Data Parallelism in LLM Training

Autumn 2025

Lecturer: Yuedong (Steven) Xu

Shenzhen Loop Area Institute

yuedongxu@slai.edu.cn

Fudan University

ydxu@fudan.edu.cn

Disclaimer

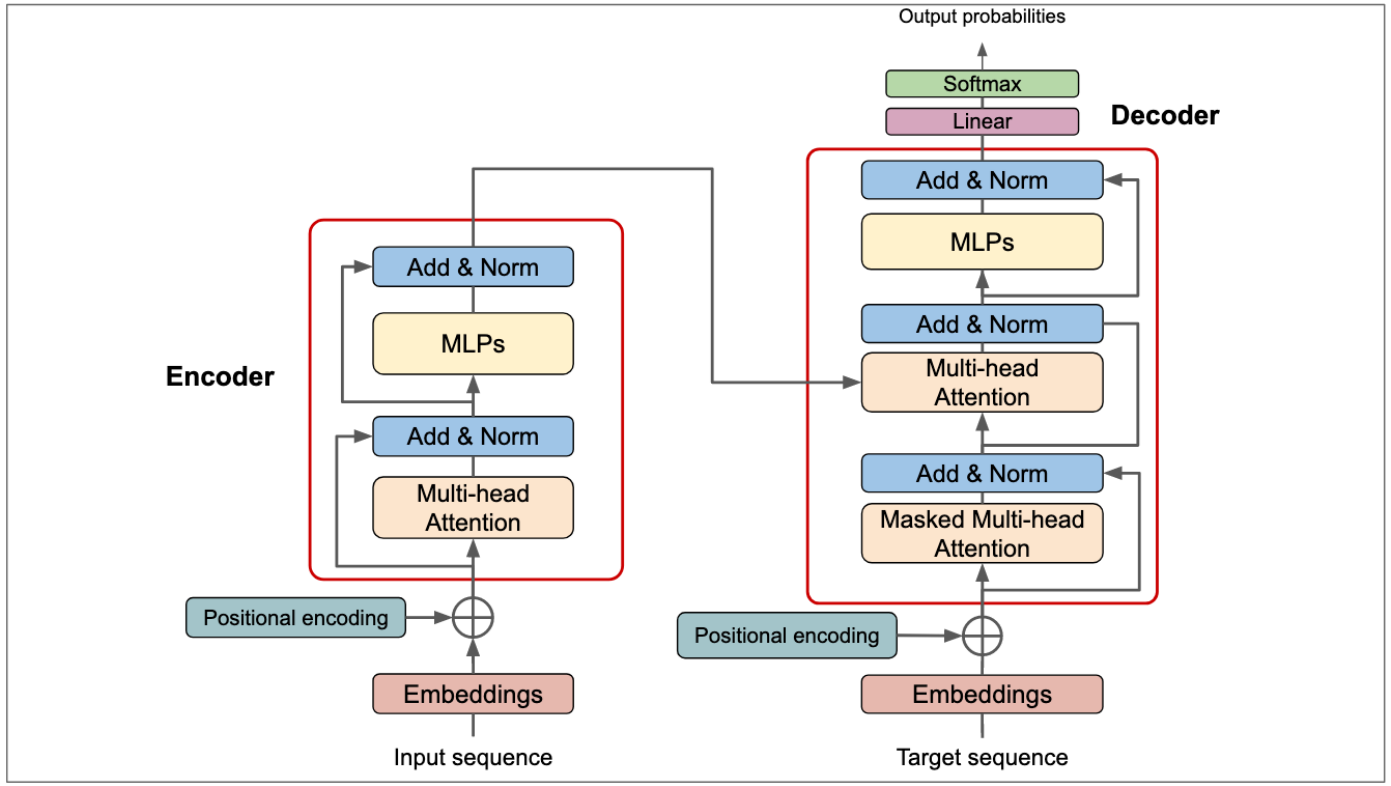
Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blogposts, research talks, tutorial videos, and other materials shared by the research community. We sincerely appreciate their efforts and assistance, and try our best to cite the sources of the materials used in this course.

Distributed LLM Training: Outline

- **Transformer: A Quick Overview**
- Data Parallelism
  - Parameter-Server
  - All-Reduce
  - Memory Optimization
- Model Parallelism
  - Pipeline Parallelism
  - Tensor Parallelism
  - Sequence Parallelism
- Mixture of Experts

Transformer Overview

- Transformer architecture



A minimalist description of Transformer *

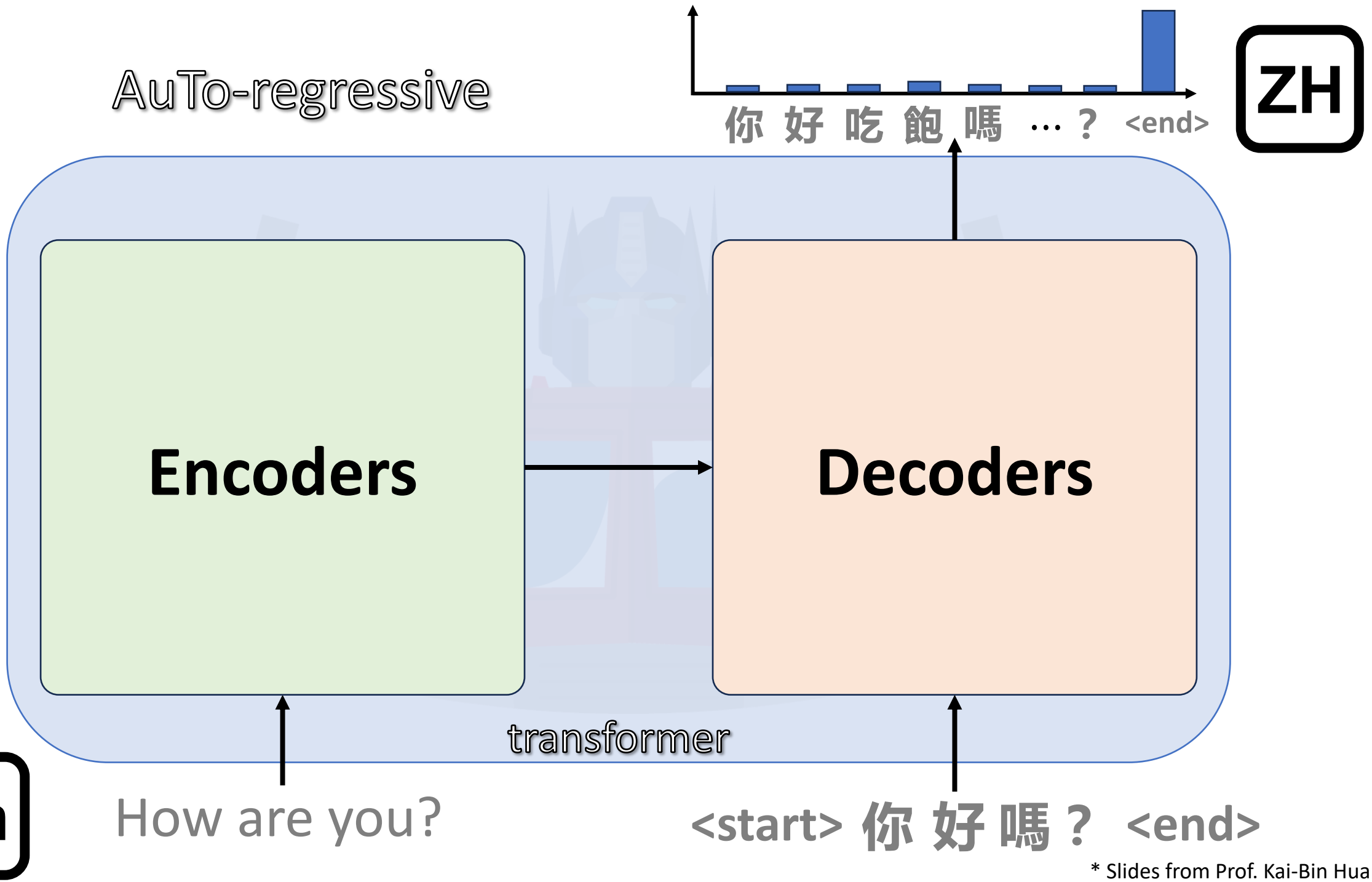
What to include

- Execution dependency graph
- Coarse-grained computation
- Fine-grained communication

What ***NOT*** to include

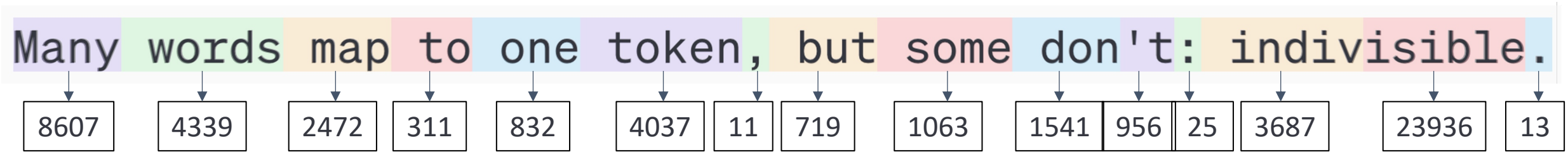
- Model architecture
- Downstream applications
- Exhausting system optimization techniques

* https://re-cinq.com/blog/llm-architectures

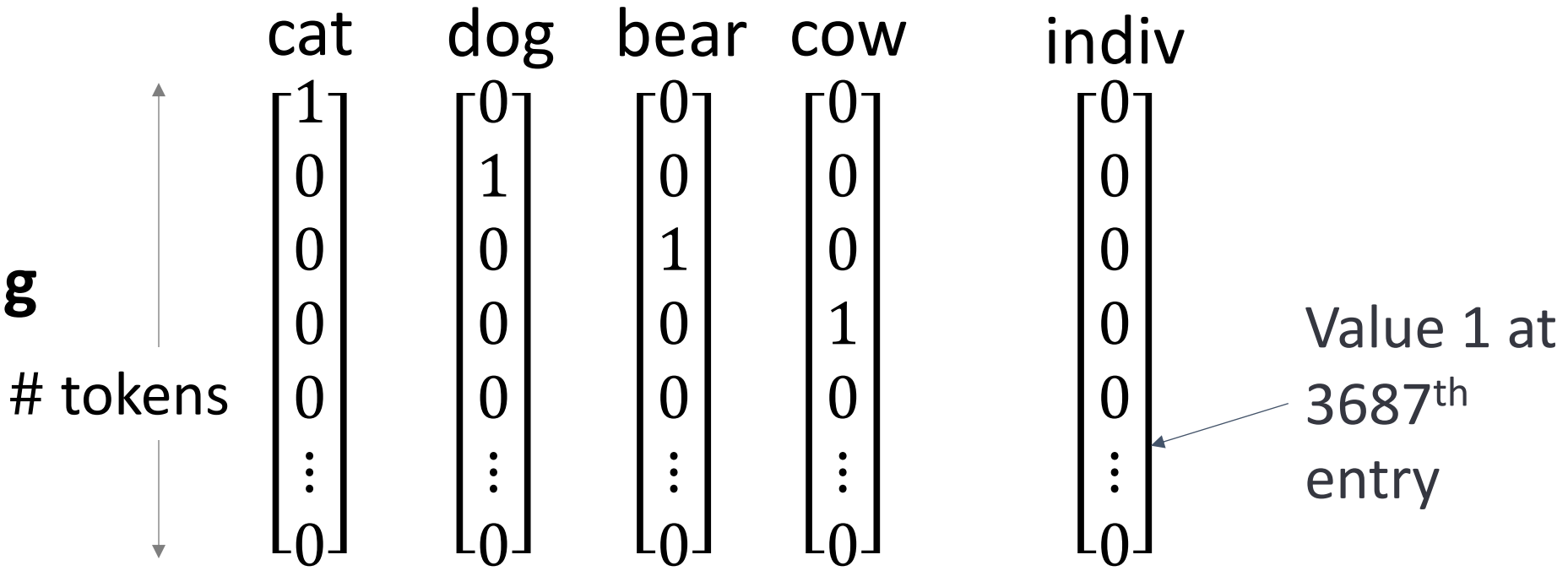


* Slides from Prof. Kai-Bin Huang

Tokenization

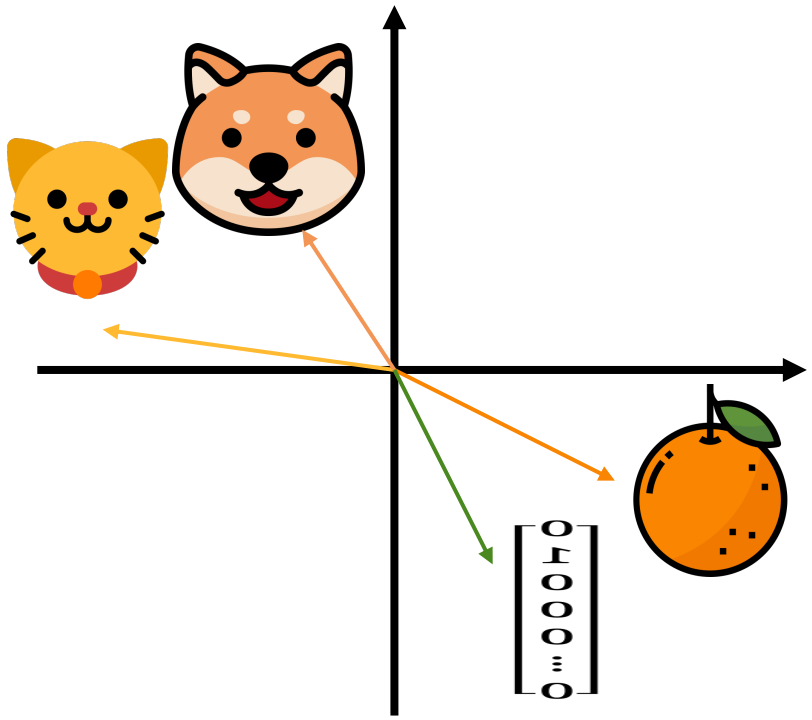


One-hot encoding



* Slides from Prof. Kai-Bin Huang

TOKEN EMBEDDING



Embedding Space

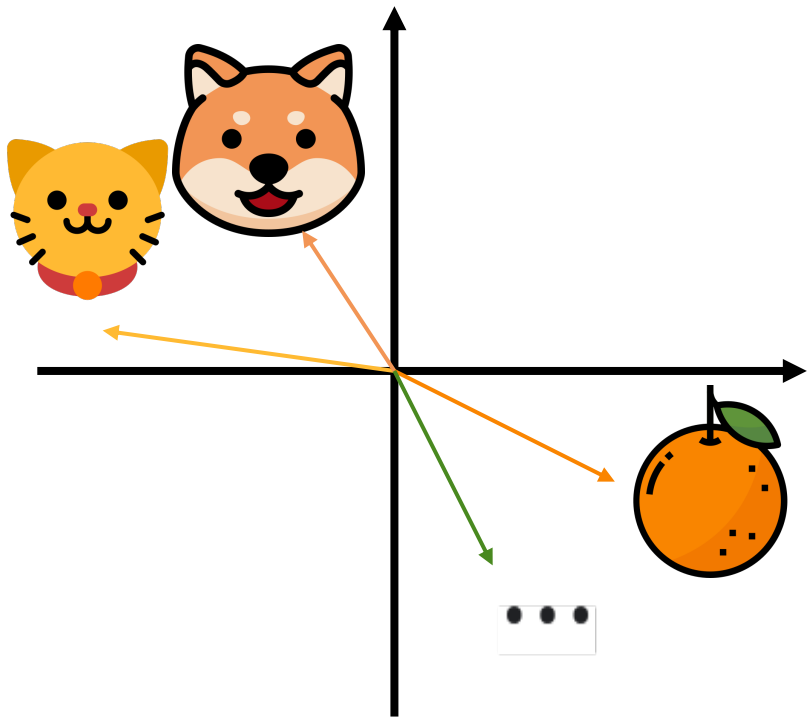
$$d\begin{bmatrix}0.5\\2.7\\1.2\\\vdots\\0.2\end{bmatrix} = d\begin{bmatrix} \overset{\text{Vocabulary size}}{\overset{\longleftarrow \text{\# tokens} \longrightarrow}{\boldsymbol{W_E}}} \end{bmatrix} \overset{\text{dog}}{\begin{bmatrix}0\\1\\0\\0\\0\\\vdots\\0\end{bmatrix}}$$

Embedded token

Embedding Matrix

* Slides from Prof. Kai-Bin Huang

TOKEN EMBEDDING



Embedding Space

Matrix Multiplication

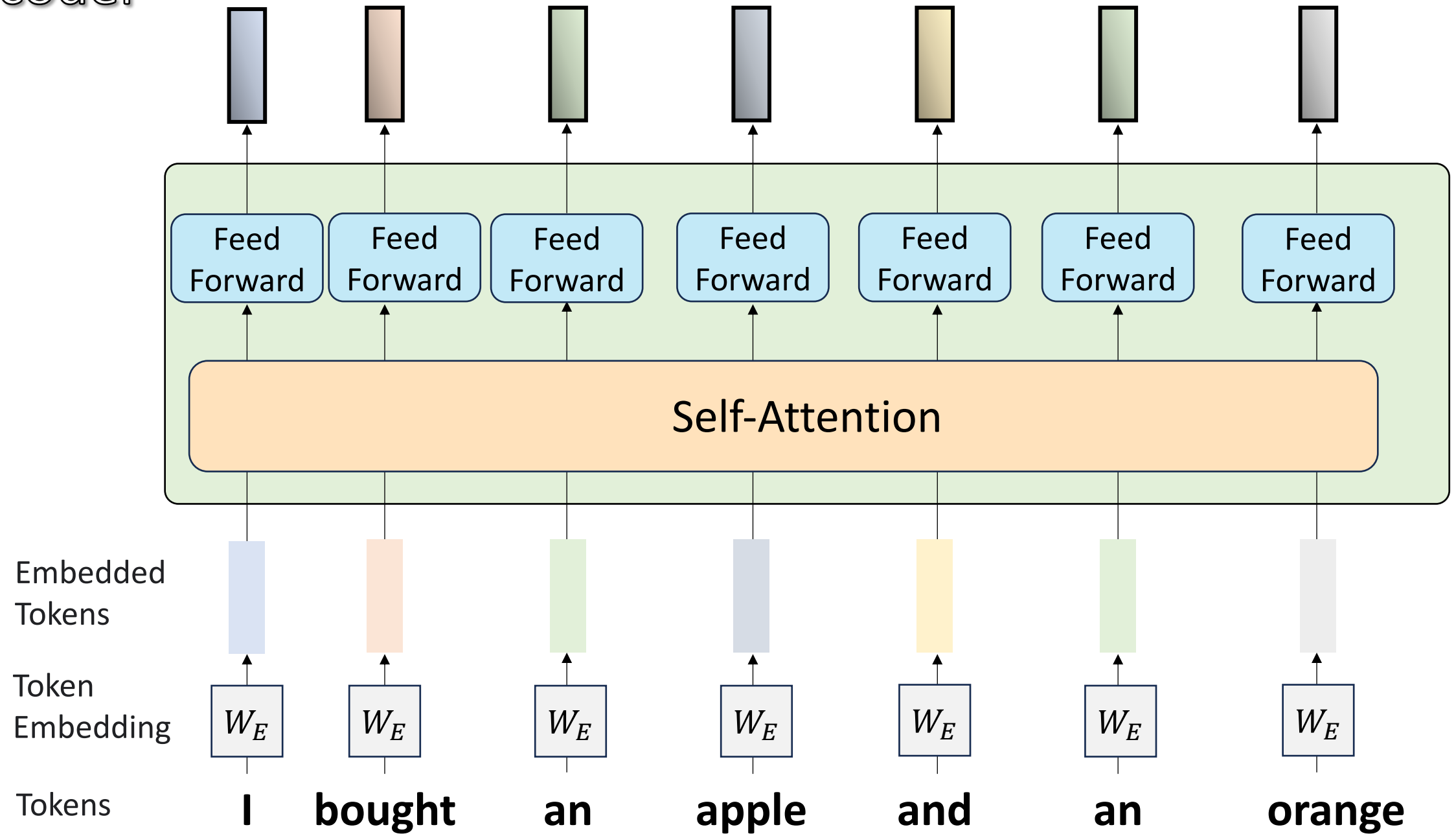
$$d \begin{bmatrix} 0.5 \\ 2.7 \\ 1.2 \\ \vdots \\ 0.2 \end{bmatrix} = d \begin{bmatrix} & \overleftarrow{\text{\# tokens}} & \overrightarrow{} \\ & \cdots & \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{matrix} \text{dog}$$

Embedded token

Embedding Matrix

* Slides from Prof. Kai-Bin Huang

Encoder



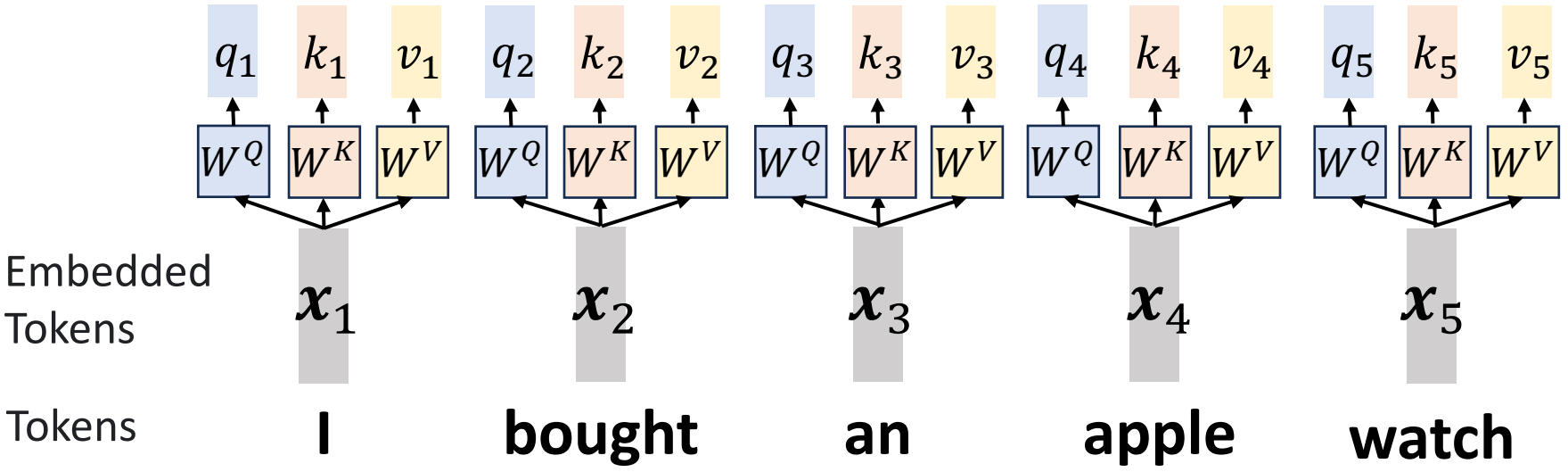
* Slides from Prof. Kai-Bin Huang

Single-head attention

$$\text{Attention}(Q, K, V) = V\ \text{softmax}\left(\frac{K^\top Q}{\sqrt{d_k}}\right)$$

$W^Q \in R^{d_k \times d}$ $W^K \in R^{d_k \times d}$ $W^V \in R^{d_v \times d}$

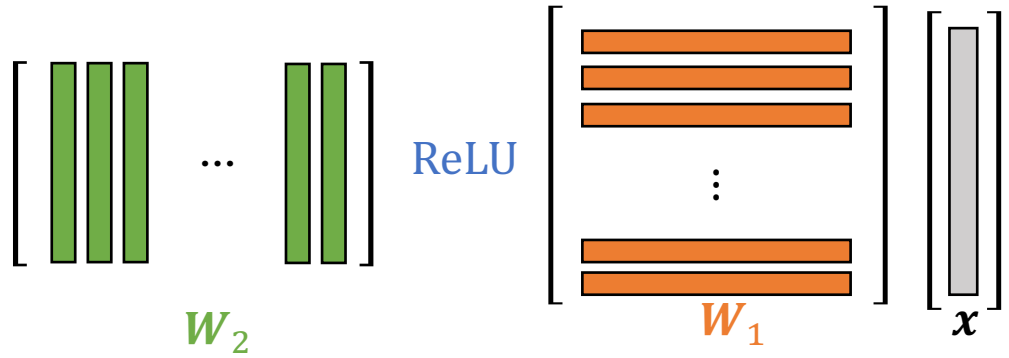
$$Q = W^Q\ x_1\ x_2\ x_3\ x_4\ x_5 \qquad K = W^K\ x_1\ x_2\ x_3\ x_4\ x_5 \qquad V = W^V\ x_1\ x_2\ x_3\ x_4\ x_5$$

