Large Scale & Distributed Optimization

LauzHack Deep Learning Bootcamp
2024-07-19

Atli Kosson

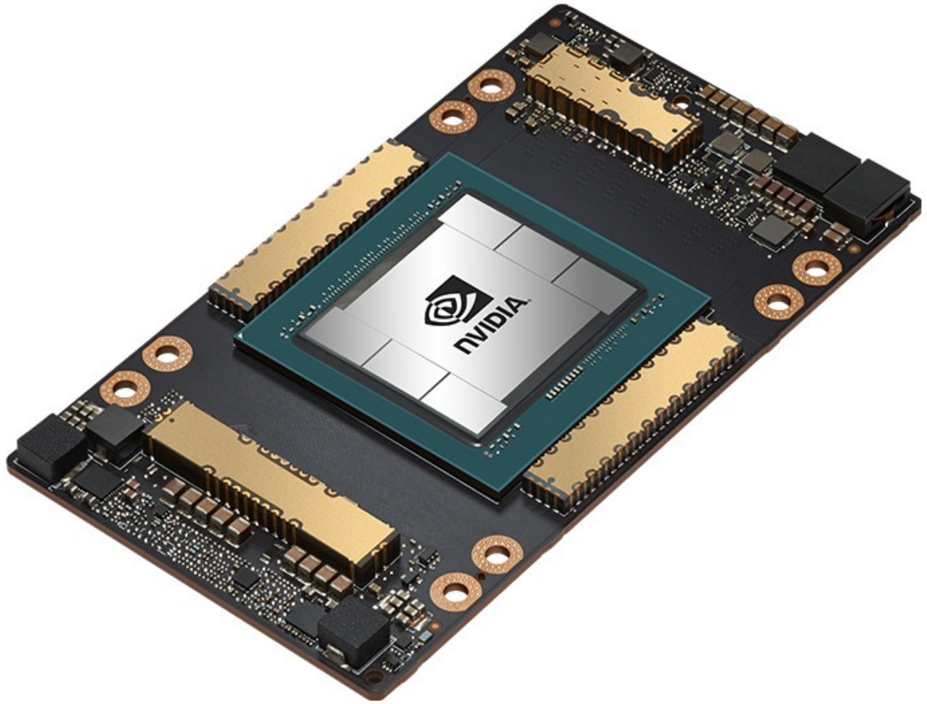
Outline

- Part 1: Efficient Single GPU Training
  - GPUs
  - Mixed Precision Training
  - Mini-batch Parallel Training
  - The Critical Batch Size / Limits of Parallelization
  - Hyperparameters
- Part 2: Distributed Training

GPUs: Overview

- Graphical Processing Units - Most common hardware for training neural networks

- Nvidia A100 - Typical High End GPU:
  - 312 TFLOPS (fp16)
  - 80GB HBM2e
  - Memory bandwidth 2,039 GB/s
  - 400W
  - ~$20k



- GPUs are made up of many small cores

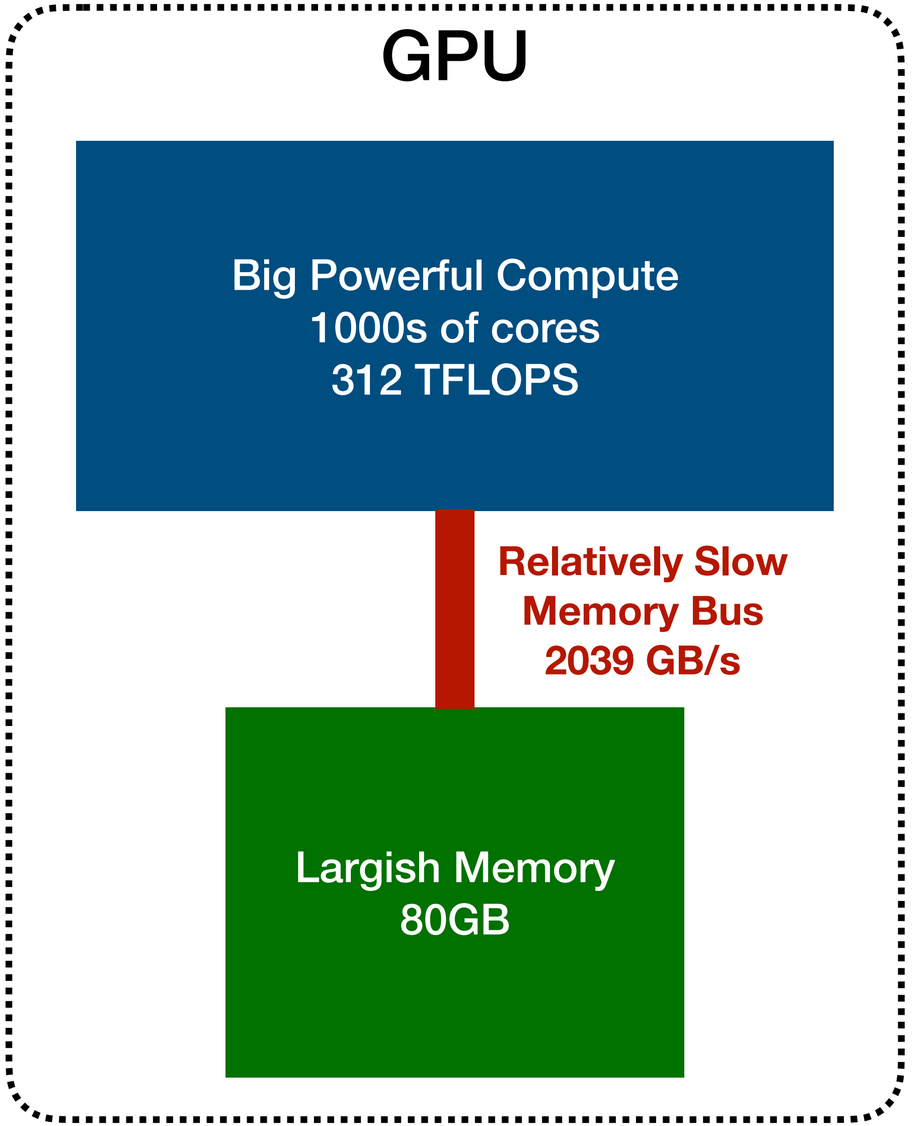
- Good for highly parallel workloads

GPUs: Bottlenecks

- GPUs have powerful compute and large memory but can only access it slowly

- Best for large compute heavy operations
  - Do multiple operations per memory load
  - Good: Matrix-Matrix Multiplication
  - Bad: Element-wise operations (e.g. addition)
  - Bad: Operations on many small inputs

- Tips for good performance:
  - Use large batch sizes (parallelize)
  - Fuse small and element-wise operations (torch.compile)
  - Pre-load data to avoid stalling the GPU



Mixed Precision Training: Floating Point Formats

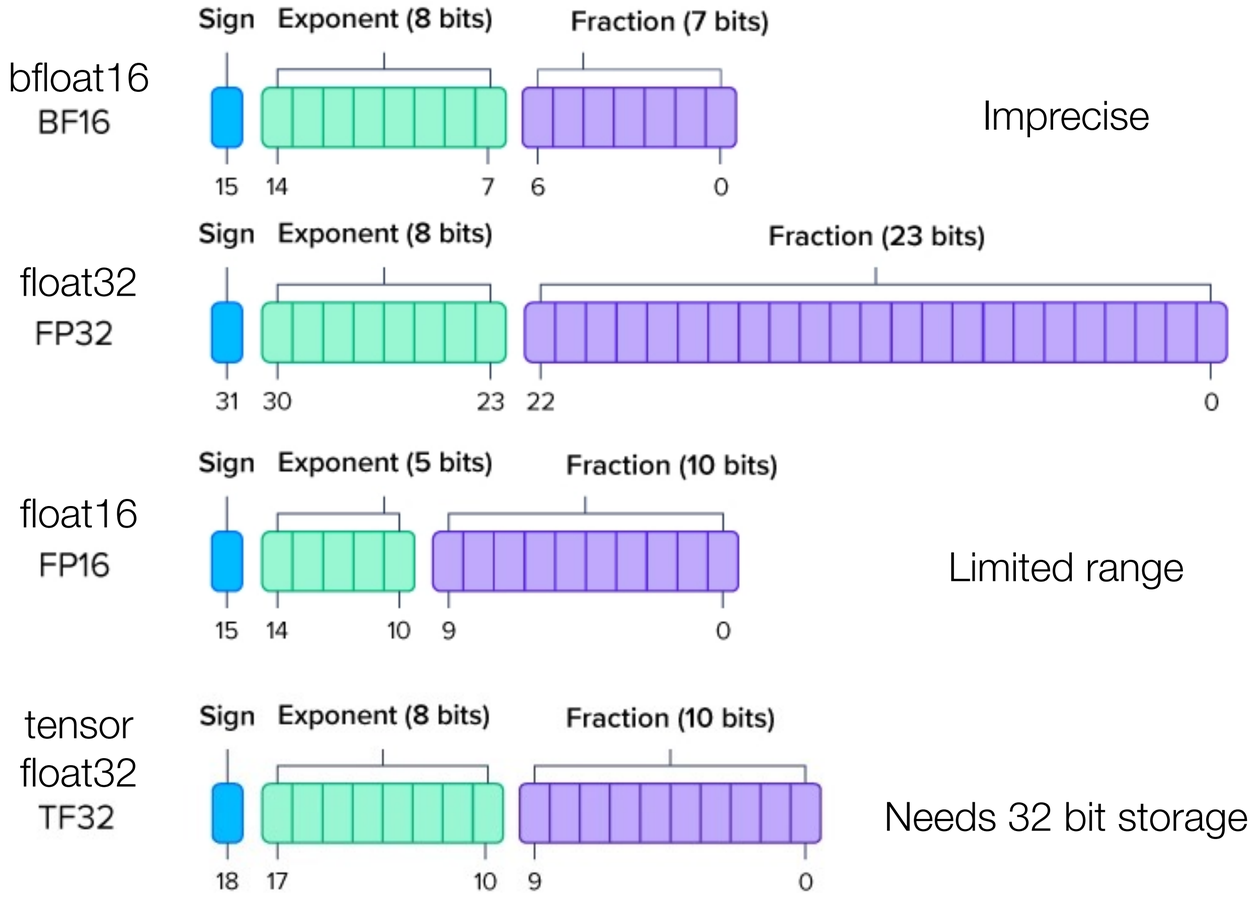
- Computers approximate real numbers with floating point numbers
- Correspond to the scientific notation in decimal for example $\pm 1.23 \cdot 10^{-3}$
- Formats allow for different numbers of significant digits and exponent range
- Get numerical approximation errors (e.g. with 2 significant digits and exponent in $[-2,2]$)
  - Rounding: $10/3 + 10/3 + 10/3 = 3.3 + 3.3 + 3.3 = 9.9 \neq 10.0$
  - Swamping: $\sum_{i=1}^{1000} 1.0 = 1.0 \cdot 10^2 \neq 1000$ if done directly
    because with limited precision $1.0 \cdot 10^2 + 1.0 \cdot 10^0 = 1.0 \cdot 10^2$
  - Underflow: $0.01/10^2 = 0.0$
- Less precise formats are cheaper to load and operate on => speedup

