Lecture 11:
Large-Scale Distributed Training

Administrative

Reminders:

- Friday 5/9: Midterm Review Session
- Tuesday 5/13: In-Class Midterm

Today:
Large-Scale Distributed Training

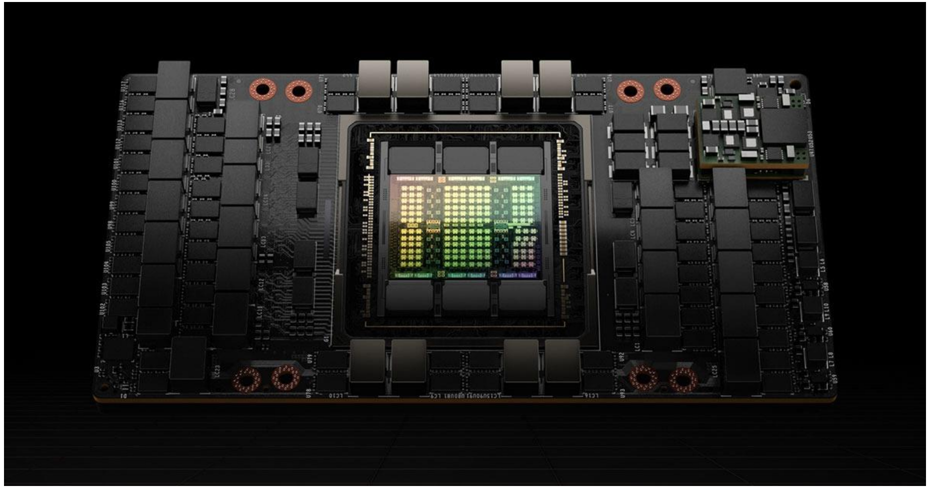
Running Example: Llama3-405B

- GPT4 kicked off a trend of not sharing any model details:
  “Given both the competitive landscape and the safety implications of large-scale models like GPT-4, this report contains no further details about the architecture (including model size), hardware, training compute, dataset construction, training method, or similar.”
- Llama3: Open-source LLM released by Meta in April 2024; paper shares many model and training details
- Llama4: Released initial models April 2025, but no paper yet

Llama Team, “The Llama 3 Herd of Models”, https://arxiv.org/abs/2407.21783
OpenAI, ”GPT4 Technical Report”, arXiv 2023

GPUs and How to Train On Them

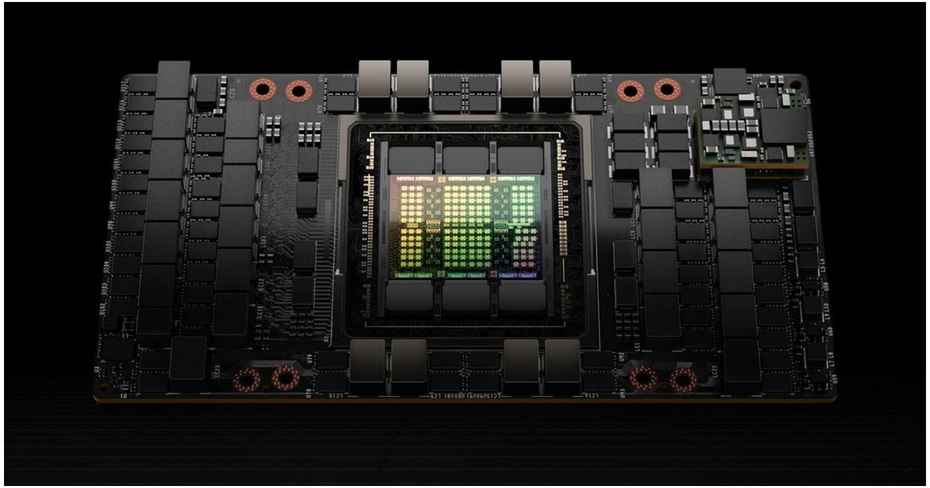
A bit about GPU hardware



How to train on lots of GPUs

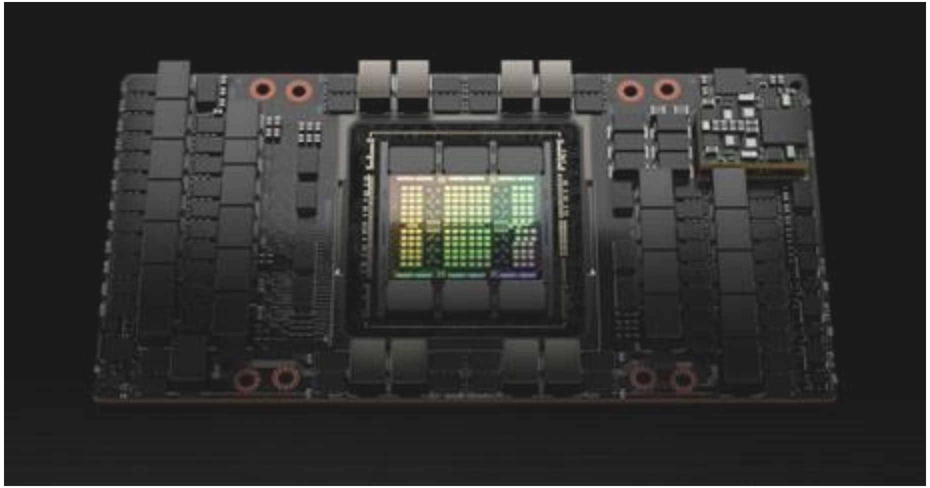
GPUs and How to Train On Them

A bit about GPU hardware



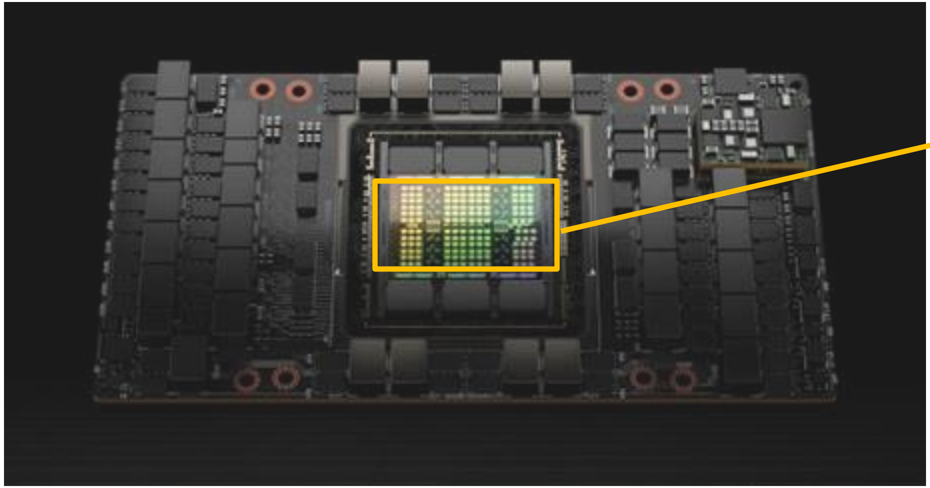
How to train on lots of GPUs

Inside a GPU: NVIDIA H100



GPU = Graphics Processing Unit
Originally for graphics
Now a general parallel processor

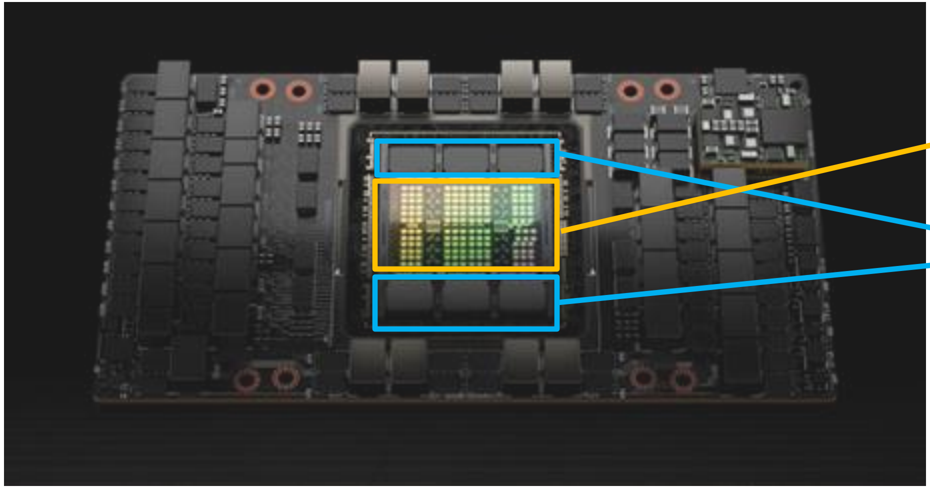
Inside a GPU: NVIDIA H100



GPU = Graphics Processing Unit
Originally for graphics
Now a general parallel processor

Compute Cores

Inside a GPU: NVIDIA H100

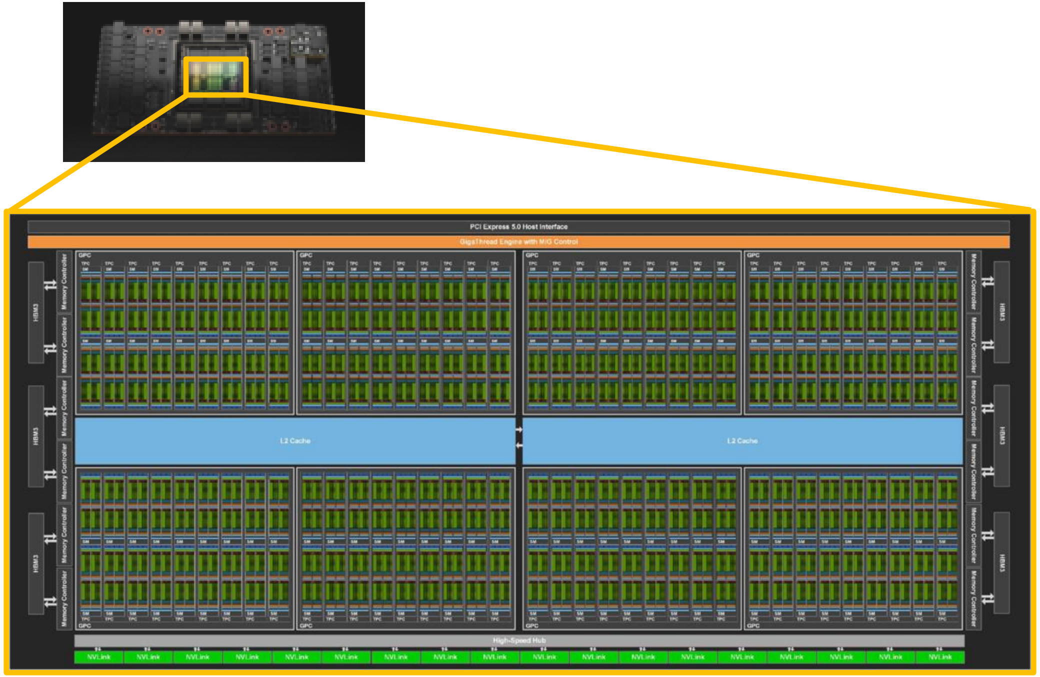


GPU = Graphics Processing Unit
Originally for graphics
Now a general parallel processor

Compute Cores

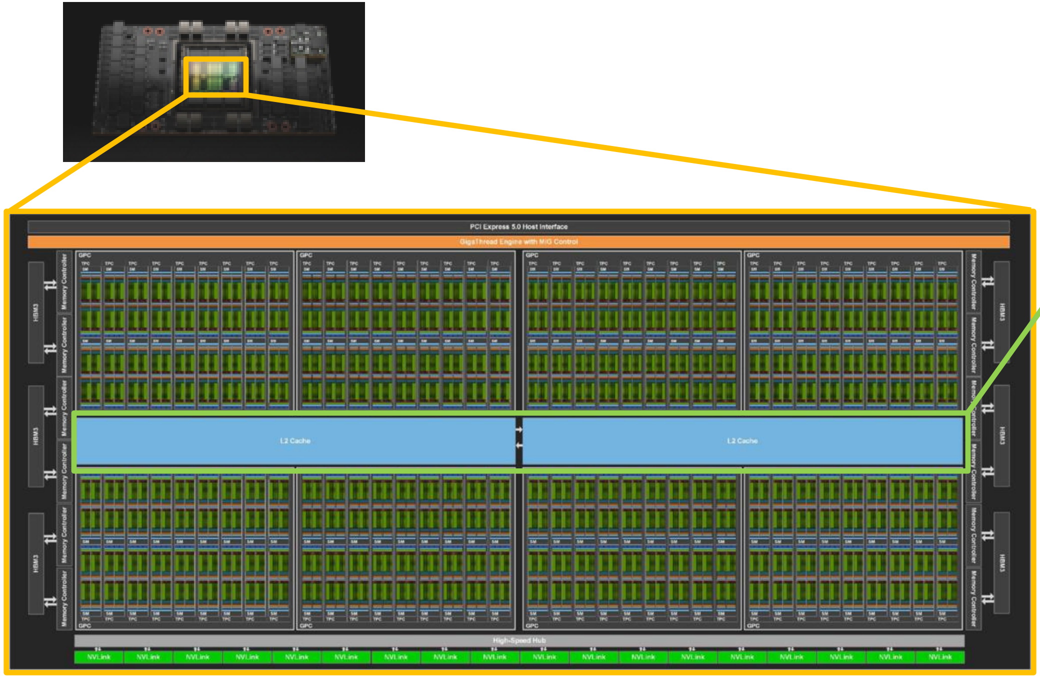
80 GB of HBM Memory
3352 GB/sec bandwidth to cores

Inside a GPU: NVIDIA H100



H100 Compute Cores

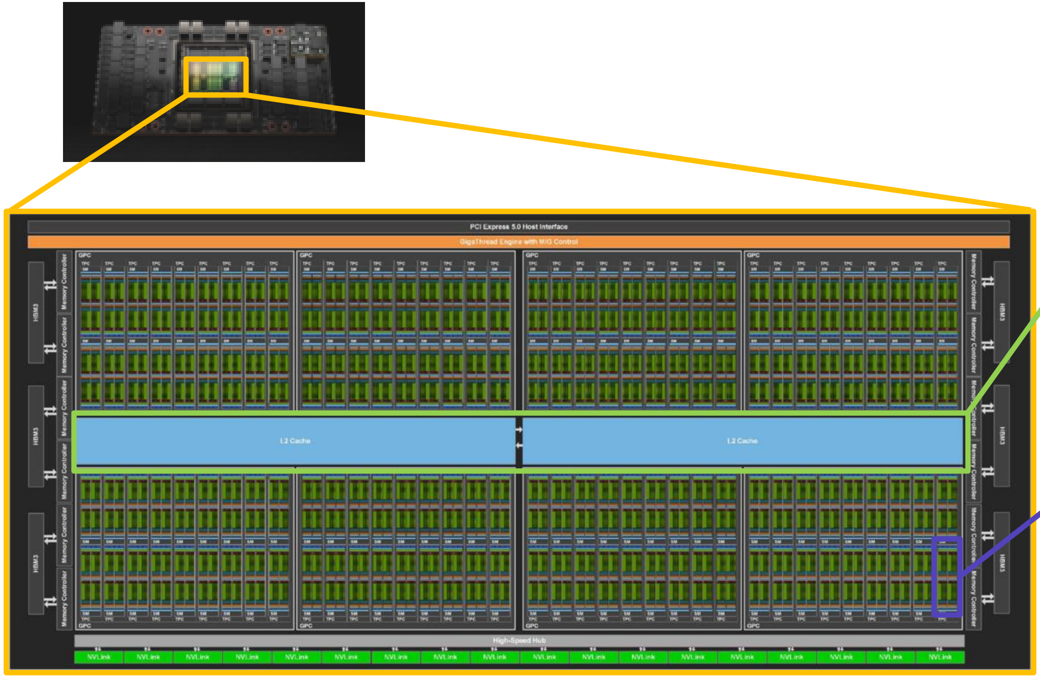
Inside a GPU: NVIDIA H100



H100 Compute Cores

50MB of L2 Cache

Inside a GPU: NVIDIA H100



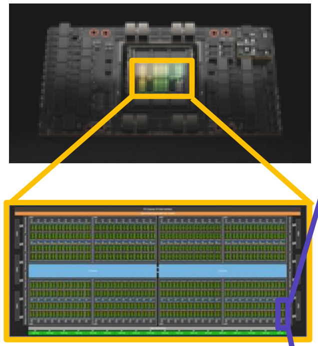
H100 Compute Cores

50MB of L2 Cache

132 Streaming Multiprocessors (SMs)
These are independent parallel cores
(Actually 144 here; only 132 are enabled due to yield)

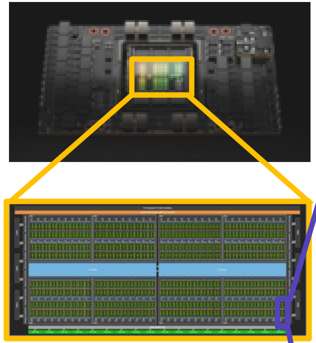
Inside a GPU: NVIDIA H100

H100 Streaming Multiprocessor



Sort of like a CPU core with vector instructions

Inside a GPU: NVIDIA H100

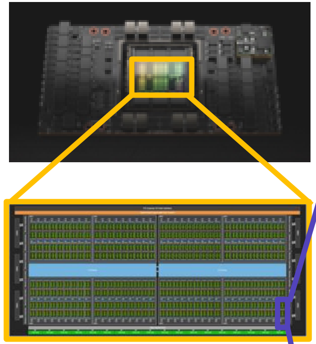


Sort of like a CPU core with vector instructions

H100 Streaming Multiprocessor

256 KB L1 cache, 256 KB registers

Inside a GPU: NVIDIA H100



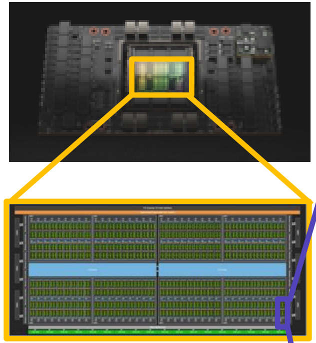
Sort of like a CPU core with vector instructions

H100 Streaming Multiprocessor

256 KB L1 cache, 256 KB registers

128 FP32 Cores
Computes a*x + b per clock cycle
2 FLOPs = Floating Point Operations
256 FLOP/cycle per SM

Inside a GPU: NVIDIA H100



Sort of like a CPU core with vector instructions

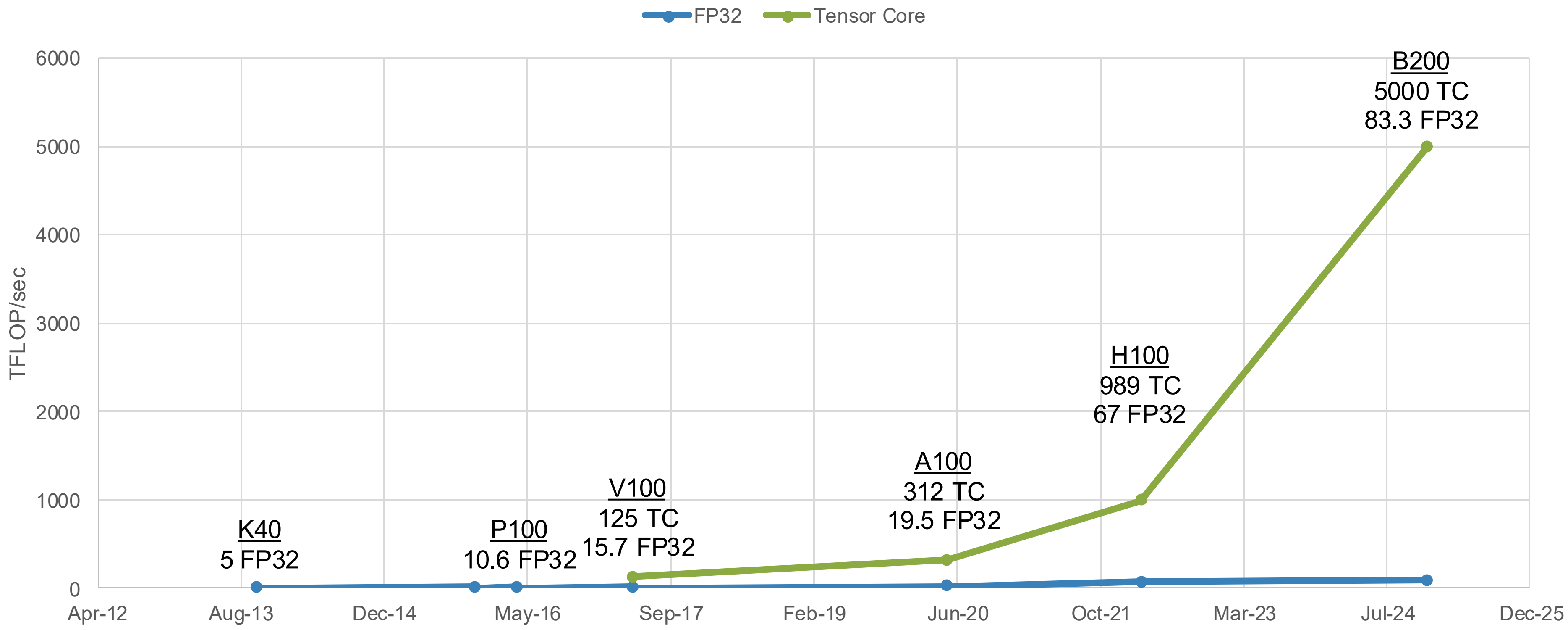
H100 Streaming Multiprocessor

256 KB L1 cache, 256 KB registers

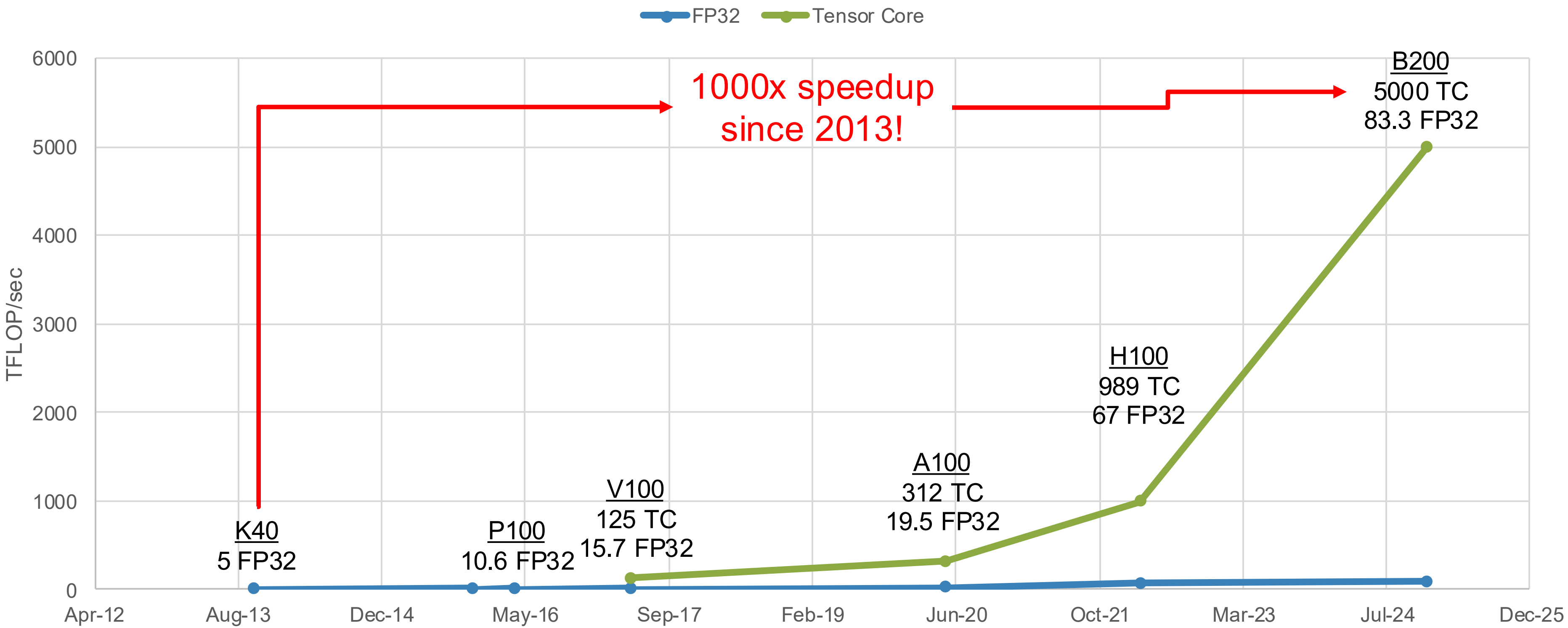
128 FP32 Cores
Computes a*x + b per clock cycle
2 FLOPs = Floating Point Operations
256 FLOP/cycle per SM

