Distributed Deep Learning

Aaron Zhao, Imperial College London

Introduction

Distributed computing is scalability

Figure 1: TPU rack - modern AI compute infrastructure

In previous lectures, we mainly looked at single-node computation.

Distributed computing is scalability

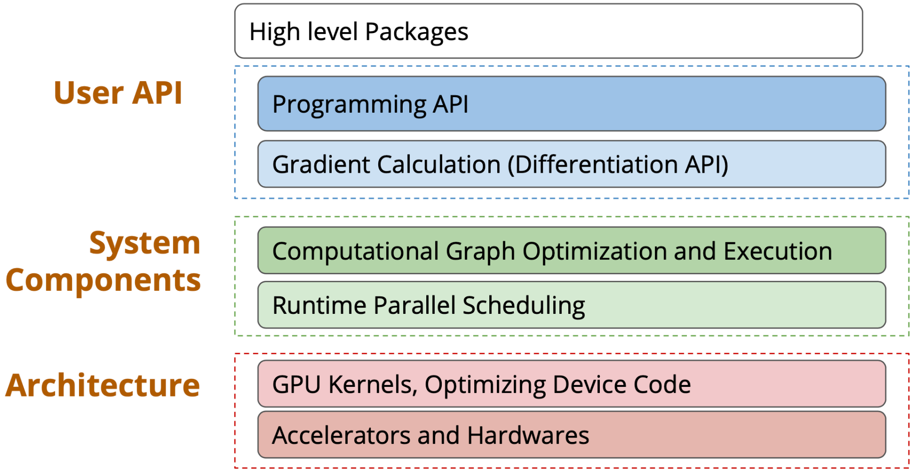

High level Packages
User API
Programming API
Gradient Calculation (Differentiation API)
System Components
Computational Graph Optimization and Execution
Runtime Parallel Scheduling
Architecture
GPU Kernels, Optimizing Device Code
Accelerators and Hardwares

Figure 2: Overview of distributed computing architecture

Distributed Computing is Scalability

- Servers are stored in racks (42U rack, normally).
- Rack servers (typically 4U) are installed in these racks, each server normally has 4-8 GPU cards, interconnected through PCIe.
- Servers in the same rack normally has a faster network (ToR).
- Racks are also connected, but a lot slower in terms of point-to-point latency.

How do we map AI workloads into such systems?

GPU Architecture and Communication

GPU Memory Hierarchy

Understanding GPU memory is crucial for distributed training:

Memory Levels (from fastest to slowest):

- **Registers**: Fastest, private to each thread (1 cycle)
- **L1 Cache/Shared Memory**: 128 KB per SM, 37.5 TB/s bandwidth
- **L2 Cache**: 40-50 MB (A100/H100), SRAM-based, tens of ns latency
- **HBM (High Bandwidth Memory)**: 40-80 GB, 2-3 TB/s per GPU, hundreds of ns latency

Key Insight: Moving data from HBM to compute units is expensive! This drives many distributed training optimizations.

GPU Interconnects: NVLink vs PCIe

PCIe (Traditional):

- PCIe Gen3: 16 GB/s per x16 link
- PCIe Gen4: 32 GB/s per x16 link
- Arbitrated by CPU - becomes a bottleneck

NVLink (NVIDIA):

- NVLink 2.0 (V100): 300 GB/s bidirectional
- NVLink 3.0 (A100): 600 GB/s
- NVLink 4.0 (H100): 900 GB/s
- NVLink 5.0 (B200): 1.8 TB/s - **60x faster than PCIe Gen4!**

Why it matters: Faster interconnects enable more efficient gradient synchronization and model parallelism.

Communication

Common collective operations:

All-Reduce: Sum gradients across all workers, distribute result
- Used in data parallelism for gradient synchronization
- Can be decomposed into Reduce-Scatter + All-Gather

All-Gather: Collect data from all workers, concatenate
- Used in FSDP/ZeRO to gather sharded parameters

Reduce-Scatter: Reduce data and scatter results
- Used in FSDP/ZeRO to average gradients

Broadcast: Send data from one worker to all others
- Used for parameter server updates

AI Compute Parallelism

Canonical view of ML parallelism

- **Data Parallelism**: data is partitioned across distributed workers, but the model is replicated. Each worker processes a different subset of the data.
- **Operator parallelism**: partition the computation of a specific operator, such as matmul, along non-batch axes, and compute each part of the operator in parallel across multiple devices.
- **Pipeline parallelism**: places different groups of ops from the model graph, referred as stages, on different workers

These parallelism strategies can also be mixed for optimal performance.

