# Parallel Training Methods

@ Intro to AI-driven Science on Supercomputers

Sam Foreman

ALCF

2024-11-05

# 👀 Overview

- 📊 Slides @ samforeman.me/talks/ai-for-science-2024/slides
  - 📄 HTML version: samforeman.me/talks/ai-for-science-2024
- argonne-lcf/ai-science-training-series
  - Series Page

# Outline

# 🚀 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

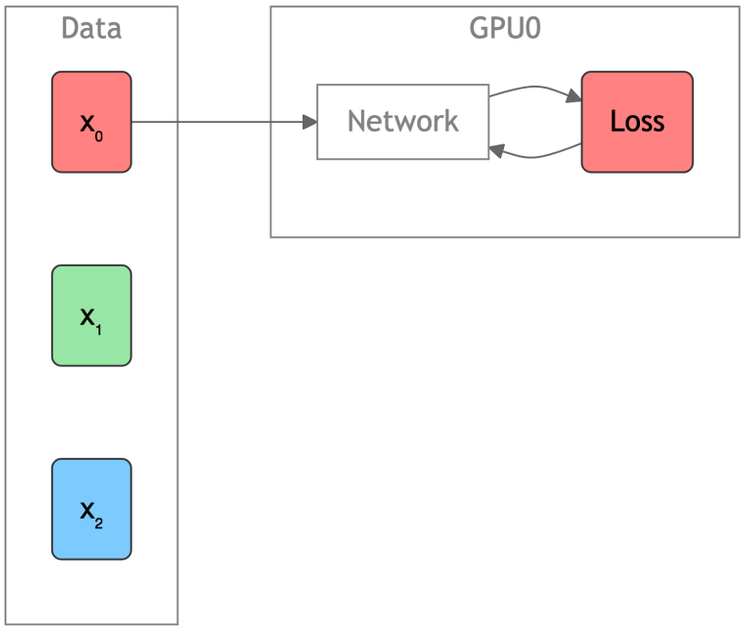



Figure 1: **SLOW** !! model size limited by GPU memory

# Training on Multiple GPUs: Data Parallelism

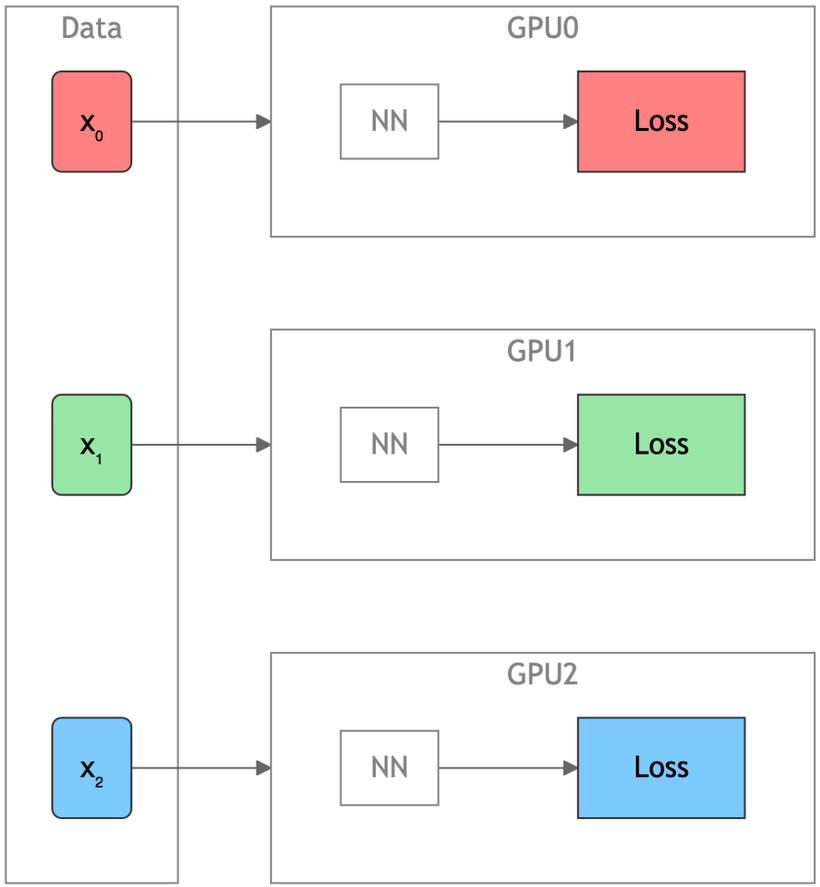



Figure 2: Each GPU receives **unique** data at each step

# Data Parallel: Forward Pass

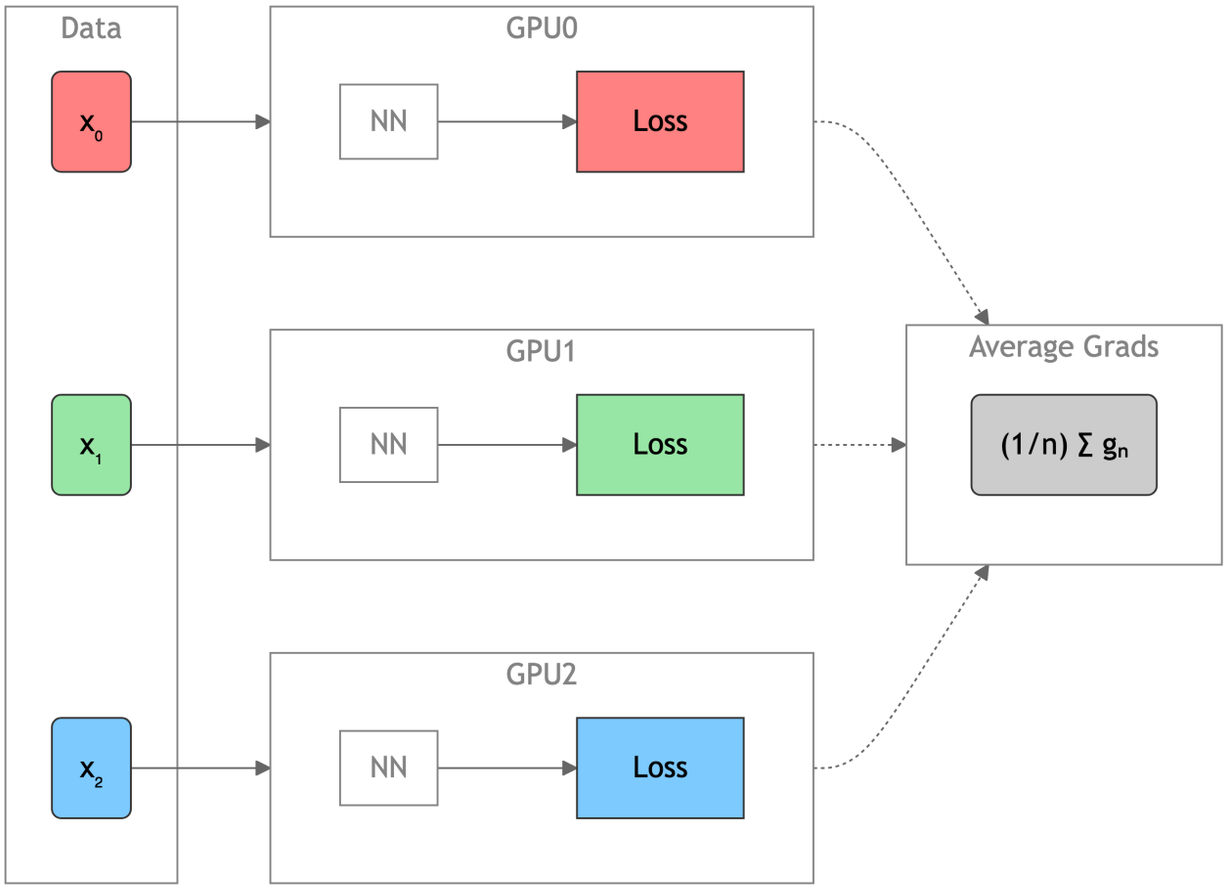



Figure 3: Average gradients across all GPUs

