ADAPT: Asymmetric Device Adaptive Parallel Training for Small-Scale Heterogeneous Systems

by

Rishab Kashyap

A Thesis
Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved February 2026 by the
Graduate Supervisory Committee

Aviral Shrivastava, Chair
Aman Arora
Nakul Gopalan

ARIZONA STATE UNIVERSITY
May 2026

ABSTRACT

The increasing computational requirements of Deep Learning models have outpaced the development of more powerful hardware. This has led to an increased dependence on multi-accelerator training parallelization, often in large data centers. However, this is not always accessible or cost-effective for smaller organizations or research teams that rely on '*Small-Scale Heterogeneous Systems*', such as workstations with a few Graphics Processing Units (GPUs) and a Central Processing Unit (CPU). These systems may consist of different GPUs in the same system and are often shared across multiple users, creating resource contention and thus creating an asymmetric computation environment. While some training parallelization techniques have recently been introduced, they either target data-center inference use-cases, only target GPUs, or require manual model partitioning from the user. Additionally, none of these techniques address real-time resource contention during training or include the CPU. To address these limitations, we introduce the ***A***symmetric ***D***evice ***A***daptive ***P***arallel ***T***raining (ADAPT) framework. ADAPT uses fixed and real-time metrics in conjunction with an analytical cost model to generate near-optimal model or dataset partitions for training models on heterogeneous and overloaded homogeneous computational systems. It provides a novel solution to train across general small-scale systems by allowing the use of multiple device types while simultaneously ensuring the fastest communication between them. ADAPT shows upto a 4× improvement in Pipeline Parallelism and Data Parallelism across a wide range of models on Heterogeneous and overloaded Homogeneous systems. Even on Homogeneous Systems with no resource contentions, ADAPT shows a 10-15% improvement in training over the default configurations generated by PyTorch.

DEDICATION

I would like to dedicate this thesis to my family and friends. I dedicate this thesis to my father for his immense support and patience, and my mother for her unconditional love and faith in my abilities. I dedicate this thesis to my sister for making me believe I can achieve anything when I put my mind to it. Most importantly, I dedicate my hardwork, patience, and determination I have had through the years of my research this to my grandfather, who has been my biggest inspiration to pursue this

ACKNOWLEDGMENTS

I would like to sincerely thank my parents for supporting me financially and emotionally, and my sister for giving me the strength to push through any hurdles I have gone through. I sincerely thank Professor Aviral Shrivastava for his patience in guiding me through this process, and his trust in me for being able to achieve what I could. I would like to thank Kaustubh Harapanahalli and Vinayak Sharma for their support in completing my thesis and research the right way. Last, but not the least, I sincerely thank Deeksha Prahlad and Pawan Kumar for supporting me through this tough period, for their encouragement greatly contributed to my success today. To all the people who have supported me, you have my immense gratitude to help me achieve my dream. Thank you!

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Modern Deep Learning (DL) models have been growing in terms of the number of parameters, which in turn increases the demand for higher computational capabilities. However, with the new devices being manufactured, it is still challenging to fit the largest deep learning models, such as GPT-3 and Llama 3.2, on a single device. This requires the training to be distributed across multiple devices, leading to the development of numerous parallel training strategies, such as Data Parallelism (DP), Model Parallelism, Pipeline Parallelism (PP), Tensor Parallelism, Sharded Parallelism, and hybrid techniques that combine all of these approaches. Deep learning frameworks, such as PyTorch (Paszke *et al.* (2019)) and TensorFlow (Abadi *et al.* (2015)), build their parallelism training strategies based on approaches like ZeRO (Rajbhandari *et al.* (2020)) and GPipe (Huang *et al.* (2019)). However, these techniques are primarily designed for homogeneous systems. They cannot account for real-world shared devices or asymmetric systems consisting of different types of devices, such as CPUs and GPUs with varying capabilities.

A key point of failure for existing distributed training implementations lies in their inability to effectively handle asymmetric compute systems. These systems, characterized by devices with varying compute capabilities (e.g., CPUs and GPUs) or fluctuating performance due to resource contention, present a significantly expanded search space for optimal workload splitting compared to homogeneous setups. For example, Figure 1.2 illustrates three common failure modes: (1) **Out of Memory (OOM) Errors**, where naive even splitting (e.g., 50GB per device) exceeds the capacity of smaller devices (e.g., 48GB GPUs) while larger resources (e.g., 128 GB CPU) remain

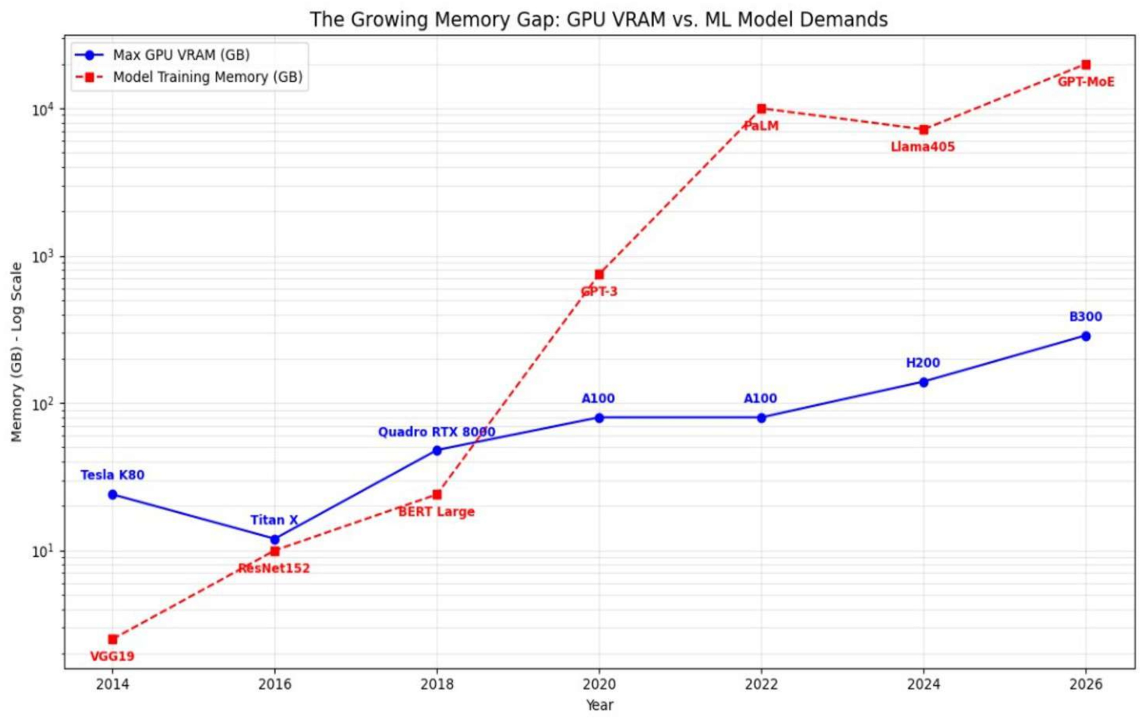


Figure 1.1: Motivation: Current hardware compute capabilities do not meet the criteria for large model training requirements. Multiple compute units are required. Small-scale systems are limited in the number of such devices that can be fit together on it.

idle; (2) **Asymmetric Compute Imbalance**, where uneven compute capabilities cause some devices to be over-utilized (100%) while others are underutilized (60% or 0%), creating compute, memory and communication bottlenecks; and (3) **Device Overloading**, where static assignments fail to account for existing load, pushing devices into OOM or high latency states. Current asymmetric-aware approaches, such as Whale (Jia *et al.* (2022)), primarily focus on memory-based load balancing, often neglecting critical factors like throughput and communication latency. Furthermore, these methods typically rely on static, compile-time configurations, failing to account for dynamic changes in resource availability or the potential to leverage underutilized resources like CPUs during training. They also lack the capability to automatically switch between communication protocols (e.g., NCCL vs. Gloo) based on the active

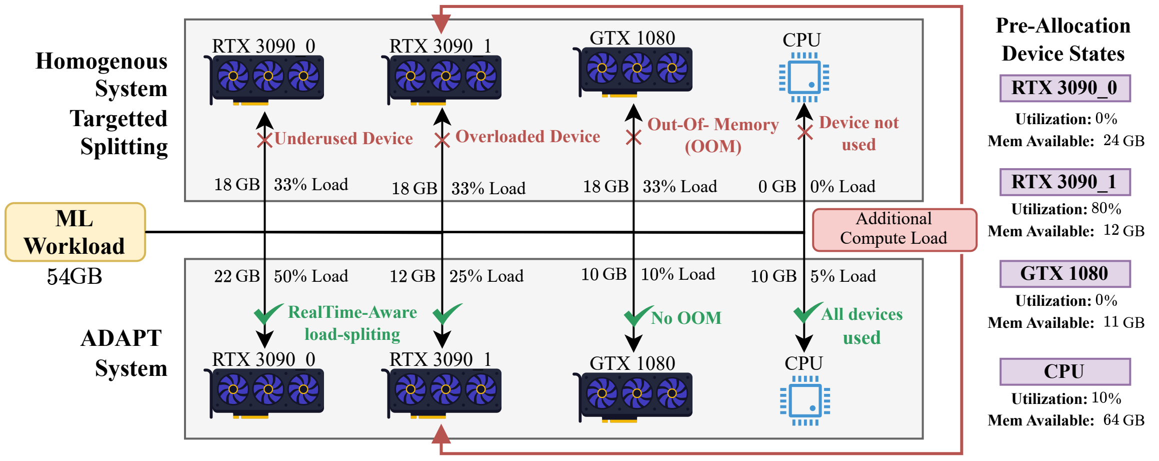


Figure 1.2: Motivation of the ADAPT System - Most systems would not be able to assign heterogeneous splits based on different device types and include CPUs and GPUs together. They might result in OOM errors or even bottlenecks due to overloading. ADAPT offers the solution by automatically assigning heterogeneous-aware splits at runtime

device topology, leading to suboptimal performance in mixed-device environments. Consequently, creating an automated, adaptable training framework for such diverse and dynamic systems remains a significant challenge. Even in the much simpler homogeneous device setting, there are few, if any, methods for automated model splitting for pipeline parallelism.