Data Parallelism Example: We replicate the same model to various workers (N GPUs). The data is then split into N parts to be deployed on these N GPUs. Each GPU computes gradients independently, then gradients are synchronized.

Operator and model parallelism

- Different colors represent different GPU devices
- We can separate the model and put different portions of the model to different devices
- Some people say tensor parallelism – this is if you chop inside an operator, and operator/model parallelism normally refers to the fact that you are chopping at the operator granularity.

Operator and model parallelism (ii)

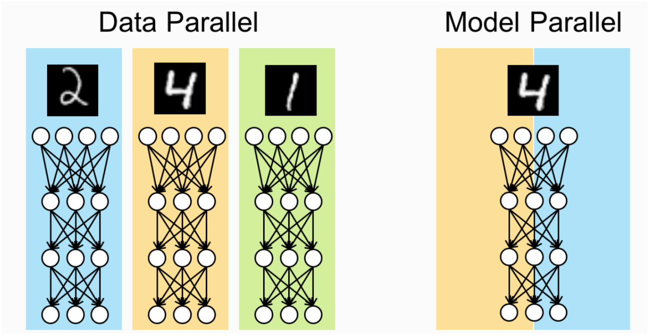


Figure 3: Model parallelism: partitioning model across devices

Pipeline parallelism

- Very similar to CPU pipelining
- Works both in inference and training

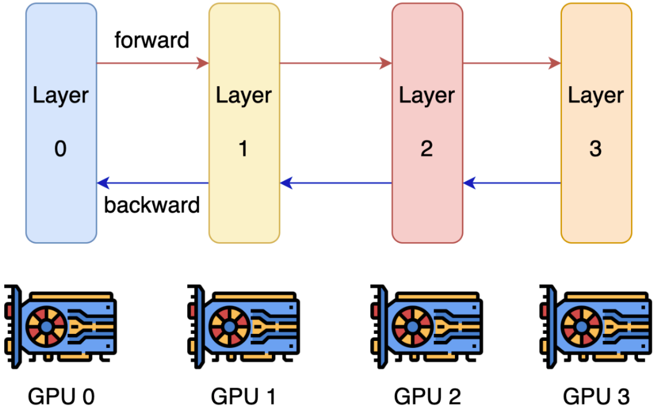


Figure 4: Pipeline parallelism: different stages on different devices

An alternative view of ML parallelism

- **Intra-operator parallelism**:

  An operator works on multi-dimensional tensors. We can partition the tensor along some dimensions, assign the resulting partitioned computations to multiple devices, and let them execute different portions of the operator at the same time.

- **Inter-operator parallelism**:

  We define inter-operator parallelism as the orthogonal class of approaches that do not perform operator partitioning, but instead, assign different operators of the graph to execute on distributed devices.

ML parallelism

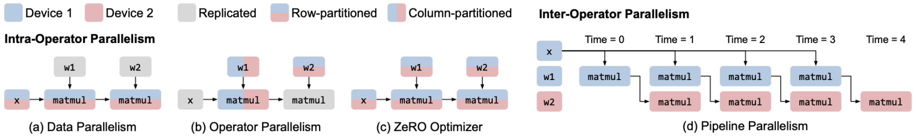


Figure 5: Alpa framework: intra-operator vs inter-operator parallelism

Model and Pipeline Parallelism Details

Model Parallelism: Split model layers across devices

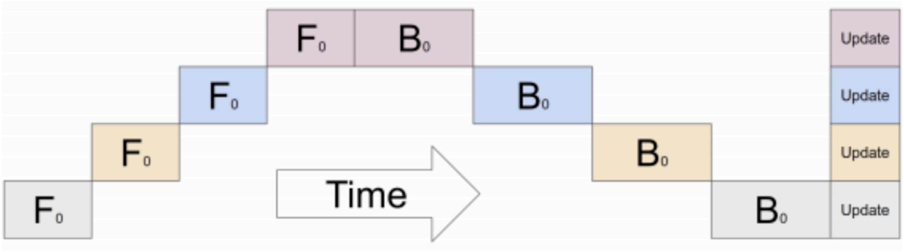
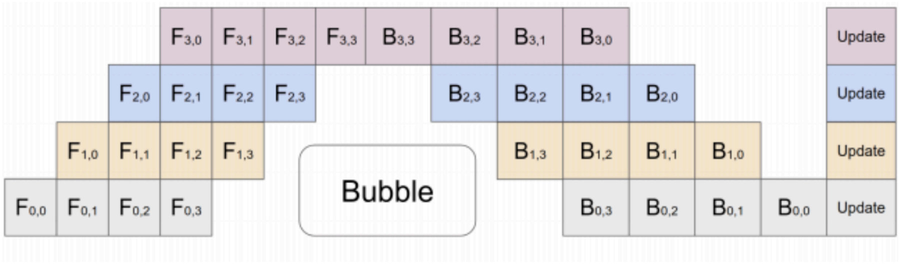


Figure 6: Model parallelism implementation details

Pipeline Parallelism: Split model into stages with micro-batching

Model and Pipeline Parallelism Details (ii)



$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ Update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ Update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ Update
Bubble

Figure 7: Pipeline parallelism with micro-batching

Tensor Parallelism

What is Tensor Parallelism?