# Data Parallel: Backward Pass

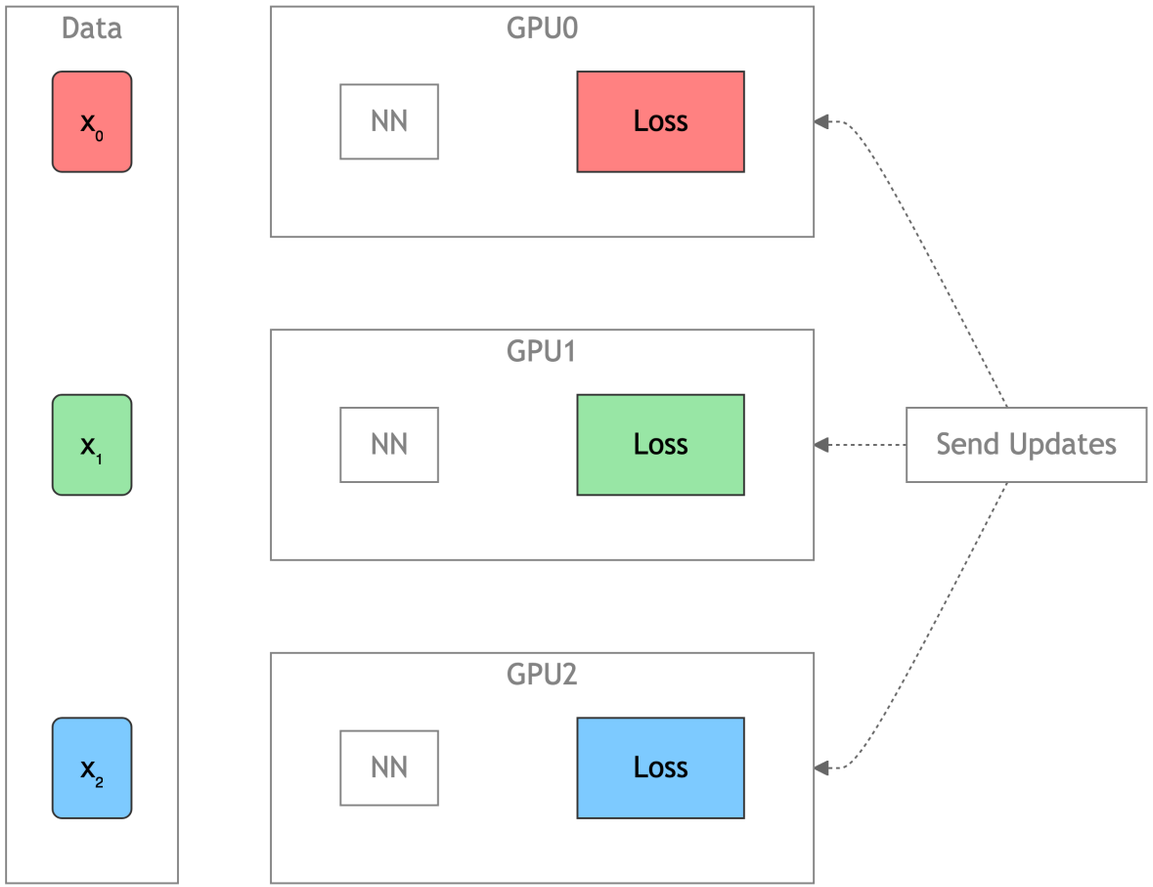



Figure 4: Send global updates back to each GPU

# Data Parallel: Full Setup

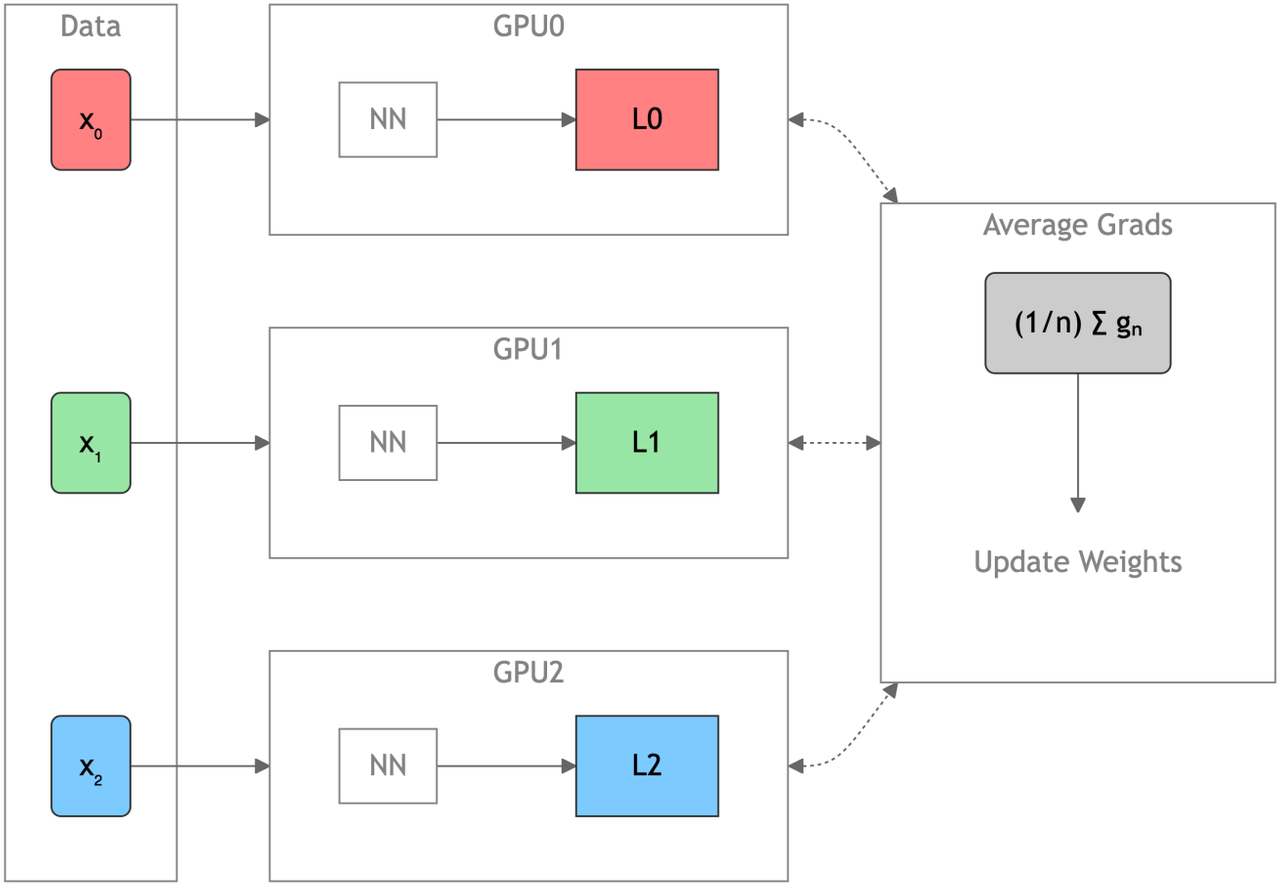



Figure 5: See: PyTorch / Distributed Data Parallel

# 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) !!

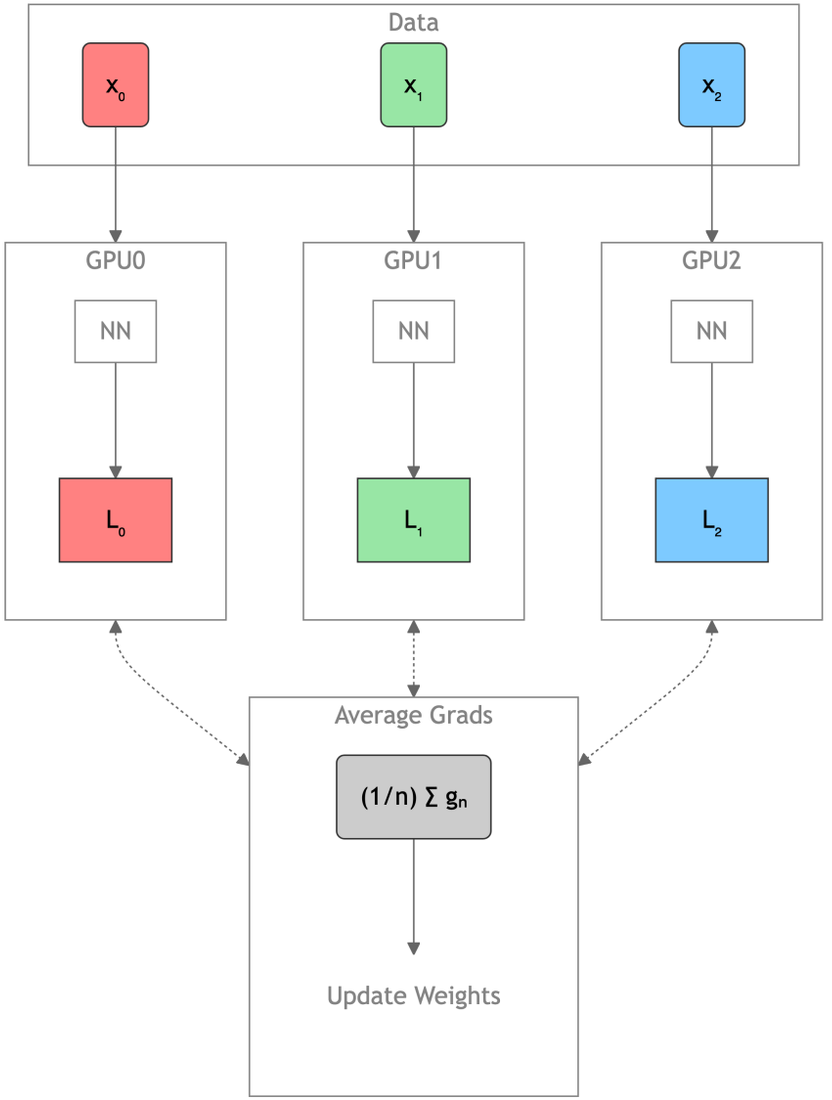



Figure 6

# 🗣️ 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.

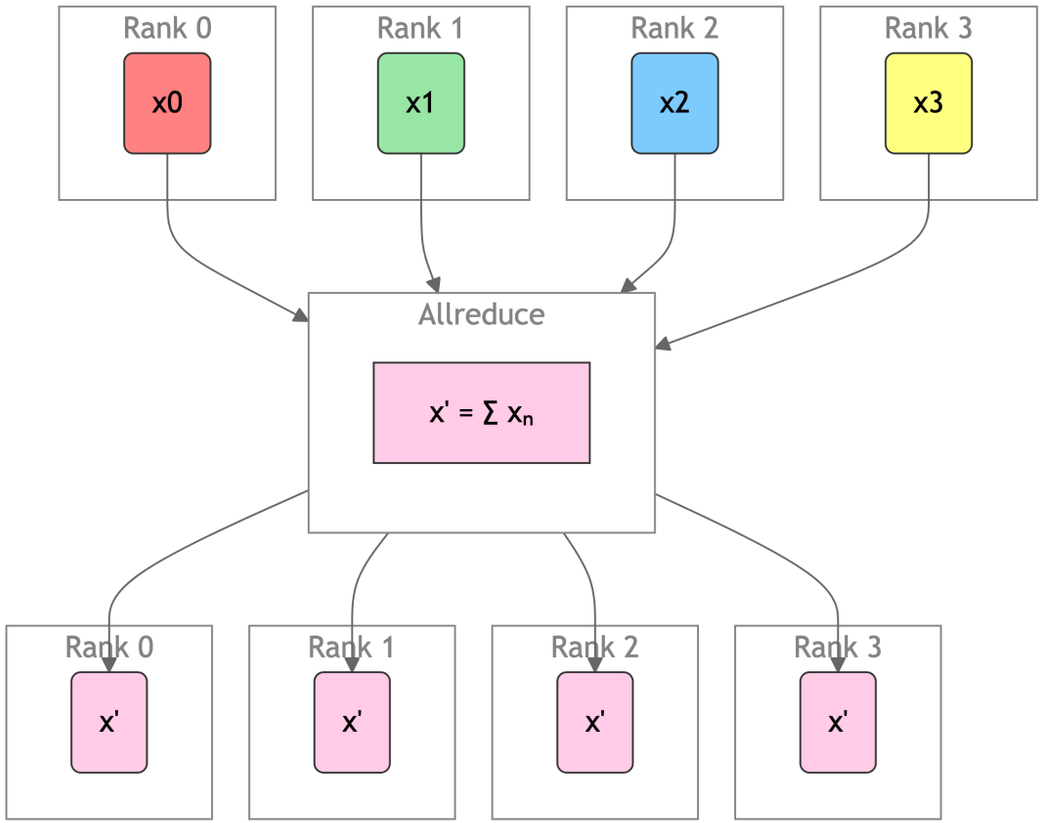



Figure 7: All-Reduce operation: each rank receives the reduction of input values across ranks.

# Reduce

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

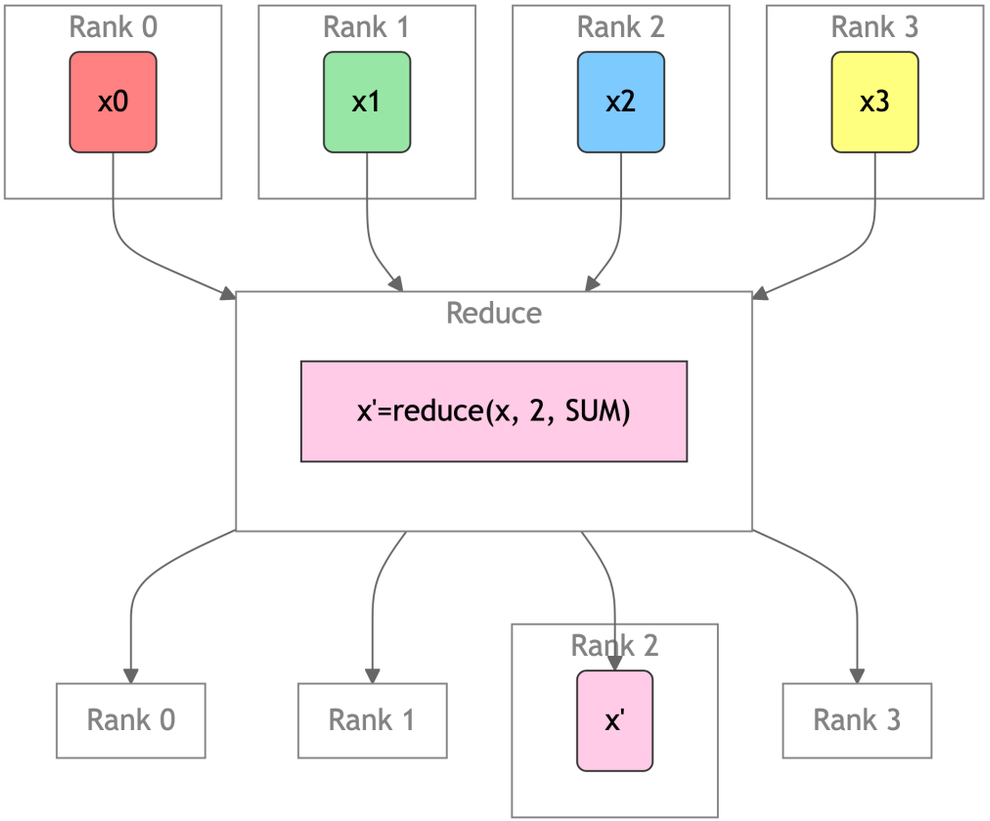



Figure 8: Reduce operation: one rank receives the reduction of input values across ranks