The following thesis work introduces ***A***symmetric ***D***evice ***A***daptive ***P***arallel ***T***raining (ADAPT), a novel framework to enable efficient training across asymmetric systems. ADAPT creates data and pipeline parallel training configurations for asymmetric systems with varying compute capabilities, and dynamically adjusts to changes in device performance due to resource contention or other real-world factors during training. ADAPT employs an analytical cost model for data and Pipeline Parallelism to efficiently operate based on real-time compute, memory, and communication characteristics of the devices in the system. This allows ADAPT to explore the search space of configurations efficiently. ADAPT also introduces heuristic optimization strategies to find optimal configurations efficiently. This allows ADAPT to dynamically adapt the training configuration during runtime as the performance characteristics of the devices change, with minimal overhead.

The cost function and optimizer allows ADAPT to improve the execution time across different small-scale systems. It ensures near-optimal load balancing based on different device types is achieved on Heterogeneous systems, granting more powerful device a larger share of the load, while reducing the effect of bottleneck devices. ADAPT can perform load balancing on an overloaded Homogeneous system, where multiple users using different devices can cause differences in GPU utilization in real time. It monitors these changes with active profiling and adjusts the training workload during runtime to ensure minimal execution time loss. ADAPT ensures every device inclusive of CPUs are considered when searching for the optimal execution strategy. To ensure the quickest communication media are maintained between devices, ADAPT also deploys a mixed communication strategy that can simultaneously use NCCL NVIDIA (2019) and GLOO PyTorch (2017) for GPU-GPU and CPU-GPU communications respectively, thus preventing communication bottlenecks.

The key contributions of this thesis can be summarized as:

1. We introduce the **ADAPT framework**, an automated and adaptive distributed training system designed to efficiently handle asymmetric environments. ADAPT *dynamically profiles and adjusts training configurations* to mitigate resource contention and leverage heterogeneous devices (e.g., CPUs and GPUs) effectively.

2. We implement a **novel analytical cost model for Data and Pipeline Parallelism** respectively that *incorporates real-time compute throughput, memory capacity, and communication latency.* This model enables precise workload

splitting and protocol selection (e.g., switching between NCCL and Gloo) to optimize performance across diverse hardware topologies.

3. We introduce **heuristic optimization strategies** that efficiently prune the vast search space of parallel configurations. These strategies allow *ADAPT to actively adapt to runtime performance fluctuations with minimal overhead*, ensuring sustained training efficiency even under varying load conditions.

We demonstrate that ADAPT achieves a reduction of up to 30% in training time in both homogeneous systems with resource contention and asymmetric systems, compared to state-of-the-art distributed training frameworks, across a range of hardware configurations. We also demonstrate that ADAPT can leverage previously underutilized CPU resources in workstations to accelerate training by up to 5% further. Finally, we illustrate the adaptability of ADAPT to actively update configurations based on additional workloads on devices during training.

Chapter 2

BACKGROUND

Three significant advantages of using distributed computing on heterogeneous devices would be **(i) Reduced execution time:** Since multiple compute units process different parts of the application concurrently, we can expect that the overall time taken for execution would be greatly reduced. **(ii) Maximized compute power:** Having a multi-GPU system to implement a machine learning application takes advantage of the compute power of multiple GPUs together to implement a large application that may not fit on a single GPU. **(iii) Distributed memory usage:** A single compute unit has limited memory for processing applications. Distributing the application across different devices eases the memory usage, thus ensuring smoother execution. This section provides context on different parallel training strategies and factors that influence how to implement them.

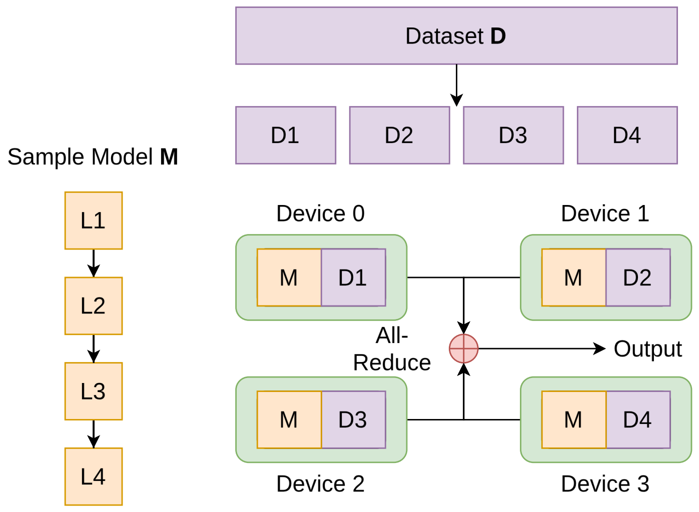


Figure 2.1: Data Parallelism across four devices with Ring Reduce Operation

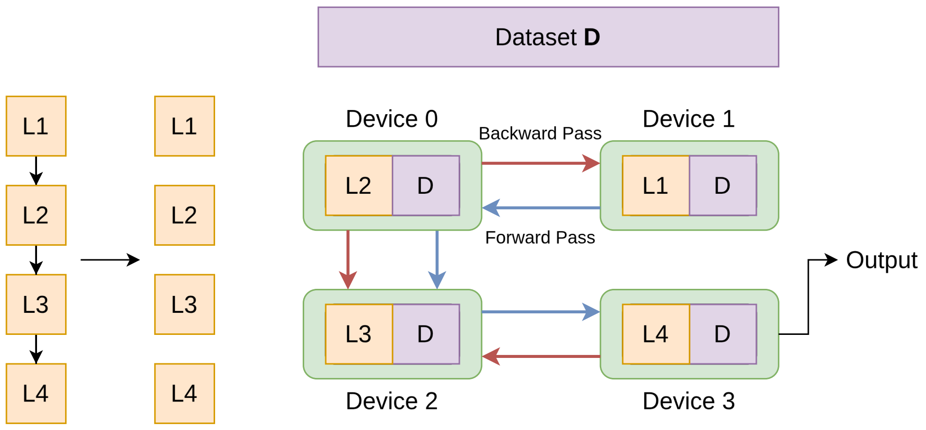


Figure 2.2: Pipeline Parallelism across four devices using a GPipe-based execution method

Data Parallelism

This is the most common form of parallelism, where a copy of the model is distributed across all the available devices of the heterogeneous system. The dataset is divided among these devices. This allows multiple batches of data to be trained simultaneously across all the devices, where each device holds a smaller amount of data. The gradients computed by each device are collected and averaged across all the models using the all-reduce operation from NVIDIA NCCL (NVIDIA (2019)) before applying the optimizer across all of them. This can be seen in Figure 2.1.

There is an exponential number of combinations of data splits to devices that can be used to execute data parallelism. To enumerate every possibility, assume a training dataset consisting of **N** rows, and the model is running on a heterogeneous system having **K** devices. In such a case, the number of ways to split the dataset across the different devices would be:

$$\# \; Splits \; = \binom{N+K-1}{K-1} \tag{2.1}$$

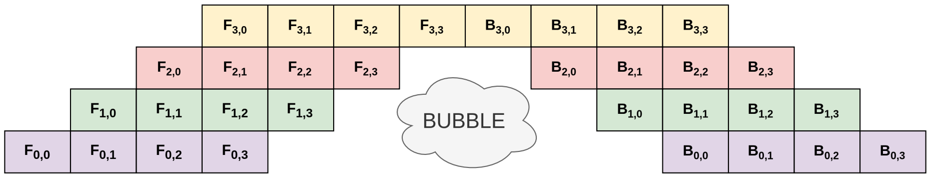

$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ BUBBLE $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$

Figure 2.3: GPipe execution strategy

Pipeline Parallelism

Pipeline parallelism splits the given model into different stages, each one mapped to a device. The dataset is fed as input to the device holding the first layer of the ML model. As shown in Figure 2.2, each stage is mapped to the respective device, which are interconnected based on the flow of the model graph, and the execution is pipelined per epoch. A single stage can consist of any number of layers. Hence, the enumeration formula remains the same as seen for data parallelism, but here, **N** is the number of layers of the model being trained.

Pipeline parallelism is a basic method of splitting the model into different sub-modules and assigning each one to a different device, as shown in Figure 2.2. This form of parallelism is used in cases where a large model, such as an LLM, cannot fit on a single device. Optimizing this strategy by allowing different batches to simultaneously execute on different devices can be done by introducing a pipelining effect. This ensures that no device is left idle for too long, thus maximizing device utilization. In this work, we utilize the GPipe Huang *et al.* (2019) execution strategy to implement pipeline parallelism, where each batch of data is divided into a number of micro-batches and sent in a pipelined fashion across the devices for the forward pass first, followed by the backward pass. This can be seen in Figure 2.3

The search space for different ways to assign these sub-modules or ”stages” to a unique set of given devices is exponential, and depends on the number of devices available to us. Furthermore, it is important to keep track of the execution order of these layers since the outputs processed on one device must go to the device that has the next set of contiguous layers of the chosen model. Hence, the ordering of devices matters as well. If D represents the set of available devices, and S represents the set of stages that the model is divided into, the total number of configurations that we can generate can be given as:

$$\sum_{d=1}^{min(len(S),len(D))} \binom{len(S)-1}{d-1} \cdot \binom{len(D)}{d} \cdot d!$$

where $d \in D$ represents the available device we take into consideration.

Optimizing Parallel Training

Choosing the correct parallel training strategy requires a clear understanding of the ML workload and the heterogeneous environment used to train the model. Many factors, such as the training latency, memory requirements, and resource utilization per device, need to be monitored to ensure smooth training. Frameworks such as Megatron-LM (Shoeybi *et al.* (2019)) train massive models over homogeneous systems but do not look into the utilization statistics to determine the number of devices that can be used for the system. Works such as Alpa (Zheng *et al.* (2022)) introduce a cost-based search strategy that allows them to estimate the optimal training setup. This allows for better results due to being well-informed about the system. Hence, it is critical to understand the heterogeneous system before performing distributed training for more optimal results.

Chapter 3

RELATED WORK

Homogeneous splitting is considered the safest option for parallel execution since identifying balanced split points is not a trivial task. Existing frameworks such as PyTorch (Paszke *et al.* (2019)) implement Distributed Data Parallel (DDP) for homogeneous data parallelism, and utilize an extended version of PiPPy (Reed *et al.* (2022)) for pipeline parallelism. Both methods automatically identify the number of participating devices and split their respective workload equally across them. This leads to diminished training time in the case of heterogeneous systems, and even more hazardous in homogeneous systems with either a defective or overloaded device included in the participating devices.

