Abstract

With the rise of Large Language Models (LLMs) and Vision Based Large Neural Networks, the need of efficient computing and frameworks for Deep Neural Networks(DNNs) has risen rapidly over the past 2 years. In this project, we explore various strategies to efficiently train large scale neural networks across CPU and GPU heterogeneous platforms. The need of parallelization strategies becomes critical as the training time of these models is reaching upto 3 to 4 months. First, we conduct a in depth survey of existing works, where we briefly classify the existing parallelization strategies to Data, Model, Pipeline, Hybrid and CPU-GPU parallelization strategies. Next, we present a detailed analysis of the existing systems and techniques used in these machine learning systems to parallelize DNNs. Based on the survey, we combine and explore the possibilities of hybrid CPU and Pipeline GPU parallelization strategies. To achieve this, we propose 2 strategies Coarse Grained Parallelization, and Fine Grained Parallelization Strategies. We present the system model for the same and a few results to demonstrate the ongoing experiments of the proposed system model. Finally, we conclude the project work by setting the targets and expectations for the future work. Detailed study on the hardware and software systems have also been presented in this work.

Acknowledgements

I extend my heartfelt gratitude to Dr. Sathish Vadhiyar, Professor, Department of Computational and Data Sciences, Indian Institute of Science (IISc), for his exceptional guidance and unwavering support throughout this project.His expertise and mentorship have been instrumental in shaping the successful outcome of my endeavors.

I'm also grateful to Dr. Shikha Tripathi, Chairperson, Department of Electronics and Communication, PES University for her support in completing this project.

A heartfelt gratitude to Dr.Rashmi Seethur, Faculty Coordinator, Department of Electronics and Communication, PES University for her cooperation and support for successful completion of the project.

Finally, I my parents, my friends and to all the people who were directly or indirectly involved, for their cooperation and support for guiding and supporting me in completion of this project work.

Dheemanth R Joshi (PES1UG20EC059)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, Deep Neural Networks (DNNs) have proved to be significantly effective in automating and solving complex and large problems in the fields of enterprise, autonomous driving and various professional and user applications. As the complexity of the problems expected to be solved by DNNs (Usually complex datasets) increases, the number of parameters required to fit the data of the DNN increases. The number of trainable parameters has been growing rapidly in the recent past, making it challenging to meet the computational requirements to train these models. Latest large language models (LLMs) [4] scale up to hundreds of billions of parameters [5], which requires significant compute units and time (upto months) to achieve efficient training of the model. Parallelizing these models on CPU and GPU platforms helps to bring down the training time of these models significantly.

Extensive research has been conducted to explore and propose efficient parallelization strategies which reduce the training time of these neural networks. Majorly, these methods include Data Parallelism, Model Parallelism, Pipeline Parallelism and Hybrid Parallelism.

In Data Parallelism, model is replicated and spawned across multiple accelerators

(CPU/GPU). These accelerators can be termed as workers. The dataset is divided across these workers and each worker trains its model copy in parallel with other workers. The gradients generated during the backward pass by these workers are communicated with each other with various communication strategies to enable synchronization of the gradients. Finally, parameters of the model are updated with the synchronized gradients to complete one iteration of training.

Model Parallelism approach divides the neural network into multiple partitions. Each partition of the neural network is assigned to a worker in the system. The forward activations and backward gradients are communicated across the workers to train the model. This approach overcomes the memory constraints posed in data parallelism, which limits the model size to the size of single device memory and reduces synchronization significantly.

Pipeline Parallelism is an extended work of model parallelism. In this strategy, mini batches of the dataset are further divided into micro batches, and are pipelined across various stages of the model residing in different workers. This reduces the idleness incurred in model parallelism, in which only one worker stays active in a given time stamp.

Hybrid Parallelization strategies combine the above mentioned strategies to achieve improved latency and memory footprint. Popular hybrid strategies include 2D / 3D parallelization strategies. However, these strategies also require a significant amount of communication and synchronization.

The strategies mentioned above usually target GPUs. For few scenarios CPUs are preferred but because of limited computing capability, are usually are not kept in focus. However, few works utilize CPU's large memory to store a relatively large model and periodically transfer parameters to the GPU to compute the forward and

backward pass. Parameter update stage is usually scheduled on the CPU. We term this strategy as 'Hybrid CPU-GPU parallelization' strategy.

In this project, we explore the possibility of combining Pipeline Parallelization strategy with CPU-GPU parallelization strategy. Unlike existing works [], this work tries to exploit the dual socket CPU infrastructure to parallelize pipelining and training on CPU GPU heterogeneous platforms. For this project, we present the idea of coarse grained and fine grained parallelization strategy. We first make 2 copies of the model for training on CPU and a group of GPUs. Then we launch multiple processes to parallelize training on CPU and GPUs. Finally, we synchronize the gradients in coarse grained/fine grained fashion to obtain the trained model. We demonstrate partial experiments and results of our work in this report.

Following are our contributions for this project:

1) The idea of exploiting CPU for training alongside pipeline GPU group.

2) Proposal of coarse grained and fine grained parallelization strategies.

3) Synchronization strategy for seamless integration with PyTorch

4) Partial results for demonstration.

Chapter 2

Hardware Platforms

2.1 CPU Architecture and Programming

CPU (Central Processing Unit) is used in neural network training phase for variety of tasks. Currently, CPUs are basically used for their large main memory (RAM). They handle communication between processes and coordinate with the accelerators. However, with he increasing integration of AI accelerators on the CPUs, researchers harness their power for training and inference.

2.1.1 CPU Architecture

With a deep learning perspective, we look at the CPU architecture as a distributed memory multiprocessor. Where the main memory can be accesses by multiple workers (CPU cores). We also call them execution units(EUs). Each EU has a private memory, which is its personal memory in CPU, the L1 cache can generally be mapped to the private memory. All the execution units share a shared memory, which is used as a common space for multiple processes to read/write. This can be mapped to L2/L3 cache. Finally, there is a global memory, usually a RAM, in which all the

groups of execution units can read write for their respective processes. See Figure 2.1 demonstrating CPU accelerator.