# Broadcast

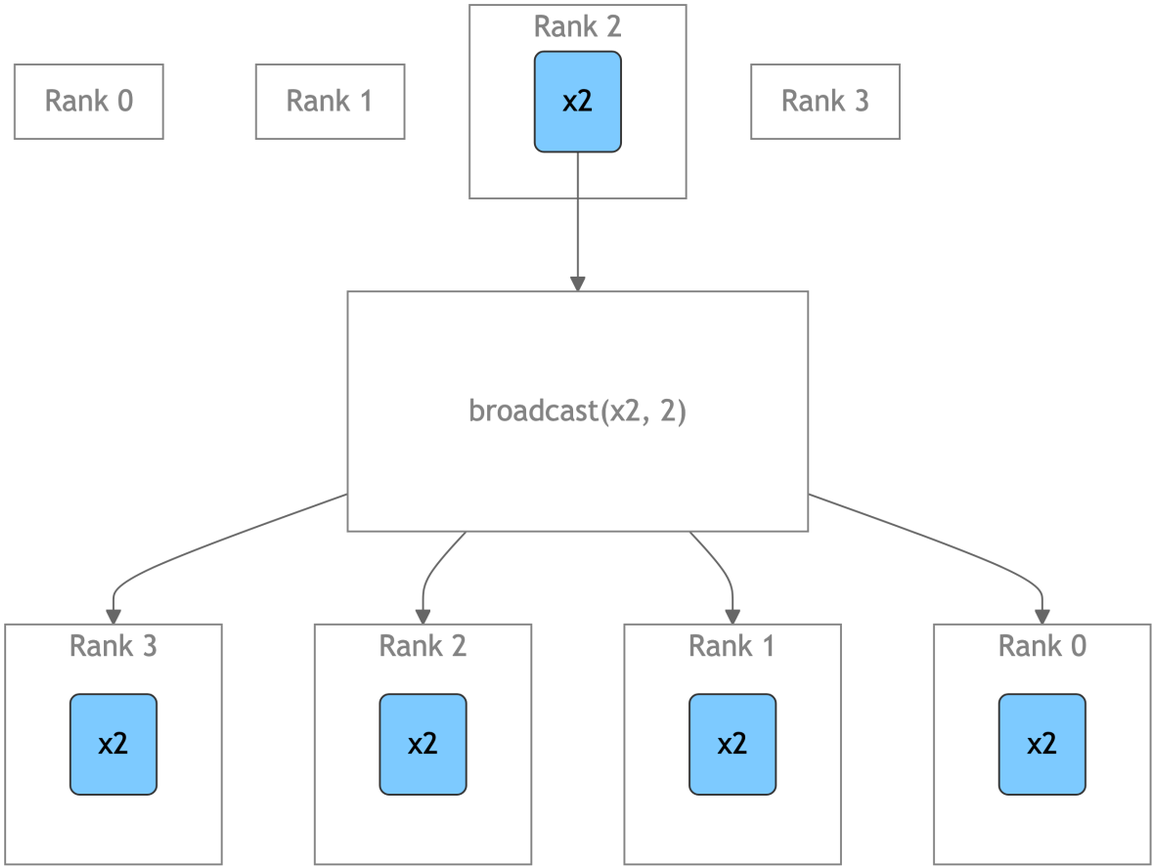



Figure 9: `broadcast` (send) a tensor $x$ from one rank to all ranks

# AllGather

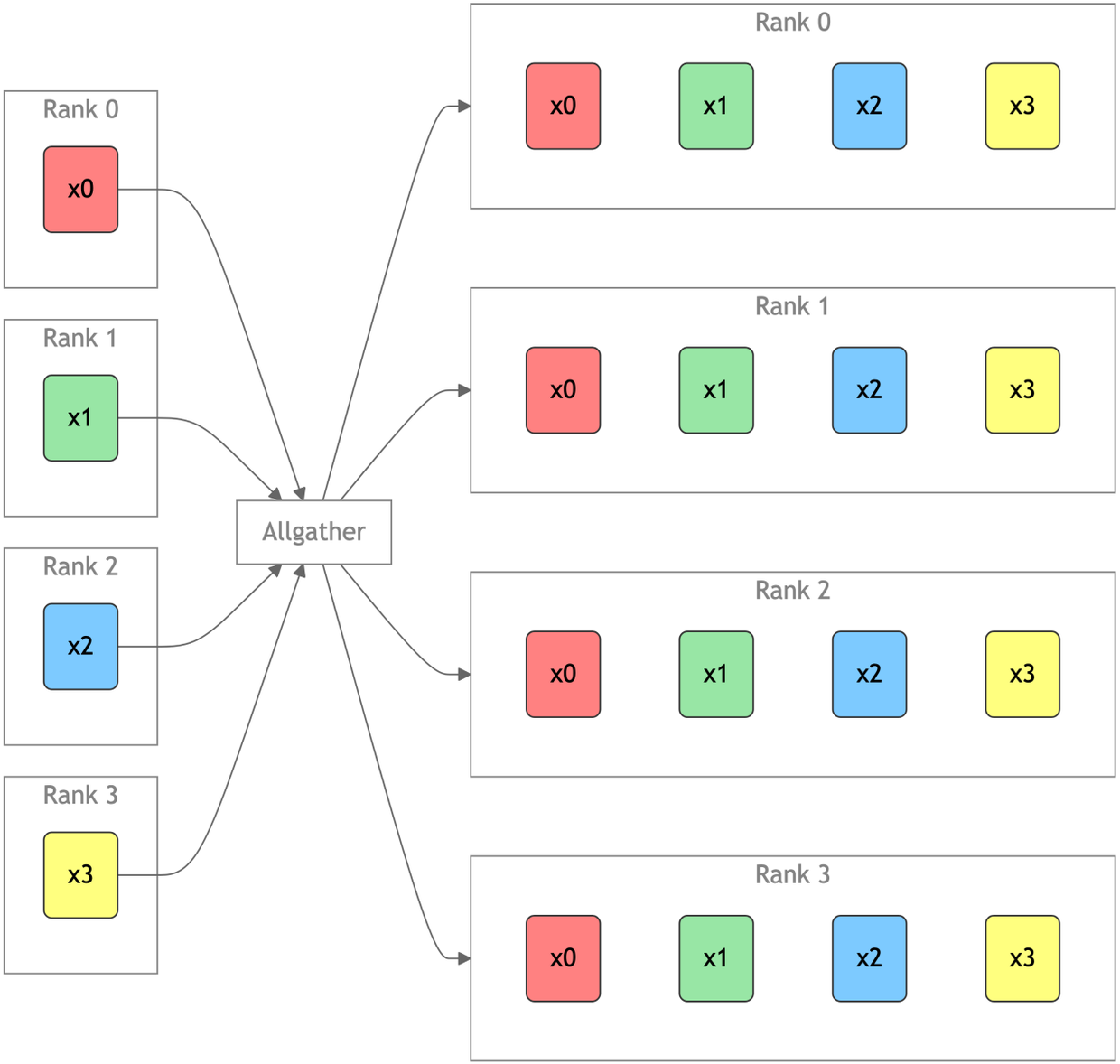



Figure 10: Gathers tensors from the whole group in a list.

# Scatter

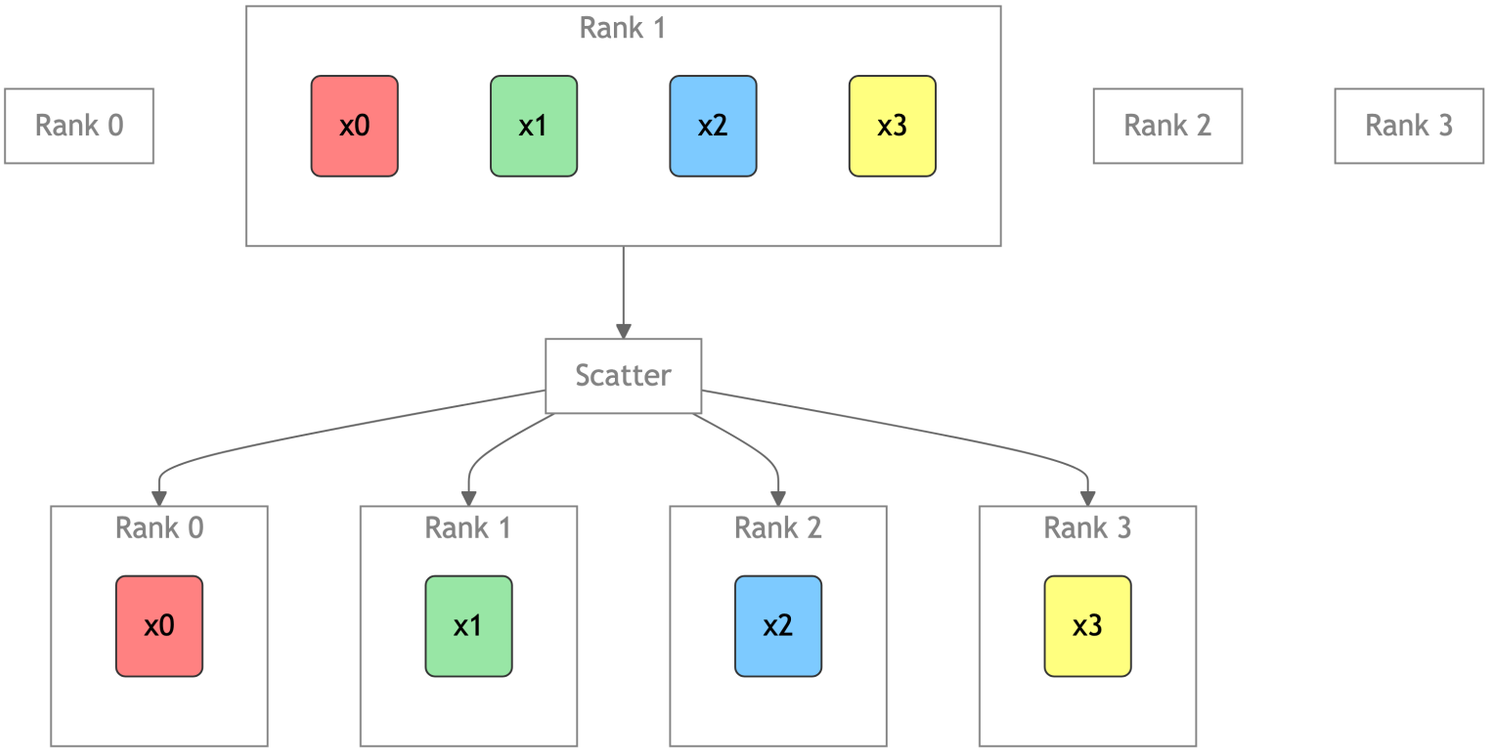



Figure 11: Scatters a list of tensors to the whole group

# ϟ Why Distributed Training?

- `N` workers each processing unique batch[1] of data:
  - `[micro_batch_size = 1]` $\times$ `[N GPUs]` $\longrightarrow$ $[\mathbf{global\_batch\_size} = \mathbf{N}]$
- Improved gradient estimators
  - Smooth loss landscape
  - Less iterations needed for same number of epochs
    - common to scale learning rate `lr *= 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

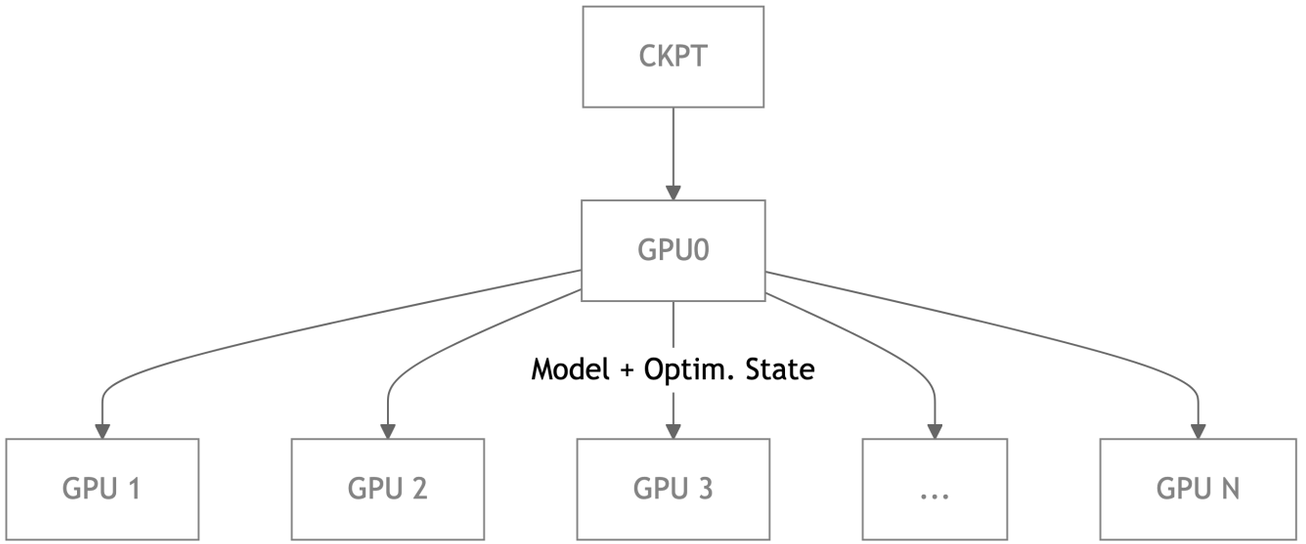



Figure 12: To ensure all workers have the same copies, we load on `RANK==0` and `broadcast`

# Best Practices

### ⏱ Keeping things in Sync

**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
    - 🔥 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:
  1. **Stage 1**: optimizer states $(P_{\text{os}})$
  2. **Stage 2**: gradients + opt. states $(P_{\text{os+g}})$
  3. **Stage 3**: model params + grads + opt. states $(P_{\text{os+g+p}})$

