The Dissertation Committee for Jayashree Mohan
certifies that this is the approved version of the following dissertation:

Accelerating Deep Learning Training :
A Storage Perspective

Committee:

Vijay Chidambaram, Supervisor

Amar Phanishayee

Emmett Witchel

Christopher J. Rossbach

Philipp Krähenbühl

Accelerating Deep Learning Training : A Storage Perspective

by

Jayashree Mohan, B.Tech., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2021

Dedicated to Amma, Appa, my little sister Swathi, and my husband Vivek :

My constant source of inspiration, encouragement, and love.

This wouldn't have been possible without you...

Acknowledgments

Over the last five years of my PhD, I've often wondered what I would write here when the time comes. Now that the moment is here, I realize I may not have the right words to express my gratitude to the people who have been a part of this wonderful journey.

I will forever be indebted to my advisor Vijay Chidambaram. Vijay took a big gamble in 2016 by hiring a freshly minted undergrad like me, with not much research experience. I joined UT, planning to do networks research; through a fortunate turn of events, I started working with Vijay on a course project. Ever since, there was no turning back. Vijay has taught me several valuable things; how to pick research problems, how to write papers, how to appreciate simplicity in research, and how to handle rejections. He has always celebrated even the smallest of our successes, and stood beside us in the toughest of times. Vijay, I've learnt from you that being kind and compassionate is the greatest virtue. Without all your help and guidance, I would not be where I am today. You're the best advisor I could have ever asked for. Thank you, Vijay!

It wouldn't be an exaggeration to say that my research career changed for good after I met Amar Phanishayee. Amar mentored me during my internship at MSR Redmond in 2019 and introduced me to the crazy yet beautiful world of systems for ML. I took a big leap of faith and changed my thesis direction after I started working with Amar and it goes without saying that Amar features prominently in all the work done as a part of this dissertation. I've always been in awe of his work strategy, and tried to imbibe the way he approaches problems. He has taught me how important it is for one to trust, love, and be passionate about one's work before they go on to convince the world of its greatness. Apart from his technical rigor, Amar is such a fun person to work with; his enthusiasm and laughter is just so contagious. With him around, there are no rough days at work. Thank you, Amar for burning the midnight oil with me during each of the deadlines, and for going above and beyond in mentoring me during my job search. I'd be very happy if in future, I can be half as good as the mentor you are!

I am thankful to my dissertation committee members, Emmett Witchel, Philipp Krahenbuhl, and Christopher Rossbach for their invaluable feedback towards my work. Their challenging questions and comments have significantly improved this dissertation. Special thanks are also due to Katie Traughber Dahm, Lydia Griffith and other staff at UTCS for their kindness and timely assistance with all administrative work throughout my time at UT.

I was fortunate to spend my summers at Microsoft Research Labs (MSR) across the globe. At MSR, I had the distinct pleasure of interacting with, and being mentored by several wonderful researchers. I am grateful to Muthian Sivathanu for giving me an opportunity to be a part of project Instalytics at MSR India and always motivating me to think big. I was very fortunate to be mentored by Ant Rowstron, Richard Black, and Andromachi Chatzieleftheriou at MSR Cambridge. Andromachi, thank you for striking a random conversation at SOSP '17, my internship wouldn't have been such a fun learning experience if not for you! Thank you, Richard and Ant for going above and beyond to motivate and mentor me not just during my internship, but also throughout my PhD. I am also thankful to MSR for generously supporting my PhD through the Microsoft Research PhD fellowship.

I was fortunate to work with several wonderful collaborators during my PhD; Ashlie Martinez, Matt Halpern, Vijay Janapa Reddi, Sanidhya Kashyap, Taesoo Kim, Melissa Wasserman, Ashish Raniwala, and Janardhan Kulkarni, who have helped me at various stages of my work - formulation, debugging, and experimentation. In particular, a special mention to Ashlie, with whom I co-authored the CrashMonkey project. Her commitment to work, and attention to detail are exemplary. Thank you for being an amazing collaborator Ashlie, I've definitely learnt some good coding practices from you!

I was incredibly fortunate to have wonderful labmates and friends who made my PhD journey joyful and fun. UT SaSLab members have been my family for the past five years - Soujanya Ponnapalli, Rohan kadekodi, Sekwon Lee, Aashaka Shah, Supreeth Shastri, thank you all for always being there, and for selflessly helping me when I needed the most both personally and professionally. I am going to terribly miss our lunch table conversations about nearly any topic under the sun. Soujanya, I am going to miss our intense technical and non-technical discussions - I've learnt so much from you. Rohan, our lab's IT support and the most selfless person I've met. I'll miss our tea times where we've discussed life over a cup of chai. Sekwon, your passion and commitment towards work has always amazed me, and I'll fondly remember our shopping adventures. Aashaka, I admire your straightforward nature and how you balance work and fun. Supreeth, I'll miss all our philosophical discussions; there was not a single dull moment with you around in the lab. I wish the pandemic did not cut short the times we all spent at lab, nevertheless I'll always fondly remember you "cubies"!

Special thanks to the amazing set of friends I made during grad school - Anisa Qazi, Naveena Sankaranarayanan, Pandian Raju, Dhathri Purohit, Chandana Amanchi, Nayan Singhal, Mit Shah, Devvrit Khatri, Kishore Punniyamurthy, Karthikeyan Haribalakrishnan, Sai Vikneshwar, Aastha Tripathi, Sharukh Shaikh, Yashwant Marathe - who have listened to my problems, and been by my side during the ups and downs of grad life. The numerous potlucks, game nights, some successful spring break trips, some failed travel plans (courtesy : Kishore), the laughter and the cries - I'll cherish all of it. Thank you, Aditya Chandrasekar, Rucha Vaidya, Shilpa Vijay, Vikranth Reddy, Aakanksha Naik and all of the 'Viterbi' gang for being an amazing support system despite being geographically distributed. Our successful mini-reunion was so refreshing.

This section would be incomplete without a special mention to Crash-Monkey & Ace - the first project I worked on as a grad student. The lessons I learned from this project, instilled the confidence in me to take on bigger challenges like venturing into a new line of research as late as in the fourth year of my PhD. CrashMonkey, I am sorry you couldn't be a part of this dissertation; but I want you to know that you will always be my most favourite!

This PhD wouldn't have been possible without the constant support and encouragement of my family. I am a proud first-generation graduate in my family. My mom and dad have been an incredible support system throughout my school, college, and graduate education. My sister Swathi added the much needed color in my life with her silly fights, arguments, and love. They stood by me and my dreams against all odds, no matter what, and celebrated every minuscule achievement of mine as if it was their biggest success. I dedicate this thesis to you, and your unconditional love. I am also grateful to my in-laws for being so understanding and supportive of my career.

Finally, the one person who I cannot thank enough - my husband, Vivek. A decade ago, he promised to stay by my side and support my dreams. A man of his words, he has made innumerable compromises till date so that we could see a future together. Thank you, Vivek for keeping me sane throughout this tremulous journey despite being half way around the world from me. I am eternally thankful to you for believing in me whenever I doubted myself. This dissertation also officially marks the end of our seemingly never-ending long distance relationship. Thank you for everything, Vivek. This degree is as much yours as it is mine!

Accelerating Deep Learning Training : A Storage Perspective

Publication No. ____________________

Jayashree Mohan, Ph.D.
The University of Texas at Austin, 2021

Supervisor: Vijay Chidambaram

Deep Learning, specifically Deep Neural Networks (DNNs), is stressing storage systems in new ways, moving the training bottleneck to the data pipeline (fetching, pre-processing data, and writing checkpoints), rather than computation at the GPUs; this leaves the expensive accelerator devices stalled for data. While prior research has explored different ways of accelerating DNN training time, the impact of storage systems, specifically the *data pipeline*, on ML training has been relatively unexplored. In this dissertation, we study the role of data pipeline in various training scenarios, and based on the insights from our study, we present the design and evaluation of systems that accelerate training.

We first present a comprehensive analysis of how the storage subsystem affects the training of the widely used DNN models by building a tool, *DS-Analyzer*. Our study reveals that in many cases, DNN training time is dominated by *data stalls*: time spent waiting for data to be fetched from (or written to) storage and pre-processed. We then describe *CoorDL*, a user-space data loading library to address data stalls in dedicated single-user servers with fixed resource capacities. Next, we design and evaluate *Synergy*, a workload aware scheduler for shared GPU clusters that mitigates data stalls by allocating auxiliary resources like CPU and memory cognizant of workload requirements. Finally, we present *CheckFreq*, a framework that frequently writes model state to storage (checkpoint) for fault-tolerance, thereby reducing wasted GPU work on job interruptions, while also minimizing stalls due to checkpointing.

Our dissertation shows that data stalls squander away the improved performance of faster GPUs. Our dissertation further demonstrates that an efficient data pipeline is critical to speeding up end-to-end training, by building and evaluating systems that mitigate data stalls in several training scenarios.

Table of Contents

Acknowledgments **v**

Abstract **x**

List of Tables **xvii**

List of Figures **xviii**

Chapter 1. Introduction **1**

1.1 Dissertation Statement 2

1.2 Dissertation Contributions 3

1.2.1 Analyzing the Impact of the Data Pipeline 4

1.2.2 Mitigating Data Stalls in Single-User Scenarios 5

1.2.3 Mitigating Data Stalls in Multi-Tenant Clusters 6

1.2.4 Mitigating Checkpoint Stalls during Interruptions 8

1.3 Dissertation Organization 9

Chapter 2. Background **11**

2.1 Deep Neural Networks 11

2.2 How DNNs are Trained 12

2.2.1 Hyperparameter Search 13

2.2.2 Train to Accuracy 13

2.3 DNN Data Pipeline 15

2.3.1 Storage Subsystem 15

2.3.2 The Data Pipeline 16

2.3.3 The DNN ETL Requirements 18

2.3.4 DALI : Fast Data Pipelining 19

2.4 DNN Training Scenarios 19

2.4.1 Single-GPU Training 20

2.4.2 Distributed training 20
2.4.3 Training in Multi-Tenant Clusters 21
2.5 DNN Checkpointing 24
2.5.1 Checkpointing strategy 24
2.5.2 Recovery Time 25
2.6 DNN Training Frameworks 25
2.7 Summary 26

Chapter 3. DS-Analyzer: Analyzing Data Stalls 27
3.1 Data Stalls 28
3.1.1 Example of data stalls in ResNet18 training 29
3.2 Methodology 30
3.2.1 Models and Datasets 31
3.2.2 Training environment 32
3.2.3 Training parameters 33
3.2.4 Training metrics 33
3.3 Measuring data stalls using DS-Analyzer 33
3.4 Data Stalls in DNN Training 35
3.4.1 When dataset resides on remote storage 35
3.4.2 When datasets cannot be fully cached 38
3.4.3 When datasets fit in memory 41
3.4.4 Data stalls exist across training frameworks 46
3.4.5 Analysis summary 48
3.5 DS-Analyzer: Predictive analysis 49
3.5.1 Example : Predicting optimal cache size 52
3.6 Limitations and Discussion 53
3.7 Summary 54

Chapter 4. CoorDL: Mitigating Data Stalls in Single-User Scenarios 55
4.1 Motivation 56
4.1.1 Assumptions and Goals 58
4.2 CoorDL 58

4.2.1 Overall Architecture 59
4.2.2 The MinIO cache 61
4.2.3 Partitioned Caching 63
4.2.4 Coordinated Prep 65
4.2.5 Implementation 68
4.3 Evaluation 70
4.3.1 Single-server Multi-GPU training 72
4.3.2 Multi-Server Distributed Training 74
4.3.3 Hyperparameter Search 76
4.3.4 Training to Accuracy with CoorDL 79
4.3.5 Resource Utilization 80
4.3.6 CoorDL on DGX-2 83
4.4 Limitations and Discussion 84
4.5 Summary 86

Chapter 5. Synergy: Mitigating Data Stalls in Multi-Tenant Clusters 87
5.1 Motivation 88
5.1.1 Example 91
5.1.2 Assumptions and Goals 93
5.2 Synergy: Design 95
5.2.1 Optimistic Profiling 96
5.2.2 Scheduling mechanism 100
5.2.3 Synergy-Greedy: Greedy Scheduling 100
5.3 Scheduling Mechanism 102
5.3.1 Synergy-Opt 102
5.3.1.1 A Upperbound on Throughput in an Optimal Solution 104
5.3.1.2 An LP formulation 104
5.3.1.3 Feasible Allocation on Multiple Machines 106
5.3.1.4 Challenges with operationalizing Synergy-Opt . 110
5.3.2 Synergy-Tune 110
5.3.3 Discussion 113

5.3.4 Implementation 114
5.4 Evaluation 115
5.4.1 Experimental setup 116
5.4.2 End-to-End Physical Cluster Experiments 119
5.4.3 End-to-end results in simulation 121
5.4.4 Simulation with production traces 123
5.4.5 Impact of workload split 127
5.4.6 Comparison to Synergy-Opt 129
5.4.7 Comparison to DRF and Tetris 131
5.5 Limitations and Discussion 133
5.6 Summary 135

Chapter 6. CheckFreq: Mitigating Checkpoint Stalls when Training with Interruptions 136

6.1 The Current State of Checkpointing 137
6.1.1 Checkpointing is Incorrect 138
6.1.2 Checkpointing is Inefficient 139
6.1.3 Summary 141
6.2 CheckFreq: Design and Implementation 142
6.2.1 Assumptions and Goals 142
6.2.2 CheckFreq Recovery Guarantees 143
6.2.3 Design 145
6.2.3.1 Checkpointing Mechanism 146
6.2.3.2 Checkpointing Policy 152
6.2.4 Implementation 155
6.3 Evaluation 156
6.3.1 Experimental setup 157
6.3.2 Accuracy implications 158
6.3.3 Performance of checkpointing mechanism 160
6.3.3.1 Checkpoint stalls 160
6.3.3.2 Breakdown of benefits 161
6.3.4 Checkpointing policy 162
6.3.5 Recovery time 164

6.3.6 End-to-end training 165
6.4 Limitations and Discussion 166
6.5 Summary 167

Chapter 7. Related Work **168**
7.1 Optimizing DNN training time 168
7.2 DNN cluster schedulers 172
7.3 Optimizing DNN checkpointing 174

Chapter 8. Conclusion and Future Work **178**
8.1 Looking Forward 179
8.2 Concluding Remarks 181

Bibliography **182**

Vita **221**

List of Tables

3.1 Models and datasets used in this analysis 31
3.2 Server configurations used in this analysis 32
3.3 Storage I/O in HP search 40
3.4 HP search with 8 jobs 46
3.5 Key findings and implications of our analysis of data stalls . . 48

4.1 Key findings and implications of data stall analysis 57
4.2 Overview of techniques 59
4.3 Impact on fetch misses and disk IO 73
4.4 HP search with CoorDL on a fully cached dataset 79

5.1 Example jobs 92
5.2 GPU-proportional allocation 92
5.3 Resource-sensitive allocation 92
5.4 Models used in this work 117
5.5 Physical cluster experiments 120
5.6 Average JCT with Synergy across different scheduling policies. 126
5.7 Cluster metrics 127

6.1 Overview of techniques used by CheckFreq 142
6.2 Server configurations 157
6.3 Breakdown of benefits 161
6.4 Checkpoint frequency 162
6.5 Adaptive frequency tuning 163
6.6 Average recovery time (CF - CheckFreq) 163

List of Figures

2.1 Deep Neural Network (DNN) 12
2.2 Storage subsystem in DNN Training 16
2.3 Recovery time across models 24

3.1 Data Stall 28
3.2 Data Pipeline for ResNet18 30
3.3 Fetch cost with remote stores 37
3.4 Training with remote stores 37
3.5 Fetch stalls 38
3.6 ResNet18 with varying cache 39
3.7 Impact of CPU cores on training 41
3.8 Impact of CPU on data prep 42
3.9 8-GPU ResNet18 training 43
3.10 Prep stall across DNNs 43
3.11 Breakdown of prep overhead 44
3.12 Impact of batch size on prep 45
3.13 Data stalls across frameworks 47
3.14 Estimating optimal cache size with DS-Analyzer 51

4.1 Architecture of CoorDL 60
4.2 Cache hits with MinIO 62
4.3 Single server training 72
4.4 Distributed training across servers 74
4.5 Distributed training with CoorDL 75
4.6 Hyperparameter search 76
4.7 Breakdown of benefits due to coordinated prep 77
4.8 Varying number of HP jobs 78
4.9 Top-1 validation accuracy during training 80

4.10 Disk I/O pattern with MinIO (ResNet18 on OpenImages) . . 81
4.11 CPU utilization with MinIO 82
4.12 Memory utilization of coordinated prep 83
4.13 Evaluation of CoorDL on DGX-2 84

5.1 CPU sensitivity 89
5.2 Resource sensitive scheduling 93
5.3 Optimistic profiling 97
5.4 Validating Synergy's optimistic profiling strategy 98
5.5 Average JCT and CDF for FIFO 122
5.6 Average JCT and CDF for LAS policy 124
5.7 Average JCT and CDF for SRTF 125
5.8 CDF for SRTF 126
5.9 Varying workload split 128
5.10 Cluster GPU utilization 130
5.11 CPU utilization 130
5.12 Scalability of our scheduling mechanism 131
5.13 Comparison to big data scheduling policies 132

6.1 Recovery time across models 138
6.2 Bounding recovery time 145
6.3 Training with CheckFreq 146
6.4 Pipelining checkpoint with compute 147
6.5 Resuming iterator state 152
6.6 Impact of resumable data iterator on accuracy 159
6.7 Runtime overhead for various models 160
6.8 End-to-end training 165

Chapter 1

Introduction

Deep learning is a branch of machine learning that uses large amounts of data and computation to solve complex tasks such as image classification. The advent of deep learning has led to significant breakthroughs in the recent times in several domains. Deep learning architectures such as Deep Neural Networks (DNNs) have allowed us to tackle problems that were previously intractable in fields like computer vision [117, 189, 91], speech recognition [86], natural language processing [204, 208], and even predictive health-care [193].

The rising success of DNNs can be attributed to the advancement in neural architectures [169, 211, 117, 161, 181, 97, 219, 195], construction of large-scale labeled datasets [179, 67, 80, 66, 39], and growing GPU compute speeds [93, 112, 162, 148, 155, 75]. Modern DNNs for tasks like image classification, object detection, and speech recognition are data intensive; they deal with multimedia data like images, videos, and audio, and have complex pre-processing pipeline involving online data transformations. Further, GPU compute speeds today are growing at an unprecedented rate. For instance, the total computational power in top-of-the-line GPUs increased by 32$\times$ over the last five years [186] and the latest NVIDIA A100's tensor cores provide up to 20× higher performance over the prior generation [162]. Finally, open-source datasets are exploding in size; for example, in contrast to the popular 140GB ImageNet-1K dataset, the youtube-8M dataset used in video models is about 1.53 TB for just frame-level features [40], while the Google OpenImages dataset is about 18 TB [80]. While these trends are a boon for DNN training, it also *emphasises the need for efficient training infrastructure to utilize them.*

What matters at the end is if we can bring in data for the GPUs to compute on, at a fast rate. While prior research focused on accelerating DNN training by reducing communication overhead [128, 217, 90, 104, 151], GPU memory optimizations [178, 54, 103], and compiler-based operator optimizations [202, 53, 108], the impact of storage subsystem (DRAM, CPU, and persistent storage), on DNN training has been relatively unexplored.

Dissertation Question. *Does the storage subsystem and the efficiency of the data pipeline have an impact on DNN training time? If so, how can we accelerate training by efficiently utilizing available resources, without additional hardware upgrades?*

1.1 Dissertation Statement

In many data-intensive DNNs, training time is dominated by data stalls*: time spent waiting for data to be fetched and pre-processed. An efficient* *data pipeline is can speed up end-to-end training significantly without additional hardware resources.* Our dissertation demonstrates this by analyzing, building and evaluating systems that mitigate data stalls in several training scenarios.

1.2 Dissertation Contributions

- The first comprehensive analysis of data stalls in DNN training and show that data stalls squander away the benefits of fast GPUs for several data-intensive DNNs using a new tool, DS-Analyzer (§3)
- The design and implementation of a new data loading library, CoorDL that accelerates DNN training by minimizing data stalls, without requiring additional hardware (§4)
- A resource-sensitivity aware scheduler, Synergy for multi-tenant, shared GPU clusters that profiles the resource sensitivity of jobs and performs multi-resource workload-aware assignments across a set of jobs using a new near-optimal online algorithm (§5)
- An automated, fine-grained checkpointing framework, CheckFreq that algorithmically determines the checkpointing frequency at iteration granularity, while pipelining checkpointing with computation to reduce checkpoint stalls (§6)

The rest of this chapter provides a brief overview of each of the thrusts mentioned above.

1.2.1 Analyzing the Impact of the Data Pipeline

During DNN training, the input *data pipeline* works as follows. Data items are first fetched from storage and then *pre-processed* at the CPU. For example, for many important and widely-used classes of DNNs that work on images, audio, or video, there are several pre-processing steps: the data is first decompressed, and then random perturbations such as cropping the image or rotating it are performed to improve the model's accuracy [167]. Once pre-processed, the data items are sent to the GPUs for processing, where the model weights are updated by performing computations on the data. We describe this data pipeline with an example in §3.1.1. This data fetch and pre-processing is normally pipelined with the GPU computation.

Data Stalls. Ideally, the data pipeline should keep the GPUs continuously busy processing data; we term this GPU-bound. Unfortunately, DNN training is often IO-bound, bottlenecked by fetching the data from storage, or CPU-bound, bottlenecked by pre-processing data in memory. Collectively, we term these bottlenecks *data stalls* and differentiate between *prep stalls* (time spent pre-processing), and *fetch stalls* (time spent on read IO).

Analyzing Data Stalls. We present the first comprehensive analysis of *data stalls* in DNN training across different models and tasks with large datasets, measured in a production cluster at Microsoft [146, 145]. Our analysis yields several interesting insights. First, our analysis shows that data stalls are prominent in popular computer vision and audio DNNs, as opposed to text-based

NLP models. We find that data stalls squander away the improved performance of faster GPUs, even on ML optimized servers like the DGX-2 [22]. Second, there is a large amount of redundant work done by the data pipeline during hyperparameter search and distributed training where the same data items are fetched and pre-processed by multiple jobs or multiple nodes. Third, relying on OS abstractions (like Page Cache) is inefficient for DNN workloads. Finally, these data stalls occur across frameworks such as PyTorch and TensorFlow.

Predictive What-If Analysis of Data Stalls. We develop a tool, DS-Analyzer, that uses differential analysis between runs (*e.g.,* comparing a run where data is completely cached vs when data needs to be fetched from storage) to identify data-stall bottlenecks. Using the measured data stalls, it answers what-if questions to help practitioners predict and analyze data stalls (*e.g.,* What would be the impact on data stalls if DRAM capacity increased by 2×?).

Mitigating Data Stalls. Our data stall analysis reveals several key insights with respect to the need for efficient data pipeline for DNN training. We build upon the insights gained from our analysis to design systems that mitigate data stalls in various DNN training scenarios described below.

1.2.2 Mitigating Data Stalls in Single-User Scenarios

One of the most common DNN training scenarios is to use exclusive on-premise servers, or cloud GPU VMs. In this scenario, a single training job runs on either (1) a single GPU, (2) across multiple GPUs in a server, or (3) across multiple servers. Further, multiple instances of the job (hyperparameter search) could be launched in a server. We present *CoorDL*, a data loading library that an individual DNN job can use to accelerate training by minimizing data stalls. CoorDL does not impact accuracy; training can sample as usual from the entire dataset, regardless of what is cached. CoorDL does not require additional hardware, running over commodity networking and storage hardware.

CoorDL introduces three techniques to overcome data stall overheads. First, it introduces MinIO, a software cache that is specialized for DNN training. Second, a *partitioned caching* technique to coordinate the MinIO caches of servers involved in distributed training over commodity network stack. Third, a *coordinated prep* technique to carefully eliminate redundancy in data prep among concurrent hyperparameter search jobs in a server. CoorDL addresses both fetch and prep stalls and accelerates several common training scenarios: hyperparameter search (by upto 5.7×), single-server DNN training (by upto 2×), and multi-server DNN training (by upto 15×).

1.2.3 Mitigating Data Stalls in Multi-Tenant Clusters

Collocating ML training workloads in a shared, multi-tenant cluster is a very common setup in several large organizations, for both research and production [105, 210]. Existing schedulers for DNN training consider GPU as the dominant resource, and allocate other resources such as CPU and memory proportional to the number of GPUs requested by the job (GPU-proportional)

[153, 209, 133, 87, 121, 166, 50, 203, 49]. Unfortunately, these schedulers do not consider the impact of a job's sensitivity to allocation of CPU, memory, and storage resources. Contrary to practice, our data stall analysis reveals that different models require different allocations of memory and CPU to mask data stalls. Based on this observation, we describe *Synergy*, a resource-sensitive scheduler for shared GPU clusters. The key insight behind Synergy is that, it is possible to exploit the heterogeneity in resource sensitivity across DNNs to allow disproportionate allocation of resources rather than using traditional GPU-proportional allocation to improve the overall cluster utilization and efficiency. While doing so, Synergy ensures a job gets less than GPU-proportional auxiliary resources *only* if such an allocation does not degrade the job throughput compared to a GPU-proportional allocation.

To achieve this, Synergy introduces a new *optimistic profiling* technique to infer the sensitivity of DNNs to different resources; it only empirically profiles the job throughput for varying CPU allocations, assuming maximum memory allocation. It then *analytically* estimates the job throughput for all combinations of CPU and memory along with the respective storage bandwidth requirement for each allocation. Synergy then performs multi-resource workload-aware assignments across a set of jobs using a new near-optimal online algorithm.

Our experiments show that across various scheduling policies like LAS, SRTF, FIFO, etc, the resource sensitive scheduling mechanisms used by Synergy can improve cluster objectives such as average job completion time (JCT) by up to 1.5× on a physical cluster of 32 GPUs. On a large simulated cluster of 128 GPUs, Synergy improves average JCT by up to 3.4× when compared to the respective GPU-proportional scheduling policy.

1.2.4 Mitigating Checkpoint Stalls during Interruptions

Interruptions to DNN training jobs are common. Be it dedicated enterprise clusters or cloud instances, failures due to software and hardware errors are inevitable [134, 88, 79, 43, 138, 152]. When interruptions occur, the long running, stateful, DNN job terminates abruptly, wiping out the model parameters in-memory; resulting in the loss of several hours of GPU work. To tackle this, the model weights and optimizer state (collectively, model state) are occasionally written to persistent storage, for fault tolerance. This is termed *checkpointing*. Traditionally, models are checkpointed at epoch boundaries [140]. Training has to briefly pause to accurately checkpoint the current state; the GPU (or any accelerator) remains idle until checkpoint completes, introducing *checkpoint stalls* in training.

With the trend in growing dataset sizes [36, 39, 119], and larger, complex model architectures [41, 161, 48], DNN epoch time and overall training time is also increasing. Therefore, it is important to frequently checkpoint training progress, at a finer granularity than epochs *i.e.,* at iteration level, while minimizing training overhead due to checkpoint stalls.

To achieve these goals, we introduce *CheckFreq*, a fine-grained checkpointing framework for DNN training. CheckFreq strikes a balance between ensuring a low runtime overhead and providing a high checkpointing frequency, so that there is minimal loss of GPU time in the event of job interruptions or failures by performing iteration-level checkpointing. CheckFreq has two major components; a checkpointing mechanism that performs *correct, low-cost checkpointing*, and a checkpointing policy that automatically determines *when to checkpoint*. To this end, we build upon a set of techniques from the High Performance Computing (HPC) and storage community, alongside novel DNN-specific optimizations such as pipelined in-memory snapshots, utilizing spare GPU capabilities for fast snapshot, and a DNN-aware systematic profiling for dynamic tuning of checkpointing frequency. Our evaluation across a variety of models, GPUs, and storage types confirms that CheckFreq reduces the wasted GPU time from order of hours to just under a minute, while incurring less than 3.5% runtime overhead, as compared to the existing epoch-based checkpointing schemes.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 reviews background concepts and relevant literature in the field; DNN training scenarios, its data pipeline, and its checkpointing strategies. Chapter 3 discusses the comprehensive analysis of data stalls in DNN training and shows that data stalls squander away the improved performance of faster accelerators. Chapter 4 introduces a coordinated data loading library CoorDL that uses the insights from our analysis to mitigate stalls in isolated, single-user training scenarios. Chapter 5 discusses how to mitigate data stalls in shared multi-tenant clusters using resource-sensitivity aware scheduling in Synergy. Chapter 6 introduces CheckFreq, an automated checkpointing framework that minimizes both lost GPU work and checkpoint stalls in training environments with frequent interruptions. Chapter 7 discusses prior work related to the techniques and systems introduced in this dissertation. Chapter 8 presents the summary and concluding remarks, along with new research problems this dissertation opens up.

Chapter 2

Background

In this chapter, we present the background required for various parts of this dissertation and explain different ML terms used in this thesis. We start by describing what DNNs are (§2.1), and how they are trained, along with various terms used throughout this dissertation (§2.2) . We next discuss the data pipeline in DNN training (§2.3). Finally, we describe different training scenarios such as distributed training, multi-tenant clusters (§2.4), and training with interruptions using checkpoints (§2.5).

2.1 Deep Neural Networks

Deep Learning is a branch of machine learning that is inspired and modeled based on how the human brain works. It allows machines to solve relatively complex problems. Deep Learning architectures such as Deep Neural Networks (DNNs) emulate the learning approach that the human brain uses to gain certain knowledge. Similar to the neurons in the brain, DNNs contain artificial neurons (nodes) that can transmit signals among one another. These nodes are connected as layers, where each layer performs a specific transformation on the input data and passes it downstream to the next layer. Signals

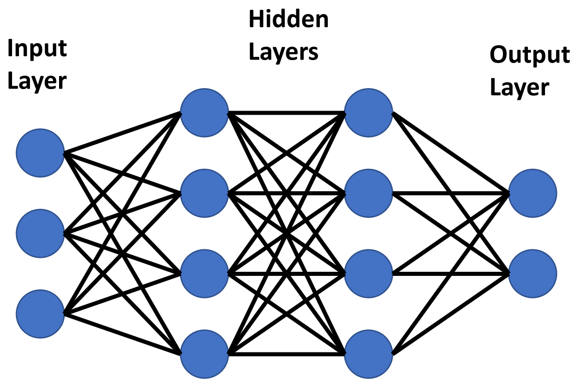


Figure 2.1: **Deep Neural Network (DNN)**.

travel from the first layer (input), to the last layer (output), traversing through multiple intermediate hidden layers as shown in Fig 2.1. When the inputs are transmitted between nodes, weights are applied to the inputs along with the bias. Weight decides how much influence the input will have on the output, while bias is a constant that is an additional input to the subsequent layer.

2.2 How DNNs are Trained

Training a DNN is the process of teaching it how to perform a task like image classification, or language translation. It is the process of determining the set of weights and bias in the network, collectively called the *learnable parameters*. During the training process, labelled data (also called training dataset) is fed to the DNN, and the DNN makes a prediction about what the data represents. Using the pre-calculated label associated with the data, any error in the prediction made by the model is used to update the weights. As the training process continues, the weights are further adjusted until the model makes predictions with sufficient accuracy.

Training a DNN model to reach a given accuracy consists of two steps: (i) finding the optimal set of hyperparameters for the learning process, and (ii) running the learning algorithm until the desired accuracy is reached.

2.2.1 Hyperparameter Search

There are many parameters for the learning algorithm that must be provided before the start of training; for *e.g.,* learning rate, its decay, dropout, and momentum. These hyperparameters influence the speed and quality of learning. During the search process, we start several training jobs; each job trains the model with different hyperparameters, on each available GPU (or a distributed job across several GPUs); progress is checked after a few epochs and the worst-performing candidates are killed and replaced by new jobs with different hyperparameters that are chosen using algorithms like Random Search [47], Population Based Training [102], Median Stopping Rule [78], or Hyperband [123]. Tuning hyperparameters is crucial for generating DNN models that have high accuracy [171].

2.2.2 Train to Accuracy

Once the hyperparameters have been decided, the second step is to obtain models with high accuracy by training it with input data.

The training process. The DNN is trained over multiple rounds termed *epochs*. Each epoch processes all items in the dataset exactly once, and consists of multiple *iterations*; each iteration processes a random, disjoint subset of the data termed a *minibatch*.

During the training process, every iteration performs the following steps in order.

- **Data Fetch**. Fetches a minibatch of data from storage and optionally caches it in memory (DRAM) for faster subsequent accesses.

- **Data augmentation (pre-processing / prep)**. Applies a set of random pre-processing operations on the fetched minibatch in memory. For *e.g.,* in popular image classification models like the ResNets, pre-processing includes randomly cropping the input image, resizing, rotating, and flipping it.

- **Forward pass**. In the forward pass, a model function is applied on the minibatch of data to obtain the prediction. Input data from the minibatch is fed to the first layer of the model. Each subsequent hidden layer accepts the input data from the prior layer, processes it based on an activation function and transmits it to the next layer. An *activation function* defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. In other words, this function helps the network prune the irrelevant data points and only use the vital information. The output of the last layer is called the *prediction*.

- **Backpropagation**. Based on the model's predicted label and the actual label of each image present in the labelled training dataset, the output layer uses a loss function to determine how much the prediction deviates from the correct answer; each layer in the DNN then computes a gradient of the loss in a backward pass.

- **Weight update**. Using the gradients computed in the backward pass, the learnable model parameters are updated.

The DNN is trained until a target accuracy is reached; this is typically for a fixed, large number of epochs. The final learned parameters are then saved to persistent storage for inference. To perform inference on the model, the DNN is initialized with the learned parameters and the output is predicted on a set of data that the model has not seen before during training.

2.3 DNN Data Pipeline

We now describe the role of storage subsystem, the steps involved in the data pipeline of DNN training, and its ETL requirements.

2.3.1 Storage Subsystem

The storage subsystem refers to the collection of hardware resources in a machine that aid data storage and access. This includes the following : (1) Persistent storage device such as Solid State Drives (SSDs), Hard Disk Drives (HDDs) or new fast storage technologies like NVMe SSDs, and Persis-

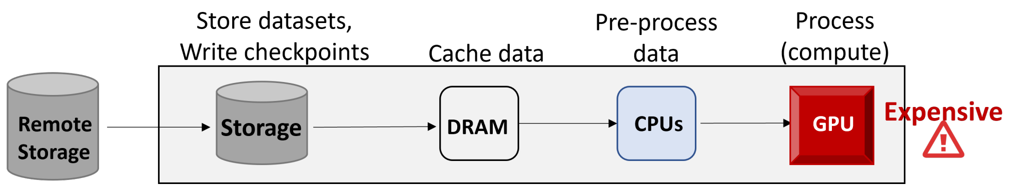


Figure 2.2: **Storage subsystem in DNN Training**. DNN training uses all the resources in a server from storage and CPU for fetching and pre-processing the dataset to the GPUs that perform computation on the transformed data.

tent Memory (PM). (2) Memory or DRAM to used to cache data for faster subsequent accesses, and also to provide a working space for process memory requirements. (3) Finally, the CPUs (typically multi-core, and multi-threaded, that is required to fetch data from storage as well as perform any in-memory processing operations on the fetched data.

In the context of this dissertation, we collectively refer to these three hardware components typically involved in any data ingestion process as the *storage subsystem*. In this dissertation, we explore the role of the storage subsystem in DNN training, and find ways to build software systems that efficiently utilize the storage subsystem to accelerate end-to-end DNN training.

2.3.2 The Data Pipeline

Training data typically resides on remote storage like AWS S3, Azure blob, or cluster filesystems like HDFS or GFS. When data resides remotely, the first epoch of training fetches data over the network and stores it locally for subsequent use. Remote data is downloaded on to local SSD when it is first accessed in epoch one, and mimics local training from the second epoch onwards. This training data is also cached in the Operating System's Page Cache to speed up subsequent accesses. It is then pre-processed at the CPUs before copying over to the GPUs for processing as shown in Figure 2.2. Additionally, there is an output data pipeline, where the model state and learned parameters are written to persistent storage periodically (checkpoints), typically at the end of an epoch. We discuss more about this in §2.5.

To summarize, there are three phases in every training iteration.

1. *Fetch.* First, a minibatch of data is fetched either from persistent storage or from cache. We call this the fetch.

2. *Prep.* Next, the fetched minibatch is pre-processed at the CPU. For example, for image classification models, common prep operations are decoding the image file, applying a random crop, flip etc.

3. *GPU Compute.* Once pre-processed, this batch is ready to be consumed at the GPU and is copied over to the GPU for processing. This comprises the forward and backward passes discussed in §2.1.

