Distributed Training Basics

September 4th, 2025

Jae-Won Chung

Jae-Won Chung

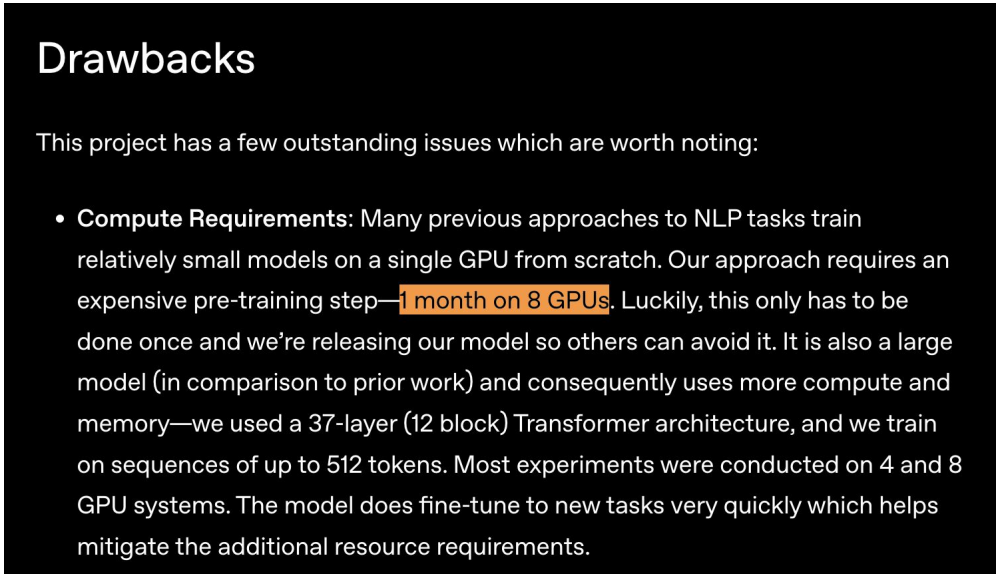
- Bio
  - 5th year PhD student working with Mosharaf
  - https://jaewonchung.me/about
- Background
  - 2017 – 2019: Machine learning, Computer vision, Meta-learning, Few-shot learning
  - 2019 – now: Systems for machine learning, Power & energy as first-class systems resources
- Three lectures
  - 08/29 (Thu) | Introduction to GenAI and Systems for GenAI
  - 09/04 (Thu) | Distributed training basics
  - 10/02 (Thu) | Inference basics

(Unofficial) Textbooks

- How to Scale Your Model (Google DeepMind)
  - Theoretical analysis of computations that are important to LLMs
  - Arithmetic intensity, compute- and memory-bound, roofline, back-of-the-envelope estimations
- The Ultra-Scale Playbook (Hugging Face)
  - Training LLMs on GPU Clusters
  - Various types of training parallelisms, implications on compute, memory, and communication
- Use as references
  - Each thing will take at least a week of full-time reading (it's worth it, though)
  - It's a fast-evolving field; the only way to be relevant is to continuously read

GPT-1 117M (OpenAI, 2018)

- 1 month on 8 GPUs = 8 GPUs x 30 x 24 hours = 5,760 GPU-hours



Drawbacks

This project has a few outstanding issues which are worth noting:

- **Compute Requirements**: Many previous approaches to NLP tasks train relatively small models on a single GPU from scratch. Our approach requires an expensive pre-training step—1 month on 8 GPUs. Luckily, this only has to be done once and we're releasing our model so others can avoid it. It is also a large model (in comparison to prior work) and consequently uses more compute and memory—we used a 37-layer (12 block) Transformer architecture, and we train on sequences of up to 512 tokens. Most experiments were conducted on 4 and 8 GPU systems. The model does fine-tune to new tasks very quickly which helps mitigate the additional resource requirements.

GPT-OSS 120B (OpenAI, Aug 2025)

- 2.1 million GPU-hours = 30 years if it were on 8 GPUs
  - So they probably used a bit more GPUs than 8

2.4 Pretraining

Data: We train the models on a text-only dataset with trillions of tokens, with a focus on STEM, coding, and general knowledge. To improve the safety of the model, we filtered the data for harmful content in pre-training, especially around hazardous biosecurity knowledge, by reusing the CBRN pre-training filters from GPT-4o [18]. Our model has a knowledge cutoff of June 2024.

Training: The gpt-oss models trained on NVIDIA H100 GPUs using the PyTorch framework [19] with expert-optimized Triton [20] kernels[2]. The training run for gpt-oss-120b required 2.1 million H100-hours to complete, with gpt-oss-20b needing almost 10x fewer. Both models leverage the Flash Attention [21] algorithms to reduce the memory requirements and accelerate training.

gpt-oss-120b & gpt-oss-20b Model Card

Llama 3 405B (Meta AI, Jul 2024)

- 30.84 million GPU hours on 16,384 H100 GPUs
  - This translates to 78 days – nearly three months

Compute. Llama 3 405B is trained on up to 16K H100 GPUs, each running at 700W TDP with 80GB HBM3, using Meta's Grand Teton AI server platform (Matt Bowman, 2022). Each server is equipped with eight GPUs and two CPUs. Within a server, the eight GPUs are connected via NVLink. Training jobs are scheduled using MAST (Choudhury et al., 2024), Meta's global-scale training scheduler.

	Training Time (GPU hours)	Training Power Consumption (W)
Llama 3.1 8B	1.46M	700
Llama 3.1 70B	7.0M	700
Llama 3.1 405B	30.84M	700
Total	39.3M	

The Llama 3 Herd of Models (left), Llama 3.1 405B Instruct model REAMDE on Hugging Face Hub (right)

Today

Bottom-up, then top-down

- A practical overview of the training stack ***(bottom-up)***
- Training parallelism: Data, tensor, and pipeline parallelism
- Case study: How tensor parallelism is executed by the stack ***(top-down)***

Essential Layers – Computation

- Programming the GPU
  - Low-level: CUDA (C++ extension)

```cpp
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void vec_add(const int *a, const int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // global thread index
    if (i < n) c[i] = a[i] + b[i];                  // each thread computes one element
}
```

- Body of the **kernel** is for one thread
- Every thread runs the exact same code
- Threads' `i` values (thread index) are different

Essential Layers – Computation

- Programming the GPU
  - Low-level: CUDA (C++ extension)

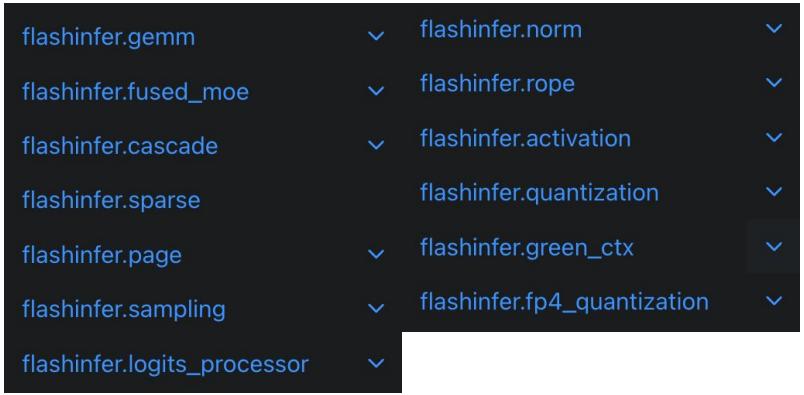
```cpp
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void vec_add(const int *a, const int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // global thread
    if (i < n) c[i] = a[i] + b[i];                  // each thread co
}
```

