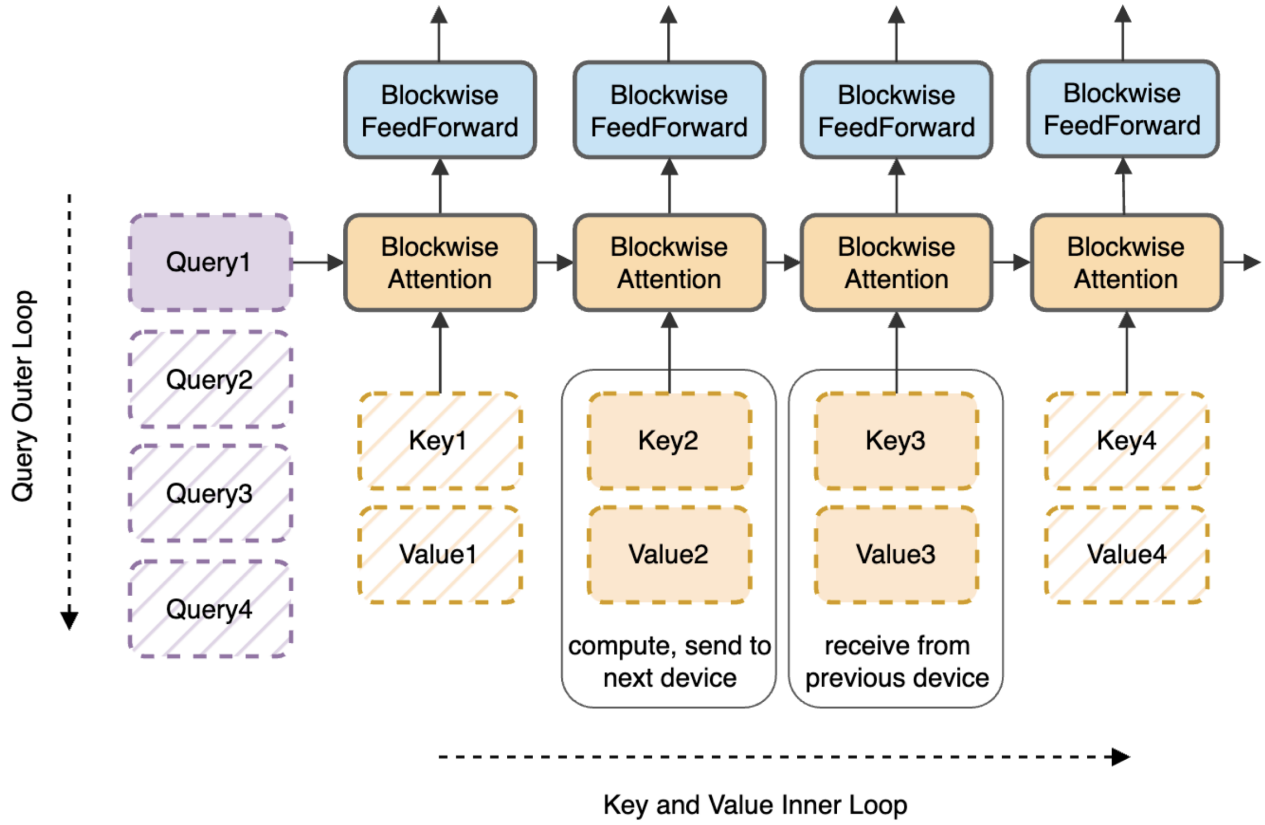
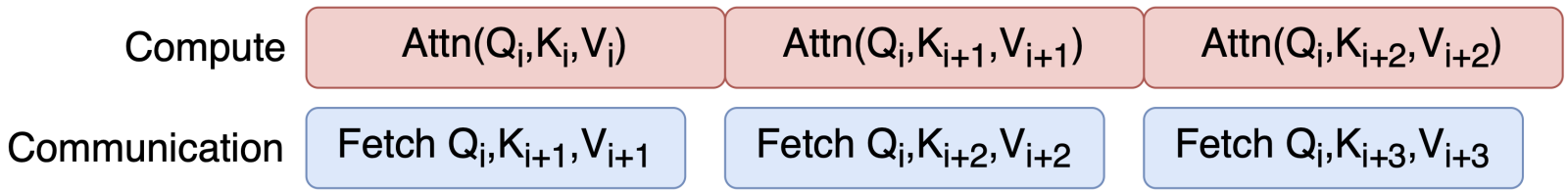