4 Tensor Cores
Computes AX + B per clock cycle
Matrix operation: [16x4][4x8] + [16x8]
16*4*8*2 = 1024 FLOPs
4096 FLOP/cycle per SM
Mixed precision: 16-bit / 32-bit

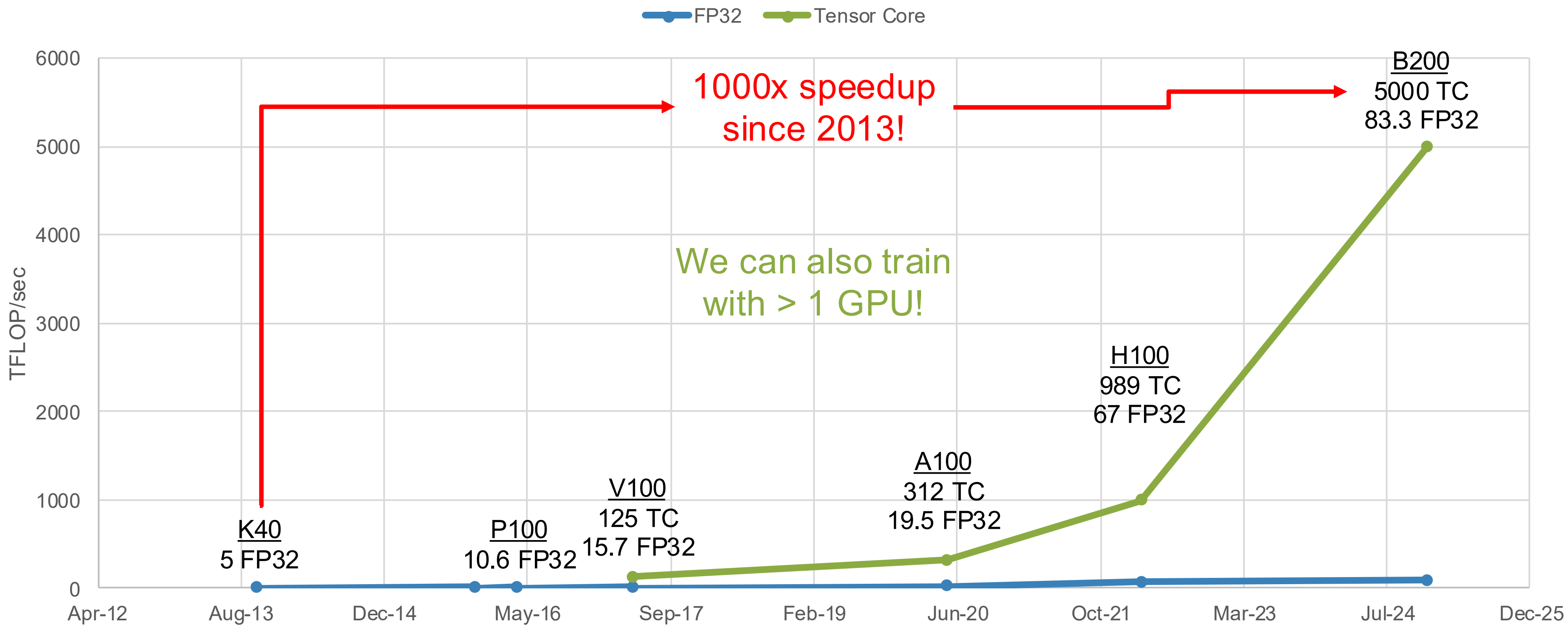
GPUs Have Gotten Much Faster!



GPUs Have Gotten Much Faster!

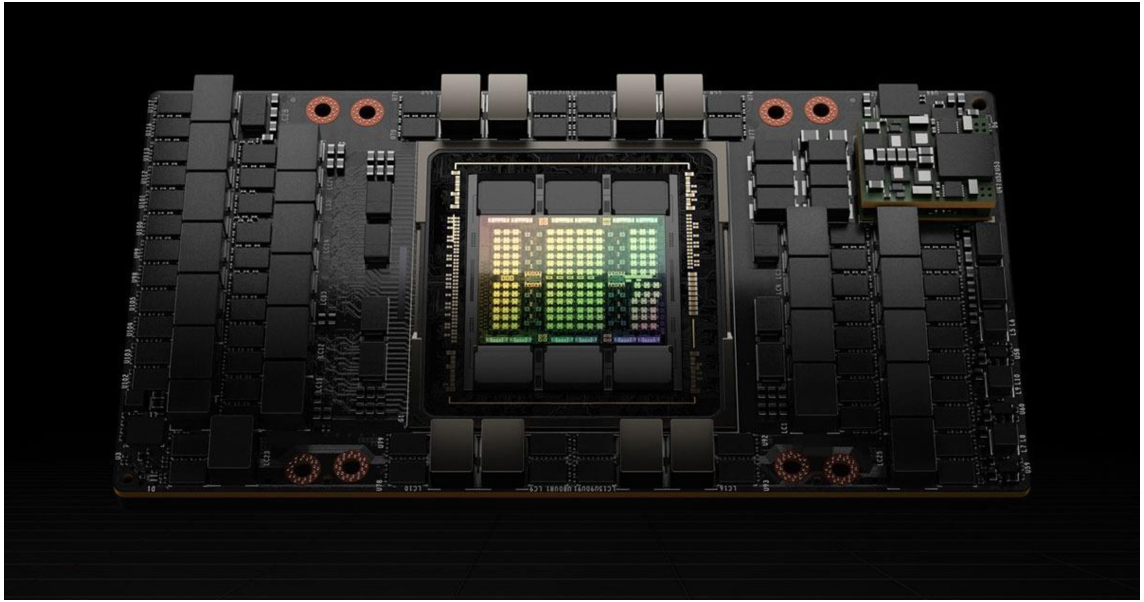


GPUs Have Gotten Much Faster!



NVIDIA H100 GPU

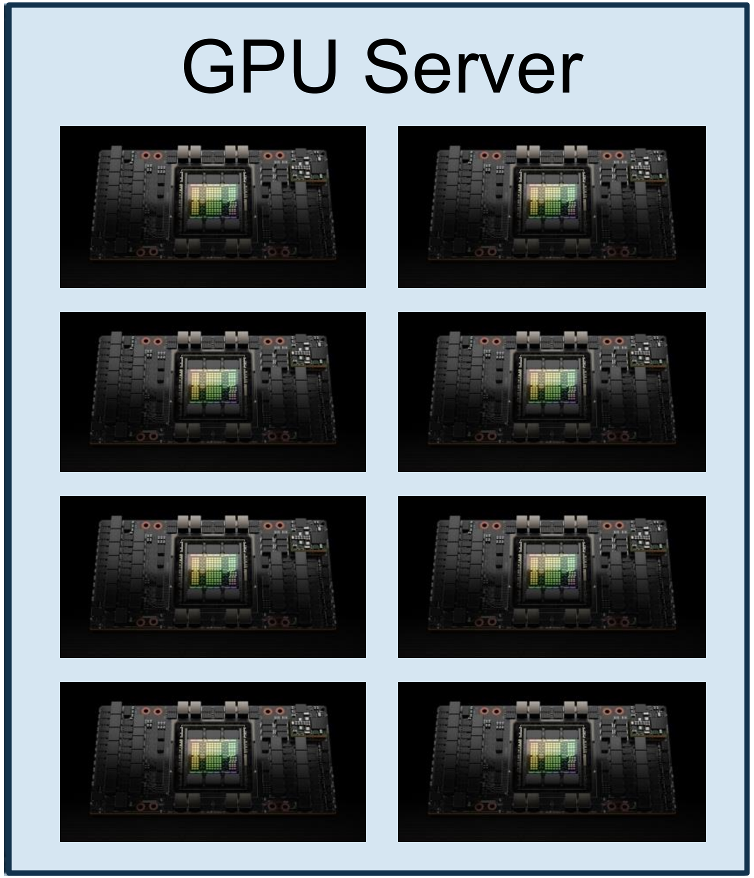
H100 GPU
3352 GB/sec inside the GPU



NVIDIA H100 GPU

H100 GPU
3352 GB/sec inside the GPU

Server = 8x GPU
900 GB/sec between GPUs

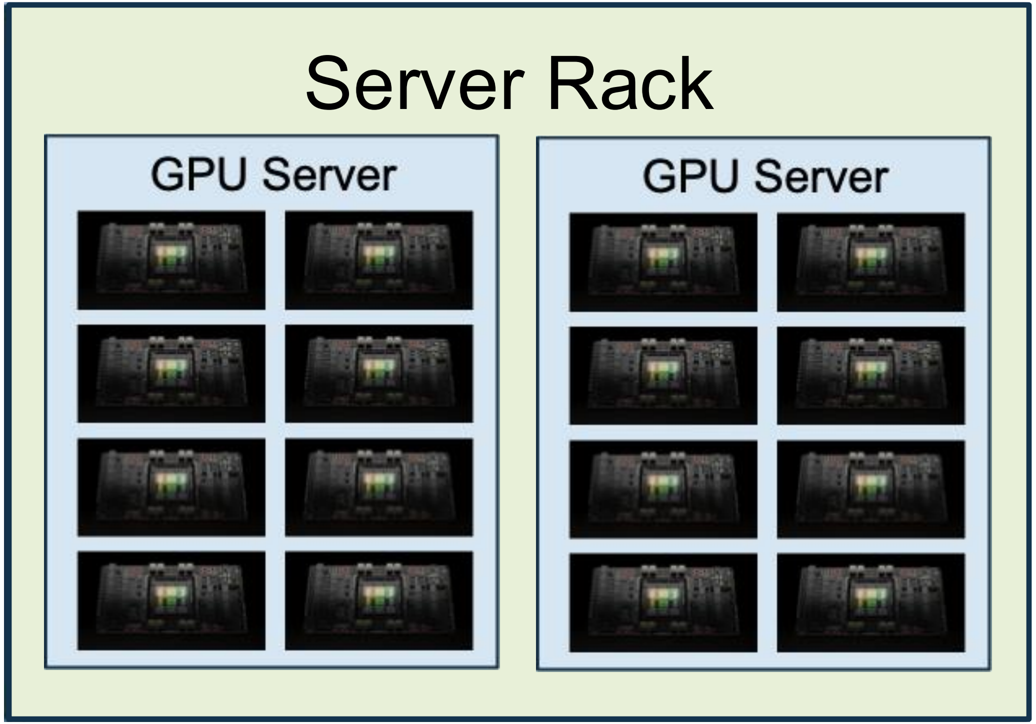


Case Study: Meta's Llama3 Cluster

H100 GPU
3352 GB/sec inside the GPU

Server = 8x GPU
900 GB/sec between GPUs

Rack = 2 Servers = 16x GPU



Llama Team, "The Llama 3 Herd of Models", https://arxiv.org/abs/2407.21783

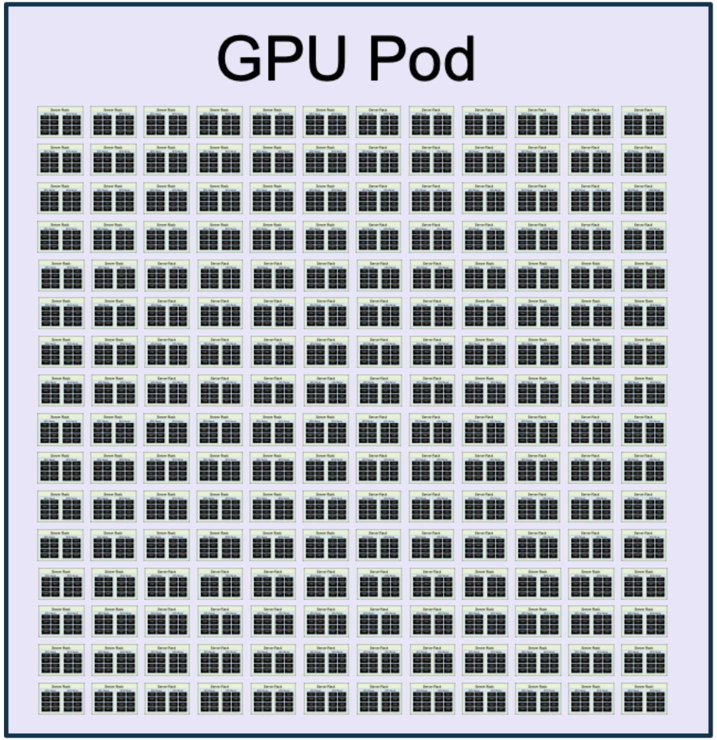
Case Study: Meta’s Llama3 Cluster

H100 GPU
3352 GB/sec inside the GPU

Server = 8x GPU
900 GB/sec between GPUs

Rack = 2 Servers = 16x GPU

Pod = 192 Racks = 3072 GPUs
50 GB/sec between GPUs



Llama Team, “The Llama 3 Herd of Models”, https://arxiv.org/abs/2407.21783

Case Study: Meta's Llama3 Cluster

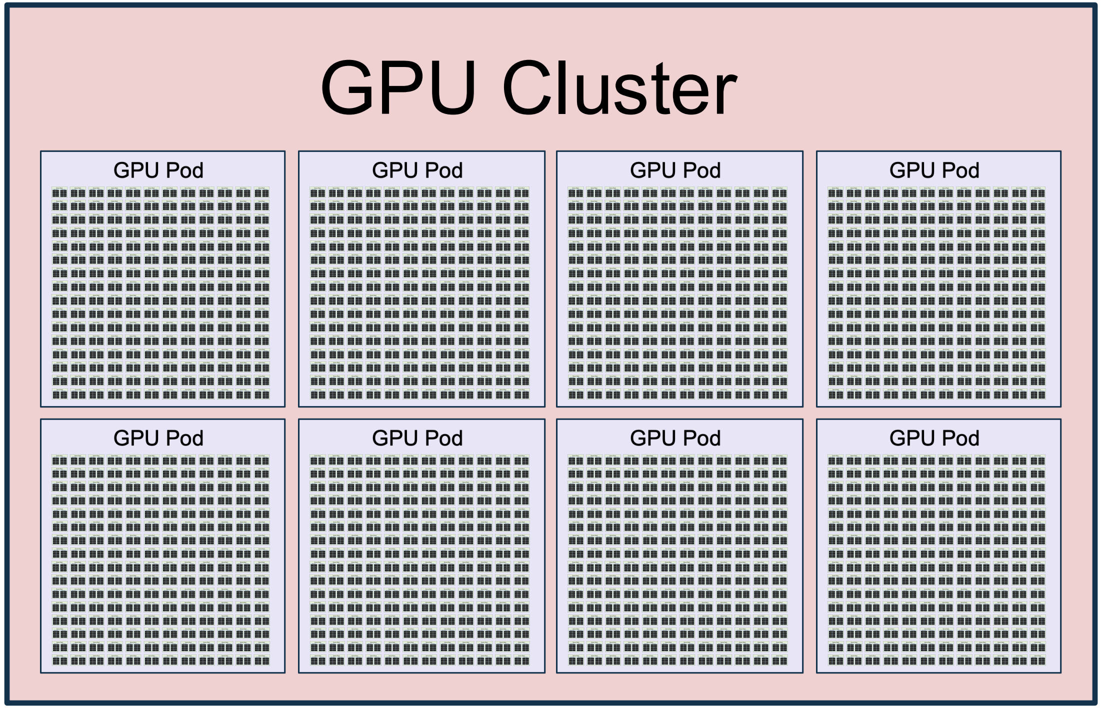
H100 GPU
3352 GB/sec inside the GPU

Server = 8x GPU
900 GB/sec between GPUs

Rack = 2 Servers = 16x GPU

Pod = 192 Racks = 3072 GPUs
50 GB/sec between GPUs

Cluster = 8 Pods = 24,576 GPUs
< 50GB/sec between GPUs

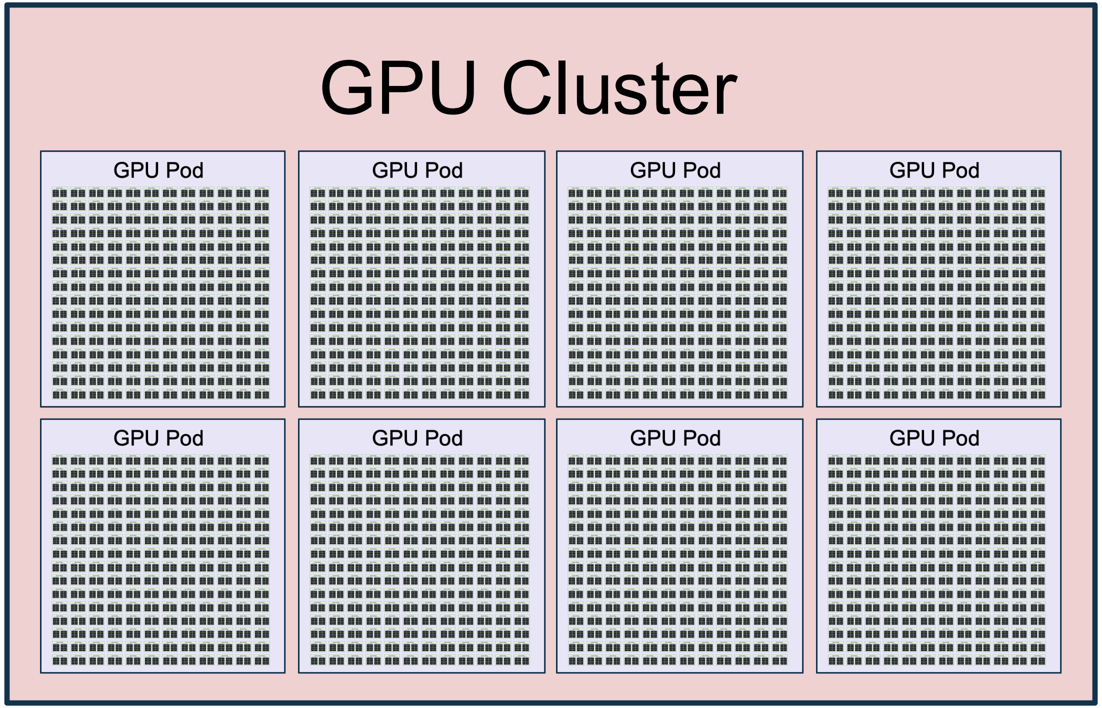


Llama Team, "The Llama 3 Herd of Models", https://arxiv.org/abs/2407.21783

GPU Cluster = One Big Computer

Total Cluster Stats
24,576 GPUs
1.875 PB of GPU memory
415M FP32 cores
13M Tensor Cores
24.3 EFLOP/sec = 24.3 x 10^{18}

<u>Goal</u>: Train one giant neural network on this cluster



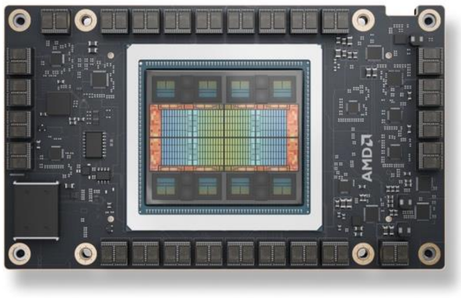
Google: Tensor Processing Units (TPUs)

Custom chips designed by Google

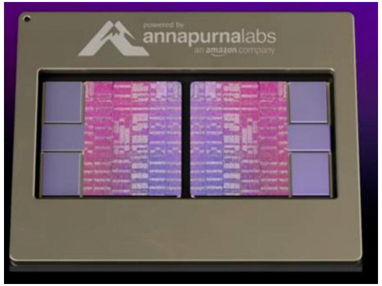
TPU v5p:
459 TFLOP/sec BF16 per chip
95GB of memory per chip
Arranged in pods of 8960 chips

Other Training Chips

AMD MI325X
1300 TFLOP/sec BF16
256GB memory

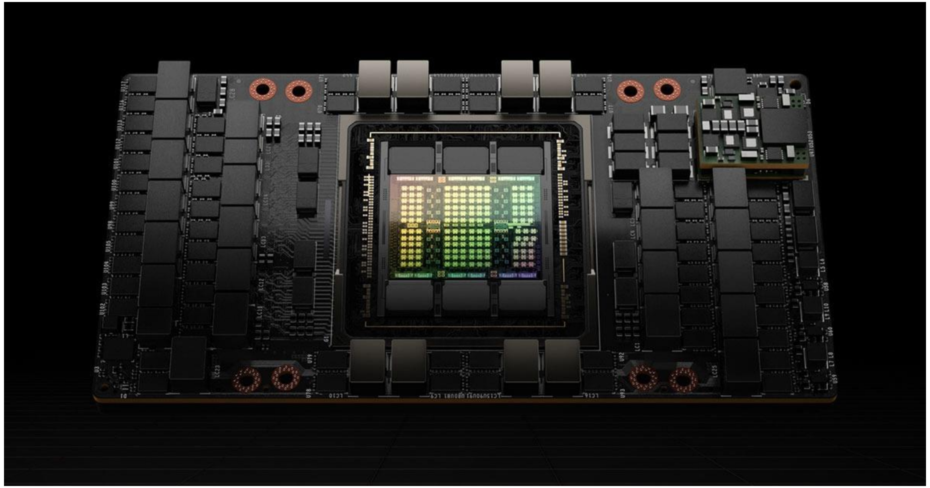


AWS Trainium2
667 TFLOP/sec BF16
96GB memory
Packed in UltraServers with 64 chips



Today: GPUs and How to Train On Them

A bit about GPU hardware



How to train on lots of GPUs

Today: GPUs and How to Train On Them

A bit about GPU hardware

How to train on lots of GPUs

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

$$L = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} \ell(x_{i,j}, W)$$

$$\frac{\partial L}{\partial W} = \frac{1}{M} \sum_{i=1}^{M} \left(\frac{1}{N} \sum_{j=1}^{N} \frac{\partial}{\partial W} \ell(x_{i,j}, W) \right)$$

Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

$$L = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\ell(x_{i,j}, W)$$

$$\frac{\partial L}{\partial W} = \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{N}\sum_{j=1}^{N}\frac{\partial}{\partial W}\ell(x_{i,j}, W)\right)$$

Each GPU computes gradient on N examples

Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

$$L = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}\ell(x_{i,j}, W)$$

$$\frac{\partial L}{\partial W} = \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{N}\sum_{j=1}^{N}\frac{\partial}{\partial W}\ell(x_{i,j}, W)\right)$$

