RESOURCE-EFFICIENT EXECUTION OF DEEP LEARNING COMPUTATIONS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Deepak Narayanan
August 2021

© 2021 by Deepak Narayanan. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This work is licensed under a Creative Commons Attribution-Noncommercial 3.0 United States License.
http://creativecommons.org/licenses/by-nc/3.0/us/

This dissertation is online at: https://purl.stanford.edu/qx792hd7022

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matei Zaharia, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kayvon Fatahalian

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Re

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Deep Learning models have enabled state-of-the-art results across a broad range of applications. Training these models, however, is extremely time- and resource-intensive, taking weeks on clusters with thousands of expensive accelerators in the extreme case. As Moore's Law slows down, numerous parallel accelerators have been introduced to meet this new computational demand. This dissertation shows how model- and hardware-aware optimizations in software systems can help intelligently navigate this heterogeneity. In particular, it demonstrates how careful automated scheduling of computation across levels of the software stack can be used to perform distributed training and resource allocation more efficiently.

In the first part of this dissertation, we study pipelining, a technique commonly used as a performance optimization in various systems, as a way to perform more efficient distributed model training for both models with small training footprints and those with training footprints larger than the memory capacity of a single GPU. For certain types of models, pipeline parallelism can facilitate model training with lower communication overhead than previous methods. We introduce new strategies for pipeline parallelism, with different tradeoffs between training throughput, memory footprint, and weight update semantics; these outperform existing methods in certain settings. Pipeline parallelism can also be used in conjunction with other forms of parallelism, helping create a richer search space of parallelization strategies. By partitioning the training graph across accelerators in a model-aware way, pipeline parallelism combined with data parallelism can be up to 5$\times$ faster than data parallelism in isolation. We also use a principled combination of pipeline parallelism, tensor model parallelism, and data parallelism to efficiently scale training to language models with a trillion parameters on 3072 A100 GPUs (aggregate throughput of 502 petaFLOP/s, which is 52% of peak device throughput).

In the second part of this dissertation, we show how heterogeneous compute resources (e.g., different GPU generations like NVIDIA K80 and V100 GPUs) in a shared cluster (either in a private deployment or in the public cloud) should be partitioned among multiple users to optimize objectives specified over one or more training jobs. By formulating existing policies as optimization problems over the allocation, and then using a concept we call effective throughput, policies can be extended to be heterogeneity-aware. A policy-agnostic scheduling mechanism then helps realize

the heterogeneity-aware allocations returned by these policies in practice. We can improve various scheduling objectives, such as average completion time, makespan, or cloud computing resource cost, by up to 3.5×, using these heterogeneity-aware policies. Towards the end of this dissertation, we also touch on how the dynamic pricing information of spot instances can be plugged into this heterogeneity-aware policy framework to optimize cost objectives in the public cloud. This can help reduce cost compared to using more expensive on-demand instances alone.

Acknowledgements

It truly takes a village to produce a Ph.D. The 6 years that ultimately culminated in this document have had many highs and lows, and I am deeply grateful to the many people who have helped me (in small ways and large) finally find light at the end of the tunnel.

I owe a big debt of gratitude to my advisor, Matei Zaharia. When I joined Stanford, Matei was actually not even faculty at Stanford. Through a sequence of fortunate events, he ended up moving to Stanford right before my second year, right in time for my fourth rotation. One thing led to another and we ended up advisor and advisee. From the get go, Matei was incredibly supportive, always humble, and never overbearing. He allowed me to continue an internship project from Microsoft Research that ended up being the PipeDream work that features prominently in this dissertation, and had no qualms with me jumping into a nascent research area (systems for machine learning) that neither he nor I had much experience in at the time. Besides insightful technical advice, Matei taught me a lot about technical communication; my writing and speaking have improved immensely over the years from his feedback. He also has had a significant impact on how my research ethos has evolved; his experience as Chief Technologist at Databricks was always useful in grounding my research with what was going on in industry.

Amar Phanishayee took a big gamble in 2015 taking me on as an intern before I started my Ph.D. at Stanford. I had scarce research experience at that point, and Amar really taught me the ropes: how to formulate questions and hypotheses, how to design experiments that tested these hypotheses, and how to automate as much as one possibly could to make it easy to run these experiments. Amar's enthusiasm in our almost daily morning checkins was contagious, and I could not help but feel excited about the work we were doing together. I spent a total of four wonderful summers at Microsoft Research over the course of my Ph.D., and needless to say, Amar features prominently in the work presented in this dissertation.

I am grateful to Chris Ré and Kayvon Fatahalian for serving on my reading committee and greatly improving this document. More generally, Chris and Kayvon have been hugely inspirational figures for me in the Stanford CS department. Chris's various projects that found a way to marry systems building with strong theoretical foundations, and Kayvon's systems that produced incredibly cool demos, were always exemplars of great research for me.

Mohammad Shoeybi was kind enough to respond to a cold email regarding a potential collaboration in June 2020. Working with him, Jared Casper, Patrick LeGresley, Vijay Korthikanti, Mostofa Patwary, and Bryan Catanzaro on the NVIDIA ADLR team for a year was immensely rewarding. I learnt a lot about how machine learning models are trained in industry, and also got to deploy my research at scales that only seemed like a pipe dream (apologies for the pun :P) at Stanford.

The work in this dissertation would not have been possible without my collaborators. I strongly believe that research is best done when people with different expertises come together, and I was lucky to have some amazing co-authors, who taught me so much: Aaron Harlap, Akshay Agrawal, Amar Phanishayee, Anil Shanbhag, Bryan Catanzaro, Chris Ré, Cody Coleman, Daniel Kang, Dmitri Vainbrand, Edward Gan, Fiodar Kazhamiaka, Gina Yuan, Gregory R. Ganger, Holger Pirk, James Thomas, Jared Casper, Jian Zhang, Julie Bernauer, Keshav Santhanam, Kexin Rong, Kunle Olukotun, Luigi Nardi, Malte Schwarzkopf, Matei Zaharia, Mohammad Shoeybi, Mostofa Patwary, Nikhil R. Devanur, Parimarjan Negi, Patrick LeGresley, Peter Bailis, Peter Kraft, Phillip B. Gibbons, Pratiksha Thaker, Prethvi Kashinkunti, Rahul Palamuttam, Sahaana Suri, Saman Amarasinghe, Samuel Madden, Shoumik Palkar, Srikanth Kandula, Stephen Boyd, Tian Zhao, Vijay Korthikanti, and Vivek Seshadri.

The saying goes that one only really appreciates the value of something in absentia. I certainly believe this to be the case with 432 and my officemates: Firas Abuzaid, Shoumik Palkar, and James Thomas. Firas was the energizer bunny of our office, always full of life and basketball wisdom (a direct quote from Firas: "my game is modeled on Steph Curry, but I'm not *quite* as good"). Shoumik was the funny one, always with a joke or incredibly accurate impersonation up his sleeve. He and I had great fun as roommates at various conferences. James was the perpetually late one who would show up at the office just in time to leave for lunch. I have been lucky to be friends with James from MIT when we lived in the same undergraduate dormitory; the last year and a half of the pandemic were made much more tolerable with our lunches at the dining hall, and games of football and basketball. Unfortunately, our time together in 432 was cut short by the shelter-in-place order, but I will look back at our times together in that office with great fondness.

I joined the FutureData group in its infancy, when it was just a bunch of second years (also by default, the "senior" students in the group) and the PIs, Peter Bailis and Matei. The group has become a tiny bit larger since (:P), but still retains that vibrancy and friendliness from our early days, while also featuring a breadth of expertise and interests that I think is hard to find in an academic lab. I have been fortunate to work with Cody, Daniel, Deepti, Edward, Fiodar, Gina, Kai Sheng, Keshav, Kexin, Lingjiao, Omar, Peter B., Peter K., Pratiksha, Sahaana, and Trevor, in some shape or form over the last 5 or so years, and have learnt many things, both technical and otherwise, along the way in my interactions with them.

I am appreciative of my friends through the years at Stanford and outside: thank you for giving me joy (and also keeping me sane outside of work and the constant grind of paper deadlines).

Last, but definitely the most: a huge thanks to my mom, who has been the main, always pervasive, guiding light in my academic journey. It is not hyperbolic to say that this dissertation would not be possible without her. She was instrumental in recognizing and nurturing my interest in math and science when I was very young, nudged me towards research when the time came to decide on a career path, and continues to this day to push me to reach my full potential. Through no fault of her own, she often had to deal with me at my lowest points, which cannot be a pleasant experience. She was kind enough to visit me every year of my Ph.D. (apart from the last one due to COVID-19) from India for extended periods of time. I dedicate this dissertation to her.

To my mom.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Deep Neural Networks (DNNs) have facilitated tremendous progress across a range of applications, including image classification [102, 154, 84], translation [171], language modeling [118, 45], and video captioning [167]. As DNNs have become more widely deployed, they have also become more computationally expensive to train. For example, training the state-of-the-art GPT-3 language model [45] requires trillions of floating point operations. These computations will only become more expensive going forward as ML models and training datasets become larger.

The end of Moore's Law has led to the rapid adoption of a number of parallel architectures, such as multicore CPUs (with SIMD), GPUs, FPGAs, and domain-specific accelerators like the TPU, each with different programming models and performance characteristics (e.g., number of cores, SIMD lane width, cache sizes) to meet this new computational demand. Achieving high performance on these architectures is challenging for non-expert programmers like Machine Learning engineers, who do not want to understand the low-level performance intricacies of complicated parallel hardware. At the same time, it is increasingly becoming important to achieve high device utilization in order to reduce the runtime and cost of training, and keep training computationally feasible.

ML models are composed of different operators (or layers). The types of operators used are highly task-dependent, e.g., convolutions are used for vision tasks, transformers with various multi-head attention mechanisms are used for language tasks, and multi-layer perceptrons are used for recommendation tasks. Each of these operator types perform differently across hardware architectures. Consequently, ML models display *performance heterogeneity*, and executing a given model's computation the same way across accelerator types can lead to significant performance underutilization. For example, distributing training over multiple accelerators using the same parallelization strategy can lead to sub-optimal results (e.g., up to 90% of total time can be spent on communication when using data parallelism [Figure 2.1]).

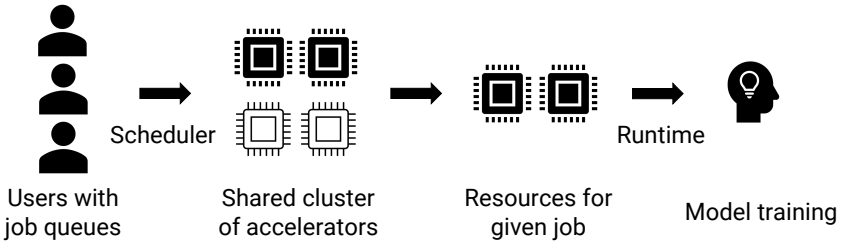


Figure 1.1: Typical model training workflow: a scheduler first determines how shared resources should be allocated to various users while optimizing a specified macro-objective; a runtime then determines how to best use these resources to train a given model. This dissertation addresses two concrete problems in this pipeline: resource allocation to determine how a pool of resources should be shared among multiple users, and distributed training to determine how a given job's resource allocation should be optimally used to train the target model as fast as possible.

Consequently, model- and hardware-aware optimization is essential, particularly as heterogeneity in models and hardware architectures will only increase going forward.

To amortize cost, compute resources in industry and academia are often available as part of a shared cluster. *Cluster schedulers* allocate resources to various users based on their demands and a globally optimized objective function (e.g., fairness). Once given resources, users can then use a *training framework* like PyTorch or TensorFlow [134, 36] to train their model. This end-to-end workflow is shown in Figure 1.1. As we shall show in this dissertation, inefficiencies exist in both stages of this end-to-end workflow.

1.2 Dissertation Overview

> **Thesis Statement:** *Careful automated scheduling of computation on (heterogeneous) resources across the software stack (e.g., cluster scheduler, training execution runtime) can significantly increase model training throughput.*

This dissertation introduces ideas that try to make it easier for programmers to achieve high performance on parallel hardware for model training. In particular, the central focus of this dissertation is on the design of software systems that can execute deep learning computations in a more resource-efficient and scalable way with *minimal* user supervision.

In demonstrating the central thesis, this dissertation examines the two related but orthogonal problems shown in Figure 1.1: resource allocation across jobs and distributed execution within a job. Both of these are scheduling problems but at different granularities. Concretely, we try to answer the following questions:

1. At the micro level, given a budget of training resources (e.g., n GPUs of a specific type), how

should operators in a single deep neural network (DNN) model be partitioned among these resources to maximize overall training throughput?

2. At the macro level, how should heterogeneous resources in a shared cluster be allocated to ML training jobs to optimize scheduling objectives specified over one or more jobs (e.g., fairness, cost) in both private and public cloud cluster deployments?

To address the first question, we study how to adapt *pipelining*, an optimization used in conventional compilers and runtime systems [105, 39, 37, 47], to accelerate DNN training performance with little to no reduction in the final accuracy of the model. Pipelining makes it possible to assign each participating device a subset of the layers in the model, thus facilitating more communication-efficient parallelization schemes for *certain* types of models. Existing work [86, 54] has looked at using pipeline parallelism for a narrow set of models, but does not clearly outline the associated tradeoffs of the proposed strategies, and also suffers from expensive pipeline stalls. We make the following concrete contributions: (a) we discuss the challenges associated with using pipeline parallelism for distributed training, (b) we introduce new strategies for pipeline parallelism that address these challenges, and discuss the tradeoffs associated with each along the dimensions of throughput, memory footprint, and weight update semantics (Table 1.1). These new strategies can outperform existing approaches by as much as 3.2×, c) we observe that pipeline parallelism can be composed with other existing modes of parallelism, but these various modes of parallelism interact in *non-trivial* ways. We empirically and analytically analyze the interactions of pipeline parallelism with data and tensor model parallelism. The principled combination of these parallelism methods can train models with up to a trillion parameters using 3000+ GPUs with high efficiency (52% of theoretical peak device throughput, including communication across GPUs and data loading), d) we show that an optimizer can automatically determine how to compose a subset of these parallelism modes (given a number of workers to work with) to maximize training throughput. Our automated partitioning algorithm recommends combinations of pipeline and data parallelism that are up to 5× faster than data parallelism alone.

To address the second question, we introduce a general way to convert a wide range of scheduling policies into heterogeneity-aware policies, improving diverse objectives in an automated way in a system called Gavel. In Gavel, we show that existing policies can be expressed as optimization problems, and that these optimization problems can be extended easily to be heterogeneity-aware using a concept we call *effective throughput*. Using this framework, we can write policies that optimize for a host of objectives, including fairness, makespan, and dollar cost. We use a round-based scheduling mechanism to ensure that jobs subsequently actually achieve their computed optimal allocation in practice. The dollar cost policies can also be adapted to determine how to allocate ephemeral resources (e.g., spot instances) in the public cloud, whose price and availability can change with time, to various long-running ML training jobs. On heterogeneous clusters, Gavel is able to improve objectives such as average job completion time by as much as 3.5×.

1.2.1 Non-Goals

We observe that generating efficient low-level code given a higher-level description of computations (as done by systems like TVM and Halide [139, 52]) or automatically discovering semantics-preserving transformations for model sub-graphs (as done by systems like TASO [95]) can also be thought of as types of micro-scheduling optimizations; however, these are outside the scope of this dissertation. Instead, we focus on a narrow type of micro-scheduling optimizations: efficient parallelization given a budget of training resources.

1.3 Accelerating Distributed Model Training using Pipelining

As DNN models and training datasets become larger, many organizations are adopting distributed DNN training to either decrease training time or train very large models that do not fit on a single accelerator (e.g., language models like OpenAI's GPT-3 [45]). Today, distributed training is largely performed using *intra-batch parallelism* techniques (data parallelism, model parallelism, and hybrid parallelism that combines the two), where training for a single batch of input samples is parallelized over multiple workers. These techniques, however, all hit fundamental scaling limits, either by introducing expensive all-to-all communication into the computation graph, or by lowering compute resource utilization by forcing workers to wait for intermediate outputs from other workers (in inter-layer model parallelism). We show how to use *pipelining* as a parallelization dimension for DNN training: a batch is broken into smaller microbatches and workers process *different* microbatches concurrently (one pipeline-parallelism schedule is shown in Figure 1.2). Pipelining enables new distributed training strategies that can outperform previous methods, achieving low communication overhead and high resource utilization for certain types of models.

Pipelining is a common performance optimization used in various systems, such as for instruction-level parallelism in processors. However, pipelining in distributed model training presents one key difference over previous computer systems that use pipelining: training is bidirectional and stateful (Chapter 2). A forward pass through the model is followed by a backward pass for the same set of samples which updates weight parameters, and intermediate outputs and weight parameters used in the forward pass are needed in the backward pass. This is shown in Figure 1.3. Naïve pipelining can lead to weight version mismatches across forward and backward passes that compromise the accuracy of the final trained model.

PipeDream [80, 125] is a system that versions state (weight parameters and intermediate activations) to ensure clean weight update semantics. In steady state, each worker in PipeDream processes a forward pass for one microbatch followed by a backward pass for a potentially different microbatch (called a 1F1B schedule). PipeDream supports multiple ways of stashing weight versions to trade off between memory footprint, throughput, and the number of samples over which weight gradients are averaged before updating model parameters. PipeDream's memory-efficient modes

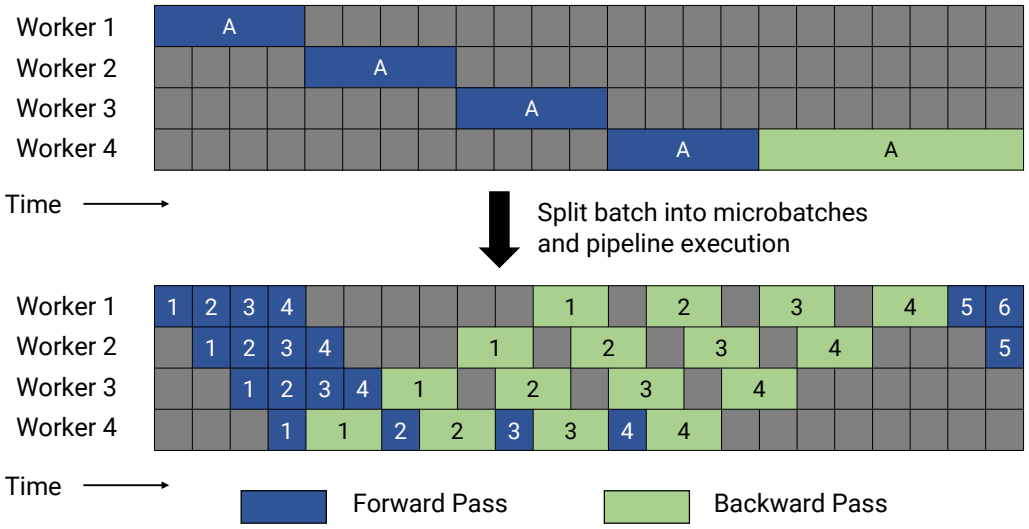


Figure 1.2: With pipeline parallelism, a batch of samples is split into microbatches, and then execution is pipelined across the microbatches. Here, the batch A is split into 4 microbatches. In this particular pipelining schedule, the pipeline is first flushed at the end of a batch, and then the optimizer is stepped.

