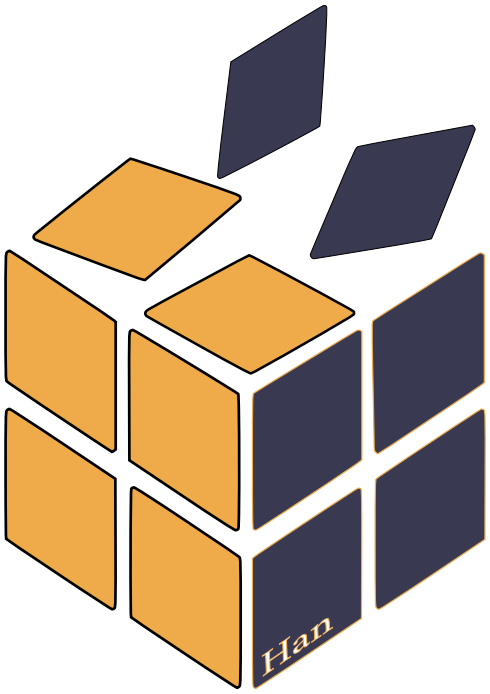
Deep Learning System Design



Han Cheol Moon
tabularasa8931@gmail.com

The logo depicts a cube puzzle gradually coming together, reflecting the journey of learning. Each piece represents a fragment of knowledge, and as they fall into place, they reveal the larger structure of understanding. It conveys the idea that growth is a process — knowledge is completed bit by bit.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Operations challenges with LLMs

- **Long download times** (*e.g.,* Bloom LLM is 330GB).
- **Longer deploy times** (*e.g.,* Bloom takes $30 \sim 45$ mins to load the model into GPU).
- Along with increases in model size often come increases in **inference latency**.
- **Managing GPUs**
- **Peculiarities of text data**: unlike other fields, texts have ambiguities.
- **Token limits for a model** create bottlenecks
- **Hallucinations cause confusion**
- **Bias and ethical considerations**
- **Security concerns**
- **Controlling costs**: *e.g.,* GPUs, infra, storage, operational costs like energy consumption during both training and inference.

1.2 LLMOps Essentials

- **Compression** is the practice of making models smaller.
  - **Quantizing** is the process of reducing precision in preference of lowering the memory requirements.
  - **Pruning** is the process of weeding out and removing any parts of the model we deem unworthy.
  - **Knowledge distillation** takes the large LLM and train a smaller language model to copy it.
  - **Low-rank approximation** is a trick to simplify large matrices or tensors to find a lower dimensional representation.

  - **Mixture of Experts** (MoE) is a technique where we replace the feed-forward (FF) layers in a transformer with MoE layers instead. FF layers are notorious for being parameter-dense and computationally intensive, so replacing them with something better can often have a large effect. MoEs are a group of sparsely activated models. They differ from ensemble techniques in that typically only one or a few expert models will be run, rather than combining results from all models. The sparsity is often induced by a *gate mechanism* that learns which experts to use and/or a router mechanism that determines which experts should even be consulted.

- **Distributed computing** is a technique used in DL to parallelize and speed up large, complex neural networks by dividing the workload across multiple devices or nodes in a cluster. This approach significantly reduces training and inference times by enabling concurrent computation, data parallelism, and model parallelism.

  - **Data parallelism**: splitting up the data and running them through multiple copies of the model or pipeline.
  - **Tensor parallelism**: This approach takes advantage of matrix multiplication properties of split up the activations across multiple processors, running the data through and then combining them on the other side of the processors.
  - **Pipeline parallelism**: This creates a pipeline, as input data will go to the first GPU, process, then transfer to the next GPU, and so on until it's run through the entire model.
  - **3D parallelism**: We want to take advantages of all three parallelism practices as they can all be run together. This is known as 3D parallelism, which combines data, tensor and pipeline parallelism (DP+TP+PP) together. Since each technique and thus dimension will require at least two GPUs to run 3D parallelism, we will need at least eight GPUs to get started.

1.3 LLM Operations Infrastructures

- Data infrastructure is the foundation of DataOps. (*e.g.,* Airflow, Prefect, and Mage)
- Experiment trackers (*e.g.,* MLFlow and Weights & Biases)
- Model registry
- Feature stores (*e.g.,* Feast) is a centralized system for managing, storing, and serving features (the inputs to machine learning models) in a consistent and reliable way. It sits between your raw data sources and your ML models, making it easier to build, deploy, and maintain production ML systems. Why do we need it? In ML, the same features are used in two places:

  - Training – when building the model.
  - Serving/Inference – when the model is deployed to make predictions.

- Vector databases (*e.g.,* Qdrant, Pinecone, and Milvus) are specialized databases that store vectors along with some metadata around the vector, which makes them great for storing embeddings. The power of vector databases isn't in their storage but in the way that they search through the data.

- Monitoring systems (*e.g.,* whylogs and): ML models are often fail silently (*e.g.,* data drift [1]).
- GPU-enabled workstations (*e.g.,* H100 provides 80GB and NVL [2])
  - If you aren't sure which GPU you need to run which model, multiply the number of billions of parameters by two, since most models at inference will default to run at half precision, FP16 or BF16, which means we need at least 2 bytes for every parameter. You will need a little extra as well for embedding model, which will be about another gigabyte, and more for the actual toknes. One token is about 1MB. For 16 bathes of this size, you will need an extra 8GB of space.
  - For training you will need a lot more space (*e.g.,* full precision, optimizer tensors and gradients). Roughly you need 16 bytes for every parameter, so to train a 7B parameter model, you will need 112GB of memory.
- Deployment service (*e.g.,* NVIDIA Triton Inference Service, MLServer, Seldon, BentoML)

[1] Data drift in Machine Learning (ML) systems refers to changes in the statistical properties of input data (or the relationship between inputs and outputs) over time, which can degrade model performance if not monitored and addressed.