The *fetch* and *prep* phases are collectively called the *data loading phase* for a given minibatch. In most training frameworks, including PyTorch, data loading and the GPU compute phase execute in a pipelined fashion; i.e., subsequent minibatches are *prefetched and pre-processed* by data preparation threads, using multiple CPU cores on the machine, as the GPU computes on the current minibatch of data. Such pipelining ensures that the expensive accelerator devices are constantly fed with data and do not stall waiting for the next minibatch.

2.3.3 The DNN ETL Requirements

In every epoch of training, the input dataset is subjected to a ETL (extract-transform-load) before being processed at the GPU (or any other accelerator). The ETL process in the data pipeline of popular image-based DNN training imposes several unique data ordering constraints to ensure model convergence and achieve state-of-the-art accuracy.

- *Ordering invariant.* The dataset must be shuffled every epoch to ensure the order in which data items are accessed are random in each epoch
- *Frequency invariant.* An epoch must use *all* data items in the dataset *exactly* once
- *Transformation invariant.* In every epoch, the data transformations(preprocessing) must be random; the same transformed item should not be used across epochs.

Prior work has theoretically and empirically demonstrated that relaxing these constraints will affect the convergence rate of SGD [135, 59, 126, 167]. While some NLP and recommendation models may not require random preprocessing and data shuffling every epoch, the focus of our work is computer vision and audio models where random data augmentation and shuffling is the default and common practice [140, 176]. Therefore, in this dissertation, all our experiments abide by the aforementioned ETL requirements.

2.3.4 DALI : Fast Data Pipelining

State-of-the-art data loading and pre-processing libraries like DALI can be used as a drop in replacement for the default dataloaders in frameworks like PyTorch, TensorFlow, or MxNet. DALI can accelerate data pre-processing operations on Nvidia GPUs using the `NVJpeg` image decoding library, and by GPU-accelerated data augmentation operations [23, 21]. DALI also prefetches and pipelines the data fetch and pre-processing with the GPU compute, similar to the default dataloader in PyTorch.

2.4 DNN Training Scenarios

DNN training can happen in a variety of scenarios. It could be a single-GPU training on a machine, or in parallel across GPUs in a single machine, or distributed across several machines. This can happen on isolated, single-user machines that are either part of a larger cluster, or rented on the cloud [5, 6], or in shared multi-tenant clusters managed by a scheduler like YARN [203], or other custom schedulers. We provide more background on multi-tenant training in §2.4.3.

2.4.1 Single-GPU Training

Training a model using a single GPU on a server is the most straightforward training strategy. A GPU is assigned to the training job until completion and no network or communication overhead is incurred.

2.4.2 Distributed training

In many cases, models need to be trained across several GPUs as they need high computational power to train faster. In distributed training, learning is performed on multiple workers (GPUs). These workers can be either in one machine or across multiple machines. The common distributed training paradigms are data parallelism [115], and model parallelism .

Data Parallelism. Data parallel training uses the same model on each worker, but feeds it with different parts of the dataset. At regular synchronization points (typically after each iteration), the workers publish their resultant gradients to other workers (or to a parameter server from where all workers pull updates). Finally, each worker updates its own model taking into account the combined knowledge learned by all of the models. For the distributed training experiments in this dissertation, we consider data parallelism as it is the most commonly used technique. We also roll over the communication time between GPUs as the GPU compute time. Optimizing the communication time is not the focus of this dissertation; prior work has looked at this optimization [151, 212, 99, 19].

Model Parallelism. Model parallelism splits the model among the GPU workers and uses the same data for every worker. It splits the weights of the network equally among the workers and each worker computes on a single minibatch. The generated output is then synchronized after each layer to provide the input to the next layer.

2.4.3 Training in Multi-Tenant Clusters

The widespread popularity of DNNs makes training such models an important workload in both enterprises and cloud data centers. Enterprises typically setup large multi-tenant clusters, with expensive hardware accelerators like GPUs, to be shared by several users and production groups [105, 210]. In addition to the model-specific parameters and scripts, jobs specify their GPU demand before being scheduled to run on available servers. These jobs are scheduled and managed either using traditional big-data schedulers, such as Kubernetes [49] or YARN [203], or using modern schedulers that exploit DNN job characteristics for better performance and utilization [153, 209, 133, 87, 121, 166, 50]. These DNN schedulers decide how to allocate GPU resources to many jobs while implementing complex cluster-wide scheduling policies to optimize for objectives such as average job completion times (JCT), makespan, or user-level fairness.

Note that in the scope of this dissertation, we do not consider co-scheduling jobs on the same GPU concurrently; i.e., two jobs do not space share the GPU. Collocated jobs on the same machine, run on different GPUs in the machine.

Scheduling policy and mechanism. When jobs are submitted to a scheduler, a *scheduling policy* such as First In, First Out (FIFO), Shortest Remaining Time First (SRTF), or Least Attained Service (LAS) decides the next job (J) to be run on the cluster. A *scheduling mechanism* then identifies where job J should be run, and how much resources to allocate to the job. In this dissertation, we do not propose a new scheduling policy; we propose a scheduling mechanism that appropriately allocates resources, and maps jobs onto available servers.

Cluster Metrics. The efficacy of a scheduling policy is evaluated based on performance metrics such as average job completion time (JCT), makespan, and fairness metrics measured across a set of jobs in a loaded cluster.

We define a *loaded cluster* as the point in time, when the cluster load (sum of GPU demand across all runnable jobs at the moment) is higher than the available cluster resources. We now describe each cluster metric.

- **Average JCT**. JCT of a job is the time elapsed between its arrival and termination (including any queuing delays). Average JCT is the mean of JCT across the set of jobs evaluated in a loaded cluster. Lower the average JCT, better the cluster efficiency; i.e., on average, jobs complete faster. Typically this is the performance metric for a cluster with dynamic job arrivals; i.e., jobs arrive continuously.

- **Makespan**. Makespan is the time to complete execution of all the jobs in the cluster. It is the time between the arrival of the first job to the completion of the last job. Lower the makespan, better the overall cluster performance. Makespan is typically used as a performance metric for static job traces; one where all jobs arrive at the beginning of the workload.

- **Fairness**. Fairness is a metric used to ensure multiple users get a fair share of resources to run their jobs. A fair policy assigns resources to jobs such that all jobs on average get an equal share of resources over time.

GPU-proportional allocation. Users can request jobs to be scheduled on the cluster, specifying the number of GPUs required by the job, in addition to the model-specific parameters and scripts. Apart from the GPU (user-given), each DNN job requires CPU to pre-process the dataset, and memory to cache the dataset during training. Existing DNN schedulers in literature [209, 153, 133, 51, 87, 121], and those used in real-world GPU clusters [18], allocate CPU and memory resources to a job using a GPU-proportional allocation. They allocate auxiliary resources like memory and CPU, proportional to the number of GPUs allocated to the job. For instance, consider a server with 4 GPUs, 12 CPUs and 200 GB memory. The GPU-proportional allocation for 1 GPU-job is 3 CPUs and 50GB memory. If a job is allocated 2 GPUs on this server, then it is given 6 CPUs and 100GB memory.

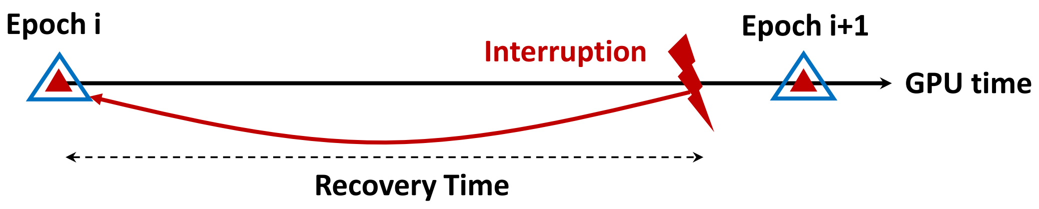


Figure 2.3: **Recovery time across models**. The amount of GPU work lost and has to be *redone* on recovery is termed the recovery time.

2.5 DNN Checkpointing

The long-running DNN training jobs could be interrupted for a variety of reasons; server crash, process failure, job migration, job or VM preemptions, etc. When interrupted, all the model state learned so far in GPU memory is wiped out, resulting in the loss of several hours of GPU work. To overcome this, the model state is typically written to persistent storage at regular intervals - typically at epoch boundaries. This process is called *checkpointing*.

2.5.1 Checkpointing strategy

Checkpointing is typically done at epoch boundaries so as to minimize complexities due to violating the data invariant of training (§2.3.3). This state is typically a few hundred MBs to a few hundred GBs in size [163]. This checkpoint can then be loaded when the training job resumes to ensure that progress is not entirely lost.

2.5.2 Recovery Time

When a DNN training job is interrupted, it rolls back to the last completed epoch that was checkpointed and persisted on disk as shown in Figure 6.1. Note that, all the GPU work performed between the last checkpoint and the point of interruption is *lost* and has to be *redone* when training resumes. The amount of GPU time lost due to an interruption is termed the *recovery time*. In other words, this is the time spent to bring the model to the same state as it was prior to the interruption.

2.6 DNN Training Frameworks

Today there exists a myriad of frameworks like PyTorch [29], TensorFlow [38], MxNet [52], CNTK [185], etc which provide clean abstractions that can be used to easily build, train, and deploy complex DNNs at scale. Each framework is built in a different manner for different purposes.

PyTorch. Facebook's PyTorch is one of the most popular frameworks known for its simplicity, ease of use, and dynamic computational graphs. Owing to its ease of use, PyTorch is a widely used framework in academia and research.

TensorFlow. Google's TensorFlow is a framework widely used by companies, and businesses develop new systems. Its reputation traces back to its robustness, scalable production and deployment support. The core of Tensorflow is built in C++ with Python bindings to simplify model creation and deployment. However, the underlying complexity of the framework can make debugging challenging.

MxNet. Apache MXNet is a computationally efficient framework used in business as well as in academia. It supports a wide range of languages like JavaScript, Python, and C++. However, unlike TensorFlow, it has a smaller open-source community.

While the above and several other open-source frameworks aid model development, debugging, and deployment, in this dissertation, we identify root causes of problems that are not specific to training frameworks; we show in our analysis that the problems we identify exist across different frameworks. However, PyTorch is the primary development framework used in this dissertation for both analysis and evaluation of the systems we introduce.

2.7 Summary

In this chapter, we presented background material necessary for this dissertation. We discussed what DNNs are, and how they are trained. We introduced the role of storage subsystem in DNN training and explained the DNN data pipeline. We then provide background on different scenarios of DNN training such as distributed training and training in multi-tenant clusters. We then described how DNNs are checkpointed, and their recovery time in the event of job interruption; Finally, we discussed various DNN training platforms available to end-user today.

Chapter 3

DS-Analyzer: Analyzing Data Stalls

Chapter 2 introduced the importance of the data pipeline in DNN training. In this chapter, we introduce the concept of data stalls in DNN training, and its categorization into fetch and prep stalls. We then try to answer the question, *Do data stalls commonly exist in DNN training, and if so, what are the fundamental reasons they occur?*

To answer this, we perform a comprehensive analysis of data stalls on several DNNs by varying a number of factors, such as the number of GPUs, GPU generation, the size of the DRAM cache, the number of CPU threads etc. We further introduce a new tool, DS-Analyzer to precisely measure data stalls in the highly concurrent DNN training jobs, as well as preform predictive what-if analysis with respect to data stalls[1].

In the rest of this chapter, we first introduce data stalls in §3.1. We next present our analysis methodology in §3.2, describe how DS-Analyzer measures data stalls in §3.3, and present our major findings in §3.4. Finally, we discuss how DS-Analyzer can be used for predictive analysis in §3.5.

[1]This Chapter is based on the work, Analyzing and Mitigating Data Stalls, published in VLDB 21 [146]

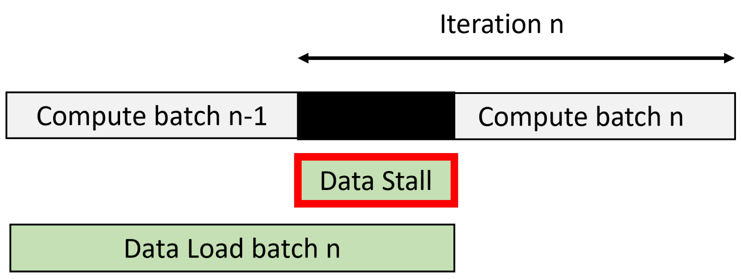


Figure 3.1: **Data Stall**. Data stalls occur when the data pipeline cannot keep up with the speed of GPU compute, leaving the GPUs stalled for the next minibatch of data

3.1 Data Stalls

Training a DNN model to its target accuracy involves the following steps in each iteration of an epoch: 1) A minibatch of data items is fetched from storage. 2) The data items are pre-processed: *e.g.,* in image classification, data items are decompressed, and then randomly cropped, resized, and flipped. 3) The minibatch is then processed at the GPU to obtain the model's prediction in a forward pass. 4) A loss function is used to determine how much the prediction deviates from the right answer; both weight and activation gradients are computed across the different layers of the DNN. 5) Weights in the model's layers are updated using gradients computed in the backward pass.

Ideally, most of the time in each epoch should be spent on Steps 3–5 (which we collectively term the *GPU compute* time), i.e., training is *GPU bound*. When performing multi-GPU training, individual GPUs (workers) exchange weight gradients with other workers before performing weight update. In this dissertation work, we roll the communication time for gradient exchange during multi-GPU training into computation time.

As shown in Figure 3.1, if the GPU is waiting for Steps 1–2 (data load) to complete, we term it a *data stall*. Specifically, if training is blocked on *fetch*, we call it a fetch stall; the training is *I/O bound* in this case. Training blocked due to *prep* is termed prep stall; this causes the training to be *CPU bound*. Data stalls cause the GPU to be idle, and must be minimized to increase GPU utilization.

The rate at which data items can be fetched from storage depends primarily on the storage media. The rate at which data items can be pre-processed depends upon the pre-processing operations and the number of CPU cores available for pre-processing.

3.1.1 Example of data stalls in ResNet18 training

Let us examine the data pipeline for the ResNet18 model. Figure 3.2 shows the data fetching and pre-processing pipeline for ResNet18, along with the throughput of various components in the pipeline. This experiment is run on a machine with eight V100 GPUs, and 24 CPU cores, a typical configuration for training machine-learning models. The raw data can be fetched from hard drives at 15 MB/s or from solid state drives at 530 MB/s. If we assume that 35% of the dataset is cached in DRAM, then the effective throughput from the storage stack (assuming 35% of dataset fetched at memory bandwidth, and 65% fetched at disk bandwidth) is 802 MB/s. Pre-processing with 24 CPUs provides an overall throughput of 735 MB/s using DALI (or 1062MB/s if some

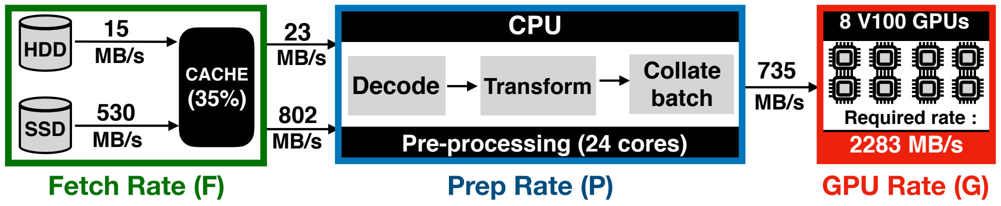


Figure 3.2: **Data Pipeline for ResNet18**. This figure shows the data pipeline with DALI for the ResNet18 model along with the throughput of each component in the pipeline. On a server with 8 `V100` GPUs and 24 physical CPU cores, the overall throughput of the data pipeline is lower than the expected ingestion rate at the GPU, resulting in data stalls.

pre-processing is offloaded to the GPU), far short of the 2283 MB/s required by the GPUs. As a result, the GPUs stall waiting for data to be fetched and pre-processed.

In general, if we prefetch data at rate F, pre-process it at rate P and perform GPU computation on it at rate G, then data stalls appear if $G > min(F, P)$, i.e., GPU processes data at a rate faster than it can be prefetched or pre-processed. The fetch and prep stalls reported in this work are unmasked stall time; i.e., the stall time that shows up in the critical path, inspite of being pipelined with compute. From now on, we call data prefetching simply *fetch*, and pre-processing *prep*.

3.2 Methodology

We now describe our analysis methodology. We describe the models and datasets used in our analysis, our training environment, parameters used for the models, and metrics we evaluate the models on.

Task	Model	Dataset (Size)
Image Classification	Shufflenetv2 [219]	
	AlexNet [117]	ImageNet-22k [17]
	Resnet18 [91]	(1.3TB)
	SqueezeNet [100]	OpenImages-Extended
	MobileNetv2 [181]	[119, 182] (645GB)
	ResNet50 [91]	Imagenet-1k [180]
	VGG11 [189]	(146GB)
Obj Detection	SSD+Res18 [130]	OpenImages [119] (561GB)
Audio Classify	M5 [62]	Free Music [66] (950GB)

Table 3.1: **Models and datasets used in this analysis**.

3.2.1 Models and Datasets

We analyze **nine** state-of-the-art DNN models across three different tasks and four different datasets as shown in Table 3.1. This section focuses on the smaller ImageNet-1K dataset for image classification models. Evaluation with large datasets like ImageNet-22k and OpenImages is presented in Chapter 4 (§4.3). The image and audio classification models are taken from TorchVision [34] and TorchAudio [33] respectively; for object detection, we use NVIDIA's official release of SSD300 v1.1 [24]. For all DNNs, we use the same pre-processing as in the original papers. Additionally, we evaluated data stalls on two language models; Bert-Large [69] on Wikipedia & BookCorpus dataset [222] for language modeling and GNMT [208] on WMT16 [37] (EN-De) dataset for translation. These models are GPU compute heavy and do not exhibit data stalls in our training environment (hence, results excluded from analysis). But, if compact representations for these models with lower

	GPU Config	GPU Mem(GB)	Storage Media	Rand Read (MBps)
`SSD-V100`	8xV100	32	SSD	530
`HDD-1080Ti`	8x1080Ti	11	HDD	15 – 50

Table 3.2: **Server configurations used in this analysis**. We use two representative ML optimized server SKUs; each server has 24 CPU cores, 500GiB DRAM, and 8 GPUs

computational requirements show up, or if compute speed increases due to newer generations of GPUs, data stalls may show up in these models as well.

3.2.2 Training environment

All experiments are performed on PyTorch 1.1.0 using the state-of-the-art NVIDIA data loading pipeline, DALI. We have empirically verified that DALI's performance is strictly better than PyTorch's default data loader. We use two distinct server configurations for our analysis as shown in Table 3.2. `Config-SSD-V100` has configuration closest to AWS p3.16xlarge [6] with gp2 storage [13], while `Config-HDD-1080Ti` is closest to AWS p2.8xlarge [5] with st1 storage [13]. Both our servers have 500GB DRAM, 24 physical CPU cores, and 8 GPUs per server. Both these server types are a part of internal clusters at a large cloud provider; they resemble publicly available cloud GPU SKUs [6, 5] as well as publicly available information on typical production cluster SKUs [18, 106].

3.2.3 Training parameters

For experiments on `Config-SSD-V100`, we use a batch size of 512 per GPU for all image classification models, 128 per GPU for SSD-Res18, 16 per GPU for M5 and perform weak scaling for distributed training (while ensuring that the global batch size is consistent with those widely used in the ML community). Since V100 GPUs have tensor cores, we use Apex mixed precision training with LARC (Layer-wise Adaptive Rate Clipping), and state-of-the art learning rate warmup schedules [82]. On `Config-HDD-1080Ti`, we use the maximum batch size that fits the GPU memory (less than 256 for all models) and perform full-precision training.

3.2.4 Training metrics

We run all the experiments presented here for three epochs, and report the average epoch time (or throughput in samples per second), ignoring the first epoch. Since we start with a cold cache in our experiments, first epoch is used for warmup. Measuring data stall time does not require training to accuracy; per-epoch time remains stable.

3.3 Measuring data stalls using DS-Analyzer

We develop a standalone tool, *DS-Analyzer* that profiles data stalls in DNN training. Frameworks like PyTorch and TensorFlow provide an approximate time spent on data loading and pre-processing per minibatch, by simply placing timers in the training script. This is insufficient and inaccurate for two reasons. First, this technique cannot accurately provide the split up of time spent in data fetch (from disk or cache) and pre-processing operations. To understand if the training is bottlnecked on I/O or CPU, it is important to know this split. Second, frameworks like PyTorch and libraries like DALI use several concurrent processes (or threads) to fetch and pre-process data; for a multi-GPU data parallel training job, a data stall in one of the data loading processes may reflect as GPU compute time for the other processes, because all GPU processes wait to synchronize weight updates at batch boundaries. Naively adding timers around data path does not provide accurate timing information. Therefore, DS-Analyzer uses a differential approach. DS-Analyzer runs in three phases;

1. **Measure ingestion rate**. First, DS-Analyzer pre-populates synthetic data at the GPUs and runs the job for a fixed number of epochs. This identifies the max data ingestion rate at the GPUs, with no fetch or prep stalls.
2. **Measure prep stalls**. Next, DS-Analyzer executes the training script with the given dataset by ensuring that the subset of data used is cached in memory, using all available CPU cores, and estimates the training speed. Since this run eliminates fetch stalls, any drop in throughput compared to (1) is due to prep stalls.
3. **Measure fetch stalls**. Finally, DS-Analyzer runs the training script by clearing all caches, and setting maximum cache size to a user-given limit, to account for fetch stalls. The difference between (2) and (3) is

the impact of fetch stalls.

Additionally, DS-Analyzer collects low level metrics such as the throughput of the storage device, memory and network bandwidth, cache size, and memory utilization.

In all experiments presented in §3.4, we use DS-Analyzer to run the training script for each model for a total of 3 epochs. To accurately measure fetch stalls, we consider the first epoch as warmup (as we start from a cold cache), and report the average metrics (§3.2) of the two subsequent epochs.

3.4 Data Stalls in DNN Training

Our analysis aims to answer the following important questions:

Fetch Stalls (Remote)	Is remote storage a bottleneck for training?	§3.4.1
Fetch Stalls (Local)	When does the local storage device (SSD/HDD) become a bottleneck for DNN training?	§3.4.2
Prep Stalls	When does data prep at the CPU become a bottleneck for DNN training?	§3.4.3
Generality	Do fetch and prep stalls exist in other training platforms like TensorFlow?	§3.4.4

3.4.1 When dataset resides on remote storage

Datasets used for training DNNs could reside locally on the persistent storage of a server, or on shared remote storage such as distributed file systems (HDFS, GlusterFS - GFS), or object stores (S3, Azure blobs). We analyze the impact of two kinds of remote backends; a distributed file system, GlusterFS

(GFS) and the Azure blob object store accessed via blobfuse.

When data resides remotely, the first epoch of training fetches data over the network and stores it locally for subsequent use. Cluster file systems like GFS use the OS Page Cache to speed up subsequent accesses. Blobfuse downloads the dataset on to local SSD, and mimics local training from the second epoch. Figure 3.3 compares the epoch time for ResNet18 on `Config-SSD-V100` using GFS, blobfuse, and local SSD for the first epoch with cold cache, and a stable-state epoch with warmed up cache.

The data stall overhead of BlobFuse is especially high in the first epoch when it downloads the entire dataset to local storage, and can result in 20× higher training time as compared to GFS due to blocking IO. Unsurprisingly, during the steady state epochs, data stall overheads when using the local SSD and BlobFuse are similar (as the blob data is cached on the local SSD); GFS results in more data stalls as it validates metadata of cached data items over the network every time a data item is accessed. Blobfuse does not incur any network cost beyond first epoch, if the dataset fits on local SSD.

As shown in Figure 3.4, for the ImageNet1K dataset, for BlobFuse, the cost of downloading the entire dataset in the first epoch is amortized as we train for a longer number of epochs, making the remote blob store a better fit compared to GFS when models are trained to accuracy for over 60 epochs.

Although datasets are growing in size, large datasets that are publicly available fit entirely on local storage (but not in memory) [36, 39, 119, 66, 17,

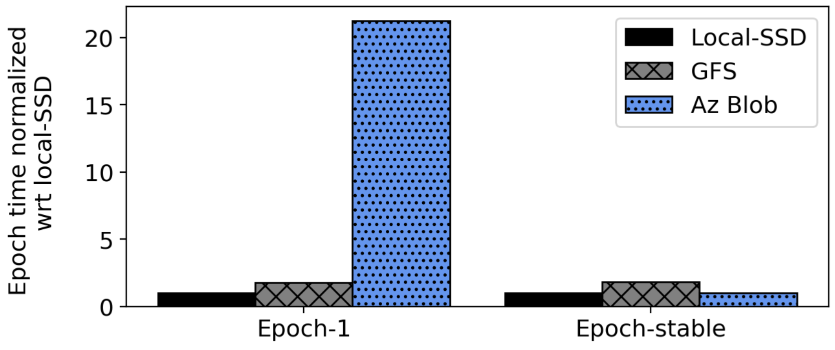


Figure 3.3: **Fetch cost with remote stores**. The initial fetch cost is high with blobs, but it provides local-disk performance for subsequent epochs; GFS on the other hand has higher than local-disk fetch cost in stable state.

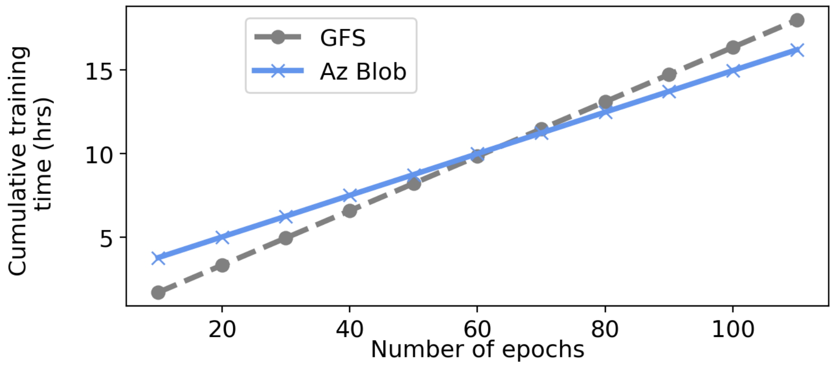


Figure 3.4: **Training with remote stores**. The high download cost of blob is amortized over training for a large number of epochs

180]. Therefore, a common training scenario is to pay a one-time download cost for the dataset, and reap benefits of local-SSD accesses thereafter (default and recommended mode in the Microsoft clusters). Therefore, in the rest of the work, we analyze fetch stalls in scenarios where dataset is present locally on a server, but is not entirely cached in memory.

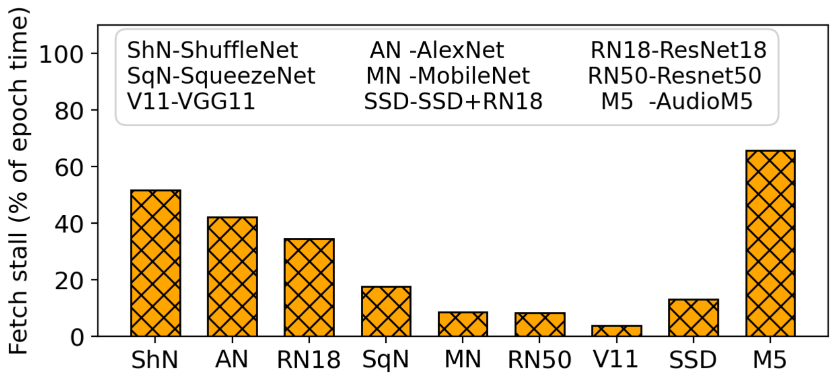


Figure 3.5: **Fetch stalls**. Several DNNs experience significant stalls waiting for I/O, when training on `Config-SSD-V100` with 35% of their dataset cached.

3.4.2 When datasets cannot be fully cached

Datasets used for training DNNs are growing in size [36, 39, 119]. Even the ML-optimized cloud servers with 500GB DRAM can only cache 35% of ImageNet-22K, or 45% of the FMA dataset, or 65% of the OpenImages dataset. Popular datasets like ImageNet-1K cannot be fully cached on commonly used cloud SKUs like AWS `p3.2xlarge`, which has 61 GiB DRAM. When datasets don't fit in memory, and the fetch rate(F) < compute rate $(min(P, G))$, fetch stalls occur.

Fetch stalls are common if the dataset is not fully cached in memory. Figure 3.5 shows the percentage of per epoch time spent on I/O for nine different DNNs when 35% of their respective datasets can be cached in memory on `Config-SSD-V100`. DNNs spend 10 –70% of their epoch time on blocking I/O, despite pipelining and prefetching, simply because the compute rate is higher than fetch rate.

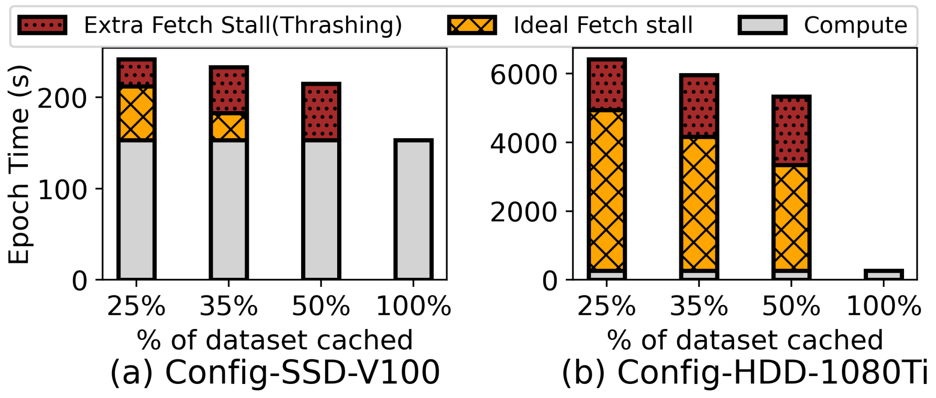


Figure 3.6: **ResNet18 with varying cache**. This stacked bar chart splits epoch time into time spent in compute, ideal fetch stalls, and the additional fetch stall due to thrashing.

OS Page Cache is inefficient for DNN training. DNN training platforms like PyTorch, TensorFlow and libraries like DALI, rely on the operating system's Page Cache to cache raw training data in memory. Unfortunately, the OS Page Cache leads to thrashing as it is not efficient for DNN training. If 35% of the data can be cached, then an effective cache should provide 35% hits; instead, the Page Cache provides a lower hit rate. For a 146 GiB data set, each epoch should see only 65% of the dataset, or 95GiB, fetched from storage. Instead, we observe 85% of the dataset fetched from storage every epoch; the 20% difference is due to thrashing. Figure 3.6 shows the fetch stalls, including those due to thrashing, when using PyTorch with DALI. An effective cache for DNN training must eliminate thrashing to reduce fetch stalls to the minimum shown in Figure 3.6.

Lack of coordination among caches leads to redundant I/O in dis-

# Jobs	1x8GPU	2x4GPU	4x2GPU	8x1GPU
Storage I/O (GB)	125	258	467	884
Read amplification	1×	2×	3.7×	7×

Table 3.3: **Storage I/O in HP search**. This table shows the redundant storage IO incurred during HP search.

tributed training. In distributed training jobs, the data to be fetched and processed is divided randomly among servers. The division is random and changes every epoch. As a result, each server often has to fetch data from storage every epoch; this is done even if the required data item is cached in an another server that is a part of the distributed training job. This lack of coordination among caches makes distributed training storage I/O-bound. When training Resnet50 on ImageNet-1K (146GiB) across two servers having a total cache size of 150GiB, each server fetches 45GiB from storage in each epoch (despite the fact that the other server might have this data item in its cache). On `Config-HDD-1080Ti`, this leaves ResNet50 stalled on I/O for 75% of its epoch time.

Lack of coordination in HP search results in redundant I/O. HP search is performed by launching several parallel jobs with different HP on all available GPUs in a server [125]. All HP jobs access the same dataset in a random order in each epoch, resulting in cache thrashing and read amplification at the storage device. Table 3.3 shows the total storage I/O for different configurations of HP search jobs on `Config-SSD-V100`. When 8 single-GPU jobs are run in a server (35% cache), there is 7× read amplification per epoch

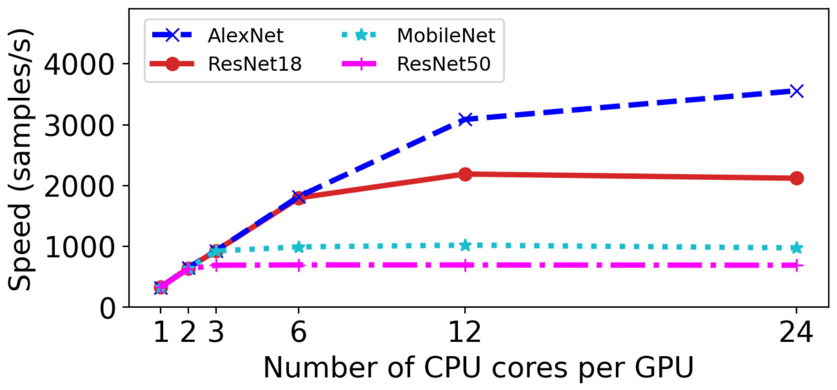


Figure 3.7: **Impact of CPU cores on training**. DNNs need between 3 – 24 cores per GPU to mask prep stalls.

(884 GiB read off storage compared to 125 GiB for one job), which slows down HP search on ResNet18 by 2× on `Config-SSD-V100`.

3.4.3 When datasets fit in memory

We now analyze the impact of CPU pre-processing on DNN training in the scenario where the entire dataset is cached in memory, thus eliminating fetch stalls due to storage I/O.

DNNs need 3–24 CPU cores per GPU for pre-processing. Figure 3.7 shows how DNN training throughput changes as we vary the number of CPU pre-processing threads (per V100 GPU) for four models. For computationally complex models like ResNet50, 3 – 4 CPU cores per GPU is enough to prevent prep stalls; for computationally lighter models like ResNet18 or AlexNet, as many as 12 – 24 CPUs per GPU are needed to mask prep stalls. Since prep is CPU-intensive, using more threads (vCPUs) than the number of physical

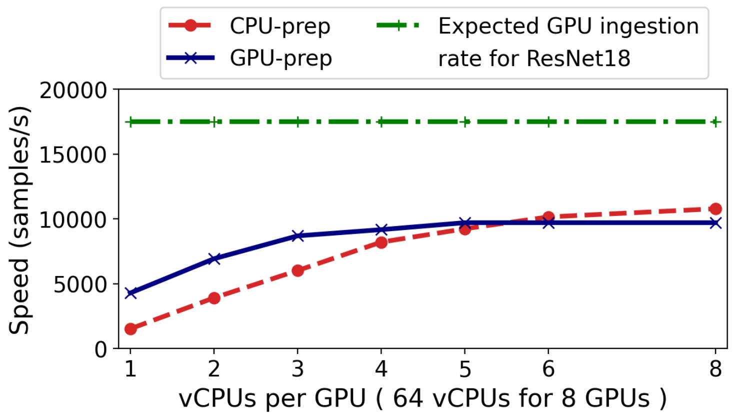


Figure 3.8: **Impact of CPU on data prep**. This graph plots the epoch time for ResNet18 on a server with 64 vCPUs and 8 V100 GPUs as we vary the number of vCPUs per GPU. ResNet18 incurs about 37% prep stalls. With the GPU-prep of DALI, we do not increase threads beyond 5 per GPU as it results in GPU OOM.

CPU cores does not help much; For a 8-GPU V100 server with 32 CPU cores (64 vCPUs), ResNet18 spends 37% of the epoch time on prep stalls as shown in Figure 3.8. Note that this is the same server configuration as that of AWS P3 (p3.16xlarge) [6]. Even on NVIDIA's AI-optimized DGX-2, there are only three CPU cores per GPU; many models will have prep stalls on the DGX-2.

DALI is able to reduce, but not eliminate prep stalls. DALI uses the GPU for pre-processing operations, and is thus able to reduce prep stalls, as shown in Figure 3.9 (a). The effectiveness of DALI depends on the GPU speed; for example, on the slower 1080Ti, DALI is able to eliminate prep stalls using three CPU threads and the GPU. On the faster V100 though, DALI still results in 50% prep stalls when using three CPU threads and the GPU. Figure 3.10 shows that our observations hold across different DNNs when training with

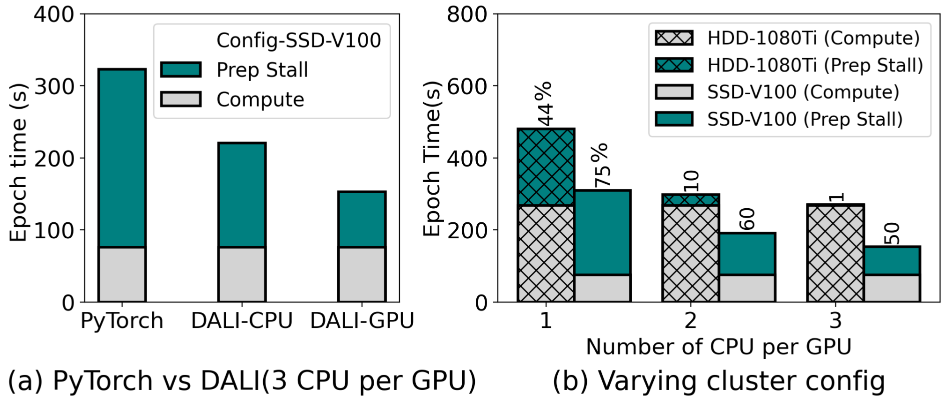


Figure 3.9: **8-GPU ResNet18 training**. Even with DALI, faster GPUs like V100 have upto 50% prep stalls.

