

Large Language Models at Scale

Shilpika

Argonne National Laboratory

Contributor: Sam Foreman

Outline

1. Scaling: Overview
2. Data Parallel Training
 - Communication
 - Why Distributed Training?
3. Beyond Data Parallelism
 - Additional Parallelism Strategies
4. Hands On

Scaling: Overview

Goal:

Minimize: Cost (i.e. amount of time spent training)

Maximize: Performance

Note:

See  [Performance and Scalability](#) for more details

Training on a Single Device

See 🤗 [Methods and tools for efficient training on a single GPU](#)

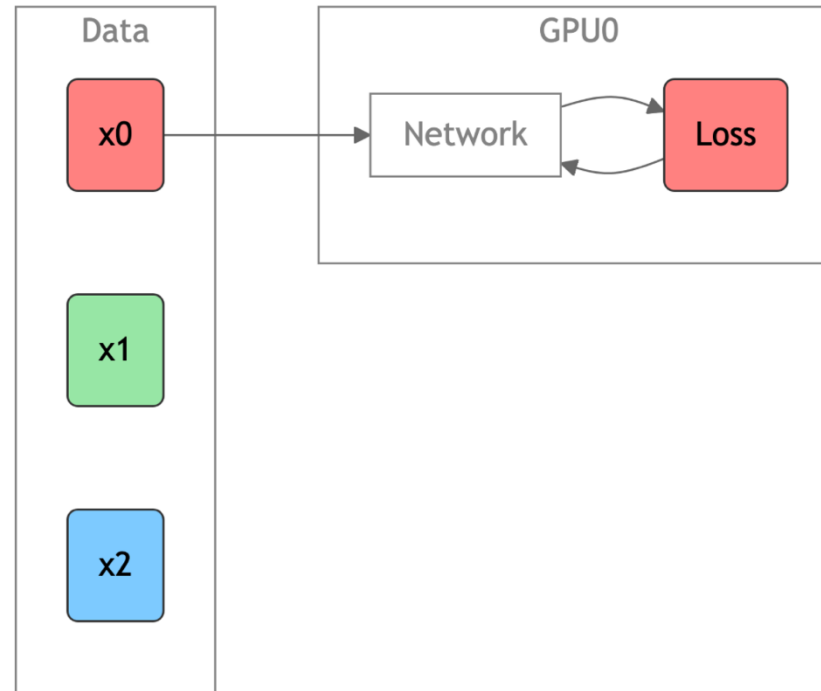


Figure 1: **SLOW** !! model size limited by GPU memory

Training on a Single Device

See 🤗 [Methods and tools for efficient training on a single GPU](#)

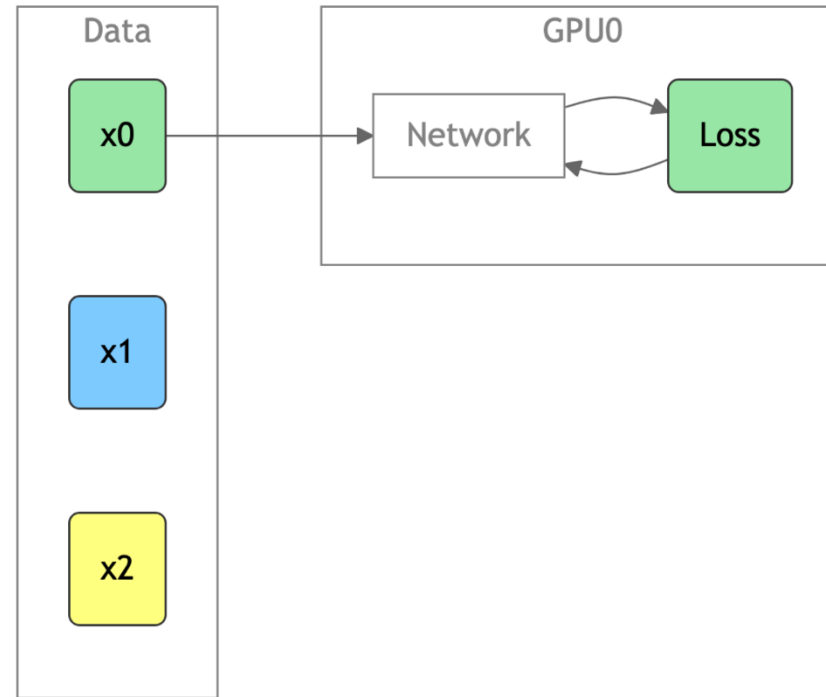


Figure 2: **SLOW** !! model size limited by GPU memory

Training on a Single Device

See 🤗 [Methods and tools for efficient training on a single GPU](#)