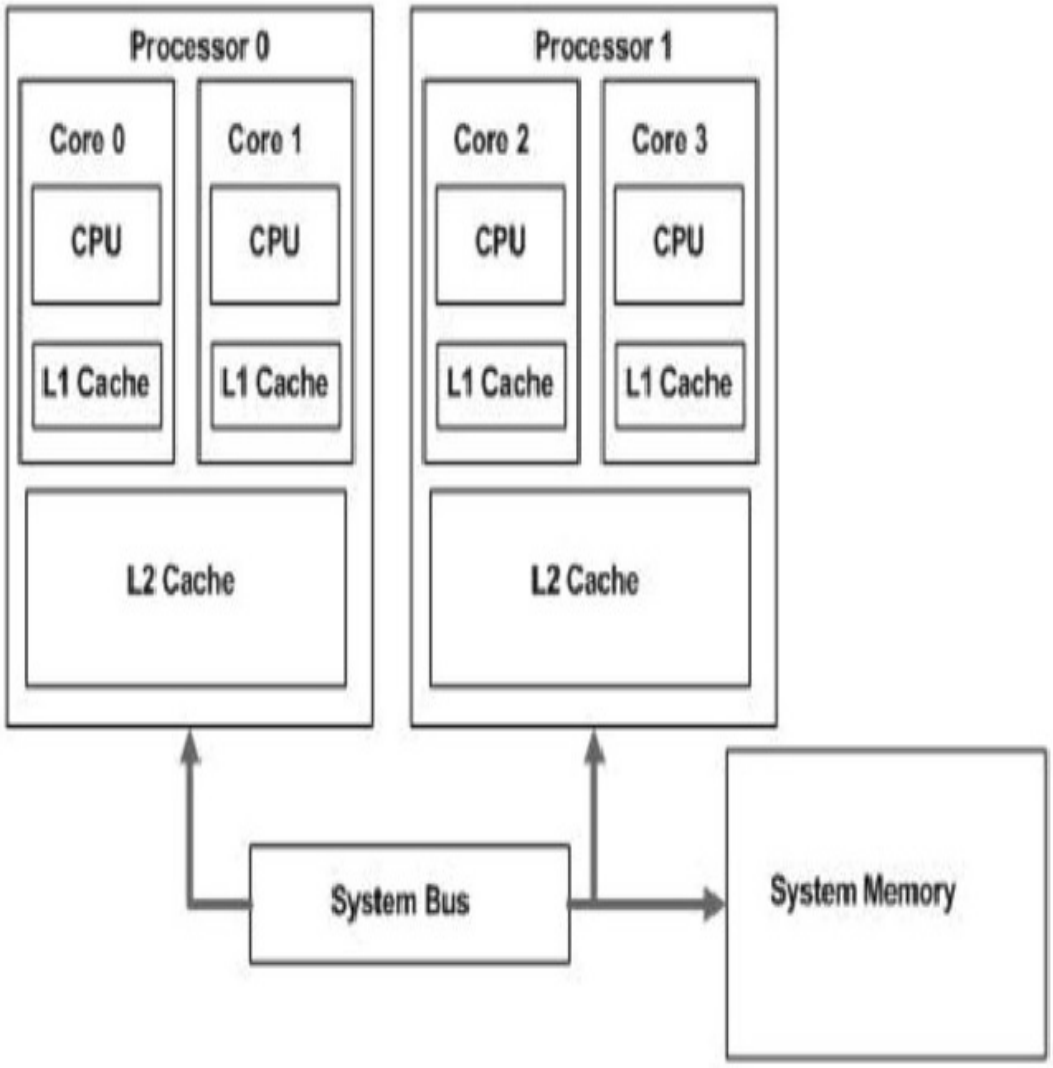

Figure 2.1: CPU accelerator, Courtesy: ecomputertips

2.1.2 CPU Programming

We use traditional programming stack for CPU programming, however (Python - C++ - Instructions), there are some optimizations and custom instruction sets to target the CPU mounted accelerators for example Intel uses Vector Neural Network

Instruction Sets (VNNI) to target matrix multiplication and vector engines to accelerate deep learning routines.

Accelerating CPU using Shared Memory Multiprocessing: OpenMP

OpenMP is a open source, a C/C++ tool which comprises of set of primitives and directives for the compiler to compile the given code in parallel process fashion. OpenMP is a shared memory system programming environment, which implies that all the processes can share a common ground on memory.

2.2 GPU Architecture and Programming

GPUs have gained a huge popularity amid deep learning break through. With the ability to perform at scale parallelism, GPUs are a go to solution for modern day large scale learning. GPUs are comprised of thousands of CUDA cores, each grouped to belong to a particular block. Each cuda core, may be a bit backward in terms of options of instructions, but is a perfect choice for large scale multiplication and additions.

2.3 GPU Architecture and Programming

The GPU architecture in generalised terms is quite similar to that of what we defined as in the CPU architecture. The main difference is the number of cores GPU holds and the attribute limitation of each core. An example of the GPU unit can be seen in Figure 2.2