Alpa (Zheng *et al.* (2022)) is an automatic pipeline parallelism method implemented in JAX, which utilizes a latency-based cost function to minimize execution time. It identifies near-optimal split points across the model pipeline based on the available device clusters, employing a dynamic programming approach to infer statistics from model training and identify the optimal configuration. Though it provides a massive advantage over general homogeneous systems, it does not cover the premise of unequally partitioning the workload across a heterogeneous system effectively, and could sometimes be detrimental to the overall performance. It suffers from the same issue of not being able to identify when devices are overloaded or are in use in a homogeneous setup as well.

Whale (Jia *et al.* (2022)) implements a parallel training technique for heterogeneous GPU systems by using a memory-aware algorithm to identify usable devices. Here, GPUs with higher VRAM are given a major portion of the overall workload, thus allowing for irregular splitting for different parallelism types. However, it is limited to GPUs, and can misjudge in situations where an overloaded GPU, although more powerful, may be given lesser layers due to lesser memory. Furthermore, Whale utilizes a static splitting approach, where the spits are pre-determined before runtime. This limits the capability to detect a difference in system loads during runtime, thus affecting execution time.

ADAPT thus addresses the above shortcomings by deploying a method that maximizes system resource utilization on any general workstation-like system. ADAPT implements balanced splitting across heterogeneous devices and imbalanced homogeneous devices as well by constantly monitoring system resource availability. It ensures the inclusion of both GPUs and CPUs to execute the model training, while only selecting the devices that results in the fastest execution time. Furthermore, ADAPT enables real-time re-balancing by monitoring system resources throughout the training process to adapt to system changes. Optimizations are performed for Data and Pipeline Parallelism formats to test our hypothesis, where Data Parallelism follows the standard DDP (Paszke *et al.* (2019)) implementation, while pipelining follows the GPipe (Huang *et al.* (2019)) execution style.

Chapter 4

ADAPT ARCHITECTURE

ADAPT focuses on addressing the issues highlighted in chapter 3, by implementing a dynamic optimization strategy that accounts for compute and memory asymmetry at start-up and runtime resource contention during runtime process execution for asymmetric systems. This is powered by two analytical cost models for Data and Pipeline Parallelism for efficient search space exploration and optimization.

As outlined in Figure 4.1, ADAPT consists of 2 main stages:

1. **Initial System Start-Up Stage**: ADAPT probes an asymmetric system to discover all computational resources, and generates an initial training configuration for Data and Pipeline Parallelism.

2. **Runtime Stage**: ADAPT probes the asymmetric compute system during the execution of the model training, runs profiling-assisted optimization to account for resource contention and any other variability in terms of compute, memory, and communication, modifies the training configuration, and continues to train the model.

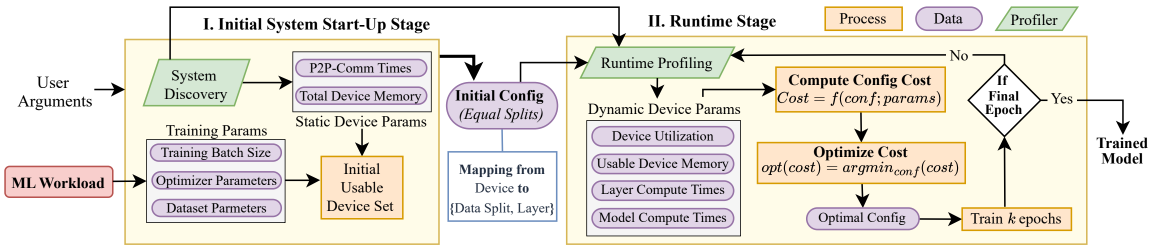


Figure 4.1: Our Proposed Methodology: Four input values are fed to the system. The First step utilizes a Benchmark Epoch to generate the first training configuration and all the profiling values. Based on the profiled values, it uses an informed search to generate the new training configuration. The training is resumed, and the cycle repeats.

Initial System Start-Up Stage

When a training process is triggered, ADAPT automatically probes the asymmetric system to discover all compute and memory resources, and their peer-to-peer (P2P) device communication latencies. The framework combines this information with the model and training hyperparameter details to create an initial training configuration. This initial configuration is the result of pruning devices that cannot accommodate the training workload in any capacity. For example, a device that cannot fit the model in memory is pruned for data parallelism. Similarly, a device that cannot fit a single layer of the model in memory is pruned for Pipeline Parallelism. ADAPT is flexible in accepting user-defined arguments, such as disregarding specific devices or setting a maximum number of devices to use. It is essential to note that all information collected during the discovery stage is static and does not require updates during training; therefore, the discovery stage is only run once.

Runtime Stage

During runtime, ADAPT considers the initial training configuration generated during the initial system startup stage. It then, during runtime, profiles all selected devices to capture real-time resource contention, differing compute capabilities, and pairs them with known communication latencies. An analytical cost model and optimizer then use this information to generate an optimal partition of the model or dataset. ADAPT then trains the model for a set number of epochs (k). If training reaches the final epoch, then the process is completed. Otherwise, ADAPT repeats the process from the runtime profiling stage to check for any changes in the system due to external processes. If changes exceeding a specific delta are detected, the training configuration is updated to reflect these changes.

Cost Model and Optimizer

ADAPT implements distinct optimizers and cost models for data and Pipeline Parallelism to account for the unique challenges of each parallelism type. However, both cost models are analytical, allowing the runtime to be a small fraction of the total training time. At a high level, the cost models are functions of the profiled values and the training hyperparameters and can be defined as:

$$Cost(config) = f(T_{comp}, T_{comm}, M_{usable}, \mathcal{H}) \tag{4.1}$$

Where, T_{comp} = Compute time, T_{comm} = Communication time, M_{usable} = Usable memory, $\mathcal{H}$ = Set of Training hyperparameters. The specific cost model and optimizer are detailed in Chapters 5 and 6.

Chapter 5

DATA PARALLELISM WITH ADAPT

A '**configuration**' in data parallelism as shown in Eq. 5.1 refers to the number of samples per batch or the *batch size* assigned to each device as well as the '*ring ordering*' of the devices. Dropping a device in this context is equivalent to assigning 0 samples per batch to it. The batch size assigned to each device models the total

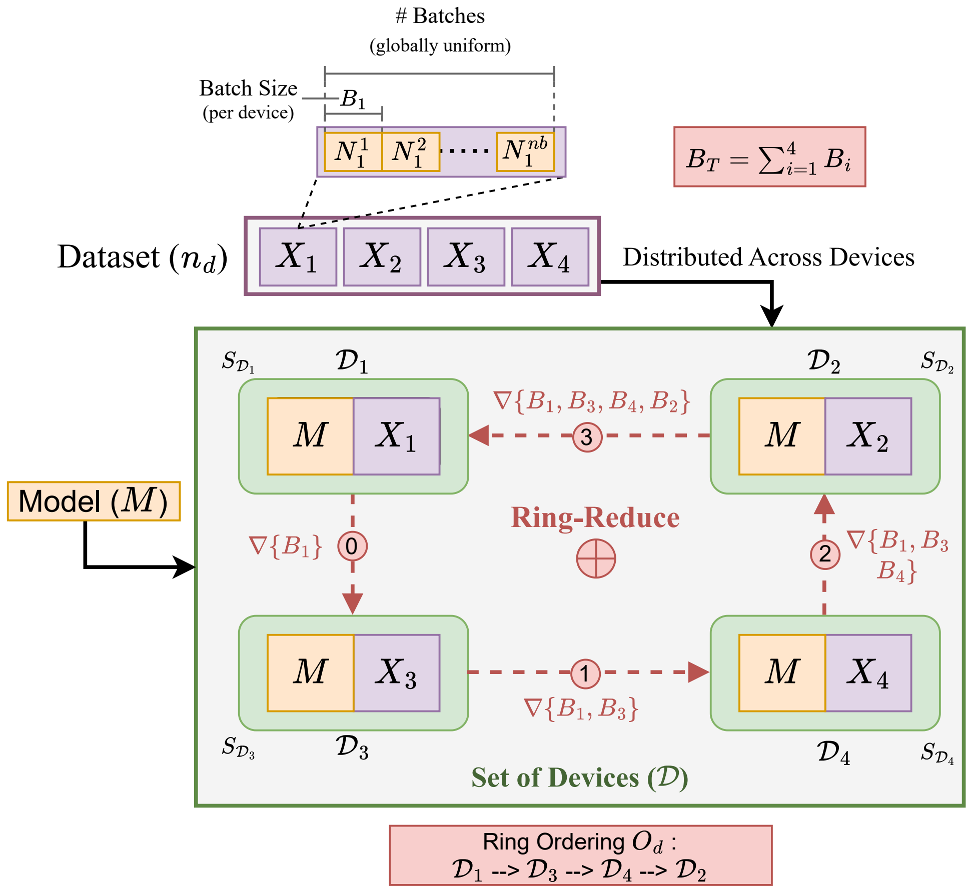


Figure 5.1: ADAPT addition over traditional Data Parallelism. Variable batching and ring ordering are used to perform heterogeneous load balancing.

computation load on each device. While this is relatively trivial in homogeneous device systems, it becomes more complicated when applied to asymmetric systems with different compute capabilities, network protocols, and resource contention.

$$\mathcal{C}_P = \{(D : (B_d, O_d)) \quad \forall D \in \mathcal{D} \ , \qquad \text{such that} \quad \sum_{D \in \mathcal{D}} B_d = B_T\} \tag{5.1}$$

where $\mathcal{D}$ is the set of devices available in the system, B_d is the batch size assigned to device d, B_T is the total batch size that gradients are averaged over, and O_d is the ring ordering of device d. As all the variable batch sizes add up to the total B_T, the model training characteristics are unaffected by the batch size differences between individual devices. The following subsections (5, 5) detail the cost function and optimization strategy.

As highlighted in Figure. 5.2, if we assign an equal number of batches to each device, as is the case in homogeneous systems, the faster devices will complete their computation before the slower devices, leading to idle time. This can happen due to a variety of reasons, such as asymmetric compute capabilities and resource contention.