Each GPU computes gradient on N examples

Average gradients across M GPUs

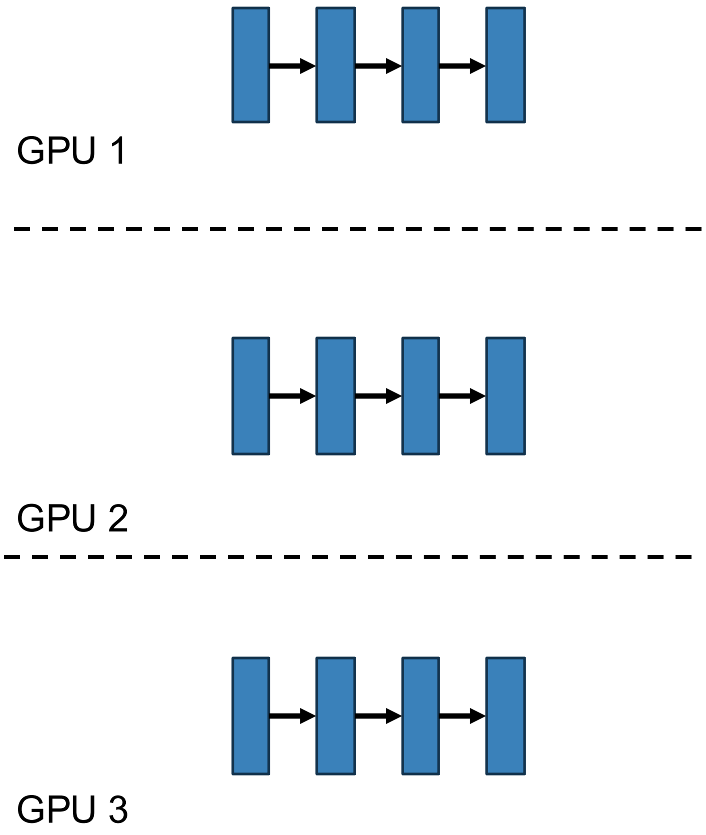
Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer



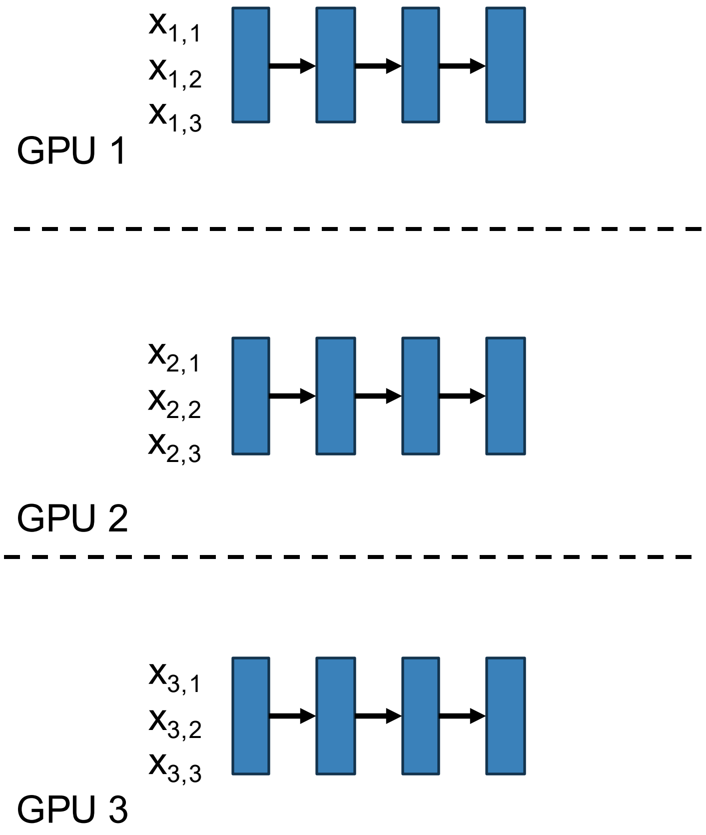
Data Parallelism

<u>Recall</u>: Loss is usually averaged over a minibatch of N samples

<u>Idea</u>: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer
2. Each GPU loads its own batch of data



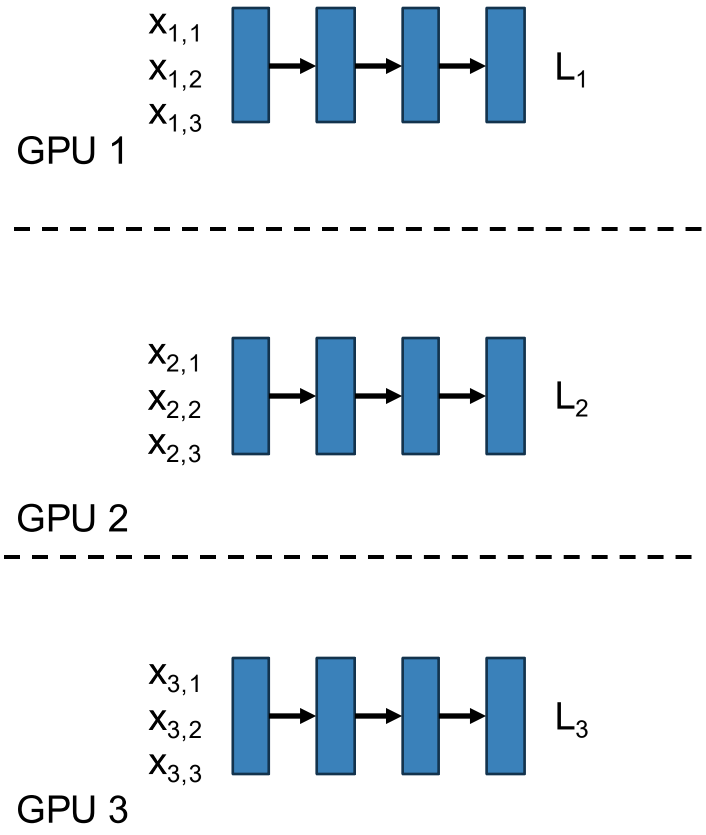
Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it’s own copy of model + optimizer
2. Each GPU loads its own batch of data
3. Each GPU runs forward to compute loss



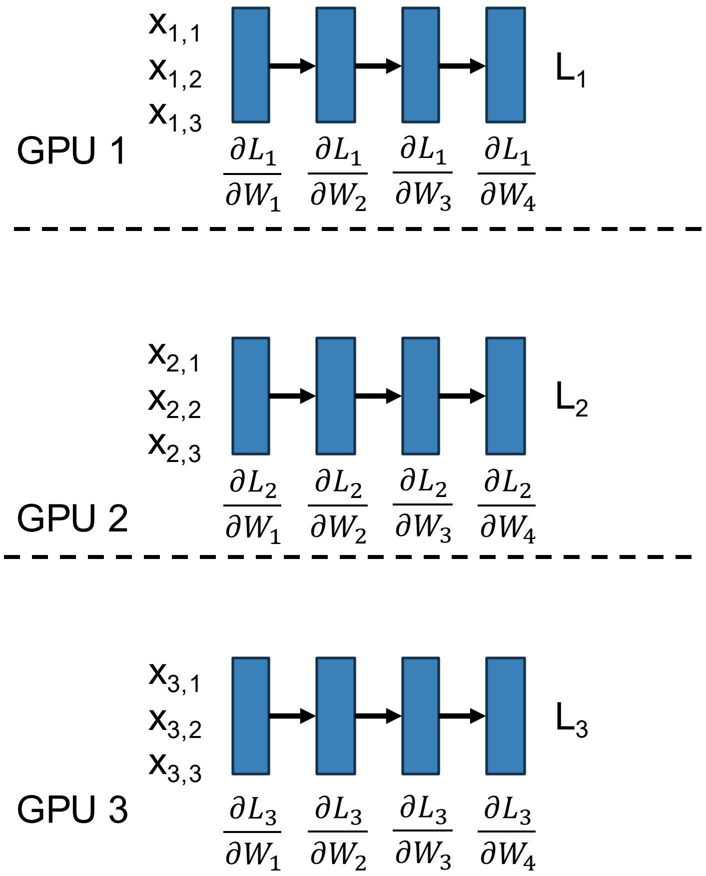
Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer
2. Each GPU loads its own batch of data
3. Each GPU runs forward to compute loss
4. Each GPU runs backward to compute gradients



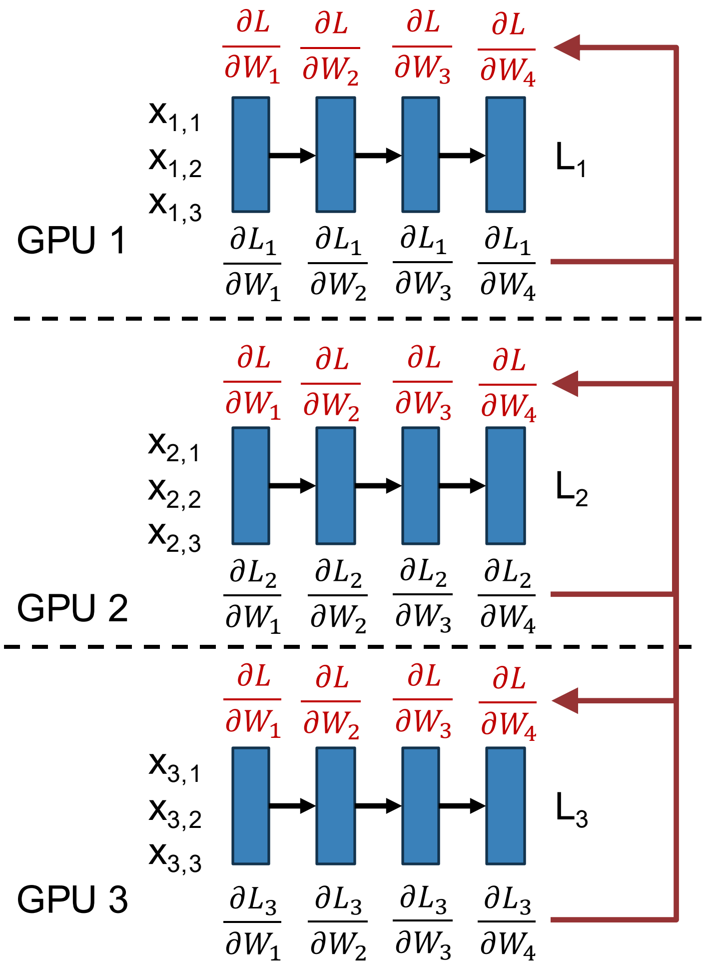
Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer
2. Each GPU loads its own batch of data
3. Each GPU runs forward to compute loss
4. Each GPU runs backward to compute gradients
5. Average gradients across all GPUs



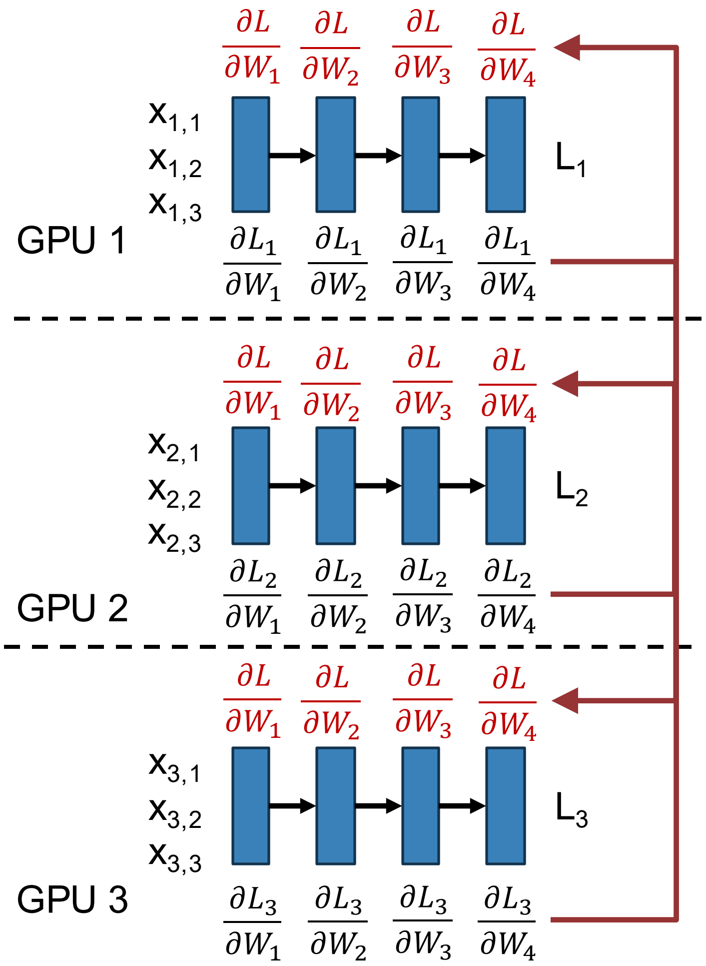
Data Parallelism

<u>Recall</u>: Loss is usually averaged over a minibatch of N samples

<u>Idea</u>: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer
2. Each GPU loads its own batch of data
3. Each GPU runs forward to compute loss
4. Each GPU runs backward to compute gradients
5. Average gradients across all GPUs
6. Each GPU updates its own weights



Data Parallelism

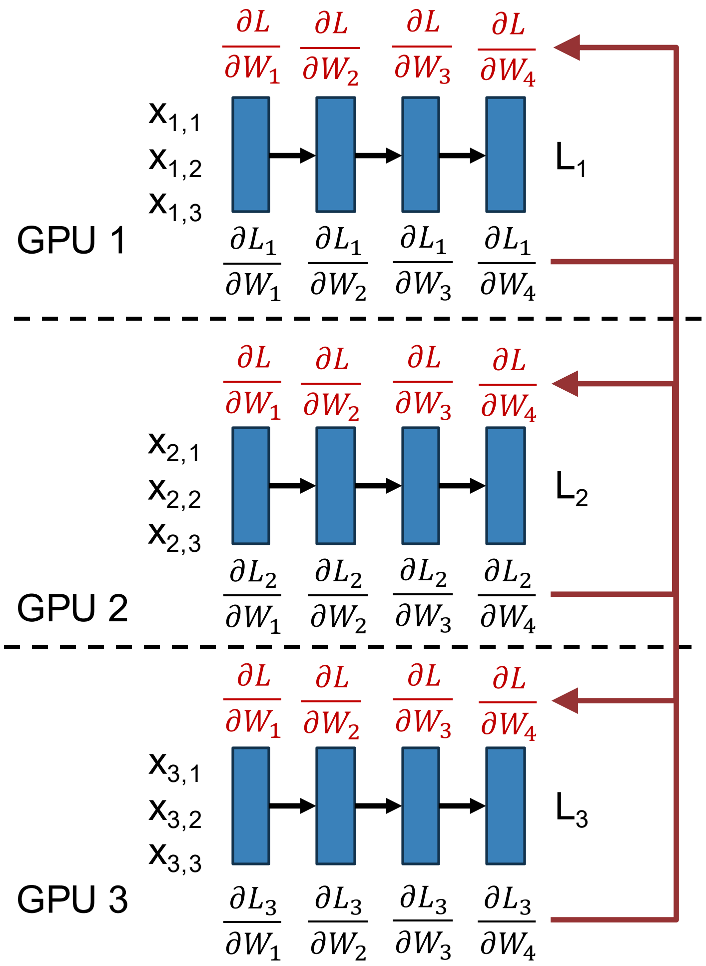
Recall: Loss is usually averaged over a minibatch of N samples

Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

1. Each GPU has it's own copy of model + optimizer
2. Each GPU loads its own batch of data
3. Each GPU runs forward to compute loss
4. Each GPU runs backward to compute gradients
5. Average gradients across all GPUs
6. Each GPU updates its own weights

(4) and (5) can run in parallel!



Data Parallelism

Recall: Loss is usually averaged over a minibatch of N samples

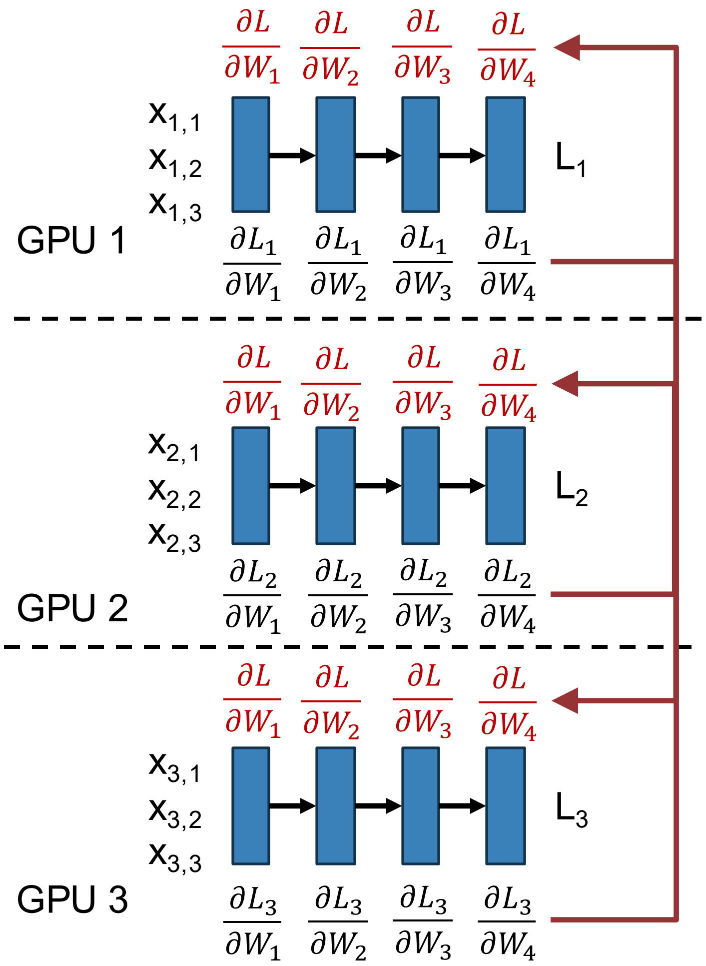
Idea: Use minibatch of MN samples, split over M GPUs

Gradients are linear, so each GPU computes its own gradient:

Problem: Model size constrained by GPU memory.

Each weight needs 4 numbers (weight, grad, Adam β_1, β_2). Each number needs 2 bytes.

1B params takes 8GB; 10B params fills up 80GB GPU.



Data Parallelism

<ins>Recall</ins>: Loss is usually averaged over a minibatch of N samples

<ins>Idea</ins>: Use minibatch of MN samples, split over M GPUs

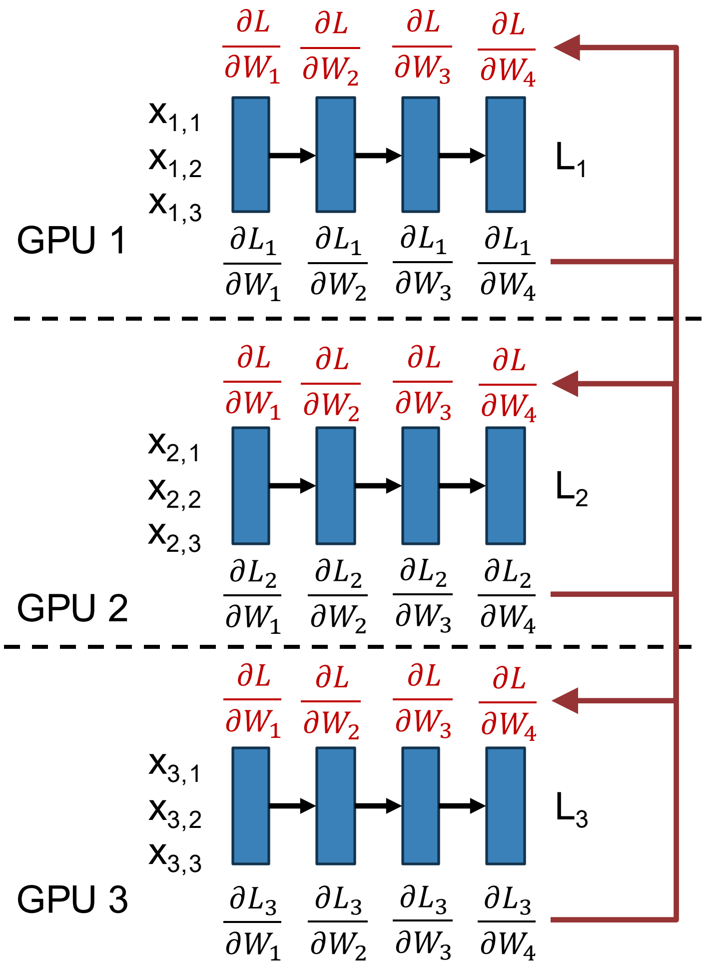
Gradients are linear, so each GPU computes its own gradient:

<ins>Problem</ins>: Model size constrained by GPU memory.

Each weight needs 4 numbers (weight, grad, Adam β_1, β_2). Each number needs 2 bytes.

1B params takes 8GB; 10B params fills up 80GB GPU.

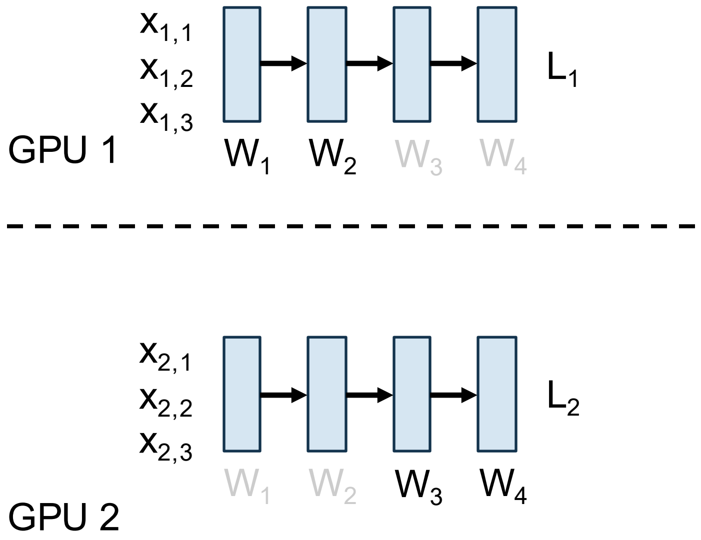
<ins>Solution</ins>: Split model weights across GPUs



Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states



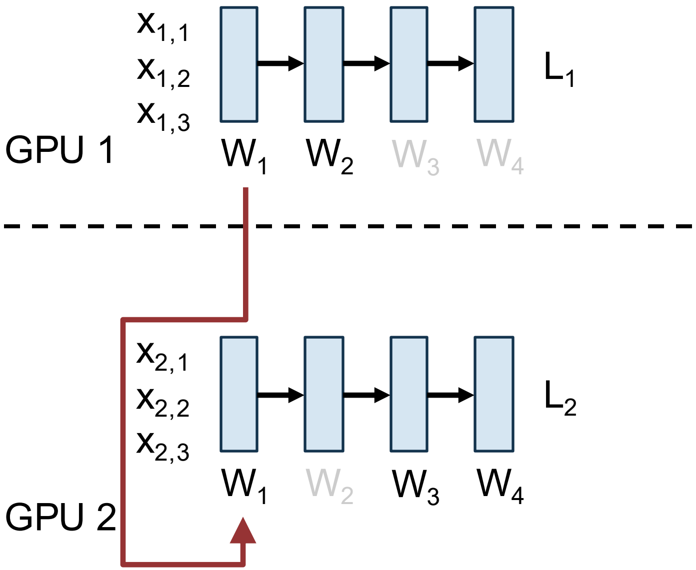
Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs



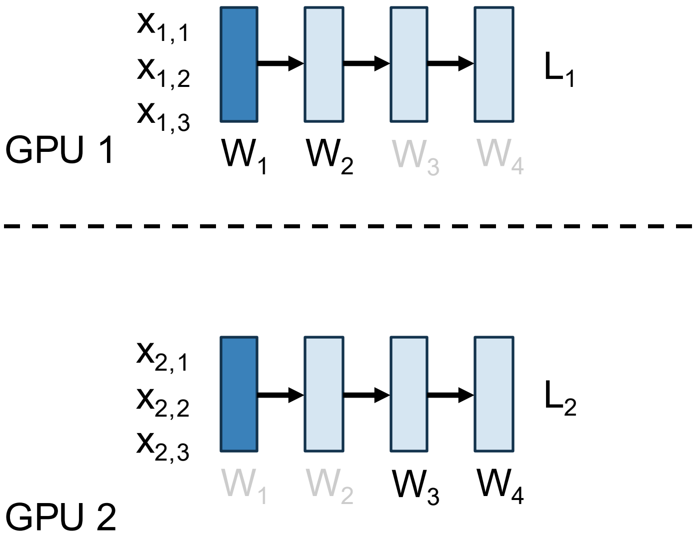
Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