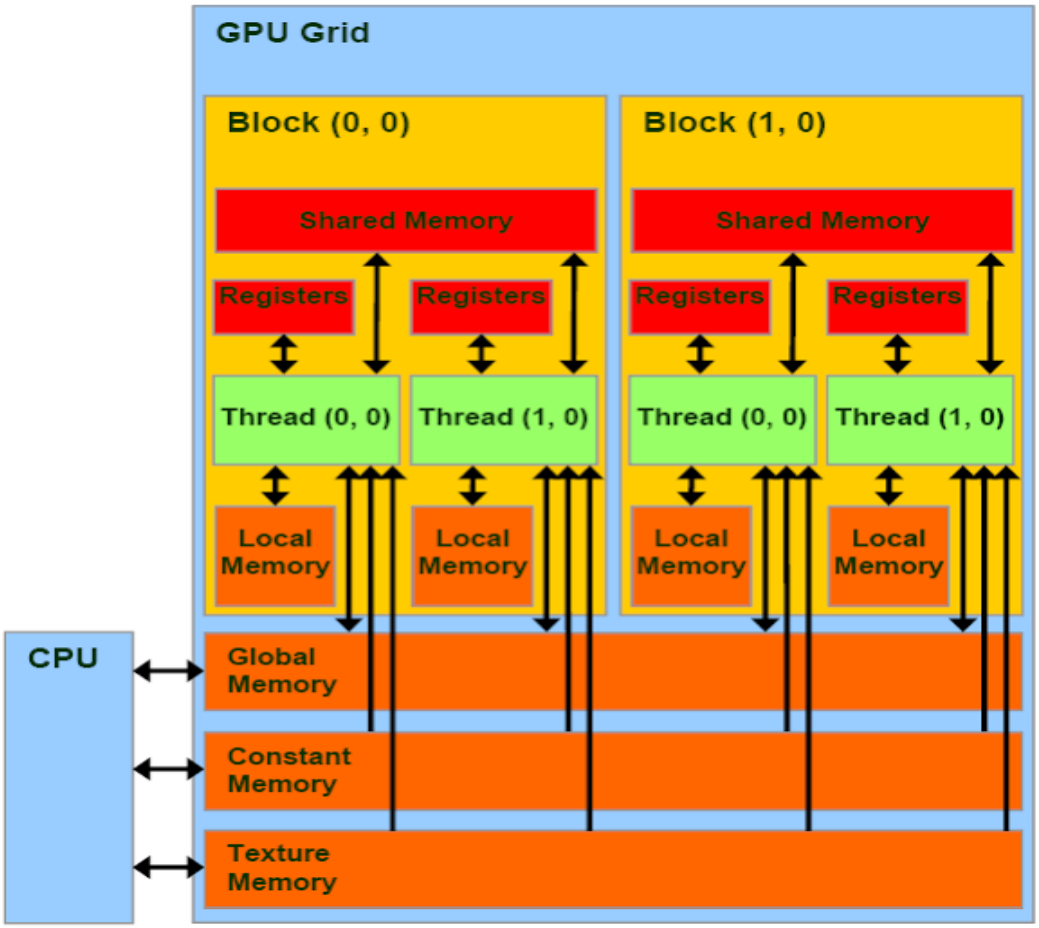

Figure 2.2: GPU accelerator, Courtesy: selkie

2.3.1 GPU Programming

for programming in GPU CUDA language (a C extension) is used. CPU in this case is treated as the host and GPU is called as an accelerator device. Programs for host and device are written and compiled separately. Host manages the code launch control over the GPU, the code is termed as a kernel.

Kernel and thread

a kernel code is written to launch the routines on the GPU. Kernel with set of directives is mapped as one thread on the GPU to one CUDA core. The number of threads to be launched can be controlled by the user during the compile time.

Blocks, Kernel Grid

CUDA imagines the cuda cores as points in the Cartesian space of computing. Where each core corresponds to a single location with a unique location across compute dimensions. Each dimension is divided to multiple blocks, which basically is introduced to bring in locality and controllability to kernels as a function of location. Groups of blocks which make up the entire compute space is known as Kernel Grid.

Chapter 3

Deep Neural Network Training Procedure

In this chapter, we will discuss the basic terminologies and fundamental ideas of deep neural network training. For ease of understanding of upcoming chapters, we will be using the same terminology used in this chapter. The upcoming sections will comprise of various terminologies and their significance in neural network training. Furthermore, we use PyTroch framework to build/test our strategies therefore, most terms/definitions used in this chapter will correlate with the pytorch implementation.

3.1 Dataset

A simplified definition of a dataset is a look up table, containing an input and the corresponding output (also known as the target) for a given input. Input and target can be any representation of data including but not limited to images, speech, text etc...

3.2 Deep Neural Network (DNN)

We define a deep neural network (DNN) as a graph of linear algebra operations. A DNN consists of an input layer, output layer and some hidden layers. These layers consist of learnable parameters (weights, biases, embeddings, attention scores etc...). All the layers defined in a DNN are heterogeneous i.e they need not be of same architecture. Most of the neural network operations deduce to matrix multiplications. To train a neural network, there are majorly three steps Forward Pass, Backward Pass and Parameter Update. We will discuss these processes and the attributes involved in these attributes in detail in later sections.

3.3 Forward Pass

This process involves passing the input of the neural network through the entire neural network graph once. The term "pass" refers to map the input X to output Y through some operator $*$ with the weight matrix W.

$$Y = W * X \tag{3.1}$$

every time the input is passed through a layer the output also known as activations of the neural network are passed through an activation function to yield the final activations of that layer. These activations are stored for backward pass in the compute node of the graph and it also becomes the input to the next layer. Once all the activations are computed the activations of the last layer are compared with the target to measure an average incorrectness of the model. This measure is often termed as loss. Loss is used to tune the parameters of the model to be as correct as possible.

3.4 Backward Pass

Backward pass involves passing the entire task graph backwards to compute the gradients (sensitivity of the parameter with the loss). Loss is kept as the metric to find the gradients. Gradients are computed using a recursive algorithm called back propagation. To compute the gradients, the gradients of the previous layer and the activations computed during the forward pass are required. once we have the gradients we update the model parameters to new state.

3.5 Optimizers and Parameter Update

Once the gradients of the layer are computed we need to update the parameters with the new gradients. Optimizers formulate an equation involving model's current parameters and the computed gradients. The result of the equation is the updated set of parameters, which has learnt the trends of passed input. Advanced optimizers have been invented to achieve a smooth and optimal convergence with the target. Some of these optimizers also include optimizer states, which usually have the same dimensions of the layers of the model. Therefore, optimizers also consume significant amount of memory.

3.6 Mini Batches

Passing the entire dataset for training the model can be compute intensive and may not fit the data optimally. Therefore, the dataset is divided into mini batches (chunks of input - output pairs) and each forward pass, backward pass and parameter update processes are performed on each of the mini batch. This ensures optimal fitting of the data and makes the compute space relatively less dense.

3.7 Training Iteration

Training iteration is defined as performing a single forward pass, backward pass and parameter update for a given mini batch of the data.

3.8 Training Epoch

Training process is said to have completed one epoch once it completes training iterations for all the mini batches. In other words, training process has completed one epoch once it has seen the entire dataset once.

Chapter 4

Literature Review

Our literature study majorly focused on parallel and distributed training, in which High Performance Computing Systems are used to train large scale deep neural networks to reduce latency and improve compute efficiency. This is majorly achieved through distributing and parallelizing the DNNs across multiple nodes consisting of High Performance CPU and GPU cores.

We conducted a detailed study on existing parallelization strategies. These strategies include Data Parallelism, Model Parallelism, Pipeline Parallelism, Hybrid Parallelism and CPU-GPU parallelism.

