Efficiently Training Deep Learning Models on Elastic and Heterogeneous Cloud Resources

by

Runsheng (Benson) Guo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Runsheng (Benson) Guo 2026

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Fiodar Kazhamiaka Senior Researcher, Azure Systems Research Group Microsoft
Supervisor:	Khuzaima Daudjee Research Professor, Cheriton School of Computer Science University of Waterloo
Internal Members:	Sihang Liu Assistant Professor, Cheriton School of Computer Science University of Waterloo
	Hong Zhang Assistant Professor, Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Seyed Majid Zahedi Assistant Professor, Dept. of Electrical and Computer Engineering University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some portions of this thesis are based on joint work with Khuzaima Daudjee, Utkarsh Anand, Victor Guo, Antonio Kim, Josh Hildred, Arthur Chen and Rathijit Sen, in which I am the first author and the primary contributor [58, 57, 59].

Abstract

Deep Neural Networks (DNNs) have demonstrated remarkable success across diverse domains, but their training requires substantial computational resources and is typically parallelized across large GPU clusters. However, such clusters are prohibitively expensive for most organizations to own and manage. Hence, instead of owning and managing their own clusters, organizations often rent clusters on cloud platforms to meet their training needs. While cloud environments offer elastic scalability and heterogeneous hardware options, they also introduce significant challenges for efficient distributed DNN training. Specifically, existing training frameworks lack support for dynamic reconfiguration during training, limiting the exploitation of cloud elasticity. Additionally, most systems assume homogeneous clusters, which rarely reflect the heterogeneous GPU clusters that organizations commonly use due to hardware availability constraints. Furthermore, heterogeneous network conditions in cloud environments create communication bottlenecks that limit the scalability of existing approaches.

This thesis presents three systems that collectively address these limitations to enable efficient distributed DNN training on elastic and heterogeneous cloud resources. First, Hydrozoa leverages cloud elasticity through serverless containers, enabling dynamic scaling and configuration changes during training without the traditional pitfalls of serverless computing. By combining data and model parallelism with fine-grained resource provisioning, Hydrozoa achieves cost-effective training while eliminating cluster management overhead. Second, Cephalo addresses heterogeneous GPU clusters by independently balancing compute and memory resources across GPUs with different capabilities. Unlike existing approaches that tie workload assignment to computational speed, Cephalo separately optimizes compute distribution through proportional batch sizing and memory utilization through intelligent partitioning of training state, activation checkpointing, and gradient accumulation strategies. Third, Zorse tackles heterogeneous network conditions, which are particularly common in heterogeneous clusters, by efficiently combining memory-efficient data parallelism with pipeline parallelism. Through interleaved pipelining, parameter and activation offloading, and heterogeneous pipeline parallelism configurations, Zorse achieves both communication and memory efficiency for training large DNN models across diverse network topologies.

The experimental evaluation demonstrates that these systems significantly improve training efficiency and resource utilization compared to existing approaches. Hydrozoa reduces training costs while providing seamless scalability, Cephalo simultaneously achieves high compute and memory utilization in heterogeneous clusters, and Zorse maintains high throughput under varying network conditions. Together, these contributions make

distributed DNN training more accessible, cost-effective, and efficient in modern cloud environments, advancing the state of the art in large-scale machine learning infrastructure.

Acknowledgements

I am deeply grateful to the many people who made this thesis possible. It would be impossible to thank everyone by name, but I remain indebted to all of my colleagues, mentors, friends, and family who supported me – including those not explicitly mentioned here.

First and foremost, I thank my supervisor, Khuzaima Daudjee, for his guidance, support, and steady encouragement over this journey. From countless hours spent discussing research ideas and polishing papers to giving helpful career advice, he taught me not only how to iterate on challenging problems quickly but also how to communicate ideas with clarity and precision. He also gave me the freedom and opportunity to explore my interests and to collaborate with both graduate and undergraduate students. My PhD under his mentorship helped me grow into an independent researcher and gave me the confidence to tackle complex, open-ended research problems.

I am grateful to my committee members – Sihang Liu, Hong Zhang, Fiodar Kazhamiaka, and Seyed Majid Zahedi – for their time, insightful discussions, and valuable feedback on this work. Even prior to being a part of my committee, Hong graciously took the time to review and give suggestions on my research direction.

I am very thankful for my parents and my brother, Victor, for believing in me throughout a long PhD journey, especially during moments when I considered giving up. I was fortunate to spend more time with my parents during the pandemic; their help with everyday life let me focus on research. Victor collaborated extensively with me on one of my first papers. I also appreciate my extended family including my aunt, uncle, grandparents, and cousins for their support in my studies. In particular, my uncle, the first in our family to earn a PhD, inspired me with his path.

I want to thank Zifan for endless support and encouragement that helped me push this thesis over the finish line, and for knowing exactly how to lift my spirits on those tough days.

I benefited from collaborations with many talented undergraduate students. Among them, Utkarsh Anand was my closest peer collaborator and worked on several research projects with me, even after his graduation. This thesis would not be possible without his contributions and it was a great pleasure to work on my research with him. I will never forget the many late nights we spent pair programming and debugging NCCL problems together.

Other notable contributors include Victor Guo and Antonio Kim on Hydrozoa. I would

also like to thank Arthur Chen, Maria Mihu, Kevin Li, Jerry Xie, Andy Wang, and Kaixi Tan for their contributions to my research.

I also thank senior PhD students who mentored me when I started, especially Michael Abebe and Brad Glasbergen. Moreover, it was a fun and enriching experience to collaborate with fellow grad students Petar Basta, Mohammad Khalaji, and Kerem Akillioglu on research outside of my thesis.

I am grateful to Raquel Urtasun for taking a chance on me as an undergraduate with little machine-learning experience and welcoming me into her self-driving lab at Uber ATG and later at Waabi. There, I was incredibly lucky to learn from outstanding mentors including Bin Yang, Wenyuan Zeng, Sergio Casas, Ming Liang, and Chris Zhang. I learned how to conduct ML research, including how to implement and debug training systems, design experiments, and approach research problems.

Thank you to Emad Soroush, Tyler Johnson, and Li Li for the opportunity to work on large-scale distributed training at Apple. This was my first chance to run distributed training jobs at the scale of thousands of GPUs. It is from this experience that I learned a lot about the internals of PyTorch and FSDP, which gave me the inspiration for large parts of this thesis.

I also appreciate the Gray Systems Lab (GSL) and Brandon Haynes and Rathijit Sen at Microsoft for the unique opportunity to work on large-scale distributed query scheduling. I am especially thankful to Brandon for invaluable career advice, and to Rathijit for entertaining my research ideas and collaborating with me on my thesis.

Earlier in my studies, Tamer Ozsu, Khuzaima Daudjee, and Khaled Ammar gave me the chance to explore distributed systems research as an undergraduate research assistant. I am thankful for these opportunities, as they set me on the path to pursue my PhD in distributed systems.

I owe special thanks to Lori Paniak for his tireless support in setting up, maintaining, and tailoring compute clusters to our research needs.

I am grateful for the friendships I made in Waterloo during my PhD, including Fadhil Abubaker, Ronak Pradeep, Ben Zhang, Kerem Akillioglu, Gaurav Sehgal, Anurag Chakraborty, Sairaj Voruganti, Tejasvi Kashi, Kriti Kathuria, Josh Hildred, Shirley Chen, Xiangyuan Zeng, and Shaokai Wang. In particular, Fadhil, Ronak, and Ben became my roommates at certain points in my studies, and were great friends to hang out with in and outside of school.

I am also appreciative of my friends from high school, undergrad, and outside of school who I had the pleasure of hanging out with during this period and who helped balance my

life outside of school.

Finally, I gratefully acknowledge financial and institutional support from the Province of Ontario, the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Cheriton School of Computer Science. I also thank Amazon Web Services, Microsoft Azure, and Waterloo.AI for the generous cloud compute credits that enabled many of the experiments in this thesis.

Dedication

This is dedicated to my parents, who have always supported and believed in me.

Table of Contents

Examining Committee Membership ii

Author's Declaration iii

Statement of Contributions iv

Abstract v

Acknowledgements vii

Dedication x

List of Figures xvii

List of Tables xxi

1 Introduction 1

1.1 Research Challenges and Motivation 2

1.1.1 Challenge 1: Serverless Elasticity for DNN Training 2

1.1.2 Challenge 2: Resource Imbalance in Heterogeneous GPU Clusters . 3

1.1.3 Challenge 3: Communication-Efficient Training in Network-Heterogeneous Environments 4

1.2 Thesis Contributions 4

1.2.1 Hydrozoa: Dynamic Hybrid-Parallel Training on Serverless Containers 4
1.2.2 Cephalo: Compute-Memory Decoupling for Heterogeneous Clusters 5
1.2.3 Zorse: Communication-Memory Efficient Training for Network Heterogeneity . . . 6
1.3 Thesis Outline . . . 6

2 Background **8**
2.1 Deep Neural Networks . . . 8
2.1.1 Architectures . . . 8
2.1.2 Scaling Laws and Performance . . . 9
2.2 Training Deep Neural Networks . . . 10
2.2.1 Optimization Algorithms . . . 10
2.2.2 Memory Requirements . . . 11
2.2.3 Computational Requirements . . . 12
2.2.4 Training Hardware . . . 13
2.3 Distributed Training . . . 13
2.3.1 Communication Primitives and Collective Operations . . . 13
2.3.2 Data Parallelism and Parameter Sharding . . . 15
2.3.3 Model Parallelism Techniques . . . 16
2.3.4 Hybrid Parallelism Strategies . . . 17
2.4 Cloud Computing for Machine Learning . . . 17
2.4.1 Cloud Service Models and Pricing . . . 18
2.4.2 Resource Elasticity and Management . . . 20
2.4.3 Containerization and Orchestration . . . 21
2.5 Heterogeneous and Network-Aware Training . . . 22
2.5.1 Resource Heterogeneity in Cloud Environments . . . 22
2.5.2 Network Topology and Communication Efficiency . . . 23

3 Related Work **25**
3.1 Dynamic Training Reconfiguration . . . 25
3.2 Serverless Computing for Machine Learning . . . 26
3.3 Fault Tolerance and Elastic Training . . . 27
3.4 Performance Optimization in Cloud Training . . . 28
3.5 Memory Management and Optimization Techniques . . . 29
3.6 Automated System Configuration and Optimization . . . 31
3.7 Heterogeneous GPU Clusters . . . 32
3.8 Heterogeneous Networks . . . 33
3.9 Training-Adjacent Workloads . . . 34

4 Leveraging Cloud Elasticity with Serverless Training **36**
4.1 Design . . . 36
4.1.1 Serverless Containers . . . 37
4.1.2 Hybrid-Parallelism . . . 38
4.1.3 Partitioning Algorithm . . . 40
4.1.4 Planner . . . 43
4.1.5 Dynamic Worker Scaling . . . 45
4.1.6 API . . . 45
4.2 Hydrozoa Implementation . . . 46
4.2.1 Worker . . . 46
4.2.2 Planner . . . 46
4.2.3 Coordinator . . . 47
4.3 Experiments . . . 47
4.3.1 Experimental Setup . . . 48
4.3.2 Comparison to Training on Serverless Functions . . . 49
4.3.3 Comparison to Training on VMs . . . 52
4.3.4 Dynamic Worker Scaling . . . 55
4.4 Summary . . . 56

5 Training on Heterogeneous GPU Clusters **57**

5.1 Design . . . 59

5.1.1 Division of Compute and Training State . . . 59

5.1.2 Managing Memory for Compute . . . 60

5.1.3 Performance Modeling . . . 62

5.1.4 Planner . . . 65

5.2 Implementation . . . 68

5.2.1 Profiler . . . 68

5.2.2 Optimizer . . . 69

5.2.3 Trainer . . . 69

5.2.4 Layered Gradient Accumulation . . . 70

5.3 Experiments . . . 70

5.3.1 Experimental Setup . . . 71

5.3.2 Training Throughput . . . 72

5.3.3 Larger Cluster Experiments . . . 74

5.3.4 Ablation Study . . . 77

5.3.5 Gradient Accumulation Optimizations . . . 78

5.3.6 Performance Model Accuracy . . . 79

5.3.7 Planner Optimization Time . . . 79

5.3.8 Optimized Training Configurations . . . 80

5.4 Discussion . . . 82

5.4.1 Adapting to Slower Networks . . . 82

5.4.2 Adapting to Heterogeneous Networks . . . 82

5.5 Conclusion . . . 83

6 Efficient Training Over Heterogeneous Networks **84**
6.1 Challenges in Heterogeneous Training . . . 85
6.1.1 Tensor Parallelism in Heterogeneous Clusters . . . 85
6.1.2 Data Parallelism in Heterogeneous Networks . . . 87
6.1.3 PP + DP in Heterogeneous Clusters . . . 88
6.1.4 Symmetric Parallelism Configuration . . . 89
6.1.5 Pipelining Overheads . . . 91
6.2 Limitations of Existing Systems . . . 91
6.3 Design . . . 92
6.3.1 Pipeline-Efficient ZeRO Data Parallelism . . . 93
6.3.2 Heterogeneous Pipeline Parallelism . . . 96
6.3.3 Planner . . . 97
6.4 Implementation . . . 103
6.4.1 Pipeline-Efficient ZeRO DP . . . 103
6.4.2 Communication Libraries . . . 104
6.4.3 Computation-Networking Overlap . . . 104
6.4.4 Computation-Offloading Overlap . . . 104
6.5 Experiments . . . 104
6.5.1 Training Throughput . . . 107
6.5.2 Cluster Scaling . . . 110
6.5.3 Comparison to Homogeneous Training . . . 111
6.5.4 Pipeline-Efficient ZeRO DP Analysis . . . 112
6.5.5 Ablation Study . . . 114
6.5.6 Pipeline-Efficient ZeRO DP vs TP . . . 115
6.5.7 Planner Optimization Time & Accuracy . . . 116
6.5.8 Resource Selection . . . 117
6.6 Discussion . . . 119
6.7 Conclusion . . . 121

7 Conclusion **122**

7.1 Summary of Research Challenges . . . 122

7.2 Research Contributions and Achievements . . . 123

7.2.1 Hydrozoa: Serverless Container-Based Training . . . 123

7.2.2 Cephalo: Decoupled Compute-Memory Optimization . . . 123

7.2.3 Zorse: Unified Memory and Communication Efficiency . . . 124

7.3 Future Research Directions . . . 124

7.3.1 Additional Parallelization Strategies . . . 124

7.3.2 Multi-Vendor Hardware Heterogeneity . . . 125

7.3.3 Adjacent Applications . . . 125

7.4 Concluding Remarks . . . 125

References **126**

Appendices **154**

A Hydrozoa . . . 154

A.1 Partitioning Algorithm . . . 154

B Zorse . . . 155

B.1 NCCL Communication Challenges . . . 155

B.2 Gloo-Based Solution . . . 156

C Cluster Profiling . . . 158

C.1 Network Bandwidth Profiling . . . 158

D Training Configurations . . . 160

E Tensor Parallelism Integration . . . 161

List of Figures

4.1 Hybrid-Parallelism in Hydrozoa 39

4.2 Pipelined Model-Parallelism in Hydrozoa 39

4.3 Hydrozoa Architecture 40

4.4 Hydrozoa ablation study on ResNet-34. Raw throughput (left) and throughput-per-dollar (right). 50

4.5 Hydrozoa ablation study on AlexNet. Raw throughput (left) and throughput-per-dollar (right). 50

4.6 Training curves for Hydrozoa with dynamic worker scaling versus fixing the number of workers. 51

4.7 Comparison of data-parallelism performance on Hydrozoa, Azure ML and SageMaker for ResNet-34. 54

4.8 Comparison of throughput for BERT-12 on Hydrozoa and SageMaker for different parallelism strategies. 55

5.1 AWS GPU availability (12-hour period). 57

5.2 GPU performance versus memory capacity for various GPU models. 58

5.3 Cephalo system architecture. 59

5.4 Gradient accumulation in FSDP (top) vs Cephalo (bottom) when training with 2 microbatches on a model consisting of 3 FSDP units. F_{ij} and B_{ij} are the forwards and backwards passes of the ith FSDP unit on the jth microbatch. AG_i and RS_i are the AllGather and ReduceScatter collectives for the ith FSDP unit. 61

5.5 Activation & Gradient Offloading in Layered Gradient Accumulation: This figure illustrates the transfer of activations and gradients between CPU and GPU during forward and backward passes for two consecutive model layers (i and $i+1$) across three microbatches. In the backward pass, activations are recomputed (RA). Here, GC^a_{ij} indicates moving the activation from GPU to CPU for layer i and microbatch j, with CG^a_{ij} representing the reverse. Similarly, GC^g_{ij} denotes transferring the activation gradient from GPU to CPU, and CG^g_{ij} signifies the reverse. 63

5.6 Training latency and memory allocated for compute as the microbatch size increases (Bert-Large). 64

5.7 Left: Throughput (TFLOPs) with different heterogeneous cluster configurations. Right: Throughput on Cluster B vs. a 32×A10G homogeneous cluster. 75

5.8 Mixed Precision (FP16) Training on Cluster B (Llama 7B, batch size 1024, sequence length 2048). 77

5.9 Throughput at different batch sizes with, and without, compute and memory balancing. 78

5.10 Speedup and memory reduction from gradient accumulation optimizations (LGA+CO+S+O). 79

5.11 Performance model absolute relative error. 80

5.12 Training configurations (ViT-G & Llama 3B). 81

6.1 Hardware Flops Utilization (HFU [22]) of tensor parallelism (TP) vs data parallelism (DP) with ZeRO-3 on 8 GPU AWS VMs. 86

6.2 Unidirectional bandwidth on AWS. Left: Heatmap of inter-node bandwidth between VMs of different GPUs across two regions, us-east-1 and us-east-2. Right: Intra-node bandwidth within VMs. For each bandwidth value the mean of several measurements is reported. Variability was observed to be low with a standard deviation of less than 2.5%. 87

6.3 Comparison of pipeline parallelism (PP) with ZeRO-2 vs ZeRO-3 across different Llama[177, 208] model sizes on 8 V100s + 8 T4s. OOM indicates Out-of-Memory. 88

6.4 This diagram contrasts the forwards pass in Zorse compared to breadth-first pipeline parallelism. The model has 20 layers (L_i) and there are 3 microbatches. There are 3 GPU groups, each with 4 ministages (Same as Figure 6.7 of the paper). In breadth-first pipeline parallelism, communication cannot be effectively overlapped with computation efficiently since the subsequent stage cannot start until the current stage reaches the first microbatch of the last layer. 90

6.5 Interleaved pipelining in Zorse. The diagram shows a training iteration for a model with 20 layers (L_i) and 3 microbatches. There are 3 GPU groups, each with 4 ministages. Due to different computational speeds, groups have different numbers of layers per ministage. Each group maintains the current and next ministage, offloading others to CPU memory. 93

6.6 Interleaved optimizer updates in Zorse vs. traditional pipeline parallelism (4 ministages, 3 microbatches). Interleaved optimizer updates free gradient memory earlier and better overlap with computation. 95

6.7 Heterogeneous pipeline parallelism in Zorse compared to traditional pipeline parallelism. 97

6.8 Architecture of Zorse's planner: ① profile workload and cluster, ② partition cluster into GPU groups, ③ partition model and data across GPU groups, and ④ select the best plan considered. 98

6.9 Training TFlops and HFU of Zorse compared to baselines for GPT models on Cluster A. 108

6.10 PFlops (1k TFlops) and HFU scaling as heterogeneous GPUs are added to the clusters. 110

6.11 GPU Utilization on heterogeneous clusters vs homogeneous subgroups of GPUs within the clusters. 111

6.12 TFlops and memory utilization for varying ministages per GPU. Values are normalized to 1 microstage. 113

6.13 TFlops and memory utilization of Zorse optimizations (Cluster A, LLama 65B). Baseline runs out of memory. 114

6.14 Training TFlops and Memory Utilization of Zorse with PP + TP vs Zorse with PP + DP on Cluster A with 4xH100s and 16xA100s. 115

6.15 Planner runtime breakdown per cluster. 116

6.16 Planner latency model absolute relative error. 117

6.17 Performance comparison of different homogeneous and heterogeneous subsets of clusters A-C. 118

1 Bandwidth comparison heatmaps between Gloo (left) and NCCL (right) communication backends for 4 GPUs distributed across 2 machines. GPUs 0 and 1 are NVIDIA H100 GPUs within the same machine, while GPUs 2 and 3 contain NVIDIA A100 GPUs within the same machine. The color intensity represents bandwidth in GB/s, with NCCL showing significantly higher bandwidth intra-node communication between similar GPU types. . 157

List of Tables

4.1 Platform comparison for DNN training. 38

4.2 A comparison of the models used for evaluation. 47

4.3 Cost per hour for ACI resources used by Hydrozoa. 48

4.4 Cluster A, C for Azure ML and B, D for SageMaker. 48

4.5 Summary of results comparing Hydrozoa to VM-based training on Azure ML and AWS SageMaker on K80 clusters. Strategy refers to the data × model parallelism employed. DP and HP correspond to data- and hybrid-parallelism, respectively. For example, 8×2 means the model is partitioned into 2 shards, and data-parallelism is applied on 8 copies of the model. . . 52

4.6 Summary of results comparing Hydrozoa to VM-based training on Azure ML and AWS SageMaker on V100 clusters. Strategy refers to the data × model parallelism employed. DP and HP correspond to data- and hybrid-parallelism, respectively. 53

5.1 Notation and Definitions 65

5.2 Model Statistics 72

5.3 GPU Specifications (FP32 TFlops) 72

5.4 Throughput comparison of different models/batch sizes on 8-GPU Cluster A (Part 1/2). *OOM* denotes Out-of-Memory. 73

5.5 Throughput comparison of different models/batch sizes on 8-GPU Cluster A (Part 2/2). *OOM* denotes Out-of-Memory. 73

5.6 Throughput comparison on 64-GPU Cluster B. 75

5.7 Breakdown of profiling and optimization runtime. 82

6.1 Comparison of heterogeneous training systems. 92
6.2 Memory and communication comparison of different training strategies. . . 94
6.3 Model Statistics 105
6.4 Datacenter-class GPU Specifications 106
6.5 Cluster GPU Configurations 106
6.6 Throughput (TFlops) and GPU utilization (HFU) of Zorse compared to other systems across different model sizes and clusters (higher is better). *OOM* denotes Out-of-Memory. 107

1 Zorse's training configurations used in the evaluation. 160

Chapter 1

Introduction

Deep Neural Networks (DNNs) have demonstrated remarkable success across diverse domains, including image classification [63, 164, 38], natural language processing [24, 180, 15], autonomous driving [113, 195, 203], and data systems [186, 88, 117, 60]. The training process for DNNs involves substantial matrix multiplications, which, although computationally intensive, are highly parallelizable. As a result, training is often accelerated using GPUs, leveraging their thousands of cores for parallel processing.

The exponentially increasing size and complexity of state-of-the-art DNNs demand substantial computational resources and potentially years of GPU hours [177] for training. For instance, GPT-3 required 3.14×10^{23} exaflops of compute for training [15], while more recent large language models like GPT-4 and PaLM demand orders of magnitude more resources. Training a single Llama-2 70B model requires approximately 1.7 million GPU hours [177]. To address these enormous computational requirements, distributed training techniques like data parallelism and model parallelism are widely adopted. Data parallelism [155, 97] involves replicating the model and training it on different data partitions across multiple workers simultaneously. Conversely, model parallelism partitions the model itself across workers [70, 128, 163], with each worker processing the same data batch for its assigned model segment.

Although distributed training significantly reduces training time [54], the scale and resource demands of modern DNNs incur prohibitively large costs for hardware ownership and maintenance. Consequently, researchers and practitioners increasingly utilize cloud platforms for DNN training. Cloud environments offer elastic scalability, enabling users to provision large numbers of GPUs on demand. This elasticity allows organizations to train large models without substantial upfront hardware investment and to dynamically adjust

resources according to workload requirements. Moreover, cloud platforms provide diverse heterogeneous hardware options, granting users flexibility in selecting optimal hardware for their tasks, while also managing the underlying infrastructure, thereby relieving the user of maintenance overhead.

While the elasticity and heterogeneity of cloud resources offer flexibility in scaling training jobs, they also introduce challenges for efficient DNN training. For instance, elasticity permits dynamic scaling, but allocating new nodes can temporarily interrupt training. This process necessitates reconfiguration of the parallelization strategy, a feature not natively supported by common DNN training frameworks [143, 174, 127]. Furthermore, dynamic scaling can alter the effective training batch size, potentially impacting model convergence [35].

The availability of heterogeneous hardware options in the cloud gives users the flexibility to select cost-effective GPUs for their training tasks. However, high demand and limited availability of the most powerful GPUs can constrain access to large homogeneous GPU allocations. This has motivated recent systems to explore training over mixed GPU types as a way to aggregate the available compute resources [57, 179, 139]. Effectively training on such heterogeneous clusters requires methods for balancing computation, memory usage, and communication across GPUs with different capabilities.

Finally, cloud environments often feature heterogeneous network infrastructure, characterized by variations in intra-node, inter-node, and inter-datacenter communication links. This heterogeneity is amplified in clusters composed of diverse GPU types, as nodes equipped with different GPUs may also possess disparate networking hardware. Consequently, there is a need for communication-efficient training strategies capable of adapting to network heterogeneity while simultaneously addressing imbalances in compute and memory resources across devices.

1.1 Research Challenges and Motivation

This thesis addresses three fundamental challenges in modern distributed DNN training:

1.1.1 Challenge 1: Serverless Elasticity for DNN Training

Traditional distributed training systems are designed for fixed clusters of virtual machines (VMs), requiring explicit cluster management and often resulting in costly overprovisioning

of resources. While serverless computing offers elasticity and cost efficiency, existing serverless systems for DNN training suffer from significant limitations. Current serverless function platforms like AWS Lambda are restricted to CPU-only workloads and rely on indirect communication through external storage, making them orders of magnitude slower than GPU-based training on VMs.

Existing attempts at serverless DNN training achieve only CPU-based performance while sacrificing the GPU acceleration that is essential for faster training times [74, 16]. Hence, they are not suitable for the resource-intensive and communication-heavy requirements of distributed DNN training.

Research Question: How can one design a distributed training system that harnesses the elasticity and cost benefits of serverless containers while achieving GPU-accelerated performance competitive with traditional VM-based approaches?

1.1.2 Challenge 2: Resource Imbalance in Heterogeneous GPU Clusters

The limitations in cloud GPU availability incentivizes many organizations to assemble heterogeneous clusters combining different GPU generations and types. However, most existing distributed training methods assume homogeneous clusters and distribute the workload evenly [72, 172]. On the other hand, systems designed for heterogeneous clusters distribute workload based solely on compute capacity, resulting in a mismatch between compute and memory utilization: a GPU's memory capacity does not always scale proportionally with its compute speed. For example, while an H100 GPU offers significantly faster computation than an A100, both have the same 80 GB memory capacity, potentially preventing the H100 from handling proportionally more work due to memory constraints.

This compute-memory mismatch leads to two critical inefficiencies: (1) underutilizing compute on GPUs with high compute-to-memory ratios, and (2) underutilizing memory on GPUs with low compute-to-memory ratios. Existing heterogeneous training systems cannot effectively balance these resources because they tightly couple compute and memory allocation.

Research Question: How can one decouple compute and memory allocation to fully utilize both the aggregate computational power and memory capacity of heterogeneous GPU clusters without being constrained by the least capable resource?

1.1.3 Challenge 3: Communication-Efficient Training in Network-Heterogeneous Environments

Large-scale training increasingly requires combining GPUs connected through different networking infrastructure with vastly different bandwidths, with variations spanning 35-fold between intra-node NVSwitch connections and inter-datacenter links. This is especially true for clusters with heterogeneous GPUs, as older GPUs will not be compatible with newer interconnects. However, this can also be true for homogeneous GPUs connected through different machines and datacenters. Traditional parallelization strategies like data parallelism and tensor parallelism are often bottlenecked by communication in such environments, while communication-efficient pipeline parallelism cannot be used in isolation due to its scalability limitations.

Existing approaches that combine multiple parallelization strategies also force a trade-off between memory efficiency and communication efficiency [59]. For example, combining pipeline parallelism with parameter replicating data parallelism is communication-efficient at the cost of memory efficiency. Pipeline parallelism with ZeRO-style data parallelism that shards parameters is memory-efficient but incurs significantly more communication overhead.

Research Question: How can one design hybrid parallelization strategies that achieve both memory efficiency and communication efficiency while adapting to the network and compute heterogeneity common in modern cloud environments?

1.2 Thesis Contributions

This thesis presents three complementary systems that address the challenges outlined above. Each system targets a specific aspect of the cloud training problem while contributing to a comprehensive and increased understanding of efficient distributed training in elastic and heterogeneous cloud environments.

1.2.1 Hydrozoa: Dynamic Hybrid-Parallel Training on Serverless Containers

To address the challenge of exploiting serverless elasticity, this thesis presents Hydrozoa [58], the first system to combine serverless containers, GPU acceleration, and hybrid-parallel

training. Hydrozoa overcomes the limitations of existing serverless training approaches by leveraging serverless containers instead of functions, enabling GPU provisioning and direct worker-to-worker communication.

Key Research Contributions:

- **Serverless Container Architecture**: First system to demonstrate GPU-accelerated DNN training on serverless containers, achieving direct communication while maintaining serverless elasticity and cost benefits.
- **Hybrid-Parallel Planner**: Novel optimization algorithm that automatically determines the optimal combination of data and model parallelism for a given model and cluster configuration.
- **Dynamic Worker Scaling**: Integration of elastic scaling policies during training that maintain convergence properties while adapting to changing resource availability.
- **Performance Achievements**: Up to 155.5× higher throughput-per-dollar compared to existing serverless systems and 5.4× higher throughput-per-dollar than VM-based training while eliminating cluster management overhead.

1.2.2 Cephalo: Compute-Memory Decoupling for Heterogeneous Clusters

To address resource imbalance in heterogeneous clusters, this thesis presents Cephalo [57], which holistically balances compute and memory usage by decoupling compute distribution from training state assignment. Cephalo enables independent optimization of computational workload and memory utilization across GPUs with varying capabilities.

Key Research Contributions:

- **Flexible Training State Partitioning**: Novel extension of FSDP that enables uneven parameter sharding independent of compute distribution, allowing simultaneous efficient use of compute and memory across heterogeneous hardware.
- **Layered Gradient Accumulation**: Optimized gradient accumulation implementation that reduces communication overhead while managing memory efficiently across different microbatch configurations.

- **Planning Algorithm**: Dynamic programming solution that simultaneously optimizes compute distribution, memory allocation, and gradient accumulation parameters to maximize training throughput.
- **Performance Achievements**: 1.2×–10.8× higher training throughput compared to state-of-the-art heterogeneous training systems while supporting larger models and batch sizes.

1.2.3 Zorse: Communication-Memory Efficient Training for Network Heterogeneity

To address communication challenges in network-heterogeneous environments, this thesis presents Zorse [59], which introduces Pipeline-Efficient ZeRO DP, a novel integration of pipeline parallelism and data parallelism that achieves both memory and communication efficiency simultaneously. This is carefully integrated with heterogeneous pipeline parallelism that allows training configurations to be more flexibly adapted for heterogeneous clusters.

Key Research Contributions:

- **Pipeline-Efficient ZeRO DP**: First system to combine the memory efficiency of ZeRO-3 data parallelism with the communication efficiency of pipeline parallelism through interleaved pipeline stage computation and intelligent parameter offloading.
- **Heterogeneous Pipeline Parallelism**: Support for asymmetric pipeline stage configurations that accommodate varying GPU counts and types within the same training job.
- **Automated Configuration Planner**: Two-phase optimization algorithm that efficiently navigates the large search space of configurations, automatically optimizing training for a given workload and cluster
- **Performance Achievements**: Up to 3× higher training throughput compared to state-of-the-art systems across diverse heterogeneous cluster configurations for models scaling to 65 billion parameters.

1.3 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2 provides background on deep neural networks, how they are trained, and distributed training strategies, covering data parallelism, model parallelism, and hybrid approaches. It also introduces cloud computing paradigms and examines the specific challenges posed by resource heterogeneity in modern ML infrastructure.

Chapter 3 surveys related work in distributed training systems, serverless computing for ML workloads, and heterogeneous resource management. This chapter highlights the limitations of existing approaches that motivate the research contributions within this thesis.

Chapters 4–6 present the three main contributions of this thesis:

- Chapter 4 describes Hydrozoa and demonstrates how serverless containers can achieve competitive training performance while providing superior cost efficiency and elastic scaling capabilities.
- Chapter 5 presents Cephalo and shows how decoupling compute and memory allocation enables efficient utilization of heterogeneous GPU clusters without sacrificing training throughput.
- Chapter 6 introduces Zorse and demonstrates how novel pipeline and data parallelism integration can achieve both memory and communication efficiency in heterogeneous GPU clusters.

Chapter 7 concludes with a summary of the research contributions and findings developed within this thesis, and outlines directions for future research in cloud-based distributed training.

Chapter 2

Background

This chapter covers background information related to training and deploying deep neural networks in distributed and cloud environments.

2.1 Deep Neural Networks

This chapter introduces core deep neural network concepts, including common architectures and the scaling behaviors that drive modern model design and training requirements.

2.1.1 Architectures

Deep Neural Networks (DNNs) are a class of machine learning models composed of multiple layers of interconnected neurons that learn hierarchical representations from data. The fundamental building block is the perceptron, which computes a weighted sum of inputs followed by a non-linear activation function [150]. Modern DNNs stack these layers to create complex function approximators capable of learning intricate patterns in high-dimensional data.

Convolutional Neural Networks (CNNs) are a specialized type of DNN primarily used for processing grid-like data, such as images [94, 90]. CNNs employ convolutional layers that apply learnable filters across the input data. These filters are parameter-efficient and detect hierarchical spatial features. Key architectural innovations in CNN models include skip connections in ResNet [63], which enable training of very deep networks by addressing the vanishing gradient problem.

The Transformer layer is the backbone of modern language models. The core innovation is the self-attention mechanism, which computes attention weights between all pairs of positions in the input sequence, enabling parallel computation and effective modeling of long-range dependencies. The layer then outputs a new representation of the input based on the attention weights.

Transformers have achieved remarkable success in natural language processing and have become the standard architecture for processing textual data [36, 15, 55]. The BERT model [36] uses an encoder-decoder architecture, where the encoder processes the input sequence bidirectionally to create rich contextual representations that can attend to both past and future tokens simultaneously. This bidirectional context allows BERT to excel at tasks requiring full sequence understanding, such as question answering and text classification. In contrast, the GPT family [15] uses a decoder-only architecture that processes sequences autoregressively, generating tokens one at a time while attending to only previous tokens in the sequence. This unidirectional constraint makes GPT particularly well-suited for generative tasks like text completion and language modeling. More recently, Vision Transformers (ViTs) [38] have shown that the transformer architecture can also excel in computer vision tasks when provided with sufficient training data.

Large Language Models (LLMs) scale the Transformer architecture to billions or even trillions of parameters, leveraging massive datasets to acquire general-purpose language understanding and generation capabilities [15, 55, 22]. This scaling has led to breakthroughs in few-shot and zero-shot learning, where LLMs can perform tasks with little or no task-specific training. The emergence of capabilities like in-context learning, chain-of-thought reasoning, and instruction-following has made LLMs versatile tools for a wide range of applications.

2.1.2 Scaling Laws and Performance

The performance of deep neural networks exhibits predictable scaling behaviors with respect to model size, dataset size, and computational resources. Kaplan et al. [84] demonstrated that the performance of autoregressive language models follows power-law relationships with respect to the number of parameters N, dataset size D, and compute budget C. Specifically, the test loss scales inversely with respect to the number of parameters.

These scaling laws have profound implications for resource allocation and model design. The research shows that performance improvements from scaling are predictable but subject to diminishing returns, and that different scaling dimensions (parameters, data, compute) must be balanced optimally. Hoffmann et al. [67] refined these scaling laws

and demonstrated that many large models are under-trained, leading to the development of compute-optimal models like Chinchilla that achieve better performance with fewer parameters but more training data.

The scaling laws also reveal the massive computational requirements of state-of-the-art models. Training GPT-3 [15] required approximately 3.14×10^{23} floating-point operations, while more recent models like PaLM [22] and LLaMA [177] require even more computational resources. These computational demands necessitate distributed training across hundreds or thousands of accelerators, making efficient parallelization strategies crucial for practical model development.

Furthermore, the scaling laws highlight the importance of efficient resource utilization. Since performance scales sublinearly with compute, maximizing the utilization of available computational resources becomes critical for cost-effective training. This drives the need for optimized distributed training systems that can effectively harness compute clusters.

2.2 Training Deep Neural Networks

This chapter summarizes how DNNs are optimized in practice and outlines the memory, compute, and hardware considerations that determine training performance at scale.

2.2.1 Optimization Algorithms

DNNs are trained iteratively using variants of stochastic gradient descent (SGD) [149]. The training process involves several key steps: during the *forward pass*, the model processes a batch of input data to generate predictions; a *loss function* quantifies the difference between predictions and target values; the *backward pass* calculates gradients of the loss with respect to model parameters using backpropagation [151]; and an *optimizer* updates parameters using these gradients.

While basic SGD uses a fixed learning rate for all parameters, adaptive optimization algorithms have become standard in modern deep learning. Adam [86] combines momentum with adaptive learning rates for each parameter, maintaining running averages of both gradients and their squared values. This allows Adam to adapt the learning rate based on the historical gradient information, often leading to faster convergence and better performance than basic SGD.

AdamW [110] addresses weight decay regularization issues in Adam by decoupling weight decay from gradient-based updates. Other popular optimizers include RMSprop [66], which

adapts learning rates based on recent gradient magnitudes, and AdaGrad [41], which scales learning rates inversely to the square root of cumulative squared gradients. More recent developments include LAMB [198] for large batch training, Lion [20] for memory-efficient optimization, and Muon [81], which provides improved memory efficiency and optimization effectiveness for large language models.

Learning rate scheduling plays a crucial role in optimization effectiveness. Common strategies include exponential decay, cosine annealing [111], and warm restarts [111]. Warmup strategies, where the learning rate is gradually increased from zero to its target value over the first few epochs, have proven essential for training large models and preventing early training instability [83].

Advanced optimization techniques include gradient clipping to prevent exploding gradients [205], gradient accumulation to simulate larger batch sizes with limited memory, and mixed precision training [125] to reduce memory usage and accelerate training while maintaining numerical stability.

2.2.2 Memory Requirements

Deep neural network training imposes substantial memory requirements that grow with model size, batch size, and sequence length. The memory footprint consists of several components: model parameters, gradients, optimizer states, and activations [144]. For a model with P parameters using the Adam optimizer, the optimizer alone requires $8P$ bytes of memory (4 bytes each for momentum and variance estimates), while parameters and gradients require an additional $8P$ bytes in mixed precision training.

Activation memory consumption depends on the model architecture and input batch size. For transformer models, activation memory scales quadratically with sequence length due to the attention mechanism, making long-sequence training particularly memory-intensive [13]. The total activation memory for a transformer can be approximated as $O(B \cdot L^2 \cdot H + B \cdot L \cdot H^2)$, where B is the batch size, L is the sequence length, and H is the hidden dimension [142].

Memory optimization techniques have become essential for training large models. Activation checkpointing [142] trades computation for memory by recomputing activations during the backward pass instead of storing them. Gradient checkpointing can significantly reduce activation memory while increasing the amount of computation required by approximately a third.

Model state sharding techniques like ZeRO [144] partition optimizer states, gradients,

and parameters across multiple devices to reduce per-device memory requirements. ZeRO-1 shards optimizer states, ZeRO-2 additionally shards gradients, and ZeRO-3 shards parameters, with each level providing greater memory savings at the cost of increased communication overhead.

Offloading strategies move data between GPU memory, CPU memory, and storage based on usage patterns. ZeRO-Offload [2] moves optimizer computations to CPU, while more aggressive approaches can offload parameters and gradients to CPU or even storage when not actively used [146].

2.2.3 Computational Requirements

The computational cost of training deep neural networks scales with model size, dataset size, and training duration. For a transformer model with P parameters trained on T tokens, the computational cost can be approximated as $6PT$ floating-point operations [84]. The computational cost of state-of-the-art models is extraordinarily large, when considering they have hundreds of billions of parameters and are trained on trillions of tokens [55].

Matrix multiplication operations dominate the computational profile of neural network training, particularly in transformer architectures where attention and feedforward layers involve large matrix operations. These operations are highly parallelizable and benefit significantly from specialized hardware accelerators like GPUs and TPUs [82].

Computational efficiency can be improved through various techniques. Mixed precision training [125] uses 16-bit floating-point arithmetic for most operations while maintaining 32-bit precision for operations requiring higher numerical precision. Modern accelerators are optimized to achieve an order of magnitude higher throughput with half precision arithmetic over full precision arithmetic [31, 30].

Compilation and kernel optimization can significantly improve computational efficiency. Systems like XLA [153] and TensorRT [135] optimize computational graphs and generate efficient kernels for specific hardware. Flash Attention [34] demonstrates how algorithm-hardware co-design can achieve significant speedups by optimizing memory access patterns for GPU hierarchies.

The computational requirements also vary significantly across different phases of training. The forward pass typically accounts for about 1/3 of the total computation, while the backward pass accounts for about 2/3 due to the need to compute gradients for all parameters [22]. Communication overhead can become a significant factor in distributed training, particularly with slower interconnects or larger model sizes.

2.2.4 Training Hardware

Modern deep learning relies heavily on specialized hardware accelerators designed for parallel matrix operations. GPUs have become the dominant training hardware due to their high computational throughput and memory bandwidth. NVIDIA's GPU architectures, from Pascal to the latest Hopper generation, have been specifically optimized for deep learning workloads with features like Tensor Cores for mixed precision operations [32].

GPU memory architecture significantly impacts training performance. High-bandwidth memory (HBM) provides the memory bandwidth necessary for feeding the thousands of cores in modern GPUs. Memory capacity limits the size of models that can be trained on a single device, with high-end GPUs like the H100 providing 80 GB of HBM3 memory. Memory bandwidth, rather than computational throughput, often becomes the bottleneck for many neural network operations, particularly those with low arithmetic intensity.

Interconnect technology plays a crucial role in multi-GPU training. NVIDIA's NVLink [138] provides high-bandwidth, low-latency communication between GPUs within a node, while InfiniBand and Ethernet provide inter-node connectivity. The bandwidth hierarchy, with intra-GPU bandwidth being much greater than intra-node bandwidth, which in turn is much greater than inter-node bandwidth, significantly influences the design of distributed training algorithms.