[2] NVIDIA Link is NVIDIA's high-speed interconnect technology. It allows GPUs (and sometimes CPUs) to communicate with each other much faster than PCIe. In multi-GPU systems (like deep learning servers, HPC, or AI supercomputers), GPUs need to share data frequently (*e.g.,* gradients and parameters). If they only use PCIe, communication is slower, creating a bottleneck.

Chapter 2

Preliminaries

2.1 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

Data Engineering

Chapter 3

Data Engineering for LLMs

Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning.

There isn't more valuable asset than your data. All successful AI and ML initiatives are built on a good data engineering foundation. It's important then that we acquire, clean, and curate our data.

3.1 Models and the Foundation

The most important dataset you will need to collect when training is the model weights of a pretrained model.

3.1.1 Evaluating LLMs

When evaluating a model, you will need two things: (*i*) a *metric* and (*ii*) a *dataset*.

Metrics

- ROUGE (Recall-Oriented Understudy for Gisting Evaluation)
- BLEU (BiLingual Evaluation Understudy)
- BPC (*e.g.,* Perplexity): The bits per character (BPC) evaluation is an example of an entropy-based evaluation for language models.

Industry benchmarks

- GLUE (General Language Understanding Evaluation) is essentially a standardized test for language models to measure performance versus humans and each other on language tasks meant to test understanding.

- SuperGLUE
- MMLU (Massive Multitask Language Understanding).

Responsible AI benchmarks

- HONEST evaluation metric compares how hurtful prompt completions are for different genders.
- Some datasets:
  - WinoBias dataset focuses on gender bias.
  - CALM
  - WinoQueer

Developing your own benchmark

- OpenAI's Evals library
- Huggingface's Evaluate

Evaluating code generators The basic setup looks like this:

1. Have your model generate code based on docstrings.
2. Run the generated code in a safe environment on prebuilt tests to ensure they work and that no errors are thrown
3. Run the generated code through a profiler and record the time it takes to complete.
4. Run the generated code through a security scanner and count the number of vulnerabilities.
5. Run the generated code against architectural fitness functions to determine artifacts like how much coupling, integrations, and internal dependencies there are.
6. Run steps 1 to 5 on another LLM.
7. Compare results.

Evaluating model parameters There's a lot you can learn by simply looking at the parameters of an ML model. For instance, an untrained model will have a completely random distribution.

```
import weightwatcher as ww
from transformers import GPT2Model

gpt2_model = GPT2Model.from_pretrained("gpt2")
gpt2_model.eval()

watcher = ww.WeightWatcher(model=gpt2_model)
details = watcher.analyze(plot=False)
print(details.head())
#    layer_id       name          D  ...       warning        xmax        xmin
```

```
# 0          2  Embedding  0.076190  ... over-trained 3837.188332    0.003564
# 1          8     Conv1D  0.060738  ...               2002.124419  108.881419
# 2          9     Conv1D  0.037382  ...                712.127195   46.092445
# 3         14     Conv1D  0.042383  ...               1772.850274   95.358278
# 4         15     Conv1D  0.062197  ...                626.655218   23.727908
```

The spectral analysis plots evaluate the frequencies of eigenvalues for each layer of a model. These plots tell you whether a model (or layer) looks well-trained and generalizes well or is unstable/poorly conditioned. Shape of the Spectrum (How eigenvalues are distributed)

- Power-law exponent (α):
  - Good if between 2 and 6: the layer is well-trained.
  - Bad if $\alpha > 6$: layer might be undertrained or over-regularized.
- Fit quality (Dks):
  - Low Dks: spectrum matches the expected "heavy-tailed" shape, reliable.
  - High Dks: poor fit, unstable or unstructured layer.

3.2 Data for LLMs

It has been shown that data is the most important part of training an LLM.

Table 3.1: Summary of datasets

Dataset	Contents	Size	LastUpdate
WikiText	English Wikipedia	<1GB	2016
Wiki-40B	Multi-lingual Wikipedia	10GB	2020
Europarl	European Parliament proceedings	1.5GB	2011
Common Crawl	The internet	$\sim$ 300GB	Ongoing
OpenWebText	Curated internet using Reddit	55GB	2019
The Pile	Everything above plus specialty datasets (books, law, med)	825GB	2020
RedPajama	GitHub, arXiv, Books, Wikipedia, StackExchange , and multiple version of Common Crawl	5TB	2023
OSCAR	Highly curated multilingual dataset with 166 languages	9.4TB	Ongoing

3.2.1 Data cleaning and preparation

If you pulled any of the previously mentioned datasets, you might be surprised to realize most of them are just giant text dumps. There are no labels or annotations, and feature engineering hasn't been done at all.

LLMs are trained via self-supervised manner to predict the next word or a masked workd, so a lot of traditional data cleaning and preparation processes are unneeded. This fact leads many to believe that data cleaning as a whole is unnecessary.

Data cleaning and curation are difficult, time-consuming, and ultimately subjective tasks that are difficult to tie to key performance indicators (KPIs). Still, taking the time and resources to clean your data will create a more consistent and unparalleled user experience.

The right frame of mind when preparing your dataset:

1. Take your pie of data and determine a schema for the features
2. Make sure all the features conform to a distribution that makes sense for the outcome your're trying to get through normalization or scaling.
3. Check the data for bia/anomalies (most businesses skip this step by using automated checking instead of informed verification).
4. Convert the data into a format for the model to ingest (for LLMs, it's through tokenization and embedding).
5. Train, check, and retrain.