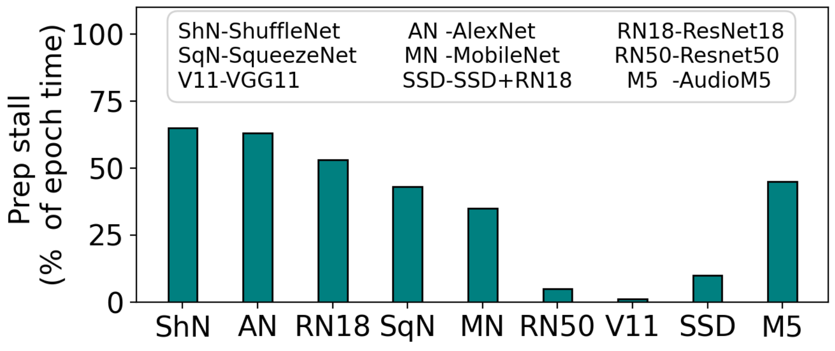


Figure 3.10: **Prep stall across DNNs**. This graph plots prep stall as a percentage of the epoch time, when training various DNNs across 8-GPUs on `Config-SSD-V100`. DNNs spend $5-65\%$ of their epoch time on blocking prep.

eight GPUs each with 3 CPUs.

Decoding is very expensive! To understand what makes prep expensive, we run a microbenchmark to breakdown the cost of different stages in a typical prep pipeline for image classification tasks. An input image is first loaded into memory, decoded, and then randomly transformed (crop, flip), and finally

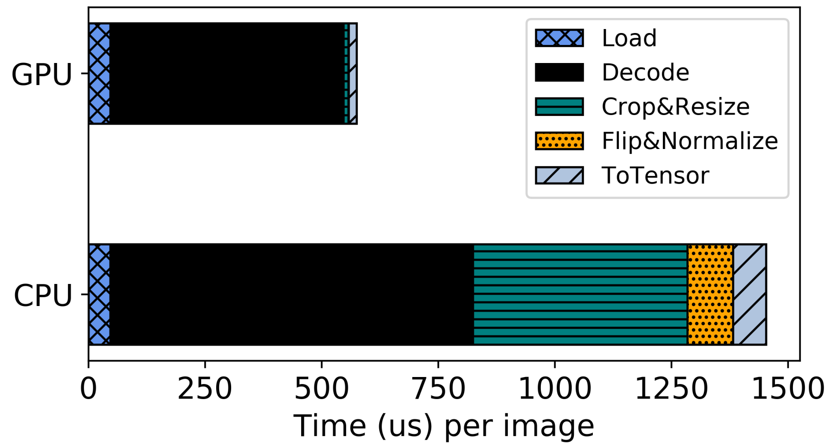


Figure 3.11: **Breakdown of prep overhead**. This graph shows the split up of time spent on each phase of prep for a image classification workload on ImageNet-1k. GPU-based prep is significantly faster than CPU-based, at the expense of additional GPU memory usage. Decoding images takes the most time in the pipeline.

copied over to the GPU as a tensor that can be processed. Fig 3.11 shows the time taken for each operation when prep is done on CPU and GPU by DALI. In the GPU-based prep, decode operation is partly done on CPU and then offloaded to GPU. All subsequent operations are performed at the GPU. There are two main takeaways from this graph. First, we see that offloading prep to the GPU provides significant speedup at the expense of GPU memory usage (+5GB!). This increased memory consumption adversely affects the maximum batch size that a model can use for training. Furthermore, for models that are already GPU-intensive, offloading the prep pipeline interferes with training and results in reduced throughput (ResNet50, VGG). Second, a majority of time during prep is spent in decoding images.

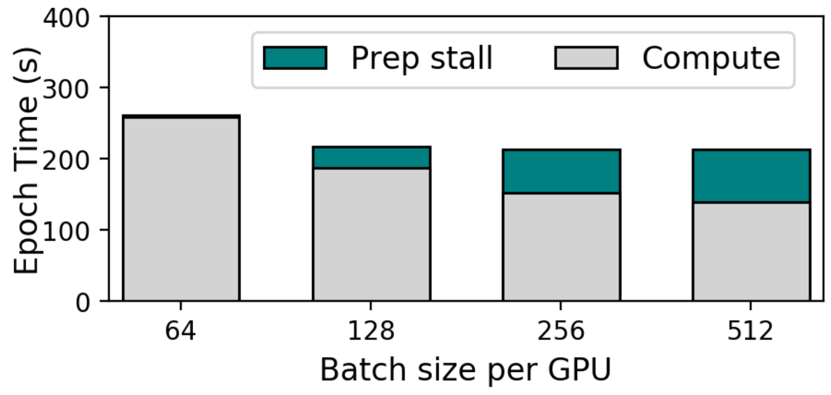


Figure 3.12: **Impact of batch size on prep**. This graph plots the epoch time for MobileNetv2 on `Config-SSD-V100` with 8-GPUs as we vary the per-GPU batch size. Epoch time does not improve because training is bottlenecked by data prep.

Impact of batch size. The impact of batch size on GPU computational efficiency is well studied [191, 95]; larger batch sizes utilize the massive GPU parallelism better, and also reduce the number of weight updates (inter-GPU communication) per epoch, resulting in faster training. Figure 3.12 shows the impact of varying the batch size on epoch time and the percentage of epoch time spent on prep stalls for MobileNetv2. As computational efficiency increases with larger batches, training becomes CPU bound due to data prep. Note that, although the required GPU compute time dropped with a larger batch size, per epoch time remained same due to prep stalls. This graph makes an important point; as compute gets faster (either due to large batch sizes, or the GPU getting faster), data stalls squander the benefits due to fast compute.

Redundant pre-processing in HP search results in high prep stalls.

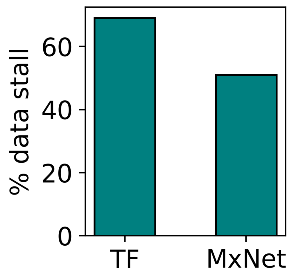
	Speed (samples/s)						
Config	ShN	AN	RN18	SqN	MN	RN50	V11
3-CPU	1731	1702	1242	927	824	726	778
24-CPU	3387	4437	2283	1461	1157	782	801

Table 3.4: **HP search with 8 jobs**. (ShN-ShuffleNet, AN-AlexNet, RN18-Resnet18, SqN-SqueezeNet, MN-MobileNet, RN50-ResNet50, V11-VGG11)

During HP search, concurrent jobs process the same data. Currently, there is no coordination; if there are 8 HP jobs, the same data item is processed eight times. This is made worse by the fact that all HP jobs share the same set of CPU threads, leading to fewer CPU threads per GPU, and higher prep stalls. When 8 single-GPU ResNet18 HP jobs run on `Config-SSD-V100`, each job gets 3 CPU for prep and incurs a 50% prep stall as shown in Figure 3.10. Coordinating these HP search jobs on a single server can potentially eliminate prep stalls, as all available CPU (24 cores) can be used to prep the dataset exactly once per epoch and reused across jobs. Table 3.4 shows the difference in overall training throughput for different models when using 3 and 24 CPU per GPU respectively.

3.4.4 Data stalls exist across training frameworks

To generalize our findings on data stalls across different training platforms and data formats, we analyze the prep and fetch stalls in TensorFlow using the binary TFRecord format. Unlike PyTorch, TensorFlow does not store training data as small individual raw files. Instead, it shuffles the small



(a) 100% Cache

% cached (Sz:146GB)	8-GPU job Cache Miss	8-job HP I/O (GB)	8-job HP Read amp
50%	91%	860	6.14×
35%	94%	1010	7.21×
25%	97%	1019	7.28×

(b) Varying cache (TensorFlow)

Figure 3.13: **Data stalls across frameworks**. The plot shows that data stalls exist in other training frameworks like MxNet and TensorFlow in various training scenarios.

random files, serializes them, and stores them as a set of files (100-200MB each) called TFRecords. TFRecords make reads more sequential. Training platforms like MXNet also use a similar serializing technique for data called RecordIO [137].

Figure 3.13a shows that both native TF and MxNet spend 65% and 50% of the epoch time on prep stall for a 8-GPU ResNet18 training job when the dataset is entirely cached in memory. Next, Table 3.13b shows the percentage of misses in the Page Cache for a 8-GPU training job and the I/O amplification due to lack of coordination in HP search for varying cache sizes in TensorFlow. Similar to PyTorch, TF can also use DALI's GPU based pre-processing and exhibit prep stalls similar to PyTorch. TFRecord format results in 40% higher cache misses than the ideal because, the sequential access nature of TFRecords (and RecordIO) is at odds with LRU cache replacement policy of the Page Cache, resulting in a pathological case for LRU. The lack of co-ordination in

Finding	***Insights***
OS Page Cache is inefficient for DNN training due to thrashing	DNN-aware caching can eliminate thrashing across epochs
DNNs need anywhere between 3 – 24 CPU cores *per GPU* for data pre-processing	If hardware is upgraded to overcome workload bottlenecks, it must be done carefully with an eye towards designing balanced server SKUs.
DNNs spend upto 65% of the epoch time in data pre-processing, primarily on redundant decoding	Decoded data can be cached (as opposed to caching encoded data), if space amplification due to decoding can be addressed
Lack of coordination among local caches lead to redundant I/O in distributed training across servers	To overcome local storage I/O bottlenecks, local in-memory caches of servers allocated to a job can be coordinated to fetch data from distributed in-memory caches
Hyperparameter search workloads perform redundant I/O & prep	Hyperparameter search jobs must coordinate data fetch & prep to mitigate data stalls

Table 3.5: **Key findings and implications of our analysis of data stalls**.

HP jobs results in upto 7.2× read amplification; although all jobs read the same 140 GiB dataset, the total disk I/O was 1.1 TB.

3.4.5 Analysis summary

Table 3.5 summarizes our key findings pertaining to data stalls across DNN training frameworks, models, and hardware configurations. Our analysis also highlights that data stalls are a consistent problem across different training platforms.

3.5 DS-Analyzer: Predictive analysis

While all the experiments that vary cache size, and the number of CPU cores in §3.4 are run on physical servers, we extend DS-Analyzer to help a user simulate these experiments without having to run all different configurations on physical servers. While there exists prior work that predict the performance of a DNN, they focus on profiling the layer-wise performance of DNN [9, 27], low level perf counters for accelerators [25, 101], or finding optimization opportunities at the neural network layer level [220]. In contrast, DS-Analyzer analyzes the implication of CPU, memory, and storage on the performance of a DNN and answers what-if questions.

This is a powerful means of analyzing whether throwing more hardware at the problem will solve the issue of data stalls. For instance, if training is dominated by fetch stalls (bottlenecked on disk bandwidth), then increasing the number of CPU cores on the machine has no benefit; either DRAM capacity has to be increased, or the disk must be replaced with a higher bandwidth one. Similarly, if the training job is bottlenecked on prep, then increasing DRAM has no effect on training time. DS-Analyzer is useful in scenarios like this, to predict the performance of a model as we scale up CPU, memory, or storage. However, DS-Analyzer needs to be run atleast once for every new GPU and DNN architecture to collect the required information, and predicts the impact of varying auxiliary resources like CPU, memory, or storage on data stalls.

Estimating data stalls. Consider the different components involved in a typical DNN data pipeline as in Figure 3.2; data is fetched from cache (and store) with an effective prefetch rate F, pre-processed at the CPU at a rate P and processed at the GPU at a rate G. To perform predictive analysis, DS-Analyzer measures several metrics related to the data pipeline of the model; the maximum ingestion rate at the GPU (G), the rate of CPU prep (P), the rate of cache fetch (C), and the rate of storage fetch (S), by training for a fixed number of iterations (default:100) using the differential technique discussed in §3.3. Using these metrics, DS-Analyzer models the training iteration to answer what-if questions such as, how much DRAM cache is required for this model to eliminate fetch stalls?

DS-Analyzer collects these metrics for a model as follows.

(i) Measure ingestion rate (G). To find the maximum possible speed at which the DNN can train, DS-Analyzer first runs the job script for a fixed number iterations (default:100) with synthetic data that is pre-populated at the GPUs. It then calculates G as,

$$G = \frac{\text{Total samples processed in (i)}}{\text{Time for (i)}} \tag{3.1}$$

$$\text{Samples processed} = \#\text{iterations} \times \text{global batch size} \tag{3.2}$$

(ii) Measure prep rate (P). Next, DS-Analyzer executes the training script with the given dataset by ensuring that the subset of data used is cached in memory, using all available CPU cores. Additionally, the GPU computation is disabled to only run the data loader. This is required because, if $P \geq G$,

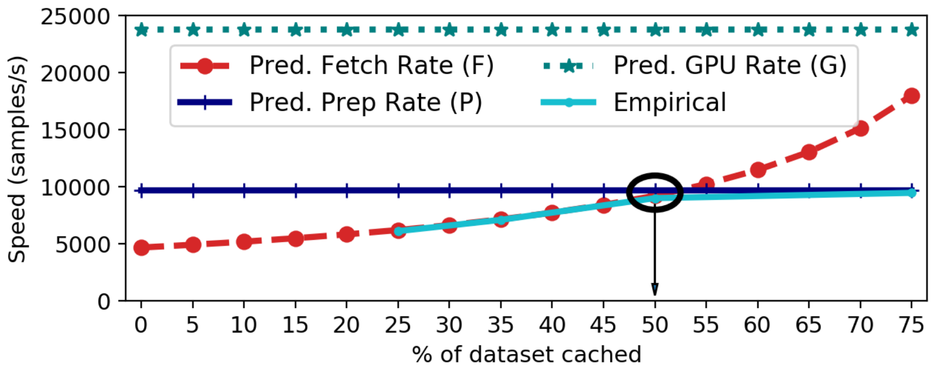


Figure 3.14: **Estimating optimal cache size with DS-Analyzer**.

then we cannot measure P using the knowledge of runs (i) and (ii), as prep will be pipelined with GPU compute. Therefore, DS-Analyzer disables GPU computation and estimates P in the same way as Eq (3.1).

(iii) Measure storage fetch rate (S). Storage fetch rate is the maximum random read throughput of the storage device. To measure this, DS-Analyzer runs the data loader (with a cold cache, disabling both pre-processing and GPU compute), with all CPU cores.

(iv) Measure cache fetch rate (C). To measure the rate at which data can be fetched from cache, DS-Analyzer uses a microbenchmark to measure memory bandwidth and uses it as an approximation for C. Note that run (ii) actually includes the time to fetch cached items as well; however we see that the cache fetch rate is very high (few tens of GBps), and does not add noise to the measurement of prep rate.

3.5.1 Example : Predicting optimal cache size

We now describe an example of what-if analysis with DS-Analyzer. We show how DS-Analyzer answers the question : *how much DRAM cache does the DNN need to eliminate fetch stalls?* To predict the implication of cache size, DS-Analyzer calculates the effective prefectch rate (F) for a given cache size (x % of the dataset). Here, we assume that the cache implements an efficient policy like MinIO; i.e., a cache of size x items has atleast x hits per epoch. F is computed as follows. Say the size of the dataset is D samples, and cache is x% of the dataset. Therefore, in an epoch, the total time to read the dataset is given by

$$T_f = \frac{D \times x}{C} + \frac{D \times (1 - x)}{S} \tag{3.3}$$

The fetch rate is then calculated as,

$$F = \frac{D}{T_f} = \frac{D}{\frac{D \times x}{C} + \frac{D \times (1-x)}{S}} \tag{3.4}$$

Since $C >> S$, $F \propto \frac{1}{1-x}$, i.e, the effective fetch rate increases, as the number of uncached items per epoch decreases. Since DS-Analyzer has already estimated values of D, C, and S, given a cache percentage x, DS-Analyzer can predict the fetch rate using Eq (3.4).

To evaluate how accurately DS-Analyzer can answer this question, we run the actual experiment by varying cache size on a physical server (empirical), and comparing it to the predictions of DS-Analyzer for AlexNet on `Config-SSD-V100` with Imagenet-1K. Figure 3.14 plots the predicted values of F, P, and G, alongside empirical speed while varying cache size. First, we observe that the predicted training speed ($min(F, P, G)$) is a maximum of 4% off the empirical results. Second, using these predictions, DS-Analyzer can estimate the optimal cache size for the model by comparing it with prep rate (P) and GPU ingestion rate (G). To eliminate fetch stalls, $F > min(P, G)$ as shown by the intersection in Figure 3.14. At lower cache sizes, training is I/O bound, however, a cache that is 50% of the dataset size is sufficient to eliminate fetch stalls; larger cache (more DRAM) is not beneficial beyond this point, as training becomes CPU-bound. A comprehensive list of data pipeline rates (G, P, F) for several models, datasets, and configurations is in the Appendix of the extended version of our paper on analyzing data stalls [145].

3.6 Limitations and Discussion

In the scope of our data stall analysis, we do not tease out the overhead due to communication (network) between GPUs in a machine, or across machines. We instead roll this over as GPU compute time. Our analysis environment had high-bandwidth network interconnects; this eliminated any possible network overheads. However, in scenarios where network bandwidth may be a bottleneck, a future direction is to analyze *network stalls*.

3.7 Summary

In summary, in this chapter, we presented a comprehensive analysis of data stalls in DNN training. We built a tool DS-Analyzer, using which we measured data stalls by varying factors like amount of memory allocated to the job, number of CPU cores used for pre-processing, GPU generation .

Our analysis yields several interesting insights. First, a large number of DNN models have data stalls. Second, these data stalls occur across frameworks such as PyTorch and TensorFlow. Third, there is a large amount of redundant work done by the data pipeline during HP search and distributed training where the same data items are fetched and pre-processed by multiple jobs or multiple servers. Finally, when the dataset is larger than available memory, current caching policies used by DNN training frameworks are inefficient, resulting in high disk I/O with unwanted evictions in the OS Page Cache.

Our study has directly guided the design of systems that mitigate data stalls. We discuss these in detail in the subsequent chapters. We also believe our study opens doors to several interesting research directions with respect to mitigating data stalls, which we discuss in Chapter 8.

Chapter 4

CoorDL: Mitigating Data Stalls in Single-User Scenarios

In the previous chapter, we presented an analysis of data stalls in DNN training and identified why data stalls occur. Our analysis reveals three key insights with respect to the causes for data stalls when models are trained in isolation. First, our analysis corroborates that relying on OS abstractions (like Page Cache) is inefficient for DNN workloads. Second, lack of coordination among caches leads to redundant I/O in distributed training. Finally, lack of coordination in HP search results in redundant I/O and data prep among concurrently running jobs.

In this chapter, we exploit these insights to design and build CoorDL, a new data loading library that accelerates DNN training by minimizing data stalls[1]. In the rest of this chapter, we first present the overall architecture of CoorDL (§4.2). We then introduce the three techniques that CoorDL introduces to mitigate data stalls; the new MinIO software cache that is specialized for DNN training (§4.2.2), a *partitioned caching* technique to co-ordinate local

[1]This Chapter is based on the work, Analyzing and Mitigating Data Stalls, published in VLDB 21 [146]

MinIO caches in distributed training (§4.2.3), and *coordinated prep*, which coordinates data fetch and pre-processing among concurrent HP search. Fnally, we evaluate CoorDL on hyperparameter tuning, single-server, and multi-server distributed training scenarios (§4.3).

4.1 Motivation

Consider a cluster of ML-optimized cloud servers with V100 GPUs and 500 GiB of memory [6]; 400GiB is allocated to cache the input dataset. We would like to train ResNet50 [91] using the 645 GiB OpenImages [119, 182] dataset in PyTorch with DALI. When we perform HP Search for this model with eight jobs on a single server, a staggering 1.7 TiB of data (2.8$\times$ the size of the entire dataset) is fetched from storage during *each* epoch because the data pipeline of each of the eight jobs fetches and pre-processes the dataset independently. After determining the hyperparameters, when we perform distributed training on 16 GPUs across two servers, in each epoch of training both servers process a random disjoint half of the dataset (so that they collectively process the entire dataset once per epoch). Despite enough memory across two servers (800 GiB) to cache the entire dataset, each server fetches 119 GiB (from storage) per epoch when training, as the random data items being requested may not be cached locally at each server. If the server uses hard drives for storage, training incurs fetch stalls. If we instead train the model on a single server, then 65% of the dataset can be cached in the OS Page Cache. So, we expect 35% of the dataset to be accessed from storage

Finding	***CoorDL Insights***
OS Page Cache is inefficient for DNN training due to thrashing	Optimize DNN cache to eliminate thrashing across epochs (*MinIO* §4.2.2)
Lack of coordination among local caches lead to redundant I/O in distributed training across servers	Local caches of servers can be coordinated to fetch data from the remote cache to overcome storage I/O bottlenecks (*Partitioned Cache* §4.2.3)
No coordination in HP search leads to redundant I/O & prep	HP search jobs must coordinate data fetch & prep (*Coordinated Prep* §4.2.4)

Table 4.1: **Key findings and implications of data stall analysis**.

in every subsequent epoch; however, we noticed 53 – 87% of dataset accesses per epoch in two different data access modes of DALI. Such increased disk IO results in fetch stalls in training.

Table 4.1 summarizes these key findings pertaining to data stalls and the insight we exploit to design a corresponding technique to mitigate it in CoorDL.

CoorDL introduces three techniques to overcome data stall overheads. First, CoorDL introduces the novel MinIO software cache that is specialized for DNN training. MinIO exploits the unique data access pattern in DNN training to minimize the amount of data fetched from storage for training on a single server. Next, CoorDL introduces *partitioned caching*, where the dataset is partitioned and cached among the servers involved in distributed training for each job. On a local MinIO cache miss, data is fetched from the memory of a remote server (over the commodity TCP stack) rather than from local storage. The dataset is thus fetched from storage exactly once for the entire distributed training job. Finally, it introduces *coordinated prep*, which coordinates data fetch and pre-processing among concurrent HP search jobs. Coordinated prep takes advantage of the fact that all HP jobs are operating on the same data; all concurrent jobs can share one epoch's worth of pre-processed data. In each epoch, data is fetched and pre-processed exactly once for all concurrent HP jobs, eliminating a significant amount of redundant work.

4.1.1 Assumptions and Goals

CoorDL is designed to be a user-space data loading library usable by individual DNN training jobs on cloud GPU VMs, or dedicated servers in a cluster. The focus of CoorDL is to accelerate different training scenarios such as single-server training, distributed training, and hyperparameter search, by utilizing available hardware resources efficiently.

CoorDL does not impact accuracy; training can sample as usual from the entire dataset, regardless of what is cached. does not alter the randomness of DNN data access pattern and does not tweak the learning algorithm in any way. CoorDL does not require specialized hardware, and runs over commodity networking and storage hardware.

4.2 CoorDL

CoorDL coordinates fetching data from storage, pre-processing data, and creating minibatches for DNN training. Using insights from our analysis, CoorDL minimizes fetch and prep stalls using three core techniques enumer-

Technique	***Impact***	***Benefits***
MinIO Cache	DNN-aware caching to minimize IO by reducing cache misses per epoch (§4.2.2)	Single-server training
Partitioned MinIO Cache	Eliminate redundant fetch by coordinating remote MinIO caches (§4.2.3)	Distributed training
Coordinated Prep	Eliminates redundant fetch and prep across jobs (§4.2.4)	Single-server training

Table 4.2: **Overview of techniques**.

ated in Table 4.2. First, CoorDL uses the novel MinIO software cache that exploits the data-access pattern of DNN training workloads to eliminate cache thrashing. Second, CoorDL coordinates the local MinIO caches of individual servers during distributed training; if there is a cache miss in a server's MinIO cache, CoorDL fetches data preferentially from a remote MinIO cache rather than local storage. Finally, CoorDL introduces the novel coordinated-prep technique, that coordinates fetch and prep of data items across all concurrent jobs in a server, if they operate on the same dataset (such as in HP search).

4.2.1 Overall Architecture

The overall architecture of CoorDL is shown in Figure 4.1. The training dataset resides on a local storage device like SSD or HDD. If the data resides on a remote storage service, the data is cached in local storage when it is first accessed [118]. For all later epochs, the data is fetched from local storage. In each training iteration, a minibatch of data must be fetched from disk

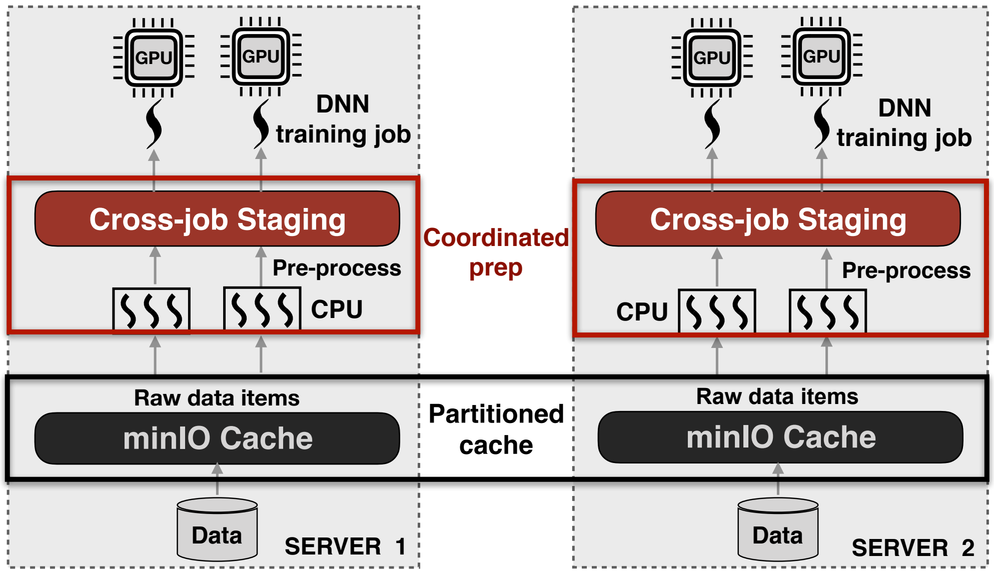


Figure 4.1: **Architecture of CoorDL**. Raw data items from the local storage are cached in the MinIO cache. Multiple CPU threads fetch items from the local (or remote) MinIO cache, pre-process and create minibatches, which are then staged for sharing across jobs, if there are multiple jobs.

(or cache), pre-processed to apply random transformations and collated to a tensor that can be copied over to the GPU for DNN computation. CoorDL manages its own MinIO cache of the raw data items (before any stochastic pre-processing transformations are applied). The data sampling and randomization is unmodified; in each epoch, every minibatch is sampled randomly from the dataset. Every data item is then subjected to the random pre-processing pipeline specified in the training workload. The prepared minibatch is then placed in a cross-job staging area for consumption by the GPU. If a single data-parallel job is running across multiple GPUs in a server, then the minibatches in the staging are used exactly once per epoch and discarded; if there are concurrent HP jobs on a server, then the staging area retains minibatches until each concurrent job has used it exactly once in the current epoch. Any minibatch that satisfies this criteria is evicted from the staging area to make way for newer batches.

We now discuss CoorDL's three core techniques in detail.

4.2.2 The MinIO cache

DNNs suffer from fetch stalls if the dataset cannot be fully cached in memory and has to be fetched from the storage during training (§3.4). Recall from Fig 3.2 that fetch stalls occur when the rate of data fetch is lower than the rate of compute (despite prefetching and pipelining data fetch with compute). When fetch stalls occur, training proceeds at the rate at which uncached data items can be fetched from storage; therefore it is important to minimize the amount of data fetched from storage in each epoch. MinIO tackles this problem by ensuring that every item in the cache is used effectively in each epoch; thereby minimizing the amount of disk IO per epoch to the ideal minimum.

DNN training has a unique data access pattern: it is *repetitive across epochs* and *random within an epoch.* Training is split into epochs: each epoch accesses all the data items in the dataset exactly once in a random order.

Currently, DNN training platforms rely on the OS Page Cache to cache training data. Every data item read from the storage device is cached in the Page Cache to speed up future accesses. When the Page Cache reaches its

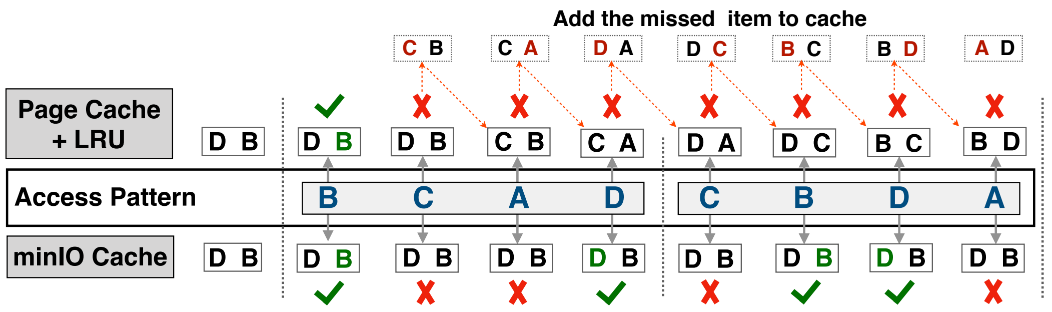


Figure 4.2: **Cache hits with MinIO**. Cache activity for two "epochs" of training for page cache and MinIO.

capacity, a *cache replacement policy* decides which of the existing items to evict to make space for the new one. Linux uses a variant of Least Recently Used (LRU) for cache replacement [81].

However, we make a key observation about the DNN access pattern that is at odds with such cache replacement policies. *All* data items in the dataset have equal probability of access in an epoch. Therefore, it is *not important* which data item is cached. Instead, it is crucial that cached items are not replaced *before* they are used, to minimize storage I/O per epoch.

Therefore, MinIO recommends a simple and unintuitive solution; *items, once cached, are never replaced* in the DNN cache. MinIO works as follows. In the first epoch of the training job, MinIO caches random data items as they are fetched from storage, to populate the cache. Once the cache capacity is reached, MinIO will not evict any items in the cache; instead, the requests to other data items default to storage accesses. The items in the MinIO cache survive across epochs until the end of the training job. Every epoch beyond the first gets exactly as many hits as the number of items in the cache; this reduces the per-epoch disk I/O to the difference in the size of dataset and the cache.

Figure 4.2 contrasts the caching policy of the OS Page Cache and MinIO. Consider a dataset of size 4 (with items A – D) and a cache of size 2 (50% cache). Let's say after warmup, the cache has two items D and B. Figure 4.2 shows the state of the cache for two training epochs. MinIO only incurs capacity misses per epoch (here 2); the Page Cache on the other hand, can result in anywhere between 2-4 misses per epoch because of thrashing. For instance, in the first epoch, D is in the cache to begin with, but kicked out to make way for a new item C, and later in the same epoch it is requested again (thrashing). We empirically verified this using large datasets and varying cache sizes (§6.3) and found that Page Cache results in close to 20% more misses than MinIO due to thrashing.

MinIO's no replacement policy simplifies the design of the cache as we do not need bookkeeping about the access time or frequency of data items; if we were to implement a replacement policy, such metadata needs to be tracked. The strength of MinIO thus lies in its simplicity and effectiveness.

4.2.3 Partitioned Caching

MinIO reduces the amount of disk I/O (fetch stalls) in single-server training. In distributed training, the dataset is partitioned and processed by a group of servers. Each server operates on a random shard of the dataset per epoch.

The MinIO cache is not efficient in this setting. For example, consider a distributed training job across two servers, each of which can cache 50% of the dataset. In every epoch, each server has to process a random 50% partition of the dataset, some of which may be hits in the local MinIO cache but the misses result in storage I/O, which is expensive and results in fetch stalls.

We observe that the cross-node network bandwidth in publicly available cloud GPU instances and our clusters(10-40 Gbps) is upto 4× higher than the read bandwidth of local SATA SSDs (530 MBps). Data transfer over commodity TCP stack is much faster than fetching a data item from its local storage, on a cache miss. Therefore, CoorDL introduces partitioned caching across the DRAM of all servers in the distributed job. While MinIO ensures that each epoch gets maximum hits in the cache, partitioned cache further reduces fetch stalls by increasing the rate at which uncached data items are fetched.

Partitioned caching works as follows. In the first epoch, the dataset is sharded across all servers, and each server populates it's local MinIO cache with data items in the shard assigned to it. At the end of the first epoch, CoorDL collectively caches a part of the dataset of size equal to the sum of capacities of individual MinIO caches. To route data fetch requests to the appropriate server, CoorDL maintains metadata about data items present in each server's cache. Whenever a local cache miss happens in the subsequent epoch at any server, the item is first looked up in the metadata; if present, it is fetched from the respective server over TCP, else from its local storage.

If the aggregate memory on the participating servers is large enough to cache the entire dataset, then partitioned caching ensures that there is no storage I/O on any server beyond the first epoch; the entire dataset is fetched exactly once from disk in the duration of distributed training.

In settings where the local MinIO cache has capacity larger than the shard assigned to it, CoorDL first caches the entire shard and fills up any available spots in the background with supplementary items that are not in the assigned shard. Note that while this introduces redundancy in the global cache, supplementary items are not added to the metadata used by partitioned cache; they are only meant to maximize local cache hits. This helps balance the load of data fetch requests on each server.

4.2.4 Coordinated Prep

Hyperparameter (HP) search for a model involves running several concurrent training jobs, each with a different value for the HP and picking the best performing one. Our analysis shows that co-locating HP search jobs on the same server results in both fetch and prep stalls (Chapter 3) due to lack of coordination in data fetch and prep among these jobs.

CoorDL introduces *coordinated prep* to address this issue. Each job in the HP search operates on the same data; hence, instead of accessing data items independently for each job, they can be coordinated to fetch and prep the dataset exactly once per epoch. Each epoch is completed in a synchronized fashion by all HP jobs; as a result, pre-processed minibatches created by one job can be reused by all concurrent jobs.

Coordinating HP search jobs must be done carefully to ensure this invariant holds: *each job processes the entire dataset exactly once per epoch.* A naive way of doing this is to pre-process the dataset once and reuse across all HP jobs and all epochs. This approach will not work for two reasons. First, reusing pre-processed data across epochs may result in lower accuracy, as the random transformations are crucial for learning. Second, the pre-processed items are 5–7× larger in size when compared to the raw data items. Caching pre-processed items will overflow the system memory capacity quickly. If we store them on storage, we may incur fetch stalls.

Coordinated prep addresses these challenges by staging pre-processed minibatches in memory for a short duration *within an epoch.* Since each job has identical per-minibatch processing time, the minibatch is short lived in the staging area. Coordinated prep works as follows.

Each HP search job on a server receives a random shard of the dataset when they start. Each job fetches and pre-processes the assigned shard, creating minibatches as they would normally do. When ready, these minibatches are exposed to the other jobs in the cross-job staging area. This is a memory region that is accessible to all running jobs on the server. Additionally, each minibatch has a unique ID and an associated atomic counter that tracks how many jobs have used this minibatch so far in the current epoch. When a job needs a minibatch for GPU processing, it retrieves it from the staging area and updates its usage counter. A minibatch is deleted from the staging area when it is used exactly once by all running jobs, as we want to ensure that it is not used across epochs. We empirically show in §6.3 that the addition of cross-job staging area does not introduce additional memory overhead.

Thus, coordinated prep ensures one sweep over the dataset per epoch for both data fetch and pre-processing, eliminating redundant work. Note that coordinated prep allows addition or removal of jobs only at epoch boundaries; this is not an issue because popular HP search algorithms evaluate the objective function (for e.g., accuracy), and make decisions on terminating or continuing the job at epoch boundaries [102, 123]

Handling job failures and terminations. The progress of each HP search job in CoorDL is dependent on the progress of all other running jobs, because each job is responsible for pre-processing a shard of the dataset. Therefore, if one of the jobs is killed by the user in the middle of an epoch, or terminates abruptly, all other jobs may stall waiting for minibatches that the job was responsible for preparing. To address this, CoorDL uses a failure detection module to monitor the status of running jobs.

