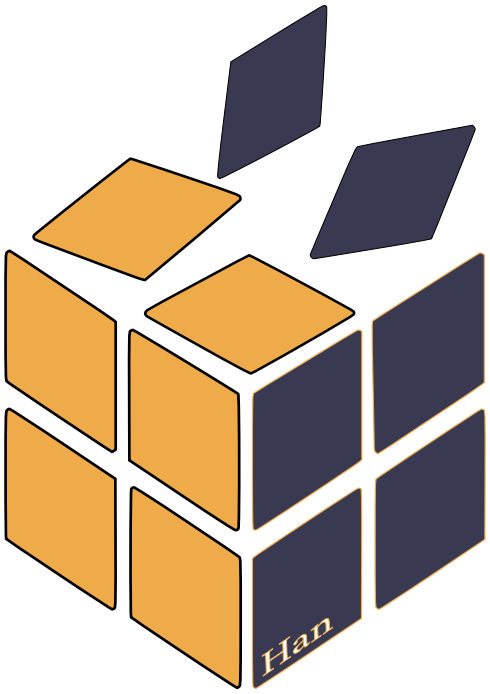
Deep Learning System Design



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.

Part III

Transformers

2.3 Flash Attention

Chapter 3

Tokenization

Chapter 4

Model Compression