Mixed Precision Training: Floating Point Formats

- A100 performance:
  - fp32 19.5 TFLOPS
  - fp16 312 TFLOPS
  - bf16 312 TFLOPS
  - tf32 156 TFLOPS

- Using lower precision formats can be much faster but may result in numerical issues

- Mixed Precision Training uses different formats depending on the type of operation, resulting in speedups while (hopefully) avoiding the downsides



Img src: https://www.exxactcorp.com/blog/hpc/what-is-fp64-fp32-fp16

Mixed Precision Training: In Practice

```
use_amp = True

net = make_model(in_size, out_size, num_layers)
opt = torch.optim.SGD(net.parameters(), lr=0.001)
scaler = torch.cuda.amp.GradScaler(enabled=use_amp)

start_timer()
for epoch in range(epochs):
    for input, target in zip(data, targets):
        with torch.autocast(device_type=device, dtype=torch.float16, enabled=use_amp):
            output = net(input)
            loss = loss_fn(output, target)
        scaler.scale(loss).backward()
        scaler.step(opt)
        scaler.update()
        opt.zero_grad(set_to_none=True)
```

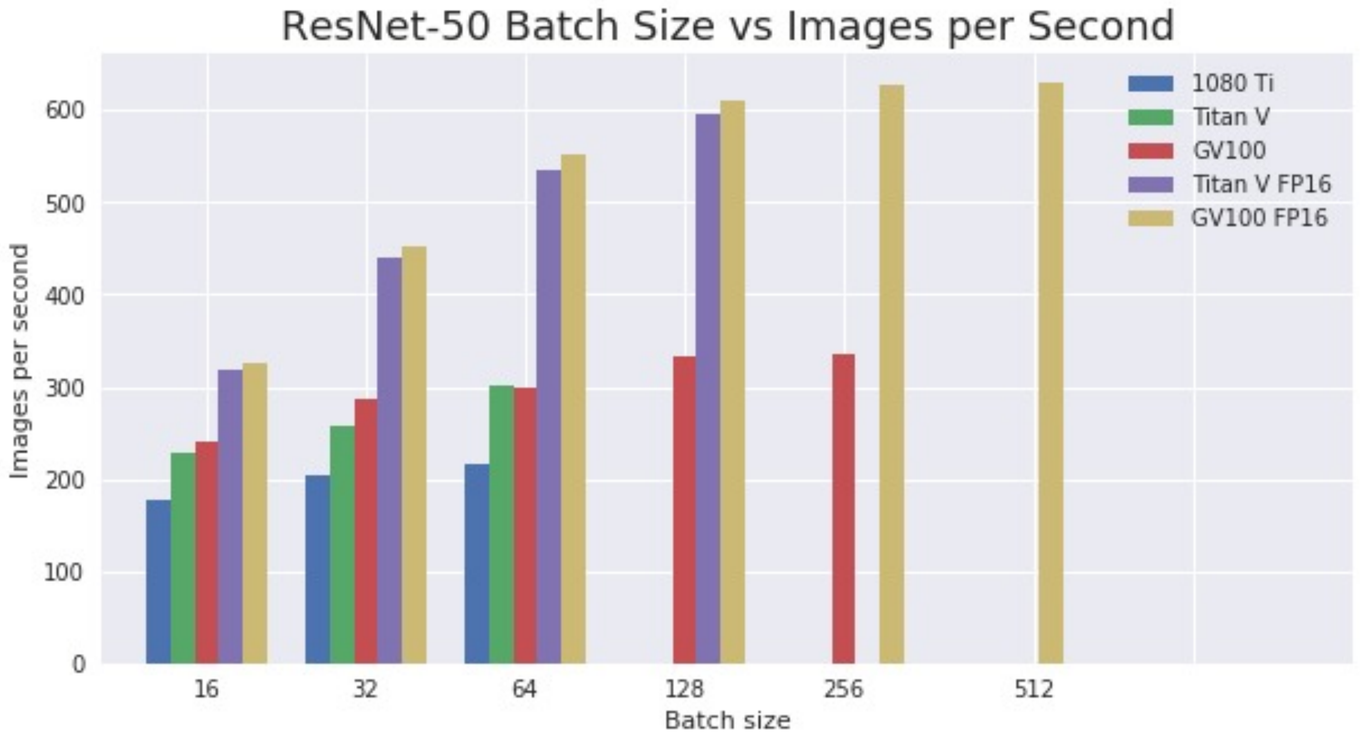
- PyTorch handles most of this for you:
  - torch.autocast
  - torch.cuda.amp.GradScaler (needed for float16, but not bfloat16)
  - TF32 is used by default for some ops if available but need to enable for matmuls with torch.backends.cuda.matmul.allow_tf32 = True

- Note: Some extra memory overhead and optimizer states are all kept in float32

Tutorial: GPU Efficiency

Mini-batch Parallel Training

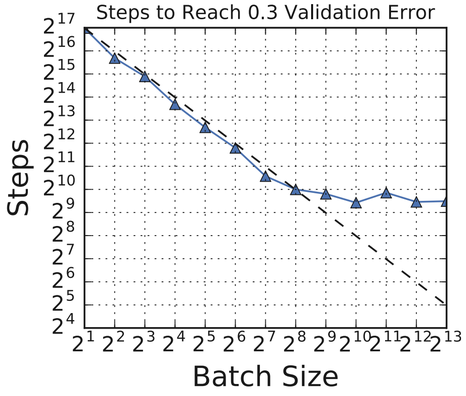
- Using mini-batches instead of batch size of one is a simple form of data parallel training
- Helps with GPU utilization:
  - More data reuse in matmuls
  - Fewer optimization steps
  - Fewer kernel launches
- Classical recommendation: Go with the largest batch size that fits on your GPU
- However, this is not always efficient!



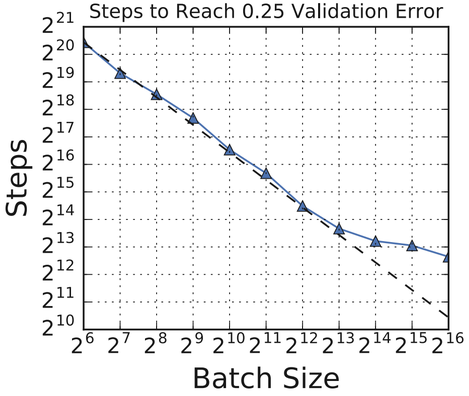
Img src: https://www.pugetsystems.com/labs/hpc/gpu-memory-size-and-deep-learning-performance-batch-size-12gb-vs-32gb-1080ti-vs-titan-v-vs-gv100-1146/

The Critical Batch Size - Limits Effective Parallelization

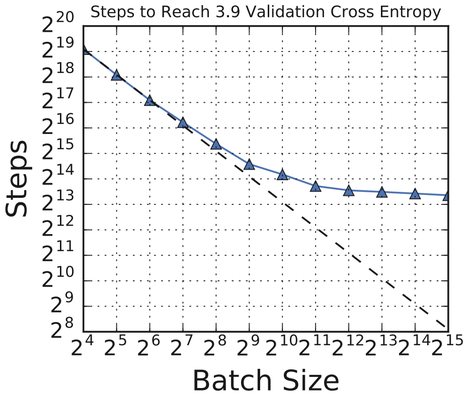
- Beyond a certain size larger batch sizes are not effective, the number of steps required doesn't decrease significantly while each step is more expensive
- Larger batch sizes give better gradient estimates, beyond a certain point this is no longer cost effective



