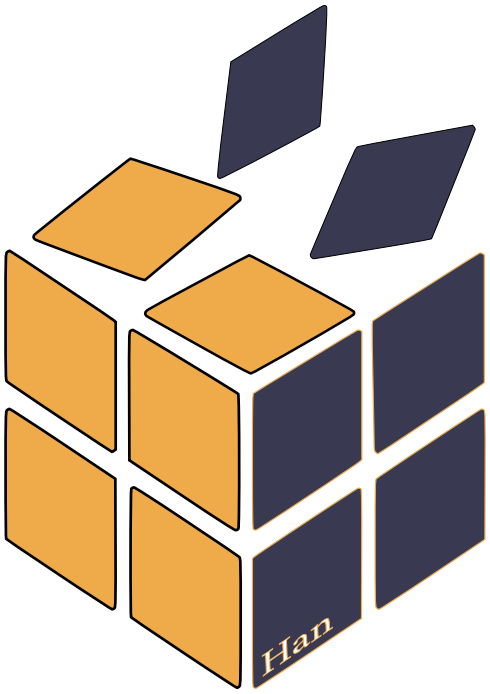
Deep Learning System Design



Engineering and Service Architectures

Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University

Singapore
hancheol001@e.ntu.edu.sg
February 9, 2025

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Complexity of Matrix Multiplication

Matrix multiplication is a fundamental operation in many computational tasks, including neural networks. The complexity of multiplying two matrices depends on their dimensions. Let's dive into the specifics.

- Let A be a matrix of size $m \times k$.
- Let B be a matrix of size $k \times n$.
- The result C will be a matrix of size $m \times n$.

Standard Matrix Multiplication: For each element c_{ij} in the resulting matrix C:

$$c_{ij} = \sum_{l=1}^{k} a_{il} \cdot b_{lj}$$

This involves:

- Multiplications: k multiplications for each element c_{ij}.
- Additions: $k - 1$ additions for each element c_{ij}.

Complexity

- The total number of elements in C is $m \times n$.
- Therefore, the total number of multiplications is $m \times n \times k$.
- The total number of additions is $m \times n \times (k - 1)$.

Thus, the total complexity is $O(m \times n \times k)$.

Even though there are several advanced methods, the standard $O(m \times n \times k)$ complexity is often used in practice, due to the simplicity and efficiency of implementation on modern hardware. Optimized libraries (like BLAS, cuBLAS for GPUs) leverage hardware-specific optimizations to improve practical performance.

Part II

Parallelism

Chapter 2

Data Parallelism

2.1 Data Parallel

The first step of the typical training loop for deep learning models is to split a dataset into batches so that we can feed them into the model and compute gradients corresponding to them. As the model size grows up, we couldn't fit the model into a single GPU. The *data parallelism* tries to tackle the issue by clone the model across multiple GPUs so that each GPU can take a small portion of the batches for each iteration. Data Parallel (sometimes referred to as "single-node data parallel") is typically used when you have **multiple GPUs on a single machine**.

Let's say the batch size is 10 and we have 5 GPUs. Then, each GPU takes 2 batches and calculate gradients by on its own. The calculated gradients are then synchronized across the GPUs pretending they are computed on a single GPU. Finally, the synchronized gradient information is going to be distributed to them.

There are some important things to mention:

1. One process (or master thread) becomes a bottleneck for gradient aggregation and parameter updates.
2. As you increase the number of GPUs, or try to involve multiple machines, communication overhead grows significantly and can slow down training.
3. Each GPU holds a copy of the entire model, which can be large.

2.2 Distributed Data Parallel

To alleviate such issues, we can adopt an approach called *Distributed Data Parallel* (DDP), which is designed to scale training across many GPUs, potentially across multiple machines (nodes). Modern deep learning frameworks (like PyTorch torch.nn.parallel.DistributedDataParallel) typically recommend DDP as the best practice for multi-GPU/multi-node training due to better performance and scalability. During backpropagation, gradients are shared among GPUs through efficient communication primitives, resulting in synchronized model parameters across all GPUs.

Key benefits:

- Scalability: You can increase the number of GPUs (and even add more machines) to handle large datasets and bigger models.
- Performance: DDP typically provides better performance than older methods like NN.DATAPARALLEL (in PyTorch) because it uses *all-reduce* and eliminates the single "master" bottleneck.
- Flexibility: You can combine DDP with other parallelization strategies (*e.g.,* model parallel, sharded data parallel, pipeline parallel) if needed.