- Body of the **kernel** is for one thread
- Every thread runs the exact same code
- Threads' `i` values (thread index) are different

```cpp
int main() {
    const int N = 8;
    int h_a[N] = {1,2,3,4,5,6,7,8};
    int h_b[N] = {8,7,6,5,4,3,2,1};
    int h_c[N], *d_a, *d_b, *d_c;

    cudaMalloc(&d_a, N*sizeof(int));
    cudaMalloc(&d_b, N*sizeof(int));
    cudaMalloc(&d_c, N*sizeof(int));
    cudaMemcpy(d_a, h_a, N*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, N*sizeof(int), cudaMemcpyHostToDevice);

    vec_add<<<1, N>>>(d_a, d_b, d_c, N);   // 1 block, N threads
    cudaMemcpy(h_c, d_c, N*sizeof(int), cudaMemcpyDeviceToHost);

    for (int i=0; i<N; i++) printf("%d ", h_c[i]);
    printf("\n");

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

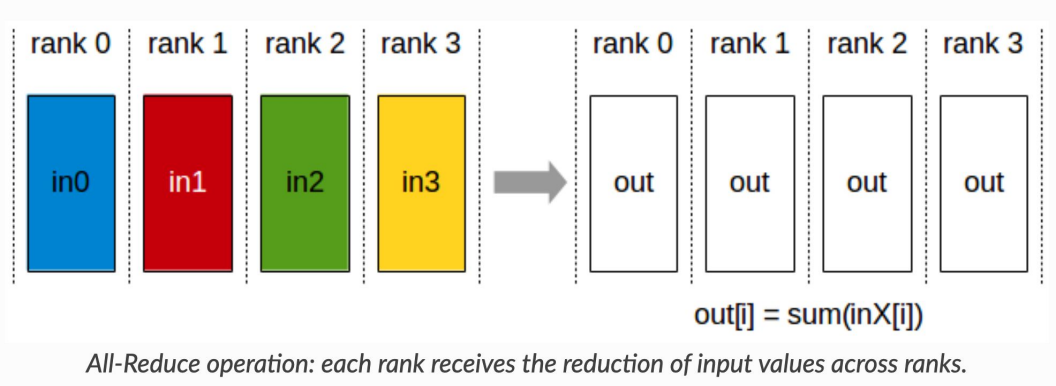
Essential Layers – Computation

- Optimized kernels are hard to write
- Kernel libraries
  - cuBLAS (linear algebra), cuDNN (deep learning), cuRAND (random number generation)
  - FlashAttention (all kinds of attention)
  - FlashInfer (all sorts of kernels relevant to LLM serving)



Essential Layers – Communication

- NCCL
  - Point-to-point (**P2P**): Sending data between from one GPU to another
  - **Collective communication**
    - All-Reduce

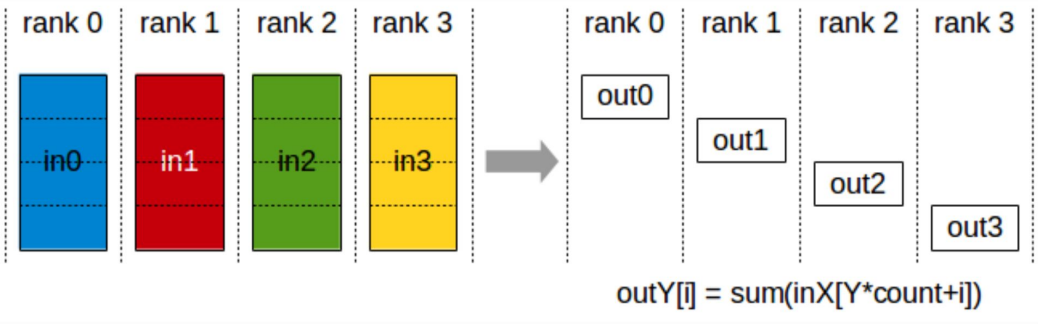


All-Reduce operation: each rank receives the reduction of input values across ranks.

NCCL Collective Operations

Essential Layers – Communication

- NCCL
  - Point-to-point (**P2P**): Sending data between from one GPU to another
  - **Collective communication**
    - All-Reduce, Reduce-Scatter

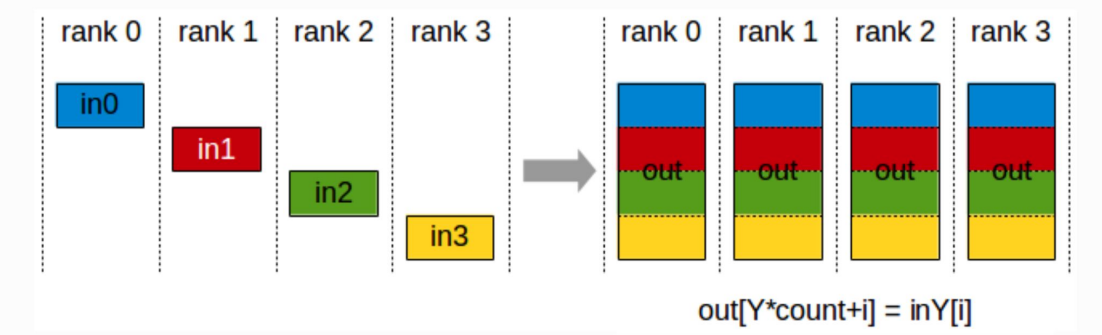


Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.

NCCL Collective Operations

Essential Layers – Communication

- NCCL
  - Point-to-point (**P2P**): Sending data between from one GPU to another
  - **Collective communication**
    - All-Reduce, Reduce-Scatter, All-Gather

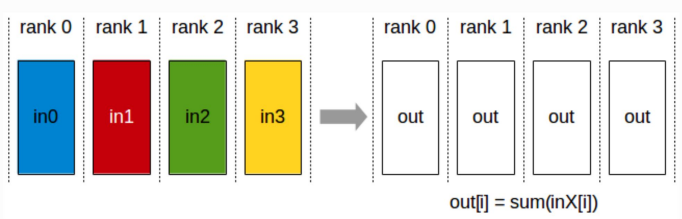


AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

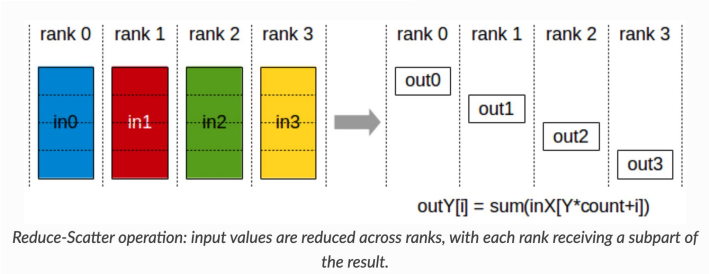
NCCL Collective Operations

Essential Layers – Communication

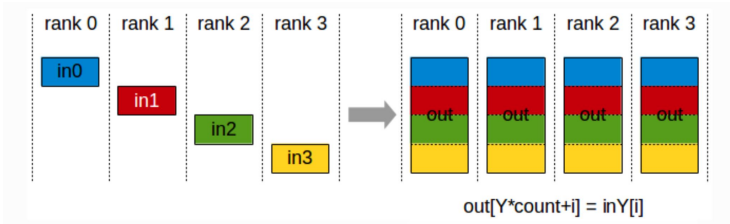
- NCCL
  - Point-to-point (**P2P**): Sending data between from one GPU to another
  - **Collective communication**
    - All-Reduce, Reduce-Scatter, All-Gather



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



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.



AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

NCCL Collective Operations

Essential Layers – Training Framework

- Where everything meets: PyTorch
  - Write tensor computations with a Pythonic API
  - Define neural network modules to train
  - Just write forward computations, and PyTorch Autograd will figure out backward for you

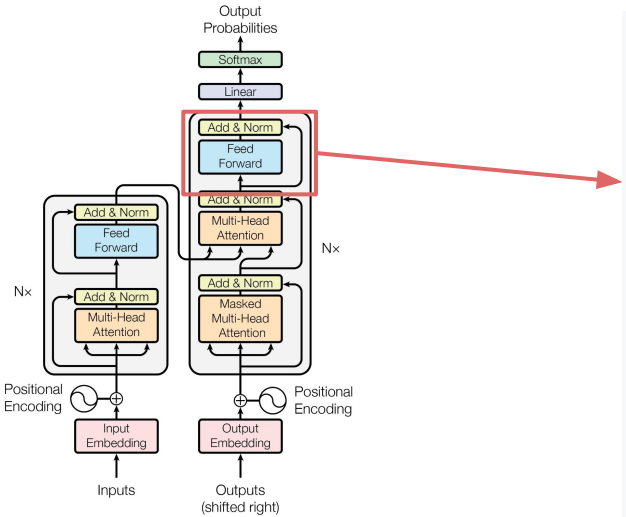


Figure 1: The Transformer - model architecture.