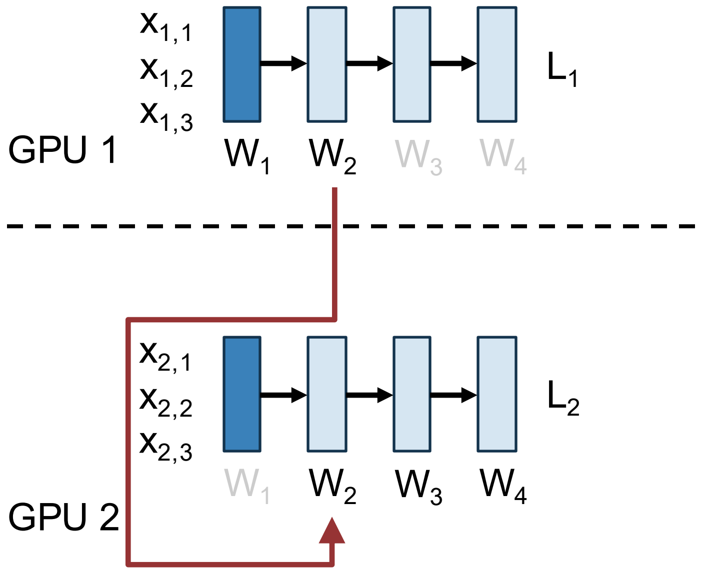
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

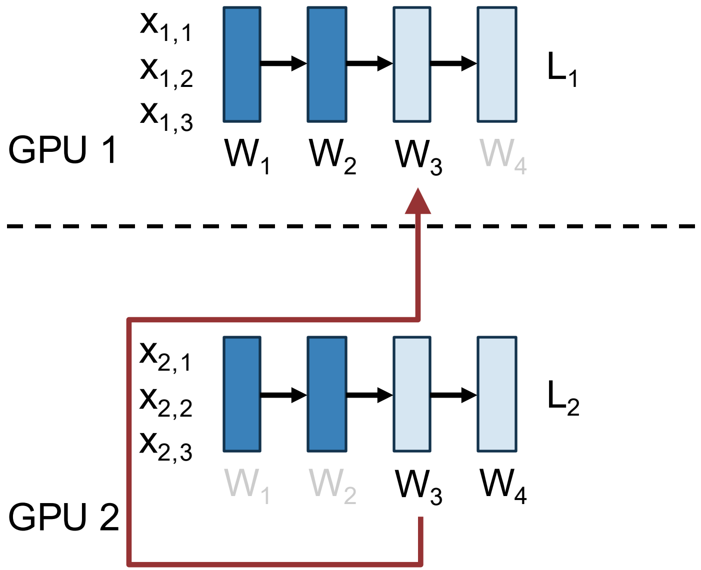
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

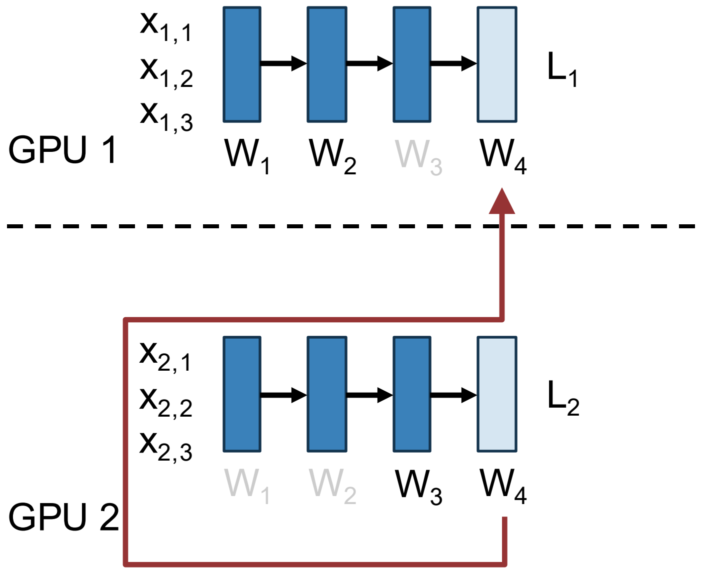
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

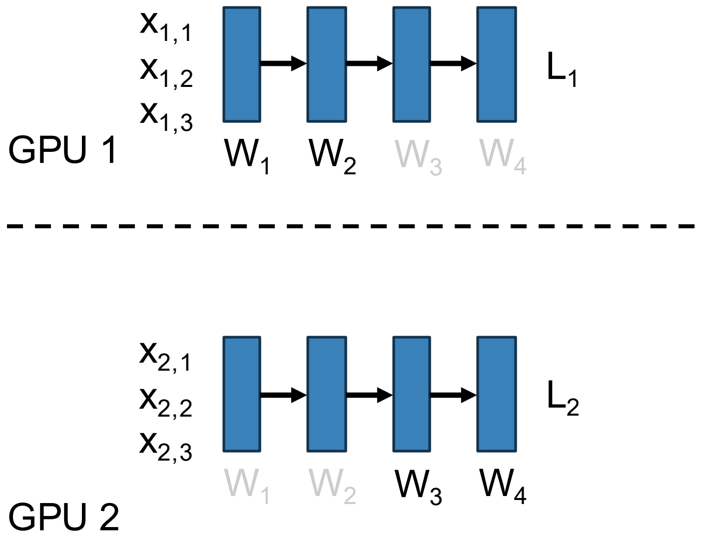
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

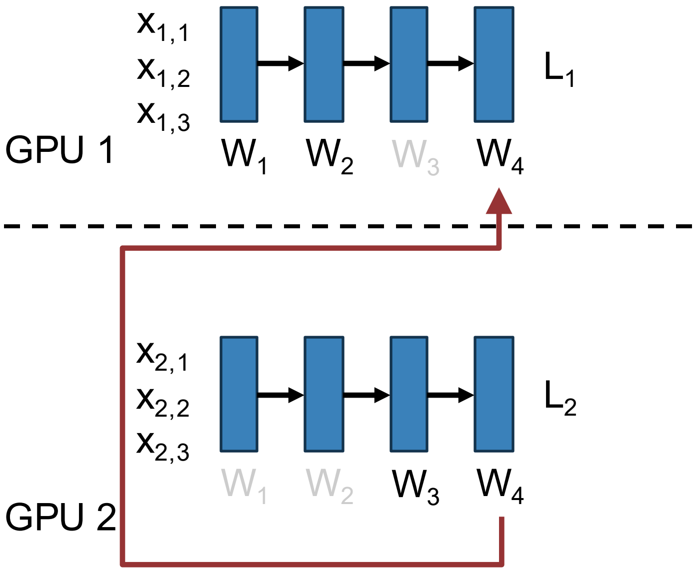
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

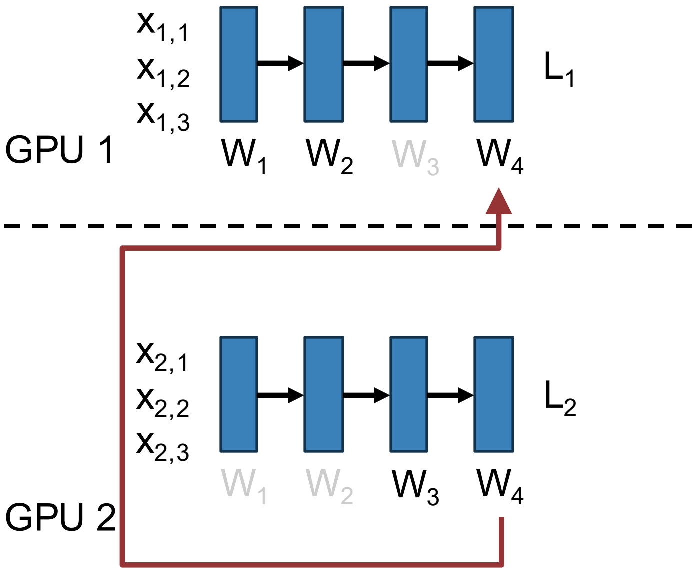
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

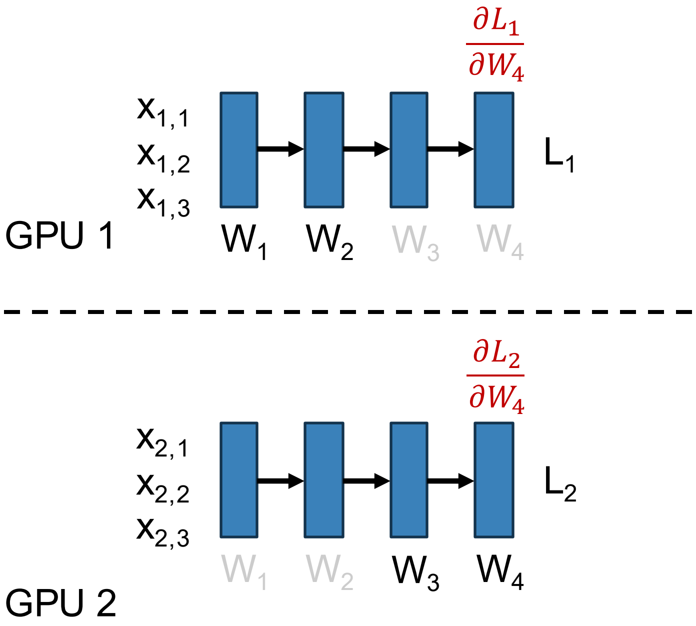
Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

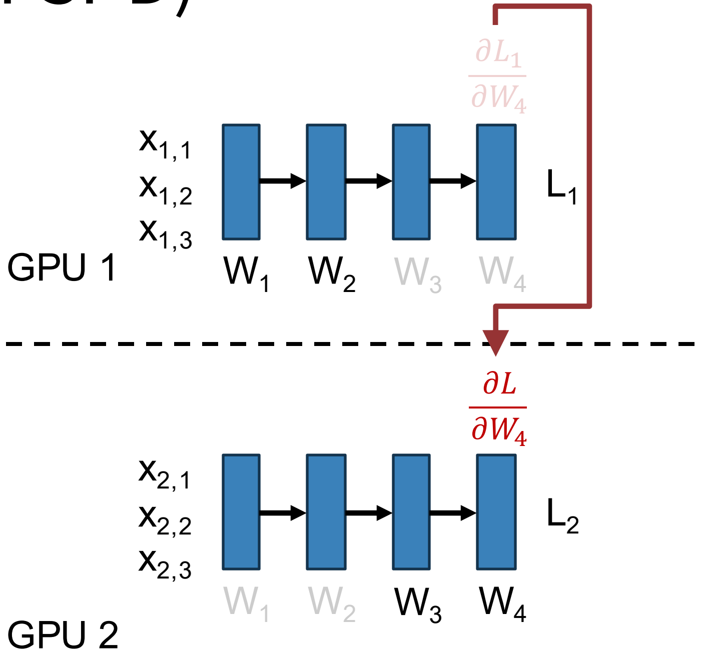
Optimization: don't delete last weight at end of forward to avoid immediately resending it

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

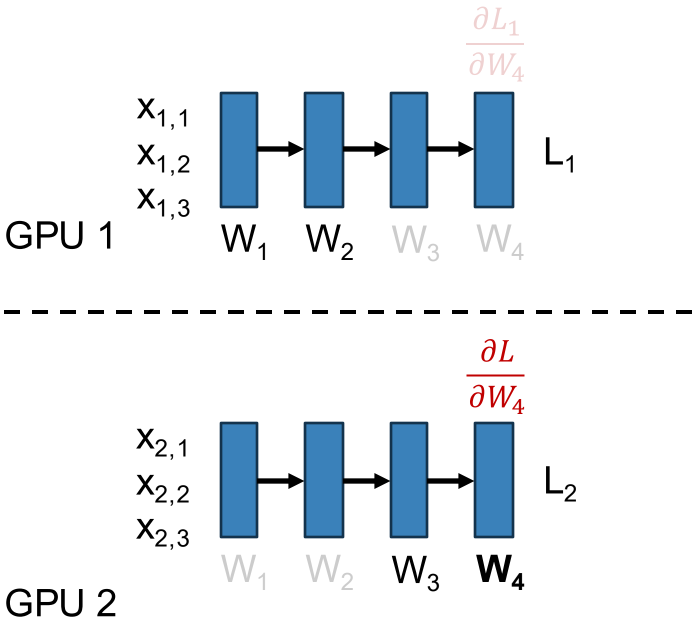
Optimization: don't delete last weight at end of forward to avoid immediately resending it

Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

Fully Sharded Data Parallelism (FSPD)

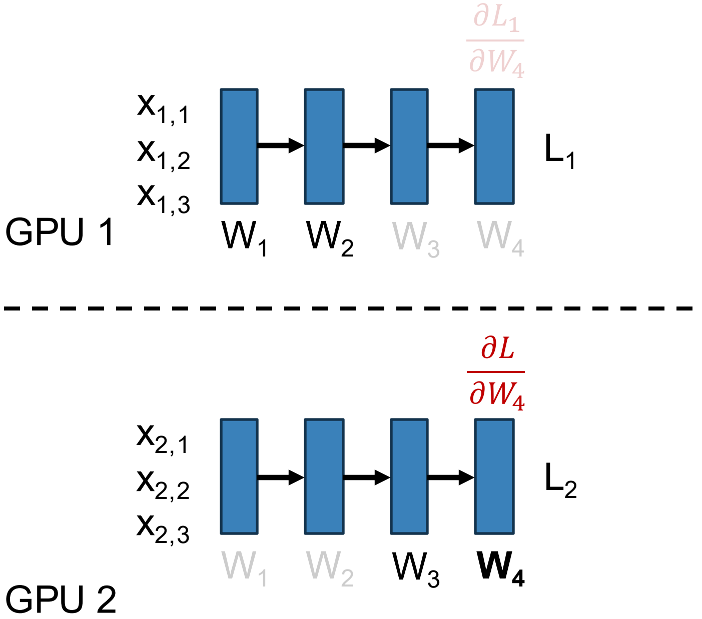
Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

Fully Sharded Data Parallelism (FSPD)

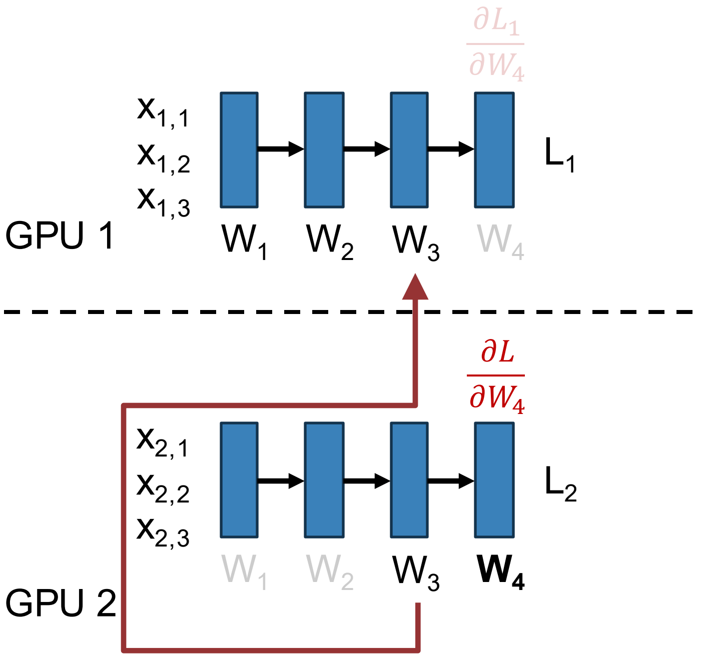
Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

Fully Sharded Data Parallelism (FSPD)

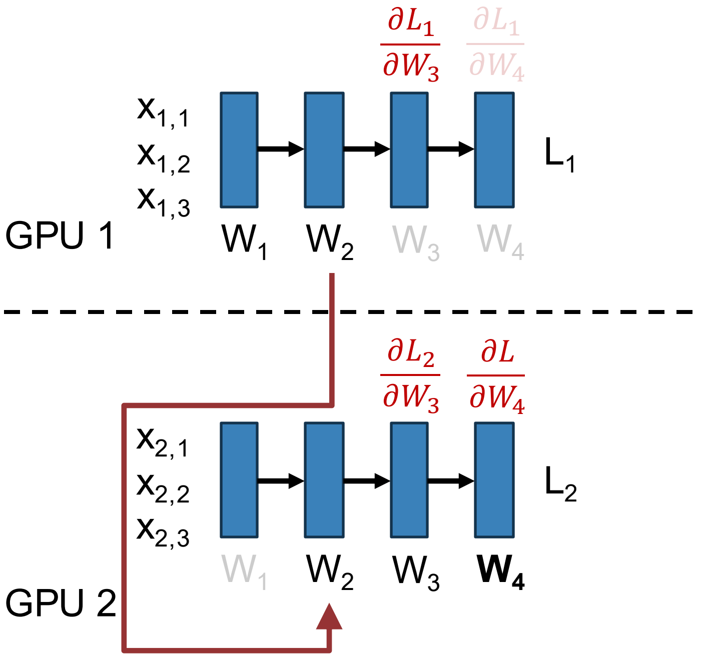
Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

Fully Sharded Data Parallelism (FSPD)

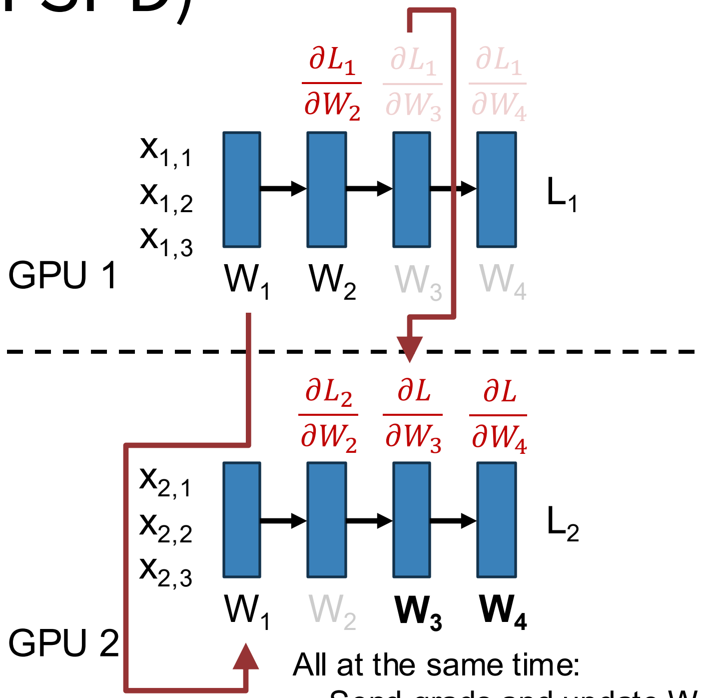
Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}



All at the same time:
- Send grads and update W_3
- Run backward with W_2
- Fetch W_1

Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

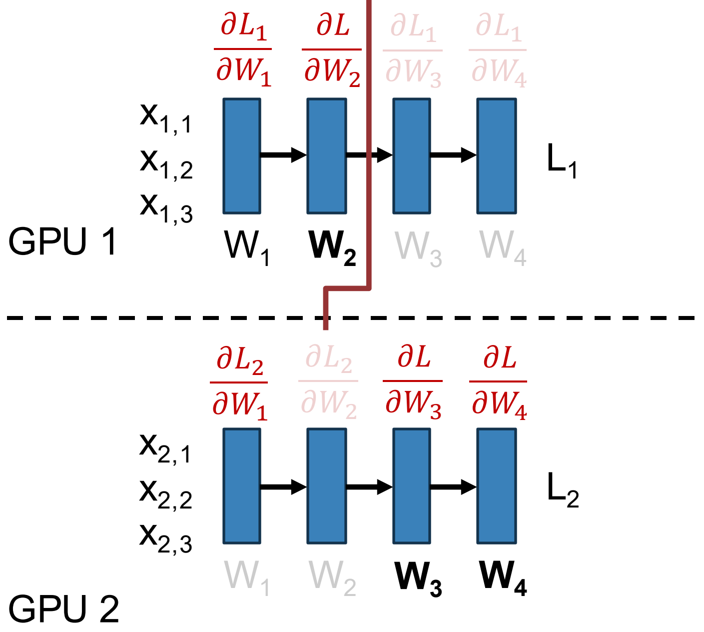
Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}

Optimization: don't delete last weight at end of forward to avoid immediately resending it



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Fully Sharded Data Parallelism (FSPD)

Split model weights across GPUs

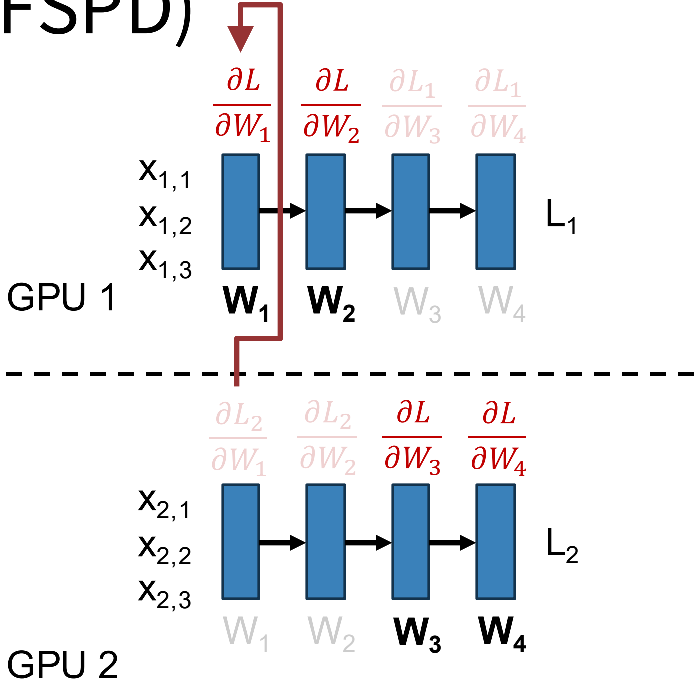
Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}

Optimization: don't delete last weight at end of forward to avoid immediately resending it



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Optimization: don't delete last weight at end of forward to avoid immediately resending it

Fully Sharded Data Parallelism (FSPD)

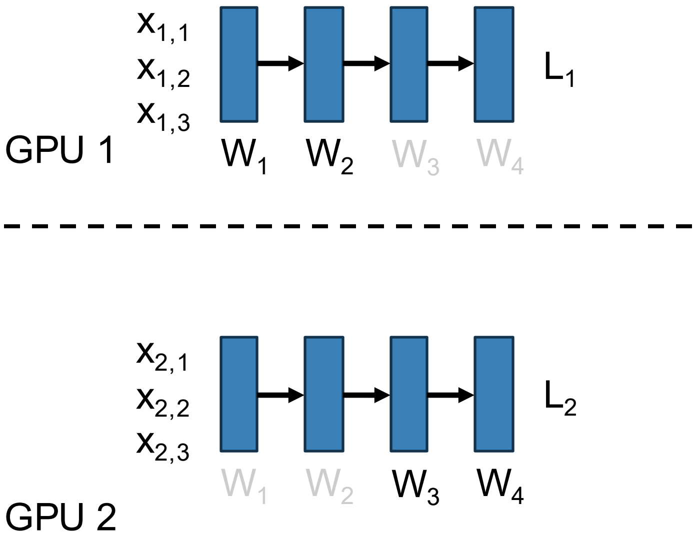
Split model weights across GPUs

Each weight W_i is owned by one GPU, which also holds its grads and optim states

1. Before forward for layer i, the GPU that owns W_i broadcasts it to all GPUs
2. All GPUs run forward for layer i, then delete their local copy of W_i
3. Before backward for layer i, owner broadcasts W_i to all GPUs
4. All GPUs run backward for layer i to compute local dL/dW_i and delete W_i
5. After backward for layer i, all GPUs send local dL/dW_i to owning GPU and delete local dL/dW_i
6. Owner of W_i makes gradient update

Fetch W_{i+1} while computing forward with W_i

Fetch W_i while computing with W_{i+1}; send dL/dW_i and update W_i while computing with W_{i-1}



Repeat with next batch of data
Data was being pre-fetched during forward+backward

Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

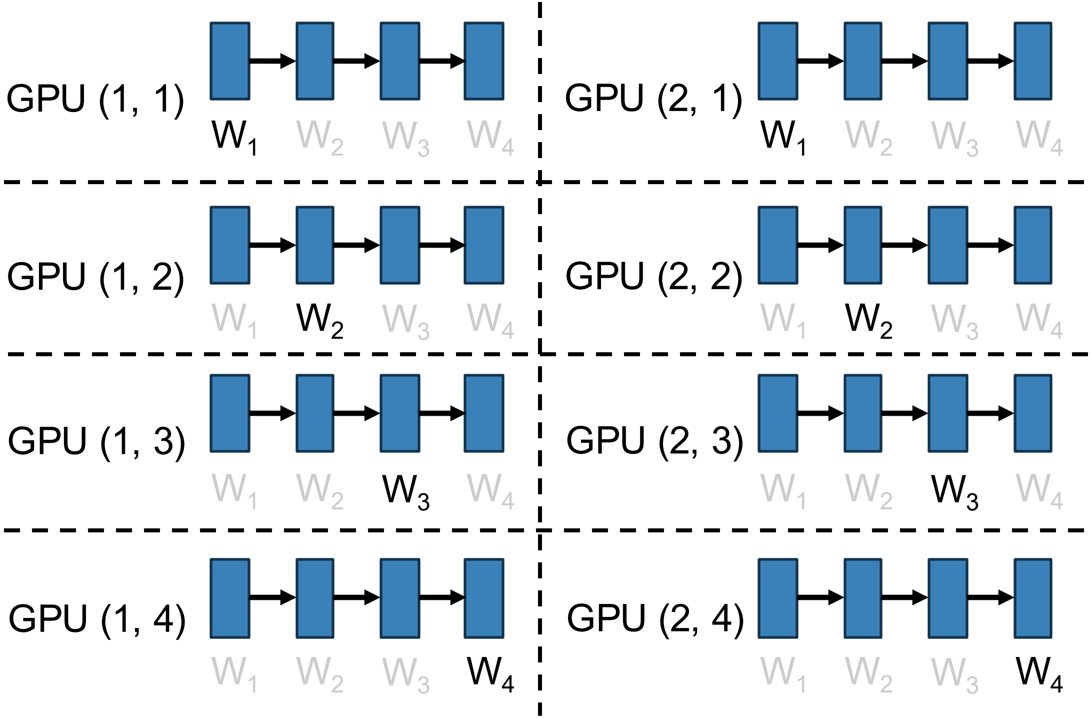
Hybrid Sharded Data Parallel (HSDP)

Split N = M*K GPUs into M groups of K

Each group of K GPUs does FSDP, splits model weights across all K GPUs. K can be O(100) GPUs.