Idea: Split individual tensor operations across GPUs

Unlike model parallelism (split layers), tensor parallelism splits **within** a layer:

Column Parallel: Split weight matrix along columns
- Each GPU computes independent output features
- Example: Split FFN intermediate dimension

Row Parallel: Split weight matrix along rows
- Each GPU computes partial outputs, then all-reduce
- Example: Split attention heads or FFN output

Key Benefit: Better load balancing than layer-wise model parallelism

Tensor Parallelism in Transformers

MLP Block:

1. First linear layer: Column parallel (split along output dim)
2. Activation (GeLU): Applied independently on each GPU
3. Second linear layer: Row parallel (split along input dim, all-reduce output)

Attention Block:

- Split attention heads across GPUs
- Each GPU computes subset of heads
- Concatenate results (or use all-reduce)

Why it works: Minimizes communication - only 2 all-reduces per transformer block (forward + backward)

Sequence Parallelism

Problem: Layer norm and dropout activations are replicated in tensor parallelism

Solution: Shard along sequence dimension for these ops

Sequence Parallel:

- Activations are sharded along sequence length
- Reduces memory for activation storage
- Particularly important for long sequences (>2K tokens)

Combined Strategy: Tensor parallelism (within layer) + Sequence parallelism (for activations)

3D Parallelism and Hybrid Strategies

The 3D Parallelism Cube

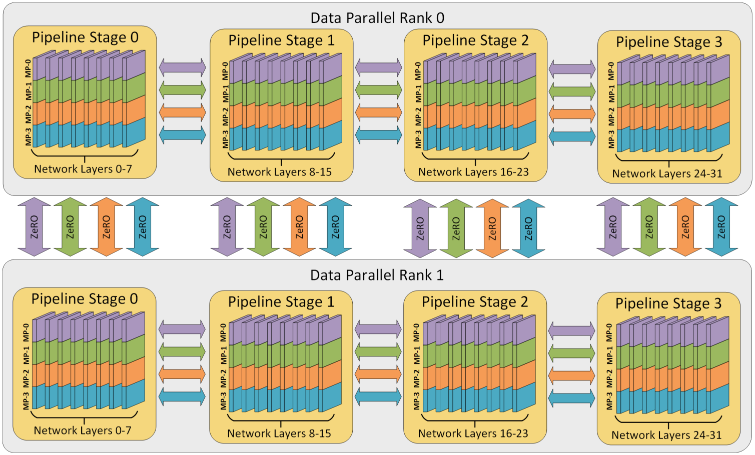


Figure 8: 3D Parallelism: Combining data, pipeline, and tensor parallelism to train trillion-parameter models

How 3D Parallelism Works

Device Mesh: Organize GPUs in 3D grid

- **Data Parallel dimension**: Replicate model across data shards
- **Tensor Parallel dimension**: Split layers within each replica
- **Pipeline Parallel dimension**: Split model stages across devices

Example Configuration (64 GPUs):

- 4-way data parallelism
- 4-way tensor parallelism
- 4-way pipeline parallelism
- Total: $4 \times 4 \times 4 = 64$ GPUs

Minimum: Need at least 8 GPUs for full 3D parallelism ($2 \times 2 \times 2$)

Pipeline Parallelism with Micro-batching

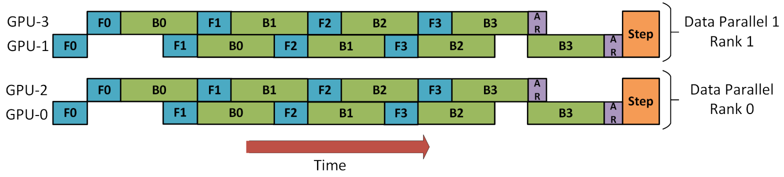


Figure 9: Pipeline schedule with micro-batches: F=forward, B=backward, AR=all-reduce

Problem: Naive pipeline has bubbles (idle time)

Solution: Split batch into micro-batches

- Overlap computation across pipeline stages
- Reduces pipeline bubble from 50% to less than 10%

Choosing the Right Strategy

Rules of Thumb:

Intra-node (8 GPUs with NVLink):

- Use tensor parallelism (fast interconnect needed)

Inter-node (multiple servers):

- Use pipeline parallelism or data parallelism (tolerates slower network)

Very large models:

- Tensor parallel within nodes (2-8 way)
- Pipeline parallel across nodes (4-16 stages)
- Data parallel for remaining GPUs
- Add FSDP/ZeRO if still memory-constrained

Distributed Training

Parameter server

It follows the data parallel approach.