Every prepared minibatch fetched from the staging area has an associated timeout. We empirically set this timeout to be equal to the duration of processing 10 minibatches, because it was sufficient to mask the discrepancies in data loading time of individual jobs. If any job times out waiting for a minibatch in the staging area, it notifies the driver process of a possible failure. Note that, the global sequence of data items seen by each job is identical;

therefore, all the jobs can deterministically identify which job failed to populate the batch it is waiting on. CoorDL's failure detection module verifies if the reported job is alive or dead; if alive, it issues a broadcast to all the jobs to retry fetching the minibatch from staging area, else it spawns a new process to resume data loading for the shard that failed.

4.2.5 Implementation

We implement CoorDL by adding 1.5K lines of C++ code to DALI. Cross-batch staging is implemented as a binding between DALI and PyTorch in 935 lines of Python code. We implement DS-Analyzer in Python with 1.1K LOC. We have also implemented our techniques in the native PyTorch data loader (Py-CoordDL- details and evaluation are presented in the extended version of our data stalls paper [145]). Since DALI is an optimized data preprocessing library for DNN training and performs strictly better than default PyTorch, it is a stronger baseline to compare against. DALI (on PyTorch), and the native PyTorch dataloader.

MinIO implementation. MinIO is implemented as a software cache using file-backed shared memory. Each data item is stored as a byte stream in a shared memory file in `/dev/shm`. To check if an item is present in cache, the data worker first tries to attach to the shared memory segment indexed by the name of the data item. If it fails, a disk access is incurred. If cached, it reads the raw byte stream from the attached shared memory segment and performs the pre-processing steps. This design (1) eliminates the need for serialization and pickling which is typically incurred if we use Python's shared memory data structures and (2) allows multiple processes to access the cached data items for concurrent reads.

MinIO is completely in user-space, making it easy to use in scenarios where the user has no root privileges to modify the kernel (e.g., in production clusters, users can run their training code in a docker container with the fixed CPU and memory allocated to them, but not a VM with a custom kernel). This calls for a flexible user-space library rather than in-kernel changes. Therefore, we chose to avoid kernel modifications such as changing the Linux page cache eviction algorithm.

MinIO can be alternately implemented using primitives such as `madvise()` and `mlock()` to prevent data items from being paged by locking the process's virtual address space in memory for the cached entries [177, 129]. However, since data workers and the training on each GPU (for multi-GPU jobs) are implemented as individual processes in PyTorch, we choose to use file-backed shared memory to enable cache sharing across all processes spawned by the training job.

Partitioned cache implementation. An instance of MinIO cache server runs on each node across which the training is distributed. Partitioned caching coordinates the MinIO caches at each server to preferentially fetch data from remote DRAM using TCP connections. To eliminate frequent TCP connection overheads, each server pre-establishes as many connections as the number of data fetch threads on the nodes. This connection is kept active for the lifetime of the job. Each server participating in the distributed training maintains metadata of MinIO cache contents at other servers to redirect data fetch requests to remote servers.

Coordinated prep. The cross-job staging area is implemented using Python's multiprocessing hashmap. Each data worker inserts the pre-processed minibatch into the shared hashmap in parallel. Minibatches in this hashmap are serialized and returned to each DNN job via sockets. CoorDL additionally maintains a counter for each inserted minibatch that tracks the number of DNN jobs that have used this batch; when a batch has been consumed by all live HP search jobs, the minibatch is deleted from the staging area. The job failure detection module uses an initial timeout that is 10 times the duration of an iteration(batch). Empirically, for all models we tested on, this duration was sufficient to mask the minor differences in the per-batch duration across jobs.

CoorDL can be used as a drop-in replacement to either native PyTorch dataloader, or DALI, with no modifications to the training script. Using DS-Analyzer requires about 10 – 15 lines of additions to the DNN training script.

4.3 Evaluation

We now evaluate the efficacy of CoorDL on three different aspects of the training process: hyperparameter tuning, multi-GPU training on a single server, and distributed training across multiple servers. We evaluate our techniques on **nine** models, performing **three** different ML tasks (image classification, object detection and audio classification) on four different datasets, each over 500GB as shown in Table 3.1. Since DALI strictly outperforms PyTorch DL, we use DALI (best of CPU or GPU based prep) as the baseline in our experiments.

Experimental setup. We evaluate CoorDL on two representative server configurations (Tbl 3.2) each with 500 GiB DRAM, 24 CPU cores, 40 Gbps Ethernet, eight GPUs, and 1.8 TiB of storage space. `Config-SSD-V100` uses V100 GPUs and a SATA SSD, while `Config-HDD-1080Ti` uses 1080Ti GPUs and a magnetic hard drive. `Config-SSD-V100` is similar to the AWS `p3.16xlarge` instance [6], while `Config-HDD-1080Ti` is similar to the AWS `p2.8xlarge` instance [5]. We use the same training methodology we used for analysis (§3.2).

We seek to answer the following questions:

- How does the MinIO cache affect multi-GPU training on a single server ? (§4.3.1)
- How does partitioned caching improve training time for jobs distributed across multiple servers? (§4.3.2)
- How does coordinated prep benefit HP search? (§4.3.3)
- Does CoorDL affect end-to-end DNN training accuracy? (§4.3.4)
- Does CoorDL enable better resource utilization compared to DALI? (§4.3.5)

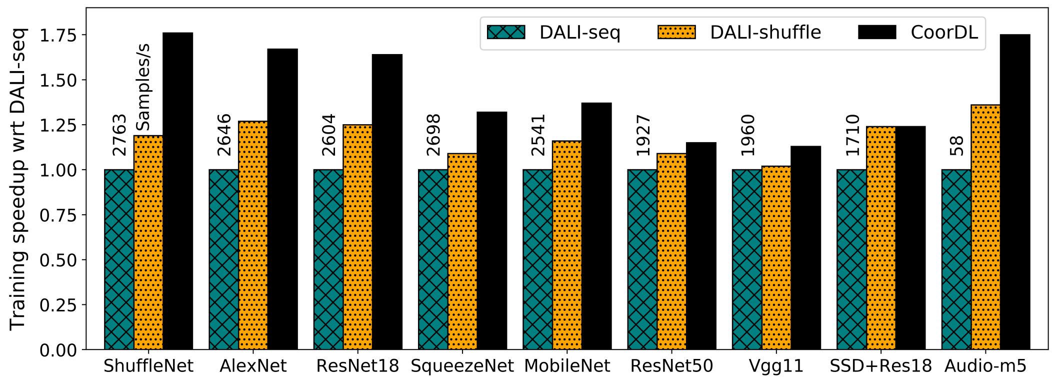


Figure 4.3: **Single server training**. This graph compares DALI against CoorDL for single server training. CoorDL significantly accelerates training by efficiency utilizing available memory with MinIO

- Does CoorDL mitigate data stalls in state-of-the-art ML optimized servers like the DGX-2? (§4.3.6)

4.3.1 Single-server Multi-GPU training

CoorDL speeds up a single-server training job by reducing fetch misses using the MinIO cache. Figure 4.3 plots the relative speedup with respect to DALI while training the image classification and object detection models on the OpenImages dataset, and audio classification on FMA dataset. We evaluate MinIO against two modes of DALI. DALI's default mode is *DALI-seq*, where it reads data sequentially off storage and shuffles them in memory [11]. *DALI-shuffle* accesses the dataset in a randomized order (doing random reads, similar to the native dataloader of PyTorch).

MinIO results in upto 1.8× higher training speed compared to DALI-seq by eliminating thrashing on `Config-SSD-V100`. When the image classification

	DALI-seq	DALI-shuffle	CoorDL
Cache miss	66%	53%	35%
Disk IO (GB)	422	340	225

Table 4.3: **Impact on fetch misses and disk IO**. When training ResNet18 on OpenImages (645GB), CoorDL reduces cache misses from 66% to 35%. `Config-SSD-V100` caches 65% of the dataset, so this is the minimum miss rate.

models are trained with ImageNet-22k dataset, CoorDL results in up to 1.5× speedup. On `Config-HDD-1080Ti`, CoorDL accelerates ResNet50 training on OpenImages by 2.1× compared to DALI-seq and 1.53× compared to DALI-shuffle respectively.

Reduction in cache misses. We measure the disk I/O and number of cache misses when training ShuffleNet on OpenImages dataset on `Config-SSD-V100`. This server can cache 65% of the dataset. CoorDL reduces misses to the minimum number of 35%, resulting in 225 GB of I/O. In contrast, DALI-Seq results in 66% cache misses, increasing I/O by 87% to 422 GB; DALI-shuffle results in 53% cache misses, increasing I/O by 50% compared to CoorDL to 340 GB.

Note that, when the whole dataset does not fit in memory, DALI-shuffle performs better than DALI-seq (because sequential access is a pathological case for the Linux LRU page cache). Therefore, our evaluation in the rest of this section compares CoorDL to the stronger baseline, DALI-shuffle.

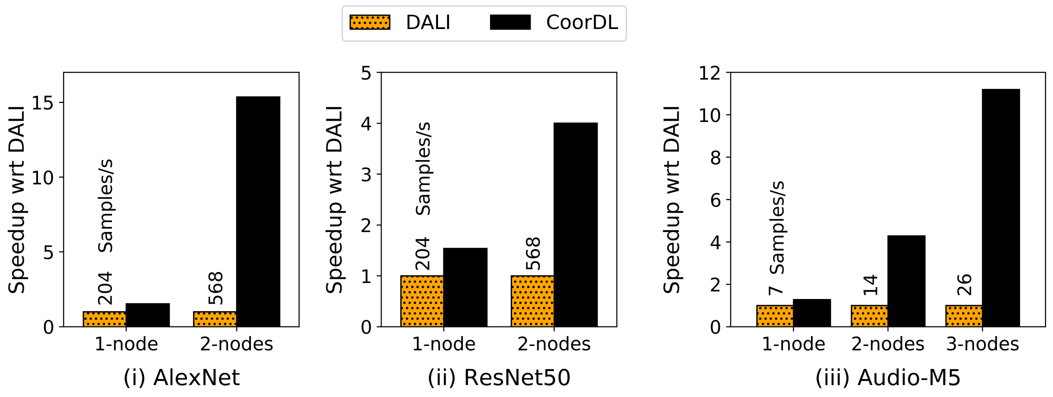


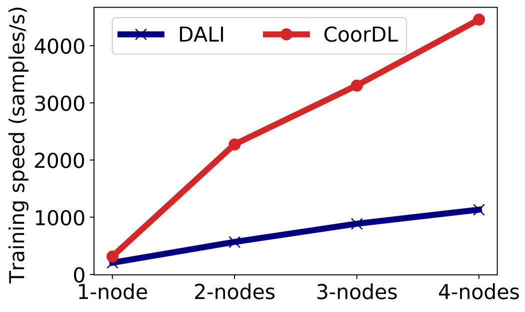
Figure 4.4: **Distributed training across servers**. This graph compares DALI against CoorDL for multi-server training. CoorDL significantly accelerates training by optimizing remote MinIO fetches

4.3.2 Multi-Server Distributed Training

We now evaluate CoorDL on a distributed training scenario. The lack of cache co-ordination between the participating servers results in fetch misses that lead to disk I/O. CoorDL uses partitioned caching to avoid redundant I/O.

Figure 4.4 shows that CoorDL improves the throughput of distributed training jobs by upto 15× (AlexNet on OpenImages) when trained across two `Config-HDD-1080Ti` servers (16 GPUs). On `Config-HDD-1080Ti` servers, 65% of the OpenImages dataset can be cached on a single server; and it can be fully cached in the aggregated memory of two servers. Therefore, CoorDL moves the training job from being I/O bound to GPU bound.

When trained across two servers on `Config-SSD-V100`, CoorDL accelerates ShuffleNet on ImageNet-22k by 1.3×, and Audio-M5 on FMA by 2.9×.



(a) Distributed training

Nodes	Disk IO(GB)
1	342
2	119
3	70
4	50

(b) Disk IO

Figure 4.5: **Distributed training with CoorDL**. The plot compares DALI with CoorDL when training ResNet50 across upto 4 nodes. Even when each node can cache 65% of the dataset, DALI results in I/O bound training due to disk fetch, while CoorDL results in zero disk accesses beyond first epoch.

The relative gains are lower on `Config-SSD-V100` because the cost of a fetch miss is lower on SSDs due to its high random read throughput, as compared to HDDs on `Config-HDD-1080Ti`.

Scalability of partitioned caching. When we distribute training to a large number of servers, such that their aggregate memory is higher than the total dataset size, CoorDL continues to outperform DALI as shown in Figure 4.5a. In this experiment, we train ResNet50 on OpenImages on `Config-HDD-1080Ti`, where each server can cache 65% of the dataset. When training extends to 24 GPUs(3 servers), or 32 GPUs(4 servers), the throughput with CoorDL increases because, training is not bottlenecked on I/O and more GPUs for training naturally results in faster training due to increase in GPU parallelism. With DALI, although the throughput increases, it is still I/O bound; the increase in throughput is due to the reduced disk I/O per server when training

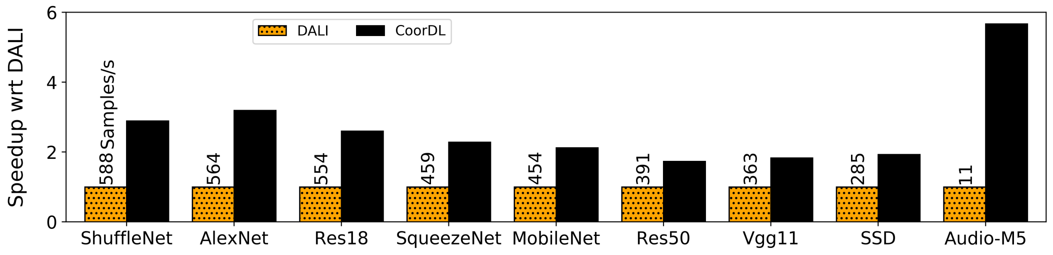


Figure 4.6: **Hyperparameter search**. This graph compares DALI against CoorDL for HP search. CoorDL significantly accelerates training using coordinated prep

is distributed as shown in Table 4.5b. Although the I/O per server decreases with DALI as we distribute training across more servers, note that the GPU parallelism is also proportionately increasing; the GPU compute rate (G) and prefetch rate (F), are proportionately increasing, leaving the performance gap the same. CoorDL however, masks this gap by eliminating storage I/O by exploiting the high bandwidth Ethernet between servers.

4.3.3 Hyperparameter Search

Figure 4.6 plots the relative increase in throughput of individual jobs across several models when eight concurrent HP search jobs are run on a `Config-SSD-V100` server. On less computationally complex models like AlexNet and ShuffleNet, CoorDL increases training speed by 3×, because these models are originally CPU bound due to prep.

For the audio model, CoorDL increases the training speed by 5.6×. CoorDL reduces the total disk IO from 3.5TB to 550GB, moving the job from being I/O bound to GPU bound. The reduced I/O results from CoorDL avoid-

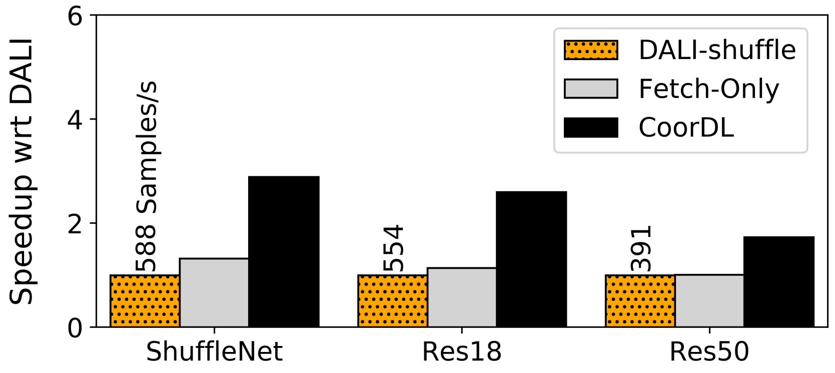


Figure 4.7: **Breakdown of benefits due to coordinated prep**. This graph shows the split of speedup due to coordinated fetch alone, and CoorDL (coordinated fetch and prep)

ing cache thrashing using coordinated prep. Similarly, on `Config-HDD-1080Ti`, CoorDL results in 5.3× faster training on the audio model, and 4.5× faster training on ResNet50.

On `Config-HDD-1080Ti`, CoorDL results in 5.3× faster training on the audio model, and 4.5× faster training on ResNet50 by coordinating data fetch and prep.

Split of coordinated prep benefits. Next, we show the breakdown of speedup due to coordination of data fetch and prep during HP search. When fetch is coordinated, concurrent jobs use data fetched by other jobs; but each job performs its own data prep. CoorDL coordinates both; eliminating redundant fetch and prep. Figure 4.7 plots the results on `Config-SSD-V100`. In this case, data stall is dominated by prep, which CoorDL mitigates unlike prior work like Quiver [118] that only coordinates fetch.

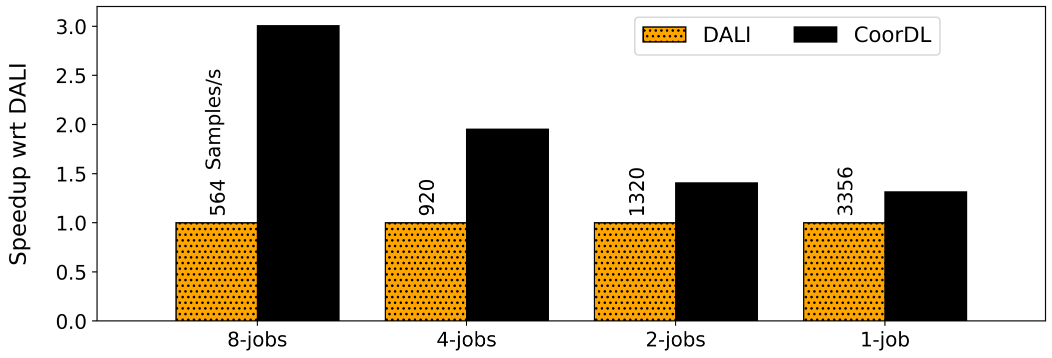


Figure 4.8: **Varying number of HP jobs**. This graph compares DALI against CoorDL as we vary the number of concurrent HP jobs. The benefit with CoorDL is higher as the number of concurrent jobs increases.

Multi-GPU HP search jobs. Figure 4.8 evaluates the efficacy of CoorDL for different configurations of HP search jobs on a machine; 8 1-GPU jobs, 4 2-GPU jobs, 2 4-GPU jobs, or 1 8-GPU job for AlexNet on OpenImages. For a single job case, the benefit is due to the MinIO cache; in other configurations, it is due to coordinated prep. When there are a lot of concurrent jobs, pre-processing becomes the bottleneck; coordinated prep is able to improve performance significantly.

HP search with fully cached dataset. CoorDL's ability to speed up HP search jobs comes from coordinating pre-processing to overcome the imbalance in the ratio of CPU cores to GPU. We perform HP search with 8 jobs on `Config-SSD-V100` with ImageNet-1k dataset that fits entirely in memory. As shown in Table 4.4, CoorDL sped up HP search by 1.9× on AlexNet and and 1.2× on ResNet50 by eliminating redundant prep.

HP search on servers with more CPU cores. `Config-SSD-V100` has

	Per job speed (Samples/s)	
Model	DALI	CoorDL
ShuffleNet	1441	1.81×
AlexNet	1399	1.87×
ResNet18	1056	1.53×
SqueezeNet	835	1.50×
MobileNet	752	1.35×
ResNet50	569	1.21×
VGG11	552	1.22×

Table 4.4: **HP search with CoorDL on a fully cached dataset**. On `Config-SSD-V100`, when training with the small ImageNet-1k dataset that fits in memory, CoorDL provides upto 1.87× speedup by eliminating redundant pre-processing

3 CPU cores per V100 GPU. To understand if servers like AWS p3.16xlarge with more CPU cores exhibit data stalls due to lack of co-ordination in pre-processing, we perform HP search with 8 1-GPU jobs on a server with 64 vCPUs and 8 V100s. Our experiment considers a fully-cached dataset to eliminate any I/O stalls. When training ResNet18 with OpenImages, CoorDL's co-ordinated prep accelerated training by 2× even when a total of 64vCPUs are used (8 vCPUs per GPU).

4.3.4 Training to Accuracy with CoorDL

CoorDL does not change the randomness of data augmentation techniques involved. Its techniques do not affect the learning algorithm. To demonstrate this, we train ResNet50 to accuracy on ImageNet-1K using 16

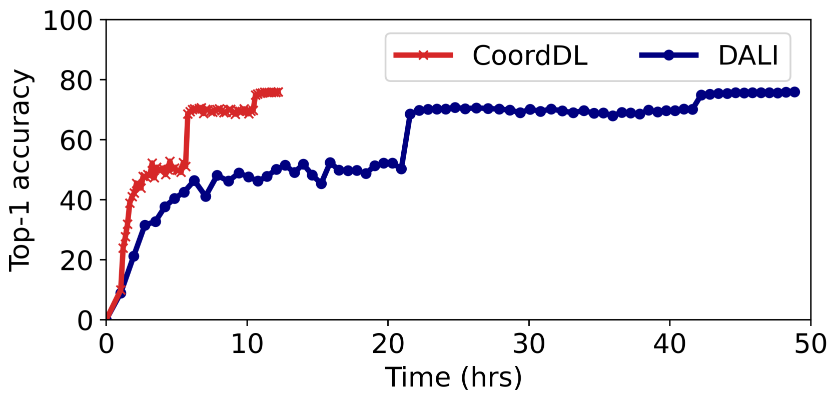


Figure 4.9: **Top-1 validation accuracy during training**. In training ResNet50 with ImageNet-1K on 16x 1080Tis across 2 servers, CoorDL reduces the time to accuracy by 4× by coordinating the caches across the job's individual servers.

GPUs across two `Config-HDD-1080Ti` servers, where each server is capable of caching 50% of the dataset. Figure 4.9 shows that CoorDL reduces the time to target accuracy (75.9%) from two days to just 12 hours (4× better), due to partitioned caching.

4.3.5 Resource Utilization

MinIO results in lower disk I/O and better CPU utilization. Figure 4.10 shows the I/O for two epochs of training ResNet18 on OpenImages on `Config-SSD-V100`. The I/O behavior is similar across models and server configurations.

DALI observes cache hits at the beginning of the epoch, but soon becomes I/O bound (disk bandwidth: 530 MB/s). Since MinIO is caching a random subset of the dataset, cache hits are uniformly distributed across the

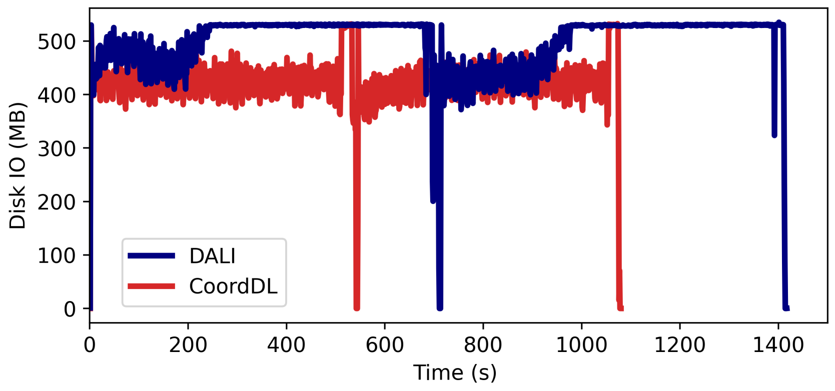


Figure 4.10: **Disk I/O pattern with MinIO (ResNet18 on OpenImages)**. DALI gets cache hits at the start of every epoch; however due to thrashing, all requests result in storage access beyond a point. CoorDL results in a more uniform I/O pattern and faster training.

epoch in CoorDL. This results in a predictable I/O access pattern and faster training (epochs end earlier in Figure 4.10).

Profiling the CPU during training shows in Figure 4.11 that the pre-processing threads in DALI are often stalled waiting for I/O. Since MinIO reduces the total disk I/O, CoorDL is able to better utilize the CPU threads for pre-processing. The combination of lower disk I/O and better CPU utilization leads to shorter training times when using CoorDL.

CoorDL uses a fraction of available network bandwidth. CoorDL shards the dataset equally among all servers in distributed training to ensure load balancing. We track the network activity during the distributed training for ResNet50 on OpenImages across two, three, and four servers with DALI and CoorDL. CoorDL used 5.7 Gbps per server of network bandwidth (14%

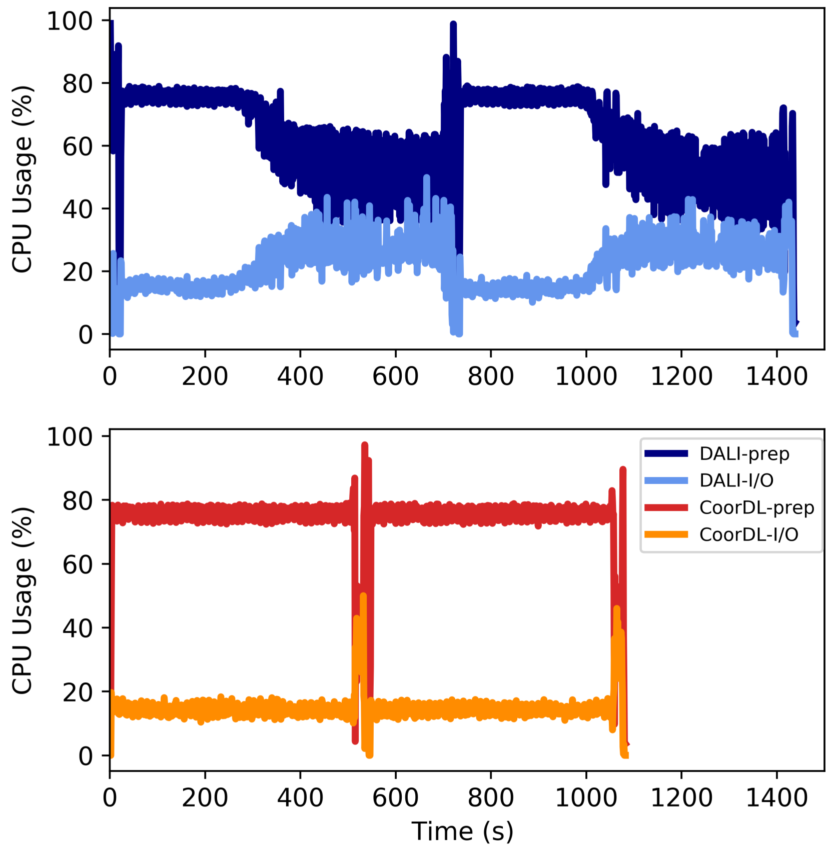


Figure 4.11: **CPU utilization with MinIO**. This plot shows the CPU utilization over time for DALI and CoorDL when training ResNet18 on OpenImages. CoorDL uses cache effectively to reduce disk I/O, therefore utilizing CPU on useful pre-processing rather than waiting on I/O

of the 40 Gbps available). DALI used 1.18 Gbps of network bandwidth per server. CoorDL used 4.8× higher network bandwidth to train 4.3× faster than DALI.

Co-ordinated prep has low memory overhead. By design, co-ordinated prep has the same memory requirements as DALI. To experimentally validate

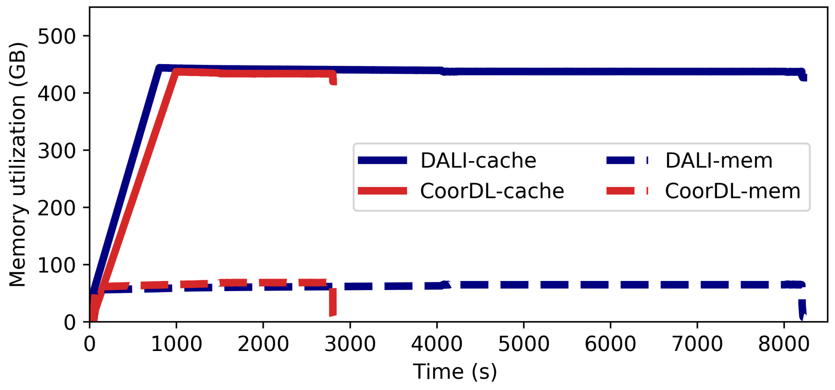


Figure 4.12: **Memory utilization of coordinated prep**. This plot shows the memory utilization for two epochs of HP search using AlexNet on Open-Images, with 8 concurrent jobs. CoorDL uses 5GB of extra process memory; resulting in 5GB lower cache space. Total memory utilization at the node is constant.

this, we track the memory utilization of running hyperparameter search on AlexNet on OpenImages on a `Config-SSD-V100` server using eight concurrent jobs. Figure 4.12 plots the memory utilization over time for both the process working memory, and the cache. CoorDL uses 5 GB of extra process memory to store prepared mini-batches in memory until all hyperparameter jobs consume it. We reduce the cache space given to CoorDL by 5 GB (keeping the total memory consumption same for CoorDL and DALI). Despite the lower cache space, CoorDL still accelerated training by 2.9×.

4.3.6 CoorDL on DGX-2

We now compare CoorDL against DALI on the bleeding-edge ML optimized server DGX-2 while performing HP search across 16 GPUs using Open-

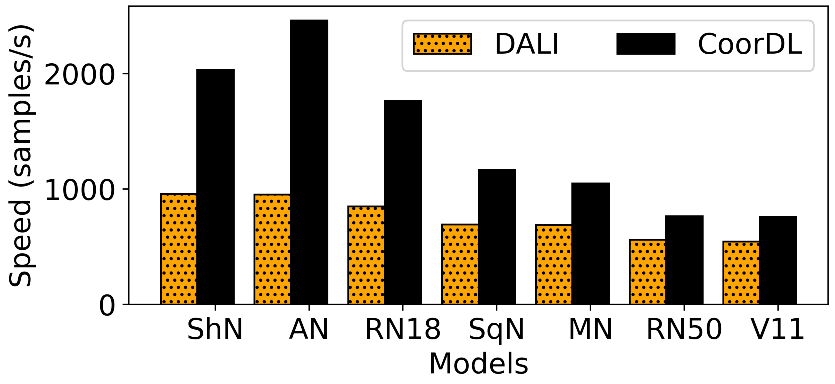


Figure 4.13: **Evaluation of CoorDL on DGX-2**. This graph compares DALI against CoorDL for HP search using 16 GPUs on the state-of-the-art ML optimized server DGX-2. CoorDL outperforms the best GPU-based prep mode of DALI which uses 5GB of GPU memory for pre-processing. CoorDL allows training larger batch sizes, while providing better performance than DALI.

Images dataset. Since this dataset can be fully cached in the memory of DGX-2 (1.5TB DRAM), we observe no stalls due to data fetch beyond the first epoch. However, the imbalance in the ratio of CPU-GPU results in prep stalls which CoorDL mitigates by coordinating pre-processing. CoorDL accelerates HP search by $1.5\times - 2.5\times$ over DALI by eliminating redundant data prep, enabling efficient usage of CPU to mask prep stalls as shown in Figure 4.13.

4.4 Limitations and Discussion

CoorDL mitigates data stalls in a user-space library to better utilize the computational capabilities of GPUs and thereby accelerate training. In this section, we discuss some limitations and scope for future work.

Trade-off between convergence rate and epoch time for other SGD variants. In this dissertation, we focus on the most common case of mini-batch SGD with a random shuffling of the data in every epoch which is the default for the models we analyzed. However, for performance, practitioners choose to shuffle datasets once very few epochs, or not perform cross-machine shuffling in case of multi-machine training. A future direction is to understand the impact of relaxing the ETL requirements assumed in this thesis (such as random prep and shuffling every epoch) on epoch time and model convergence. Although relaxing these constraints may reduce data stalls and hence epoch time, it may prolong convergence, or affect the accuracy of some models. It is worth investigating this behavior theoretically and empirically.

Remote Data Fetches. CoorDL assumes that the local storage on the machine is large enough to store the training dataset. In cases where the dataset size exceeds the size of local storage device, stalls may occur due to remote data fetches. We assume that this is not a common occurrence. We have consistently found that most important DNN training models that provide state-of-the-art results, across tasks (image classification, object detection, language models, etc.), use publicly available datasets that entirely fit on a single SSD - ImageNet-1K [179], ImageNet-22K [17], OpenImages-600 [80], or music datasets [66].

4.5 Summary

In summary, we present the design of CoorDL, a coordinated caching and pre-processing library for DNN training. We evaluate CoorDL against the state-of-the-art data pipeline DALI, on PyTorch using three different tasks (image classification, object detection and audio classification) on two representative server configurations and four large datasets. CoorDL provides significant training speedups of upto 5.7× for HP search, 2.1× for multi-GPU training on a single server, and upto 15× for multi-server distributed training. The techniques behind CoorDL are simple and intuitive, easing adoption in production systems.

Chapter 5

Synergy: Mitigating Data Stalls in Multi-Tenant Clusters

This chapter discusses how to mitigate data stalls in multi-tenant GPU clusters. Enterprises typically setup large multi-tenant clusters, with expensive hardware accelerators like GPUs, to be shared by several users and production groups [105, 210]. Users can request jobs to be scheduled on the cluster, specifying the number of GPUs required by the job.

These jobs are scheduled and managed either using traditional big-data schedulers, such as Kubernetes [49] or YARN [203], or using modern schedulers that exploit DNN job characteristics for better performance and utilization [153, 209, 133, 87, 121, 166, 50]. These DNN schedulers decide how to allocate GPU resources to many jobs while implementing complex cluster-wide scheduling policies to optimize for objectives such as average job completion times (JCT), makespan, or user-level fairness.

In this chapter [1], we exploit one of the main observations from our data stall analysis : different DNNs exhibit different levels of sensitivity to the

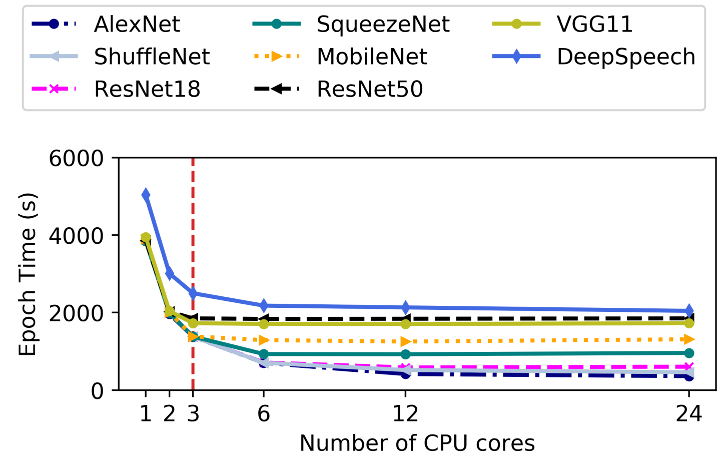
[1]This Chapter is based on the arXiv preprint, Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters [144]

amount of CPU, memory, and storage bandwidth available to the job. Therefore, when such DNNs are co-scheduled, it is possible to improve the overall cluster utilization and efficiency by performing resource-aware allocations using a new scheduler, Synergy.

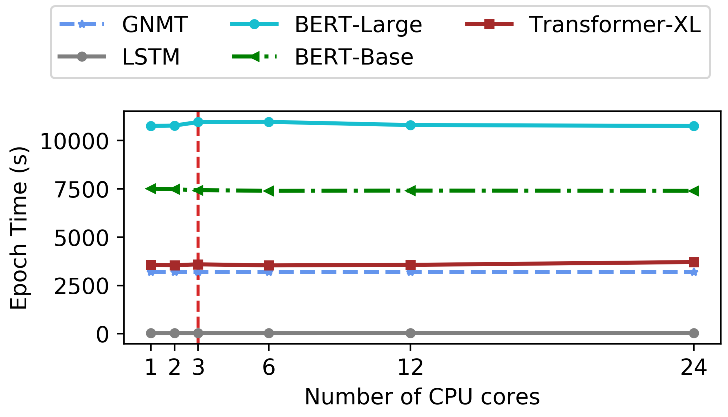
The rest of this chapter is organized as follows. We first motivate the need for resource-sensitivity aware scheduling in Section §5.1. We then present Synergy, a resource-sensitivity aware scheduler that optimistically profiles the DNN's sensitivity to resources and makes disproportionate allocations, ensuring no job achieves lower than GPU-proportional throughput in Section §6.2. We present a heuristic scheduling mechanism Synergy-Tune that maps the allocations calculated by the profiler onto the cluster, while better utilizing the resources compared to a GPU-proportional allocation in Section §5.3. Finally, we show via extensive experimentation on physical and simulated clusters that Synergy's techniques improve average JCT by up to 3.4×, allowing a higher input load in Section §6.3.

5.1 Motivation

Insight. The main insight that motivates our work is that DNNs co-scheduled on a cluster exhibit different levels of sensitivity to CPU and memory allocations during training. Therefore, it is possible to improve the overall cluster utilization and efficiency by performing resource-aware allocations instead of the state-of-the-art GPU-proportional allocation without any hardware changes. Prior work on characterization study of jobs in Microsoft's



(a) DNNs sensitive to CPU allocation



(b) DNNs in-sensitive to CPU allocation

Figure 5.1: **CPU sensitivity**. Since the server has 8 GPUs and 24 CPUs in total, a GPU-proportional share is 3 CPUs. Some jobs such as Transformers need as few as 1 CPU to achieve maximum training speed; others like AlexNet need more than 12 CPUs per GPU.