Image Credit: https://arxiv.org/abs/2512.20968v1

Overlapping Compute and Communication

- Compute attention against one chunk while simultaneously sending and receiving the next chunk

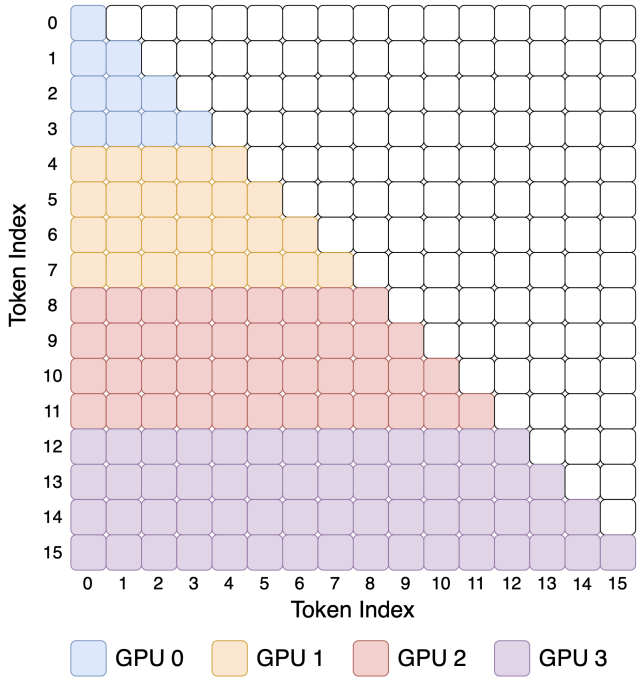


Overlapping Compute and Communication

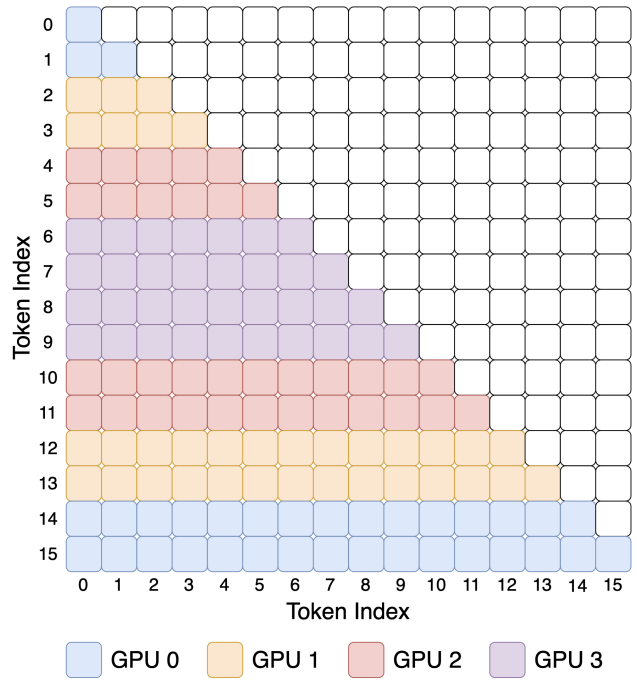
- For a block of size B, the compute cost per block is $\underbrace{2B^2d}_{QK^\top} + \underbrace{2B^2d}_{(\text{attn})\cdot V} = 4B^2d$ FLOPs
- The communication cost of each ring step is sending one K block and one V block to the next device $Bd + Bd = 2Bd$, total bidirectional traffic is $4Bd$
- For communication to be fully hidden behind computation, $\frac{4B^2d}{\text{FLOP/s}} \gtrsim \frac{4Bd}{\text{bandwidth}} \iff B \gtrsim \frac{\text{FLOP/s}}{\text{bandwidth}}$