Do DP across the M groups.

Example: HSDP with M=2 groups of K=4 GPUs



Hybrid Sharded Data Parallel (HSDP)

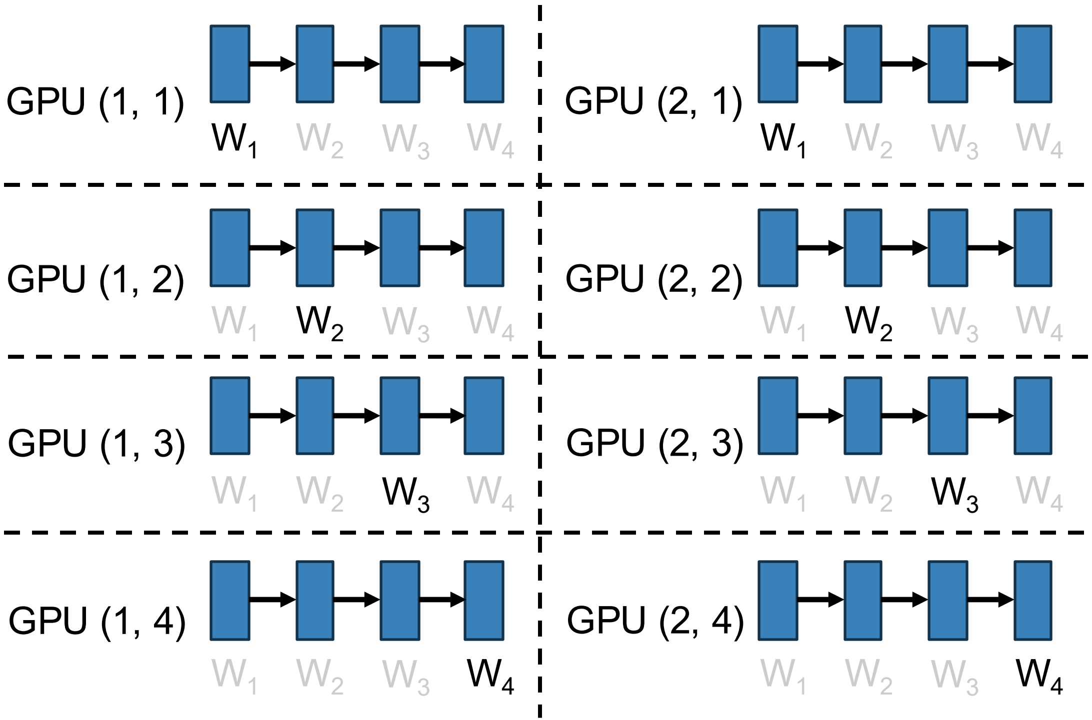
Split N = M*K GPUs into M groups of K

Each group of K GPUs does FSDP, splits model weights across all K GPUs. K can be O(100) GPUs.

Do DP across the M groups.

Multidimensional parallelism: Use different parallelism strategies at the same time! Organize GPUs in a 2D grid

Example: HSDP with M=2 groups of K=4 GPUs



Hybrid Sharded Data Parallel (HSDP)

Split N = M*K GPUs into M groups of K

Each group of K GPUs does FSDP, splits model weights across all K GPUs. K can be O(100) GPUs.

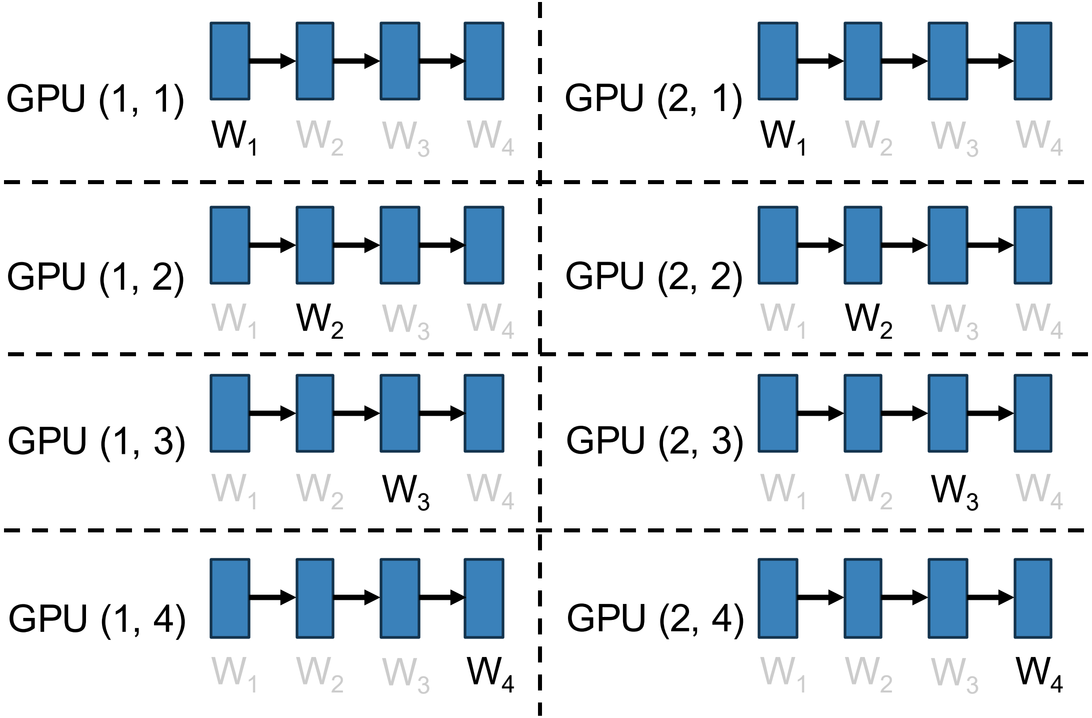
Do DP across the M groups.

Multidimensional parallelism: Use different parallelism strategies at the same time! Organize GPUs in a 2D grid

3x communication inside each group of K:
W in forward, W + dL/dW in backward.
Keep them in the same node / pod.

1x communication across the M groups: dL/dW in backward. Can use slower communication.

Example: HSDP with M=2 groups of K=4 GPUs

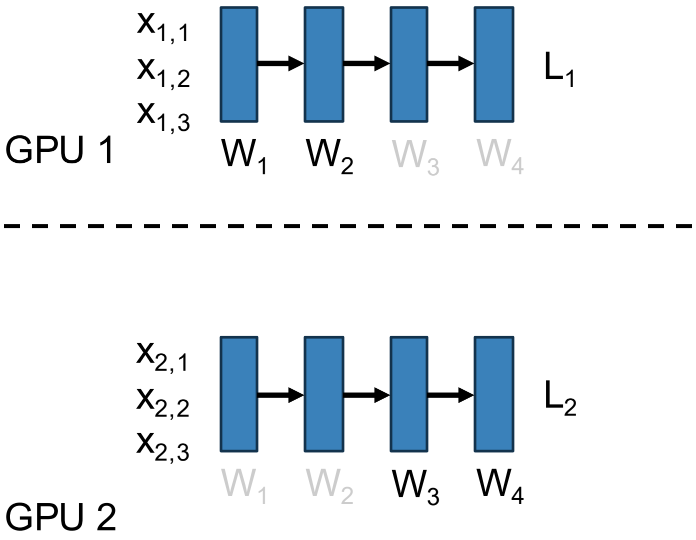


Data Parallelism (DP, FSPD, HSDP)

Split data and model weights across GPUs

Can now scale up to big models that don't fit in a single GPU!

A model with 100B params needs 4 numbers per param (param, grad, Adam β_1, β_1);
2 bytes per number takes 800GB;
splitting over 80 GPUs is just 10GB per GPU!



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

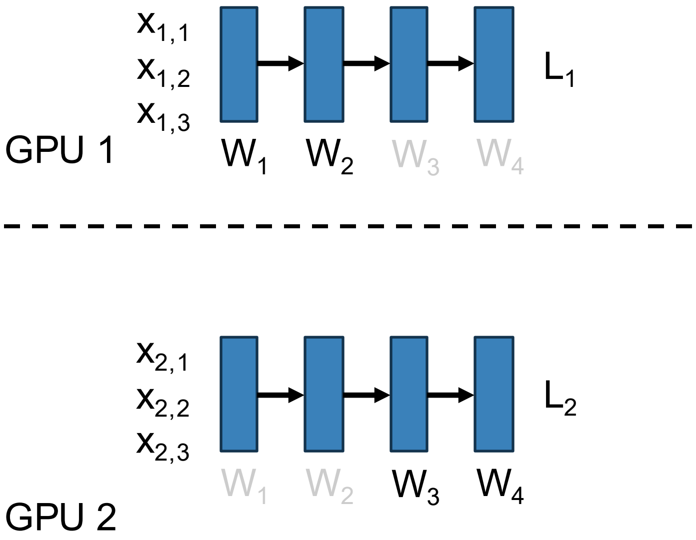
Data Parallelism (DP, FSPD, HSDP)

Split data and model weights across GPUs

Can now scale up to big models that don't fit in a single GPU!

A model with 100B params needs 4 numbers per param (param, grad, Adam β_1, β_1);
2 bytes per number takes 800GB;
splitting over 80 GPUs is just 10GB per GPU!

<u>Problem</u>: Model activations can fill up memory. Llama3-405B Transformer has 126 layers, D=16,384, seq length 4096. Just FFN hidden activations need 2*126*(4*16384)*4096 bytes = 63GB; plus need other activations.



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Data Parallelism (DP, FSPD, HSDP)

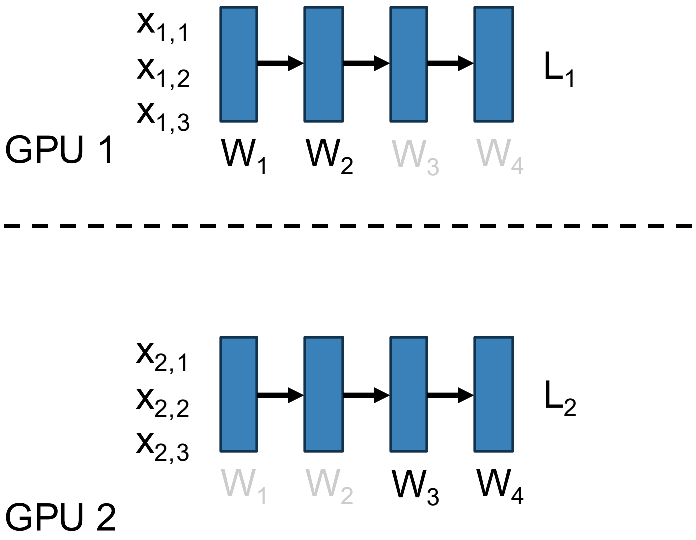
Split data and model weights across GPUs

Can now scale up to big models that don't fit in a single GPU!

A model with 100B params needs 4 numbers per param (param, grad, Adam β_1, β_1);
2 bytes per number takes 800GB;
splitting over 80 GPUs is just 10GB per GPU!

Problem: Model activations can fill up memory. Llama3-405B Transformer has 126 layers, D=16,384, seq length 4096. Just FFN hidden activations need 2*126*(4*16384)*4096 bytes = 63GB; plus need other activations.

Solution: Don't keep all activations in memory; recompute them on the fly!



Rajbhandrari et al, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models", arXiv 2019

Activation Checkpointing

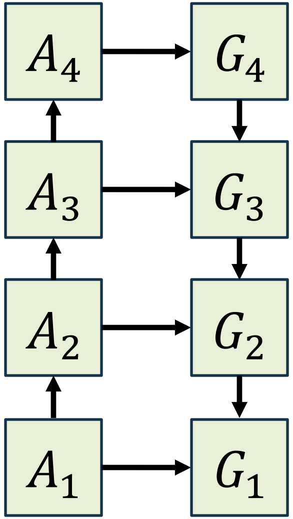
Each layer in the network is two functions:

<u>Forward</u>: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

<u>Backward</u>: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$



Activation Checkpointing

Each layer in the network is two functions:

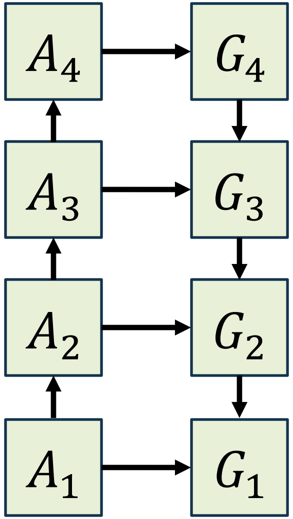
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

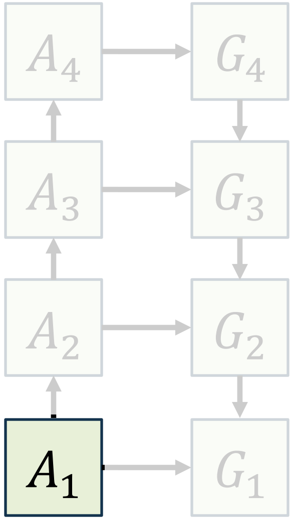
$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.



Activation Checkpointing

Compute: 1
Current Memory: 1
Peak Memory: 1



Each layer in the network is two functions:

Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.

Activation Checkpointing

Compute: 2
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

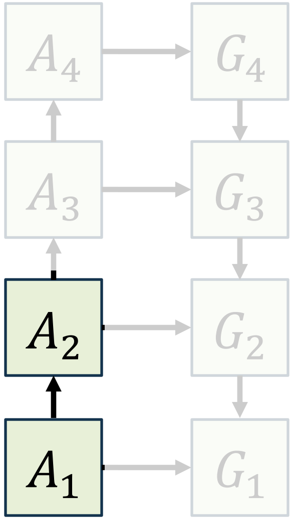
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.



Activation Checkpointing

Compute: 3
Current Memory: 3
Peak Memory: 3

Each layer in the network is two functions:

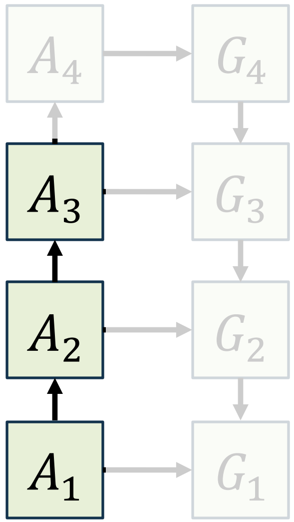
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

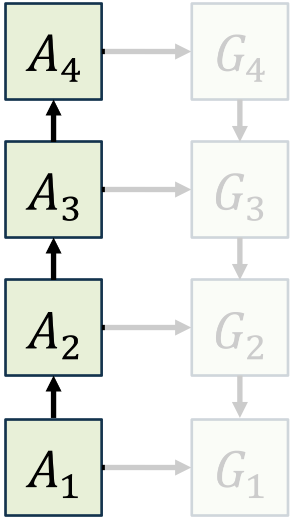
$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.



Activation Checkpointing

Compute: 4
Current Memory: 4
Peak Memory: 4



Each layer in the network is two functions:

Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

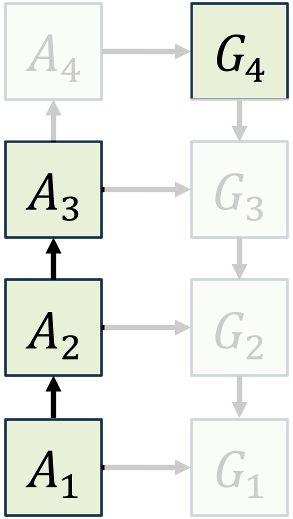
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.

Compute: 5
Current Memory: 4
Peak Memory: 4



Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

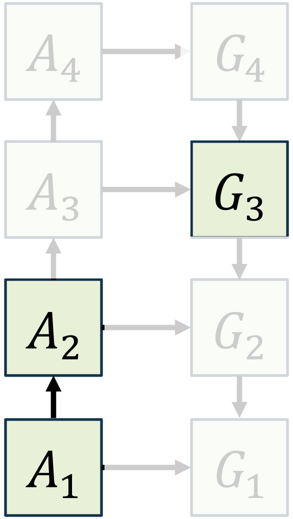
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.

Compute: 6
Current Memory: 3
Peak Memory: 4



Activation Checkpointing

Compute: 7
Current Memory: 2
Peak Memory: 4

Each layer in the network is two functions:

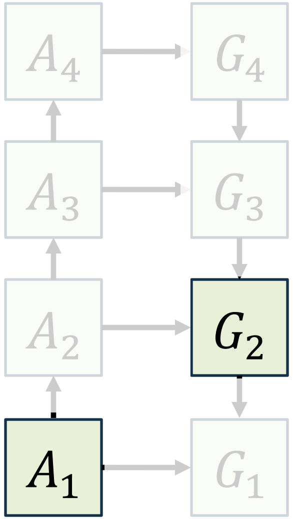
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.



Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

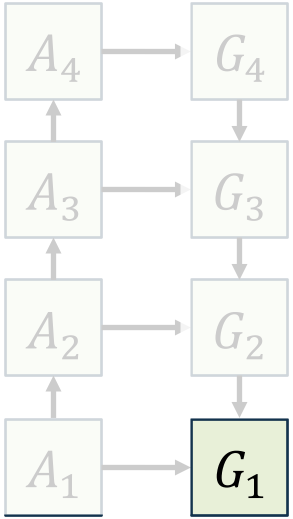
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Q: How much compute and memory does this take? Assume each $F_i^{\rightarrow}$ and $F_i^{\leftarrow}$ is O(1) compute and memory.

Compute: 8
Current Memory: 1
Peak Memory: 4



Activation Checkpointing

Compute: 8
Current Memory: 1
Peak Memory: 4

Each layer in the network is two functions:

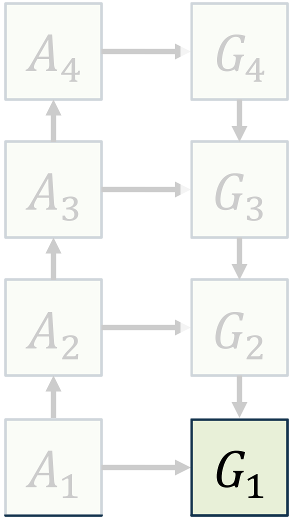
<u>Forward</u>: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

<u>Backward</u>: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Activation Checkpointing

Compute: 8
Current Memory: 1
Peak Memory: 4

Each layer in the network is two functions:

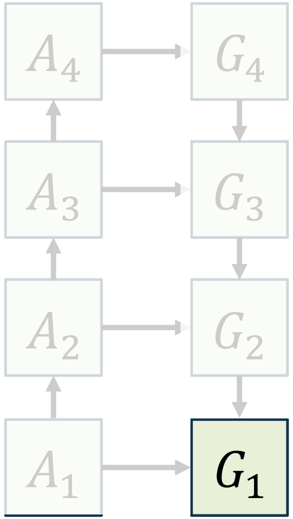
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 1
Current Memory: 1
Peak Memory: 1

Each layer in the network is two functions:

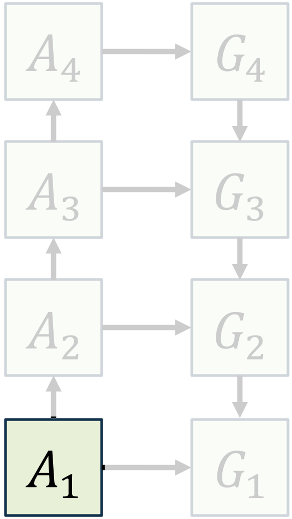
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

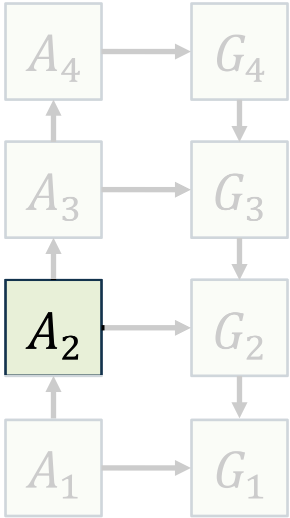
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory

Compute: 2
Current Memory: 1
Peak Memory: 1



Idea: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

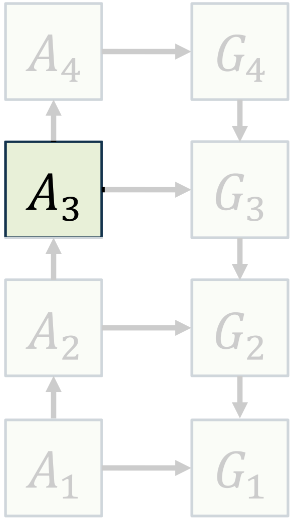
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory

Compute: 3
Current Memory: 1
Peak Memory: 1



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 4
Current Memory: 1
Peak Memory: 1

Each layer in the network is two functions:

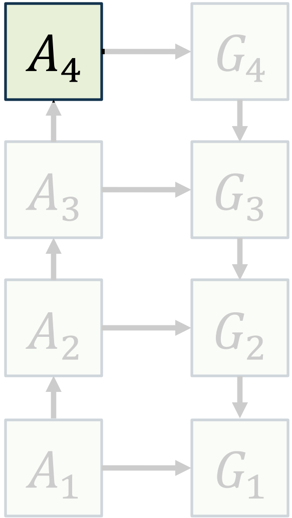
<u>Forward</u>: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

<u>Backward</u>: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



<u>Idea</u>: Recompute activations during the backward pass

Activation Checkpointing

Compute: 5
Current Memory: 1
Peak Memory: 1

Each layer in the network is two functions:

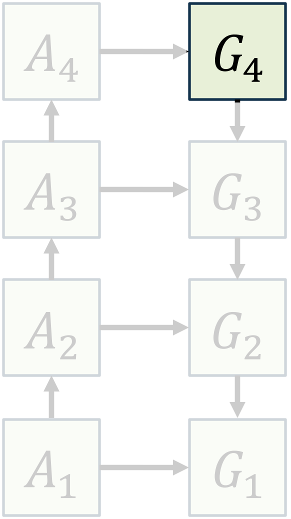
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 6
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

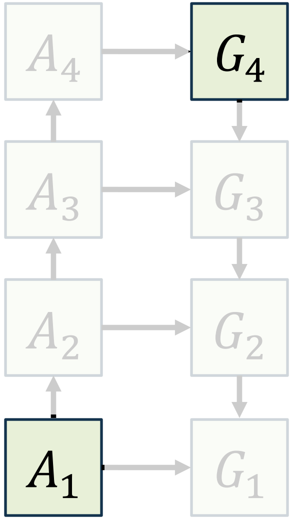
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 7
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

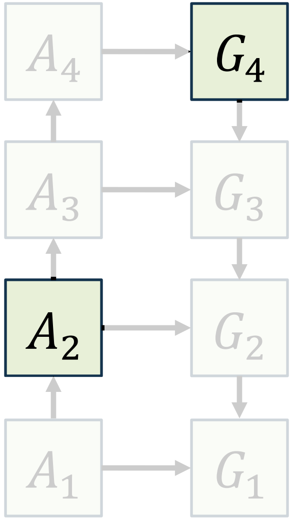
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 8
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

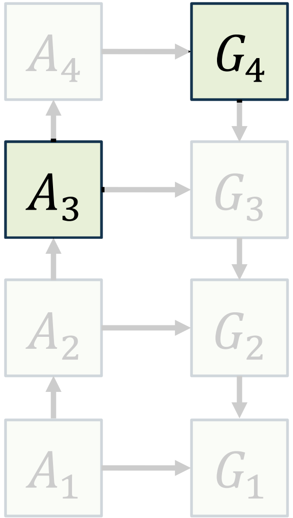
<u>Forward</u>: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

<u>Backward</u>: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



<u>Idea</u>: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

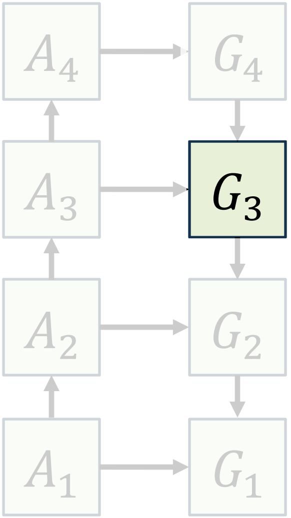
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory

Compute: 9
Current Memory: 1
Peak Memory: 2



Idea: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

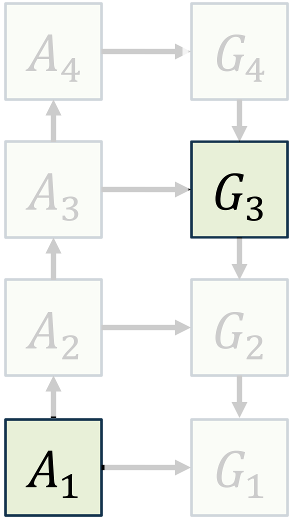
$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory

Compute: 10
Current Memory: 2
Peak Memory: 2



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 11
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

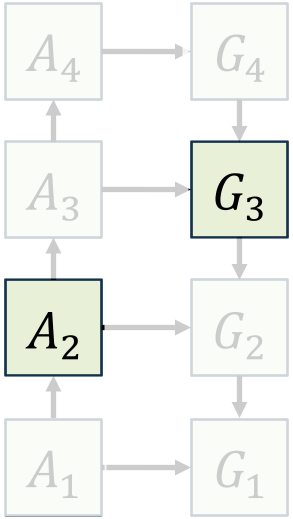
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 12
Current Memory: 1
Peak Memory: 2

Each layer in the network is two functions:

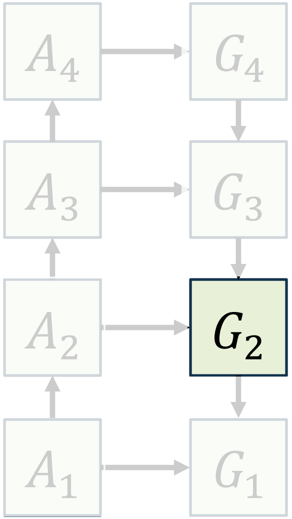
<u>Forward</u>: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

<u>Backward</u>: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



<u>Idea</u>: Recompute activations during the backward pass

Activation Checkpointing

Compute: 13
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

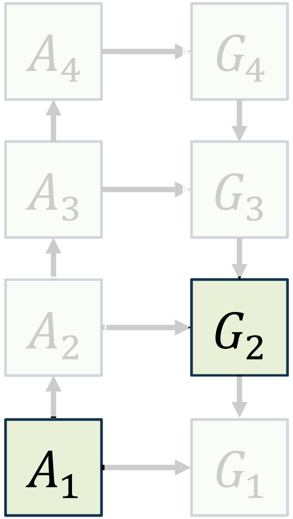
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 14
Current Memory: 1
Peak Memory: 2

Each layer in the network is two functions:

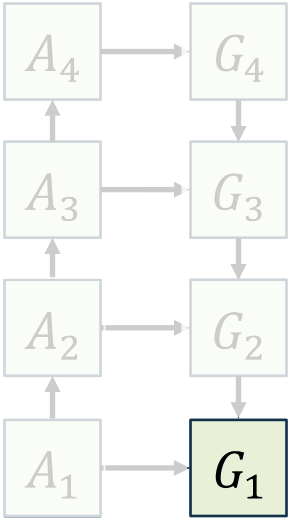
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Compute: 14
Current Memory: 1
Peak Memory: 2

Each layer in the network is two functions:

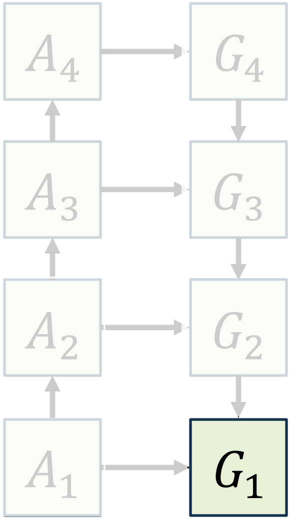
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory
Full Recomputation: O(N^2) compute, O(1) memory



Idea: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

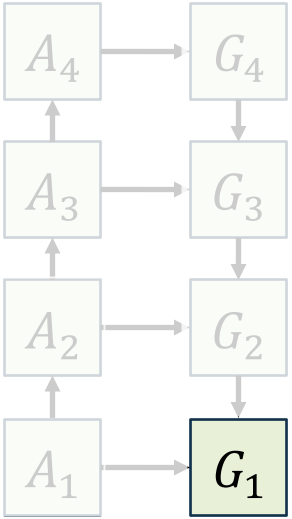
Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory
Full Recomputation: O(N^2) compute, O(1) memory

Problem: N^2 compute is bad!

Compute: 14
Current Memory: 2
Peak Memory: 2



Idea: Recompute activations during the backward pass

Activation Checkpointing

Each layer in the network is two functions:

Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

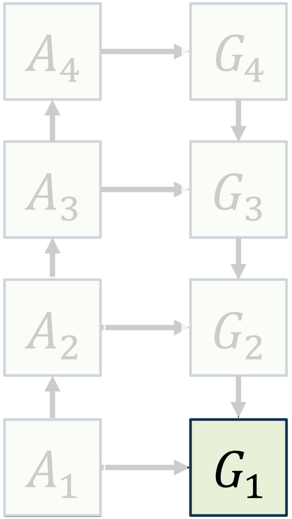
Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory
Full Recomputation: O(N^2) compute, O(1) memory

Problem: N^2 compute is bad!

Compute: 14
Current Memory: 2
Peak Memory: 2



Idea: Don't recompute everything; save a checkpoint every C layers

Activation Checkpointing

Compute: 14
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

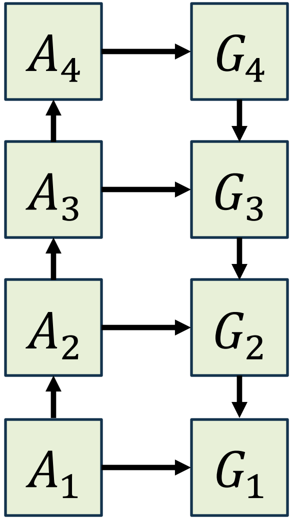
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory
Full Recomputation: O(N^2) compute, O(1) memory
C checkpoints: O(N^2/C) compute, O(C) memory



Idea: Don't recompute everything; save a checkpoint every C layers

Activation Checkpointing

Compute: 14
Current Memory: 2
Peak Memory: 2

Each layer in the network is two functions:

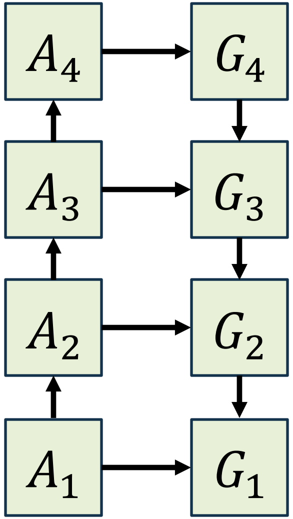
Forward: Compute next-layer activations

$$A_{i+1} = F_i^{\rightarrow}(A_i)$$

Backward: Compute prev-layer gradients

$$G_i = F_i^{\leftarrow}(A_i, G_{i+1})$$

Forward+backward: O(N) compute, O(N) memory
Full Recomputation: O(N^2) compute, O(1) memory
C checkpoints: O(N^2/C) compute, O(C) memory
$\sqrt{N}$ checkpoints: O(N $\sqrt{N}$) compute, O($\sqrt{N}$) memory



Idea: Don't recompute everything; save a checkpoint every C layers

How to train on lots of GPUs

HSDP + Activation checkpointing can take you a long way!

Scaling recipe:

1. Use **data parallelism** up to ~128 GPUs, models with ~1B params
2. Always set per-GPU batch size to max out GPU memory
3. If your model is >1B params, consider **FSDP**
4. Add **activation checkpointing** to fit larger batches per GPU
5. If you have >256 GPUs, consider **HSDP**
6. If you have >1K GPUs, models >50B params, or sequence lengths > 16K then use more advanced strategies (CP, PP, TP)

Problem: Lots of knobs to tune! How should we set them?
Solution: Maximize Model Flops Utilization (MFU)

Hardware FLOPs Utilization (HFU)

Recall: H100 can theoretically do 989.4 TFLOP/sec of 16-bit matrix multiplies on Tensor Cores

Question: How much throughput can we see in practice?

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Hardware FLOPs Utilization (HFU)

Recall: H100 can theoretically do 989.4 TFLOP/sec of 16-bit matrix multiplies on Tensor Cores

Question: How much throughput can we see in practice?

Hardware FLOPs Utilization (HFU): The fraction of theoretical matmul performance we actually achieve

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Hardware FLOPs Utilization (HFU)

Recall: H100 can theoretically do 989.4 TFLOP/sec of 16-bit matrix multiplies on Tensor Cores

Question: How much throughput can we see in practice?

Hardware FLOPs Utilization (HFU): The fraction of theoretical matmul performance we actually achieve

Benchmark for the best-case scenario: only matrix multiply

```
h100_tflop_per_sec = 989.4
sizes = [512, 1024, 2048, 4096,
         8192, 16_384, 32_768]
for N in sizes:
    x = torch.randn(N, N, device="cuda",
                    dtype=torch.bfloat16)
    flops = 2 * N * N * N
    times = []
    for i in range(12):
        t0 = time.time()
        y = x @ x
        if i > 2: times.append(time.time() - t0)
    sec = np.mean(times)
    tflops_per_sec = flops / sec / 10 ** 12
    hfu = 100 * tflops_per_sec / h100_tflop_per_sec
    print(f"N: {N}, "
          f"TFLOP/sec: {tflops_per_sec:.2f}, "
          f"HFU: {hfu:.2f}%")
```

Run this with CUDA_LAUNCH_BLOCKING=1, otherwise GPU kernels launch async and measurements are wrong

Chowdhery et al, “PaLM: Scaling Language Modeling with Pathways”, arXiv 2022

Hardware FLOPs Utilization (HFU)

Recall: H100 can theoretically do 989.4 TFLOP/sec of 16-bit matrix multiplies on Tensor Cores

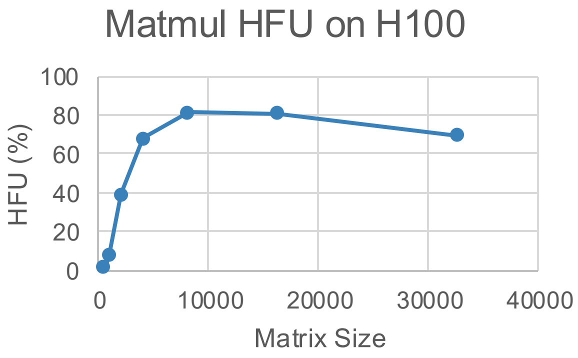
Question: How much throughput can we see in practice?

Hardware FLOPs Utilization (HFU): The fraction of theoretical matmul performance we actually achieve

Benchmark for the best-case scenario: only matrix multiply

```
h100_tflop_per_sec = 989.4
sizes = [512, 1024, 2048, 4096,
         8192, 16_384, 32_768]
for N in sizes:
    x = torch.randn(N, N, device="cuda",
                    dtype=torch.bfloat16)
    flops = 2 * N * N * N
    times = []
    for i in range(12):
        t0 = time.time()
        y = x @ x
        if i > 2: times.append(time.time() - t0)
    sec = np.mean(times)
    tflops_per_sec = flops / sec / 10 ** 12
    hfu = 100 * tflops_per_sec / h100_tflop_per_sec
    print(f"N: {N}, "
          f"TFLOP/sec: {tflops_per_sec:.2f}, "
          f"HFU: {hfu:.2f}%")
```

Large matrix multiply gets ~80% HFU on H100



Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Hardware FLOPs Utilization (HFU)

Recall: H100 can theoretically do 989.4 TFLOP/sec of 16-bit matrix multiplies on Tensor Cores

Question: How much throughput can we see in practice?

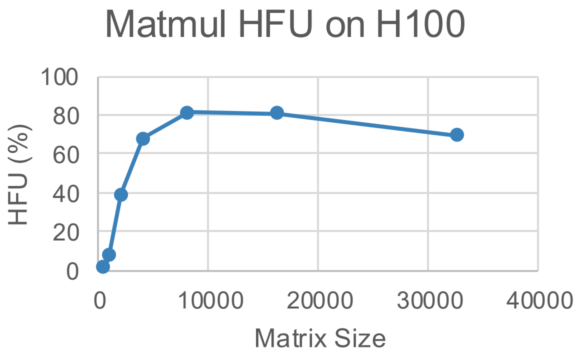
Hardware FLOPs Utilization (HFU): The fraction of theoretical matmul performance we actually achieve

Problem: HFU does not account for activation checkpointing or “helper” computation like data augmentation, optimizer, preprocessing

Benchmark for the best-case scenario: only matrix multiply

```python
h100_tflop_per_sec = 989.4
sizes = [512, 1024, 2048, 4096,
         8192, 16_384, 32_768]
for N in sizes:
    x = torch.randn(N, N, device="cuda",
                    dtype=torch.bfloat16)
    flops = 2 * N * N * N
    times = []
    for i in range(12):
        t0 = time.time()
        y = x @ x
        if i > 2: times.append(time.time() - t0)
    sec = np.mean(times)
    tflops_per_sec = flops / sec / 10 ** 12
    hfu = 100 * tflops_per_sec / h100_tflop_per_sec
    print(f"N: {N}, "
          f"TFLOP/sec: {tflops_per_sec:.2f}, "
          f"HFU: {hfu:.2f}%")
```

Large matrix multiply gets ~80% HFU on H100



Chowdhery et al, “PaLM: Scaling Language Modeling with Pathways”, arXiv 2022

Model FLOPs Utilization (MFU)

Idea: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

1. Compute $\text{FLOP}_{\text{theoretical}}$ = total number of matrix multiply FLOPs in the forward + backward pass
   (can approximate backward = 2x forward)
   (Ignore nonlinearities, normalization, elementwise ops like residuals. They will run on FP32 cores)
2. Look up $\text{FLOP/sec}_{\text{theoretical}}$ = theoretical max throughput of your device (H100: 989 TFLOP/sec)
3. Compute $t_{\text{theoretical}} = \text{FLOP}_{\text{theoretical}} / \text{FLOP/sec}_{\text{theoretical}}$
4. Measure t_{actual} = Actual time for a full iteration of data loading, forward, backward, optimizer step
5. $\text{MFU} = t_{\text{theoretical}} / t_{\text{actual}}$

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Model FLOPs Utilization (MFU)

Idea: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

1. Compute $\text{FLOP}_{\text{theoretical}}$ = total number of matrix multiply FLOPs in the forward + backward pass (can approximate backward = 2x forward) (Ignore nonlinearities, normalization, elementwise ops like residuals. They will run on FP32 cores)
2. Look up $\text{FLOP/sec}_{\text{theoretical}}$ = theoretical max throughput of your device (H100: 989 TFLOP/sec)
3. Compute $t_{\text{theoretical}} = \text{FLOP}_{\text{theoretical}} / \text{FLOP/sec}_{\text{theoretical}}$
4. Measure t_{actual} = Actual time for a full iteration of data loading, forward, backward, optimizer step
5. $\text{MFU} = t_{\text{theoretical}} / t_{\text{actual}}$

```python
L, D, N = 8, 8192, 8192

flop_fwd = N * L * 2 * D * D
flop_bwd = 2 * flop_fwd
flop_theoretical = flop_fwd + flop_bwd
t_theoretical = flop_theoretical / (989.4 * 10 ** 12)

layers = []
for _ in range(L):
    layers += [torch.nn.Linear(D, D), torch.nn.ReLU()]
model = torch.nn.Sequential(*layers).cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

for _ in range(20):
    torch.cuda.synchronize()
    t0 = time.time()
    x = torch.randn(
        N, D, device="cuda",
        dtype=torch.float32
    )
    with torch.autocast(
        device_type="cuda",
        dtype=torch.bfloat16,
    ):
        y = model(x)
    loss = ((x - y) ** 2.0).sum()
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    torch.cuda.synchronize()
    t_actual = time.time() - t0
    mfu = t_theoretical / t_actual
    print(f"MFU: {100*mfu:.2f}%")
```

Example: Wide MLP with big batch size gets ~49% MFU on H100

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Model FLOPs Utilization (MFU)

<u>Idea</u>: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

1. Compute $\text{FLOP}_{\text{theoretical}}$ = total number of matrix multiply FLOPs in the forward + backward pass (can approximate backward = 2x forward) (Ignore nonlinearities, normalization, elementwise ops like residuals. They will run on FP32 cores)
2. Look up $\text{FLOP/sec}_{\text{theoretical}}$ = theoretical max throughput of your device (H100: 989 TFLOP/sec)
3. Compute $t_{\text{theoretical}} = \text{FLOP}_{\text{theoretical}} / \text{FLOP/sec}_{\text{theoretical}}$
4. Measure t_{actual} = Actual time for a full iteration of data loading, forward, backward, optimizer step
5. $\text{MFU} = t_{\text{theoretical}} / t_{\text{actual}}$

Optimize distributed training setup to maximize MFU!

```
L, D, N = 8, 8192, 8192

flop_fwd = N * L * 2 * D * D
flop_bwd = 2 * flop_fwd
flop_theoretical = flop_fwd + flop_bwd
t_theoretical = flop_theoretical / (989.4 * 10 ** 12)

layers = []
for _ in range(L):
    layers += [torch.nn.Linear(D, D), torch.nn.ReLU()]
model = torch.nn.Sequential(*layers).cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

for _ in range(20):
    torch.cuda.synchronize()
    t0 = time.time()
    x = torch.randn(
        N, D, device="cuda",
        dtype=torch.float32
    )
    with torch.autocast(
        device_type="cuda",
        dtype=torch.bfloat16,
    ):
        y = model(x)
    loss = ((x - y) ** 2.0).sum()
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    torch.cuda.synchronize()
    t_actual = time.time() - t0
    mfu = t_theoretical / t_actual
    print(f"MFU: {100*mfu:.2f}%")
```

<u>Example</u>: Wide MLP with big batch size gets ~49% MFU on H100

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Model FLOPs Utilization (MFU)

Idea: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

MFU >30% is good, >40% is excellent

Model FLOPs Utilization (MFU)

Idea: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

MFU >30% is good, >40% is excellent

Model	# of Parameters (in billions)	Accelerator chips	Model FLOPS utilization
GPT-3	175B	V100	21.3%
Gopher	280B	4096 TPU v3	32.5%
Megatron-Turing NLG	530B	2240 A100	30.2%
PaLM	540B	6144 TPU v4	46.2%

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

Example: Llama3-405B training on H100 GPUs

GPUs	TFLOPs/GPU	BF16 MFU
8,192	430	43%
16,384	400	41%
16,384	380	38%

Llama Team, "The Llama3 Herd of Models", arXiv 2024

Model FLOPs Utilization (MFU)

Idea: What fraction of the GPU's theoretical peak FLOPs is being used for "useful" model computation?