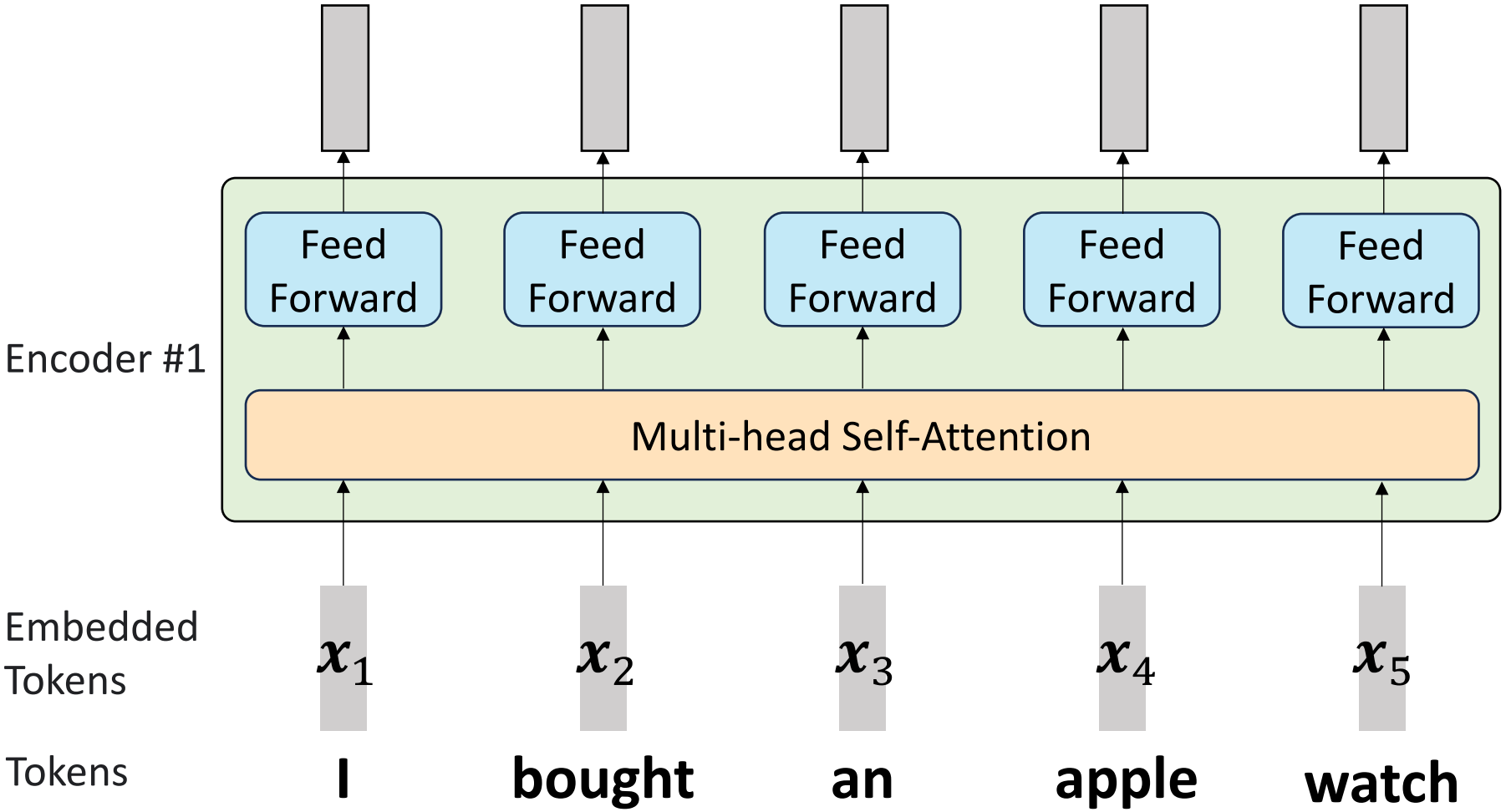


* Slides from Prof. Kai-Bin Huang

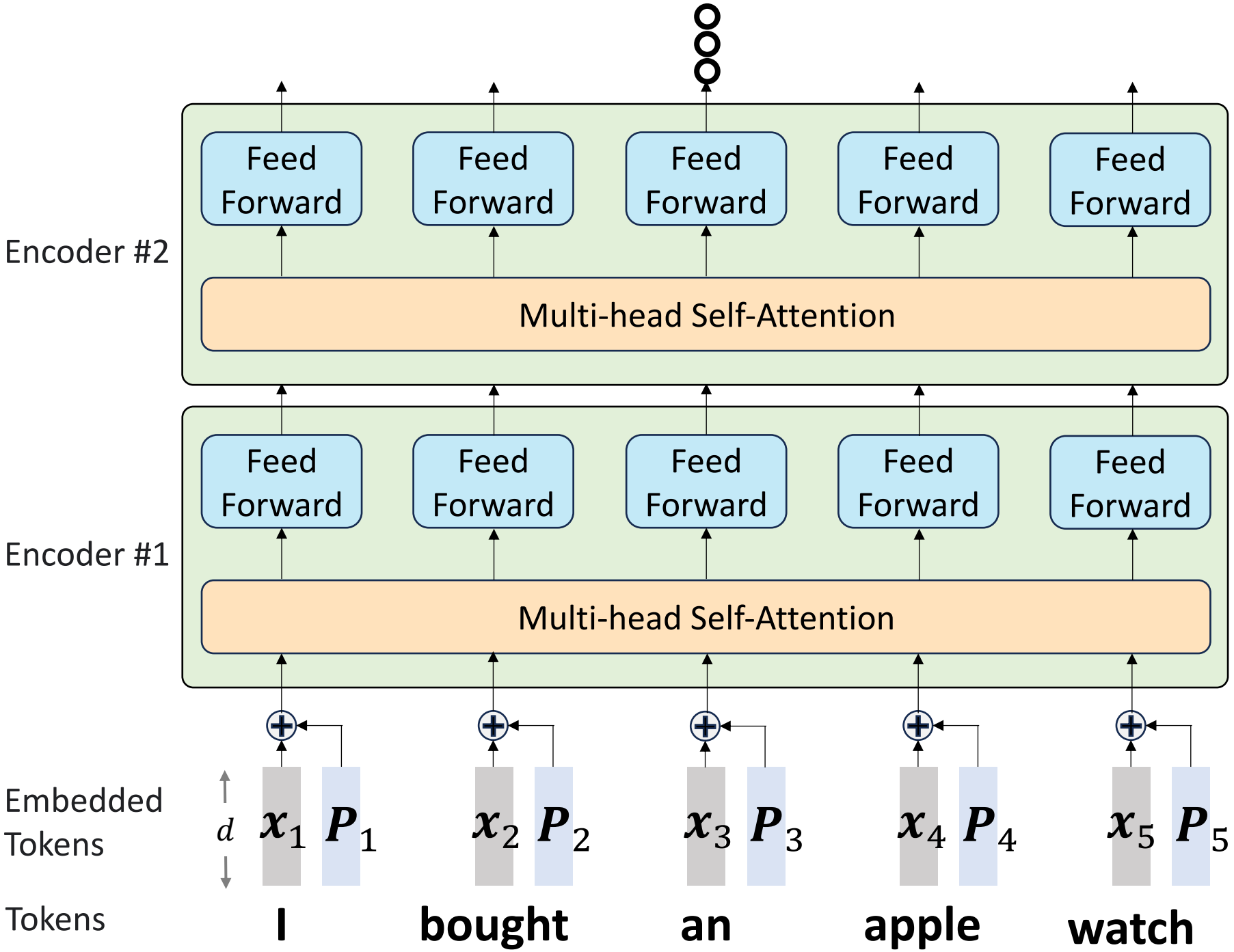
Feed Forward Network (FFN)

$$FFN(\boldsymbol{x}) = \boldsymbol{W}_2 \mathrm{ReLU}(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2$$

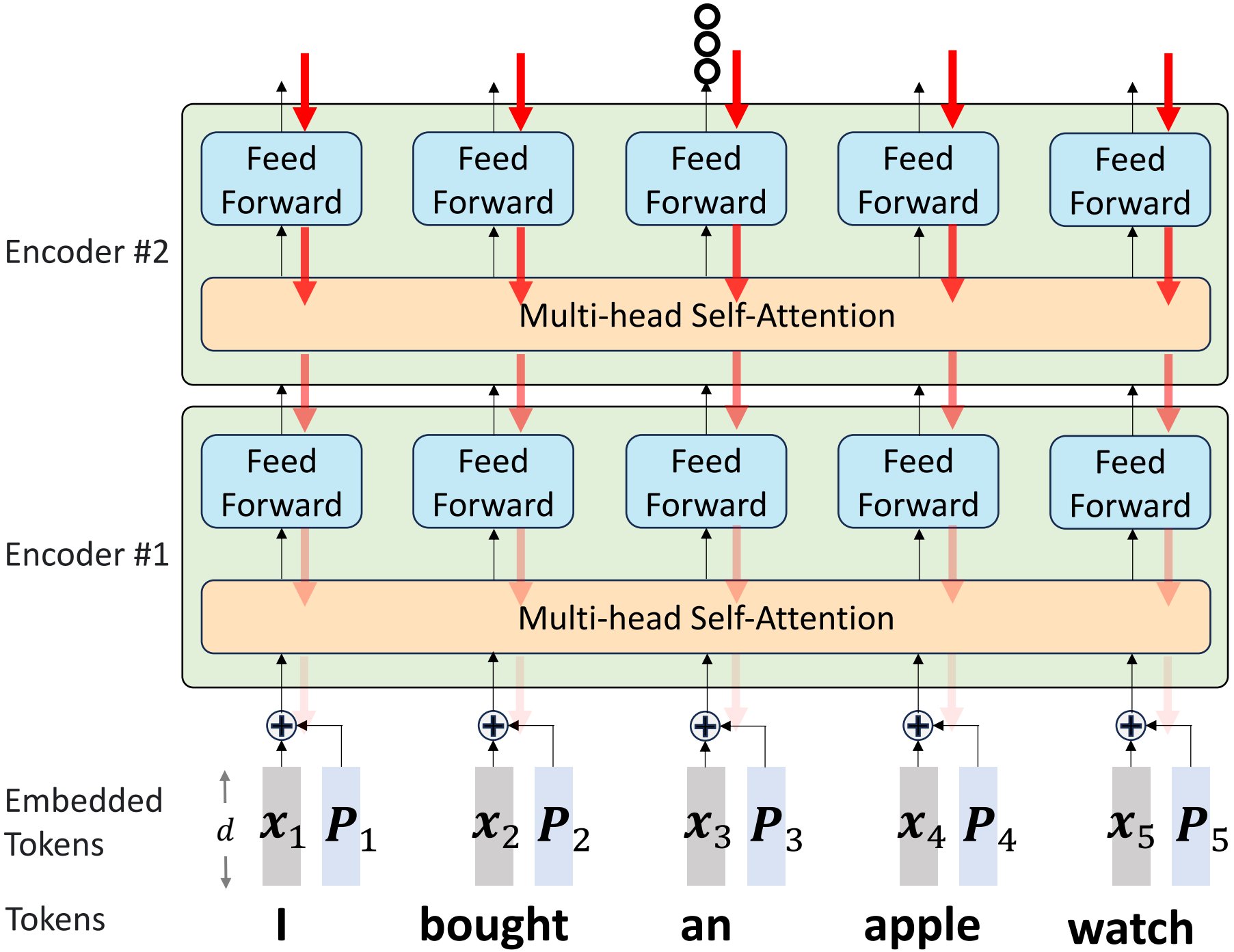




* Slides from Prof. Kai-Bin Huang



* Slides from Prof. Kai-Bin Huang



* Slides from Prof. Kai-Bin Huang

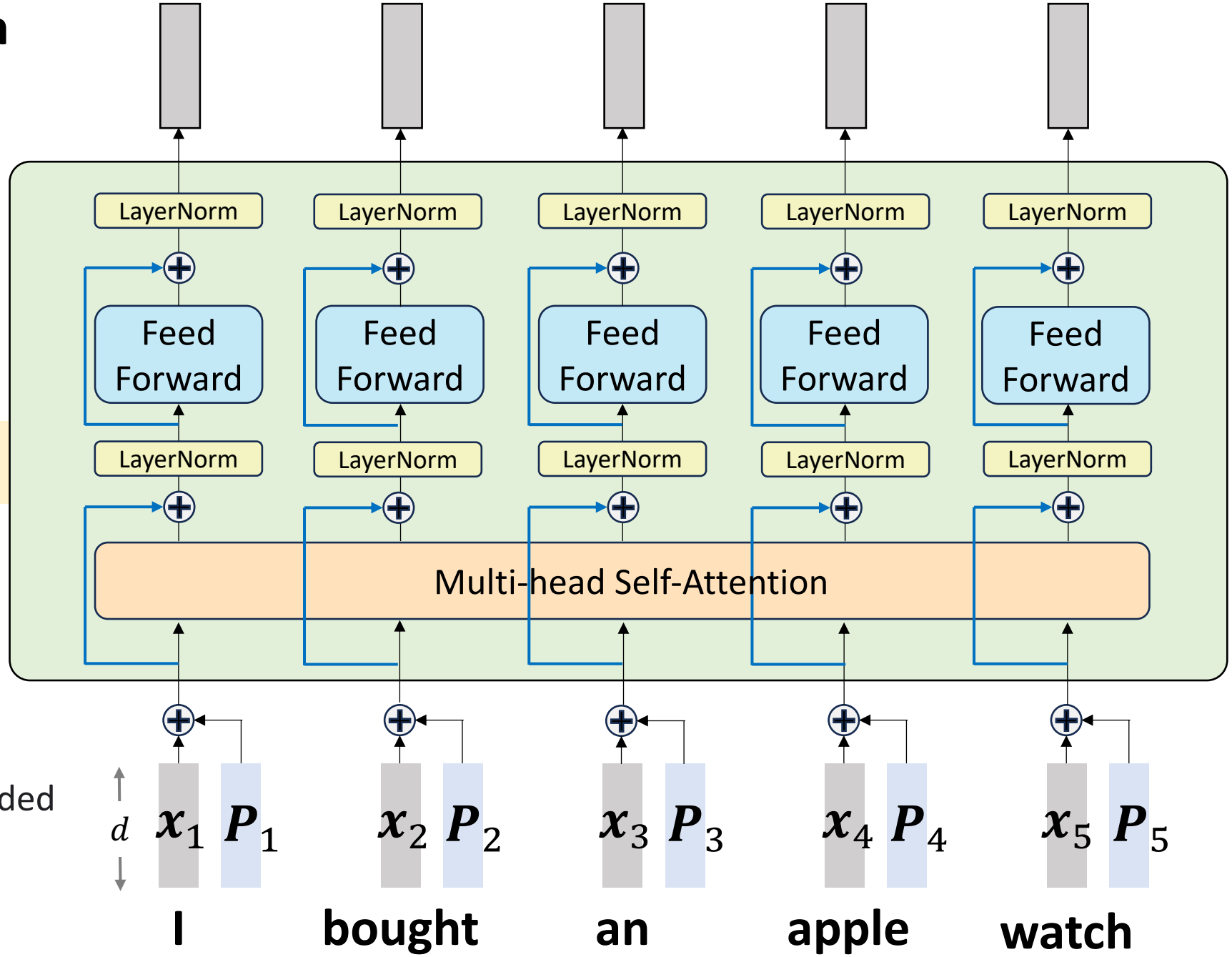
Residual connection

Layer normalization

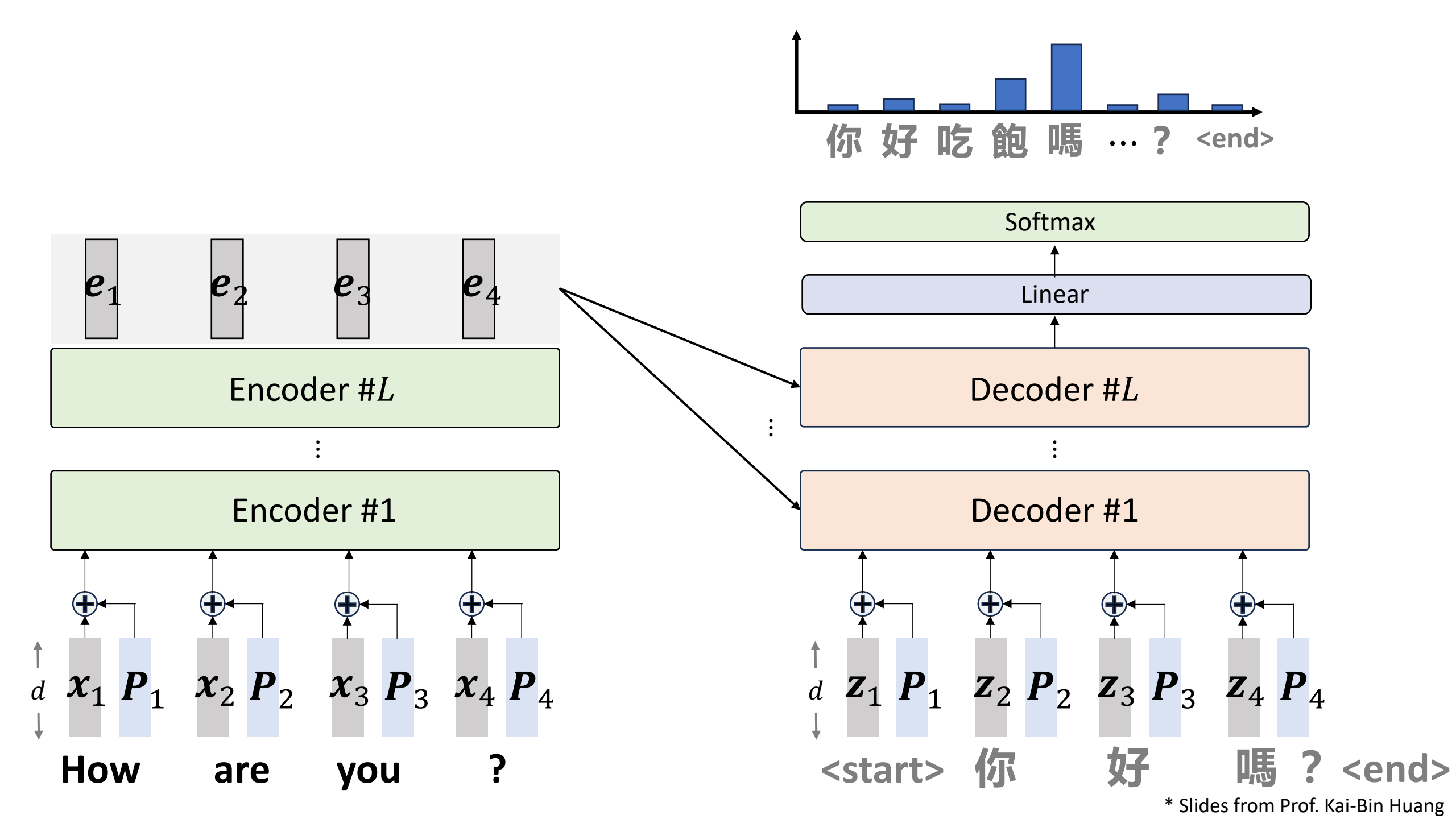
$$\text{LayerNorm}(\boldsymbol{x}) = \gamma\left(\frac{\boldsymbol{x} - \text{mean}(\boldsymbol{x})}{\sqrt{\text{Variance}(\mathbf{x}) + \epsilon}}\right) + \beta$$

$\gamma, \beta \in R$

Learnable parameters



* Slides from Prof. Kai-Bin Huang



* Slides from Prof. Kai-Bin Huang

Distributed LLM Training: Outline

- What to memorize
  - Computation flow instead of functionalities of different modules
  - Computational cost and memory consumption

- What to know beforehand
  - Block matrix multiplication
  - Some basic software, hardware and networking knowledge

Why do we need “distributed training”?

Quantitative Analysis

- GPT-175B models: parameters
  - Embedding layer

n_{vocab} : size of vocabulary

n_{model}: model dimension

$n_{embed} = n_{vocab} \times n_{model}$

  - Multi-head Attention

d_k: dimension of head Q and K

d_v: dimension of head V

n_{heads}: number of heads

n_{layers}: number of Transformer layers

d_{head}: dimension of head, generally equal to d_k and d_v

$W_i^Q, W_i^K, W_i^V \in R^{d_{model} \times d_{head}}$, and $W^O \in R^{d_{head} n_{heads} \times d_{head}}$

Quantitative Analysis

- GPT models: parameters
  - Multi-Layer Perceptron (MLP)

d_{ffn} : dimension of fully connected layer

$b_1 \in R^{d_{ffn}}, b_2 \in R^{d_{model}}$ and $W_1, W_2 \in R^{d_{ffn} \times d_{model}}$

$$n_{MLP} = 2\, d_{ffn} \times d_{model} + d_{ffn} + d_{model}$$

  - Total number of parameters

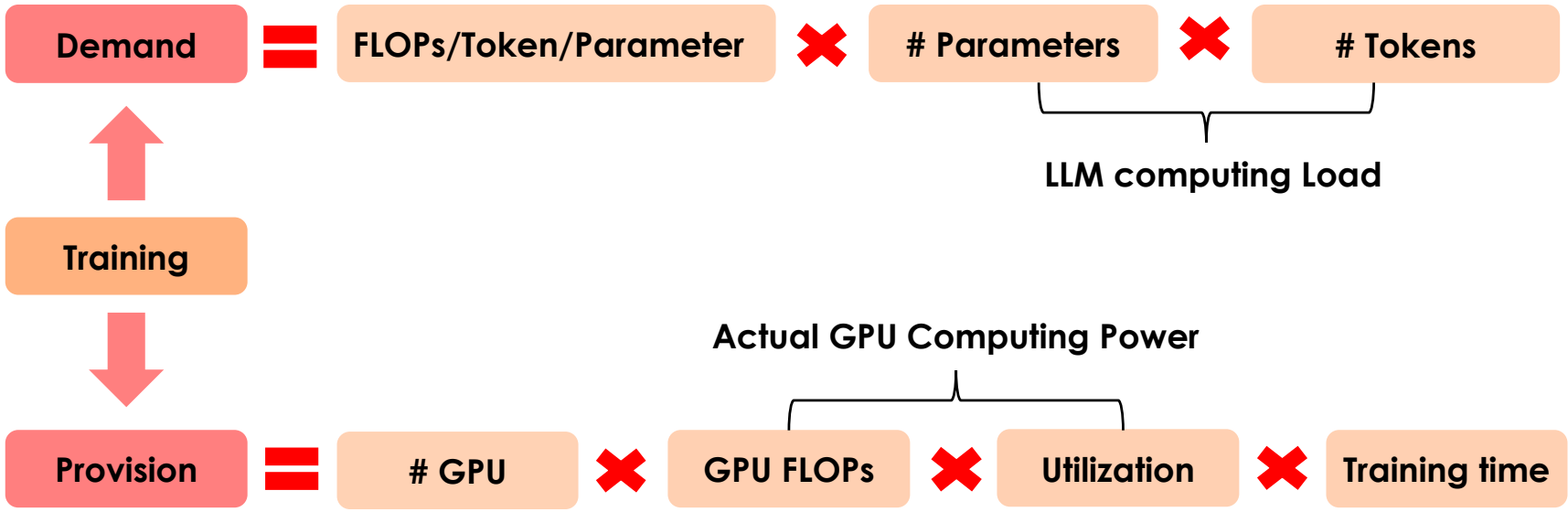
$$n_{total} = n_{vocab}\, n_{model} + n_{layers}(4\, n_{heads}\, d_{model}\, d_{head} + 2\, d_{ffn}\, d_{model})$$

Since $d_{model} = d_{head} n_{head}$ and letting $d_{ffn} = 4 d_{model}$

$$n_{total} = n_{vocab}\, n_{model} + n_{layers}(12 d_{model}^2 + 5 d_{model})$$

Quantitative Analysis

- How many GPUs do we need?