Specialized AI accelerators like Google's TPUs [82] and other domain-specific architectures offer alternative approaches to neural network acceleration. These devices are often optimized for specific operations like matrix multiplication and may provide better energy efficiency or cost-effectiveness for certain workloads.

2.3 Distributed Training

This chapter presents the communication and parallelization foundations used to scale DNN training across multiple accelerators and machines.

2.3.1 Communication Primitives and Collective Operations

Due to the high memory and computational demands of training DNNs, the process is typically distributed across a cluster of GPUs. Distributed training relies heavily on collective communication operations that coordinate data exchange across multiple workers.

These primitives form the foundation for implementing various parallelization strategies and directly impact training efficiency [137].

AllReduce is the most fundamental collective operation in distributed training. It computes a reduction (typically summation) across all workers and distributes the result to all participants. For gradient averaging in data parallelism, AllReduce enables each worker to obtain the global average gradient. The operation can be implemented using various algorithms, including ring-based approaches that achieve optimal bandwidth utilization with $O(\log P)$ latency for P participants [155].

AllGather collects data from all workers and makes the concatenated result available to all participants. This operation is essential for parameter sharding approaches like ZeRO, where workers must collect parameter shards before computation. The bandwidth requirement scales linearly with the number of participants, making it potentially expensive for large clusters.

Reduce-Scatter performs a reduction operation and distributes different parts of the result to different workers. This operation is often used as a building block for implementing AllReduce efficiently and is crucial for gradient sharding in ZeRO-2 [144].

Point-to-point communication enables direct data exchange between specific worker pairs and is essential for pipeline parallelism.

The efficiency of collective operations depends heavily on the underlying network topology and the algorithms used. Ring-based algorithms perform well on networks with limited bisection bandwidth, while tree-based algorithms can be more efficient on networks with higher connectivity. Hierarchical approaches that exploit the bandwidth hierarchy (intra-node vs. inter-node) are particularly important for modern clusters [65].

Communication libraries like NCCL [137], MPI [167], and Gloo [44] provide optimized implementations of these primitives for different hardware configurations. These libraries often include topology-aware algorithms that adapt to the specific network characteristics of the deployment environment. For example, NCCL will select between different ring-based and tree-based implementations based on the message sizes, topology, and available interconnects.

For example, consider training on two servers with eight GPUs each. NCCL can first reduce gradients within each server over high-bandwidth NVLink and then exchange aggregated results across servers over lower-bandwidth InfiniBand. By separating fast intra-node communication from slower inter-node communication, the collective reduces cross-node traffic and improves end-to-end step time.

2.3.2 Data Parallelism and Parameter Sharding

Data parallelism is the most widely used distributed training strategy, where each worker maintains a full copy of the model and processes different subsets of the training data [155]. After computing gradients on their local data, workers use AllReduce to average gradients across all participants, ensuring that all workers apply consistent parameter updates.

The main advantage of data parallelism is its simplicity and broad applicability to different model architectures. However, traditional data parallelism requires each worker to store the full model, limiting the size of models that can be trained based on the memory capacity of individual devices.

Parameter sharding techniques address this limitation by distributing model state (parameters, gradients, and optimizer states) across workers while maintaining the data parallel training paradigm. The ZeRO family of optimizations [144] provides a systematic approach to parameter sharding with three levels:

ZeRO-1 partitions optimizer states across workers, reducing memory consumption by the number of workers while requiring minimal additional communication. Each worker maintains full copies of parameters and gradients but only stores optimizer states for its assigned parameter partition.

ZeRO-2 additionally partitions gradients, further reducing memory consumption. During the backward pass, each worker accumulates gradients only for its assigned parameters, requiring a Reduce-Scatter operation to distribute gradient accumulation across workers.

ZeRO-3 (FSDP [213] in PyTorch) represents the most aggressive sharding approach, partitioning parameters themselves across workers. This requires AllGather operations to collect parameter shards before computation and parameter redistribution after computation. While ZeRO-3 provides the greatest memory savings, it also introduces the highest communication overhead since layer parameters need to be collected through AllGather collectives prior to the forward and backward pass. Parameters are resharded after computation such that only the parameters for the current layer and next layer being prefetched are fully materialized in memory.

Gradient compression techniques can reduce communication overhead in data parallelism by compressing gradients before AllReduce operations [103]. Methods include quantization, sparsification, and error feedback mechanisms that maintain convergence properties while reducing communication volume.

2.3.3 Model Parallelism Techniques

Model parallelism addresses the fundamental limitation of data parallelism by distributing the model itself across multiple workers, enabling training of models larger than the memory capacity of individual devices. Two primary forms of model parallelism have emerged: pipeline parallelism and tensor parallelism.

Pipeline Parallelism partitions a model into sequential stages, with each stage assigned to different workers [70, 128]. The global batch is divided into smaller microbatches that are pipelined through the stages, enabling parallel computation across stages while maintaining the sequential dependency of layer computations.

GPipe [70] introduced synchronous pipeline parallelism with gradient accumulation across microbatches. While simple to implement, GPipe suffers from significant pipeline bubbles during startup and shutdown phases, leading to low hardware utilization. PipeDream [128] addresses this limitation with asynchronous pipeline execution, maintaining multiple versions of parameters to handle the asynchrony between forward and backward passes.

The effectiveness of pipeline parallelism depends on load balancing across stages. Uneven computation across stages creates pipeline bubbles that reduce overall throughput. Additionally, pipeline parallelism requires careful memory management for storing intermediate activations and multiple parameter versions in asynchronous implementations.

Tensor Parallelism partitions individual operations within layers across multiple workers, enabling parallelization at a finer granularity than pipeline parallelism [161, 163]. For matrix multiplications $Y = XW$, tensor parallelism can partition along different dimensions: partitioning W column-wise requires an AllReduce after computation, while row-wise partitioning requires input replication.

Megatron-LM [131] demonstrates effective tensor parallelism for transformer models by carefully partitioning attention and feedforward operations to minimize communication requirements. The key insight is to alternate between column-wise and row-wise partitioning across layers to reduce the frequency of AllReduce operations.

Tensor parallelism requires high-bandwidth interconnects for efficiency due to frequent communication of intermediate results. It is typically most effective within nodes where NVLink or similar high-speed interconnects are available, while pipeline parallelism is better suited for inter-node communication over slower networks.

2.3.4 Hybrid Parallelism Strategies

Modern large-scale training systems combine multiple parallelization strategies to optimize resource utilization and training efficiency. 3D parallelism refers to the combination of data parallelism (DP), pipeline parallelism (PP), and tensor parallelism (TP) [131, 214].

The key challenge in hybrid parallelism is determining the optimal configuration for a given model, cluster, and performance target. Different parallelization strategies have different communication patterns, memory requirements, and computational characteristics that must be balanced.

Megatron-LM [131] employs heuristics to select the most appropriate parallelization strategies: tensor parallelism is used within nodes with fast interconnects, pipeline parallelism is applied across nodes with slower connections, and data parallelism is used for further scaling. This hierarchical approach aligns communication patterns with the bandwidth hierarchy of typical clusters.

Alpa [214] automates the search for optimal parallelization strategies using a combination of integer linear programming and dynamic programming. The system models the costs of different parallelization approaches and searches for configurations that minimize total training time while respecting memory constraints.

Other systems like DeepSpeed [146], Colossal-AI [98], and Metis [179] provide flexible frameworks for implementing various parallelization strategies. These systems often include automated tools for configuration selection and performance optimization using a mix of linear programming, dynamic programming, profiling, and heuristics.

The effectiveness of hybrid parallelism depends on careful load balancing and communication optimization. Uneven load distribution across any parallelization dimension can create bottlenecks that limit overall performance. For example, a worker with a larger workload than the others will take longer to finish its workload and delay synchronous communication collectives. Additionally, the interaction between different parallelization strategies can create complex communication patterns that require careful optimization.

2.4 Cloud Computing for Machine Learning

This chapter introduces cloud service models and orchestration frameworks that shape how machine learning training is provisioned, scaled, and priced in practice.

2.4.1 Cloud Service Models and Pricing

Cloud computing platforms provide machine learning practitioners with access to vast computational resources without the capital investment and operational overhead of maintaining dedicated infrastructure. The cloud service landscape for machine learning spans multiple service models, each offering different levels of abstraction and control [168].

Infrastructure as a Service (IaaS) provides the most fundamental level of cloud resources, offering virtualized computing resources including virtual machines, storage, and networking. For machine learning workloads, IaaS platforms like Amazon EC2 [156], Google Compute Engine [109], and Azure Virtual Machines provide direct access to GPU-enabled instances with full control over the software stack. Users can configure custom environments, install specific frameworks, and have complete control over resource allocation and scheduling.

Cloud VM pricing varies significantly and can substantially impact the cost-effectiveness of training large models. *On-demand pricing* provides immediate access to resources with per-hour billing, offering maximum flexibility but at the highest cost. *Reserved instances* require upfront commitments but provide significant discounts (often 30-60%) for predictable workloads with long-term resource needs.

Spot instances offer substantial cost savings (often 50-90% discounts) by utilizing excess cloud capacity, but come with the risk of interruption when demand increases. For fault-tolerant training workloads with checkpointing capabilities, spot instances can dramatically reduce training costs [129].

Serverless Computing represents a paradigm shift in cloud resource management, abstracting away infrastructure concerns while providing automatic scaling and pay-per-use billing models. *Serverless Functions* like AWS Lambda [159], Azure Functions [27], and Google Cloud Functions [107] were designed for stateless, short-duration tasks with limited resource requirements. These platforms provide automatic scaling, zero infrastructure management, and billing based on actual execution time and resource consumption. However, they impose significant constraints that make them challenging for distributed deep learning applications.

Key limitations of traditional serverless functions include strict execution time limits (typically 15 minutes), memory restrictions (up to 10 GB), absence of GPU support, and lack of direct inter-function communication capabilities. Functions must communicate indirectly through external storage services like AWS S3, introducing substantial latency overhead that is prohibitive for the frequent communication required in distributed training. The

stateless nature of functions also complicates maintaining training state and coordinating distributed workloads.

Despite these limitations, serverless computing offers compelling advantages for machine learning workloads: fine-grained resource allocation that can precisely match workload requirements, automatic scaling without explicit cluster management, and cost optimization through pay-per-use billing that eliminates charges for idle resources. These benefits make serverless platforms attractive for certain classes of machine learning tasks, particularly those with variable resource demands or intermittent execution patterns.

Serverless containers address some limitations of traditional serverless functions. Platforms like Azure Container Instances [3], AWS Fargate [158], and Google Cloud Run [108] provide serverless execution of containerized applications with more flexible resource allocation, longer execution times, and support for custom networking configurations. Unlike serverless functions, serverless containers can be provisioned with GPUs and configured with public IP addresses for direct communication, making them potentially suitable for distributed training workloads.

However, existing serverless container platforms still face challenges for DNN training. Most lack native support for distributed training frameworks, require custom orchestration for multi-container coordination, and do not provide optimized collective communication libraries. Additionally, cold start latencies can be significant for GPU-enabled containers, potentially impacting training efficiency during dynamic scaling operations.

ML as a Service (MLaaS) abstracts away infrastructure management while providing development and deployment platforms optimized for machine learning. Services like Google AI Platform [106], Amazon SageMaker [157], and Azure Machine Learning [28] provide a Cloud suite of industrial-strength tools for building, training and deploying machine learning pipelines. In particular, Azure ML [28] and AWS SageMaker [157] offer services for DNN training on a cloud-managed cluster of VMs. Users benefit from the convenience of serverless computing while also enjoying the performance of VM-based training. However, users are still susceptible to over-provisioning resources on MLaaS. Resources cannot be scaled dynamically, and must be provisioned in coarse-grained increments of a VM size, which come in limited configurations and are unlikely to fit the specific resource requirements of the workload at hand. For example, AWS VMs with the A100 come with exactly 8 GPUs, 96 vCPUs, and 1.1 TB of CPU RAM. It is unlikely that this particular configuration will perfectly match the resource requirements of a training job, resulting in users paying for extra resources they do not need. For example, DNN training workloads perform the bulk of their computations on the GPU. As a result, VM users may pay for bundled CPU resources that are underutilized during training.

2.4.2 Resource Elasticity and Management

One of the key advantages of cloud computing for machine learning is the ability to dynamically scale resources based on workload demands. Resource elasticity enables organizations to avoid over-provisioning infrastructure while ensuring adequate resources are available during peak training periods.

On the Cloud, users have a variety of compute platforms to choose from for deploying training jobs. The most common option is to reserve a cluster of virtual machines (VMs). However, using VM clusters comes with challenges in management and code deployment. Users must decide what type of resources and how many to provision. Moreover, cluster resources need to be scaled to accommodate changes in training workload. This requires significant effort and is mitigated in practice by spending extra money to over-provision resources so as to satisfy peak usage. Drivers and libraries need to be installed and machines need to be configured to communicate with each other prior to deploying code. These challenges detract deep learning practitioners from the task at hand and waste money on underutilized resources.

On the other hand, elastic scaling for distributed machine learning presents significant challenges to ML practitioners. Adding or removing workers during training requires reconfiguration of the parallelization strategy, redistribution of model state, and potentially adjustment of batch sizes. Traditional deep learning frameworks like PyTorch and TensorFlow do not natively support dynamic reconfiguration, requiring specialized systems or manual intervention [116].

Serverless computing represents an extreme form of elasticity where the cloud provider automatically manages all infrastructure concerns. Serverless functions abstract away the complexity of provisioning machines and deploying code. They can scale elastically without manual cluster management, and each function can be provisioned with fine-grained resources tailored to the workload at hand. Moreover, the customer pays for compute resources only when they are actively being used. However, serverless functions were primarily designed for stateless, lightweight tasks and hence are not optimized for training DNNs, which involves stateful, heavyweight processes. Serverless functions cannot communicate with each other directly and existing approaches communicate indirectly using external storage services like AWS S3 [74]. Since there is an extra hop required for serverless communication, communication becomes a performance bottleneck for DNN training workloads. Furthermore, serverless function platforms impose strict resource limits on memory, CPU, storage, and cannot use GPUs.

2.4.3 Containerization and Orchestration

Containerization has become essential for deploying machine learning workloads in the cloud, providing consistent execution environments across different infrastructure providers and simplifying dependency management. Containers encapsulate applications along with their dependencies, libraries, and configuration files, ensuring reproducible execution regardless of the underlying host environment.

Docker [121] has emerged as the dominant containerization platform for machine learning applications. Docker containers provide lightweight virtualization with minimal performance overhead compared to virtual machines. For deep learning workloads, specialized Docker images often include pre-configured frameworks like TensorFlow, PyTorch, and CUDA drivers, enabling rapid deployment without complex environment setup.

Container images for machine learning can be substantial due to the size of frameworks and dependencies. Optimization techniques include multi-stage builds to reduce image size, layer caching to accelerate builds, and base image selection to minimize security vulnerabilities and startup time. Container registries like Docker Hub, Amazon ECR, and Google Container Registry provide storage and distribution for container images.

Kubernetes [47] has become the standard for container orchestration, providing automated deployment, scaling, and management of containerized applications. For machine learning workloads, Kubernetes offers several key capabilities: resource management and scheduling, service discovery and load balancing, storage orchestration, and automated rollouts and rollbacks.

Kubernetes resource management enables fine-grained control over CPU, memory, and GPU allocation for containers. Resource requests and limits ensure predictable performance while enabling efficient resource sharing. GPU support in Kubernetes requires additional components like the NVIDIA GPU Operator and appropriate resource scheduling policies.

Kubeflow [25] extends Kubernetes with machine learning-specific capabilities, providing a comprehensive platform for ML workflows. Kubeflow includes components for model training (TFJob, PyTorchJob), hyperparameter tuning (Katib), model serving (KFServing), and pipeline orchestration (Kubeflow Pipelines). These abstractions simplify the deployment and management of complex ML workflows while leveraging Kubernetes' underlying orchestration capabilities.

2.5 Heterogeneous and Network-Aware Training

This chapter examines resource and network heterogeneity in cloud clusters and motivates training strategies that remain efficient under these non-uniform conditions.

2.5.1 Resource Heterogeneity in Cloud Environments

Cloud environments inherently exhibit resource heterogeneity across multiple dimensions, creating both opportunities and challenges for distributed machine learning. Understanding and effectively managing this heterogeneity is crucial for optimizing training efficiency and cost-effectiveness in cloud deployments.

Compute Heterogeneity arises from the diversity of processor architectures and generations available in cloud platforms. GPU offerings span multiple generations (e.g., K80, P100, V100, A100, H100) with vastly different computational capabilities, memory capacities, and architectural features. For example, H100 GPUs provide roughly 9× the training throughput of V100 GPUs for transformer models while offering 2.5× the memory capacity.

This compute heterogeneity enables cost optimization through strategic resource selection. Organizations can use high-performance instances for computation-intensive tasks while utilizing lower-cost instances for less demanding workloads. However, heterogeneous clusters introduce load balancing challenges since faster devices may be underutilized while waiting for slower devices to complete their assigned work.

Memory Heterogeneity manifests in both capacity and bandwidth differences across instance types. High-end GPU instances may provide 80 GB of HBM memory with 2 TB/s of bandwidth, while entry-level instances offer only 16 GB with proportionally lower bandwidth. This heterogeneity influences optimal model partitioning strategies and memory optimization techniques, as different devices can accommodate different model components.

Network Heterogeneity varies across multiple dimensions: intra-node connectivity (NVLink vs. PCIe), inter-node networking (InfiniBand vs. Ethernet), and geographic distribution (intra-region vs. inter-region). These differences can span multiple orders of magnitude in bandwidth and latency, significantly impacting communication-intensive distributed training algorithms.

Availability & Pricing Heterogeneity refers to the dynamic resource availability and pricing in cloud environments. The latest GPU instances are high in demand and often have limited availability [57], forcing users to use older generation hardware. Instance

availability varies by region and instance type, creating opportunities for both performance and cost optimization strategies.

Effective utilization of heterogeneous resources requires sophisticated training and planning algorithms that can adapt to the differing capabilities of available resources. Traditional homogeneous scheduling approaches often lead to significant resource underutilization in heterogeneous environments, motivating the development of heterogeneity-aware training systems.

2.5.2 Network Topology and Communication Efficiency

Network topology and communication efficiency play critical roles in distributed training performance, particularly in cloud environments where network characteristics can vary significantly across different deployment configurations. Understanding these factors is essential for designing communication-efficient training strategies.

Hierarchical Network Structure characterizes most modern cloud environments, with distinct performance tiers based on proximity and connection types. Intra-GPU communication within multi-GPU cards occurs over high-speed interconnects like NVLink, providing bandwidths of 400-600 GB/s with sub-microsecond latencies. Intra-node communication between GPUs on the same host typically uses PCIe or NVSwitch connections, offering 64-128 GB/s bandwidth with slightly higher latencies.

Inter-node communication within the same data center typically utilizes high-speed Ethernet or InfiniBand networks, providing 25-200 Gbps bandwidth with latencies in the tens of microseconds. Inter-region communication occurs over wide-area networks with significantly higher latencies (tens to hundreds of milliseconds) and more limited bandwidth.

Network Topology Awareness is crucial for optimizing communication patterns in distributed training. Ring-based AllReduce algorithms perform well on networks with limited bisection bandwidth, distributing communication load evenly across all links. Tree-based algorithms can be more efficient on networks with higher connectivity but may create bottlenecks at intermediate nodes.

Hierarchical communication strategies exploit the bandwidth hierarchy by performing reductions within high-bandwidth domains (e.g., within nodes) before communicating across slower links (e.g., between nodes). This approach can significantly reduce communication overhead for large-scale deployments [65]. RAT [182] adjusts hierarchical aggregation for AllReduce communication to consider the physical topology in addition to network oversubscription and failures.

Communication-Computation Overlap techniques hide communication latency by overlapping data transfers with computation. FSDP schedules AllGather and ReduceScatter communication such that parameters can be gathered and gradients reduced while workers continue computing. Pipeline parallelism can overlap communication between workers with computation within each stage, though careful scheduling is required to maximize utilization.

Fault Tolerance and Network Reliability become increasingly important as training scales to larger clusters with more potential failure points. Network partitions, temporary connectivity issues, and hardware failures can disrupt training if not properly handled. Robust communication protocols and checkpoint-restart mechanisms are essential for maintaining training progress in unreliable network environments [49, 78, 40].

The heterogeneous nature of cloud networks requires training systems to be adaptive and resilient to varying network conditions. Static optimization approaches that assume homogeneous network characteristics often perform poorly in real cloud deployments, motivating the development of dynamic, network-aware training strategies that can adapt to changing conditions and resource availability.

Chapter 3

Related Work

This chapter discusses challenges in deploying, managing, and training DNNs on the Cloud, as well as related work designed to address these challenges.

3.1 Dynamic Training Reconfiguration

The global batch size of inputs processed during every training iteration scales linearly with the number of workers. A larger batch size can be divided amongst more workers, allowing for more parallelization and enabling shorter training times to process the same number of input examples. However, it has been shown that larger batch sizes result in statistically inefficient training [54]. A larger batch size requires more epochs of training to reach a given accuracy and converges to a lower final accuracy. Adjusting the batch size during training can preserve the statistical efficiency of small batch sizes while achieving the training speed of large batch sizes [35, 116, 119]. For training CNNs, AdaBatch [35] achieves this by starting with a small batch size and increasing the batch size on a fixed schedule, doubling the batch size every 20 epochs. Similar strategies have been explored for training LLMs, where training with small batches early on helps stabilize optimization and improve generalization, while progressively increasing the batch size allows for efficient scaling on large hardware clusters [79, 95]. For example, Llama-3 [55] was trained in 3 stages with increasingly large batch sizes of tokens. However, popular frameworks [143, 174, 127] do not support dynamically configuring the batch size or parallelization strategy during training.

Systems like KungFu [116] and Tenplex [181] facilitate such strategies by dynamically adjusting the number of workers participating in training and the parallelism strategy

in order to maintain the desired batch size. However, these systems do not manage the dynamic scaling of the underlying cluster resources. For instance, KungFu operates on a fixed-size cluster of machines, which must be over-provisioned to accommodate the maximum number of workers required during training. On the flip side, frameworks like Terraform [61], Pulumi [33], and AWS CloudFormation [8] allow for easier deployment and scaling of cloud resources, but are not integrated with training frameworks to support dynamic reconfiguration during training. In Chapter 4, we present Hydrozoa, a system that can dynamically reconfigure training while adjusting cluster resources accordingly. This enables Hydrozoa to leverage cloud elasticity to reduce over-provisioning and improve training efficiency.

3.2 Serverless Computing for Machine Learning

Serverless computing represents a paradigm shift in cloud resource management, abstracting away infrastructure management while providing automatic scaling and pay-per-use billing models. However, traditional serverless platforms present significant challenges for distributed deep learning workloads due to their design assumptions and limitations (Section 2.4.2).

Despite these limitations, multiple systems have been designed to adapt serverless functions for machine learning workloads. In Siren [185], workers use S3 as a parameter server for data parallel training. Feng et al. [45] use multiple layers of parameter servers to reduce the data transfer time. Cirrus [16] reduces inter-worker communication latency by using a parameter server deployed on an EC2 cluster. Jiang et al. [74] compare the performance of serverless functions versus IaaS for a variety of ML tasks. They conclude that serverless functions are effective for only workloads requiring limited communication. This is generally not true for training DNNs, which can be large in size and require many rounds of communication until convergence. Hydrozoa is the first serverless system for DNN training that demonstrates performance competitive with GPU-based training on VMs for large DNNs.

Serverless computing has also been explored for other distributed workloads. PyWren [80] is a general purpose framework that allows users to run Python functions on serverless compute platforms. ExCamera [46] and Sprocket [10] demonstrated serverless video processing by decomposing video encoding tasks into thousands of parallel functions, achieving low-latency processing through careful workload partitioning. For linear algebra workloads, NumPyWren [160] and Pockets [87] provide distributed primitives and storage disaggregation capabilities in serverless environments. Starling [141] and Cloudburst [169]

explore serverless query execution, demonstrating the potential for database operations in function-as-a-service environments. Additionally, serverless approaches have been applied to IoT applications [21, 64], leveraging the automatic scaling and pay-per-use model for edge computing scenarios. Hydrozoa brings the benefits from serverless scalability and billing to DNN training.

3.3 Fault Tolerance and Elastic Training

As distributed training systems scale to larger clusters and longer training durations, fault tolerance and elastic scaling capabilities become critical for maintaining training progress and optimizing resource utilization. This section examines approaches for handling failures and dynamic resource changes in distributed training environments.

Checkpointing and Recovery Mechanisms form the foundation of fault tolerance in distributed training. Traditional approaches save complete model state, optimizer state, and training metadata at regular intervals, enabling recovery from arbitrary failures. However, checkpointing large models can introduce substantial overhead, particularly when using network-attached storage or object storage systems.

Recent advances in incremental checkpointing [42] reduce overhead by saving only modified state components, while distributed checkpointing approaches [101] parallelize checkpoint operations across multiple storage devices. Some systems explore in-memory checkpointing across multiple nodes to reduce I/O overhead while maintaining fault tolerance [188].

Spot Instance Utilization presents unique challenges for fault tolerance due to the preemptible nature of spot instances. While spot instances can provide 50-90% cost savings compared to on-demand instances, they can be terminated with minimal notice when demand increases. Effective spot instance utilization requires sophisticated fault tolerance mechanisms that can handle frequent preemptions.

Bamboo [175] and Parcae [40] provide frameworks for training on spot instances with automatic migration and recovery capabilities. These systems must balance checkpoint frequency with training efficiency, as frequent checkpointing can significantly impact performance while infrequent checkpointing increases the amount of work lost during preemptions.

Elastic Training Systems enable dynamic addition and removal of workers during training, allowing systems to adapt to changing resource availability and optimize cost-performance trade-offs. However, elastic scaling introduces significant technical challenges related to state migration, load balancing, and maintaining convergence properties.

Traditional deep learning frameworks like PyTorch and TensorFlow do not natively support dynamic reconfiguration, requiring specialized systems for elastic training. KungFu [116] provides elastic data parallelism with support for dynamic batch size adjustment and worker scaling. Tenplex [181] extends this approach to support multiple parallelization strategies and more sophisticated scaling policies.

Model Convergence is impacted by the global batch size of inputs processed during each training iteration, which in turn depends on the number of workers and parallelization strategy. For example, in data parallelism, the global batch size scales linearly with the number of workers. A larger batch size enables training with more workers in parallel and hence leads to shorter training times to process the same number of input examples.

However, it has been shown that larger batch sizes result in statistically inefficient training [54]. A larger batch size requires more epochs of training to reach a given accuracy and converges to a lower final accuracy. McCandlish et al. [119] coin the term critical batch size, referring to the largest batch size beyond which further increasing batch size gives diminishing returns in terms of statistical training efficiency. They propose the gradient noise scale, which is computed based on the covariance of the gradient and the gradient, to estimate the critical batch size. Adjusting the batch size during training can preserve the statistical efficiency of small batch sizes while achieving the training speed of large batch sizes [35, 116, 119]. For example, AdaBatch [35] starts at a small batch size and gradually increases the batch size on a fixed schedule, gradually supporting more parallelism as the training progresses. Hydrozoa enables ML practitioners to integrate batch size scaling methods like AdaBatch into distributed training without needing to handle the complexities of cluster scaling.

3.4 Performance Optimization in Cloud Training

The substantial computational requirements of modern DNNs, combined with the high cost of GPU resources, have made performance optimization a critical consideration for training DNNs in practice.

Many systems assume a fixed cluster configuration and optimize the training configuration to minimize training time and cost [131, 128, 214, 144]. Narayanan et al. [129] explore the potential cost reductions possible with dynamic resource allocation by leveraging the variable pricing of spot instances. They show that the optimal instance choice for a model is dependent on both the objective (e.g., cost, time, or combination) and the model's performance characteristics. They also show that dynamically migrating training jobs is an

effective way to leverage the fluctuating cost and availability of spot instances. Yang et al. [196] use linear programming techniques to reduce the cost of training jobs using spot instances by up to 50% over on-demand instances.

Sailor [172] is a system that automatically determines the set of compute resources to allocate for a given training job in order to optimize for a given object like cost or throughput. Other works investigate the cost-efficiency of cross-region resource deployments for DNN training [171, 43]. Beyond performance, sustainability-aware serving has emerged as a complementary objective. As DNN training continues to exponentially grow in scale, the environmental impact and sustainability of training is becoming an important consideration, including energy utilization and the operational and embodied carbon across GPU generations and workloads. Several works explore how to measure and model the energy and carbon footprint of DNN training [197, 132, 104, 114, 193]. Carbon Explorer [4] is a framework that enables users to design more carbon efficient datacenters. EmbAdvisor [176] optimizes LLM caching during model serving in order to reduce carbon emissions associated with storage. Zeus [197] and Perseus [23] are frameworks that find pareto-efficient training configurations along the energy-latency trade off curve.

3.5 Memory Management and Optimization Techniques

Modern deep neural networks, particularly large language models, require extensive memory optimization techniques to overcome the challenge that model sizes often exceed the memory capacity of individual accelerators. This chapter examines the various approaches developed to address memory constraints in distributed training.

Training State Sharding Strategies are commonly used to reduce memory utilization during training. The ZeRO (Zero Redundancy Optimizer) family of techniques [144] (equivalently, FSDP [213] in PyTorch) eliminates training state memory redundancy in data parallel training. ZeRO-1 partitions optimizer states across workers, ZeRO-2 extends this approach by additionally partitioning gradients, using Reduce-Scatter operations to distribute gradient accumulation across workers. ZeRO-3 further partitions model parameters themselves and requires AllGather operations to collect parameter shards before computation.

ZeRO++ [183] extends ZeRO-3 to support hierarchical sharding strategies. For example, model parameters can be sharded within a node, where GPUs are connected with higher bandwidth interconnects, while parameters are replicated across nodes, avoiding extra inter-node communication.

Memory Offloading Techniques extend memory capacity by utilizing the memory hierarchy beyond GPU memory. ZeRO-Offload [2] moves optimizer computations to CPU, reducing GPU memory requirements while leveraging CPU resources for optimization steps. More aggressive offloading strategies [146] can move parameters, gradients, and optimizer states to CPU memory or even persistent storage when not actively needed [134].

Cephalo [57] (Chapter 5) and Zorse [59] (Chapter 6) offload activation checkpoints and gradients to CPU memory, avoiding an accumulation in memory utilization when increasing the batch through gradient accumulation. Memory transfers are scheduled such that they can be overlapped with computation, reducing memory utilization with minimal training overhead.

Activation Memory Optimization addresses the substantial memory requirements of storing intermediate activations during the forward pass. Gradient checkpointing (also known as activation checkpointing) [131] trades computation for memory by selectively storing only a subset of activations and recomputing others during the backward pass. The optimal checkpointing strategy depends on the model architecture and available memory, with recent work developing automatic checkpointing algorithms [62]. Beaumont et al. [12] propose a hybrid optimization that jointly optimizes gradient checkpointing and offloading to reduce memory utilization with minimal training overhead.

Sequence-parallel approaches [99] distribute activations across multiple devices for transformer models, exploiting the fact that attention computations can be parallelized across the sequence dimension. This approach is particularly effective for long sequences where activation memory can dominate parameter memory.

Mixed Precision Training [125] reduces memory consumption by using 16-bit floating-point representations for most computations while maintaining 32-bit precision for operations requiring higher numerical precision. Automatic mixed precision frameworks dynamically determine the appropriate precision for each operation, maintaining training stability while reducing memory utilization.

Recent developments in quantization techniques [209] explore even lower precision representations, including 8-bit and 4-bit training, though these approaches often require careful handling of numerical stability and convergence properties.

3.6 Automated System Configuration and Optimization

The complexity of modern distributed training systems, with their numerous configuration parameters and trade-offs between memory, computation, and communication efficiency, has motivated the development of automated optimization approaches that can navigate the vast configuration space more effectively than manual tuning.

Search-Based Parallelization systems attempt to automatically discover optimal parallelization strategies for a given model and cluster configuration. Alpa [214] employs a hierarchical approach, using integer linear programming for inter-device parallelization and dynamic programming for intra-device optimization. The system models computation, memory, and communication costs to search for configurations that minimize total training time while respecting memory constraints.

Galvatron [124] focuses specifically on transformer models, using decision trees for search space pruning followed by dynamic programming to find efficient hybrid parallelization strategies. The system considers multiple parallelization dimensions simultaneously, including data, pipeline, and tensor parallelism, and can adapt to different memory budgets and cluster sizes.

FlexFlow [73] provides a more general framework for automatic parallelization that supports arbitrary DNN architectures. The system uses a combination of graph optimization techniques and performance modeling to discover efficient parallelization strategies, including support for flexible parallelization patterns that combine multiple parallelism types within the same model.

Accurate **Performance Modeling** for computation time, memory usage, and communication overhead is critical for systems to find efficient parallelization strategies while pruning infeasible configurations. Existing approaches use analytical models, machine learning models, and a combination of both to achieve high prediction accuracy across diverse hardware configurations and model architectures.

Analytical models typically predict performance based on a smaller set of profiled values. For example, a model for predicting ring all-reduce latencies is used to predict communication latency based on profiled bandwidth values [171, 59]. Hydrozoa, Cephalo, and Zorse use analytical models to predict memory utilization, and training latency based on the latencies and number of parameters in individual layers of the model. Other techniques use machine learning to directly predict model latency [50, 207, 17]. While this avoids the overhead of profiling, they have difficulty generalizing to unseen model architectures.

Habitat [199] uses a combination of analytical models and machine learning models to predict DNN kernel latencies, extrapolating performance to unseen hardware configurations.

3.7 Heterogeneous GPU Clusters

Many distributed training systems assume a homogeneous GPU cluster, dividing compute and memory demands equally. However, most organizations lack large homogeneous clusters due to frequent GPU release cycles, high upgrade costs, GPU shortages, and limited cloud availability [40, 122, 190, 173, 171, 76]. As a result, organizations often assemble clusters with GPUs from different generations to achieve higher compute power in aggregate. Thus, training on heterogeneous clusters has gained attention as it allows organizations to leverage all available GPU resources for training [139, 179, 72, 210, 194, 85].

Existing training strategies designed for homogeneous clusters are susceptible to under-utilizing GPU resources on a heterogeneous cluster since faster GPUs will be idle while waiting to synchronize with slower GPUs. Systems like Whale [72] and other approaches [126, 178] propose to mitigate bottlenecks in data parallelism by assigning uneven batch sizes to GPUs based on their relative compute speed, ensuring that faster GPUs process a greater portion of the data. However, with this strategy a GPU with fast compute but relatively low memory may run out of memory before it can process a large enough batch size to fully saturate its compute capabilities, leading to suboptimal resource utilization.

In pipeline parallelism [128, 139], balancing compute latency across stages is crucial. The stages run in parallel, so the slowest stage bottlenecks the pipeline. In homogeneous clusters, dividing the layers evenly across stages is effective for balancing computation since transformer layers are typically identical [131]. In heterogeneous clusters, layers can instead be partitioned based on the relative compute speed of their GPUs, such that faster GPUs process more layers. However, achieving an efficient partition that balances compute may not be possible, as the fastest GPUs may lack sufficient memory to handle the layers required to maximize their compute capacity, while slower GPUs might fully utilize their compute capacity but leave a significant portion of their memory underutilized.

HAP [210] distributes workloads unevenly in data and tensor parallelism to align with GPU compute capacities, though it still assumes faster GPUs have more memory. Additionally, tensor parallelism requires high-bandwidth GPU interconnects for efficiency, which are unlikely to be available in heterogeneous clusters with lower-end GPUs. Metis [179] and FlashFlex [194] integrate heterogeneous data, pipeline, and tensor (3D) parallelism, offering greater flexibility for heterogeneous training configurations but inherit limitations of each parallelism type.

Recent advances in heterogeneous training include more sophisticated workload balancing approaches. Cannikin [133] provides optimal adaptive batch size selection for heterogeneous clusters, using accurate performance modeling to predict optimal configurations. The system addresses the challenge that directly adapting existing techniques for homogeneous clusters results in poor utilization in heterogeneous environments.

3.8 Heterogeneous Networks

In addition to heterogeneity in compute and memory, heterogeneity in networking is also prevalent in Cloud environments, regardless of whether homogeneous or heterogeneous GPUs are used [128, 131]. Intra-region networking is typically much faster than inter-region networking due to machine proximity. Intra-node networking between GPUs can be orders of magnitude faster than inter-node networking in the presence of high-speed interconnects like NVSwitch [138]. Finally, different VMs may connect GPUs with different underlying networking hardware, which can result in underutilization of network links. For example, p4d [7] instances on AWS with A100 GPUs are connected to other p4d instances with up to 400 Gbps networking. However, g5 [6] instances with A10G GPUs are equipped with only 100 Gbps networking. Moreover, GPUs within each p4d instance are connected with high bandwidth NVLink, which is not present in g5 instances. Thus, collective communication on a cluster of both g5 and p4d instances can be bottlenecked by the slower networking on g5 instances.

Cross-Cloud and Multi-Region Training presents an environment of network heterogeneity when resources span multiple cloud providers, data centers, and geographic regions. Cross-region training [171, 200, 43] explores training DNN models across geographically distributed resources with varying network characteristics. These approaches must balance networking bandwidths and latencies that differ by orders of magnitude across different GPUs.

Prior systems like Megatron-LM [131], PipeDream [128], and Alpa [214] consider heterogeneity in networking when selecting an optimized parallelism strategy. However, these systems do not support balancing compute and memory across heterogeneous GPU clusters.

HexiScale [194] and Metis [179] are designed to explicitly consider heterogeneity in compute and memory and offer more flexible configuration of parallelization strategies. For example, while frameworks for homogeneous clusters assume a fixed degree of data, pipeline, and tensor parallelism across the clusters, they allow for varying degrees of parallelism across different subsets of GPUs. This is important for heterogeneous clusters where different

GPUs have different VM configurations and benefit from different parallelization strategies. However, data parallelism in these frameworks do not shard the model parameters, which can require a prohibitively large amount of memory for larger models. Hence, they rely on pipeline parallelism, which has limited scalability, and tensor parallelism, which is extremely bandwidth-intensive to reduce memory usage.

Zorse [59] uses a novel integration of data and model parallelism, which is able to jointly resolve challenges in balancing compute and memory across heterogeneous GPUs while minimizing communication overhead in heterogeneous network environments.

Several works scsale training over multiple datacenters with asynchronous training [200, 39, 48], relaxing semantics of SGD. While these techniques can be effective from a hardware utilization perspective, asynchronous training typically results in worse model performance compared to synchronous training.

3.9 Training-Adjacent Workloads

Beyond training, many systems have also been developed for DNN inference and cluster scheduling for DNN jobs on the Cloud. DNN inference is consuming an increasingly large share of compute resources, particularly given the widespread usage of LLMs and development of test-time compute scaling techniques to improve model performance [189, 166]. Inference systems like Shepherd [204] and AlpaServe [100] optimize hardware utilization and model serving goodput (i.e. amount of requests served per second satisfying a latency constraint). vLLM and SGLang are Inference systems developed to tackle the unique inference challenges of LLMs, using optimizations like paged attention and KV caching [91, 215].

Several works optimize LLM serving on heterogeneous GPUs [120, 77]. Splitwise [140] and Lamina [18] leverage the relative performance advantages in heterogeneous GPUs to split the prefill and decode phases of LLM serving. GreenLLM [162] combines higher-end with lower-end GPUs to reduce carbon emissions for LLM serving.

Large clusters are often shared across many tenants within an organization in order to improve resource utilization and amortize costs. In such environments it is important for cluster scheduling algorithms to be able to schedule many concurrent jobs such that the cluster is utilized efficiently while maintaining fairness across users [201]. Various systems such as Tiresias and Themis [56, 115] optimize job order and placement in order to achieve these objectives. Other systems such as Gavel and Sia optimize cluster scheduling to consider heterogeneous compute but restrict each job to homogeneous GPU resources,

precluding heterogeneous training for a single job [5, 173, 130, 191]. Cephalo and Zorse could be integrated with these cluster schedulers to improve utilization in shared heterogeneous clusters.

Chapter 4

Leveraging Cloud Elasticity with Serverless Training

Recall from Section 2.4.2 that DNN training jobs are typically deployed on the Cloud through VMs, serverless functions, or on end-to-end managed ML-as-a-service (MLaaS) platforms. However, these options are prone to over-provisioning resources in the case of VMs and MLaaS, or are not well suited for training workloads in the case of serverless functions.

Moreover, models are often trained in stages, where the training configuration (e.g., batch size, parallelism strategy) and resources utilized (i.e. number of GPUs) change between stages. However, existing training frameworks either do not support changes to the training configuration [143, 174, 127] or do not handle the adjustment of cluster resources between stages [181, 116].

In light of the limitations from prior work, we designed a system called Hydrozoa [58]. This chapter focuses on the contributions of Hydrozoa, which simultaneously supports dynamically modifying the training configuration and scales the cluster resources to support these changes leveraging the elasticity of the cloud.

4.1 Design

Hydrozoa trains DNN models with a combination of data and model parallelism while using serverless containers to dynamically scale the number of worker GPUs participating in training. With this architecture, Hydrozoa is able to avoid the traditional pitfalls of

serverless DNN training, including inefficient communication and limited compute resources. At the same time, it leverages the benefits of serverless computing: fine-grained scalability, zero cluster management overhead, and a pay-per-use billing model. The rest of this chapter describes the main components of Hydrozoa's design.

4.1.1 Serverless Containers

Containers run applications in a virtual environment, packaged with their libraries and dependencies. Azure Container Instances (ACI) [3] is a platform that allows users to execute containers in a serverless fashion, much like serverless functions. A user submits a container along with a set of resource requests, and ACI will provision the requested resources and manage the execution of the container on those resources. Users are charged based on the GPU, CPU and memory resources used for the execution of the container on a per second basis. Hence, ACI enables code to be executed with as much ease, scalability and cost-efficiency as serverless functions. Moreover, ACI does not have the same stringent start-up requirements as serverless functions, and can be configured with both GPUs and a public IP for communication. This approach is referred to as *serverless containers*. Hydrozoa is built on top of serverless containers in ACI. The following subsections describe why this design choice is particularly suitable for DNN training.

GPU Compute: DNN training is much faster on GPUs compared to CPUs since DNN computations are highly parallelizable, making it a highly suitable workload for exploiting the high compute throughput and memory bandwidth in GPUs. Serverless containers in ACI can be provisioned with GPUs.