MFU >30% is good, >40% is excellent

Model	# of Parameters (in billions)	Accelerator chips	Model FLOPS utilization
GPT-3	175B	V100	21.3%
Gopher	280B	4096 TPU v3	32.5%
Megatron-Turing NLG	530B	2240 A100	30.2%
PaLM	540B	6144 TPU v4	46.2%

Chowdhery et al, "PaLM: Scaling Language Modeling with Pathways", arXiv 2022

More recent devices sometimes get *worse* MFU since their peak FLOPs increases much faster than their memory bandwidth

Example: Llama3-405B training on H100 GPUs

GPUs	TFLOPs/GPU	BF16 MFU
8,192	430	43%
16,384	400	41%
16,384	380	38%

Llama Team, "The Llama3 Herd of Models", arXiv 2024

A100 => H100:
3.1x FLOPs
2.1x memory bandwidth

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

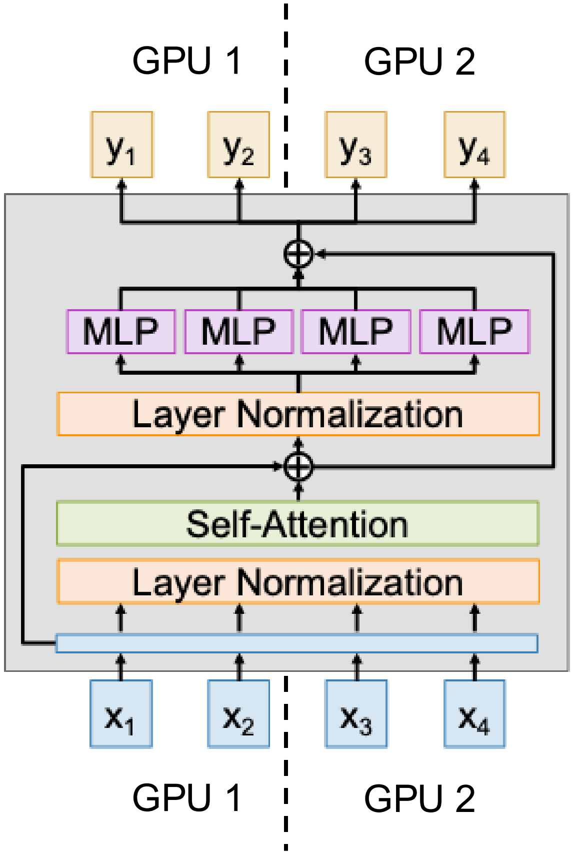
Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

Context Parallelism (CP)

(Usually for Transformers)

Idea: Transformers operate on L-length sequences.
Use multiple GPUs to process a single long sequence

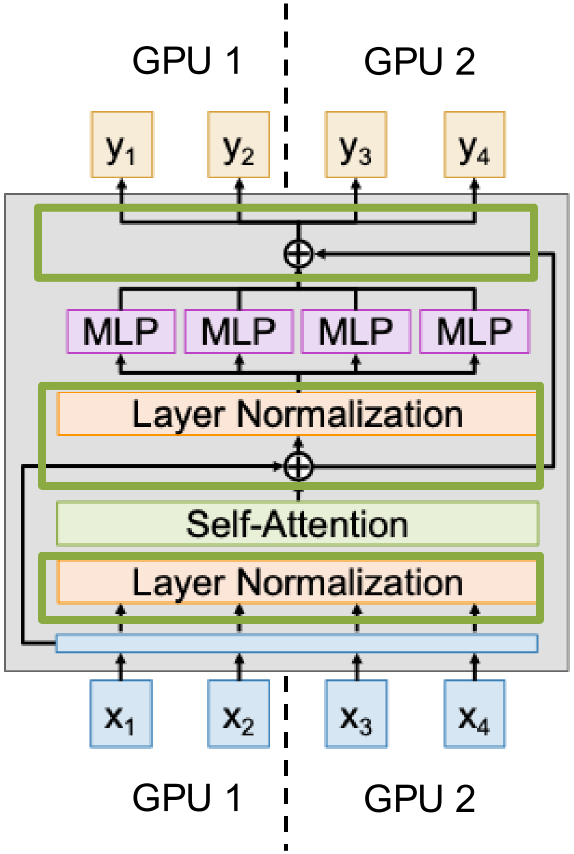


Context Parallelism (CP)

(Usually for Transformers)

Idea: Transformers operate on L-length sequences. Use multiple GPUs to process a single long sequence

Normalization, residual connections: Easy, they have no weights and trivially parallelizable



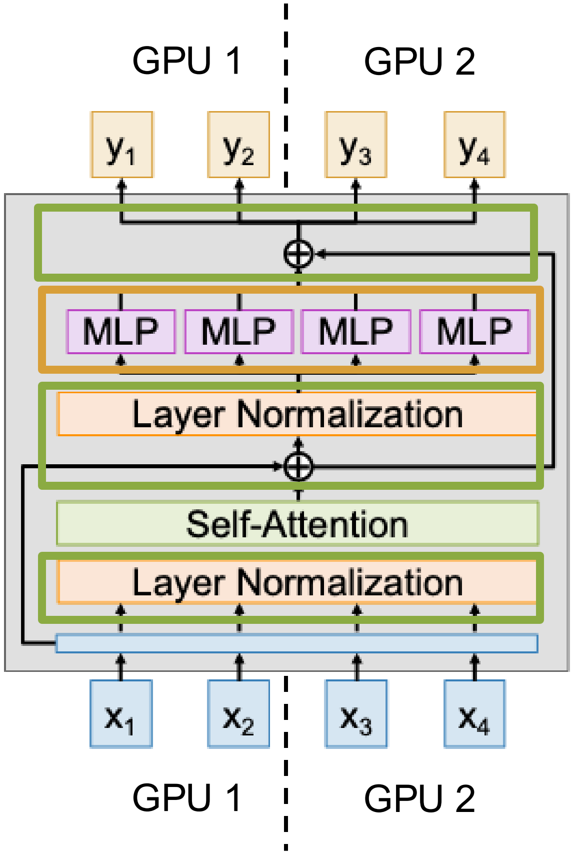
Context Parallelism (CP)

(Usually for Transformers)

Idea: Transformers operate on L-length sequences. Use multiple GPUs to process a single long sequence

Normalization, residual connections: Easy, they have no weights and trivially parallelizable

MLP: Trivially parallelizable, but has weights. Each GPU keeps a copy of the weights and communicates gradients like in DP



Context Parallelism (CP)

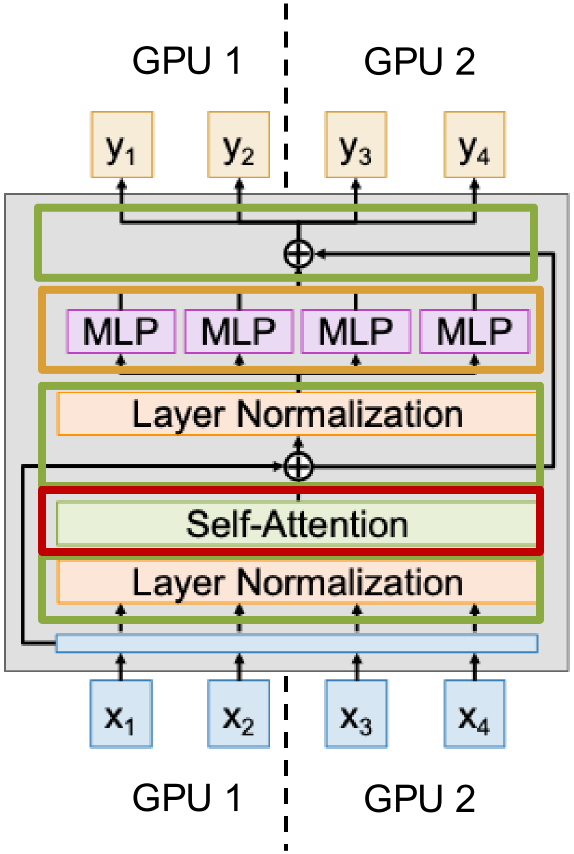
(Usually for Transformers)

<u>Idea</u>: Transformers operate on L-length sequences. Use multiple GPUs to process a single long sequence

Normalization, residual connections: Easy, they have no weights and trivially parallelizable

MLP: Trivially parallelizable, but has weights. Each GPU keeps a copy of the weights and communicates gradients like in DP

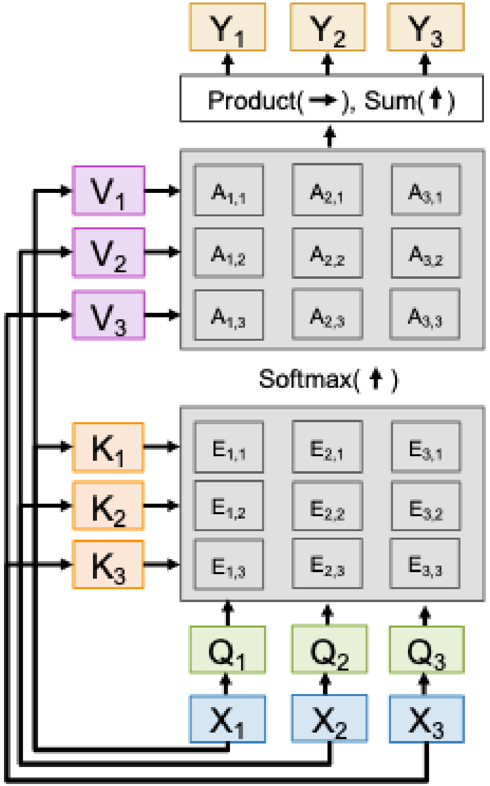
Attention: More complex, need to dig in



Context Parallelism (CP)

(Usually for Transformers)

<u>Idea</u>: Transformers operate on L-length sequences.
Use multiple GPUs to process a single long sequence

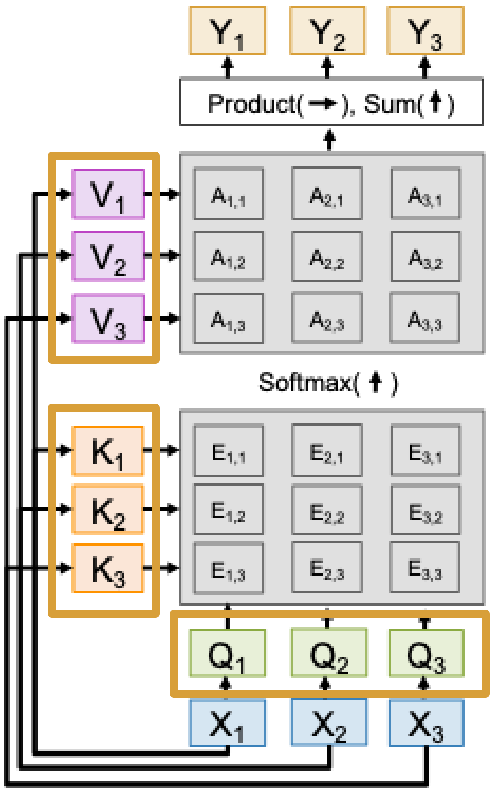


Context Parallelism (CP)

(Usually for Transformers)

<u>Idea</u>: Transformers operate on L-length sequences. Use multiple GPUs to process a single long sequence

QKV Projection: Same as MLP, parallelize over the sequence and sync gradients as in DP



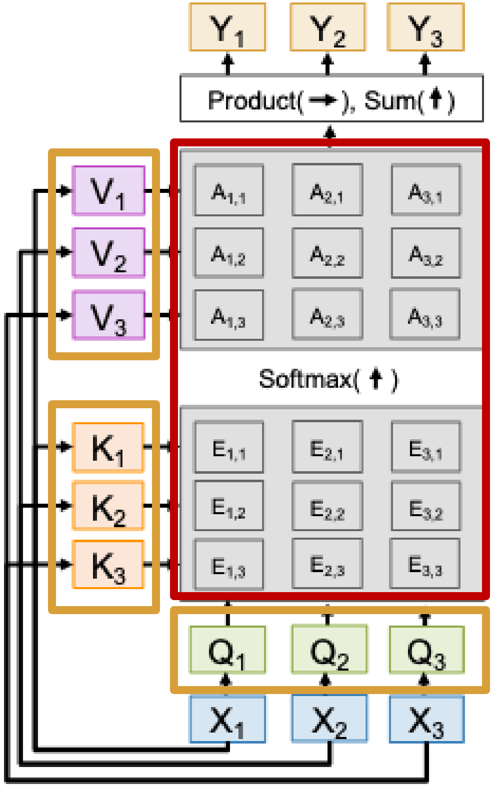
Context Parallelism (CP)

(Usually for Transformers)

Idea: Transformers operate on S-length sequences. Use multiple GPUs to process a single long sequence

QKV Projection: Same as MLP, parallelize over the sequence and sync gradients as in DP

Attention operator: Hardest to parallelize



Context Parallelism (CP)

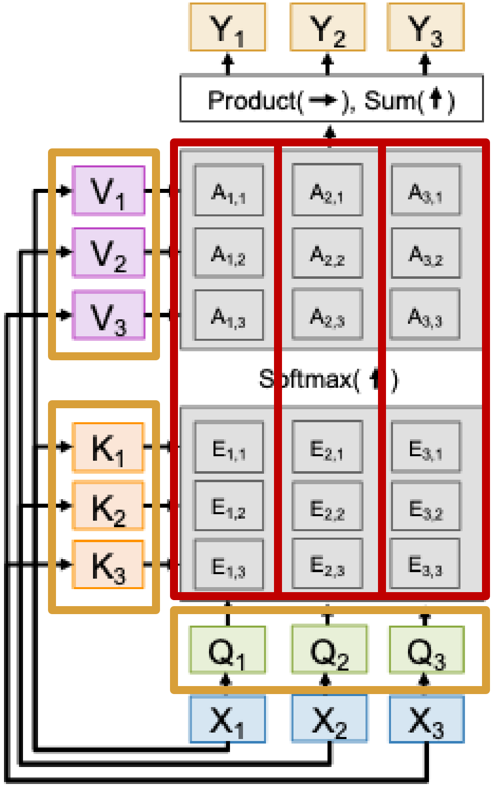
(Usually for Transformers)

Idea: Transformers operate on S-length sequences. Use multiple GPUs to process a single long sequence

QKV Projection: Same as MLP, parallelize over the sequence and sync gradients as in DP

Attention operator: Hardest to parallelize

(Option 1) Ring Attention: Divide into blocks and distribute over GPUs. Inner loop over keys/values, outer loop over queries. Complex to implement but can scale to very long sequences.



Liu et al, "Ring Attention with Blockwise Transformers for Near-Infinite Context", arXiv 2023

Context Parallelism (CP)

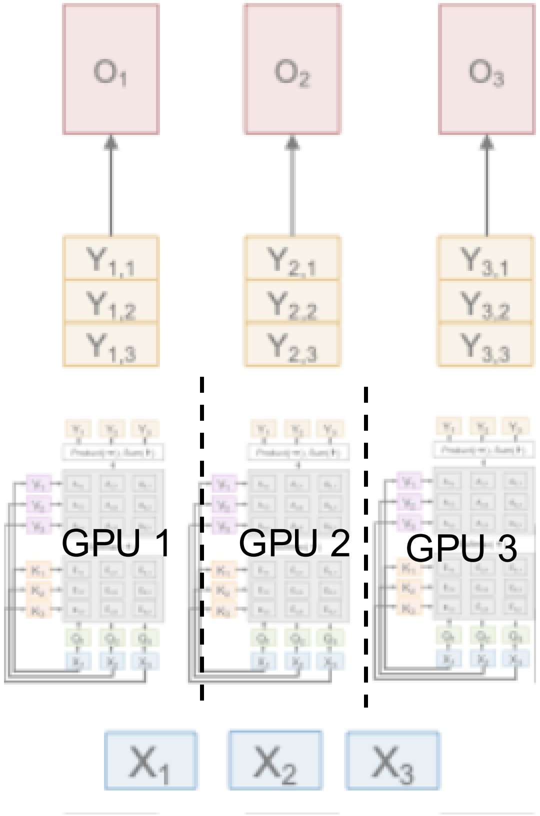
(Usually for Transformers)

Idea: Transformers operate on S-length sequences.
Use multiple GPUs to process a single long sequence

QKV Projection: Same as MLP, parallelize over the sequence and sync gradients as in DP

Attention operator: Hardest to parallelize

(Option 2) Ulysses: Don't try to distribute attention matrix, instead parallelize over heads in multihead attention. Simpler, but max parallelism = number of heads



Jacobs et al, "DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models", arXiv 2023

Context Parallelism (CP)

(Usually for Transformers)

<u>Idea</u>: Transformers operate on S-length sequences.
Use multiple GPUs to process a single long sequence

Often used for long-sequence finetuning.

<u>Example</u>: Llama3-405B training:
- Stage 1: S=8192, no context-parallelism
- Stage 2: S=131,072, 16-way context-parallelism (8192 per GPU)

Llama Team, "The Llama3 Herd of Models", arXiv 2024

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

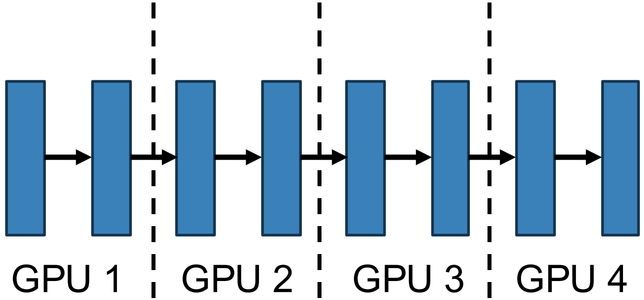
Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

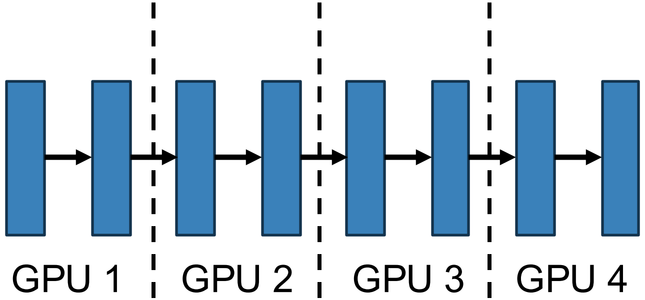
Pipeline Parallelism (PP)

Idea: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.



Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

Pipeline Parallelism (PP)



Idea: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.

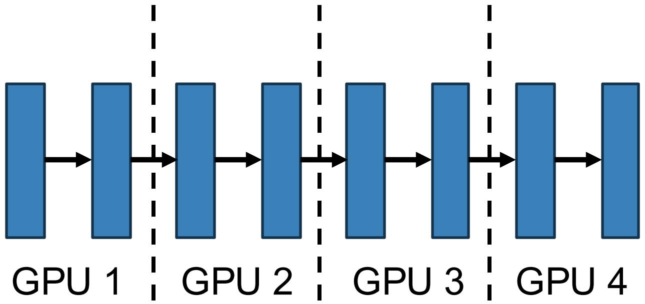
Problem: Sequential dependencies; GPUs are mostly sitting idle.

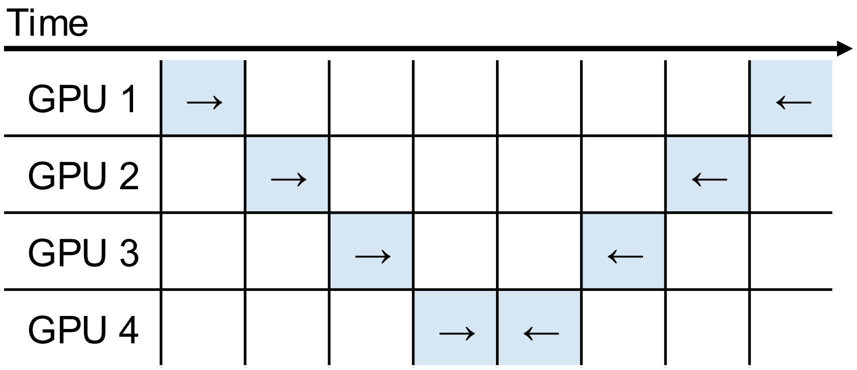
Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

Pipeline Parallelism (PP)

<u>Idea</u>: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.

<u>Problem</u>: Sequential dependencies; GPUs are mostly sitting idle.
Max MFU with N-way PP is 1/N



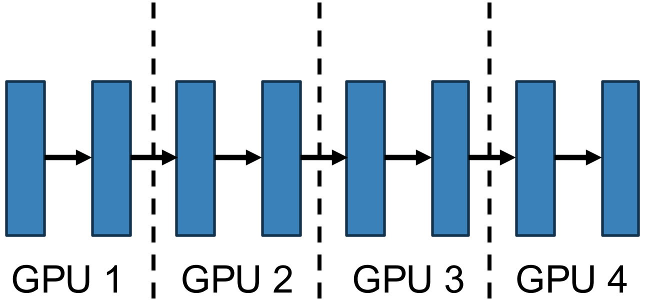


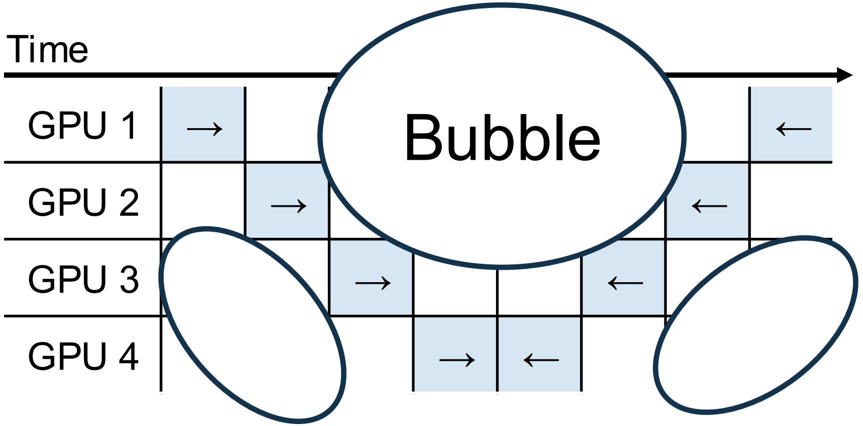
Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

Pipeline Parallelism (PP)

Idea: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.

Problem: Sequential dependencies; GPUs are mostly sitting idle. Max MFU with N-way PP is 1/N





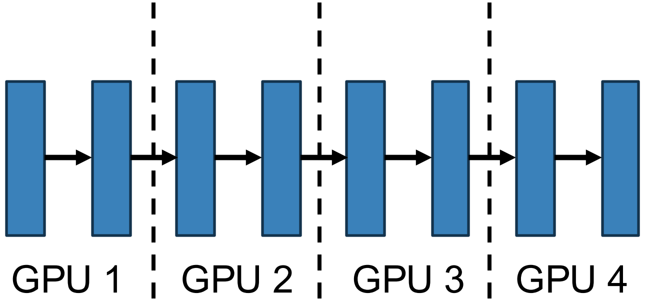
Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

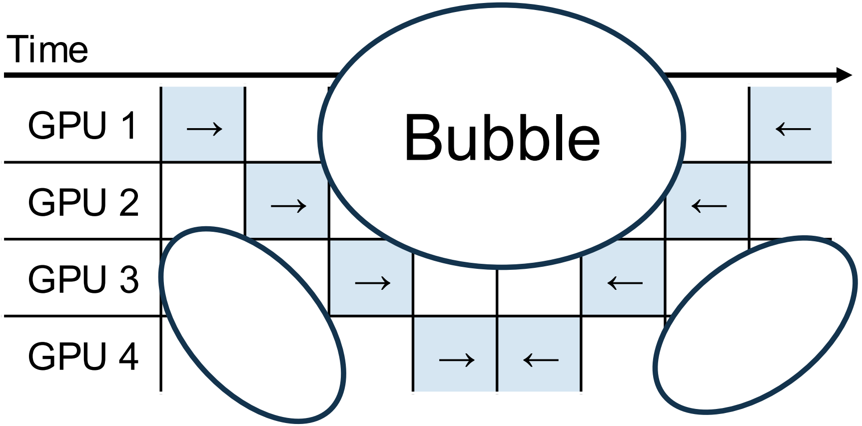
Pipeline Parallelism (PP)

Idea: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.

Problem: Sequential dependencies; GPUs are mostly sitting idle.
Max MFU with N-way PP is 1/N

Solution: Run multiple **microbatches** at the same time, pipeline them through the GPUs

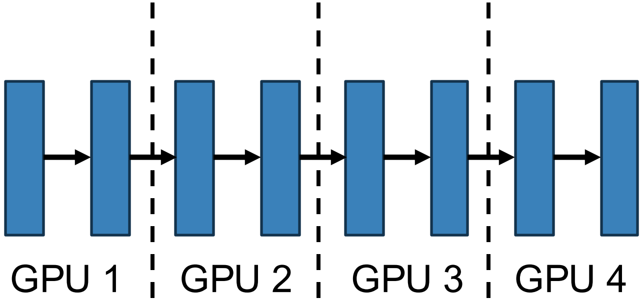




Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

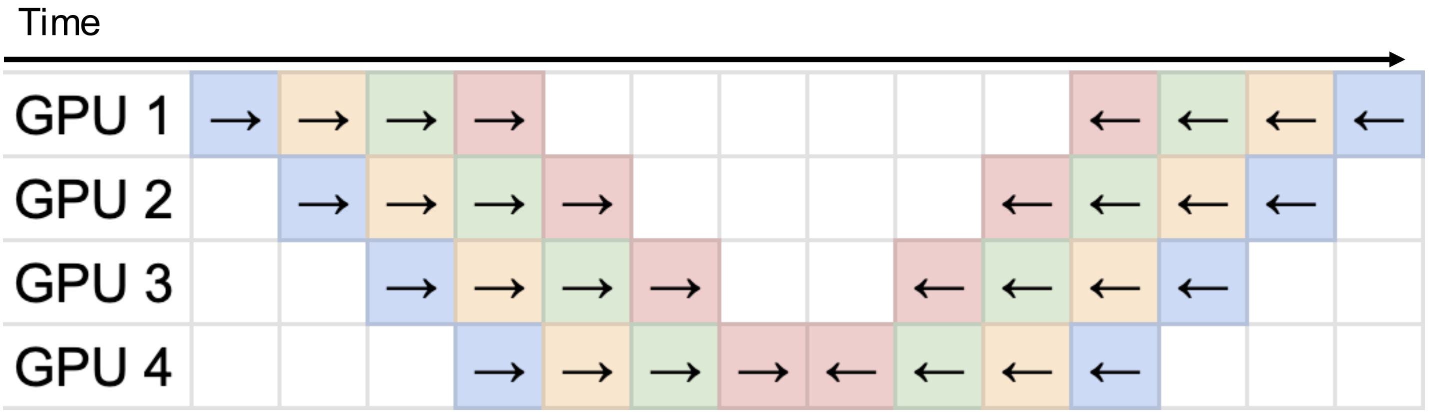
Pipeline Parallelism (PP) - Microbatches

<u>Idea</u>: Split the layers of the model across GPUs. Copy activations between layers at GPU boundaries.