```python
import torch.nn as nn
import torch.nn.functional as F

class TransformerMLP(nn.Module):
    """Transformer MLP block."""

    def __init__(self, d_model: int = 256) -> None:
        """Initialize the Transformer MLP block."""
        super().__init__()
        self.fc1 = nn.Linear(d_model, d_model * 4)
        self.fc2 = nn.Linear(d_model * 4, d_model)

    def forward(self, x):
        """Forward pass of the Transformer MLP block."""
        hidden = self.fc1(x)
        hidden = F.gelu(hidden)
        hidden = self.fc2(hidden)
        return x + hidden
```

Essential Layers – Training Framework

- Where everything meets: PyTorch
  - PyTorch also lets you launch collective communication with tensors

```python
import torch
import torch.distributed

rank = torch.distributed.get_rank()
world_size = torch.distributed.get_world_size()
assert world_size == 4

tensor = torch.ones(4, 4) * rank
torch.distributed.all_reduce(tensor, op=torch.distributed.ReduceOp.SUM)

print(f"Rank {rank} has data {tensor}")
```

Essential Layers – Training Framework

- Distributed training system
  - NVIDIA Megatron-LM and NeMo, Microsoft DeepSpeed, PyTorch/Meta TorchTitan
  - Composes PyTorch-level APIs to actually make training happen across 16k GPUs