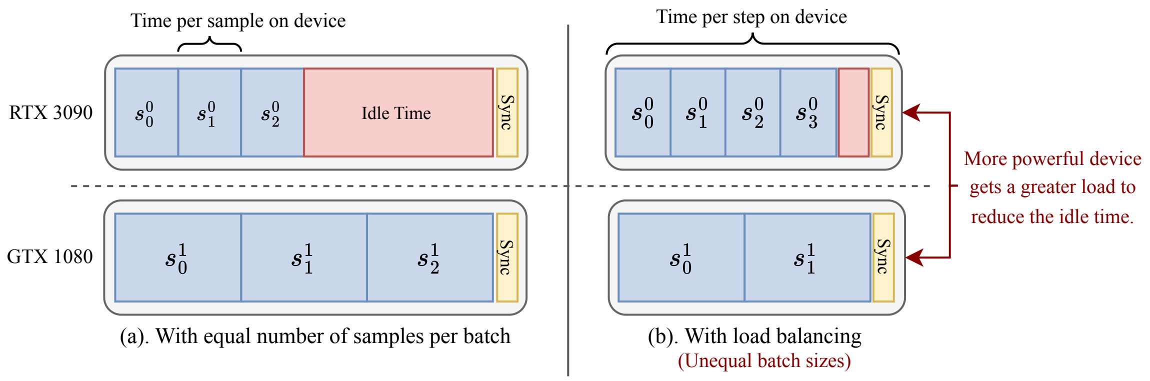


Figure 5.2: Bottlenecks in Data Parallelism. If the faster devices complete their computation before the slower devices, the faster devices will be idle. Creating a more optimal allocation of workload mitigates this issue.

The cost model in ADAPT addresses all of these issues.

Cost Function

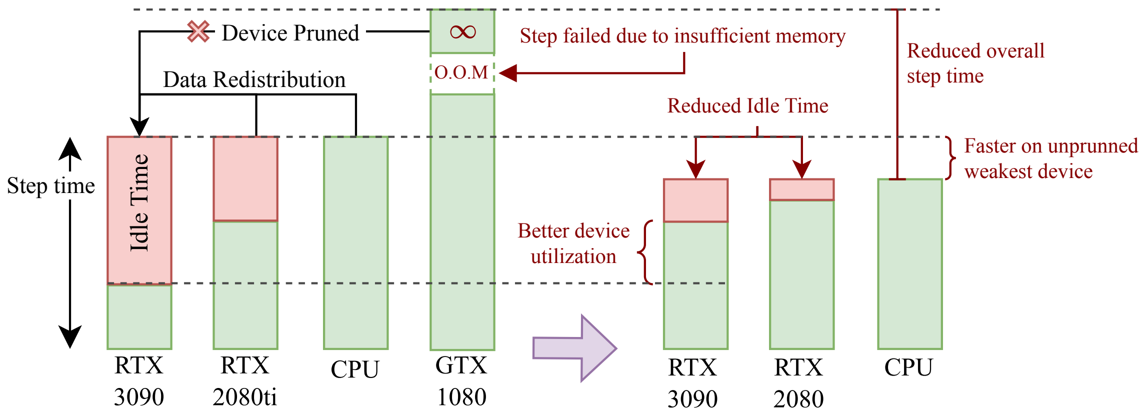
The cost model in ADAPT models the cost of running a given workload based on 3 factors:

1. **Communication Cost**: The time taken for communication between devices. This metric also takes into consideration the most optimal communication protocol to be used, given all the devices being considered.

2. **Computation Cost**: The time taken for computing the forward and backward pass on each device for a single batch without considering communication time.

3. **Memory Cost**: Checks if the model can fit in the memory of the device. If not, it returns infinity.

ADAPT uses a multi-stage cost function where each individual device has an associated cost function. The cost of the configuration is the sum of the cost of each device. The device cost function is given in Eq. 5.2.



Device Pruned
Data Redistribution
∞
O.O.M
Step failed due to insufficient memory
Reduced overall step time
Reduced Idle Time
Faster on unprunned weakest device
Step time
Idle Time
Better device utilization
RTX 3090
RTX 2080ti
CPU
GTX 1080
RTX 3090
RTX 2080
CPU

Figure 5.3: The overall training time is improved by reducing the execution time on the slowest device and pruning devices that are slower than their communication time. This leads to better utilization of the most powerful devices, while reducing the step time on the weakest devices and as a whole.

$$C_d = B_d \cdot S_d + \infty \cdot int(M^{B_d} \leq M_d^u) \tag{5.2}$$

Where, S_d is the time to compute the forward and backward pass for each data sample of the device 'd', which is calculated as T_d/B_d^p, where T_d is the time taken to compute the forward and backward pass for a single batch on device d and B_d^p is the batch size used during the profiling epoch. B_d is the batch size assigned to device d. M_d^u is the memory available on the device. M^{B_d} is the memory required for a single batch.

As depicted in Figure. 5.3, data parallelism is a bottle-necked process. Therefore, minimizing the cost of the configuration is equivalent to minimizing the maximum cost of any device in the configuration. Therefore, **the cost of the overall configuration is maximum device cost** as shown in Eq. 5.3.

$$Cost_{config} \;=\; max(C_d\ ;\ \forall d \in Devices) + \max\left(\{Comm(d, \text{Neighbor}(d))\ \mid\ \forall d_i \in \mathcal{D}\}\right)$$

Where, $Comm(d, d_i)$ is the communication time between device d and device d_i. We only consider the maximum communication time because *AllRingReduce* is only limited by bottle-necked devices.

Increasing batch size B_d up to a point on the most powerful device increases its effective utilization and reducing the idle time. Since B_T is fixed, as the batch size of the most powerful device increases, the batch size of the weakest device decreases, thereby reducing its time per step. This effectively reduces the computational bottle-necked. However, the communication time between devices is also a bottle-necked process and needs to be optimized. Therefore, **optimization in Data Parallelism is equivalent to minimizing 2 disjoint bottlenecks**.

Optimization

ADAPT optimizes for two main bottlenecks in data parallelism - (1). AllRingReduce communication bottleneck and (2). Computation bottlenecks.

Optimizing Communication Bottleneck

The *AllRingReduce* operation (Sergeev and Balso (2018)) is the bedrock of communication in data parallelism. In RingAllReduce, each device only communicates with its neighbors, which makes it a bottle-necked process. The total communication time in this process is directly proportional to the maximum communication time between two nodes in the ring.

The ADAPT framework during the Initial System Start-Up Stage from Figure. 4.1, profiles the peer-to-peer communication time between all the devices in the system. This is done for every supported communication protocol (NCCL(NVIDIA Corporation (2024)) and Gloo(Paszke *et al.* (2019))) supported by the devices in the asymmetric system. This is to account for the lower communication times that might result from dropping a device, for example, being able to use NCCL in a purely GPU system upon dropping the CPU.

The communication time between a pair of devices (d_i, d_j) is a function of the latency (L_{comm}) and bandwidth (BW) of the communication between the devices. The latency is the time taken for a single communication operation to complete, and the bandwidth is the amount of data that can be transferred in a unit of time. The communication time is given by:

$$Comm(d_i, d_j) = L_{comm}(d_i, d_j) + \frac{S(\mathcal{T})}{BW(d_i, d_j)} \tag{5.3}$$

where, $S(\mathcal{T})$ is the size of the tensor being communicated, $BW(d_i, d_j)$ is the bandwidth of the communication between the devices, and $L_{comm}(d_i, d_j)$ is the latency of the communication between the devices. Since both the L_{comm} and BW are device pair dependent, the ordering of devices in the ring is critical to minimizing the communication time.

Whenever a device d_i is dropped in ADAPT, an optimal ordering of the remaining devices is computed using a Traveling Salesman (TSP) based search to minimize the communication time. The pairwise communication time used is dependent on the protocol supported by all devices in the system. For example, when a CPU is dropped, the recomputed communication time assumes NCCL as the protocol instead of Gloo, which was used when the CPU was included.

Optimizing Compute Bottleneck

ADAPT optimizes for the compute bottleneck in data parallelism by a combination of compute-ratio guided splitting and device dropping. The Data Parallel Compute Optimizer, or **DPCO**, recursively performs a two-step process -

1. **Workload Redistribution** - The workload assigned to each device is adjusted based on the ratio of their compute throughput. This ensures that the fastest device has the highest workload.

2. **Device Drop Test** - The device with the highest drop affinity (see section. 5) is dropped from the configuration, if the new configuration has a lower cost, the device is dropped, otherwise the older configuration is returned.

This process is repeated until only one device remains in the configuration or the device drop test returns the older configuration.

Device Drop Affinity

The drop affinity is used to determine which device to drop first. ADAPT will first try to drop devices where the communication time with all its neighbors is greater than the compute time. If no such device is found, the device with the highest compute time is dropped.

$$Drop\_Affinity(d) = B_d \cdot S_d + \infty \cdot int(Comm(d_i, d_j) > T_d^{comp})$$

Chapter 6

PIPELINE PARALLELISM WITH ADAPT

Pipeline Parallelism in ADAPT defines its cost function and optimization strategy based on a state space graph representation of a model ($\mathcal{M}$) layer-device assignment as shown in Figure. 6.3. To define the cost functions and optimization strategy it is important to understand what defines a Pipeline Parallelism configuration in ADAPT. In Figure. 6.3, **each leaf node represents a configuration** ($\mathcal{C}_P$), which comprises the complete assignment of every layer of a model to a device at a given level of *'depth'*. The level of depth is defined by the user, and defines the unrolling of layer models into sublayers, for example, a ResNet block into its constituent layers. Each node in the tree is defined by a combination of a layer and a device. We can therefore