2.2.1 Concepts and Terminology

All-Reduce is a collective communication operation commonly used in distributed computing (especially in high-performance computing and deep learning). In simple terms:

- Each process (or GPU) starts with its own data (*e.g.,* local gradients).
- These data are combined (usually via a reduction operation like sum, mean, min, or max) across all processes.
- The result of that reduction (*e.g.,* the summed gradients) is then shared back so that every process receives the same reduced value.
- Hence the name: "all" (everyone gets the result) + "reduce" (combine data).

Basic Terms:

- World Size: The total number of processes engaged in the distributed job. Often, we run one process per GPU, so world size is the number of GPUs.
- Rank: A unique integer ID assigned to each process. Ranks typically range from 0 to $world\_size - 1$. Rank 0 is often referred to as the "leader" or "master" process, but in DDP, every process does roughly the same work.
- Local Rank: When multiple GPUs reside on a single node, local rank identifies which GPU a specific process is mapped to on that local machine (*e.g.,* 0 for the first GPU, 1 for the second, etc.).
- Backend: The communication backend used for synchronization (*e.g.,* nccl). For GPU training, NCCL is typically recommended because it's optimized for high-performance GPU-to-GPU communication.
- Initialization Method: Describes how processes connect with each other (*e.g.,* a TCP store, a file-based store). This allows all processes to know who's who in the cluster.

2.2.2 How DDP Works Under the Hood

1. Process Per GPU: Each GPU runs the same script in its own process.
2. Data Subset: A DistributedSampler ensures that each process sees a unique subset of data. This prevents overlap in data usage among GPUs.
3. Full Model Copy: Each GPU has a full replica of the model in memory.

- For massive models, consider *Sharded DDP* (*e.g.,* PyTorch's FSDP or DeepSpeed ZeRO) to split parameters across GPUs.

4. All-Reduce Gradient Sync: After backprop, gradients are summed (or averaged) across processes with an all-reduce operation. This keeps all models in sync.

Chapter 3

Pipeline Parallelism

3.1 Introduction

The basic idea of the data parallel is to distribute the model across GPUs. However, if the model size is bigger than the VRAM of GPU, the model wouldn't fit in a single GPU. To resolve the issue, we have to split the model across GPUs. For instance, we can put the half of the model into the fist GPU and the remaining half into the second GPU. This technique is called *pipeline parallelism*.

Pipeline Parallelism is a strategy for distributing large deep learning models across multiple devices (GPUs) by splitting the model layers into sequential stages. Rather than replicating the entire model on each GPU or sharding the parameters themselves, pipeline parallelism assigns a subset of layers to each device in a pipeline-like fashion. This technique is especially helpful when:

- The model is too large to fit on a single GPU, but it can be split into chunks (layers/stages).
- You want to keep multiple GPUs actively working on different portions (stages) of the forward and backward pass concurrently.

3.1.1 Illustration of the Pipeline

In pipeline parallelism, the model is divided into N stages, and each stage is placed on a different GPU (or sometimes on multiple GPUs if you have many layers). Think of it like an assembly line:

- Stage 1: Layers $1 - k$
- Stage 2: Layers $(k + 1) - m$
- Stage 3: Layers$(m + 1) - \ldots$
- and so on.

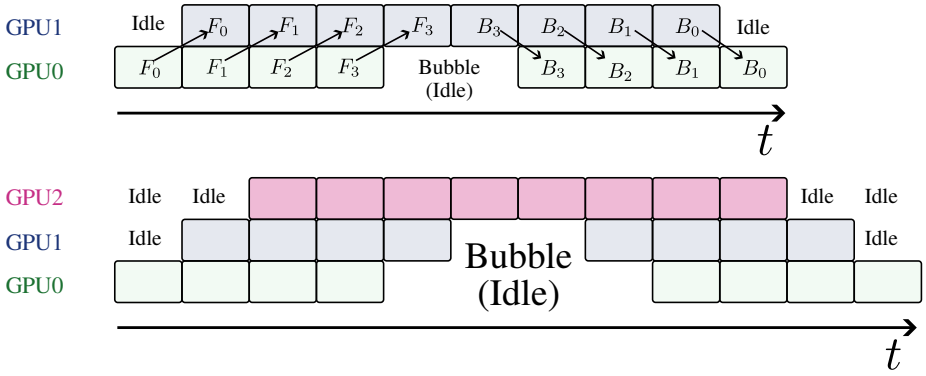


Figure 3.1: The Illustration of the pipeline parallel on two GPUs. As you can see the bubble tends to grow as we increase the number of GPUs

The input minibatch is then split into smaller micro-batches (smaller pieces of data), which flow sequentially through these stages. In other words, the micro-batch is the basic unit of the input to the pipeline parallelism.