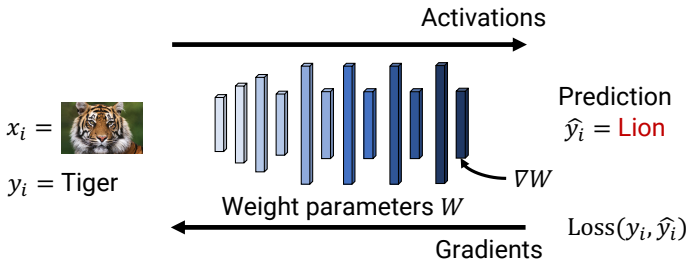


Figure 1.3: Deep Neural Network (DNN) models are composed of operators stacked one on top of each other, called layers. Model training proceeds in iterations. In each iteration, a forward pass through the model is followed by a backward pass where model gradients are computed; these gradients can then be used to update the model's parameters to prevent it from making the same mistakes (e.g., incorrectly predicting that a picture of a "tiger" is in fact a "lion").

like `2BW` (Chapter 3) offer a way to train large models (e.g., GPT-3 [45]) with training footprints much larger than the memory capacity of a single worker by stashing fewer weight versions on each worker. The specific pipelining strategy used has an impact on the throughput, memory footprint, and weight update semantics; Table 1.1 shows these tradeoffs.

PipeDream automatically determines how best to partition operators across workers by reasoning about the computation times of each operator and the sizes of the tensors communicated across workers. Instead of using the same parallelization strategy for all models, PipeDream ensures that

Pipelining Scheme	Throughput Overhead	Memory Footprint	Update Semantics
GPipe [86]	High	Medium	Strict
PipeDream (Chapter 2)	Zero	High	Relaxed
PipeDream-2BW (Chapter 3)	Zero	Low	Relaxed*
PipeDream-Flush (Chapter 3)	High	Very Low	Strict
Interleaved (Chapter 4)	Medium	Very Low	Strict

Table 1.1: Comparison of various pipelining approaches discussed in this dissertation along three dimensions: throughput overhead imposed from pipelining, memory footprint, and weight update semantics. For overhead and memory footprint, lower is better. PipeDream-2BW performs gradient accumulation: its relaxed weight updates use gradients averaged over more samples compared to PipeDream, which might not always be feasible.

the partitioning is model- and hardware-aware.

PipeDream is able to train models to the same accuracy target up to 5× faster than data parallelism. PipeDream, when optimizing for lower memory footprint (using the 2BW memory-efficient scheme), can train large language models with 3.5 billion parameters up to 6.9× faster than model parallelism (data parallelism cannot be deployed in settings where models are too large to fit on a single worker). PipeDream and PipeDream-2BW train models with similar convergence trajectories to existing widely-used approaches like data parallelism, indicating that weight stashing and 2BW provide data parallelism-like weight update semantics.

Pipeline parallelism can also be composed with other parallelization strategies like data and tensor model parallelism, since each of these strategies in isolation break down at large accelerator counts: data parallelism is limited by the batch size, pipeline parallelism by the number of layers in the model, and tensor model parallelism by the number of GPUs in a single server. The composition of these techniques, which we call PTD-Parallelism (PTD-P for short) allows us to train GPT models with up to a trillion parameters on 3072 GPUs with high efficiency (52% of theoretical peak). PTD-P is described in Chapter 4.

1.4 Heterogeneous Resource Allocation for Deep Learning in Shared Clusters and Clouds

Different types of DNN models display highly heterogeneous performance behavior across accelerator types, e.g., a ResNet-50 image classification model is about 10× faster on a later-generation Nvidia V100 GPU compared to an older-generation K80 GPU, whereas a Transformer model is only about 3.3× faster (Figure 1.4). We expect heterogeneity to increase as newer accelerator generations and domain-specific accelerators are released. This raises a difficult question for ML users: how should an organization allocate accelerators, which usually span multiple generations, among its workloads in either a private cluster or in the public cloud? This is especially challenging since

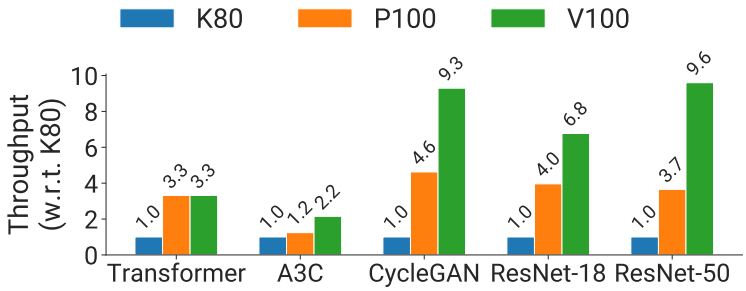


Figure 1.4: Training throughputs for various ML models. The magnitude of speedup across GPU generations varies significantly across models.

organizations typically wish to optimize for a wide range of objectives, such as inter-user fairness or total dollar cost. Prior resource allocation algorithms that optimize these objectives generally do not consider device heterogeneity. One way to deal with heterogeneous resources is to manage them separately and defer resource choice to the user; however, this can lead to sub-optimal outcomes (e.g., all users picking the fastest resource type available, increasing the queuing delay for these in-demand resources, while leaving other slower resources idle).

Gavel [129] is a scheduling system that determines how heterogeneous resources in on-premise and cloud deployments should be automatically shared among training jobs from multiple users to optimize a wide range of classical resource allocation objectives (Chapter 5). We observe that existing policy objectives can be expressed as a function of a job's observed throughput. Consequently, policies can be formulated as optimization problems over the allocation. We show how to extend these optimization problems to consider heterogeneity by extending allocations to represent the fractions of time each job should spend on each resource type, and using effective throughput, i.e., the time-weighted average of throughputs jobs observe on each resource type, in the policy objectives. Gavel's heterogeneity-aware policies can also consider performance optimizations such as space sharing (concurrent execution of applications to improve utilization), by changing the allocation representation. Commonly used policies can be expressed as linear problems, which can be solved efficiently using off-the-shelf solvers. Gavel also introduces a policy-agnostic round-based scheduling mechanism that takes the allocation returned by the policy and ensures that each job receives compute time on resources according to the computed allocation. This round-based scheduling mechanism makes it possible to use Gavel for new policies; previous systems would need complete system rewrites in order to support objectives that they were not originally designed for.

Gavel's heterogeneity-aware policies reduce objectives like average job completion time by 3.5× compared to previous schedulers that are heterogeneity-agnostic, and sustain up to 1.5× higher load using the *same* cluster (Figure 1.5) by more efficiently giving resources to compatible jobs (e.g., jobs that are very slow on a specific GPU type are not given time on that GPU type).

Figure 1.5: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-single trace.

In this dissertation, we also consider the implications of using heterogeneity-aware policy formulations in an elastic spot market, where prices and availability of instances can change with time (Chapter 6). Heterogeneity-aware scheduling in this regime can lead to significant cost savings (up to 3.5×) by moving ML workloads across instances as needed as prices and availability change.

1.5 Overview of Results

In this dissertation, we show that we can train models with low training footprints up to 5× faster than existing methods like data parallelism, reach 52% of theoretical peak device throughput when running training iterations for a model with a trillion parameters (which has a training memory footprint far larger than the memory capacity of a single GPU) using 3072 GPUs, and improve average job completion time by 3.5× on a cluster with heterogeneous resources, by carefully scheduling computation on heterogeneous resources. In particular, we have designed and built automatic partitioning and scheduling algorithms that take in model profiles as input (either fine-grained at the operator level for distributed model training, or coarse-grained at the model or job level for resource allocation) and determine how best to place and orchestrate computation on the available resources.

1.6 Previously Published Work

This dissertation features the following previously published work:

- **PipeDream: Generalized Pipeline Parallelism for DNN Training** [125].

  Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, Matei Zaharia. *SOSP 2019*.

- **Memory-Efficient Pipeline-Parallel DNN Training** [127].

Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, Matei Zaharia. *ICML 2021*.

- **Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM** [131].

  Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, Matei Zaharia. *SuperComputing 2021*.

- **Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads** [129].

  Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, Matei Zaharia. *OSDI 2020*.

- **Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training** [128].

  Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, Matei Zaharia. *DISPA 2020 (workshop at VLDB 2020)*.

1.7 Roadmap

This dissertation is organized into two parts.

Part I describes how we can distribute tasks for training jobs in a heterogeneity-aware way with the help of pipeline parallelism:

- **Chapter 2** introduces the challenges that need to be solved in applying pipeline parallelism to distributed model training, and outlines solutions to these challenges for models that fit on a single worker.
- **Chapter 3** describes how pipeline parallelism can be adapted to train models with training footprints much larger than the memory capacity of a single GU.
- **Chapter 4** describes the limitations of existing parallelization strategies in isolation at large scale (thousands of GPUs), and shows how a principled combination of data, tensor, and pipeline parallelism can be used to train models of up to a trillion parameters.

Part II describes how we can allocate heterogeneous resources (both in private clusters and in public clouds) to different training jobs:

- **Chapter 5** introduces a way to allocate heterogeneous resources to different types of training jobs while optimizing for various objectives (e.g., fairness, makespan).
- **Chapter 6** shows how this policy framework can be used to optimize for cost-based objectives, and also studies how the availability and price of spot instances change with time, and the implications of these on ML training workloads running on public cloud infrastructure.

Part I

Scheduling at the Microscale: Pipeline Parallelism for Efficient Distributed Training of Single Jobs

Chapter 2

Pipeline Parallelism and the PipeDream System

2.1 Introduction

DNN training proceeds in iterations of forward and backward pass computations. In each iteration, the training loop processes a batch of input data and performs an update to the model parameters. Current approaches to distributed training focus on parallelizing each iteration of the optimization algorithm across a set of workers. For example, data parallelism partitions the input data across workers [102], model parallelism partitions operators across workers [62, 55], and hybrid schemes partition both [94, 96, 100]. Unfortunately, such parallelization schemes can suffer from high communication costs at large scale. For example, Figure 2.1 shows the communication overhead for data parallelism across five different DNN models on three different types of multi-GPU servers. Over 32 GPUs, the communication overhead for some models, computed as the percentage of total time spent on communication stalls, is as high as 90% due to expensive cross-server `all_reduce` communication. Communication overheads are high even on servers where GPUs within the server are connected by dedicated interconnects like `NVLink` [22]. Moreover, rapid increases in GPU compute speed over time will further shift the bottleneck of training towards communication for all models.

In this chapter, we outline the challenges with applying pipelining, a common optimization used in a variety of systems, to distributed model training. With pipeline parallelism, the model is divided among available workers, with a group of consecutive operators (called layers in DNN terminology) in the operator graph assigned to each worker. Computation and communication of *different inputs* is then overlapped in a pipelined fashion. This process can greatly reduce inter-worker communication because it limits the communication to layer inputs and outputs (activations in the forward pass and gradients in the backward pass) across consecutive layers assigned to different workers, which for

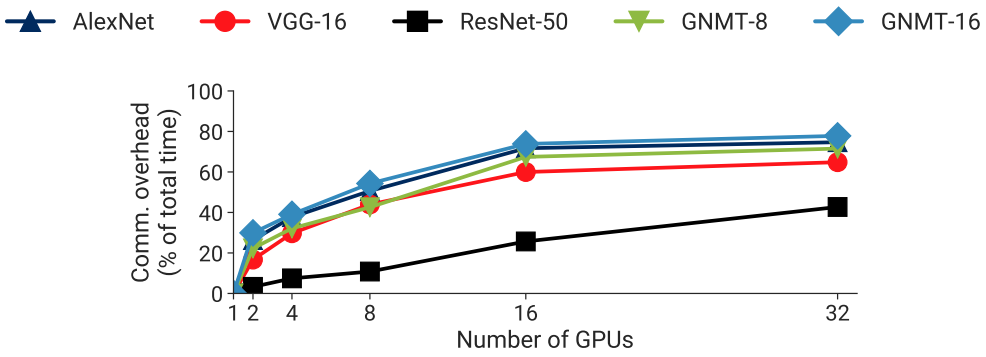
many models are much smaller than the size of the entire model.

Despite its potential, pipelining with DNN training poses an important challenge not present in traditional pipelining: DNN training is *bi-directional*—the forward pass is followed by a backward pass through the same layers in reverse order, using state and intermediate results from the forward pass. To keep the pipeline full and thus achieve high hardware efficiency, a naïve scheduling mechanism might inject all input batches in an epoch into the pipeline, first completing forward passes for all input batches followed by backward passes. However, this approach suffers from low statistical efficiency [58] and high memory footprint, increasing the number of passes through the dataset needed to produce a high-quality model (or preventing the model from reaching the desired target accuracy, since gradients are averaged over all training samples [43, 116]) and the amount of stashed state needed to complete backward passes. To improve statistical efficiency, one could inject only a subset of m inputs into the pipeline, and apply weight updates every m inputs, as recently proposed by GPipe [86]. However, this reduces hardware efficiency due to more frequent pipeline flushes. Inter-layer model parallelism corresponds to an extreme case of this (m is 1).

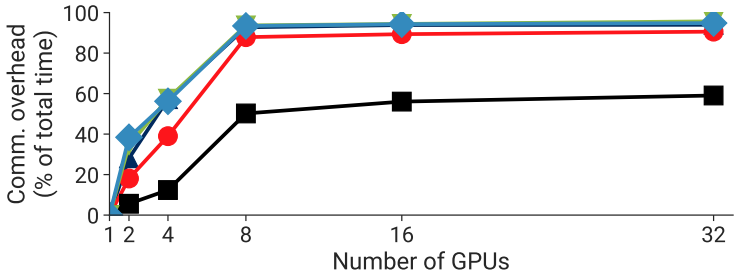
In this chapter, we introduce PipeDream, a system we built that uses *pipeline parallelism* to enable faster DNN training. PipeDream, as we introduce it in this chapter, presents one possible solution to the challenges imposed from using pipelining for distributed model training. However, other solutions are also possible; we describe alternate solutions in Chapters 3 and 4 of this dissertation.

PipeDream achieves high hardware efficiency with no pipeline stalls in steady state, and comparable statistical efficiency to data parallelism using the same number of workers. Given a pipeline of groups of consecutive layers executed on different workers (called a stage), PipeDream uses a scheduling algorithm called 1F1B to keep hardware well utilized while achieving semantics similar to data parallelism. In 1F1B's steady state, each worker strictly alternates between forward and backward passes for its stage, ensuring high resource utilization (negligible pipeline stalls, no pipeline flushes) even in the common case where the backward pass takes longer than the forward pass. 1F1B also uses *different versions* of model weights to maintain statistical efficiency comparable to data parallelism. Each backward pass in a stage results in weight updates; the next forward pass uses the latest version of weights available, and "stashes" a copy of these weights to use during the corresponding backward pass. Although the forward pass will not see updates from incomplete in-flight inputs, learning is still effective because model weights change relatively slowly and bounded staleness has been found effective in improving training speeds [59, 142]. However, for the backward pass to compute *numerically correct* gradients, the same weight version used during the forward pass must be used. This scheme results in slightly relaxed weight update semantics compared to GPipe (see Table 1.1). PipeDream limits the number of "in-pipeline" inputs to the minimum needed to keep the pipeline full, reducing memory overhead.

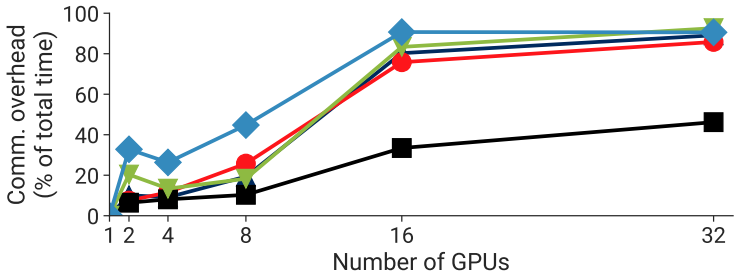
Operating the pipeline at peak throughput also requires that all stages in the pipeline take



(a) Instances with 8 1080Tis (private cluster).



(b) Instances with 4 V100s (Azure).



(c) Instances with 8 V100s and `NVLink` (EC2).

Figure 2.1: Communication overhead of data-parallel training using different multi-GPU server instances using PyTorch 1.1, NCCL [18], and `fp32` precision. We use the largest per-GPU batch size that fits in GPU memory, and keep the per-GPU batch size constant as the number of GPUs are scaled up (weak scaling).

roughly the same amount of time, since the throughput of a pipeline is bottlenecked by the slowest stage. PipeDream *automatically* determines how to schedule computation using the provided number of GPUs. In particular, its optimizer partitions the operators of the DNN based on a short profiling run performed on a single GPU, balancing computational load among the different stages while minimizing communication for the target platform. PipeDream effectively load balances even in the presence of model diversity (computation and communication) and platform diversity (interconnect topologies and hierarchical bandwidths). As DNNs do not always divide evenly among available workers, PipeDream may decide to use data parallelism for some stages—multiple workers can be assigned to a given stage, processing different inputs in parallel. Note that vanilla data parallelism corresponds to the pipeline having a single stage that is replicated. PipeDream extends 1F1B to incorporate round-robin scheduling across data-parallel stages, while making sure that gradients in a backward pass are routed to the corresponding worker from the forward pass since the same weight version and intermediate outputs need to be used for a correct gradient computation. The combined scheduling algorithm, 1F1B-RR, produces a static schedule of operators that each worker runs repeatedly, keeping utilization high across all workers. Thus, PipeDream executes a principled combination of pipeline and data parallelism.

Our evaluation, encompassing many combinations of DNN models, datasets, and hardware configurations, confirms the training time benefits of PipeDream's pipeline parallelism. Compared to data parallelism, PipeDream reaches a high target accuracy on multi-GPU machines up to 5.3× faster for image classification tasks, up to 3.1× faster for machine translation tasks, 4.3× faster for language modeling tasks, and 3× faster for video captioning models. PipeDream is also 2.6× – 15× faster than model parallelism, up to 1.9× faster than hybrid parallelism, and 1.7× faster than other approaches to pipelining such as GPipe.

2.2 Background and Related Work

A DNN model is composed of many operators organized into *layers*. When parallelizing DNN training, these layers may be partitioned over the available workers in different ways. In this section, we cover the broad parallelization strategies already proposed in the literature. We also highlight the challenges posed by DNN model and hardware diversity for effective parallelization.

2.2.1 Parallelization Strategies

Existing parallelization strategies split a single training iteration across available workers.

Data Parallelism. In data parallelism, inputs are sharded across workers. Each worker maintains a local copy of the model weights and trains on its own partition of inputs while periodically synchronizing weights with other workers, using either collective communication primitives like

`all_reduce` [76] or parameter servers [108]. The amount of data communicated is proportional to the number of model weight parameters and the number of workers participating in training.

The most commonly used form of data parallelism, referred to as *bulk synchronous parallel* or BSP [163][1], requires each worker to *wait* for gradients from other workers. Despite optimizations such as Wait-free Backpropagation [180], where weight gradients are sent as soon as they are available (common in modern frameworks), communication stalls are inevitable for large models where the time needed to synchronize gradients across workers can dominate computation time.