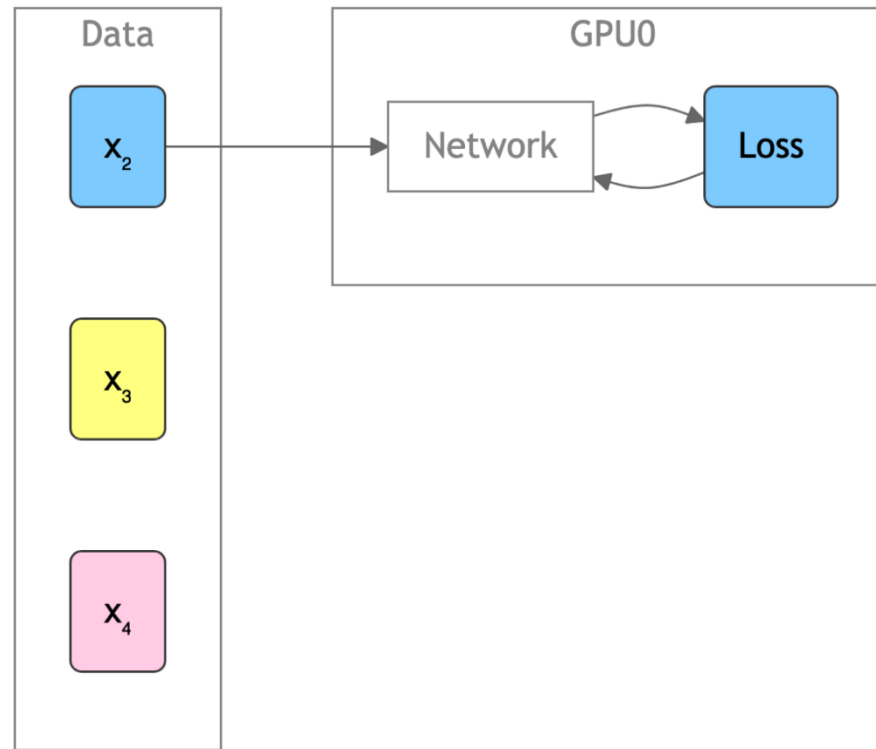


Figure 3: **SLOW** !! model size limited by GPU memory

Training on Multiple GPUs: Data Parallelism

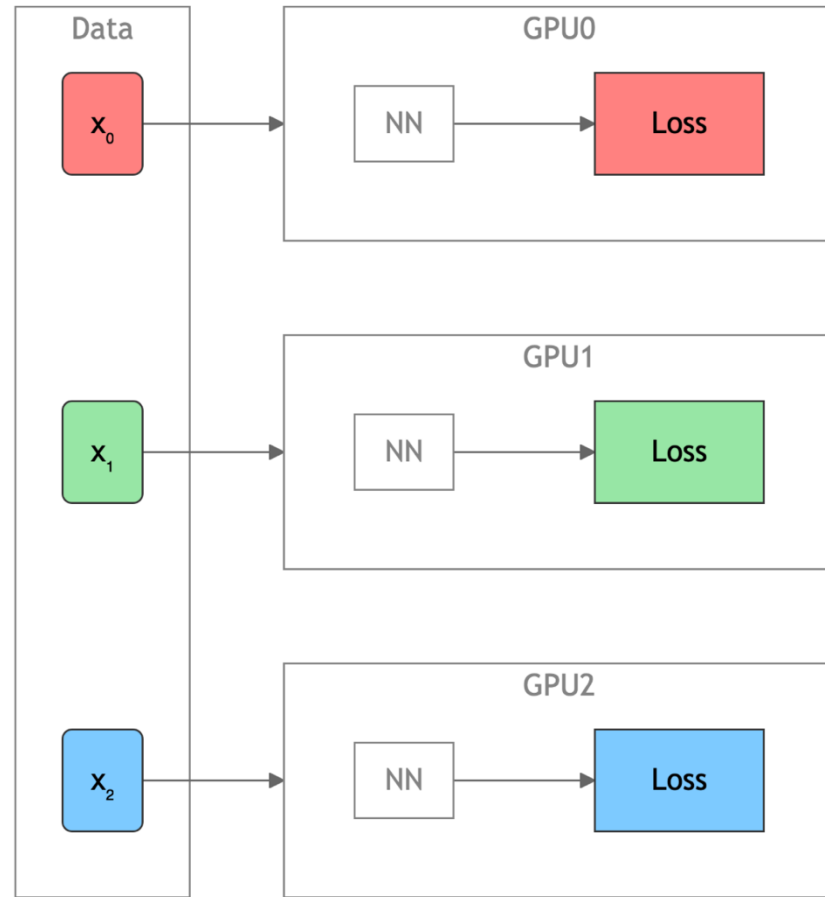


Figure 4: Each GPU receives **unique** data at each step

Data Parallel: Forward Pass

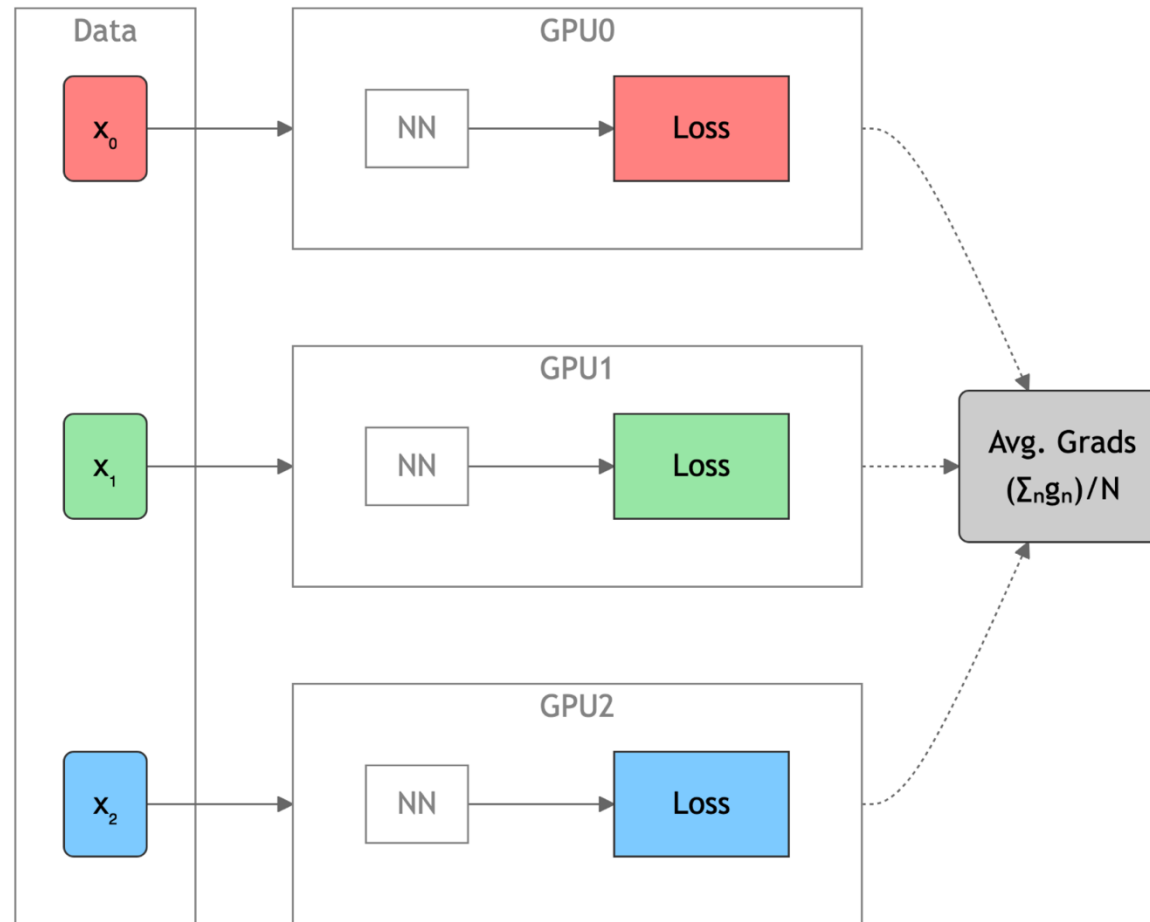


Figure 5: Average gradients across all GPUs

Data Parallel: Backward Pass

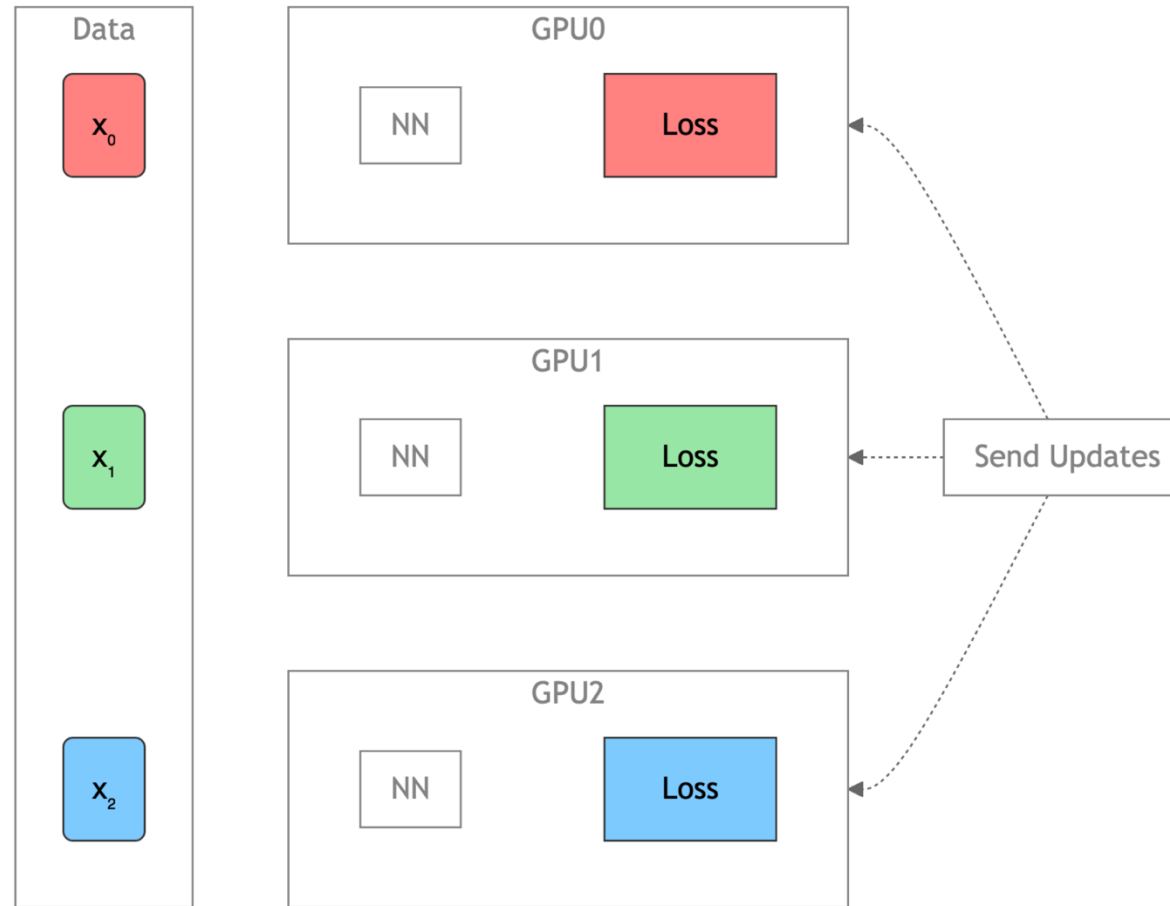


Figure 6: Send global updates back to each GPU

Data Parallel: Full Setup

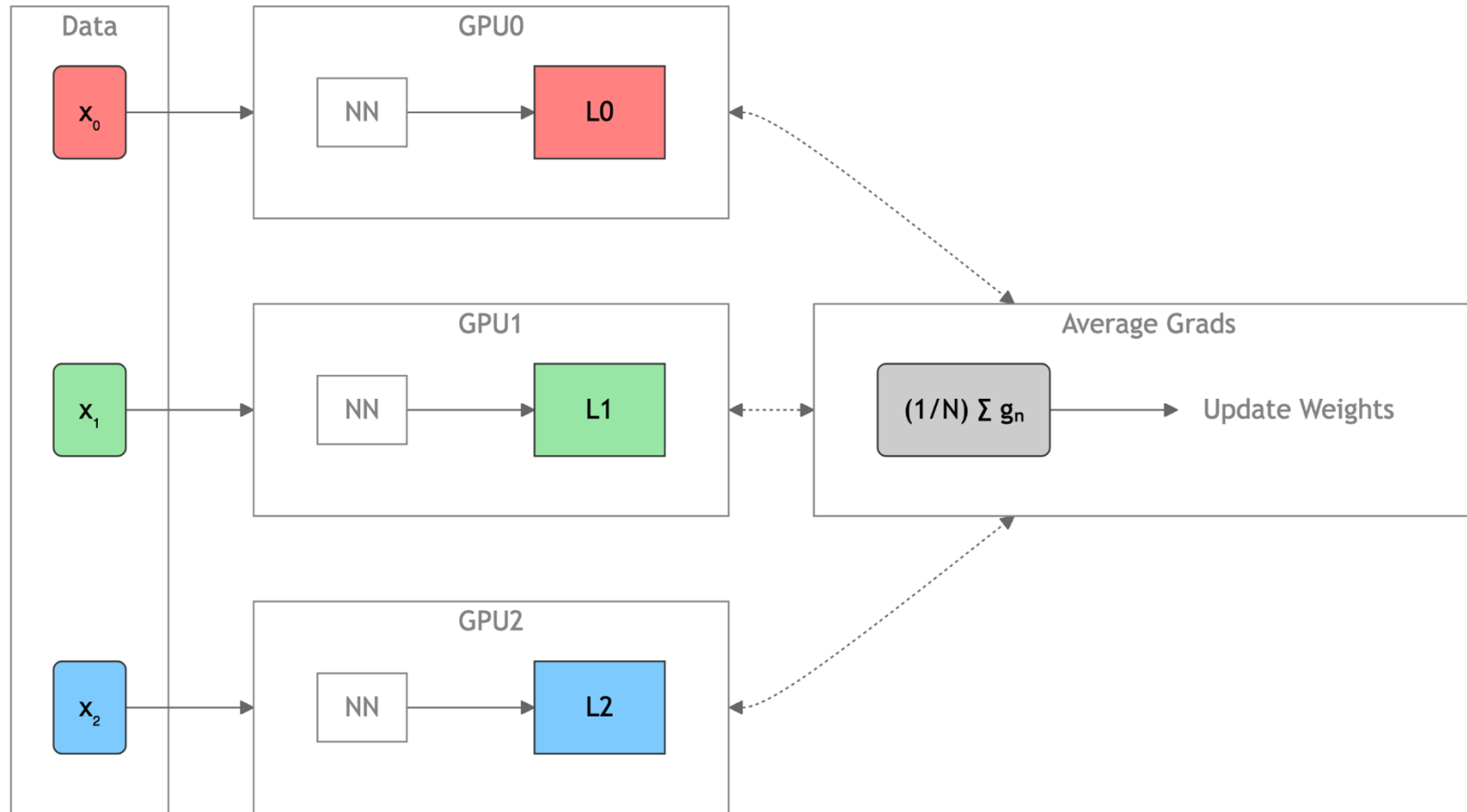


Figure 7: See: [PyTorch / Distributed Data Parallel](https://pytorch.org/distributed-data-parallel)

Data Parallel: Training

Each GPU:

- has **identical copy** of model
- works on a **unique** subset of data

Easy to get started (minor modifications to code):

 [saforem2/ezpz](#)

 [PyTorch / DDP](#)

 [HF / Accelerate](#)

 [Microsoft / DeepSpeed](#)

Requires **global** communication

- every rank *must participate* (collective communication) !!



Communication

Need mechanism(s) for communicating across GPUs:

- [mpi4py](#)
- [torch.distributed](#)

Collective Communication:

- [Nvidia Collective Communications Library \(NCCL\)](#)
- [Intel oneAPI Collective Communications Library \(oneCCL\)](#)



Timeouts

- Collective operations have to be called for each rank to form a complete collective operation.
 - Failure to do so will result in other ranks waiting **indefinitely**

AllReduce

Perform *reductions* on data (e.g. sum, min, max) across ranks, send result back to everyone.