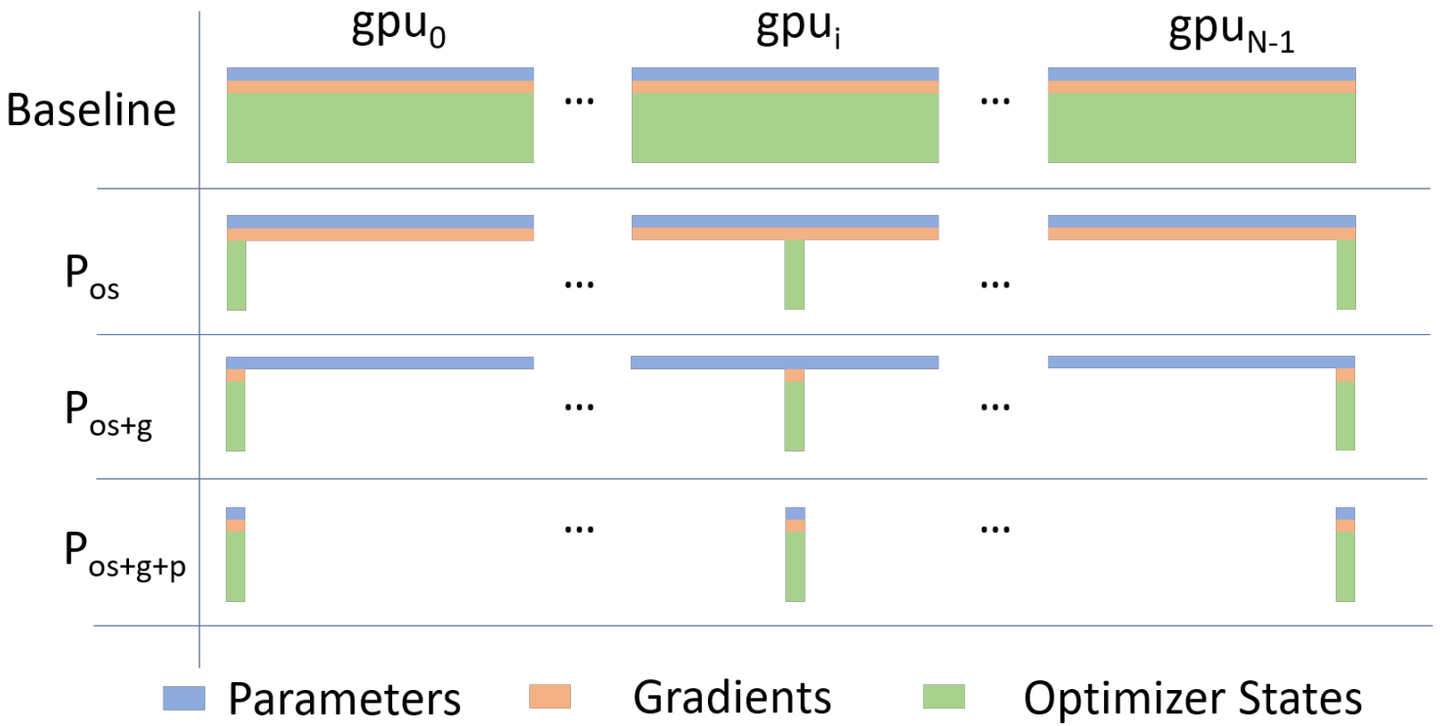



Figure 13: DeepSpeed + ZeRO

# 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 to CPU
- [Introducing PyTorch Fully Sharded Data Parallel (FSDP) API | PyTorch]