  - Input
    - Model definition (`nn.Module`)
    - Dataset (sequence of batches) – training, validation, and test sets
    - Hardware specs and parallelism configuration

Essential Layers – Training Framework

- Distributed training system
  - NVIDIA Megatron-LM and NeMo, Microsoft DeepSpeed, PyTorch/Meta TorchTitan
  - Composes PyTorch-level APIs to actually make training happen across 16k GPUs
  - Input
    - Model definition (`nn.Module`)
    - Dataset (sequence of batches) – training, validation, and test sets
    - Hardware specs and parallelism configuration
  - The reality
    - Everything is intermixed with everything else
    - You need to write your model in a parallelism-aware manner
    - Your parallelism implementation has to be model-aware
    - Even simple optimizations can lead to numerics issues

Parallelism in Large Model Training

- Pipeline parallelism

- You have an H100 GPU (80GB VRAM)
- You have a 70B model ⇒ 140GB in half precision
- Memory capacity is a *hard constraint*; everything must fit within GPU memory

Parallelism in Large Model Training

- Pipeline parallelism

- You have an H100 GPU (80GB VRAM)
- Now each stage weight takes up 35GB

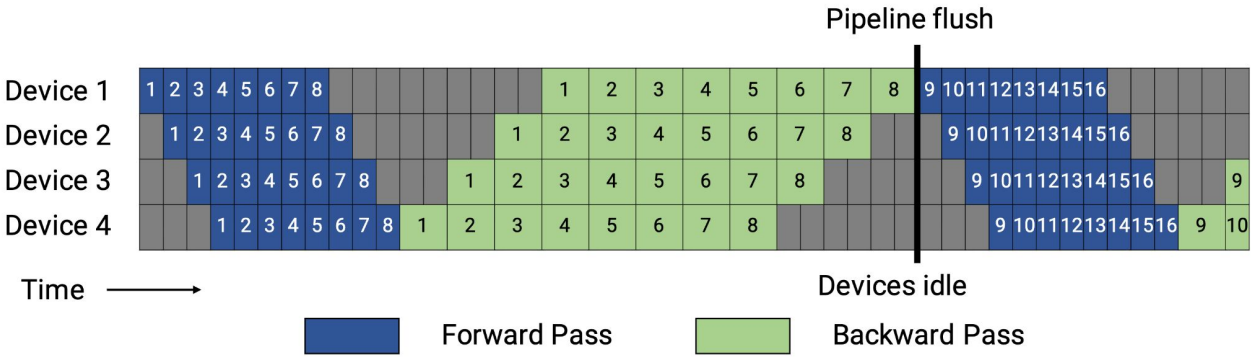
Parallelism in Large Model Training