Philly cluster [107] shows that CPU cycles are under-utilized in multi-tenant clusters; we use this as motivation to show that we can *exploit the disparity* *in resource requirements* ***across jobs*** *to improve overall cluster utilization without hardware upgrades* (storage, CPU, or memory).

Figure 5.1 plots the per-epoch time for various DNNs when trained on 1 GPU (in isolation, one job at a time) by varying the number of CPUs allocated to the job (ensuring that the dataset is fully cached for each job). It is trained on a server whose GPU-proportional allocation per GPU is 3 CPUs and 62GB of memory.

Figure 5.1a shows that most image and speech models are sensitive to CPU allocations; smaller models like ShuffleNet and ResNet18 require 9–24 CPU cores per GPU to pre-process data items. Increasing the CPU allocation from the GPU-proportional share 3 to 12 results in 3.1× faster training for AlexNet, and increasing it to 9 results in 2.3× faster training for ResNet18. On the other hand, most language models are insensitive to CPU allocations as shown in Figure 5.1b. This is because they have light-weight input data pre-processing. Transformer models for instance pre-processes the entire dataset before training begins, unlike image classification models which perform several unique data augmentation operations in every epoch [146].

Takeaway. *When two jobs have to be scheduled on the same server, it is possible to co-locate a CPU-sensitive job with a CPU-insensitive one. This allows CPU allocation to be performed in a resource-sensitive manner rather than GPU-proportional allocation. We can thus give more CPUs to the sensitive job and fewer to the insensitive one without hurting its throughput.*

Next, to understand the importance of memory allocations, we train two models; an image classification model - ResNet18 on OpenImages [80] and a language model GNMT on WMT, with varying memory allocations on a server whose GPU-proportional share of memory per GPU is 62GB. We observe that GNMT is insensitive to memory allocation; even if only 20GB memory is allocated (which is the required process memory for training), the training throughput is unaffected. However, increasing the memory from 62GB to 500GB for ResNet18 speeds up training by almost 2×. This is because, language models like GNMT, and transformers are GPU compute bound. Therefore, fetching data items from storage if they are not available in memory does not affect training throughput. On the other hand, image and speech models benefit from larger DRAM caches. If a data item is not cached, the cost of fetching it from the storage device can introduce fetch stalls in training [146].

Takeaway. *When co-locating two jobs on a server, it is always beneficial to pack a memory-sensitive job with an insensitive one, allowing a disproportionate resource-sensitive share of memory to improve the aggregate cluster throughput.*

5.1.1 Example

We now show how resource-sensitivity aware scheduling can improve cluster efficiency using a small example. Consider two servers each with 8 GPUs, 24 CPUs and 500GB DRAM. Let's say we have 4 jobs in the scheduling queue, each requesting 4 GPUs as shown in Table 5.1. We schedule these jobs

Job	Model
J_1	ResNet18
J_2	Audio-M5
J_3	Transformer
J_4	GNMT

Table 5.1: Example jobs

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	12	250
	$J2$	4	12	250
S_2	$J3$	4	12	250
	$J4$	4	12	250

Table 5.2: GPU-proportional allocation

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	23	400
	$J3$	4	1	100
S_2	$J2$	4	12	450
	$J4$	4	12	50

Table 5.3: Resource-sensitive allocation

using FIFO scheduling policy.

We consider two different schedules; (1) GPU-proportional allocation and (2) resource-sensitivity aware allocation. The results of these schedules are shown in Table 5.2 and Table 5.3. Figure 5.2 compares the epoch time of each of these jobs in the two scenarios. The increased resource allocation to CPU

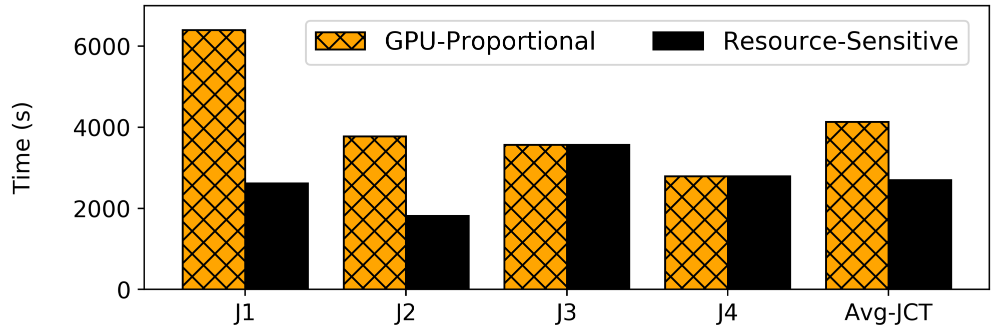


Figure 5.2: **Resource sensitive scheduling**. We compare the runtime of the jobs with two different schedules; GPU-proportional and resource-sensitive. By allocating resources disproportionately, CPU and memory sensitive jobs see increased throughputs which reduces the average JCT by 1.5×.

and memory sensitive jobs in Schedule 2 speeds up $J1$ $J2$ significantly, while leaving the runtimes of $J3$ and $J4$ unaffected. The average job completion time in the cluster thus drops by 1.5× by performing resource-aware allocations.

5.1.2 Assumptions and Goals

Collocating ML training workloads in a shared, multi-tenant cluster is a very common setup in several large organizations, for both research and production [106, 51, 209, 153, 87, 133]. Synergy targets state-of-the-art multi-tenant clusters similar to the ones published by prior large-scale studies by organizations like Microsoft [106] and Alibaba [210]. These clusters use on-premise servers or cloud VMs with pre-defined GPU, CPU, and memory resources. The cluster itself is shared by multiple users and jobs, and each server can host more than one job each with varying resource usage (some heavy on CPU side pre-processing, while others heavy on GPU computation). For example, a server with 8 GPUs can host 8 single GPU jobs from different users. Since the jobs that run on these clusters have varied resource profiles, Synergy shows that it is possible to improve the overall cluster utilization and efficiency by performing resource-aware allocations instead of the state-of-the-art GPU-proportional allocation without any hardware changes. The techniques introduced in Synergy can be used behind-the-scenes for cloud GPU VMs as well by building a virtualization infrastructure to perform resource allocations as per job needs instead of using fixed VM SKUs.

Synergy assumes that the GPU cluster is homogeneous; it currently does not exploit performance heterogeneity across accelerators. Synergy also assumes that the dataset is present locally at each server and ignores data locality-awareness in its placement decisions as discussed further in §5.5. Synergy performs resource-aware allocations; it does not introduce a new scheduling policy. One of the major advantages of Synergy is that, rather than constraining to a particular scheduling policy like LAS, or FTF, Synergy improves a wide range of scheduling policies. Synergy's innovation lies in identifying and solving the resource disparity across co-located jobs in a cluster, and coming up with an allocation that best utilizes all the cluster resources while improving individual job throughputs and using the existing scheduling policy chosen by the cluster administrator.

Efficiently exploiting the heterogeneity in resource sensitivity among DNN jobs raises two important problems which have not been tackled by prior work:

1. What is the ideal resource requirement for each job (with fixed GPU demand) and how can this be determined quickly?
2. How should we pack these jobs onto servers along multiple resource dimensions efficiently, especially when we can tune the job's demand for these resources?

5.2 Synergy: Design

Synergy is a round-based scheduler for GPU clusters that arbitrates multi-dimensional resources(GPU, CPU, and memory) in a homogeneous cluster among the set of runnable jobs in every round, which is decided based on an existing scheduling policy like SRTF, FTF, FIFO etc. Synergy does not introduce a new scheduling policy; it imparts and augments *resource sensitivity awareness* to the existing scheduling policies. Synergy accomplishes this in two steps. First, it identifies the job's best-case CPU and memory requirements using an *optimistic profiling* technique (§5.2.1). Synergy then identifies a set of runnable jobs for the given round using an existing scheduling policy such that their collective GPU demand is less than or equal to the GPUs available in the cluster. Then, using the profiled job resource requirements, Synergy packs these jobs on to the available servers along multiple resource dimensions using a close-to-optimal heuristic mechanism (§5.3). At the end of a round, the set of runnable jobs are updated using the scheduling policy, and their placement decisions are recomputed.

We now discuss both the components of Synergy in detail.

5.2.1 Optimistic Profiling

A DNN job is profiled for its resource sensitivity once per lifetime of the job, i.e. on job arrival. Each incoming job is profiled by varying the CPU and memory allocated to the job. A resource sensitivity matrix is then constructed for discrete combinations of CPU and memory allocations as shown in Figure 5.3. Since DNN training has a highly predictable structure, empirically evaluating training throughput for a few iterations gives a fair estimate of the actual job throughput [209, 146].

It is easy to see that naively profiling different combinations of CPU and memory can be very expensive. For instance, if the cost of profiling one combination of CPU, and memory for a job is 1 minute, then to profile all discrete combinations of CPU and memory (assuming allocation in units of 50GB) on a server with 24 CPUs and 500GB DRAM takes about 24*10 = 240 minutes (4 hours)!

To tackle this problem, Synergy introduces an optimistic profiling technique that exploits the predictability in the relationship between job throughput and memory allocation. We observe that, with DNN-specific caches like MinIO [146], it is easy to model the job throughput behaviour as wevary the amount of memory allocated to a job at fixes CPU allocation. This is because, MinIO ensures that a job gets x% cache hits in every epoch if the memory allocated to the job holds x% of the dataset. For a given CPU allocation that

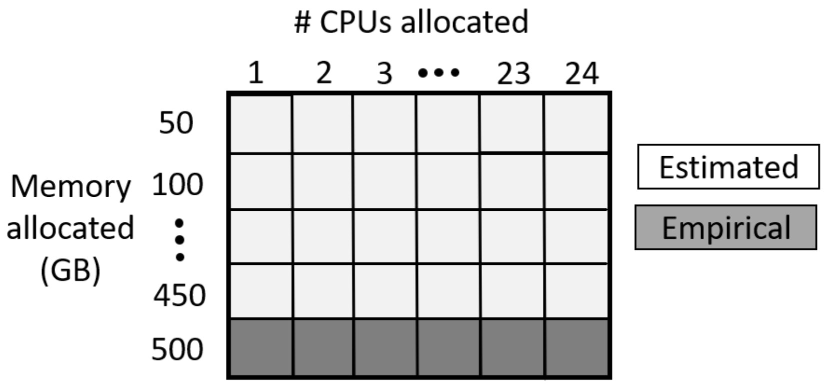
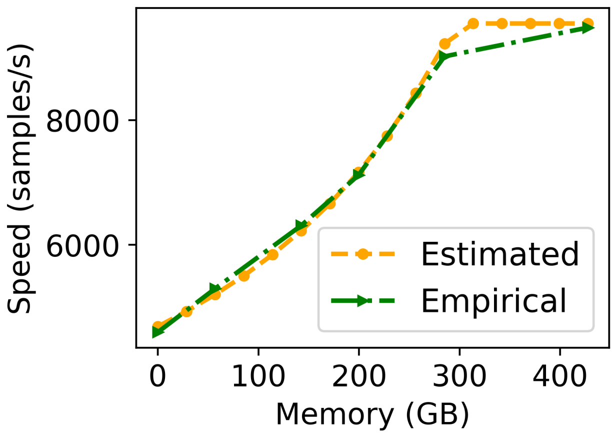


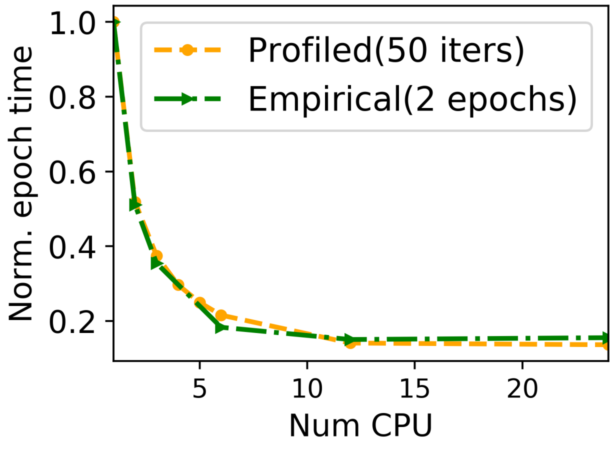
Figure 5.3: **Optimistic profiling**. We empirically evaluate the sensitivity of a model to varying # CPUs assuming a fully cached dataset. Rest of the matrix is completed using estimation

determines the CPU pre-processing speed, and a known storage bandwidth, it is easy to analytically model the job throughput or varying memory allocation. Therefore, we only need to empirically profile the job for varying CPU values at full memory allocation, as indicated by the last row of the matrix in Figure 5.3. All the other entries can be estimated using the above technique. This leads to a 10× reduction in profiling time, bringing it down to 24 minutes! We experimentally validate this in Figure 5.4a. For a 8-GPU Resnet18 job, we compare the modeled job throughput using Synergy to the empirical results obtained by training the job for 2 epochs with varying memory allocations. As we see in Figure 5.4a, Synergy's estimations are within 3% of the empirical results, without having to actually run the model.

To further optimize profiling time, we observe that we do not require exact throughput values for a job with varying CPU allocations. We instead need a curve depicting the empirical job throughput. Therefore, instead of



(a) Memory Validation



(b) CPU validation

Figure 5.4: **Validating Synergy's optimistic profiling strategy**. The graphs compare the profiling results to empirical runs for CPU and memory demands in ResNet18

profiling the job for all possible CPU values, we pick discrete points for CPU profiling using the following algorithm. We start with the maximum CPU allocation and do a binary search on the CPU values to estimate job throughput.

If the profiled point resulted in a throughput improvement that is less than a fixed threshold (say 10%), then we continue binary search on the lower half of CPU values, else we profile more points on the upper half. The idea here is to empirically profile CPU regions that show significant difference in job throughput, while skip those regions with little to no improvement in throughput. We experimentally show the efficacy of our CPU profiling technique in Fig 5.4b for a 1-GPU ResNet18 job. We compare the normalized job runtime (wrt 1 CPU) using empirical results averaged over 2 epochs of the job and Synergy's optimistic profiling averaged over 50 iterations (approximately, a minute per profile). Synergy is able to mimic the empirical job performance very closely, in under 8 minutes (using just 8 CPU profile points instead of 24). We believe that this is a reasonable overhead as it is incurred only once per lifetime of the job, which typically runs for hours.

After profiling a job on arrival, the job along with its resource sensitivity matrix is enqueued into the main scheduling queue, from which the scheduling policy picks a set of runnable jobs every round. When the job is scheduled and run on the cluster, its performance is monitored by a local agent in the cluster, which then updates the empirical performance for a given CPU-memory allocation combination online, if it deviates from the value in the sensitivity matrix.

5.2.2 Scheduling mechanism

Synergy performs round-based scheduling. At the beginning of each scheduling round (a pre-defined time interval, say 15 minutes), Synergy identifies a set of runnable jobs from the scheduling queue that can be packed on the cluster in the current round duration. The set of runnable jobs is identified using a known scheduling policy such as FIFO, SRTF, LAS, or FTF. Using the resource sensitivity matrix, each of these jobs are packed onto the available servers in the cluster while satisfying the multi-dimensional resource constraints as opposed to simply performing a GPU-proportional allocation of CPU and memory resources.

Job demand vector. To pack the jobs onto servers, we first construct a job demand ector that indicates the GPU demand, and best-case CPU and memory requirements for the job. We identify the best-case values using the resource sensitivity matrix. We pick the minimum value of CPU and memory that saturates the job throughput.

Packing a job with multi-dimensional resource demands is analogous to multi-dimensional bin packing problem which is NP hard [207]. Therefore, we first evaluate the efficacy of a naive greedy scheduling mechanism as an approximation to tackle the multi-dimensional resource allocation problem.

5.2.3 Synergy-Greedy: Greedy Scheduling

A naive greedy multi-resource packing algorithm translates to a first-fit approximation of the multi-dimensional bin packing problem [74]. Given a job demand vector populated as described in §5.2.2, the greedy algorithm picks the next runnable job decided by the objective function (or the scheduling policy), and places it on the server that can satisfy the jobs demands in all dimensions. If no such server exists, the job is skipped over for this round and the next available job is scheduled. This greedy approach has two main drawbacks

- This scheme can result in CPU and memory resources being exhausted by jobs, while leaving GPU resources underutilized. This results in GPU fragmentation in the cluster. We experimentally show that GPU fragmentation in Synergy-GREEDY severely degrades cluster objectives such as average job completion times (5.4.5). GPUs being the most expensive resource in a GPU-cluster, thus have to be utilized efficiently to run DNN jobs.
- This scheme also hurts the fairness of the scheduling policy as some jobs can be skipped over for a long time if their resource demands cannot be satisfied in the cluster.

Synergy-GREEDY thus introduces two major problems in the cluster - fragmentation and violation of scheduling fairness. The challenge ahead of us is to design a scheduling mechanism that eliminates GPU under-utilization due to fragmentation, and upholds the fairness properties of the given scheduling policy, while performing multi-dimensional resource allocation.

However, any greedy heuristic solution we come up with, is not optimal in our problem setting as greedy solutions do not make global repacking decisions when new jobs arrive; it is quite expensive to do so. Therefore, one pertinent question is to understand how good is the allocation produced by our heuristic when compared to an optimal solution

To this end, we first formulate a theoretical upper bound on the optimal throughput achieved by the cluster given a set of runnable jobs and their resource sensitivity profiles. We then discuss the challenges associated with materializing the optimal allocation on a physical cluster and introduce Synergy-Tune, an empirically close-to-optimal heuristic solution.

5.3 Scheduling Mechanism

We first present our formulation of an optimal allocation that provides an upper bound on the achievable cluster throughput. This provides a solid basis for us to empirically evaluate how close to optimal the heuristic solution we design gets.

5.3.1 Synergy-Opt

Problem Definition. Our goal is to allocate CPU and memory to each job so as to maximize the throughput, while guaranteeing that each job makes at least as much progress as it would do if we allocate its *GPU-proportional share.*

Notation

- s: The number of machines or servers.

- For each machine $i \in [s]$, we denote G_i, C_i, M_i as the total GPU, CPU, and memory available on machine i.
- We denote the total GPU available across all machines by G. That is, $G = \sum_i G_i$. Similarly, we denote C, M as the total CPU and Memory capacity across all machines.
- We denote jobs by indices j. The GPU requirement of job j is denoted by g_j.
- For each machine $i \in [s]$, we denote C_g, M_g as the GPU-proportional allocation of CPU and memory. That is, $C_g = C_i/G_i * g_j$ and $M_g = M_i/G_i * g_j$.
- J_t: The set of jobs that needs to be scheduled in each *round*. J_t is the set of runnable jobs for this round, identified by the scheduling policy such that the total GPU requirements of jobs in J_t is at most the total GPU capacity of the cluster. In notation, $\sum_{j \in J_t} g_j \leq G$.
- n: We denote the number of jobs in the set J_t by n. In notation, $n = |J_t|$.
- W_j: We assume that resource sensitivity matrix for each job j is given as input. $W_j[c, r]$ denote the amount of progress made on job j per round if c units of CPU and r units of (RAM) memory are allocated to job j.
- With a baseline GPU-proportional allocation strategy the progress a job makes in each round is equal to $W[Cg, Mg]$.

5.3.1.1 A Upperbound on Throughput in an Optimal Solution

It is not hard to show that our problem is NP-hard. So, we resort to finding approximate solutions. Towards that we first find an *upperbound* on the throughput achievable by an optimal solution. We achieve that by formulating our problem as a linear program (LP). Moreover, we assume an *idealized setting*: We assume that all the CPU and memory available across all the machines is present in one (super) machine. That is, there is only one machine with C units of CPU and M units of memory. Note that in reality C units of CPU and M units of memory are spread across s machines. This means that in our throughput allocation, we do not take into account the effect of network when resources are allocated to jobs across multiple machines. Therefore, the true optimal solution of our problem can only do worse than the idealized allocation.

5.3.1.2 An LP formulation

We get an upperbound on the optimal allocation via an LP formulation. The variables of our LP are denoted by $y_{\{c,m,j\}}$, which should be interpreted as follows. If for a job $j \in J_t$, $y_{\{c,m,j\}} = 1$, then it means that in the LP solution c units of CPU and m units of memory are allocated. We further note that for every job j, there is a variable $y_{\{c,m,j\}}$ for *for every possible* allocation of CPU and memory. We consider these variables in the discrete space as identified by our resource sensitivity matrix.

- Our objective function is to maximize the throughput. We formulate it as follows using our LP variables.

$$\text{Maximize} \quad \sum_{j \in J_t} \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \tag{5.1}$$

Now, we enforce constraints such that LP solution is feasible in the idealized setting we talked about.

- First constraint we enforce is that the total CPU allocated to jobs is no more than the total capacity available:

$$\sum_{j \in J_t} \sum_{[c,m]} c \cdot y_{\{c,m,j\}} \leq C \tag{5.2}$$

- Similarly, we make sure that the total memory allocated to jobs is no more than the total capacity available:

$$\sum_{j \in J_t} \sum_{[c,m]} m \cdot y_{\{c,m,j\}} \leq M \tag{5.3}$$

- We want LP to allocate only one configuration of CPU and memory to each job.

$$\text{For each job } j \in J_t\text{:} \quad \sum_{[c,m]} y_{\{c,m,j\}} = 1 \tag{5.4}$$

- Finally, we want LP solution to be as good as the fair allocation.

$$\text{For each job } j \in J_t\text{:} \quad \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \geq W_j[C_g, M_g] \tag{5.5}$$

It is easy to verify that the optimal solution for our problem defines a feasible solution to our LP. On the other hand, as the LP solution can be fractional, in the sense $y_{\{c,m,j\}}$ variables can take fractional values, the optimum solution for LP can be no smaller than the true optimum solution, and thus always an upper bound on the throughput one can achieve for our problem. By enforcing the integrality constraints on $y_{\{c,m,j\}}$ variables one can getting a tighter upper bound. Indeed, in our experiments we solve this as a Integer Linear Program (ILP) where $y_{\{c,m,j\}}$ takes boolean values. For every job, we define the total CPU (c_j^*) and memory (m_j^*) allocated by the optimal ILP solution as follows.

$$\text{For each job } j, \text{ define} \quad c_j^* := c \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{5.6}$$

$$and \quad m_j^* := m \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{5.7}$$

5.3.1.3 Feasible Allocation on Multiple Machines

Recall that in the LP(1-5), we assumed that all the resources are present on a single machine. However, in reality these resources are spread across multiple machines. So, now we need to make an allocation taking into account this fact. We achieve that by solving another linear program.

Now our goal is the following:

- For each job $j \in J_t$, allocate g_j units GPU, c_j^* units of CPU, and r_j^* units of memory across s machines such that each job is fully scheduled on a single machine. We call (g_j, c_j^*, r_j^*) as the demand vector of job j.

Again, the above problem is an instance of multi-dimensional bin packing problem, so it is NP-hard. Instead, we try to reduce the number of jobs that get fragmented. So, our new goal is:

- For each job $j \in J_t$, allocate g_j units GPU, c_j^* units of CPU, and r_j^* units of memory across s machines such that the number of jobs that get fragmented is minimized.

A Feasible Allocation via Second LP

The variables of the second LP are denoted by $x_{i,j}$. Here index i denotes the machine and j denotes the job. The variables $x_{i,j}$ are interpreted as follows: if $x_{i,j} = 1$, it means that resources of job j (that g_j units of GPU, c_j^* units of CPU, and r_j^* units of memory) are allocated on machine i.

Now we are ready to find a feasible allocation minimizing the number of fragmented jobs.

- First constraint we enforce is that the total GPU allocated to jobs is no more than the total capacity available on the machine:

$$\text{For each machine } i \text{ in } [s]: \quad \sum_{j \in J_t} g_j \cdot x_{i,j} \leq G_i \tag{5.8}$$

- Next, we make sure that the total CPU allocated is no more than the total capacity available on the machine:

$$\text{For each machine } i \text{ in } [s]: \quad \sum_{j \in J_t} \sum_{[c,m]} c_j^* \cdot x_{i,j} \leq C_i \tag{5.9}$$

- Similarly, we make sure that the total memory allocated is no more than the total capacity available on the machine:

$$\text{For each machine } j \text{ in } [s]: \quad \sum_{j \in J_t} \sum_{[c,m]} r_j^* \cdot x_{i,j} \leq M_i \tag{5.10}$$

- We make sure that every job is allocated all the resources it demands:

$$\text{For each job } j \in J_t \quad \sum_{i \in [s]} x_{i,j} \geq 1 \tag{5.11}$$

- Finally, We make sure that variables are positive.

$$\text{For each job } j \in J_t \text{ and } i \in [s] \quad x_{i,j} > 0 \tag{5.12}$$

Using linear programming theory, we now prove a structural property about our LP that states that most of the variables are integral.

Theorem 5.3.1. Suppose we assume that no job demands more CPU, GPU or memory available on a single machine. Then, the total number of jobs that get fragmented in the LP (8-12) is at most 3s.

Proof. Let $\{x^*_{i,j}\}_{i,j}$ denote an optimal solution to the LP(8-12). We know from linear programming theory that for every LP, there is an optimal solution which is a *vertex* of the polytope. Let P denote the set of positive variables in the LP solution. That is set of $x_{i,j}$ such that $x_{i,j} > 0$. A vertex solution is defined by a linearly independent family of tight constraints. A tight constraint means that in the LP solution a constraint is satisfied with an equality (=). A tight constraint of the form $x_{i,j} = 0$, only leads to variables not in P. Therefore, we only we need to consider tight constraints of the form (8), (9), (10), and (11). Therefore, number of variables taking positive values in P is bounded by

$$|P| \leq 3s + n \tag{5.13}$$

The above equation is true because there is only 1 constraint of the type (8), (9), and (10) for each machine and there are s machines. Further more there is one constraint of type (10) and there are n jobs.

Now let N_1 denote the number of jobs that got fragmented in the LP (8-12) solution. Now each such job contributes at least 2 variables to P. This implies,

$$2N_1 + (n - N_1) \leq |P| \leq 3s + n \tag{5.14}$$

Therefore, $N_1 \leq 3s$, and it concludes the proof.

□

5.3.1.4 Challenges with operationalizing Synergy-Opt

While the allocations identified by Synergy-Opt provides an upper bound on the optimal cluster throughput, it is challenging to operationalize these allocations in the real world due to two main reasons;

1. Solving two LPs per scheduling round is a computationally expensive task. As cluster size and the number of jobs per round increases, the time to find an optimal allocation exponentially increases (§5.4.6)

2. The allocation matrix obtained with the second LP can result in fractional GPU allocations when jobs are split across servers; for instance, a valid allocation might assign 3.3 GPUs on server 1 and 2.7 GPUs on server 2 for a 6 GPU job. Realizing such an allocation requires a heuristic rounding off strategy to ensure non-fractional GPU allocations, as GPU time or space sharing, and its impact on job performance is considered beyond the scope of our work.

5.3.2 Synergy-Tune

We now describe Synergy-Tune, our heuristic scheduling mechanism.

Goal. Our goal is to design a scheduling mechanism that allocates multi-resource demand job onto available servers in the cluster each round. In doing so, we want to ensure that (1) We do not affect the fairness properties of the scheduling policy used. (2) The expensive GPU resources are not fragmented and underutilized.

Allocation Requirements. Synergy-TUNE's allocation must satisfy the following requirements.

- The GPU, CPU, and memory resources requested by a single-GPU job must all be allocated on the same server.
- A multi-GPU job can either be consolidated on one server, or split across multiple servers. In the latter case, the CPU and memory allocations must be proportional across servers. For instance, if the job demands (2GPU, 12 CPU, 300GB DRAM), then while splitting it across two servers, we need to ensure that each server gets (1GPU, 6CPU, 150GB DRAM). This is because, multi-GPU jobs train on a separate process on each GPU, and synchronize at regular intervals, i.e., after one or many iterations. The job performance will vary across processes if each GPU does not get the same ratio of resources, and will eventually proceed at the speed of the process with the lowest allocation of CPU and memory.

In a shared cluster where users share the available resources across their jobs, it is import to enforce fairness in terms of throughput achieved by individual jobs. We need to ensure that no job runs at a throughput lower than what it would have achieved if we allocated a GPU-proportional share of CPU and memory resources. Additionally, we need to respect the priority order of jobs identified by the scheduling policy. For instance, a FIFO scheduling policy can be implemented using a priority queue sorted by job arrival times. Our mechanisms must schedule the top n jobs from this priority queue in every round that fit the cluster. Understanding the term *fit* is important.

As we mentioned, a DNN job can run as long as the requested number of GPUs are available; irrespective of the amount of memory and CPU available. Therefore, Synergy-Tune identifies a set of runnable jobs as the top n jobs from the scheduling queue, whose GPU demands can be exactly satisfied by the available servers in the cluster. Synergy-Tune picks this runable job set irrespective of the job's other resource demands - which are fungible. Note that, unlike Synergy-Greedy, we do not skip over any jobs unless it cannot be scheduled (GPU demand cannot be met). Therefore, we never underutilize the GPUs when the cluster is at full load.

Next, Synergy-Tune greedily packs each of these runnable jobs along multiple resource dimensions on one of the available servers, with the objective to minimize fragmentation. To achieve this, Synergy-Tune sorts the runnable jobs by their GPU demands, followed by CPU, and memory demand. For each job j in order,

Synergy-Tune picks the server with the least amount of free resources just enough to fit the demand vector of j. If it is a multi-GPU job, then we find a minimum set of servers with sufficient GPU availability that can fit the job's demands in entirety. However, it is possible that the job cannot fit in the cluster along all dimensions. In such a case,

1. We check if the job's demand vector is greater than proportional share of resources, In this case, we switch the job's demand to GPU-proportional share and retry allocation.

2. If the job still does not fit the cluster, or if the job's demand vector was less than or same as GPU proportional allocation in step (1), then to ensure fairness, we cannot further reduce the current job's demand. In this case, we do the following.

    (a) We repeat step (1) ignoring the job's CPU and memory requirements. We find a server that can just satisfy the job's GPU requirements. We know by construction that there is atleast one job on this server, which is allocated more than GPU-proportional share of resources. We identify the job or a set of jobs (J_s) on this server by switching whom to GPU-proportional share, we can release just as much resources required by the current job j. We switch the jobs in J_s to fair-share and by design, job j will fit this server.

    (b) We continue this recursively for all runnable jobs.

In the worst case, all the running jobs in a round could be allocated GPU-proportional share of resources. Therefore, Synergy ensures that its allocations never degrades a job throughput to less than GPU-proportional allocation.

5.3.3 Discussion

While Synergy-Tune is simple to implement and has many appealing properties that it makes local changes, which in principle can be parallelized, and converges to allocations where every job strictly achieves as much throughput as its GPU-proportional allocation, one pertinent question is to understand how good is the allocation produced by Synergy-Tune compares to an optimal solution. One natural way is to compare Synergy-Tune to an allocation produced by an Integer Program (IP). In our experiments, we see that for smaller instances Synergy-Tune produces allocations that are very close to the optimum.

5.3.4 Implementation

We implement Synergy and an associated simulator in 8000 lines of Python code. Our scheduler is event-driven. There is a global event queue where job arrivals, schedule event, and deploy events are queued. These events are handled in the order of their arrival time. There is a priority job queue, where all the jobs arriving into the cluster are added. This queue is sorted by the priority metric decided by the scheduling policy; for instance, SRTF sorts the jobs in the order of job remaining time.

When a schedule event occurs, the scheduler collects a list of runnable jobs from the job queue and identifies the appropriate placement for these jobs for the following round, either using Synergy-Greedy, Synergy-Tune or Synergy-Opt. Then when a deploy event is triggered, these allocations are deployed on to the cluster. By default, every job requests for a lease update to continue running on the same server, if its allocations haven't changed for the next round [153]. The scheduler then either grants a lease update or terminates the lease for the job, adding it back to the job queue.

The scheduler and the DNN jobs interact via a thin API provided by the Synergy data iterator. DNN job scripts must be updated to call the Synergy iterator which is a wrapper around the default PyTorch [29] and DALI [21] iterators. The iterator handles registering the job with the scheduler, and appropriately sending lease updates, It also checkpoints the job to a shared storage if its lease is terminated. The iterator also synchronizes across GPU processes for a multi-GPU job to ensure that each process makes identical progress. We use `gRPC` [3] for communicate between the scheduler and the jobs.

We implement Synergy-Opt in `cvxpy` [73] for use in our simulator. The optimistic profiling module is also implemented in Python, and it profiles the incoming jobs hooked to the Synergy iterator, prior to the job's initial addition to the scheduling queue. This is a one time overhead, which is negligible when compared to the long runtime of our jobs.

5.4 Evaluation

In this section, we use a number of microbenchmarks, trace-driven simulations from production cluster traces, and physical cluster deployment to evaluate the efficacy of Synergy's scheduling and profiling mechanism. Our evaluation seeks to answer the following questions.

- Does Synergy's data-aware scheduling mechanism improve objective metrics such as makespan and average JCT in a physical cluster (§5.4.2) and in trace-driven simulations of large-scale clusters (§5.4.3, §5.4.4) ?

- How does Synergy-Tune and Synergy-Greedy perform with different workload splits and how well do they utilize available resources (§5.4.5)?
- How does Synergy compare to Synergy-Opt (§5.4.6) ?

5.4.1 Experimental setup

Clusters. Our experiments run on both a physical and a large simulated cluster. Our experiments are performed on state-of-the-art internal servers at Microsoft - these servers are part of larger multi-tenant research/production clusters. We run physical cluster experiments on a cluster with 32 V100 GPUs across 4 servers. Each server has 500GB DRAM, 24 CPU cores, and 8 GPUs. In all our experiments, fair-share CPU allocation is 3 cores per GPU and fair-share memory allocation is 62.5GB per GPU.

For simulated experiments, we assume two cluster sizes; a 128 GPU cluster across 16 servers and a 512 GPU cluster across 64 machines, where each machine resembles the physical server configuration mentioned above.

Models. Our experiments consider 10 different DNNs (CNNs, RNNs, and LSTMs) spanning across image classification (AlexNet, ResNet18, ShuffleNet, MobileNet, ResNet50), speech recognition (DeepSpeech), music classification (M5), language translation (GNMT), and language modeling (LSTM, Transformer-XL). We categorize these models by task (image, language, and speech) and assign a certain weight to these tasks in our traces, called the *split*. For in-

Task	Model	Dataset
Image	Shufflenetv2 [219]	ImageNet [180]
	AlexNet [117]	
	Resnet18 [91]	
	MobileNetv2 [181]	
	ResNet50 [91]	
Language	GNMT [208]	WMT16 [37]
	LSTM []	Wikitext-2 [136]
	Transformer-XL [63]	Wikitext-103 [136]
Speech	M5 [62]	Free Music [66]
	DeepSpeech [89]	LibriSpeech [165]

Table 5.4: **Models used in this work**.

stance, if the split for a given trace is (30,40,30), then the percentage of image, language, and speech models in the job trace is 30%, 40% and 30% respectively. All experiments are performed on PyTorch 1.1.0.

We run our physical and simulated experiments using both production traces from Microsoft Philly cluster [18] and traces derived using the Philly trace.

In the production trace, we use the job GPU demand, arrival time, and duration as is from the trace and fix an appropriate cluster size for simulation. We assign a model to each job from Table 5.4 based on a chosen workload *split*.

In the production-derived trace, we extract job GPU demand from the production trace and assign a model based on the chosen *split*. We then appropriately scale the job runtime and arrival for the chosen cluster size, while keeping the job duration distribution similar to the one in Philly trace.

We achieve this as follows:

- **Duration**. The duration of each job for GPU-proportional allocation is sampled from an exponential distribution: the job duration is set to 10^x minutes, where x is drawn uniformly from [1.5,3] with 80% probability, and from [3,4] with 20% probability similar to the trace duration used in prior work [153]. The iterations for this job is calculated using the assigned runtime and the per-iteration time obtained from the empirically profiled value.

- **Arrival**. We classify derived traces into two kinds based on the job arrival time : (1) a *static* trace where all the jobs arrive at the start of the workload, and (2) a *dynamic* trace, where the job arrival time is determined by a Poisson distribution at a rate λ.

Such a derived trace gives us the flexibility to vary the load on the cluster, the distribution of job duration and the composition of jobs in the workload. It thus helps evaluate our scheduling mechanism across a range of workload scenarios.

The experiments with the production Philly trace uses a 512 GPU cluster with a subrange of the trace containing 8000 jobs. The derived traces with varying job arrival rates uses a 128 GPU cluster. In both cases, we report the average metrics across a set of 1000 jobs in steady state.