Load Balancing Under a Causal Mask

- Change the assignment of token blocks to devices to balance work



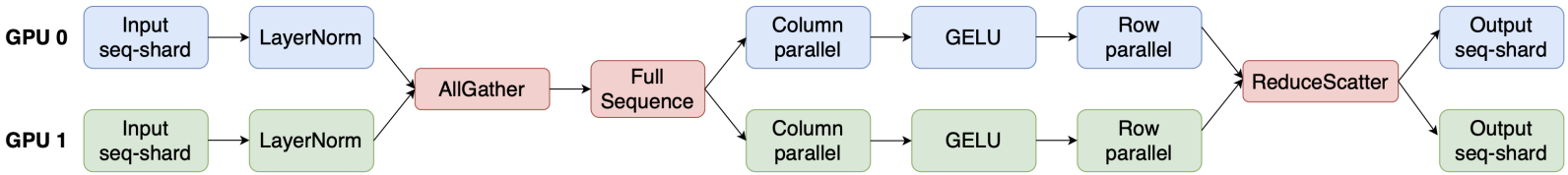
(a) Unbalanced load



(b) Balanced load

Sequence Parallelism

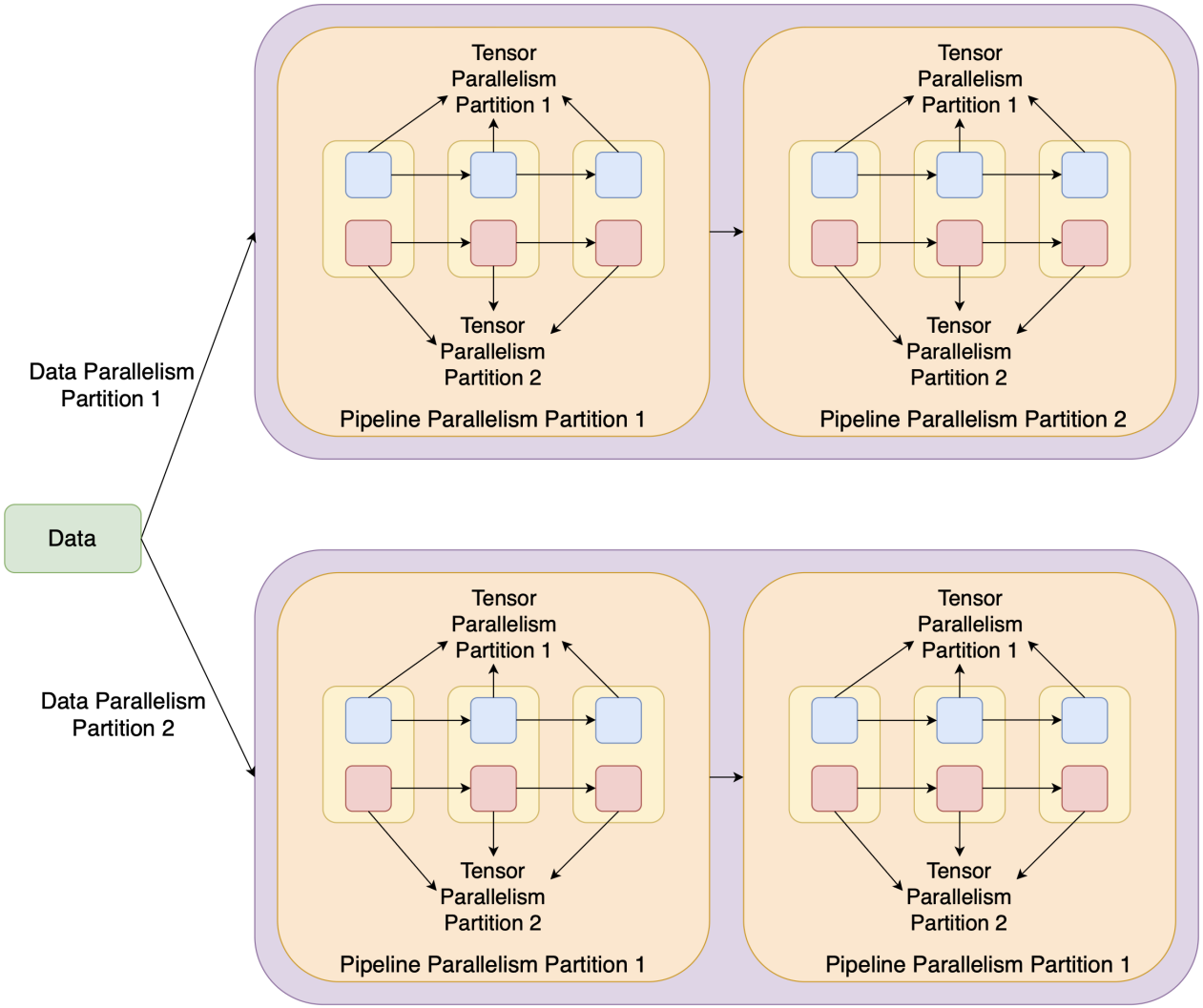
- Standard TP shards across hidden dimension, but LayerNorm/Dropout need per-token stats (μ, σ^2) across full hidden vector h
- **Sequence parallelism** shards such operations along sequence dimension instead
- When composed with TP, needs sharding across hidden and sequence dimensions alternately