Note

For more information, check out Fundamentals of Data Engineering, WizardLM, and LIMA: Less Is More for Alignment.

Instruct Schema is one of the most effective and widely used data formats for fine-tuning models. Instruction tuning works on the principle that providing a model with explicit instructions for a task leads to better performance than simply giving it raw prompts and answers. In this approach, the data explicitly demonstrates what the model should do, making it clearer and more aligned with human intent. However, preparing such datasets is more demanding than assembling general web data, since each entry must be carefully constructed to match a structured format, typically including an instruction, optional input, and the expected output. You need to prepare your data to match a format that will look something like this:

```
###Instruction

{user input}

###Input

{meta info about the instruction}

###Response

{model output}
```

It is a structured way of formatting data so that each example clearly contains:

- An instruction (what the model should do).
- An input (optional context or data the model works on).
- An output (the desired response).

For instance,

```
{
  "instruction": "Translate the following English text into Korean.",
  "input": "The stock market saw significant volatility today due to global
    economic concerns.",
  "output": <Translations>
}
```

Note

- EvolInstruct: WizardLM
- Self-instruct format, Alpaca

Ensuring proficiency with speech acts When preparing a dataset for training a model, the most important factor is ensuring the data truly reflects the task you want the model to perform. Misaligned or overly generic data reduces performance and can cause unpredictable behavior.

Dataset alignment:

- Training data must match the intended task (e.g., don't train on Titanic survivors if you want to predict Boston housing prices).
- In real-world use cases (like fast-food ordering), interactions are more diverse and unpredictable than generic datasets suggest.

Robustness and Risks:

- Instruction datasets require intentional design: if a model is only trained on "helpful" responses, it might follow harmful instructions (e.g., "help me take over the world").
- With tool access (Google, HR docs), this becomes even riskier.

Understanding speech acts (directives, representatives, commissives, expressives, declarations, verdictives) helps design datasets that match realistic user interactions.

- In language learning, this means learners should not only know grammar/vocabulary but also how to perform speech acts appropriately:
  - How to make polite requests
  - How to refuse without sounding rude
  - How to apologize or thank in culturally acceptable ways
- In AI / LLM context, it means training the model to:
  - Generate outputs that correctly perform the intended communicative function (e.g., distinguish between an instruction, a suggestion, or a formal declaration).
  - Handle pragmatic nuances, politeness, indirectness, etc.

Speech acts refer to the various functions language can perform in communication beyond conveying information. They are a way of categorizing utterances based on their intended effect or purpose in a conversation. In short, it is an action performed through speaking. For example:

- Assertives → stating something true/false.
- Directives → requesting, commanding (*e.g.,* "Get it done in the next three days").
- Commissives → promising, committing (*e.g.,* "I swear").

- Expressives → greetings, apologizing (*e.g.,* "You are the best").
- Declarations → enacting something by saying it (*e.g.,* "I now pronounce you married").
- Questions (*e.g.,* "What is this?")

Annotating the data Annotation is labeling your data, usually in a positionally aware way. For speech recognition tasks, annotations would identify the different words as noun, verb, adjective, or adverb. Annotations essentially give us metadata that makes it easier to reason about and analyze our datasets.

There are tools to help with the task:

- Prodigy: multimodal annotation tool.
- doccano: Open-source web-based platform for data annotation.
- Praat: The audio annotation tool.
- Galileo: Galileo's LLM studio helps create prompt, evaluate and speed up annotation.

3.3 Text Processors

We need to transform our dataset into something that can be consumed by the LLM. Simply, we need to turn the text into numbers.

3.3.1 Tokenization

The tokenization is often ignored when working with an LLM through an API, but it is actually vitally important for every subsequent step, and it affects the LLM's performance significantly.

Word-based Word-based tokenizers most commonly split on whitespace, but there are other methods like using regex, dictionaries, or punctuation.

Character-based Character-based encoding methods are the most straightforward and easiest to implement since we split on the UTF-8 character encodings. However, it comes with a major loss of information and fails to keep relevant syntax, semantics, or morphology (morpheme like prefix and suffixes) of the text.

Subword-based Subword-based tokenizers have proven to be the best option so far.

- BPE
- WordPiece
- SentencePiece

3.3.2 Embeddings

Embeddings provide meaning to the vectors generated during tokenization.

Part III

Training LLMs

Chapter 4

Training LLMs: How to generate the generator

4.1 Multi-GPU environments

Training is a resource-intensive endeavor. A model that only takes a single GPU to run inference on may take 10 times that many to train if, for nothing else, to parallelize your work and speed things up so you aren't waiting for a thousand years for it to finish training.

4.1.1 Setting up

It should be pointed out up front that while multi-GPU environments are powerful, they are also expensive. For the rest of us, setting up a virtual machine (VM) in Google's Compute Engine is one of the easiest methods.

Google Virtual Machine One of the easiest ways to create a multi-GPU environment is to set up a VM on Google's cloud.

1. Create a Google Cloud Project (GCP).
   - Set up billing
   - Download the gcloud CLI.
2. After setting up your account, GCP sets your GPU quotas to 0. Quotas are used to manage your costs. You need to increase to 2 or more, since we plan to use multiple GPUs.
3. Init by "`gcloud init`"
4.

4.1.2 Libraries

DeepSpeed: DeepSpeed is an optimization library for distributed deep learning. DeepSpeed is powered by Microsoft and implements various enhancements for speed in training and inference,

like handling extremely long or multiple inputs in different modalities, quantization, caching weights and inputs, and, probably the hottest topic right now, scaling up to thousands of GPUs.

To install,

1. Install PyTorch
2. `pip install deepspeed`

Accelerate: From HuggingFace, Accelerate is made to help abstract the code for parallelizing and scaling to multiple GPUs away from you so that you can focus on the training and inference side.