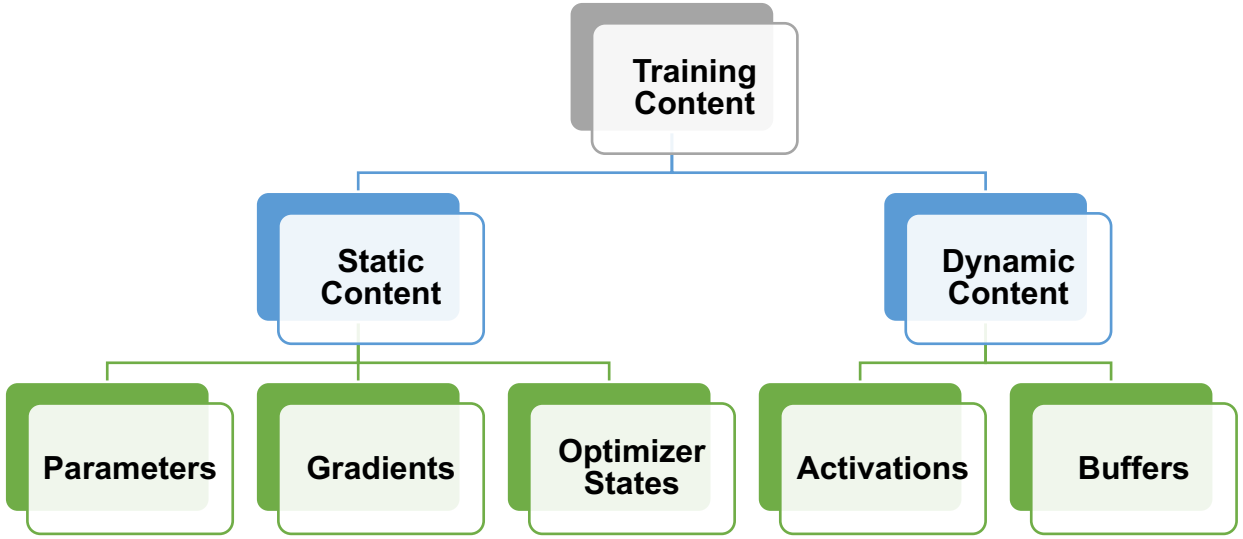
$$6 \times 175B \times 300B \;/\; 160TFLOPs \;/\; 3600 * 24 * 30 = 800$$

parameters | # tokens | TFLOPs/GPU | Second | # GPUs

《Language Models are Few-Shot Learners》
《Efficient large-scale language model training on GPU clusters using megatron-LM》

Quantitative Analysis

- GPU HBM Content During Training



- An example of GPT-3 175B
  - **Optimizer States**
    - 32-bit Parameter (700 GB)
    - Adam Moment (700 GB)
    - Adam Variance (700 GB)
  - 16-bit Parameter (350 GB)
  - 16-bit Gradient (350 GB)
  - Activations (depending on batch size)
  - Buffer and Fragmentation

Quantitative Analysis

- Adam Optimizer (Adaptive Moment Estimation)
  - First moment (mean) estimate

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

  - Second moment (variance) estimate

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial L}{\partial w_t} \right)^2$$

  - Bias correction

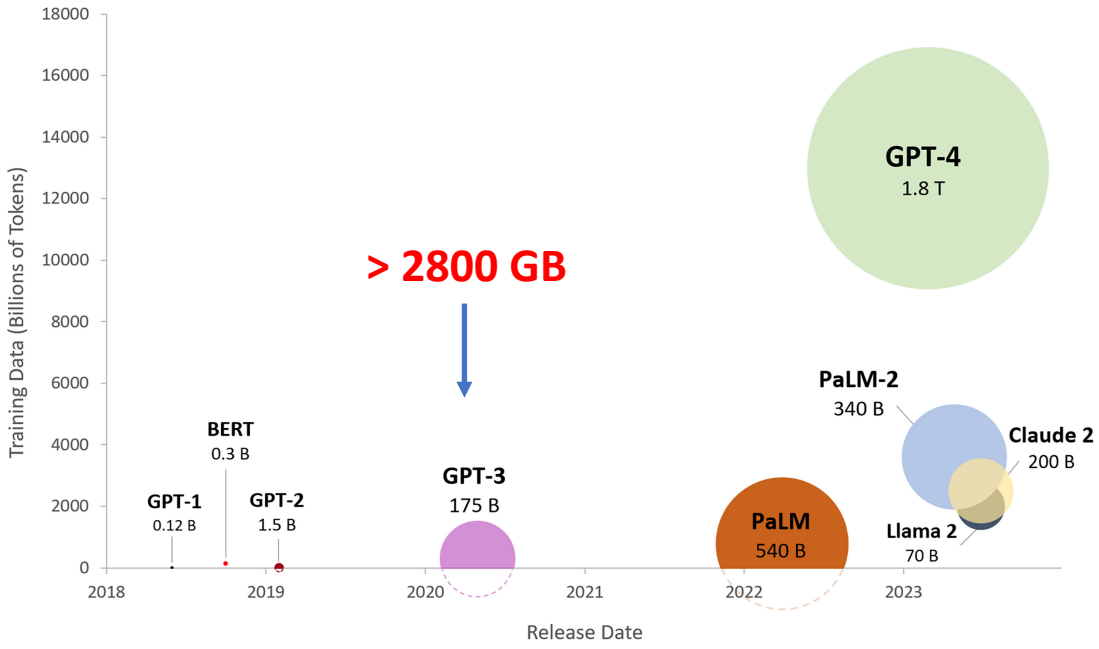
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

  - Final parameter

$$w_{t+1} = w_t - \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \alpha$$

Quantitative Analysis

- GPT-175B models: storage



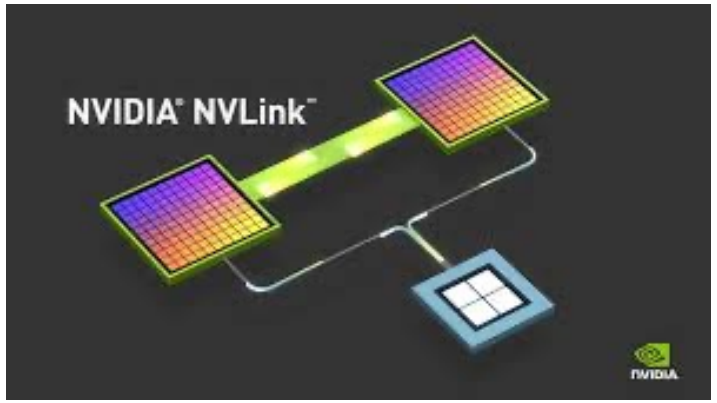
Models become larger and larger

GPU	Memory	Type
Tesla P100	16GB	HBM2
Tesla V100	32GB	HBM2
Tesla A100	40/80GB	HBM2E
Tesla A800	80GB	HBM2E
Tesla H100	80GB	HBM3
Tesla H800	80GB	HBM3

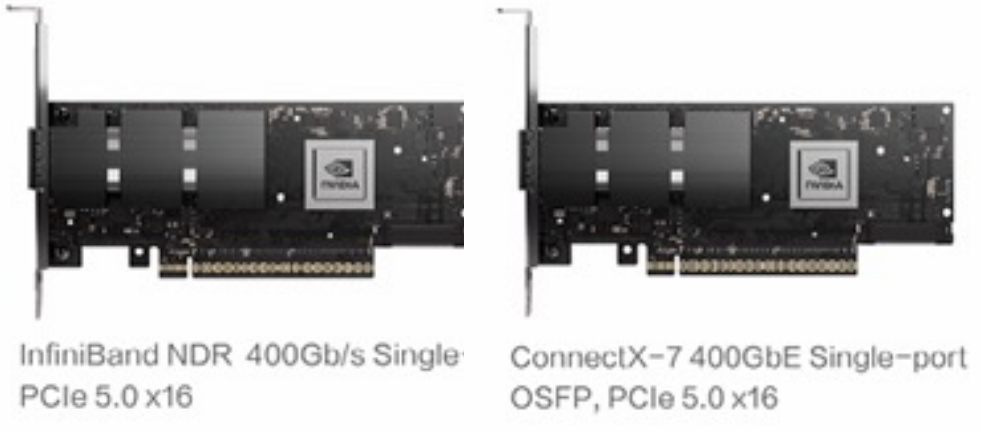
GPU HBM Size Remains Small

Distributed Training

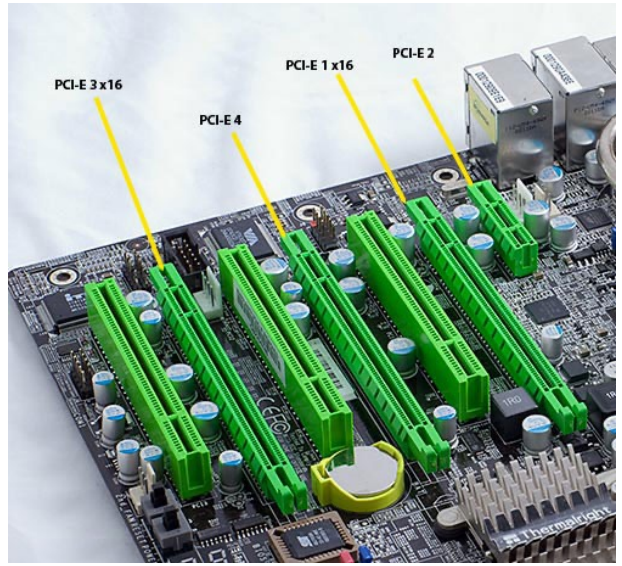
- Multiple GPUs compute collaboratively
  - Data Parallel, Pipeline Parallel, Tensor Parallel, Expert Parallel, Sequence Parallel, Context Parallel
- GPU interconnection



NVLink 5.0 (e.g. 1.8TB/s)



RDMA (RoCE/InfiniBand, e.g. 400Gbps)



PCIe 5.0 (e.g. 128GB/s, 16lanes)

Distributed Training

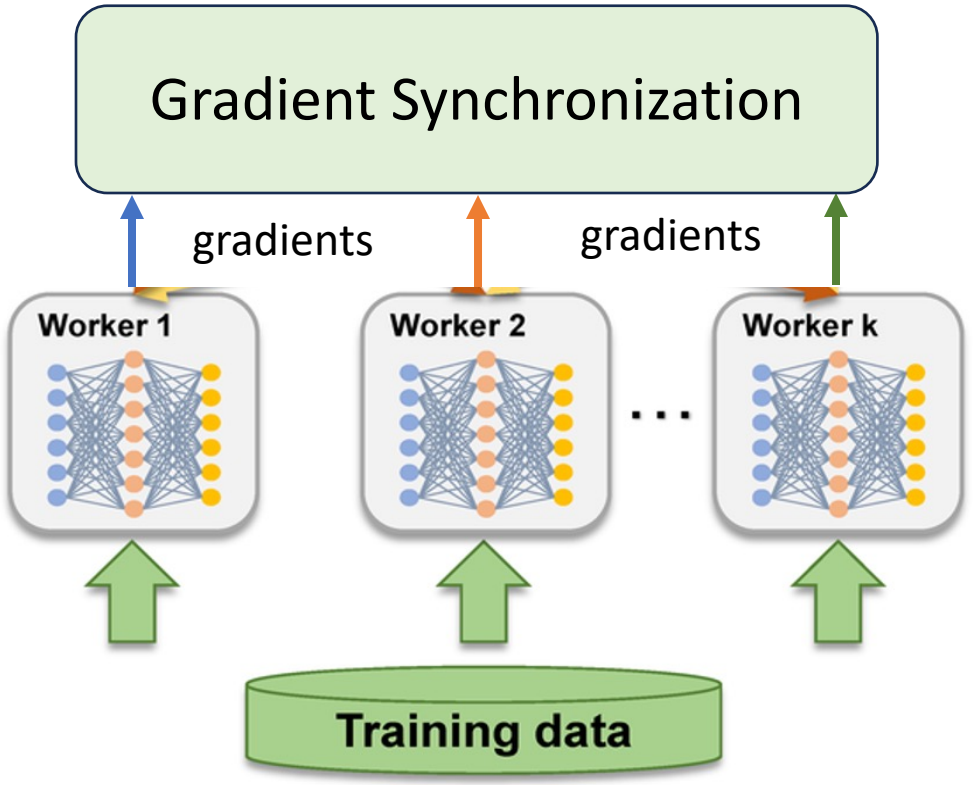
- Importance of Communication
  - Traffic volume
    - 16-bit gradient of GPT3-175B: 350GB needs to be transmitted in every iteration
    - Even more data communication if model is partitioned
  - Imbalanced technology upgrading
    - Ampere A100 (FP16 312TFLOPS, released in 2020) → Blackwell B300 (FP8 72PFLOPS, 2025
    - NVLink 3.0 (600GB/s, 2020) → NVLink 4.0 (900GB/s, 2022) → NVLink 5.0 (1.8TB/s, 2025)

Distributed LLM Training: Outline

- Data Parallelism
  - **Parameter-Server**
  - All-Reduce
  - Memory Optimization
- Model Parallelism
  - Pipeline Parallelism
  - Tensor Parallelism
  - Sequence Parallelism
- Mixture of Experts

Distributed LLM Training: Outline

- Data Parallelism
  - Distribute data to workers
  - Each worker work independently
  - Synchronize gradients via different approaches
  - Repeat the above procedures until model convergence



⇨ *Aggregate compute power*

Parameter Server Architecture

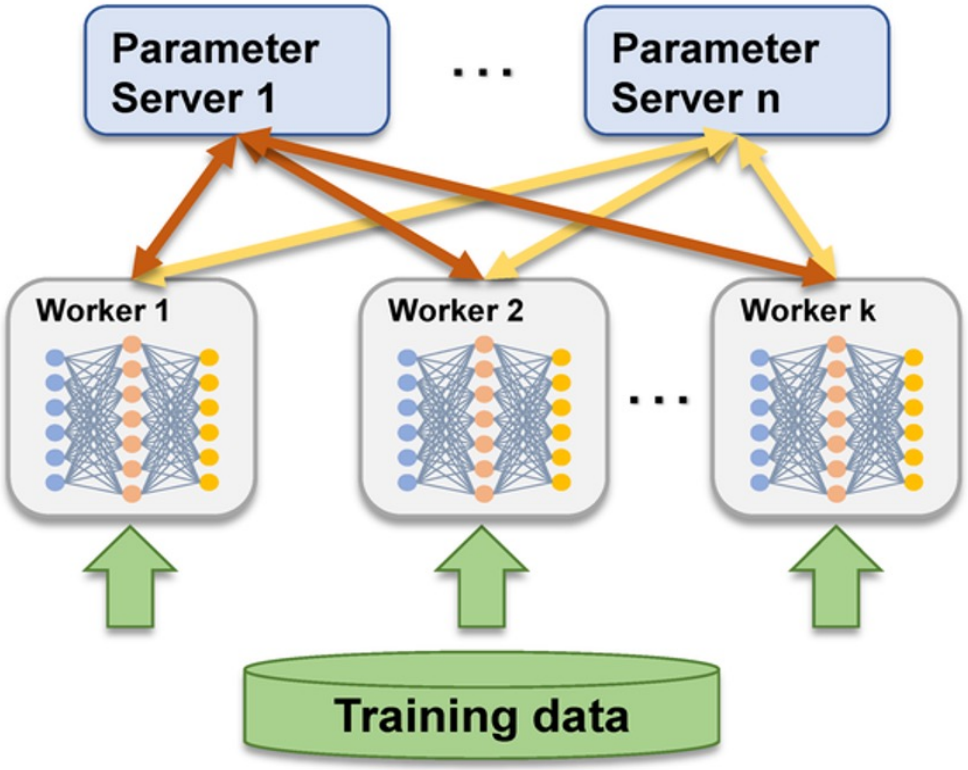
4. $Update$: $W^{new} = W - \gamma \Delta W$

5. $Pull$: W^{new}

3. $Push$: ΔW

2. $back\ propagation$

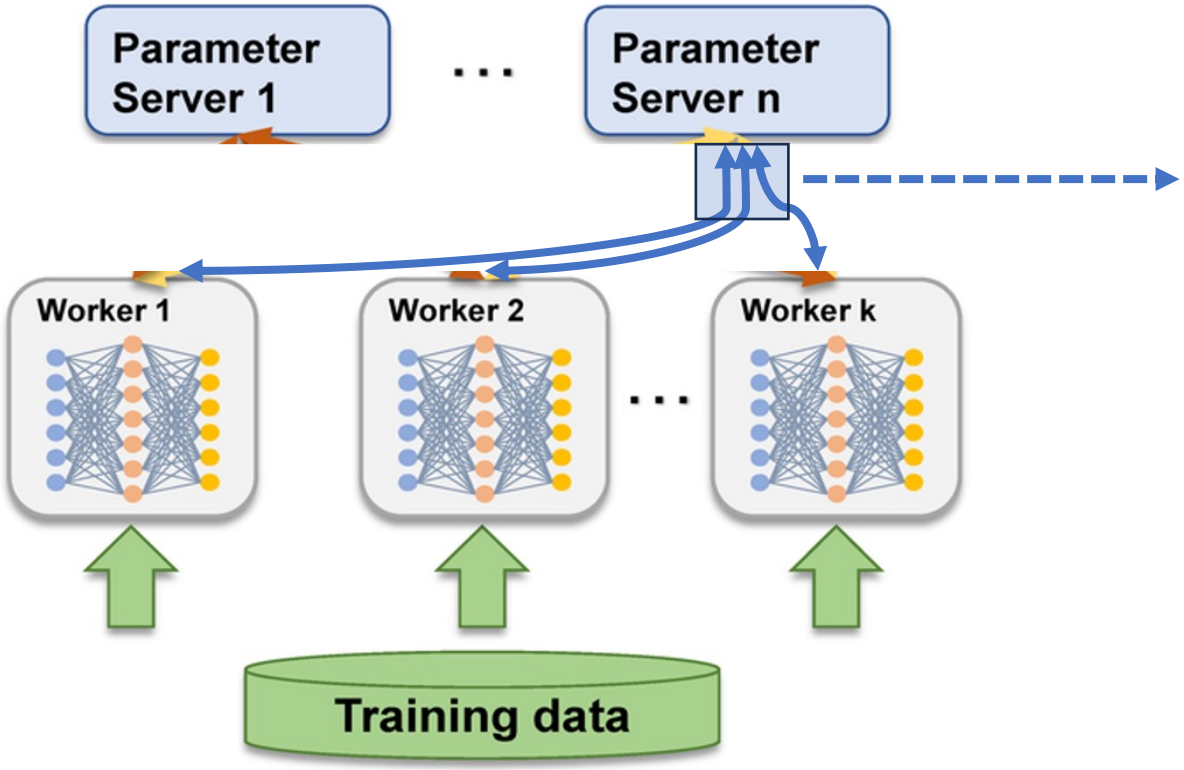
1. $forward\ propagation$



Distributed Parameter Server Architecture:

Augmenting Bandwidth

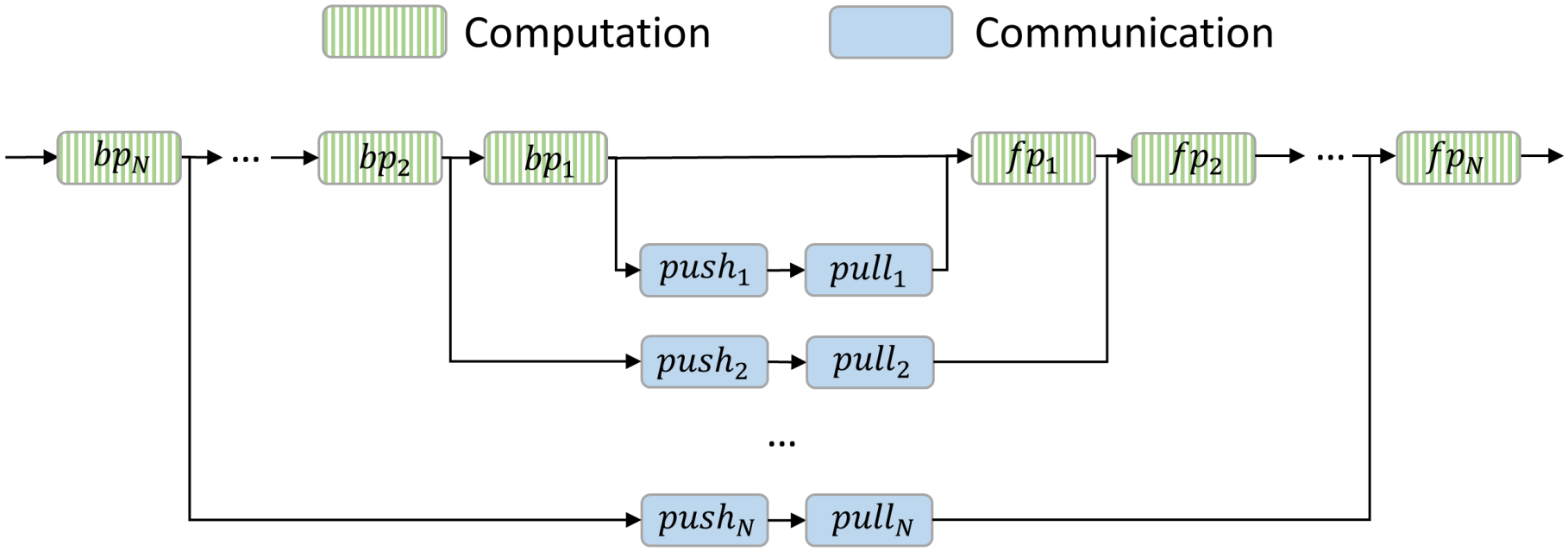
Parameter Server Architecture



Network Bottleneck

- Shared bottleneck
  - Communication throttles computation
  - Reduced bandwidth efficiency due to multi-flow competition
  - Difficult to overlap comp. and comm., push and pull

Optimizing PS Architecture via Scheduling

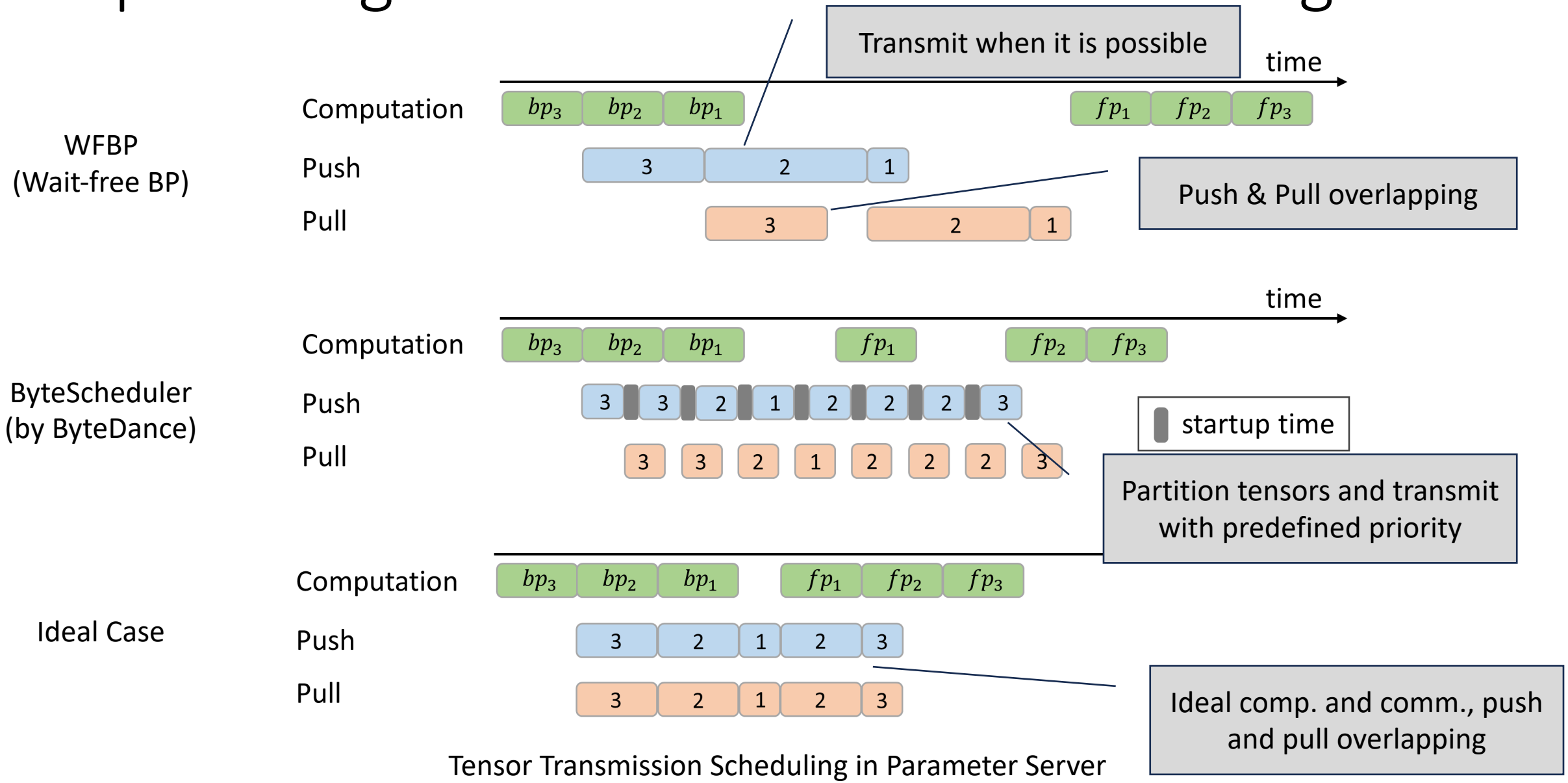


Computation order: bp_N → bp_{N-1} → … → bp_2 → bp_1 → fp_1 →fp_2 → … → fp_N

Data availability order: $gradient_N$ → $gradient_{N-1}$ → … → $gradient_2$ → $gradient_1$

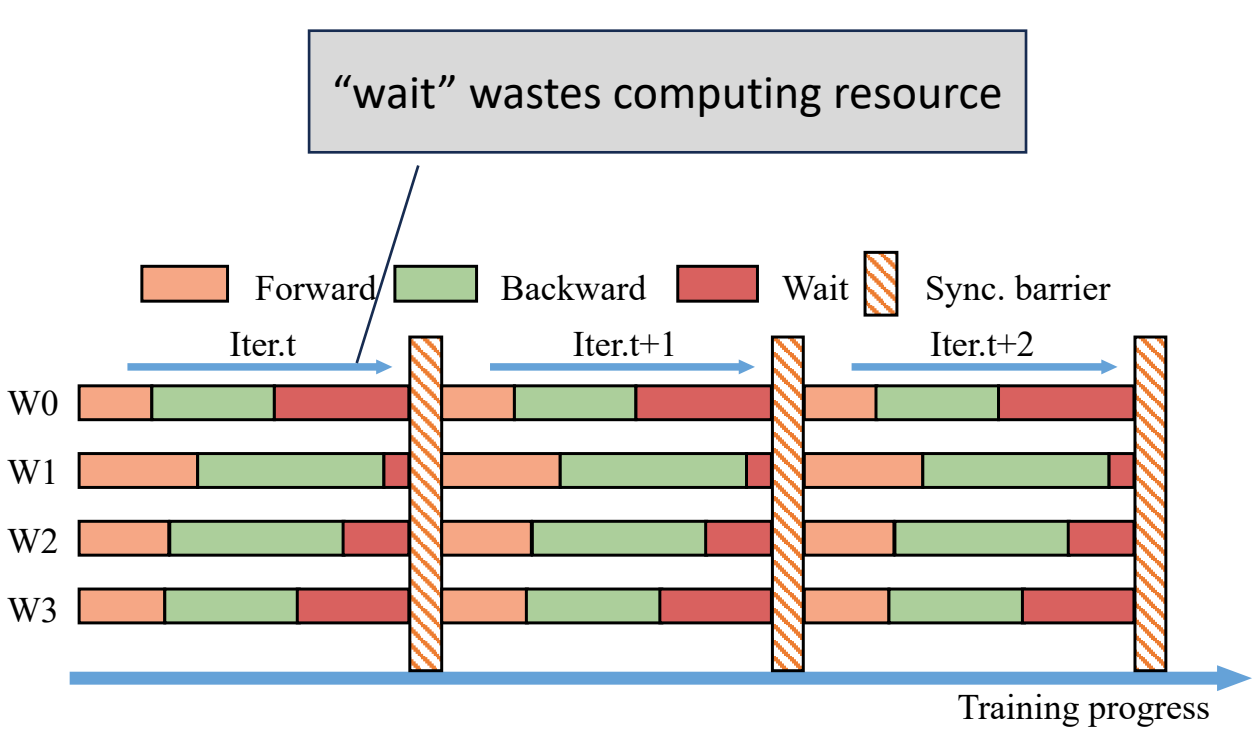
Layer-wise Computation and Communication