Direct Communication: Serverless function platforms do not support direct communication between functions. Functions must rely on external storage systems such as S3 [152] or Redis [147] to indirectly exchange information, which incurs significant latency. This is problematic for distributed DNN training, which requires frequent communication between workers. This is not an issue in ACI, since containers can communicate with each other directly through public IP addresses.

Fine-Grained Scaling: In ACI, containers can be provisioned resources at a fine-grained level. Memory can be configured in GB increments; GPUs and vCPUs can be scaled independently. Hydrozoa right-sizes resource allocations to fit the specific compute requirements of each job, saving costs by avoiding resource over-provisioning.

Pay-Per-Use: ACI charges for resources only while they are in use, and on a per-second basis. On the other hand, VMs reserved on IaaS are billed regardless of whether or not

	GPUs	Direct Communication	Fine-Grained Scaling	Pay Per-Use	Management Free
VM (IaaS)	✓	✓	×	×	×
VM (MLaaS)	✓	✓	×	✓	✓
Serverless Functions	×	×	✓	✓	✓
Serverless Containers	✓	✓	✓	✓	✓

Table 4.1: Platform comparison for DNN training.

they are actively being used, and often on a per-hour basis. The pay-per-use billing model provides yet another avenue in which costs can be cut.

Management Free: Training DNNs on IaaS requires provisioning and configuring VMs, launching the distributed tasks, and deprovisioning the VMs when done. This process is time-consuming and prone to human error, especially when launching multiple training jobs or scaling resources on-the-fly. Hydrozoa automates this process on ACI.

Table 4.1 summarizes the availability of these features across different platforms. Serverless containers are the only platform type that supports all of these features.

4.1.2 Hybrid-Parallelism

Hydrozoa trains DNNs with hybrid-parallelism – a combination of data- and model-parallelism. Figure 4.1 illustrates this training architecture on a model partitioned into 3 shards. Workers 1-1, 1-2, and 1-3 are each responsible for training one model shard. They form a *worker group*, training one copy of the model with model-parallelism. Data-parallelism is then added on top of model-parallelism by instantiating multiple such worker groups. Workers training the same shard across different worker groups synchronize model updates with each other using *AllReduce* [155], a communication collective that aggregates data from all workers and sends the result back to each worker. The *parallelism strategy* in Figure 4.1 corresponds to 2× data-parallelism applied on top of 3-shard model-parallelism. This is denoted as 2×3 for short and this notation is used in the rest of this chapter. Since model-parallelism is a special case of hybrid-parallelism where there is no data-parallelism, and vice-versa, Hydrozoa is also able to train models with pure data-parallelism or model-parallelism. Hydrozoa's *planner* optimizes the parallelism strategy for each training job (Section 4.1.4).

Model Partitioning: To perform model-parallel training, Hydrozoa first partitions the layers of a model M into shards $p_1, p_2, \ldots, p_k$ such that evaluating the model is equivalent to

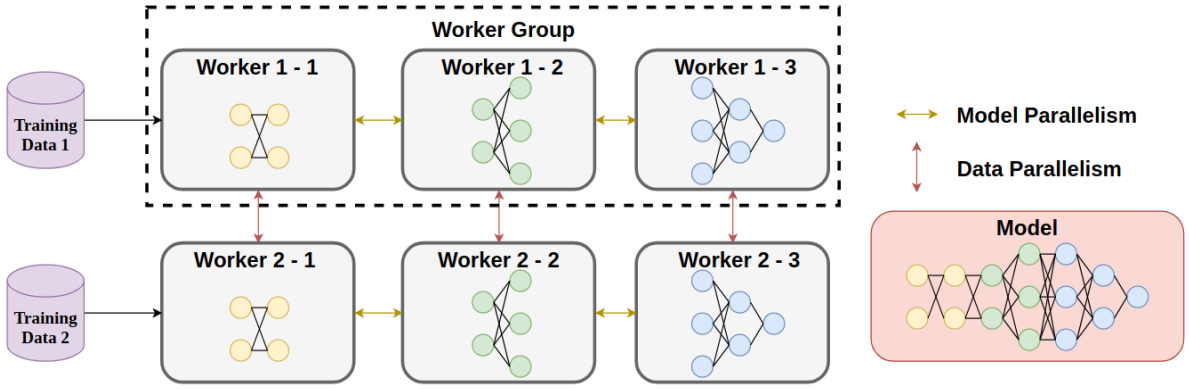


Figure 4.1: Hybrid-Parallelism in Hydrozoa

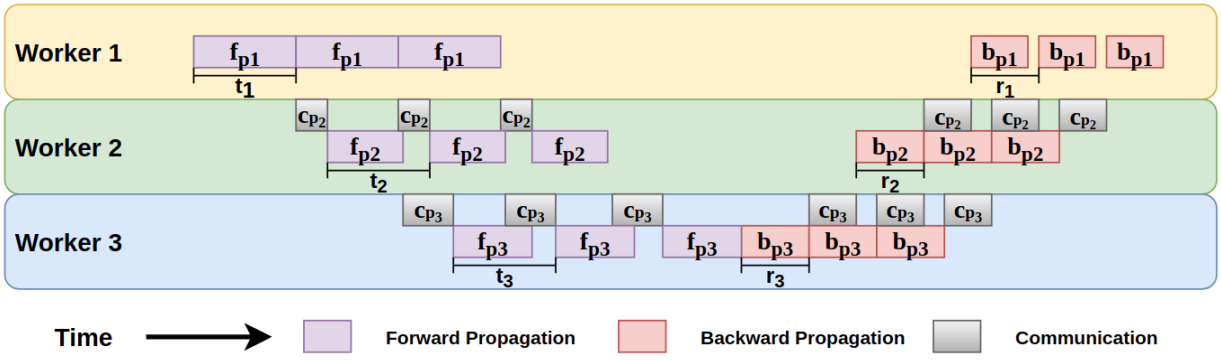


Figure 4.2: Pipelined Model-Parallelism in Hydrozoa

evaluating the composition of the shards. In other words, $M = p_k \circ \ldots \circ p_2 \circ p_1$. The model partition can have a significant impact on the speed of model-parallel training. However, finding an effective partition manually is difficult since there are an exponential number of partitions to consider. Moreover, the performance of a partition is dependent on a variety of factors including hardware, networking, and model architecture. Part of Hydrozoa's planner includes a partitioning algorithm that produces partitions optimized to run on a target environment.

Pipelined Model-Parallelism When model-parallelism is implemented naively, sequential dependencies in forward and backward propagation permit only one worker to perform computation at a time. Instead, pipeline-parallelism is employed which splits a minibatch of inputs into (smaller) microbatches [70, 128], then pipelines microbatches across the workers.

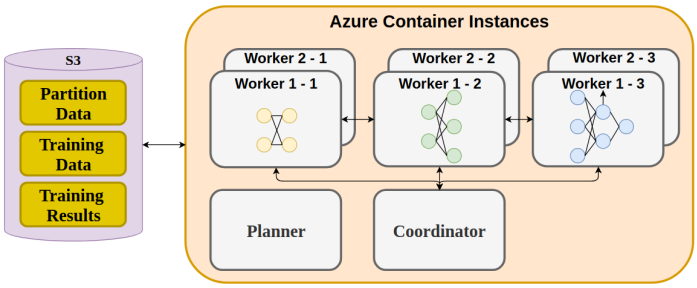


Figure 4.3: Hydrozoa Architecture

The pipeline is further optimized with workers prefetching activation and gradient inputs asynchronously during computation.

Figure 4.2 demonstrates pipelining for a model partitioned into 3 shards and a minibatch of data split into 3 microbatches. The pipeline starts on worker 1 with forward propagation on shard 1 with microbatch 1. The results can be passed to worker 2, which begins forward propagation on shard 2 with microbatch 1. At the same time, worker 1 can begin forward propagation on microbatch 2. Similarly, during backward propagation, worker 2 can begin backward propagation with microbatch 1 while worker 3 performs backward propagation with microbatch 2.

With the prefetching optimization, workers receive activation and gradient values as soon as they become available. With a well-balanced model partition, forward and backward propagation can start on subsequent microbatches without communication delay. In Figure 4.2, the gradients for microbatch 2 are transferred from worker 3 to worker 2 while worker 2 is doing backward propagation on microbatch 1. Consequently, worker 2 is able to start backward propagation on microbatch 2 immediately after microbatch 1.

4.1.3 Partitioning Algorithm

Hydrozoa's partitioning algorithm aims to partition the model into shards $p_1, p_2, \ldots, p_k$ in a way that minimizes the total training time of a minibatch of inputs. To minimize training time, the partitioning algorithm ensures that computation is evenly distributed across each worker while minimizing the data communicated between workers. Partitions that evenly distribute the computation enable efficient microbatch pipelining, and minimizing the amount of data exchanged avoids communication bottlenecks.

Profiling The partitioning algorithm first profiles information about the model's per-layer computation time, output sizes, and network speed. Let the profiled forward and backward propagation times for layer i be f_i and b_i. Then, for any shard p_l that spans layers x through y, the forward and backward propagation time of p_l can be estimated as $f_{p_l} = \sum_{i=x}^{y} f_i$ and $b_{p_l} = \sum_{i=x}^{y} b_i$. The time c_{p_i} to transfer activations and gradients between shard p_{i-1} and shard p_i is estimated to be $a_{p_{i-1}}/D$, where $a_{p_{i-1}}$ is the activation size of the last layer in shard p_{i-1} and D is network bandwidth.

Model-Parallelism Runtime Model Using the profiled information, the runtime for model-parallelism is modeled. Suppose there are k workers for shards $p_1, p_2, ..., p_k$. The forward propagation of a minibatch finishes when worker k finishes forward propagation.

It can be seen from Figure 4.2 that worker k starts forward propagation at time $\sum_{i=1}^{k-1} f_{p_i} + \sum_{i=2}^{k} c_{p_i}$. In this figure, notice that in worker 2 there can be a slight delay between the forward propagation of microbatches. This delay occurs whenever a worker needs to wait for the activations from a previous worker. In fact, the time elapsed between the start of two consecutive forward microbatches in worker 2 equals the forward time of a microbatch in worker 1. This can be generalized. Denote the time elapsed between two consecutive forward microbatches in worker i as t_i (as labeled in Figure 4.2). In general, $t_i = \max(\max_{1 \le j \le i} f_{p_j}, \max_{2 \le j \le i} c_{p_j})$ – which is equivalent to the maximum of forward propagation and communication times encountered up to the i^{th} shard. If the number of microbatches is M, it follows that worker i will require $(M-1)t_i + f_{p_i}$ time to perform forward propagation. Hence, worker k finishes forward propagation at time $\sum_{i=1}^{k} f_{p_i} + \sum_{i=2}^{k} c_{p_i} + (M-1)t_k$; this is also when the forward propagation of the minibatch finishes.

Similarly, denote the time elapsed between consecutive backward microbatches in worker i as r_i. It follows that $r_i = \max(\max_{i \le j \le k} b_{p_j}, \max_{i+1 \le j \le k} c_{p_j})$ and by symmetry, the backpropagation of the minibatch requires a total time of $\sum_{i=1}^{k} b_{p_i} + \sum_{i=2}^{k} c_{p_i} + (M-1)r_1$.

Hence, given any partition, the total runtime (forward plus backward propagation) for

a minibatch on that partition can be modeled with the equation

$$\sum_{i=1}^{k}(f_{p_i}+b_{p_i})+2\sum_{i=2}^{k}c_{p_i}+(M-1)(t_k+r_1).$$

Algorithm The partitioning algorithm aims to produce the partition that minimizes the runtime of processing a minibatch of inputs. This runtime is modeled with the equation described above, which incorporates communication time, computation time, and pipeline stalls. t_k and r_1 model the time between consecutive batches of forward and backward propagation.

$$\sum_{i=1}^{k}(f_{p_i}+b_{p_i})+2\sum_{i=2}^{k}c_{p_i}+(M-1)(t_k+r_1).$$

Dynamic programming is used to find a partition that minimizes this runtime model.

Since $\sum_{i=1}^{k}(f_{p_i}+b_{p_i})$ is constant for all partitions, it suffices to minimize $2\sum_{i=2}^{k}c_{p_i}+(M-1)(t_k+r_1)$.

It is difficult to optimize this term jointly due to the interdependencies between c_{p_i}, t_k, r_1. However, if t_k, r_1 were fixed, the partition that minimizes $2\sum_{i=2}^{k}c_{p_i}$ (communication time) could be found through dynamic programming. The optimal partition must then be one of the partitions produced from minimizing the communication time for some t_k, r_1 combination. However, there are an exponential number of t_k, r_1 combinations to consider, so to reduce the search space, Hydrozoa picks V evenly spaced values as bounds for t_k and similarly, another V bounds are used for r_1. We denote the bounds for t_k as $B_1^{(1)}, ..., B_1^{(V)}$, where $B_1^{(j)} \in [\min_{1\le i\le L} f_i, \max(\sum_{i=1}^{L} f_i, \max_{2\le i\le L} c_i)]$ for all $1 \le j \le V$, where c_i is the communication time between layer $i-1$ and i. Notice that $[\min_{1\le i\le L} f_i, \max(\sum_{i=1}^{L} f_i, \max_{2\le i\le L} c_i)]$ is the interval of all possible values for t_k. The bounds for r_1 are denoted as $B_2^{(1)}, ..., B_2^{(V)}$ and are defined in a similar fashion.

For each combination of $B_1^{(i)}$ and $B_2^{(j)}$, the partitioning algorithm uses dynamic programming to find the partition that minimizes communication time while satisfying $t_k \le B_1^{(i)}$ and $r_1 \le B_2^{(j)}$. In the dynamic programming algorithm, $C(i, j)$, iteratively solved, denotes

the minimal communication time for a partition of the first j layers in the model with i shards. Using the optimal subproblem property, it follows that

$$C(i,j) = \min_{i-1 \leq u \leq j-1} C(i-1,u) + c_{u+1}.$$

In the end, all partitions found from every combination of $B_1^{(i)}$ and $B_2^{(j)}$ are compared, and Hydrozoa selects the partition with the lowest estimated minibatch runtime.

Notice that the dynamic programming algorithm computes the partition with minimal communication for each possible partition size en-route to the final answer. In addition to the overall best solution, the partitioning algorithm will also keep track of and return the best solution per partition size.

The dynamic programming algorithm returns the overall best partition found in addition to the best partition found for each possible partition size. This allows the planner to consider partitions of different sizes when choosing a hybrid-parallelism strategy.

Theorem 1 *Hydrozoa's partitioning algorithm can produce a partition with runtime arbitrarily close to the optimal solution. Proof: In Appendix A.1.*

Selecting Size and Number of Microbatches The optimal partition depends on the microbatch size and the number of microbatches. Hydrozoa performs an exponential search to select these hyperparameters since they can have a big impact on performance but are tedious to tune. The search generates partitions for microbatch sizes in increasing powers of 2, until they no longer fit in GPU memory. Pipelining is more efficient as the number of microbatches increases. Thus, for each microbatch size, binary search is used to maximize the number of microbatches within the GPU memory limits. The partitioning algorithm then returns the best partition produced across all the sizes and number of microbatches considered.

4.1.4 Planner

Hydrozoa's planner considers the exponential search space of hybrid-parallel training strategies and selects one that is optimized for the training workload, VM type, and desired number of VMs to use for training. The planner assumes a fixed global batch size and optimizes the hybrid-parallel training configuration under this constraint. As a result, the choice of parallelism strategy affects training efficiency but not model convergence.

The planner first uses the partitioning algorithm to produce the most efficient partitions for each partition size. For each such partition, the planner considers the hybrid-parallelism strategies using that partition for model-parallelism, and maximizes the level of data-parallelism while not exceeding the desired VMs. The planner uses a model for hybrid-parallel runtime to select the strategy with the lowest estimated runtime.

Hybrid-Parallelism Runtime Model When processing a minibatch with hybrid-parallelism, Hydrozoa first runs model-parallelism within each worker group and then synchronizes weights across worker groups through data-parallelism. Thus, Hydrozoa's model for hybrid-parallelism runtime combines the runtime model for model-parallelism described earlier, with a runtime model for data-parallelism.

Suppose a model is being trained with parallelism-strategy A×B (data-parallelism of A and model-parallelism of B). Worker w_{ij} corresponds to the worker on the i^{th} worker group processing the j^{th} partition. Moreover, let D be the network bandwidth, $|p_k|$ be the size of partition p_k in bytes, and α, β be model parameters.

The runtime model from the partitioning algorithm is used to compute the model-parallelism runtime of w_{ij} as $T_{ij}^{(M)}$.

The following equation is used to model data-parallelism (implemented in Hydrozoa with ring allreduce) runtime:

$$T_{ij}^{(D)} = (B-1)(\frac{|p_j|}{B \cdot D} + \alpha\frac{|p_j|}{B} + \beta).$$

The term $\frac{|p_j|}{B \cdot D}$ models the data transfer time for each of the $2(B-1)$ rounds in allreduce. $\alpha\frac{|p_j|}{B} + \beta$ models any other overhead that occurs during every round of communication, such as serialization and deserialization. The planner learns α, β through linear regression after collecting some runtime statistics on randomly generated inputs.

Thus, the total runtime of w_{ij} is then

$$T_{ij} = T_{ij}^{(M)} + T_{ij}^{(D)}.$$

Then the entire round of hybrid-parallelism ends when all workers finish, which is modeled as

$$T = \max_{i,j} T_{ij}.$$

Algorithm 1 Parallelism Strategy Selection

```
1: procedure SelectHybridStrategy(clusterSize)
2:     bestStrategyTime ← inf
3:     for 1 ≤ mp ≤ clusterSize do
4:         dp ← ⌊clusterSize/mp⌋
5:         runtime ← runtimeModel(dp, mp)
6:         if runtime < bestStrategyTime then
7:             bestStrategyTime ← runtime
8:             bestStrategy ← (dp, mp)
       return bestStrategy
```

Selecting Parallelism Strategy Algorithm 1 shows how the planner selects a hybrid-parallelism strategy for a desired cluster size based on the runtime model. It considers all possible combinations of data- and model-parallelism that fit within the cluster size and returns the strategy with the lowest estimated runtime.

4.1.5 Dynamic Worker Scaling

Hydrozoa allows users to specify policies for adjusting the number of workers at each epoch boundary. Hydrozoa adjusts serverless resources dynamically in response to decisions to scale up or scale down workers, and coordinates workers such that they are aware of changes in the training topology. Users can use worker scaling in Hydrozoa to implement strategies that improve statistical training efficiency [35, 119]. Dynamic worker scaling would be difficult to implement otherwise, since the user would need to manually scale the cluster, and popular ML frameworks such as MXNet and PyTorch assume the number of workers are fixed throughout training.

4.1.6 API

Hydrozoa exposes an API for data loading, planning, hybrid-parallel training, and worker scaling. It was used to train image-classification and language models in Section 4.3.

4.2 Hydrozoa Implementation

Hydrozoa's implementation consists of a (1) worker task, (2) planner, and (3) coordinator. Worker tasks execute training logic for a model shard and run on containers in ACI. The planner and coordinator also run as separate ACI containers. The coordinator manages the worker tasks throughout the training process. Training metadata and results are stored on S3. Figure 4.3 illustrates Hydrozoa's architecture.

4.2.1 Worker

Each worker is part of a worker group and assigned a model shard. The worker executes model-parallelism within its worker group and data-parallelism with the corresponding shard in other worker groups. The machine learning library MXNet is used to implement the training process.

Communication Workers exchange activations and gradients with each other through ZeroMQ [1], a high-speed messaging library. ZeroMQ is chosen since it is lightweight, heavily optimized, and offers convenient abstractions for implementing complex distributed communication patterns. Each worker is connected to the coordinator for querying network topology information and neighbouring workers for exchanging data during hybrid-parallel training.

Pipelined Model-Parallelism with Prefetching Activations for microbatches are cached during forward propagation so that they can be reused in backward propagation. Separate threads on each worker handle the transfer of gradient and activation values asynchronously with forward and backward propagation. Likewise, a separate thread is responsible for prefetching training data.

4.2.2 Planner

The planner runs on ACI so that profiled computational and network statistics are consistent with the actual training environment. After determining the model partitioning and hybrid-parallel strategy selection, the planner collects memory utilization statistics for each shard, which is used by the coordinator for resource allocation decisions. In practice, the planner runs within a few minutes in all of the experiments, which is an insignificant amount of time compared to the overall training time of the models.

Task	Model	Parameters	FLOPs / Input
Image Classification	AlexNet	57M	0.7G
	ResNet-34	21.5M	3.7G
	ResNet-152	58.5M	11.3G
	VGG19	143.7M	19.6G
Question Answering	BERT-12	108.9M	65.3G

Table 4.2: A comparison of the models used for evaluation.

4.2.3 Coordinator

The Coordinator orchestrates the training job, running as a container on minimal resources in ACI. It is responsible for starting worker tasks with the appropriate configuration based on model partition information and training hyperparameters. The coordinator right-sizes the memory allocation for workers based on data profiled from the planner. Workers share their IP and port information with the coordinator that forwards this information to other workers that need it.

Dynamic Worker Scaling When dynamic worker scaling is enabled, the coordinator is responsible for adjusting provisioned resources as the number of workers change. GPU containers on ACI can take several minutes to start up, so the coordinator will pre-start workers to ensure they are ready when actually needed. The coordinator collects information about worker startup time and training speeds to anticipate when to pre-start workers. The coordinator also informs existing workers about new workers so that they are aware of changes in network topology for allreduce and are assigned new subsets of the dataset.

4.3 Experiments

This chapter evaluates the performance of Hydrozoa on DNN training for image classification and question answering tasks. In Section 4.3.2, it is demonstrated why Hydrozoa is superior to existing systems built on serverless functions. In Section 4.3.3, it is shown that Hydrozoa is able to improve on VM-based training throughput while costing significantly less through hybrid-parallelism and fine-grained resource allocation. Finally, in Section 4.3.4, the performance benefits achievable through dynamic worker scaling are explored.

	vCPU	Memory (GB)	K80	V100
CPU Container	$0.04050	$0.00445	N/A	N/A
GPU Container	$0.03165	$0.00425	$0.36	$2.8548

Table 4.3: Cost per hour for ACI resources used by Hydrozoa.

Cluster	VMs	GPU	Cost / Hour
A	16 × Azure NC6	K80	$14.40
B	16 × AWS p2.xlarge	K80	$18.00
C	4 × Azure NC6s v3	V100	$12.24
D	1 × AWS p3.8xlarge	4 × V100	$14.69

Table 4.4: Cluster A, C for Azure ML and B, D for SageMaker.

4.3.1 Experimental Setup

Models: Hydrozoa is evaluated on training 5 popular models for image classification and question answering – AlexNet [90], ResNet-34, ResNet-152 [63], VGG19 [164], and BERT-12 [36] – featuring varying sizes and computational requirements shown in Table 4.2.

Datasets: The CIFAR-10 [89] dataset is used for image classification and SQuAD [145] for question answering. CIFAR-10 images are scaled to standard size $3 \times 224 \times 224$ images.

Clusters: Table 4.4 outlines the details of the clusters used for the experiments. Each cluster has either 16× K80 GPUs, or 4× of the relatively more powerful V100 GPUs. Since there were no VMs on AWS with a single V100 GPU, the p3.8xlarge instance was used, which contains 4× V100 GPUs.

Batch Sizes & Evaluation Methodology: Data-parallel training is configured to use the largest per-GPU minibatch that fits within memory to optimize utilization of the GPUs. For hybrid-parallel training, Hydrozoa selects the microbatch size and number of microbatches processed per worker group.

The main evaluation metrics used for system comparison are *throughput* and *throughput-per-dollar*. Throughput (T) is measured as the number of inputs processed per second. Throughput-per-dollar is defined as the number of inputs the system can process per dollar spent, i.e., T/C, where C is the per-second cost of running the system. Throughput is used

to compare the raw speed of each system, while throughput-per-dollar enables comparing performance normalized by cost. Results for system comparisons are averaged over at least 3 independent experimental runs. Costs are computed based on current Cloud pricing in the US East region (Tables 4.3 and 4.4).

4.3.2 Comparison to Training on Serverless Functions

Hydrozoa breaks away from the limitations of serverless functions to improve performance significantly over these benchmarks:

(i) **Hydrozoa-Lambda (H-L)**: A re-implementation of Hydrozoa using serverless functions. H-L is built on Lambda, passes data through S3, and is representative of state-of-the-art serverless training systems [74]. Each worker is configured to have the maximum compute of 6 vCPUs and 10 GB RAM, which cannot be scaled independently on Lambda.

(ii) **Hydrozoa-CPU (H-C)**: Hydrozoa, but limited to training on CPUs. Each worker is configured to have 4 vCPUs, which is the maximum for containers without GPUs. This runs on serverless containers, and can benefit from direct communication between workers.

(iii) **Hydrozoa-GPU (H-G)**: Hydrozoa, using a single worker with a K80 GPU and 4 vCPUs.

Data-parallelism throughput and throughput-per-dollar are compared for H-L and H-C as the number of workers are scaled from 1 to 16, on ResNet-34 in Figure 4.4. The performance of H-G with a single worker is also plotted as a baseline. The cost of ACI resources used by Hydrozoa are given in Table 4.3. Note that ACI charges more for vCPU and memory resources on CPU-only containers. For Lambda, each worker provisioned with 10 GB of RAM costs $0.60 per hour.

Hydrozoa-Lambda (H-L) vs Hydrozoa-CPU (H-C) As the number of workers increases, the throughput gap between H-L and H-C widens since H-L scales poorly. With 16 workers, the throughput of H-C is 1.34× higher than H-L for ResNet-34 (Fig. 4.4). Poorer scaling can be attributed to differences in communication. Messages in H-L are exchanged indirectly through S3, requiring higher latency than direct communication. This inefficiency is amplified as the number of workers increases, since communication during

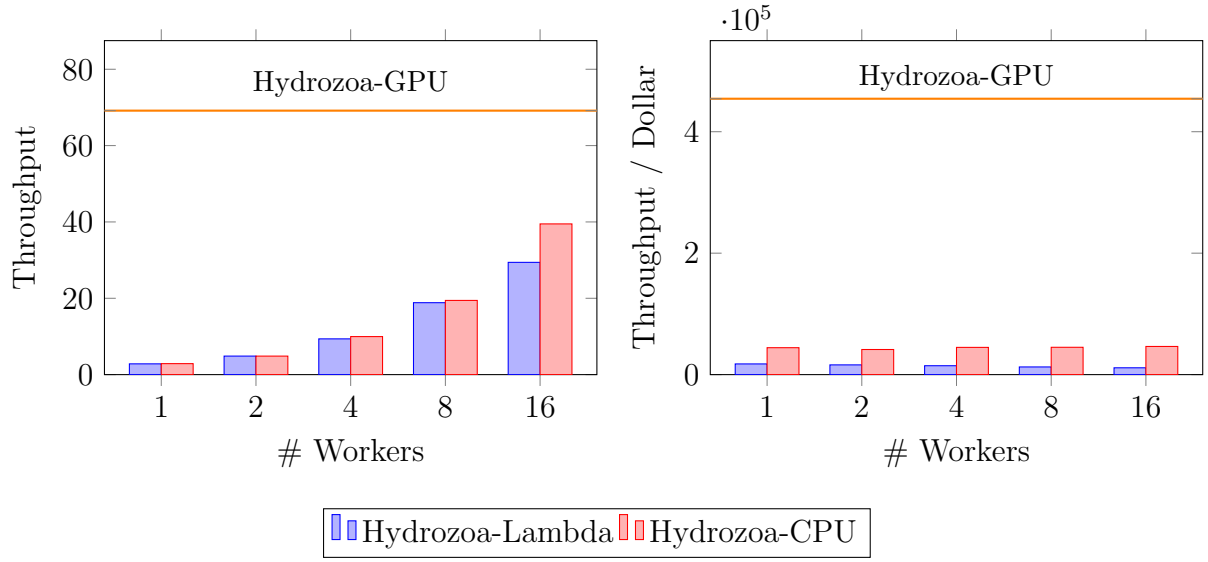


Figure 4.4: Hydrozoa ablation study on ResNet-34. Raw throughput (left) and throughput-per-dollar (right).

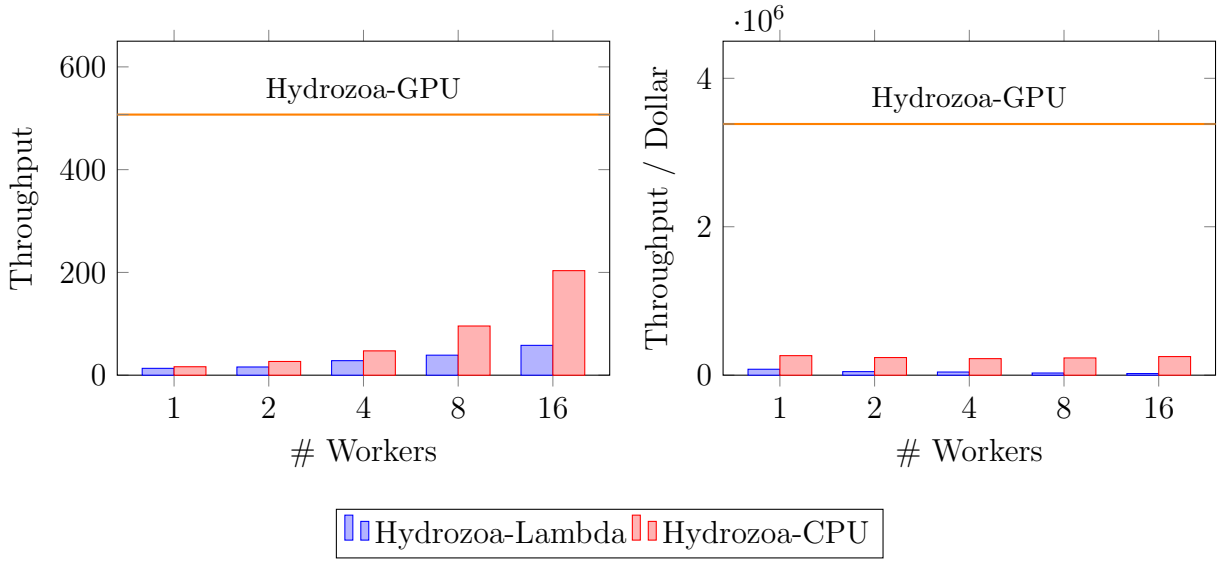


Figure 4.5: Hydrozoa ablation study on AlexNet. Raw throughput (left) and throughput-per-dollar (right).

allreduce increases with the number of workers. This is also echoed in throughput-per-

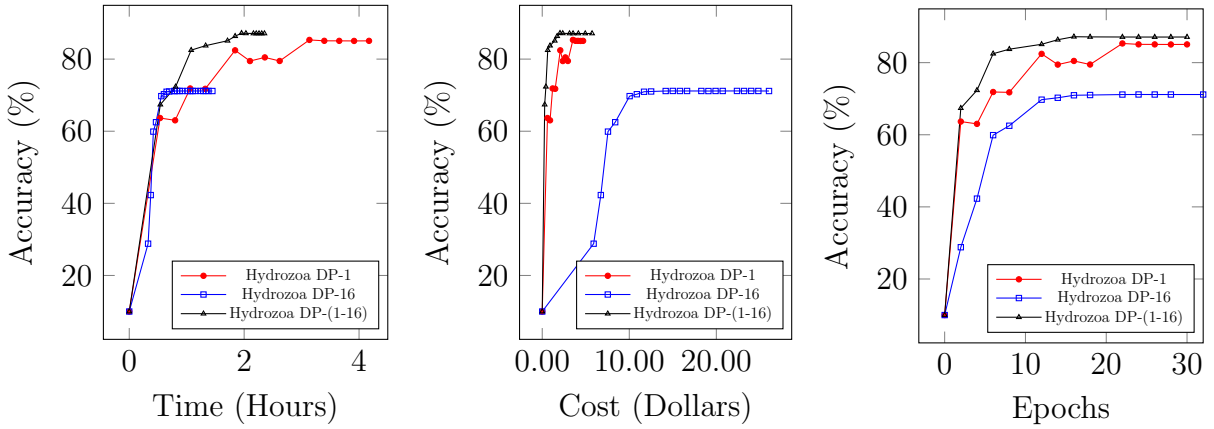


Figure 4.6: Training curves for Hydrozoa with dynamic worker scaling versus fixing the number of workers.

dollar. While the throughput-per-dollar increases for H-L with an increase in workers, it stays relatively constant for H-C due to near-linear scaling. On AlexNet (Fig. 4.5), the throughput scaling of H-L is even worse. This is not surprising since AlexNet is relatively large in size compared to computation – it is almost 3 times as large as ResNet-34 in size but requires 80% fewer FLOPs to train. With 16 workers, the throughput of H-C is 4.6× higher than that of H-L on AlexNet. Cheaper resources on Azure ML over Lambda yield even higher throughput-per-dollar improvements of 4.1× and 11.5× for these models.

Hydrozoa-CPU (H-C) vs Hydrozoa-GPU (H-G) Comparing H-C to H-G demonstrates the benefit of using GPUs over CPUs. H-G with a single K80 GPU delivers 1.8× and 2.5× higher throughput than H-C with 16 workers for ResNet-34 and AlexNet, respectively. This performance also comes at significantly lower cost – compared to H-C with 16 workers, the throughput-per-dollar of H-G is 9.8× and 13.5× higher. From *both* a performance and cost perspective, GPUs should be the choice over CPUs to accelerate DNN training.

These experiments show why Hydrozoa is superior to state-of-the-art DNN training systems on serverless functions. Hydrozoa is able to scale better with an increasing number of workers due to more efficient communication. Moreover, by training on GPUs instead of CPUs, Hydrozoa is able to further boost throughput while reducing costs. As a result, H-G is able to train ResNet-34 and AlexNet with throughput-per-dollar values that are 40.4× and 155.5× higher, respectively, than H-L.

Model	Hydrozoa Strategy	# Microbatches × Microbatch Size	Compared System	Compared Strategy	Throughput Speedup	Cost Reduction	Throughput / Dollar Improvement
ResNet-34	DP - 16×1	1×256	Azure ML	DP - 16×1	1.03×	1.76×	1.81×
			SageMaker	DP - 16×1	1.03×	2.19×	2.26×
ResNet-152	HP - 4×4	16×4	Azure ML	DP - 16×1	1.03×	1.70×	1.75×
			SageMaker	HP - 8×2	1.05×	2.15×	2.25×
VGG19	HP - 8×2	70×2	Azure ML	DP - 16×1	1.98×	1.63×	3.23×
			SageMaker	DP - 16×1	1.70×	2.04×	3.47×
BERT-12	HP - 4×4	13×4	Azure ML	DP - 16×1	2.05×	1.64×	3.36×
			SageMaker	DP - 16×1	2.66×	2.05×	5.43×

Table 4.5: Summary of results comparing Hydrozoa to VM-based training on Azure ML and AWS SageMaker on K80 clusters. Strategy refers to the data × model parallelism employed. DP and HP correspond to data- and hybrid-parallelism, respectively. For example, 8×2 means the model is partitioned into 2 shards, and data-parallelism is applied on 8 copies of the model.

4.3.3 Comparison to Training on VMs

Hydrozoa is compared to VM-based training in Azure ML and AWS SageMaker:

(i) Azure ML: Distributed training is run in PyTorch with Horovod [155], which, like Hydrozoa, employs data-parallelism with allreduce.[1]

(ii) AWS SageMaker: Training is also performed with PyTorch on SageMaker in order to use its optimized distributed training API, which supports data-parallelism, model-parallelism, and also hybrid-parallelism.

Azure ML and SageMaker are MLaaS services on Azure and AWS, so the deployment and execution of the distributed training jobs are managed by their respective cloud service providers. There should be no performance difference between training on Azure ML/SageMaker and running the equivalent training logic on a self-managed cluster of VMs provisioned from IaaS. In fact, training performance is expected to be better on SageMaker because it is able to take advantage of custom distributed training libraries that are industrial-strength and optimized for AWS' network infrastructure. The results on a cluster of 16× K80 GPUs are summarized in Table 4.5. Hydrozoa achieves better throughput and throughput-per-dollar through a combination of fine-grained resource allocation and hybrid-parallelism.

Unlike Hydrozoa, Azure ML and SageMaker do not select a parallelism strategy for the user. All parallelism strategies for these systems were tried and the results obtained

[1]Azure ML does not support MXNet.

Model	Hydrozoa Strategy	# Microbatches × Microbatch Size	Compared System	Compared Strategy	Throughput Speedup	Cost Reduction	Throughput / Dollar Improvement
VGG19	HP - 1×4	34×4	Azure ML	DP - 16×1	1.89×	1.02×	1.91×
			SageMaker	DP - 16×1	0.86×	1.21×	1.04×
BERT-12	MP - 1×4	24×2	Azure ML	DP - 4×1	3.66×	1.02×	3.71×
			SageMaker	DP - 4×1	0.95×	1.22×	1.15×

Table 4.6: Summary of results comparing Hydrozoa to VM-based training on Azure ML and AWS SageMaker on V100 clusters. Strategy refers to the data × model parallelism employed. DP and HP correspond to data- and hybrid-parallelism, respectively.

from their best-performing configuration were compared to Hydrozoa. The results for K80 clusters are summarized in Table 4.5.

Hydrozoa achieves better throughput and throughput-per-dollar through a combination of fine-grained resource allocation and hybrid-parallelism. The contribution of each of these factors is broken down next.

Results for V100 clusters are provided in Table 4.6. Since V100 has more compute power than K80 GPUs, evaluation is performed on the computationally-intensive VGG19 and BERT-12 models. As in the K80 experiments, Hydrozoa achieves higher throughput-per-dollar than Azure ML and SageMaker on both models. Since the AWS p3.8xlarge VM contains 4 GPUs locally, it is able to leverage NVLink for extremely high throughput GPU-to-GPU communication. Consequently, SageMaker was able to outperform Azure ML.

Fine-Grained Resource Allocation ResNet-34 is a relatively small model with low computational requirements, so data-parallelism is more effective than model- and hybrid-parallelism for all systems. Figure 4.7 compares the throughput and throughput-per-dollar as the number of workers used for data-parallelism are scaled. Hydrozoa's throughput scales similarly to the VM-based benchmarks, since they all use the same parallelism strategy. However, Hydrozoa attains this performance at much lower cost due to its fine-grained resource allocation. For example, the p2.xlarge VM in SageMaker is the smallest sized VM containing a K80 GPU, yet it comes with 61 GB RAM. Since training occurs on the GPU, only a fraction of that memory is used. Hydrozoa uses profiling data to allocate no more resources than needed – a K80, 4 VCPUs, and 6 GB of RAM per worker. Consequently, each worker in Hydrozoa costs \$0.512/hour compared to more than double the price tag of \$1.125/hour for each SageMaker VM. With these cost savings, Hydrozoa achieves 1.81× and 2.26× higher throughput-per-dollar than Azure ML and SageMaker, respectively.

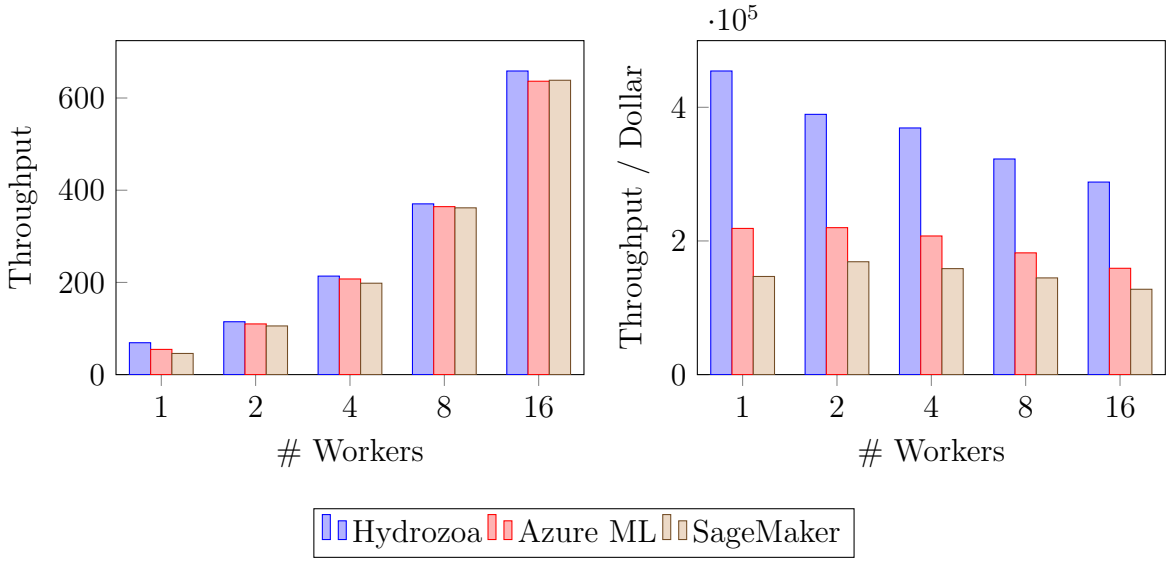


Figure 4.7: Comparison of data-parallelism performance on Hydrozoa, Azure ML and SageMaker for ResNet-34.

Hybrid-Parallelism: ResNet-152, VGG19, and BERT-12 are much larger models than ResNet-34, causing communication to be a bottleneck in data-parallel training. Hydrozoa mitigates this issue by using hybrid-parallelism to greatly reduce the communication overhead while balancing computation evenly across workers. Azure ML supports only data-parallelism. On the other hand, SageMaker supports hybrid-parallelism but it was ineffective for training any of the models other than ResNet-152.