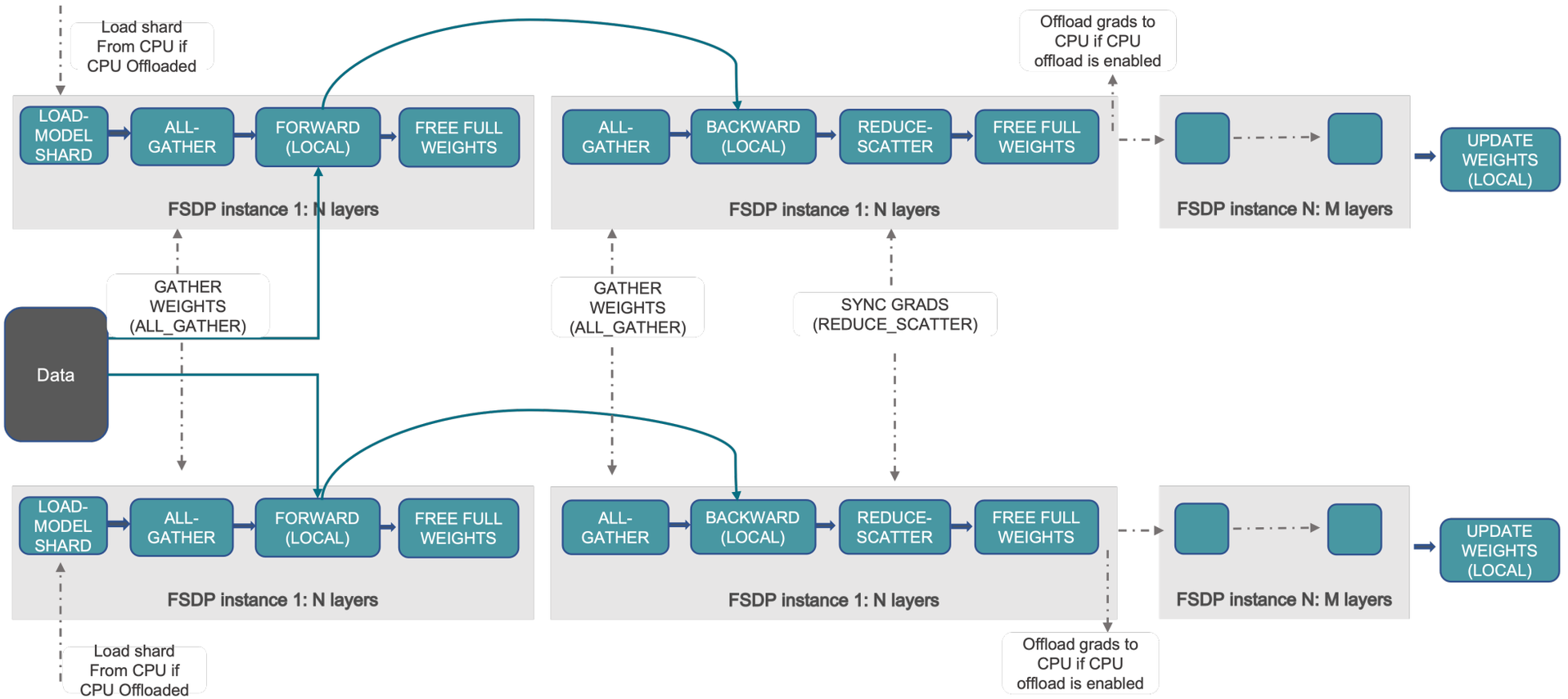



Figure 14: FSDP Workflow. Source

# 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)

# 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

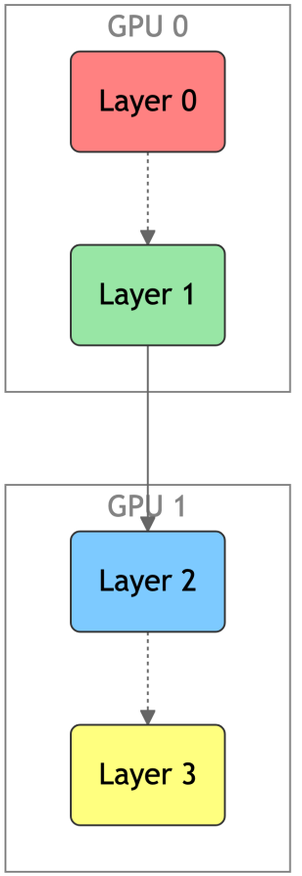



Figure 15: Pipeline Parallelism

# 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

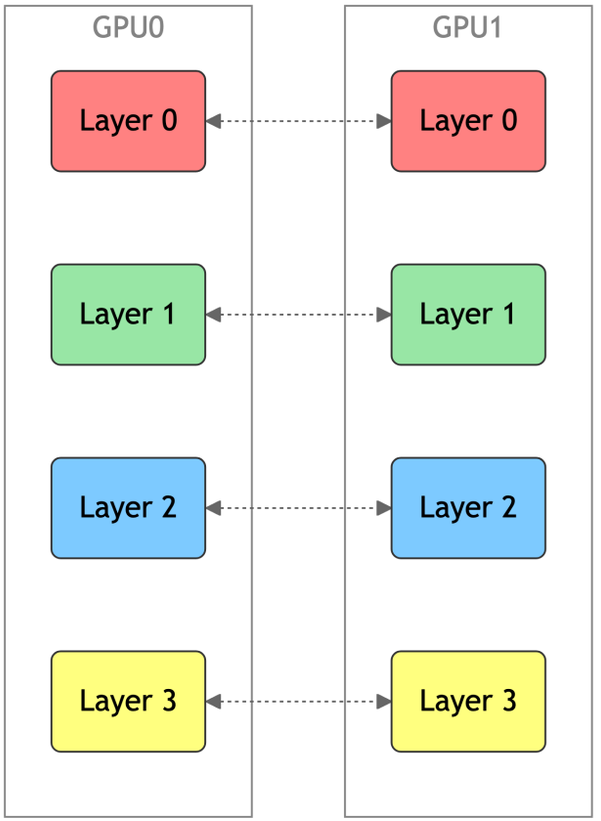



Figure 16: Tensor Parallel Training

# 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

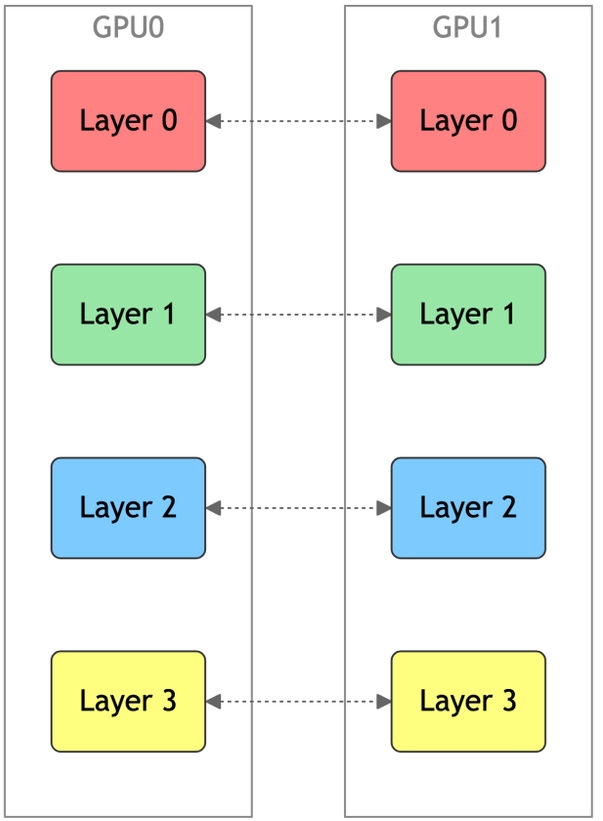



Figure 17: Tensor Parallel Training

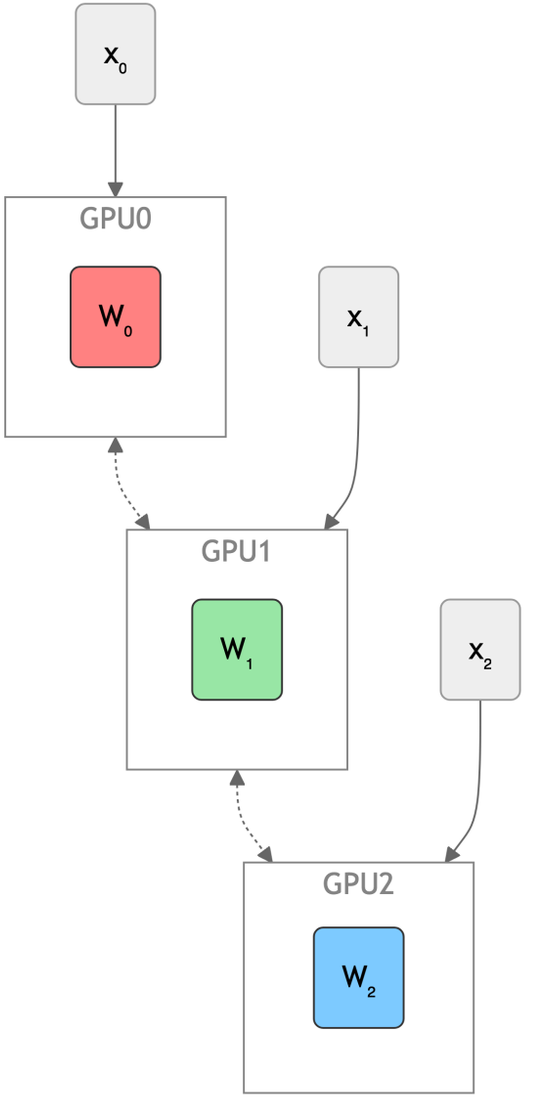
# Tensor (/ Model) Parallel Training: Example

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

1. Compute: $x_0 W_0 \rightarrow$ `GPU1`
2. Compute: $x_0 W_0 + x_1 W_1 \rightarrow$ `GPU2`
3. Compute: $y = \sum_i x_i W_i$ ✅



# Tensor (Model) Parallelism[1]

- In **Tensor Paralleism** each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.