To install,

1. `pip install accelerate`

PyTorch FSDP Install PyTorch with distributed support (CUDA version must match drivers):

```
pip install torch torchvision torchaudio

# (Optional) For speedups
pip install torchmetrics accelerate
```

Ensure passwordless SSH between nodes if you're on a bare-metal or HPC cluster.

Create `train_fsdp.py`:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp.wrap import transformer_auto_wrap_policy

# --- Simple Transformer block ---
class ToyBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1024, 4096)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(4096, 1024)

    def forward(self, x):
        return self.fc2(self.act(self.fc1(x)))

class ToyModel(nn.Module):
    def __init__(self, depth=6):
        super().__init__()
        self.layers = nn.Sequential(*[ToyBlock() for _ in range(depth)])

    def forward(self, x):
        return self.layers(x)

def main():
    dist.init_process_group("nccl")
    torch.cuda.set_device(dist.get_rank() % torch.cuda.device_count())
```

```
    model = ToyModel().cuda()

    # Auto-wrap large layers with FSDP
    auto_wrap_policy = transformer_auto_wrap_policy
    model = FSDP(model, auto_wrap_policy=auto_wrap_policy)

    optimizer = optim.AdamW(model.parameters(), lr=1e-4)

    for step in range(20):
        x = torch.randn(8, 1024).cuda()
        y = model(x).mean()
        y.backward()
        optimizer.step()
        optimizer.zero_grad()
        if dist.get_rank() == 0:
            print(f"Step {step} done.")

    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

If your node has 4 GPUs:

`torchrun -nproc_per_node=4 train_fsdp.py`

- 4 processes (one per GPU).
- NCCL backend handles GPU communication.

Let's try running multiple nodes

- Node 0: 10.0.0.1
- Node 1: 10.0.0.2
- 4 GPUs per node
- Total world size = 8 (2 nodes × 4 GPUs)

```
**On Node 0 (rank 0):**

torchrun --nnodes=2 --nproc_per_node=4 \
         --node_rank=0 \
         --master_addr=10.0.0.1 \
         --master_port=29500 \
         train_fsdp.py

**On Node 1 (rank 1):**

torchrun --nnodes=2 --nproc_per_node=4 \
         --node_rank=1 \
         --master_addr=10.0.0.1 \
         --master_port=29500 \
         train_fsdp.py
```

- `-nnodes=2`: total number of nodes.

- `-nproc_per_node=4`: GPUs per node.
- `-node_rank`: each node's unique index.
- `-master_addr`: IP/hostname of rank 0 node.
- `-master_port`: open port for coordination.

- Activation checkpointing: Saves memory by discarding intermediate activations during forward pass and recomputing them during backward pass.
- Mixed precision: Uses FP16 or BF16 for computations (faster, less memory) while keeping FP32 for stability in some ops.

Full Example with FSDP + Checkpointing + AMP:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp.wrap import transformer_auto_wrap_policy
from torch.utils.checkpoint import checkpoint

# --- A block with activation checkpointing ---
class CheckpointedBlock(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(1024, 4096)
        self.act = nn.ReLU()
        self.fc2 = nn.Linear(4096, 1024)

    def forward(self, x):
        def forward_fn(x):
            return self.fc2(self.act(self.fc1(x)))
        # checkpoint will discard activations & recompute in backward
        return checkpoint(forward_fn, x)

# --- Toy Model ---
class ToyModel(nn.Module):
    def __init__(self, depth=6):
        super().__init__()
        self.layers = nn.Sequential(*[CheckpointedBlock() for _ in range(depth)
    ])

    def forward(self, x):
        return self.layers(x)

def main():
    dist.init_process_group("nccl")
    torch.cuda.set_device(dist.get_rank() % torch.cuda.device_count())

    # Build model
    model = ToyModel().cuda()
    auto_wrap_policy = transformer_auto_wrap_policy
    model = FSDP(model, auto_wrap_policy=auto_wrap_policy)

    optimizer = optim.AdamW(model.parameters(), lr=1e-4)

    # Use mixed precision autocast (bf16 preferred if hardware supports it)
    scaler = torch.cuda.amp.GradScaler(enabled=True)  # works for fp16
```

```
45
46      for step in range(20):
47          x = torch.randn(8, 1024).cuda()
48
49          with torch.cuda.amp.autocast(dtype=torch.bfloat16):  # or torch.float16
50              y = model(x).mean()
51
52          # backward with gradient scaler
53          scaler.scale(y).backward()
54          scaler.step(optimizer)
55          scaler.update()
56          optimizer.zero_grad()
57
58          if dist.get_rank() == 0:
59              print(f"Step {step} done.")
60
61      dist.destroy_process_group()
62
63  if __name__ == "__main__":
64      main()
```

- Checkpointing: Wrap the forward function with `torch.utils.checkpoint.checkpoint`.
- AMP (Automatic Mixed Precision):
  - Use `torch.cuda.amp.autocast` for forward pass.
  - Use `torch.cuda.amp.GradScaler` for loss scaling (needed for FP16, not BF16).
- BF16 vs FP16:
  - Use BF16 if your GPUs are A100/H100 (more stable).
  - Use FP16 + GradScaler for V100 or older cards.

- Full State Dict: Gather the full parameters on rank 0 and save them. (simpler, larger files).
- Sharded State Dict: Each rank saves only its shard. (efficient, but needs all shards to reload).
- Rank 0 only writing: Typically only rank 0 writes to disk to avoid file conflicts.