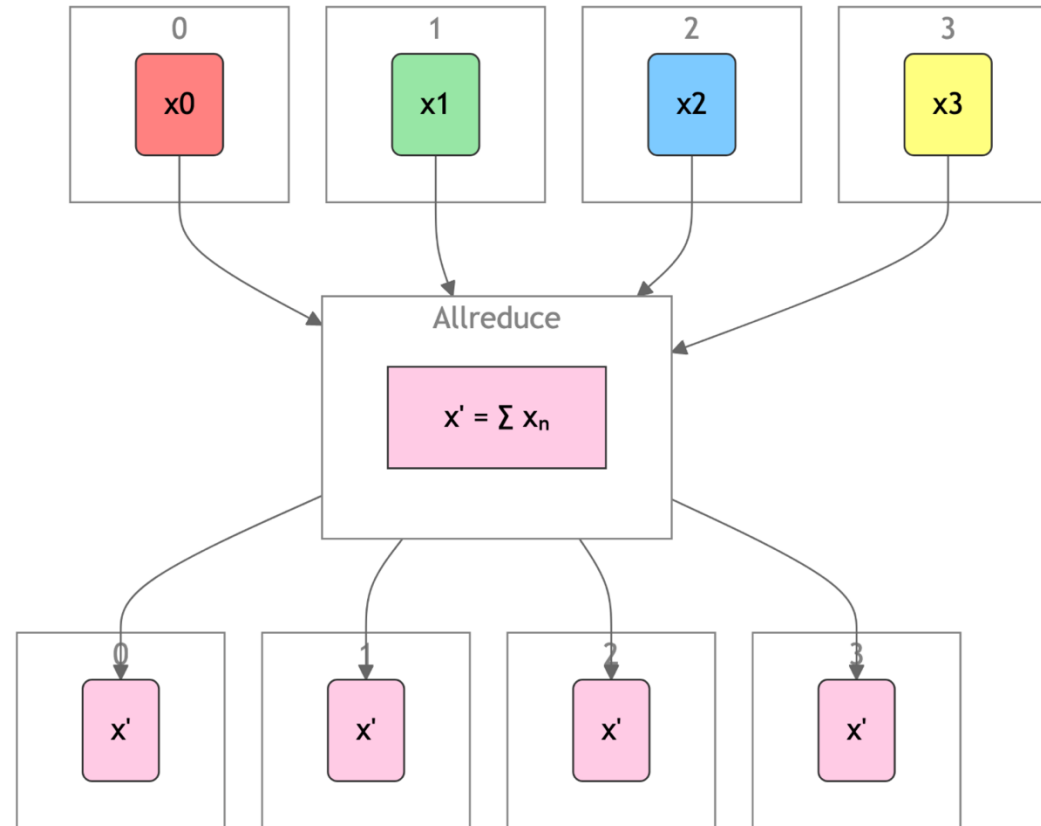


Figure 8: All-Reduce operation: each rank receives the reduction of input values across ranks.

Reduce

Perform a *reduction* on data across ranks, send to individual

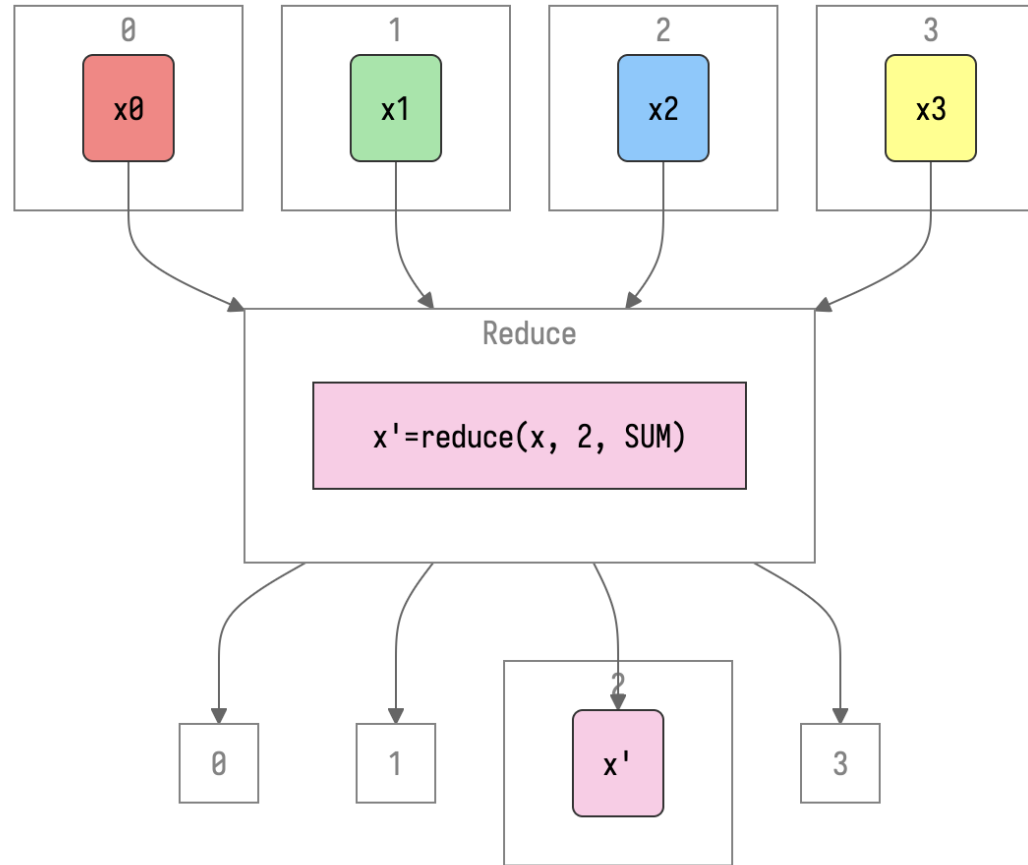


Figure 9: Reduce operation: one rank receives the reduction of input values across ranks