These strategies were found to be most efficient in terms of compute utilization and memory footprint. In the following sections we will go through each parallelization strategy in detail, including various methods and improvisations built on top of these strategies. An overview of our understanding of the literature is demonstrated in Figure 4.1

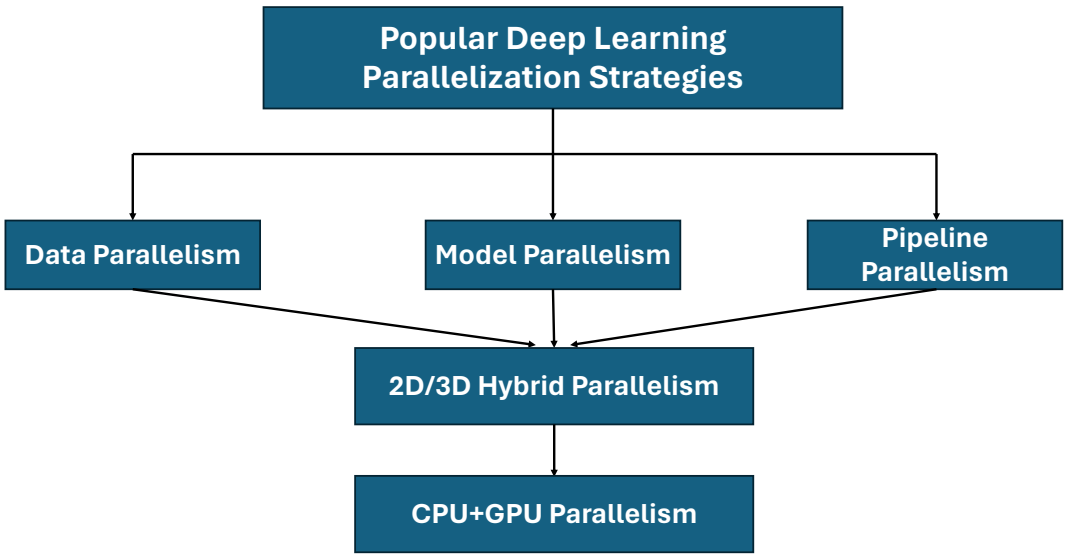

Figure 4.1: An overview of the literature survey

4.1 Data Parallelism

Data Parallelism is a method of training large scale neural networks on large datasets. It involves a process of creating copies of the neural network and spawning it across multiple nodes / devices. Each node / device is assigned a batch of the dataset. All the nodes / devices train their copy of the model in parallel. Assigned batches are further divided into mini batches to perform training iterations. The nodes / devices further periodically communicate with each other to synchronize the gradients generated by the networks during the backward pass.

Various works have been proposed to improvise the communication and synchronization overhead. Following are some implementations of data parallelism

4.1.1 Distributed Data Parallel (DDP) Strategy

Distributed data parallel [6] overcomes the issue of compute and communication bottlenecks caused by the vanilla all reduce strategy. It employs ring all reduce strategy,

to communicate gradients only with their neighbors while parallely computing the training iterations. This eliminates the aggregator and loads are balanced across all the devices. DDP implementation is provided by PyTorch library.

Gradient Bucketing

[6] argues that communicating small, individual tensors through the NCCL / GLOO is inefficient compared to the efficiency acquired when large matrices are sent through the channels. therefore authors in [] propose a gradient bucketing strategy, in which gradients are bucketed through few layers and sent at once to achieve higher communication efficiency. This is implemented by inserting pytorch hooks to each layer in the model.

4.1.2 Parameter Server Approach

unlike DDP strategy which is synchronous, (devices communicate with each other to acquire gradients) the parameter server approach is asynchronous, where a single node/device holds the parameters of the global model. workers periodically communicate only with the server to read and write the latest trained parameters of the model. This approach provides robust fault tolerance due to its asynchronicity, but fails to scale for larger number of devices due to communication bottleneck.

4.2 Model Parallelism

Model Parallelism was introduced in the community to overcome the issue of growing size of models. With the number of parameters rapidly increasing, issues occurred to fit an entire model on a single device. Model parallelism overcomes this issue by splitting the model to various partitions, and placing each partition on a different

device. The activations and the gradients obtained during forward and backward passes are communicated between devices to continue their part of the work.

Though Model parallelism solves the problem of large model sizes, it faces a major issue of lack of inefficiency of hardware resources. As can be observed from figure [], only one device actively performs its tasks for a given time instance. This impacts the throughput of the system.

4.3 Pipeline Parallelism

Pipeline parallelism was introduced as a improved version of model parallelism. In pipeline parallelism, the model is partitioned into different pipeline stages, where each stage is assigned to a single device. The devices communicate with each other to share gradients and activations like in the model parallel approach. Unlike model parallelism, the minibatches used for training the model are further divided into microbatches, and these microbatches are pipelined through the defined stages. This approach ensures that the majority of the devices stay active during the compute phase and throughput is maximized.

There has been a significant progress in the process of parallelizing the model through pipelining. Some of them are listed below.

4.3.1 Gpipe Implementation

Gpipe [7] implements pipeline parallelism in its vanilla form, where microbatches are pipelined through multiple stages to perform the forward pass, later the pipeline continues from the back to perform the backward pass. Once all the microbatches complete the two way pass, all the stages parallely update their parameters through the respective optimizers.

Activation Recompute Strategy

Gpipe introduces a activation recompute strategy, in which the activations of intermediate layers are initially discarded during the forward pass. Later, are recomputed during the backward pass. This ensures low memory footprint/memory efficiency.

4.3.2 Hippie Implementation

Hippie [2] Introduces the last stage schedule method, which alternatively performs forward and backward passes at the last stage of the pipeline. In addition to this, the gradient communication and parameter update between the stages are scheduled when the devices are bubbled. This ensures that computation overlaps with communication. Authors of this paper also introduce custom memory efficiency relation which takes memory foot print into account. Figure 4.2 represents the last stage schedule from Hippie work.