(c) ResNet-8 on CIFAR-10



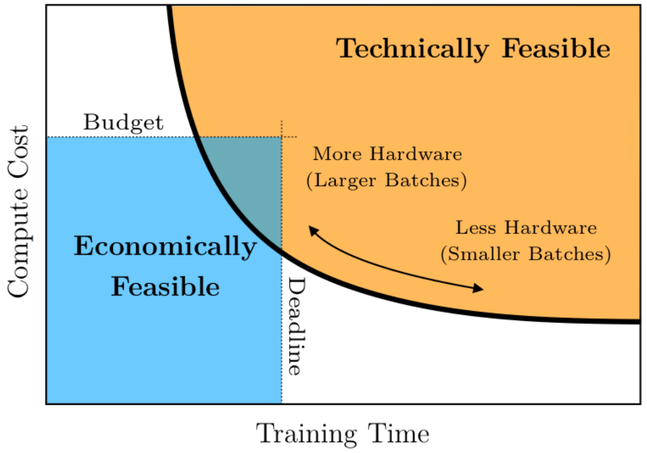
(d) ResNet-50 on ImageNet

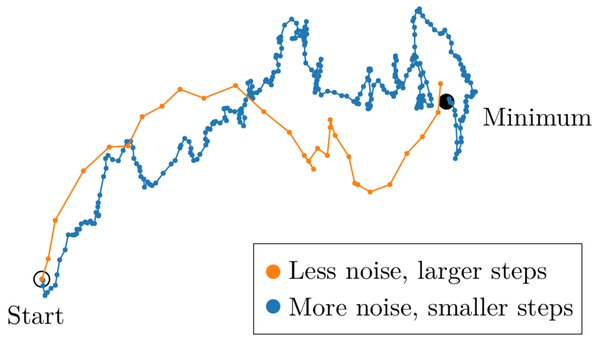


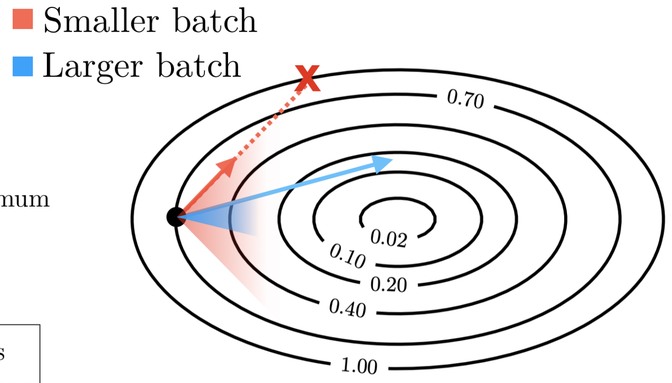
(f) Transformer on LM1B

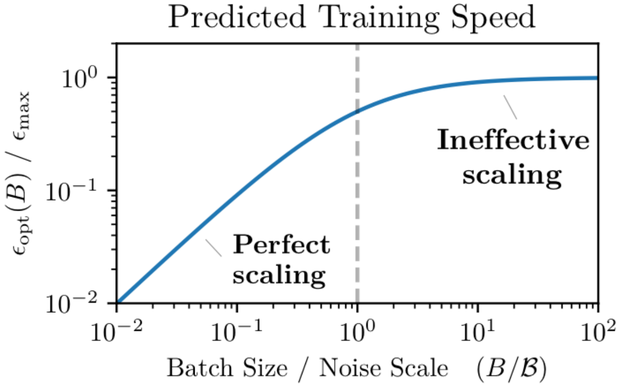
Measuring the Effects of Data Parallelism on Neural Network Training
arxiv.org/abs/1811.03600

An Empirical Model of Large-Batch Training
arxiv.org/abs/1812.06162



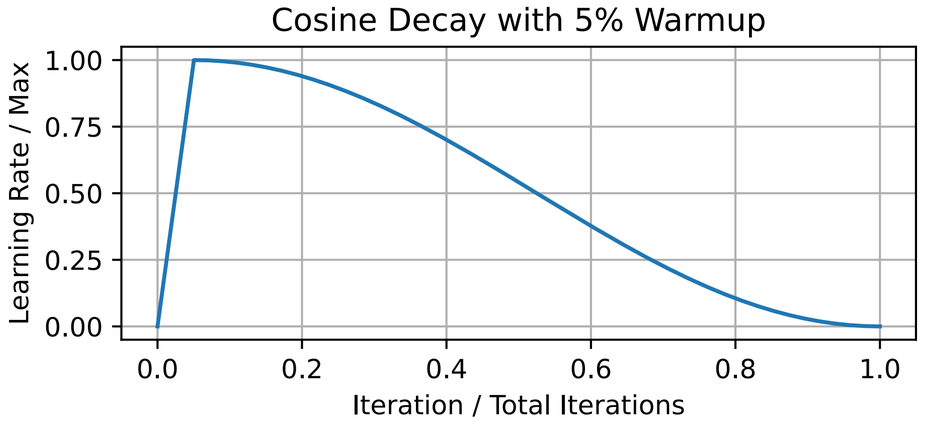






Hyperparameters

- AdamW + Cosine decay schedule with warmup
- Learning rate η should increase with larger batch sizes, roughly up to the critical batch size
- Weight decay λ should increase or stay the same
- My recommendation: $\eta \propto \sqrt{B}$ and $\lambda \propto \sqrt{B}$
- Momentum β_1 should be high (>= 0.9)
- Second moment β_2 may need to be lower (~0.95)



AdamW

$$\boldsymbol{m}_t = \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t$$
$$\boldsymbol{v}_t = \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2$$
$$\boldsymbol{p}_t = \boldsymbol{p}_{t-1} - \eta \cdot \left(\frac{\boldsymbol{m}_t/(1-\beta_1^t)}{\sqrt{\boldsymbol{v}_t/(1-\beta_2^t)}+\varepsilon} + \lambda \boldsymbol{p}_{t-1} \right)$$

Outline

- Part 1: Efficient Single GPU Training
- Part 2: Distributed Training
  - The need for parallelism
  - What to parallelize - Optimizer + Model
  - Nodes and clusters
  - Megatron case study of different types of parallelism (data, tensor, pipeline)
  - Alternative methods

How do we efficiently train large models?

- Exponential growth in model size & compute

- System challenges:
  - Compute: 288 V100 GPU years for GPT-3
  - Memory: >100 V100s to store GPT-3 state
  - Bandwidth: Sync large tensors fast

- Will mostly focus on the Megatron Paper:
  - Study how to parallelize training effectively
  - Pipeline, tensor and data parallelism combo
  - Demonstrate 1 trillion parameter training

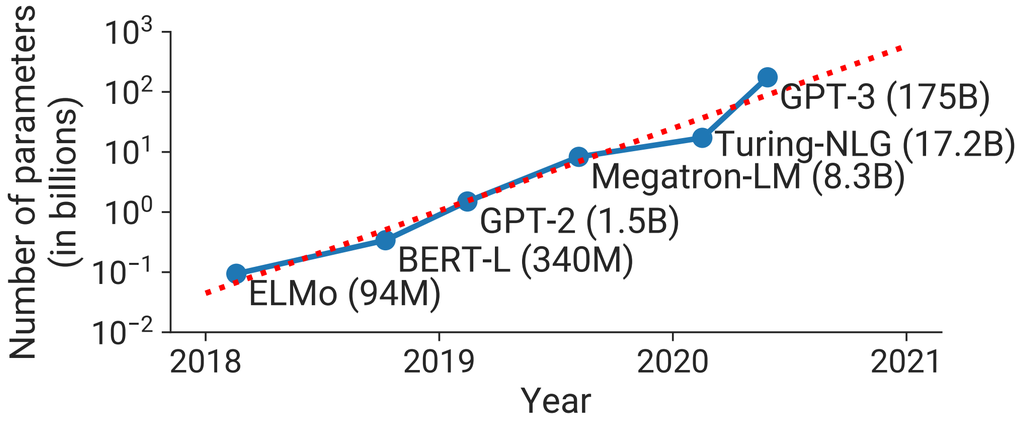


Figure 1 from arXiv:2104.04473 (PTD-P)

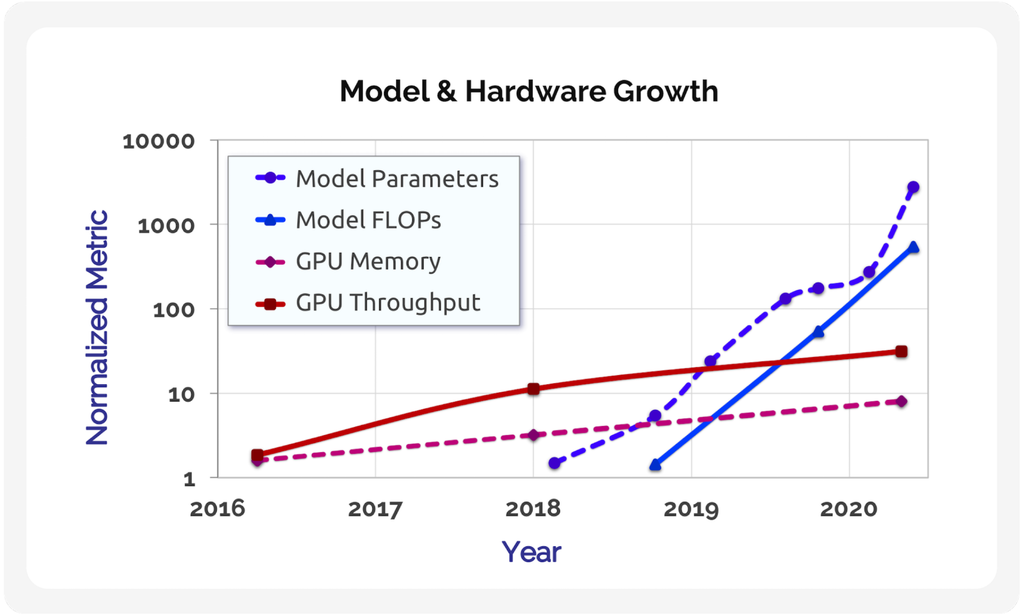


Figure 19 from arXiv:2108.07258 (Foundation Models)