```
1  import os
2  import torch
3  from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
4  from torch.distributed.fsdp import StateDictType, FullStateDictConfig
5
6  CHECKPOINT_DIR = "./checkpoints"
7
8  def save_checkpoint(model, optimizer, step):
9      # Ensure only rank 0 writes the file
10     rank = torch.distributed.get_rank()
11     os.makedirs(CHECKPOINT_DIR, exist_ok=True)
12
13     # Switch to FULL state dict (gathered on rank 0)
14   # offload_to_cpu=True: keeps memory usage down when gathering full state.
15     full_sd_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
16     with FSDP.state_dict_type(model, StateDictType.FULL_STATE_DICT,
       full_sd_config):
17         model_state = model.state_dict()
```

```
        optim_state = optimizer.state_dict()

    if rank == 0: # only rank 0 writes to disk.
        save_path = os.path.join(CHECKPOINT_DIR, f"step_{step}.pt")
        torch.save({"model": model_state, "optimizer": optim_state, "step":
    step}, save_path)
        print(f"[Rank 0] Saved checkpoint to {save_path}")
```

```
def load_checkpoint(model, optimizer, load_path):
    # Load only on rank 0
    rank = torch.distributed.get_rank()
    map_location = "cpu" if rank == 0 else "meta"  # meta avoids OOM on other
    ranks
    checkpoint = torch.load(load_path, map_location=map_location)

    # Use FULL state dict context
    full_sd_config = FullStateDictConfig(offload_to_cpu=True, rank0_only=True)
    with FSDP.state_dict_type(model, StateDictType.FULL_STATE_DICT,
    full_sd_config):
        model.load_state_dict(checkpoint["model"])

    # Broadcast model weights to all ranks
    torch.distributed.barrier()
    optimizer.load_state_dict(checkpoint["optimizer"])
    print(f"[Rank {rank}] Loaded checkpoint from {load_path}")
```

Library	**Core Idea**	**Strengths / Use Cases**
DeepSpeed	Distributed training engine with ZeRO (Zero Redundancy Optimizer) sharding	- ZeRO-1/2/3: memory savings via parameter/gradient/optimizer sharding - Offloading to CPU/NVMe - Mixture of Experts (MoE) support - Great for very large dense or MoE models
FSDP (Fully Sharded Data Parallel)	Native PyTorch module for full parameter, gradient, optimizer sharding	- First-party, stable, integrated in PyTorch - ZeRO-3–like memory scaling - Easy to use with Hugging Face Accelerate/Lightning
Megatron-LM	Parallelism library from NVIDIA (TP, PP, SP, EP)	- Tensor Parallelism (TP) for matmuls - Pipeline Parallelism (PP) for layer distribution - Sequence/Expert Parallelism for long-context and MoE - Standard for 30B–100B+ scale

Table 4.1: Comparison of Core Libraries for Multi-GPU LLM Training

- Up to 13B on few GPUs → FSDP (simpler, first-party).
- >30B or MoE, multi-node → Megatron-LM + (FSDP or DeepSpeed ZeRO).
- When GPU memory is tight → DeepSpeed ZeRO-3 (with offload).

4.2 Basic Training Techniques

Unlike traditional ML models, LLMs are often trained in stages:

- Optuna: Open source hyperparameter optimization (HPO) framework for machine learning, including Large Language Models (LLMs).

Chapter 5

Parameter Efficient Fine-Tuning

5.1 LoRA

When fine-tuning a neural network for a new task, we adjust its weights W by learning an update ΔW, yielding

$$W' \;=\; W + \Delta W.$$

LoRA (Low-Rank Adaptation) avoids updating W directly. Motivated by the **intrinsic rank hypothesis**—the idea that task-critical changes lie in a low-dimensional subspace—LoRA parameterizes the update as a low-rank product:

$$\Delta W \;=\; BA, \qquad W' \;=\; W + BA,$$

while keeping W frozen. Here $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times d}$ with $r \ll d$, so BA is a low-rank approximation to the full update.

This factorization dramatically reduces trainable parameters. Instead of learning all d^2 entries of ΔW (for a $d \times d$ weight), LoRA learns only the factors B and A, totaling $2dr$ parameters—much smaller when $r \ll d$. (The same idea applies to non-square $W \in \mathbb{R}^{d\times k}$, using $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, for a cost of $r(d + k)$ instead of dk.)

5.2 QLoRA

While LoRA reduces the number of *trainable* parameters, it still requires storing and using the full-precision base model W during fine-tuning. For very large LLMs (tens or hundreds of billions of parameters), this memory demand can exceed the capacity of a single GPU.

QLoRA (Quantized Low-Rank Adaptation) addresses this by combining LoRA with parameter *quantization*. Instead of keeping W in 16- or 32-bit precision, QLoRA stores it in a lower-bit format (typically 4-bit). During training:

- The frozen base weights W are kept in 4-bit quantized form, greatly reducing memory usage.

- On-the-fly, W is *dequantized* into higher precision (e.g., 16-bit) for computations.
- As in LoRA, trainable low-rank adapters A, B are introduced to capture task-specific updates.

The update rule remains

$$W' = W + BA,$$

but now W is quantized, while A, B are small full-precision matrices.

Part IV

LLM Services

Chapter 6

LLM Services: A Practical Guide

Production refers to the phase where the model is integrated into live or operational environment to perform its intended tasks or provide services to end users. It's a crucial phase in making the model available for real-world applications and services. In this chapter, we will explore how to package up an LLM into a service or API so that it can take on-demand requests. Then, we will study how to set up a cluster in the cloud where you can deploy this service.

6.1 Creating an LLM service

6.1.1 Model Compilation

The success of any model in production is dependent on the hardware it runs on. Unfortunately, when programming in a high-level language like Python-based frameworks like PyTorch or TensorFlow, the model won't be optimized to take full advantage of the hardware. This is where compiling comes into play. Compiling is the process of taking code written in a high-level language and converting or lowering it to machine-level code that the computer can process quickly. Compiling your LLm can easily lead to major inference and cost improvements.

Kernel Tuning In DL, and high-performance computing, a kernel is a small program or function designed to run on a GPU or other similar processors. These routines are developed by the hardware vendor to maximize chip efficiency.

During kernel tuning, the most suitable kernels are chosen from a large collection of highly optimized kernels.

Kernel Fusion A **GPU kernel** is the tiny function that runs on each element. **Fusion** = do several elementwise ops in one kernel so each element is read from VRAM once, processed in registers, then written back once.

The `y = ReLU(x + b)` example

Assume `x` and `b` are the same length (or `b` is broadcast on the last dim).
Without fusion (two kernels: add, then ReLU):

1. For every element i:
2. Read x[i] from VRAM
3. Read b[i] from VRAM
4. Compute t = x[i] + b[i]
5. Write t to VRAM ← intermediate array
6. Read t from VRAM
7. Compute y = max(t, 0)
8. Write y to VRAM

Global-memory ops per element: 5 (read x, read b, write t, read t, write y) Two kernel launches (one for add, one for ReLU).
With fusion (one kernel: add+ReLU together)

1. For every element i:
2. Read x[i] from VRAM
3. Read b[i] from VRAM
4. Compute t = x[i] + b[i] (kept in a register, not VRAM)
5. Compute y = max(t, 0)
6. Write y to VRAM

Global-memory ops per element: 3 (read x, read b, write y) One kernel launch.

ReLU Example:

```
import torch, time

B, C = 4096, 4096
x = torch.randn(B, C, device="cuda", dtype=torch.float32)
b = torch.randn(C,     device="cuda", dtype=torch.float32)

def add_relu_unfused(x, b):
    # Eager mode typically launches two kernels (add, then relu)
    return torch.relu(x + b)

# PyTorch 2.x compiler (Inductor) generates fused kernels automatically
# This fuses common elementwise chains (like  a d d ReLU ) into single kernels.
add_relu_fused = torch.compile(add_relu_unfused, backend="inductor")

# Correctness check
with torch.no_grad():
    y1 = add_relu_unfused(x, b)
    y2 = add_relu_fused(x, b)
    print("max |diff| =", (y1 - y2).abs().max().item())

```

```
# Simple timing
def bench(fn, iters=50, warmup=10):
    for _ in range(warmup):
        fn(x, b); torch.cuda.synchronize()
    t0 = time.perf_counter()
    for _ in range(iters):
        fn(x, b)
    torch.cuda.synchronize()
    return (time.perf_counter() - t0) * 1000 / iters

print("Unfused  :", bench(add_relu_unfused), "ms/iter")
print("Fused    :", bench(add_relu_fused),  "ms/iter")
```

Graph Optimization Graph optimization = semantics-preserving rewrites of the op graph (fold, fuse, simplify, relayout, retype, reschedule) so the lowered kernels do less work with less memory traffic

TensorRT NVIDIA TensorRT is an SDK that converts a trained model into an optimized "engine" and then runs that engine with very low latency/high throughput on NVIDIA GPUs. It includes an optimizer (compiler) and a lightweight runtime.

How it works (typical workflow):

1. Import your model (usually ONNX; there are parsers and framework bridges).
2. Build an optimized engine: TensorRT selects fast kernels ("tactics"), fuses layers, lowers precision (FP16/INT8) if allowed, and specializes to your shapes/hardware.
3. Serialize the engine to a .plan file (so you can load it instantly in production).
4. Run it via the TensorRT runtime (C++/Python).

```
import tensorrt as trt

logger  = trt.Logger(trt.Logger.WARNING)
builder = trt.Builder(logger)
network = builder.create_network(1 << int(trt.NetworkDefinitionCreationFlag.
    EXPLICIT_BATCH))
parser  = trt.OnnxParser(network, logger)

with open("model.onnx", "rb") as f:
    assert parser.parse(f.read()), parser.get_error(0)

config = builder.create_builder_config()
config.set_memory_pool_limit(trt.MemoryPoolType.WORKSPACE, 4 << 30)  #
    workspace budget
config.set_flag(trt.BuilderFlag.FP16)  # enable mixed precision if supported

# Dynamic-shape profile for input "input" (name must match your ONNX)
profile = builder.create_optimization_profile()
profile.set_shape("input", min=(1,3,224,224), opt=(8,3,224,224), max
    =(32,3,224,224))
config.add_optimization_profile(profile)