Figure 4.8 compares the throughput of Hydrozoa to SageMaker for model- and hybrid-parallelism on BERT-12. The left graph compares the two systems as the partition size for model-parallelism increases. Hydrozoa's partitioning algorithm is able to produce partitions that scale throughput effectively up to 4 shards. 8 shards improves the throughput marginally over 4. SageMaker's model-parallelism cannot scale well to even 2 shards. In fact, all of the configurations involving model-parallelism on SageMaker performed worse than the trivial partition of size 1, suggestive of their partitioning algorithm being unable to find an effective partition. Since hybrid-parallelism applies data-parallelism on top of model-parallelism, it is not surprising that hybrid-parallelism in SageMaker also performs worse. The graph on the right in Figure 4.8 compares the throughput on the different hybrid-parallel configurations that fit on 16 GPUs. Hydrozoa achieves the highest throughput with 4× data-parallelism applied on top of 4-shard model-parallelism. Using less than 4× data-parallelism is inefficient due to the relative communication overhead incurred from

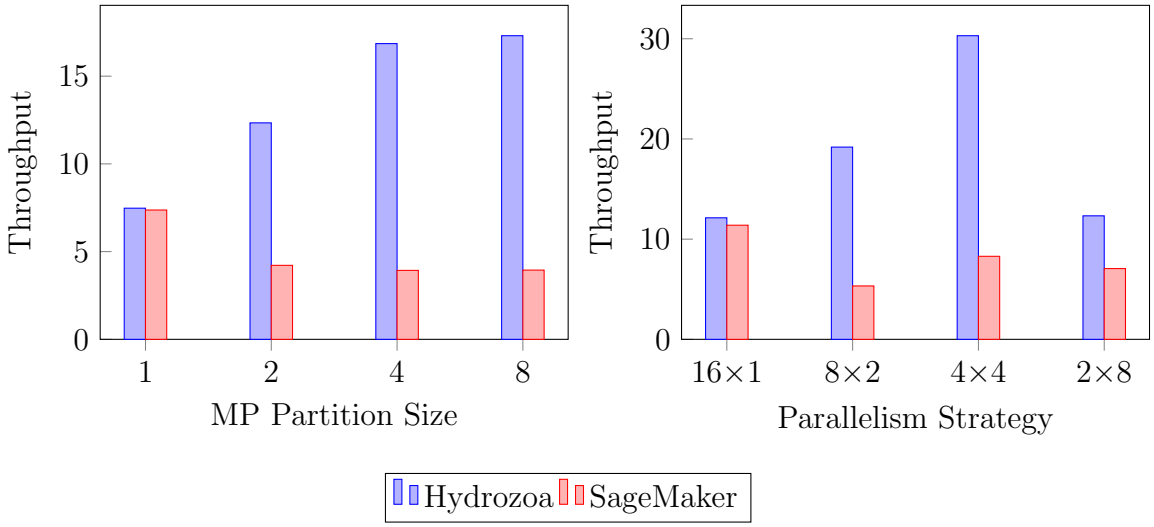


Figure 4.8: Comparison of throughput for BERT-12 on Hydrozoa and SageMaker for different parallelism strategies.

allreduce. On the contrary, with more than 4 shards, the benefit from model-parallelism is marginal, and so further scaling with data-parallelism is more effective. In SageMaker, using pure data-parallelism (16×1) was the fastest configuration.

Thus, Hydrozoa is able to achieve 1.70× and 2.66× higher throughput than SageMaker for training VGG19 and BERT-12, respectively. Hydrozoa provides an even larger improvement in throughput-per-dollar at 3.47× and 5.43×, because it factors in the cost savings from fine-grained resource allocation. Compared to Azure ML, Hydrozoa improves throughput-per-dollar by 3.23× and 3.36×. Hydrozoa's cost reduction from Azure ML is still significant, yet slightly less than AWS since Azure VMs are cheaper.

4.3.4 Dynamic Worker Scaling

This chapter shows how dynamic worker scaling in Hydrozoa can achieve performance gains and cost savings over prior VM-based training systems which either do not support dynamically adjusting parallelism during training or do not manage the scaling of resources. An adaptation of AdaBatch [35] was implemented in Hydrozoa in which the equivalent batch size scaling is achieved by increasing the number of workers and not the per-worker minibatch size.

ResNet-34 is trained on CIFAR-10 with the Adam Optimizer [86], an initial learning

rate of 10^{-3} and a minibatch size of 256 per worker. The baselines are data-parallelism with 1 (DP-1) and 16 workers (DP-16). All experiments are on Hydrozoa so that the performance results are attributed solely to worker scaling and not affected by system differences. The learning rate is decayed every 5 epochs by a factor of 0.375 to avoid stagnation during training. This is compared to DP-(1-16), which starts with 1 worker, and gradually scales to 16 workers. Instead of decaying the learning rate by a factor of 0.375, the batch size is doubled and the learning rate is decayed by a factor of 0.75 [165]. The batch size is doubled by doubling the number of workers, which also doubles the parallelism.

The training curves are shown in Figure 4.6. While DP-16 maximizes training speed, the statistical efficiency is poor. It reaches a lower accuracy for the same number of epochs processed compared to DP-1 and converges to a lower accuracy of 71% vs 85%. On the other hand, DP-1 achieves good statistical efficiency but is slow because there is only one worker training. DP-(1-16) achieves the best of both approaches. By scaling the number of workers, it can maintain the same statistical efficiency as DP-1 while reaching the training speed of DP-16. Thus, DP-(1-16) converges to an accuracy of 85% in about 45% less time than DP-1. Moreover, since DP-(1-16) is more statistically efficient than DP-16, it converges to an accuracy of 71% in fewer epochs than DP-16 and at 96% lower cost. While other scaling strategies can be implemented, this experiment demonstrates that dynamic worker scaling in Hydrozoa can bring significant benefits in training efficiency and cost.

4.4 Summary

In this chapter Hydrozoa was presented, a serverless system for training DNNs. Hydrozoa overcomes existing limitations of serverless DNN training with a novel architecture that combines serverless containers with hybrid-parallel training and supports dynamic worker scaling. Hydrozoa attains higher throughput at a lower cost, achieving throughput-per-dollar improvements of up to 5.4× over existing VM-based training and 155.5× over serverless approaches while relieving the user from the burden of managing machine clusters.

Chapter 5

Training on Heterogeneous GPU Clusters

Recall from Section 3.7 that while training systems are typically designed for homogeneous GPU clusters, most organizations do not have access to large homogeneous clusters due to the high costs of owning GPUs and limited GPU availability on the Cloud. Cloud platforms like AWS offer VMs with a variety of GPU models, but due to high demand, each model is available only in limited quantities. Figure 5.1 plots a trace of GPU availability on AWS over a 12-hour period in the us-west region. High-end GPUs (A100, H100) are almost always unavailable, and even mid-tier GPUs (A10G, V100, T4) are limited due to capacity and quota. Thus, it is challenging to reserve a large homogeneous cluster of GPUs.

By assembling *heterogeneous* clusters with different GPU models, users can circumvent this limitation and leverage a larger pool of compute resources for training. However, existing systems are unable to utilize resources efficiently in heterogeneous clusters. Existing systems for training on homogeneous clusters divide the training workload evenly across GPUs

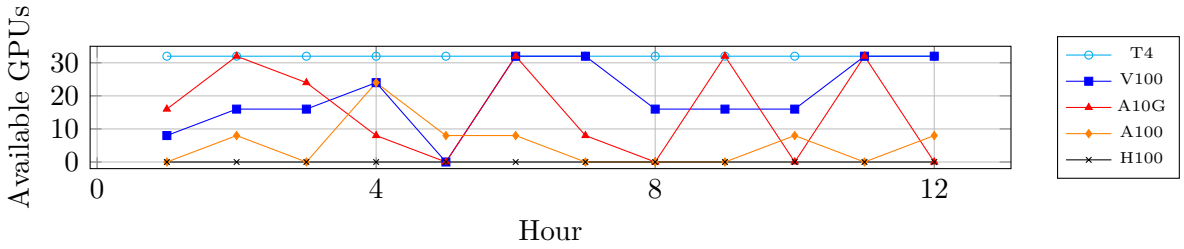


Figure 5.1: AWS GPU availability (12-hour period).

[214, 102, 128], which leads to faster GPUs being bottlenecked by slower GPUs when waiting to synchronize model updates.

Existing systems designed for training on heterogeneous clusters attempt to balance the workload by assigning a batch size or model shard proportional to the computational speed on each GPU [194, 72, 179, 210]. However, this requires faster GPUs to have more memory to process a larger workload, which is not always the case. As shown in Figure 5.2, GPU memory does not scale proportionally with compute speed. Thus, existing systems are susceptible to both: (i) underutilizing compute on GPUs with low memory capacity relative to compute speed, and (ii) underutilizing memory on GPUs with high memory capacity relative to compute speed.

In light of the limitations from prior work, we designed Cephalo [57], a system capable of effectively utilizing the aggregate compute and memory resources in heterogeneous GPU clusters when training transformer models. This work focuses on transformer models because they are a particularly memory and compute intensive class of DNNs which are also widely used in a variety of applications.

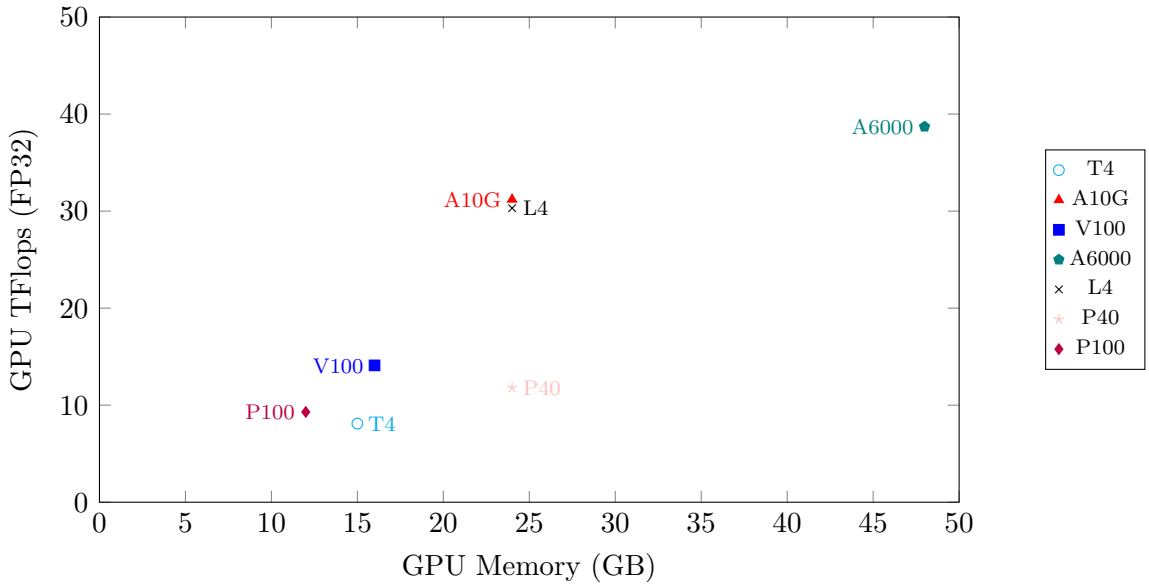


Figure 5.2: GPU performance versus memory capacity for various GPU models.

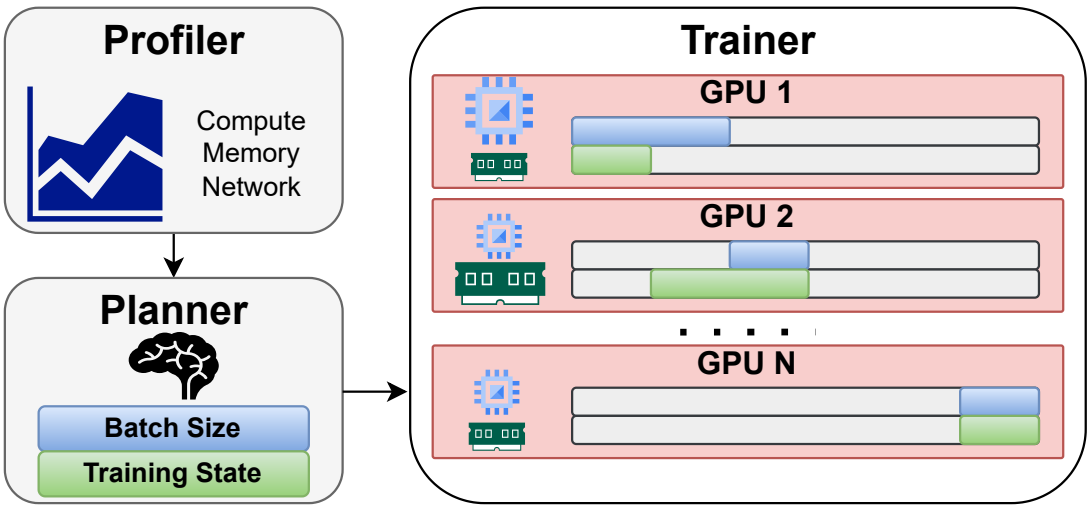


Figure 5.3: Cephalo system architecture.

5.1 Design

Cephalo independently balances compute and memory during training in heterogeneous GPU clusters, enabling the full utilization of the aggregate resources available in the cluster. To balance compute, Cephalo assigns a batch size to each GPU proportional to its compute speed. To balance memory utilization, Cephalo partitions the training state and decides on configurations for gradient accumulation, activation checkpointing, and activation/gradient offloading according to the relative memory capacities of each GPU. Given a model and target cluster, Cephalo profiles the model to build performance models predicting computation time, memory usage, and communication time across configurations. The planner then leverages these models to configure batch size, training state shard, and gradient accumulation for each GPU to maximize training throughput. The system architecture is illustrated in Figure 5.3.

5.1.1 Division of Compute and Training State

A key feature of Cephalo is its ability to decouple the distribution of compute and memory loads across GPUs, essential for optimizing performance in heterogeneous clusters where GPU memory capacity does not necessarily scale with compute power. Cephalo efficiently

allocates compute and training state across GPUs, leveraging the combined compute and memory resources of the cluster. The mechanisms Cephalo uses for this division are described next.

Compute Partitioning. Given a global batch size B, Cephalo partitions the workload across GPUs by assigning each GPU i a local batch size b_i such that $\sum_i b_i = B$. To minimize iteration times, Cephalo balances b_i to reduce the maximum runtime on any GPU. To maintain equivalency with standard training, each GPU's local gradient is adjusted by $N \cdot b_i/B$, resulting in a final gradient of:

$$\nabla = \frac{1}{N}\sum_{i=1}^{N}(\frac{N \cdot b_i}{B})\frac{1}{b_i}\sum_{j=1}^{b_i}\nabla_{ij} = \frac{1}{B}\sum_{i=1}^{N}\sum_{j=1}^{b_i}\nabla_{ij}. \tag{5.1}$$

Training State Partitioning. The training state – comprising model parameters, gradients, and optimizer states – consumes significant memory. Cephalo is built on top of FSDP, which partitions this state evenly across GPUs to reduce memory utilization. The model is divided into FSDP units which manage the computation and communication for their respective parameters. As in prior work [142], each unit in Cephalo contains a single transformer layer. Each unit shards its portion of the training state evenly. During the forward and backward passes for an FSDP unit, an *AllGather* collective gathers the full set of parameters on each GPU. Parameters are subsequently resharded to ensure only the current and prefetched next units are materialized in memory. After a unit's backward pass, a *ReduceScatter* collective averages gradients and distributes them to the responsible GPUs. FSDP overlaps AllGather and ReduceScatter collectives with the computation of adjacent units to hide communication latencies. In contrast to a fixed even partitioning, Cephalo assigns each GPU i a specific training state ratio r_i (where $\sum r_i = 1$), enabling fine-grained memory control independent of the compute distribution.

5.1.2 Managing Memory for Compute

Beyond storing training state, significant GPU memory is needed during computation for storing intermediate activations. Gradient accumulation [128, 92] is employed to enable training with larger effective batch sizes while reducing memory usage. Instead of computing gradients for the full batch size b at once, b is split into smaller microbatches of size m and gradients are accumulated over ℓ microbatches, where $b = \ell \cdot m$. This approach allows each GPU to process an effective batch size of b while reducing memory demands by managing smaller microbatches. In Cephalo, an optimized implementation of gradient

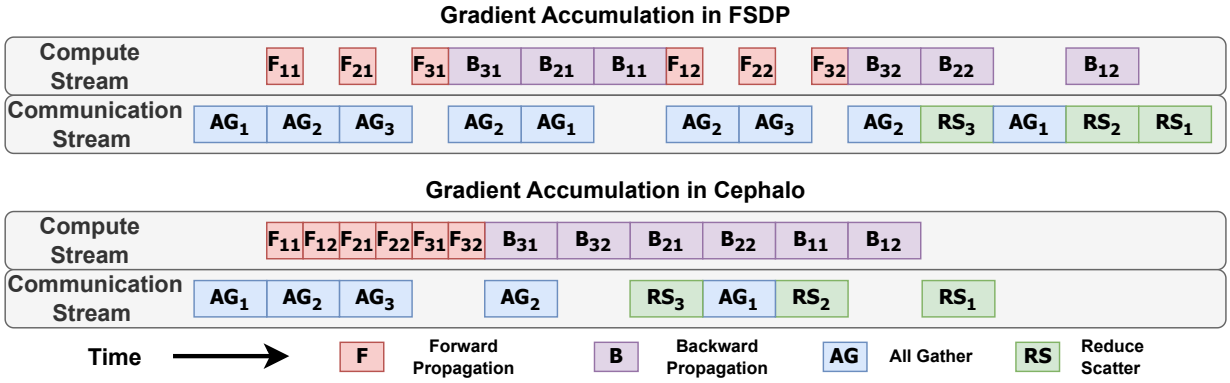


Figure 5.4: Gradient accumulation in FSDP (top) vs Cephalo (bottom) when training with 2 microbatches on a model consisting of 3 FSDP units. F_{ij} and B_{ij} are the forwards and backwards passes of the ith FSDP unit on the jth microbatch. AG_i and RS_i are the AllGather and ReduceScatter collectives for the ith FSDP unit.

accumulation for FSDP is developed, and configured to control the amount of memory used for computation.

Layered Gradient Accumulation. Traditional gradient accumulation in FSDP performs the full forward and backward pass for each microbatch sequentially. This requires ℓ times more AllGather collectives than training without gradient accumulation because sharded parameters must be gathered for each microbatch. To reduce this overhead, Cephalo implements *layered* gradient accumulation [92], which processes *all* microbatches for a layer before moving to the next. Sequentially processing microbatches allows parameters to be gathered only once per pass for all microbatches.

Figure 5.4 illustrates the difference between gradient accumulation in FSDP and Cephalo. Cephalo's implementation calls AllGather to prefetch the next FSDP unit while the current one is executing. This communication is overlapped with all executing microbatches of the current FSDP unit, effectively hiding the communication overhead even when networking is slow relative to compute. Gradient accumulation can add minor runtime overhead as smaller microbatches may not fully utilize GPU cores, introducing a tradeoff between memory savings and compute efficiency. Unlike prior systems, Cephalo automatically optimizes gradient accumulation with compute and training state partitioning, balancing this tradeoff effectively.

Activation Checkpointing and Offloading. While layered gradient accumulation reduces communication overhead, it introduces significant memory overhead compared

to traditional gradient accumulation. This is because activations must be stored for all microbatches of a layer until the backward pass, whereas traditional gradient accumulation maintains activations for only a single microbatch. For some models, this additional activation storage can exceed the memory savings gained from smaller batch sizes.

Cephalo addresses memory overhead in layered gradient accumulation with a combination of activation checkpointing and offloading. Activation checkpointing saves activations only at layer boundaries during the forward pass [128, 131], allowing intermediate activations to be recomputed in the backward pass, which significantly reduces memory usage. However, even storing boundary activations adds overhead. To mitigate this, Cephalo offloads boundary activations to CPU memory until needed during backwards. Similarly, Cephalo offloads boundary gradients in the backwards pass. Combined, activation checkpointing and offloading eliminate memory overhead when accumulating additional microbatches. Hence, for a fixed microbatch size, Cephalo can increase the number of microbatches without using additional memory. Activations and gradients are offloaded and prefetched between the GPU and CPU asynchronously with compute, hiding transfer overheads. In the forwards pass:

1. Microbatch activations are offloaded to the CPU while the next microbatch runs.
2. Input activations for the next microbatch are prefetched while the current microbatch runs.

In the backwards pass:

1. Microbatch gradients are offloaded to the CPU while the next microbatch runs.
2. Input activations for recomputing the next microbatch's activations are prefetched from the CPU.
3. The subsequent layer's gradients are prefetched from the GPU to compute the current layer's gradients.

This process is illustrated for gradient accumulation with 3 microbatches in Figure 5.5.

5.1.3 Performance Modeling

The profiler runs training iterations on small batch sizes in the target cluster to build predictive models for compute latency and memory usage based on batch size. Linear

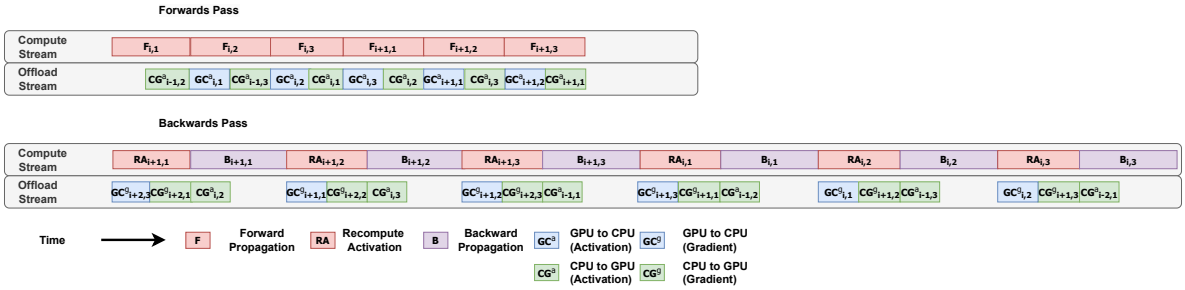


Figure 5.5: Activation & Gradient Offloading in Layered Gradient Accumulation: This figure illustrates the transfer of activations and gradients between CPU and GPU during forward and backward passes for two consecutive model layers (i and $i+1$) across three microbatches. In the backward pass, activations are recomputed (RA). Here, GC^a_{ij} indicates moving the activation from GPU to CPU for layer i and microbatch j, with CG^a_{ij} representing the reverse. Similarly, GC^g_{ij} denotes transferring the activation gradient from GPU to CPU, and CG^g_{ij} signifies the reverse.

models are used, as they are simple, require minimal profiling to fit, and accurately predict both metrics. Communication latency for collectives is profiled with an evenly sharded training state and a conservative model is applied to adjust for latency when the state is unevenly sharded. The planner then uses these models to find a configuration that maximizes throughput while respecting each GPU's memory capacity.

Compute Latency Model. In the left plot of Figure 5.6, the compute latency of a single transformer layer is profiled as the batch size increases. For small batch sizes, the latency increases sublinearly as the batch size is not large enough to fully utilize the compute on the GPU. Non-linearities can also arise from certain batch sizes aligning better with GPU memory architecture, or using better kernel algorithms selected by the compiler. As the GPU compute is saturated for larger batch sizes, there is a strong linear relationship. Latency is modeled by using the profiled data for smaller batches to capture non-linearities, then extrapolating linearly for larger batches. Profiling a single layer reduces time and resources, and since transformer layers are typically identical, this can be used to extrapolate the entire model's latency.

Let $T_f(m)$ and $T_b(m)$ be the latency models for forwards and backwards compute as a function of the microbatch size m. The latency of a single microbatch is linearly scaled by the number of microbatches ℓ to derive the total forwards $T_f(m, \ell)$ and backwards $T_b(m, \ell)$ compute latencies.

Memory Utilization Model. During training, GPU memory utilization includes memory

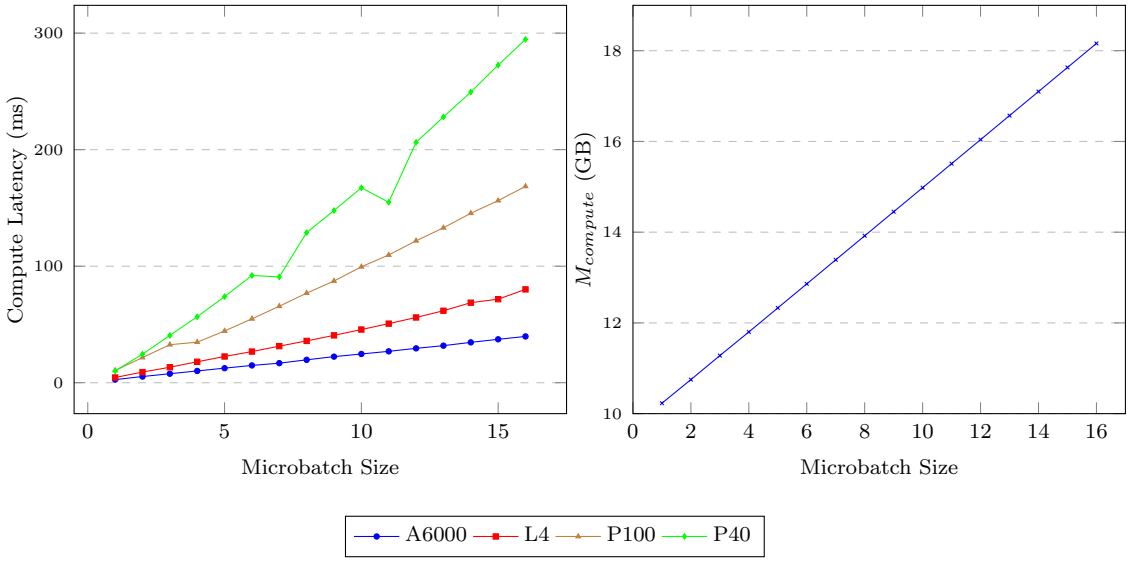


Figure 5.6: Training latency and memory allocated for compute as the microbatch size increases (Bert-Large).

for the training state, M_{state}, and computation, $M_{compute}$, resulting in a total memory usage of $M = M_{state} + M_{compute}$. M_{state} is derived from the parameters in a GPU's model shard $|P|$. Standard full-precision training with the Adam optimizer is assumed, where each parameter requires 4 bytes for the parameter, 4 bytes for its gradient, and 8 bytes for the first and second gradient moments. Thus, the total memory needed for the training state is $M_{state} = 16 \cdot |P|$. $M_{compute}$ encompasses memory for materializing layer parameters, executing GPU kernels, storing activations, and other framework state. In the right plot of Figure 5.6, $M_{compute}$ is plotted against batch size by subtracting M_{state} from the total memory usage, showing a strong linear relationship. $M_{compute}$ is profiled for small batch sizes to fit a linear model based on microbatch size. The linear increase is due to the need to run kernels and store activations for larger batch sizes. Notably, $M_{compute}$ is unaffected by the number of microbatches, as activations are checkpointed and offloaded after computation in Cephalo.

Communication Latency. FSDP uses NCCL [137] for inter-GPU communication, using *AllGather* to collect parameters and *ReduceScatter* to average gradients. With even training state sharding, inputs to NCCL collectives are equal in size; however, uneven sharding introduces variable input sizes. Cephalo employs generalized collective implementations which handle uneven inputs but incur overhead from extra GPU memory copies [213]. It was observed empirically that the overhead of using uneven sharding remained within 25%

of even sharding latency regardless of the input skew, defined to be the ratio of the size of the largest shard to the total input size. Thus, collective latency is profiled under even input sizes and a conservative assumption of 25% overhead for uneven sharding is applied. Since transformer layers are identical, profiling a single layer suffices.

5.1.4 Planner

Given a model, cluster of N GPUs, and a target batch size B to train with, the optimizer decides how to divide the computation, training state, and configure gradient accumulation to maximize training throughput. Next, this chapter describes how the optimizer formulates this as an optimization problem and solves it with dynamic programming.

Symbol	Description
N, B	Number of GPUs and global batch size
m_i, ℓ_i, g_i	Microbatch size and number of microbatches for ith GPU g_i
$M(m, \ell)$	Compute memory for ℓ microbatches of size m
$M_{cap}^{g_i}$	Memory capacity of g_i
$T_f^{g_i}(m, \ell)$	Forwards latency of g_i for ℓ microbatches of size m
$T_b^{g_i}(m, \ell)$	Backwards latency of g_i for ℓ microbatches of size m
AG, RG	*AllGather* and *ReduceScatter* latency
M_{state}^{es}	Memory required to store an even training state share

Table 5.1: Notation and Definitions

Optimization Formulation. Training throughput is maximized by minimizing the latency for one iteration of training. Under the typical assumption that transformer layers are identical, this problem is equivalent to minimizing the forwards and backwards pass for a single transformer layer. Each transformer layer is wrapped as an FSDP unit [142], which efficiently overlaps communication during the forwards and backwards pass. The forwards pass runs in

$$T_f = \max(\max_i(T_f^{g_i}(m_i, \ell_i)), AG), \tag{5.2}$$

where variables are defined in Table 5.1. The forwards pass waits on the slowest GPU to finish its computation, as well as the AllGather that is running concurrently to fetch the next FSDP unit. Similarly, with activation recomputation, the backwards pass runs in

$$T_b = \max(\max_i(T_f^{g_i}(m_i, \ell_i) + T_b^{g_i}(m_i, \ell_i)), RS + AG), \tag{5.3}$$

where a *ReduceScatter* is required to average the gradient. The training state must be unevenly sharded if, for any GPU, its combined compute memory and memory holding the evenly distributed training state exceeds its memory capacity. Then, the goal is to minimize the layer latency $T_f + T_b$ subject to the constraints: (I) Batch size: $B = \sum_i b_i = m_i \cdot \ell_i, \ell_i \in \mathbb{Z}_{>0}$ (II) Individual memory: $M(m_i) \leq M_{cap}^{g_i}, \forall i$ (III) Aggregate memory: $M_{state} + \sum_i M(m_i) \leq \sum_i M_{cap}^{g_i}$.
The second constraint specifies that the memory used for compute cannot exceed the memory capacity of the GPU. The last constraint specifies that the aggregate GPU memory in the cluster is at least as much as the sum of the memory required to store the complete training state and perform computation on each GPU. Under these conditions, Cephalo is able to train the model without running out of memory.

Dynamic Programming Solution. The optimization problem is solved using dynamic programming. Let $D(i, j, k)$ be the minimum achievable runtime for the first i GPUs to process a total batch size of j and total microbatch size of k. That is, the sum of the batch sizes on the first i GPUs is j, and the sum of their microbatch sizes is k. Suppose that the optimal solution assigns ℓ microbatches of size m (batch size of $\ell \cdot m$) to the ith GPU. Then the optimal solution can be constructed by combining this assignment with the solution to $D(i-1, j-\ell \cdot m, k-m)$. Thus, by this optimal subproblem property, $D(i, j, k)$ can be computed as

$$D(i, j, k) = \min_{m,\ell} \max(D(i-1, j-\ell \cdot m, k-m), T_{i,\ell,m}) \ , \tag{5.4}$$

where $\ell \cdot m \leq j, m \leq k, M(m) \leq M_{cap}^{g_i}$ and $T_{i,\ell,m}$ is the runtime of forwards and backwards for ℓ microbatches of size m on the ith GPU using Eqs. 5.2 and 5.3.

From the memory model, the aggregate memory utilization can be computed using the sum of the microbatch sizes, k. Hence, the last dimension in the recurrence represents the aggregate memory utilization. This dimension is needed in the recurrence to ensure constraint (III) is satisfied. The minimum latency is $\min_k D(N, B, k)$ over all k meeting the memory constraint. Backtracking is then performed to find the batch and microbatch sizes that achieve this latency (Pseudocode is provided in Algorithm 2).

Partitioning Training State. After determining the compute partitioning, the optimizer allocates training state to minimize the maximum memory utilization across GPUs, balancing each GPU's memory usage relative to its capacity. This avoids out-of-memory issues and reduces memory allocation overheads when utilization approaches capacity. This allocation is determined with a water-filling-style greedy algorithm [14]:

1. Sort: Sort the GPUs in ascending order based on their total memory utilization, which includes memory required for computation and memory allocated for the training

Algorithm 2 Throughput Maximization using DP

Input: # of GPUs N, Batch Size B
Output: Max achievable throughput, training configuration
Initialize $DP[0 \dots N][0 \dots B][0 \dots B]$ with ∞
$DP[0][0][0] \leftarrow 0$
for $i \leftarrow 1$ **to** N, $j \leftarrow 1$ **to** B, $k \leftarrow 1$ **to** j,
    $m \leftarrow 1$ **to** k, $\ell \leftarrow 1$ **to** $\lfloor j/m \rfloor$ **do**
    if $M(m, \ell) > M_{cap}^{g_i}$ **then**
        continue with the next m
    $AG' = AG$, $RS' = RS$
    if $M(m, \ell) + M_{state}^{es} > M_{cap}^{g_i}$ **then**
        $AG' = AG_{uneven}$, $RS' = RS_{uneven}$
    $T_{i,\ell,m} \leftarrow \max(T_f^{g_i}(m, \ell), AG') + \max(T_b^{g_i}(m, \ell), AG' + RS')$
    $R \leftarrow \max(DP[i-1][j - \ell \cdot m][k - m], T_{i,\ell,m})$
    $DP[i][j][k] \leftarrow \min(DP[i][j][k], R)$
$optimalThroughput \leftarrow 0, solution \leftarrow None$
for $k \leftarrow 1$ to B **do**
    if $B/DP[N][B][k] > optimalThroughput$ **then**
        $optimalThroughput \leftarrow B/DP[N][B][k]$
        $solution \leftarrow Backtrack(DP[N][B][k])$
return $optimalThroughput, solution$

state by this algorithm.

2. Assign: Allocate parameters to the GPU with the lowest utilization until its memory matches the GPU with the second-lowest utilization.

3. Iterate: Repeat this for $k = 2, \ldots, N$. In the kth iteration, evenly distribute parameters to the GPUs with the kth lowest memory utilization until their utilization matches the GPU with the $(k+1)$st lowest utilization.

This process ensures that the parameters are distributed to the GPUs with the lowest memory utilization first in order to minimize the maximum memory utilization across GPUs.

Complexity Analysis. The optimizer runtime is dominated by the dynamic programming algorithm which runs in $O(N \cdot B^3 \cdot \log B)$, where N is the GPU count and B the global batch size. This arises from $O(N \cdot B^2)$ states, each requiring $O(B \cdot \log B)$ to compute. The greedy algorithm for training state partitioning runs for N iterations, where each iteration compares and assigns parameters in $O(N)$ time. Thus, the greedy algorithm runs in $O(N^2)$.

5.2 Implementation

This chapter details Cephalo's implementation and optimizations. Cephalo is implemented on top of PyTorch FSDP and consists of a profiler, optimizer and model trainer (Figure 4.3).

5.2.1 Profiler

The profiler conducts lightweight profiling to model compute latency, memory usage, and communication latency. It profiles a few training iterations for each batch size from 1 to B, fitting linear models for compute latency and memory usage. In practice, $B = 8$ suffices for accuracy. Additionally, the profiler measures *AllGather* and *ReduceScatter* latencies by averaging profiled latencies over 50 runs to obtain a representative sample of network performance.

5.2.2 Optimizer

The optimizer uses models built by the profiler to configure Cephalo for maximum training throughput (Section 5.1.4). It determines each GPU's microbatch size, number of microbatches, and assigned portion of the global batch size and training state. To avoid memory allocation bottlenecks as usage nears capacity, the optimizer caps GPU memory usage at 80%. It completes within 20 minutes for all workloads — a meager overhead to generate optimized, time-saving, training configurations compared to the GPU-years required to train these models [177].

In practice, the optimizer runtime is further reduced by observing that scaling the number of microbatches (ℓ_i) by some constant C on each GPU increases the global batch size (B) without affecting computation or memory distribution. With Cephalo's offloading optimizations (Section 5.1.2), memory utilization remains constant for a microbatch regardless of the number of microbatches. Moreover, constant scaling maintains a proportional compute load across GPUs. Consequently, optimization for a batch size of B can be performed by running the optimizer on a smaller batch size B/C and later adjusting the number of microbatches to $\ell_i' = C \cdot \ell_i$. This approach reduces optimization time for larger batch sizes.

5.2.3 Trainer

Compute and Training State Division. The *trainer* trains the model using the batch size and training state assignments set by the optimizer. Each process's data loader is configured to load its assigned batch size. Implementing uneven parameter sharding required modifying FSDP's *shard* and *unshard* operations to follow the training state divisions configured by the optimizer. The gradient reduction logic in the backward pass was also modified to average gradients according to this training state division.

When the training state is unevenly sharded, Cephalo uses generalized *AllGather* and *ReduceScatter* implementations to handle uneven input sizes. It was observed that uneven sharding incurs up to a 25% runtime overhead, but does not have a strong correlation with the input skew in shard sizes. Therefore, a greedy strategy is applied to minimize uneven sharding across FSDP units. For instance, if two identical FSDP units are split across two GPUs in a 3:1 ratio, one unit would be sharded evenly (1:1) and the other as 1:0, thereby incurring uneven sharding overhead for only one unit.

5.2.4 Layered Gradient Accumulation

The trainer implements a training loop for layered gradient accumulation (LGA), splitting each batch into microbatches and processing them one at a time through each FSDP unit before moving to the next. This order differs from FSDP's assumed sequential unit-by-unit execution, which it uses to overlap communication with computation. Consequently, several changes were needed in FSDP to avoid unnecessary communication and support communication-computation overlap with this new order of execution.

FSDP reshards parameters after each forward pass, assuming the next unit runs next and the current is no longer needed. However, in LGA, the same unit runs all microbatches before moving to the next. FSDP was modified to reshard parameters only after all microbatches are processed, avoiding unnecessary *AllGather*'s.

Communication-Computation Overlap. To maintain efficient overlap, the prefetching logic was updated to align with the layered gradient accumulation order and parameter unsharding logic was scheduled on its own GPU stream to avoid communication from blocking backwards computation. Furthermore, post-backward logic was adjusted to accumulate gradients across microbatches and reset the execution state only after all microbatches are processed. The FSDP prefetcher was configured (with the *limit_all_gathers* flag [26]) to retain only the current and subsequent layers' parameters in memory, avoiding memory overhead from excessive prefetching.

Compute Synchronization. Severe memory fragmentation was observed from PyTorch scheduling multiple microbatches simultaneously, leading to out-of-memory errors even below 50% memory usage. This fragmentation is avoided by synchronizing the GPU's compute stream to process one microbatch at a time.

Offloading. PyTorch's default activation offloading incurred large runtime overheads due to synchronous CPU-GPU transfer, which blocked GPU computation. Consequently, custom offloading and prefetching logic was implemented on a dedicated GPU stream, enabling GPU-CPU memory transfer to run in parallel with computation.

In Section 6.5.5, it is shown that these optimizations for LGA are essential for achieving high performance.

5.3 Experiments

This chapter evaluates the performance of Cephalo compared to state-of-the-art training methods on 9 popular transformer models across 2 heterogeneous GPU clusters. End-to-end

results are presented in Section 5.3.2, and larger-scale experiments in Section 5.3.3. Sections 6.5.5 and 5.3.5 analyze how Cephalo's design components impact performance. Section 5.3.6 evaluates the accuracy of Cephalo's performance model and Section 5.3.7 evaluates the optimization time of Cephalo's planner. Section 5.3.8 presents training configurations generated by Cephalo.

5.3.1 Experimental Setup

The evaluation includes popular transformer models used for text classification (TC), text generation (TG), and image classification (IC) following the training setup from [142]. Activations are checkpointed after each transformer layer and models are trained in full precision with the Adam optimizer, using a sequence length of 512 for language modeling (unless otherwise specified). Table 6.3 provides further details on the models.

Clusters. Cephalo was evaluated on environments representative of typical heterogeneous GPU clusters used by ML practitioners. Cluster A was assembled with four types of GPUs acquired over several years. Cluster B is a mix of higher- and lower-end GPU VMs on AWS, selected to reflect the typical quantities available for reservation.

- Cluster A: 2 machines (8 GPUs), connected via a 50 Gbps link. One contains 2×L4, 1×A6000, and 1×P40; the other contains 2×P40 and 2×P100.
- Cluster B: 8 VMs (64 GPUs), equipped with 100 Gbps bandwidth. 2×g5.48xlarge (8×A10G), 2×p3.16xlarge (8×V100-16 GB), and 4×g4dn.metal (8×T4) VMs.

A summary of GPU specifications appear in Table 6.4.

Baselines. The comparison is made against representative state-of-the-art techniques for training on heterogeneous GPU clusters:

- Megatron-Het [131]: Pipeline parallelism across nodes and data/tensor parallelism within nodes. It was adapted for heterogeneous training by partitioning the model proportionally to each node's compute capacity.
- HexiScale [194]: Combines ZeRO-2 data [144] (optimizer state and gradient sharding), tensor, and pipeline parallelism. An optimizer balances memory and compute.
- Whale [72]: Balances computational loads with data parallelism by assigning batch sizes to GPUs based on their runtime profiles.

- HAP [210]: Uses tensor parallelism across nodes and data parallelism within nodes. The batch size and parameters are sharded unevenly to balance the workload.

Task	Model	Layers	Embd. Size	Attn. Heads	Parameters
IC	ViT-G [202]	48	1664	16	1.8B
IC	ViT-e [19]	56	1792	16	3.9B
TC	BERT-Large [36]	24	1024	16	0.4B
TC	BERT-XLarge [36]	36	1536	24	1.2B
TG	GPT 2.7B [15]	32	2560	80	2.7B
TG	GPT 6.7B [15]	32	4096	128	6.7B
TG	Tiny Llama [208]	22	2048	32	1.1B
TG	Llama 3B [52]	26	3200	32	3.5B
TG	Llama 7B [177]	32	4096	32	6.7B

Table 5.2: Model Statistics

Cluster	GPUs	Generation	Memory	TFlops	Total Memory	Total TFlops
A	3×P40	Pascal	24 GB	11.8	192 GB	153.3
	2×P100	Pascal	12 GB	9.3		
	1×A6000	Ampere	48 GB	38.7		
	2×L4	Ada	24 GB	30.3		
B	16×V100	Volta	16 GB	14.1	1120 GB	984.0
	32×T4	Turing	15 GB	8.1		
	16×A10G	Ampere	24 GB	31.2		

Table 5.3: GPU Specifications (FP32 TFlops)

5.3.2 Training Throughput

Cephalo's end-to-end training throughput was evaluated against baselines, measuring throughput as samples processed per second (images for image classification models, sequences for language models). Experiments on Cluster A included models of up to 3.9

System	ViT-G		ViT-e		Bert-Large		Bert-XLarge	
	128	**256**	**128**	**256**	**128**	**256**	**128**	**256**
Megatron-Het	3.41	0.79	*OOM*	*OOM*	19.77	20.57	6.40	6.80
HexiScale	2.88	2.97	1.38	1.4	25.64	28.90	8.63	9.06
Whale	*OOM*	*OOM*	*OOM*	*OOM*	27.13	28.84	*OOM*	*OOM*
HAP	*OOM*	*OOM*	*OOM*	*OOM*	17.48	18.54	*OOM*	*OOM*
Cephalo	**6.38**	**6.41**	**3.02**	**3.23**	**33.56**	**33.69**	**11.47**	**11.72**

Table 5.4: Throughput comparison of different models/batch sizes on 8-GPU Cluster A (Part 1/2). *OOM* denotes Out-of-Memory.