Broadcast

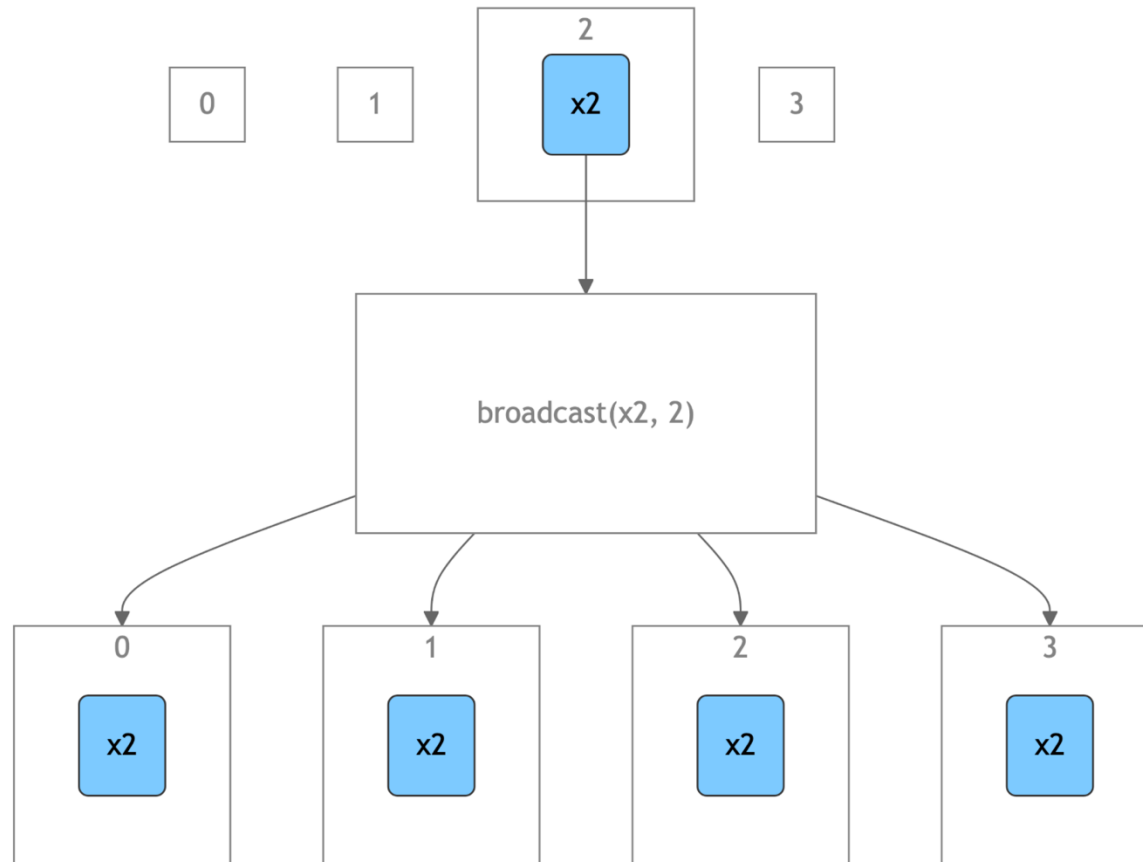


Figure 10: `broadcast` (send) a tensor x from one rank to all ranks

AllGather

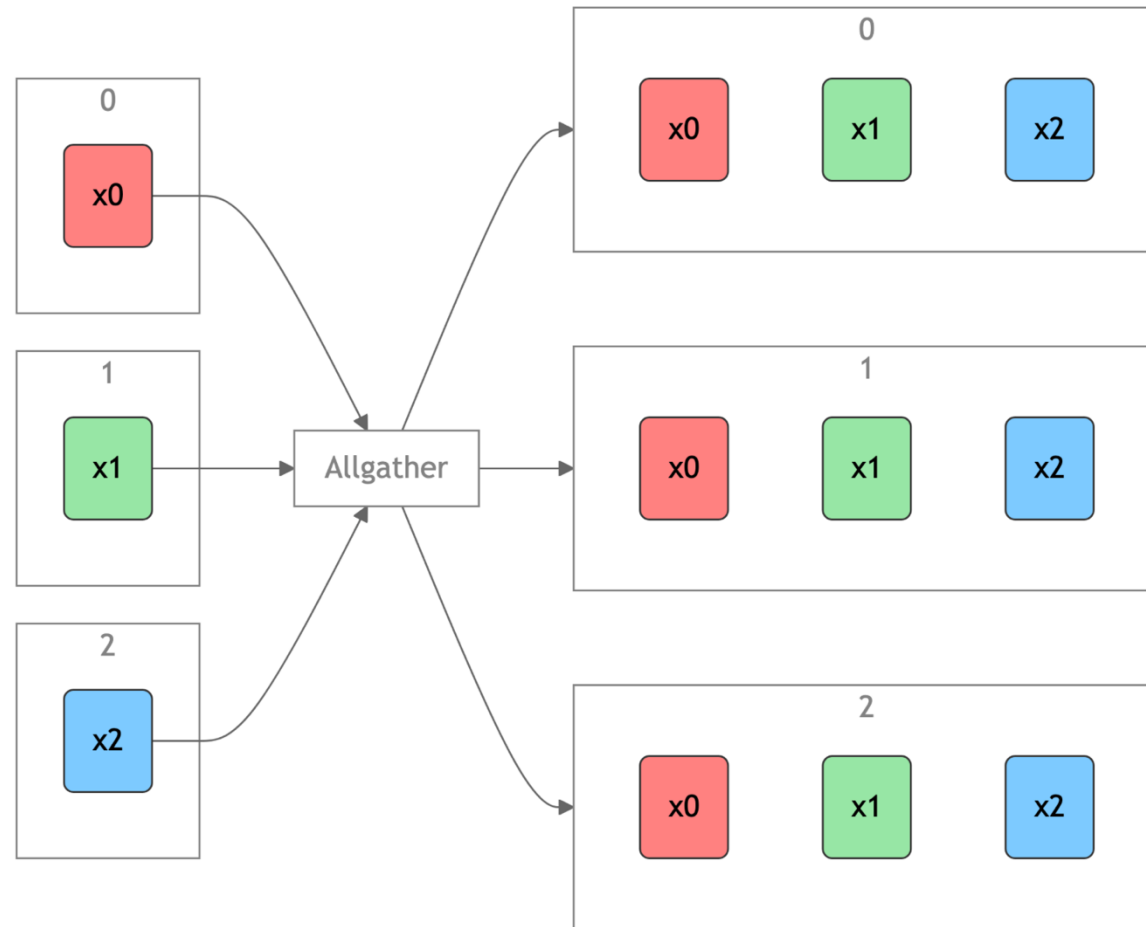


Figure 11: Gathers tensors from the whole group in a list.

Why Distributed Training?

- N workers each processing unique batch¹ of data:
 - $[\text{micro_batch_size} = 1] \times [N \text{ GPUs}] \rightarrow [\text{global_batch_size} = N]$
- Improved gradient estimators
 - Smooth loss landscape
 - Less iterations needed for same number of epochs
 - common to scale learning rate $\text{lr} \propto \sqrt{N}$
- See: [Large Batch Training of Convolutional Networks](#)

¹ micro_batch_size = batch_size per GPU

Why Distributed Training? Speedup!

Table 1: Recent progress

| Year | Author | GPU | Batch Size | # GPU | TIME (s) | ACC |
|------|----------|------|------------|-------|----------|--------|
| 2016 | He | P100 | 256 | 8 | 104,400 | 75.30% |
| 2019 | Yamazaki | V100 | 81,920 | 2048 | 72 | 75.08% |

Dealing with Data

At each training step, we want to ensure that **each worker receives unique data**

This can be done in one of two ways:

1. Manually partition data (ahead of time)
 - Assign **unique subsets** to each worker
 - Each worker can only see their local portion of the data
 - Most common approach
2. From each worker, randomly select a mini-batch
 - Each worker can see the full dataset
 - ⚠ When randomly selecting, it is important that each worker uses different seeds to ensure they receive unique data

Broadcast Initial State

At the start of training (or when loading from a checkpoint), we want all of our workers to be initialized consistently
Broadcast the model and optimizer states from rank() = 0 worker

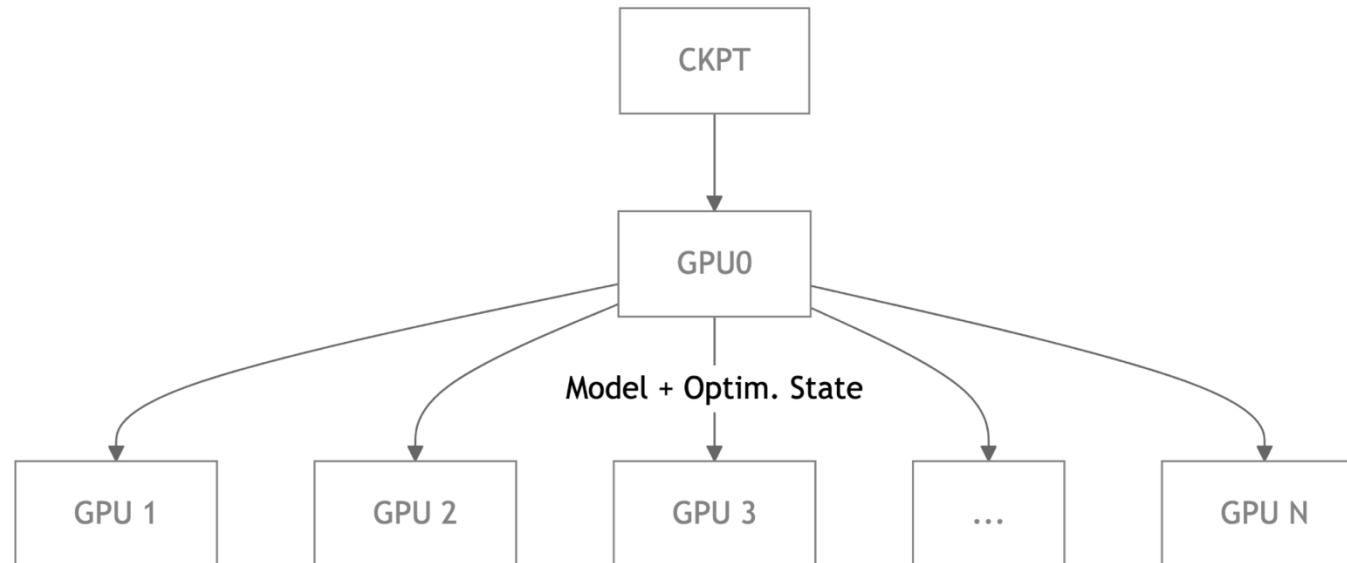


Figure 13: To ensure all workers have the same copies, we load on `RANK==0` and broadcast





Best Practices

Computation stalls during communication !!