<u>Example</u>:
4-way PP with 4 microbatches.

Max MFU increases from 1/4 = 25% to 16/28 ≈ 57.1%



Huang et al, "Gpipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism", arXiv 2018

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

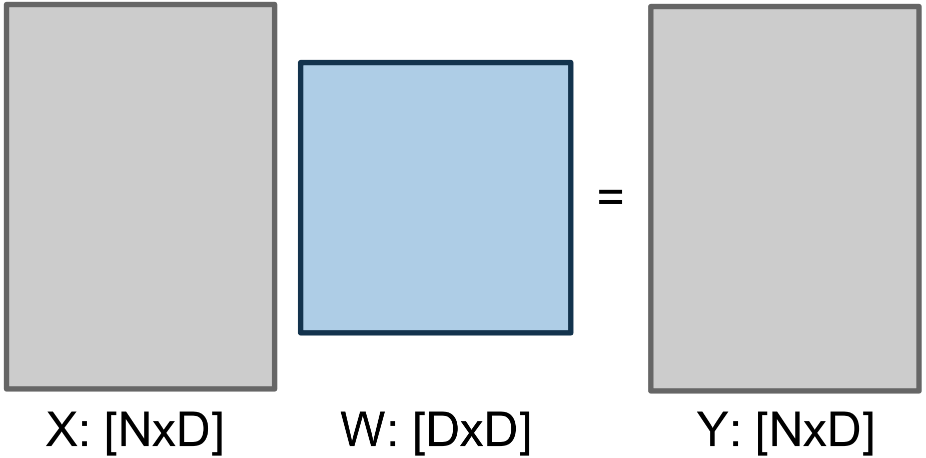
Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

Tensor Parallelism (TP)

Idea: Split the weights of each linear layer across GPUs, use block matrix multiply

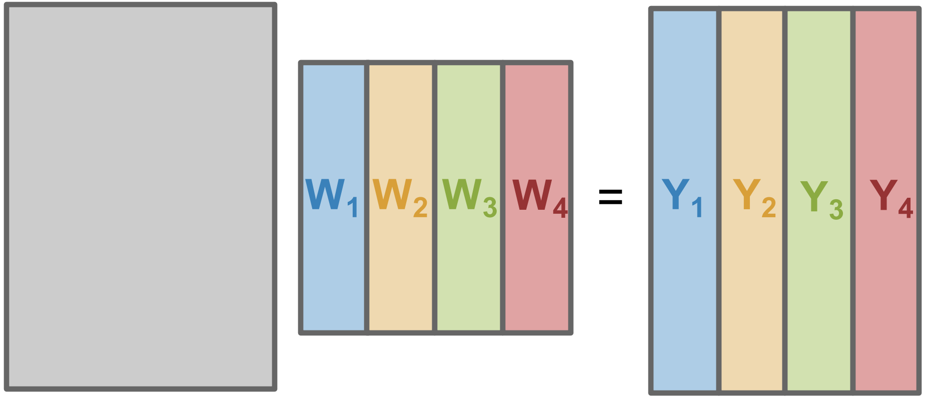
XW = Y (1 GPU)



Tensor Parallelism (TP)

Idea: Split the weights of each linear layer across GPUs, use block matrix multiply

XW = Y (4-way TP)



	X: [NxD]	W: [DxD]	Y: [NxD]
Block shapes:	[1x1]	[1x4]	[1x4]

GPU i computes

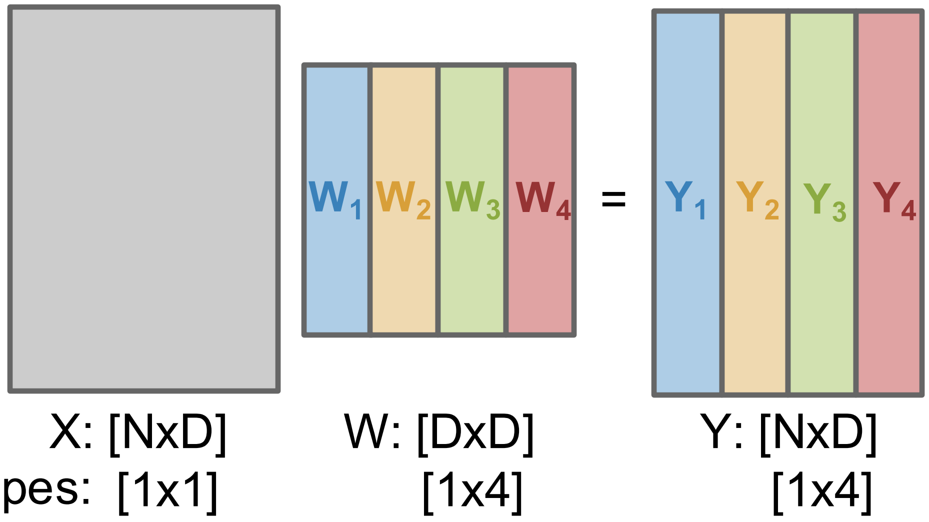
$XW_i = Y_i$

Tensor Parallelism (TP)

Idea: Split the weights of each linear layer across GPUs, use block matrix multiply

Problem: Need to gather parts of Y after forward, can't overlap with communication

XW = Y (4-way TP)



GPU i computes

$XW_i = Y_i$

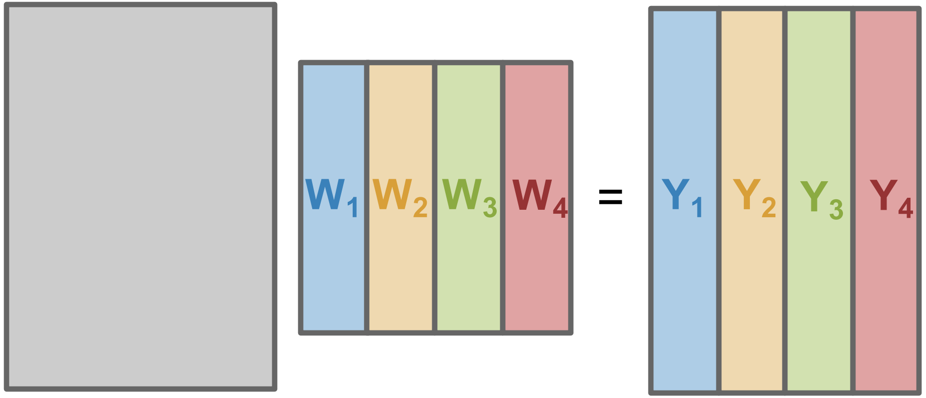
Tensor Parallelism (TP)

Idea: Split the weights of each linear layer across GPUs, use block matrix multiply

Problem: Need to gather parts of Y after forward, can't overlap with communication

Trick: With 2 consecutive TP layers, shard first over row and second over column to avoid communication

$XW = Y$ (4-way TP)



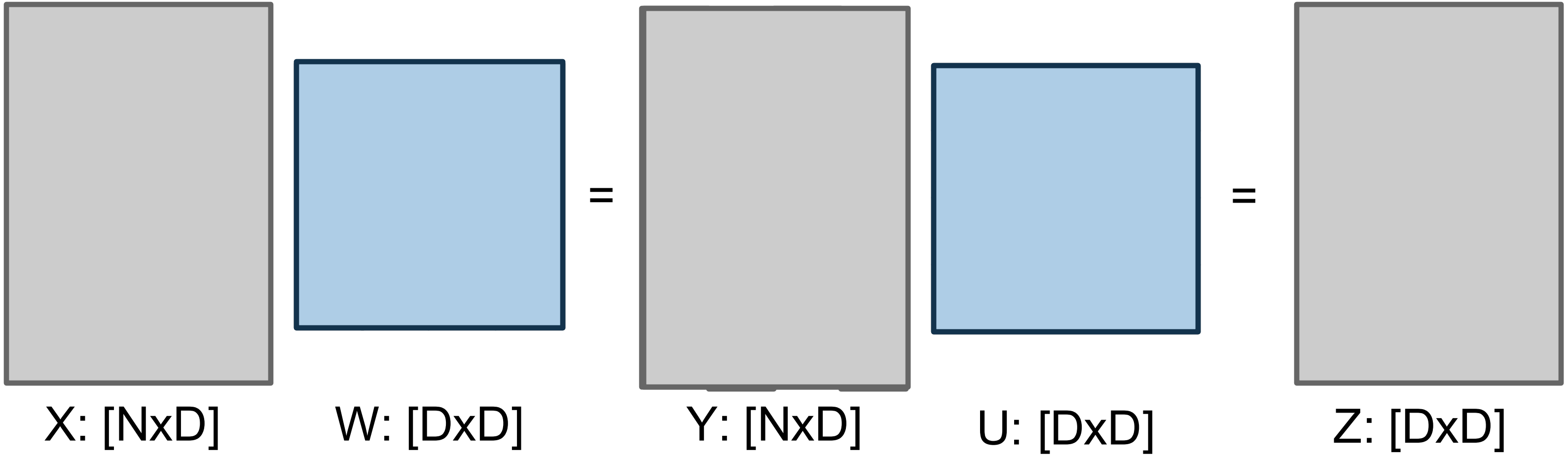
	X: [NxD]	W: [DxD]	Y: [NxD]
Block shapes:	[1x1]	[1x4]	[1x4]

GPU i computes

$XW_i = Y_i$

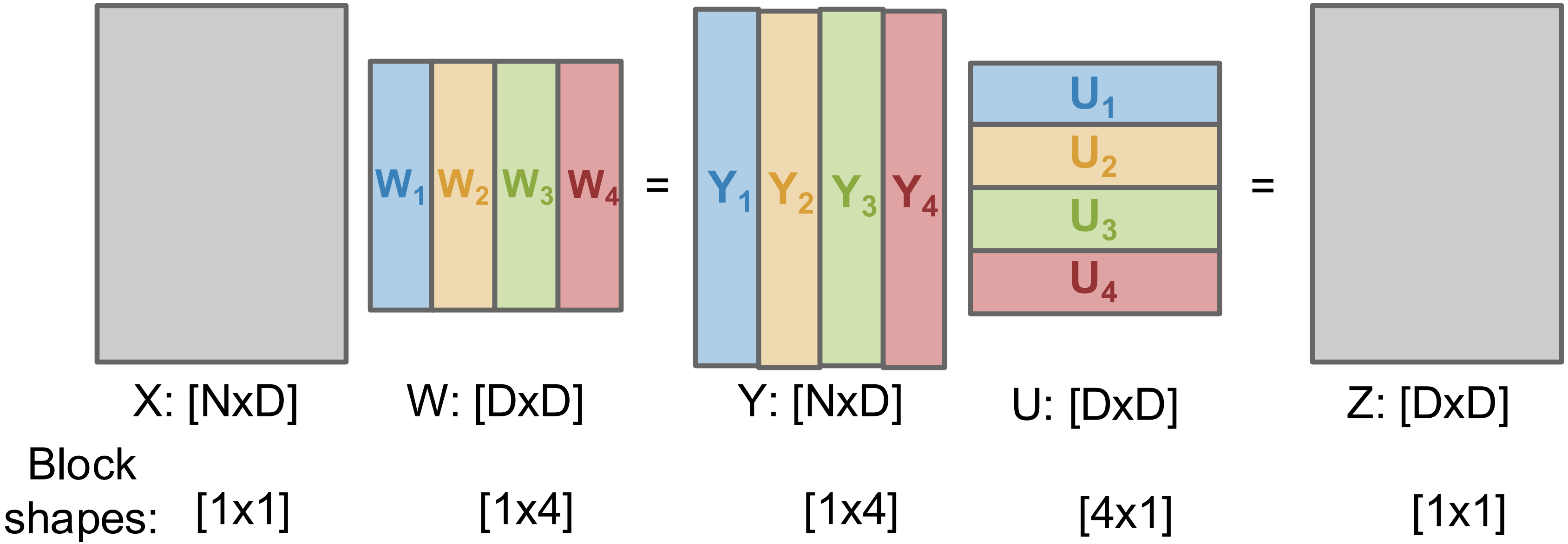
Tensor Parallelism (TP) – Two Layers

(4-way TP)
XW = Y (layer 1)
YU = Z (layer 2)



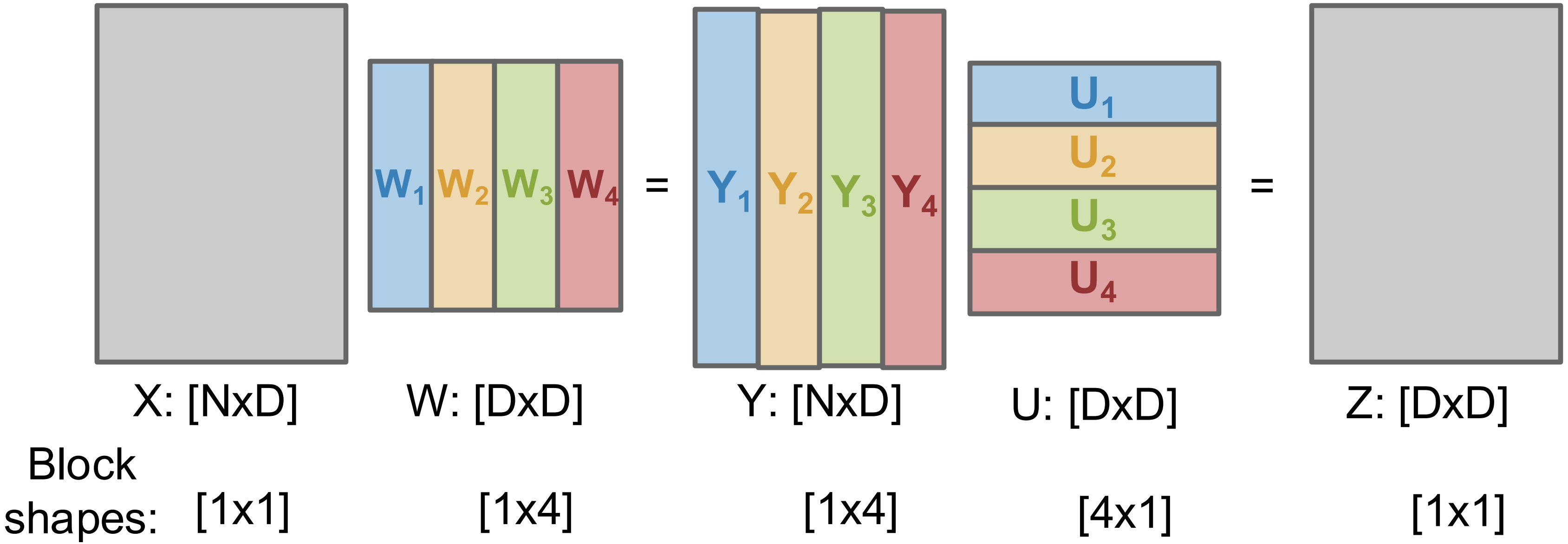
Tensor Parallelism (TP) – Two Layers

(4-way TP)
XW = Y (layer 1)
YU = Z (layer 2)



Tensor Parallelism (TP) – Two Layers

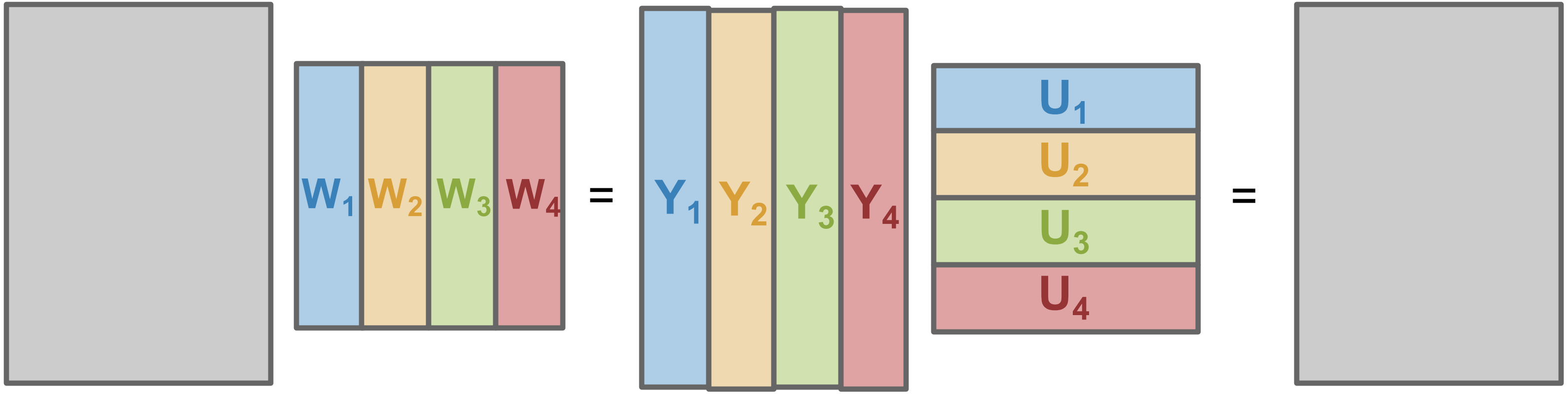
(4-way TP)
$XW = Y$ (layer 1)
$YU = Z$ (layer 2)



$$Z = Y_1U_1 + Y_2U_2 + Y_3U_3 + Y_4U_4$$

Tensor Parallelism (TP) – Two Layers

(4-way TP)
$XW = Y$ (layer 1)
$YU = Z$ (layer 2)



X: [NxD] W: [DxD] Y: [NxD] U: [DxD] Z: [DxD]

Block shapes: [1x1] [1x4] [1x4] [4x1] [1x1]

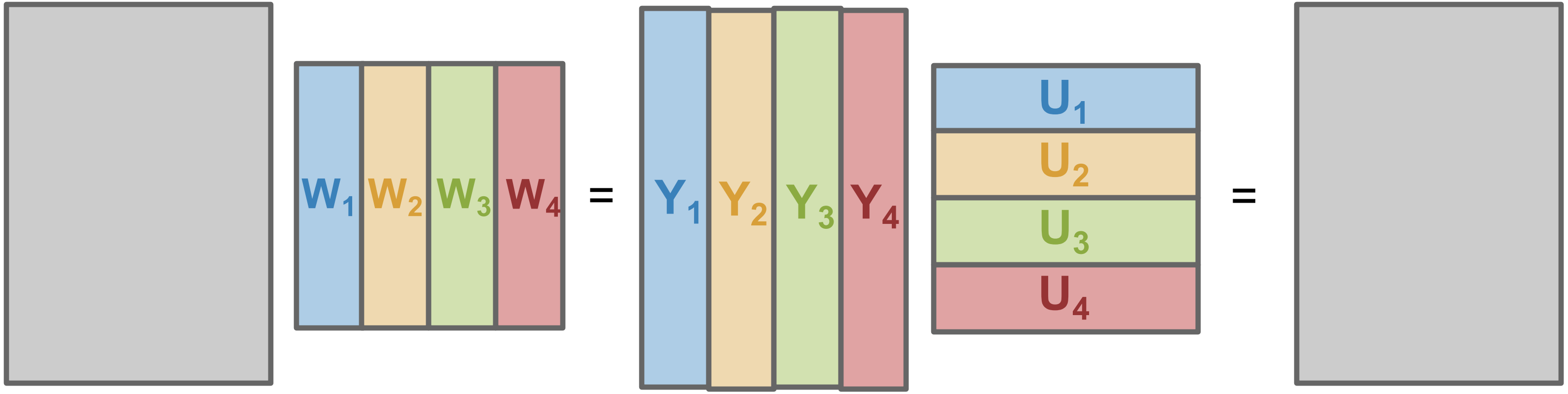
GPU i computes
$XW_i = Y_i$

GPU i computes
$Y_iU_i = Z_i$

$Z = Y_1U_1 + Y_2U_2 + Y_3U_3 + Y_4U_4$

Tensor Parallelism (TP) – Two Layers

(4-way TP)
$XW = Y$ (layer 1)
$YU = Z$ (layer 2)



	X: [NxD]	W: [DxD]	Y: [NxD]	U: [DxD]	Z: [DxD]
Block shapes:	[1x1]	[1x4]	[1x4]	[4x1]	[1x1]

No need for communication after XW=Y! Each GPU computes one term of Z, then broadcasts to all other GPUs

$$Z = Y_1U_1 + Y_2U_2 + Y_3U_3 + Y_4U_4$$

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

How to train on lots of GPUs

A model with L layers operates on tensors of shape (Batch, Sequence, Dim)

Data Parallelism (DP)
Split on Batch dimension

Context Parallelism (CP)
Split on Sequence dimension

Q: Which to use for largest models?
A: All of them!

Pipeline Parallelism (PP)
Split on L dimension

Tensor Parallelism (TP)
Split on Dim dimension

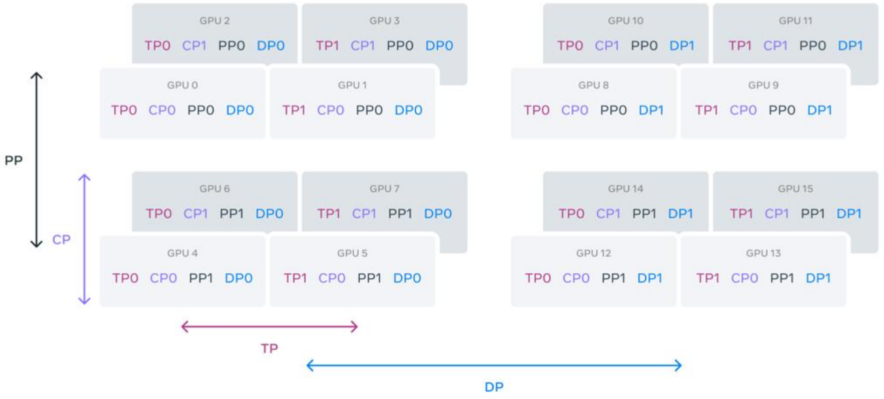
ND Parallelism

Use TP, CP, PP, and DP all at the same time!

Arrange GPUs in a 4D grid

GPUs index in the grid gives its rank along each parallelism dimension

Optimize setup to maximize MFU



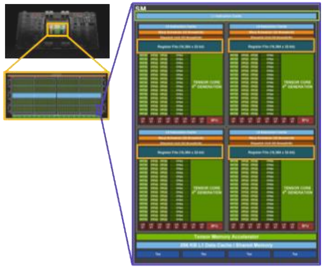
Example: LLama3-405B

GPUs	TP	CP	PP	DP	Seq. Len.	Batch size/DP	Tokens/Batch	TFLOPs/GPU	BF16 MFU
8,192	8	1	16	64	8,192	32	16M	430	43%
16,384	8	1	16	128	8,192	16	16M	400	41%
16,384	8	16	16	8	131,072	16	16M	380	38%

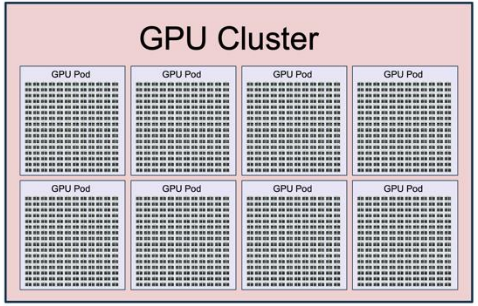
Llama Team, "The Llama3 Herd of Models", arXiv 2024

Summary: Large-Scale Distributed Training

A GPU is a parallel processor with hundreds of cores



A GPU cluster has O(10K) GPUs



Split up the computation along different axes
Consider a model with many Layers, operating on tensors of shape (Batch, Seq, Dim)

- **Data Parallel (DP)**: Split on Batch
- **Context Parallel (CP)**: Split on Seq
- **Pipeline Parallel (PP)**: Split on Layers
- **Tensor Parallel (TP)**: Split on Dim

Activation Checkpointing saves memory by recomputing during backward

Tune parallelism recipe to maximize **Model Flops Utilization (MFU)**

Next Time:
Self-Supervised Learning