engine = builder.build_engine(network, config)
with open("model.plan", "wb") as f:
    f.write(engine.serialize())
```

ONNX Runtime

6.1.2 LLM storage strategies

Part V

Parallelism

Chapter 7

Data Parallelism

7.1 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.

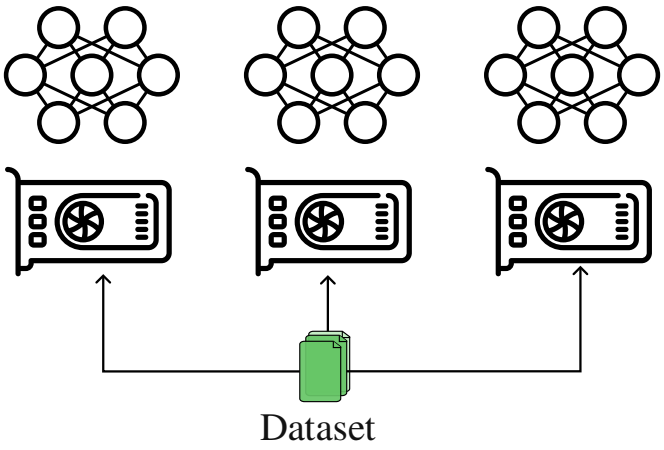
There are some important things to mention:

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.

7.2 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.

7.2.1 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.

7.2.2 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 8

Pipeline Parallelism

8.1 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 approach is often called *model parallelism*. Let's closely look at one of the model parallelism approaches, 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.

8.1.1 Illustration of the Pipeline

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

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

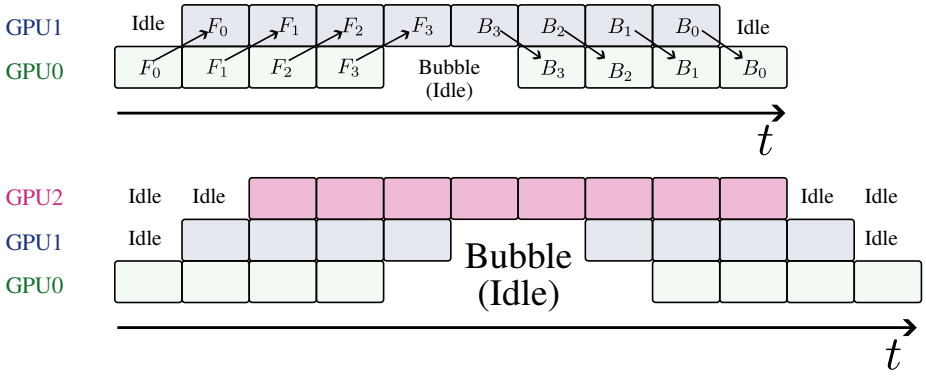


Figure 8.1: The Illustration of the pipeline parallel on two GPUs. As you can see the *bubble* (*i.e.,* underutilization) tends to grow as we increase the number of GPUs

The input minibatch is then split into smaller micro-batches (smaller pieces of data), which flow sequentially through these sub-networks. 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.

8.1.2 Pipeline Bubbles

When using pipeline parallelism, you often hear about *pipeline bubbles* (or underutilization). 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.

The percentage of idle can be computed as follows:

$$\frac{1-m}{m+n-1},$$

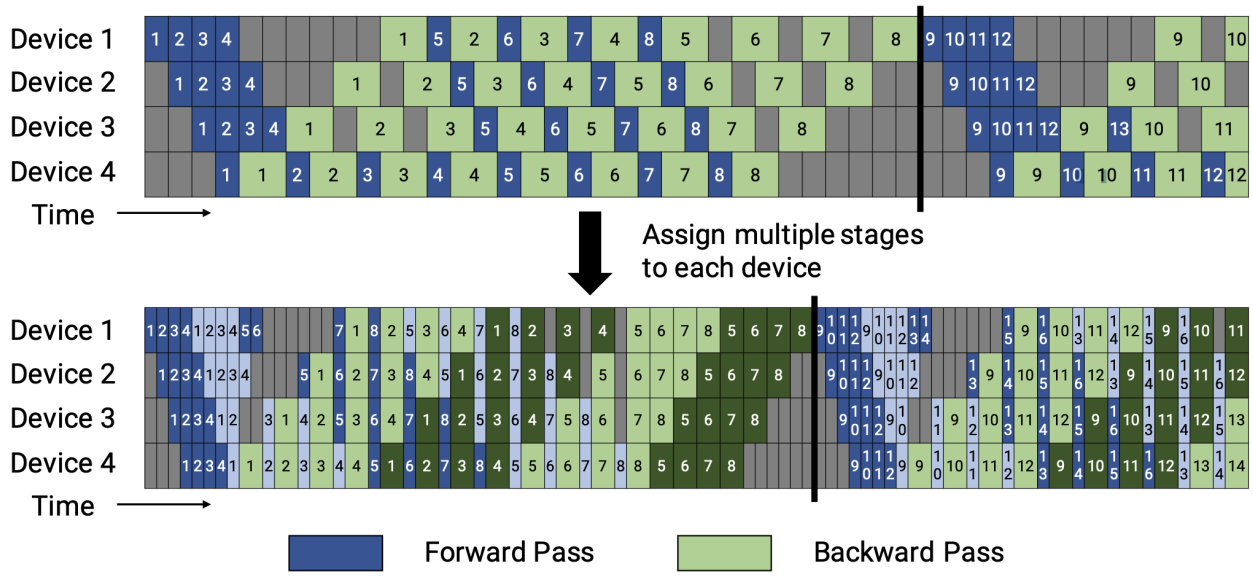
where m is the number of microbatches and n is the number of GPUs.

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.

8.1.3 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.



8.2 1F1B

One of the issues is that the model parameters keep changing while processing the forward passes. This means at every time step, minibatches are going to be forwarded through different weights. Thus, it is necessary to keep different states of the model parameters. Thus, 1F1B increases the memory requirements while increasing the processing speed.

8.2.1 Non-interleaved Schedule

The non-interleaved schedule can be divided into two states. The first state is the startup state (or warm-up state). In the startup state, After completing the forward pass for the first minibatch, it performs the backward pass for the same minibatch, and then starts alternating between performing forward and backward passes for subsequent minibatches. As the backward pass starts propagating to earlier stages in the pipeline, every stage starts alternating between forward and backward pass for different minibatches. As shown in the above figure, in the steady state, every machine is busy either doing the forward pass or backward pass for a minibatch.

8.2.2 Interleaved Schedule

This schedule requires the number of microbatches to be an integer multiple of the stage of pipeline. In this schedule, each device can perform computation for multiple subsets of layers(called a model chunk) instead of a single contiguous set of layers. *i.e.,* Before device 1 had layer 1-4; device 2 had layer 5-8; and so on. But now device 1 has layer 1,2,9,10; device 2 has layer 3,4,11,12; and so on. With this scheme, each device in the pipeline is assigned multiple pipeline stages and each pipeline stage has less computation. This mode is both memory-efficient and time-efficient.

8.3 Zero Bubble

Chapter 9

Tensor Parallelism

9.1 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:

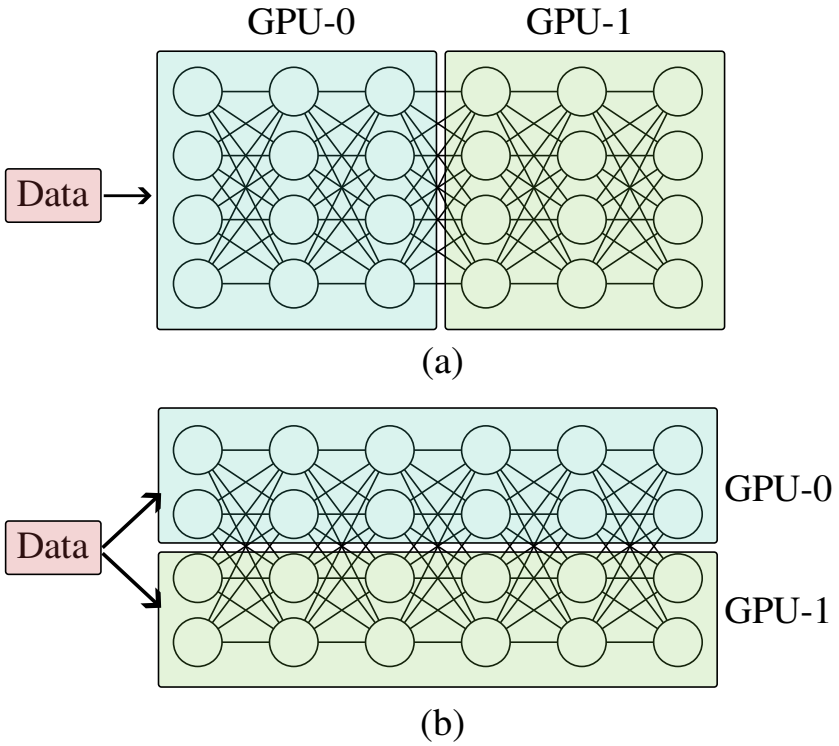


Figure 9.1: (a): Pipeline parallelism. (b) Tensor parallelism.

- 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.

In Megatron-LM, all Transformer layers, except normalization layer, are using row or column parallelism.

Tensor parallelism can be costly primarily due to the significant communication overhead involved when distributing large model layers across multiple GPUs, requiring frequent data exchange between devices which can become a bottleneck, especially when dealing with very large models and limited network bandwidth; this communication cost often outweighs the benefits of parallel computation, making it a major drawback of tensor parallelism.

Chapter 10

N-Dim Parallelism

Chapter 11

DualPipe

11.1 Introduction

11.1.1 All-to-All vs Point-to-Point

When orchestrating multiple GPUs, we need them to communicate with each other to share information like gradients and model parameters. There are two main types of communication patterns:

1. All-to-all communication.
2. Point-to-point communication.

All-to-all communication involves every GPU in the system simultaneously exchanging data with all other GPUs. The canonical analogy is a group chat where everyone needs to share their updates with everyone else. All-to-all communication is expensive and involves a ton of communication overhead. There are several clever algorithms like ring-AllReduce that can reduce this overhead, but it's still often a bottleneck.

Point-to-point communication, on the other hand, is a communication between just two GPUs (the analogy here is a private conversation). One GPU sends data directly to another specific GPU without involving the rest of the system. This is much more efficient in terms of network bandwidth and latency. In practice, point-to-point communication is strongly preferred when possible because it's significantly cheaper in terms of computational resources.

11.2 DualPipe

Finer-Grained stages: divide each chunk into 4 components:

- Attention,
- All-to-all dispatch(Handles communication between devices),
- MLP(Multi-Layer Perceptron),

Figure 11.1: An illustration of dualpipe.

- All-to-all combine(merge output across devices).

For a backward chunk, the attention and MLP split further into two parts: backward for input(B) and backward for weights(W) like Zero Bubble.

Bidirectional pipeline scheduling which feeds micro-batches from both ends of the pipeline simultaneously and a significant portion of communications can be fully overlapped (See the 2 black arrows in following diagram). In order to support bidirectional pipeline scheduling, DualPipe requires keeping two copies of the model parameters. If we have 8 devices with a 8 layers model, in the Zero Bubble Schedule, each device has a corresponding layer. But in the DualPipe Schedule, in order to handle bidirectional pipeline, the device 0 should have model's layer0 and layer7, and the device 7 should have model's layer7 and layer0.

Part VI

Transformers

11.3 Flash Attention

Chapter 12

Tokenization

Part VII

Compression

Chapter 13

Model Compression

13.1 Introduction

haha

Bibliography

[1] C. Brousseau and M. Sharp. *LLMs in Production: From Language Models to Successful Products*. Manning, 2025.

[2] Yiwei Li, Huaqin Zhao, Hanqi Jiang, Yi Pan, Zhengliang Liu, Zihao Wu, Peng Shu, Jie Tian, Tianze Yang, Shaochen Xu, Yanjun Lyu, Parker Blenk, Jacob Pence, Jason Rupram, Eliza Banu, Ninghao Liu, Linbing Wang, Wenzhan Song, Xiaoming Zhai, Kenan Song, Dajiang Zhu, Beiwen Li, Xianqiao Wang, and Tianming Liu. Large language models for manufacturing, 2024.