For the physical cluster experiment, we choose a fixed arrival rate for the derived trace that keeps our cluster at *full load* (GPU demand of all runnable jobs > available GPUs in the cluster). For the simulated experiments, we vary the load λ on the cluster to evaluate its impact on cluster metrics. For the simulated experiments, we show results for two trace categories - (1) all jobs request single-GPU (2) multi-GPU jobs that request 1, 2, 4, 8, or 16 GPUs.

Policies and metrics. We evaluate Synergy against GPU-proportional for 4 different scheduling polices; FIFO, SRTF, LAS, and FTF. For a static trace, we measure makespan (time to complete all jobs submitted at the beginning of the trace) and for the dynamic job traces, we measure the average job completion time (JCT) of a subset of jobs in steady state (cluster at full load), and their CDF. This is same as the evaluation metrics used by prior related work [209, 133, 153].

5.4.2 End-to-End Physical Cluster Experiments

For the physical cluster experiments, we run a Synergy-Tune (*tune*) and GPU-proportional allocation (*proportional*) for two different workload traces. (1) A static production-derived trace of 100 jobs with a *split* (60,30,10), scheduled using FIFO and evaluated for makespan. (2) A dynamic production-derived trace with continuous job arrivals and a split of (30,60,10), scheduled using SRTF and evaluated for average and 99th percentile JCT. Both scenarios use an appropriately sized trace that keeps the cluster fully loaded. We compare the obtained results to that of the simulator by replaying the same trace. Additionally, we compare our metrics to the upper bound generated by the optimal solution, Synergy-Opt (*opt*). The results are shown in Table 5.5.

Policy (Metric)	**Workload Split**	**Mechanism**	**Time (hrs) Deploy**	**Time (hrs) Simulate**
FIFO (Makespan)	60-30-10	Proportional	16	15.67
		Tune	11.6	11.33
		Opt	-	11.01
SRTF (Avg JCT)	30-60-10	Proportional	4.81	4.52
		Tune	3.21	3.19
		Opt	-	3.06
SRTF (99 Percentile JCT)	30-60-10	Proportional	17.32	16.85
		Tune	8.59	8.54
		Opt	-	8.21

Table 5.5: **Physical cluster experiments**. This table sshows the comparison of makespan and average JCT, and 99th percentile JCT for two different traces; (1) a static trace with a workload composition of 60% image, 30% language and 10% speech models using FIFO scheduling (2) a dynamic trace with a workload composition of 30% image, 60% language and 10% speech models, using SRTF scheduling policy. Synergy-Tune improves makespan by 1.4×, average JCT by 1.5 × and 99th percentile JCT by 2×.

Synergy-Tune reduces the makespan of static trace by 1.4× when compared to a data-agnostic GPU-proportional allocation mechanism. For the dynamic trace, Synergy-Tune reduces average JCT of steady-state jobs by 1.5× while reducing the 99th percentile JCT of these jobs by 2× as shown in Table 5.5.

We compare the observed results from physical experiments to the same trace replayed on our simulator. As shown in Table 5.5, the difference between metrics in real and simulated clusters are less than 5%, demonstrating the fidelity of the simulator.

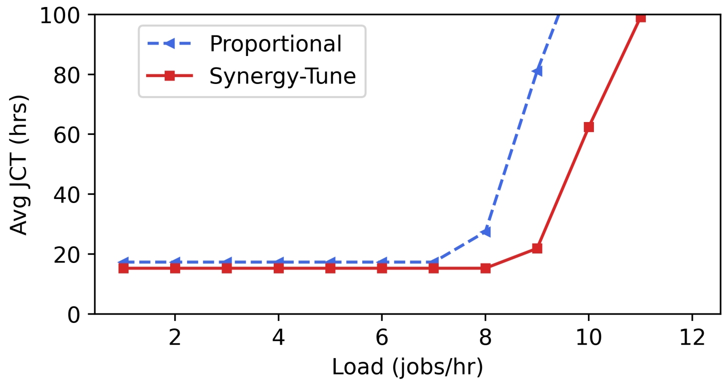
We also see from Table 5.5 that the cluster objectives achieved by

Synergy-Tune are within 4% of the optimal solution in this case. We do not deploy the optimal allocations due to the challenges enumerated in §5.3.1.4

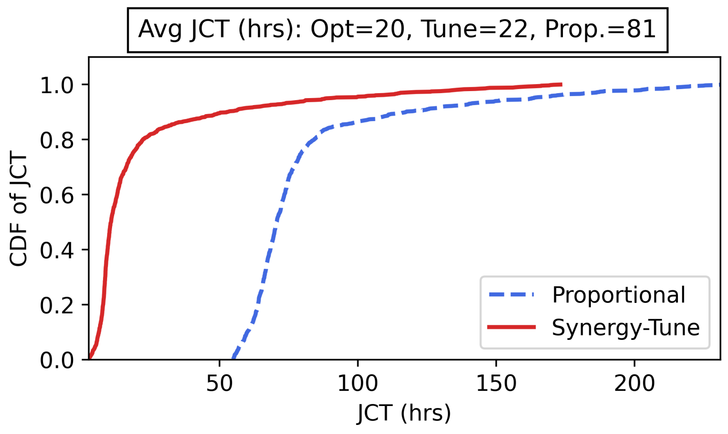
5.4.3 End-to-end results in simulation

We run simulated experiments on a cluster of 128 GPUs across 16 servers using production-derived traces. Each server is setup to mimic the real-world server used in our physical cluster experiments. We evaluate Synergy against GPU-proportional allocation mechanism for 4 different scheduling policies - FIFO, SRTF, LAS and FTF. We run dynamic workload traces, where jobs arrive continuously throughout the workload, at a rate *lambda*, as such a workload closely resembles real-world traces [18]. We show results for both single-GPU traces (where all jobs request 1 GPU) and multi-GPU traces (where jobs request multiple GPUs). Our metric of evaluation is the average Job Completion Time (JCT) of a set of 1000 jobs in cluster steady state.

Comparing Synergy-Tune to GPU-proportional allocation. We plot the average JCT for a set of jobs in steady state cluster for varying cluster loads. We show the results for three scheduling policies, FIFO (single GPU trace) in Figure 5.5, LAS (multi-GPU trace) in Figure 5.6, and SRTF (multi-GPU trace) as shown in Figure 5.7. In all cases, we assume a workload split of (20,70,10). We plot both average JCT and the CDF of job completion times for a specific cluster load in all the scenarios described above. For the multi-GPU traces, we split the CDF into those for short and long jobs to distinctly differentiate the tail of the distribution. We make three key observations.



(a) Avg JCT across load



(b) CDF of JCT at 9 jobs/hr

Figure 5.5: **Average JCT and CDF for FIFO**. Comparison of baseline GPU-proportional allocation with Synergy-Tune and Synergy-Opt.

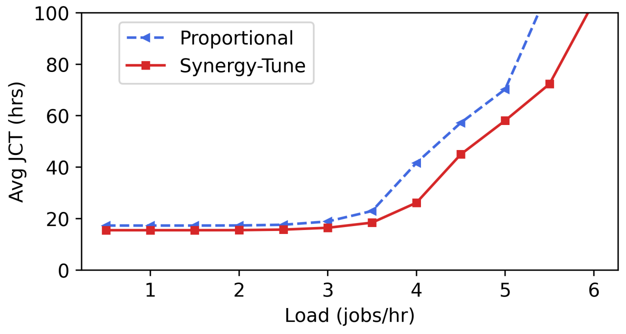
First, Synergy-Tune improves average JCT by up to 3.4× in the single-GPU trace, and up to 1.6× in the multi-GPU trace by speeding up resource sensitive jobs with disproportionate allocation. The improvement in average JCT is higher as the load increases, because at low load the cluster is not at full capacity. As load increases, jobs start to get queued and incur queuing delay before being scheduled on the cluster. Since Synergy significantly speeds up individual jobs using disproportionate resource allocation, pending jobs can get scheduled faster, thereby reducing their queuing delays. Therefore Synergy improves cluster metrics by both reducing qeuing delays and speeding up individual jobs. Second, Synergy-Tune is able to sustain a larger cluster load than GPU-proportional allocation. For multi-GPU scheduling with LAS, Synergy-Tune reduced the 95th percentile JCT of long jobs by 2×. Third, the average JCT achieved with Synergy-Tune is within 10% of the optimal solution in all cases.

Similarly, for FTF scheduling policy, Synergy-Tune observed 2.3× and 2× improvement in average JCT for a single-GPU and multi-GPU trace respectively.

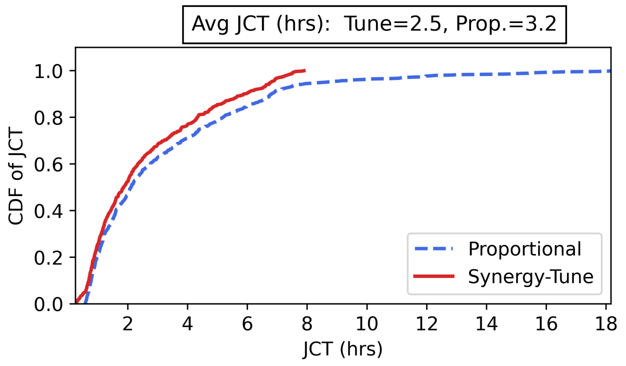
5.4.4 Simulation with production traces

We run simulated experiments on a cluster of 512 GPUs across 64 servers using a subrange of the Philly trace published by Microsoft [18]. We assume a workload split of (20,70,10) for this trail. Table 5.6 lists the average JCT with Synergy and GPU-proportional scheduling for three different scheduling policies. Across all policies, Synergy is able to reduce the average JCT compared to GPU-proportional scheduling due to better split of resources between jobs.

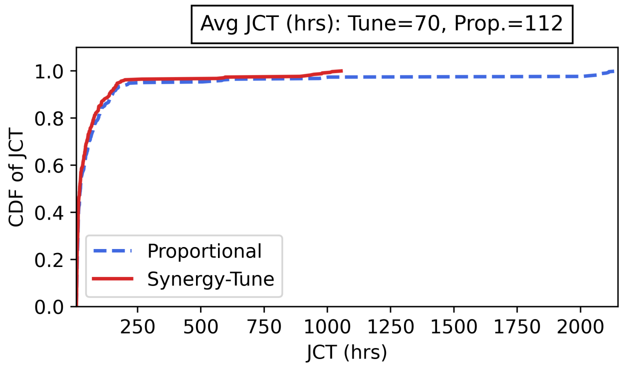
We further plot the CDF of job completion time for one of the scheduling policies, SRTF as shown in Figure 5.8. We split the set of 1000 monitored



(a) Avg JCT

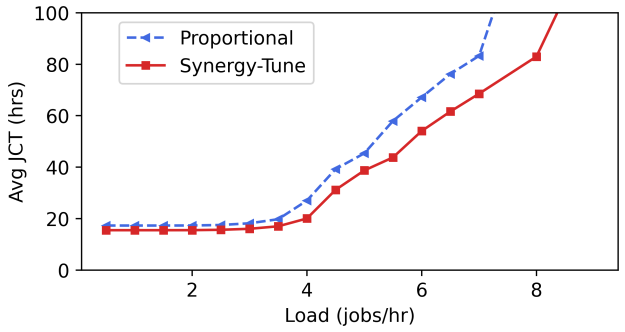


(b) CDF of JCT at 4 jobs/hr (short)

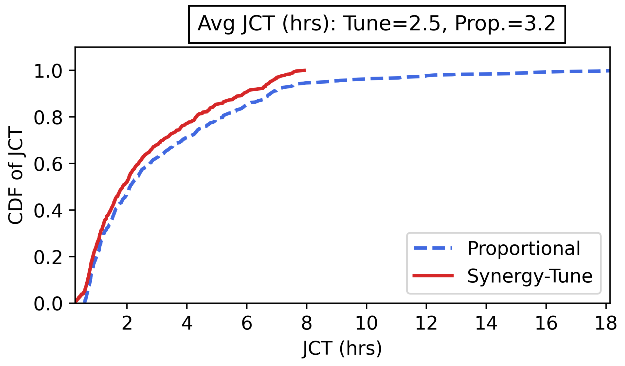


(c) CDF of JCT at 4 jobs/hr (long)

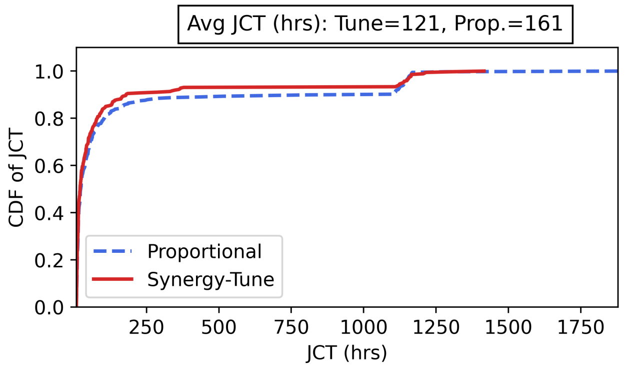
Figure 5.6: **Average JCT and CDF for LAS policy**.



(a) Avg JCT



(b) CDF of JCT at 5.5 jobs/hr (short)

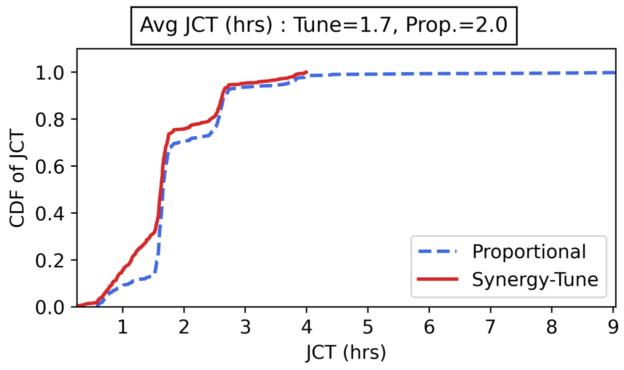


(c) CDF of JCT at 5.5 jobs/hr (long)

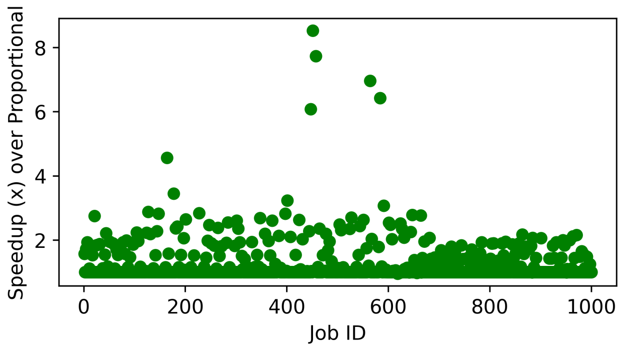
Figure 5.7: **Average JCT and CDF for SRTF**.

Policy	SRTF	LAS	FIFO
GPU-prop.	30	32	71
Synergy	26	28	62

Table 5.6: Average JCT with Synergy across different scheduling policies.



(a) CDF of JCT for short jobs



(b) JCT speedup across jobs

Figure 5.8: **CDF for SRTF**. This graph compares the baseline GPU-proportional allocation with Synergy-TUNE for SRTF scheduling policy for long and short jobs using a subrange of real-world Philly trace. The JCT of individual jobs improves by upto 9× with Synergy.

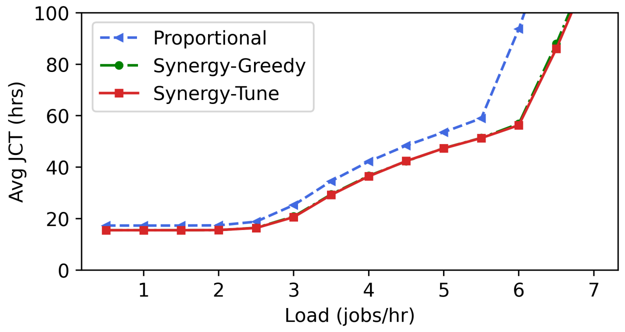
JCT (hrs)		**Short**	**Long**
Avg	Prop.	2	80
	Synergy	1.7	68
99p	Prop.	9	660
	Synergy	4	641

Table 5.7: **Cluster metrics**. This table shows the average and 99th percentile JCT for short and long jobs with SRTF

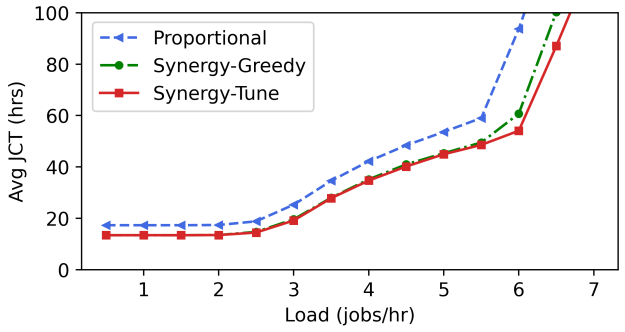
jobs into short (JCT $<$ 4 hrs) and long jobs. Synergy reduces the tail of the distribution by 2.2$\times$ for short jobs and the average JCT of both long and short jobs by 15% as shown in Table 5.7. For each of the 1000 monitored jobs, we plot the individual job speedup with repect to GPU-proportional scheduling in Figure 5.8b. We see that Synergy speeds up jobs by upto 9$\times$ using better resource allocations.

5.4.5 Impact of workload split

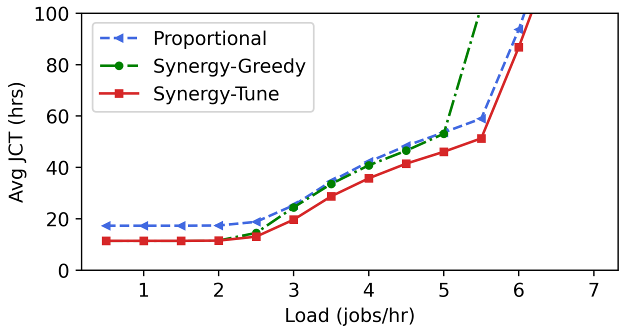
The workload split decides the percentage of resource sensitive jobs in the workload. As the percentage of speech and image models increase in the trace, there may not be enough spare CPU and memory resources to perform disproportionate allocation, as they are mostly CPU- and memory-hungry. Figure 5.9 plots the average JCT with varying load for 3 different workload splits with FIFO scheduling for multi-GPU jobs. As the percentage of resource-sensitive jobs increase, we observe that Synergy-Greedy breaks down, and ends up degrading JCTs significantly compared to a GPU-proportional alloca-



(a) Split=(20,70,10)



(b) Split=(33,33,33)



(c) Split=(50,0,50)

Figure 5.9: **Varying workload split**.

tion. This is because, the naive greedy technique results in resource fragmentation when the demand along CPU and memory dimensions are high, leaving several GPUs underutilized. Whereas, by the design of Synergy-Tune, it allocates at least as many resources required to achieve the throughput of GPU-proportional allocation; therefore, even in the worst case workload split shown in Figure 5.9c, where all the jobs are CPU- and memory-sensitive, Synergy-Tune performs as good as GPU-proportional allocation.

Resource utilization. Figure 5.10 plots the GPU allocation over time for the workload in Figure 5.9c at a high load of 5.5 jobs/hr where the cluster GPU demand is higher than 100%. While Synergy-Tune is able to sustain a higher load by finishing jobs faster at low cluster load, Synergy-Greedy severely underutilizes GPU resources throughout the workload, trading it off for higher CPU and memory allocation. At low loads as shown in Figure 5.11, GPU-proportional allocation only utilized 60% of the available CPU resources, while Synergy-Tune utilized it up to a 90%, resulting in up to 1.5× lower average JCT.

5.4.6 Comparison to Synergy-Opt

Calculating optimal allocations for every scheduling round with Synergy-Opt can be quite expensive, especially for large cluster sizes. We plot the time taken for per-round allocations for Synergy-Opt against that of Synergy-Tune for varying cluster sizes in Figure 5.12. For Synergy-Opt, the time to solve the allocation problem quickly escalates, exponentially with increas-

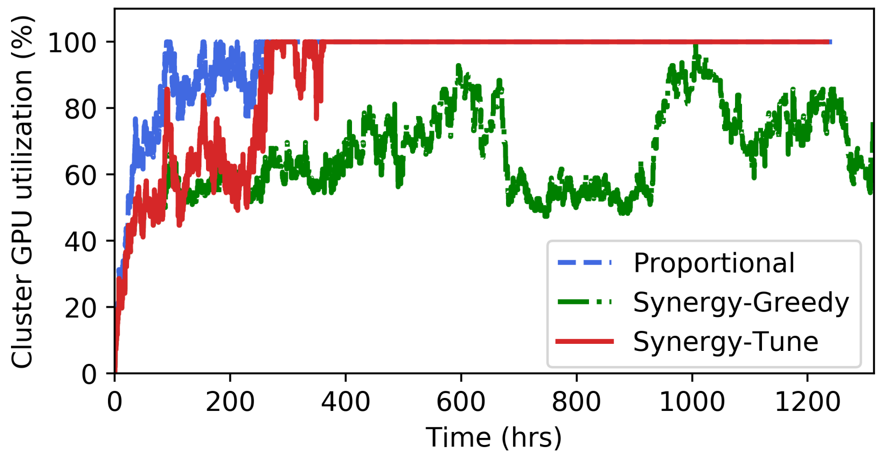


Figure 5.10: **Cluster GPU utilization**. This graph plots the GPU utilization in the cluster at load 5.5 jobs/hr in Figure 5.9c. Synergy-Greedy degrades average JCT due to GPU fragmentation and under utilization.

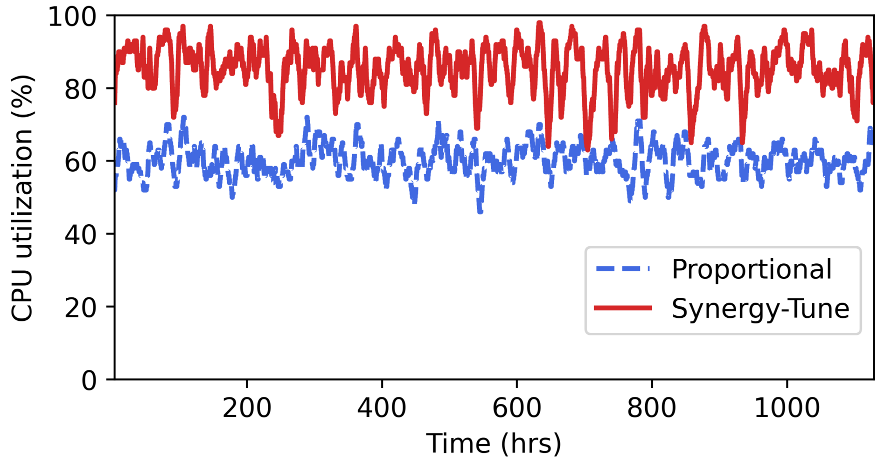


Figure 5.11: **CPU utilization**. This graph plots the CPU utilization in the cluster at low load. GPU-proportional scheduling under-utilizes the available CPU resources while Synergy is able to efficiently utilize spare resources to improve cluster metrics.

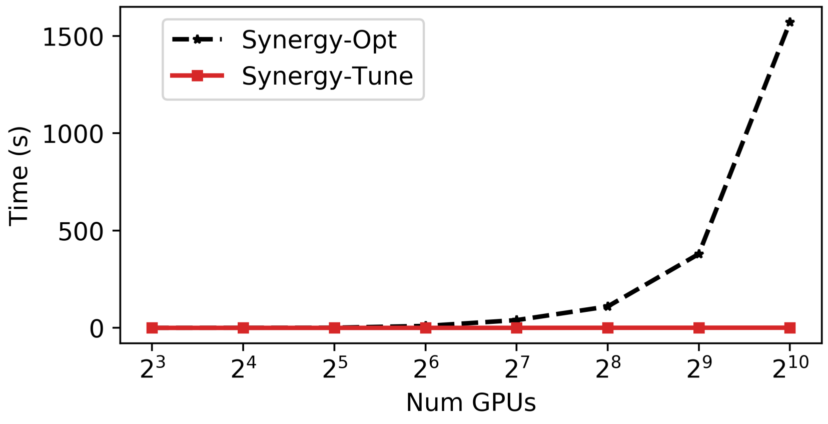


Figure 5.12: **Scalability of our scheduling mechanism**.

ing cluster sizes, while that for Synergy-Tune is hardly a second. We also show experimentally that the allocations given by Synergy-Tune are close to those estimated by Synergy-Opt in §5.4.2 and §5.4.3. At a cluster size of 128 GPUs used in our experiments, Synergy-Tune converges at allocations that are within 10% of the optimal value, 200× faster than Synergy-Opt.

5.4.7 Comparison to DRF and Tetris

Big data schedulers like Dominant Resource Fairness (DRF) [77] and Tetris [83] have explored multi-dimensional resource allocation for map-reduce jobs. DNN jobs have different properties when compared to big-data jobs. DNN jobs are gang-scheduled, meaning they can run only when all the GPUs requested by them are available on the cluster at once. Further, the auxiliary resource requirements like CPU and memory are fungible unlike the GPU demand. DRF and Tetris assume resources to be statically allocated throughout the lifetime of a job, whereas Synergy assumes these resources to be fungible

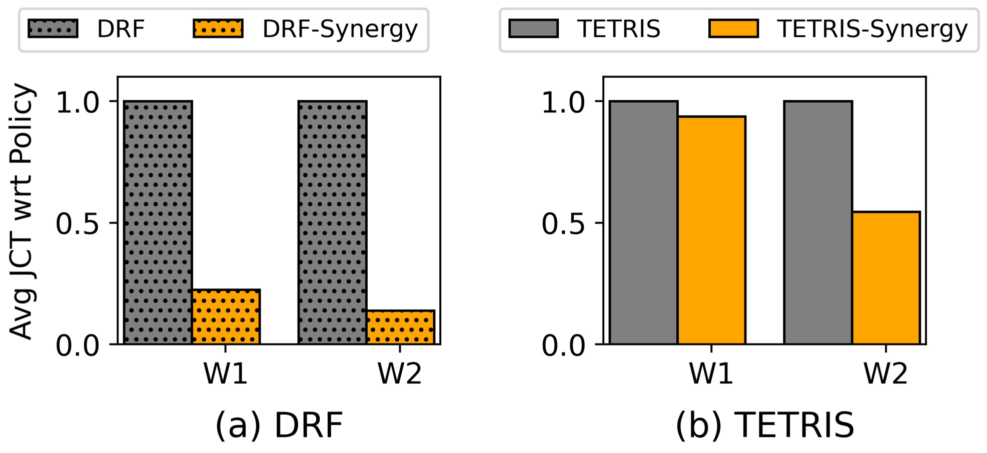


Figure 5.13: **Comparison to big data scheduling policies**. This graph compares existing multi-resource allocation policies for big data workloads such as DRF and TETRIS against Synergy for two different workload compositions (lower the better).

and could result in varied allocations throughout the lifetime of a DNN job. Furthermore, profiling the DNN job's resource demands is unique to Synergy; big data schedulers assume that the job request already encodes resource demands across all dimensions. To evaluate Synergy against these policies, we assume that the best-case resource requirement for CPU and memory is fed as input to the bigdata scheduling policies using Synergy's profiling mechanism.

On a cluster of 128 GPUs, we evaluate these policies on two different workload compositions : W1 (20,70,10), and W2 (50,0,50) and compare the naive policy with its Synergy-variant, which allows resource tuning. W1 represents a workload split with a good mix of resource-sensitive as well as resource-insensitive jobs. W2 is a workload dominated by resource-sensitive jobs, which is one of the worst-case scenarios for multi-dimensional scheduling as it could lead to GPU fragmentation (explained in §5.4.5)

We plot the results in Figure 5.13. Tuning resource allocation across jobs using Synergy reduced the average JCT of DRF by 7.2× and that of Tetris by 1.8× for the workload split W2. This is because Synergy is able to allocate auxiliary resources in a fungible-manner every round, whereas the big-data scheduler's static allocations performs similar to greedy techniques, resulting in GPU fragmentation, and thereby degrading the overall cluster metrics. Synergy performs the best in each scenario as it uses the best-case resource demands of jobs to perform fungible, disproportionate allocation.

5.5 Limitations and Discussion

We now discuss Synergy's limitations and scope for future work.

Tradeoff between consolidation and allocation. When multi-GPU jobs are split across physical servers, they may incur a penalty due to network communication [151, 209]. DNN jobs therefore prefer consolidation. In Synergy, we assume that no more than a server's worth of CPU or memory resources can be allocated to a job if its GPU demands can be satisfied by one server. However, we find that some jobs may benefit from giving up consolidation if the throughput gain due to increased CPU or memory allocation is higher than the penalty due to splitting. It is an interesting future direction to explore the trade off between consolidation and allocation, while taking into account the network overhead.

Dataset locality. When DNN training jobs are submitted to a cluster scheduler, the input datasets for the job reside on a remote storage [7, 188]. When training begins, the dataset is first downloaded locally on the machine, and then loaded into the server memory. The former is constrained by the remote storage and network bandwidth, while the latter is restrained by the local storage bandwidth.

Migrating large datasets, or downloading them from cloud storage is expensive. Therefore, jobs with large datasets that are sensitive to storage bandwidths must be placed on the same server until completion, when possible. In this work, we assume that the datasets are present locally on each server. We leave it to future work to study the impact of remote storage fetch and dataset locality on scheduling decisions.

Heterogeneous server resources. In this work, we assume that the server resources in a cluster are homogeneous; *i.e.,* each server is identical. If we relax this assumption, we need to carefully evaluate how the performance of a DNN job changes with heterogeneity in GPU generation; different GPUs exhibit heterogeneous performance behavior across model architectures. The scope of this work was to demonstrate the importance of resource-sensitivity awareness in scheduling. Therefore, we leave it to future work to extend Synergy to be aware of accelerator performance heterogeneity.

5.6 Summary

This chapter introduced Synergy, a resource-sensitive DNN cluster scheduler. Synergy is based on the insight that, not all jobs exhibit the same level of sensitivity to CPU and memory allocation during DNN training. Our experiments how that Synergy can reduce average JCT by upto 3.4× compared to a GPU-proportional allocation strategy using the same scheduling policy.

Chapter 6

CheckFreq: Mitigating Checkpoint Stalls when Training with Interruptions

In this chapter, we discuss how to optimize the output data pipeline i.e., checkpointing in DNN training. Due to the large runtime of DNN training, the model weights and optimizer state (collectively, model state) are occasionally written to persistent storage, for fault tolerance; else, an interruption to the job due to process failure, or node crash can wipe out all the job state, resulting in loss of several hours of GPU work. This is termed *checkpointing*. Traditionally, models are checkpointed at epoch boundaries [140].

With the recent trend in growing size of datasets [36, 39, 119], and larger, complex model architectures [41, 161, 48], DNN epoch time and overall training time is increasing. Therefore, it is critical to frequently checkpoint training progress, at a finer granularity than epochs - at iteration level.

In this chapter, we explore how to perform fine-grained checkpointing automatically in a model- and hardware-agnostic manner, without intrusive changes to the training workload using a framework we introduce - CheckFreq[1].

[1]This Chapter is based on the work Frequent, Fine-Grained DNN Checkpointing, published in FAST 21 [146]

Specifically, we aim to provide (1) frequent iteration-level checkpointing to persistent storage, (2) ability to resume training at iteration boundaries, and (3) ensure low data stalls due to checkpointing.

The rest of this chapter is organized as follows. We first analyzes the state of DNN checkpointing today and highlight the need for fine-grained checkpointing and the challenges involved in achieving it (§6.1). We then present the design and implementation of CheckFreq, an automatic, fine-grained checkpointing framework for DNN training that exploits the DNN computational model to provide low-cost, pipelined checkpointing (§6.2) FI-nally, we show experimental results demonstrating the efficacy of CheckFreq in reducing the recovery time from hours to seconds, across a range of models and hardware configurations (§6.3).

6.1 The Current State of Checkpointing

We analyze the current state of checkpointing in popular open source ML training frameworks like PyTorch [29], TensorFlow [38], and MxNet [52]. We analyze training workloads from MLPerf submissions v0.7, and the official workloads released by NVIDIA, TensorFlow and PyTorch. We find that checkpointing in open source ML training frameworks is incorrect and inefficient.

- **Correctness**. The checkpointing mechanism used in the training scripts could result in loss or corruption of checkpoint files in the event of job failure or interruption.

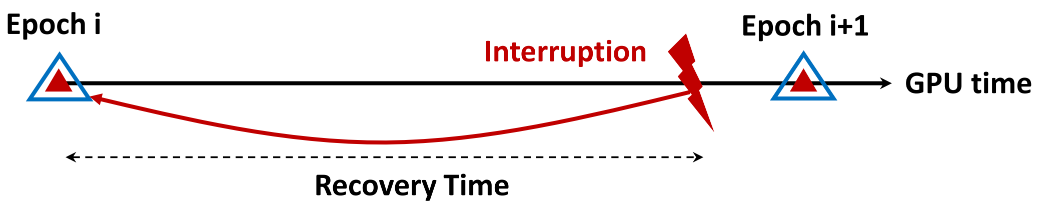


Figure 6.1: **Recovery time across models**. The amount of GPU work lost and has to be *redone* on recovery is termed the recovery time.

- **Efficiency**. Checkpointing is inefficient. The frequency of checkpointing is determined in an ad-hoc fashion, typically at epoch-boundaries which results in loss of several hours of GPU time for recovery. Furthermore, there is lack of support for checkpointing at fine granularity; existing data iterators do not support resuming training state at iteration boundaries and results in high checkpoint stalls.

6.1.1 Checkpointing is Incorrect

Corruption due to overwrites. Some of the official training workloads maintained by PyTorch [176], overwrite the same checkpoint file at the end of each epoch to reduce storage utilization. However, this exposes the risk of corrupting the checkpoint file in the event of a crash during the checkpoint operation. Prior work [170] has shown that different filesystems treat overwrites differently; a crash could result in non-atomic data update in the writeback mode of ext3 resulting in data corruption, while it could truncate the file on ext4, resulting in data loss. In either case, the checkpoint file becomes unusable; training has to restart from the first epoch.

The checkpoint file may not persist. Analyzing the primitives used by training frameworks for checkpointing, such as `torch.save` reveal that they do not `fsync()` the checkpoint file. We verified that this can lead to data loss. Moreover, naively performing frequent synchronous `fsync()` affects training performance significantly (§6.3.3.1).

To validate this, we perform a simple crash test, where we checkpoint random tensors to local storage on a VM, and force shut the VM after the checkpoint operation successfully returns, to simulate a crash. As expected, we observe that the checkpoint file is lost when the VM restarts. While the importance of `fsync()` for persistence is widely acknowledged and understood by the storage community, DNN training has unfortunately failed to embrace this knowledge.

Infact, despite issuing `fsync()`, some filesystems may fail to persist the file reliably on disk due to crash-consistency bugs. These situations are out of the scope of this dissertation. More details on finding crash-consistency bugs in filesystems can be found in our prior work [141, 142].

6.1.2 Checkpointing is Inefficient

Checkpointing is performed sparingly in an ad-hoc fashion. There is no systematic checkpointing policy in the training jobs; checkpointing interval is chosen in an ad-hoc fashion. For example, some jobs do not checkpoint during training, while some others start checkpointing only after a large number of epochs (60% of training) have elapsed. In general, we observe that checkpointing is typically performed at epoch boundaries, providing only modest fault-tolerance; in the event of a job interruption, the training will resume from the last completed epoch, which potentially loses several hours of GPU training time that has to be redone. For instance, when ResNext101 is trained using ImageNet on a V100 GPU, *two* hours of GPU time is lost on average if the job is interrupted (§6.3.5).

A naive frequent checkpointing schedule results in checkpoint stalls. Providing higher fault-tolerance requires checkpointing to be performed more frequently than at epoch boundaries; *i.e.,* at iteration boundaries. However, naively increasing the frequency of checkpointing introduces a large checkpoint stall in training. Since model weights are constantly updated between iterations, checkpointing requires the training to briefly pause to capture the model weights accurately. We term this overhead (i.e, the time GPU is idle, waiting for the checkpoint to complete) as the *checkpoint stall.* Therefore, it is crucial to find the correct checkpointing frequency given a DNN (because the size of checkpoint varies from 100MBs to 100GBs across DNNs), and the storage bandwidth, to minimize checkpoint stalls.

Violating the data invariant during training can affect model accuracy. Each epoch performs a full pass over the dataset, in a random order and holds the invariant that *each data item is seen exactly once per epoch.* One of the benefits of checkpointing at epoch boundaries is that, the data iterator state need not be persisted, as it is reset at the end of epoch. Checkpointing at a finer granularity (i.e. at iterations), requires infrastructure support to resume the state of data iterator as well. We note that the support to persist iterator state exists in some custom dataloaders of NLP models which do not perform random pre-processing operations for every batch. However, for image and video models that apply random transformations on the input data every batch, the existing dataloaders in PyTorch, MxNet, and state-of-the-art data pipelines like NVIDIA's DALI are *not resumable* at iteration boundaries. As a result, they violate the data invariant in the presence of interruptions, resulting in upto the 13% drop in accuracy for popular models ResNet18 (Fig 6.6).