Compute Requirements

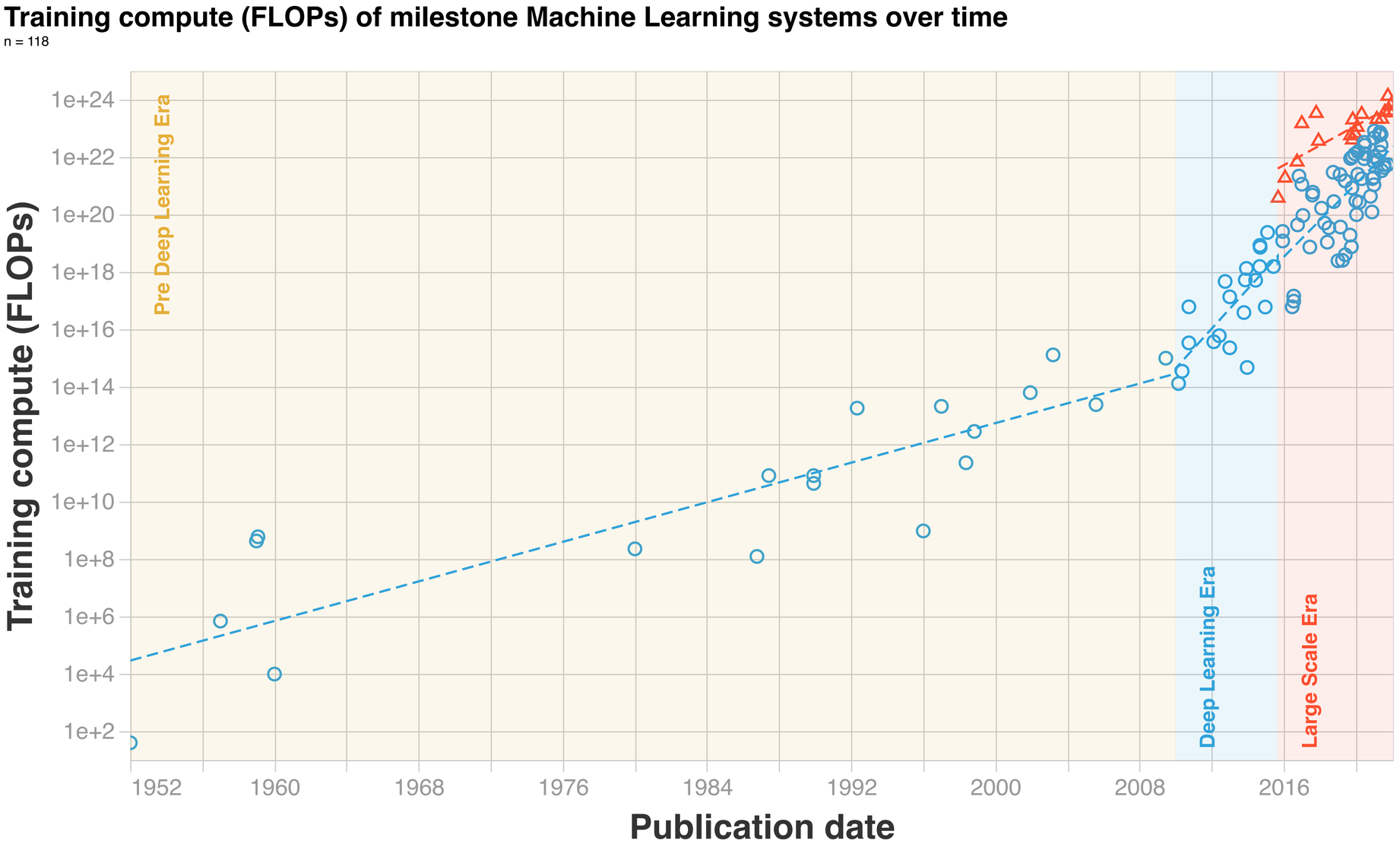


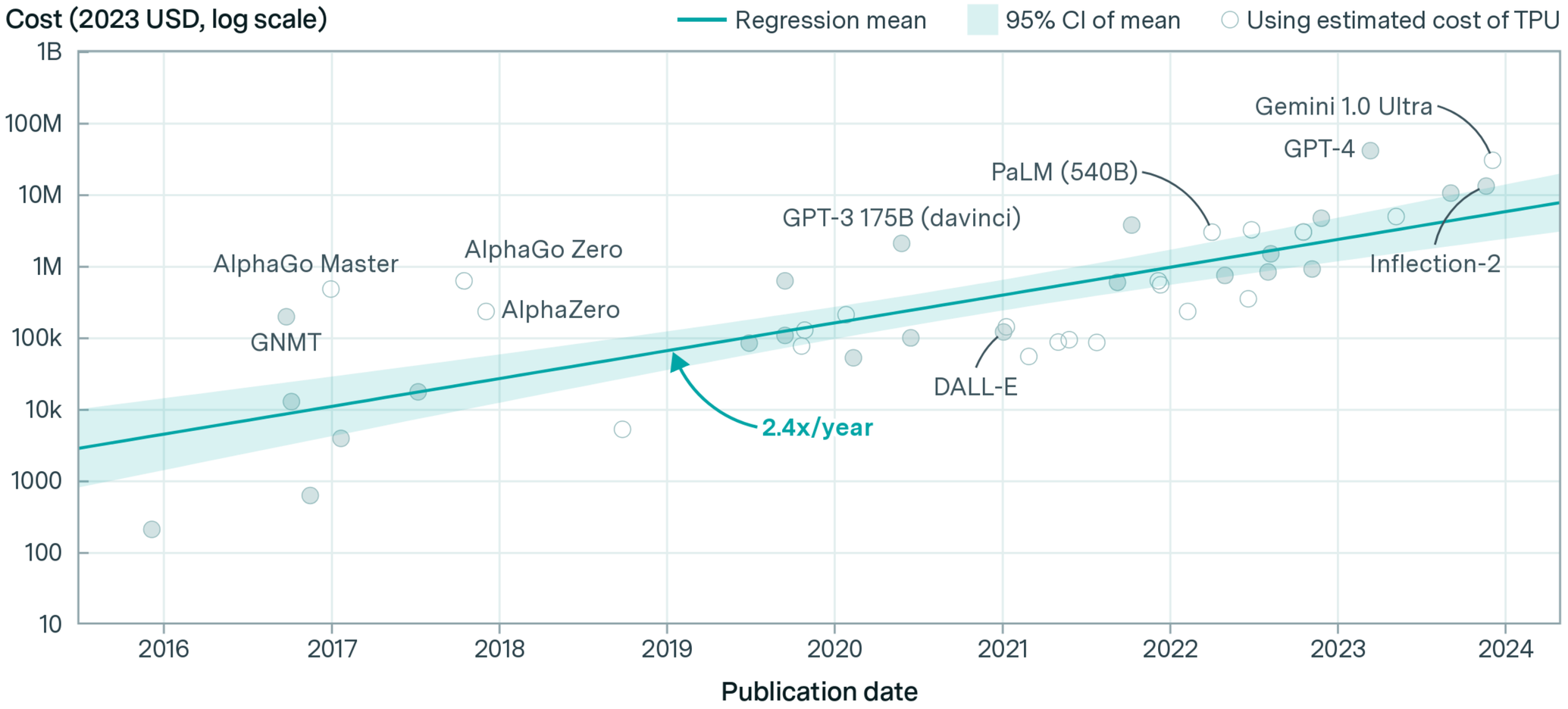
Figure 1: Trends in $n = 118$ milestone ML models between 1952 and 2022. We distinguish three eras. Notice the change of slope circa 2010, matching the advent of Deep Learning; and the emergence of a new large-scale trend in late 2015.

https://arxiv.org/abs/2202.05924

Cost of Training

Amortized hardware and energy cost to train frontier AI models over time

EPOCH AI



https://epochai.org/blog/how-much-does-it-cost-to-train-frontier-ai-models

What we want to run: Model and Optimizer

- Model - Stacked transformer blocks:
  - Attention subblock
  - MLP subblock

- Mixed precision training:
  - float16 / bfloat16 activations + weights
  - Optimizer state in float32

- Optimizer - AdamW:
  - First gradient moment
  - Second gradient moment
  - 20 bytes / parameter with mixed precision

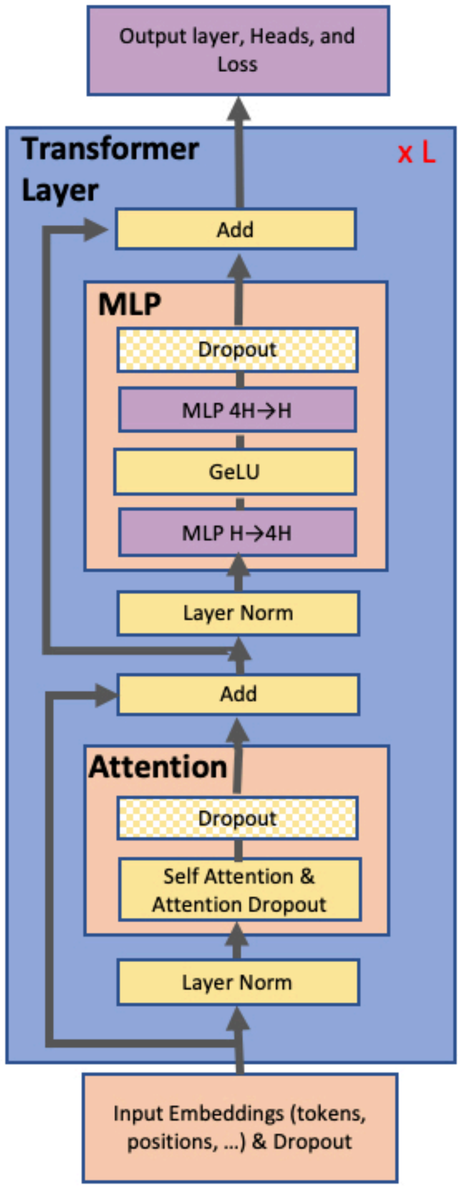
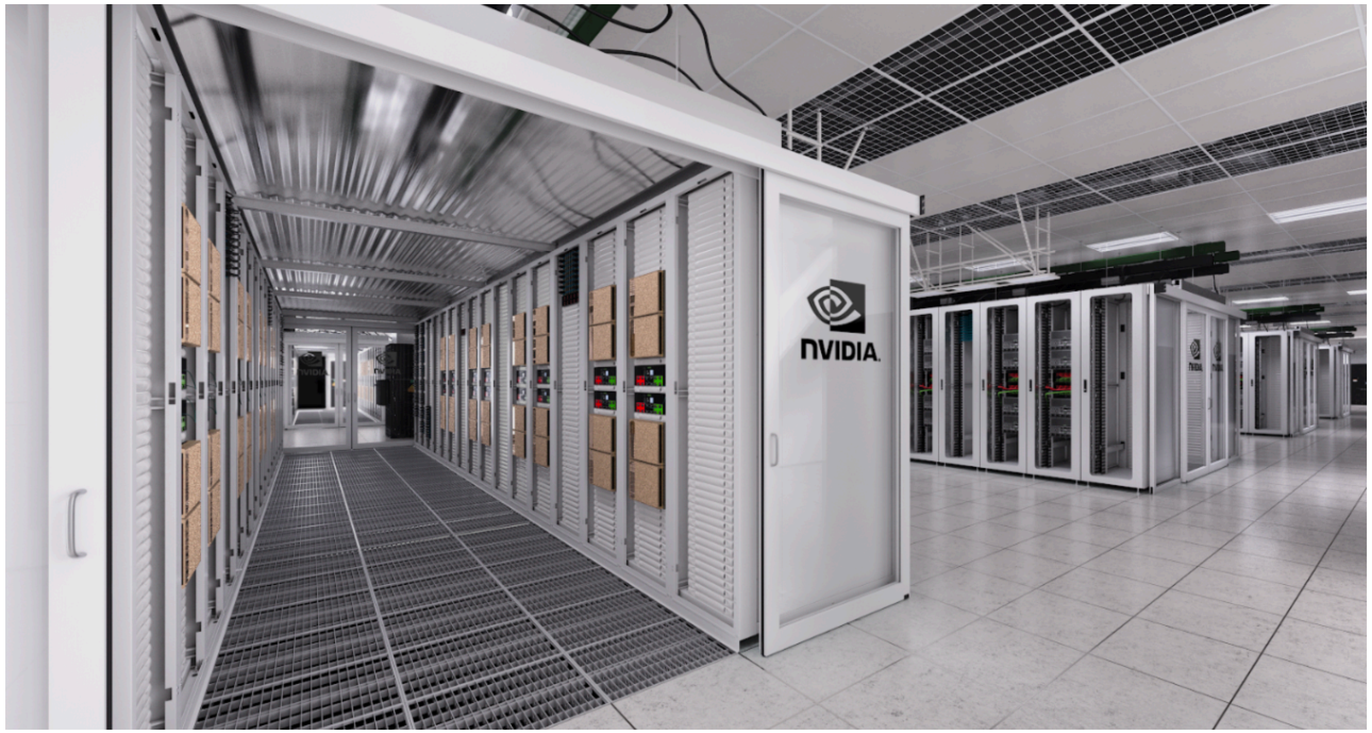