- While Stage 1 is processing the next micro-batch, Stage 2 can concurrently work on the intermediate outputs from Stage 1's previous micro-batch.

Example: Imagine a 2-stage pipeline parallel setup (for simplicity):

- GPU 0: Holds Layers 1–3
- GPU 1: Holds Layers 4–6

If you have a batch of data with 32 samples, you might split it into 4 micro-batches of size 8 each. Then, forward Pass can be processed as follows:

1. Micro-Batch 1
   (a) Step A: GPU 0 processes layers 1–3 for micro-batch 1.
   (b) Step B: Once GPU 0 is done with those layers, it sends the activations for micro-batch 1 over to GPU 1.
   (c) Step C: GPU 1 then processes layers 4–6 for micro-batch 1.
2. Micro-Batch 2
   (a) As soon as GPU 0 finishes Step A for micro-batch 1 and passes the data to GPU 1, GPU 0 is free to start micro-batch 2 (layers 1–3).
   (b) Meanwhile, GPU 1 is busy processing micro-batch 1 (layers 4–6).
   (c) Once GPU 0 finishes its part for micro-batch 2, it sends those activations to GPU 1—which will be ready to handle them as soon as it's done with micro-batch 1.
3. Micro-Batch 3 and 4
   (a) This pattern continues in an overlapping fashion: while GPU 1 is busy with micro-batch 2, GPU 0 can start on micro-batch 3, and so on.

The key benefit is concurrency:

- While GPU 0 is processing micro-batch 2, GPU 1 can process micro-batch 1.
- This overlap leads to higher GPU utilization.

Backward pass is a bit more complex because:

- You need gradient signals to flow in the reverse order of the forward pipeline.
- Each stage waits until it receives the gradient from the next stage before it can compute its own local gradients and pass them back to the previous stage.

However, the overall concept is similar-multiple stages can run backprop (on different micro-batches) in parallel, thereby keeping all GPUs busy.

3.1.2 Pipeline Bubbles

When using pipeline parallelism, you often hear about *pipeline bubbles*. This refers to idle times on some GPUs before the assembly line is fully loaded or after it starts to wind down.

- Start-up Bubble: In the very beginning, GPU 1 must wait until GPU 0 finishes the first forward pass for micro-batch 1. GPU 1 sits idle during that initial delay.
- Wind-down Bubble: After the last micro-batch enters GPU 0, GPU 1 continues to process the pipeline while GPU 0 is idle.

These bubbles can lead to less-than-ideal speedups, but you can mitigate them by using enough micro-batches to keep the pipeline busy most of the time.

3.1.3 Combining Pipeline Parallelism with Other Forms of Parallelism

In practice, pipeline parallelism is often combined with:

- Data Parallelism: You still replicate each stage across multiple GPUs to handle separate shards of data.
- Tensor Parallelism / Model Parallelism: Instead of giving entire layers to one GPU, you split the parameters or compute of a single layer across multiple GPUs (common in large language model setups, *e.g.,* Megatron-LM).
- Sharded Optimizer Approaches (*e.g.,* ZeRO, FSDP): Distribute optimizer states and gradients to reduce memory overhead.

Chapter 4

Tensor Parallelism

4.1 Introduction

Let's go over an example:

- x is a row vector of shape $[1, d_{\text{in}}]$ (the input).
- W is a weight matrix of shape $[d_{\text{in}}, d_{\text{out}}]$.
- output is $[1, d_{\text{out}}]$.

We have two GPUs, GPU 0 and GPU 1. We want to split (shard) the weight matrix W across two GPUs. One common approach is column parallelism:

- GPU 0 holds columns $[0, 1]$
- GPU 1 holds columns $[2, 3]$

This means each GPU stores some columns of W. Let's denote:

$$W = \left[W_{\text{left}} \mid W_{\text{right}}\right]$$

where

- W_{left} is a 4×2 matrix on GPU 0,
- W_{right} is a 4×2 matrix on GPU 1.