Optimizing PS Architecture via Scheduling

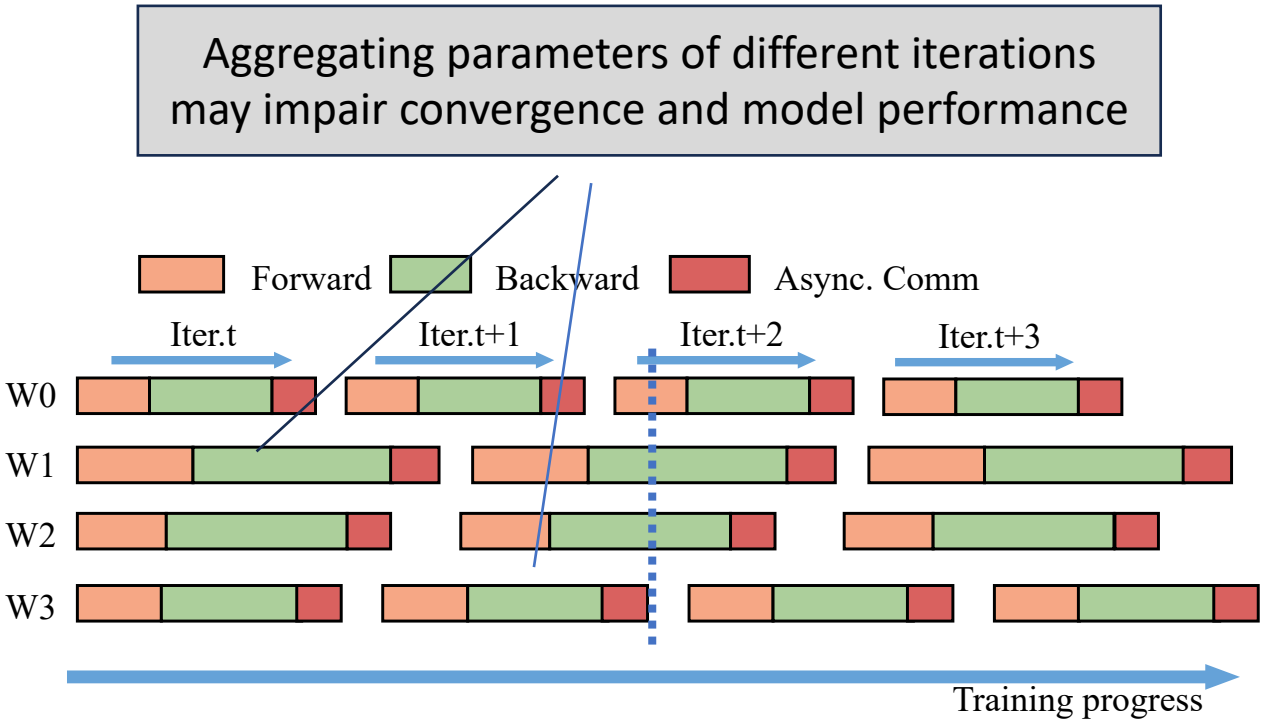


Tensor Transmission Scheduling in Parameter Server

Optimizing PS Architecture via Asynchronism



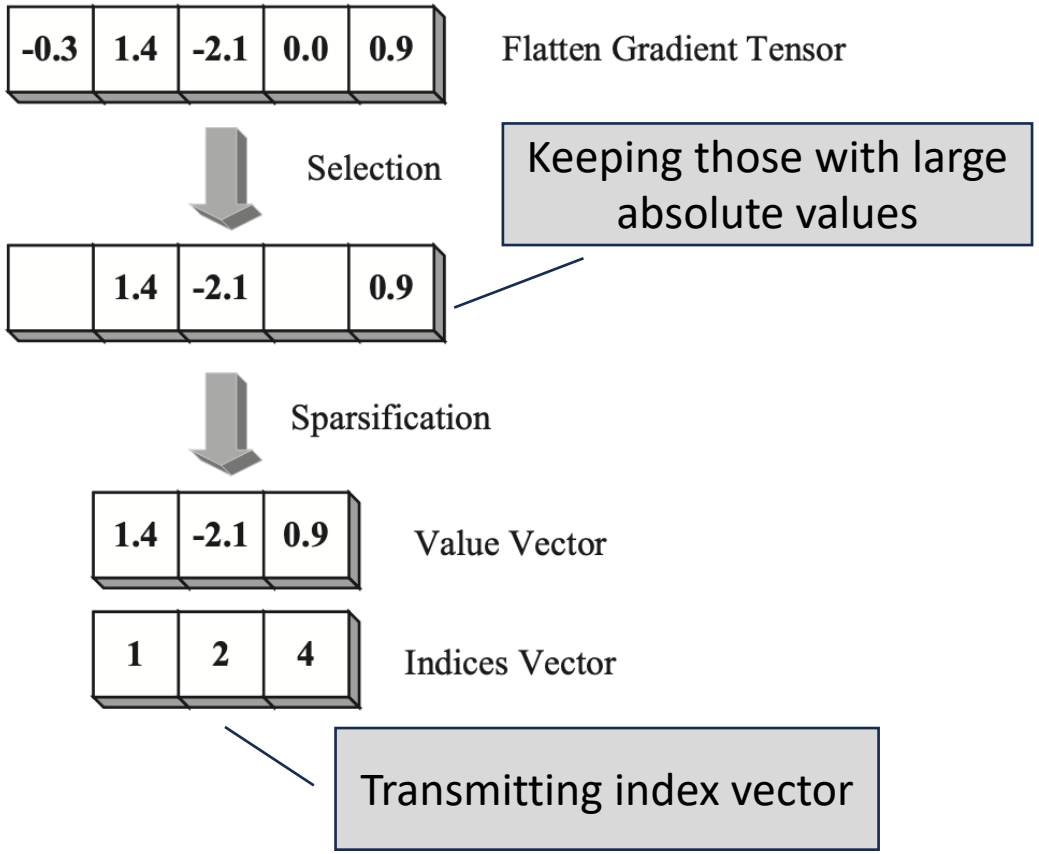
Synchronous Training: Some machines compute or communicate faster, but randomly faster



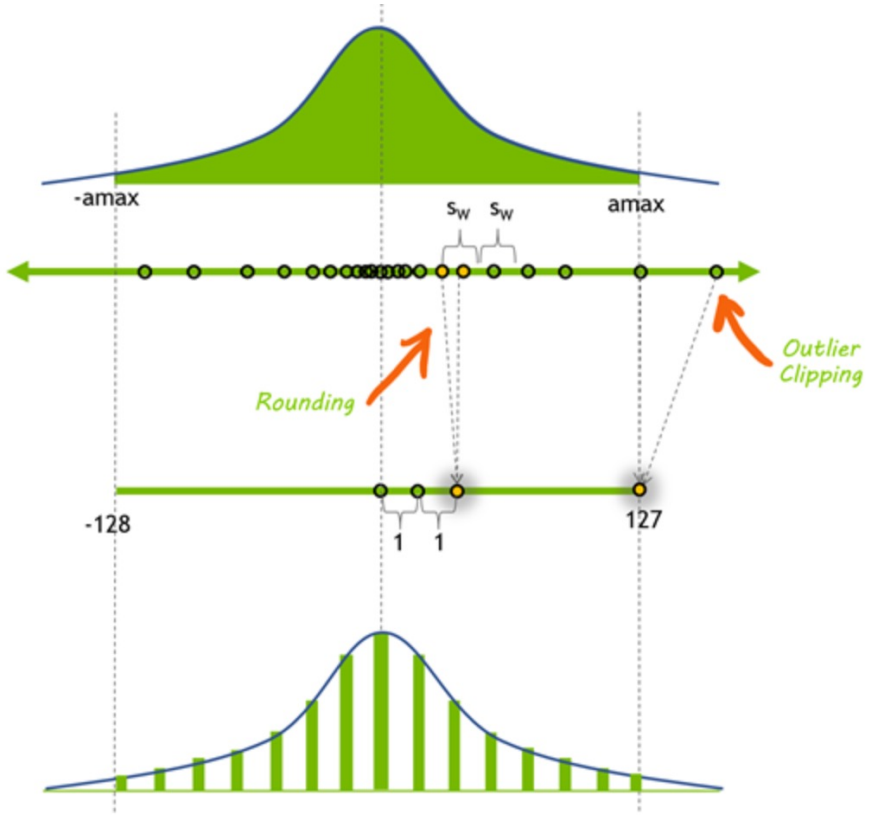
Asynchronous Training: no synchronization barrier

Asynchronous Training: Mitigating Stragglers

Optimizing PS Architecture via Compression



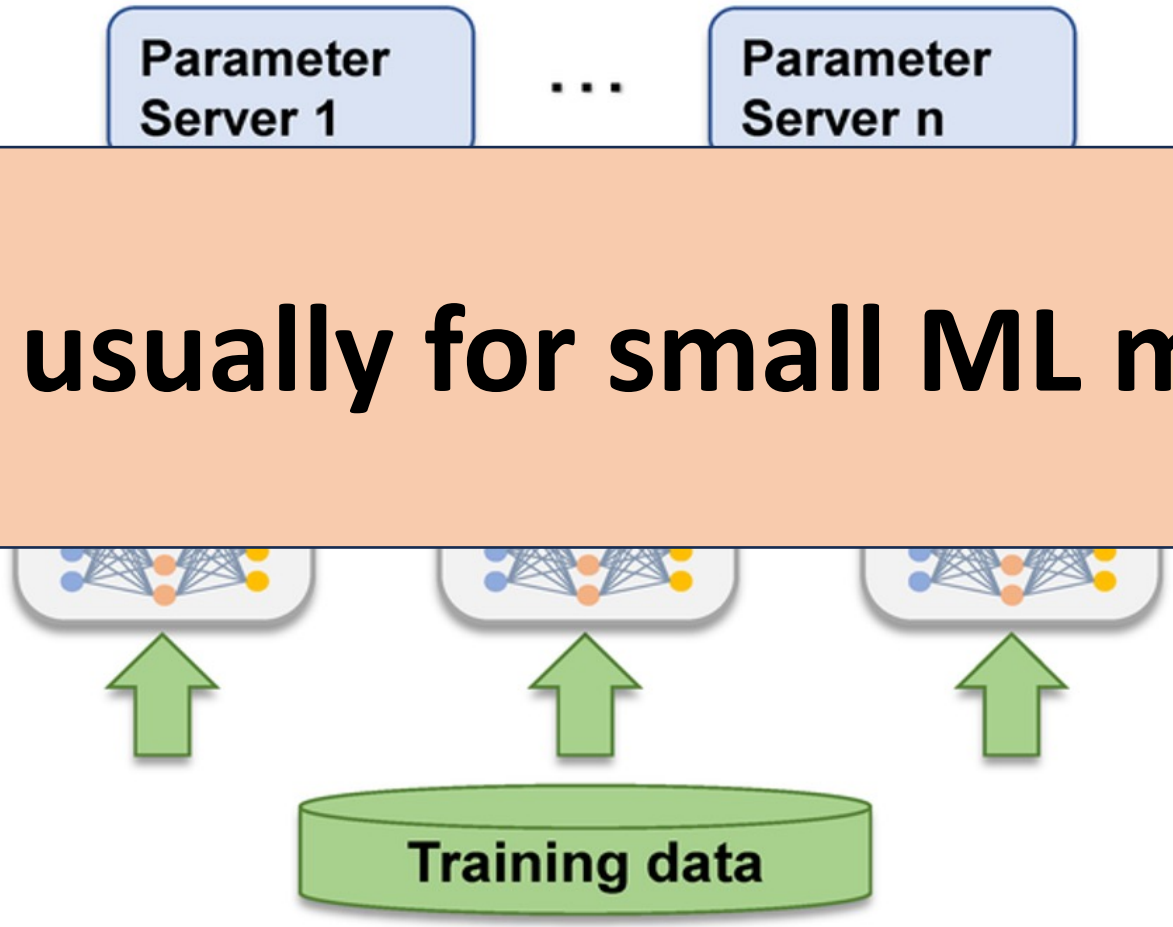
Sparsification: Reducing # of parameters



Mapping gradients into low-precision ones w/wo the consideration of value distribution

Sparsification and Quantization
《Communication Compression Techniques in Distributed Deep Learning: A Survey》

Parameter Server Architecture



Simple, usually for small ML models

Distributed LLM Training: Outline

- Data Parallelism
  - Parameter-Server
  - **All-Reduce**
  - Memory optimization
- Model Parallelism
  - Pipeline Parallelism
  - Tensor Parallelism
  - Sequence Parallelism
- Mixture of Experts

All-Reduce

- Collective Communication

> *"**Collective communication** is communication that involves a group of processing elements (termed nodes in this entry) and affects a data transfer between all or some of these processing elements. Data transfer may include the application of a reduction operator or other transformation of the data." 《Encyclopedia of Parallel Computing》*

集合通信是指一个进程组的所有进程都参与的全局通信操作。

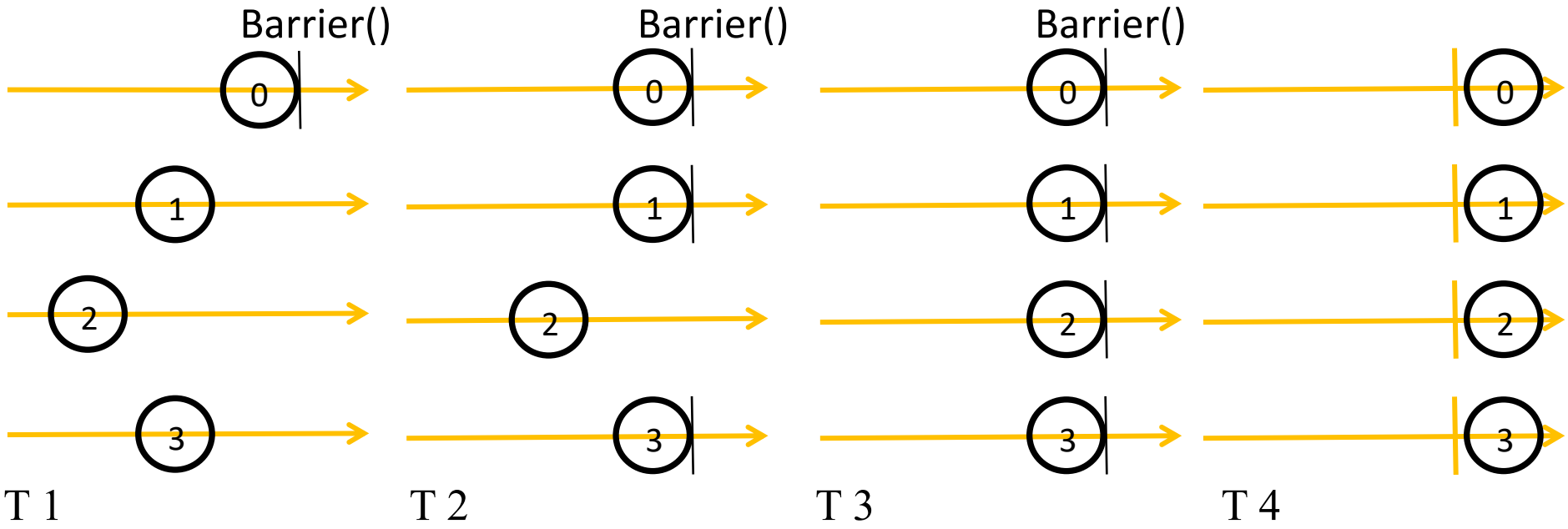
全部点对点通信完成才算集合通信完成

All-Reduce

- Why Collective Communication
  - Simplified Programming Interface
    - Developers don't need to manually code complex synchronization and data distribution logic for every scenario.
  - Scalability for Large-Scale Systems
    - As systems grow to thousands of nodes or GPUs (e.g., in supercomputers or cloud clusters), managing communication becomes exponentially complex. These libraries support sparse data handling, fault tolerance, and observability features to ensure reliable operation at scale.
  - Decompose Compute and Communication
    - Allowing machine learning researchers and system engineers to work on their own.

All-Reduce

- Collective Communication: Most Basic Operations
  - **SEND**, **RECEIVE**, COPY, BARRIER, SIGNAL+WAIL (in Message Passing Interface, i.e. MPI)



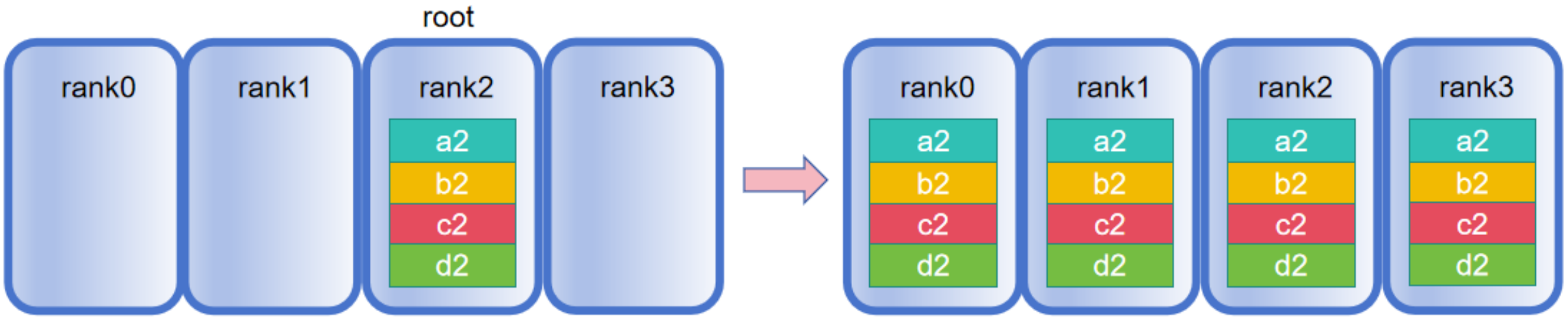
* Pictures replotted, but originated from Mengyuan @ Zhihu

All-Reduce

- Collective Communication: More Advanced Operations
  - Broadcast: one-to-many
  - Gather: many-to-one, and All-Gather: many-to-many
  - Scatter: one-to-many
  - Reduce: many-to-one, and All-Reduce: many-to-many
  - Reduce-Scatter: aggregate data and then transmit
  - All-to-All: many-to-many

All-Reduce

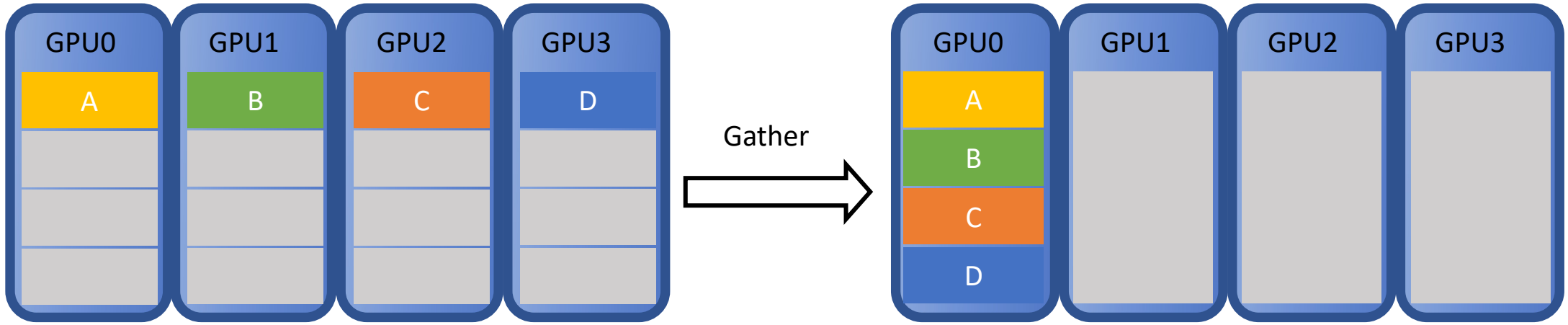
- Collective Communication: More Advanced Operations
  - Broadcast



After broadcasting, every GPU owns the same data

All-Reduce

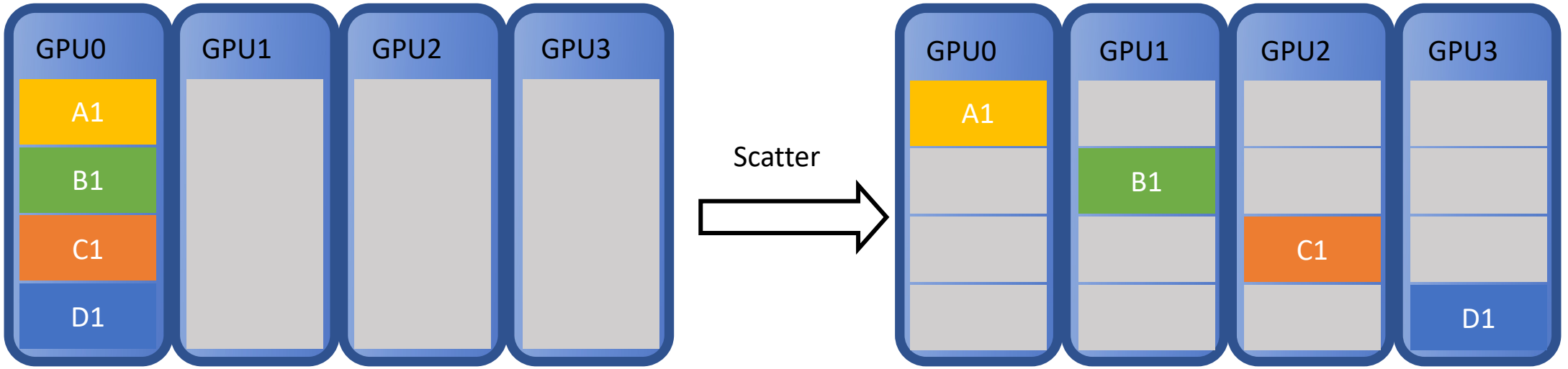
- Collective Communication: More Advanced Operations
  - Gather



A GPU collects data shards on different GPUs

All-Reduce

- Collective Communication: More Advanced Operations
  - Scatter



Disseminate each shard to a different GPU

All-Reduce

- Collective Communication: More Advanced Operations
  - Reduce

Transform data shards on different GPUs into one shard (via MIN, MAX, SUM …) at a GPU

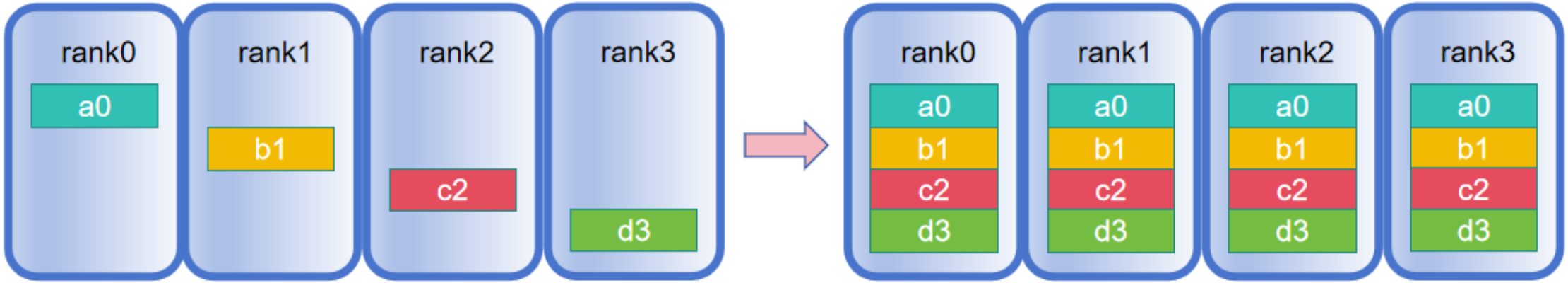
All-Reduce

- Collective Communication: More Advanced Operations
  - All-Reduce

Transform data shards on different GPUs into one shard at **every** GPU

All-Reduce

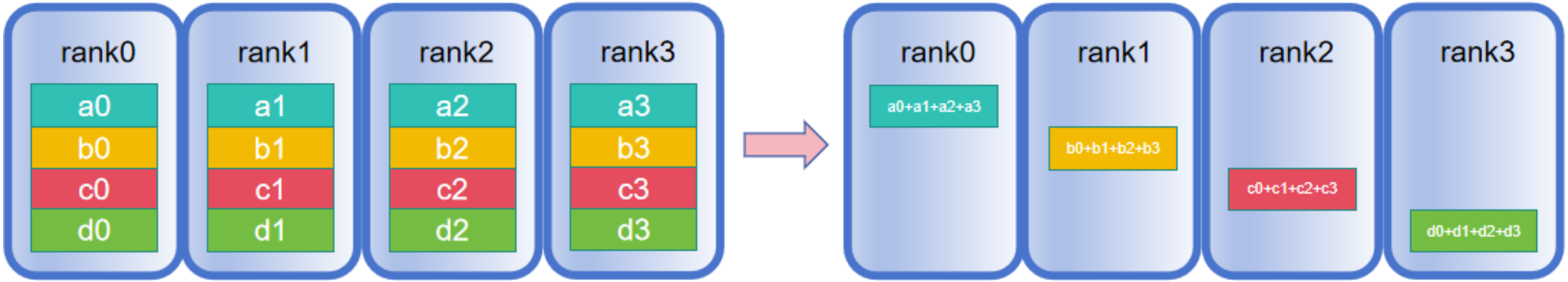
- Collective Communication: More Advanced Operations
  - All-Gather