Figure 2.1 quantitatively shows the fraction of training time spent in communication stalls with data parallelism for different classes of DNNs using three types of servers: 8-1080Ti GPU instances linked over PCIe within servers and 25Gbps interconnects across servers, 4-V100 GPU instances without `NVLink` and 10Gbps interconnects across servers, and 8-V100 GPU instances with `NVLink` interconnects within servers and 25Gbps interconnects across servers.

We focus on four key takeaways. First, the communication overhead for many of these models is high despite using multi-GPU servers and state-of-the-art communication libraries like NCCL. Data parallelism scales well for models like ResNet-50, which have a large number of convolutional layers with compact weight representations, but scales less well for other models with LSTM or fully-connected layers, which have more dense weight representations. Second, applications distributed across multi-GPU servers are bottlenecked by slower inter-server links, as evidenced by communication overheads spiking and then plateauing when training scales out to multiple servers. Data parallelism for such hierarchical networks can be a poor fit, since the same number of bytes are sent over both high- and low- bandwidth channels. Third, as the number of data-parallel workers increases, communication overheads increase for all models, even if training is performed on a multi-GPU instance with `NVLink`. Coleman et al. [57] showed similar results. Fourth, as GPU compute speeds increase (1080Tis to V100s), communication overheads also increase for all models.

Other Data Parallelism Optimizations. Asynchronous parallel training (ASP) allows each worker to proceed with the next input batch before receiving the gradients from the previous batch. This approach improves hardware efficiency (time spent in each iteration) over BSP by overlapping computation with communication, but also introduces staleness and reduces statistical efficiency (number of iterations needed to reach a particular target accuracy) [60, 50].

Seide et al. [147, 146] looked at quantizing gradients to decrease the amount of data needed to be communicated over the network. This approximation strategy is effective in limited scenarios but lacks generality; it does not hurt convergence for some speech models [148], but has not been shown to be effective for other types of models. Others have explored techniques from the HPC literature to reduce the overhead of communication [76, 160, 41, 162], often using highly specialized networking hardware. Our work is complementary to these techniques and focuses mainly on

[1] In this dissertation, we use DP to refer to data-parallelism with BSP.

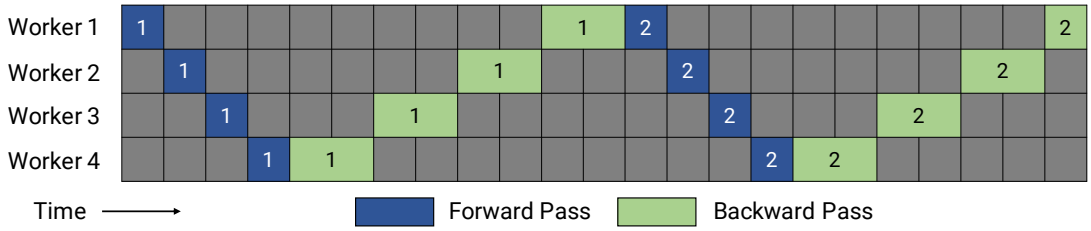


Figure 2.2: Model parallel training with 4 workers. Numbers indicate input ID, and backward passes takes twice as long as forward passes. For simplicity, we assume that communicating activations/-gradients across workers has no overhead.

improving the performance of parallel DNN training when using commodity accelerators and interconnects available in public clouds; our work looks at fundamentally different ways of partitioning the model training graph over training resources to reduce the number of bytes of data that need to be communicated between workers.

Recent work has demonstrated that using large batches is effective for training ResNet-50, especially when combined with Layer-wise Adaptive Rate Scaling (LARS) [76, 92, 177]. Large batches reduce the communication overhead by exchanging parameters less frequently; however, our experiments show that such techniques lack generality beyond ResNet-50 and pipeline parallelism can outperform the fastest LARS data-parallel option.

Model Parallelism. Model parallelism is used traditionally to train large models that do not fit on a single worker. With model parallelism [62, 55], the weight parameters in a model are split over available workers, with intermediate activations and gradients communicated across workers. Different forms of model parallelism are possible based on how operators are partitioned over workers. Inter-layer model parallelism (where each worker is assigned a subset of the layers or operators in the model) underutilizes resources since at most a single worker is active at any point in time (Figure 2.2). Tensor (intra-layer) model parallelism [153] involves splitting each layer over multiple workers, and leads to multiple all-to-all communication calls in the critical path (which are expensive collectively), limiting the number of model partitions to the number of GPUs in a single server. Chapter 4 discusses this in more detail.

Model parallelism requires programmers to determine how to partition their models across multiple GPUs [100], resulting in point solutions. Recent work explores the use of Reinforcement Learning to automatically perform device placement [121]. However, these techniques are time- and resource- intensive, and do not leverage the fact that DNN training can be thought of as a computational pipeline consisting of groups of consecutive layers – these assumptions make the optimization problem more tractable, allowing for exact solutions in polynomial time as we show in §2.4.1. FlexFlow [96] shows how to split a model graph using model and data parallelism, but does not consider pipelining, and can still suffer from poor resource utilization when sharding operators over

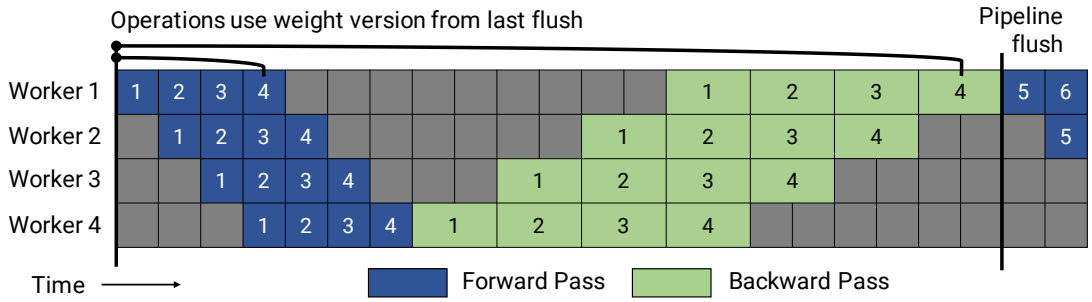


Figure 2.3: GPipe's pipeline parallelism approach. Frequent pipeline flushes lead to idle time where workers do not have inputs to process.

multiple workers or GPUs.

Hybrid Parallelism. Recent work has proposed splitting a single iteration of the optimization algorithm among *multiple* dimensions. One Weird Trick (OWT) [100] split the then-popular AlexNet model by hand, using data parallelism for convolutional layers that have a small number of weight parameters and large outputs, while choosing to not replicate fully connected layers that have a large number of weight parameters and small outputs. OWT does not use pipelining. FlexFlow [94] proposed splitting a single iteration along samples, operators, attributes, and parameters, and describes an algorithm to determine how to perform this splitting in an automated way. However, FlexFlow does not consider pipelining in its search space.

Pipeline Parallelism. Chen et al. [54] explored the potential benefits of pipelining batches in model-parallel training, but did not address the conditions necessary for good statistical efficiency and performance across a wide variety of real-world models. Huo et al. [88] explored parallelizing the backward pass. Our proposed solution parallelizes both forward and backward passes.

GPipe [86] uses pipelining in the context of model-parallel training for very large models. GPipe does not specify an algorithm for partitioning a model, but assumes a partitioned model as input. GPipe further splits a batch into m microbatches, and performs forward passes followed by backward passes for these m microbatches (see Figure 2.3, where m is 4). With a focus on training a large model like AmoebaNet, GPipe optimizes for memory efficiency; it uses existing techniques such as weight gradient aggregation and trades computation for memory by discarding activation stashes between the forward and the backward pass, instead opting to re-compute them when needed in the backward pass [53]. As a result, it can suffer from reduced hardware efficiency due to re-computation overheads and frequent pipeline flushes if m is small (§2.5.4).

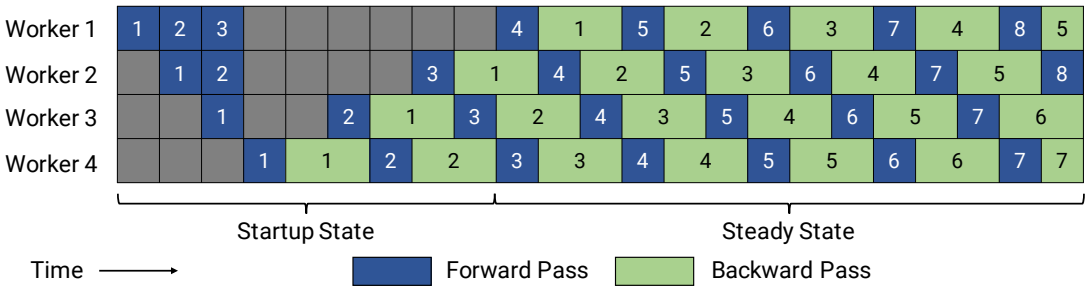


Figure 2.4: PipeDream pipeline schedule with 4 workers, with startup and steady states indicated. In this example, the backward pass takes twice as long as the forward pass.

2.2.2 DNN Model and Hardware Diversity

DNN models are diverse, with convolutional layers, LSTMs [171], attention layers [164], and fully-connected layers commonly used. These different types of models exhibit vastly different performance characteristics with different parallelization strategies, making the optimal parallelization strategy highly model-dependent.

Picking an optimal parallelization scheme is challenging because the efficacy of such a scheme depends on the characteristics of the target deployment hardware as well; GPUs, ASICs, and FPGAs have very different compute capabilities. Moreover, interconnects linking these accelerators have different topologies and capacities; cloud servers are linked by 10Gbps to 100Gbps networks, accelerators within servers might be connected over shared PCIe trees (10 to 15GBps), and specialized expensive servers, such as the DGX-1 [20], use `NVLink` with point-to-point 30GBps bandwidth capabilities. This diversity in models and deployments makes it extremely hard to manually come up with an optimal parallelization strategy. PipeDream automates this process, as we discuss in §2.4.1.

2.3 Pipeline Parallelism as a Distributed Training Paradigm

Pipeline parallelism is a parallelization strategy that combines pipelining with inter-layer model parallelism. Pipeline-parallel computation involves partitioning the layers of a DNN model into multiple *stages*, where each stage consists of a *consecutive* set of layers in the model. Other assignments of layers to compute resources are possible; we defer discussion of such interleaved assignments (where each worker gets a strided set of operators in the model) to Chapter 4. Each stage is mapped to a separate GPU that performs the forward pass (and backward pass) for all layers in that stage.[2]

In the simplest case, only one input is active in the system, as in traditional model-parallel training (Figure 2.2); in this setup, at most one GPU is active at a time. Ideally, we would like all GPUs to be active. With this in mind, we inject multiple inputs into the pipeline one after the

[2]We use GPUs as a concrete instance of accelerators and use the terms "GPU", "device", and "worker" interchangeably.

other. On completing its forward pass for an input, each stage asynchronously sends the output activations to the next stage, while simultaneously starting to process another input. The last stage starts the backward pass on an input immediately after the forward pass completes. On completing its backward pass, each stage asynchronously sends the gradient to the previous stage while starting computation for the next input (Figure 2.4).

Pipeline parallelism (PP) can outperform data parallelism (DP) for two reasons.

Pipelining communicates less. PP often can communicate far less than DP. Instead of having to aggregate gradients for all parameters and send the result to all workers, as is done in data-parallel approaches (using either collective communication or a parameter server), each worker in a PP execution has to communicate only subsets of the gradients and output activations, to only a single other worker. For certain models, these intermediate activations and input gradients are much smaller than the full weight gradients. This can result in large reductions in communication for *some* models (e.g., >85% reduction for VGG-16, AWD LM).

Pipelining overlaps computation and communication. Asynchronous communication of forward activations and backward gradients across stages results in significant overlap of communication with the computation of a subsequent input. This computation and communication are completely independent with *no* dependency edges, since they operate on different inputs, leading to easier parallelization.

However, to realize the opportunity of pipeline parallelism, we must overcome three challenges.

2.3.1 Challenge 1: Work Partitioning

With pipeline parallelism, model training can be treated as a computation pipeline, with each worker executing a subset of the model as a stage. Like with any pipeline, the steady state throughput of the resulting pipeline is the throughput of the slowest stage. Having each stage process inputs at vastly different throughputs can lead to bubbles in the pipeline, starving faster stages of inputs to work on and resulting in resource under-utilization. Excessive communication between workers can also lower the throughput of the training pipeline. Moreover, the allocation of stages to workers needs to be model- and hardware-aware to be effective, and there may be cases where no simple partitioning across the GPUs achieves both limited communication and perfect load balance.

2.3.2 Challenge 2: Work Scheduling

Unlike traditional uni-directional pipelines, training a DNN model with pipelining involves a bi-directional pipeline, where an input proceeds through the computation pipeline first forward and

then backward (this is fundamental to the most natural and widely used form of backpropagation: the backward pass is needed to compute weight gradients that are then used to update the model's parameters). This is shown in Figure 1.3. Each active input in the pipeline may be in a different stage, either in the forward pass or backward pass. As a result, at any point in time, each worker in the system needs to make decisions on the following:

1. Should it perform a forward pass for an input, pushing the subsequent output activation to downstream workers?
2. Should it perform a backward pass for a (different) input, pushing the subsequent input gradient (gradient of the loss with respect to the input tensor to the stage) to upstream workers?
3. How should inputs be routed through replicated stages?

These decisions need to be made in such a way that we can still ensure that the final model obtained is high quality, convergence rate (or statistical efficiency, the number of iterations needed to train the model up to a particular accuracy target) is not hampered, and memory footprint is low.

2.3.3 Challenge 3: Effective Learning

In a naïvely pipelined system, each stage's forward pass for an input is performed using one version of parameters and its backward pass is performed using a different version of parameters. Figure 2.4 illustrates this using a partitioning with four workers and no stage replication. In stage 1, the forward pass for input 5 is performed after the updates from input 1 are applied, whereas the backward pass for input 5 is performed after updates from inputs 2, 3, and 4 are applied. As a result, in the backward pass for input 5 on stage 1, the gradient is computed using a different set of weights than the ones used in the corresponding forward pass; this discrepancy in weight versions results in invalid gradients and can prevent or slow down model convergence.

2.4 PipeDream System Design

In this section, we discuss PipeDream's specific solutions to the challenges presented in the previous section. However, as mentioned before, other strategies exist for pipeline parallelism, leading to other tradeoffs. We discuss a few other strategies in Chapters 3 and 4. In discussing PipeDream's specific solutions, we will refer to Figure 2.5, which shows PipeDream's high-level workflow.

PipeDream assumes that each input is composed of a fixed pre-configured number of samples (the microbatch size). PipeDream, as described in this chapter, does not perform additional gradient accumulation within the pipeline, which means the batch size and microbatch size within the pipeline are the same. Chapter 3 shows an alternative approach where this is no longer true.

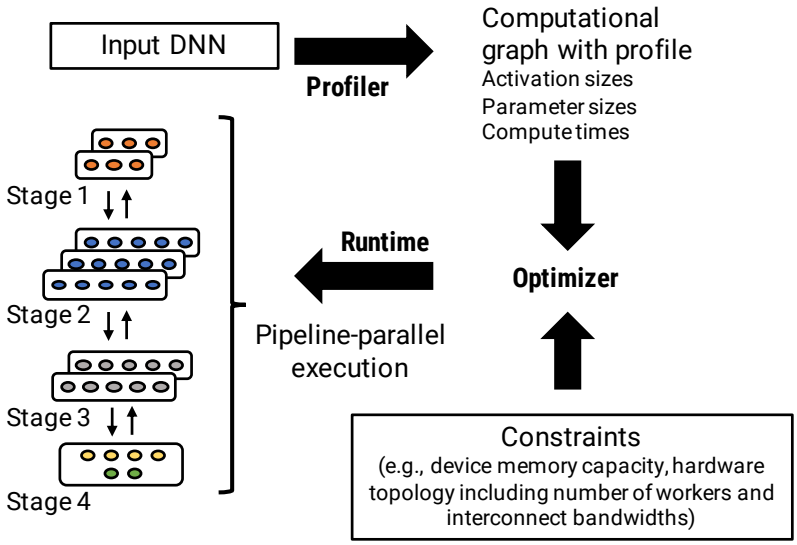


Figure 2.5: PipeDream's automated mechanism to partition DNN layers into stages. PipeDream first profiles the input DNN, to get estimates for each layer's compute time and output size. Using these estimates, PipeDream's optimizer partitions layers across available machines, which is then executed by PipeDream's runtime.

2.4.1 Profiling and Partitioning

PipeDream's optimizer outputs a balanced pipeline. Its algorithm partitions DNN layers into stages such that each stage completes at roughly the same rate, while trying to minimize communication across workers in a topology-aware way (for example, large outputs should be sent over higher bandwidth links if possible). To further improve load balancing, PipeDream goes beyond straight pipelines, allowing a stage to be replicated (i.e., data parallelism is used on the stage). This partitioning problem is equivalent to minimizing the time taken by the slowest stage of the pipeline, and has the *optimal sub-problem property*: a pipeline that maximizes throughput given a worker count is composed of sub-pipelines that maximize throughput for smaller worker counts. Consequently, we use dynamic programming to find the optimal solution.

PipeDream exploits the fact that DNN training shows little variance in computation time across inputs. PipeDream records the computation time taken by the forward and backward pass, the size of the layer outputs, and the size of the associated parameters for each layer as part of an initial profiling step; this profile is used as the input to the optimizer's partitioning algorithm (Figure 2.5). The partitioning algorithm also takes into account other constraints such as hardware topology and bandwidth, number of workers, and memory capacity of the compute devices.

Figure 2.6: An example 2-level hardware topology. Solid green boxes represent GPUs. Each server (dashed yellow boxes) has 4 GPUs connected internally by links of bandwidth B_1; each server is connected by links of bandwidth B_2. In real systems, $B_1 > B_2$. Figure best seen in color.

Profiler

PipeDream records three quantities for each layer l, using a short (few minutes) profiling run of 1000 iterations or so on a single GPU of the target type:

1. T_l, the total computation time across forward and backward passes for layer l on the GPU for a single input (we assume that the microbatch size is the same across the full computation).
2. a_l, the size of the output activations of layer l in bytes.
3. w_l, the size of weight parameters for layer l in bytes.

PipeDream estimates the communication time by dividing the amount of data that needs to be transferred by the network bandwidth of the communication link. In data-parallel configurations with m workers, each worker sends $\left(\frac{m-1}{m} \cdot |w_l|\right)$ bytes to other workers, and receives the same amount; this is used to estimate the time for weight synchronization for layer l when using data parallelism with m workers.

Partitioning Algorithm

Our partitioning algorithm takes the output of the profiling step, and computes:

1. A partitioning of layers into stages.
2. The replication factor (number of workers) for each stage.
3. The optimal number of in-flight inputs to keep the training pipeline busy.

PipeDream's optimizer assumes that the machine topology is hierarchical and can be organized into levels, as shown in Figure 2.6. Bandwidths within a level are the same, while bandwidths across levels are different. We assume that level k is comprised of m_k components of level $(k-1)$, connected by links of bandwidth B_k. In Figure 2.6, m_2 is 2 and m_1 is 4. In addition, we define m_0 to be 1; m_0 is the number of compute devices within the first level (solid green boxes in Figure 2.6).