In training, we split the data, and compute update at each local replica.

We then aggregate the gradients and propagate back the updated weights.

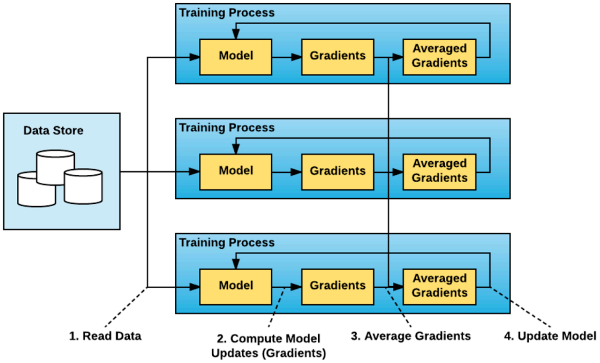


Figure 10: Parameter server architecture for distributed training

Parameter server (ii)

We need to sync the gradients and updated weights.

We gather the gradients in parameter server or parameter servers.

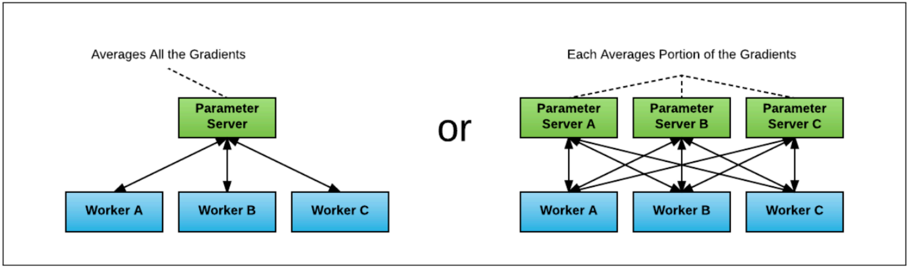


Figure 11: Gradient synchronization via parameter servers

The ring all reduce pattern

We need to sync the gradients and updated weights.

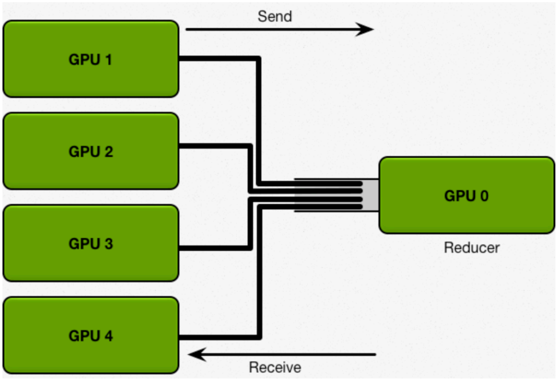


Figure 12: All-to-all communication pattern - not scalable

The most obvious communication pattern is to allow a crossbar connection, however, this is not very scalable.

The ring all reduce pattern

With the following physical connections

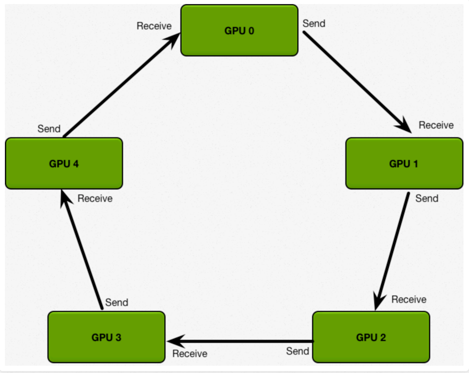


Figure 13: Ring topology for efficient gradient synchronization

The ring all reduce pattern

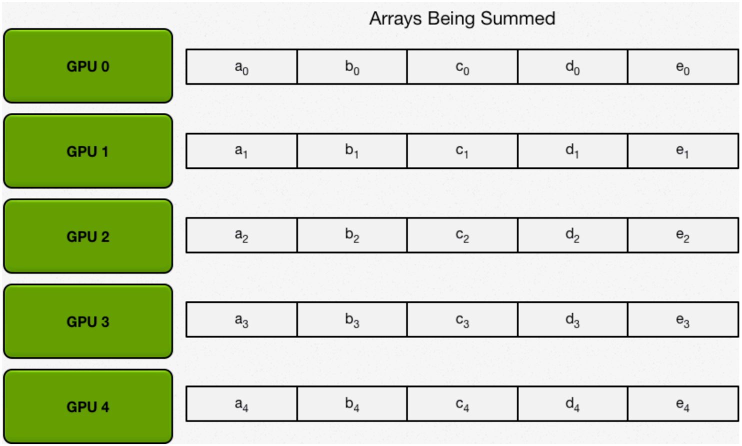

Arrays Being Summed
GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
a_0 b_0 c_0 d_0 e_0
a_1 b_1 c_1 d_1 e_1
a_2 b_2 c_2 d_2 e_2
a_3 b_3 c_3 d_3 e_3
a_4 b_4 c_4 d_4 e_4