Collect data shards on different GPUs at **every** GPU

All-Reduce

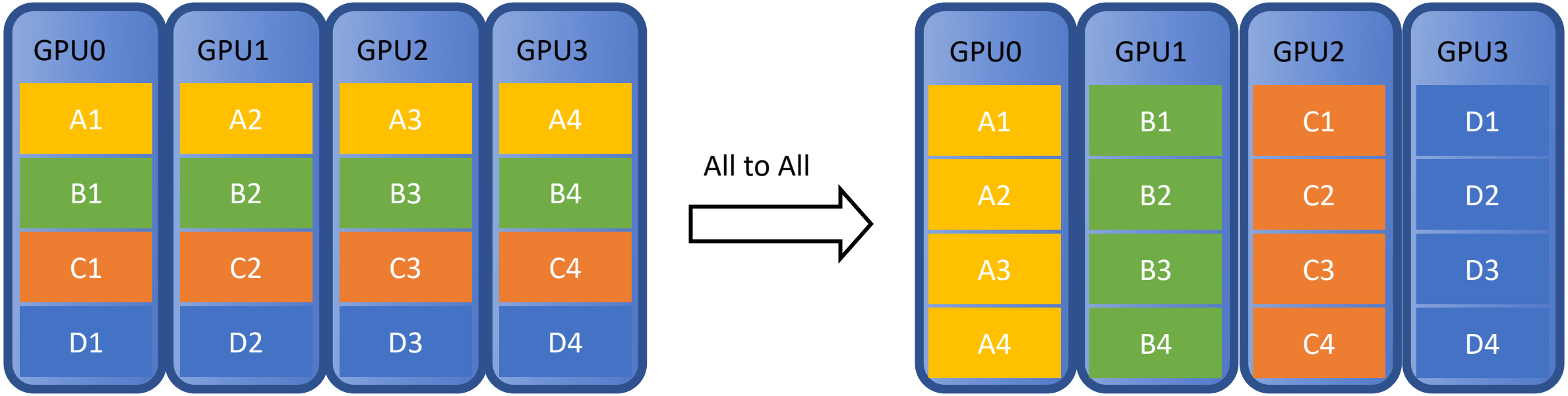
- Collective Communication: More Advanced Operations
  - Reduce-Scatter



Aggregate data chunks and store one shard at a GPU

All-Reduce

- Collective Communication: More Advanced Operations
  - All-to-All



GPU i sends j-th chunk to GPU j, and GPU j stores the chunk from GPU i at the i-th location

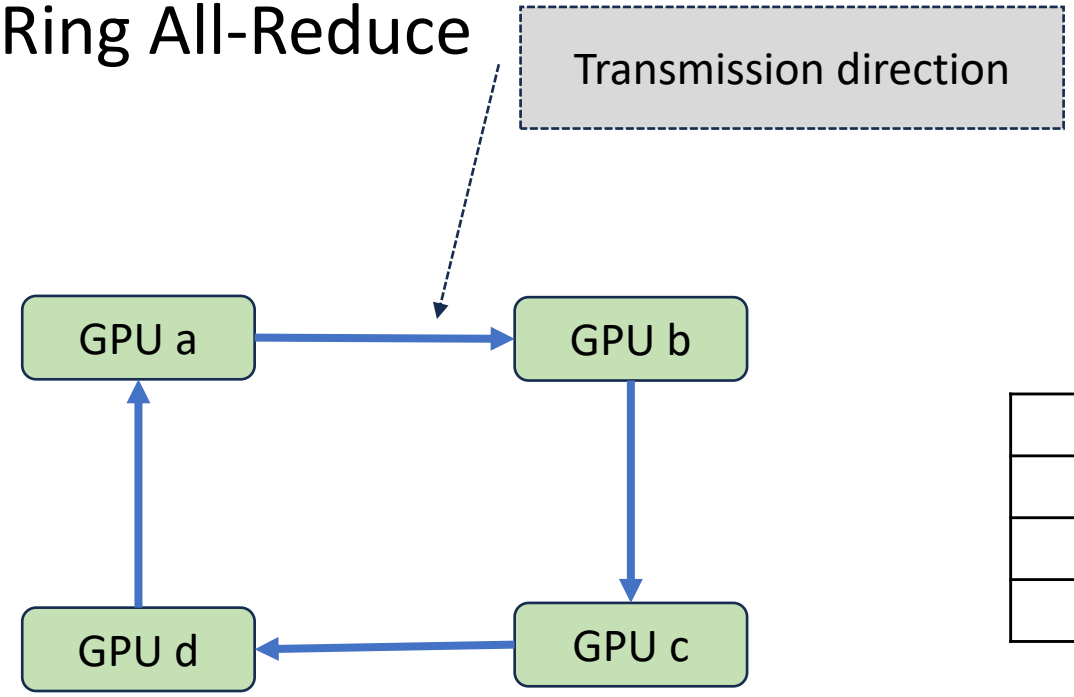
Starting from the most important “All-Reduce”

Ring All-Reduce

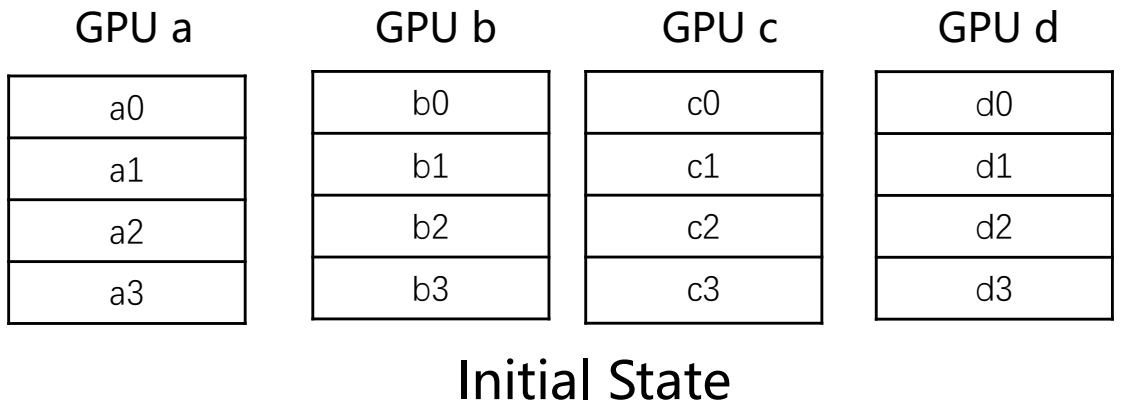
- Ring All-Reduce
- Tree All-Reduce
- Topology-aware All-Reduce

Ring All-Reduce

- Ring All-Reduce



Ring All-Reduce Overlay Topology

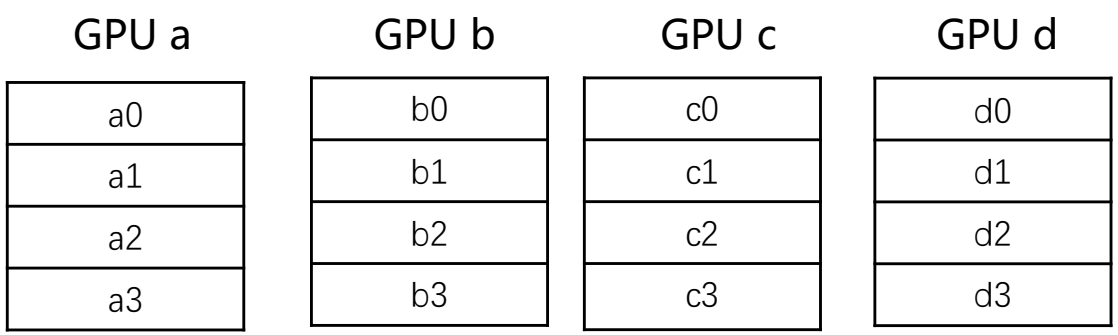


Initial State

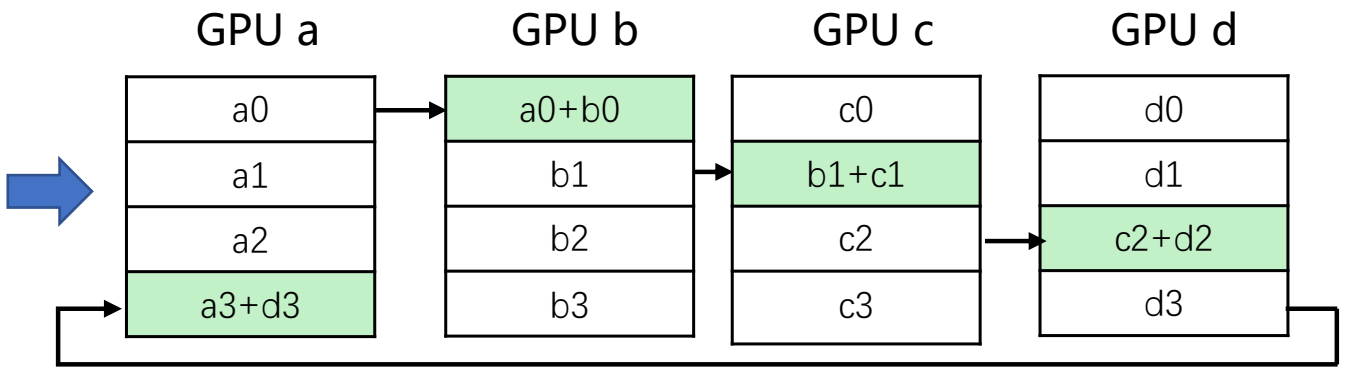
Local gradient after back propagation

Ring All-Reduce

- Ring All-Reduce



Initial State

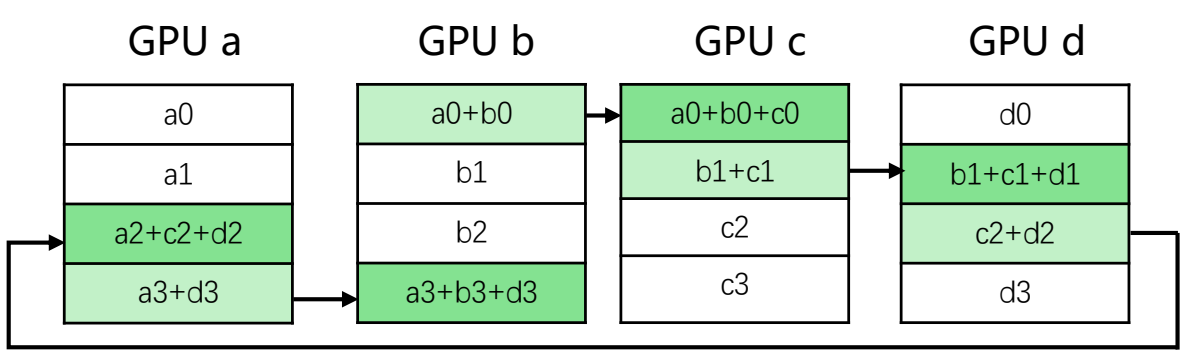


Reduce-scatter step 1

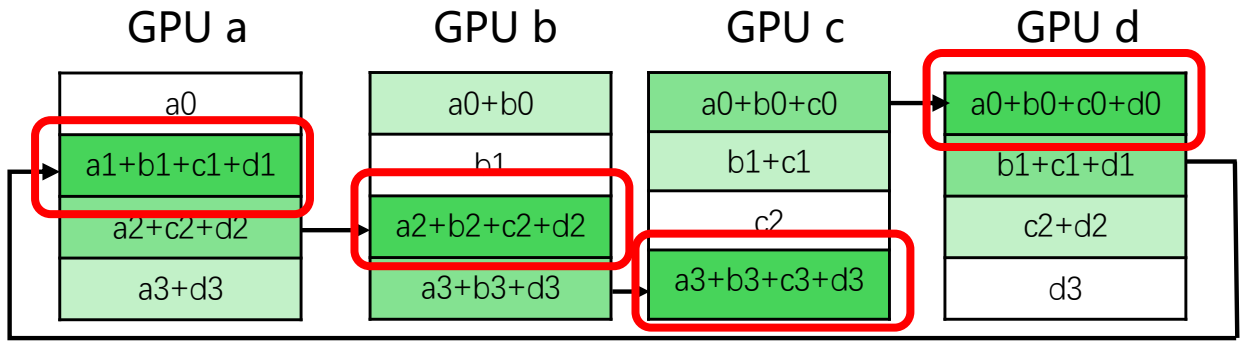
Reduce-Scatter

Ring All-Reduce

- Ring All-Reduce



Reduce-scatter step 2

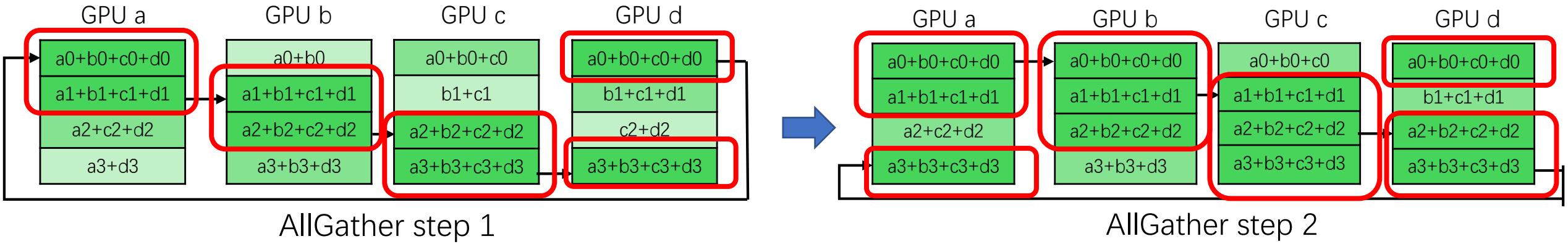


Reduce-scatter step 3

Reduce-Scatter

Ring All-Reduce

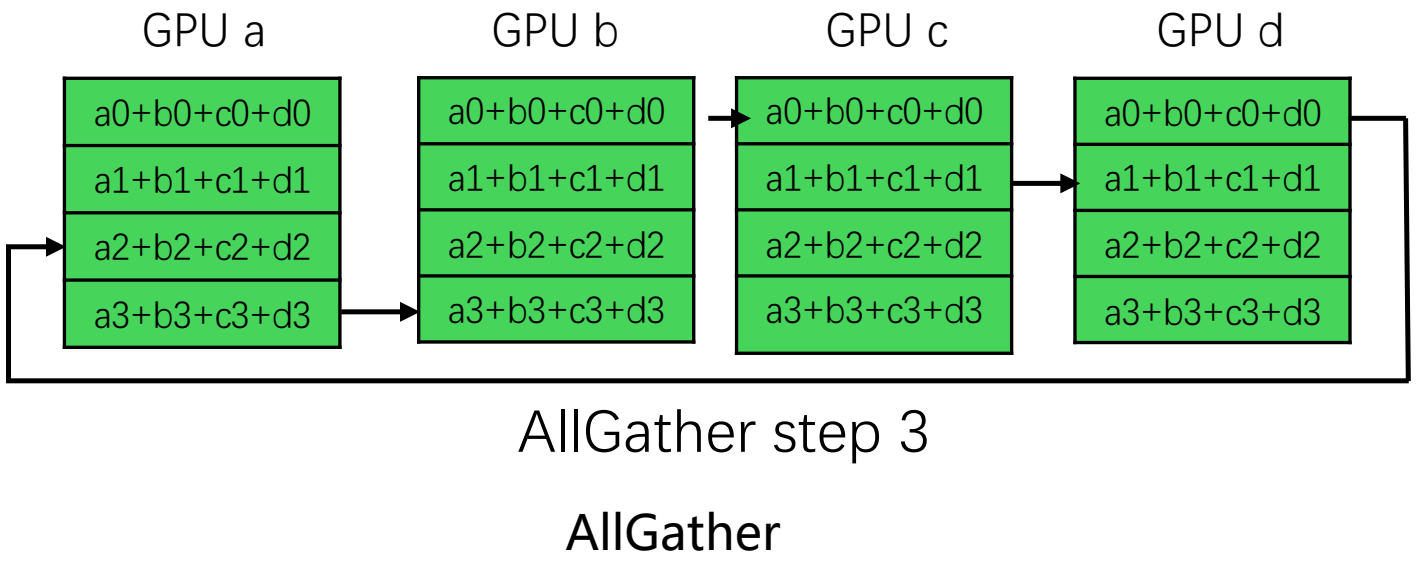
- Ring All-Reduce



All-Gather

Ring All-Reduce

- Ring All-Reduce



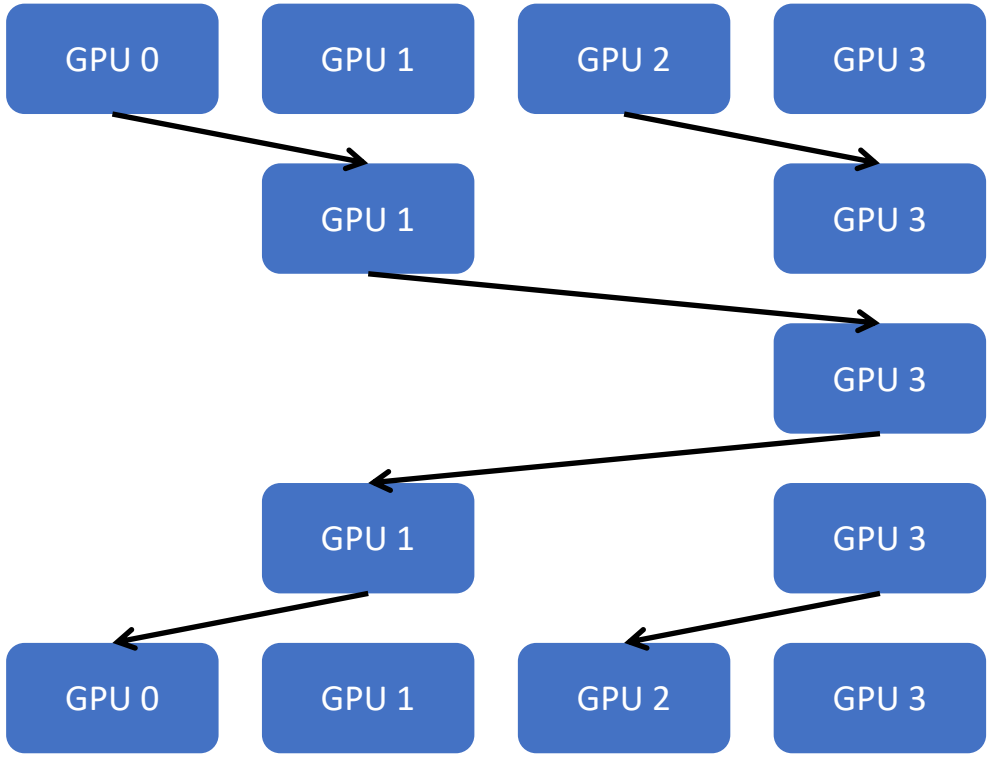
- Reduce-Scatter
  - N-1 rounds
  - S/N data transmission per round
- All-Gather
  - One round
  - (N-1)*S/N
- Total traffic per GPU
  - **2*(N-1)*S/N**

Ring All-Reduce

“All-Reduce = Reduce-Scatter + All-Gather”

(Recursive) Halving Doubling All-Reduce

- Halving Doubling All-Reduce

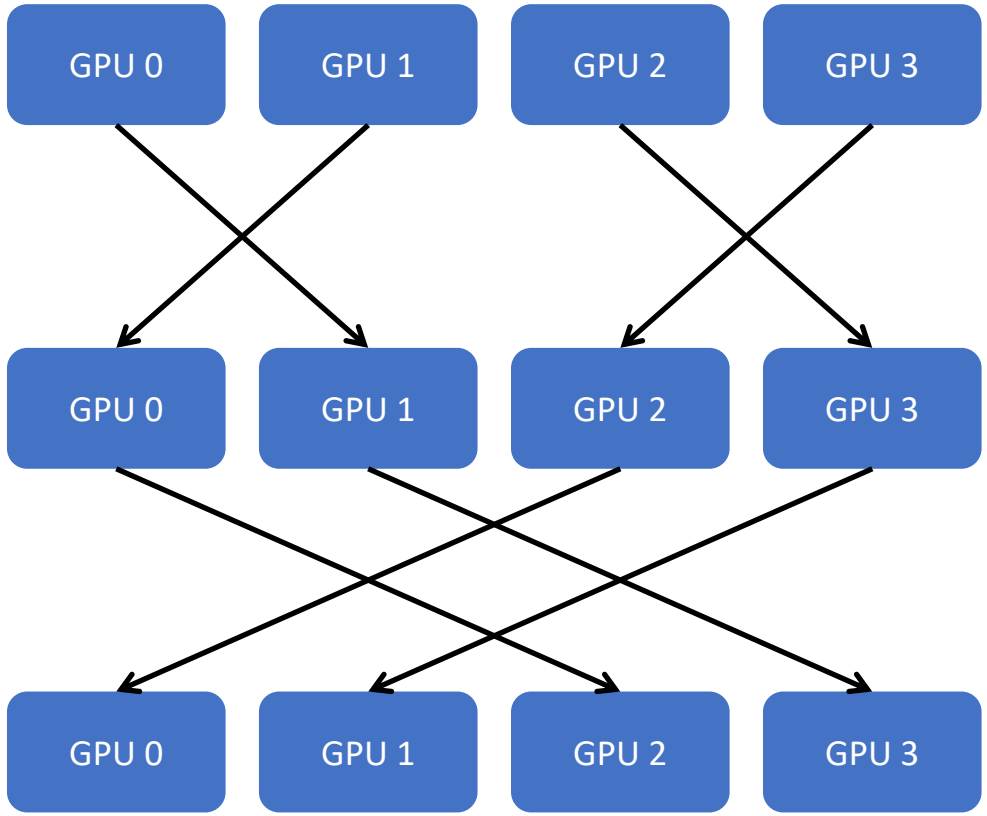


Halving and Doubling

- Halving
  - $\log_2 N$ rounds
  - S data per round
- Doubling
  - $\log_2 N$ rounds
  - S data per round
- Total traffic per GPU
  - $2*S*\log_2 N$

Butterfly All-Reduce

- Butterfly Reduce

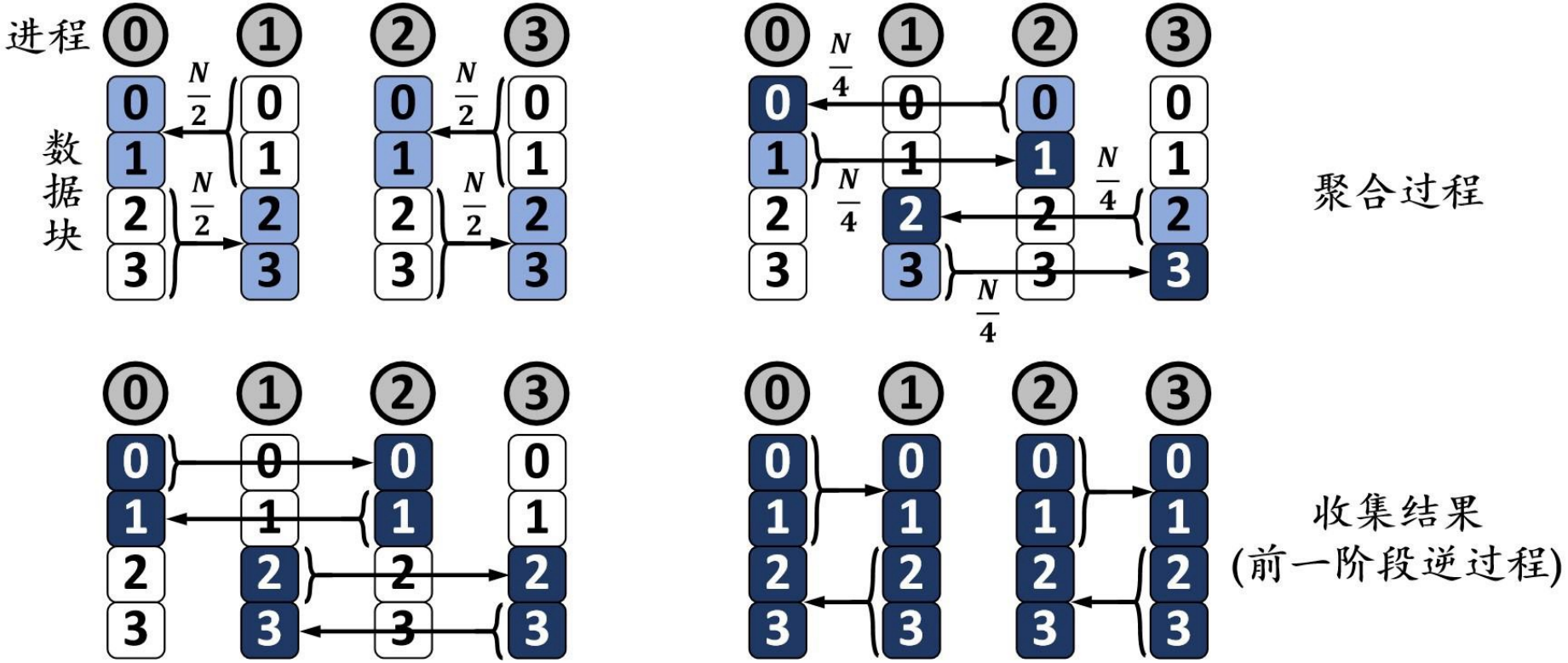


Butterfly Reduce: Utilizing bidirectional bandwidth

- Butterfly Reduce
  - $\log_2 N$ rounds
  - S data per round
- Total Traffic per-GPU
  - $\log_2 N * S$

Rabenseifner All-Reduce

- Rabenseifner Reduce



Rabenseifner Algorithm (an enhanced naïve halving-doubling)

《Allreduce算法及其硬件加速方法介绍》

Cross-Comparison

- Communication Cost
  - Assumption: each GPU can send and receive data simultaneously
  - Classical α-β model: latency = $\alpha + \beta \cdot \frac{S}{B}$