PipeDream's optimizer solves dynamic programming problems progressively from the lowest to the highest level. Intuitively, this process finds the optimal partitioning within a server and then uses these partitions to split a model optimally across servers.

Notation. Let $A^k(i \rightarrow j, m)$ denote the time taken by the slowest stage in the optimal pipeline between layers i and j using m workers at level k. The goal of our algorithm is to find $A^L(0 \rightarrow N, m_L)$, and the corresponding partitioning, where L is the highest level and N is the total number of layers in the model.

Let $T^k(i \rightarrow j, m)$ denote the total time taken by a single stage spanning layers i through j for both forward and backward passes, replicated over m workers using bandwidth B_k.

Formulation. For all k from 1 to L,

$$T^k(i \rightarrow j, m) = \frac{1}{m} \max \begin{cases} A^{k-1}(i \rightarrow j, m_{k-1}), \\ \frac{2(m-1)\sum_{l=i}^{j} |w_l|}{B_k}. \end{cases}$$

where the first term inside the $\max$ is the total computation time for all the layers in the stage using level $k-1$ as the computation substrate, and the second term is the time for data-parallel communication among all layers in the stage. The result of the $\max$ expression above gives the effective time spent processing m inputs while performing compute and communication concurrently; thus, the effective time spent processing a single input is this term divided by m.

The optimal pipeline can now be broken into an optimal sub-pipeline consisting of layers from 1 through s with $m - m'$ workers followed by a single stage with layers $s+1$ through j replicated over m' workers. Then, using the optimal sub-problem property, we have:

$$A^k(i \rightarrow j, m) = \min_{i \leq s < j} \min_{1 \leq m' < m} \max \begin{cases} A^k(i \rightarrow s, m - m'), \\ 2a_s / B_k, \\ T^k(s+1 \rightarrow j, m'). \end{cases}$$

where the first term inside the max is the time taken by the slowest stage of the optimal sub-pipeline between layers i and s with $m - m'$ workers, the second term is the time taken to communicate the activations and gradients of size a_s between layers s and $s+1$, and the third term is the time taken by the single stage containing layers $s+1$ to j in a data-parallel configuration of m' workers.

When solving for level k, we use $A^{k-1}(i \rightarrow j, m_{k-1})$, which is the optimal total computation time for layers i through j using all workers available in a single component at level $(k-1)$ (in the expression $T^k(i \rightarrow j, m)$). In Figure 2.6, this would represent determining how best to partition intermediate layers of the model using all workers in a yellow server.

Initialization. Level 0 uses the profiled computation times: $A^0(i \rightarrow j, m_0) = \sum_{l=i}^{j} T_l$. For $k > 0$, optimal compute times with all compute devices in the previous level are used: $A^k(i \rightarrow j, 1) = A^{k-1}(i \rightarrow j, m_{k-1})$.

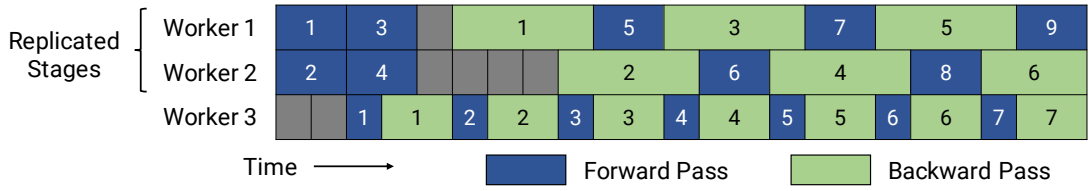


Figure 2.7: An example PipeDream pipeline with 3 workers and 2 stages. We assume that forward and backward passes in the first stage take two and four time units, while forward and backward passes in the second stage take one and two time units. The first stage in this pipeline is replicated twice so that each stage sustains roughly the same throughput. Here, we assume that the backward pass takes twice as long as the forward passes, but this is not a requirement of our approach.

Runtime Analysis. For a given level k, the total number of sub-problems is $O(N^2 m_k)$. Time complexity per sub-problem is $O(N m_k)$, leading to a total time complexity of $O(N^3 m_k^2)$ for level k. Total time complexity is $\sum_{k=1}^{L} O(N^3 m_k^2)$. In our experiments, the running time is under 8 seconds.

2.4.2 1F1B(-RR) Schedule

In the startup phase, the input stage admits enough inputs to keep the pipeline full in steady state. Based on the partitioning generated by our algorithm, the optimal number of inputs admitted *per input stage replica* to keep the pipeline full in steady state is given by:

$$\texttt{NUM\_OPT\_ACTIVE\_MINIBATCHES}\ (NOAM) = \lceil \text{(\# workers)} \ / \ \text{(\# of replicas in the input stage)} \rceil.$$

Once in steady state, each stage *alternates* between performing its forward pass for an input and its backward pass for an earlier input. We call this the *one-forward-one-backward* (1F1B) schedule. 1F1B ensures that every GPU is occupied with an input in a balanced pipeline, with each stage producing outputs in aggregate at roughly the same rate. It also ensures backward passes from inputs are applied at regular intervals of time. As we show later in this dissertation, this schedule helps keep the memory footprint low by keeping the number of in-flight inputs as small as possible while still ensuring that every worker in the pipeline is active (thus minimizing pipeline stalls).

Figure 2.4 shows the corresponding compute timeline for a pipeline with 4 stages. The NOAM for this configuration is 4. In the startup phase, the input stage admits exactly four inputs that propagate their way to the output stage. As soon as the output stage completes its forward pass for the first input, it performs its backward pass for the same input, and then starts alternating between forward and backward passes for subsequent inputs. As the first input propagates up the pipeline to earlier stages (to complete its backward pass), every stage starts alternating between forward and backward passes for different inputs. As shown in the figure, every worker is performing either a forward or backward pass for some input in steady state.

When a stage is run in a data-parallel configuration (replicated across multiple GPUs), we use

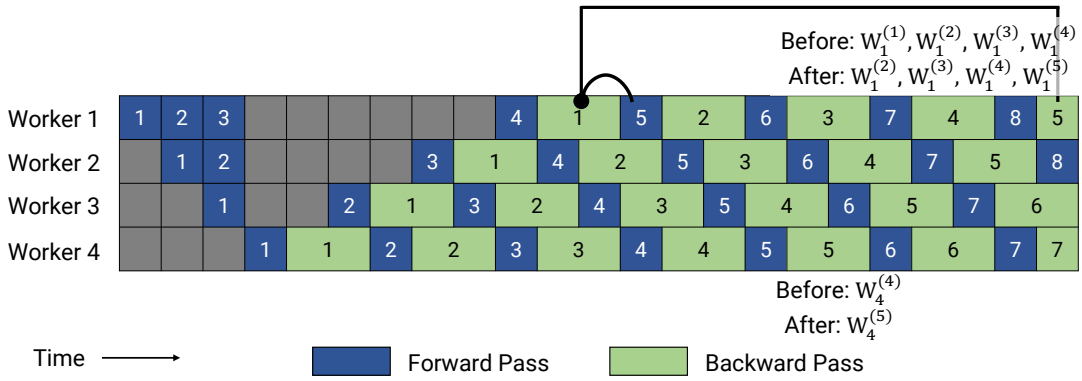


Figure 2.8: Weight stashing as input 5 flows across stages. Arrows point to weight versions used for forward and backward passes for input 5 at the first stage. For simplicity, we assume that the forward pass takes one time unit, and the backward pass takes two time units on each worker.

deterministic round-robin load balancing based on an input identifier to spread work across the replicas. Such deterministic load-balancing ensures that each input is routed to the same worker for both the forward and backward passes of the stage, which is important since parameters and intermediate outputs from the forward pass are needed for the backward pass. This mechanism, which we call *one-forward-one-backward-round-robin* (1F1B-RR), is a *static* policy that is executed without expensive distributed coordination. Figure 2.7 shows this mechanism in action for a simple 2-1 configuration, with the first stage replicated twice, and the second stage un-replicated. In the first stage, all inputs with even input IDs are processed by worker 1, while inputs with odd input IDs are processed by worker 2. Worker 3 in the second stage processes all inputs. All workers perform a forward pass followed by a backward pass on a different input.

For 1F1B-RR to be effective, it is *not necessary* for the forward pass to take as long as the backward pass. In fact, we observe that the backward pass is always larger than the forward pass in practice. 1F1B-RR remains an effective scheduling mechanism, as highlighted in Figure 2.4.[3]

2.4.3 Weight Stashing and Vertical Sync

In this chapter, we present two techniques (weight stashing and vertical sync) that ensure that numerically-correct gradients are computed. However, these are not the only solutions, and we discuss other solutions in Chapters 3 and 4, along with the corresponding tradeoffs.

Weight Stashing. PipeDream uses a technique called *weight stashing* to avoid a fundamental mismatch between the version of weights used in the forward and backward pass. Weight stashing maintains multiple versions of the weights, one for each active input. Each stage processes an input

[3] 1F1B-RR produces a full steady state pipeline even for cases where the ratio of backward- to forward-pass time is not an integer (e.g., 3 to 2).

using the latest version of weights available in the forward pass. After completing the forward pass, PipeDream stores the weights used for that input. The *same weight version* is then used to compute the weight update and upstream weight gradient in the input's backward pass.

Weight stashing ensures that *within a stage*, the same version of model parameters are used for the forward and backward pass of a given input. For example, in Figure 2.8, input 5 uses parameter updates from input 1 on machine 1 and from 2 on machine 2. Weight stashing does not guarantee the consistency of parameter versions used for a given input *across* stages.

Vertical Sync. Vertical sync is an optional technique in PipeDream that eliminates the potential inconsistency *across stages*. For example, in Figure 2.4, input 5 uses parameters updated by input 1 on all workers for both its forward and backward passes when using vertical sync. Each input t that enters the pipeline is associated with the latest weight version $W^{(t-x)}$ seen at the input stage. This information is propagated along with the activations and gradients as the input t flows through the pipeline in the forward direction. Across all stages, the forward pass for t uses the stashed weights $W^{(t-x)}$ as opposed to the latest weight update. After performing the backward pass for t (using stashed weights $W^{(t-x)}$), each stage independently applies weight updates to create the latest weights ($W^{(t)}$), and can then delete $W^{(t-x)}$. This coordination across stages is asynchronous.

The semantics of vertical sync are different from GPipe (and data parallelism). In particular, gradients are not aggregated over all in-flight inputs (called microbatches in GPipe) in the system – vertical sync merely ensures that the same weight versions are used to compute gradients across different workers (but the weight versions to which gradients are applied are different from those used to compute the gradients). The batch size with weight stashing and vertical sync is thus just the microbatch size (the number of samples in an input); the batch size with GPipe is $b \cdot m$, where m is the number of inputs injected into the pipeline.

Staleness. We can now formalize the degree of staleness of weight updates for each of these techniques. For this discussion, we assume a straight pipeline (i.e., no stage replication) with the model split into n stages; the weights in each stage are represented as W_1, W_2, and so on. In addition, we denote $W_l^{(t)}$ as the weights W_l after t inputs. We assume that the number of pipeline stages is p.

Now, after every input batch, we compute $\nabla f(W_1, W_2, \ldots, W_p)$, which is the gradient averaged over all samples in the batch. Vanilla batch SGD (f is the loss function, ν is the learning rate) has the following gradient update:

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W_1^{(t)}, W_2^{(t)}, \ldots, W_p^{(t)})$$

With weight stashing, gradients in stage 1 are computed with weights that are $p-1$ steps delayed, gradients for stage 2 are computed with weights that are $p-2$ steps delayed, etc. Mathematically,

this means the weight update looks like:

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W_1^{(t-p+1)}, W_2^{(t-p+2)}, \ldots, W_p^{(t)})$$

Without weight stashing, the weight update is not a valid gradient of the loss function f for any vector $W_1, \ldots, W_p$.

Adding vertical sync alters the weight update to:

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W_1^{(t-p+1)}, W_2^{(t-p+1)}, \ldots, W_p^{(t-p+1)})$$

This is semantically similar to data parallelism with BSP synchronization on p workers with the same per-worker batch size and staleness (but gradients averaged over a p times smaller batch).

Memory Overhead. Pipelining does not significantly increase per-worker memory usage relative to data parallelism, even with weight stashing. Consider a straight pipeline (no data-parallel stages), where a model is divided across p workers, with each worker holding $1/p$ of the weights. With non-pipelined model-parallel training, each worker would need $1/p$ of the memory compared to data parallel training. Admitting p inputs into the pipeline, as PipeDream does, increases this by at most a factor of p, because a version of <weights, activations> is needed for each in-flight input. Thus, PipeDream's peak per-worker memory usage is on par with data parallelism.

PipeDream's memory footprint can be further reduced by using existing techniques: efficient encoding or compression of intermediate data [89], gradient aggregation where weight gradients are accumulated into a single buffer at a stage for m inputs before performing a weight update, and trading computation time for activation-stash memory by discarding them in the forward pass and recomputing them as needed during the backward pass [53]. We discuss the usage of such techniques to train models with large training footprints in the next chapter.

PipeDream's default semantics exclude vertical sync as it requires more metadata to be stored at every stage in the pipeline. Our evaluation demonstrates the effectiveness of weight stashing across models, datasets, and hardware configurations.

2.4.4 Implementation

The interface to PipeDream is implemented as a standalone Python library of ∼3000 LOC that manages device memory, schedules work, and handles communication. PipeDream uses PyTorch [134] for auto-differentiation and to execute operators; however, PipeDream is extensible and can work with other ML frameworks such as Tensorflow [36], MXNet [51], and CNTK [146]. As a proof of concept, we also integrated PipeDream with Caffe [93].

PipeDream first profiles the model on a single GPU with a subset of inputs from the training dataset (Figure 2.5). It then runs the optimization algorithm described in §2.3.1 to partition the DNN model into stages, with some stages possibly replicated.

PipeDream's optimizer returns an annotated operator graph, with each model layer mapped to a stage ID. PipeDream performs a BFS traversal of this graph and generates code for each stage as a separate `torch.nn.Module`, ordering operators in each stage to make sure their input-output dependencies from the original PyTorch model graph are respected. The PipeDream runtime then assigns each stage (including replicas for replicated stages) to a single worker.

Parameter State. PipeDream maintains all parameters associated with the layers assigned to the stage directly in GPU memory. PipeDream applies updates to the most recent parameter version when the weight update becomes available if the stage is not replicated. The weight updates are synchronized across replicas prior to being applied if the stage is replicated. When a newer version of the parameters becomes available, the prior version is *not* immediately discarded. Parameters are discarded only once a backward pass that uses fresher parameters is performed.

Intermediate State. Each stage's input and output data is assigned a unique blob ID. Upon receiving intermediate data from the prior stage (or from disk in the case of the input stage), PipeDream copies the intermediate data to GPU memory and places a pointer to the associated buffer in a work queue. Intermediate data from the forward pass is not discarded until the associated batch completes that stage's backward pass. Intermediate data from the backward pass is freed as soon as the worker finishes using it, and if necessary, after it is sent to the next stage.

Stage Replication. PipeDream uses PyTorch's `DistributedDataParallel` library [24] to synchronize parameters for layers of data-parallel stages. Using wait-free back propagation, weight gradients are communicated to servers as soon as they are computed, rather than waiting for computation to finish for all layers. Since we support replication of individual stages, data-parallel training is effectively a special case in our framework – we represent this as a single stage that contains all the layers of the DNN model, and replicate the stage across all available GPUs. We use the NCCL communication backend [18] for data-parallel baselines as we find it to be faster than Gloo [8] for the large tensors exchanged in DP. PipeDream uses Gloo for all inter-GPU communication when performing pipeline-parallel training.

Checkpointing. PipeDream supports periodic checkpointing of model parameters for fault tolerance, with default checkpoints made across stages at the end of every epoch. Checkpoints don't require expensive global coordination. Each stage dumps its model parameters locally when it performs the backward pass for the last batch in an epoch. Restarting a run due to failures entails starting from the last successfully created checkpoint for all stages.

Cluster name	Server SKU	GPUs per server	Interconnects Intra-, Inter-server
Cluster-A	Azure NC24 v3	4x V100	PCIe, 10 Gbps
Cluster-B	AWS p3.16xlarge	8x V100	NVLink, 25 Gbps
Cluster-C	Private Cluster	1 Titan X	N/A, 40 Gbps

Table 2.1: Characteristics of servers used in experiments.

2.5 Evaluation

This section evaluates the effectiveness of PipeDream for seven different DNNs on three different clusters. The results of our experiments support a number of important findings:

1. PipeDream achieves significant speedups in time-to-target-accuracy across a wide range of different learning tasks on different hardware deployments.
2. PipeDream is more efficient than other recently proposed pipeline parallelism approaches.
3. PipeDream greatly reduces overheads of communication and does not significantly increase memory footprint compared to data-parallel training.
4. Combining pipelining, model parallelism, and data parallelism outperforms model-, data-, or hybrid-parallelism in isolation.

2.5.1 Experimental Setup

Tasks and Datasets. We use four tasks and four datasets in our experiments:

1. Image Classification, using the ImageNet-1K (ILSVRC12) [144] dataset.
2. Translation, using the WMT16 English to German dataset for training, and the `newstest2014` dataset for validation.
3. Language Modeling, using the Penn Treebank (PTB) [120] dataset.
4. Video Captioning (S2VT), using the Microsoft Video description corpus (MSVD) [49].

Clusters. We use three different clusters in our experiments, summarized in Table 2.1. *Cluster-A* has servers with 4 NVIDIA V100 GPUs each (Microsoft Azure NCv3 instances), with 16 GB of GPU device memory, and a 10 Gbps Ethernet interface. *Cluster-B* has servers with 8 V100s each (AWS EC2 p3.16xlarge instances), with 16 GB of GPU device memory, and a 25 Gbps Ethernet interface. GPUs within servers are connected via a shared PCIe interconnect on Cluster-A, and via point-to-point `NVLink` on Cluster-B. All servers run 64-bit Ubuntu 16.04 with CUDA toolkit 10.0 and cuDNN

v7.4. *Cluster-C* has servers with 1 NVIDIA Titan X GPU and 12 GB of GPU device memory, connected via 40 Gbps Ethernet. Unless otherwise stated, all our experiments are run on multi-GPU servers (Cluster-A and Cluster-B).

Models. We use seven different DNN models in our experiments across the four applications: 1) VGG-16 [154], 2) ResNet-50 [84], 3) AlexNet [102], 4) Google Neural server Translation (GNMT) with 8 LSTM layers [171], 5) GNMT with 16 LSTM layers, 6) AWD Language Model (LM) [118], and 7) the S2VT [167] sequence-to-sequence model for video transcription.

Batch Sizes and Training Methodology. We use the largest per-GPU batch that fits in one GPU's memory – anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable throughput on a single device. Unless otherwise stated, we report per-GPU batch sizes (G); for data-parallel runs with n workers, the global batch size is $n \cdot G$. The global batch sizes we use are consistent with those used by the ML community and reported in the literature for these models. We use a per-GPU batch size of 64 per GPU for VGG-16, 256 for AlexNet, 128 for ResNet-50 (e.g., BS $= 1024$ for 8 GPUs), 64 for GNMT, 80 for S2VT, and batch size of 80 for LM. We train the VGG-16, ResNet-50, Language Modeling, and S2VT models using SGD with an initial learning rate of 0.01, 0.1, 30.0, and 0.01 respectively. For GNMT, we use the Adam optimizer [98] with an initial learning rate of 0.0003. We use full (`fp32`) precision.