Figure 2 arXiv:1909.08053 (Megatron)

Megatron Training Hardware

- Nvidia Selene:
  - Top 10 supercomputer
- DGX Node:
  - 8 A100s
  - 600 GB/s GPU-to-GPU
  - 8x200Gb/s networking
  - ~$200k



Nvidia Selene, src: Nvidia

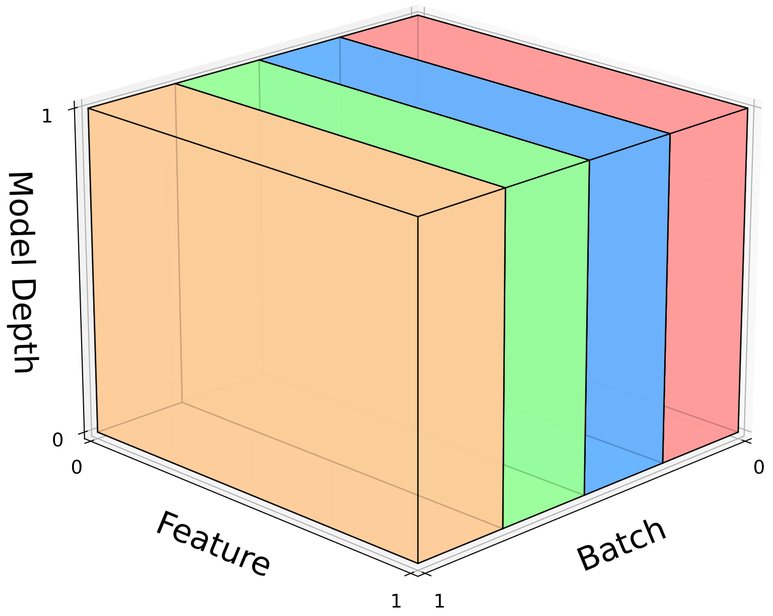
- Heterogeneous connectivity, faster within nodes, affects parallelism trade-offs

Types of Parallelism

- For fwd pass through a neural network, the total work is computing activations across three dimensions, the mini-batch, features (model width) and layers (model depth)

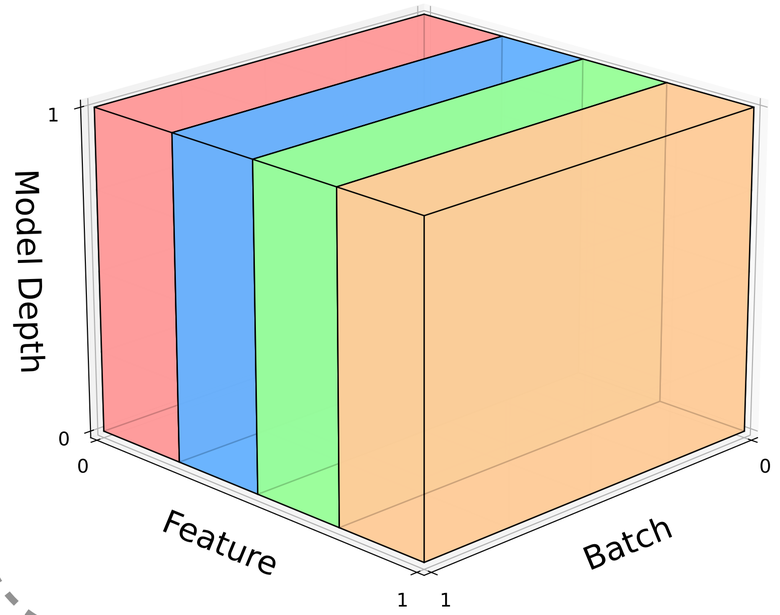
- Each axis can be parallelized across GPUs (colors)

Data Parallel

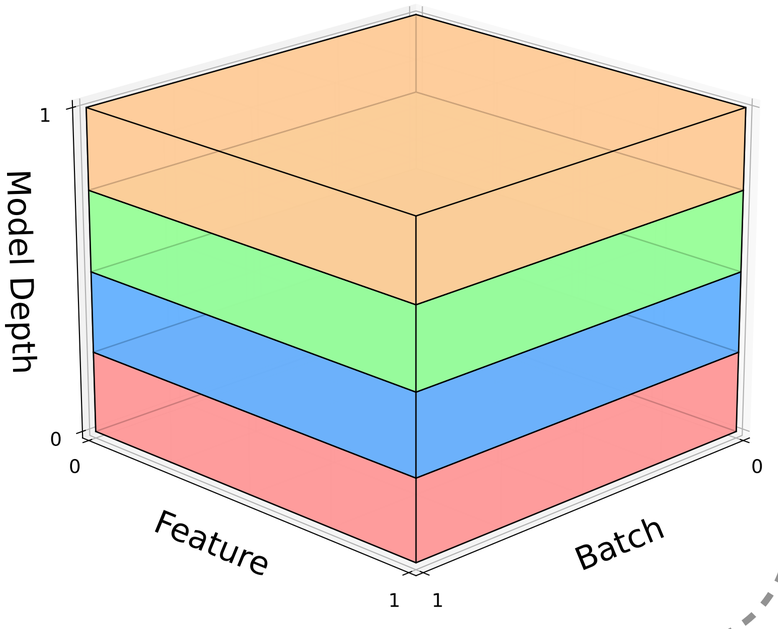


Model Parallelism

Tensor Parallel



Pipeline Parallel

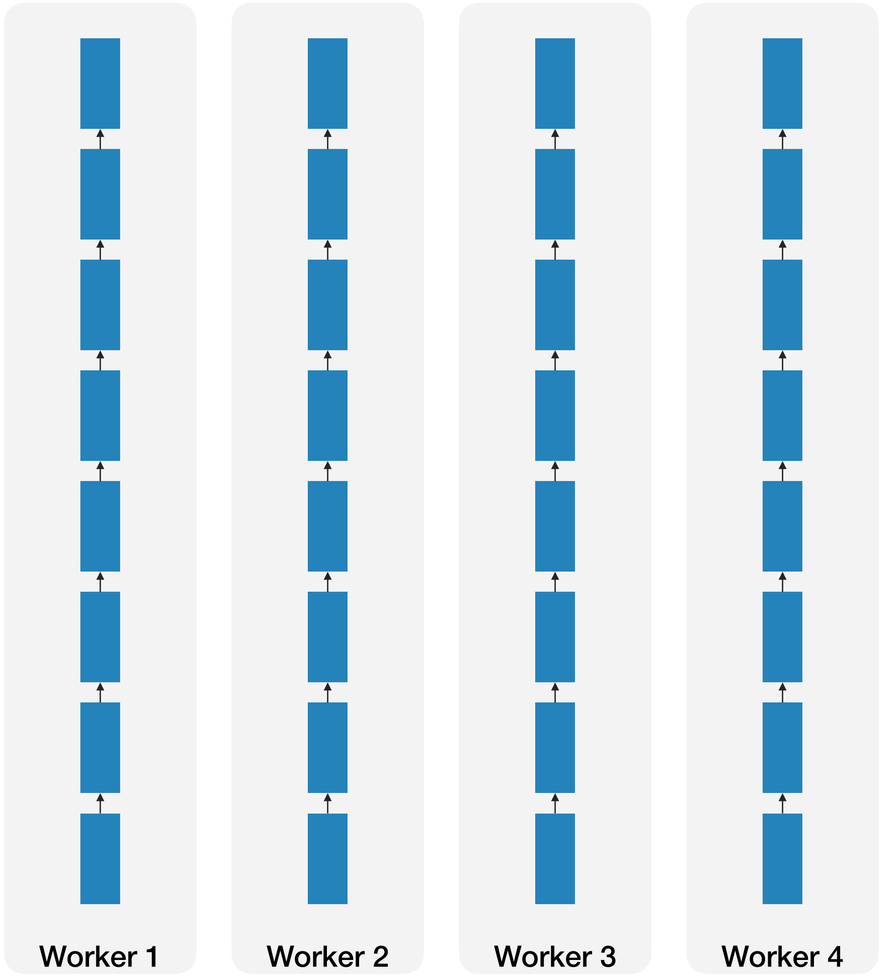


Parallelization Technique 1: Distributed Data Parallel

- Method:
  - Split mini-batch between workers
  - Gradients all-reduced over workers
  - Each worker has entire model state

- Strengths:
  - Near perfect scaling at small scales
  - Communication + computation largely overlapped
  - Gradient accumulation for reduced overhead

- Weaknesses:
  - No memory savings for model state
  - Limited by global batch size (e.g. critical batch size)



Each worker processes different samples
Communicate gradients for all weights

Sidenote: Gradient Accumulation

```
acc_steps = 8

for batch_idx, (inputs, targets) in enumerate(data_loader):
    inputs, targets = inputs.to(device), targets.to(device)

    preds = model(inputs)
    loss = criterion(preds, targets)
    loss = loss / acc_steps
    loss.backward()

    # weights update
    if ((batch_idx + 1) % acc_steps == 0) or (batch_idx + 1 == len(data_loader)):
        optimizer.step()
        optimizer.zero_grad()
```

- Sometimes you want to replicate DDP training with less resources
- We can emulate larger total batch sizes by accumulating the gradients across multiple smaller batches
- Divide loss by accumulation steps (for mean losses) + only step optimizer periodically
- Allows you to reuse the hyperparameters, mathematically equivalent assuming randomness and data is handled correctly