"start-up" delay

transmission efficiency

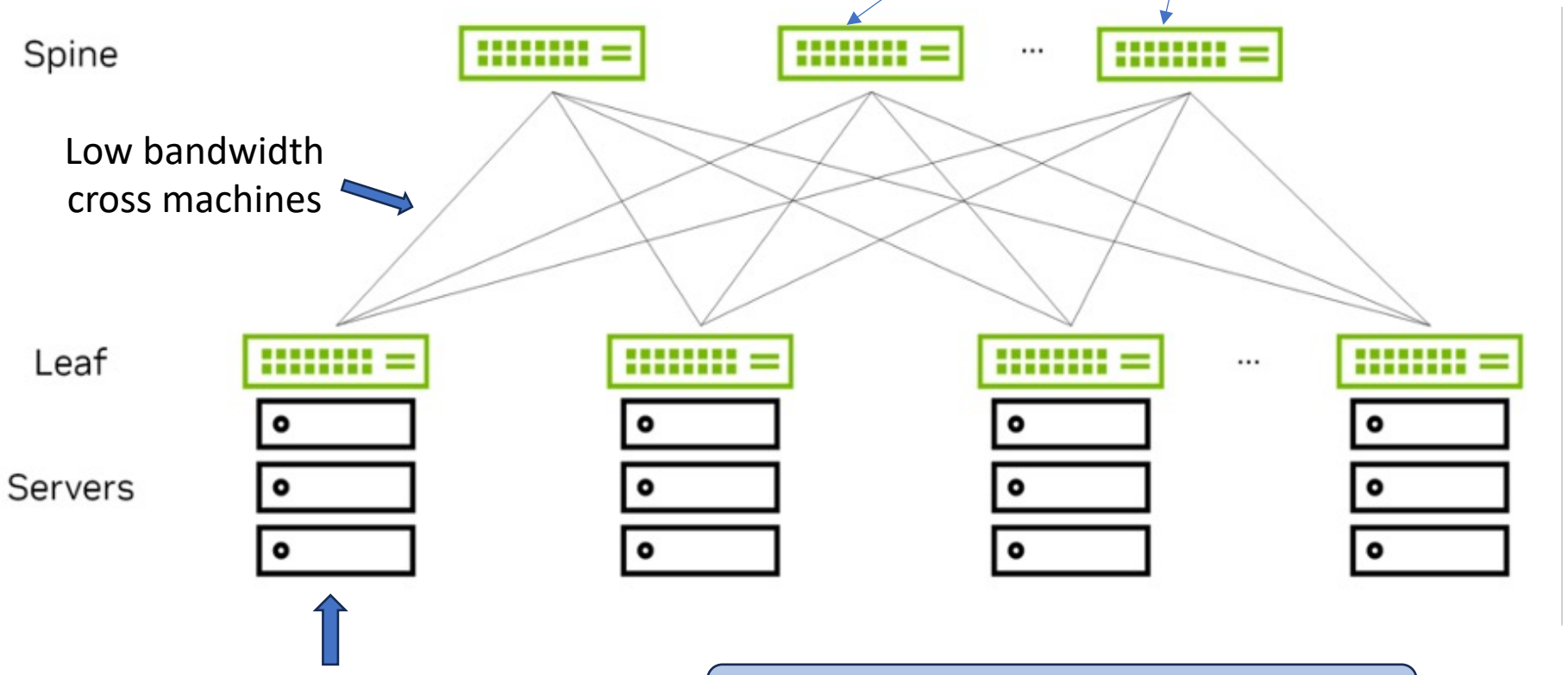
Message size/bandwidth

Algorithm	Rounds	Traffic Load	Cost	Pros and Cons
Ring	2(N-1)	2S(N-1)/S	*2(N-1)*(α + S/B)*	▪ Large data chunk aggregation ▪ Relatively small # of GPUs ▪ Not suitable for short chunks ▪ Ease of implementation ▪ Utilizating bidirectional links
Recursive Halving-Doubling (Rabenseifner)	2 $\lceil \log N \rceil$	2S(N-1)/S	$2\lceil \log N \rceil \alpha + 2(N-1)S/N * \beta$	▪ Relatively smaller data chunk ▪ Large # of GPUs ▪ Periodically changing communication pairs

Challenge of Scaling Out

Network congestion, load balancing

- Two-tier spine-leaf topology



Topology aware CCL

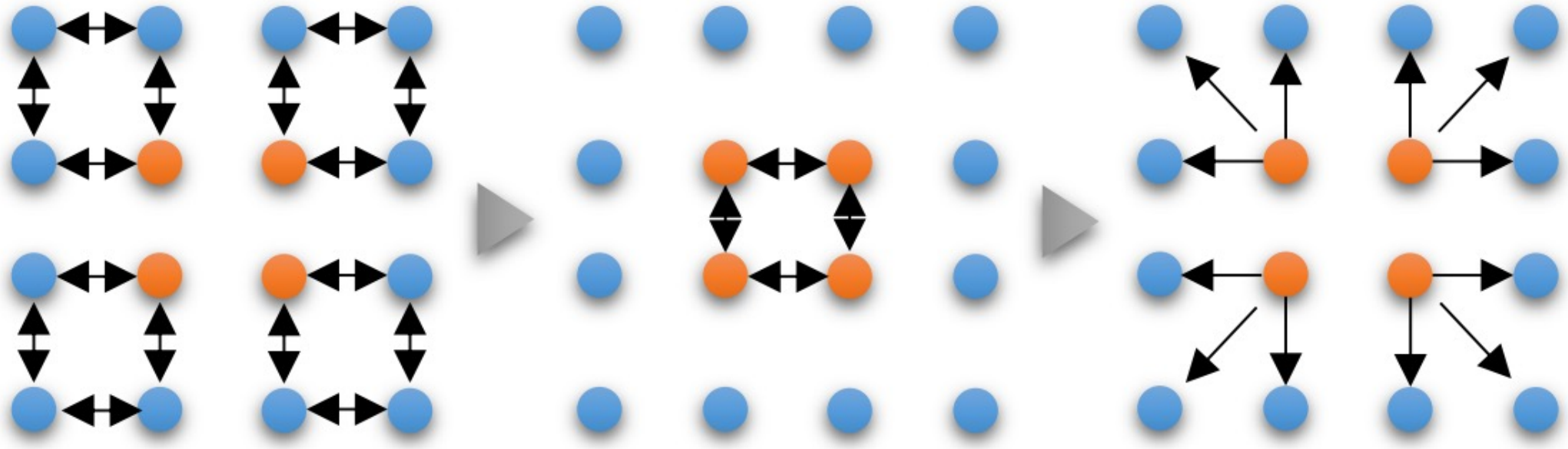
All-Reduce for LLM training

- Each company develops its own collective communication libraries
  - NCCL (NVIDIA CCL)
  - MSCCL (Microsoft CCL)
  - Gloo
  - HCCL (Huawei CCL)
  - ACCL (Alibaba CCL)
  - TCCL (Tencent CCL)
  - Many to be added

Tailored for their own harware, datacenter network topology, etc.

All-Reduce for LLM training

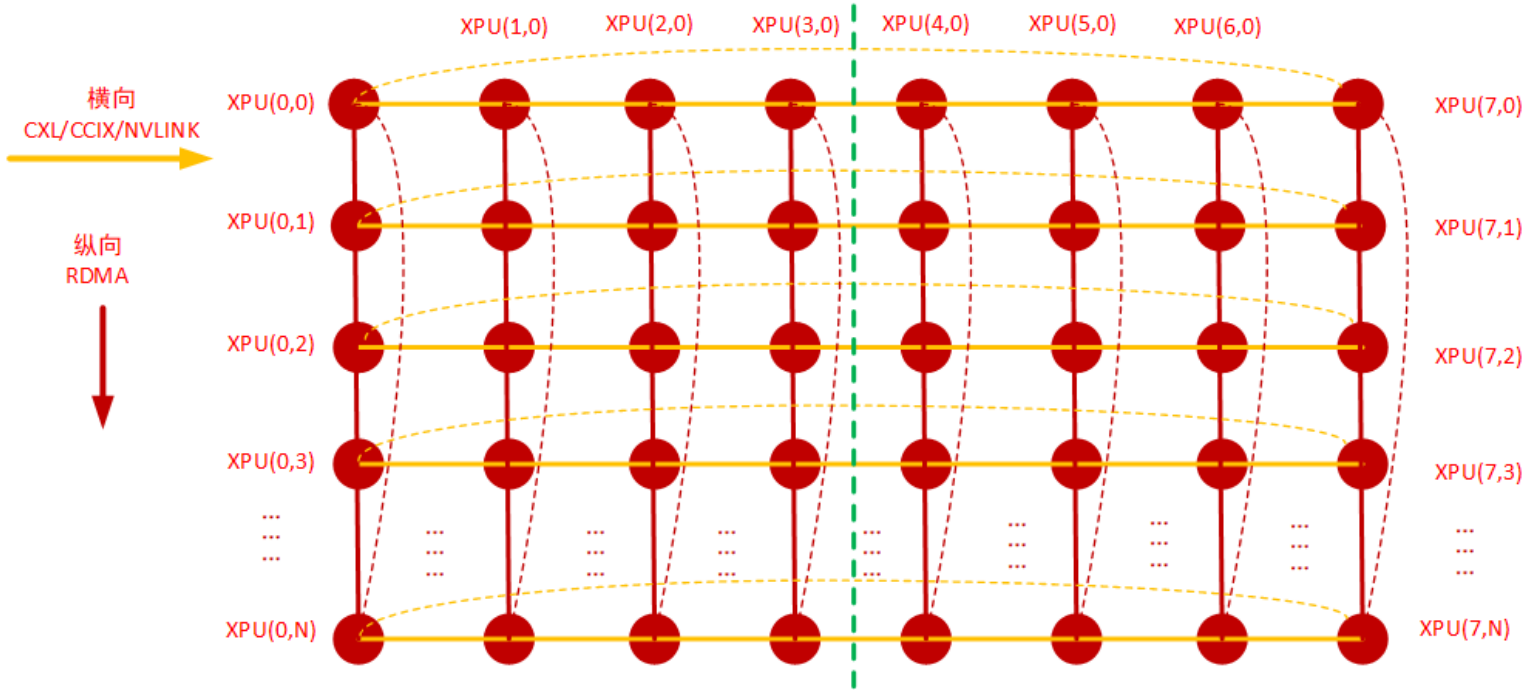
- Hierarchical All-Reduce



《Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes》

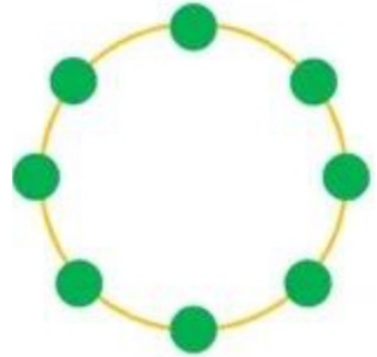
All-Reduce for LLM training

- 2D Torus All-Reduce

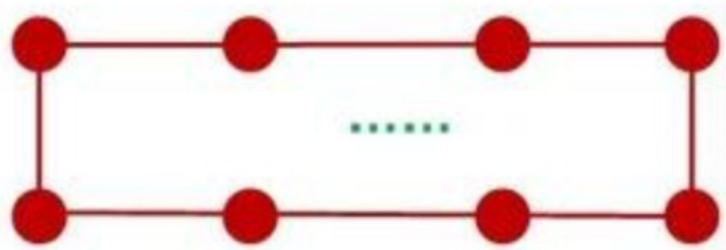


Horizontal: intra-machine GPU interconnect via NVLink/CXL
Vertical: inter-machine GPU interconnect via RDMA NICs

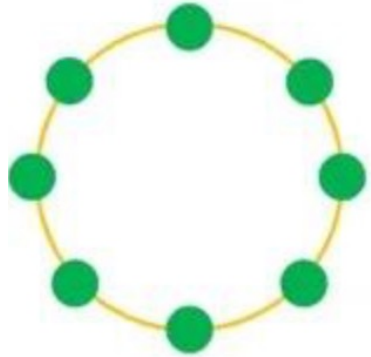
Intra-machine NVLink/CXL; Inter-machine RDMA (≥ 2 NICs)
Designed by SONIC



Step 1: Intra-ring Reduce-Scatter



Step 2: Inter-ring All-Reduce

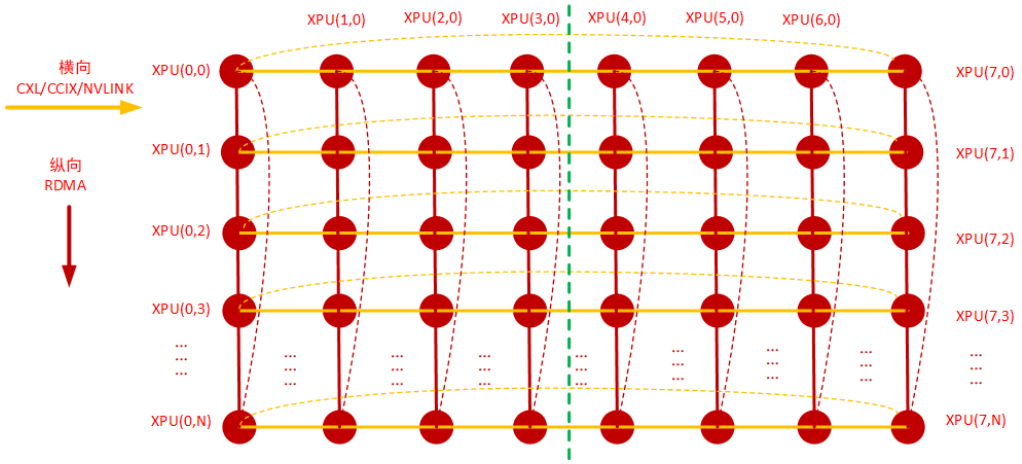


Step 3: Intra-ring All-Gather

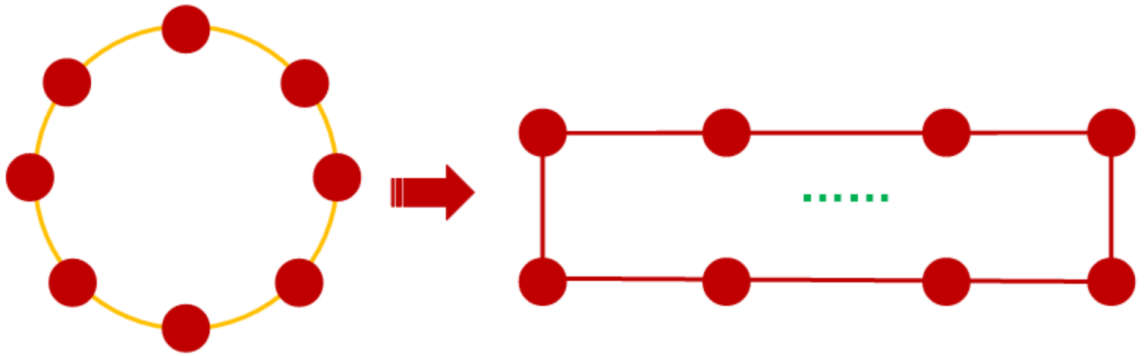
《分布式训练常用的网络结构及集合通信拓扑算法》知乎

All-Reduce for LLM training

- 2D Mesh All-Reduce



2D Mesh: removing links connecting the first and the last GPUs of each row and column



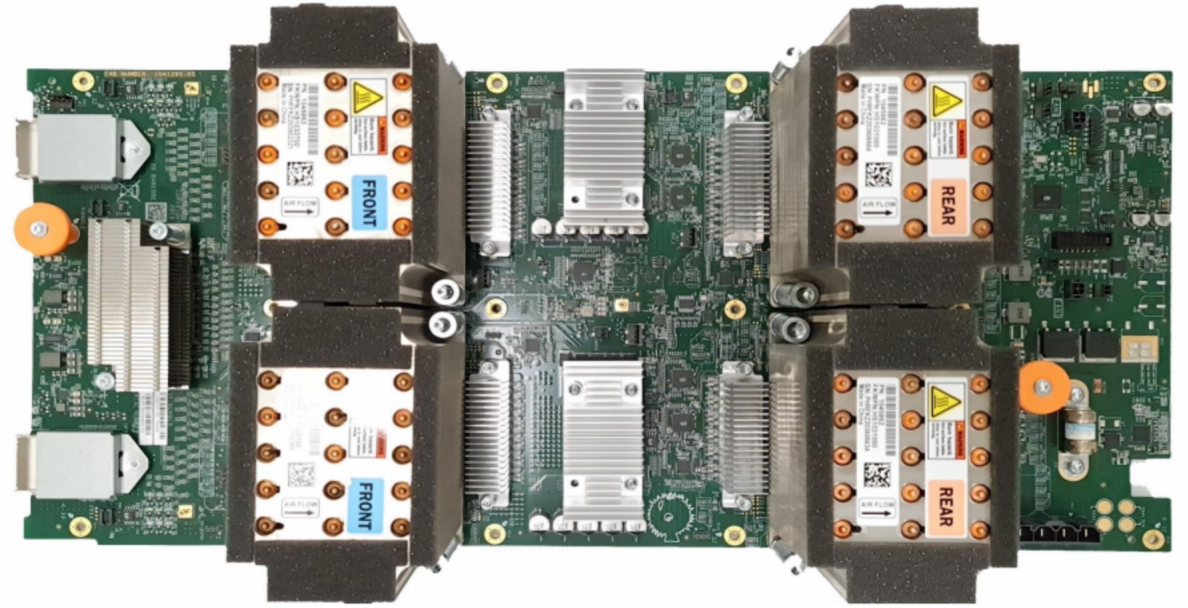
Step 1: Intra-Ring All-Reduce

Step 2: Inter-Ring All-Reduce

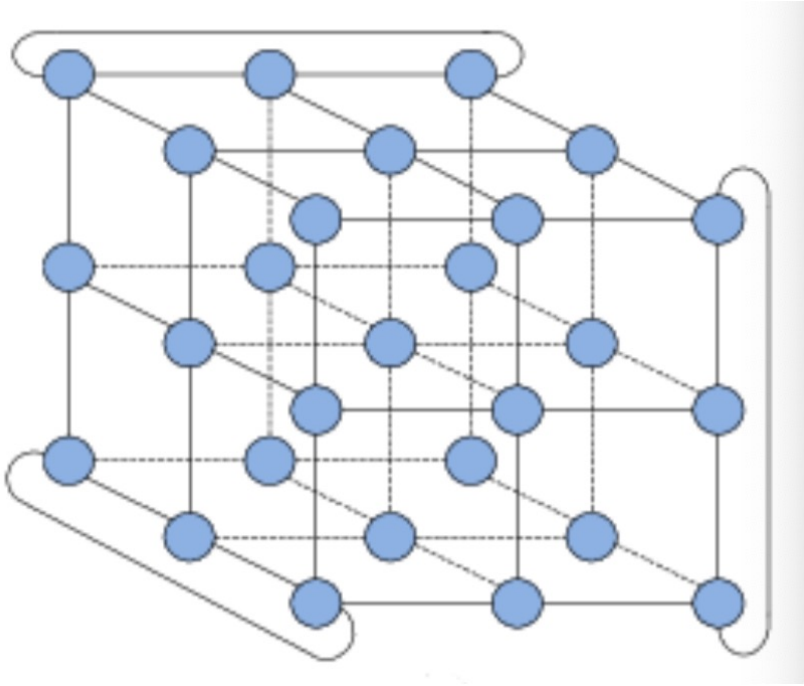
《分布式训练常用的网络结构及集合通信拓扑算法》知乎

All-Reduce for LLM training

- 3D Torus All-Reduce



TPUv4i board with 4 chips that are connected by ICI.



3D Torus (3-ary 3-cube)
Initially designed by IBM

5D, 6D Torus in HPC areas

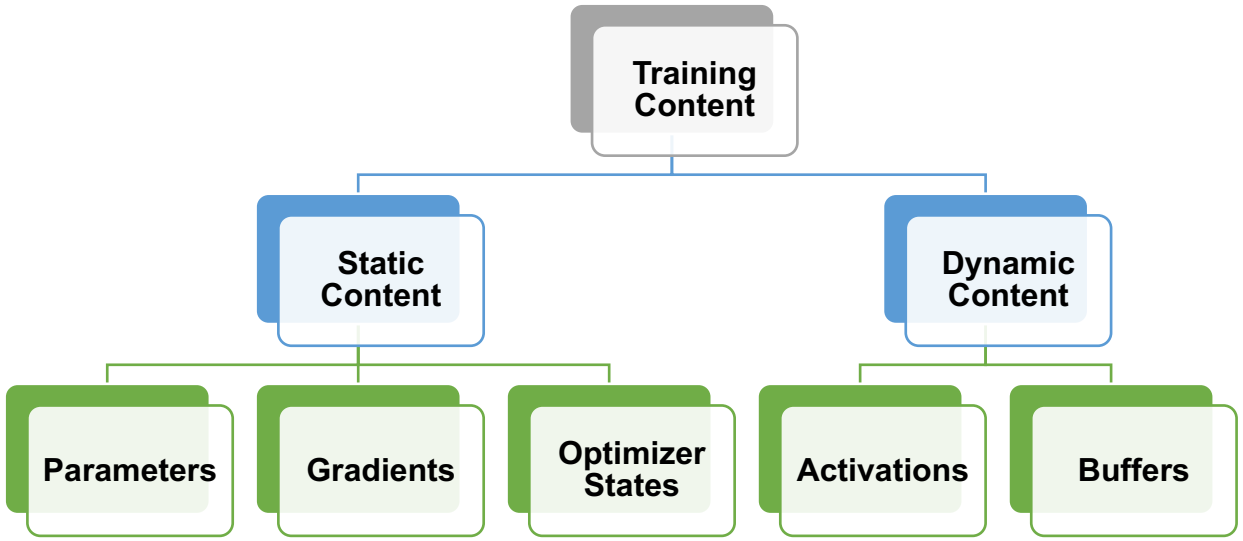
《TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings》

Distributed LLM Training: Outline

- Data Parallelism
  - Parameter-Server
  - All-Reduce
  - **Memory optimization**
- Model Parallelism
  - Pipeline Parallelism
  - Tensor Parallelism
  - Sequence Parallelism
- Mixture of Experts

LLM training: DP with Memory Optimization

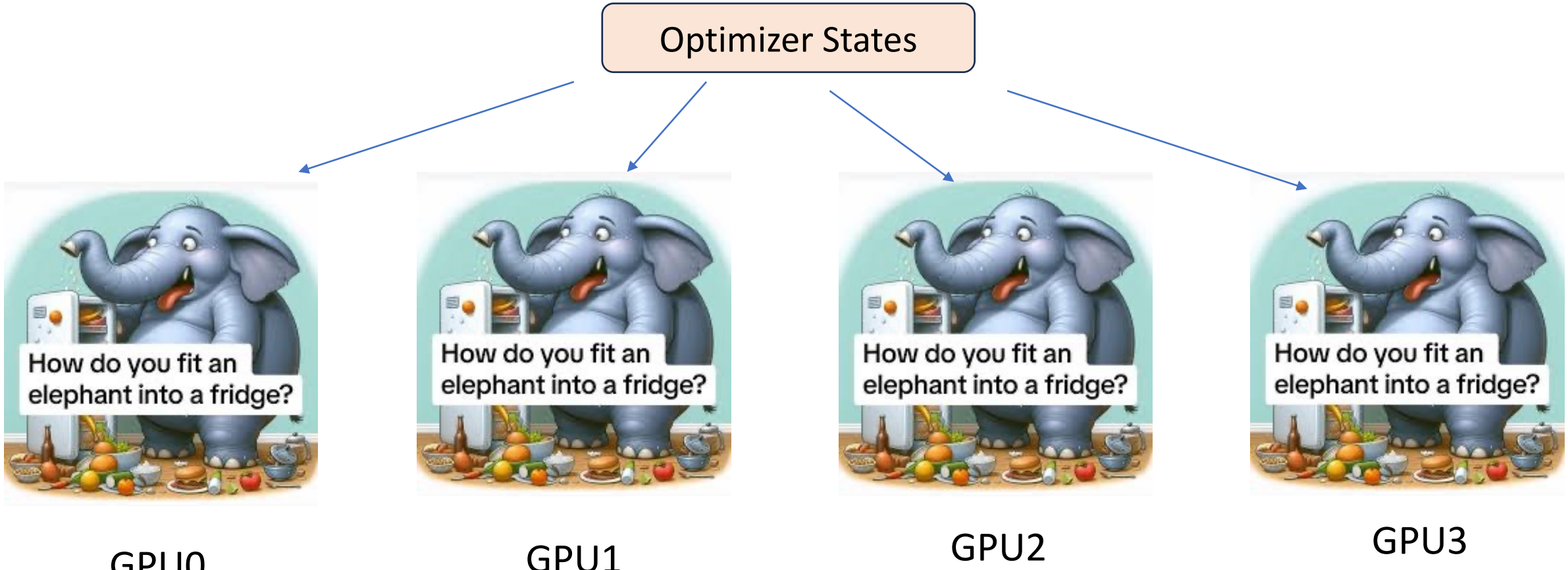
- Retrospect: GPU HBM Content During Training



- An example of GPT-3 175B
  - **Optimizer States**
    - 32-bit Parameter (700 GB)
    - Adam Moment (700 GB)
    - Adam Variance (700 GB)
  - 16-bit Parameter (350 GB)
  - 16-bit Gradient (350 GB)
  - Activations (depending on batch size)
  - Buffer and Fragmentation

LLM training: DP with Memory Optimization

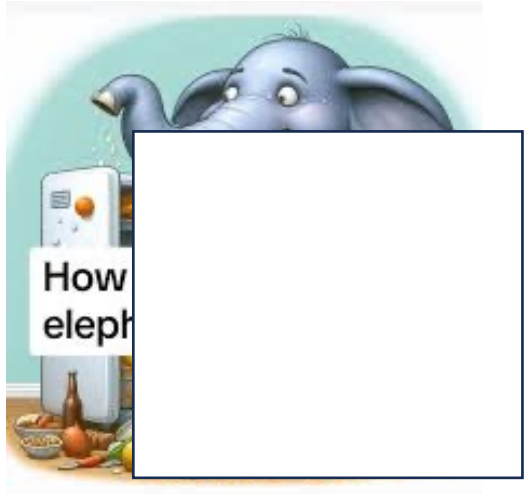
- How to put an elephant into a fridge?