- Pipeline parallelism

- You have an H100 GPU (80GB VRAM)
- Now each stage weight takes up 35GB

Parallelism in Large Model Training

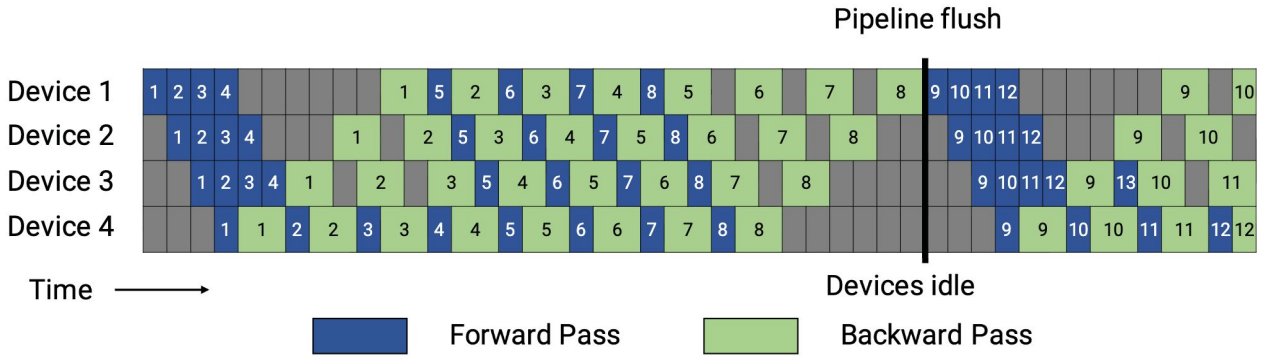
- Pipeline parallelism
  - Four **stages**, eight **microbatches**
  - **Schedule**: GPipe



Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, SC, 2021

Parallelism in Large Model Training

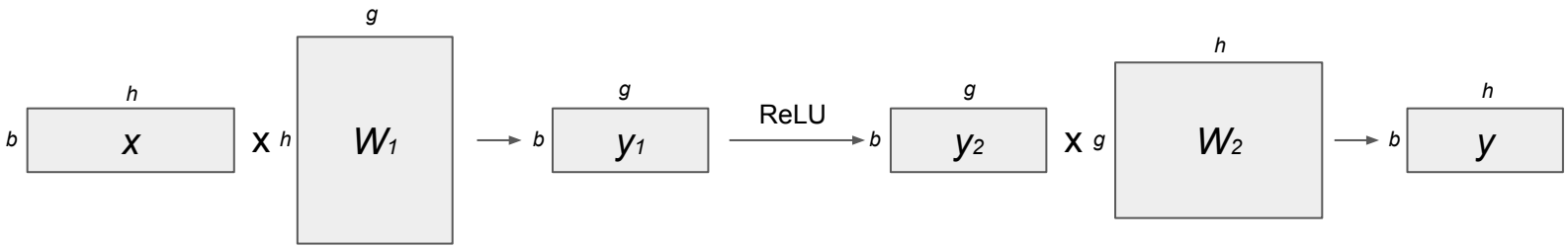
- Pipeline parallelism
  - Four **stages**, eight **microbatches**
  - **Schedule**: GPipe, 1F1B



Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, SC, 2021

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

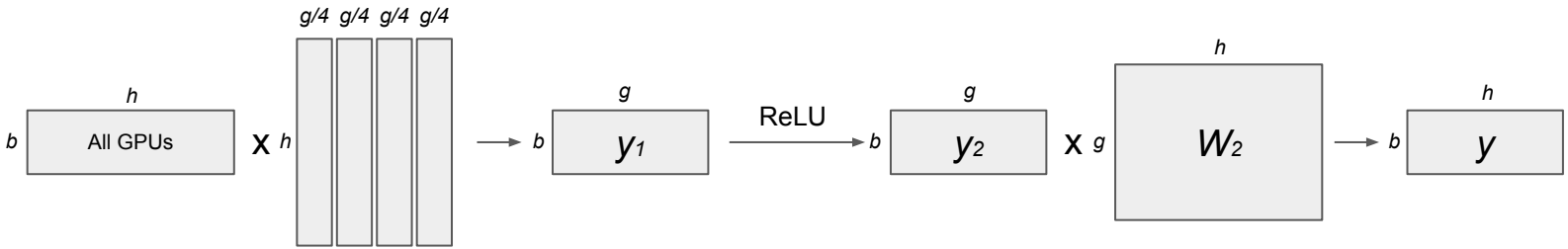


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

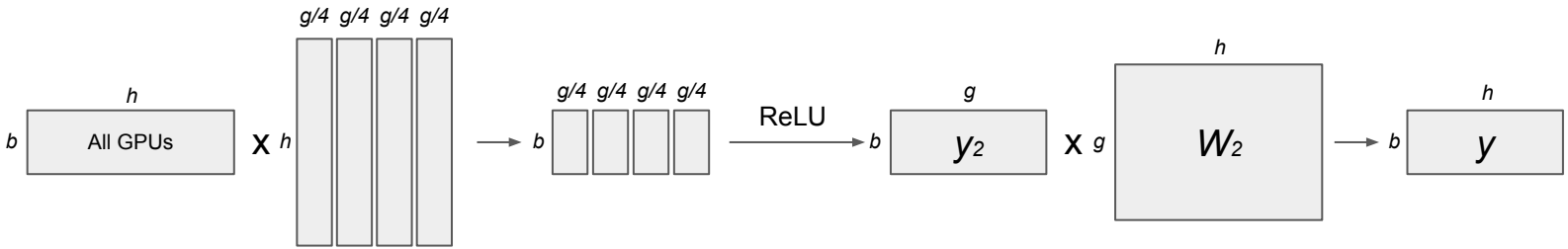


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

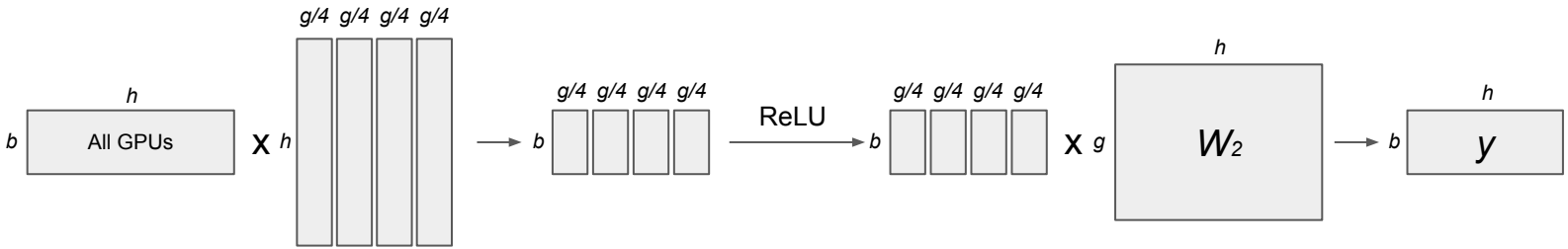


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

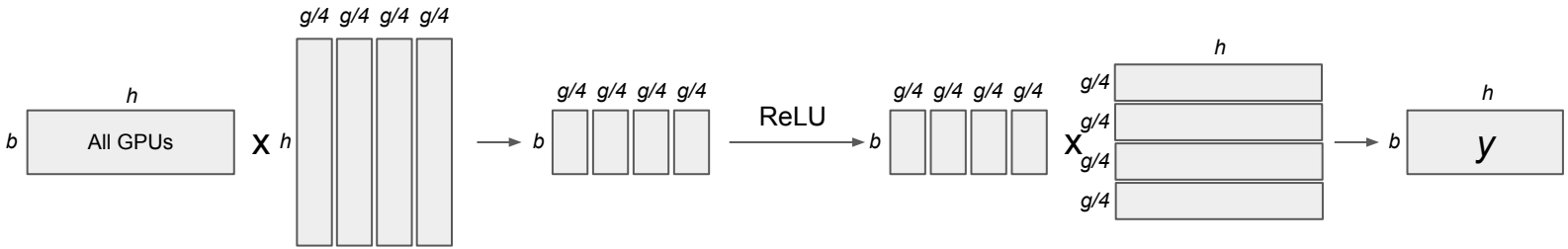


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

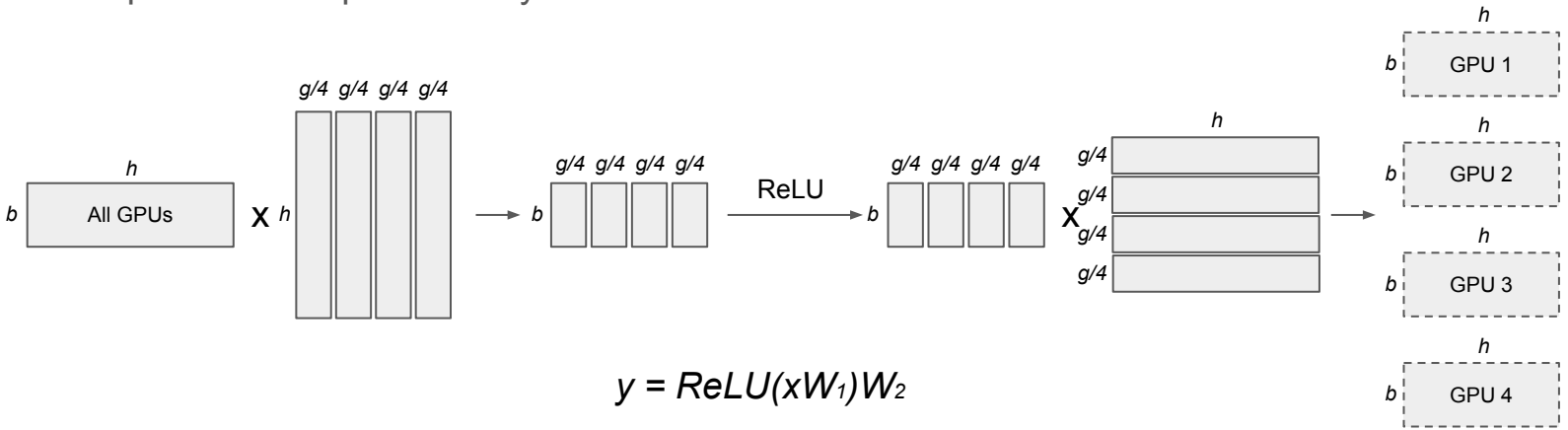


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

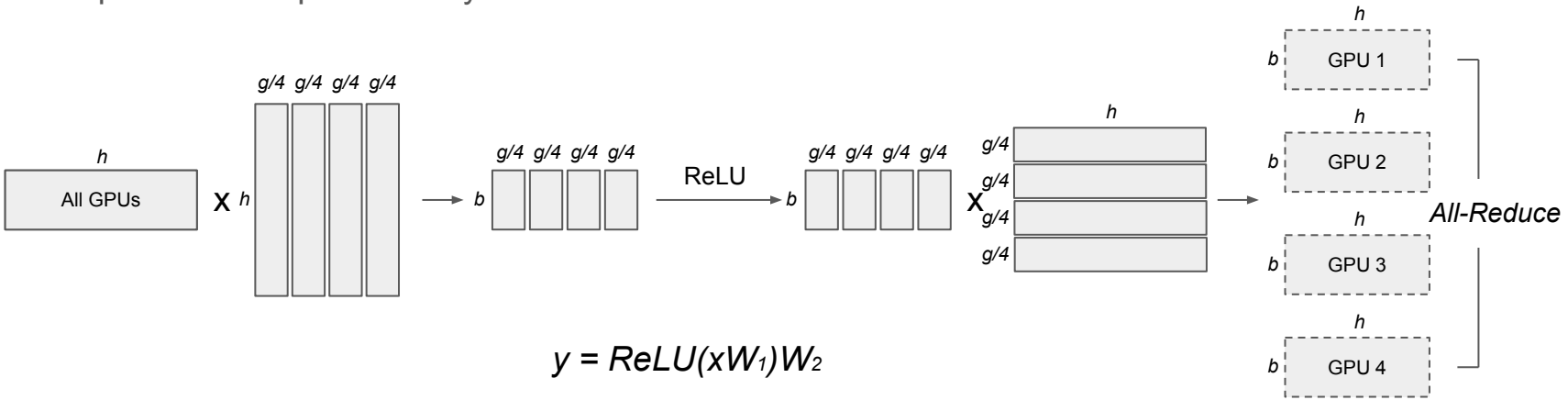


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

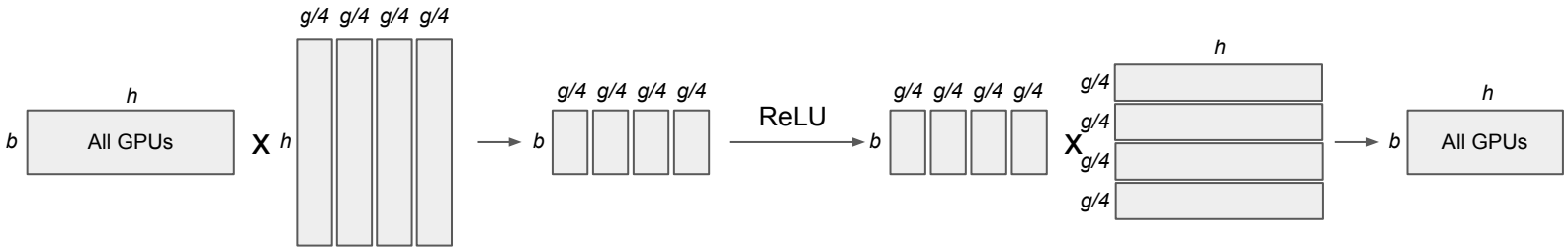