Sidenote: Zero Redundancy Optimizer

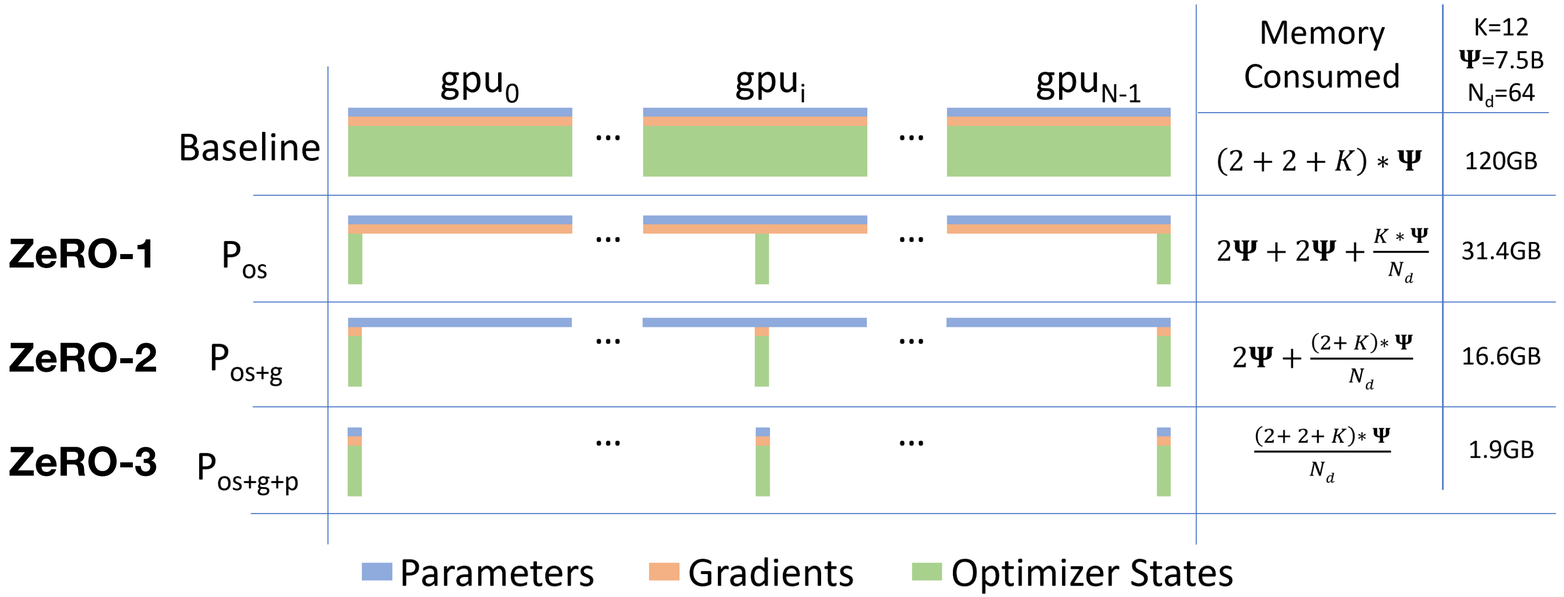


Figure 1 arXiv:1910.02054 (ZeRO)

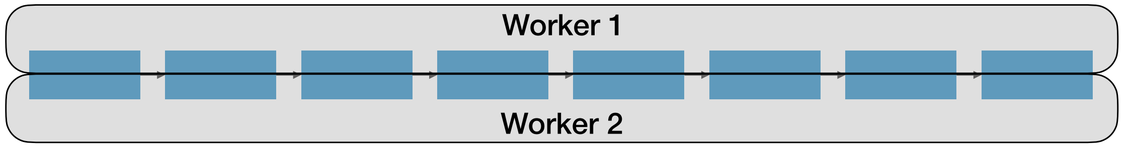
- Sharded Data Parallelism: Divide model state between workers to save memory
  - Necessary weights are sent just in time and deleted after use
- ZeRO-1: Optimizer state partitioned
- ZeRO-2: Optimizer state + gradients
- ZeRO-3: Optimizer state + gradient + weights (aka Fully Shared Data Parallel)
- ZeRO-3 requires sending weights twice (fwd, bwd), extra communication overhead
- **Typically easier to use than model parallelism methods (torch.distributed.fsdp)**

Parallelization Technique 2: Tensor Parallel

- Method:
  - Split computation of a single tensor over multiple workers
  - Model parallelism

- Strengths:
  - Reduced memory usage for parameters
  - Potential memory savings for activations

- Weaknesses:
  - Frequent communication / synchronization
  - Communication per compute scales poorly



We can split matrix multiplication:

$$XA$$

By rows of A (partial sums):

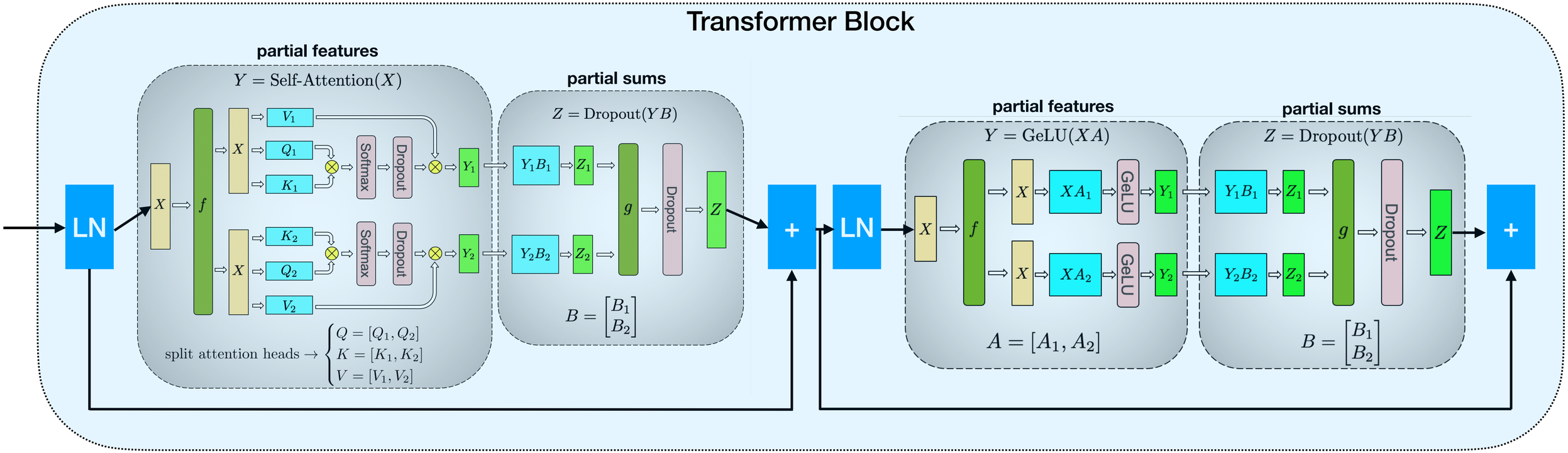
$$XA = [X_1, X_2]\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = X_1A_1 + X_2A_2$$

By columns of A (partial features):

$$XA = X[A_1, A_2] = [XA_1, XA_2]$$

Both save model memory
Different communication properties, can mix the two

Parallelization Technique 2: Tensor Parallel

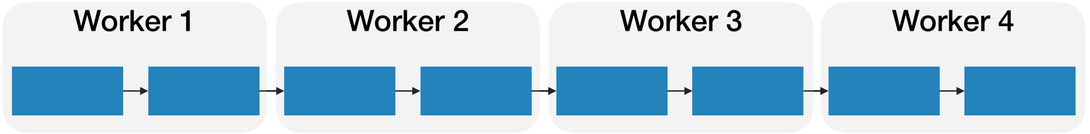


Modified Figure 3 arXiv:1909.08053 (Megatron)

- Megatron setup:
  - Split attention heads + channels
  - Combine with 2 all-reduce / block
  - All workers perform dropout 2 and 3 (seeds synchronized) + normalization

Parallelization Technique 3: Pipeline Parallel

- Method:
  - Split successive portions of model over multiple workers
  - Split batch into microbatches and feed in sequentially
  - Different type of model parallelism



- Strengths:
  - Reduced memory usage for parameters
  - Low communication (activations at partition boundaries)

- Weaknesses:
  - Pipeline bubbles
  - Need to balance throughput of workers
  - Uneven activation storage requirements
  - Limited by model depth