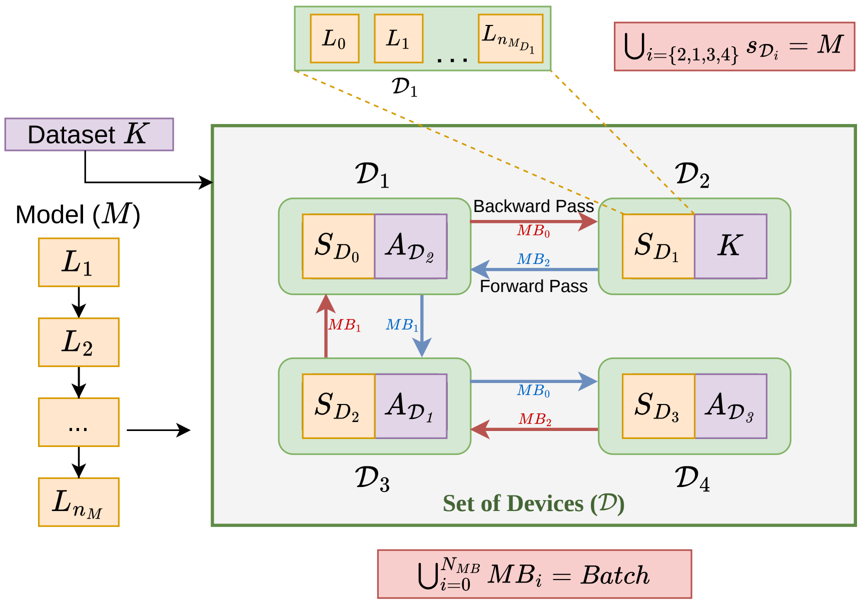


Figure 6.1: ADAPT addition over traditional Pipeline Parallelism. Stages retain correct layer ordering while identifying variable micro-batch sizes.

define a full configuration as a the following set which can be represented as a path from root to leaf in the state space graph -

$$\mathcal{C}_P = \{(L \rightarrow D) \quad \forall L \in \mathcal{M} \quad , \quad \exists D \in \mathcal{D}\} \tag{6.1}$$

where $\mathcal{D}$ is the set of devices available in the system and $\mathcal{M}$ is the set of layers defining the model. Therefore, each configuration is a set of assignments of layers to devices at a given level of depth for the model. The following subsections (6, 6) detail

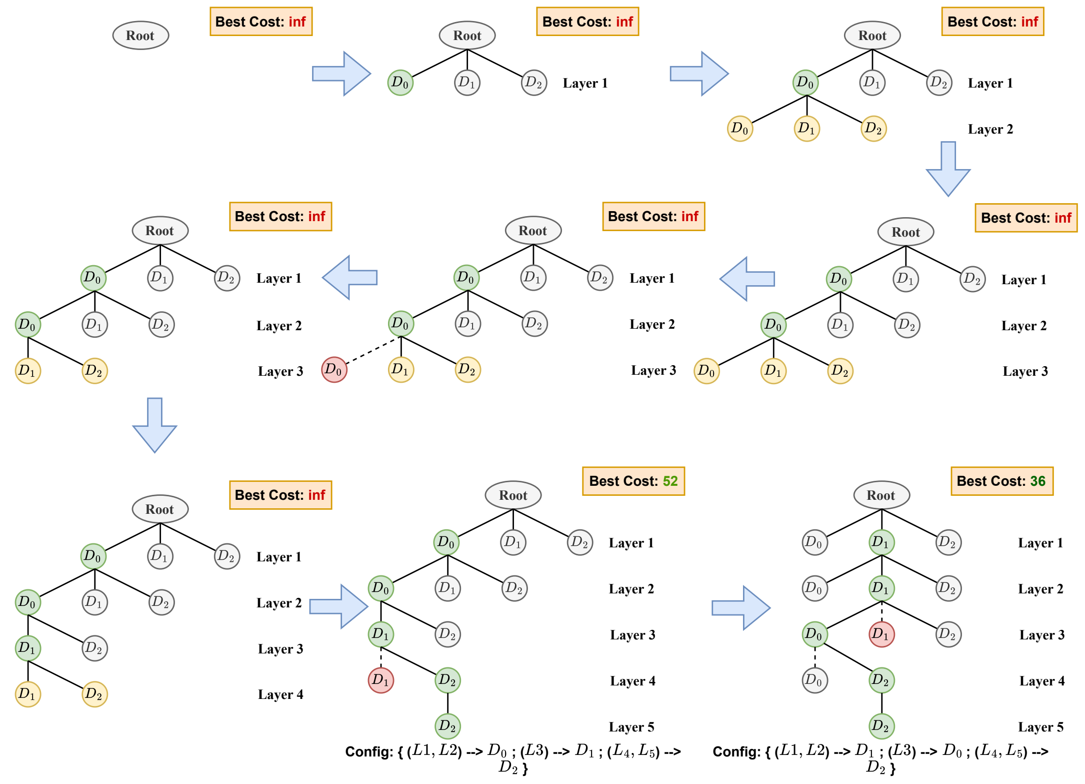

Best Cost: inf
Best Cost: 52
Best Cost: 36
Root
Layer 1
Layer 2
Layer 3
Layer 4
Layer 5
Config: { $(L1, L2)$ --> D_0 ; $(L3)$ --> D_1 ; (L_4, L_5) --> D_2 }
Config: { $(L1, L2)$ --> D_1 ; $(L3)$ --> D_0 ; (L_4, L_5) --> D_2 }

Figure 6.2: Pipeline ParallelismCost Optimization stages. The tree starts at the root node, from which each depth represents a layer to be mapped. Devices act as individual nodes, which are explored using DFS through the tree

the cost function and optimization strategy.

Cost Function

ADAPT defines 2 types of cost functions for Pipeline Parallelism - (1). Node Cost and (2). Configuration Cost. Both analytical cost models rely on the path cost from the root to a given node in the state space graph. We can define the cost of the edge between any two nodes N_0 and N_1 who are parent and child respectively as -

$$\begin{aligned} \mathcal{C}ost(N_0, N_1) = C_{N_0} \\ + Comp(N_1.\text{layer}, N_1.\text{device}) \\ + Comm(N_1.\text{device}, N_0.\text{device}) \end{aligned} \tag{6.2}$$

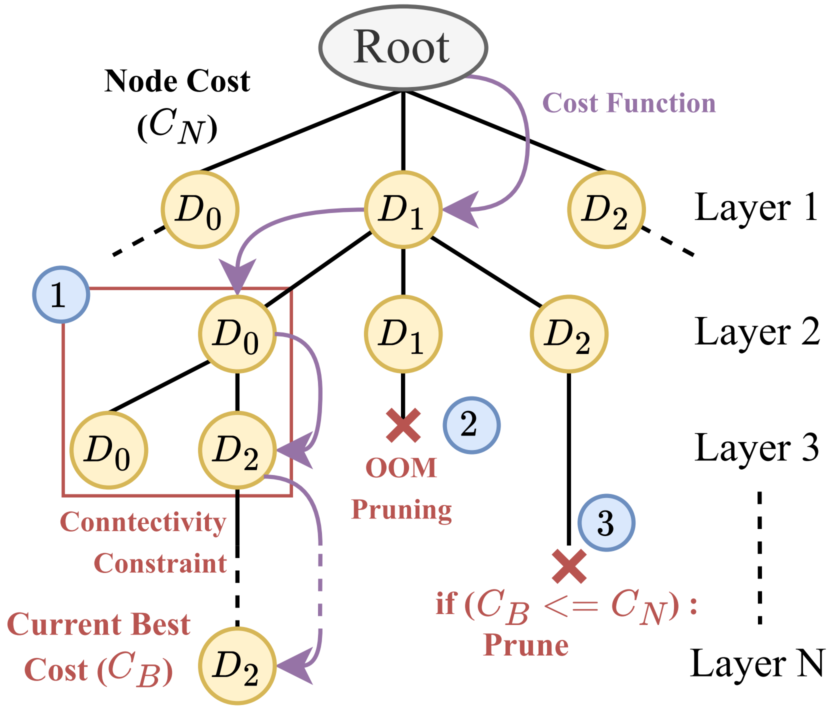


Figure 6.3: Pipeline Parallelism Optimization - ADAPT uses a state space DFS where each level of the tree represents a layer and each node represents a device. The path from root to leaf represents the assignment of devices per layer for the model as well as the cost of the assignment.

where C_{N_0} is the cost of the parent node, $Comp(N_1.\text{layer}, N_1.\text{device})$ is the time in ms for the layer of N_1 to be computed on the device of N_1. $Comm(N_1.\text{device}, N_0.\text{device})$ is the communication time in ms between the devices of N_1 and N_0. **This creates an inductive cost function where the cost of each node is additive to the cost of the parent node**.

Node Cost

The cost of every node is defined as the cost of the path ($\mathcal{P}(N)$) from the root to the node N and is the sum of all the edge costs in the path. We can mathematically define this as -

$$\mathcal{C}ost_{Node}(N) = \sum_{N_0 \in \mathcal{P}(N)} \mathcal{C}ost(N_0, N) \tag{6.3}$$

Configuration Cost

Each leaf node defines a complete configuration of a model. Therefore, when we compute the **node cost for a leaf node, it is the configuration cost**.

This creates an analytical cost function which is very fast to compute and representative of the factors that affect the performance of a configuration. The following subsections detail the key components of the cost function in greater detail.

Compute Cost

The compute cost is defined as the time in ms for a layer to be computed on a device. This includes the forward and backward pass time for the layer. ADAPT profiles this information on a per layer and per devices basis during the *'Runtime Profiling'* step. These profiled values are used to define *'stage compute cost'* ($\mathcal{C}omp(s_i)$) for each stage s_i in the model. Each stage is the set of layers assigned to the same device D_i. Therefore, the compute cost of a node is defined as follow:

$$Comp(N) = \sum_{s_i \in S} \mathcal{C}omp(s_i) \\ + (N_{micro-batch} - 1) \times max(\{\mathcal{C}omp(s_i) \forall s_i \in S\}) \tag{6.4}$$

where S is the set of stages in the model and $N_{micro-batch}$ is the number of micro-batches. This defines the bottle-necked, pipelined compute cost of the configuration.

Communication Cost

The communication cost in ADAPT is defined as the *'protocol-aware'* time in ms for a layer to be computed on a device. The 'protocol-aware' component accounts for the time taken to transfer data using proprietary protocols such as NCCL when the set of available devices all support it. We profile the per MB *'Latency'* and *'Bandwidth'* between evey pair of devices in the systems for the NCCL and GLOO (NVIDIA (2019); PyTorch (2017)) protocols. The communication times are then computed as a function of the profiled values and the size of the activations of the layer.