  - The main building block of any transformer is a fully connected `nn.Linear` followed by a nonlinear activation GeLU.
    - `Y = GeLU(XA)`, where X and Y are the input and output vectors, and A is the weight matrix.
  - If we look at the computation in matrix form, it's easy to see how the matrix multiplication can be split between multiple GPUs:

1. Efficient Large-Scale Language Model Training on GPU Clusters

# Tensor Parallelism



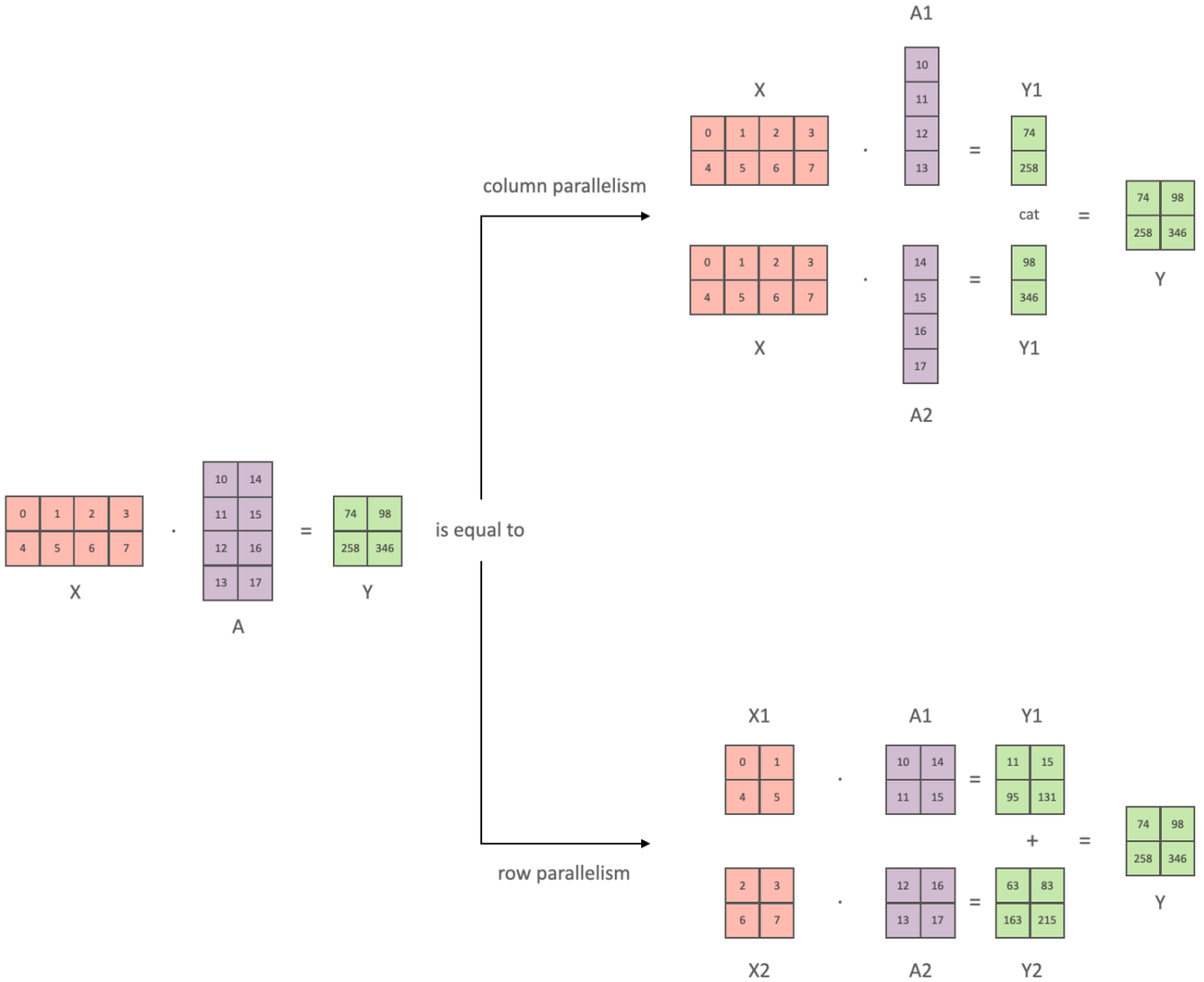



Figure 18: Tensor Parallel GEMM. This information is based on (the much more in-depth) TP Overview by @anton-l

# 3D Parallelism

- DP + TP + PP (3D) Parallelism

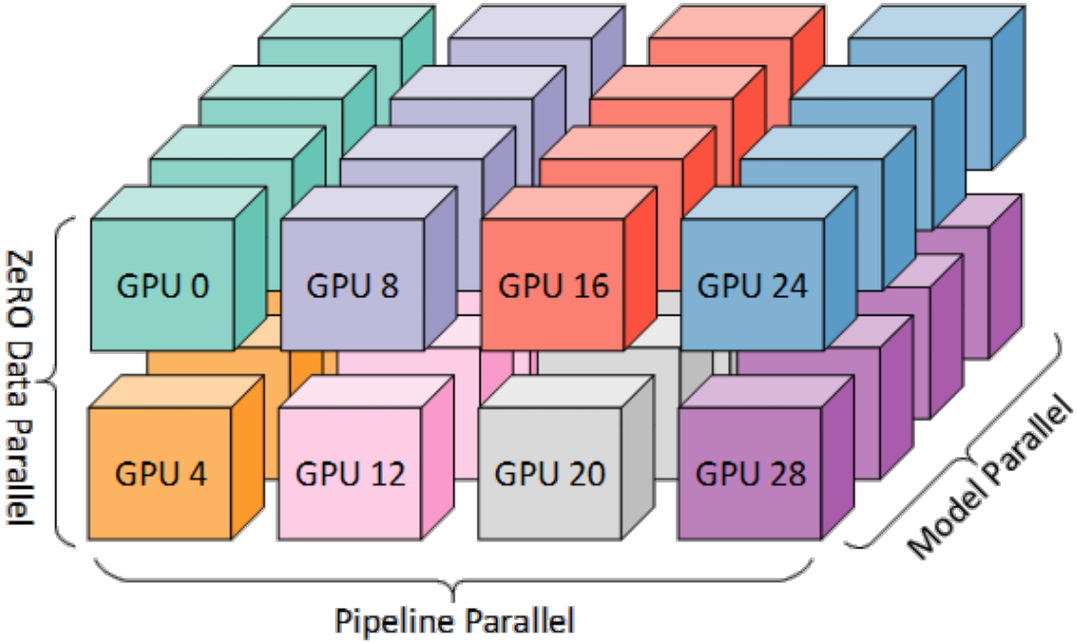



Figure 19: Figure taken from [3D parallelism: Scaling to trillion-parameter models]

# Deciding on a Parallelism Strategy

Single GPU | Single Node / Multi-GPU | Multi-Node / Multi-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.