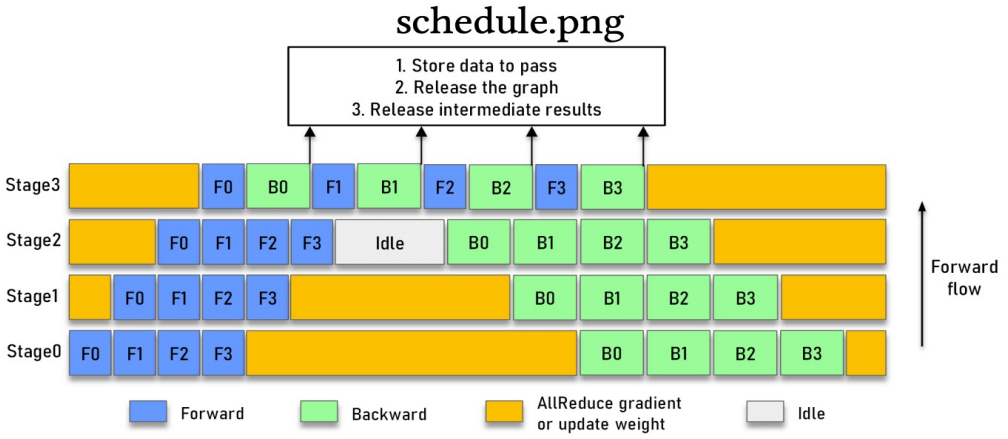

Figure 4.2: Illustration of last stage schedule in Hippie [2]

4.4 Hybrid Parallelization strategies

Hybrid Parallelization strategies tend to combine Data, Model and Pipeline parallelism to effectively leverage each parallelization strategies, while simultaneously hiding the drawbacks of each other. [8] is one of the works which implement hybrid parallelization strategies.

4.5 CPU - GPU parallelism

Data parallelism has proved to be the most efficient technique due to its simple implementation and its effective nature to enable overlap of computation with communication. However, due to limited device memory, this technique usually gets second thoughts to get picked. Various techniques have been proposed to overcome this issue. One of the most noted works proposed to store the entire model on the CPU memory (see figure 4.3). Due to CPU's large RAM, the entire model can fit into it, and the GPU becomes the heavy computer, where it periodically picks the parameters from the CPU and performs compute intensive tasks like forward and backward passes and CPU handles the communication and other light weight tasks.

4.5.1 ZeRO-Offload [1]

This work stores the entire model, including the optimizer states on the CPU. The parameters are periodically transferred to the GPU for forward passes and gradient computation. To bring in CPU to the work, authors propose to perform parameter update stage on the CPU. To assure that CPU doesn't become the bottleneck, authors implement 1-bit Adam optimizer, which effectively uses multiple CPU cores to compute the optimizer states and parameter update stages. This also ensures high

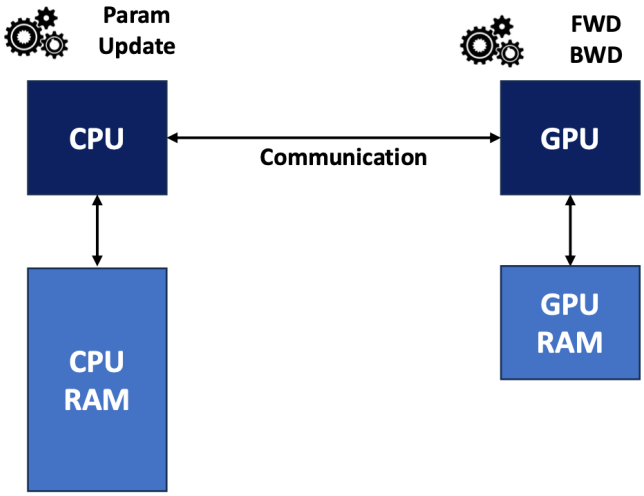

Figure 4.3: CPU GPU Parallelization Strategy

throughput and anti bottleneck strategy. Some memory optimizations are performed for better throughput [9].

4.5.2 CoTrain Implementation

Authors of this work [3] build atop ZeRO offload, where they argue that offloading parameter update kernels of all the layers to the CPU could cause a compute bottleneck leading GPU to stall in successive iterations. To this sovle this issue, authors proposed a dynamic scheduling strategy, where the parameter update task of initial layers were scheduled on the GPU post backward pass. This ensured that the memory utilization in the accelerator was high and the load was balanced between CPU and GPU. This was achieved through calculating the reuse distance between CPU and GPU every iteration. Constraints assured that CPU and GPU finish all the tasks of the iteration at almost the same time. CoTrain implementation seamlessly integrated with the PyTorch framework. See figure 4.4 for better understanding of reuse distance

based scheduling.

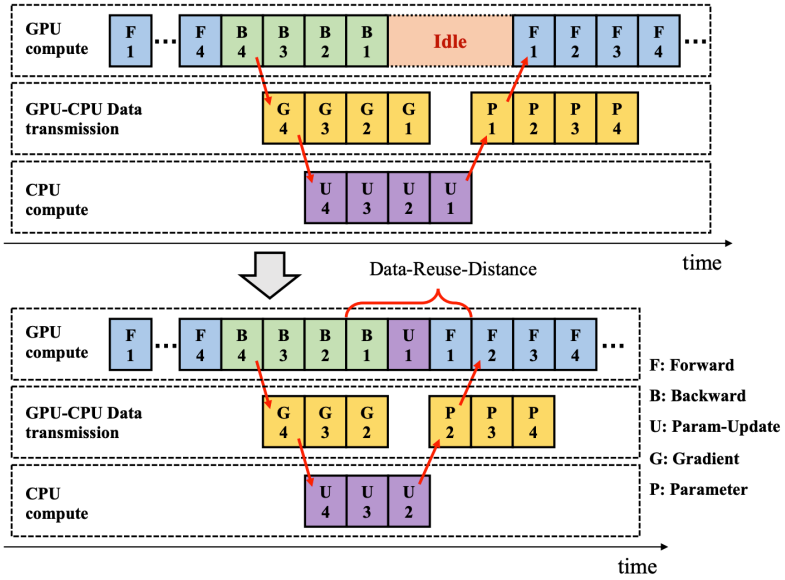

Figure 4.4: Overview of CoTrain [3]: Reuse Distance Based Scheduling on CPU-GPU

4.5.3 Hyscale GNN Approach

HyScale GNN [10] proposes to accelerate graph neural networks on CPU-GPU and CPU-FPGA platforms. Authors propose Dynamic resource management schemes, which adaptively resizes the batches of data assigned to CPU and accelerators. Unlike the previous works, this works leverages the CPU to train the GNN. This work also introduces a two stage prefetching, ensuring the overlap of computation and communication. further, customized hardware kernels are designed for efficient routing and placement of the FPGA.

Chapter 5

System Model and Implementation

In this work, we propose two parallelization strategies which employs pipeline GPUs and multicore CPUs to enable coarse grained and fine grained implementations to train neural networks. In upcoming sections, we will present the concept of coarse grained and fine grained parallelization strategies.