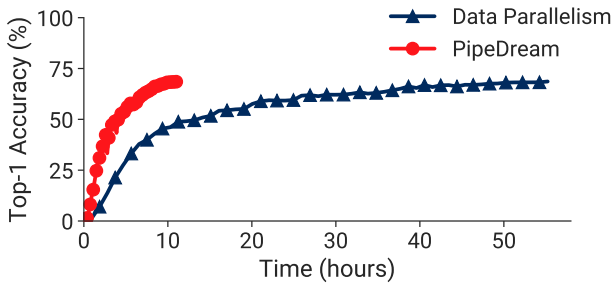
For all experiments (other than AlexNet), we measure the time taken to train to a target validation accuracy: top-1 accuracy of 68% for VGG-16 [26], top-1 accuracy of 75.9% for ResNet-50, BLEU score of 21.8 for GNMT, a validation perplexity of 98 for LM, and a METEOR [65] score of 0.294 for S2VT. Guided by prior work, we adjust the learning rate during training to converge to the desired result faster [156, 98] and utilize learning rate warm-up for large global batch sizes [76]. We use the same learning rate schedules for PipeDream and data-parallel training. For AlexNet, we use synthetic data (otherwise, data loading is the bottleneck) and measure throughput.

Task	Model	Dataset	Accuracy Threshold	# Servers × # GPUs (Cluster)	PipeDream Config	Speedup over DP	
						Epoch time	TTA
Image Classification	VGG-16 [154]	ImageNet [144]	68% top-1	4x4 (A)	15-1	5.3×	5.3×
				2x8 (B)	15-1	3×	2.5×
	ResNet-50 [84]	ImageNet [144]	75.9% top-1	4x4 (A)	16	1×	1×
				2x8 (B)	16	1×	1×
	AlexNet [102]	Synthetic Data	N/A	4x4 (A)	15-1	5×	N/A
				2x8 (B)	15-1	2×	N/A
Translation	GNMT-16 [171]	WMT16 EN-De	21.8 BLEU	1x4 (A)	Straight	1.5×	2.2×
				4x4 (A)	Straight	2.3×	2.9×
				2x8 (B)	Straight	3.1×	3.1×
	GNMT-8 [171]	WMT16 EN-De	21.8 BLEU	1x4 (A)	Straight	1.5×	1.5×
				3x4 (A)	Straight	3×	3×
				2x8 (B)	16	1×	1×
Language Model	AWD LM [118]	Penn Treebank [120]	98 perplexity	1x4 (A)	Straight	4.3×	4.3×
Video Captioning	S2VT [167]	MSVD [49]	0.294 METEOR	4x1 (C)	2-1-1	3×	3×

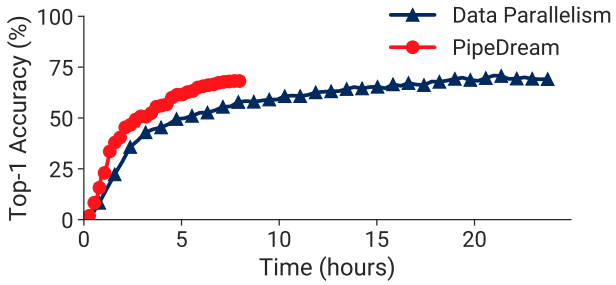
Table 2.2: Summary of results comparing PipeDream with data parallelism (DP) when training models to advertised final accuracy. A PipeDream config of "2-1-1" means the model is split into three stages with the first stage replicated across 2 workers, and a "straight" configuration is a pipeline with no replicated stages—e.g., "1-1-1-1" on 4 workers. Batch sizes used to train these models are reported in §2.5.1.

2.5.2 Comparison to Data Parallelism

Table 2.2 summarizes results comparing PipeDream with data-parallel training (DP). The table shows PipeDream's auto-generated configurations and their speedups in training time-to-accuracy over corresponding data-parallel training configurations.[4]



(a) Cluster-A.

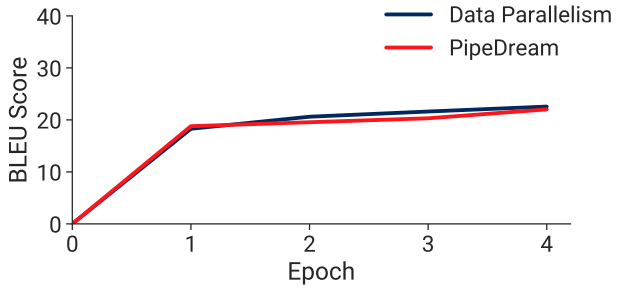


(b) Cluster-B.

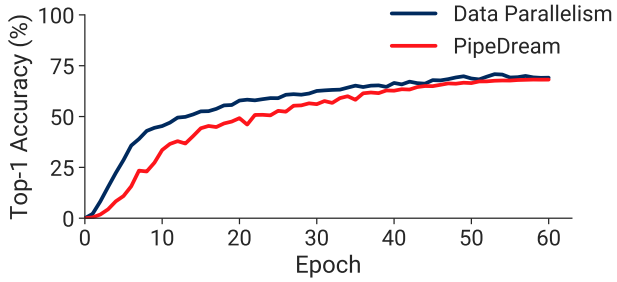
Figure 2.9: Accuracy vs. time for VGG-16 using 16 GPUs. Each circle or triangle represents two epochs of training.

PipeDream Configurations. As described in §2.3.1, given a DNN model and a set of servers with GPUs, PipeDream's optimizer automatically chooses to partition the model into stages, while also deciding the optimal replication factor for each stage. Although most prior research has focused on improving data-parallel training, our results indicate that the best configurations for many models is not data parallelism, despite the use of many important optimizations such as wait-free back propagation. In all but one of our experiments, the best PipeDream configuration combines model parallelism, pipelining, and sometimes data parallelism; each of these configurations outperforms purely data-parallel training, highlighting the importance of combining pipeline parallelism with data parallelism. PipeDream's optimizer recommends data parallelism for ResNet-50 because its weight representations are small and its outputs are large. PipeDream's optimizer, besides determining the optimal configuration, also automatically decides where to partition the DNN training

[4]A configuration indicates how layers are partitioned into stages amongst workers.



(a) GNMT-16



(b) VGG-16

Figure 2.10: Accuracy vs. epoch using 16 GPUs on Cluster-B.

graph; these partitioning decisions are not shown in Table 2.2.

Image Classification. We compare the time-to-accuracies for PipeDream and data parallelism (DP) on the VGG-16 model using 4 servers in Cluster-A (4x4 (A) in Table 2.2). PipeDream reaches target accuracy 5.3× faster than DP on a single server due to a reduction in inter-server communication. Figure 2.9 (a) shows this comparison as the DNN is trained over time. In the 4-server configuration, PipeDream's optimizer (§2.3.1) recommends a 15-1 configuration – in this case, VGG-16's convolutional layers are replicated, while the large fully connected layers are not, reducing communication overhead. Moreover, pipelining across the two stages helps keep all workers busy.

Compared to Cluster-A, which has 4 GPUs per server connected via PCIe, Cluster-B has 8 GPUs per server connected over faster NVLink interconnects. On 2 servers on Cluster-B (16 GPUs total), PipeDream reaches target accuracy 3× faster than DP when training VGG-16. Due to the faster interconnects on Cluster-B, both PipeDream and DP reach target accuracy faster than on Cluster-A (see Figure 2.9).

For training ResNet-50 on Cluster-A, PipeDream's partitioning algorithm recommends data parallelism as the optimal configuration (no pipelining or model parallelism). Later, in §2.5.5, we show the reason for this recommendation: configurations that do not use data parallelism incur

Model	Scale (# V100s)	Cluster-B / official MLPerf v0.5
GNMT-8	256	1.9×
SSD	64	3.3×
Mask R-CNN	64	2.3×

Table 2.3: Increase in per-epoch times for data-parallel training when moving from dedicated clusters used in official MLPerf v0.5 entries to public clouds like Cluster-B. The *same* code is used for both sets of runs.

higher communication overheads than data parallelism for ResNet-50, since ResNet-50 is composed of convolutional layers which have compact weight representations but large output activations. For AlexNet, we compare throughput of PipeDream on Cluster-A and Cluster-B. On Cluster-A, PipeDream achieves a time-per-epoch speedup of 4.9× with 4 servers. On Cluster-B, PipeDream achieves a speedup of 2× when using 16 GPUs.

Translation. We show results for the GNMT model with 8 LSTM layers (GNMT-8) and 16 LSTM layers (GNMT-16) in Table 2.2). Using 1 server on Cluster-A, PipeDream reaches target accuracy ∼1.5× faster than DP for GNMT-8 and GNMT-16. When using 4 servers (16 GPUs) on Cluster-A, PipeDream reaches target accuracy 2.9× (GNMT-8) and 3× (GNMT-16) faster than DP. We show in §2.5.5 that PipeDream significantly reduces communication compared to DP, thus reducing its time to target accuracy.

On 2 servers (16 GPUs) of Cluster-B, PipeDream reaches target accuracy 3.1× faster than DP for GNMT-16, choosing a "straight" configuration (no stage replication). For GNMT-8, PipeDream falls back to data parallelism, since the smaller model has lower communication overhead on servers with fast `NVLink` interconnects between GPUs on the same server, and GNMT-8 does not have enough layers for a 16-deep straight pipeline.

Language Modeling. This model is made up of six LSTM layers that contain a large number of model parameters (0.41GB), making data-parallel training inefficient. Using a single server on Cluster-A, PipeDream reaches target accuracy 4.3× faster than DP. PipeDream chooses a "straight" configuration that reduces communication by 88% compared to DP.

Video Captioning. PipeDream chooses to use a 2-1-1 configuration for the S2VT on Cluster-C, reducing communication by 85% compared to DP, which in turn allows it to reach target accuracy 3× faster than DP.

Comparison to MLPerf v0.5. For ResNet-50 and GNMT-8, we observe that our data-parallel baseline on a single server with 8 GPUs in Cluster-B is comparable to the MLPerf v0.5 entry that uses a

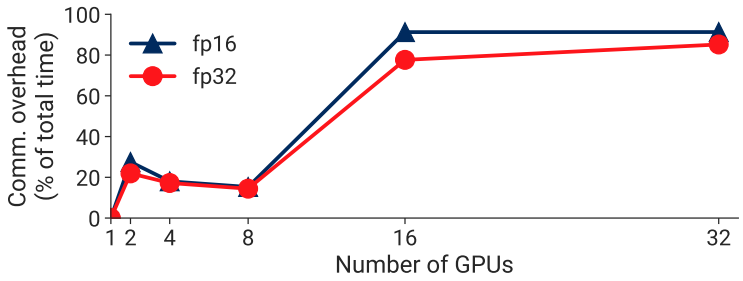


Figure 2.11: Communication overhead of data-parallel training using different server instances using PyTorch 1.1 and NCCL [18] for a GNMT-8 model with `fp16` and `fp32` precision.

similar hardware configuration. However, we observe that per-epoch times on public cloud servers are slower than official MLPerf v0.5 entries for multi-server DP deployments, since slower communication links on public cloud servers (compared to dedicated clusters used in the MLPerf entries) make `all_reduce` communication slower. We cannot measure this difference in time-to-accuracy at the scales used by the MLPerf entries as it is cost prohibitive, but Table 2.3 compares the advertised training throughput of official MLPerf v0.5 [16] entries with data-parallel runs on `p3.16xlarge` instances using the *same code*. Coleman et al. observed similar results [57], both for official DAWNBench and MLPerf entries.

Furthermore, with 8 GPUs, for GNMT-8, while full precision is slower than the entry using mixed precision, we use a `fp32` baseline to be consistent with the rest of the evaluation in this chapter. Figure 2.11 shows that communication overheads for data parallelism with mixed precision are higher than with full precision, and thus the speedups we highlight with pipeline parallelism should carry over (or improve) with mixed precision training.

Comparison to DP with large batches. Recent work has demonstrated that using large batches is effective for training ResNet-50 and AlexNet models, especially when combined with Layer-wise Adaptive Rate Scaling (LARS). [76, 177, 92]. LARS uses different learning rates for each layer based on the ratio of the weight norm to the gradient norm. Large batches decrease the frequency of communication, reducing the communication overhead for data parallelism. Figure 2.12 shows 8-server results for data-parallel training of VGG-16 using LARS and large batches on Cluster-C. Batches of 1024 had the fastest time-to-target-accuracy, while batches of 4096 and 8192 failed to reach target accuracy, highlighting the lack of generality of such approaches. PipeDream still reaches target accuracy over 2.4× faster than the fastest data-parallel option (1024 with LARS).

Comparison to Asynchronous Parallelism (ASP). ASP can reduce communication overhead in data-parallel training. Unlike BSP, which synchronizes parameters after every batch, ASP has no synchronization overheads, and workers use the most recent parameter data *available*. The result

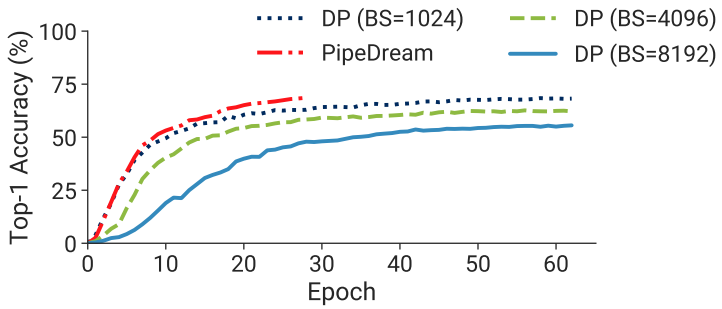


Figure 2.12: Statistical efficiency (accuracy vs. epoch) using LARS (VGG-16, 8 GPUs).

is often poor statistical efficiency. For example, when training VGG-16 on 4 Cluster-B servers, ASP takes 7.4× longer than PipeDream to reach a 48% accuracy (when we terminate ASP for taking too long to converge), even though ASP has minimal communication delays. Similar results have been shown by Chen et al. [50].

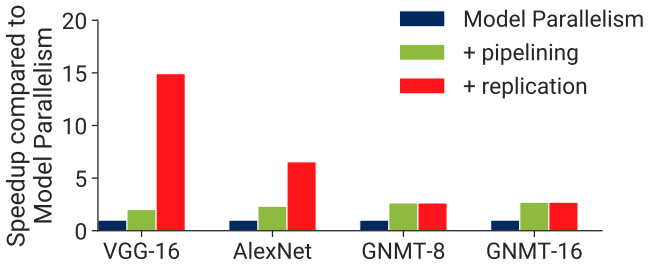
Statistical Efficiency. Figure 2.10 shows accuracy vs. epoch for VGG-16 and GNMT-16 on Cluster-B. We consistently observe that PipeDream reaches target accuracy in a similar number of epochs as DP (as can be seen by the fact that TTA and epoch time speedups are the same for many rows in Table 2.2). This highlights the fact that PipeDream's weight stashing mechanism is able to achieve statistical efficiency comparable to data parallelism, and that PipeDream's speedups are due to better system performance.

2.5.3 Comparison to Other Parallelism Schemes

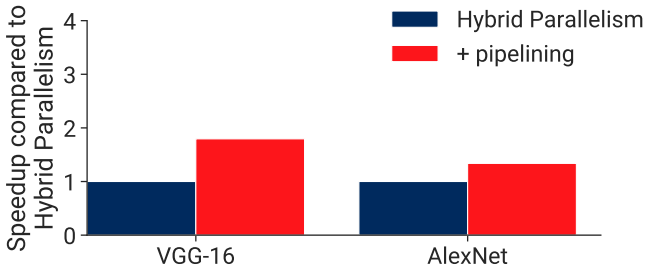
This section compares PipeDream to other parallelization techniques besides data parallelism.

Model Parallelism. Figure 2.13a compares model parallelism (blue bars), straight pipelines without replication (green bars), and pipelining with stage replication (red bars). For all four models, pipelining alone increases throughput by 2× or more. For GNMT-8 and GNMT-16, PipeDream's optimizer chooses not to replicate any stages, resulting in identical configurations for the green and red bars. For VGG-16 and AlexNet, PipeDream replicates the first stage, leading to speedups of 14.9× and 6.5× compared to model parallelism.

Hybrid Parallelism. Figure 2.13b shows that pipelining for a configuration that combines data and model parallelism (similar to those proposed by Krizhevsky et al. [100] and FlexFlow [96, 94]) increases throughput by as much as 80%. In running FlexFlow for AlexNet on Cluster-B (not shown



(a) Model Parallelism.



(b) Hybrid Parallelism.

Figure 2.13: Comparison of PipeDream (red) to non-DP parallelism techniques for 4-GPU configurations on Cluster-A.

in Figure 2.13b), we observe that PipeDream is 1.9× faster; a speedup due to pipelining over hybrid parallelism. Note that the same number of bytes are being communicated across workers with and without pipelining. Speedups are achieved by overlapping compute and communication, and consequently better utilization of compute resources.

2.5.4 Comparison to GPipe

We compare training GNMT-16 using PipeDream and our implementation of GPipe using 16 GPUs on Cluster-A and Cluster-B. GPipe does not provide an algorithm for partitioning work across stages, so we use the same partitions as PipeDream. GPipe also does not provide an algorithm for how many inputs should be permitted into the pipeline. When we set the number of inputs to be equivalent to "NOAM" in PipeDream (§2.3.2), GPipe experiences 55% and 71% throughput slowdowns compared to PipeDream on Cluster-A and Cluster-B, respectively. Setting the number of inputs in the pipeline for GPipe to the largest number that does not cause an out-of-memory exception, leads to throughput slowdowns of 35% and 42% on Cluster-A and Cluster-B, respectively. These throughput slowdowns are due to more frequent pipeline flushes compared to PipeDream (Figures 2.3 and 2.4).

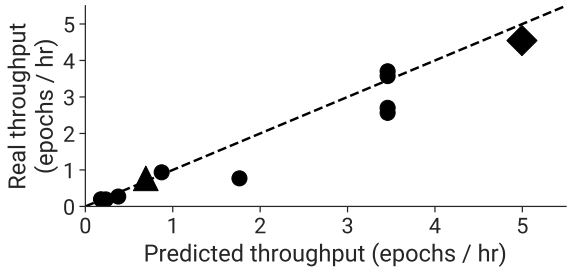


Figure 2.14: Real vs. optimizer's predicted throughput for VGG-16 with 16 workers. Each symbol represents a different partition, including the triangle for vanilla data-parallelism and the diamond for the optimizer's selection.