Figure 14: Ring all-reduce: Scatter-reduce phase - Step 1

The ring all reduce pattern (ii)

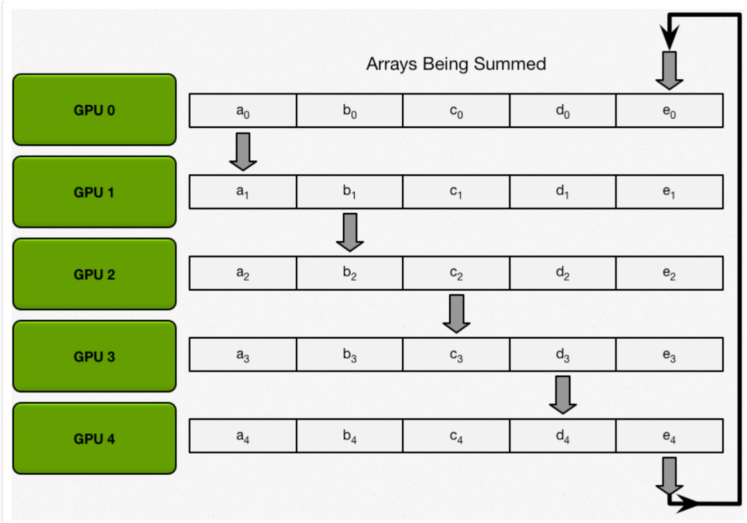
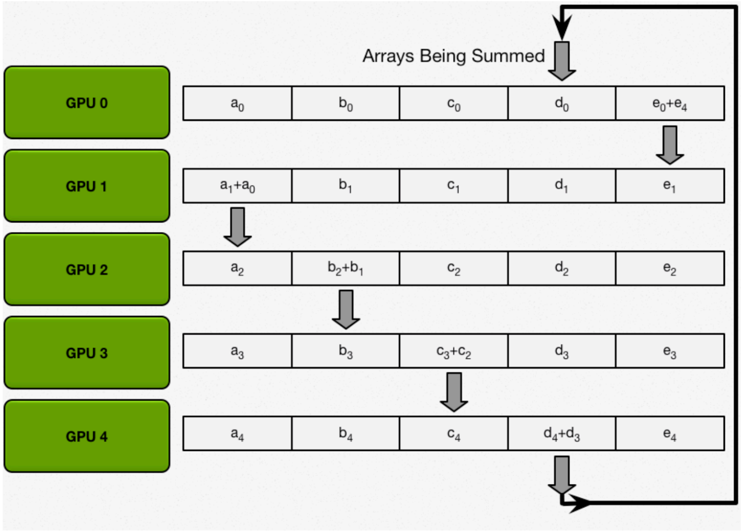

Arrays Being Summed
GPU 0
GPU 1
GPU 2
GPU 3
GPU 4
a_0 b_0 c_0 d_0 e_0
a_1 b_1 c_1 d_1 e_1
a_2 b_2 c_2 d_2 e_2
a_3 b_3 c_3 d_3 e_3
a_4 b_4 c_4 d_4 e_4

Figure 15: Ring all-reduce: Scatter-reduce phase - Step 2

The ring all reduce pattern (iii)



Arrays Being Summed
GPU 0
a_0 b_0 c_0 d_0 e_0+e_4
GPU 1
a_1+a_0 b_1 c_1 d_1 e_1
GPU 2
a_2 b_2+b_1 c_2 d_2 e_2
GPU 3
a_3 b_3 c_3+c_2 d_3 e_3
GPU 4
a_4 b_4 c_4 d_4+d_3 e_4

Figure 16: Ring all-reduce: Scatter-reduce phase - Step 3

The ring all reduce pattern (iv)