Keeping the communication to computation ratio small is important for effective scaling.

- Use parallel IO whenever possible
 - Feed each rank from different files
 - Use MPI IO to have each rank read its own batch from a file
 - Use several ranks to read data, MPI to scatter to remaining ranks
 - Most practical in big *at-scale* training
- Take advantage of data storage
 - Use [striping on lustre](#)
- Use the right optimizations for Aurora, Polaris, etc.
- Preload data when possible
 - Offloading to a GPU frees CPU cycles for loading the next batch of data
 - **minimize IO latency this way**

Going Beyond Data Parallelism

-  Useful when model fits on single GPU:
 - ultimately **limited by GPU memory**
 - model performance limited by size
-  When model does not fit on a single GPU:
 - Offloading (can only get you so far...):
 -  [DeepSpeed + ZeRO \(ZeRO++\)](#)
 -  [PyTorch + FSDP](#)
 - Otherwise, resort to [model parallelism strategies](#)

Going Beyond Data Parallelism : DeepSpeed + ZeRO(++)

Depending on the ZeRO stage (1, 2, 3), we can offload:

Stage 1: optimizer states (P_{os})

Stage 2: gradients + opt. states (P_{os+g})

Stage 3: model params + grads + opt. states (P_{os+g+p})

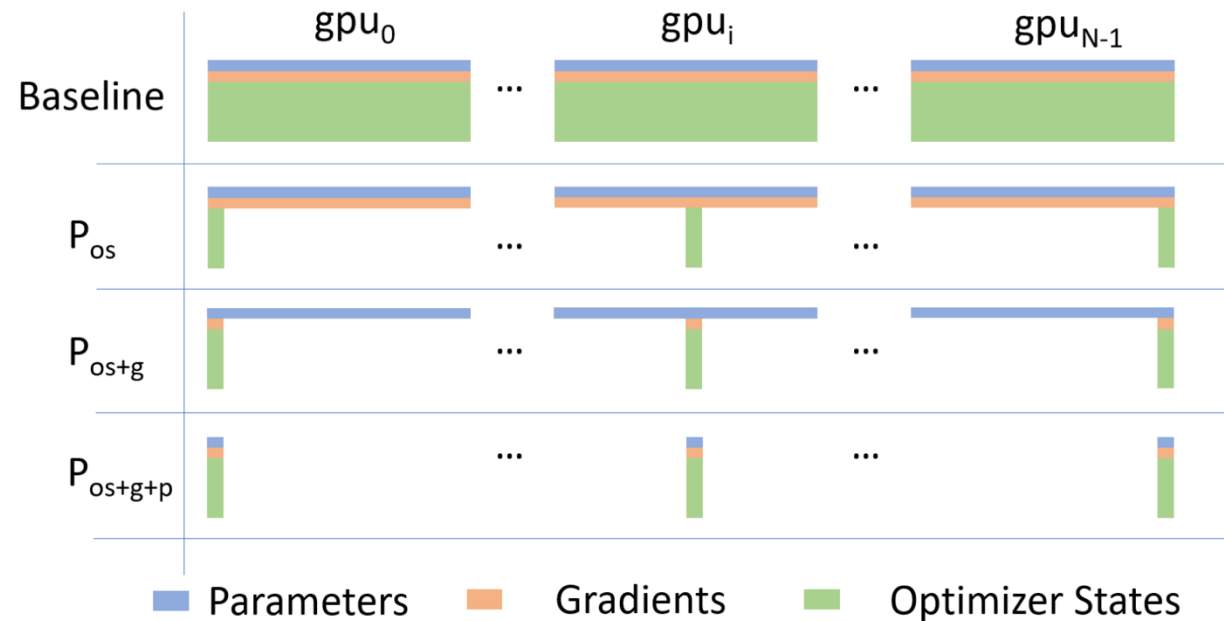


Figure 14:  DeepSpeed +  ZeRO

Fully Sharded Data Parallel: 🦋 PyTorch + FSDP

- Instead of maintaining per-GPU copy of {params, grads, opt_states}, FSDP shards (distributes) these across data-parallel workers
 - can optionally offload the sharded model params and grads to CPU
- [Introducing PyTorch Fully Sharded Data Parallel \(FSDP\) API | PyTorch](#)

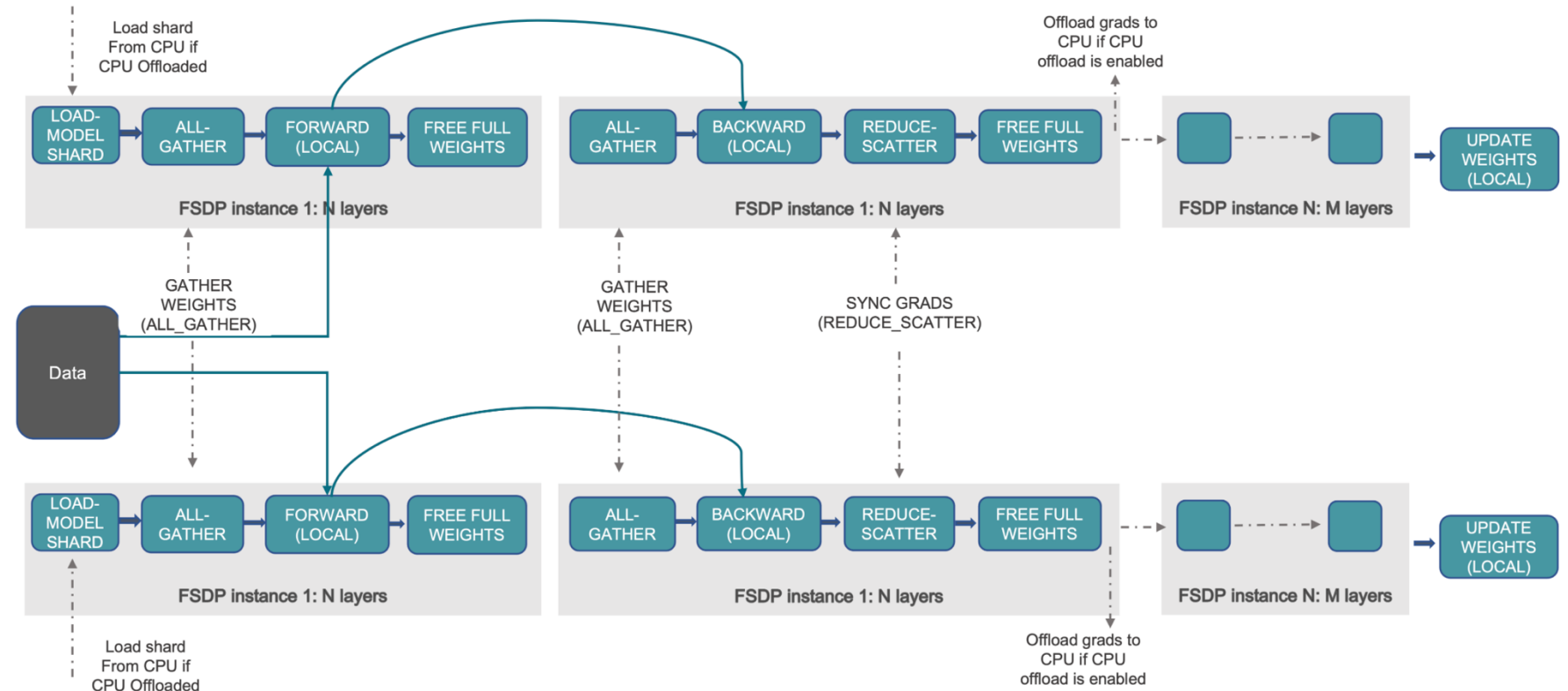


Figure 15: FSDP Workflow. [Source](#)