System	GPT 1.3B		GPT 2.7B		Tiny Llama		Llama 3B	
	128	**256**	**128**	**256**	**128**	**256**	**128**	**256**
Megatron-Het	4.18	4.35	1.82	1.82	7.93	8.63	*OOM*	*OOM*
HexiScale	5.81	5.83	2.79	2.83	8.67	8.75	1.91	1.83
Whale	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*
HAP	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*	*OOM*
Cephalo	**6.83**	**7.09**	**4.57**	**4.67**	**12.58**	**12.91**	**4.51**	**4.85**

Table 5.5: Throughput comparison of different models/batch sizes on 8-GPU Cluster A (Part 2/2). *OOM* denotes Out-of-Memory.

billion parameters with global batch sizes of 128 and 256. Cluster A is highly heterogeneous, with four GPU types varying substantially in compute and memory. Baselines do not auto-configure pipeline parallelism, so various microbatch sizes (powers of 2) were tested, with the best results reported in Tables 5.4 and 5.5. Cephalo consistently achieved significantly higher throughput without out-of-memory (*OOM*) errors across all models and batch sizes.

Comparison to Megatron-Het. Megatron uses four pipelines of two GPUs each across the two nodes. However, each pipeline must be partitioned identically, despite the mixed GPU types on each node. This results in different GPUs being assigned the same stage across pipelines, causing compute bottlenecks due to the slower P40 GPUs, which underutilizes faster L4 and A6000 GPUs, reducing throughput. For larger models (GPT 2.7B and Llama 3B), Megatron applies tensor parallelism within each node, further decreasing throughput due to high communication overhead. Megatron is optimized for clusters with fast interconnects like NVSwitch, which Cluster A and most AWS VMs do

not have (except mostly unavailable A100 and H100 VMs).

Comparison to HexiScale. Like Cephalo, HexiScale trains larger batch sizes with a reduced memory footprint by using smaller microbatches and gradient accumulation. However, smaller microbatches may not fully utilize GPU compute, and frequent gradient accumulation reduces pipeline parallelism efficiency. Cephalo automatically optimizes the microbatch size and gradient accumulation configuration, whereas HexiScale requires manual tuning. Additionally, Cephalo's layered gradient accumulation implementation does not incur extra communication overhead. HexiScale, like Megatron-Het, relies on communication-heavy tensor parallelism for larger models. These factors enable Cephalo to achieve significantly higher throughput across all configurations.

Comparison to Whale. Like Cephalo, Whale optimizes compute utilization in the cluster by assigning varying local batch sizes to GPUs based on their compute capabilities. However, it is able to train only the smallest model, Bert-Large, without running out of memory. In this cluster, although P40 GPUs have similar compute speeds to P100s, they have twice the memory (24 GB). Despite this, to maintain compute balance, Whale assigns similar batch sizes to both, causing P100s to run out of memory when P40s have utilized only 50% of their memory. Cephalo avoids this issue by partitioning compute independently from memory. It assigns a similar batch size for both P40 and P100 GPUs, but stores a larger share of the training state in the P40 GPUs to balance memory utilization. Whale also consumes considerably more memory than Cephalo since data parallelism replicates the entire training state across each GPU. Cephalo saves memory by sharding the training state at the cost of extra communication. However, Cephalo effectively masks this extra communication by overlapping it with computation.

Comparison to HAP. HAP, like Cephalo, can partition compute by dividing the batch size unevenly across GPUs. However, HAP relies on tensor parallelism to partition the training state, which is proportional to the amount of compute assigned. HAP does not consider the memory constraints on the GPUs, so it runs out of memory on all models but Bert-Large. Despite the compute partitioning, HAP is unable to train efficiently due to the high communication overheads of tensor parallelism, which requires high-bandwidth interconnects.

5.3.3 Larger Cluster Experiments

Cephalo's scalability was evaluated on the larger Cluster B featuring 64 GPUs (16 V100s, 16 A10Gs, and 32 T4s) using ViT-e, GPT-6.7B, and Llama-7B models with batch sizes of 512 and 1024. Cephalo consistently delivered 2–10× higher throughput than other systems.

System	ViT-e		GPT 6.7B		Llama 7B	
	512	**1024**	**512**	**1024**	**512**	**1024**
Megatron-Het	12.06	12.12	3.59	1.71	5.53	1.65
HexiScale	12.84	13.37	4.78	4.99	5.42	5.47
Cephalo	**20.37**	**26.08**	**11.62**	**17.04**	**13.12**	**17.74**

Table 5.6: Throughput comparison on 64-GPU Cluster B.

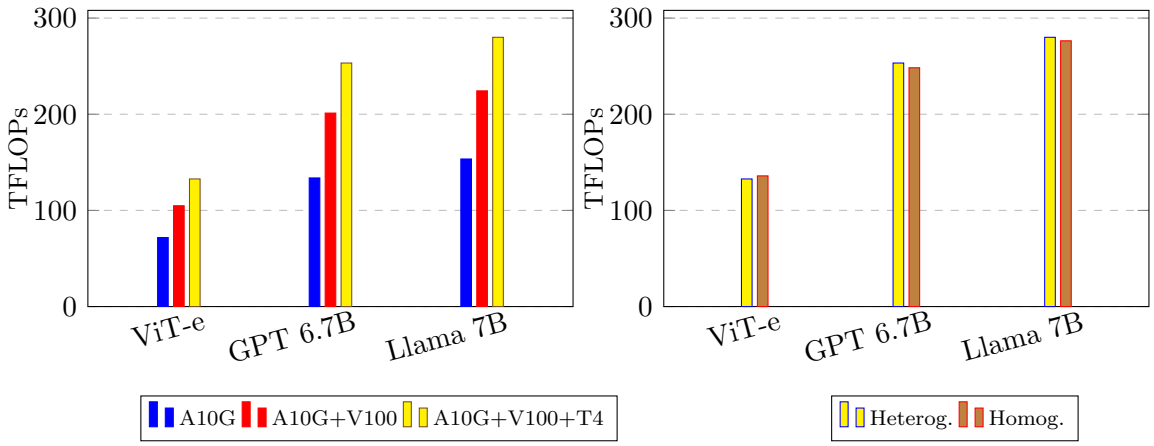


Figure 5.7: Left: Throughput (TFLOPs) with different heterogeneous cluster configurations. Right: Throughput on Cluster B vs. a 32×A10G homogeneous cluster.

At a batch size of 512, Megatron uses ZeRO-2 data parallelism within each node. Since it does not shard the model parameters like Cephalo, Megatron needs to configure pipeline parallelism with a smaller microbatch size and a suboptimal model partitioning to avoid running out of memory. It is unable to fully utilize compute on the V100 GPUs since it has similar memory to the T4 despite being significantly faster. At a batch size of 1024, Megatron uses tensor parallelism to manage memory. However, this reduces throughput for GPT 6.7B and Llama 7B, as V100 GPUs' NVLink lacks all-to-all connectivity and is not fast enough to offset the communication overhead of tensor parallelism.

HexiScale is able to more flexibly parallelize training, supporting a different degree of tensor parallelism per pipeline stage and a different number of GPUs for each pipeline. This enables faster training at a batch size of 1024 when memory pressure is larger. However, it still relies on tensor parallelism (albeit less than Megatron) and partitions layers into pipeline stages according to memory, rather than compute, to avoid running out of memory. This partitioning assigns the T4s a similar workload as the V100s, despite being slower, resulting in a performance bottleneck.

In contrast, Cephalo leverages FSDP to shard training state, reducing memory requirements and enabling training at a batch size of 1024 without tensor parallelism. Additionally, independent partitioning of training state from compute allows Cephalo to fully utilize each GPU by assigning batch sizes proportional to its compute capacity.

Scaling Heterogeneous GPUs. In the left plot of Figure 5.7, the training throughput (in TFLOPs) of Cephalo is compared as the system scales from using only the fastest A10G GPUs in Cluster B, to using the A10G and V100 GPUs, to finally using all GPUs. The training throughput almost doubles when comparing using only A10G to utilizing all the heterogeneous GPUs in the cluster. Cephalo is able to achieve a significant improvement in training throughput by utilizing all of the (heterogeneous) GPUs available on the cluster.

Comparison to Homogeneous Training. In the right plot of Figure 5.7, Cephalo's training TFLOPs on Cluster B are compared to a homogeneous cluster of 32×A10Gs with similar peak TFLOPs (984 vs. 998). Despite Cluster B's mix of lower-memory lower-compute GPUs, Cephalo is able to achieve comparable TFLOPs to the homogeneous cluster, demonstrating effective utilization of heterogeneous GPUs.

Mixed Precision Training. The evaluation focuses on full precision training since it preserves model convergence guarantees and older GPUs in cluster A were not optimized for lower precision. However, Cephalo's design is also effective for lower-precision training, where non-proportional scaling of GPU memory and compute still exists. For example, the V100 and T4 have similar memory capacity but the V100 has 2× higher FP16 TFLOPs. Cephalo's effectiveness at lower precisions is validated by showing it outperforms other

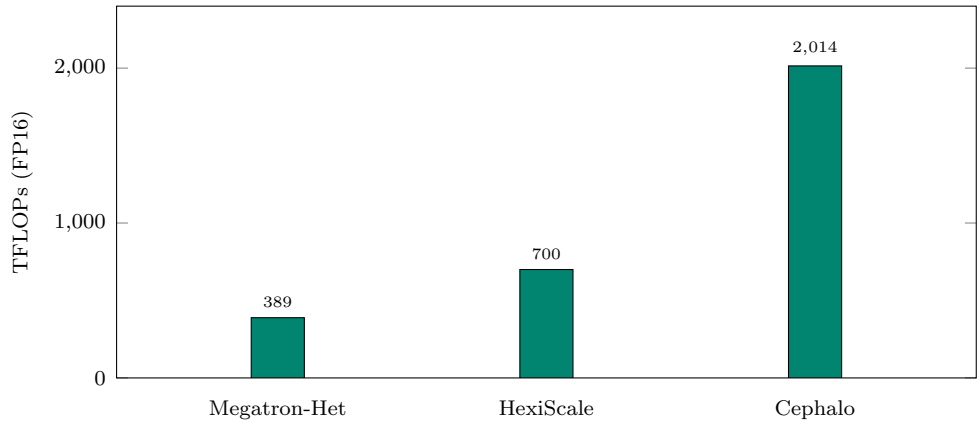


Figure 5.8: Mixed Precision (FP16) Training on Cluster B (Llama 7B, batch size 1024, sequence length 2048).

systems when training Llama 7B with FP16 mixed precision [125], a batch size of 1024, and a sequence length of 2048 on Cluster B (Figure 5.8). Megatron-Het and HexiScale use more communication intensive configurations due to memory constraints and obtain lower throughput due to communication overhead. Generally, hiding latencies through communication-computation overlap at lower precisions is more difficult as both computation and communication latencies decrease, but computation latency at a greater rate. Despite this, Cephalo manages to effectively hide communication latencies with its memory and communication efficient implementation of FSDP with layered gradient accumulation.

5.3.4 Ablation Study

An ablation study was conducted to assess the individual and joint contributions of compute and memory balancing to Cephalo's performance. Cephalo's throughput was compared with two variants: compute balancing only (Cephalo-CB) and memory balancing only (Cephalo-MB), alongside baseline FSDP. Experiments were run on Cluster A with ViT-e, GPT-2.7B, and Llama-3B, scaling batch sizes to 256, as shown in Figure 5.9. Cephalo-CB improves throughput over FSDP by balancing compute but encounters out-of-memory (OOM) issues beyond a batch size of 100 for all models, with throughput declining as it nears max memory capacity. Cephalo-MB prevents OOM by balancing memory with uneven training state partitioning and using gradient accumulation with a microbatch size of 1. However, its throughput is lower than FSDP's, as gradient accumulation with such a small microbatch size fails to fully utilize GPU compute, underscoring the need

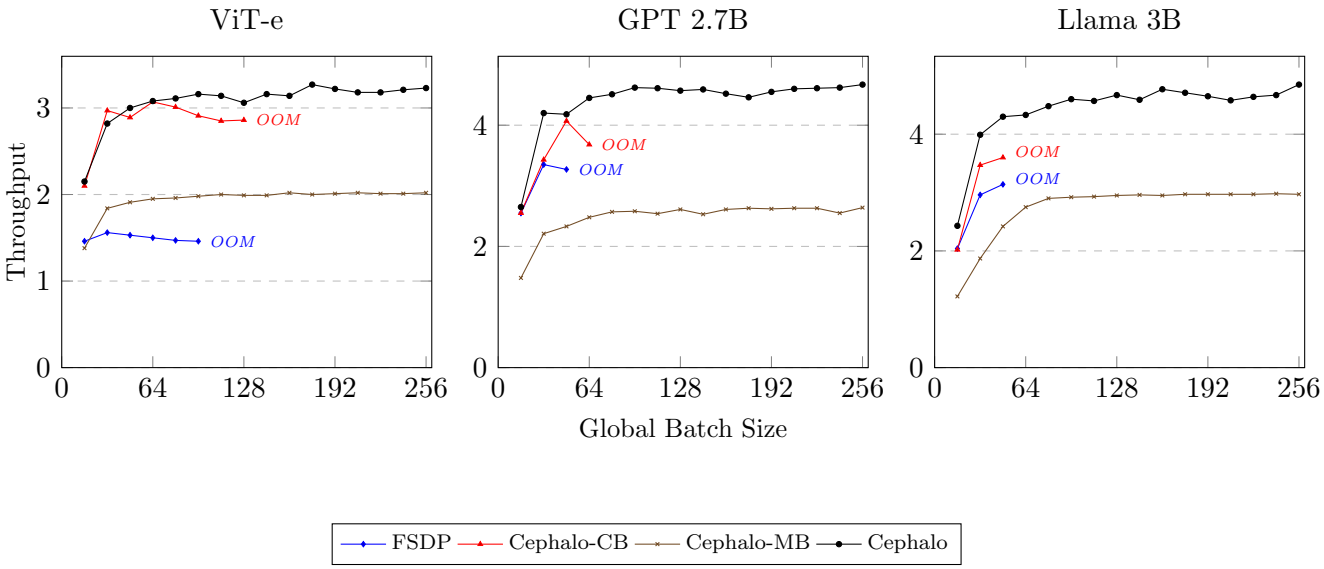


Figure 5.9: Throughput at different batch sizes with, and without, compute and memory balancing.

for prudently configuring gradient accumulation. Cephalo overcomes Cephalo-CB and Cephalo-MB limitations by jointly balancing compute, memory, and gradient accumulation, essential for high throughput on heterogeneous clusters. It achieves the highest training throughput across all batch sizes and sustains high throughput up to a batch size of 256 without running OOM.

5.3.5 Gradient Accumulation Optimizations

In Figure 5.10, the throughput and memory improvements obtained from Cephalo's gradient accumulation optimizations are investigated. Starting from the existing gradient accumulation in FSDP (FSDP-GA), layered gradient accumulation (LGA) is introduced, then communication overlap with computation (CO), compute synchronization (S), and offloading (O) are added. The GPT 6.7B model is trained with a batch size of 256 (16 microbatches of size 1 per GPU). A homogeneous cluster of 16×V100 GPUs is used to isolate from the effects of heterogeneous GPUs.

While FSDP-GA encounters communication bottlenecks, LGA achieves a 6× speedup by minimizing communication overhead and increases throughput by 22% through full communication overlap with gradient accumulation. Additionally, compute synchronization

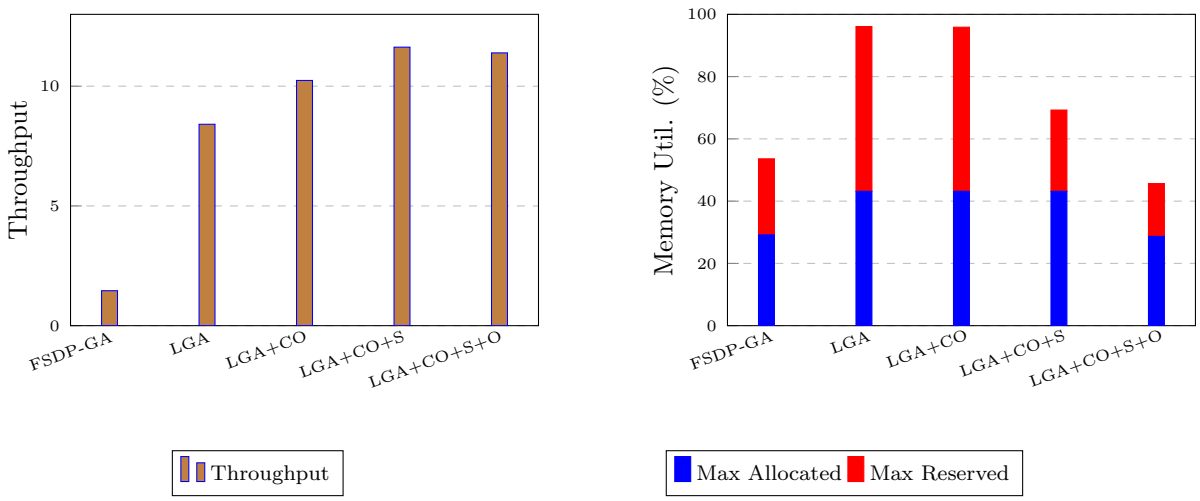


Figure 5.10: Speedup and memory reduction from gradient accumulation optimizations (LGA+CO+S+O).

and offloading eliminate memory overhead and fragmentation, boosting throughput by an extra 11%. The final implementation with all optimizations (LGA+CO+S+O) delivers a 7.8× speedup over FSDP-GA while reducing memory usage.

5.3.6 Performance Model Accuracy

Cephalo's optimizer uses a performance model to predict runtime across training configurations, which is essential for efficiently navigating the large search space and optimizing configurations. Figure 5.11 shows the absolute relative error (ARE) between predicted and actual latencies on Cluster A. Across all models and batch sizes, errors remained within 10%, with a mean absolute relative error of 2.9%. Notably, error rates did not increase for larger models or batch sizes, demonstrating the performance model's robustness.

5.3.7 Planner Optimization Time

To generate a training configuration, Cephalo profiles the model and network in addition to running the planner to partition compute and training state. This runtime depends on the number of GPUs in the cluster, size of the model, and batch size. Even in the largest experiment with 64 GPUs, GPT 6.7B and a batch size of 512, it took less than 20 minutes

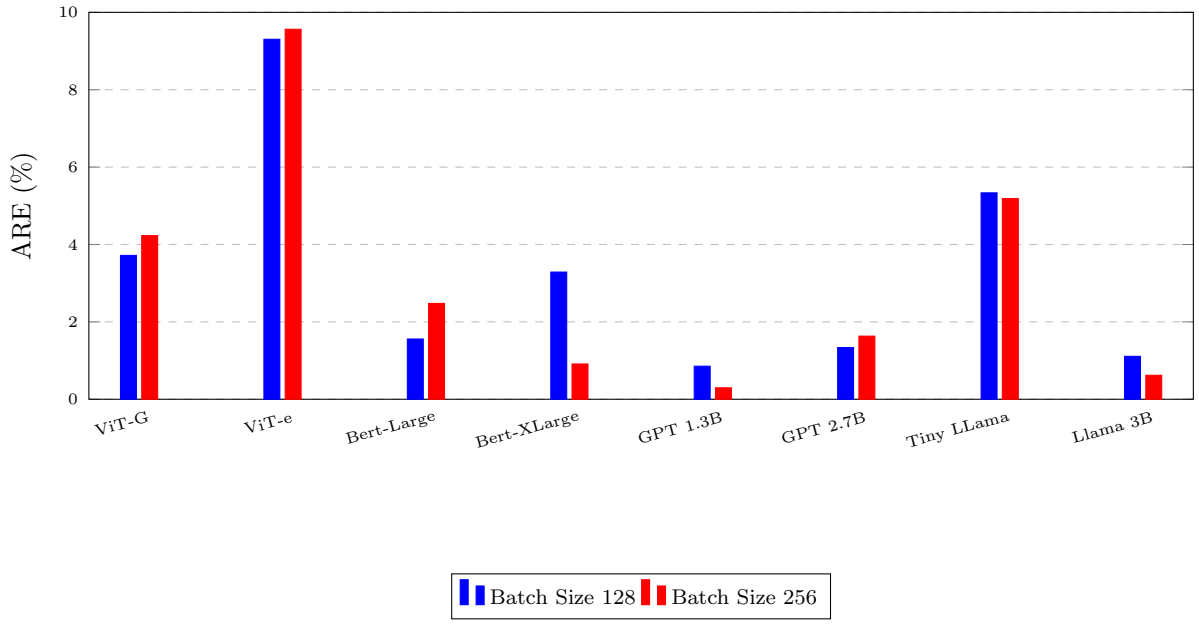


Figure 5.11: Performance model absolute relative error.

to generate the training configuration. The search time is negligible compared to the long times required to train these large models. Moreover, the profiling tasks need to be run only once for a given model and cluster. The planner can reuse the profiling data to generate configurations for different batch sizes and GPUs. Table 5.7 shows the runtime breakdown for each subtask in the optimization process.

5.3.8 Optimized Training Configurations

In Figure 5.12, Cephalo's optimized configurations for ViT-G and Llama 3B on Cluster A with batch size 256 are shown. The A6000 GPU, being faster and having more memory than the L4s, P100s, and P40s, is assigned the largest portion of the training state and compute. The L4s, with about half the compute and memory of the A6000, receive roughly half the batch size and training state. P100s and P40s are assigned smaller batch sizes, with the P40 handling a larger training state due to its greater memory capacity.

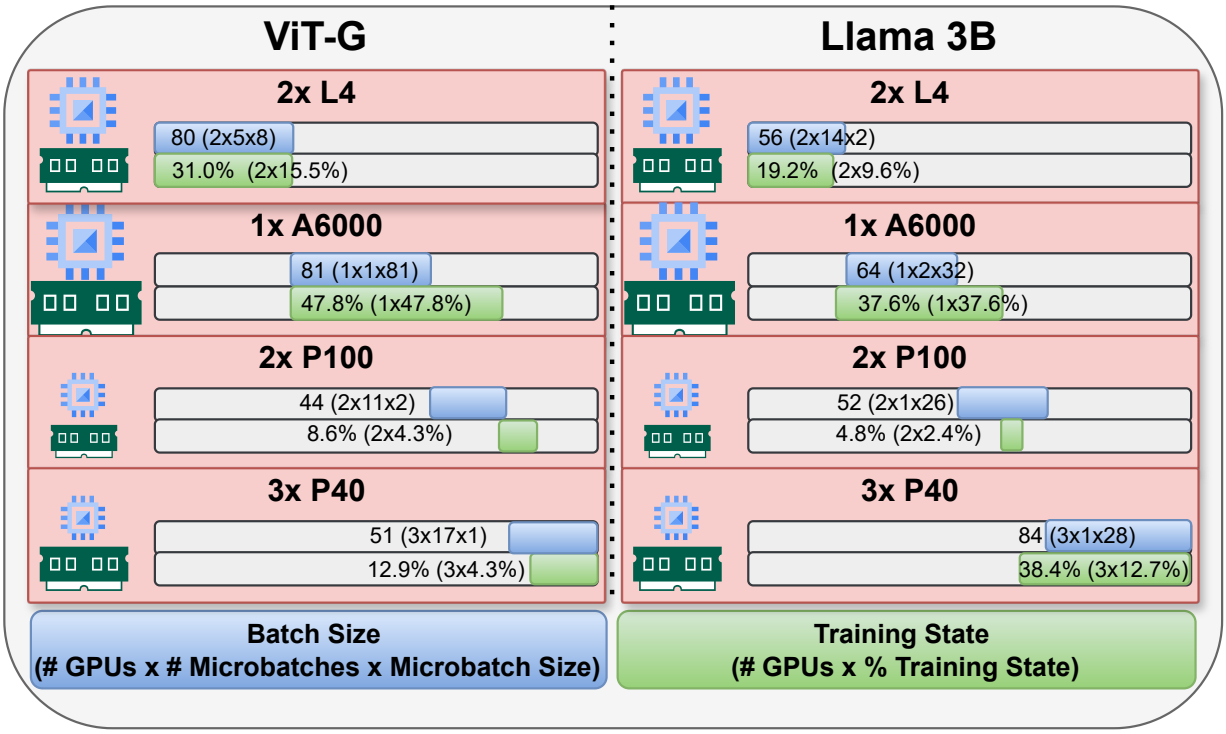


Figure 5.12: Training configurations (ViT-G & Llama 3B).

Subtask	Runtime (s)
Profile Compute	23
Profile Memory	486
Profile Communication	150
Partition Compute DP	327
Partition State	1
Total	**987**

Table 5.7: Breakdown of profiling and optimization runtime.

5.4 Discussion

Cephalo is designed to address compute and memory imbalances in heterogeneous clusters. This chapter discusses Cephalo's adaptability to various networking environments, though this aspect is orthogonal to the focus of this chapter.

5.4.1 Adapting to Slower Networks

Communication over the network can potentially bottleneck training when networking is slow relative to compute. However, with the layered gradient accumulation optimizations in Cephalo, both communication and GPU memory can be kept constant while scaling the number of microbatches. Therefore, if networking latency is high enough to limit training performance, users can increase the global batch size to scale computation time such that it can fully overlap and hide communication overhead. While extremely large batch sizes could impact model convergence, the few organizations training at this scale typically own very large, dedicated, homogeneous GPU clusters [55, 78], obviating the need for heterogeneous training solutions. Cephalo targets mid-sized clusters where heterogeneous compute is commonly needed [76, 123, 139, 210] and batch sizes remain well within the limits at which modern transformers can achieve efficient convergence [55].

5.4.2 Adapting to Heterogeneous Networks

In heterogeneous networks, varying interconnect speeds can cause *AllGather* and *ReduceScatter* collectives in FSDP to be bottlenecked by slower links. While increasing batch size can mitigate networking overheads, changing the parallelism strategy can also be considered.

Hierarchically-sharded variants of FSDP replicate model parameters within GPU subgroups of a cluster with faster interconnects [213, 183, 192]. This approach reduces communication across slower links when parameters are gathered but increases memory usage.

Integrating pipeline parallelism with FSDP offers an orthogonal optimization. By partitioning the model into stages, FSDP can be applied within each stage among nodes with faster interconnects, while nodes with slower interconnects are assigned to different stages. This approach leverages communication-efficient pipeline parallelism over slower links and utilizes FSDP across faster links. However, naive integration of layered gradient accumulation (LGA) with pipeline parallelism can introduce significant pipelining overhead. In LGA, all microbatches are processed for a layer before moving to the next, delaying computation pipelining with subsequent FSDP groups until the final layer in the stage begins. To mitigate this, each pipeline stage can be divided into smaller microstages assigned in a round-robin fashion across FSDP groups [131, 92]. Processing smaller microstages enables earlier pipelining, reducing pipeline overhead. In stages with very fast interconnects, tensor parallelism can be combined with FSDP to scale computation without increasing batch size. Optimizations in compute-communication overlap for tensor parallelism can help maintain high training efficiency [187, 184].

5.5 Conclusion

Cephalo is the first system that jointly resolves imbalances in compute and memory across GPUs when training on a heterogeneous cluster. It decouples compute and memory requirements for each GPU through uneven compute division, parameter sharding, and gradient accumulation. Cephalo models compute, memory, and communication holistically and uses an optimizer to optimally allocate training state, batch size, and gradient accumulation across GPUs. Evaluations on multiple clusters show that Cephalo achieves significantly higher training throughput while supporting larger models and batch sizes than existing systems.

Chapter 6

Efficient Training Over Heterogeneous Networks

While Cephalo was shown to achieve efficient heterogeneous training in Section 5.3, communication latency can be a bottleneck in Cephalo when there is a large degree of heterogeneity in networking bandwidth between GPUs. This is because Cephalo only employs data parallelism (DP), which uses collective operations like AllReduce that synchronize model updates across GPUs using all-to-all communication patterns. These collectives utilize links inefficiently in heterogeneous networks and can significantly increase communication latencies since faster links are bottlenecked by slower links.

Recall from Section 3.8 that heterogeneity in networking interconnects is common in Cloud VMs, for both homogeneous and heterogeneous clusters. This is due to differences in (1) intra-node networking, (2) inter-node networking, and (3) intra- and inter-datacenter networking. Figure 6.2 illustrates the network heterogeneity in bandwidth between common GPUs/VMs on AWS.

Instead of relying on DP alone, DP can be combined with pipeline parallelism (PP) which requires significantly less communication and can be used over slower inter-node or inter-datacenter links [171, 131]. Moreover, modern LLMs require far more memory than available on high end GPUs; hence using memory efficient variants of DP such as ZeRO-3 [144] (FSDP) are necessary to avoid running out of memory. However, combining ZeRO-3 with PP is inefficient since ZeRO-3 gathers parameter shards before each forward and backward computation (parameters are sharded afterwards), for each PP microbatch of computation. Consequently, the frequency of parameter gathering increases proportionally with the number of microbatches, creating a significant communication bottleneck and

slowing down training unless extremely fast interconnects are available, which is uncommon in heterogeneous clusters.

In light of the limitations from prior work, Zorse [59] was designed as a system which combines memory efficient DP with PP such that it is both simultaneously communication and memory efficient for training LLMs on heterogeneous Cloud clusters. Zorse directly addresses the inefficiencies of Cephalo when deployed in highly heterogeneous networking environments, and Section 6.2 and 6.6 provide a more detailed comparison of Cephalo and Zorse. Zorse combines PP and DP by partitioning the model into stages, pipelining computation across them, and applying DP within each stage. Zorse employs *Pipeline-Efficient ZeRO DP*, which reduces memory consumption in DP by sharding optimizer states and gradients. However, unlike existing PP approaches, each stage is divided into ministages and computation is interleaved in a way that avoids fully materializing stage parameters and gradients in memory. Moreover, both activations and parameters are offloaded to CPU memory when not in use, significantly reducing memory overhead. Zorse also supports *Heterogeneous Pipeline Parallelism*, allowing different stages to use varying numbers of GPUs, and each stage to be parallelized with different GPU configurations. This flexibility enables the configuration of more efficient training strategies in heterogeneous clusters. Finally, a planner was designed to efficiently navigate the large search space of configurations in Zorse, automatically optimizing training for a given workload and cluster.

Note that Hydrozoa, Cephalo, Zorse all employ planners to find efficient training configurations. However, each planner is tailored to optimize for the characteristics and configurable options available in each system.

6.1 Challenges in Heterogeneous Training

This chapter provides insights and establishes key challenges to overcome for achieving efficient training on heterogeneous GPU clusters.

6.1.1 Tensor Parallelism in Heterogeneous Clusters

Tensor parallelism (TP) is typically deployed only in clusters of high-end GPU machines interconnected by very high bandwidth technologies such as NVSwitch [131]. This is because TP requires frequent all-to-all communication between model layers. Without high-bandwidth interconnects, the overhead from these communications becomes prohibitively large, despite optimizations that overlap communication with computation [187]. Figure 6.1

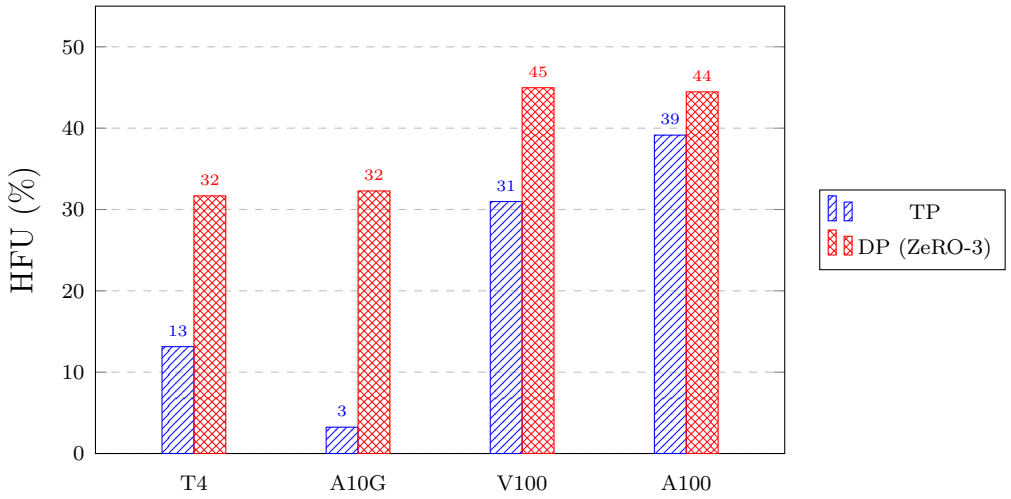


Figure 6.1: Hardware Flops Utilization (HFU [22]) of tensor parallelism (TP) vs data parallelism (DP) with ZeRO-3 on 8 GPU AWS VMs.

compares the GPU utilization when training with TP versus data parallelism (DP) across common AWS VMs. The plot shows that GPU utilization with TP is low relative to DP for most VMs, and is only comparable to DP on the 8×A100 VM that has high bandwidth NVSwitch. Moreover, TP is generally advantageous only in extremely large training scales where DP or PP become impractical:

(1) Exceptionally large models: TP can train very large models where individual layers exceed the GPU's memory capacity, as it partitions the computation within each layer.

(2) Extremely large clusters: On very large clusters with 1024+ GPUs, using only DP can result in excessively large batch sizes that impact model convergence. TP can be used to distribute training across more GPUs without further increasing the batch size [131, 187].

However, organizations using heterogeneous clusters are typically resource constrained, and therefore would not be training at the scale of models and batch sizes that require TP. Moreover, heterogeneous clusters often feature low- or mid-tier GPUs with limited networking bandwidth, since it is well established that high-end GPUs are limited in availability [57, 139, 179].

Takeaway #1: TP is unsuitable for most heterogeneous training due to its high communication overhead and misalignment with practical heterogeneous training scenarios.

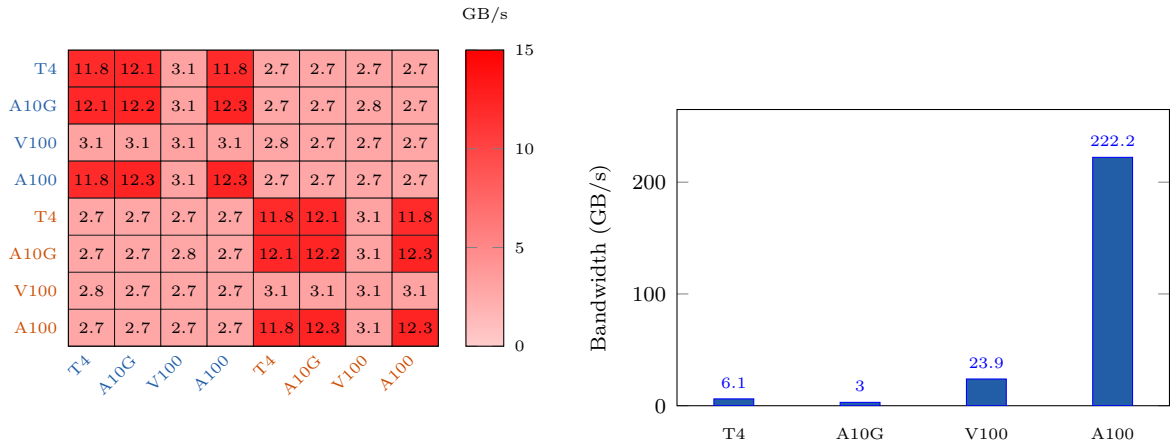


Figure 6.2: Unidirectional bandwidth on AWS. Left: Heatmap of inter-node bandwidth between VMs of different GPUs across two regions, us-east-1 and us-east-2. Right: Intra-node bandwidth within VMs. For each bandwidth value the mean of several measurements is reported. Variability was observed to be low with a standard deviation of less than 2.5%.

6.1.2 Data Parallelism in Heterogeneous Networks

In homogeneous clusters with high-bandwidth interconnects, DP has been shown to scale efficiently to large clusters of over 512 GPUs [183, 144, 134]. However, the scalability of DP is limited in heterogeneous clusters due to substantial network variability caused by differences in (1) intra-node networking, (2) inter-node networking, and (3) intra- and inter-datacenter networking. Figure 6.2 illustrates the network heterogeneity in bandwidth between common GPUs/VMs on AWS.

DP uses collective operations like AllReduce, which synchronize model updates across GPUs using all-to-all communication patterns. However, these collectives utilize links inefficiently in heterogeneous networks and can significantly increase communication latencies since faster links are bottlenecked by slower links. Moreover, inter-datacenter bandwidth is typically insufficient to train efficiently with DP [171, 200, 43]. Instead of relying on DP alone, DP can be combined with pipeline parallelism (PP) which requires significantly less communication and can be used over slower inter-node or inter-datacenter links [171, 200].

Takeaway #2: Efficiently scaling training over heterogeneous networks with slow links necessitates a combination of data parallelism and pipeline parallelism.

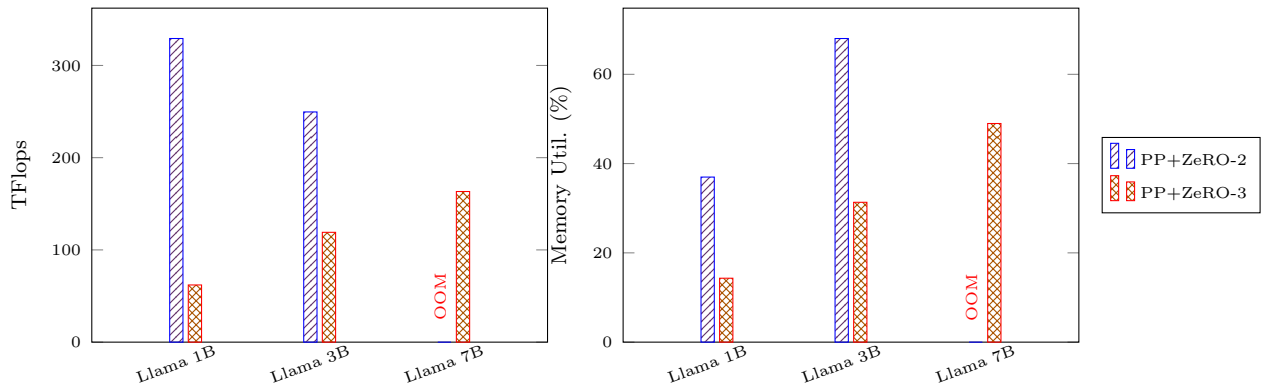


Figure 6.3: Comparison of pipeline parallelism (PP) with ZeRO-2 vs ZeRO-3 across different Llama[177, 208] model sizes on 8 V100s + 8 T4s. OOM indicates Out-of-Memory.

6.1.3 PP + DP in Heterogeneous Clusters

ZeRO-3 [144, 183] (or FSDP [213] in PyTorch) is a popular variant of DP designed to avoid redundant storage of training state data across GPUs. While this method significantly reduces the memory requirement for storing training state data by a factor proportional to the number of GPUs in the cluster, it also increases communication overhead. Specifically, ZeRO-3 must gather parameter shards before each forwards and backwards computation (parameters are sharded afterwards). This overhead is exacerbated when ZeRO-3 is combined with PP, as PP divides a batch of inputs into microbatches that are pipelined to overlap computation across pipeline stages. Consequently, the frequency of parameter gathering increases proportionally with the number of microbatches, creating a significant communication bottleneck and slowing down training unless extremely fast interconnects are available, which is uncommon in heterogeneous clusters.

Breadth-first PP [93, 92, 55] amortizes the communication cost of gathering parameters in ZeRO-3 by swapping the order of computation [57, 92]. Instead of processing all layers for one microbatch before moving to the next, these approaches sequentially process all microbatches for one layer before proceeding to the next layer. However, this strategy requires every stage of the model to contain a single layer. While this works well for homogeneous clusters, on heterogeneous clusters, single-layer stages will often result in imbalanced computational loads across PP groups due to differing computational speeds across GPUs. However, breadth-first PP also introduces a large pipelining overhead when ministage computation is balanced to have more than 1 layer. Figure 6.4 illustrates this pipelining overhead with an example of how Zorse's pipeline compares with a breadth-first

strategy.

Appendix B explains the large pipelining overhead that is introduced in heterogeneous clusters.

Alternatively, PP can be combined with ZeRO-2, which shards optimizer state and gradients but keeps a full set of parameters in memory. This avoids the need to gather parameters before each computation, achieving higher training throughput but significantly increases memory usage, as shown in Figure 6.3. This is especially problematic in heterogeneous clusters, where some GPUs can be memory constrained relative to their compute capacity [57]. In Figure 6.3, although both V100s and T4s have the same memory, V100s are more than three times faster. As model size increases, V100s in PP + ZeRO-2 are unable to process a workload proportional to its compute capacity due to memory constraints, resulting in lower training throughput.

Takeaway #3: Existing variants of DP cannot be combined with PP without introducing significant communication, pipelining, or memory overhead.

6.1.4 Symmetric Parallelism Configuration

Most state-of-the-art distributed training systems [102, 146, 124, 131] integrate PP with DP by organizing GPUs into a 2D mesh, with PP along one axis and DP on the other. This configuration simplifies system design and narrows the parallelism plan search space. However, in heterogeneous clusters, this approach may overlook efficient training configurations. Such clusters often contain asymmetry, with variations in GPU count per node, hardware, and datacenter location. Efficient training configurations would ideally group GPUs of the same datacenter, node, or GPU type for DP, while applying PP across different groups, ensuring that GPUs within each DP group share similar networking and computational capabilities. Moreover, GPUs may need to be grouped into varying group sizes to accommodate differences in memory capacity (e.g. GPUs with less memory require smaller groups to reduce the memory overhead of parameter replication). Supporting such configurations is important for heterogeneous training, and requires a system capable of asymmetrically partitioning GPUs into PP stages of different sizes.

Takeaway #4: Supporting PP with variably-sized DP stages is essential for optimizing training on heterogeneous clusters.