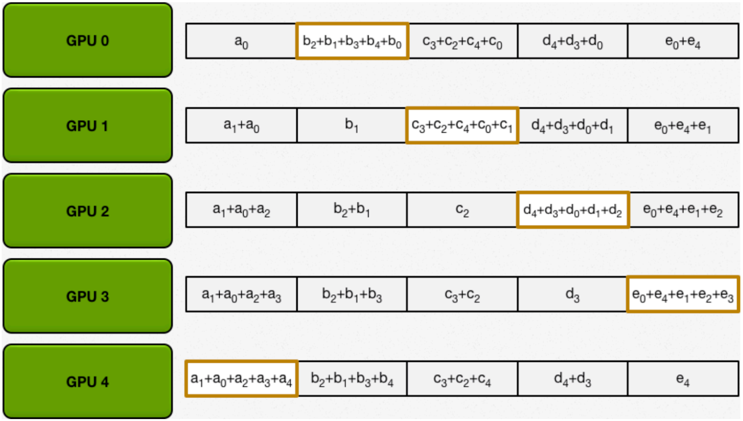


Figure 17: Ring all-reduce: Scatter-reduce phase - Final

The ring all reduce pattern (v)

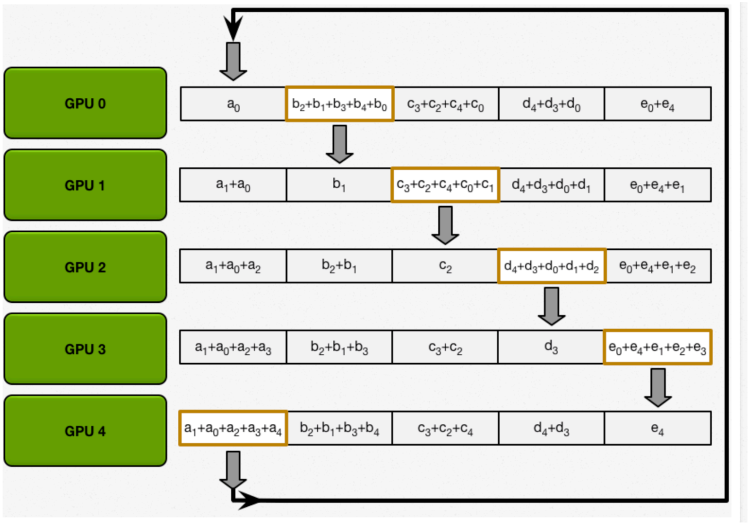


Figure 18: Ring all-reduce: All-gather phase - Step 1

The ring all reduce pattern (vi)

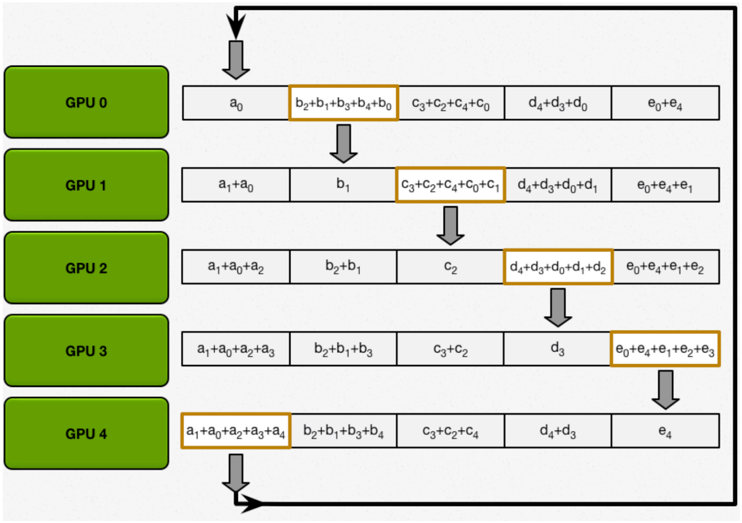


Figure 19: Ring all-reduce: All-gather phase - Step 2

The ring all reduce pattern (vii)

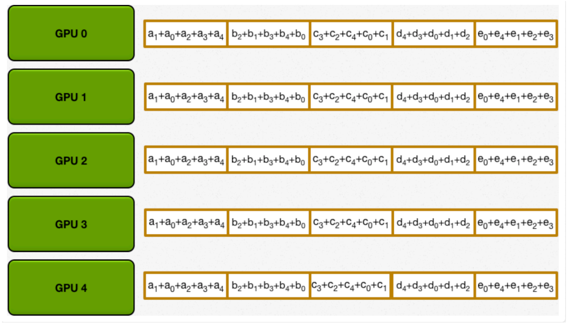

GPU 0
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
GPU 1
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
GPU 2
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
GPU 3
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$
GPU 4
$a_1+a_0+a_2+a_3+a_4$ $b_2+b_1+b_3+b_4+b_0$ $c_3+c_2+c_4+c_0+c_1$ $d_4+d_3+d_0+d_1+d_2$ $e_0+e_4+e_1+e_2+e_3$

Figure 20: Ring all-reduce: All-gather phase - Complete

Normally you do not need to worry. Typically, you don't need to be concerned about the distributed training strategy, as PyTorch, NCCL, and MPI collectively manage most of it for you. However, you must consider the distributed training strategy if you are using more than 8 GPUs, which exceeds the capacity of a single node.