5.1 Coarse Grained Parallelization Strategy

This strategy tries to parallelize deep neural networks in its vanilla form, i.e, the 2 processes running on CPU and group of GPUs each will train a batch of data for one iteration, gradients generated by the processes are synchronized and the system proceeds to next iteration. the group of GPUs construct to form multiple stages of the pipeline parallelism process, and CPUs leverage multiple threads to accelerate the training routines. System illustrated in Figure 5.1

5.2 Fine Grained Parallelization Strategy

This strategy aims to hide certain stages of training routines by parallelizing them on the system platform. This simply implies that we try to parallelize the pipelined

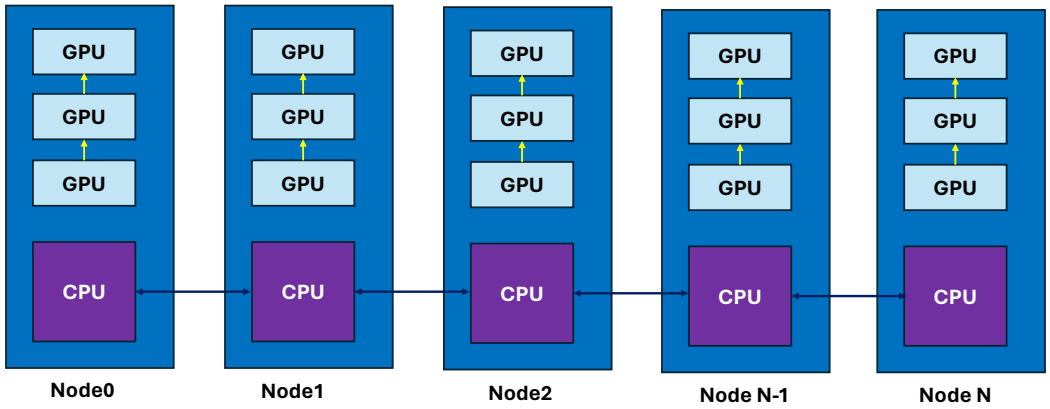

Figure 5.1: System Model: Coarse Grained Parallelization Strategy

GPU's backward pass with the parameter update stage of the training task graph. Unlike the work in [3] where the parts of entire model's state were communicating between CPU and GPU, we try to leverage pipelining strategies, through which only gradient communication takes place between the CPU and GPU (entire model communication is not required). See Figure 5.2

5.3 Parallelization Methods

5.3.1 Multiprocessing

One method is to achieve parallelism by launching multiple processes and assigning the process its relevant task. This is achieved by python's multiprocessing module, where multiple python interpreters are launched on multiple cores of the CPU. Each process assigned with a dedicated task, we achieve CPU GPU parallelization.

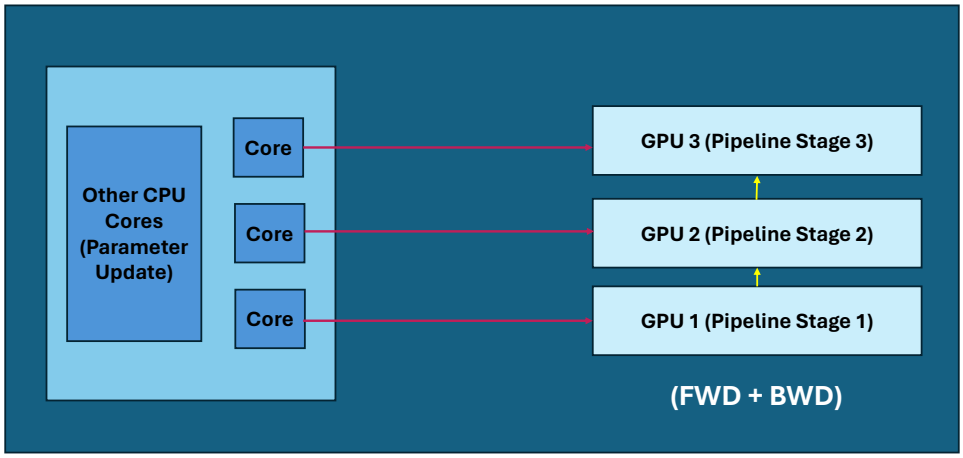

Figure 5.2: System Model: Fine Grained Parallelization Strategy

5.3.2 CUDA Streams

PyTorch has a built in cuda stream context manager, through which asynchronous GPU tasks are launched on the CPU and GPU. This implies, that CPU launches the GPU routine, and continues to execute its task on the CPU core.

5.4 Communication Strategies

5.4.1 NCCL / GLOO

NCCL (Nvidia Collective Communications Library) is an wrapper around native MPI for inter process communication. With NCCL, GPUs directly communicate with each other without CPU becoming the bottleneck. GLOO is a standard inter process communication library used between CPU to CPU.

5.4.2 Multiprocessing Communication Strategies

5.4.3 Process Manager

Through multiprocessing's manager class, data structures specific to deep learning (tensors) can easily be communicated with a common shared memory data base. These data structures include lists, custom objects and particularly dictionaries (Hash Maps).

Chapter 6

Experimental Results

We use vision based neural network (Multi layer Perceptron (MLP)) and GPT based neural networks (decoder only Transformers) to test our system model. For this project, we just demonstrate the parallelization results. Synchronization implementation is kept under progress and will be presented as future work. Additionally, we will be experimenting our system on a 16 core Intel i7 core processor and Nvidia Geforce GTX 1650 GPU.

6.1 MLP Experiments

The model was implemented with MLP. Hyperparameters are provided in table 6.1. We can observe that loss is gradually decreasing in the loss graph for both CPU and GPU.

6.1.1 Implementation

We launched two processes each process training one network each on respective CPU cores and GPU respectively. We use CUDA streams to asynchronously offload and retrieve gradients and data from the GPU. CPU parallely computes the gradients.

Table 6.1: Hyperparameters of the MLP

Parameter	**Value**
Optimizer : ADAM(lr)	(0.001)
Epochs	100
Dataset	MNIST
Dataset Size	60000
Minibatch size	100

The loss convergence of the 2 processes (CPU and GPU Process) is demonstrated in Figure 6.1 and Figure 6.2 respectively.