# Large Language Models

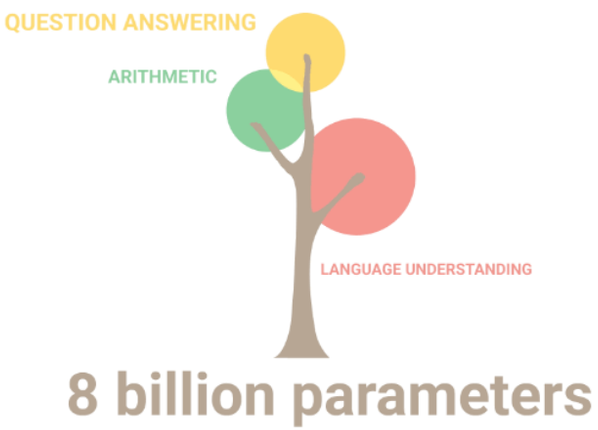



Figure 20: Large Language Models have (LLM)s have taken the ~~NLP community~~ **world** by storm[1].

1. Source: Hannibal046/Awesome-LLM

# Emergent Abilities

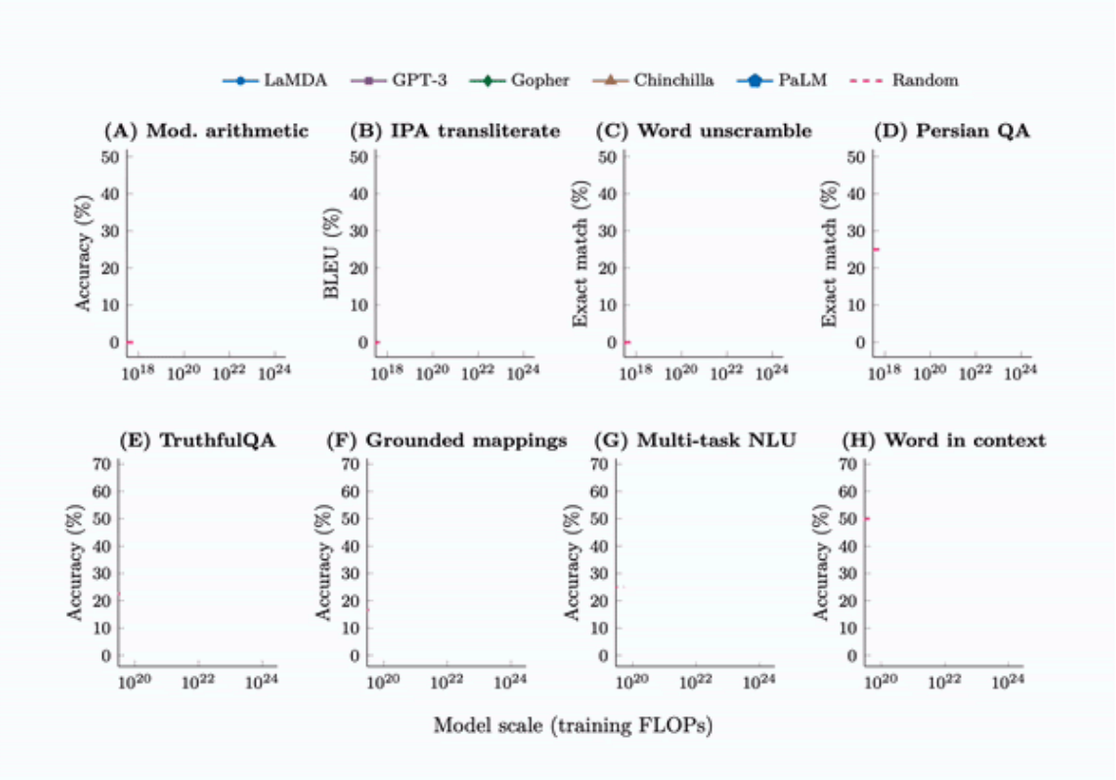



Figure 21: See Wei et al. (2022), Yao et al. (2023)

# Training LLMs

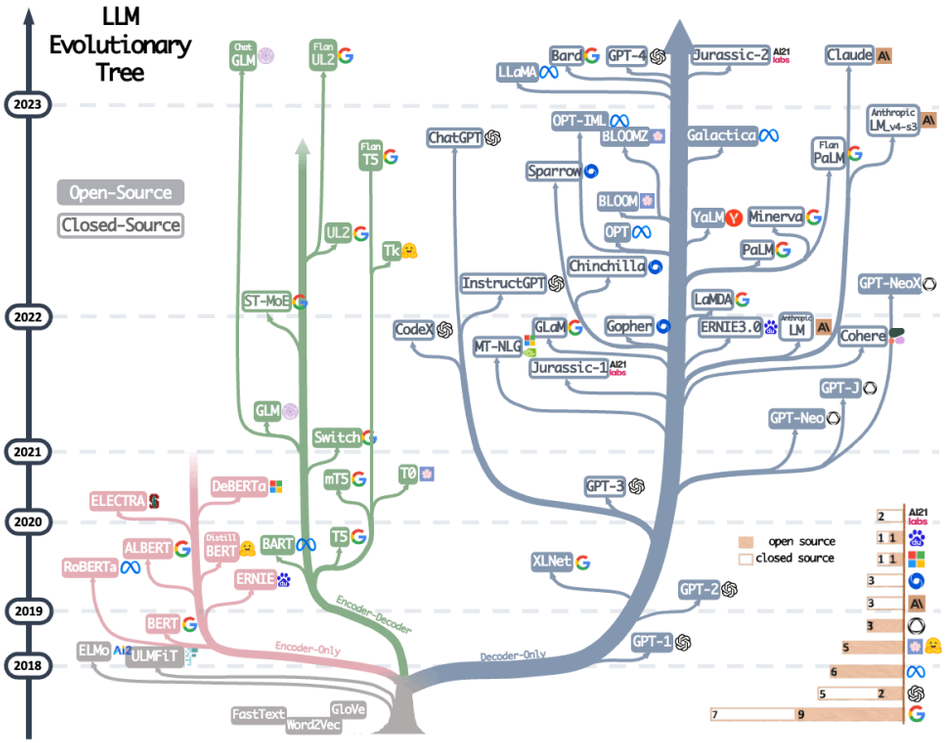



Figure 22: Visualization from Yang et al. (2023)

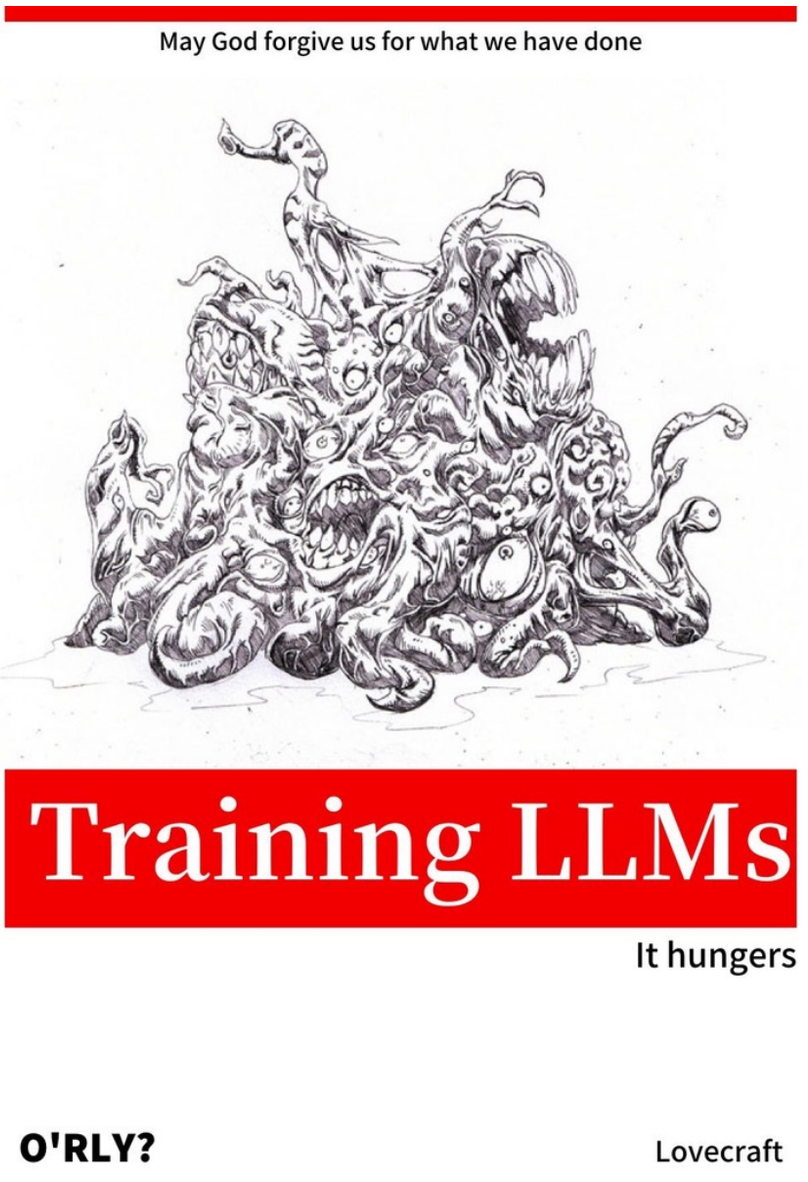



Figure 23: It's hungry! Wei et al. (2022)

# Life-Cycle of the LLM

1. Data collection + preprocessing
2. **Pre-training**
    - Architecture decisions, model size, etc.
3. Supervised Fine-Tuning
    - Instruction Tuning
    - Alignment
4. Deploy (+ monitor, re-evaluate, etc.)