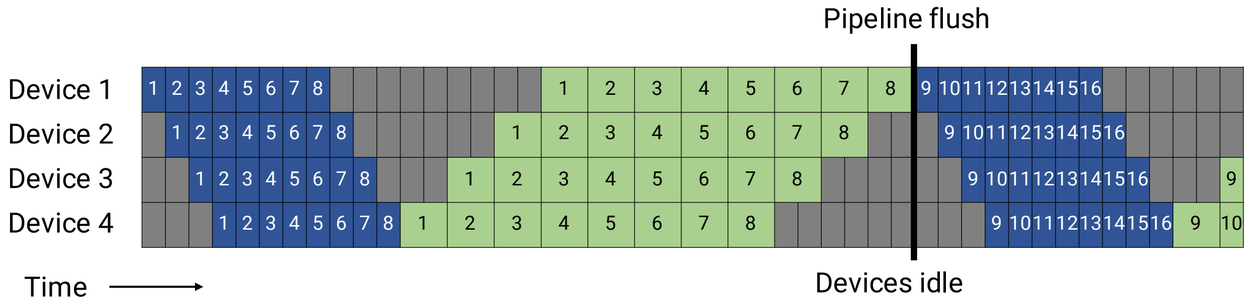
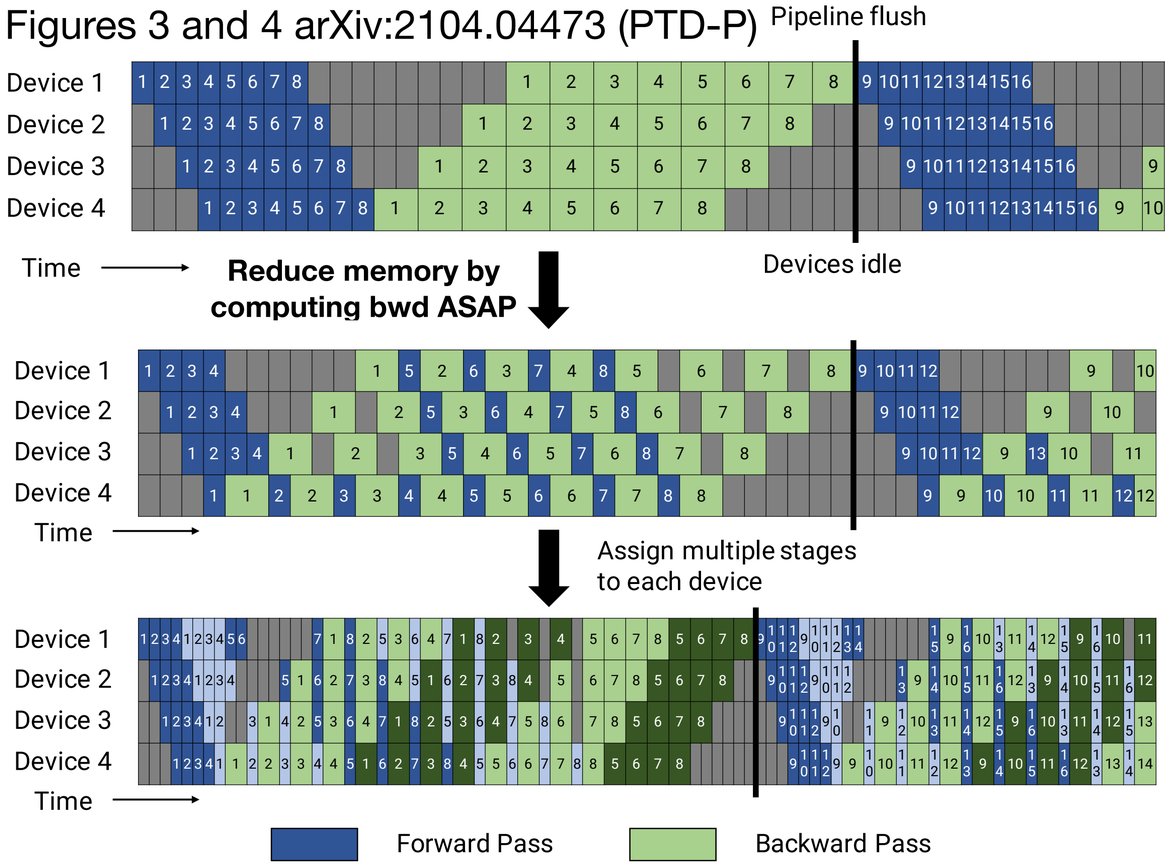
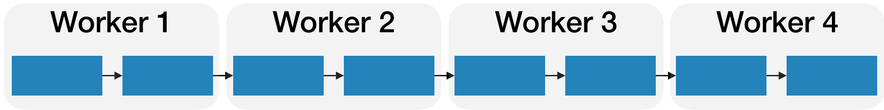


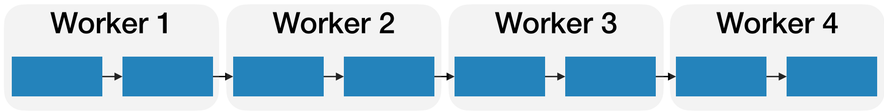
Figure 3 arXiv:2104.04473 (PTD-P)

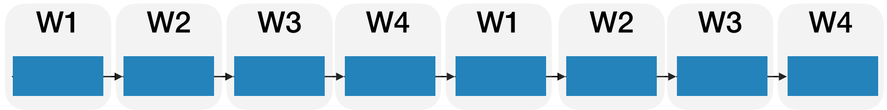
Parallelization Technique 3: Pipeline Parallel

Figures 3 and 4 arXiv:2104.04473 (PTD-P)









More boundaries → Increased communication

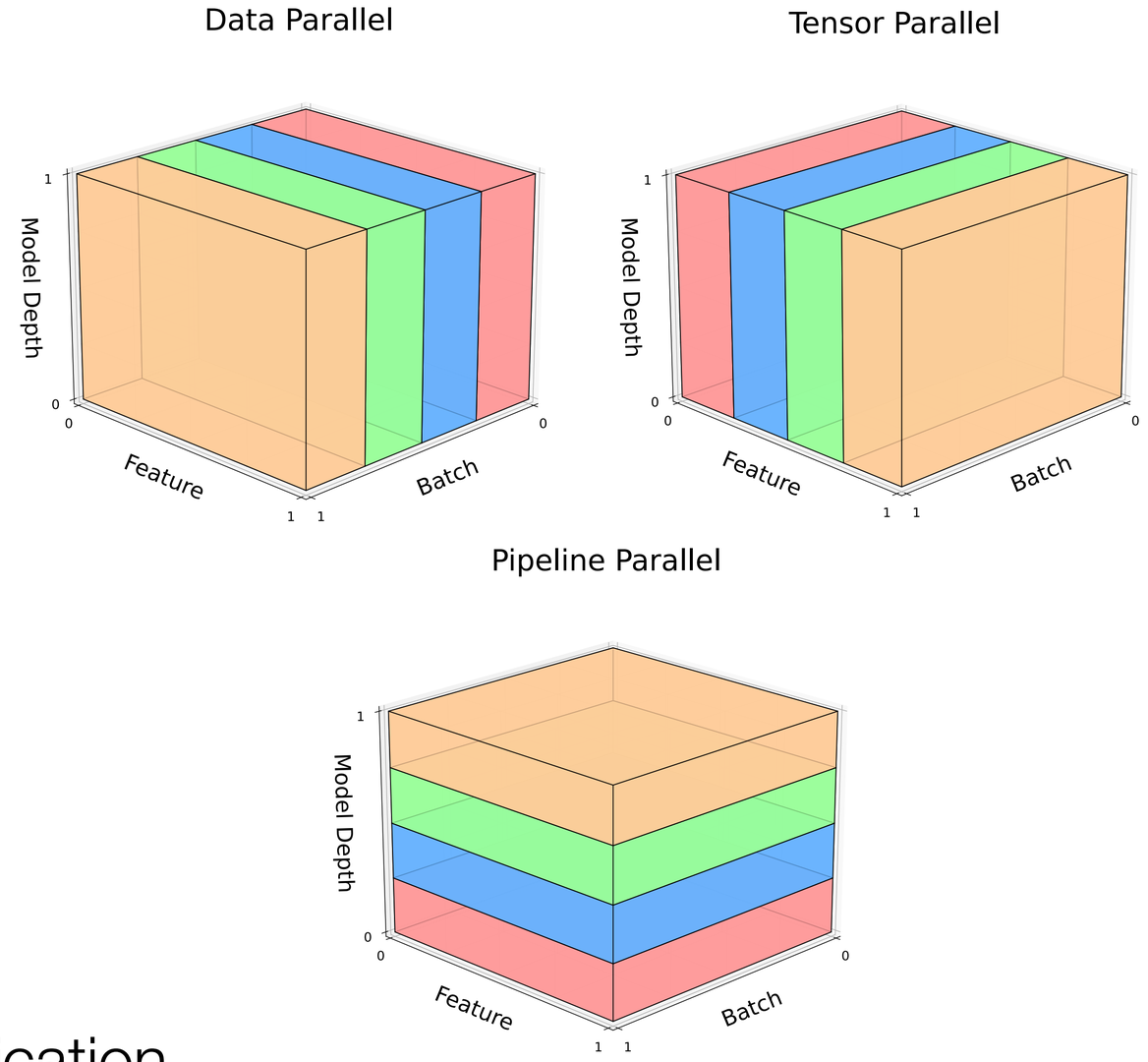
- Number of pipeline stages p, microbatches per pipeline m, stages per worker v
- Bubble time: $\frac{p-1}{vm}$

Combined Parallelism: PTD-P / 3D

- Why combine different types of parallelism?
  - Work around the limitations of each
  - Sometimes interact positively

- Pipeline and tensor parallelism:
  - Save memory
  - Reduce communication for data parallel
  - Parallelize beyond global batch size

- Data Parallelism:
  - Avoid pipeline bubbles / frequent communication
  - Parallelize beyond depth



Combined Parallelism: PTD-P / 3D

- Optimal mix?

- Pipeline vs Tensor Parallelism:
  - Both save memory
  - Pipeline bubbles vs increased bandwidth
  - Optimal: Use tensor parallelism within node

- Model parallelism vs Data Parallelism:
  - Data parallelism faster but doesn't save memory
  - Data parallelism limited by overall global batch size
  - Optimal (Megatron): Use sufficient model parallelism for memory and data parallel after that if possible

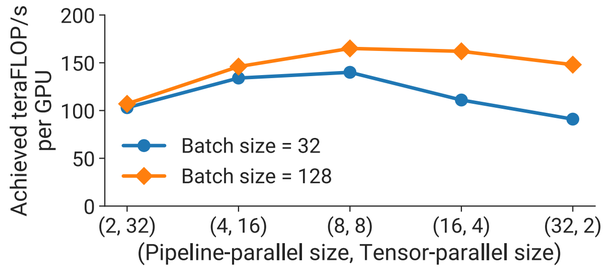


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

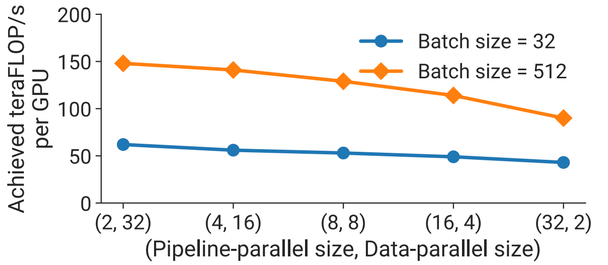


Figure 14: Throughput per GPU of various parallel configurations that combine data and pipeline model parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