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

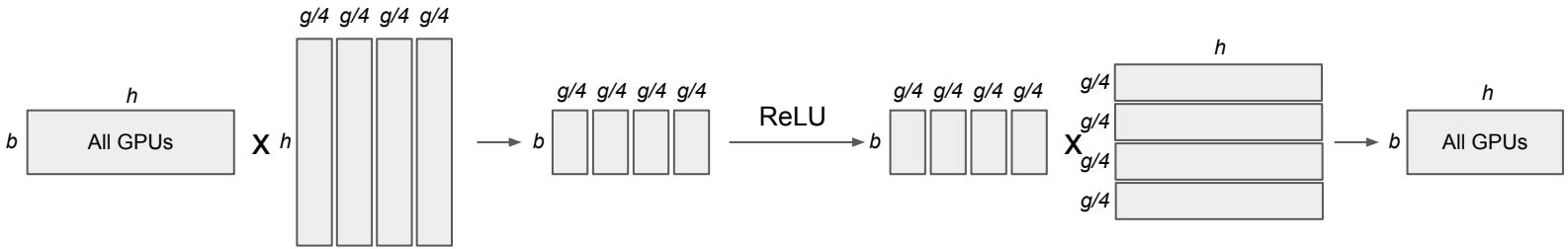


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP



Column Parallel Linear

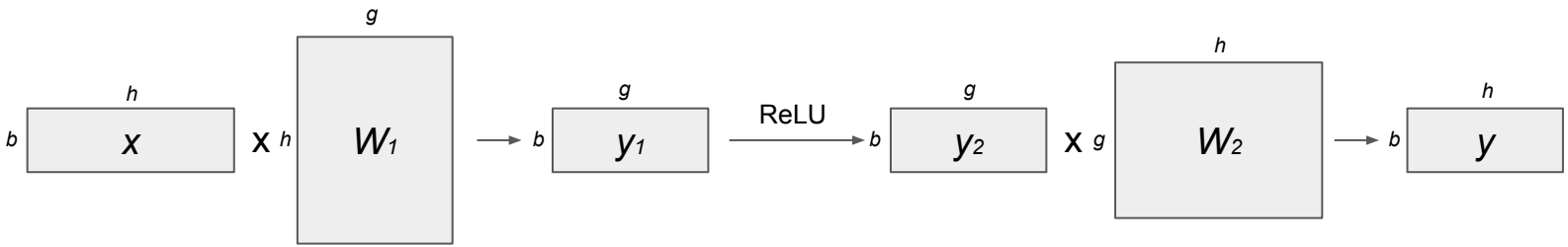
Row Parallel Linear

$y = ReLU(xW_1)W_2$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

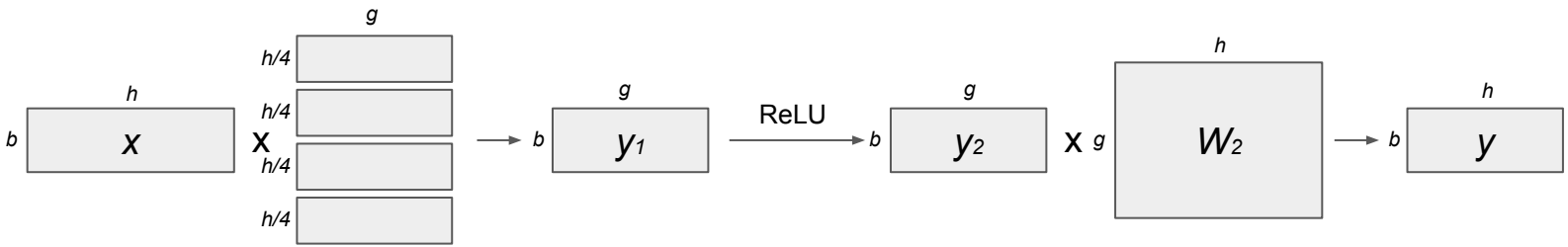


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

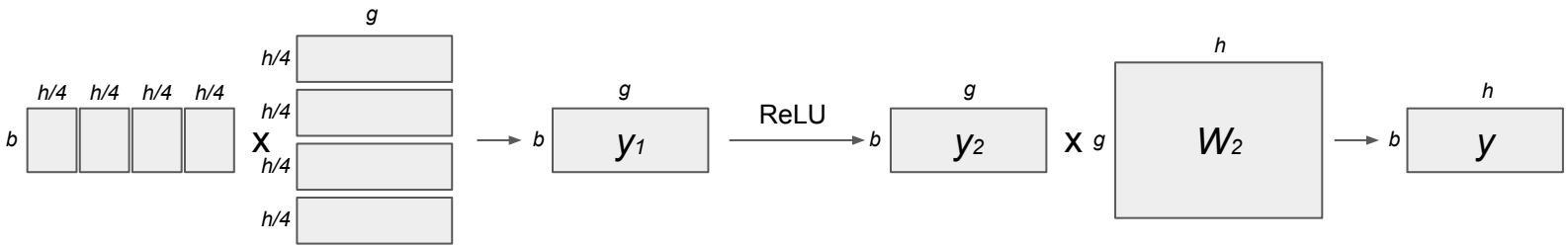


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP

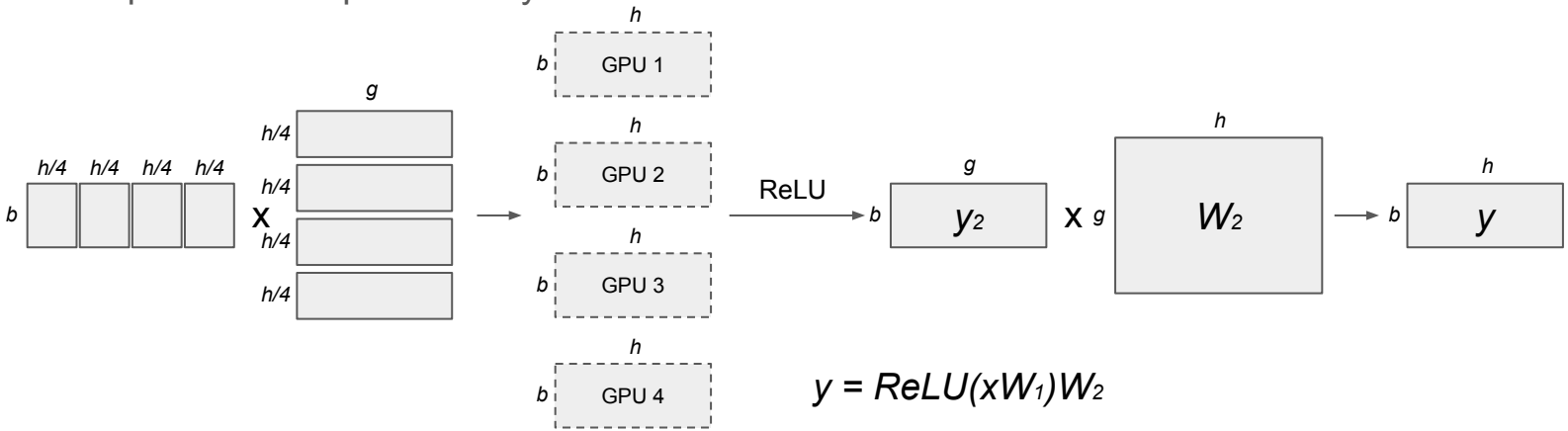


$$y = ReLU(xW_1)W_2$$

Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Tensor parallelism
  - A generic category for whenever a single operation's weights are split into N devices
  - Important example: Two-layer MLP



Now you can't apply ReLU, unless another All-Reduce collective is inserted here

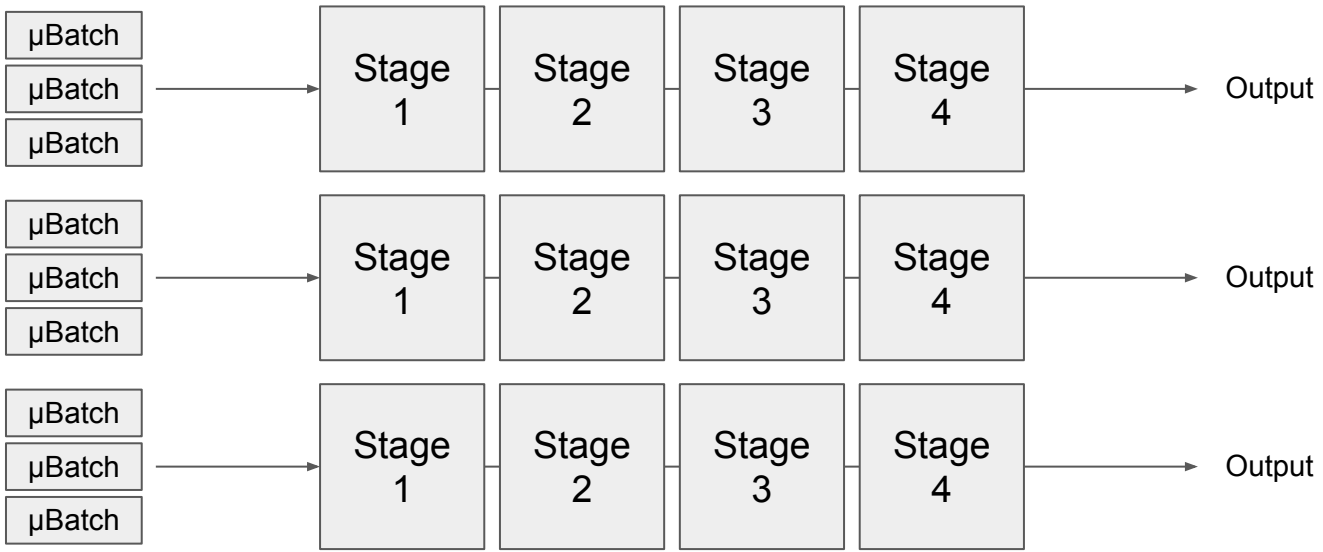
Note: *ReLU(a) = max(a, 0)* – **Elementwise**

Parallelism in Large Model Training

- Data parallelism
  - Replicate whole pipelines and run them in parallel

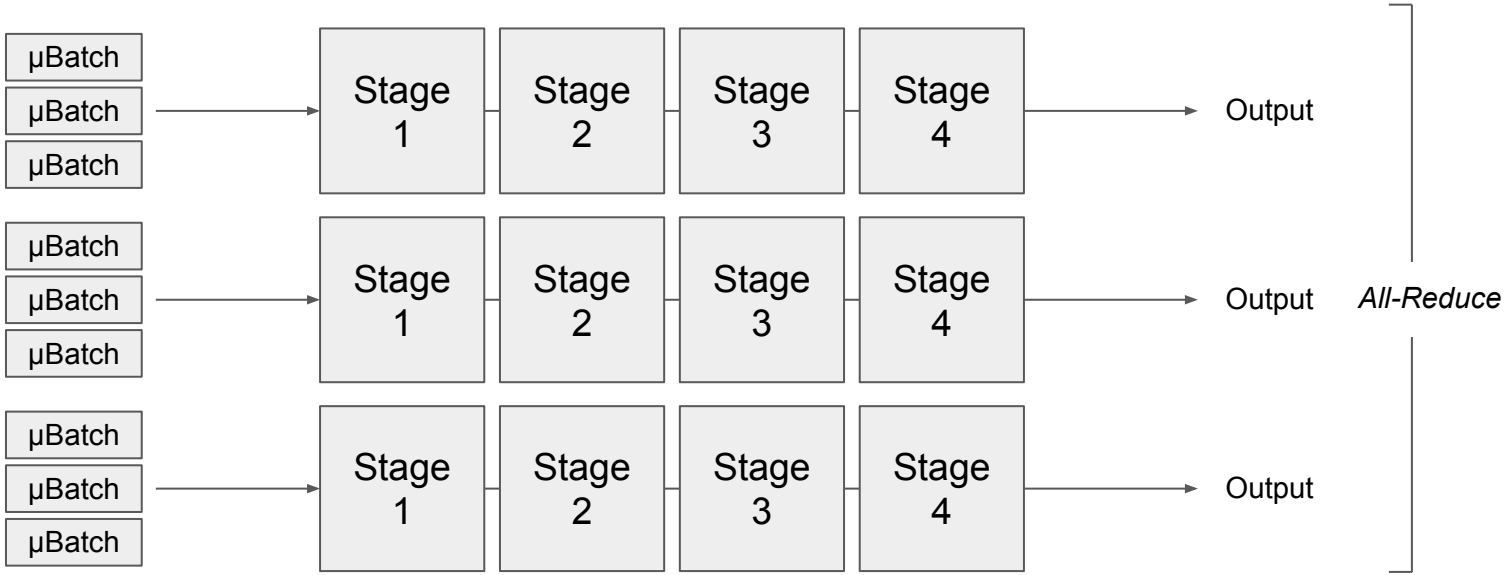
Parallelism in Large Model Training

- Data parallelism
  - Replicate whole pipelines and run them in parallel

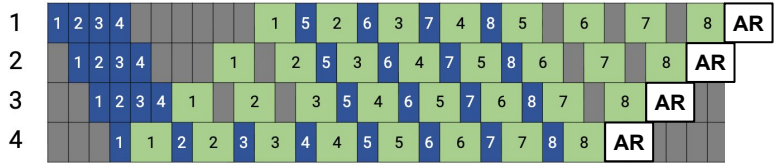