LLM training: DP with Memory Optimization

- How to put an elephant into a fridge?

Sharding the elephant into four partitions, and each GPU HBM holds one chunk!



GPU0

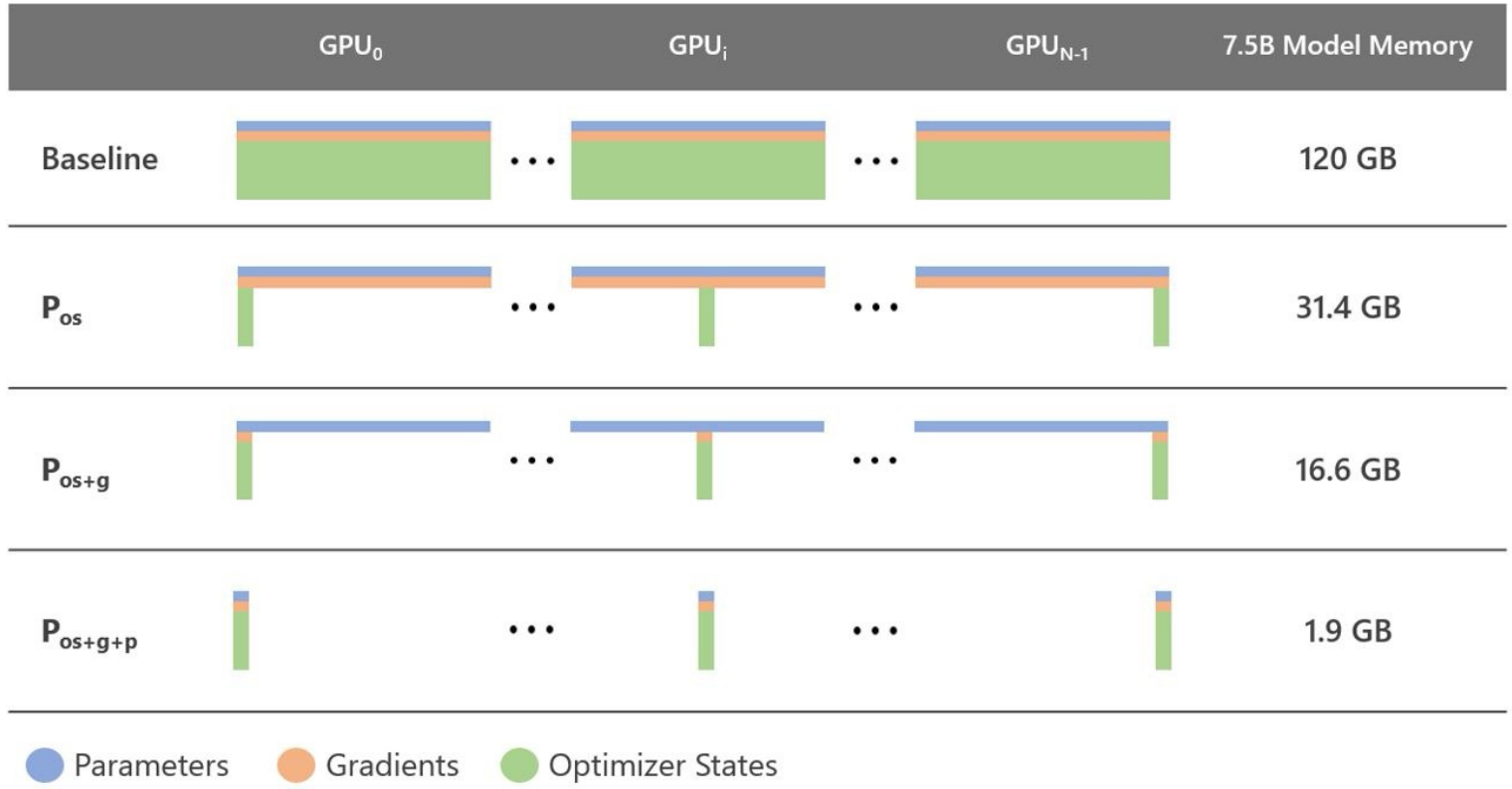
GPU1

GPU2

GPU3

LLM training: DP with Memory Optimization

- ZeRO (Zero Redundancy) optimization: an overview



HBM usage with DP degree 64

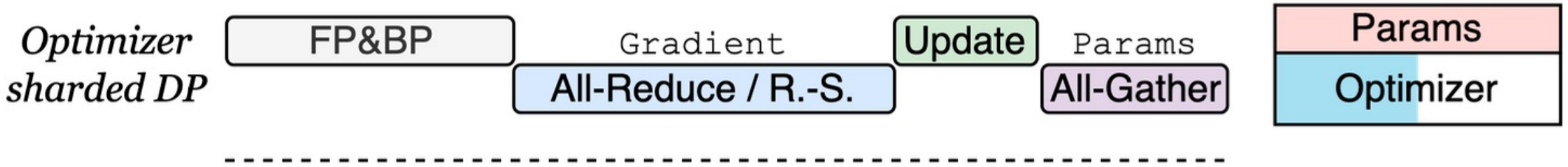
《Fit More and Train Faster With ZeRO via DeepSpeed and FairScale》

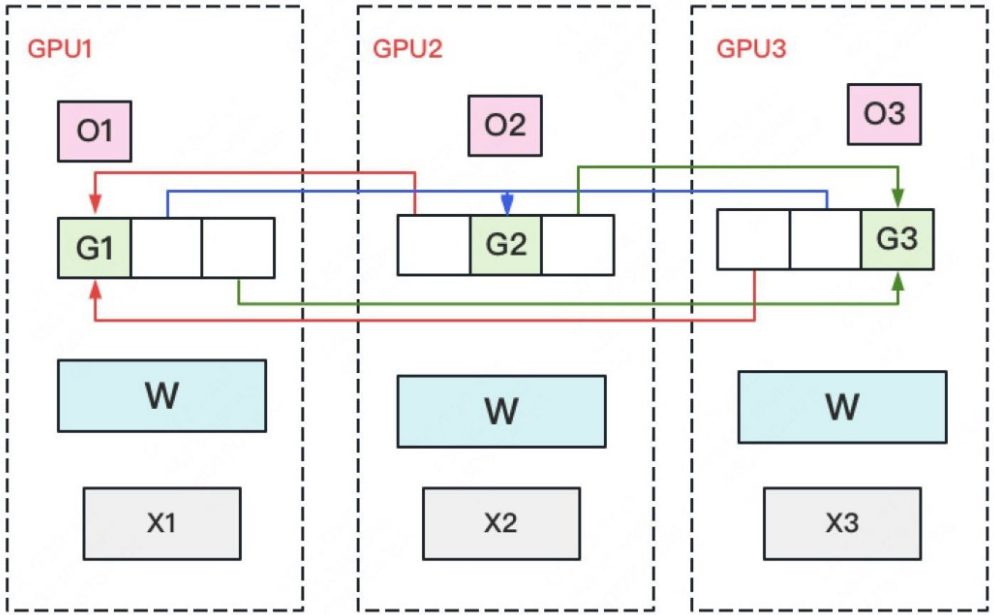
LLM training: DP with Memory Optimization

- ZeRO-1&2
  - *Forward* is OK because each GPU holds the complete *parameter* ✓
  - *Backward* is OK because of the same reason ✓
  - *AllReduce* = ReduceScatter + AllGather
    - *ReduceScatter is OK* ✓
    - *Update optimizer state shards* ✓
      - *Incomplete optimizer states at every GPU*
    - *Update parameter shards* ✓
    - *AllGather remaining parameter shards to assembly the complete parameter* ✓

LLM training: DP with Memory Optimization

- ZeRO-2 (All-Reduced is optimized by Reduce-Scatter+All-Gather)





- Reduce-Scatter
  - Green shards are global **gradients** after reduce-scatter
- Update OS and Param.
- All-Gather
  - Each GPU acquires complete model **parameters**

LLM training: DP with Memory Optimization

- ZeRO-3 (parameter, gradient and optimizer sharding)
  - *Forward* is ***NOT*** OK because of incomplete *parameter*
    - *Need to fetch parameter shards from other GPUs via BROADCAST*
  - *Backward* is ***NOT*** OK because of incomplete parameter
    - *Need to fetch parameter shards from other GPUs via BROADCAST again*
  - *Aggregating local gradient shards to obtain global gradient shards*
    - *Reduce Scatter is OK*
  - *Updating optimizer shards and parameter shards are OK*
    - *No "AllGather" operation afterwards*

LLM training: DP with Memory Optimization

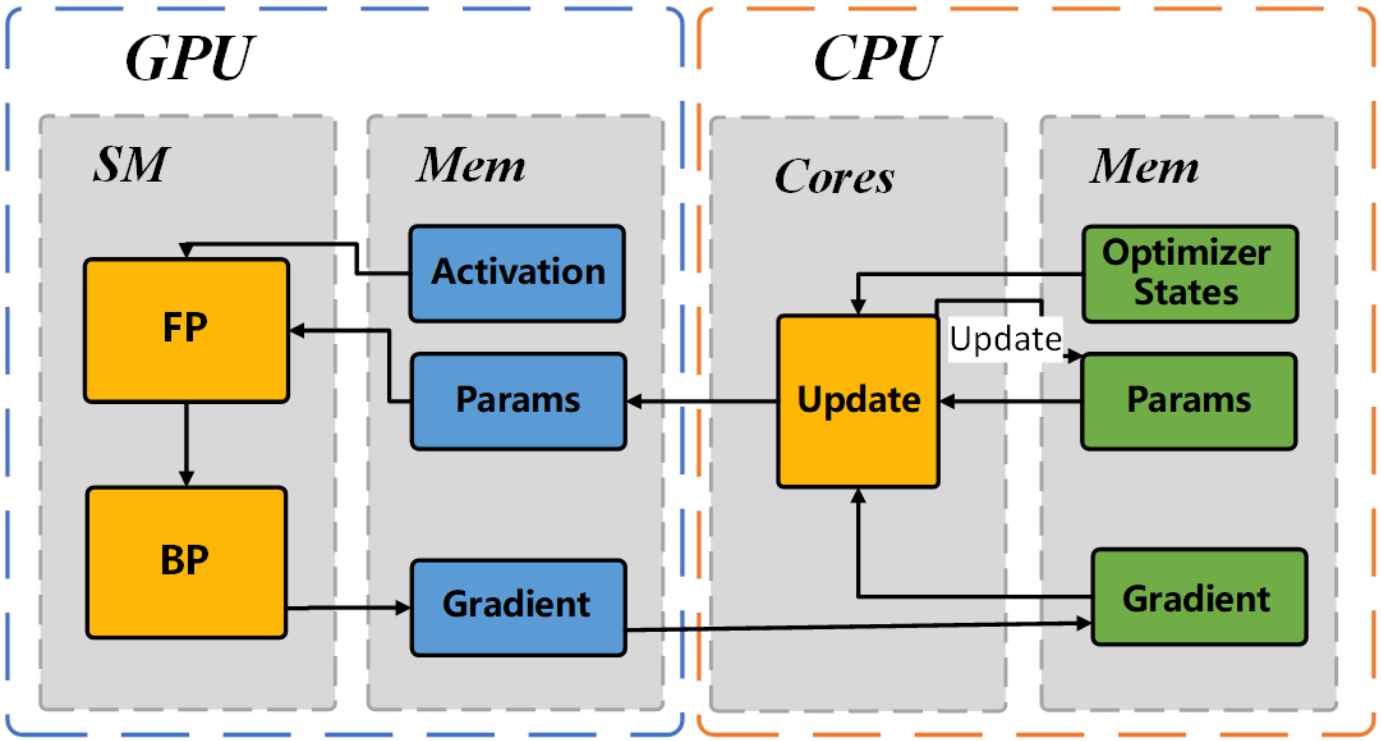
- ZeRO-3: An animation (without optimizer and parameter updates)

- All-Gather
  - Collecting parameters for forward propagation
- All-Gather
  - Collecting parameters for back propagation
- Reduce-Scatter
  - Obtaining global gradient
- Update OS and param.

Source: microsoft official site

LLM training: DP with Memory Optimization

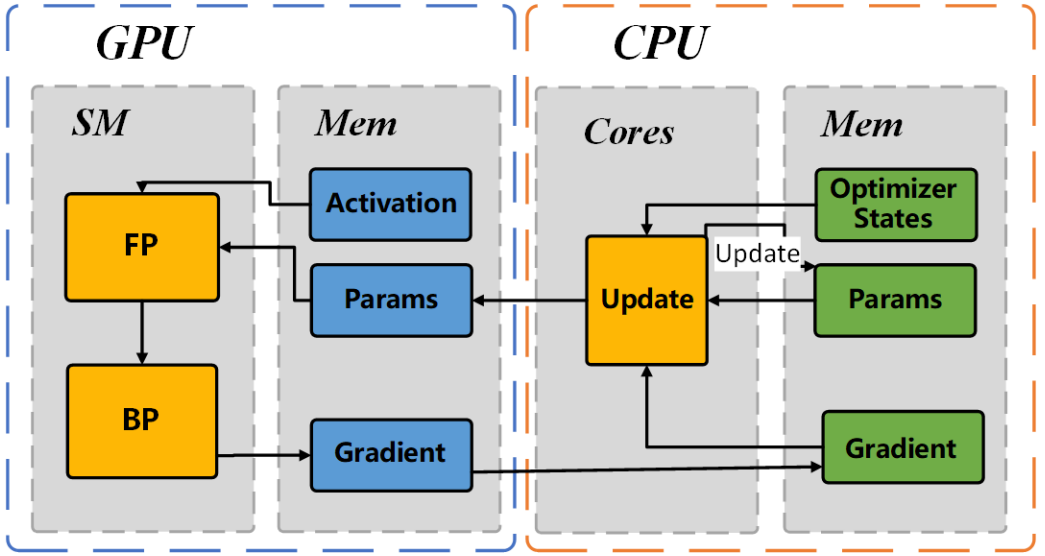
- ZeRO Offload



- Store OS in CPU memory
  - ~1TB vs 80GB HBM
- Connection speed
  - PCIe 5.0 (16 lanes) ~ 63GB/s
  - NVIDIA H100 NVLink 4.0 ~ bidirectional 900GB/s
- GPU←→CPU bottleneck

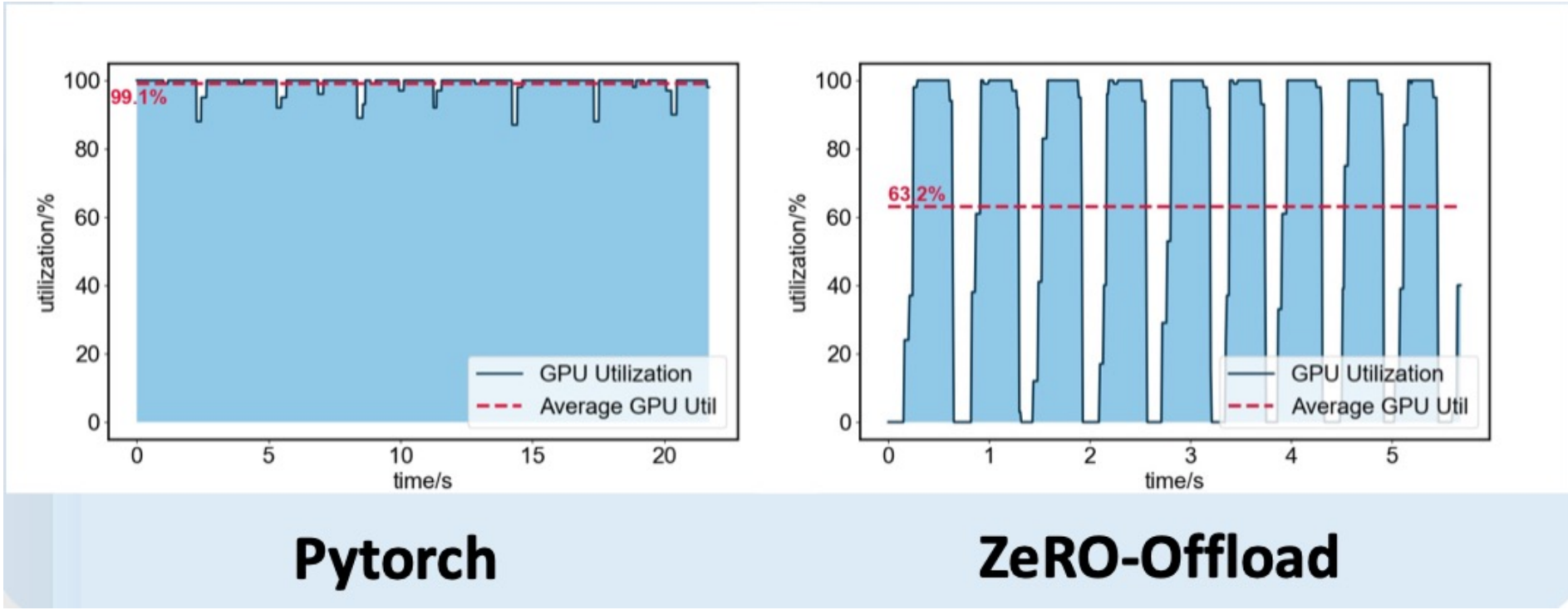
LLM training: DP with Memory Optimization

- ZeRO Offload: workflow
  - Forward propagation at **GPU HBM**
  - Backward propagation at **GPU HBM**
  - Optimizer state is at CPU main memory
  - Transmit gradient to from GPU to CPU
  - Update optimizer states at CPU memory (which is relatively slow)
  - Update parameter at CPU memory
  - Transmit new parameter from CPU to GPU, and repeat



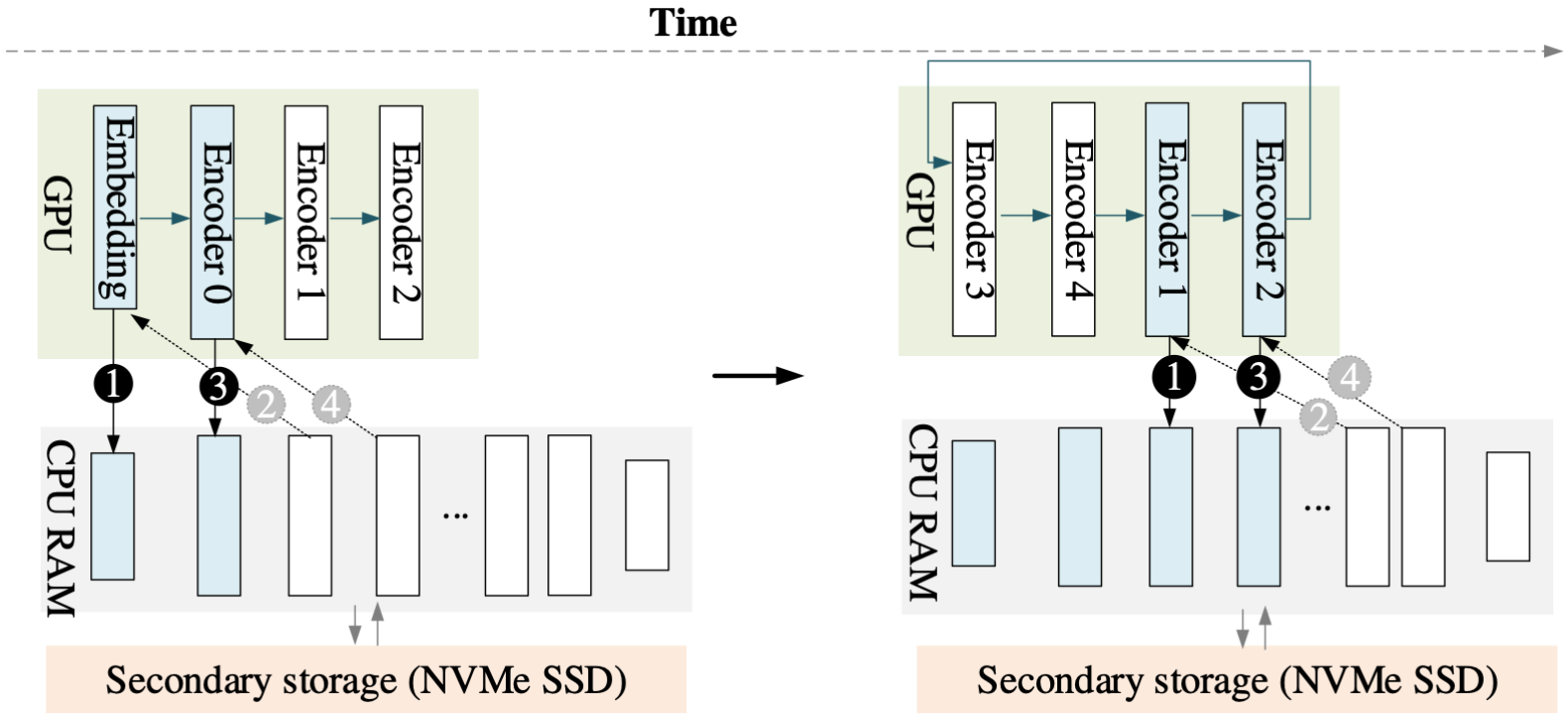
LLM training: DP with Memory Optimization

- Pros and Cons
  - Training GPT2 on A100: reduced GPU utilization



LLM training: DP with Memory Optimization

- ZeRO Offload Variants

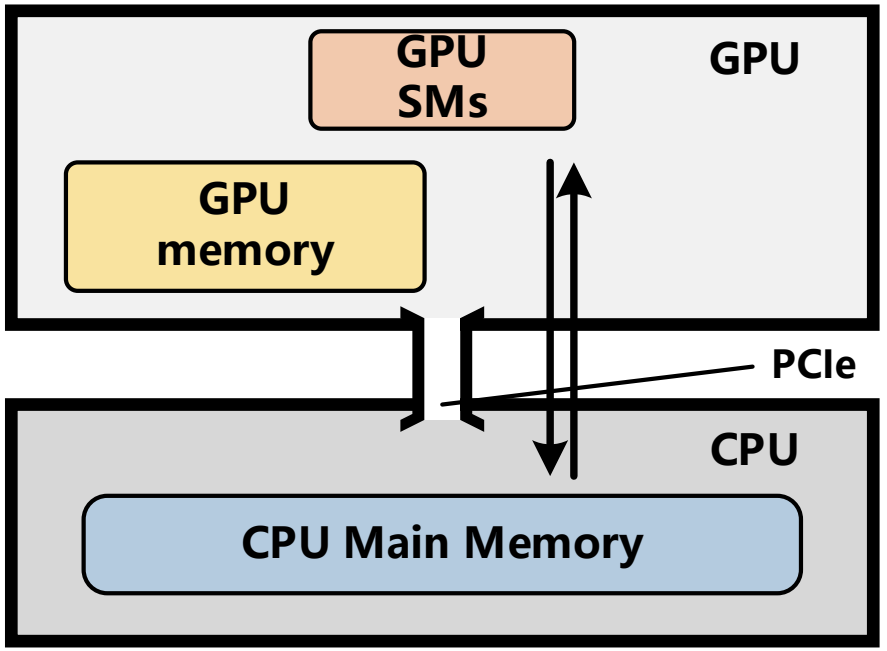


STRONGHOLD stores some DNN layers in the GPU memory and swapping out the finished layer states to the CPU RAM.