The Memory Problem in Data Parallelism

Traditional DDP (Distributed Data Parallel):

- Each GPU holds a complete copy of the model
- Memory usage: Parameters + Gradients + Optimizer States
- For Adam optimizer: parameter size explodes (fp16 params + fp32 copy + momentum + variance)

Problem: For large models like GPT-3 (175B parameters):

- Memory needed: 700 GB per GPU!
- But A100 GPU only has 40-80 GB HBM

Solution: Shard model states across GPUs instead of replicating them

FSDP: All-Reduce Decomposition

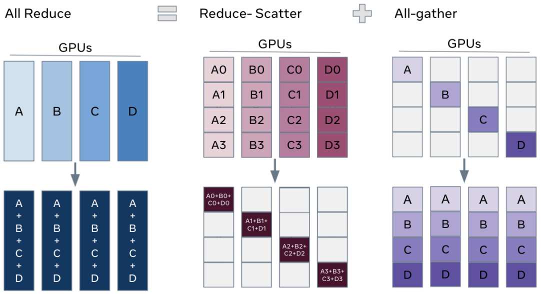
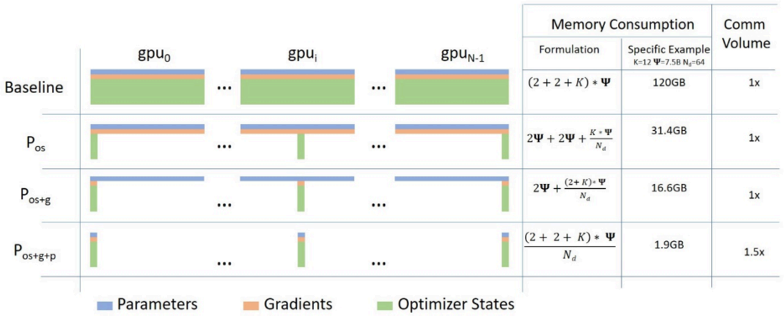


Figure 21: FSDP decomposes all-reduce into reduce-scatter and all-gather phases

Key Idea:

- All-Reduce = Reduce-Scatter + All-Gather
- Each GPU only stores a shard of parameters
- Dynamically gather full parameters when needed for forward/backward

ZeRO: Zero Redundancy Optimizer



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

Figure 22: ZeRO stages: Progressive sharding of optimizer states, gradients, and parameters

ZeRO Optimization Stages

ZeRO Stage 1: Partition optimizer states

- 4x memory reduction
- Same communication volume as DDP

ZeRO Stage 2: Partition optimizer states + gradients

- 8x memory reduction
- Same communication volume as DDP

ZeRO Stage 3: Partition optimizer states + gradients + parameters

- Linear memory reduction with number of GPUs (Nx reduction for N GPUs)
- 1.5x communication increase
- Enables training 100B+ parameter models!

When to Use What?

Standard DDP:

- Models fit comfortably in GPU memory
- Maximum training speed (no communication overhead)

FSDP / ZeRO Stage 2:

- Models barely fit in memory
- Good balance between memory and speed

ZeRO Stage 3:

- Very large models (>10B parameters)
- Must trade some speed for memory efficiency
- Can train models 64x larger than DDP!

Communication and Performance

Communication Bottlenecks

Network Bandwidth Hierarchy:

- NVLink (intra-node): 900 GB/s - 1.8 TB/s
- InfiniBand/RoCE (inter-node): 100-400 GB/s per node
- Ethernet (inter-rack): 10-100 GB/s

Ratio: Intra-node can be 10-100x faster than inter-node!

Design Principle:

- Put communication-heavy parallelism (tensor) intra-node
- Put communication-light parallelism (pipeline, data) inter-node

Optimizing Communication

Overlap Communication and Computation:

- Start gradient all-reduce while still computing other gradients
- Pipeline next micro-batch while waiting for gradients

Gradient Accumulation:

- Accumulate gradients over multiple micro-batches
- Reduce communication frequency (at cost of larger batch size)

Mixed Precision:

- Communicate fp16/bf16 gradients (2x less data)
- Keep fp32 master weights for numerical stability

Compression:

- Quantize gradients to int8 or even 1-bit
- Trade accuracy for 16-32x communication reduction

Federated Learning

The concept

Goal: Train models without centralizing user data - Let’s do not let the user data leave their phone (tricky!).