6.1.3 Summary

In summary, we observe that the checkpointing mechanism today is incorrect; resulting in potential checkpoint data loss or corruption. Additionally, the checkpointing policy is ad-hoc; there is no systematic way of determining how frequently one must checkpoint, to both minimize recovery time and incur low checkpoint stalls.

The solution to minimize recovery time is to perform frequent, iteration-level checkpointing. However, performing correct and efficient fine-grained checkpointing is challenging. We need (1) low-cost checkpointing mechanisms, (2) light-weight, resumable data iterators that preserve the model accuracy, and (3) a way to systematically determine the frequency of checkpointing.

Technique	***Benefits***
Checkpointing mechanism (How to checkpoint?)	
2-phase checkpointing	Splits checkpointing into two phases and pipelines them carefully with compute to make checkpoints cheap
Recoverable data iterator	Maintains data invariant, allows resuming training at iteration boundaries without affecting accuracy
Checkpointing policy (When to checkpoint?)	
Systematic online profiling	Automatically determines checkpointing frequency, cognizant of model characteristics
Adaptive rate tuning	Dynamically tunes checkpointing frequency to reduce overhead due to interference

Table 6.1: **Overview of techniques used by CheckFreq**.

6.2 CheckFreq: Design and Implementation

We present the goals of CheckFreq and the recovery guarantees it provides. We then present an overview of the overall architecture of CheckFreq, and discuss the techniques used by CheckFreq to achieve the enlisted goals.

6.2.1 Assumptions and Goals

CheckFreq focuses on optimizing checkpointing, which is by far the predominant way in which DNN training jobs recover from failures. While it is possible to use transparent checkpointing techniques such as CRIU [2] to backup the entire VM state, our work focuses on the dominant approach to DNN fault-tolerance; framework-assisted checkpointing of model state. CheckFreq focuses on data-parallel training. We discuss the applicability of Check-

Freq to model- and pipeline-parallelism in §6.4.

CheckFreq aims to achieve the following goals:

Correctness. CheckFreq aims to provide frequent, iteration-level checkpointing that is consistent, and persistent.

No impact on model accuracy. CheckFreq aims to not impact the statistical efficiency of the model by ensuring that the data invariant holds when training resumes after interruption.

Automatic frequency selection. CheckFreq aims to determine and tune the frequency of checkpointing *automatically* based on the model being trained, and the training environment (GPU gen, storage type, iteration time). Checkpointing frequency influences the recovery time, *i.e.,* time to bring model state to what it was prior to the interruption.

Low checkpoint stalls. CheckFreq aims to reduce checkpoint stalls during training, so that there is low runtime overhead to frequent checkpointing (*e.g.,* <5%).

Minimal code changes. CheckFreq aims to require minimal changes to the training code to automate checkpoint management and restoration.

6.2.2 CheckFreq Recovery Guarantees

An interrupted job resumes training from the latest available checkpoint on disk. In the traditional epoch-based checkpointing, irrespective of when the job is interrupted, training resumes from the previous epoch boundary as shown in Fig 6.1. If a job performs n iterations per epoch and takes time t_i per iteration, then the average recovery time R_{avg} for this job is :

$$R_{avg} = \frac{n}{2} * t_i$$

This is because, when interrupted in the middle of an epoch, work done so far in the epoch must be redone when resumed, as the state is reset to the end of previous epoch. Thus, recovery time R for epoch-based checkpointing is bounded by:

$$0 \leq R \leq n * t_i$$

Note that $n * t_i$ is the duration of an epoch; it can be as large as a few hours. CheckFreq aims to provide a tight bound on recovery time and takes a more fine-grained approach to checkpointing at iteration boundaries. CheckFreq guarantees that *there is at most one ongoing checkpoint operation in the system at any point in time*. When interrupted, it *rolls back at most one checkpoint* - either the last initiated checkpoint (if it completes), or the one prior as shown in Fig 6.2. If the frequency automatically determined by CheckFreq is k iterations, then CheckFreq guarantees that the recovery time R is bounded by

$$0 \leq R \leq 2 * k * t_i$$

$$R_{avg} = k * t_i \quad (k << n)$$

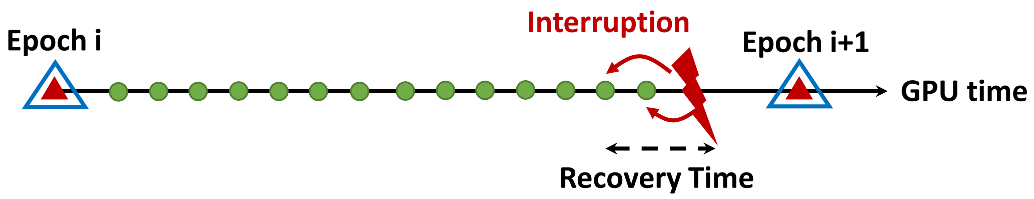


Figure 6.2: **Bounding recovery time**. CheckFreq guarantees that training rolls back at most one checkpoint.

The chosen checkpointing frequency k is $100 - 300\times$ less than n, as we show later in evaluation (§6.3.4), thereby resulting in orders of magnitude reduction in recovery time compared to epoch based checkpointing.

6.2.3 Design

We now present an overview of the architecture of CheckFreq and how it uses various techniques to provide frequent checkpointing at a bounded cost described in §6.2.2. Table 4.2 lists the different techniques used by CheckFreq and the benefit of each technique.

Overview. The architecture of CheckFreq is shown in Figure 6.3. CheckFreq has three major components; a recoverable data iterator that returns a minibatch of data to the training job, a feedback-driven checkpointing policy that determines when to trigger a checkpoint, and a low-cost checkpointing mechanism that is split into a `snapshot()` and a `persist()` phase. CheckFreq monitors the runtime overhead incurred in each checkpoint interval; this is used as feedback to dynamically tune the checkpointing frequency to ensure that the runtime overhead does not exceed a user-given limit p (*e.g.*, 5%). When interrupted, CheckFreq restores the latest available checkpoint

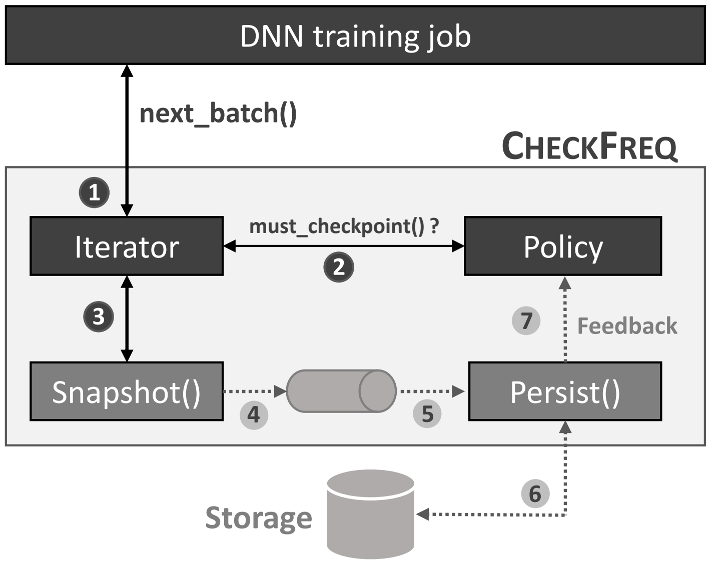
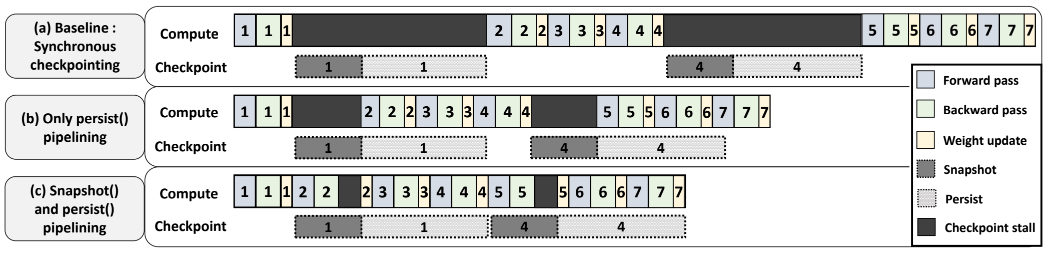


Figure 6.3: **Training with CheckFreq**. CheckFreq's policy determines the checkpointing frequency. The checkpointing mechanism then snapshots and persists the model and iterator state at the identified frequency in a pipelined manner. If a failure occurs, CheckFreq rolls back the model and iterator state to the latest available checkpoint and resumes training.

and resumes training. We describe each component in detail below.

6.2.3.1 Checkpointing Mechanism

DNN checkpointing today is performed synchronously; training is paused until the checkpoint operation is complete. However, synchronous checkpointing introduces large *checkpoint stalls*, which results in large runtime overhead if performed frequently. In other words, the cost of a checkpoint (T_c) is high for synchronous checkpointing. For example, consider a policy that checkpoints every three iterations. The model state is written to disk after the weight



(a) Baseline : Synchronous checkpointing
(b) Only persist() pipelining
(c) Snapshot() and persist() pipelining
Compute
Checkpoint
Forward pass
Backward pass
Weight update
Snapshot
Persist
Checkpoint stall

Figure 6.4: **Pipelining checkpoint with compute**. This figure contrasts three checkpointing mechanisms, when checkpointing is performed every 3 iterations. (a) performs checkpointing synchronously and incurs a high checkpoint stall. (b) takes a snapshot of the model state synchronously but pipelines disk IO (`persist()`) with compute, allowing it to proceed in the background. CheckFreq takes a more nuanced approach by carefully pipelining `snapshot()` with the subsequent iteration's forward and backward pass and incurs lower checkpointing stalls as shown in (c)

update phase which updates weights based on the gradients computed in the backward pass. As shown in Figure 6.4a, the checkpoint cost is incurred in the critical path, resulting in high checkpoint stalls, which can significantly slow down the end-to-end training time. To mask such high checkpoint costs within an overhead p, checkpointing needs to be performed infrequently, which in turn results in high recovery cost.

Two-phase checkpointing. CheckFreq aims to reduce the recovery cost in the event of an interruption by reducing checkpoint stalls. To achieve low checkpoint cost, CheckFreq introduces a *DNN-aware* two-phase checkpointing mechanism. CheckFreq splits checkpointing into two phases; `snapshot()` and `persist()` and pipelines each phase with computation. The main insight behind CheckFreq's two-phase checkpointing is that it exploits the DNN computational model (chapter 2) to pipeline checkpointing operations on modern accelerators such as the GPUs.

1. **Phase 1 : `snapshot()`.** The first is a `snapshot()` phase, performed after the weight update step of the iteration. Here, a copy of the model state is captured in memory, so that it can be written out to storage asynchronously. Since the model state resides in GPU memory, `snapshot()` involves copying the model parameters from GPU to CPU memory. Performing this operation synchronously in the critical path results in non-trivial `snapshot()` overhead as shown in Figure 6.4b. Therefore, CheckFreq carefully *pipelines* `snapshot()` with compute.

   Pipelining `snapshot()` with compute has to be performed cautiously to ensure consistency of model parameters and preserve correctness of Stochastic Gradient Descent (SGD), which is a popular optimization technique used by learning algorithms. Naively pipelining them can result in an inconsistent snapshot that contains part of the weight updates from one iteration and the rest from the other. CheckFreq exploits the *DNN learning structure* to achieve correct, pipelined snapshots.

   We observe that the learnable model parameters are updated in GPU memory after the backward pass of an iteration; in a step called the *weight update*. Therefore, we can pipeline `snapshot()` of iteration i with compute, until the weight update of iteration $i+1$. If `snapshot()` does not complete by then, then iteration $i+1$ waits until the ongoing `snapshot()` successfully completes as shown in Figure 6.4c. This tight coupling is required to ensure a consistent snapshot; else we might capture a state that is partially updated by the subsequent iteration that in turn affects the correctness of the learning algorithm [135].

GPU-based `snapshot()`. Although `snapshot()` is pipelined with compute of the following iteration, it may result in checkpointing stalls in cases where it is not possible to completely hide the cost of copying model state from GPU to CPU. Therefore, CheckFreq further optimizes this operation using a GPU-based `snapshot()` when feasible. We observe that the cost of performing a `snapshot()` in GPU memory is an order of magnitude cheaper than performing it to CPU memory, as the latter involves a GPU to CPU copy in the critical path. Therefore CheckFreq takes the following approach.

(a) When spare GPU memory is available in the training environment to hold a copy of the snapshot, we `snapshot()` in the GPU on GPU memory. The `persist()` phase then asynchronously copies the snapshot to CPU memory and then to disk.

(b) If not, CheckFreq snapshots directly into CPU memory. This can introduce stalls in critical path.

(c) CheckFreq adjusts the frequency of checkpointing appropriately to minimize the overhead of `snapshot()`, which can be especially large in (b), and stalls in `persist()`.

2. **Phase 2 : `persist()`.** The second phase in checkpointing is the `persist()` phase which asynchronously writes the snapshot to persistent storage similar to well explored asynchronous checkpointing techniques [187, 158, 198,

157]. However, to provide bounded rollback guarantees discussed in §6.2.2, `persist()` is *tightly coupled with compute*. CheckFreq performs the `persist()` operation as a background process; and monitors its progress. When a subsequent checkpoint is triggered as determined by the policy, the progress of the ongoing `persist()` operation is checked. If the `persist()` has not completed, then the compute process waits until the ongoing checkpoint operation is complete. This ensures that there is at most one ongoing checkpoint operation at any point in time, and if the job is interrupted, it rolls back to at most one prior checkpoint.

While it may be tempting to abandon an ongoing checkpoint if the next one is triggered, it is a tricky and risky operation. Suppose we abandon the current checkpoint and begin writing the next one, a failure at this point may end up losing both the checkpoints. This could be a chain reaction; a failure could result in rolling back to a significantly old checkpoint if all the recent ones were abandoned, resulting in a high recovery time. Since CheckFreq aims to guarantee that we roll back to at most one prior checkpoint, it does not abandon any running checkpoints.

Resumable light-weight data iterator. The DNN training workload interacts with CheckFreq using a thin API provided by a data iterator. The function of a data iterator in DNN training is to return a pre-processed batch of data items to the GPU, such that the data invariant holds - *each epoch processes all the data items exactly once, in a random order.* While the native iterator in PyTorch and those provided by state-of-the-art data pipelines like DALI [21] support this in the common case, they lack *resumability* if the training is interrupted.

For example, consider a dataset with eight data items from 1 – 8. In an epoch, the order of data items processed could be as shown in Fig 6.5a. Assume that we checkpoint the model state at the end of every iteration which processes *one* data item. If training is interrupted in the middle of this epoch, the data iterator loses state, and resumes with a random shuffled order of the dataset as shown in Fig 6.5b, resulting in data items being repeated and missed in a epoch, violating the data invariant.

CheckFreq's data iterator uses the following techniques to support resumption:

- It shuffles data items every epoch using a seed that is a function of the epoch number. Therefore, to recreate the same shuffle order, it is sufficient to persist the current epoch ID, and the number of data items processed so far (which makes iterator checkpointing lightweight).
- When training resumes, the iterator reconstructs the shuffle order, and deterministically restarts from where it left off at the last checkpoint as shown in Fig 6.5c.

Summary. Two-phase checkpointing mechanism along with the resumable data iterator provides correct, low-cost checkpointing. The next important question to answer is, *how frequently should we checkpoint the model?*

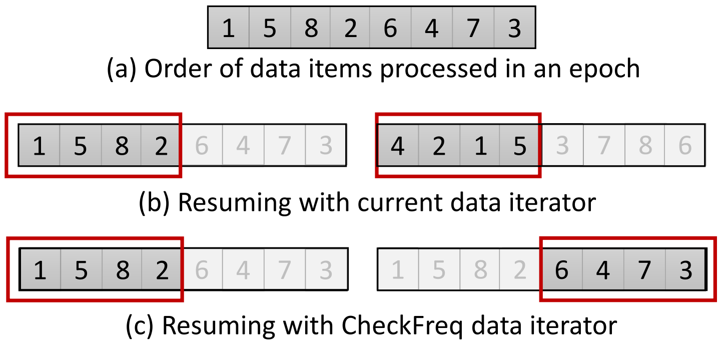

1 5 8 2 6 4 7 3
(a) Order of data items processed in an epoch
1 5 8 2 6 4 7 3
4 2 1 5 3 7 8 6
(b) Resuming with current data iterator
1 5 8 2 6 4 7 3
1 5 8 2 6 4 7 3
(c) Resuming with CheckFreq data iterator

Figure 6.5: **Resuming iterator state**. When iterator state is not resumable, an epoch might miss data items when job is interrupted (items 3,6,7 are missed in b). CheckFreq (c) ensures that training resumes from exactly where it left off.

6.2.3.2 Checkpointing Policy

To perform automatic, iteration-level checkpointing, we must determine the frequency at which checkpointing is performed. On one hand, we can checkpoint after every iteration, providing low recovery cost but possibly high runtime overhead. On the other hand, we can perform coarse grained checkpointing at epoch boundaries, resulting in high recovery cost but low runtime overhead. An effective checkpointing policy must find the right balance between recovery cost and runtime overhead, minimizing both. The main idea behind CheckFreq's checkpointing policy is to initiate checkpoints every k iterations (called the checkpointing frequency), such that the overhead of one checkpointing operation can be amortized over k iterations. While prior work in HPC have explored ways of identifying the checkpointing frequency based on failure distribution in the cluster [71, 64, 72], CheckFreq finds the shortest interval that masks the overhead of checkpointing based on the DNN and hardware characteristics.

Systematic online profiling. CheckFreq takes a systematic profile-based approach to determine the checkpointing frequency. It should be chosen such that the runtime overhead introduced due to checkpointing is within a percentage p of the actual compute time, where p is the permissible overhead decided by the user (say 5%).

CheckFreq determines the initial checkpointing frequency as follows. When a training job starts, CheckFreq's data iterator (§6.2.3.1) automatically profiles several iteration-level and checkpoint-specific metrics which influences the checkpointing frequency - the iteration time (T_i), time to perform weight update (T_w), time to create an in-memory GPU copy (T_g), time to create an in-memory CPU copy (T_c), time to write to storage (T_s), size of checkpoint (m), peak GPU memory utilization (M), and total GPU memory (M_{max}). Based on CheckFreq's 2-phase checkpointing mechanism, the frequency determination algorithm is as shown in Algorithm 1.

The algorithm provides two outputs; 1) the checkpointing frequency k which is the number of iterations elapsed between every checkpoint, and 2) the `snapshot()` *mode* (CPU or GPU-based). The algorithm first determines the snapshot mode based on available free GPU memory; if there is enough space to snapshot the model state in GPU memory, then the mode is set to GPU, else the preferred mode is set to CPU-based snapshotting. Based on the chosen

Algorithm 1 : Checkpointing frequency determination

```
Input: $Ti, Tw, Tc, Tg, Ts, m, M, M_{max}, p$
  $T_{oc} \leftarrow max(0, T_c - (T_i - T_w)$
  $T_{og} \leftarrow T_g$
  if $M_{max} - M > m$ and $T_{og} \leq T_{oc}$ then
    $T_o \leftarrow T_{og}$
    $mode \leftarrow GPU$
  else
    $T_o \leftarrow T_{oc}$
    $mode \leftarrow CPU$
  end if
  $k \leftarrow \frac{T_c + T_s - T_o}{T_i}$
  $k_{min} \leftarrow \left\lceil \frac{T_o}{p * T_i} \right\rceil$
  $k \leftarrow max(k, k_{min})$
Output: $k, mode$
```

mode, the algorithm estimates the overhead in the critical path incurred after pipelining checkpointing and compute in a tightly coupled manner as described earlier (§6.2.3.1). It then determines the number of iterations required to amortize this overhead such that the total runtime overhead incurred is below the threshold p. For example, consider the cost of a checkpoint operation and the duration of an iteration are both 1 time unit. If the threshold on runtime overhead is set to 5%, then CheckFreq chooses to checkpoint every 20 iterations.

Adaptive rate tuning. A static, profile-based frequency determination works well when the training environment of the model remains unchanged throughout the runtime of the job. However, in practice, the checkpoint cost estimated by the online profiler can deviate, resulting in higher than estimated runtime overheads. For instance, a job could face write interference by concurrently running jobs sharing storage for read/write, which affects the time to write a checkpoint.

Therefore, CheckFreq uses an adaptive rate tuning technique to perform feedback-driven frequency changes. CheckFreq's iterator monitors the runtime of the job and the actual cost of checkpointing during runtime (after the initial frequency determination). If the observed runtime exceeds the desired overhead, then these values are used to recalculate the checkpointing frequency. The idea is to ensure that the overall runtime overhead does not exceed the threshold p.

6.2.4 Implementation

We implement CheckFreq as a pluggable module for PyTorch. The data iterator of CheckFreq is implemented on top of the state-of-the-art data pipeline DALI for PyTorch. CheckFreq can be used as a drop-in replacement to the existing data loader in PyTorch.

CheckFreq determines the initial checkpointing frequency by profiling the first 1% of the iterations in the first epoch, or the first 50 iterations, whichever is the minimum. Therefore, no checkpointing is performed during this initial phase, which is a very small fraction of the total runtime. Additionally, we cache the profiled metrics and the determined policy on persistent storage so that profiling can be skipped when the job resumes after a crash.

CheckFreq internally uses `torch.save()`, followed by a `fsync()` to perform `persist()`, and thus guarantees persistence. To eliminate chances of data corruption, CheckFreq always writes checkpoints to a new file. However, to keep space utilization bounded, CheckFreq only maintains two checkpoints on disk at any given time; one completed checkpoint and the other in-flight. Additionally, checkpoints performed at epoch boundary are preserved (can be turned off by the user). CheckFreq wraps the weight update step in the optimizer with a semaphore that waits on the ongoing `snapshot()` to ensure that a copy of the model state is completed before it is updated by the next iteration.

6.3 Evaluation

In this section we use a number of microbenchmarks and end-to-end training to accuracy with interruptions to evaluate the efficacy of CheckFreq with respect to the current epoch-based checkpointing scheme across a variety of DNNs. Our evaluation seeks to answer the following questions.

- Can CheckFreq's iterator make iteration-level checkpointing feasible without affecting the accuracy? (§6.3.2)
- Does CheckFreq's 2-phase checkpoint mechanism reduce checkpoint stalls compared to the existing synchronous strategy? (§6.3.3)
- Can CheckFreq checkpoint more frequently than epoch-based checkpointing, while incurring low runtime overhead? (§6.3.4)
- Does CheckFreq reduce the recovery cost when DNN training is interrupted? (§6.3.5)

	GPU Type	GPU Mem(GB)	CPU Mem(GB)	Storage Media
`Conf-Pascal`	1080Ti	11	500	HDD
`Conf-Volta`	V100	32	500	SSD

Table 6.2: **Server configurations**. We use two ML server SKUs; each with 24 CPU cores, 500GB DRAM, and 8 GPUs

- What is the end-to-end benefit of training to accuracy with CheckFreq in the presence of job interruptions in a real preemptive training environment? (§6.3.6)

6.3.1 Experimental setup

We evaluate the efficacy of CheckFreq against the state-of-the-art epoch-based checkpointing in PyTorch using the state-of-the-art data pipeline DALI [21].

Servers. We evaluate CheckFreq on two generations of GPU; a Volta V100 GPU with a 1.8TB SSD for persistent storage, and a Pascal 1080Ti GPU with a 1.8TB HDD for persistent storage as shown in Table 6.2. Both these servers have 8 GPUs, 24 CPU cores and 500GB of DRAM. Both servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and PyTorch 1.1.0.

Models. We use 7 DNNs in our evaluation. ResNet18 [91], ResNet50 [91], ResNext101 [211], DenseNet121 [98], VGG16 [189], InceptionV3 [195] all on Imagenet-1k dataset [180], and Bert-Large pretraining [69] on Wikipedia & BookCorpus dataset [222]. For each model, we use the default minibatch size reported in the literature for these models.

Baseline. We use the epoch-boundary checkpointing as the baseline for all the models except BERT. BERT trains in units of iterations; therefore we use the default checkpointing interval of 200 iterations as the baseline [41]. To perform persistent and correct checkpoints, we explicitly flush the checkpoint file after the checkpoint operation returns.

6.3.2 Accuracy implications

We first show the need for resumable data iterator to make fine grained iteration-level checkpointing feasible. Using the existing state-of-the-art data iterators to perform iteration-level checkpointing results in violation of the DNN data invariant as described in (§6.2.3.1). To demonstrate this, we perform the following experiment. We train a ResNet18 job for 70 epochs or to a target accuracy of 69.5% (whichever is earliest) in three different scenarios;

- **No interrupt**. This is the normal training scenario where the job is not interrupted until its completion. There is no checkpointing performed here.

- **Baseline-interrupt**. This scenario uses the existing DALI iterator (same with the native PyTorch iterator) to perform checkpoints at the iteration right before the job is interrupted. We interrupt the job once very 7 minutes (approx every two epochs). This corresponds to commonly used round durations in preemptive schedulers [209, 153, 133, 87].

- **CheckFreq-interrupt**. This setting uses the CheckFreq data iterator

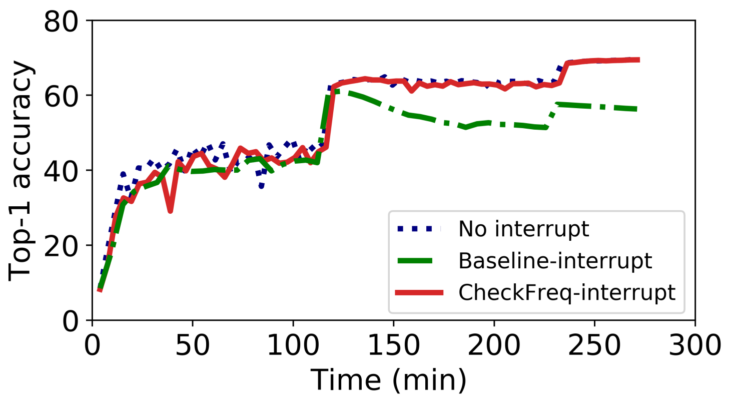


Figure 6.6: **Impact of resumable data iterator on accuracy**. Performing iteration-level checkpointing with baseline non-resumable data iterator violates the data invariant, results in significant loss of accuracy if job is interrupted. However, CheckFreq's iterator does not affect the final accuracy.

that is capable of performing a light-weight checkpoint of iterator state and correctly resuming it. We checkpoint, interrupt, and resume the job exactly as described in the prior setting.

We plot the Top-1 validation accuracy against cumulative training time. Figure 6.6 shows that it is not possible to perform iteration-level checkpointing using existing iterator, without affecting the model accuracy. This is because, the model state is checkpointed at iteration boundaries, but the data loader state is lost. However, with CheckFreq's iterator, the model reaches the target accuracy in the almost the same time as the setting where the job ran without any interruption.

Storage overhead. Checkpointing data iterator state does not have a significant space overhead; it requires persisting two integers - epoch and iteration

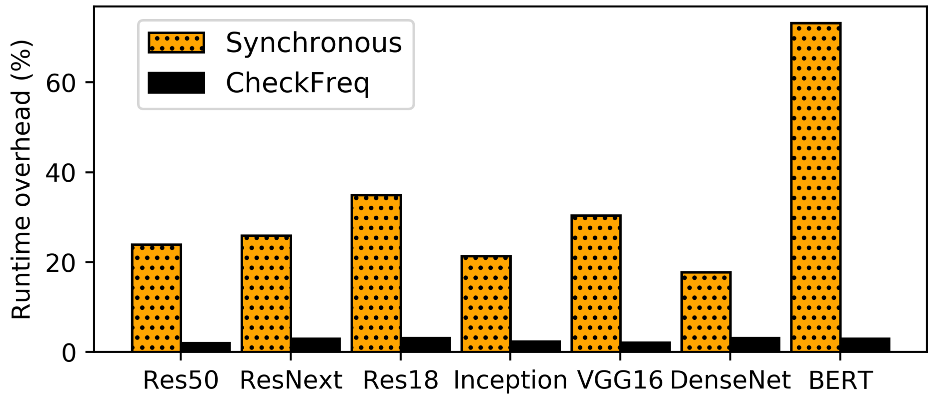


Figure 6.7: **Runtime overhead for various models**. At a frequency chosen by CheckFreq, synchronous checkpointing incurs upto 70% overhead while CheckFreq's pipelined checkpointing reduces runtime overhead to under 3.5%

number, that take up a few bytes on disk. CheckFreq thus provides lightweight, resumable data iterators that do not affect the accuracy of DNNs.

6.3.3 Performance of checkpointing mechanism

We now evaluate the performance of the two-phase checkpointing strategy of CheckFreq, and compare it against the synchronous strategy. We further provide a split of benefits due to pipelining `persist()` and `snapshot()` operations.

6.3.3.1 Checkpoint stalls

Figure 6.7 shows the runtime overhead incurred due to checkpoint stalls with CheckFreq and the baseline checkpointing mechanism while checkpointing at a frequency chosen for that model by CheckFreq on `Config-HDD-1080Ti`.

	Checkpoint stall (seconds)		
	Synchronous	***IO pipelining***	***CheckFreq***
`Config-SSD-V100`	3.6	1.5	0.3
`Config-HDD-1080Ti`	10.7	1.3	0.07

Table 6.3: **Breakdown of benefits**. This table shows the split of checkpoint stall incurred in critical path for VGG16 on two different hardwares

The frequency varies across models, but is kept constant for CheckFreq and baseline for a given model. While CheckFreq is able to bound the runtime overheads to about 3.5%, the baseline incurs 17 – 73% runtime overhead due to frequent checkpointing. The reduction in runtime overhead is due to the two-phase checkpointing and pipelining it with computation.

6.3.3.2 Breakdown of benefits

To understand how much each phase of the checkpointing mechanism contributes to the reduction of checkpoint stalls, we train VGG16 on the two servers using identical batch size of 64 that is the maximum that can fit on `Config-HDD-1080Ti`. Checkpointing is performed at a frequency chosen independently for the two servers. We evaluate three settings in Table 6.3; 1) The baseline synchronous mode, 2) CheckFreq with only `persist()` pipelining (indicated by IO pipelining) and `snapshot()` performed synchronously, 3) CheckFreq with both `persist()` and `snapshot()` pipelining.

On both hardware, CheckFreq is able to significantly reduce the checkpoint cost by 5 – 18× by pipelining both phases of checkpointing with compute

Model	Res18	Res50	ResNext	VGG16	BERT
Freq	147	125	238	83	100
Size(MB)	90	195	482	1055	5000

Table 6.4: **Checkpoint frequency**. This table shows the number of checkpoints per epoch and the size of each checkpoint

as compared to only pipelining `persist()`. On `Config-HDD-1080Ti`, the benefit due to pipelining `persist()` is prominent due to the slower storage device. On `Config-SSD-V100` with fast storage, the CPU cost of `snapshot()` and the storage cost of `persist()` contribute equally to the checkpointing cost. Therefore, pipelining `snapshot()` with compute provides significant speedup.

6.3.4 Checkpointing policy

We compare the checkpointing frequency determined by CheckFreq for a threshold overhead p of 3.5%. Table 6.4 shows the number of checkpoints performed per epoch for various models along with per-checkpoint size when performing distributed data parallel training across across 8 GPUs on `Config-HDD-1080Ti`. There are two main takeaways here. First, the checkpointing frequency varies with model; therefore frequency selection must take into account the model characteristics. Second, CheckFreq is able to perform $83 - 278\times$ more frequent checkpointing when compared to that performed at epoch boundaries, while incurring $\leq 3.5\%$ overhead. On `Config-SSD-V100`, CheckFreq resulted in $25 - 100\times$ more frequent checkpointing than the epoch-based policy. More frequent checkpoints directly translate to faster recovery

Setting	Isolated	Static	Adaptive
Overhead	5%	**35%**	5%
Frequency (# iterations)	14	14	19

Table 6.5: **Adaptive frequency tuning**. Adaptive frequency tuning is able to dynamically adjust checkpointing frequency to maintain the same overhead as if the job is run in isolation.

Model	***Recovery (seconds)***	
	Baseline	***CF***
ResNet18	840	5
ResNet50	2100	24
VGG16	5700	25
ResNext101	7080	32
DenseNet121	2340	7
Inceptionv3	3000	27
BERT	4920	85

(a) 1 GPU (V100)

Recovery (seconds)	
Baseline	***CF***
180	3
540	8
1320	31
1680	14
600	4
780	42
4500	43

(b) 8 GPU (1080Ti)

Table 6.6: **Average recovery time (CF - CheckFreq)**.

times which we evaluate in Section 6.3.5.

Adaptive tuning of frequency. To demonstrate the importance of adaptive frequency tuning, we perform the following experiment. We run a VGG16 training job on a single GPU (Job-A), allowing it to checkpoint at an initial frequency chosen by CheckFreq (with an overhead of 5%). After 100 iterations have elapsed, we trigger another VGG16 job on a different GPU on the same machine (Job-B), so that the two jobs contend for storage bandwidth to write checkpoints. We measure the runtime for 500 iterations of Job-A with and without adaptive frequency tuning. The results are as shown in Table 6.5. When Job-A runs in isolation, it incurs an overhead of 5% while checkpointing every 14 iterations. However, when Job-B is introduced after 100 iterations of Job-A, if there is no adaptation across the two jobs, the checkpointing frequency is statically fixed to 14 iterations and the runtime overhead for Job-A increases to 35% (indicated as static in Table 6.5). This is because, the jobs compete for storage bandwidth, increasing checkpoint cost. In contrast, CheckFreq's adaptive rate tuning dynamically adjusts the checkpointing frequency and keeps the overhead bounded at 5%.

6.3.5 Recovery time

To understand the benefits of using CheckFreq in the presence of job interruptions, we evaluate the recovery time with the epoch-based checkpointing and CheckFreq. With epoch-based checkpointing, irrespective of when during the epoch the job is interrupted, the job rolls back to the previously completed epoch. Therefore, in the best case, if a failure occurs immediately after the finish of an epoch, then the recovery time is the same as CheckFreq. However, on average, half an epoch's worth of work can be lost if the job is interrupted in the middle of an epoch. And in the worst case, the entire epoch must be redone if the job fails just before the completion of an epoch. For the seven different models, we compare the average case recovery time in two distinct scenarios; 1) a single-GPU training job on `Config-SSD-V100` in Table 6.6a and 2) a 8 GPU data-parallel job on `Config-HDD-1080Ti` in Table 6.6b.

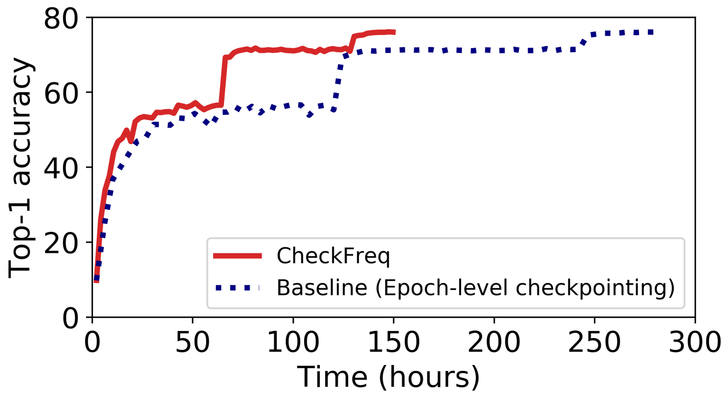


Figure 6.8: **End-to-end training**. We train Resnet50 using a `Config-HDD-1080Ti` GPU with interruptions every 5 hours. CheckFreq trains to state-of-the-art accuracy (76.1%) 2× faster than epoch-based checkpointing by reducing recovery time.

As can be seen, CheckFreq is able to reduce recovery time from several minutes (and hours) to just a few seconds, while incurring less than 3.5% run-time overhead. For *e.g.*,, when training ResNext101 on a V100 GPU, CheckFreq reduces the recovery time from 2 hours to 32 seconds on average.

6.3.6 End-to-end training

We evaluate the end-to-end benefit of training with CheckFreq by simulating a preemptive cluster scenario. We consider a cluster with a premeptive scheduler similar to the one in large production clusters like Philly [106, 18]. We consider an average preemption interval of 5 hours. Figure 6.8 plots the total training duration against top-1 validation accuracy for the epoch-based baseline checkpointing strategy and CheckFreq for training ResNet50 using a GPU on `Config-HDD-1080Ti` to state-of-the-art accuracy. CheckFreq results in 2× faster training by reducing recovery time from 1.9 hours to under a minute for every interruption. A similar experiment on `Config-SSD-V100` resulted in 1.6× faster training time to accuracy for ResNext101.