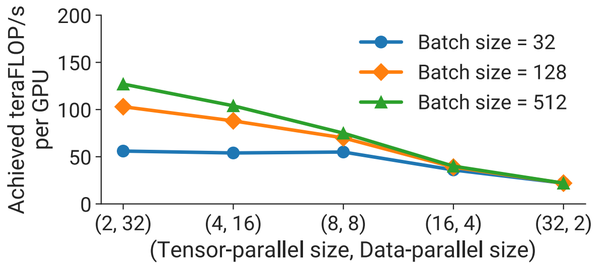
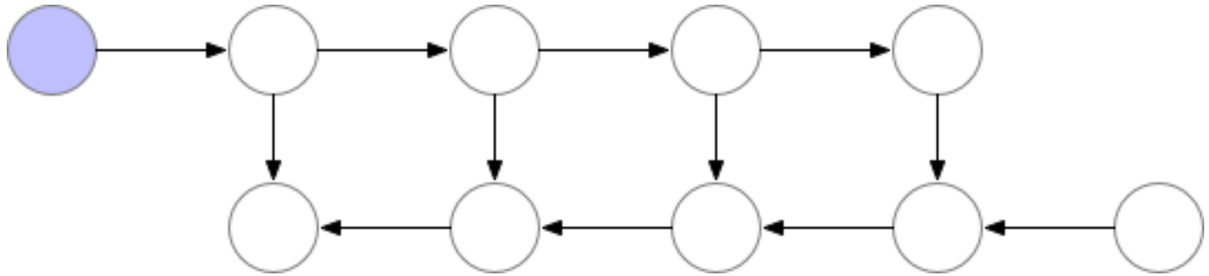


Figure 15: Throughput per GPU of various parallel configurations that combine data and tensor model parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

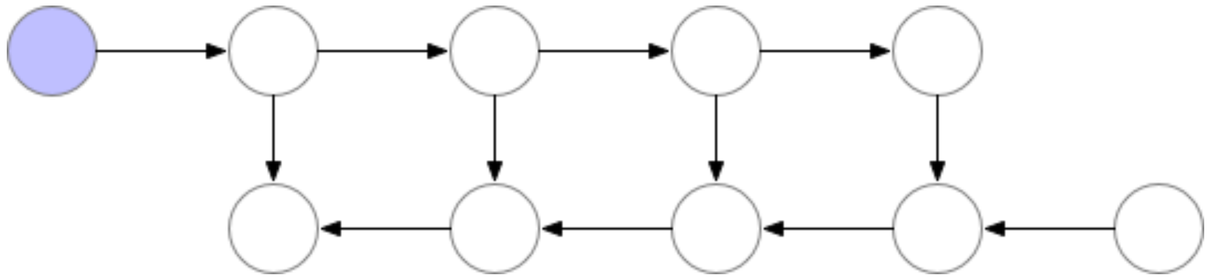
Additional Trick: Gradient Checkpointing

- Activation Recomputation (gradient checkpointing):
  - Don’t keep all activations in memory for bwd
  - Recompute them for a single block at a time
  - One micro-batch at a time for pipeline

- Vanilla backprop (blue activations in memory):



- Checkpointing example:



Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM

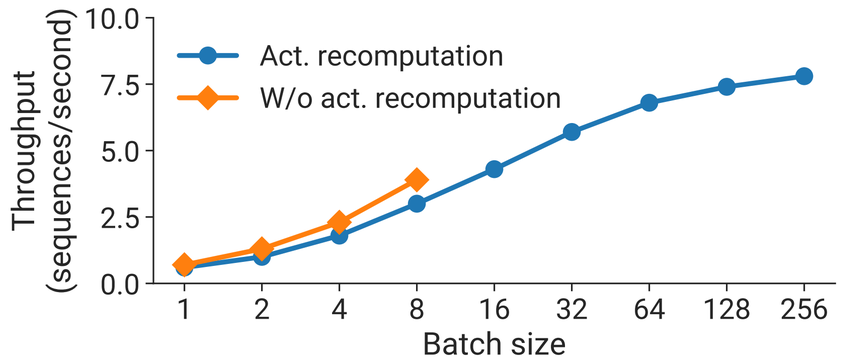


Figure 17: Throughput (in sequences per second) with and without activation recomputation for a GPT model with 145 billion parameters using 128 A100 GPUs ($(t, p) = (8, 16)$).

https://github.com/cybertronai/gradient-checkpointing

More Tricks

- Scatter-Gather Optimization:
  - Divide pipeline parallel communication over all 8 links of DGX boxes

- Fused Operators (+11% throughput):
  - Element-wise ops are inefficient
  - Combine sequential ones into a single kernel
  - Bias + GeLU
  - Bias + Dropout + Add

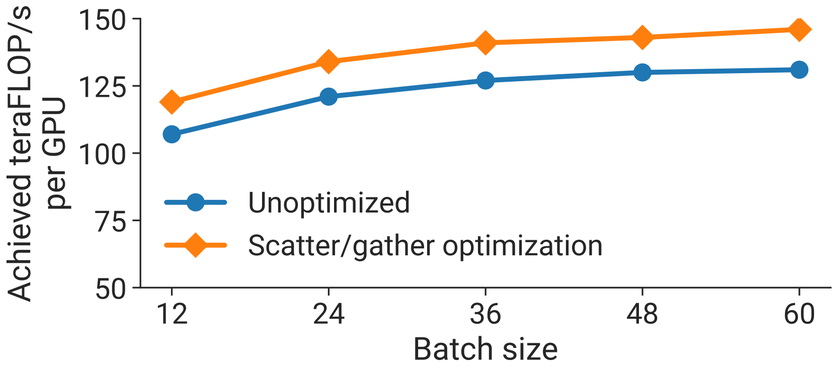
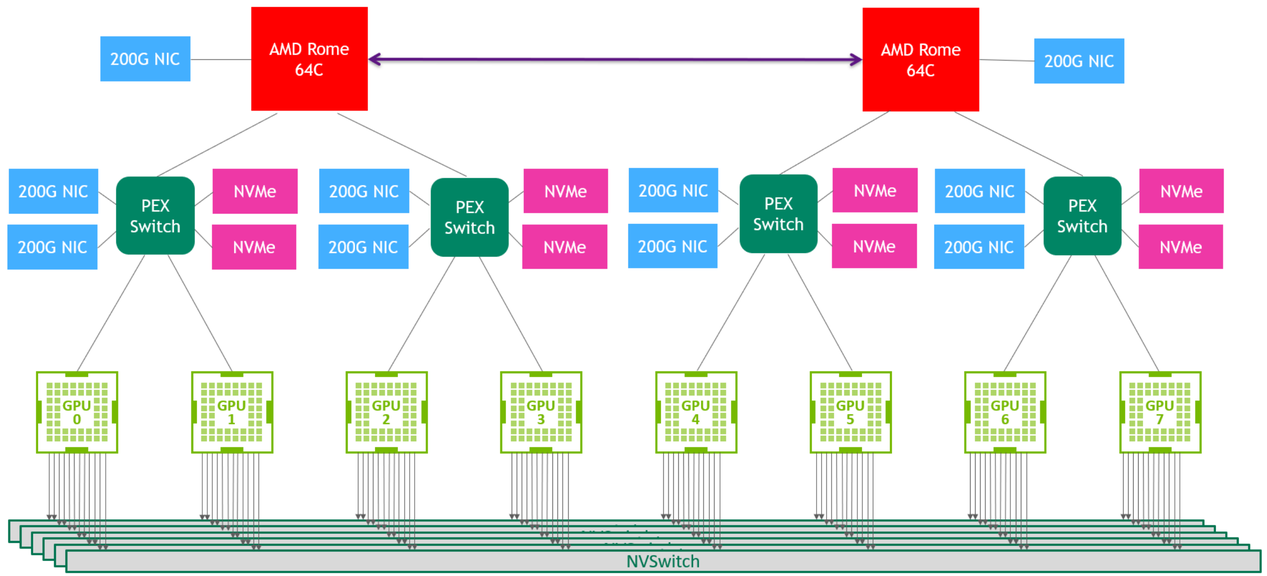


Figure 18: Throughput per GPU with and without the scatter/gather optimization for a GPT model with 175 billion parameters using 96 A100 GPUs and the interleaved schedule.



Megatron Results: Utilization vs Model Size

Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7	24	2304	24	1	1	32	512	137	44%	4.4
3.6	32	3072	30	2	1	64	512	138	44%	8.8
7.5	32	4096	36	4	1	128	512	142	46%	18.2
18.4	48	6144	40	8	1	256	1024	135	43%	34.6
39.1	64	8192	48	8	2	512	1536	138	44%	70.8
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0

Table 1: Weak-scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

- Successfully scales to large model sizes
- Objectively high device utilization ≈52% (≈39% without counting recompute)

PyTorch Distributed Data Parallel

```python
import os
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data import Dataset, DataLoader
from torch.utils.data.distributed import DistributedSampler

# Initialize the distributed group
def init_distributed():
    dist.init_process_group(backend='nccl')
    torch.cuda.set_device(int(os.environ['LOCAL_RANK']))
```

- Multiple separate processes
- Initialize distributed group, needs additional info to set up
  - Env: MASTER_PORT, MASTER_ADDR, WORLD_SIZE, RANK, LOCAL_RANK
  - torchrun --nproc_per_node=8 --nnodes=2 --node_rank=0 --master_addr=123.456.123.456 --master_port=1234 train.py
- Important: Set seeds and split data!
- Wrap model in DDP

```python
def main():
    # Initialize the distributed group
    init_distributed()
    # Set different random seeds for each rank
    rank = dist.get_rank()
    torch.manual_seed(42 + rank)
    # Create and split the dataset
    dataset = get_dataset(1000)
    sampler = DistributedSampler(dataset)
    dataloader = DataLoader(dataset, batch_size=32, sampler=sampler)
    # Create the model and wrap it in DDP
    model = get_model().cuda()
    model = DDP(model, device_ids=[int(os.environ['LOCAL_RANK'])])
    # Define loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(model.parameters())
    # In practice: scheduler, weight decay only on matrices
    # Training loop
    for epoch in range(10):
        sampler.set_epoch(epoch)
        for batch, (data, labels) in enumerate(dataloader):
            data, labels = data.cuda(), labels.cuda()
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, labels)
            loss.backward()
            optimizer.step()
            if batch % 10 == 0 and rank == 0:
                print(f"Epoch {epoch}, Batch {batch}, Loss: {loss.item()}")
```

Asynchronous and Decentralized Methods

- All previous methods discussed are equivalent to using a large batch size on a sufficiently powerful single GPU
- Keeping everything consistent requires synchronization with overheads
- Asynchronous methods also exist:
  - Examples: Federated learning, asynchronous SGD, pipelined backpropagation
  - Each worker does not wait for the others, receives gradients later
  - More computationally efficient
  - Cause optimization issues, delayed gradients and model inconsistency
- Still a research area, not used much in practice especially for large models