Micro-batch Estimation

A key component of Pipeline Parallelism is the micro-batch size. A micro-batch in Pipeline Parallelism is the size of the data that is processed in parallel by the model. The number of micro-batches is therefore the batch size of the model divided by the micro-batch size. ADAPT employs a best-case micro-batch estimation strategy for intermediate nodes. This is computed as -

$$\text{micro-batch Size} = \frac{BatchSize}{min(max(N_L, N_D) + N_{UD}, M.MB)} \tag{6.5}$$

where N_L is the number of layers remaining in the model, N_D is the number of available devices, N_{UD} is the number of used devices, $M.MB$ is the maximum micro-batch size hyperparameter. This formulation assumes maximum spliting for the remaining model. The higher the splitting, the lower the micro-batch size and faster the execution time per device.

Optimization Strategy

ADAPT use depth-first-search (DFS) with pruning to explore the state space graph and find the optimal configuration. As highlighted in Figure. 6.3, ADAPT prunes nodes based on 3 conditions -

1. The Connectivity Constraint: Current deep learning frameworks do not support disconnected layers of a model on the same device or incur massive performance penalties. This is a result of needing to launch multiple CUDA kernels to execute disconnected layers and not being able to apply graph optimization techniques to reduce the number of kernels launched. Therefore, if the device of the parent node is not the same as the device of the child node, then we remove the device of the parent node from the list of available devices for the child node.

2. The Memory Constraint - If the memory required for the configuration exceeds the available memory, then the configuration will fail to run or incur massive performance penalties. Every time a layer is assigned to a device for a given branch, we subtract the memory required for the layer parameters, buffers and activations from the available memory. If the available memory goes below 0, then the node is pruned.

3. Worse than Current best cost - Due to the inductive nature of the cost function, if the cost of a given node is worse than the current best cost, then the cost of all the children of that node will be worse than the current best cost and therefore not useful to explore. Therefore, we prune all the children of that node.

We prove in Lemma 1 (Appendix) that ADAPT will always find the optimal configuration given an ideal node cost function.

Dynamically Switchab;e Communication Protocols

Since the fastest communication medium for CUDA devices in PyTorch is NCCL (NVIDIA (2019)), it renders the CPU useless in current methods. ADAPT utilizes the communication backend sub-grouping technique to assign NCCL pipelines for GPU-to-GPU communications and PCIe medium (GLOO) for any communications involving CPUs. This is showcased in all the results that utilize the CPU for processing as well. Thus, this allows inclusion of CPUs without affecting the communication time between GPUs.

Chapter 7

EXPERIMENTAL SETUP

For our experimental setup, we used 3 systems. All the experiments were done using PyTorch Version 2.6 and CUDA Version 12.6. We define 3 main categories of system configurations for our experiments:

1. **Homogeneous Base**: Two identical GPUs (A6000 or Quadro RTX 5000) along with a high core count CPU.

2. **Homogeneous Overloaded**: Two identical GPUs (A6000 or Quadro RTX 5000), where one GPU has an additional computational load, paired with a high core count CPU.

3. **Heterogeneous**: Two different GPUs (RTX 3090 and GTX 1080) paired with a high core count CPU.

We test across 10 models, which represent the most popular model architectures as well as a variety of model sizes :

1. **EfficientNet-V2-S**: A small ($\sim$ 24M parameter) convolutional neural network architecture.

2. **EfficientNet-V2-Large**: A larger variant of the original EfficientNet V2 ($\sim$ 117M parameter)

3. **ConvNext-Base**: A heavy ($\sim$ 89M parameter) modern convolutional neural network architecture.

Table 7.1: ADAPT speedup results on a Homogeneous Overloaded system for Pipeline Parallelism. A system with 2 × A6000 and 1 × CPU is used here, with one A6000 GPU computationally overloaded by 40% and memory loaded by 30%

Model	Depth	BatchSize	MicroBatchSize	Baseline_time (s)	ADAPT_time (s) ↓	Speedup (x) ↑	ADAPT_Devices
ConvNext	2	512	32	142.82	65.01	2.20	2 × GPU + CPU
ConvNext	2	512	64	137.94	63.53	2.17	2 × GPU
ConvNext	3	512	64	138.07	59.35	2.33	2 × GPU
ConvNext	4	512	16	151.06	54.44	2.77	2 × GPU
ConvNext	4	512	32	142.80	53.11	2.69	2 × GPU
ConvNext	4	512	64	138.30	53.55	2.58	2 × GPU
LLAMA_IMDB	3	512	64	22.39	14.49	1.54	2 × GPU
LLAMA_IMDB	4	512	32	25.14	15.36	1.64	2 × GPU
ViT	3	128	16	422.76	270.85	1.56	2 × GPU
ViT	3	128	32	425.54	223.23	1.91	2 × GPU
ViT	4	128	16	423.43	245.86	1.72	2 × GPU
ViT	4	128	32	424.66	248.45	1.71	2 × GPU
MLP_Mixer	2	256	16	163.02	148.71	1.10	2 × GPU
MLP_Mixer	2	256	32	150.20	137.74	1.09	2 × GPU
MLP_Mixer	2	256	64	149.25	127.88	1.17	2 × GPU
MobileNetV4	4	256	64	36.35	26.47	1.37	2 × GPU
VGG19	2	256	32	160.99	126.14	1.28	2 × GPU
VGG19	2	256	64	162.49	120.50	1.35	2 × GPU
VGG19	3	256	32	160.35	126.87	1.26	2 × GPU
VGG19	3	256	64	153.19	123.45	1.24	2 × GPU
VGG19	4	256	16	176.52	138.33	1.28	2 × GPU
VGG19	4	256	32	163.23	128.74	1.27	2 × GPU
VGG19	4	256	64	162.84	125.25	1.30	2 × GPU
ConvNext_Large	2	128	64	22.76	19.23	1.18	2 × GPU + CPU
EfficientNet_Large	2	128	32	71.99	69.72	1.03	2 × GPU
EfficientNet_Large	2	128	64	43.03	39.01	1.10	2 × GPU + CPU
ConvNext_Large	2	128	16	63.93	47.97	1.33	2 × GPU + CPU
ConvNext_Large	2	128	32	45.45	27.51	1.65	2 × GPU
ConvNext_Large	2	128	64	36.03	20.29	1.78	2 × GPU + CPU
ConvNext_Large	3	128	32	25.97	24.66	1.05	2 × GPU
ConvNext_Large	3	128	64	23.41	21.00	1.11	2 × GPU + CPU
ConvNext_Large	4	128	32	25.47	23.53	1.08	2 × GPU
ConvNext_Large	4	128	64	21.82	20.25	1.08	2 × GPU + CPU

4. **ConvNext-Large**: The larger variant of the ConvNext model ($\sim$ 198M parameter).

5. **ViT-Base**: A medium scale ($\sim$ 86M parameter) vision transformer architecture.

6. **Llama-1B**: A small ($\sim$ 1B parameter) large language model architecture.

7. **VGG19**: A massive convolution Net architecture ($\sim$ 140M parameter)

8. **MobileNetV4**: A tiny architecture for small-scale systems ($\sim$ 8.4M parameter)

9. **MLP_Mixer**: Exclusively Multi-Layer Perceptron based architecture ($\sim$ 59M parameter)

10. **Swin Transformer Tiny**: A popular CV-based transformer with non-overlapping patching ($\sim$ 30M parameter)

We test against the PyTorch Automatic Equal Splitting techniques, which we use as our baseline for both Data Parallelism (Paszke *et al.* (2019)) and Pipeline Parallelism (Reed *et al.* (2022)).

Table 7.2: ADAPT speedup results on a Homogeneous Base system for Pipeline Parallelism. A system with 2 $\times$ A6000 GPUs and 1 $\times$ CPU is considered here.

Model	Depth	BatchSize	MicroBatchSize	Baseline_time(s)	ADAPT_time(s) $\downarrow$	Speedup_Factor(x) $\uparrow$	ADAPT_Devices
ConvNext	2	512	32	76.243	65.336	1.167	2 $\times$ GPU + CPU
ConvNext	3	512	32	64.034	47.181	1.357	2 $\times$ GPU + CPU
ConvNext	3	512	64	62.500	46.635	1.340	2 $\times$ GPU
ConvNext	4	512	16	66.609	44.812	1.486	2 $\times$ GPU
ConvNext	4	512	32	63.984	41.844	1.529	2 $\times$ GPU + CPU
ConvNext	4	512	64	62.816	41.525	1.513	2 $\times$ GPU
LLAMA_IMDB	3	512	32	12.423	12.081	1.028	2 $\times$ GPU
ViT	4	128	16	211.181	193.038	1.094	2 $\times$ GPU
ViT	4	128	32	200.904	191.796	1.047	2 $\times$ GPU
ViT	4	128	64	250.949	277.934	0.903	2 $\times$ GPU + CPU

Chapter 8

RESULTS

As explained in section 7, ADAPT was tested across multiple asymmetric as well as homogeneous systems. Through the results obtained, we aim to prove four important claims

Claim 1: ADAPT Significantly Boosts Heterogeneous Training Performance

We use the Heterogeneous system setup as explained from the Experimental Setup section 7 to test different models. Since the GTX1080 is the weaker GPU compared to the RTX 3090, ADAPT tends to shift a majority of the workload onto the stronger GPU, and in some cases includes the CPU in case of memory shortages, or small