Key Idea: Bring the computation to the data, not the data to the computation.

Phase 1 and 2: The server selects a subset of clients, with each algorithm employing a distinct sampling strategy. These clients download the latest model weights from the server.

The concept (ii)

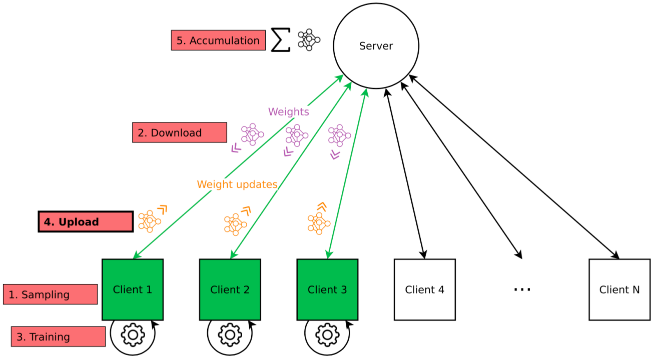


Figure 23: Federated learning: Phase 1-2 - Client selection and model distribution

Phase 3 - Local Training:

The concept (iii)

Each client performs training on a model replica locally, using its own private data. This training is always running concurrently, no matter a device is chosen or not.

The server also normally gives a time constraint on this training. If a client did not finish its training in time, it is ignored (this is known as the **straggler problem**).

The concept (iv)

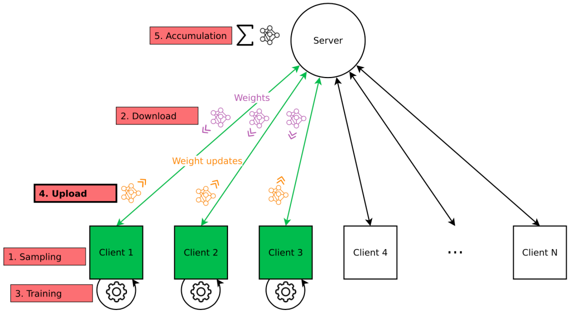


Figure 24: Federated learning: Phase 3 - Local training on devices

Phase 4 and 5 - Aggregation:

Clients now transfer back weight updates (gradients) across E local epochs back to the server.

The concept (v)

The server accumulates these updates using an aggregation algorithm (typically weighted averaging based on dataset sizes).

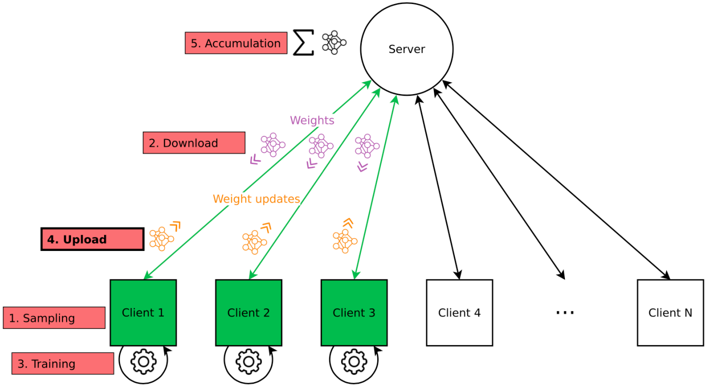


Figure 25: Federated learning: Phase 4-5 - Gradient aggregation

Pros and Cons

Biggest advantage

No “data” leaves the user device!

Other advantages

- Some level of privacy protections
- Give people an opportunity to do differential privacy

Disadvantages

- Can take very long to train since effectively there is a subsampling.
- If server is not honest, bad things can happen.

Summary

GPU Architecture and Communication:

- Memory hierarchy: Registers $\longrightarrow$ L1 $\longrightarrow$ L2 $\longrightarrow$ HBM
- NVLink (1.8 TB/s) vs PCIe (32 GB/s) - 60x difference!
- Collective operations: All-Reduce, All-Gather, Reduce-Scatter

Parallelism Strategies:

- Data Parallelism: DDP, FSDP, ZeRO (Stages 1-3)
- Tensor Parallelism: Column/Row parallel, sequence parallelism
- Pipeline Parallelism: Micro-batching to reduce bubbles
- 3D Parallelism: Combines all three for trillion-parameter models

Key Takeaways:

- Memory is the bottleneck for large models (FSDP/ZeRO helps!)
- Communication hierarchy matters: NVLink intra-node, IB/Ethernet inter-node
- Choose strategy based on model size, GPU count, and network topology
- Federated Learning: Privacy-preserving distributed training