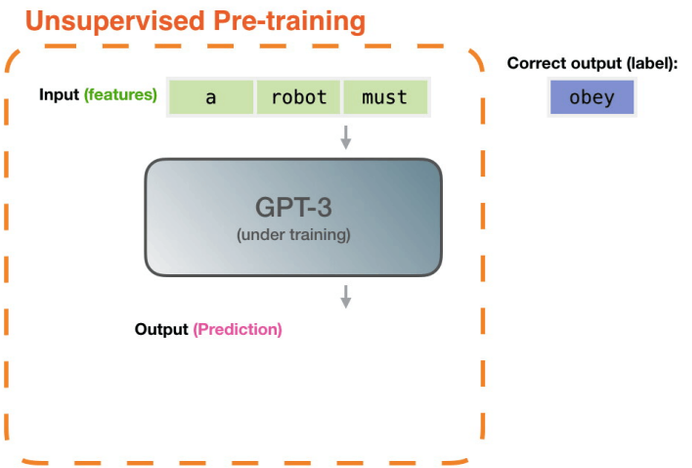



Figure 24: **Pre-training**: Virtually *all of the compute* used during pre-training[1].

1. Figure from The Illustrated Transformer

# 🎀 Life-Cycle of the LLM

1. Data collection + preprocessing
2. Pre-training
    - Architecture decisions, model size, etc.
3. **Supervised Fine-Tuning**
    - Instruction Tuning
    - Alignment
4. Deploy (+ monitor, re-evaluate, etc.)

1. Figure from The Illustrated Transformer

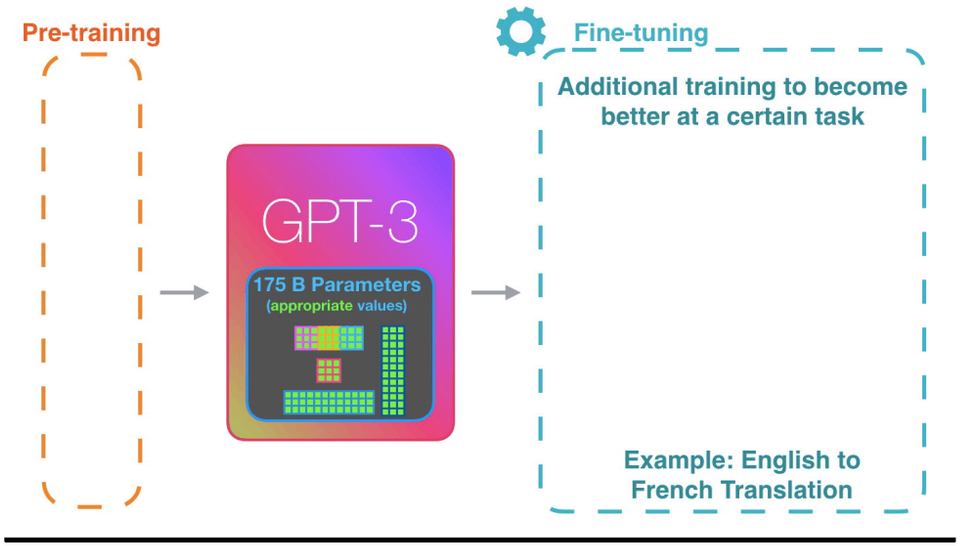
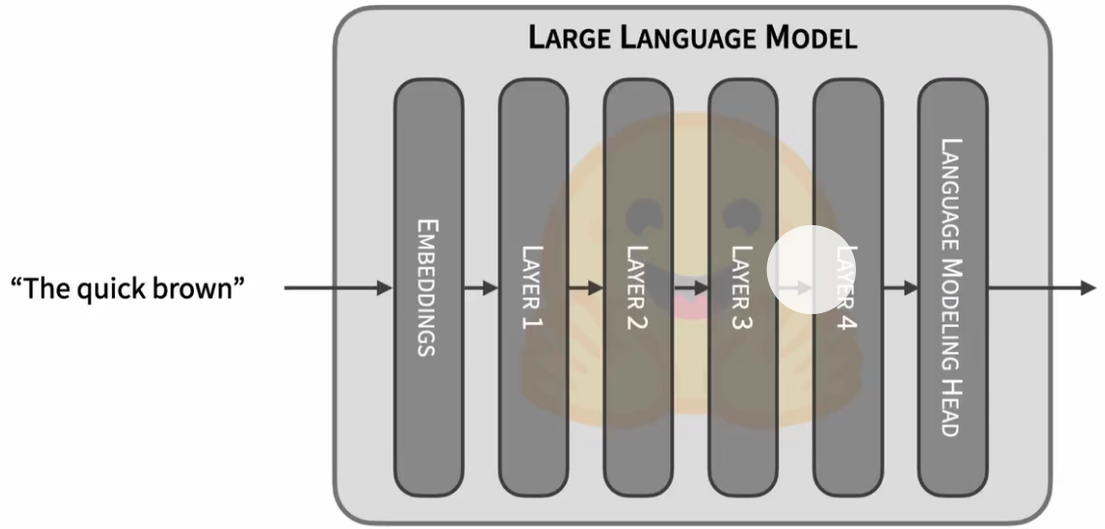



Figure 25: **Fine-tuning**: Fine-tuning actually updates the model's weights to make the model better at a certain task[1].

# ⏩ Forward Pass




0:00 / 0:09

Video

Figure 26: Language Model trained for causal language modeling[1].

1. Video from: 🤗 Generation with LLMs

# 💬 Generating Text

0:00 / 0:19

Video

Figure 27: Language Model trained for causal language modeling[1].

1. Video from: 🤗 Generation with LLMs

# 👋 Hands On

ai-science-training-series / 06_parallel_training

# 🧑‍💻 Hands On: Getting Started

1. 🌱 Clone Repo(s):
   - saforem2/wordplay

     ```
     1  git clone https://github.com/saforem2/wordplay
     2  cd wordplay
     ```

   - saforem2/ezpz

     ```
     1  git clone https://github.com/saforem2/ezpz deps/ezpz
     ```

2. 🐍 Setup Python:

   ```
   1  export PBS_O_WORKDIR=$(pwd) && source deps/ezpz/src/ezpz/bin/utils.sh
   2  ezpz_setup_python
   3  ezpz_setup_job
   ```

# 📦 Install {ezpz, wordplay}

1. Install Python packages:
    1. saforem2/ezpz:

        ```
        python3 -m pip install -e "./deps/ezpz" --require-virtualenv
        ```

    2. saforem2/wordplay:

        ```
        # from inside `wordplay/`
        python3 -m pip install -e . --require-virtualenv
        ```

2. Test distributed setup:

    ```
    mpirun -n "${NGPUS}" python3 -m ezpz.test_dist
    ```

    See: 🍋 ezpz/test_dist.py

# ezpz: Example [video]



Figure 28: Example: using 🍋 `ezpz.test_dist` to train a small model using DDP

# Install wordplay 🎮💬


Figure 29: The simplest, fastest repository for training / finetuning GPT based models. Figure from karpathy/ nanoGPT

# Prepare Data

```
$ python3 wordplay/data/shakespeare_char/prepare.py
Using HF_DATASETS_CACHE=/home/foremans/tmp/polaris-talk/2024-07-17-073327/wordplay/data/shakespeare_char/
length of dataset in characters: 1,115,394
all the unique characters:
 !$&\',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
vocab size: 65
train has 1,003,854 tokens
val has 111,540 tokens
```

# Launch Training (DDP)

```
launch python3 -m wordplay \
    train.backend=DDP \
    train.eval_interval=100 \
    data=shakespeare \
    train.dtype=bf16 \
    model.batch_size=64 \
    model.block_size=1024 \
    train.max_iters=1000 \
    train.log_interval=10 \
    train.compile=false \
    | tee wordplay-gpt2-DDP.log
```

# Training: Example Output

```
$ launch python3 -m wordplay \
    train.backend=DDP \
    train.eval_interval=100 \
    data=shakespeare \
    train.dtype=bf16 \
    model.batch_size=64 \
    model.block_size=1024 \
    train.max_iters=1000 \
    train.log_interval=10 \
    train.compile=false \
    | tee wordplay-gpt2-DDP.log
[2024-07-17 07:42:11.746540][INFO][__init__:156] - Setting logging level to 'INFO' on 'RANK == 0'
[2024-07-17 07:42:11.748763][INFO][__init__:157] - Setting logging level to 'CRITICAL' on all others 'RAN
[2024-07-17 07:42:11.749453][INFO][__init__:160] - To disable this behavior, and log from ALL ranks (not
[2024-07-17 07:42:11.772718][INFO][configs:81] - Setting HF_DATASETS_CACHE to /home/foremans/tmp/polaris-
[2024-07-17 07:42:15.341532][INFO][dist:358] - [device='cuda'][rank=2/3][local_rank=2/3][node=0/0]
[2024-07-17 07:42:15.342381][INFO][dist:358] - [device='cuda'][rank=1/3][local_rank=1/3][node=0/0]
[2024-07-17 07:42:15.342430][INFO][dist:358] - [device='cuda'][rank=3/3][local_rank=3/3][node=0/0]
[2024-07-17 07:42:15.348657][INFO][dist:95] -
```

# wordplay: Example [video]

Figure 30: Training a LLM to talk like Shakespeare using saforem2/wordplay 🎮💬

# ❤️ Thank you!

- Organizers
- Feel free to reach out!

**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

# References


- Title slide (Tetris animation) from: https://emilhvitfeldt.github.io/quarto-iframe-examples/tetris/index.html

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.

Yang, Jingfeng, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. "Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond." https://arxiv.org/abs/2304.13712.

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/