Additional Parallelism Strategies

- **Tensor (/ Model) Parallelism (TP):**
 - 😊 [Tensor Parallelism](#)
 - 🔥 [Large Scale Transformer model training with Tensor Parallel \(TP\)](#)
- **Pipeline Parallelism (PP):**
 - 🔥 [PyTorch](#), [DeepSpeed](#)
- **Sequence Parallelism (SP):**
 - [DeepSpeed Ulysses](#)
 - [Megatron / Context Parallelism](#)
 - [Unified Sequence Parallel \(USP\)](#)
 - [feifeibear/long-context-attention](#)
- [argonne-lcf/Megatron-DeepSpeed](#)
 - Supports 4D Parallelism (DP + TP + PP + SP)

Additional Parallelism Strategies: Pipeline Parallelism (PP)

- Model is split up **vertically** (layer-level) across multiple GPUs
- Each GPU:
 - has a portion of the full model
 - processes *in parallel* different stages of the pipeline (on a small chunk of the batch)
- See:
 - [PyTorch / Pipeline Parallelism](#)
 - [DeepSpeed / Pipeline Parallelism](#)

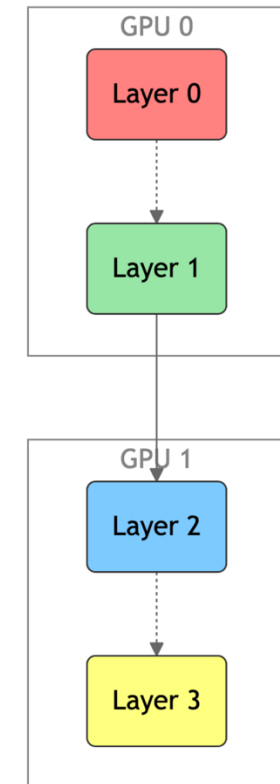


Figure 16: Pipeline Parallelism

Additional Parallelism Strategies: Tensor Parallel (TP)

- Each tensor is split up into multiple chunks
- Each shard of the tensor resides on its designated GPU
- During processing each shard gets processed separately (and in parallel) on different GPUs
 - synced at the end of the step
- See: 😊 [Model Parallelism](#) for additional details

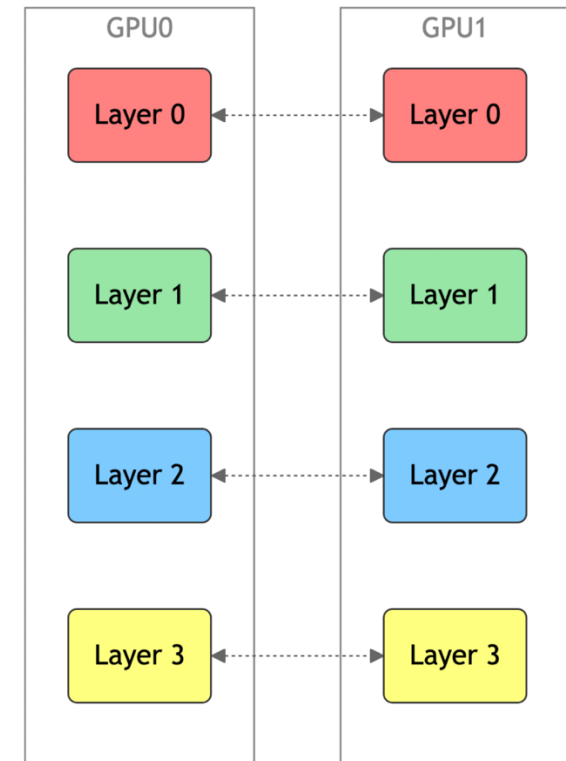




Figure 17: Tensor Parallel Training

Additional Parallelism Strategies: Tensor Parallel (TP)

- Suitable when the model is too large to fit onto a single device (CPU / GPU)
- Typically, more complicated to implement than data parallel training
 - This is what one may call horizontal parallelism
 - Communication whenever dataflow between two subsets
-  argonne-lcf/Megatron-DeepSpeed
-  huggingface/nanotron

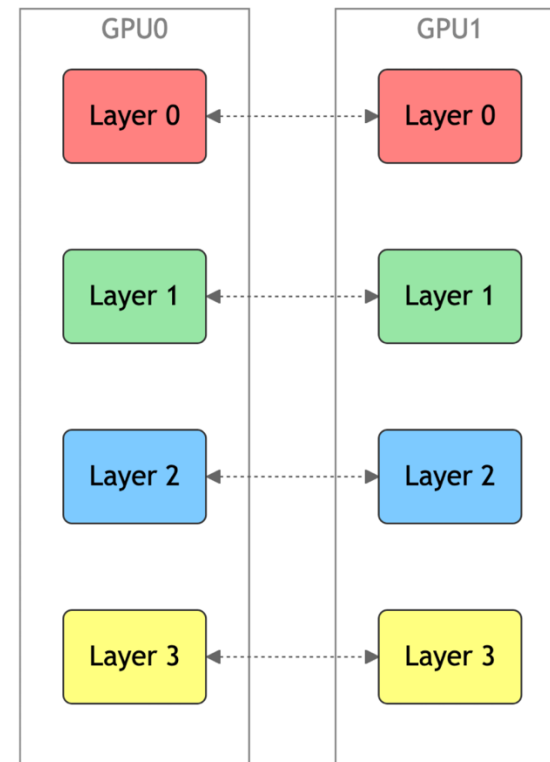


Figure 17: Tensor Parallel Training

Tensor (/ Model) Parallel Training: Example

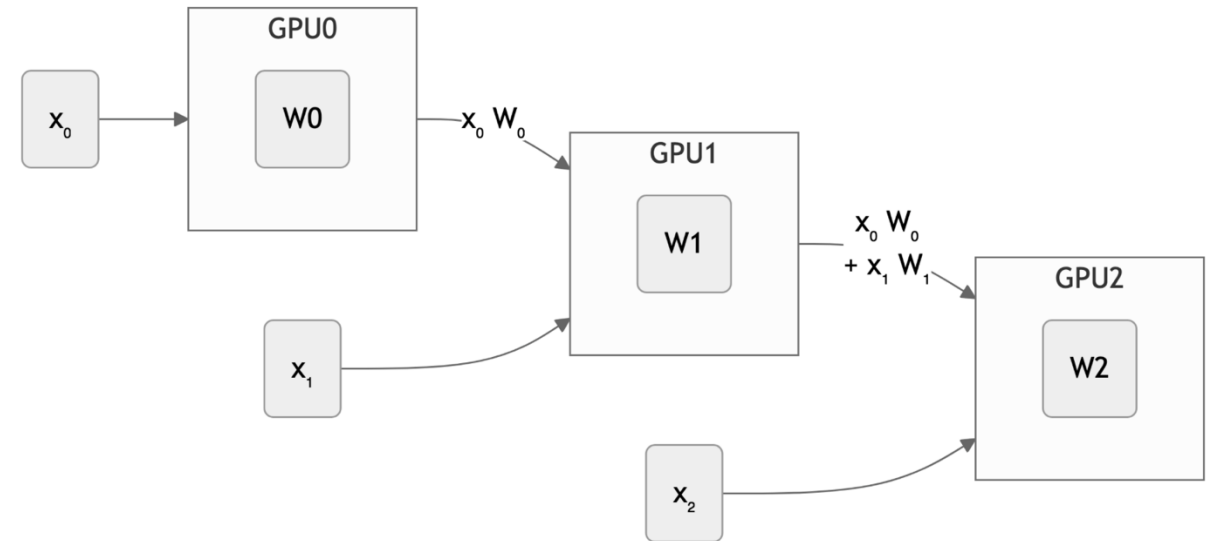
Want to compute: $y = \sum_i x_i W_i = x_0 * W_0 + x_1 * W_1 + x_2 * W_2$