Parallelism in Large Model Training

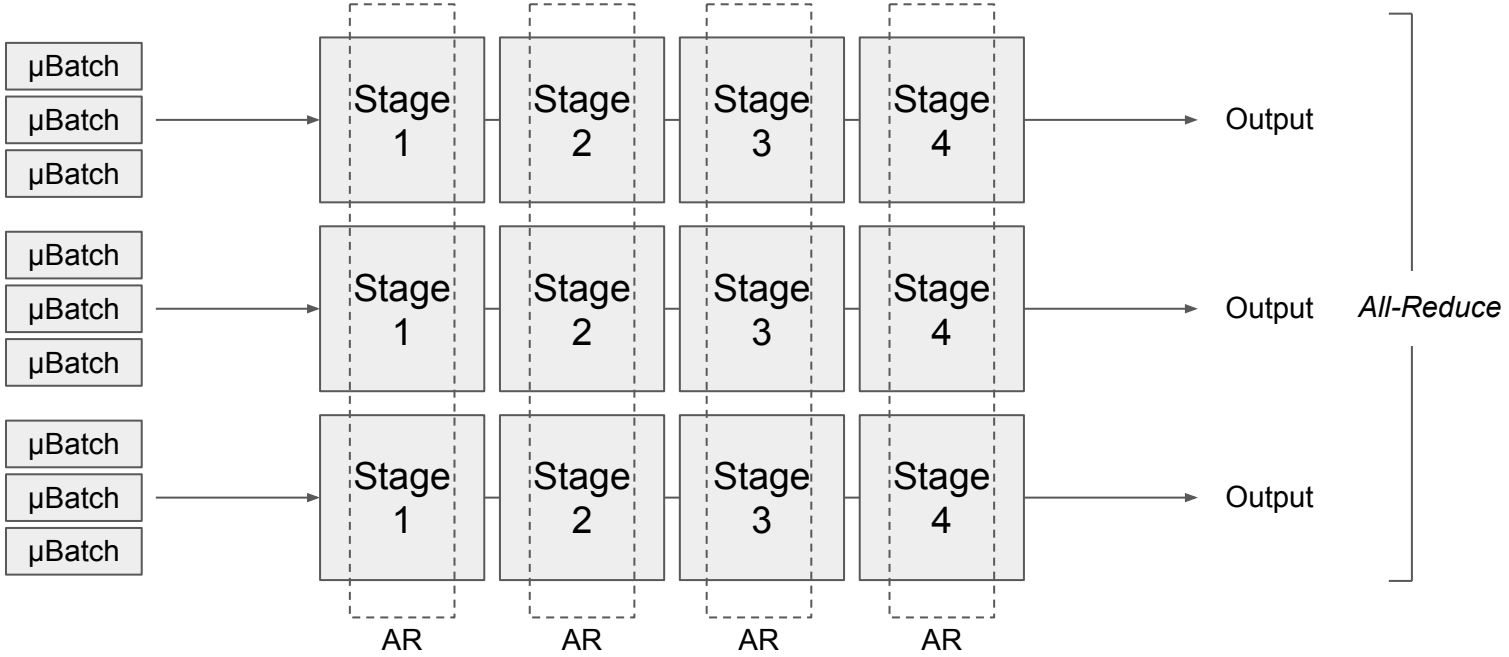
- Data parallelism
  - Replicate whole pipelines and run them in parallel



Parallelism in Large Model Training



- Data parallelism
  - Replicate whole pipelines and run them in parallel



Essential Layers – Training Framework

- Distributed training system
  - NVIDIA Megatron-LM and NeMo, Microsoft DeepSpeed, PyTorch/Meta TorchTitan
  - Composes PyTorch-level APIs to actually make training happen across 16k GPUs
  - Input
    - Model definition (`nn.Module`)
    - Dataset (sequence of batches) – training, validation, and test sets
    - Hardware specs and parallelism configuration

Model layers are already implemented with TP support

96 H100 GPUs each with 80GB memory
Pipeline parallel 4, tensor parallel 8, data parallel 3

Additionally
- Training progress monitoring
- Asynchronous parallel checkpointing & restoring
- Mixed precision training (e.g., bf16 and fp8)
- … and more

Essential Layers – Training Framework

- Where everything meets: PyTorch
  - PyTorch also lets you launch collective communication with tensors

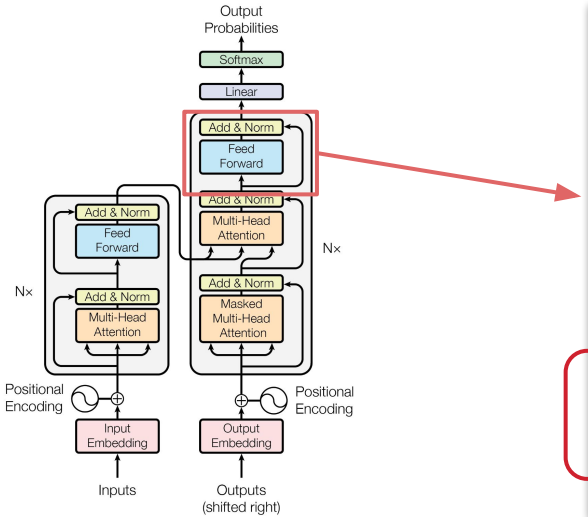


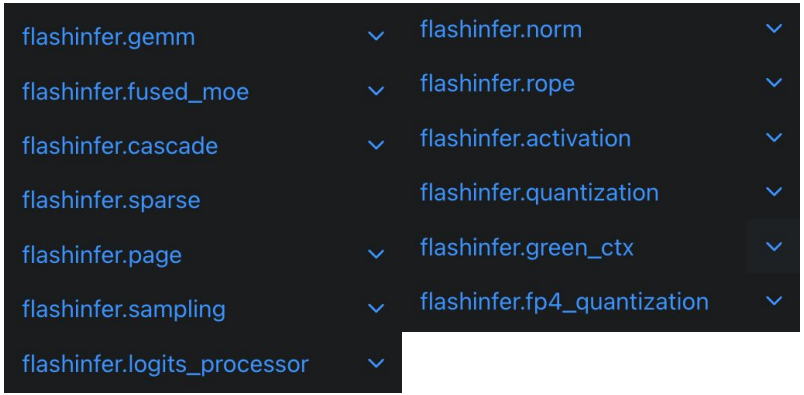
Figure 1: The Transformer - model architecture.