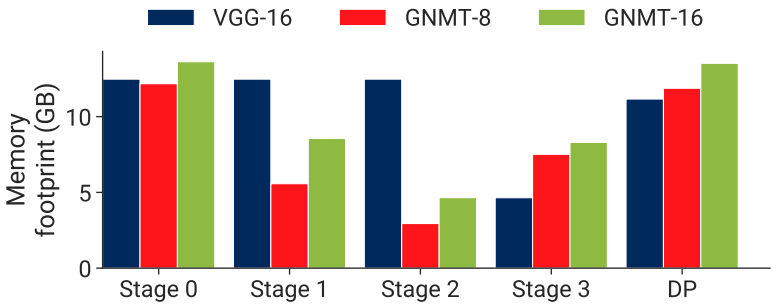


Figure 2.15: Memory footprint for various models using 4 GPUs. Per-GPU memory footprint is shown for data parallelism, and is identical on all GPUs.

2.5.5 Microbenchmarks

We evaluate PipeDream's optimizer, its communication overhead and memory footprint, and the effect of the number of in-flight inputs on throughput and memory footprint.

Optimizer. PipeDream's optimizer is efficient, generating optimal training configurations in under 8 seconds for all models and hardware deployments evaluated. As one example, Figure 2.14 shows real vs. predicted throughputs for various configurations for VGG-16 with 16 workers. Predicted and real throughputs are strongly linearly correlated, and the optimizer picks the best configuration among those tested.

Memory Footprint. Figure 2.15 shows the per-stage memory footprint of PipeDream for 4-stage configurations for three different models. PipeDream's worst-case memory footprint is on par with that of data parallelism, even though PipeDream stashes multiple weight and activation versions. This is because each stage in PipeDream is responsible for only a fraction of the total number of weights and activations in the model. As PipeDream scales to include more stages, the memory

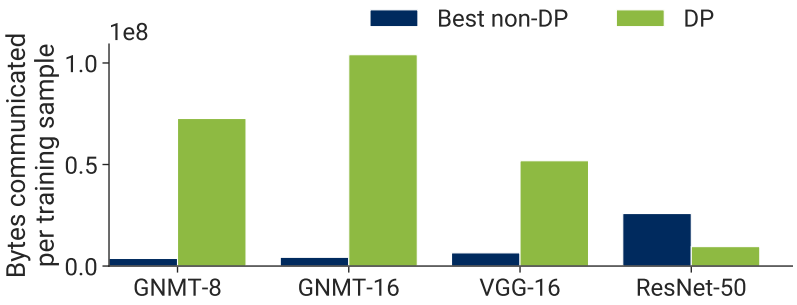


Figure 2.16: Bytes communicated per training sample by data-parallel (DP) and the best non-DP configurations for 4 GPUs on Cluster-A.

footprints remain consistent as discussed in §2.3.3.

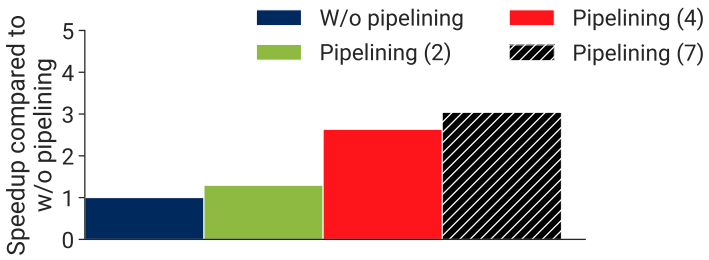
Communication Overhead. Figure 2.16 shows the amount of communication performed *per training sample* in the best non-DP configuration compared to the amount of communication performed in data-parallel training. For GNMT-8, GNMT-16, and VGG-16, the communication overhead for the best non-DP configuration is far less than the communication overhead for the DP configuration. For ResNet-50, the amount of communication for the best *non-data-parallel* configuration is higher than the DP configuration, thus explaining why PipeDream's optimizer chooses to perform ResNet-50 training using a data-parallel configuration.

Effect of Number of In-Flight Inputs. Figure 2.17 shows the effect of varying the number of in-flight inputs on throughput and memory overhead for GNMT-8. We make three observations:

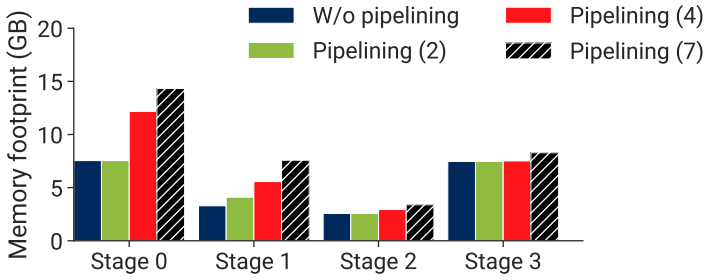
1. Memory footprint with no pipelining is different across stages, since PipeDream's optimizer tries to load balance compute and communication, and *not* memory footprint (the working set still fits comfortably in GPU memory).

2. As the number of in-flight inputs increases from 2 to 7, memory footprint increases because the number of weights and activations that need to be stashed increases proportionally.

3. In our experiments, setting the number of in-flight inputs to 4 (NOAM) and 7 give the highest throughput. While the working set of stages fits in GPU memory (16 GB), if required, the number of in-flight inputs can be decreased to trade throughput for reduced memory footprint. Throughput increases as this number increases since communication can be more easily hidden as the number of inputs in the pipeline increases.



(a) Throughput.



(b) Memory Overhead.

Figure 2.17: Effect of number of in-flight inputs (number in parentheses in legend) on throughput and memory overhead for GNMT-8 on 4 V100s in Cluster-A.

2.6 Summary

Pipeline parallelism can help reduce the communication overheads that can bottleneck data parallelism. PipeDream automatically partitions DNN training across workers, combining pipeline parallelism with data parallelism to better overlap computation with communication while minimizing the amount of data communicated. PipeDream proposes a pipelining schedule with relaxed semantics compared to data parallelism, but can still achieve large end-to-end speedups in time-to-accuracy. Compared to state-of-the-art approaches, PipeDream's automated scheduling approach helps complete training up to 5.3× faster across a range of DNNs and hardware configurations.

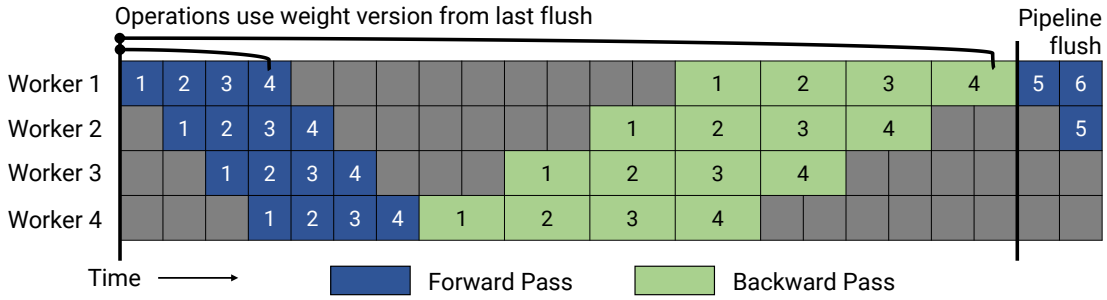
Chapter 3

Memory-Efficient Pipeline Parallelism for Large Model Training

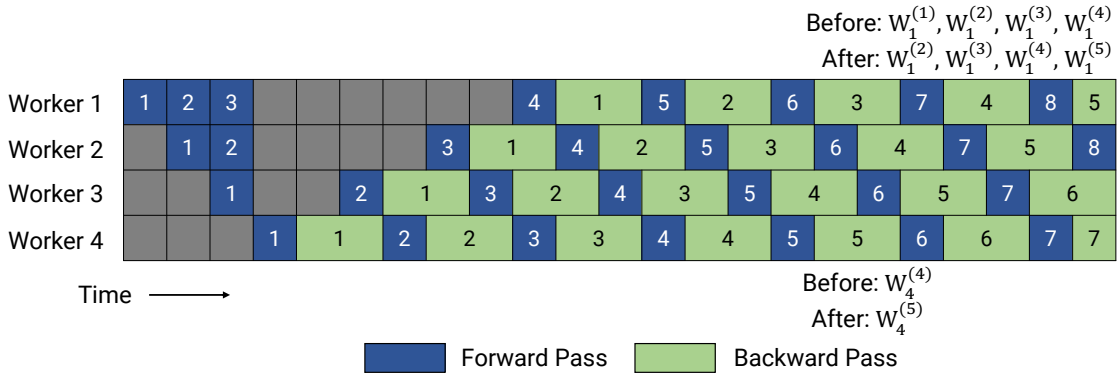
3.1 Introduction

In the quest to achieve higher accuracy across a range of tasks, DNN models have grown in size, often by scaling up the number of parameters in existing architectures [66, 135, 136, 45]. It is challenging to train large models with billions of parameters. Modern accelerators have limited memory, which means that the model parameters and intermediate outputs that need to be in accelerator memory during training might not fit on a single accelerator. One of the solutions researchers and practitioners have turned to is model-parallel training [62, 55], where a model is partitioned over multiple accelerator devices. However, model parallelism, when traditionally deployed, can either lead to resource under-utilization [125] or high communication overhead with good scaling only within a multi-GPU server [153], and consequently an increase in training time and dollar cost.

Recent work has proposed *pipelined* model parallelism to accelerate model-parallel training. For example, GPipe [86] and PipeDream (Chapter 2) push multiple inputs in sequence through a series of workers that each manage one model partition (contiguous layers in the model), allowing different workers to process different inputs in parallel. Naïve pipelining can harm model convergence due to inconsistent weight versions between the forward and backward passes of a particular input. Existing techniques trade off memory footprint and throughput in different ways to avoid this. GPipe maintains a single weight version, but has periodic *pipeline flushes* where the pipeline is drained of inputs to update weights (Figure 3.1a); these flushes limit overall throughput as resources are idle. PipeDream does not periodically flush the pipeline but stores multiple weight versions, which increases throughput but also increases the memory footprint, making the training of large models infeasible due to memory constraints. Efficient training of large models requires an approach with



(a) GPipe.



(b) PipeDream.

Figure 3.1: Timelines of different pipeline-parallel executions. Without loss of generality, forward and backward passes are assumed to take twice as long as forward passes; forward passes are shown in blue and backward passes are shown in green. Numbers indicate microbatch ID, time is shown along x-axis, per-worker utilization is shown along the y-axis. GPipe maintains a single weight version, but periodically flushes the pipeline. PipeDream does not introduce periodic pipeline flushes, but maintains multiple weight versions. For PipeDream, weight versions before and after the backward pass of input 5 are shown.

both high throughput and low memory footprint.

Additionally, the performance of a pipeline-parallel system is dependent on how DNN model operators are partitioned over workers. This is challenging for three reasons:

- **Memory Capacity Constraints:** Parameters and intermediate activations associated with a model partition need to fit in the main device memory of the accelerator.
- **Heterogeneous Network Interconnects:** Training deployments today feature heterogeneous network topologies, with higher-bandwidth links between devices on the same server.
- **Large Search Space for Operator Placement:** As model sizes increase, splitting an operator graph becomes computationally expensive since the number of distinct partitionings is exponential in the model size.

In this chapter, we introduce *double-buffered weight updates* (2BW), a pipeline schedule for efficient (high throughput *and* low memory footprint) pipeline-parallel training of DNN models with billions of parameters. 2BW reduces the memory footprint of training while avoiding pipeline flushes. We leverage the fact that every input's generated gradient does not need to be applied to weights immediately, and instead can be accumulated into a "coalesced" gradient to limit the number of weight versions maintained. Instead of flushing the pipeline before using newly updated weights, 2BW uses the new weights for inputs newly admitted into the pipeline, while using the previous weight version, called the *shadow version*, for already in-flight inputs. This double buffering of weights at each worker yields a pipelining scheme with higher throughput than GPipe (no pipeline flushes) and better memory efficiency than PipeDream (2 weight versions, versus worst case of d in PipeDream for a depth-d pipeline). 2BW introduces a *constant* weight delay term of 1, consistent across stages, while updating weights (weight update equation of $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)})$), which we show has empirically similar model convergence to vanilla weight updates (§3.4.1). We also present a variant of 2BW (called the *PipeDream-Flush* schedule) that trades off throughput for even lower memory footprint and vanilla semantics (weight update equation of $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$).

Second, we provide a planning algorithm that yields effective parallelization schemes for many of today's large model architectures. The 2BW *planner* partitions DNN operators over the available workers while taking into account the memory capacities of the accelerator devices, and addresses the three challenges highlighted earlier. The 2BW planner exploits the repetitive structure of large DNNs, e.g., transformer layers in BERT [66], to explore the space of schedules where each stage in the pipeline is replicated *equally*. This choice reduces the size of the search space explored drastically compared to existing work like PipeDream and FlexFlow [96], while still providing effective model splits in practice. The planner determines the size of each model partition, batch size, and whether to use memory-saving optimizations like *activation recomputation* [53, 77]: it considers the impact of these decisions on both throughput and memory footprint, unlike PipeDream and FlexFlow. Finally, the planner tries to ensure expensive communication stays on high-speed intra-server interconnects. This facilitates the automated scheduling of operators in the training computation graph for large transformer-based language models widely used in Natural Langauge Processing applications.

We find that the Adam optimizer with 2BW has a similar training loss trajectory to vanilla Adam with the same batch size, with similar accuracy on downstream finetuning tasks. PipeDream-2BW achieves end-to-end speedups of 1.3× to 20× for various GPT models compared to an optimized model-parallel baseline. PipeDream-2BW is up to 3.2× faster than GPipe, and is able to train large transformer models that vanilla PipeDream cannot fit in memory.

3.2 PipeDream-2BW System Design

PipeDream-2BW uses memory-efficient pipeline parallelism to train large models that do not fit on a single accelerator. Its *double-buffered weight update (2BW)* and *flush* mechanisms ensure high throughput, low memory footprint, and weight update semantics similar to data parallelism. PipeDream-2BW splits models into stages over multiple workers, and replicates each stage an equal number of times (with data-parallel updates across replicas of the same stage). Such *parallel pipelines* work well for models where each layer is repeated a fixed number of times (e.g., transformer models).

3.2.1 Double-Buffered Weight Updates (2BW)

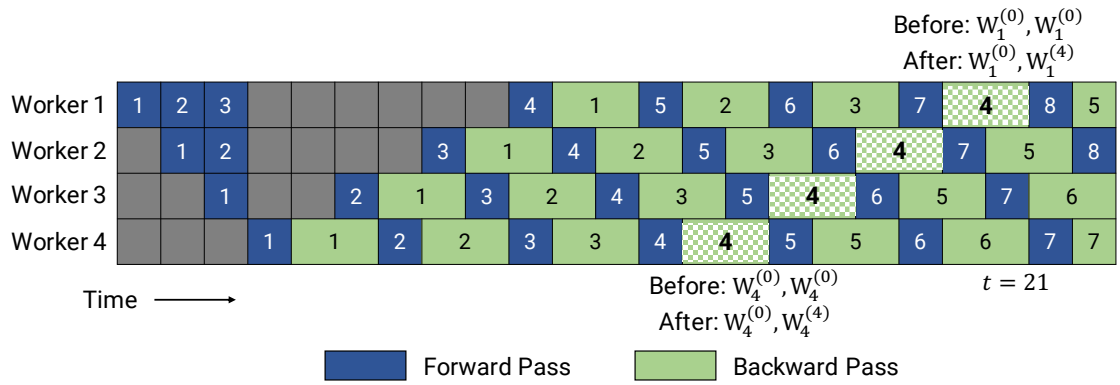


Figure 3.2: Timeline showing PipeDream-2BW's double-buffered weight update (2BW) scheme with time along x-axis. *Without loss of generality*, backward passes are assumed to take twice as long as forward passes. PipeDream-2BW only stashes two weight versions at every worker, reducing the total memory footprint while no longer requiring expensive pipeline stalls. $W_i^{(v)}$ indicates weights on worker i with version v (contains weight gradient generated from input v). New weight versions are generated in checkered green boxes; $W_4^{(4)}$ is first used for input 9's forward pass.

PipeDream-2BW uses a novel double-buffered weight update (2BW) scheme in conjunction with 1F1B scheduling [125], where each worker alternates between forward and backward passes for different inputs, to ensure that the same weight version is used in both the forward and the backward pass for a particular input (Figure 3.2). 2BW has a lower memory footprint than PipeDream and GPipe, and also avoids GPipe's expensive pipeline flushes.

Gradients are computed at the granularity of smaller *microbatches*. For any input microbatch, PipeDream-2BW uses the same weight version for an input's forward and backward passes. Updates are accumulated over multiple microbatches before being applied at the granularity of a batch, limiting the number of weight versions generated and maintained. Figure 3.2 shows an example timeline of 2BW. PipeDream-2BW generates a new weight version once every m microbatches ($m \geq p$, the number of pipeline stages). For simplicity, we will initially assume that $m = p$ (p is 4 in

Figure 3.2). A new weight version *cannot* be used immediately. In particular, in-flight inputs cannot use the newest weight version for their backward passes (for example, input 7 on worker 3 at $t = 21$), since the forward pass for these inputs was already initiated using an older weight version on a different stage. Thus, newly generated weight versions need to be buffered for future use. However, the total number of weight versions that need to be maintained is at most 2, since the weight version used to generate a new weight version can immediately be discarded (no future inputs that pass through that stage use the old weight version any longer). For example, in Figure 3.2, each worker can discard $W_i^{(0)}$ once they are done processing the *backward pass* for input 8 since all subsequent inputs use a later weight version for both their forward and backward passes.

The weight version a given input microbatch k (1-indexed) uses is $\max(\lfloor (k-1)/m \rfloor - 1, 0)$, where m is the number of microbatches in a batch (4 in Figure 3.2). This weight version is the same for both the forward and backward passes for input k. m can be any number $\geq p$; additional gradient accumulation (larger m) increases the global batch size.

Memory Footprint. PipeDream-2BW maintains 2 weight versions, and activation stashes for all in-flight microbatches. The number of in-flight microbatches at any stage is at most the number of pipeline stages (p); this follows from reusing the 1F1B schedule from Chapter 2. With activation recomputation, PipeDream-2BW's memory footprint can be decreased, since only input activations (as opposed to the full intermediate activation) need to be maintained for all in-flight microbatches. With activation recomputation, PipeDream-2BW's worst-case memory footprint is $\frac{2|W|}{p} + \frac{|A^{\text{total}}(b)|}{p} + p|A^{\text{input}}(b)|$. $|W|$ is the size of weight parameters for the full model, $|A^{\text{total}}(b)|$ is the size of intermediate activations for microbatch size b for the full model, and $|A^{\text{input}}(b)|$ is the size of input activations for microbatch size b *for a pipeline stage*.

In comparison, GPipe needs to checkpoint potentially a much larger number of input activations – proportional to the total number of microbatches accumulated within the pipeline before applying a weight update (m). With activation recomputation, GPipe's memory footprint with a per-GPU microbatch size b is $\frac{|W|}{p} + \frac{|A^{\text{total}}(b)|}{p} + m|A^{\text{input}}(b)|$. Since $|W| \ll |A(b)|$ for even small b for most models [89], the memory savings from maintaining one fewer weight version is small. To achieve high throughput, GPipe must use a large value of m to amortize away the cost of pipeline flushes; at such high m, its memory footprint is higher than PipeDream-2BW. Additionally, due to its higher memory footprint, GPipe must always use activation recomputation. Activation recomputation, however, reduces throughput by about 33%, and should be avoided if possible.