where each GPU only has only its portion of the full weights as shown below

Compute: $y_0 = x_0 * W_0 \rightarrow$ GPU1

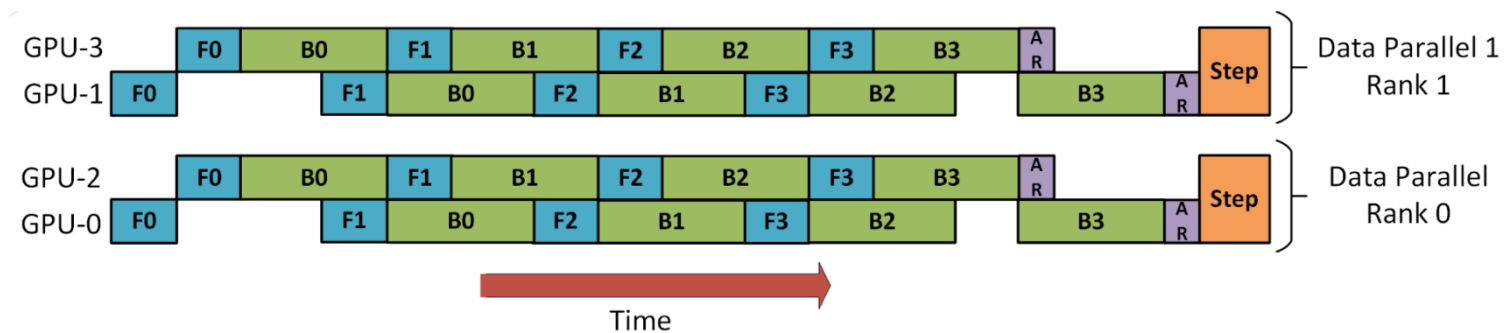
Compute: $y_1 = y_0 + x_1 * W_1 \rightarrow$ GPU2

Compute: $y = y_1 + x_2 * W_2 = \sum_i x_i W_i$



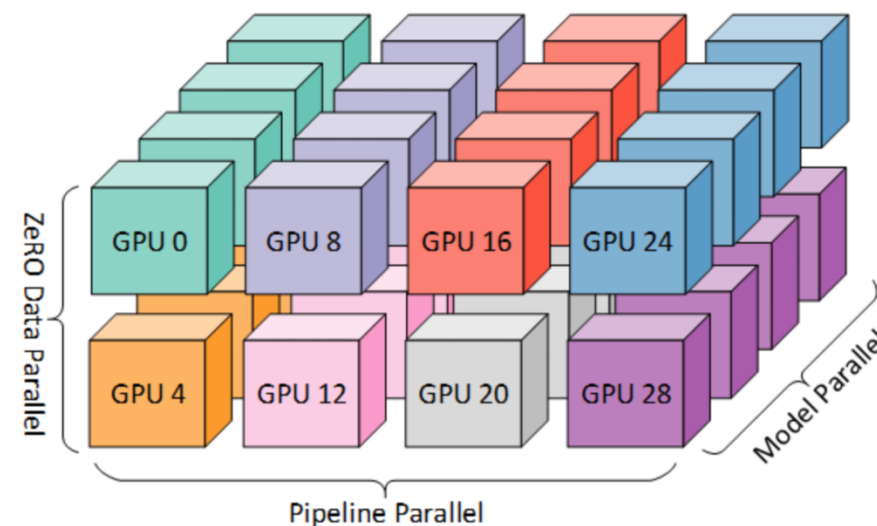
2D/3D Parallelism

DP + PP



Credit: DeepSpeed [pipeline tutorial](#)

DP + TP + PP (3D) Parallelism



Credit: [3D parallelism: Scaling to trillion-parameter models](#)

Deciding on a Parallelism Strategy: Single GPU

- Model fits onto a **single GPU**:
 - Normal use
- Model **DOES NOT** fit on a single GPU:
 - ZeRO + Offload CPU (or, optionally, NVMe)
- Largest layer **DOES NOT** fit on a single GPU:
 - ZeRO + Enable [Memory Centric Tiling \(MCT\)](#)
 - MCT Allows running of arbitrarily large layers by automatically splitting them and executing them sequentially.

Deciding on a Parallelism Strategy: Single Node/Multi GPU

- Model fits onto a single GPU
 - [DDP](#)
 - [ZeRO](#)
- Model **DOES NOT** fit onto a single GPU
 - [Pipeline Parallelism \(PP\)](#)
 - [ZeRO](#)
 - [Tensor Parallelism \(TP\)](#)

With sufficiently fast connectivity between nodes, these three strategies should be comparable.

- Otherwise, $PP > ZeRO \simeq TP$.

Deciding on a Parallelism Strategy: Multi Node/Multi GPU

- When you have fast inter-node connectivity:
 - ZeRO (virtually NO modifications)
 - PP + ZeRO + TP + DP (less communication, at the cost of MAJOR modifications)
 - when you have slow inter-node connectivity and still low on GPU memory:
DP + PP + TP + ZeRO-1

References

- Wei, Jason, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, et al. 2022. “Emergent Abilities of Large Language Models.” <https://arxiv.org/abs/2206.07682>.
- Yao, Shunyu, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. “Tree of Thoughts: Deliberate Problem Solving with Large Language Models.” <https://arxiv.org/abs/2305.10601>.
- Foreman, Sam. 2024. “Parallel Training Methods.” November 5. <https://samforeman.me/talks/ai-for-science-2024/slides>.
- Foreman, Sam. 2025. “LLMs on Aurora: Overview.” May 21. <https://samforeman.me/talks/incite-hackathon-2025/AuroraGPT/slides.html>.

Footnotes:

micro_batch_size = batch_size per GPU↔

[Efficient Large-Scale Language Model Training on GPU Clusters↔](#)

Source: [Hannibal046/Awesome-LLM↔](#)

Figure from [The Illustrated Transformer↔](#)

Figure from [The Illustrated Transformer↔](#)

Video from: 🧐 [Generation with LLMs↔](#)

Video from: 🧐 [Generation with LLMs↔](#)

Acknowledgements

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357

Hands-on

https://github.com/argonne-lcf/ATPESC_MachineLearning/blob/master/06_training_LLMs_at_scale/instructions-atpesc-2025.md



ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.

The U.S. Government retains for itself and others acting on its behalf a nonexclusive, royalty-free license in this video, with the rights to reproduce, to prepare derivative works, and to display publicly.

ARGONNE
ATPESC2025
EXTREME - SCALE COMPUTING

Thank you