```python
import torch
import torch.distributed

# Shared weights
rank = torch.distributed.get_rank()
w1_shard = w1.chunk(chunks=4, dim=1)[rank]  # Column parallel
w2_shard = w2.chunk(chunks=4, dim=0)[rank]  # Row parallel

# Tensor parallel 2-layer MLP
y1 = torch.matmul(x, w1_shard)
y2 = torch.relu(y1)
y = torch.matmul(y2, w2_shard)  # Partial output
torch.distributed.all_reduce(y, op=torch.distributed.ReduceOp.SUM)
```

Essential Layers – Computation

- Optimized kernels are hard to write
- Kernel libraries
  - cuBLAS (linear algebra), cuDNN (deep learning), cuRAND (random number generation)
  - FlashAttention (all kinds of attention)
  - FlashInfer (all sorts of kernels relevant to LLM serving)

Insanely optimized matrix multiplication kernel



Essential Layers – Computation

- Programming the GPU
  - Low-level: CUDA (C++ extension)

NB: cuBLAS is closed-source, so the source code is hidden

```
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void vec_add(const int *a, const int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // global thread index
    if (i < n) c[i] = a[i] + b[i];                  // each thread computes one element
}
```

- Body of the **kernel** is for one thread
- Every thread runs the exact same code
- Threads' `i` values (thread index) are different

Essential Layers – Computation

- Programming the GPU
  - Low-level: CUDA (C++ extension)
  - High-level: Triton (Python)

```cuda
#include <stdio.h>
#include <cuda_runtime.h>

__global__ void vec_add(const int *a, const int *b, int *c, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;  // global thread index
    if (i < n) c[i] = a[i] + b[i];                  // each thread computes one element
}
```

```python
import triton.language as tl

@triton.jit
def vec_add_kernel(a_ptr, b_ptr, c_ptr, n_elements, BLOCK_SIZE: tl.constexpr):
    pid = tl.program_id(0)                  # unique program id
    offsets = pid * BLOCK_SIZE + tl.arange(0, BLOCK_SIZE)
    mask = offsets < n_elements             # guard for tail
    a = tl.load(a_ptr + offsets, mask=mask)
    b = tl.load(b_ptr + offsets, mask=mask)
    tl.store(c_ptr + offsets, a + b, mask=mask)
```

- Body of the **kernel** resembles single-threaded
- Programming model is closer to PyTorch

Essential Layers – Training Framework

- Where everything meets: PyTorch
  - PyTorch also lets you launch collective communication with tensors

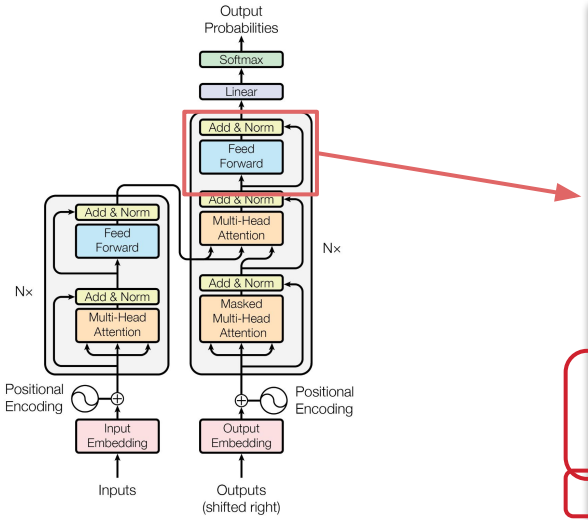


Figure 1: The Transformer - model architecture.

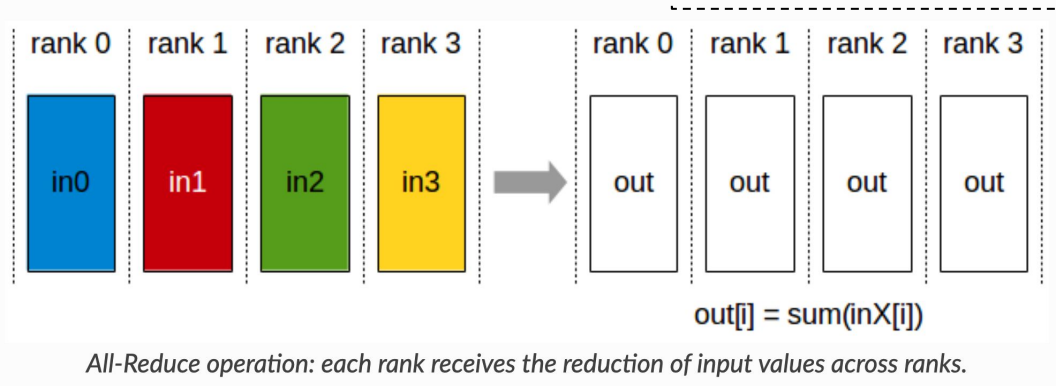
```python
import torch
import torch.distributed

# Shared weights
rank = torch.distributed.get_rank()
w1_shard = w1.chunk(chunks=4, dim=1)[rank]  # Column parallel
w2_shard = w2.chunk(chunks=4, dim=0)[rank]  # Row parallel

# Tensor parallel 2-layer MLP
y1 = torch.matmul(x, w1_shard)
y2 = torch.relu(y1)
y = torch.matmul(y2, w2_shard)  # Partial output
torch.distributed.all_reduce(y, op=torch.distributed.ReduceOp.SUM)
```

Essential Layers – Communication

- NCCL
  - Point-to-point (**P2P**): Sending data between from one GPU to another
  - **Collective communication**
    - All-Reduce (`ncclAllReduce`)

`torch.distributed` calls NCCL APIs internally

PyTorch handles NCCL communicator initialization, buffer management, process group management, synchronization with other async computations, etc.



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

NCCL Collective Operations

Essential Layers – Training Framework

- Where everything meets: PyTorch
  - PyTorch also lets you launch collective communication with tensors

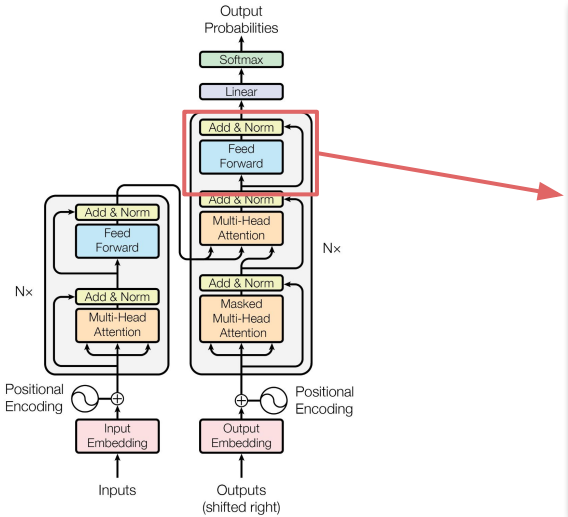


Figure 1: The Transformer - model architecture.

```python
import torch
import torch.distributed

# Shared weights
rank = torch.distributed.get_rank()
w1_shard = w1.chunk(chunks=4, dim=1)[rank]  # Column parallel
w2_shard = w2.chunk(chunks=4, dim=0)[rank]  # Row parallel

# Tensor parallel 2-layer MLP
y1 = torch.matmul(x, w1_shard)
y2 = torch.relu(y1)
y = torch.matmul(y2, w2_shard)  # Partial output
torch.distributed.all_reduce(y, op=torch.distributed.ReduceOp.SUM)
```

Congratulations! You've successfully computed two matmuls across four GPUs.

Essential Layers – Training Framework

- Distributed training system
  - NVIDIA Megatron-LM and NeMo, Microsoft DeepSpeed, PyTorch/Meta TorchTitan
  - Composes PyTorch-level APIs to actually make training happen across 16k GPUs
  - Input
    - Model definition (`nn.Module`)
    - Dataset (sequence of batches) – training, validation, and test sets
    - Hardware specs and parallelism configuration
  - The reality
    - Everything is intermixed with everything else
    - You need to write your model in a parallelism-aware manner
    - Your parallelism implementation has to be model-aware
    - Even simple optimizations can lead to numerics issues

At the same time, these are also fantastic, impactful, and very difficult challenges left to be solved!