6.2 GPT (Transformer Neural Networks)

Tests were run regarding the inference time (for forward pass analysis) for decoder only models. Following were the architecture details and hyperparameters represented in table 6.2

Table 6.2: Attributes of the decoder only model

Parameter	**Value**
Number of Attention Heads	1
Steps	100
Dataset	Raw
vocab size	10000
Context Length	1000
dropout	0.2
number of layers	10

We ran some tests with the above hyperparameters and layers of the model on the

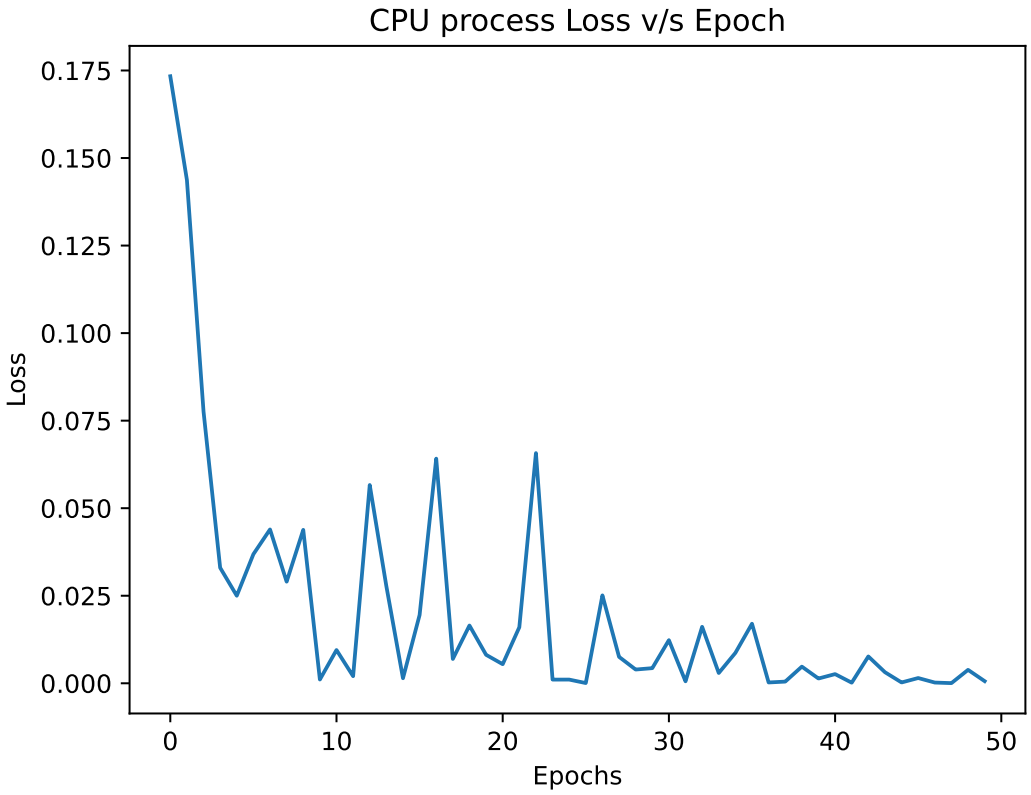

Figure 6.1: Epochs v/s Loss on CPU

mentioned hardware platform. Inference times have been recorded and demonstrated in table 6.2

6.2.1 Implementation

Two processes were launched to run the model parallely on CPU and GPU. Dataset were immediately moved/created on the accelerator device. This avoided the need of consistently moving the dataset from the CPU to GPU. Table 6.3 displays the inference time taken for forward pass.

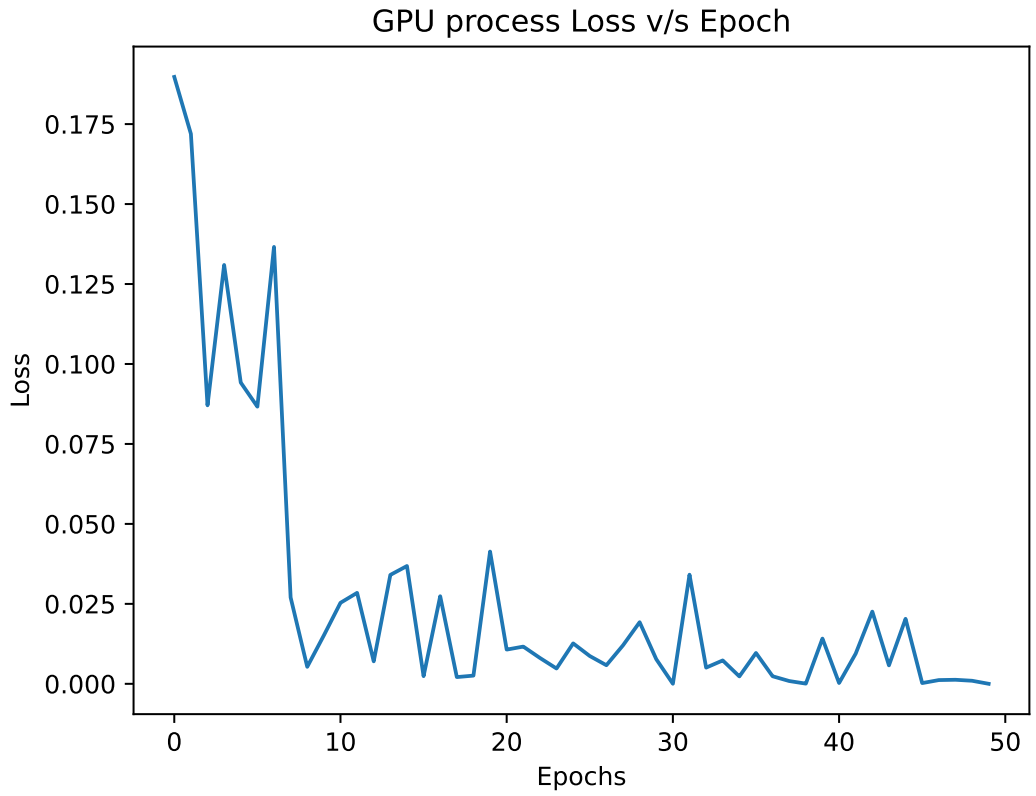

Figure 6.2: Epochs v/s Loss on GPU

Table 6.3: Inference Time of the GPT based model on CPU and GPU platform

Iteration	**Inference Time CPU (ms)**	**Inference Time GPU (ms)**
1	2294.58	654.86
2	2196.16	431.97
3	2132.47	138.63
4	2135.54	88.55
5	2144.50	60.75
6	2150.73	57.23
7	2132.31	66.65
8	2127.77	62.63
9	2129.92	64.13
10	2493.66	98.29

Chapter 7

Conclusion and Future Work

In this work, we proposed to parallelize large scale deep neural networks on CPU GPU heterogeneous platforms. Extensive literature review was conducted to understand the existing works, which mainly consisted of Data Parallelism, Model Parallelism and Pipeline Parallelism. CPU GPU parallelization strategies also was considered. We tried to combine CPU parallelization with pipeline parallelism strategies, To achieve this, we proposed two main strategies 1) Coarse Grained Strategies 2) Fine grained Strategies. We also demonstrated few training and inference experiments we performed to optimally move forward.