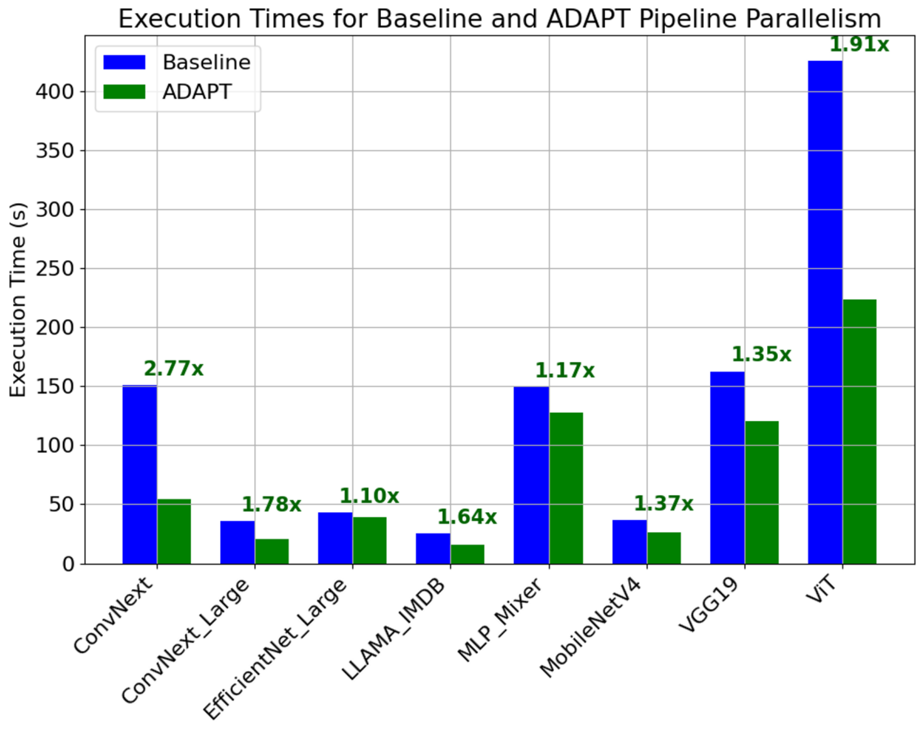


Figure 8.1: Average speedup and performances for models across pipeline parallelism for the homogeneous overload case

micro-batch sizes. Thus, slower devices have lesser workload to process, while more layers get processed quicker by more powerful devices, giving us a speedup in both cases. Table 8.1 showcases how pushing more layers on the powerful GPU results in a much faster execution time for any given model. Table 8.5 showcases the speedups for Data Parallelism.

Claim 2: Adapt Detects and Adjusts Performance Based on Symmetric GPUs with Varying Workloads

Similar to Claim 1, ADAPT identifies loaded devices, and balances the workload across the system accordingly. Here, one of the GPUs is loaded with 40% compute utilization and 30% memory utilization simulated by our custom GPU loading script. From Table 7.1, it is seen that equally splitting the layers across the GPUs results in

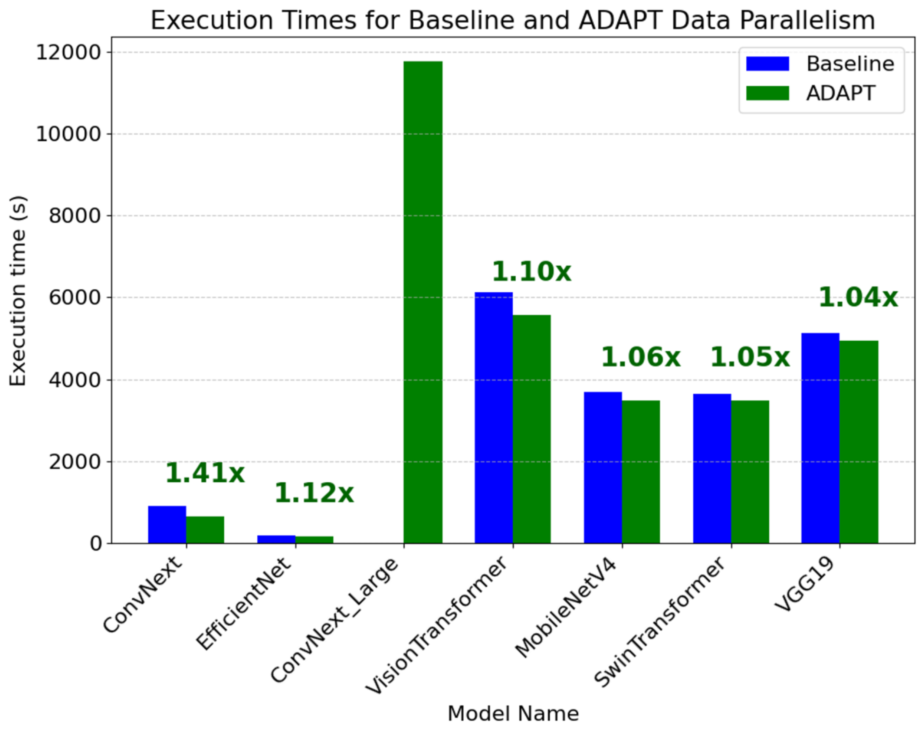


Figure 8.2: Average speedup and performances for models across data parallelism for the homogeneous overload case

Table 8.1: ADAPT speedup results on a Heterogeneous system for Pipeline Parallelism. The Heterogeneous experimental setup defined in section 7 is used here

Model	Depth	BatchSize	MicroBatchSize	Baseline_time (s)	ADAPT_time (s) ↓	Speedup_Factor (x) ↑	ADAPT_Devices
ConvNext	3	512	16	388.264	82.387	4.713	2×GPU
ConvNext	3	512	32	240.015	77.493	3.097	2×GPU
ConvNext	3	512	64	176.525	74.957	2.355	2×GPU
ConvNext	4	512	16	387.644	78.749	4.923	2×GPU + CPU
ConvNext	4	512	32	239.736	75.975	3.155	2×GPU + CPU
ConvNext	4	512	64	176.520	84.870	2.080	2×GPU
EfficientNet	3	512	64	19.078	21.506	0.887	2×GPU
LLAMA_IMDB	4	512	16	35.507	20.031	1.773	2×GPU

a highly negative impact, with the unloaded GPU having to wait for the loaded one to complete processing, leading to cascading stalls. ADAPT identifies these overload conditions and automatically shifts a majority of the workload onto other available devices in real time, hence showcasing high speedups. Table 8.4 showcases loaded examples where it can be seen speedups across different models for data parallelism.

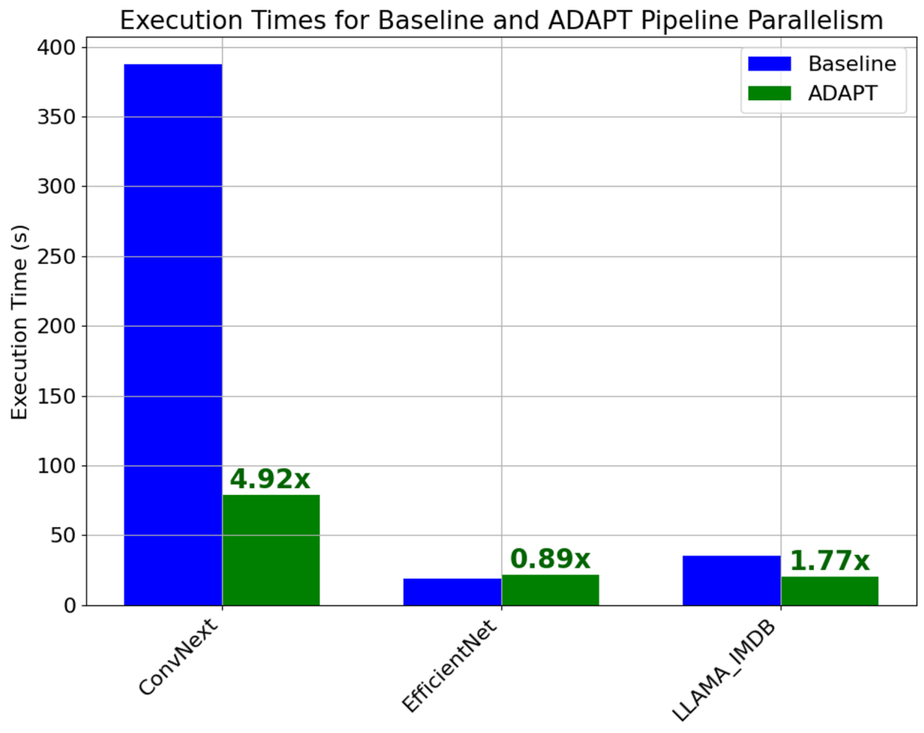


Figure 8.3: Average speedup and performances for models across data parallelism for the heterogeneous case

Claim 3: Adapt Can Include Previously Ignored Devices

Table 7.2 shows that even on regular systems, the inclusion of a CPU can actually mildly boost training performance, which may not have been possible in the case of current techniques that strictly utilize GPUs. A bigger advantage can be seen in the case of Table 7.1, where even with powerful A6000 GPUs, ConvNext still utilizes a CPU due to comparable micro-batch sizes for Pipeline Parallelism, resulting in a massive training speedup. Table 8.1 showcases multiple examples where including a

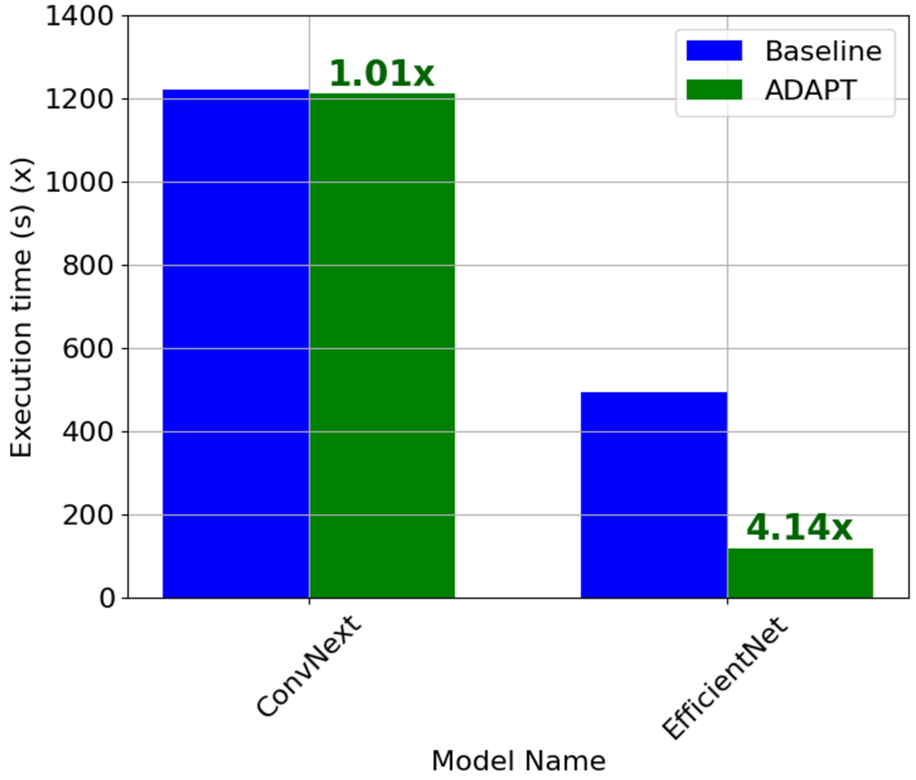


Figure 8.4: Average speedup and performances for models across data parallelism for the heterogeneous case

Table 8.2: ADAPT execution for Pipeline Parallelism overcoming OOM conditions on the Heterogeneous setup.