Hybrid Parallelism

- Data, pipeline, and tensor parallelism are combined into **hybrid parallelism**
- Tensor parallelism applied inside a GPU node
- Pipeline parallelism across GPU nodes, because of lower communication needs
- Data parallelism sits at the outermost level
- ZeRO is often layered on top of this hybrid setup

Hybrid Parallelism



Case Studies

DeepSeek-V3

- Trained on a cluster of **2048 H800 GPUs**
- Avoided tensor parallelism entirely
- The model uses **64-way expert parallelism**, with the experts spread across eight nodes
- Uses 16-way PP and ZeRO-1 style DP
- Uses DualPipe as a pipeline scheduling algorithm
- Overlap achieved with **warp specialization**
  - Some warps are dedicated to computation while others to communication

DeepSeek-V3

- Makes extensive use of mixed precision training with FP8 arithmetic to reduce both compute and memory footprint

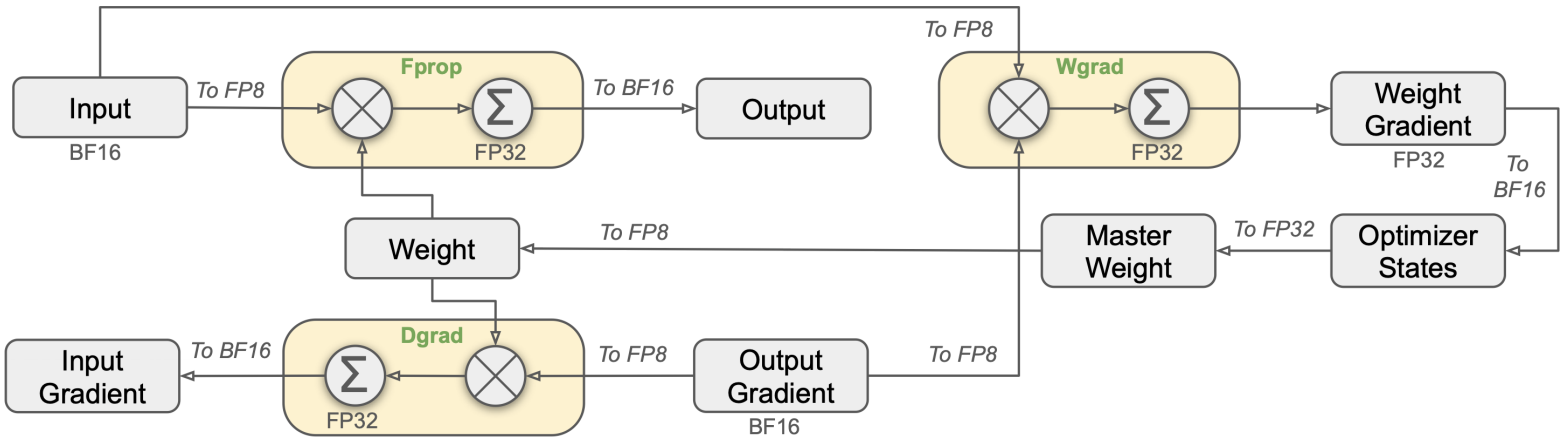


Image Credit: `https://arxiv.org/abs/2412.19437`

Llama 3

- Trained on a larger cluster of **16,384 H100 GPUs**

	8B	**70B**	**405B**
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE ($\theta = 500{,}000$)		