Semantics. We can also formalize the semantics of 2BW. For this discussion, we assume an unreplicated pipeline with p stages. If b is the per-GPU microbatch size, then gradients are averaged over m microbatches; thus, the effective batch size is $B = b \cdot m$.

We denote $W^{(t)}$ as the weight version after t batches of size B. $\nabla f(W)$ is the gradient averaged

over the B samples in the batch. Vanilla batch SGD (f is the loss function, ν is the learning rate) then has the following weight update equation(note that with 2BW, the delay term at every stage is the same; consequently, we get rid of the superscripts for brevity in this chapter):

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)}).$$

2BW's weight update semantics (with a delay term of 1 *across* all stages) are almost unchanged:

$$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)}).$$

We show that this delay term does not affect model convergence significantly in §3.4.1. Intuitively, the parameters of the model do not change significantly across single iterations, so $W^{(t)} \approx W^{(t-1)}$. The semantics with a replication factor greater than 1 is similar, with the batch size multiplied by the number of replicas (as with regular data parallelism). Other momentum-based optimizers such as Adam can be similarly analyzed (momentum term uses a weight gradient computed on a 1-stale weight version instead of latest version). Extra shadow variables are not needed. For example, m_t in batch SGD with momentum can be computed as (ignoring bias corrections):

$$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot \nabla f(W^{(t-1)}).$$

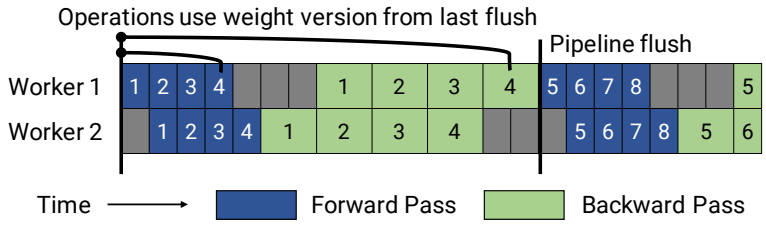
The final weight update equation is then:

$$W^{(t+1)} = W^{(t)} - \nu \cdot m_t.$$

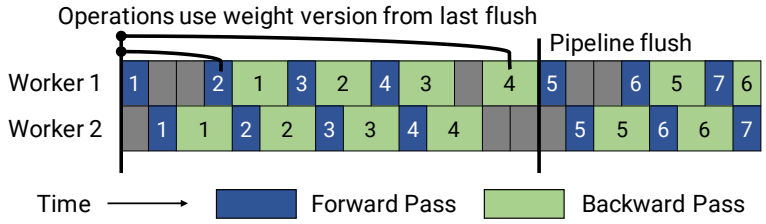
3.2.2 Weight Updates with Flushes (PipeDream-Flush)

We also propose a second memory-efficient pipeline schedule called PipeDream-Flush. It has lower memory footprint than 2BW and vanilla optimizer semantics, at the cost of lower throughput. This schedule reuses the 1F1B schedule from PipeDream [125], but maintains a single weight version and introduces periodic pipeline flushes to ensure consistent weight versions across weight updates. Timelines for PipeDream-Flush and GPipe with 2 pipeline stages are shown in Figure 3.3.

Memory Footprint. With PipeDream-Flush, the total number of in-flight "active" input activations is less than or equal to the pipeline depth, giving it lower memory footprint than GPipe, which has to maintain input activations proportional to the number of microbatches over which gradients are averaged (m). PipeDream-Flush's memory footprint is also lower than PipeDream-2BW since it only needs to maintain a single weight version (versus 2 with PipeDream-2BW).



(a) GPipe.



(b) PipeDream-Flush.

Figure 3.3: Timelines of GPipe and PipeDream-Flush for 2 stages. Both GPipe and PipeDream-Flush use pipeline flushes; PipeDream-Flush alternates between forward and backward passes in steady state to keeping memory footprint low compared to GPipe by limiting activation stashes to only in-flight microbatches.

Semantics. Periodic pipeline flushes ensure that weight updates can be performed with gradients computed using the latest weight version. This results in weight updates of the form $W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$ (same as GPipe). We compare 2BW's statistical efficiency (rate of model convergence) to the vanilla semantics of PipeDream-Flush, GPipe, and data parallelism, in §3.4.1.

3.2.3 Equi-replicated Stages (Parallel Pipelines)

PipeDream-2BW executes DNN training using a hybrid parallelization scheme which combines data and model parallelism with input pipelining. Since large deep models today feature extremely repetitive structures, with the same block repeated multiple times, a simple way of load balancing computation and communication involves breaking up a model into stages with an equal number of blocks and replication factors. Model training in PipeDream-2BW can thus be thought of as a collection of parallel pipelines (Figure 3.4), where inputs and intermediate output activations within a pipeline do not ever need to be sent to workers responsible for a different pipeline. Intermediate activations and gradients can be communicated *within* a pipeline using point-to-point communication primitives, such as `send` and `recv`. As with PipeDream, weight gradients need to be aggregated across stage replicas in different pipelines. Figure 3.4 shows an example: each model copy is split across 3 workers (number of stages, p is 3), and each stage is replicated twice (number of pipelines or data-parallel size, d is 2). Stage replicas can be placed on the same server so that expensive

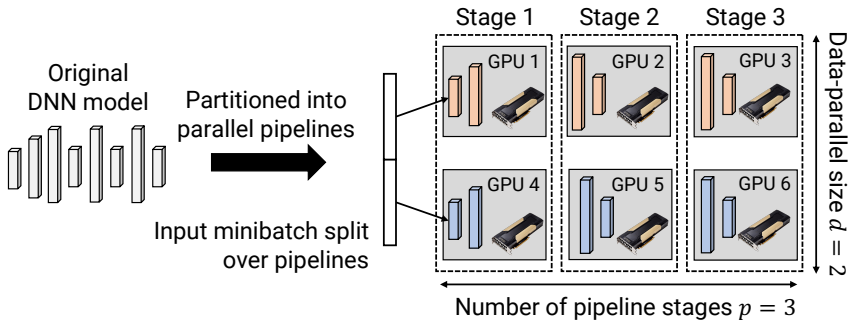


Figure 3.4: Example PipeDream-2BW $(2, 3)$ configuration. The model is partitioned into 3 stages (p is 3) and each pipeline is replicated twice (w is 2). Each pipeline replica is shown in a different color. The input batch is split over the parallel pipelines.

all-reduce updates are between GPUs on the same server with high-bandwidth interconnects.

3.3 Planner

PipeDream-2BW's *planner* determines how to split a model over the available compute devices by exhaustively searching over the *reduced* search space of all possible parallel-pipeline configurations. The planner also determines whether memory-saving optimizations should be deployed, and the per-GPU microbatch size and degree of gradient accumulation, given a maximum *safe* global batch size verified to not compromise model convergence (e.g., determined from past hyperparameter sweeps without pipelining).

PipeDream-2BW's planner uses a cost model for the compute times and memory footprints of individual blocks in the model. Computation time and memory cost functions allow PipeDream-2BW to reason about the impact of the data-parallel size, number of pipeline stages, and memory-saving optimizations (such as activation recomputation) on throughput and memory footprint. For example, a configuration with a greater number of pipeline stages has additional memory capacity, allowing for a larger maximum per-GPU microbatch size; this can increase the arithmetic intensity (number of floating point operations performed per memory load) of kernels [97], and consequently throughput. Communication times for tensors can be estimated by dividing the size of the tensor by the respective bandwidth. Expensive communication (e.g., large tensors, or all-reduce communication needed to coalesce weight gradients across stage replicas) can be placed on high-bandwidth links within the server by orienting pipelines appropriately.

Profiling for cost modeling can be done in two ways: end-to-end for each distinct configuration, or extrapolating from an individual block's measurements. End-to-end profiling is cheap (2 to 3 minutes per configuration), which means total profiling time is still a couple of hours (compared to the days to weeks needed for model training). Optimal configurations can be reused for a given

server and model deployment. We describe how per-block time and memory measurements can be extrapolated in §3.3.3 – this is even cheaper, but provides less accurate cost estimates. The highest-throughput configuration is chosen that also fits within the accelerator memory capacity.

3.3.1 Activation Recomputation

Activation recomputation is a common technique [86, 53, 77] that trades off extra computation for a lower memory footprint. With activation recomputation, activation stashes are not left materialized on the device between forward and backward passes; instead, only *input* activations on each stage are stashed, and the remaining activations needed in the backward pass are recomputed when required by re-running the forward pass. Activation recomputation trades off extra computation for a lower memory footprint.

Activation recomputation is useful for two reasons: it can enable larger per-GPU microbatch sizes to fit in memory, which can improve device throughput by increasing the arithmetic intensity of kernel. It can also enable the training of large models. Concretely, in some cases, the target accelerator device does not have sufficient memory capacity to store full activation stashes for all in-flight microbatches. This is especially true for deep pipelines, since the number of in-flight inputs with the 1F1B schedule from Chapter 2 (used by both PipeDream-2BW and PipeDream-Flush) is proportional to the number of pipeline stages (p).

3.3.2 Partitioning Algorithm

Putting it all together, given a total memory capacity M, PipeDream-2BW's planner first determines the largest per-GPU microbatch size that fits on a given worker (and the corresponding throughput) with and without each memory-savings optimization deployed using a memory cost function. The partitioning algorithm also verifies that the resulting global batch size is lower than the maximum safe batch size B . Each memory-savings optimization can be integrated into PipeDream-2BW's planner by specifying a corresponding throughput and memory cost function.

PipeDream-2BW's planner then sweeps all (d, p) values to determine the best pipeline configuration for a given model and hardware deployment. Configurations with memory footprint higher than the memory capacity M of the device (modeled by the MEMORY(.) cost function) are discarded. Gradient accumulation can be used to increase the batch size to B. The partitioning algorithm aims to pick a configuration that has a high compute-to-communication ratio, while accounting for the communication time across stages in the same pipeline and across replicated stages (modeled by the THROUGHPUT(.) cost function). Pseudocode is shown in Algorithm 1.

Algorithm 1 Algorithm for PipeDream-tbw's Planner.

Input: Model m, memory capacity M, m's associated search function SEARCH$(.)$, m's associated throughput cost function THROUGHPUT$(.)$, m's memory footprint cost function MEMORY$(.)$, maximum safe batch size B.
Return: Optimal data-parallel size and number of pipeline stages d^{opt} and p^{opt}, optimal per-GPU microbatch size b^{opt}, boolean whether activations should be recomputed r^{opt}, optimal degree of gradient accumulation g^{opt}.

Initialize $t^{\text{max}} = 0, d^{\text{opt}} = \text{NULL}, p^{\text{opt}} = \text{NULL}$
for $d = 1$ **to** N **do**
  for $p = 1$ **to** N/w **do**
    // For given data-parallel size d, number of pipeline stages p, and batch size B, find optimal microbatch size and whether activation recomputation should be performed.
    $b, r = m.\text{SEARCH}(d, p, B)$

    $t = m.\text{THROUGHPUT}(d, p, b, r)$
    if $m.\text{MEMORY}(d, p, b, r) > M$ **then**
      continue
    if $t > t^{\text{max}}$ **then**
      $t^{\text{max}} = t, d^{\text{opt}} = d, p^{\text{opt}} = p, b^{\text{opt}} = b, r^{\text{opt}} = r$
$g^{\text{opt}} = B/(N \cdot b^{\text{opt}})$ *// To reach batch size B.*

3.3.3 Closed-Form Cost Functions

For every possible configuration of data-parallel and pipeline-parallel sizes, PipeDream-2BW's planner explores the benefit of pipelining and each space-saving optimization. For example, with activation recomputation as a target memory-savings optimization, PipeDream-2BW considers three executions:

- Model and data parallelism without pipelining (with the largest per-GPU microbatch size that fits in memory).
- Hybrid parallelism with pipelining and without activation recomputation (all required weight versions and activation stashes in memory for in-flight microbatches).
- Hybrid parallelism with pipelining and recomputation.

PipeDream-2BW's planner estimates the throughput and memory footprint of each of these possible executions using a cost model. PipeDream-2BW's planner then tries to find the configuration with highest throughput that also fits in main device memory of the accelerators used (memory capacity provided as input). In this section, we show one such cost model for throughput and memory.

In our experiments, we used profile-based cost functions that run configurations end-to-end for a couple of hundred iterations. However, performance of different parallel configurations can also be estimated using closed-form expressions that use more fine-grained profile information (e.g., time and memory footprint of each transformer block). We present one such cost model here.

Cost Function for THROUGHPUT(.)

The throughput of various hybrid-parallel setups with and without pipelining can be modeled using the times of forward and backward passes obtained from a simple profiling step. Let b be the largest per-GPU microbatch size without additional weight and activation versions, and b' be the largest per-GPU microbatch size that can fit on the device when multiple versions are needed ($b' \leq b$). As before, d and p are the data-parallel size and number of pipeline stages.

Consider the following notation:

- $T_i^{\text{comp}}(b, d, p)$ is the compute time of stage i with a per-GPU microbatch size b.
- $T_{i \rightarrow j}^{\text{comm}}(b, d, p)$ is the communication time of activations and gradients between stages i and j with microbatch size b.
- $T_i^{\text{comm}}(b, d, p)$ is the communication time of exchanging gradients between d replicas of stage i with microbatch size b.

We assume that the global batch size used is B. With data-parallel size d and microbatch size b, data-parallel communication is required every $m(b, d) = B/(d \cdot b)$ microbatches.

Then, without pipelining, each microbatch of size b takes the following computation time, t:

$$t = \sum_i \max(T_i^{\text{comp}}(b, d, p) + \sum_j T_{j \rightarrow i}^{\text{comm}}(b, d, p), \frac{1}{m(b, d)} \cdot T_i^{\text{comm}}(b, d, p)).$$

With pipelining, computation of different stages can be overlapped. A microbatch of size b' can then be processed every t seconds, where t is given by the expression:

$$t = \max_i \max(T_i^{\text{comp}}(b', d, p) + \sum_j T_{j \rightarrow i}^{\text{comm}}(b', d, p), \frac{1}{m(b', d)} \cdot T_i^{\text{comm}}(b', d, p)).$$

With activation recomputation, the number of floating point operations increases, since forward passes need to be repeated to recompute the activation stashes needed in the backward pass. We use a constant multiplier c^{extra} to represent this. $c^{\text{extra}} = 4/3$ is a reasonable value for this constant, since the backward pass typically takes twice as long as the forward pass. c^{extra} can also be measured empirically. Arithmetic intensity might also increase, which is captured by $T_i^{\text{comp}}(.)$ being a function of the microbatch size b. Communication time remains unchanged from before. Every b inputs can

now be processed in time t, where t is given by,

$$t = \max_i \max(c_i^{\text{extra}} \cdot T_i^{\text{comp}}(b, d, p) + \sum_j T_{j \to i}^{\text{comm}}(b, d, p), \frac{1}{m(b, d)} \cdot T_i^{\text{comm}}(b, d, p)).$$

The throughput in samples per second of each of these setups is then the corresponding per-GPU microbatch size (b or b') divided by t.

Estimating $T^{\text{comp}}(.)$. $T_i^{\text{comp}}(b, d, p)$ is the compute time of stage i with per-GPU microbatch size b, and can be computed by summing up the forward and backward pass times of all blocks within the stage. If the number of pipeline stages is p and the total number of blocks in the model is B, then the total number of blocks in a given stage is B/p. Forward and backward pass times for each stage can be estimated by profiling 100–200 iterations of training.

Estimating $T^{\text{comm}}(.)$. Communication times can be similarly modeled. Let the size of the associated parameter with B total blocks be $|W|$, and the size of the block's input and output activations be $|A^{\text{inp.+out.}}(b)|$. With p pipeline stages, each pipeline stage has $1/p$ of the model parameters.

The time to communicate activations across stages can be computed as (factor of 2 for gradients in the backward pass),

$$T_{i \to j}^{\text{comm}}(b, w, p) = \frac{2|A^{\text{inp.+out.}}(b)| \cdot \mathbb{I}(p > 1)}{\text{bwdth}_{\text{in-pipeline}}(p)}.$$

The time to communicate weight gradients across stage replicas can be computed similarly given a bandwidth function $\text{bwdth}_{\text{cross-pipeline}}(d)$, and the number of bytes communicated during all-reduce. The number of byes communicated in an all-reduction can either be explicitly measured, or estimated using a closed-form expression.

$\text{bwdth}_{\text{in-pipeline}}(p)$ and $\text{bwdth}_{\text{cross-pipeline}}(d)$ represent the bandwidths for in-pipeline and cross-pipeline communication. These bandwidth functions can respect hierarchical network topologies. For example, if d is less than the number of workers in a single server, communication can be performed entirely within a server, using the higher intra-server bandwidth.

$$\text{bwdth}_{\text{cross-pipeline}}(d) = \begin{cases} B_{\text{high}} \text{ if } d < \text{number of GPUs in server}, \\ B_{\text{low}} \text{ otherwise.} \end{cases}$$

Cost Function for MEMORY(.)

The memory footprint can similarly be modeled using the sizes of activations and weights obtained from a profiling step. Let the total size of the weight parameters for the entire model be $|W|$, let the total size of the activations given a microbatch size b for the entire model be $|A^{\text{total}}(b)|$, and let the size of the input activations for a single stage be $|A^{\text{input}}(b)|$. With a pipeline of p stages, each pipeline stage has weight parameters of size $|W|/p$, and activations of size $|A^{\text{total}}(b)|/p$.

Without Activation Recomputation. Without activation recomputation, 2BW maintains 2 different versions of the weight parameters. PipeDream-2BW also maintains p activation versions (the total number of in-flight activations). This means the total PipeDream-2BW memory footprint is:

$$\frac{2|W|}{p} + \frac{p|A^{\text{total}}(b)|}{p} + p|A^{\text{input}}(b)|.$$

With Activation Recomputation. With activation recomputation, the total number of activation versions in GPU memory at any point in time is 1. This means that the PipeDream-2BW memory footprint with p stages is:

$$\frac{2|W|}{p} + \frac{|A^{\text{total}}(b)|}{p} + p|A^{\text{input}}(b)|.$$

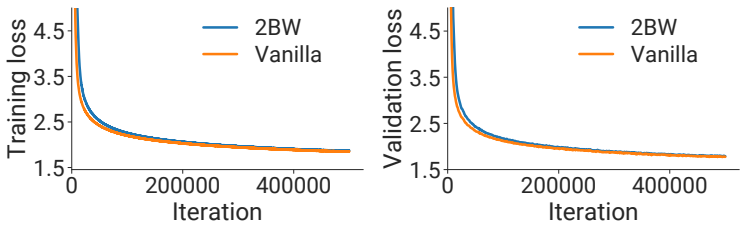
3.4 Evaluation

In this section, we show that the Adam optimizer with 2BW has similar semantics to vanilla Adam, and that PipeDream-2BW and PipeDream-Flush are able to train large models faster than existing model-parallel approaches including Megatron [153], and existing pipelining approaches like GPipe [86].