Model	Depth	BatchSize	MicrobatchSize	Baseline_Time (s)	ADAPT_Time (s) ↓	Speedup_Factor (x) ↑	ADAPT_Devices
ViT	4	128	16	-	340.2803	-	2 × GPU
ViT	4	128	32	-	333.8441	-	2 × GPU

Table 8.3: Data Parallelism Results for a Homogeneous Base system having 2 × Quaddro5000 GPUs and a CPU

Model	Data_Splits (gpu:0, gpu:1)	Baseline_time (s)	ADAPT_time (s) ↓	Speedup_Factor (x) ↑	ADAPT_devices
ConvNext	-	612.48	525.45	1.17	2 × GPU
EfficientNet	-	178.74	140.03	1.28	2 × GPU
ConvNext_Large	[25124, 24876]	1927.11	1901.02	1.01	2 × GPU
ViT	[25011, 24989]	2187.53	2168.90	1.01	2 × GPU
MobileNetV4	[25438, 24562]	555.79	552.75	1.01	2 × GPU
Swin_Transformer	[25844, 24156]	987.61	971.98	1.02	2 × GPU
VGG19	[25793, 24207]	OOM!	1756.89	-	2 × GPU

Table 8.4: Data Parallelism Results for a Homogeneous Overloaded system having 2 × Quadro RTX 5000 GPUs and a CPU, where one GPU is loaded with 50% compute load and 30% memory load.

Model	Data_Splits (gpu:0, gpu:1)	Baseline_time (s)	ADAPT_time (s) ↓	Speedup_Factor (x) ↑	ADAPT_devices
ConvNext	-	904.55	642.29	1.41	2 × GPU
EfficientNet	-	181.93	162.83	1.12	2 × GPU
ConvNext_Large	[50000, 0]	OOM!	11780.96	-	2 × GPU
VisionTransformer	[29683, 20317]	6120.03	5576.15	1.10	2 × GPU
MobileNetV4	[32290, 17710]	3690.48	3472.34	1.06	2 × GPU
SwinTransformer	[35514, 14486]	3645.28	3474.27	1.05	2 × GPU
VGG19	[33757, 16243]	5123.83	4938.45	1.04	2 × GPU

CPU improves train time significantly. Table 8.2 showcases a prime example where baseline techniques fail to train models due to OOM errors, but ADAPT detects the possibility of this and readjusts accordingly to find configurations that allows previously non-trainable models to train on such systems. It can also be seen through the example in the case of $VGG19$ in Table 8.3 and 8.4 for $ConvNext\_Large$ where the baseline methods for data parallelism OOM, but ADAPT finds configurations that can still train the model.

Claim 4: Adapt Can Adjust the Configurations at Runtime Based on Changes in Environmental Workloads

Since the cost optimization can be performed multiple times through the training process, the configurations can be adjusted based on the asymmetry across different devices in real time. As shown in Figure 8.5, training a constant equal configuration

Table 8.5: ADAPT speedup results on the Heterogeneous Base system for Data Parallelism

Model	Configuration	Baseline_Time (s)	ADAPT_Time (s) ↓	Speedup (x) ↑
ConvNext	Loaded	1221.51	1211.98	1.01
	Unloaded	1229.49	1226.97	1.00
EfficientNet	Loaded	492.23	118.82	4.14
	Unloaded	167.95	115.62	1.45

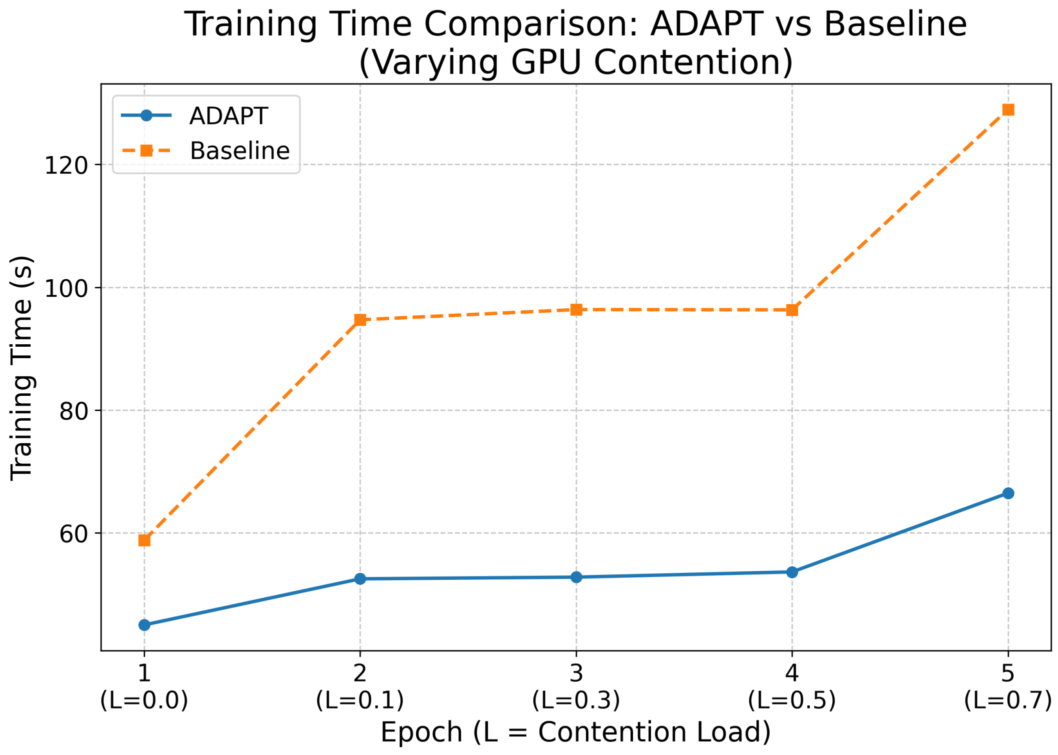


Figure 8.5: Comparative plot to show training with increase in GPU load over time for a ConvNeXt model on the Homogeneous Overloaded system

gradually becomes slower as the load on one of the GPUs increases. However, in the case of ADAPT, it can be seen how the workloads dynamically adjust the layers across the different devices.

Chapter 9

CONCLUSION

Adapting to heterogeneous setups for training is no trivial task and involves understanding the workload to be trained and the underlying system. Assigning workloads manually to different devices is a tedious process, and homogeneous splits to ease that procedure result in inefficient training. ADAPT addresses the critical issue of determining the best heterogeneous splits for different parallelism types and adjusts them as the model progresses through the training process. Future work would include testing the ADAPT system with tensor parallelism and multi-level parallelism types, which could not be included in this paper due to limited PyTorch support on implementation, as well as communication.

REFERENCES

Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems", URL `https://www.tensorflow.org/`, software available from tensorflow.org (2015).

Huang, Y., 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).

Jia, X., L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng, X. Liu and W. Lin, "Whale: Efficient giant model training over heterogeneous gpus", URL `https://arxiv.org/abs/2011.09208` (2022).

NVIDIA, "NVIDIA Collective Communications Library (NCCL)", `https://developer.nvidia.com/nccl` (2019).

NVIDIA Corporation, "NVIDIA Management Library (NVML)", `https://developer.nvidia.com/management-library-nvml` (2024).

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library", Advances in Neural Information Processing Systems **32** (2019).

PyTorch, "Gloo 100k: Collective communications library with various primitives for multi-machine training.", `https://github.com/pytorch/gloo` (2017).

Rajbhandari, S., P. Yao, E. Lopez, H. Li, J. Springer, M. Zhang, J. Rasley and Y. He, "Zero: memory optimizations toward training trillion parameter models", URL `https://doi.org/10.5555/3433701.3433727` (2020).

Reed, J., P. Belevich, K. Wen, H. Huang and W. Constable, "Pippy: A pipeline parallelism technique for pytorch", `https://github.com/pytorch/PiPPy` (2022).

Sergeev, A. and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow", URL `https://arxiv.org/abs/1802.05799` (2018).

Shoeybi, M., M. Patwary, R. Puri, P. LeGresley, J. Casper and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism", URL `http://arxiv.org/abs/1909.08053`, cite arxiv:1909.08053 (2019).

Zheng, L., Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning", in "16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)", pp. 559–578 (USENIX Association, Carlsbad, CA, 2022), URL `https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin`.

APPENDIX A

PIPELINE PARALLELISM DFS COMPLETENESS PROOF

Lemma 1: [Optimality Proof] *Given an ideal node cost function, the DFS with pruning strategy in ADAPT will always find the optimal configuration.*
Proof. As is known, DFS is an exhaustive search strategy, and it will always find the optimal configuration given an ideal node cost function. Therefore, to prove optimality, it needs to be shown that the pruning strategy will not prune the optimal configuration. As defined in section 6, ADAPT employs pruning based on 3 conditions -

1. **The Connectivity Constraint** - based on a limitation of current deep learning frameworks where disconnected layers of a model on the same device are not supported or lead to massive performance penalties, therefore any configuration that violates this constraint is provably suboptimal.

2. **The Memory Constraint** - If the memory required for the configuration exceeds the available memory, then the configuration is provably suboptimal.

3. **Worse than Current best cost** - If the configuration is worse than the current best cost, then the configuration is provably suboptimal. This is due to the inductive property of the cost function defined in section 6, where the cost of a node is additive to the cost of the parent node. Therefore, if the cost of a node is worse than the current best cost, then cost of all the children of that node will be worse than the current best cost and therefore suboptimal.

Therefore, as none of the pruning conditions will prune the optimal configuration, the DFS with pruning strategy in ADAPT will always find the optimal configuration.