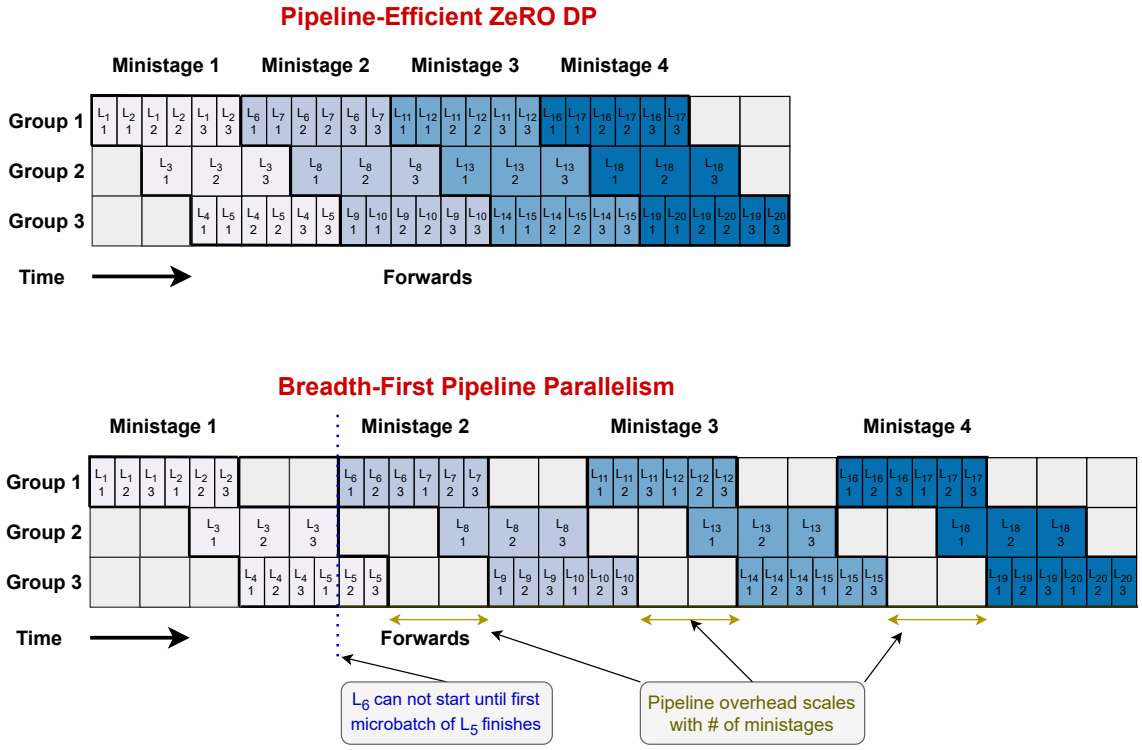


Figure 6.4: This diagram contrasts the forwards pass in Zorse compared to breadth-first pipeline parallelism. The model has 20 layers (L_i) and there are 3 microbatches. There are 3 GPU groups, each with 4 ministages (Same as Figure 6.7 of the paper). In breadth-first pipeline parallelism, communication cannot be effectively overlapped with computation efficiently since the subsequent stage cannot start until the current stage reaches the first microbatch of the last layer.

6.1.5 Pipelining Overheads

Training systems optimized for homogeneous clusters [102, 146, 124, 131, 214] typically distribute batch sizes evenly across GPUs within a DP stage. This uniform allocation results in training bottlenecks when heterogeneous GPUs share a stage, as faster GPUs will be blocked waiting for slower GPUs within the same stage. Avoiding this issue in heterogeneous environments restricts GPUs within a DP group to be of the same type, which therefore scales the number of PP stages by the number of GPU types. However, adding more pipeline stages can introduce inefficiencies, such as increased idle time (bubbles [206]) and additional inter-stage communication. Moreover, increasing the number of stages constrains the flexibility in distributing model layers across stages to achieve balanced computation, especially considering the relatively few layers in modern LLMs. Consequently, it is important for heterogeneous training systems to support balancing of computational loads across different GPUs within a stage. This capability allows for configurations with fewer stages that incur lower pipelining overhead.

> **Takeaway #5**: Balancing computational loads across heterogeneous GPUs within a DP stage facilitates training configurations with reduced pipelining overhead.

6.2 Limitations of Existing Systems

Given the challenges discussed above, this chapter examines the limitations of representative systems that embody state-of-the-art methods for heterogeneous training.

General 3D Parallel Frameworks. Systems such as TorchTitan [102] and DeepSpeed [146] support 3D parallelism with DP, PP, and TP, but are not explicitly designed for heterogeneous clusters. In heterogeneous clusters, these frameworks need to be manually configured with asymmetric model partitions to balance computation time across stages in PP. However, they do not provide the flexibility to asymmetrically partition GPUs across pipeline stages or to use different batch sizes within DP groups to accommodate heterogeneous GPUs within a stage. Additionally, its DP can be optimized for either memory or communication efficiency, but not both simultaneously. For instance, in TorchTitan, DP can configured to use either ZeRO-2 which is communication efficient but not memory efficient, or ZeRO-3 which is memory efficient but not communication efficient.

Heterogeneity-Aware 3D Parallel Frameworks. Frameworks such as HexiScale [194], Metis [179], and Sailor [172] are designed for 3D parallelism in heterogeneous clusters.

They support asymmetric PP and permit varying batch sizes within a DP group, enabling balanced computation across different GPUs. However, some parameters still require manual tuning, e.g., HexiScale requires specifying the batch size distribution for DP and microbatches for PP. Additionally, these frameworks do not support ZeRO-3 due to communication bottlenecks when integrated with PP. Instead, they employ ZeRO-2 and standard DP, respectively. When PP and DP exceed the cluster's memory capacity, TP is employed to reduce memory usage, but this leads to poor utilization in heterogeneous clusters (Section 6.1.1).

Heterogeneity-Aware ZeRO-3. Systems like Cephalo [57] and Poplar [211] utilize ZeRO-3 for training on heterogeneous clusters. Instead of using PP or TP to scale training for larger batch sizes, Cephalo employs gradient accumulation to reduce the memory requirements by splitting large batch sizes into smaller microbatches. It also reorders the computation to avoid regathering parameters for each microbatch. However, due to the lack of support for PP, Cephalo cannot efficiently utilize network resources in highly heterogeneous networks, resulting in networking bottlenecks when the batch size is insufficient to hide communication latencies [57].

System	Communication Efficient DP	Memory Efficient DP	Computation Balancing DP	PP +DP	Asymmetric PP
TorchTitan-ZeRO2 [102]	✓	✗	✗	(✓)	✗
TorchTitan-ZeRO3 [102]	✗	✓	✗	(✓)	✗
HexiScale [194]	✓	✗	(✓)	(✓)	✓
Cephalo [57]	✓	✓	✓	✗	✗
Zorse (Our System)	✓	✓	✓	✓	✓

✓ Supported, (✓) Supported with manual tuning, ✗ Not supported

Table 6.1: Comparison of heterogeneous training systems.

6.3 Design

Zorse trains LLMs with *Pipeline-Efficient ZeRO DP*, which combines PP and DP in a way that is simultaneously communication- and memory-efficient. Zorse also supports *Heterogeneous Pipeline Parallelism*, allowing asymmetric partitioning of GPUs across pipeline stages and varying batch sizes within a DP group. Finally, Zorse's planner automatically determines optimized training configurations for a given workload. Table 6.1

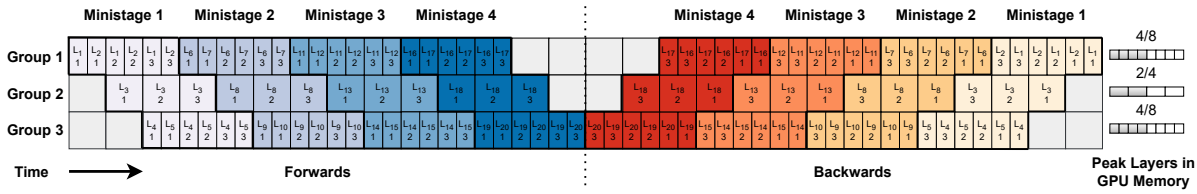


Figure 6.5: Interleaved pipelining in Zorse. The diagram shows a training iteration for a model with 20 layers (L_i) and 3 microbatches. There are 3 GPU groups, each with 4 ministages. Due to different computational speeds, groups have different numbers of layers per ministage. Each group maintains the current and next ministage, offloading others to CPU memory.

compares Zorse with existing systems. Zorse is the first to integrate all essential capabilities for efficient training on heterogeneous clusters. While Zorse focuses on using PP and DP for training, Section 6.5.6 extends Zorse to support TP and compares its performance to DP. This chapter describes the components of Zorse in detail.

6.3.1 Pipeline-Efficient ZeRO Data Parallelism

Pipeline-Efficient ZeRO DP in Zorse is a novel integration of PP and ZeRO-based DP that leverages interleaved pipelining and offloading to reduce memory utilization while avoiding large communication overheads. Instead of conventional PP which assigns one stage per GPU, Zorse uses interleaved pipelining which assigns multiple smaller *ministages* to each GPU and interleaves computation across them. However, traditional interleaved pipelining [131] uses a 1F1B [128] schedule, interleaving computation across ministages. This means that ministage parameters cannot be sharded without incurring extra communication overhead.

Interleaved Pipelining

Figure 6.5 illustrates how computation is interleaved in Pipeline-Efficient ZeRO DP. The forwards pass is processed for each ministage sequentially, finishing all microbatches for a ministage before moving on to the next. The backwards pass is processed similarly in the reverse order. Since the microbatches are all processed together for each ministage, ministages that are not actively being used are offloaded to CPU memory. With this design, Zorse needs to maintain only the parameters for 2 ministages in memory at a time (the

current ministage and the next ministage that is being prefetched in parallel), rather than all the parameters in the stage in standard PP + ZeRO-2. As in ZeRO, the gradients and optimizer state are sharded across each DP group.

Let L be the total number of layers in the stage, S be the number of stages, P_{layer} be the number of parameters in each layer, D_{dp} be the degree of DP, and M be the number of microbatches. Table 6.2 summarizes the parameter memory and communication requirements of Zorse (Pipeline-Efficient ZeRO DP) , PP + ZeRO-2, and PP + ZeRO-3. As the number of ministages increases, the memory utilization of Zorse approaches that of PP + ZeRO-3 and is significantly lower than PP + ZeRO-2. Additionally, Zorse has similar communication requirements as PP + ZeRO-2, with one AllGather per layer in the forwards and backwards pass. This is significantly lower than PP + ZeRO-3, which requires one AllGather per microbatch of each layer.

Strategy	**Materialized Parameters**	**Sharded Parameters**	**# AllGathers**
Zorse	$2 \times \frac{L}{S} \times P_{layer}$	0	$2 \times L$
PP + ZeRO-2	$L \times P_{layer}$	0	L
PP + ZeRO-3	$2 \times P_{layer}$	$(L-2) \times \frac{P_{layer}}{D_{dp}}$	$2 \times L \times M$

Table 6.2: Memory and communication comparison of different training strategies.

To offload parameters, the host machine must have sufficient memory to store all ministages for all GPUs in CPU memory. Modern GPU VMs have much more CPU RAM than GPU VRAM and this requirement is satisfied by our VMs for the largest workloads. For example, the largest model we evaluate requires 130 GB of host memory, whereas the AWS 8x A100 VMs used provide 1152 GB. The PCIe bandwidth on our VMs supports per-GPU 16 GB/s CPU-GPU bandwidth, which is sufficient to ensure that CPU-GPU transfers do not bottleneck training in practice.

Interleaved Optimizer Updates

Zorse further takes advantage of its interleaved pipelining schedule to interleave optimizer updates. Instead of starting the optimizer update after the backwards pass completes for all layers, Zorse starts the optimizer update for each ministage as soon as its own backwards pass completes, while the computation for the next ministage is simultaneously starting. This design (Figure 6.6) offers two key advantages:

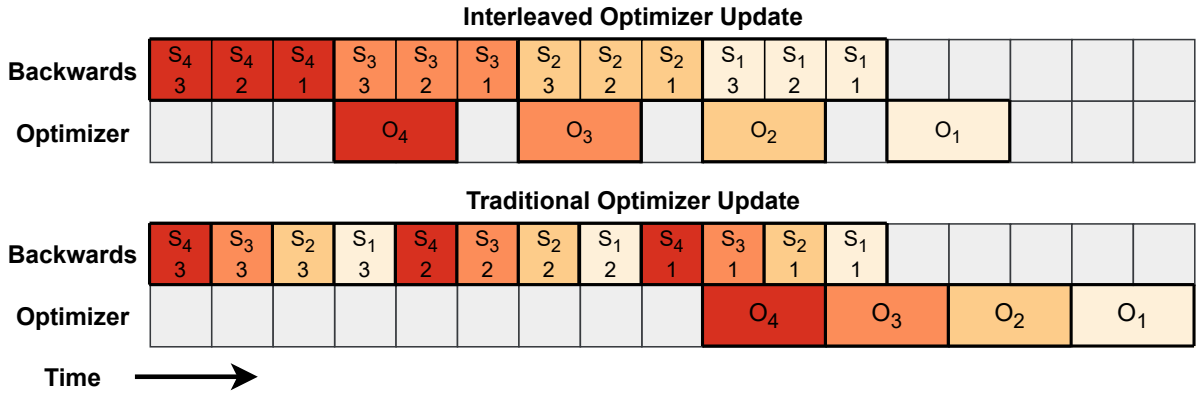


Figure 6.6: Interleaved optimizer updates in Zorse vs. traditional pipeline parallelism (4 ministages, 3 microbatches). Interleaved optimizer updates free gradient memory earlier and better overlap with computation.

(1) Reduced peak memory utilization: By freeing the gradients of each ministage as soon as its optimizer update is finished, Zorse avoids the need to retain gradients until the entire backward pass is complete.

(2) Decreased communication overhead: The optimizer update requires averaging gradients across GPUs within its DP group. By interleaving optimizer updates, this communication can overlap with the computation of subsequent ministages. This overlap is not possible if all optimizer updates start at the end of the pipeline schedule.

Activation Checkpoining & Offloading

As in prior work [131, 142, 194], Zorse checkpoints activations after each transformer layer, discarding activations between layer boundaries and recomputing them when they are needed in the backwards pass. However, the remaining activations at layer boundaries still require a significant chunk of GPU memory due to the pipeline schedule. Sequential processing of all forwards passes before backwards passes means that layer boundary activations for all microbatches of all layers need to be maintained in memory. With a batch size of B, L layers, sequences of length S, a hidden layer size of H, and D bytes per parameter, the memory overhead is $B \times L$ activations of size $S \times H \times D$ bytes. Even at smaller training scales, this can amount to many GBs of memory.

Thus, in addition to offloading parameters, Zorse also offloads layer boundary activations to CPU memory during the forwards pass, which are loaded back during the backwards

pass. With offloading, Zorse needs to maintain activations for only the current microbatch being computed and for the next microbatch being prefetched.

Section 6.4.4 describes how Zorse efficiently overlaps offloading (and loading) of activations, and model parameters with computation to hide offloading overhead.

6.3.2 Heterogeneous Pipeline Parallelism

Zorse supports *heterogeneous pipeline parallelism*, which allows for each pipeline stage to have: (1) a different number of GPUs, and (2) a combination of different GPU types.

This differs from traditional PP, which fixes a uniform number of GPUs per stage and the same GPU type within each stage. This flexibility allows Zorse to more efficiently configure PP with DP in heterogeneous clusters, aligning with the varying numbers of each GPU type, varying numbers of GPUs per VM, and varying memory and compute capabilities in heterogeneous clusters.

Figure 6.7 illustrates heterogeneous PP on a 3-node cluster with 1, 2, and 4 GPUs of different types across two regions. With heterogeneous PP, nodes in Region 1 form a single stage with 3 GPUs, while Region 2's node forms another stage with 4 GPUs. This approach leverages high-bandwidth intra-region connections for DP communication while using PP across the lower-bandwidth inter-region link, reducing pipeline stages to just two. Traditional PP, however, requires six single-GPU stages to maintain GPU uniformity, significantly increasing pipeline overhead. Moreover, with more stages, it may become infeasible to partition the model layers in a way that evenly distributes computation across stages. This is especially true for LLM models, which have a limited number of layers.

Cross-stage Communication Algorithm. In traditional PP, communication is straightforward due to the uniformity of GPUs both within and across stages, allowing for one-to-one communication between GPUs in different stages. However, in heterogeneous PP where the number and type of GPUs per stage varies, data must be reshuffled across stages, necessitating many-to-many communication patterns. Additionally, data must be balanced across heterogeneous GPUs within a stage to evenly distribute computation.

Zorse computes an optimized communication plan to redistribute microbatches across stages and balance computation within stages. Microbatches are assigned to GPUs based on their relative layer runtimes. This algorithm estimates completion times for microbatches of the current stage and remaining compute time for each GPU in the next stage. It then assigns the ith completed microbatch to the GPU with the ith highest remaining runtime. This assignment sends each microbatch to the GPU with the most work remaining to process in order to minimize overall stage runtime.

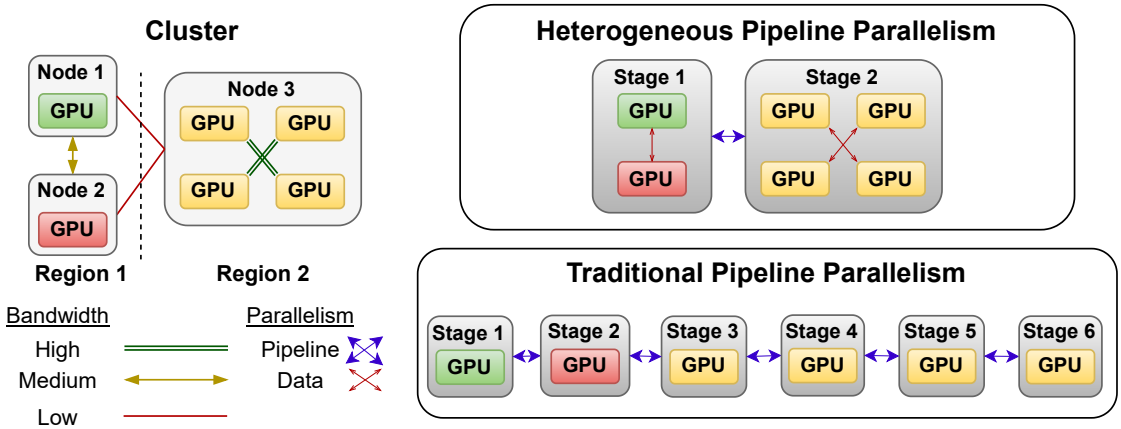


Figure 6.7: Heterogeneous pipeline parallelism in Zorse compared to traditional pipeline parallelism.

Algorithm 3 presents the pseudocode for the greedy algorithm used to compute an efficient communication plan between adjacent pipeline stages in a heterogeneous cluster. Note that each stage has $microbatchesPerRank$ defined based on the compute ratios of the GPUs. The system computes this plan statically before execution begins based on compute latencies profiled from each GPU. The algorithm takes as input the current pipeline stage configuration, the next pipeline stage configuration, and a mapping of GPU types to their compute latencies. It produces a communication plan that specifies:

1. For each microbatch in the current stage: the rank in the next stage to send outputs to
2. For each microbatch in the next stage: the rank in the current stage to receive inputs from
3. Unique tags for all forward and backward communications

6.3.3 Planner

Zorse initially profiles the cluster and workload to gather model runtime and networking statistics. This data is then utilized by the planner to determine the final training configuration. Zorse's planner employs a two-phase optimization process to optimize the training configuration:

Algorithm 3 Greedy Microbatch Distribution

Input: Current stage S_c, Next stage S_n, GPU compute time mapping L.
Output: Communication plan between stages
$C \leftarrow \emptyset$ ▷ List to store microbatch completion times
for each rank $r \in S_c$ **do**
    for each microbatch m from 0 to numMBs$(r) - 1$ **do**
        Append $(m \times L_r, r)$ to C ▷ When this microbatch completes
$W \leftarrow \emptyset$ ▷ List to store remaining work per GPU
for each rank $r \in S_n$ **do**
    for each microbatch m from 0 to numMBs$(r) - 1$ **do**
        Append $((M_r - m) \times L_r, r)$ to W ▷ Remaining work after m microbatches
Sort C by completion time (earliest first)
Sort W by remaining work (most work first)
Initialize microbatch indices for all ranks to 0
Initialize empty communication plan P
for each pair $(r_c, r_n) \in \text{zip}(C, W)$ **do** ▷ Match by sorted order
    $i_c \leftarrow$ current index for rank r_c
    $i_n \leftarrow$ current index for rank r_n
    $P[r_c][i_c]$.forward_send_to $\leftarrow r_n$ ▷ Set forward path
    $P[r_n][i_n]$.forward_recv_from $\leftarrow r_c$
    $P[r_c][i_c]$.backward_recv_from $\leftarrow r_n$ ▷ Set backward path
    $P[r_n][i_n]$.backward_send_to $\leftarrow r_c$
    Increment microbatch index for r_c and r_n
return P

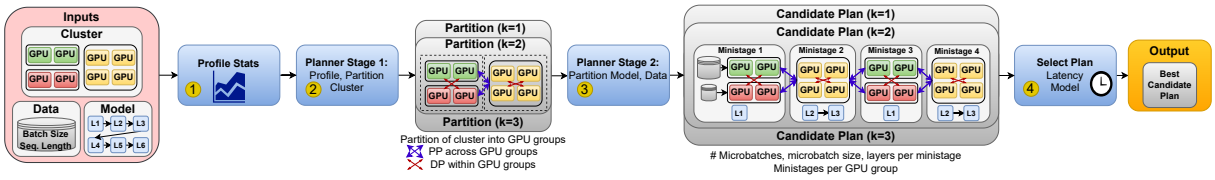


Figure 6.8: Architecture of Zorse's planner: ① profile workload and cluster, ② partition cluster into GPU groups, ③ partition model and data across GPU groups, and ④ select the best plan considered.

- In Phase 1, the cluster is partitioned into *GPU groups*. PP will be applied across GPU groups and DP within each GPU group. The planner considers the best way to partition the cluster into k GPU groups, for $k \in [1, N]$ where N is the number of GPUs in the cluster.
- In Phase 2, the planner computes an optimized configuration for each cluster partition from Phase 1. This includes the number of microbatches, the size of each microbatch, the number of ministages per GPU group, and the partitioning of the model into these ministages.

The architecture of Zorse's planner is shown in Figure 6.8.

Profiling.

The system measures inter-node and intra-node bandwidths between pairs of nodes and GPUs, respectively, for use in cluster partitioning. Model layer runtimes on each GPU are profiled for small batch sizes and a linear model is fitted to predict runtimes for unseen batch sizes, which is a lightweight yet accurate method for estimating runtime [57, 179]. These bandwidth measurements and runtime predictions are used to estimate training latencies in Phase 2. To accelerate profiling, the system: (1) parallelizes intra-node profiling, and (2) profiles inter-node bandwidths only between unique VM configurations, as large clusters often contain redundant configurations. Further profiling details are provided in Appendix C.1.

Phase 1: Cluster Partitioning

In the first phase, the planner determines how to divide the cluster nodes into k GPU groups. GPUs within the same group utilize DP, while PP is applied across groups. Therefore, the goal is to create a partition that minimizes inter-group bandwidth usage. This results in an efficient setup that uses communication-efficient PP over lower bandwidth links between groups and DP over higher bandwidth links within groups. Fixing the intra-group parallelism to DP effectively prunes the search space of possible parallelism configurations, while still capturing the most efficient ones. In particular, configurations that apply DP over lower-bandwidth links are likely to be communication bottlenecked, and are therefore unlikely to be optimal.

The cluster is represented as a fully connected graph $G = (V, E)$, where V represents the GPUs and E represents the edges with weights $w(u, v)$ indicating the bandwidth between

GPUs u and v. The partitioning task is framed as a min-k cut problem on this graph. Specifically, the objective is to divide V into k disjoint subsets $V_1, V_2, \ldots, V_k$ such that the total weight of the edges between different subsets is minimized. This is called the *min-k cut*:

$$\min \sum_{\substack{u \in V_i, v \in V_j \\ i \neq j}} w(u,v), \quad \text{where} \bigcup_{i=1}^{k} V_i = V,\ V_i \cap V_j = \emptyset \text{ if } i \neq j \quad .$$

This partitioning minimizes the bandwidth sum between GPU groups, ensuring that higher bandwidth links are used within groups. This problem is solved for each k from 1 to the total number of GPUs in the cluster, N.

Since exact min-k cut solutions have an impractically large $O(N^{k^2})$ time complexity [53], we use the SPLIT greedy approximation algorithm [154], which guarantees a solution within a factor of $2 - 2/k$ of optimal. SPLIT iteratively computes min 2-cuts of the current graph and removes those edges until k connected components remain. Thus, we can efficiently generate approximate min k cuts for all values from 2 to N in a single execution with $k = N$. With each min 2-cut requiring $O(N^3)$ time [170] and N iterations needed, the overall time complexity of cluster partitioning is $O(N^4)$.

Phase 2: Model Configuration

In the second phase, the planner utilizes cluster partitions from Phase 1 to determine an optimized training configuration, which encompasses model layer partitioning into ministages, GPU group ordering, and microbatch size selection. Due to the extensive search space, Zorse employs heuristics to prune candidate configurations before evaluating the remaining options with a latency model for performance assessment and a memory model to verify GPU memory constraints are respected.

Heuristics. In pipeline parallelism, the pipeline is bottlenecked by the slowest GPU group. Thus, to minimize training latency, runtime across GPU groups needs to be balanced by partitioning model layers proportionally to each group's aggregate processing speed, estimated as the sum of layer processing rates across GPUs in the group. Layers assigned to each GPU group are divided into evenly sized ministages, which are then ordered across groups in round robin fashion.

The order of GPU groups impacts pipeline startup time because the first ministage must gather sharded model parameters without having prior ministages to overlap this communication. GPU groups are ordered by descending intra-group bandwidth to minimize

initial communication delay and expedite pipeline startup. This arrangement enables subsequent GPU groups to better overlap their communication with computation from preceding GPU groups.

Enumeration. The system enumerates over all possible configurations of batch size and number of ministages per GPU group, which is at most the number of model layers. Hence, there are only $O(B \cdot L)$ configurations to consider, where B is the global batch size and L the number of layers. The lightweight latency and memory model is used to select the configuration with the lowest estimated latency that satisfies the memory constraints.

Latency Model.

The total training latency is modeled as:

$$L_{total} = (L_{forwards} + L_{backwards}) \cdot N_{ministages} + L_{startup} \quad . \tag{6.1}$$

where $L_{forwards}$, $L_{backwards}$ are the latencies to process one ministage in the forward and backward passes, respectively, $N_{ministages}$ is the number of ministages per GPU group, and $L_{startup}$ is the latency to initialize the pipeline.

Ministage Forward & Backward Pass Latency. Let $F_{i,g}$ be the forward pass latency for running a ministage for all microbatches on GPU g in GPU group i, and L_{send} be the latency required to send all microbatches between neighboring GPU groups. The compute latency for processing one ministage in the forward pass of the pipeline is:

$$C_{forwards} = \max_{G_i} \max_{g \in G_i} \max(F_{i,g}, L_{send}) \quad . \tag{6.2}$$

The inner max accounts for the overlap between computation and communication of microbatches, while the outer max terms model the forwards pass being bottlenecked by the slowest ministage across all GPU groups. To account for the overlap of the forward computation with the AllGather of parameters, the overall latency for a ministage in the forward pass is:

$$L_{forwards} = \max(C_{forwards}, L_{AG}, L_{offload}) \quad . \tag{6.3}$$

where L_{AG} is the latency of the AllGather for the ministage parameters and $L_{offload}$ is the latency for offloading and reloading both ministage parameters and activations. The AllGather latency is modeled based on profiled network bandwidths, assuming a ring-based AllGather implementation [75, 171]. Given a ministage size of P GB and $|G_i|$ GPUs in the group, each GPU will transmit a total of $\frac{(|G_i|-1) \cdot P}{|G_i|}$ GB of data. This data exchange occurs

in parallel and is bottlenecked by the slowest network bandwidth b_{min} between any two GPUs in the group. Thus, the communication time for AllGather is:

$$L_{AG} = \frac{(|G_i| - 1) \cdot P}{|G_i| \cdot b_{min}} \quad . \tag{6.4}$$

Offloading latency can be estimated by dividing the total bytes offloaded and loaded for model parameters and activations by the PCIe bandwidth.

The backward computation (which includes the forwards recomputation of activations) is overlapped with another AllGather of parameters along with ReduceScatter to average the gradients:

$$L_{backwards} = \max(C_{backwards}, L_{AG} + L_{RS}, L_{offload}) \quad . \tag{6.5}$$

where $C_{backwards}$ is the latency for processing a ministage in the backwards pass, L_{RS} the latency for ReduceScatter, modeled like $C_{forwards}$ and L_{AG}.

Pipeline Startup Latency. The pipeline startup latency $L_{startup}$ is computed by summing the AllGather latency of the first ministage and the send latencies for the first $K - 1$ microbatches, which are not overlappable with any computation:

$$L_{startup} = L_{AG} + L_{send} \cdot (K - 1) \quad . \tag{6.6}$$

where K is the number of GPU groups.

Memory Model.

The total per-GPU memory consumption is modeled as:

$$M_{total} = M_{params} + M_{grads} + M_{optim} + M_{activations} \quad . \tag{6.7}$$

where M_{params} represents memory for model parameters, M_{grads} gradients, M_{optim} optimizer states, and $M_{activations}$ activations. Let N_{TL} and N_{ML} be the total number of layers assigned to a GPU group G, and the number of layers per ministage, respectively. In FP16 mixed-precision training, a master copy of the model parameters is kept sharded in full precision, while at most 2 layers in half precision are materialized in memory. Then, total memory (bytes) for the model parameters is

$$M_{params} = 4 \cdot \left(\frac{N_{TL} \cdot P_{layer}}{|G|} \right) + 2 \cdot (2 \cdot N_{ML} \cdot P_{layer}) \quad .$$

With Pipeline-Efficient ZeRO DP, there are gradients for at most one ministage in memory at a time. The gradient memory is

$$M_{grads} = 2 \cdot (N_{ML} \cdot P_{layer}) \quad .$$

With a standard optimizer like Adam [86], the first and second moments of the gradient are stored as optimizer state in full precision, and so the sharded optimizer state requires

$$M_{optim} = 4 \cdot \left(2 \cdot \frac{N_{TL} \cdot P_{layer}}{|G|} \right) \quad .$$

Finally, with offloading, only two sets of boundary activations are kept in memory, which amount to

$$M_{activations} = 2 \cdot (b \cdot s \cdot h) \quad .$$

where b is the microbatch size, s is the sequence length, and h is the model hidden dimension.

6.4 Implementation

This chapter provides details on Zorse's implementation and optimizations. Zorse is built on top of FSDP [213] (ZeRO-2 and ZeRO-3 in PyTorch). Implementing Pipeline-Efficient ZeRO DP and Heterogeneous PP requires adding logic to manage ministage parameters, pipeline microbatches across GPU ministages, and communicate between GPU groups.

6.4.1 Pipeline-Efficient ZeRO DP

FSDP's parameter management is modified to support interleaved pipeline parallelism by addressing key timing assumptions. Unlike FSDP's ZeRO-2 which keeps parameters materialized between forward and backward passes, the implementation reshards and offloads ministage parameters to CPU post-forward pass, then reloads and unshards during the backward passes. To handle FSDP's premature resharding after the first microbatch's backward pass, the implementation delays resharding until all microbatches complete. Memory usage is limited by configuring the FSDP prefetcher to fetch only the next immediate ministage. Finally, per-ministage optimizers are created that execute independently after their corresponding backward passes complete.

6.4.2 Communication Libraries

NCCL [137] is used for DP communication within GPU groups. For PP communication, limitations were encountered with NCCL's blocking P2P behavior [136], which can cause deadlocks in heterogeneous PP due to cyclic dependencies. A hybrid approach is implemented using NCCL where possible and GLOO P2P (non-blocking) between GPU groups where cycles in communication are possible. While GLOO cannot utilize NVLink, this was not a performance bottleneck as NVLink typically connects GPUs within the same DP group rather than across PP groups. More details are provided in Appendix B.1 and Appendix B.2.

6.4.3 Computation-Networking Overlap

Communication-computation overlap is optimized by managing parameter sharding at the layer level instead of ministage level. By gathering layers sequentially rather than as a single large chunk, computation can begin once the first layer is ready while subsequent layers are prefetched in parallel. This approach extends to the backward pass, where CPU-offloaded parameters are prefetched ahead of when they are needed, preventing computation stalls.

6.4.4 Computation-Offloading Overlap

PyTorch's built-in CPU offloading incurs significant performance overhead by blocking GPU computation and executing optimizer steps on CPU when parameter offloading is enabled. A custom implementation utilizes separate CUDA streams for parameter and activation offloading without blocking GPU computation. Similarly, activations and parameters are prefetched in advance to prevent stalling during backward pass. Finally, optimizer steps are executed on GPU prior to offloading to avoid costly optimizer updates on CPU.

6.5 Experiments

Zorse is compared to state-of-the-art heterogeneous training systems across three representative clusters with up to 128 GPUs, and LLMs with up to 65B parameters. Training throughput and cluster utilization are evaluated in Section 6.5.1 and 6.5.2. Zorse's training efficiency on heterogeneous clusters is evaluated in Section 6.5.3, and how different components of Zorse's design contribute to its performance is investigated in Section 6.5.4 and 6.5.5. Finally, the optimizer runtime and accuracy are analyzed in Section 6.5.7.

Model	Layers	Embd. Size	Attn. Heads	Parameters
Llama 1B [208]	22	2048	32	1B
Llama 3B [177]	26	3200	32	3B
Llama 7B [177]	32	4096	32	7B
Llama 13B [177]	40	5120	40	13B
Llama 33B [177]	60	6656	52	33B
Llama 65B [177]	80	8192	64	65B
GPT 6.7B [15]	32	4096	32	6.7B
GPT 13B [15]	40	5120	40	13B

Table 6.3: Model Statistics

Models: The popular LLama 7B - 65B [177] and GPT [15] 6.7B - 13B models are used for evaluation in this chapter, and their smaller variants [208] for benchmarking PP + DP in Section 6.1. Model specifications are provided in Table 6.3.

Experimental Setup: For all systems, training is performed with a global batch size of 1 million tokens such that training dynamics and convergence are consistent. Activations are checkpointed after each transformer layer and models are trained with the Adam optimizer [86] and in FP16 mixed-precision [125] to take advantage of optimizations on newer GPUs like the A100.

Training performance is evaluated using two key metrics: (1) TFlops: the floating point operations per second achieved during training, which measures training throughput; (2) Hardware FLOPS Utilization (HFU [22]): the ratio of achieved TFlops to the cluster's peak theoretical TFlops, which quantifies GPU utilization efficiency. Note that the differences of Zorse compared to baselines in HFU is proportional to differences in throughput and inversely proportional to differences in iteration time. For example, a system with 2x higher HFU achieves 2x higher throughput and trains with half the iteration time. TFLOPs are directly comparable since all systems use the same batch size and enable activation recomputation.

Clusters. Evaluation is performed on three clusters representative of common heterogeneous training scenarios. These clusters include VMs from Azure and AWS, contain up to 128 GPUs of low-, mid-, and high-end GPUs, and span up to 2 regions. Table 6.4 details

GPU specifications, while Table 6.5 presents the cluster configurations. Sequence lengths of 4096, 1024, and 512 are used for clusters A, B, and C respectively. Longer sequences are used on clusters with more powerful GPUs.

Performance	GPU	Memory	TFlops (FP16)
High	H100-NVL	94 GB	989
	A100	40/80 GB	312
Middle	V100	16 GB	125
	A10G	24 GB	125
Low	T4	16 GB	65

Table 6.4: Datacenter-class GPU Specifications

Cluster	Cloud	# VMs	GPUs / VM	Total Regions	Total GPUs	Total TFlops (FP16)
A	Azure	2	2×H100	1	20	8332
		2	8×A100 (80GB)			
B	AWS	1	8×A100 (40GB)	1	64	8112
		2	8×A10G			
		2	8×V100			
		3	8×T4			
C	AWS	2	8×A10G	2	128	8240
		2	8×V100			
		12	8×T4			

Table 6.5: Cluster GPU Configurations

Baselines. For our baselines, we selected systems that are representative of state of the art techniques for heterogeneous training, open source, and readily adaptable to run our workloads. TorchTitan-Het is reflective of how popular homogeneous training frameworks would be adapted for heterogeneous training, while Hexiscale and Cephalo are specifically designed for heterogeneous training.

- TorchTitan-Het: 3D parallelism in TorchTitan [102] adapted for heterogeneous clusters by partitioning the model unevenly to balance compute across PP stages. Different combinations of DP (ZeRO-2), PP, and TP are tried, reporting the best performing configuration.

- HexiScale [194]: Combines DP (ZeRO-2), tensor, and pipeline parallelism, leveraging a planner to select an optimized training configuration.
- Cephalo [57]: Distributes compute workload and training state data unevenly with FSDP to utilize heterogeneous resources efficiently.

6.5.1 Training Throughput

Cluster	Model	Zorse		TorchTitan-Het		HexiScale		Cephalo	
		TFlops	**HFU**	**TFlops**	**HFU**	**TFlops**	**HFU**	**TFlops**	**HFU**
A	Llama 7B	**4370.56**	**52.46%**	4223.80	50.69%	3193.46	38.33%	1714.52	20.58%
	Llama 13B	**4917.87**	**59.02%**	3837.49	46.06%	3270.32	39.25%	1656.29	19.88%
	Llama 33B	**5281.64**	**63.39%**	944.47	11.34%	3064.22	36.78%	1943.89	23.33%
	Llama 65B	**5239.13**	**62.88%**	*OOM*	*OOM*	2048.63	24.59%	1937.64	23.26%
B	Llama 7B	**3412.88**	**43.49%**	2033.53	25.91%	1194.89	15.23%	2274.50	28.98%
	Llama 13B	**2965.64**	**37.79%**	1956.09	24.93%	1152.73	14.69%	1992.24	25.39%
	Llama 33B	**2658.29**	**33.87%**	*OOM*	*OOM*	657.16	8.37%	1373.31	17.50%
C	Llama 7B	**3936.94**	**39.24%**	2441.70	24.34%	2624.63	26.16%	1213.39	12.10%
	Llama 13B	**3357.97**	**33.47%**	2061.55	20.55%	1952.31	19.46%	1222.96	12.19%
	Llama 33B	**1548.60**	**15.44%**	*OOM*	*OOM*	*OOM*	*OOM*	775.42	7.73%

Table 6.6: Throughput (TFlops) and GPU utilization (HFU) of Zorse compared to other systems across different model sizes and clusters (higher is better). *OOM* denotes Out-of-Memory.

This chapter compares Zorse to existing systems across three representative heterogeneous training scenarios. A summary of the results is provided in Table 6.6. More details on the training configurations used by Zorse are provided in Appendix D.

A: Small-size cluster of high-end GPUs. This cluster consists of 4 H100s and 16 A100s, representative of scenarios where users have limited high-end GPUs but can combine them to form a larger cluster. GPUs within each node are connected via NVSwitch, while inter-node bandwidth is significantly lower at 50 Gbps. Zorse achieves superior performance over all baselines, with its relative speedup increasing with model size, reaching up to 3× on the Llama 65B model. Additional experiments for GPT models are provided in Figure 6.9.

TorchTitan-Het partitions the model into 5 stages, each with 4 GPUs, one grouping the 4 H100s together and the rest grouping the A100s. It employs a 2×2 ZeRO-2 DP×TP configuration, which works well for smaller models but runs out of memory on larger ones. Gradient accumulation is used to manage memory constraints, but this adds communication overhead for ZeRO-2 from extra parameter gathering.

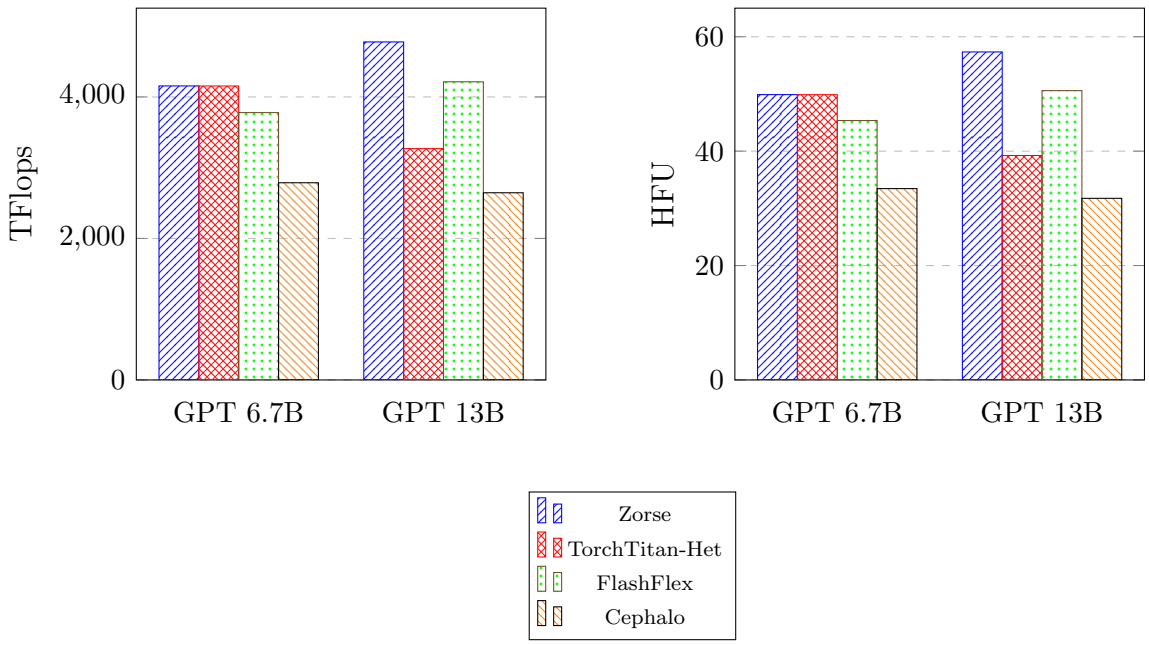


Figure 6.9: Training TFlops and HFU of Zorse compared to baselines for GPT models on Cluster A.

HexiScale employs PP + ZeRO-2 DP, adjusting PP and DP degrees to manage memory usage. However, larger models still face high memory pressure, leading to suboptimal partitioning and underutilization of H100 GPUs. Despite having nearly 3× the TFlops of A100s, H100s have only 15% more memory, and are unable to train with 3× the number of layers without running out of memory.

Cephalo uses ZeRO-3 across all GPUs, but the significant disparity in intra-node and inter-node bandwidths (over 35-fold) causes AllGather and ReduceScatter communications to bottleneck training.