In numeric form, suppose

$$W = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \\ 2 & 0 & 3 & 1 \\ -1 & 4 & 8 & 2 \end{bmatrix}.$$

Then, for column parallel:

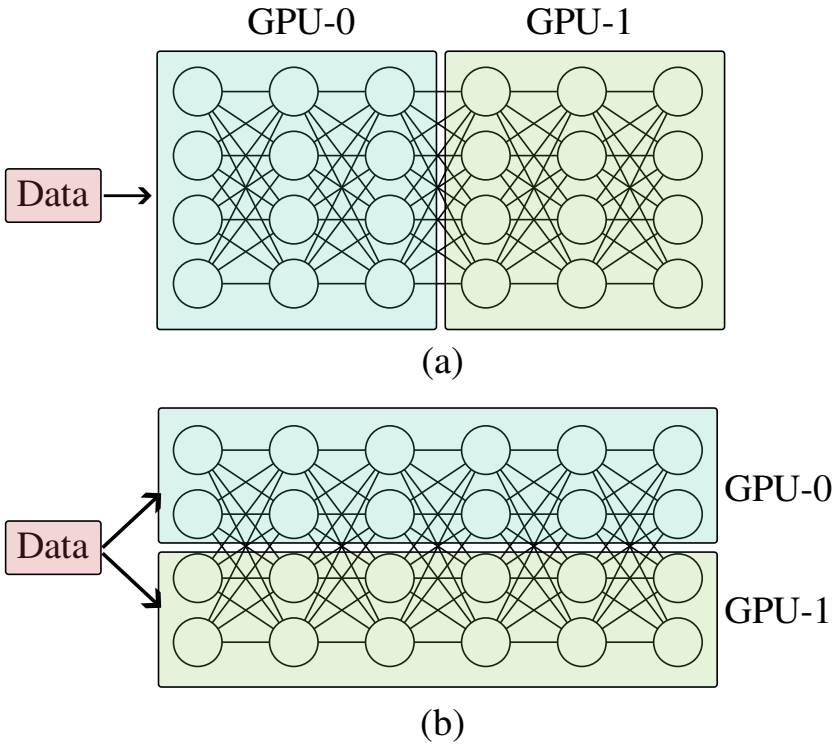


Figure 4.1: (a): Pipeline parallelism. (b) Tensor parallelism.

- GPU 0:

$$W_{\text{left}} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 2 & 0 \\ -1 & 4 \end{bmatrix}.$$

- GPU 1:

$$W_{\text{right}} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 3 & 1 \\ 8 & 2 \end{bmatrix}.$$

Given the input

$$x = [\,1,\ 2,\ 0,\ 1\,].$$

We can treat x as a row vector $[1, 4]$. For column parallelism, each GPU needs the entire input x so it can multiply by its subset of columns:

- We copy the x to both GPU 0 and GPU 1.
  - This is typically a small overhead compared to storing large weight matrices.
- Then, compute the matrix multiplications for each matrix.
- Finally, concatenate the outputs.

$$\text{output} = [\,\text{partial}_0 \,\big|\, \text{partial}_1\,] = [\,6,\ 14,\ 27,\ 24\,].$$

- Some frameworks do a ring-all-gather, or they might place this final output on one GPU if needed, etc.

When we do backprop, we can update the model's parameters in the opposite direction.

Chapter 5

N-Dim Parallelism

Part III

Transformers

5.1 Flash Attention

Chapter 6

Tokenization

Chapter 7

Model Compression

Bibliography

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[2] Rachit Singh. . `https://rachitsingh.com/elbo_surgery/`, 2017. Online; accessed 29 January 2014.

[3] Hany Hassan, Anthony Aue, Chang Chen, Vishal Chowdhary, Jonathan Clark, Christian Federmann, Xuedong Huang, Marcin Junczys-Dowmunt, William Lewis, Mu Li, Shujie Liu, Tie-Yan Liu, Renqian Luo, Arul Menezes, Tao Qin, Frank Seide, Xu Tan, Fei Tian, Lijun Wu, Shuangzhi Wu, Yingce Xia, Dongdong Zhang, Zhirui Zhang, and Ming Zhou. Achieving human parity on automatic chinese to english news translation. *CoRR*, abs/1803.05567, 2018.