Table: Model parameters for the Llama 3 family of models

Llama 3

- First ran a large sweep of much smaller training runs to empirically determine how loss depends on model size and training data
- For a range of fixed compute budgets, trained several models of different sizes, each consuming that same budget but trading off model size against number of training tokens

Llama 3

- Plotting validation loss against the number of training tokens for each compute budget produces a family of **U-shaped curves**

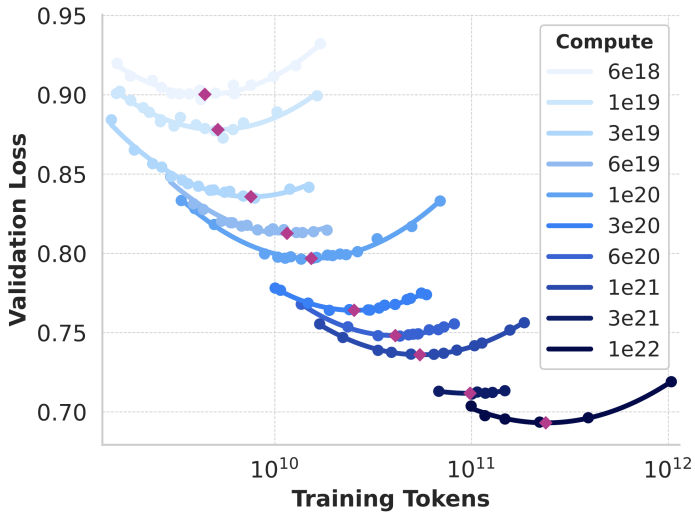


Figure: IsoFLOPs curves: validation loss vs. training tokens for a range of fixed compute budgets

Image Credit: `https://arxiv.org/abs/2407.21783`

Llama 3

- Compute-optimal training tokens scale as a **power law** in the compute budget
- Fitted power law predicts the compute-optimal number of training tokens and model size

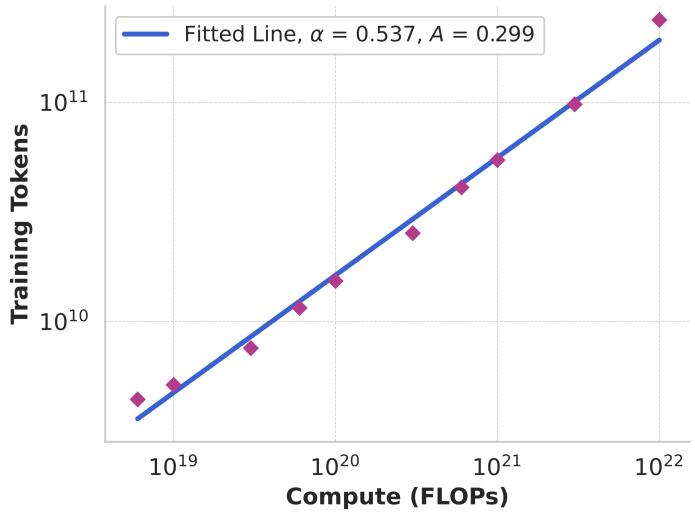


Figure: Compute-optimal training tokens vs. compute budget

Image Credit: `https://arxiv.org/abs/2407.21783`

Llama 3

- **4D parallelism** – tensor, context, pipeline, and (fully sharded) data parallelism
- TP is applied **8-way** within a node
- PP is applied **16-way**, using interleaved pipeline schedule
- Context parallelism is applied **16-way**, and is introduced specifically to handle Llama 3's long-context training
- Uses grouped-query attention, in which several query heads share a single key/value head
- DP is realized through Fully Sharded Data Parallelism **(FSDP)**