In contrast, Zorse maintains high efficiency even with larger models by grouping GPUs within each node for DP and applying PP across nodes, possible with Zorse's support for heterogeneous PP. Its memory optimizations allow it to train larger models while evenly balancing computation across GPUs, unlike other systems that must compromise performance to avoid running out of memory.

B: Medium-size cluster of low, middle, & high end GPUs. This cluster comprises 8 A100s, 16 A10Gs, 16 V100s, and 24 T4s, representing scenarios where users have limited access to GPUs with diverse performance capabilities but aim to utilize them concurrently

for training. The cluster presents challenges due to its heterogeneity in computational power, memory, and networking. For instance, V100s and A100s benefit from faster intra-node NVLink interconnects, whereas T4s and A10Gs rely on PCIe; T4s and V100s share the same memory capacity, yet V100s are twice as fast; A10Gs and V100s have similar computational speeds, but A10Gs offer 1.5 times more memory.

Zorse partitions the cluster into four groups, each consisting of VMs with the same GPUs and hardware. This setup facilitates ZeRO-2 DP within groups sharing the same networking hardware, ensuring slower interconnects do not impede faster ones. For the larger 13B and 33B models, Zorse employs 4 and 6 ministages per GPU, respectively, increasing parameter offloading and reducing memory usage. This strategy is crucial for performance, as it frees memory, allowing for more flexible layer distribution across GPUs. Zorse consistently achieves a $1.5\times - 4\times$ speedup in training throughput across all models and baselines.

TorchTitan-Het and HexiScale lack efficient activation offloading mechanisms, relying instead on partitioning models into many stages with PP to manage memory usage. However, they are unable to balance computation well across PP stages due to the limited number of model layers. For example, HexiScale divides the Llama 33B model into over 16 stages, but with only 40 layers available, the partitioning is too coarse to balance computation effectively. The A100, with $5\times$ the TFlops of the T4, would need $5\times$ more layers to achieve balance. Unlike Zorse, TorchTitan-Het cannot flexibly configure PP and DP, and runs out of memory when training Llama 33B.

Cephalo efficiently manages memory by fully sharding parameters across the cluster, balancing computation across GPUs. However, it faces bottlenecks due to collective communication in the highly heterogeneous network.

C: Large-size cluster of low- & middle-end GPUs. This cluster comprises 128 GPUs across two AWS regions: 16 A10Gs and 48 T4s in one, and 16 V100s and 48 T4s in the other. It represents scenarios where users lack high-end GPUs but have access to numerous low- and mid-tier GPUs. Training is challenging due to limited GPU memory ($\leq$ 24 GB) and high communication latencies from the large GPU count and slower cross-region links. Consequently, training throughput and utilization are generally lower. Nevertheless, Zorse consistently achieves at least a $1.5\times$ speedup over baselines.

To prevent OOM errors on larger models, Zorse employs varied DP sizes tailored to each VM configuration. For instance, all A10Gs form a single DP group, while V100s are divided into two groups of 8 GPUs each, as they need to handle a similar layer count as A10Gs but with 50% less memory. This smaller DP grouping reduces memory demands for model parameter replication. T4s, assigned fewer layers due to their slower speed, are split

into two groups of 24 within each region to avoid cross-region DP.

TorchTitan-Het, unable to asymmetrically partition GPUs, required a higher PP degree to manage memory, leading to high pipelining overhead and poor performance. HexiScale's use of TP to cut memory usage resulted in high communication overhead and also led to poor performance. Cephalo's uneven training state data partitioning allowed V100s to store less state, but the cross-region network's uneven partitioning overhead caused communication bottlenecks.

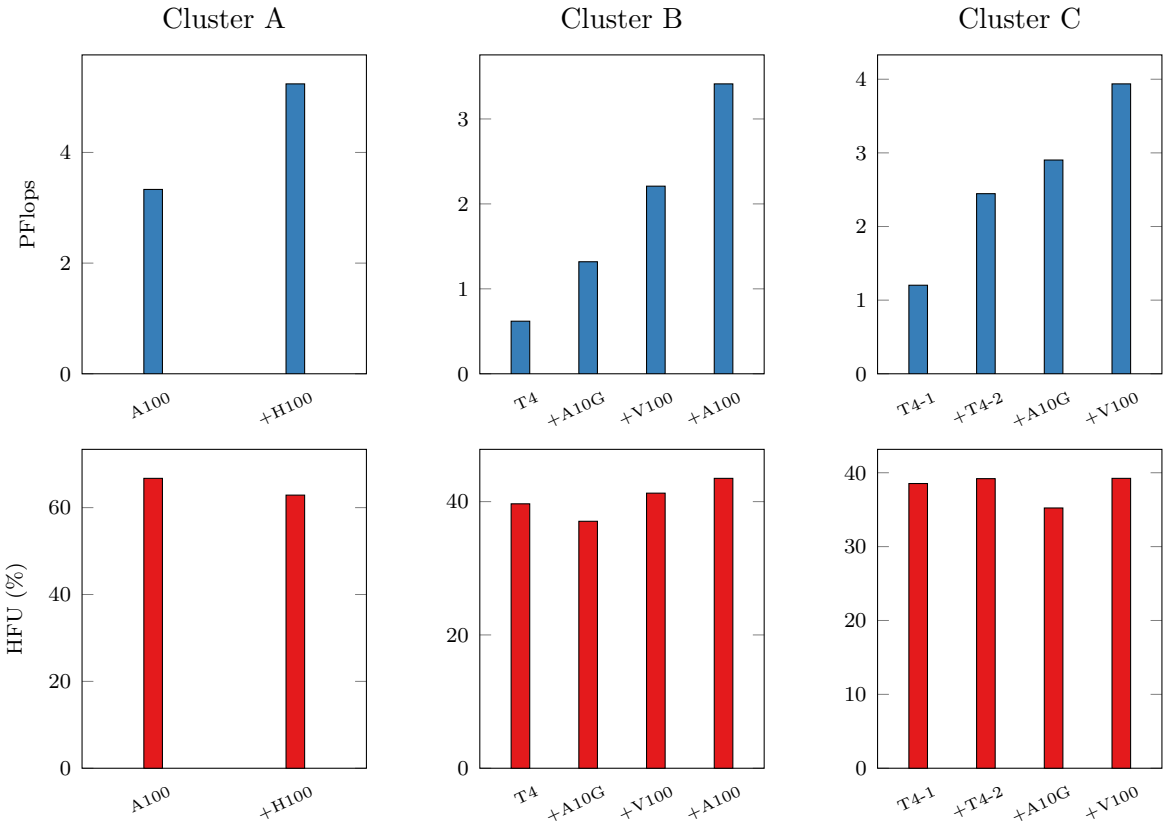


Figure 6.10: PFlops (1k TFlops) and HFU scaling as heterogeneous GPUs are added to the clusters.

6.5.2 Cluster Scaling

Zorse's scalability is validated with increasingly heterogeneous and larger clusters. For each cluster, the training performance of the largest model that can be trained on the slowest

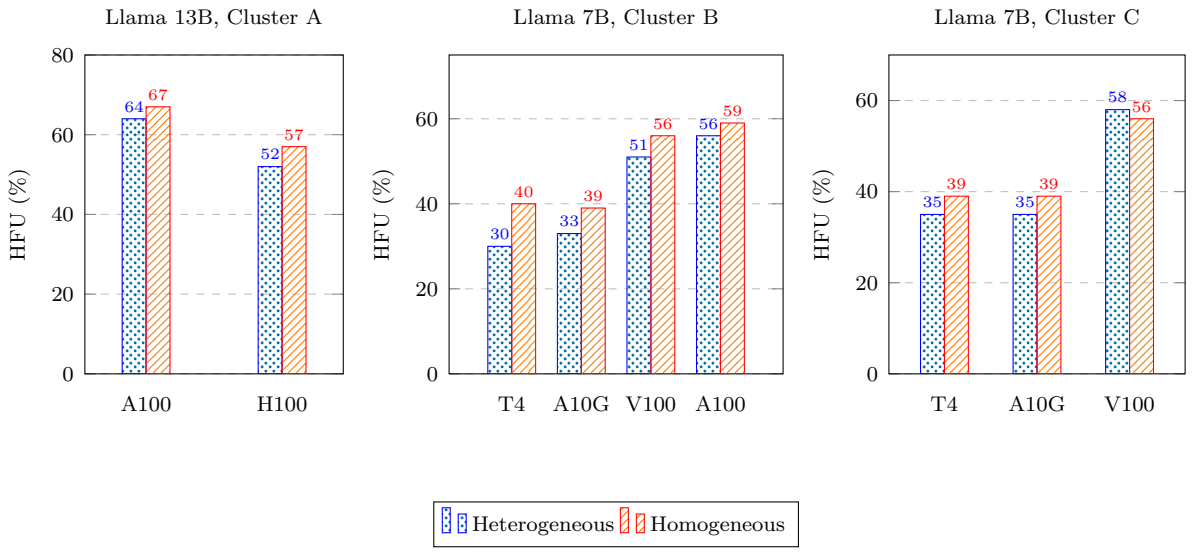


Figure 6.11: GPU Utilization on heterogeneous clusters vs homogeneous subgroups of GPUs within the clusters.

GPUs is first evaluated. Faster GPUs are then incrementally added to the training group, gradually increasing the total number of GPUs and heterogeneity until the entire cluster is utilized. Results are presented in Figure 6.10.

Adding heterogeneous GPUs into the training group across various clusters significantly improves training throughput. Furthermore, cluster utilization (HFU) generally remains stable or improves, as observed in Cluster B. Improvements can be attributed to reduced memory requirements per GPU with the addition of more GPUs, enabling more efficient training configurations that better balance computation. These results demonstrate that when homogeneous GPUs are limited, they can be pooled together to achieve higher throughput without sacrificing training efficiency with Zorse.

6.5.3 Comparison to Homogeneous Training

The training efficiency of Zorse on heterogeneous versus homogeneous clusters is assessed in Figure 6.11. For each heterogeneous cluster, the HFU achieved per GPU type when training on the entire cluster is compared against training on only the homogeneous subset of those GPUs. This evaluation uses the largest model that can be trained on the clusters without running out of memory. Homogeneous training is typically expected be more efficient due to its smaller scale and uniform hardware, which reduces communication

overhead and eliminates the need for balancing computation. Nevertheless, the results demonstrate that Zorse consistently achieves efficiency levels comparable to homogeneous training, reflecting its ability to balance network, compute, and memory heterogeneity. This highlights Zorse's capability to effectively scale training across heterogeneous clusters, achieving higher throughput than possible with homogeneous clusters, while maintaining efficiency.

6.5.4 Pipeline-Efficient ZeRO DP Analysis

Section 6.1.3 demonstrated how PP + ZeRO-3 is memory efficient but can lead to very low throughput due to high communication overhead, whereas PP + ZeRO-2 is memory inefficient but can achieve high throughput due to low communication overhead. This chapter analyzes how Pipeline-Efficient ZeRO DP in Zorse is able to achieve a good balance between low memory and high throughput, and shows the tradeoff as the interleaving factor scales, corresponding to the number of ministages assigned to each GPU.

Figure 6.12 evaluates the performance of Pipeline-Efficient ZeRO DP using two homogeneous clusters of 16 A100s, and 16 A10Gs. A homogeneous cluster is used to isolate the impact of interleaving from side effects that may result from compute imbalance in heterogeneous clusters. There is a slight drop in throughput when going from 1 ministage (no interleaving) to 2 ministages per GPU, due to increases in pipelining overhead. However, as the number of ministages increases past 2, the additional drop in throughput is minimal, while memory utilization continues to decrease. In both training setups, with maximum interleaving, memory utilization can be reduced by 40% while incurring only a 20% drop in throughput. When memory is limited in heterogeneous clusters, this is an effective tradeoff since it allows for compute to be balanced more evenly across GPUs, which can often improve throughput and make up for the additional pipelining overhead.

The performance of PP with ZeRO-2 and ZeRO-3 is also plotted for comparison. Pipeline-Efficient ZeRO DP achieves much higher throughput than PP + ZeRO-3 with comparable memory utilization, particularly on the A10Gs (due to slower links). When interleaving is maximized, Pipeline-Efficient ZeRO DP actually uses less memory since it also offloads sharded parameters. Offloading overhead is measured to be minimal, contributing at most a 3% performance impact across all workloads.

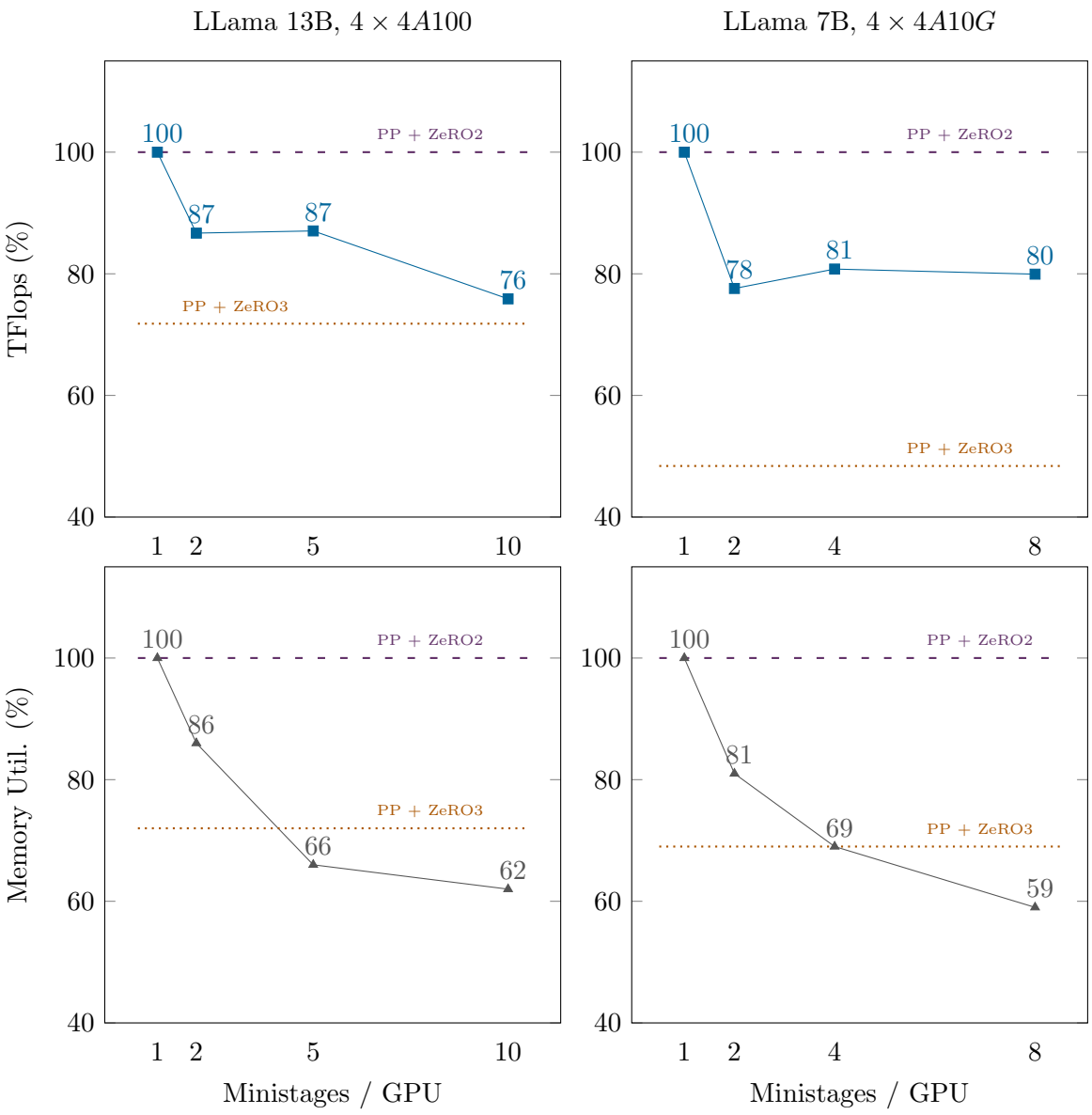


Figure 6.12: TFlops and memory utilization for varying ministages per GPU. Values are normalized to 1 microstage.

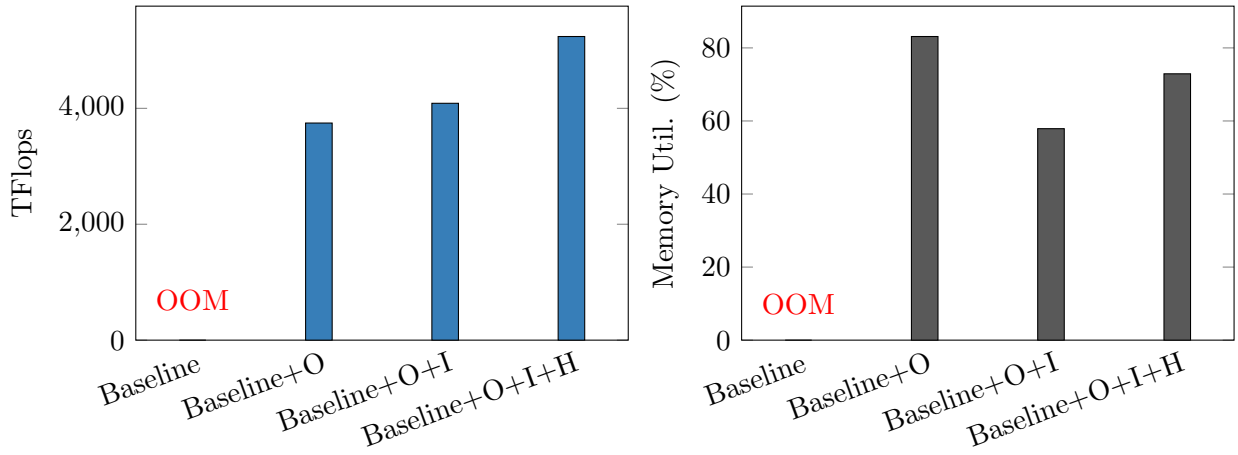


Figure 6.13: TFlops and memory utilization of Zorse optimizations (Cluster A, LLama 65B). Baseline runs out of memory.

6.5.5 Ablation Study

An ablation study is conducted to evaluate the impact of Zorse's key components on training throughput and memory utilization. Starting with a baseline PP + ZeRO-2 implementation in TorchTitan, the following are incrementally added: (1) activation offloading (O), (2) Pipeline-Efficient ZeRO DP and interleaved optimizer updates (I), and (3) Heterogeneous Pipeline Parallelism (H). These components are evaluated using Llama 65B on Cluster A, with results shown in Figure 6.13.

The PP + ZeRO-2 baseline fails to train the model due to excessive activation memory overhead. Adding activation offloading (O) enables training with a PP degree of 5 and ZeRO-2 degree of 4, but H100 GPUs remain underutilized despite being 3× faster than A100s, as they lack sufficient memory to store proportionally more layers. Pipeline-Efficient ZeRO DP and interleaved optimizer updates (I) further reduces memory utilization by 26% due to parameter offloading and discarding gradients as soon as possible. Finally, with Heterogeneous Pipeline Parallelism (H), the pipeline stages are reduced to 4 by grouping GPUs within each node. This configuration achieves balanced computation across GPUs, trading slightly higher memory utilization for significant performance improvements.

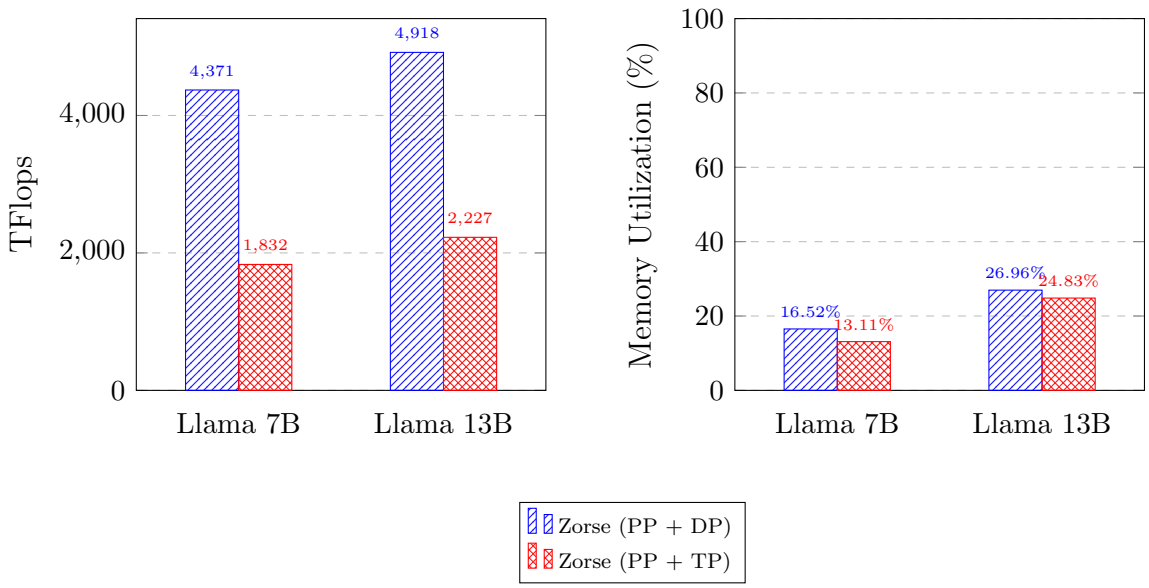


Figure 6.14: Training TFlops and Memory Utilization of Zorse with PP + TP vs Zorse with PP + DP on Cluster A with 4xH100s and 16xA100s.

6.5.6 Pipeline-Efficient ZeRO DP vs TP

As discussed in Section 6.1.1, the targeted heterogeneous workloads of this work do not typically benefit from TP, since the scale of model and cluster that necessitates TP in training is beyond the typical scale of heterogeneous training. Nevertheless, for additional flexibility and to serve as a point of comparison, TP (including Sequence Parallelism for normalization layers) was integrated in Zorse such that it can parallelize training with TP within each pipeline stage instead of DP. Figure 6.14 compares the performance of Zorse with PP + DP vs Zorse with PP + TP on cluster A, which is the most ideal scenario for using TP since there are high bandwidth NVLink interconnects connecting GPUs within each node. The model is split into 4 stages, one per node, and each node applies TP within its stage.

Zorse with TP is not able to achieve as good performance as Zorse with DP for both the Llama 7B and Llama 13B models. This is primarily due to poor communication-computation overlap within TP. Due to the sequential dependencies of computation on gathering and reducing layer inputs, it is not possible to completely overlap computation with communication, and there is a high communication overhead despite individual communication operations being fast under NVLink. In terms of memory utilization, TP

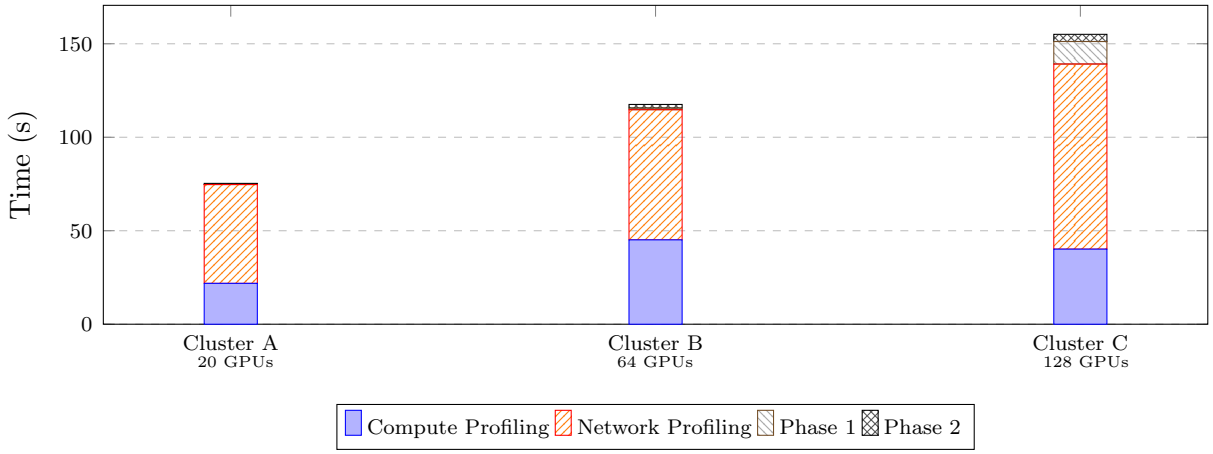


Figure 6.15: Planner runtime breakdown per cluster.

uses a comparable amount of memory as DP in Zorse. This is because both strategies shard both model activations and parameters. TP shards activations by the sequence and hidden dimension, while DP shards by the batch dimension. The amount of memory Zorse with DP uses can be adjusted by configuring the size of ministages assigned to each GPU. With larger ministages, Zorse with DP can require slightly more memory than with TP since ministages are unsharded during computation, whereas layers always remain sharded during TP. However, when Zorse is configured with a large number of small ministages, GPU utilization tied to model parameters can actually be lower than TP since ministage parameters are offloaded to the CPU when not needed. In fact, memory requirements for Zorse with TP are too high to train the Llama 65B model.

6.5.7 Planner Optimization Time & Accuracy

The search space for optimal training configurations in Zorse is extensive, encompassing asymmetric combinations of PP and DP, model partitioning, interleaved pipelining, and microbatch sizes. Despite this complexity, Zorse's planner completes within 3 minutes for all workloads. This is a negligible overhead to pay for optimizing LLM training, which typically requires hundreds of thousands of GPU hours [177]. Fast planning is achieved through a two-phase optimization approach that employs approximation algorithms and heuristics to effectively prune the search space. Figure 6.15 presents a breakdown of optimization time for the largest models in each cluster. While profiling dominates planning time, it scales sublinearly with the number of GPUs, since optimizations are made to parallelize profiling

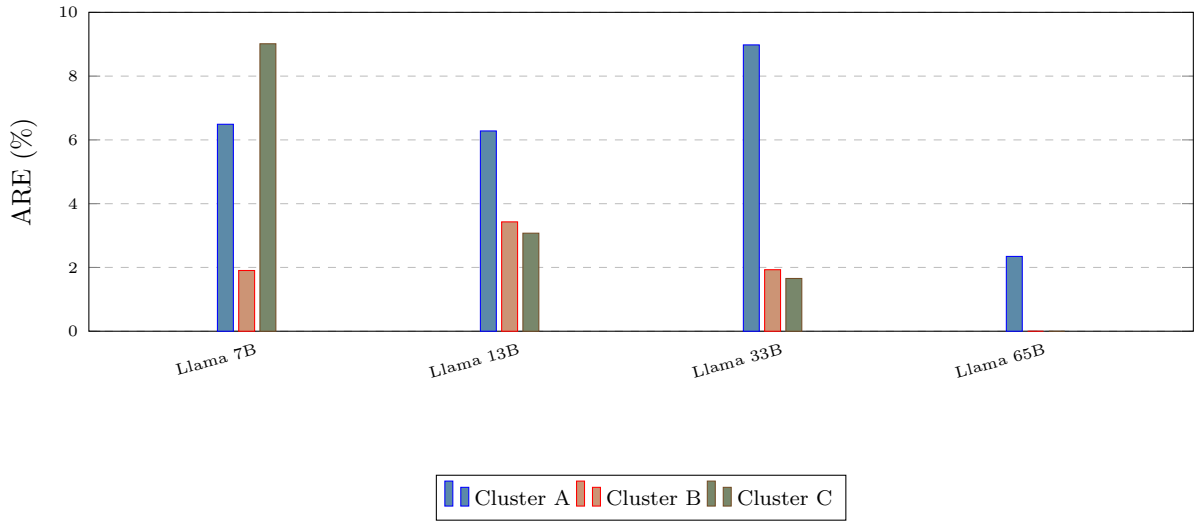


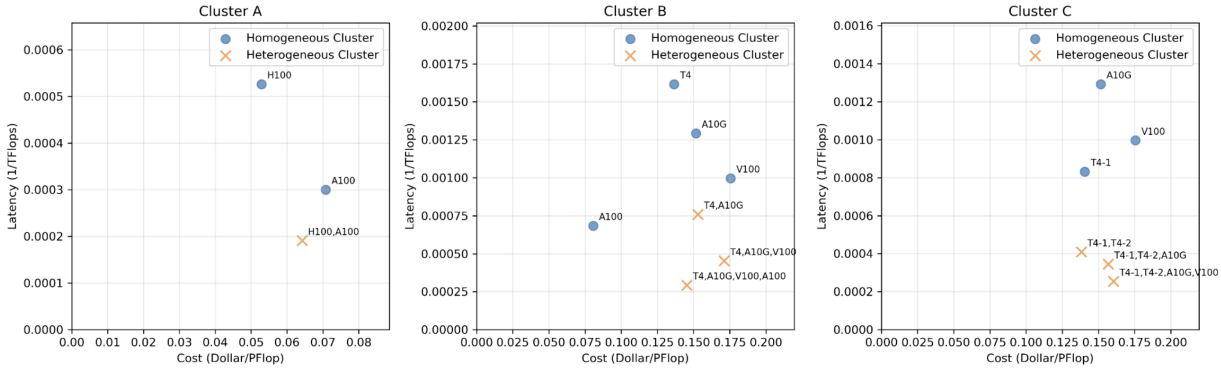
Figure 6.16: Planner latency model absolute relative error.

and avoid redundant measurements on duplicate VMs and GPUs.

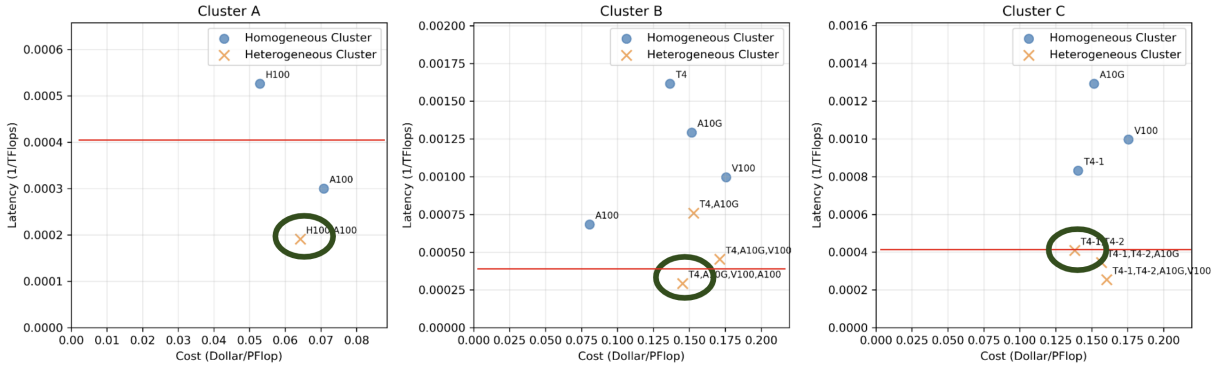
Zorse's planner relies on a latency model to estimate execution time for different training configurations, enabling efficient comparison within the vast search space during optimization. Figure 6.16 presents the absolute relative error (ARE) between model predictions and measured latencies across all clusters and workloads. The planner maintains high accuracy across all evaluated workloads, with errors consistently below 10% and an average absolute relative error of 4.5%. Importantly, prediction accuracy remains stable as model size and batch size increase, demonstrating the planner's robustness across diverse training scenarios.

6.5.8 Resource Selection

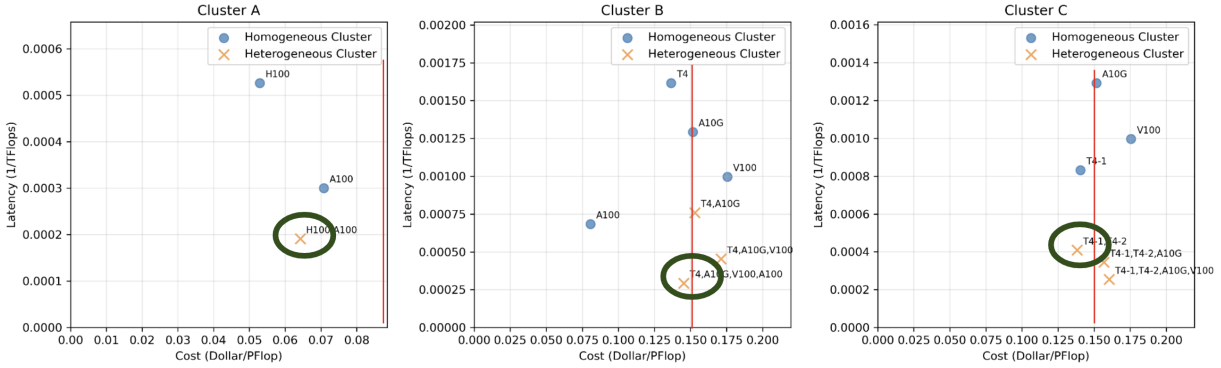
The planner initially assumes a fixed cluster and determines the best training configuration for it. However, in practice, users are not restricted to a fixed cluster and can modify the cluster composition based on changing workload demands and resource availability. Since Zorse's planner runs quickly, it can be used to optimize both the cluster configuration and the training configuration simultaneously. As shown in Figure 6.15, the planning time is primarily consumed by compute and network profiling, while the actual optimization (Phases 1 and 2) can be completed in seconds, even for the largest clusters. This is advantageous because profiling the compute and network is a fixed cost. Given the limited number of



(a) Latency versus cost tradeoff for different cluster configurations.



(b) Optimizing cost under latency constraint Y = 0.0004. Optimal cluster circled in green.



(c) Optimizing latency under cost constraint X = 0.15. Optimal cluster circled in green.

Figure 6.17: Performance comparison of different homogeneous and heterogeneous subsets of clusters A-C.

GPUs and distinct hardware configurations, profiling can be done for all configurations, and only the optimization (Phases 1 and 2) needs to be rerun for new cluster configurations. This allows Zorse to optimize both the cluster and training configuration without incurring significant optimization time.

Figure 6.17a illustrates the latency (1/TFlops) versus cost (in dollars per PFlop) for the training configurations selected by Zorse across different homogeneous and heterogeneous subsets of the three clusters. For instance, in Cluster A, H100 refers to using all 4xH100 GPUs for training, A100 refers to using all 16xA100 GPUs, and H100+A100 refers to using a combination of 4xH100 GPUs and 16xA100 GPUs. Generally, more expensive configurations achieve lower latency. Heterogeneous clusters that utilize all available GPUs will achieve lower latency with higher aggregate compute but may be slightly more expensive than training with only homogeneous GPUs due to a small drop in efficiency as heterogeneity increases.

The optimal cluster choice depends on the user's constraints. For example, in Figure 6.17b, the user has a latency constraint of Y = 0.0004 (red line). Any configuration below this line meets the constraint, and the best cluster is the one that minimizes cost, represented by the leftmost point circled in green. In Cluster B, the only viable solution is the heterogeneous configuration with all GPUs. In Cluster C, the best solution uses only the T4 GPUs, as it is the cheapest configuration that satisfies the latency constraint, even though it is not the fastest overall configuration.

If the user instead has a cost constraint of X = 0.15 (red line) as shown in Figure 6.17c, the best solution minimizes latency while satisfying the cost constraint (to the left of the red line). In Cluster A, all configurations meet the cost constraint, and the optimal configuration is the heterogeneous cluster that uses all GPUs and achieves the lowest latency. In Cluster B, the heterogeneous cluster with all GPUs also achieves the lowest latency among the solutions that satisfy the cost constraint.

6.6 Discussion

Cephalo vs Zorse. By efficiently integrating PP with ZeRO-style DP, Zorse is able to train LLMs effectively on clusters with large heterogeneity, such as those that feature NVLink, which can be an order of magnitude faster than communication without NVLink. However, in clusters with limited network heterogeneity, training with Cephalo can be more advantageous. This is because Cephalo needs to materialize only a single layer of the model at a time, whereas Zorse needs to materialize entire stages of the model at a time.

While a stage in Zorse can be configured to consist of a single layer, doing so may lead to uneven compute balance and result in computational bottlenecks. As shown in Figure 6.12, ZeRO-3–style DP, which is employed in Cephalo, generally uses less memory than most configurations employed by Zorse.

Another advantage of Cephalo is that it can more granularly partition training state across GPUs in the cluster. In Zorse, each GPU must store an equal portion of the training state for the parameters it is assigned to process. In contrast, in Cephalo, training state storage is decoupled from compute assignment, such that a GPU with low memory capacity can be assigned no share of the training state, while a GPU with high memory capacity can store the entire training state. Thus, Cephalo enables better utilization of memory within the cluster at the cost of increased communication overhead, and is preferable in heterogeneous clusters where communication bandwidth is not a bottleneck but memory is limited.

Benefits in Homogeneous Clusters. While Zorse is designed for heterogeneous clusters, several of its techniques can also be useful in homogeneous clusters. Section 6.5.4 evaluates Pipeline-Efficient ZeRO-DP in a homogeneous setting, showing substantial training speedups over PP + ZeRO-3 and significantly lower memory usage than PP + ZeRO-2.

Interleaved optimizer updates and activation offloading are low-overhead memory optimizations that provide similar benefits in homogeneous clusters. Finally, Zorse's planner remains effective in homogeneous environments, since the homogeneous clusters are a strict subset of the heterogeneous search space it is designed to handle. Moreover, even homogeneous GPU clusters exhibit communication heterogeneity (e.g., intra- vs. inter-node links), making the planner's optimizations directly applicable.

Dynamic Parallelism. Hardware performance variability and node failures during training [112, 71, 148] may require dynamic adjustment of the training configuration. However, this is a problem for distributed training in general, and not unique to heterogeneous training. Hence, handling these edge cases is orthogonal to the main focus of this work. Zorse's low-overhead planner runs within 2.5 minutes even for our largest workloads, which is comparable to the reconfiguration time for state-of-the-art systems designed for dynamic parallelism [181, 96]. This allows Zorse to be efficiently integrated with existing approaches for efficient parallelism adaptation [181, 51, 123, 96, 69] and node failure recovery [40, 49, 11] to address these challenges.

Hardware Generalizability. Like previous studies [57, 85, 210, 194, 179], Zorse uses NVIDIA GPUs for evaluation due to their widespread availability and usage in training [118]. Nevertheless, Zorse's implementation is adaptable to any accelerator with PyTorch support, including AMD GPUs [9] and TPUs [82, 68]. Furthermore, its design principles

are hardware-agnostic and applicable to other heterogeneous environments.

6.7 Conclusion

This paper presents Zorse, a system that allows users to train large language models efficiently on clusters with significant compute, memory, and networking heterogeneity. Zorse achieves this by optimizing the combination of PP with DP such that it is both memory efficient and incurs low communication overhead while balancing heterogeneous hardware utilization with flexible PP configuration. Moreover, it uses a planner to efficiently find an optimized training strategy from the vast search space of possible training configurations for heterogeneous clusters. With this design, Zorse is able to resolve data management challenges in heterogeneous training and achieve up to 3× higher training throughput compared to existing state-of-the-art systems across three diverse and representative heterogeneous training scenarios.

Chapter 7

Conclusion

This thesis addressed three important challenges in modern distributed deep neural network training that arise from increasing computational and memory requirements for training modern DNNs, as well as the elasticity, limited availability, and heterogeneity of GPU resources on the cloud. Addressing these training challenges is crucial as DNNs continue to grow exponentially in size and complexity.

7.1 Summary of Research Challenges

Challenge 1: Serverless Elasticity. While serverless computing offers compelling advantages in terms of elasticity and cost efficiency, existing serverless approaches to DNN training suffer from severe performance limitations. Traditional serverless functions are restricted to CPU-only workloads and rely on inefficient indirect communication through external storage, making them orders of magnitude slower than GPU-based training.

Challenge 2: Resource Imbalance in Heterogeneous Clusters. Cloud GPU availability constraints often force organizations to assemble heterogeneous clusters combining different GPU generations and types. However, existing training systems either assume homogeneity or distribute workload based solely on compute capacity, leading to significant underutilization of both compute and memory resources due to the mismatch between these capabilities across different hardware.

Challenge 3: Communication Inefficiency in Network-Heterogeneous Environments. Large-scale cloud training increasingly involves GPUs connected through vastly different networking infrastructure, with bandwidth variations spanning 35-fold differences.

Traditional parallelization strategies struggle in such environments, while existing hybrid approaches force trade-offs between memory efficiency and communication efficiency.

7.2 Research Contributions and Achievements

This thesis presented three novel systems that collectively advance the state-of-the-art in cloud-based distributed training, each addressing a specific challenge while contributing to a comprehensive solution for efficient training in elastic and heterogeneous environments.

7.2.1 Hydrozoa: Serverless Container-Based Training

Hydrozoa represents the first system to successfully combine serverless containers, GPU acceleration, and hybrid-parallel training. By leveraging serverless containers instead of traditional serverless functions, Hydrozoa overcomes the fundamental limitations that previously made serverless training impractical for GPU-intensive workloads.

The system's key technical innovations include a novel serverless container architecture that enables direct worker-to-worker communication while maintaining serverless benefits, a hybrid-parallel planner that automatically optimizes the combination of data and model parallelism, and dynamic scaling policies that maintain convergence properties during elastic resource changes. These contributions resulted in remarkable performance improvements: up to 155.5× higher throughput-per-dollar compared to existing serverless systems and 5.4× higher than VM-based training, while completely eliminating cluster management overhead.

7.2.2 Cephalo: Decoupled Compute-Memory Optimization

Cephalo is a system that targets decoupling compute distribution from memory allocation to achieve better resource utilization in heterogeneous clusters. This decoupling enables independent optimization of computational workload and memory utilization, addressing the core inefficiencies that plague traditional approaches to heterogeneous training.

The system's novel contributions include flexible training state partitioning that extends FSDP to support uneven parameter sharding independent of compute distribution, layered gradient accumulation that reduces communication overhead while managing memory efficiently, and a sophisticated planning algorithm that simultaneously optimizes multiple

resource dimensions. These technical contributions delivered substantial performance gains of 1.2×–10.8× higher training throughput compared to state-of-the-art heterogeneous systems while supporting larger models and batch sizes.

7.2.3 Zorse: Unified Memory and Communication Efficiency

Zorse addressed the trade-off between memory efficiency and communication efficiency in distributed training through the introduction of Pipeline-Efficient ZeRO DP. This novel integration combines the memory benefits of ZeRO-3 data parallelism with the communication advantages of pipeline parallelism, representing the first system to achieve both objectives simultaneously.