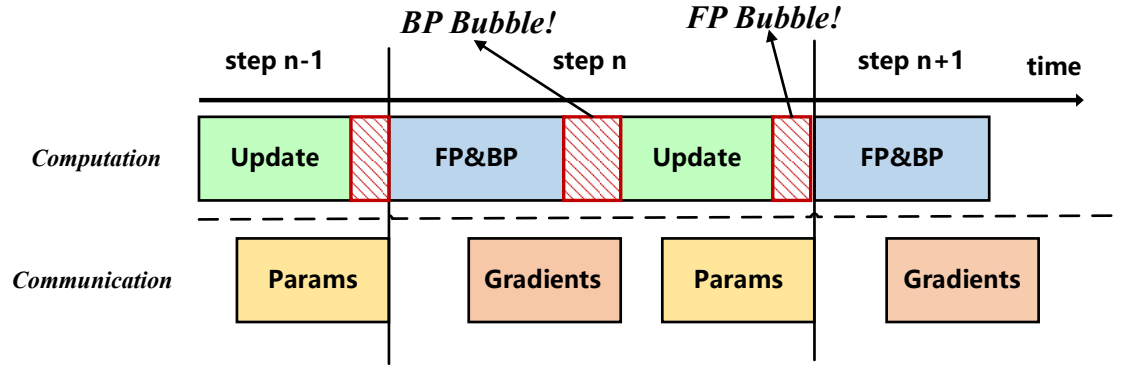
《STRONGHOLD: Fast and Affordable Billion-Scale Deep Learning Model Training》

LLM training: DP with Memory Optimization

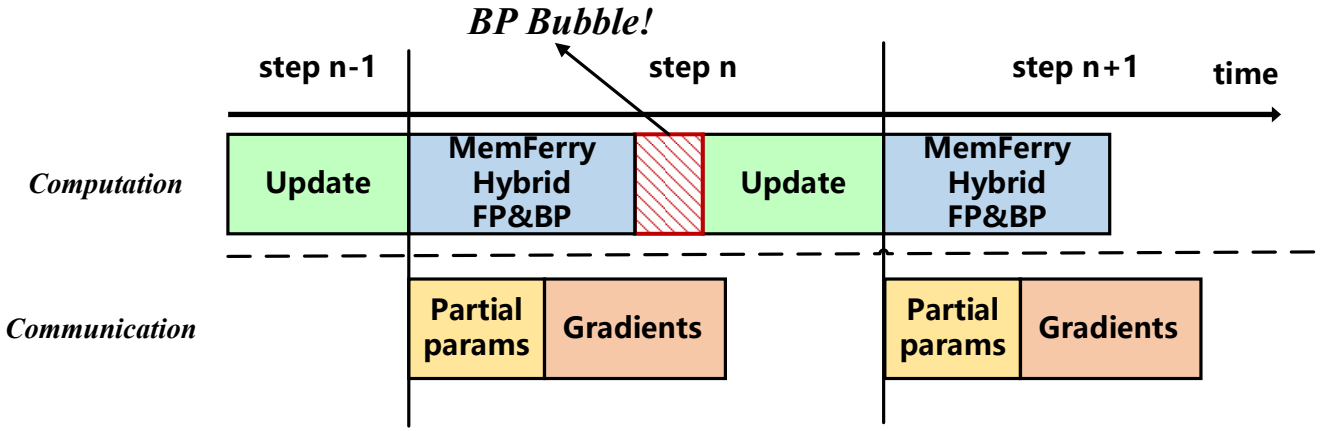
- ZeRO Offload Variants



DHA mode: GPU SMs compute tensors stored in CPU memory



Eliminating bubbles

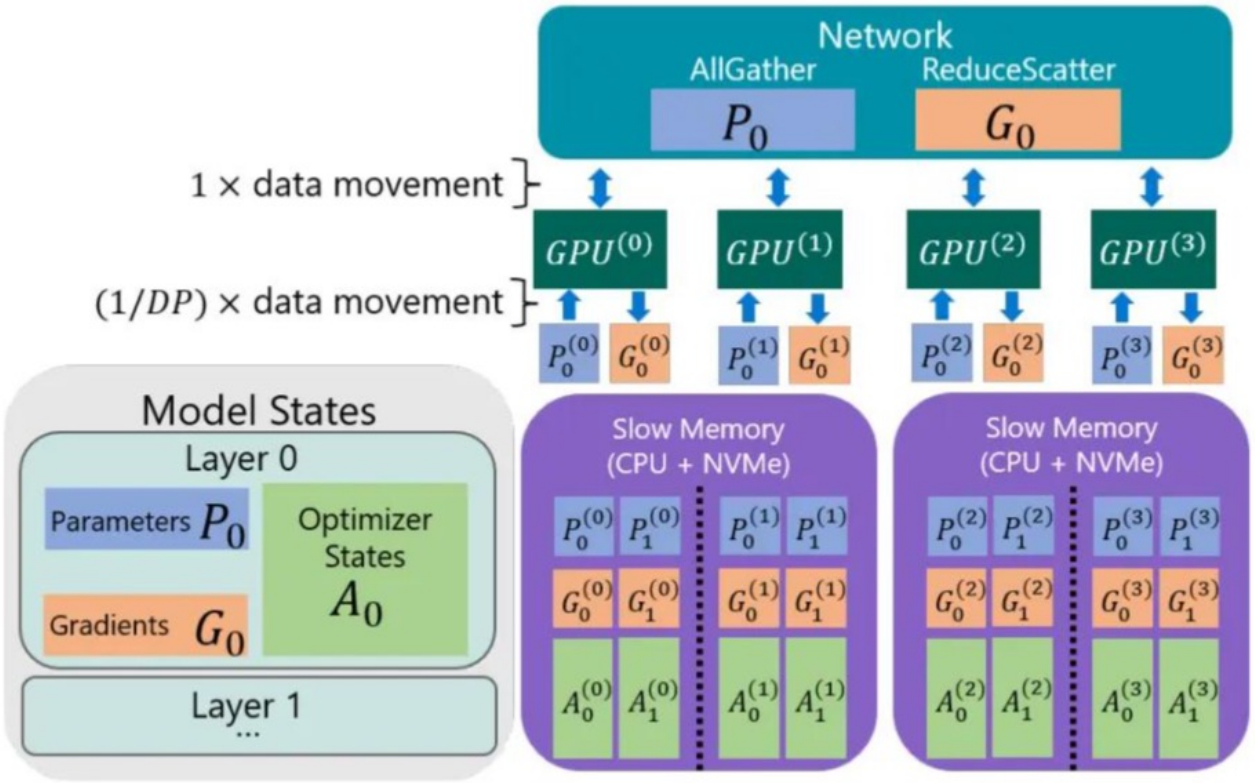


Eliminating bubbles

《MemFerry: A Fast and Memory Efficient Offload Training Framework with Hybrid GPU Computation》

LLM training: DP with Memory Optimization

- ZeRO Infinity

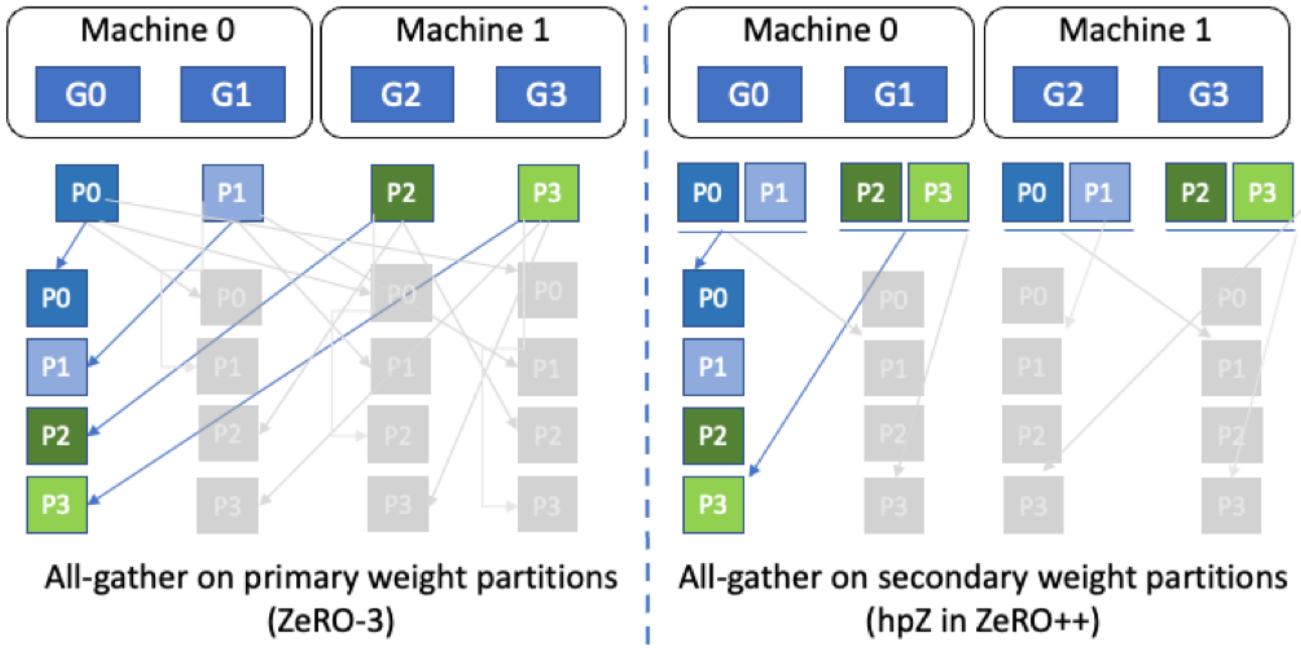


- Bandwidth
  - GPU > CPU > NVMe
- Read Data
  - Read parameter using All-Gather
- Data Path
  - CPU → GPU
  - NVMe → CPU → GPU

《ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning》

LLM training: DP with Memory Optimization

- ZeRO++: Low-bandwidth Scenario



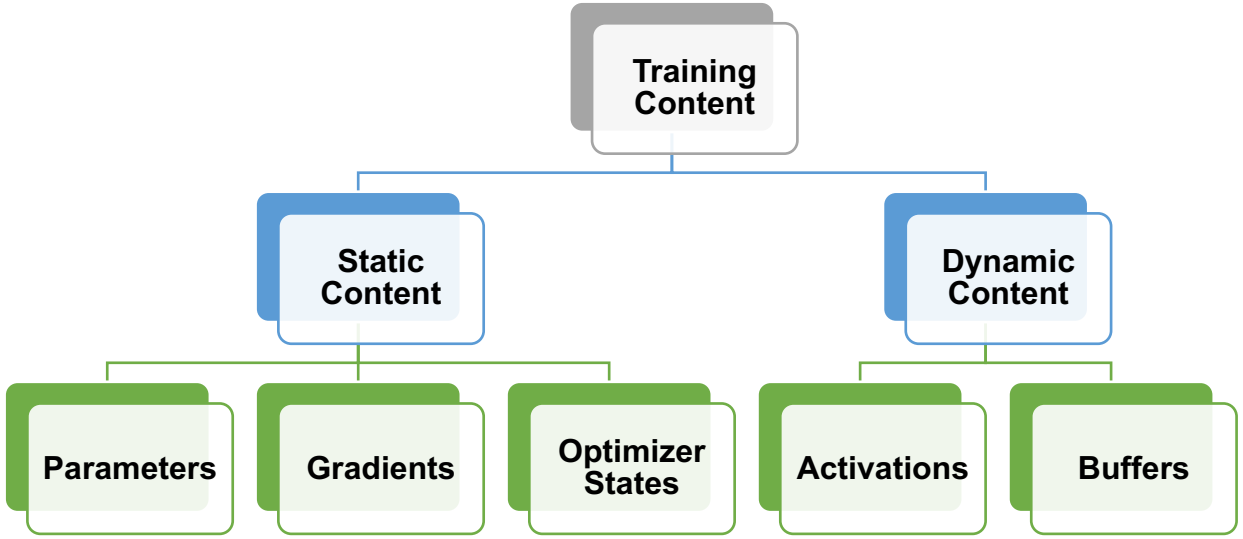
Hierarchical partitioning (not diving into details)

- Quantized Weight Communication
  - FP16 weight -> INT8 weight before All-Gather
  - block-quantization based all-gather in FP
  - INT8 -> FP16 after All-Gather
- Hierarchical Partitioning
  - (trying to) eliminate the inter-node all-gather
- Quantized Gradients Communication
  - Standard Reduce-Scatter involves a lot of quantization and dequantization steps
  - leverages all-to-all collectives to implement quantized reduce- scatter

《ZeRO++: Extremely Efficient Collective Communication for Giant Model Training 》

LLM training: DP with Memory Optimization

- Retrospect: GPU HBM Content During Training



- An example of GPT-3 175B
  - Optimizer States
    - 32-bit Parameter (700 GB)
    - Adam Moment (700 GB)
    - Adam Variance (700 GB)
  - 16-bit Parameter (350 GB)
  - 16-bit Gradient (350 GB)
  - **Activations (depending on batch size)**
  - Buffer and Fragmentation

LLM training: DP with Memory Optimization

- Activation is non-negligible
  - Forward propagation
  - Backward propagation
  - Size of activation
    - Standard transformer: b - batch size, s - sequence length, h - hidden layer dimension

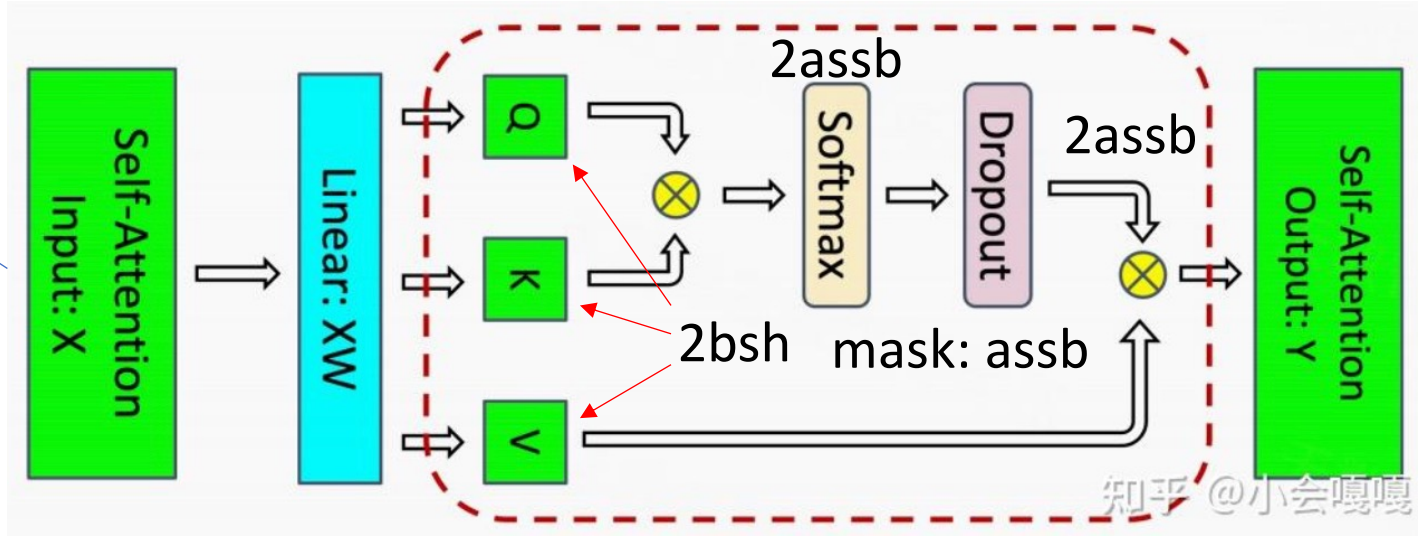
Output of a previous layer, intermediate variable

$$y = Wx + b$$

$$\frac{dL}{dW} = \frac{dL}{dy}x$$

Cannot be discarded after FP

bsh * 2 Bytes



In-total: 5abss + 8sbh

LLM training: DP with Memory Optimization

- Activation is non-negligible
  - Forward propagation
  - Backward propagation
  - Size of activation
    - Standard transformer: b - batch size, s - sequence length, h - hidden layer dimension

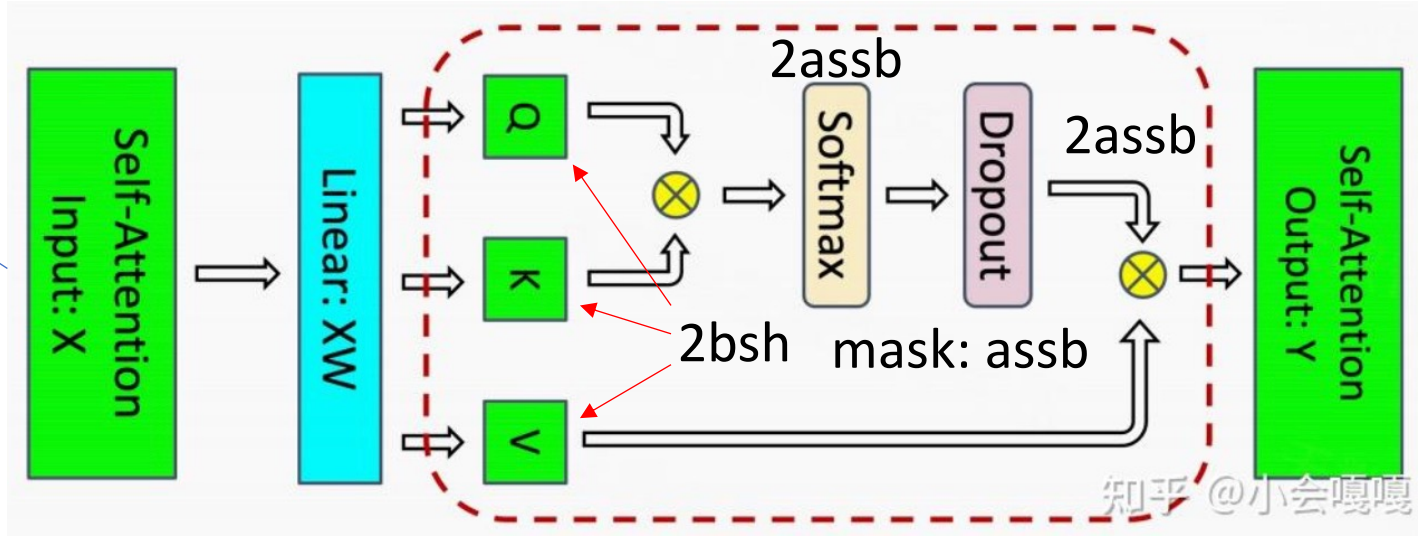
Output of a previous layer, intermediate variable

$$y = Wx + b$$

$$\frac{dL}{dW} = \frac{dL}{dy}x$$

Cannot be discarded after FP

bsh * 2 Bytes

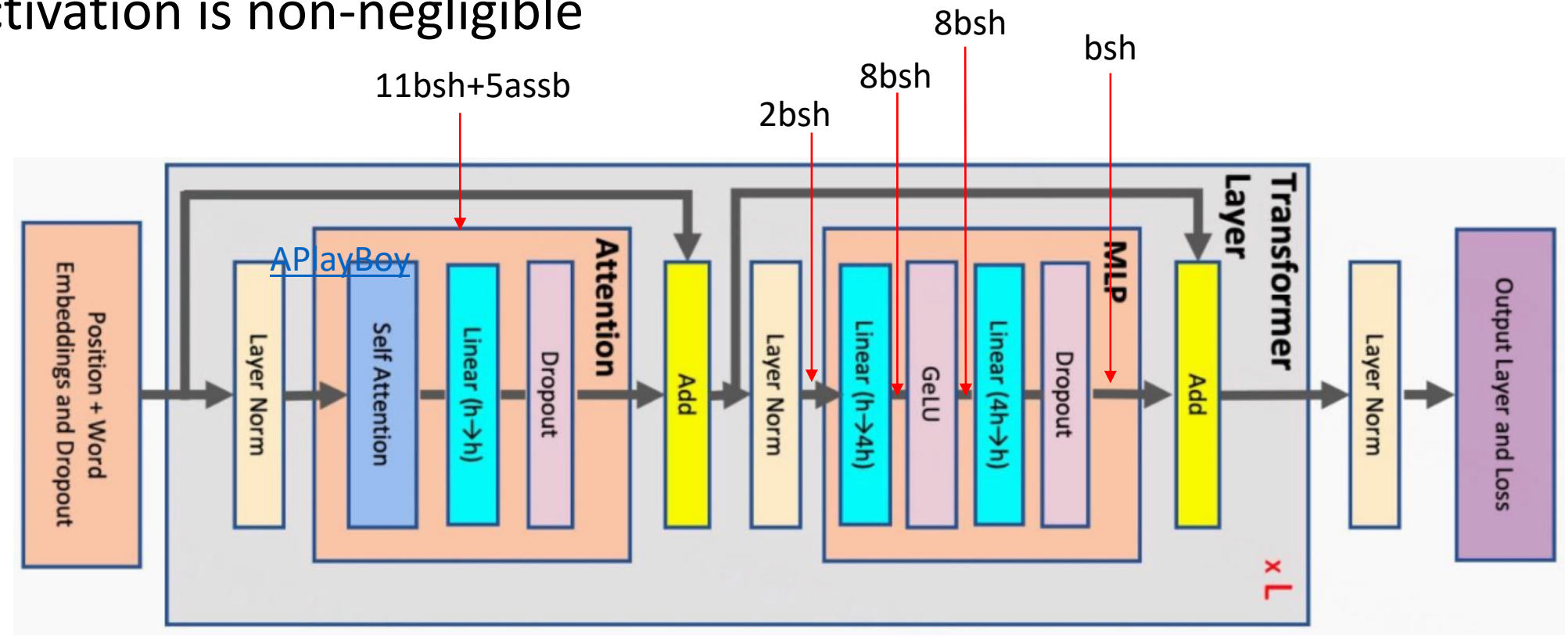


3sbh: Linear layer + dropout layer

In-total: 5abss + 11sbh

LLM training: DP with Memory Optimization

- Activation is non-negligible



In-total: 5abss + 34sbh

From Zhihu user (APlayBoy) and 《 Reducing Activation Recomputation in LLMs 》

LLM training: DP with Memory Optimization

- Activation is non-negligible
  - b: 3.2M tokens/sequence length, s: 2048 tokens, h: 12288, l: 96 layers
  - Activation size ?
  - Full activation re-computation: only keeping the initial input and recompute everything
    - Minimal memory occupation
    - Prolonging training time (doing forward propagation once again) by 30%~40%
  - Goal of strategic recomputation (not the scope of this class)
    - Significantly reducing memory occupation while slightly increasing training time
    - Softmax and dropout are more suitable to be re-computed

https://lambda.ai/blog/demystifying-gpt-3 《Reducing Activation Recomputation in LLMs》

Thanks!

Example Codes

- Training BERT model with distributed data parallel via Ring AllReduce

```
import os
import socket
import argparse

import torch
import torch.distributed as dist
import torch.multiprocessing as mp
from torch import nn
from torch.optim import Adam
from torch.utils.data import DataLoader, DistributedSampler, Dataset
from transformers import BertConfig, BertModel  # 作为示例的简化 BERT 结构

# 简易数据集，返回 input_ids、attention_mask、labels
class FakeBertLikeDataset(Dataset):
    def __init__(self, seq_len=128, vocab_size=30522, length=10000):
        self.seq_len = seq_len
        self.vocab_size = vocab_size
        self.length = length

    def __len__(self):
        return self.length
```

Example Codes

- Training BERT model with distributed data parallel via Ring AllReduce

```
    def __getitem__(self, idx):
        input_ids = torch.randint(0, self.vocab_size, (self.seq_len,), dtype=torch.long)
        attention_mask = torch.ones(self.seq_len, dtype=torch.long)
        # 简单的"标签": 下一个 token 的分类任务, 或假标签
        label = torch.tensor(0, dtype=torch.long)
        return input_ids, attention_mask, label

def setup_distributed(rank, world_size, backend='nccl', port=12355):
    # 通过 tcp rendezvous 初始化进程组
    hostname = socket.gethostname()
    init_method = f'tcp://{hostname}:{port}'
    dist.init_process_group(backend=backend, init_method=init_method,
                            rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()
```

Example Codes

- Training BERT model with distributed data parallel via Ring AllReduce

```python
# 一个简化的 BERT-like 模型：仅含一个 transformer encoder block 的堆叠
class SimpleBertLike(nn.Module):
    def __init__(self, vocab_size=30522, hidden_size=256, num_heads=4, seq_length=128, num_layers=2)
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.position = nn.Parameter(torch.zeros(1, seq_length, hidden_size))
        encoder_layer = nn.TransformerEncoderLayer(d_model=hidden_size, nhead=num_heads, dim_feedfor
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.classifier = nn.Linear(hidden_size, 2)  # 示例二分类任务

    def forward(self, input_ids, attention_mask):
        # 简化的嵌入和位置编码
        x = self.embedding(input_ids) + self.position
        # Transformer 编码
        # 注意：attention_mask 的处理在简化示例中省略，实际应将 pad token 遮蔽
        x = self.encoder(x.transpose(0,1), src_key_padding_mask=(attention_mask==0)).transpose(0,1)
        # 取 CLS token 位置做分类（这里简单取序列第一 token）
        cls_token = x[:, 0, :]
        logits = self.classifier(cls_token)
        return logits
```

Example Codes

- Training BERT model with distributed data parallel via Ring AllReduce

```
def train(rank, world_size, port, epochs=3, batch_size=32, lr=1e-4):
    setup_distributed(rank, world_size, port=port, backend='nccl')
    device = torch.device(f'cuda:{rank}' if torch.cuda.is_available() else 'cpu')

    model = SimpleBertLike().to(device)
    model = nn.parallel.DistributedDataParallel(model, device_ids=[rank])

    # 数据
    dataset = FakeBertLikeDataset()
    sampler = DistributedSampler(dataset, num_replicas=world_size, rank=rank, shuffle=True)
    loader = DataLoader(dataset, batch_size=batch_size, sampler=sampler, num_workers=2)

    criterion = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr=lr)

    for epoch in range(epochs):
        sampler.set_epoch(epoch)
        model.train()
        epoch_loss = 0.0
        for batch in loader:
            input_ids, attention_mask, label = [b.to(device) for b in batch]
            optimizer.zero_grad()
            logits = model(input_ids, attention_mask)
            loss = criterion(logits, label)
            loss.backward()
            optimizer.step()
```

Example Codes

- Training BERT model with distributed data parallel via Ring AllReduce

```
            epoch_loss += loss.item()

        if rank == 0:
            print(f"[Epoch {epoch+1}/{epochs}] Loss: {epoch_loss/len(loader):.4f}")

    cleanup()

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--world_size', type=int, required=True, help='Total number of processes (wo
    parser.add_argument('--rank', type=int, required=True, help='Rank of this process')
    parser.add_argument('--port', type=int, default=12355, help='Init method port for rendezvous')
    parser.add_argument('--epochs', type=int, default=3)
    parser.add_argument('--batch_size', type=int, default=32)
    return parser.parse_args()

def main():
    args = parse_args()
    train(rank=args.rank, world_size=args.world_size, port=args.port,
          epochs=args.epochs, batch_size=args.batch_size)

if __name__ == '__main__':
    main()
```

Example Codes

- Single machine two GPUs:
  - Using torchrun to launch 2 processes

```
torchrun --standalone --nproc_per_node=2 train_ddp_bert_ring.py --world_size=2
--rank=0 --epochs=3
```

```
torchrun --standalone --nproc_per_node=2 train_ddp_bert_ring.py --world_size=2
--rank=1 --epochs=3
```

- Multiple machines eight GPUs:
  - Two machines, each having four GPUs, i.e. world_size = 8
  - On two machines:

```
torchrun --nnodes=2 --nproc_per_node=4 --rdzv_id=bert_job --
rdzv_backend=c10d train_ddp_bert_ring.py --world_size=8 --rank=0 --port=12355

Increasing rank id
```