Next, we plan to come up with a new synchronization strategy to aggregate model parameters of CPU and GPU. We are also planning to scale parameters upto tens of billion parameter model. We will also be working on sparse mixture of experts (MoE) based neural networks and we will be trying to parallelize them.

Appendix A

7.1 PyTorch Framework

PyTorch is a open source deep learning framework to create, train and deploy deep neural networks platforms. Most of the libraries written in pytorch is implemented through C++. PyTorch also provides CUDA support to perform complex linear algebra operations like vector addition and matrix multiplication on Nvidia GPUs.

Figure 7.1 illustrates the PyTorch framework and its coordination with C++ and CUDA libraries.

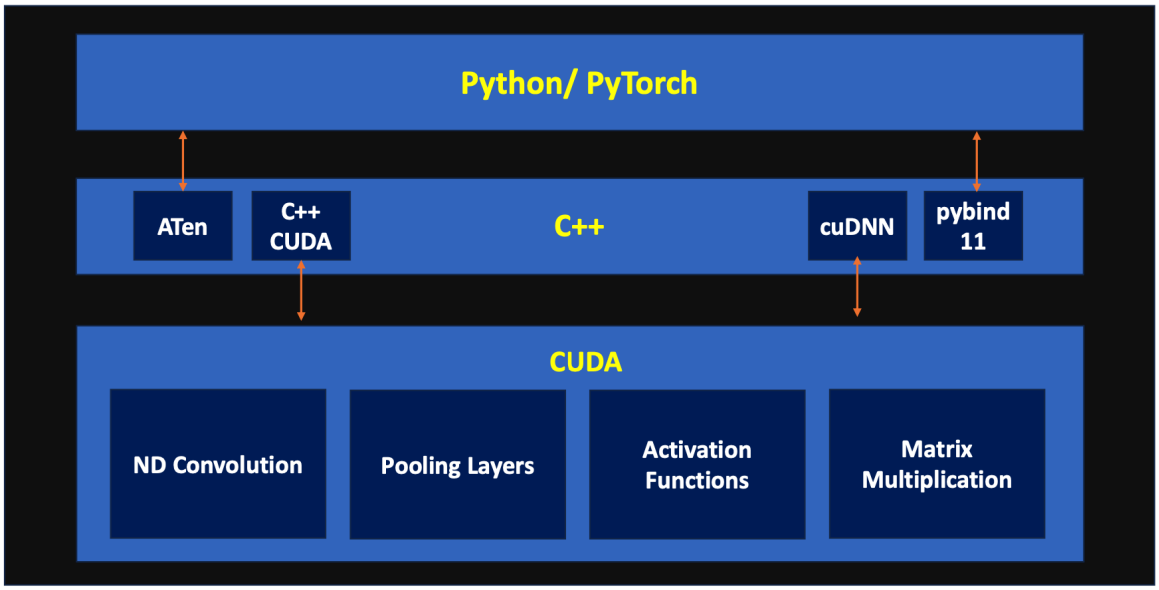

Figure 7.1: Overview of Pytorch Framework interaction with C++ and CUDA software stack

The pytorch framework creates a task graph of the defined neural network archi-

tecture. All the modules/network blocks inherit from the base class of "nn.Module". This hybrid architecture of he module and graphs enables seamless integration of models enabling higher developer efficiency.

7.1.1 nn.Module Class

base class consisting of basic neural network operations. Various complex architectures like LSTMS and Transformers inherit various modules defined by nn.Module and its children.

Forward Method and task graph generation

when the forward method of the modules is defined, pytorch creates a task graph of the neural network, in which each node consists of modules and their methods. This graph further generates a backward graph, enabling back propagation algorithm to efficiently pass through the task graph.

PyTorch framework provides wide area of frameworks which we use to train our models. We leverage distributed training routines to parallelize model training some of them are mentioned below.

7.1.2 Distributed Library

PyTorch presents a distributed library, which handles complex multiprocessing and communication optimized for deep neural networks pytorch DDP mentioned in chapter 4. Distributed library mainly manages the processes through process groups. Inter process communication and multiprocessing are handled by NCCL/GLOO.

7.1.3 ATen Library

ATen is a C++ implementation of PyTorch. Through ATen, developers can leverage the the speed of C++, to accelerate the math behind deep learning. All the linear algebra operations can be defined in the C++ interface, including CUDA programming. Later these files are bonded with the python interface using pybind11.

7.1.4 Remote Procedure Call (RPC framework)

RPC framework enables to perform remote server communication scheme to enable distributed training across remote servers. This helps developers to split a large model across nodes/servers to enable distributed training / inference.

References

[1] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "Zero-offload: Democratizing billion-scale model training," 2021.

[2] X. Ye, Z. Lai, S. Li, L. Cai, D. Sun, L. Qiao, and D. Li, "Hippie: A data-paralleled pipeline approach to improve memory-efficiency and scalability for large dnn training," in *Proceedings of the 50th International Conference on Parallel Processing*, ser. ICPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472497

[3] Z. Li, Q. Cao, Y. Chen, and W. Yan, "Cotrain: Efficient scheduling for large-model training upon gpu and cpu in parallel," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 92–101. [Online]. Available: https://doi.org/10.1145/3605573.3605647

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

[5] X. Sun, W. Wang, S. Qiu, R. Yang, S. Huang, J. Xu, and Z. Wang, "Stronghold: Fast and affordable billion-scale deep learning model training," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–17.

[6] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," 2020.

[7] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," 2019.

[8] S. Singh and A. Bhatele, "Axonn: An asynchronous, message-driven parallel framework for extreme-scale deep learning," 2023.

[9] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," 2020.

[10] Y.-C. Lin and V. Prasanna, "Hyscale-gnn: A scalable hybrid gnn training system on single-node heterogeneous architecture," 2023.