The system's key innovations include interleaved pipeline stage computation with intelligent parameter offloading, support for asymmetric pipeline configurations that accommodate varying GPU capabilities, and a two-phase optimization algorithm that efficiently navigates the complex configuration space. These contributions resulted in up to 3× higher training throughput across diverse heterogeneous configurations while scaling to models with 65 billion parameters.

7.3 Future Research Directions

The work presented in this thesis opens several promising avenues for future research in cloud-based distributed training.

7.3.1 Additional Parallelization Strategies

While this thesis focused on data, pipeline, and tensor parallelism, future work could extend these frameworks to incorporate orthogonal parallelization techniques. Namely, context parallelism, which distributes long sequences across multiple devices, and expert parallelism for Mixture-of-Experts (MoE) architectures are strategies targeting emerging model architectures and workloads. Integrating these parallelization strategies would further extend the applicability of these systems to a wider range of models and workloads.

7.3.2 Multi-Vendor Hardware Heterogeneity

The evaluations in this thesis primarily focused on NVIDIA GPUs, which remain the dominant choice for cloud-based training. However, the landscape is rapidly evolving with AMD GPUs, Google TPUs, AWS Trainium, and other specialized accelerators gaining adoption. Future research could extend these heterogeneity-aware approaches to consider cross-vendor hardware differences, including varying memory architectures, compute paradigms, and interconnect technologies. This would enable training systems to leverage the full spectrum of available cloud accelerators while optimizing for cost and performance across diverse hardware ecosystems.

7.3.3 Adjacent Applications

The elastic and heterogeneous resource management techniques developed in this thesis could be adapted to adjacent computationally intensive DNN workloads. For example, these training techniques could be integrated into cluster schedulers designed for heterogeneous clusters. Large-scale inference workloads exhibit dynamic demand patterns across a variety of models and use cases that could benefit from serverless elasticity and heterogeneous resource allocation. Similarly, reinforcement learning applications, with their varying computational requirements across training and sampling phases, present opportunities to apply these dynamic scaling and resource optimization approaches beyond traditional training-only scenarios.

7.4 Concluding Remarks

This thesis demonstrated that while cloud environments make DNN training more accessible for the broader community, they also pose several challenges in resource allocation and training efficiency that need to be addressed to unlock the full potential of cloud computing for DNN training.

Through the development of Hydrozoa, Cephalo, and Zorse, this work established new paradigms for serverless training, heterogeneous resource management, and training efficiency optimization that collectively advance the field toward more accessible, efficient, and cost-effective distributed training.

References

[1] ZeroMQ, 2025. https://zeromq.org/.

[2] Jie Ren 0015, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li 0001, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 551–564. USENIX Association, 2021.

[3] Azure Container Instances, 2025. https://azure.microsoft.com/en-us/services/container-instances/.

[4] Bilge Acun, Benjamin Lee, Fiodar Kazhamiaka, Kiwan Maeng, Udit Gupta, Manoj Chakkaravarthy, David Brooks, and Carole-Jean Wu. Carbon explorer: A holistic framework for designing carbon aware datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 118–132, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Nawras Alnaasan, Bharath Ramesh, Jinghan Yao, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. Hypersack: Distributed hyperparameter optimization for deep learning using resource-aware scheduling on heterogeneous gpu systems. In *2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 100–110, 2024.

[6] Amazon Web Services. Amazon EC2 G5 Instances, 2025. https://aws.amazon.com/ec2/instance-types/g5/.

[7] Amazon Web Services. Amazon EC2 P4 Instances, 2025. https://aws.amazon.com/ec2/instance-types/p4/.

[8] Amazon Web Services, Inc. AWS CloudFormation, 2025. https://aws.amazon.com/cloudformation/.

[9] AMD. *PyTorch on ROCm*, 2025. Accessed: 2025-02-24.

[10] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages 263–274. ACM, 2018.

[11] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 472–487, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient combination of rematerialization and offloading for training dnns. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 23844–23857. Curran Associates, Inc., 2021.

[13] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 04 2020.

[14] Dimitri P. Bertsekas and Robert G. Gallager. *Data Networks*. Prentice-Hall, 1987.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[16] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy H. Katz. Cirrus: a serverless framework for end-to-end ML workflows. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 13–24. ACM, 2019.

[17] Yuji Chai, Devashree Tripathy, Chuteng Zhou, Dibakar Gope, Igor Fedorov, Ramon Matas, David Brooks, Gu-Yeon Wei, and Paul Whatmough. Perfsage: Generalized inference performance predictor for arbitrary deep learning models on edge devices. *arXiv preprint arXiv:2301.10999*, 01 2023.

[18] Shaoyuan Chen, Wencong Xiao, Yutong Lin, Mingxing Zhang, Yingdi Shan, Jinlei Jiang, Kang Chen, and Yongwei Wu. Efficient heterogeneous large language model decoding with model-attention disaggregation. *arXiv preprint arXiv:2405.01814*, 05 2025.

[19] Xi Chen and Xiao Wang. Pali: Scaling language-image learning in 100+ languages. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2022.

[20] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[21] Bin Cheng, Jonathan Fürst, Gurkan Solmaz, and Takuya Sanada. Fog function: Serverless fog computing for data intensive iot services. In *2019 IEEE World Congress on Services (IEEE SERVICES 2019)*, Milan, Italy, July 2019.

[22] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sashank Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24(1), January 2023.

[23] Jae-Won Chung, Yile Gu, Insu Jang, Luoxi Meng, Nikhil Bansal, and Mosharaf Chowdhury. Reducing energy bloat in large model training. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 144–159, New York, NY, USA, 2024. Association for Computing Machinery.

[24] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008*, volume 307 of *ACM International Conference Proceeding Series*, pages 160–167. ACM, 2008.

[25] Kubeflow Contributors. Kubeflow: A machine learning toolkit for kubernetes. https://www.kubeflow.org/, 2025. Accessed: 2025-09-18.

[26] PyTorch Contributors. *torch.distributed.fsdp.FullyShardedDataParallel*. PyTorch, 2025.

[27] Microsoft Corporation. Azure functions: Serverless compute on demand. https://azure.microsoft.com/en-us/products/functions, 2025. Accessed: 2025-09-18.

[28] Microsoft Corporation. Azure machine learning. https://azure.microsoft.com/en-us/services/machine-learning/, 2025. Accessed: 2025-09-18.

[29] Microsoft Corporation. Azure virtual machines. https://azure.microsoft.com/en-us/services/virtual-machines/, 2025. Accessed: 2025-09-18.

[30] NVIDIA Corporation. A100 data center gpu. https://www.nvidia.com/en-us/data-center/a100/, 2025. Accessed: 2025-09-18.

[31] NVIDIA Corporation. H100 tensor core gpu. https://www.nvidia.com/en-us/data-center/h100/, 2025. Accessed: 2025-09-18.

[32] NVIDIA Corporation. Tensor Cores: Versatility for HPC & AI. https://www.nvidia.com/en-us/data-center/tensor-cores/, 2025. Accessed: 2025-09-12.

[33] Pulumi Corporation. Pulumi, 2025. https://github.com/pulumi/pulumi.

[34] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: fast and memory-efficient exact attention with io-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[35] Aditya Devarakonda, Maxim Naumov, and Michael Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *CoRR*, abs/1712.02029, 2017.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[37] Yushi Ding, Noam Botzer, and Tim Weninger. Hetseq: Distributed gpu training on heterogeneous infrastructure. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 15432–15438. AAAI Press, May 2021.

[38] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[39] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models. *arXiv preprint arXiv:2311.08105*, 11 2024.

[40] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: proactive, liveput-optimized dnn training on preemptible instances. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[41] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12(null):2121–2159, July 2011.

[42] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[43] Alexander Erben, Ruben Mayer, and Hans-Arno Jacobsen. How can we train deep learning models across clouds and continents? an experimental study. *Proc. VLDB Endow.*, 17(6):1214–1226, February 2024.

[44] Facebook Incubator. Gloo: Collective communications library with various primitives for multi-machine training. https://github.com/facebookincubator/gloo, 2025. Accessed: 2025-04-17.

[45] L. Feng, P. Kudva, D. Da Silva, and J. Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 334–341, 2018.

[46] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[47] Cloud Native Computing Foundation and Kubernetes Contributors. Kubernetes: Production-grade container orchestration. https://kubernetes.io/, 2025. Accessed: 2025-09-18.

[48] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Sen Yang, and Ji Liu. Bagua: scaling up distributed learning with system relaxations. *Proc. VLDB Endow.*, 15(4):804–813, December 2021.

[49] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[50] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. Runtime performance prediction for deep learning models with graph neural network. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 368–380, 2023.

[51] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 178–194, New York, NY, USA, 2024. Association for Computing Machinery.

[52] Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama. https://github.com/openlm-research/open_llama, 2023.

[53] Oded Goldschmidt and Dorit S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 444–451. IEEE Computer Society, 1988.

[54] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[55] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne,

Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb,

Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang, Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao,

Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 07 2024.

[56] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: a gpu cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 485–500, USA, 2019. USENIX Association.

[57] Runsheng Guo, Utkarsh Anand, Arthur Chen, and Khuzaima Daudjee. Cephalo: Harnessing heterogeneous gpu clusters for training transformer models. In *Proceedings of the 2025 International Conference on Supercomputing (ICS '25)*, Salt Lake City, UT, USA, 2025. Association for Computing Machinery.

[58] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 779–794, 2022.

[59] Runsheng Benson Guo, Utkarsh Anand, Khuzaima Daudjee, and Rathijit Sen. Zorse: Optimizing llm training efficiency on heterogeneous gpu clusters. In *MLSys 2026*, Bellevue, WA, May 2026. Ninth Annual Conference on Machine Learning and Systems, May 18–22, 2026.

[60] Runsheng Benson Guo and Khuzaima Daudjee. Research challenges in deep reinforcement learning-based join query optimization. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, pages 3:1–3:6. ACM, 2020.

[61] HashiCorp. Terraform, 2025. https://developer.hashicorp.com/terraform.

[62] Horace He and Shangdi Yu. Transcending runtime-memory tradeoffs in checkpointing by being fusion aware. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 414–427. Curan, 2023.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[64] Luis Herrera-Quintero, Julian Vega-alfonso, Klaus Banse, and Eduardo Carrillo Zambrano. Smart its sensor for the transportation planning based on iot approaches using serverless and microservices architecture. *IEEE Intelligent Transportation Systems Magazine*, 10:17–27, 07 2018.

[65] Mert Hidayetoğlu, Simon Garcia De Gonzalo, Elliott Slaughter, Pinku Surana, Wen-Mei Hwu, William Gropp, and Alexander Aiken. Hiccl: A hierarchical collective communication library. *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 950–961, 2024.

[66] Geoffrey Hinton, Nîsh Srivastava, and Kevin Swersky. Lecture 6: *Neural Networks for Machine Learning* (mini-batch gradient descent & rmsprop). https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, 2013. Accessed: 2025-09-13.

[67] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[68] Ronghang Hu, Vaibhav Singh, Jack Cao, Milad Mohammadi, Yeounoh Chung, Shauheen Zahirazami, and Ross Girshick. Scaling pytorch models on cloud tpus with fsdp, 2022. Accessed: 2025-02-24.

[69] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[71] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113, 2011.

[72] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, Carlsbad, CA, July 2022. USENIX Association.

[73] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[74] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. *CoRR*, abs/2105.07806, 2021.

[75] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[76] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Guoliang He, Xupeng Miao, Ana Klimovic, Bin Cui, Binhang Yuan, and Eiko Yoneki. Demystifying cost-efficiency in llm serving over heterogeneous gpus. *arXiv preprint arXiv:2502.00722*, 02 2025.

[77] Youhe Jiang, Fangcheng Fu, Xiaozhe Yao, Taiyi Wang, Bin Cui, Ana Klimovic, and Eiko Yoneki. Thunderserve: High-performance and cost-efficient llm serving in cloud environments. *arXiv preprint arXiv:2502.09334*, 02 2025.

[78] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. Megascale: scaling large language model training to more than 10,000 gpus. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI'24, USA, 2024. USENIX Association.

[79] Hongye Jin, Xiaotian Han, Jingfeng Yang, Zhimeng Jiang, Chia-Yuan Chang, and Xia Hu. Growlength: Accelerating llms pretraining by progressively growing training length. *arXiv preprint arXiv:2310.00576*, 10 2023.

[80] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[81] Keller Jordan, Yuchen Jin, Vlado Boza, You Jiacheng, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks. https://kellerjordan.github.io/posts/muon/, 2024.

[82] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[83] Dayal Singh Kalra and Maissam Barkeshli. Why warmup the learning rate? underlying mechanisms and improvements. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc.

[84] Jared Kaplan, Sam McCandlish, T. J. Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeff Wu, and Dario Amodei. Scaling laws for neural language models. *ArXiv*, abs/2001.08361, 2020.

[85] Kyeonglok Kim, Hyeonsu Lee, Seungmin Oh, and Euiseong Seo. Scale-train: A scalable dnn training framework for a heterogeneous gpu cloud. *IEEE Access*, 10:68468–68481, 2022.

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

[87] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[88] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[89] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

[90] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097–1105, Red Hook, NY, USA, 2012. Curran Associates Inc.

[91] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[92] Joel Lamy-Poirier. Layered gradient accumulation and modular pipeline parallelism: fast and efficient training of large language models. *arXiv preprint arXiv:2106.02679*, 06 2021.

[93] Joel Lamy-Poirier. Breadth-first pipeline parallelism. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 48–67. Curan, 2023.

[94] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[95] Conglong Li, Minjia Zhang, and Yuxiong He. The stability-efficiency dilemma: investigating sequence length warmup for training gpt models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[96] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization. *Proceedings of the ACM on Management of Data*, 3(3):1–28, Jun 2025.

[97] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed

machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA, 2014. USENIX Association.

[98] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 766–775, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2391–2404, Toronto, Canada, July 2023. Association for Computational Linguistics.

[100] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[101] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal checkpointing: Efficient and flexible checkpointing for large scale distributed training. *arXiv preprint arXiv:2406.18820*, 06 2024.

[102] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *arXiv preprint arXiv:2410.06511*, 10 2024.

[103] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. 2018.

[104] Yueying Lisa Li, Leo Han, G Edward Suh, Christina Delimitrou, Fiodar Kazhamiaka, Esha Choukse, Rodrigo Fonseca, Liangcheng Yu, Jonathan Mace, and Udit Gupta. Fair, practical, and efficient carbon accounting for llm serving. *SIGMETRICS Perform. Eval. Rev.*, 53(2):99–103, August 2025.

[105] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel dnn training through iterative bottleneck alleviation. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 163–181, New York, NY, USA, 2024. Association for Computing Machinery.

[106] Google LLC. Google ai platform. https://cloud.google.com/ai-platform, 2025. Accessed: 2025-09-18.

[107] Google LLC. Google cloud functions: Event-driven serverless functions. https://cloud.google.com/functions, 2025. Accessed: 2025-09-18.

[108] Google LLC. Google cloud run: Fully managed compute platform to run containers. https://cloud.google.com/run, 2025. Accessed: 2025-09-18.

[109] Google LLC. Google compute engine. https://cloud.google.com/compute, 2025. Accessed: 2025-09-18.

[110] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2017.

[111] Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with warm restarts. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[112] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. Srifty: Swift and thrifty distributed neural network training on the cloud. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 4, pages 833–847, 2022.

[113] Wenjie Luo, Bin Yang, and Raquel Urtasun. Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net. *CoRR*, abs/2012.12395, 2020.

[114] Jialun Lyu, Jaylen Wang, Kali Frost, Chaojie Zhang, Celine Irvene, Esha Choukse, Rodrigo Fonseca, Ricardo Bianchini, Fiodar Kazhamiaka, and Daniel S. Berger. Myths and misconceptions around reducing carbon embedded in cloud platforms. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23, New York, NY, USA, 2023. Association for Computing Machinery.

[115] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: fair and efficient gpu cluster scheduling. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 289–304, USA, 2020. USENIX Association.

[116] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter R. Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 937–954. USENIX Association, 2020.

[117] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[118] MarketsandMarkets. Nvidia's dominance in the ai chip market. https://www.marketsandmarkets.com/blog/SE/nvidia-dominance-in-the-ai-chip-market, 2023.

[119] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *CoRR*, abs/1812.06162, 2018.

[120] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Serving large language models over heterogeneous gpus and network via max-flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 586–602, New York, NY, USA, 2025. Association for Computing Machinery.

[121] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[122] Xupeng Miao, Xiaonan Nie, Yingxia Shao, Zhi Yang, Jiawei Jiang, Lingxiao Ma, and Bin Cui. Heterogeneity-aware distributed machine learning training via partial reduce. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2262–2270. ACM, 2021.

[123] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment*, 16(9):2354–2363, 2023.

[124] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proc. VLDB Endow.*, 16(3):470–479, nov 2022.

[125] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[126] Sergio Moreno-Alvarez, Juan M Haut, Mercedes E Paoletti, Juan A Rico-Gallego, Juan C Diaz-Martin, and Javier Plaza. Training deep neural networks: a static load balancing approach. *The Journal of Supercomputing*, 76:9739–9754, 2020.

[127] MXNet, 2025. https://mxnet.apache.org.

[128] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[129] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, , and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *VLDB DISPA Workshop 2020*.

[130] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[131] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[132] Sophia Nguyen, Beihao Zhou, Yi Ding, and Sihang Liu. Towards sustainable large language model serving. *SIGENERGY Energy Inform. Rev.*, 4(5):134–140, April 2025.

[133] Chengyi Nie, Jessica Maghakian, and Zhenhua Liu. Cannikin: Optimal adaptive distributed dnn training over heterogeneous clusters. In *Proceedings of the 25th International Middleware Conference*, Middleware '24, page 299–312, New York, NY, USA, 2024. Association for Computing Machinery.

[134] Xiaonan Nie, Yi Liu, Fangcheng Fu, Jinbao Xue, Dian Jiao, Xupeng Miao, Yangyu Tao, and Bin Cui. Angel-ptm: A scalable and economical large-scale pre-training system in tencent. *Proc. VLDB Endow.*, 16(12):3781–3794, August 2023.

[135] NVIDIA. TensorRT: High-Performance Deep Learning Inference SDK. https://github.com/NVIDIA/TensorRT, 2025. Accessed: 2025-09-12.

[136] NVIDIA Corporation. Point to point communication functions. NVIDIA NCCL User Guide API documentation, version 2.26.2, 2020. Accessed: 2025-04-17.

[137] NVIDIA Corporation. *NVIDIA Collective Communication Library (NCCL) User Guide: Collective Communication Functions*, 2025. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/api/colls.html.

[138] NVIDIA Corporation. NVIDIA NVLink and NVLink Switch, 2025. https://www.nvidia.com/en-us/data-center/nvlink/.

[139] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[140] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ISCA '24, page 118–132. IEEE Press, 2025.

[141] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 131–141. ACM, 2020.

[142] PyTorch. Training a 1 trillion parameter model with pytorch fully sharded data parallel on aws, 2023. https://shorturl.at/6Y4LT.

[143] PyTorch, 2025. https://pytorch.org.

[144] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[145] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics.

[146] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[147] Redis, 2025. https://redis.io/.

[148] M. Suhail Rehman and Majd F. Sakr. Initial findings for provisioning variation in cloud computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, page 473–479, USA, 2010. IEEE Computer Society.

[149] H. Robbins and S. Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[150] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain [j]. *Psychol. Review*, 65:386 – 408, 11 1958.

[151] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning representations by back-propagating errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[152] Amazon S3, 2025. https://aws.amazon.com/s3.

[153] Amit Sabne. Xla : Compiling machine learning for peak performance. https://research.google/pubs/xla-compiling-machine-learning-for-peak-performance/, 2020.

[154] Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.

[155] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[156] Amazon Web Services. Amazon ec2 (elastic compute cloud). https://aws.amazon.com/ec2/, 2025. Accessed: 2025-09-18.

[157] Amazon Web Services. Amazon sagemaker. https://aws.amazon.com/sagemaker/, 2025. Accessed: 2025-09-18.

[158] Amazon Web Services. Aws fargate: Serverless compute for containers. https://aws.amazon.com/fargate/, 2025. Accessed: 2025-09-18.

[159] Amazon Web Services. Aws lambda: Run code without thinking about servers. https://docs.aws.amazon.com/lambda/, 2025. Accessed: 2025-09-18.

[160] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018.

[161] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31, 2018.

[162] Tianyao Shi, Yanran Wu, Sihang Liu, and Yi Ding. Greenllm: Disaggregating large language model serving on heterogeneous gpus for lower carbon emissions. *arXiv preprint arXiv:2412.20322*, 12 2024.

[163] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[164] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[165] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[166] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 08 2024.

[167] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[168] Dinesh Soni and Neetesh Kumar. Machine learning techniques in emerging cloud computing integrated paradigms: A survey and taxonomy. *Journal of Network and Computer Applications*, 205:103419, 05 2022.

[169] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(12):2438–2452, July 2020.

[170] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

[171] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. ML Training with Cloud GPU Shortages: Is Cross-Region the Answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[172] Foteini Strati, Zhendong Zhang, George Manos, Ixeia Sánchez Périz, Qinghao Hu, Tiancheng Chen, Berk Buzcu, Song Han, Pamela Delgado, and Ana Klimovic. Sailor: Automating distributed training over dynamic, heterogeneous, and geo-distributed clusters. *arXiv preprint arXiv:2504.17096*, 04 2025.

[173] Suhas Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 642–657, 2023.

[174] TensorFlow, 2025. https://www.tensorflow.org.

[175] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513. USENIX Association, 2023.

[176] Yuyang Tian, Desen Sun, Yi Ding, and Sihang Liu. Embadvisor: Adaptive cache management for sustainable llm serving. *arXiv preprint arXiv:2505.23970*, 05 2025.

[177] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 02 2023.

[178] Sahil Tyagi and Prateek Sharma. Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching . In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 188–194, Los Alamitos, CA, USA, August 2020. IEEE Computer Society.

[179] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578, 2024.

[180] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[181] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic parallelism for deep learning using parallelizable tensor collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 195–210, New York, NY, USA, 2024. Association for Computing Machinery.

[182] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. Rat - resilient allreduce tree for distributed machine learning. In *Proceedings of the 4th Asia-Pacific Workshop on Networking*, APNet '20, page 52–57, New York, NY, USA, 2020. Association for Computing Machinery.

[183] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Xiaoxia Wu, Connor Holmes, Zhewei Yao, Samyam Rajbhandari, Olatunji Ruwase, Feng Yang, Lei Yang, and Yuxiong He. Zero++: Extremely efficient collective communication for large model training. In *ICLR 2024*, March 2024.

[184] Guanhua Wang, Chengming Zhang, Zheyu Shen, Ang Li, and Olatunji Ruwase. Domino: Eliminating communication in llm training via generic tensor slicing and overlapping. Technical Report MSR-TR-2024-40, Microsoft, September 2024.

[185] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *2019 IEEE Conference on Computer Communications, INFOCOM 2019, Paris, France, April 29 - May 2, 2019*, pages 1288–1296. IEEE, 2019.

[186] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. Database meets deep learning: Challenges and opportunities. *SIGMOD Rec.*, 45(2):17–22, September 2016.

[187] Yifu Wang, Horace He, Less Wright, Luca Wehrstedt, Tianyu Liu, and Wanchao Liang. Distributed w/ torchtitan: Introducing async tensor parallelism in pytorch, 2024. Accessed: 2025-02-24.

[188] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[189] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[190] Alex Woodie. How aws plans to cope with genaai's insatiable desire for compute. *Datanami*, Dec 2023. https://shorturl.at/Gx69T.

[191] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[192] Lang Xu, Quentin Anthony, Jacob Hatef, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. Scaling large language model training on frontier with low-bandwidth partitioning. In *Proceedings of the 2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 57–67, Bengaluru, India, dec 2024.

[193] Leyi Yan, Linda Wang, Sihang Liu, and Yi Ding. Ensembleci: Ensemble learning for carbon intensity forecasting. In *Proceedings of the 16th ACM International Conference on Future and Sustainable Energy Systems*, E-Energy '25, page 208–212, New York, NY, USA, 2025. Association for Computing Machinery.

[194] Ran Yan, Youhe Jiang, Xiaonan Nie, Fangcheng Fu, Bin Cui, and Binhang Yuan. Hexiscale: Accommodating large language model training over heterogeneous environment. In *MLSys 2026*, Bellevue, WA, May 2026. Ninth Annual Conference on Machine Learning and Systems, May 18–22, 2026.

[195] Bin Yang, Runsheng Guo, Ming Liang, Sergio Casas, and Raquel Urtasun. Radarnet: Exploiting radar for robust perception of dynamic objects. In *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XVIII*, volume 12363 of *Lecture Notes in Computer Science*, pages 496–512. Springer, 2020.

[196] Sheng Yang, Samir Khuller, Sunav Choudhary, Subrata Mitra, and Kanak Mahadik. Scheduling ml training on unreliable spot instances. In *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, UCC '21, New York, NY, USA, 2022. Association for Computing Machinery.

[197] Jie You, Jae-Won Chung, and Mosharaf Chowdhury. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 119–139, Boston, MA, April 2023. USENIX Association.

[198] Yang You, Jing Li, Sashank J. Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv: Learning*, 2019.

[199] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. Habitat: A Runtime-Based computational performance predictor for deep neural network training.

In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 503–521. USENIX Association, July 2021.

[200] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[201] Seyed Majid Zahedi. *Managing Shared Resources in the Datacenter Era*. PhD thesis, Duke University, Durham, NC, USA, 2018.

[202] Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling vision transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12104–12113, 2022.

[203] Chris Zhang, Runsheng Guo, Wenyuan Zeng, Yuwen Xiong, Binbin Dai, Rui Hu, Mengye Ren, and Raquel Urtasun. Rethinking closed-loop training for autonomous driving. In *Computer Vision - ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23-27, 2022, Proceedings, Part XXXIX*, pages 264–282, Berlin, Heidelberg, 2022. Springer-Verlag.

[204] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[205] J. Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv: Optimization and Control*, 2019.

[206] Jiashu Zhang, Zihan Pan, Molly Yiming Xu, Khuzaima Daudjee, and Sihang Liu. Freeride: Harvesting bubbles in pipeline parallelism. In *Proceedings of the 26th International Middleware Conference*, Middleware '25, page 126–139, New York, NY, USA, 2025. Association for Computing Machinery.

[207] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, page 81–93, New York, NY, USA, 2021. ACM.

[208] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 01 2024.

[209] Pengle Zhang, Jia Wei, Jintao Zhang, Jun Zhu, and Jianfei Chen. Accurate int8 training through dynamic block-level fallback. *arXiv preprint arXiv:2503.08040*, 03 2025.

[210] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. HAP: SPMD DNN Training on Heterogeneous GPU Clusters with Automated Program Synthesis. In *Proceedings of the European Conference on Computer Systems (EuroSys '24)*, page 18, New York, NY, USA, 2024. ACM.

[211] WenZheng Zhang, Yang Hu, Jing Shi, and Xiaoying Bai. Poplar: Efficient scaling of distributed dnn training on heterogeneous gpu clusters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22587–22595, 2025.

[212] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Trans. Parallel Distrib. Syst.*, 33(3):489–506, March 2022.

[213] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023.

[214] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[215] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: efficient execution of structured language model programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2025. Curran Associates Inc.

[216] Yonghao Zhuang, Hexu Zhao, Lianmin Zheng, Zhuohan Li, Eric P. Xing, Qirong Ho, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. On optimizing the communication of model parallelism. In D. Song, M. Carbin, and T. Chen, editors, *Proceedings of Machine Learning and Systems*, volume 5, pages 526–540. Curan, 2023.

Appendices

A Hydrozoa

A.1 Partitioning Algorithm

Proof of Theorem 1

Theorem 1 *Hydrozoa's partitioning algorithm can produce a partition with runtime arbitrarily close to the optimal solution.*

Proof. Let $\epsilon > 0$ be arbitrary. Suppose the optimal solution uses k_0 shards and partition $p'_1, p'_2, ..., p'_{k_0}$, let t'_{k_0} and r'_1 be the corresponding t_k and r_1 values for the optimal partition. Thus, the runtime of the optimal partition is $R' = \sum_{i=1}^{k_0}(f_{p'_i} + b_{p'_i}) + 2\sum_{i=2}^{k_0} c_{p'_i} + (M-1)(t'_{k_0} + r'_1)$

Now let $B_1^{(i)}$ and $B_2^{(j)}$ be the two smallest bounds such that $t'_{k_0} \leq B_1^{(i)}$ and $r'_1 \leq B_2^{(j)}$. Suppose the solution found from dynamic programming for bounds $B_1^{(i)}$ and $B_2^{(j)}$ was $p_1, ..., p_k$. The runtime of this solution is at most $R = \sum_{i=1}^{k}(f_{p_i} + b_{p_i}) + 2\sum_{i=2}^{k} c_{p_i} + (M-1)(B_1^{(i)} + B_2^{(j)})$. We can bound the runtime of this solution as follows:

$$R - R' = 2\sum_{i=2}^{k} c_{p_i} - 2\sum_{i=2}^{k_0} c_{p'_i} + (M-1)(B_1^{(i)} + B_2^{(j)}) - (M-1)(t'_{k_0} + r'_1)$$

We know that $2\sum_{i=2}^{k} c_{p_i} \leq 2\sum_{i=2}^{k_0} c_{p'_i}$ since the algorithm picks the partition with minimal communication time. Furthermore, $B_1^{(i)} - t'_{k_0} \leq \frac{B_1^{(V)} - B_1^{(1)}}{V}$ and $B_2^{(j)} - r'_1 \leq \frac{B_2^{(V)} - B_2^{(1)}}{V}$. From this it follows that $R - R' \leq 2(M-1)\frac{\Delta}{V}$, where $\Delta = \max(B_1^{(V)} - B_1^{(1)}, B_2^{(V)} - B_2^{(1)})$.

Therefore, we can choose $V \geq \frac{2(M-1)\Delta}{\epsilon}$ such that the partition selected by the partitioning algorithm is within ϵ of the optimal partition and the theorem follows. Even with this theoretical guarantee, selecting $V = 1000$ was found to work well for the models benchmarked.

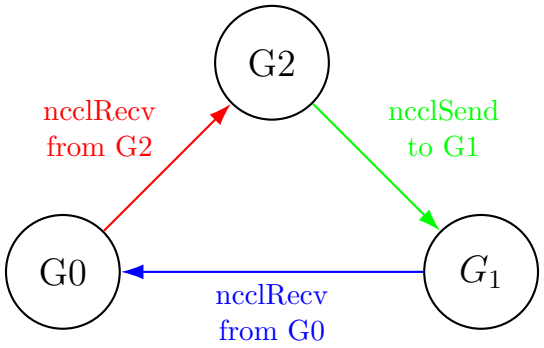
B Zorse

B.1 NCCL Communication Challenges

Challenges were encountered when using NCCL (NVIDIA Collective Communications Library [137]) for pipeline-parallel peer-to-peer communication. NCCL's communication primitives operate under strict ordering requirements that can lead to deadlocks in complex pipeline schedules.

The core issue stems from NCCL's blocking behavior in P2P communication [136]: any `ncclSend` operation blocks until the corresponding `ncclRecv` call is executed on the receiving rank, and similarly, any `ncclRecv` operation blocks waiting for the corresponding `ncclSend`. This will not block computation which can occur on a separate stream, but will block subsequent send and recv operations on the same NCCL communicator. This creates a strict global ordering requirement where communication operations must follow a consistent sequence across all GPU ranks.

Specifically, when implementing pipeline parallelism with multiple microbatches and interleaved pipeline stages of different sizes, there is significant potential for cyclic dependencies.



Due to differences in computation workload and stage execution timing, a deadlock scenario could occur as follows:

GPU 0	GPU 1	GPU 2
`ncclRecv` from G_2	`ncclRecv` from G_0	`ncclSend` to G_1
`ncclSend` to G_1	`ncclRecv` from G_2	`ncclSend` to G_0

This creates a deadlock because:

1. G_0 is waiting for data from G_2 before it can proceed to sending data to G_1
2. G_1 is waiting for data from G_0 before it can proceed to receiving data from G_2
3. G_2 needs to send data to both G_1 and G_0, but its communication operations execute in order

The key insight is that the relative ordering of `ncclSend` and `ncclRecv` operations across different GPUs creates an implicit dependency graph. If this graph contains a cycle, a deadlock will occur. In pipeline parallelism with multiple microbatches, ensuring a globally deadlock-free ordering becomes extremely challenging as the number of stages and microbatches increases.

While NCCL provides `ncclGroupStart`/`ncclGroupEnd` primitives to group operations, these mechanisms are inadequate for complex pipeline parallelism scenarios with multiple microbatches. The key limitation is that operations within a group are batched and finish together, which prevents effective pipelining when different microbatches need to be processed in parallel. Resolving these communication ordering challenges in NCCL is left to future work.

B.2 Gloo-Based Solution

To address these challenges, an alternative communication strategy was implemented using the Gloo communication library [44]. Gloo offers several advantages for pipeline communication:

- **Tag-based messaging**: Gloo's communication primitives support message tagging, allowing sends and receives to be matched based on tags rather than strict ordering
- **Non-blocking behavior**: Gloo's operations can be executed out of order without causing deadlocks
- **Flexibility**: This approach enables more dynamic communication patterns essential for efficient pipeline parallelism

The primary tradeoff with Gloo is that it communicates over CPU memory, meaning it cannot leverage high-bandwidth GPU interconnects like NVLink. This can lead to larger communication overheads when a lot of data is transferred. As the heatmaps in Figure 1

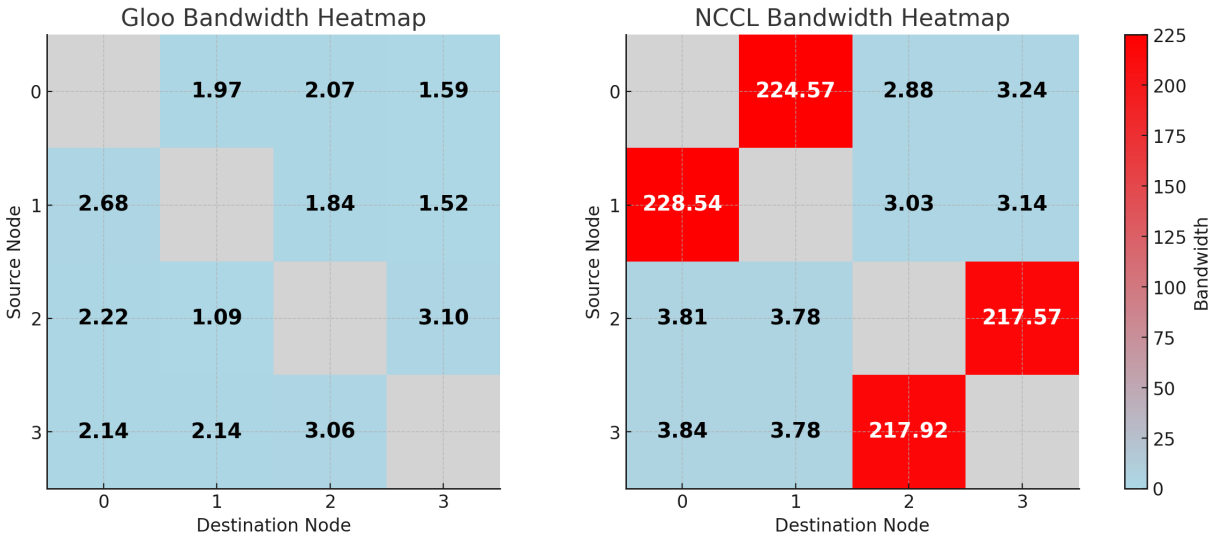


Figure 1: Bandwidth comparison heatmaps between Gloo (left) and NCCL (right) communication backends for 4 GPUs distributed across 2 machines. GPUs 0 and 1 are NVIDIA H100 GPUs within the same machine, while GPUs 2 and 3 contain NVIDIA A100 GPUs within the same machine. The color intensity represents bandwidth in GB/s, with NCCL showing significantly higher bandwidth intra-node communication between similar GPU types.

illustrate, while NCCL provides superior bandwidth for intra-node transfers, the gap narrows for inter-node communication where network bandwidth becomes the bottleneck rather than the communication backend. The use of Gloo was limited to PP communication, which is significantly less communication intensive than DP communication and can still work efficiently over slower networking. PP communication is overlapped with computation, which is typically large enough to overlap Gloo P2P communication. Moreover, NCCL is still used between GPU groups where deadlocks cannot occur.

Hybrid Communication Strategy

To maximize performance, a hybrid communication strategy was implemented that selectively uses NCCL and Gloo based on the communication context:

- **Gloo for cross-stage pipeline communication**: Gloo is used for communication between pipeline stages of different sizes where deadlocks can arise from many-to-many communication schedules.
- **NCCL for data parallelism**: Data-parallel communication uses NCCL for maximum bandwidth, since it does not conflict with P2P communication across DP groups.
- **NCCL between same-sized pipeline stages**: For pipeline stages of equal size and containing identical sets of GPUs, a predictable one-to-one communication pattern can be established that safely uses NCCL. This constraint ensures avoidance of many-to-many communication schedules that could lead to deadlocks.

C Cluster Profiling

C.1 Network Bandwidth Profiling

Accurate measurement of inter-GPU communication bandwidth is essential for optimizing distributed training configurations in the planner.

Methodology

The profiling framework systematically measures both intranode and internode bandwidth using controlled point-to-point communication tests.

1. **Cluster Information Collection**: Hardware details are gathered from each machine in the cluster, including GPU types, counts, and connectivity interfaces.
2. **Bandwidth Measurement**: Point-to-point communication tests are conducted that measure both bandwidth (Gb/s) and latency (ms) between logical GPU pairs using PyTorch's distributed communication primitives.
3. **Backend Selection**: For intranode communication, NVIDIA's NCCL backend is used, leveraging high-speed NVLink where available. For internode communication, the Gloo backend is used for P2P bandwidth and latency estimates.

Optimization Techniques

Profiling all pairs of GPUs in a large cluster would require $O(n^2)$ measurements, where n is the number of GPUs in the cluster, making the profiling process very expensive. Several optimizations are implemented to reduce profiling time:

Logical Pair Reduction for Intranode Testing Instead of measuring bandwidth between all possible GPU pairs within a node, the set of logical GPU pairs is identified for each machine with unique GPU and networking characteristics. For a machine with x unique GPU types, bandwidth is measured between $\frac{x(x-1)}{2}$ pairs, which represents all unique combinations of GPU types within the node.

For example:

- In a homogeneous machine with 8 identical GPUs, only 1 pair is measured since all GPUs have identical interconnect characteristics
- In a heterogeneous machine with 4 A100 GPUs and 4 H100 GPUs, 3 pairs are measured: (A100→A100), (H100→H100), (A100→H100)

The communication test uses non-blocking operations (isend/irecv) with sufficient warmup iterations. Furthermore, these intranode tests are conducted in parallel across all machines in the cluster.

Logical Pair Reduction for Internode Testing For internode communication, a similar logical pair reduction approach is applied:

1. **GPU Type Representation**: For each machine pair, one representative GPU of each unique type from each machine is selected. For a pair of machines with x and y unique GPU types respectively, this reduces testing complexity from $n \times m$ pairs to $x \times y$ pairs, where n and m are the total number of GPUs on each machine. In cloud environments where GPUs within a machine are typically homogeneous, this often reduces to measuring just a single pair of GPUs.
2. **Machine Type Grouping**: Further optimization is achieved by grouping machines based on:
    - Geographic region/datacenter
    - Hardware configuration (exact GPU types and counts)

For each group of identical machines, one representative machine is selected. For each pair of machine groups, bandwidth is measured between their representatives and these measurements are propagated to all machine pairs between these groups. This approach reduces the testing complexity from $O(M^2)$ to $O(G^2)$, where M is the total number of machines and G is the number of unique machine configurations (typically $G \ll M$).

D Training Configurations

In Table 1, more details are provided on the training configurations that Zorse uses for experiments in the main paper.

Cluster	Model	Batch Size	Sequence Length	Total Tokens	Microbatch Size	Ministages / GPU Group	GPU Groups
A	Llama 7B	256	4096	1M	4	1	2H100,2H100,8V100,8V100
A	Llama 13B	256	4096	1M	2	1	2H100,2H100,8V100,8V100
A	Llama 33B	256	4096	1M	2	4	2H100,2H100,8V100,8V100
A	Llama 65B	256	4096	1M	2	4	2H100,2H100,8V100,8V100
B	Llama 7B	1024	1024	1M	4	1	8A100,16V100,16A10G,24T4
B	Llama 13B	1024	1024	1M	4	4	8A100,16V100,16A10G,24T4
B	Llama 33B	1024	1024	1M	1	6	8A100,16V100,16A10G,24T4
C	Llama 7B	2048	512	1M	4	1	8V100,8V100,24T4,24T4,16A10G,24T4,24T4
C	Llama 13B	2048	512	1M	2	1	8V100,8V100,24T4,24T4,16A10G,24T4,24T4
C	Llama 33B	2048	512	1M	1	3	4V100,4V100,4V100,4V100,24T4,24T4,16A10G,24T4,24T4

Table 1: Zorse's training configurations used in the evaluation.

E Tensor Parallelism Integration

The current design of Zorse combines pipeline parallelism (PP) and data parallelism (DP) for three practical reasons. First, in a large majority of heterogeneous training use cases, using sharded DP is more communication-efficient than TP while being equally memory-efficient. As shown in Figure 1 of the paper, TP suffers from poor GPU utilization on most such VMs due to limited interconnect bandwidth, with only the 8×A100 with NVSwitch VM achieving acceptable TP performance. Second, for the targeted model sizes and cluster scales, Pipeline-Efficient ZeRO DP already reduces memory use efficiently without TP's communication overhead. The batch size constraints that TP resolves (global_batch_size $<$ num_GPUs) are only relevant for very large clusters with 1024+ GPUs, while heterogeneous clusters are typically much smaller due to resource constraints [210, 57, 76, 194, 179, 37, 133].

Future work may explore TP integration in settings where it becomes advantageous. One setting is in large clusters with widespread high-bandwidth interconnects or for models with single layers exceeding single-GPU memory capacity. TP in heterogeneous clusters would require *cross-mesh resharding*, since transitions between GPU groups with different TP configurations (e.g., 4 to 2) or between TP and FSDP demand tensor layout conversions and careful coordination to limit communication overhead [216].

The following additions to Zorse would support TP without changing Zorse's core design:

1. Layout-aware resharding routines that overlap communication with computation.
2. Updating planner latency and memory models to accurately account for TP's communication and memory overheads.
3. Considering TP when searching for the parallelization strategy in the planner.