6.4 Limitations and Discussion

We now discuss the limitations and future directions for CheckFreq.

Applicability of checkpointing to distributed cluster training. CheckFreq currently works with the distributed data parallel (DDP) mode, where only one GPU per node (rank 0) is responsible for checkpointing. While we show results for single- and multi-GPU training, extending it to multi-node settings is straightforward; checkpointing in multi-GPU and multi-node settings is the same for DDP in frameworks such as PyTorch. Model weights are synchronized across different workers (same node or in the distributed cluster) typically every iteration, or accumulated over a few tens of iterations before synchronizing; therefore each node sees the same version of weights at these synchronization points. Hence, one instance of CheckFreq runs on each node, and persists an identical checkpoint for local recovery at synchronization boundaries. Since each node persists checkpoints independently, and in parallel, there is no additional synchronization overhead for checkpointing.

Generality of CheckFreq. CheckFreq focuses on optimizing checkpointing, which is by far the predominant way in which DNN training jobs recover from failures. While our dissertation focuses on data parallel training, prior work in model or pipeline parallelism, also rely on checkpointing. Using CheckFreq, checkpointing at minibatch boundaries (every n iterations), each pipeline stage only persists a subset of parameters and optimizer state hosted by that worker. CheckFreq also enables checkpointing within minibatch boundaries during pipeline parallel training (every m microbatches), as CheckFreq's iterator controls the introduction of each microbatch into the pipeline. Checkpointing at the microbatch granularity requires storing additional model state – specifically accumulated weight gradients at every stage in addition to parameter and optimizer state. We leave it to future work to integrate CheckFreq's implementation into frameworks supporting pipeline parallelism.

While we implement CheckFreq in PyTorch, we can extend it to other frameworks like TF and MxNet by wrapping the framework-specific APIs into those exposed by CheckFreq.

6.5 Summary

This chapter presents CheckFreq, an automatic, fine-grained checkpointing framework for DNN training. CheckFreq achieves consistent, low-cost checkpoints at iteration level using a resumable data iterator, a pipelined two-phase checkpointing mechanism, and automatic determination and tuning of checkpointing frequency. When the job is interrupted, CheckFreq reduces recovery time for popular DNNs from hours to seconds, while incurring low runtime overhead. With the growing scale and complexity of DNN training, CheckFreq enables low-cost, failure resilient training.

Chapter 7

Related Work

In this Chapter, we discuss prior research efforts and systems that are related to this dissertation. First, we describe past efforts in optimizing DNN training time by optimizing the data pipeline (§7.1). We next discuss prior works on DNN cluster scheduling and big data scheduling that our dissertation draws inspiration from (§7.2). Finally, we discuss prior work related to optimizing DNN checkpointing (§7.3).

7.1 Optimizing DNN training time

A number of solutions have been proposed to reduce the training time for DNNs including specialized hardware [110, 106, 183, 164, 159, 173, 8, 16], parallel training [117, 65, 56, 109, 116, 151, 99], GPU memory optimizations [178, 54, 103], lowering communication overhead [128, 217, 90, 104], faster communication libraries [19, 205], and compiler-based operator optimizations [202, 53, 108]. This paper presents a new point in this spectrum, *data stalls*.

Importance of data ingest pipelines. There is a rich literature in the database community both in understanding data stalls and cache behaviour in DBMS [42, 218, 190], as well as building efficient data ingest pipelines for query processing [45, 201, 131]. Our work seeks motivation from such findings and analyses data stalls in the context of emerging DNN workloads.

Application-aware caching. The idea of designing a caching policy that is aware of application semantics is not new. Stonebraker highlighted the importance of domain-aware caching for databases [192]. Tomkins et.al. show that informed prefetching and caching in file systems can reduce the execution time of I/O-intensive applications [199]. Similarly, Liao et.al. show that application-aware client side caching in a parallel filesystem outperforms traditional caching approaches [124]. Our work draws parallels to such techniques by first identifying data stall times in DNN training, understanding their access pattern and then devising a caching policy based on these observations.

Hardware solutions to fetch stalls. New hardware like NVIDIA's Magnum IO [20], and PureStorage's AIRI [28] provide high throughput storage solutions to address fetch stalls. While these fast hardware may mask fetch stalls in some models, they may not help if the model is bottlenecked on prep stalls. CoorDL accelerates DNN training by mitigating data stalls with existing commodity servers as opposed to relying on expensive hardware solutions. Furthermore, as compute gets faster, the bottleneck may shift back to storage. Such high-end hardware solutions are also unavailable in public cloud.

Redundancy in DNN training. Prior work like Model Batching [154] has identified redundancy in model search; where an algorithm automatically searches for a model architecture for a given task. However, it optimizes for running multiple DNNs together on a single GPU, by sharing GPU computation across jobs. CoorDL on the other hand accelerates training in the more common setting where GPUs are not shared between jobs.

OneAccess [111] is a preliminary study that uses reservoir sampling to generate uniformly random samples of data while accessing pre-processed data sequentially. In a departure from the state-of-the-art, OneAccess stores pre-processed data across epochs to reduce prep stalls; however such an approach precludes commonly used online data-augmentation (rescaling, translations, flipping) and randomization (hue, saturation, brightness, and contrast) techniques. This can affect model convergence adversely. Furthermore, OneAccess uses a simplistic PyTorch baseline with no more than 2 CPU cores used per GPU and very small datasets (MS-COCO [127] and CIFAR-10 [114]). In contrast, CoorDL carefully eliminates redundancy while preserving accuracy and providing significant speedups.

A team of researchers at Google Brain recently proposed data echoing [57], a technique that reuses data from prior batches if the accelerator is stalled for the current batch of data.This technique relaxes the DNN ETL requirements - both data ordering and randomness. Therefore, it needs a careful selection of echoing rate and an additional hyperparameter search even for existing model architectures to achieve state-of-the-art accuracy. Unlike data echoeing, the techniques introduced in this dissertation do not alter the ETL requirements or affect the final accuracy of the model in any way.

Distributed DNN caching. Prior work like Quiver [118], DeepIO [221], and Cerebro [150] build systems for large-scale distributed training. These works have identified that data fetch from remote storage can bottleneck large-scale distributed DNN training. We now discuss how each one of these systems fall short of harnessing the insights from our analysis, and the need for a coordinated caching and pre-processing library for DNN training.

Quiver [118] is a distributed storage (SSD) cache that uses a new substitutable sampling technique co-designed with the PyTorch framework, which restricts randomness in the creation of minibatches to a subset of cached items. Unlike CoorDL that accelerates a variety of training settings, Quiver is specifically designed for HP search when the dataset is too large to fit on the local storage device ($3TB$).

DeepIO [221] also proposes an entropy-aware sampling technique, and RDMA based data shuffling for distributed training across servers. However, when the entire dataset does not fit in memory, DeepIO cache suffers from thrashing unlike MinIO. Unlike DeepIO, CoorDL does not require any specialized hardware support and provides impressive speedups over commodity TCP stack with low bandwidth requirements and no changes to the training framework.

Cerebro [150] introduces a new parallel SGD strategy for model selection tasks. It partitions the dataset across the servers in a cluster and hops the models from one server to other, instead of shuffling data. Cerebro does not improve performance for DNN training on a single server, while CoorDL presented in our dissertation optimizes performance in this scenario. Furthermore, Cerebro is designed for a specifc scenario - distributed model search; on the contrary, our analysis and CoorDL have a broader scope.

Tensorflow dataloader - tf.data. Recent work presented `tf.data`, a framework for building and executing efficient input pipelines for machine learning jobs in TensorFlow [149]. `tf.data` is based on the programming model of chaining higher-order functional transformations, executing input pipelines as dataflow graphs. It also automatically tunes parameters such as buffer sizes and the amount of parallelism to optimize the data pipeline. While `tf.data` is a comparable alternative to the PyTorch dataloader, it fails to harness the insights from our data stall analysis such as DNN optimized caching, and eliminating redundancy.

7.2 DNN cluster schedulers

A number of recent works target on cluster scheduling for DNN workloads each one focusing on improving a certain objective; Cluster utilization (Gandiva [209]), JCT (Tiresias [87]), and fairness (Themis [133], Gandiva-Fair [50]). Some works have also looked at exploting performance heterogeniety among accelerators to improve cluster objectives [153, 121]. All these schedulers assume GPU to be the dominant resource in the scheduling task; i.e., a user requests a fixed number of GPUs for her DNN job, and when the requested number of GPUs are all available, the job is scheduled to run. Other resources such as CPU and memory are allocated proportional to the number of GPUs requested by the job. Existing scheduler thus ignore *resource-sensitivity* of the DNN tasks to CPU, and memory. Synergy shows that resource-sensitivity allocation is an important factor to be considered for DNN workload scheduling on multi-tenant clusters, and can help achieve better cluster utilization.

Big data schedulers. Our work builds upon the insights drawn from the rich literature of schedulers for big data jobs [203, 94, 85, 84, 83, 77]. Big data schedulers like Tetris [83], and DRF [77] have looked at the problem of multi dimensional resource allocation for big data jobs. They propose new scheduling policies aimed at optimizing a specific cluster objective for jobs whose resource demands are prior known. Big data jobs come in with resource requirements along several dimesnions, all of which is necessary to run that task. On the contrary, the primary resource in a DNN job is the accelerator (GPU), whose requirement is specified by the job. Other resources are fungible; Our work exploits this key insight to perform disproportionate allocations by profiling job resource sensitivity, and then appropriately packing them onto available servers.

Algebriac scheduling. Prior work like alsched [200] explore how to perform optimal scheduling decisions by formulating resource requirements as hard and soft constraints using algebraic utility expressions, and solve an optimization problem. Synergy's optimal upper bound estimation shares a similar optimization approach. However, unlike alsched, Synergy does not require manual encoding of utility functions; resource requirements are automatically profiled and estimated. Moreover, Synergy is optimized for DNN training; it incorporates DNN-aware constraints such as reducing GPU fragmentation and minimizing the number of fragmented jobs.

7.3 Optimizing DNN checkpointing

While recent work like DeepFreeze [157] that perform asynchronous DNN checkpointing employ techniques similar to CheckFreq for IO pipelining, it only considers CPU clusters. It does not consider the cost of snapshotting the model state in memory when trained using state-of-the-art GPUs. Our work shows that on modern ML optimized servers, the cost of snapshotting the model state (copying from GPU to CPU) is significant, demonstrating how to pipeline this transfer with compute, and use spare GPU capabilities to enable fast snapshotting.

Furthermore, DeepFreeze requires manual intervention to tune the checkpointing frequency for a given model, hardware and training environment while CheckFreq masks these complexities from the user and analytically identifies the best parameters for checkpointing. Unlike DeepFreeze that uses a static checkpointing frequency, CheckFreq is also beneficial in shared cluster settings, as it adapts the checkpointing frequency based on memory and storage interference due to other jobs to minimize checkpoint stalls.

Recent work by Chen [55] look at reducing checkpoint IO using quantization techniques, while CheckFreq optimizes the checkpointing process without altering the checkpoint data, therefore resulting in no accuracy implications.

Asynchronous checkpointing in HPC. Prior work in HPC [187, 158, 198] uses asynchronous checkpointing to mask the IO latency. A key challenge that differentiates DNN checkpointing from traditional HPC ones is that, performing a synchronous in-memory copy of the model state from GPU to CPU is expensive due to the increasingly fast compute capabilities of the GPU. CheckFreq exploits the DNN learning structure to carefully pipeline even the in-memory snapshot with computation to perform correct, consistent checkpointing. Moreover, CheckFreq further reduces the latency of checkpointing by utilizing spare GPU memory and compute capabilities when possible to perform fast snapshots. The novelty of CheckFreq thus lies in identifying and exploiting the unique characteristics of DNN training to correctly, and consistently perform fast, online, CPU- and GPU-based snapshots and providing a practical, easy to deploy solution to fine-grained checkpointing.

Checkpoint interval estimation in HPC. Prior work [71, 64, 72] determine checkpointing interval for large scale HPC applications based on failure distributions observed in the system. CheckFreq does this in a DNN-aware fashion by exploiting the deterministic, repetitive structure of DNN training to systematically profile resource utilization at runtime.

Adaptive checkpointing. The idea of using adaptation for fault management has been used in HPC applications [120] to decide when to checkpoint, based on a failure prediction module. CheckFreq introduces adaptivity in DNN checkpointing frequency. It identifies and dynamically adapts the checkpointing frequency, based on the characteristics of the model being trained, system hardware, and interference due to other jobs.

Frequent checkpointing mechanisms. The idea of frequently checkpointing application state is well researched in the HPC community. Prior work by Tiwari explored how to reduce the I/O overhead during checkpointing using *lazy checkpointing* [198]. Similarly Di [72] and Daly [64] show how to determine checkpointing interval for large scale HPC applications based on failure distributions observed in the system. In contrast to such work, CheckFreq builds DNN specific checkpointing policy and mechanism that exploits DNN computational model to pipeline checkpoint with compute, at a frequency determined by the model characteristics.

TensorFlow Checkpoint Manager. TF checkpoint manager [196] allows checkpointing at a user-given time interval, and supports persisting iterator state. However, it has three shortcomings. First, the checkpointing frequency is decided in an ad-hoc fashion by the user; this introduces large checkpoint stalls if not chosen carefully. Second, it cannot checkpoint the iterator state if random data transformation is involved; this is common for most image based models [197]. Finally, even in cases where it can persist iterator state, TF writes the entire operator graph to storage along with prefetched items resulting in large checkpoint size. CheckFreq addresses these challenges by automatically adapting the checkpointing frequency and using a light-weight, resumable data iterator.

Framework-transparent checkpointing. Transparent checkpointing techniques such as CRIU [2] can backup entire VM state for fault-tolerance; however they do not checkpoint GPU or accelerator state. Even if they were to capture entire device state, device state alone is an order of magnitude larger than the model state captured at iteration boundaries, making frequent CRIU checkpoints impractical. Thus, in this work, we focus on the dominant approach to DNN fault-tolerance - framework-assisted checkpointing of model state.

Chapter 8

Conclusion and Future Work

The problem of data stalls in DNN training will only worsen with time; as the size of data sets increase [36, 40, 119], and GPUs become faster [112]. Our dissertation throws light on ways in which the data pipeline bottlenecks DNN training, obscuring the efforts of prior research on scaling popular DNN models [215].

In this dissertation, we present the first detailed study of data stalls in DNN training, and argue that it is possible to accelerate end-to-end DNN training time by carefully mitigating data stalls. We show that data stalls account for up to 70% of the training time in data-intensive DNNs. The insights from our study guide the design of several systems to mitigate data stalls; (1) CoorDL, a coordinated caching and pre-processing library for DNN training. (2) Synergy, a resource-aware scheduler for training DNN jobs on shared GPU clusters, and (3) CheckFreq, an automated fine-grained checkpointing framework to reduce GPU recovery time, while minimizing checkpoint stalls.

8.1 Looking Forward

With the fast evolution of new ML models, growth in the need for large-scale distributed training infrastructures, changing hardware configurations, and emerging storage technologies, it is an exciting time for storage for ML research.

While the systems we built addressed some issues unearthed by our data stall analysis, we now discuss interesting future research directions that this dissertation opens up.

Decoded cache to reduce pre-processing overhead. Our dissertation shows that decoding/decompressing raw images is one of the most expensive operations during data pre-processing. A future direction is to evaluate the benefits of caching decoded data items instead of the current approach of caching raw encoded items. Since decoding is deterministic, it is possible to cache it across epochs. However, this is non trivial; decoding increases the dataset size by $5-7\times$. A solid future direction is to explore how to make decoded caching practical without incurring the high space overhead, possibly using serialized data formats.

Disaggregated data loading. Our dissertation shows that the imbalance in CPU cores per GPU in ML optimized servers result in data stalls for several models. In such cases where single-host capacities are maxed out, a viable approach is to offload data prep to other idle host machines in a cluster. This is especially useful in production clusters with high-bandwidth Ethernet, where several jobs use the same dataset and similar pre-processing pipelines; a dedicated set of servers can be used to centrally pre-process minibatches of data, while the training jobs can request minibatch as a service, thereby entirely disaggregating learning from data management.

Automatic prep offload to accelerators. Data pipelining frameworks like DALI have the ability to perform certain image and audio based pre-processing (prep) such as crop, flip, and other transformations on GPU accelerators. However, the split of operations performed on GPU and CPU has to be decided by the user manually.

There is a memory-performance tradeoff in deciding how many prep operations are offloaded to the GPU for two reasons. (1) Performing prep at the GPU takes up a part of the already scarce GPU memory which may result in training with lower batch sizes, thereby affecting training efficiency. (2) Prep at the GPU may interfere with the computations performed by the learning algorithm; this adversely affects the overall throughput of training for computationally expensive and deeper networks. Therefore, the split of prep operations must be carefully chosen considering the model's architecture, batch size, and data stalls. While this split is determined manually by trial-and-error today, automating it with a careful eye on GPU and CPU utilization is a promising direction towards optimizing the input data pipeline.

More generally, it may be beneficial to offload data pre-processing entirely to specialized hardware accelerators like FPGAs or ASICs so that the entire GPU capacity can be used by the learning process.

Optimizing the inference data pipeline. This dissertation addresses data stalls in the training pipeline which have three distinct features from inference. (1) Training requires a large volume of data samples, (2) performs a larger set of data prep for every batch, and (3) requires backpropagation during the learning phase. While inference jobs require fewer prep steps per sample or batch, it also performs lesser GPU computation compared to training. Moreover, the limited memory and compute availability at edge devices also introduces data stalls in inference. We hope this dissertation encourages similar research and possibly unique optimizations in inference land.

8.2 Concluding Remarks

We are now in a world that is most conducive for deep learning research; accelerator speeds are growing at an unprecedented rate, and training datasets are exploding in size. This growth trend in computational power and dataset sizes is definitely a boon for DNN training. However, the teraFLOPS of compute power that the GPUs of today house, cannot be fully harnessed unless we build equally efficient infrastructure to feed data into these powerful accelerators. The history of high performance computing has taught us an important lesson; *A system can process only as fast as its slowest component.* This dissertation serves as a reminder to carefully analyze, re-evaluate, and re-invent design decisions as a system evolves.

Bibliography

[1] Amazon sagemaker tutorial. https://towardsdatascience.com/a-quick-guide-to-using-spot-instances-with-amazon-sagemaker-b9cfb3a44a68.

[2] CRIU checkpointing. https://criu.org/Main_Page.

[3] gRPC. https://grpc.io/.

[4] Why the AI Industry Needs to Rethink Storage. https://blog.purestorage.com/ai-industry-needs-rethink-storage/, July 2017.

[5] Aws instance types. https://aws.amazon.com/ec2/instance-types/#p2, 2020.

[6] Aws instance types. https://aws.amazon.com/ec2/instance-types/#p3, 2020.

[7] Blobfuse. https://github.com/Azure/azure-storage-fuse, 2020.

[8] Cerebras Wafer Scale Engine. https://www.cerebras.net/, 2020.

[9] Cloud TPU tools. https://cloud.google.com/tpu/docs/cloud-tpu-tools, 2020.

[10] Dali accelerates pre-processing. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9925-fast-ai-data-pre-processing-with-nvidia-dali.pdf, 2020.

[11] Dali: Supported operations. https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html#nvidia.dali.ops.FileReader, 2020.

[12] Distributed data parallel. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html, 2020.

[13] Ebs volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html, 2020.

[14] Fast AI Data Preprocessing with NVIDIA DALI. https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali/, January 2020.

[15] Gloo. https://github.com/facebookincubator/gloo, 2020.

[16] GraphCore Intelligence Processing Unit. https://www.graphcore.ai/, 2020.

[17] Imagenet-22k. http://www.image-net.org/releases, 2020.

[18] Microsoft philly traces. https://github.com/msr-fiddle/philly-traces, 2020.

[19] Nccl. https://developer.nvidia.com/nccl, 2020.

[20] Nvidia : Magnum-io. https://www.nvidia.com/en-us/data-center/magnum-io/, 2020.

[21] Nvidia dali. https://github.com/NVIDIA/DALI, 2020.

[22] NVIDIA DGX-2: Enterprise AI Research System. https://www.nvidia.com/en-us/data-center/dgx-2/, 2020.

[23] NVIDIA nvJPEG library. https://docs.nvidia.com/cuda/nvjpeg/index.html, 2020.

[24] Nvidia object detection. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Detection/SSD, 2020.

[25] Nvidia profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2020.

[26] Nvlink. https://www.nvidia.com/en-us/data-center/nvlink/, 2020.

[27] Profiling MXNet models. https://mxnet.apache.org/api/python/docs/tutorials/performance/backend/profiler.html, 2020.

[28] Purestorage : Airi. https://www.purestorage.com/products/flashblade/ai-infrastructure.html, 2020.

[29] Pytorch. https://github.com/pytorch/pytorch, 2020.

[30] Pytorch ddp. https://pytorch.org/docs/stable/_modules/torch/nn/parallel/distributed.html, 2020.

[31] Pytorch: DDP vs DP. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html, 2020.

[32] Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, Santa Clara, CA, February 2020. USENIX Association.

[33] Torchaudio classifier. https://pytorch.org/tutorials/beginner/audio_classifier_tu audio, 2020.

[34] Torchvision models. https://pytorch.org/docs/stable/torchvision/models.html, 2020.

[35] Trace event format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview, 2020.

[36] Training a Champion: Building Deep Neural Nets for Big Data Analytics. https://www.kdnuggets.com/training-a-champion-building-deep-neural-nets-for-big-data-analytics.html/, 2020.

[37] Wmt16. http://www.statmt.org/wmt16/, 2020.

[38] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, GA, 2016.

[39] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[40] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *CoRR*, abs/1609.08675, 2016.

[41] NVIDIA AI. Bert meets gpus. hhttps://medium.com/future-vision/bert-meets-gpus-403d3fbed848, 2020.

[42] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. Dbmss on a modern processor: Where does time go? In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 266–277. Morgan Kaufmann, 1999.

[43] Amazon. Amazon EC2 spot instances. https://aws.amazon.com/ec2/spot/?cards.sort-by=item.additionalFields.startDateTime&cards.sort-order=asc.

[44] Amazon. Amazon sagemaker. https://aws.amazon.com/sagemaker/.

[45] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 208–219. IEEE Computer Society, 1996.

[46] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(8):1798–1828, 2013.

[47] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[48] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020.

[49] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[50] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[51] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[52] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[53] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[54] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*,

2016.

[55] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On efficient constructions of checkpoints. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 1627–1636. PMLR, 2020.

[56] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, volume 14, pages 571–582, 2014.

[57] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. Faster neural network training with data echoing. *CoRR*, abs/1907.05550, 2019.

[58] Dami Choi, Alexandre Passos, Christopher J Shallue, and George E Dahl. Faster neural network training with data echoing. *arXiv preprint arXiv:1907.05550*, 2019.

[59] Jichan Chung, Kangwook Lee, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Ubershuffle: Communication-efficient data shuffling for sgd via coding theory. *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[60] Alex Clark. Pillow (pil fork) documentation, 2015.

[61] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. In *2nd USENIX Symposium on Internet Technologies and Systems, USITS'99, Boulder, Colorado, USA, October 11-14, 1999*. USENIX, 1999.

[62] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. Very deep convolutional neural networks for raw waveforms. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 421–425. IEEE, 2017.

[63] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics, 2019.

[64] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, 2006.

[65] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, and

Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[66] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. Fma: A dataset for music analysis. *arXiv preprint arXiv:1612.01840*, 2016.

[67] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[68] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[69] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *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.

[70] NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/, 2020.

[71] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[72] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[73] Steven Diamond and Stephen P. Boyd. CVXPY: A python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17:83:1–83:5, 2016.

[74] György Dósa and Jirí Sgall. First fit bin packing: A tight analysis. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 538–549. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[75] Nick Evanson. 25 years later: A brief analysis of gpu processing efficiency. https://www.techspot.com/article/2008-gpu-efficiency-historical-analysis/, 2021.

[76] Thomas R. Furlani, editor. *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), PEARC 2019, Chicago, IL, USA, July 28 - August 01, 2019*. ACM, 2019.

[77] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[78] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.

[79] Google. Preemptible VM instances. https://cloud.google.com/compute/docs/instances/preemptible#preemptible_with_gpu.

[80] Google. Open images dataset. https://opensource.google/projects/open-images-dataset, 2020.

[81] Mel Gorman. Understanding the linux virtual memory manager. https://www.kernel.org/doc/gorman/html/understand/understand013.html, 2020.

[82] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[83] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 455–466. ACM, 2014.

[84] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 65–80. USENIX Association, 2016.

[85] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[86] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[87] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 485–500. USENIX Association, 2019.

[88] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, pages 44:1–44:12. ACM, 2017.

[89] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[90] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H. Campbell. Tictac: Accelerating distributed deep learning with communication schedul-

ing. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[91] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[92] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[93] Dodi Heryadi and Scott Hampton. Characterizing performance improvement of gpus. In Thomas R. Furlani, editor, *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), PEARC 2019, Chicago, IL, USA, July 28 - August 01, 2019*, pages 4:1–4:5. ACM, 2019.

[94] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In David G. Andersen and Sylvia Ratnasamy, editors, *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[95] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural

networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

[96] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019.

[97] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[98] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 2261–2269. IEEE Computer Society, 2017.

[99] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[100] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[101] Kumar Iyer and Jeffrey Kiel. GPU debugging and profiling with NVIDIA Parallel Nsight. In *Game Development Tools*, pages 303–324. AK Peters/CRC Press, 2016.

[102] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

[103] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA '18)*, 2018.

[104] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[105] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie

Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[106] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[107] Myeongjae Jeon, Shivaram Venkataraman, Junjie Qian, Amar Phanishayee, Wencong Xiao, and Fan Yang. Multi-tenant gpu clusters for deep learning workloads: Analysis and implications. *Tech. Rep.*, 2018.

[108] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automated generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* ACM, 2019.

[109] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd SysML Conference, SysML '19*, Palo Alto, CA, USA, 2019.

[110] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb,

Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA 2017, pages 1–12, New York, NY, USA, 2017. ACM.

[111] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[112] Rupp Karl. CPU, GPU and MIC hardware characteristics over time. https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, 2020.

[113] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[114] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[115] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[116] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[117] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[118] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, pages 283–296. USENIX Association, 2020.

[119] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. The open images dataset v4: Unified image classifica-

tion, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982*, 2018.

[120] Zhiling Lan and Yawei Li. Adaptive fault management of parallel applications for high-performance computing. *IEEE Trans. Computers*, 57(12):1647–1660, 2008.

[121] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 31:1–31:16. ACM, 2020.

[122] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. Deep learning. *Nat.*, 521(7553):436–444, 2015.

[123] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[124] Wei-keng Liao, Kenin Coloma, Alok Choudhary, Lee Ward, Eric Russell, and Sonja Tideman. Collective caching: Application-aware client-side file caching. In *HPDC-14. Proceedings. 14th IEEE International Symposium on High Performance Distributed Computing, 2005.*, pages 81–90. IEEE, 2005.

[125] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[126] Tao Lin, Sebastian U. Stich, Kumar Kshitij Patel, and Martin Jaggi. Don't use large mini-batches, use local SGD. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[127] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V*, volume 8693 of *Lecture Notes in Computer Science*, pages 740–755. Springer, 2014.

[128] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

[129] Linux. mlock(). https://man7.org/linux/man-pages/man2/mlock.2.html, 2021.

[130] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[131] Chen Luo and Michael J. Carey. Efficient data ingestion and query processing for lsm-based storage systems. *Proc. VLDB Endow.*, 12(5):531–543, 2019.

[132] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018.

[133] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[134] Catello Di Martino, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 610–621. IEEE Computer Society, 2014.

[135] Qi Meng, Wei Chen, Yue Wang, Zhi-Ming Ma, and Tie-Yan Liu. Convergence analysis of distributed stochastic gradient descent with shuffling. *Neurocomputing*, 337:46–57, 2019.

[136] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[137] Apache Mesos. Recordio data format. https://mesos.apache.org/documentation/latest/recordio/, 2020.

[138] Microsoft. Use low priority VMs. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms.

[139] Microsoft. Avere systems : Performance in a flash. https://www.microsoft.com/en-us/avere?activetab=pivot1%3aprimaryr2, 2020.

[140] MLPerf. Mlperf training results v0.7. https://github.com/mlperf/training_results_v0.7, 2020.

[141] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 33–50. USENIX Association, 2018.

[142] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Crashmonkey and ACE: systematically test-

ing file-system crash consistency. *ACM Trans. Storage*, 15(2):14:1–14:34, 2019.

[143] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, 2021.

[144] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Synergy: Resource sensitive DNN scheduling in multi-tenant clusters. *CoRR*, abs/2110.06073, 2021.

[145] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *CoRR*, abs/2007.06775, 2020.

[146] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.*, 14(5):771–784, 2021.

[147] Timothy Prickett Morgan. Removing The Storage Bottleneck For AI. https://www.nextplatform.com/2018/03/29/removing-the-storage-bottleneck-for-ai/, March 2018.

[148] Hassan Mujtaba. Nvidia's ampere a100 gpu is unstoppable. https://wccftech.com/nvidia-ampere-a100-fastest-ai-gpu-up-to-4-times-

faster-than-volta-v100/, 2021.

[149] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *CoRR*, abs/2101.12127, 2021.

[150] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *Proc. VLDB Endow.*, 13(11):2159–2173, 2020.

[151] 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. ACM, 2019.

[152] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *DISPA*, 2020.

[153] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. *arXiv preprint arXiv:2008.09213*, 2020.

[154] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating deep learning workloads through efficient multi-

model execution. In *NIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[155] Vincent Natoli. A decade of accelerated computing augurs well for gpus. https://www.nextplatform.com/2019/07/10/a-decade-of-accelerated-computing-augurs-well-for-gpus/, 2021.

[156] Andrew NG. Data and dnns. https://www.wired.com/brandlab/2015/05/andrew-ng-deep-learning-mandate-humans-not-just-machines/, 2020.

[157] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*, pages 172–181. IEEE, 2020.

[158] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. Veloc: Towards high performance adaptive asynchronous checkpointing at large scale. In *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 911–920. IEEE, 2019.

[159] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Dun-

can Moss, Suchit Subhaschandra, et al. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 5–14. ACM, 2017.

[160] Misja Nuyens and Adam Wierman. The foreground-background queue: A survey. *Perform. Evaluation*, 65(3-4):286–307, 2008.

[161] NVIDIA. ResNext101 Training. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets/resnext101-32x4d.

[162] NVIDIA. Nvidia a100 tensor core gpu. https://www.nvidia.com/en-in/data-center/a100/, 2021.

[163] OpenAI. GPT-3 Checkpoint. https://github.com/openai/gpt-3/issues/1.

[164] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[165] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, pages 5206–5210. IEEE, 2015.

[166] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018.

[167] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[168] Cody Coleman Greg Diamos Paulius Micikevicius David Patterson Hanlin Tang Gu-Yeon Wei Peter Bailis Victor Bittorf David Brooks Dehao Chen Debojyoti Dutta Udit Gupta Kim Hazelwood Andrew Hock Xinyuan Huang Atsushi Ike Bill Jia Daniel Kang David Kanter Naveen Kumar Jeffery Liao Guokai Ma Deepak Narayanan Tayo Oguntebi Gennady Pekhimenko Lillian Pentecost Vijay Janapa Reddi Taylor Robie Tom St. John Tsuguchika Tabaru Carole-Jean Wu Lingjie Xu Masafumi Yamazaki Cliff Young Peter Mattson, Christine Cheng and Matei Zaharia. Mlperf training benchmark. 2020.

[169] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In Jennifer G. Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, volume 80 of *Proceedings of Machine*

Learning Research, pages 4092–4101. PMLR, 2018.

[170] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 433–448. USENIX Association, 2014.

[171] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *J. Mach. Learn. Res.*, 20:53:1–53:32, 2019.

[172] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. Tunability: Importance of hyperparameters of machine learning algorithms. *Journal of Machine Learning Research*, 20(53):1–32, 2019.

[173] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *Proceeding of the 41st Annual International*

Symposium on Computer Architecuture (ISCA), ISCA 2014, pages 13–24, 2014.

[174] PyTorch. Torch elastic. https://pytorch.org/elastic/0.2.0/index.html.

[175] PyTorch. Word-level language modeling rnn. https://github.com/pytorch/examples/tree/master/word_language_model.

[176] PyTorch. Pytorch training examples. https://github.com/pytorch/examples/tree/master/imagenet, 2020.

[177] RedHat. Using mlock to avoid page I/O. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/reference_guide/using_mlock_to_avoid_page_io, 2021.

[178] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 18:1–18:13, Piscataway, NJ, USA, 2016.

[179] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[180] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael

Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[181] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[182] Supheakmungkol Sarin, Knot Pipatsrisawat, Khiem Pham, Anurag Batra, and Luís Valente. Crowdsource by google: A platform for collecting inclusive and representative machine learning data. 10 2019.

[183] Kaz Sato. Google's first tensor processing unit. https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu, 2020.

[184] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[185] Frank Seide and Amit Agarwal. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016.

[186] Aashaka Shah, Chao-Yuan Wu, Jayashree Mohan, Vijay Chidambaram, and Philipp Krähenbühl. Memory optimization for deep networks. In

9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021.

[187] Faisal Shahzad, Markus Wittmann, Thomas Zeiser, Georg Hager, and Gerhard Wellein. An evaluation of different I/O techniques for checkpoint/restart. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 1708–1716. IEEE, 2013.

[188] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[189] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[190] Utku Sirin and Anastasia Ailamaki. Micro-architectural analysis of OLAP: limitations and opportunities. *Proc. VLDB Endow.*, 13(6):840–853, 2020.

[191] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

[192] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 24(7):412–418, 1981.

[193] Mustafa Suleyman. Using ai to give doctors a 48-hour head start on life-threatening illness. https://deepmind.com/blog/article/predicting-patient-deterioration, 2020.

[194] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[195] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[196] TensorFlow. Tensorflow checkpoint manager. https://www.tensorflow.org/api_docs/python/tf/train/CheckpointManager.

[197] TensorFlow. Tensorflow iterator checkpointing. https://www.tensorflow.org/guide/data#iterator_checkpointing.

[198] Devesh Tiwari, Saurabh Gupta, and Sudharshan S. Vazhkudai. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*

2014, Atlanta, GA, USA, June 23-26, 2014, pages 25–36. IEEE Computer Society, 2014.

[199] Andrew Tomkins, R Hugo Patterson, and Garth Gibson. Informed multi-process prefetching and caching. In *ACM SIGMETRICS Performance Evaluation Review*, volume 25, pages 100–114. ACM, 1997.

[200] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. alsched: algebraic scheduling of mixed workloads in heterogeneous clouds. In Michael J. Carey and Steven Hand, editors, *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, page 25. ACM, 2012.

[201] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141. ACM Press, 1998.

[202] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[203] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh

Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[204] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4534–4542, 2015.

[205] Guanhua Wang, Amar Phanishayee, Shivaram Venkataraman, and Ion Stoica. Blink: A fast NVLink-based collective communication library. In *Proceedings of the 3rd Conference on Machine Learning and Systems, MLSys '20*, Austin, TX, USA, 2020.

[206] Wikipedia. Page cache. https://en.wikipedia.org/wiki/Page_cache, 2020.

[207] Gerhard J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Inf. Process. Lett.*, 64(6):293–297, 1997.

[208] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[209] 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 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[210] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[211] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 5987–5995. IEEE Computer Society, 2017.

[212] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *arXiv preprint arXiv:1910.05124*, 2019.

[213] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. *arXiv preprint arXiv:1910.01196*, 2019.

[214] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*, 2017.

[215] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD batch size to 32k for ImageNet training. *arXiv preprint arXiv:1708.03888*, 6, 2017.

[216] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[217] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association.

[218] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):1568–1581, 2020.

[219] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices.

In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[220] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 337–352. USENIX Association, 2020.

[221] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156. IEEE, 2018.

[222] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

Vita

Jayashree Mohan received her Bachelor of Technology in Computer Engineering from the National Institute of Technology, Karnataka in 2016. She then joined the University of Texas at Austin as a PhD student in the CS Department in 2016 under the guidance of Prof. Vijay Chidamabaram. During her PhD, she completed a parallel Master's study and received her M.S. in Computer Science in 2020.

Her research interests include performance and reliability of storage systems. Her thesis explores the role of storage stack and data pipeline in deep learning. Her doctoral research was funded by the Microsoft Research PhD Fellowship 2019-21. During the course of her doctoral work, she has interned at various labs of Microsoft Research - India, Cambridge (UK), and Redmond. She was mentored by Dr. Amar Phanishayee at MSR Redmond for all the work done as a part of this dissertation.

Permanent address: jayashree2912@gmail.com

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.