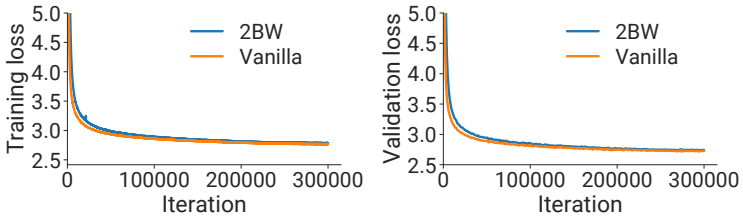
Hardware. We show results on two different hardware setups on AWS: eight 8×V100 servers (64 GPUs) with NVLink and 16GB per-GPU memory, and a single 8×V100 server (p3.16xlarge instances).

Implementation. Our implementation uses PyTorch and is adapted from the Megatron repository [14]; we verified that single-worker performance with this implementation achieves about 45 TFLOPS on a 355M-parameter GPT model and is competitive with existing state-of-the-art open source implementations from NVIDIA [19]. All results shown are with mixed precision.

Models. We evaluate PipeDream-2BW on BERT [66] and GPT [136], large transformer-based language models used for a number of NLP applications. In particular, most of our experiments are performed with GPT models with 1.3, 2.2, and 3.9 billion parameters, with similar layer dimensions to those used in the Megatron paper [153].



(a) BERT, 355M (batch size = 1024).



(b) GPT, 355M (batch size = 512).

Figure 3.5: Training and validation loss when pre-training BERT and GPT models with vanilla Adam and Adam with 2BW.

Baselines. We compare PipeDream-2BW to two types of baselines: (a) model parallelism without pipelining (tensor model parallelism used in Megatron, and inter-layer model parallelism); and (b) GPipe (we extend GPipe to use parallel pipelines, and refer to this *enhanced* version as GPipe in the rest of this chapter), which performs pipeline parallelism. We do not compare to PipeDream or data parallelism for the entire model since they cannot fit the above models in memory when using 16-GB V100 GPUs. With 64 GPUs, we use data parallelism *across stages* to scale up training.

Main Takeaways. We make the following observations:

- **Quality of Convergence:** 2BW weight update semantics yield pre-trained models which produce **comparable accuracy** on downstream finetuning tasks to vanilla Adam (GPipe and PipeDream-Flush) with the same batch size.
- **Comparison to Model Parallelism:** PipeDream-2BW is able to train a 3.8 billion-parameter GPT model up to **20**× faster compared to non-pipelining approaches.
- **Comparison to Other Pipelined Approaches:** PipeDream-2BW is up to **3.2**× faster than GPipe.

3.4.1 Quality of Convergence of 2BW

We pre-trained 355M-parameter BERT and GPT models with vanilla Adam and Adam with 2BW; we then finetuned the resulting BERT models. We note that GPipe, PipeDream-Flush, and DP have identical semantics, and hence are equivalent baselines ("Vanilla"). To provide a fair comparison,

Task	Metric	Vanilla	Vanilla (90%)	2BW
MNLI	Overall Accuracy	87.77%	N/A	87.82%
RACE	Overall Accuracy	80.06%	79.30%	79.48%

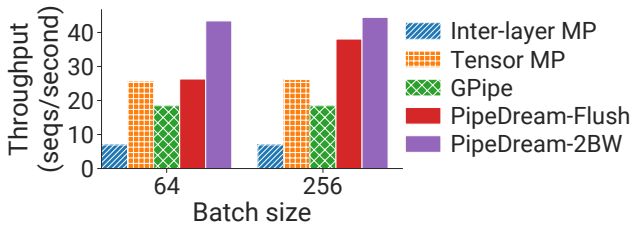
Table 3.1: Comparison of BERT models pre-trained with vanilla (all and 90% of iterations) and 2BW optimizers on finetuning tasks.

we use the *same* hyperparameters, including batch size, used by Megatron [153] to train these BERT and GPT models. For BERT, we use a batch size of 1024, and for GPT, we use a batch size of 512. We use the Adam optimizer with standard hyperparameters (learning rate of 10^{-4} with initial warmup and subsequent linear decay, maximum sequence length of 512), and mixed precision. We used the OpenWebText dataset [23] for pretraining. Figure 3.5 shows the training and validation loss for the two models. The training and validation losses for the 2BW runs track the vanilla runs almost identically after the first 100,000 iterations (when the model is changing more rapidly and the delay term matters more).

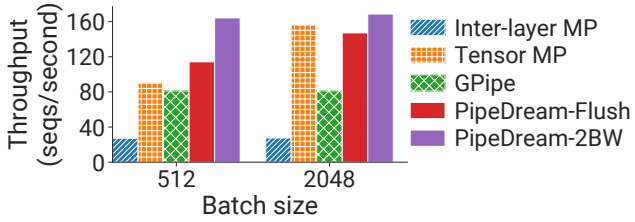
To further validate the quality of the pre-trained model, we finetuned the pre-trained vanilla and 2BW BERT models on downstream MNLI and RACE tasks [170, 104]. Both pre-training and fine-tuning were performed with the same hyperparameter and training setups, and we did not perform hyperparameter tuning for either – our goal here is to show that 2BW has nearly identical semantics to the corresponding vanilla optimizer. As shown in Table 3.1, the accuracy on each of these tasks is similar after finetuning. We also evaluated the vanilla and 2BW GPT models on the Wikitext-103 test dataset and got similar test perplexities (19.28 vs. 19.56); test perplexities match *exactly* when "Vanilla" is run for 20% fewer iterations.

3.4.2 Throughput

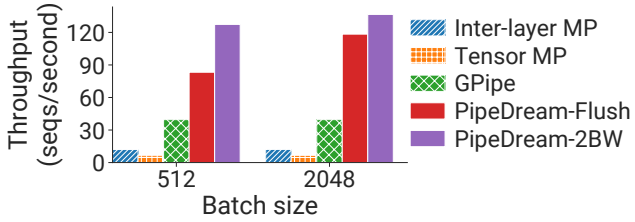
Figure 3.6 shows the throughputs of various PipeDream-2BW, PipeDream-Flush, and baseline configurations using 8 and 64 V100s with a sequence length of 512 for various large GPT models. Results with BERT models are similar (§3.4.6). We compare to two different forms of model parallelism, as well as GPipe. Data parallelism is not a viable baseline for these large models due to its high memory overhead. In these experiments, we use activation recomputation, and the largest per-GPU microbatch size that fits on the 16-GB V100 GPUs. We use the best configuration recommended by PipeDream-2BW's planner for all comparisons: 8-deep configurations for the model with 2.2 billion parameters, and 16-deep configurations for the model with 3.8 billion parameters. For each model, we show two different batch sizes to show the impact of batch size on throughput for approaches that use periodic flushes.



(a) GPT, 2.2B, 8-way model parallelism (8×V100s).



(b) GPT, 2.2B, 8-way model parallelism (64×V100s).



(c) GPT, 3.8B, 16-way model parallelism (64×V100s).

Figure 3.6: Throughput of various systems for different batch sizes for GPT models, using 8×16GB-V100 servers.

Model Parallelism without Pipelining. We compare against two model parallelism approaches: tensor model parallelism used by Megatron [153] where each layer is divided among all model-parallel workers, and inter-layer model parallelism where layers are sharded over the workers but inputs are not pipelined. On a single node, PipeDream-2BW is faster than tensor MP by **1.3**×. This grows to **20**× on 64 GPUs for the model with 3.8 billion parameters, when the all-to-all communication used by tensor MP needs to be performed across servers, which is expensive using AWS instances (bandwidth across multi-GPU servers is much lower than the bandwidth within server). Compared to inter-layer MP, pipelining with flushes increases throughput by up to **4.1**× for small batch sizes, and by up to **5.3**× for large batch sizes, on the 2.2-billion model. 2BW is up to **6.1**× faster than inter-layer MP.

GPipe. PipeDream-2BW outperforms corresponding GPipe configurations at the same global batch size by up to **3.2**× due to the lack of periodic pipeline flushes. GPipe natively has high memory

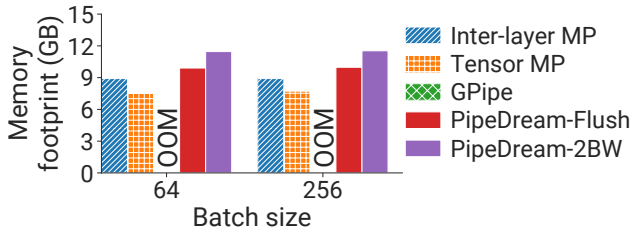


Figure 3.7: Worst-case memory footprint (in GB) of various systems with 8 V100 GPUs for a GPT model with 2.2 billion parameters.

footprint due to a large number of activation stashes: consequently, the maximum number of microbatches it can admit is small, leading to a larger pipeline bubble and **2.1**× worse throughput than PipeDream-Flush at low batch sizes, and **3**× at high batch sizes.

PipeDream-Flush and PipeDream-2BW. Figure 3.6 also compares PipeDream-2BW and PipeDream-Flush for two different batch sizes with different numbers of microbatches over which gradients are averaged ($m = p \cdot g$) within the pipeline. At low batch size, PipeDream-2BW is up to **1.6**× faster. With more gradient accumulation (batch size of 2048), this speedup drops to **15%**. However, high g is not always practical. Both PipeDream-Flush and PipeDream-2BW have weight updates with a batch size of $b \cdot w \cdot p \cdot g$, where the total number of workers is $w \cdot p$. For a large number of workers ($\gg 64$), the batch size is high even with $g = 1, m = p$, making additional gradient accumulation infeasible (batch size cannot scale to ∞ without affecting model convergence). Indeed, systems like Megatron [153], that train large transformer models using 512 GPUs, show state-of-the-art results across tasks using a global batch size ≤ 1024.

3.4.3 Memory Footprint

We measured the worst-case memory footprint of different systems on a GPT model, shown in Figure 3.7. GPipe runs out of memory at a batch size of 64, due to a larger number of activation stashes from its all-forward-all-backward schedule, even with activation recomputation (worst case of m input activation stashes with activation recomputation, compared to p for PipeDream-Flush). PipeDream-Flush has a slightly higher memory footprint compared to inter-layer model parallelism, since it needs to maintain activation stashes for more in-flight microbatches. PipeDream-2BW has a higher memory footprint than PipeDream-Flush due to an additional weight version (but still lower than GPipe's).

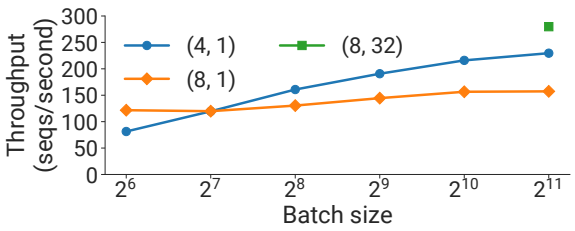


Figure 3.8: Throughput of two PipeDream-2BW configurations vs. global batch size for a 1.3-billion parameter GPT model using 64 V100 GPUs. The legend shows (p, b): the number of pipeline stages and the microbatch size.

3.4.4 Planning Decisions

In this sub-section, we analyze the implications of pipeline depth and width on performance. Figure 3.8 shows the throughputs of two PipeDream-2BW configurations for different batch sizes. We highlight relevant takeaways below.

Inter-Stage Communication. As the global batch size increases with gradient accumulation, throughput for each configuration increases due to less communication across stage replicas. This is especially true for configurations with communication across servers ($w > 8, p < 8$ for 8-GPU servers, e.g. p equal to 4) where inter-stage all-to-all communication is cross-node and more expensive.

Compute-Communication Ratio. Increasing the pipeline depth decreases the amount of computation in each pipeline stage while keeping the number of bytes communicated between stages constant. This makes the pipeline more communication-bound, decreasing throughput.

Maximum Per-GPU Microbatch Size. Increasing the pipeline depth increases the maximum microbatch size that fits in GPU memory. This leads to possibly higher arithmetic intensity and throughput. In Figure 3.8, we show throughput for two microbatch sizes for the $p = 8$ configuration; the larger microbatch size ($b = 32$) has higher throughput. Smaller pipeline depths cannot fit large microbatch sizes.

Maximum Model Size. Deeper pipelines support the training of larger models. We show the empirically measured maximum model size that can be trained with 2BW in Figure 3.9.

These observations illustrate the complexity in picking a configuration. For example, increasing pipeline depth leads to two effects (decreased compute-communication ratio within the pipeline and increased arithmetic intensity) that have opposing effects on throughput. PipeDream-2BW's planner automates this process for each combination of model, batch size, and number of GPUs.

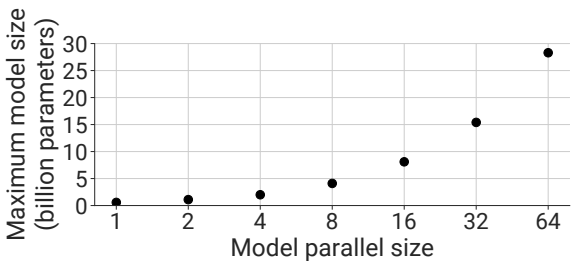


Figure 3.9: Maximum model size supported by various pipeline-parallel depths with 64 16-GB V100 GPUs using 2BW.

3.4.5 Maximum Model Size Supported

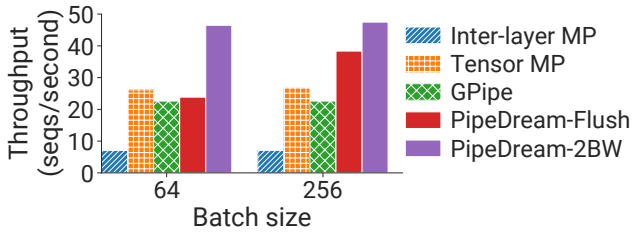
Figure 3.9 shows the empirically measured maximum model size supported by various pipeline depths while using 2BW. As can be seen in the figure, deeper configurations provide additional memory capacity. PipeDream-2BW is able to train models of up to almost 30 billion parameters using 64 16-GB GPUs. As a point of comparison, Megatron-LM [153] was able to train a model with 8.3 billion parameters with 8 32-GB GPUs ($2\times$ more memory).

3.4.6 Throughput and Memory Footprint with BERT Models

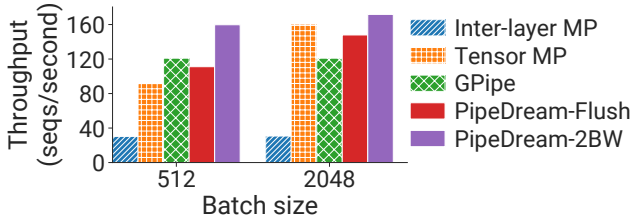
We also ran PipeDream-2BW on two BERT models: one with 2.2 billion parameters, and another with 3.8 billion parameters. Figure 3.10 compares PipeDream-2BW's throughput and Figure 3.11 compares PipeDream-2BW's memory footprint against the same baselines as before. We see that results are similar to GPT. One point of difference is that GPipe does not run out of memory at the batch size of 64 (for GPT, only a batch size of 32 fits in memory, leading to a larger pipeline bubble); however, GPipe still has higher memory footprint compared to all other baselines.

3.4.7 Impact of Activation Recomputation

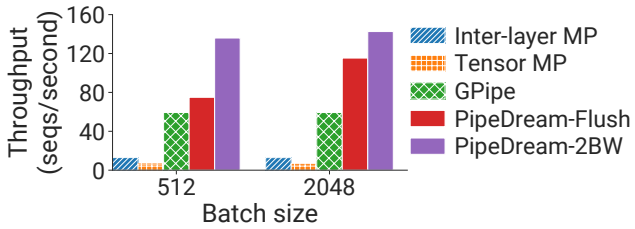
Figure 3.12 shows the effect of activation recomputation on throughput for various GPT models. For a given per-GPU microbatch size, recomputation introduces overhead (capped at 33% since the backward pass takes twice as long as the forward pass for most operators). However, recomputation allows for a larger per-GPU microbatch to fit on the worker, sometimes leading to higher throughput than without activation recomputation: activation recomputation leads to higher throughput in Figure 3.12b, but not in Figure 3.12a. In the extreme case (not pictured), recomputation makes it possible to train large models by reducing peak memory footprint of training.



(a) BERT, 2.2B, 8-way model parallelism (8×V100s).



(b) BERT, 2.2B, 8-way model parallelism (64×V100s).



(c) BERT, 3.8B, 16-way model parallelism (64×V100s).

Figure 3.10: Throughput of various systems for different batch sizes for BERT models. Results are shown with a single 8×V100 server, and with eight 8×V100 servers (with 16GB).

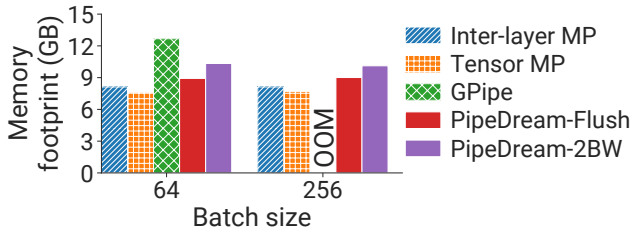
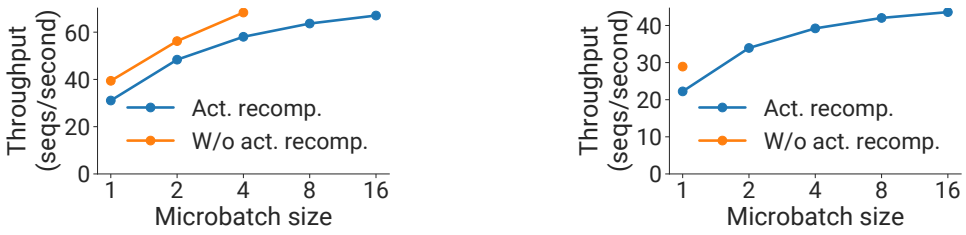


Figure 3.11: Worst-case memory footprint (in GB) with 8 V100 GPUs for a 2.2B BERT model.

3.5 Related Work and Discussion

In this section, we expand on work related to PipeDream-2BW, and place PipeDream-2BW's speedups in context with respect to PipeDream (discussed in Chapter 2), as well as other related work.



(a) GPT, 1.3B.

(b) GPT, 2.2B.

Figure 3.12: Throughput of $(1, 8)$ PipeDream-2BW configurations vs. per-GPU microbatch size for GPT models using a maximum sequence length of 512 and 8 16-GB-V100 GPUs, with and without activation recomputation. Activation recomputation helps increase the maximum per-GPU microbatch size that fits, especially for larger models, leading to higher throughput in some cases.

Model Parallelism in Real Deployments. NVIDIA used a custom intra-layer model parallelism scheme in its Megatron system [153] to train a GPT-2 model with 8.3 billion parameters on 64 32-GB V100 servers by parallelizing matrix multiplications across multiple workers. This approach can be combined with data parallelism. Multiple all-reductions are needed *per layer* to coalesce partial results produced on different GPUs, thus making training communication-bound at high numbers of model partitions (cross-node communication needed). In comparison, PipeDream-2BW trades off additional memory footprint (an extra weight version) for lower communication overhead (20× faster training when using multi-GPU servers on Amazon AWS with limited inter-node bandwidth).

Pipeline Parallelism. We showed quantitative comparisons to existing approaches for pipeline parallelism in §3.4.2. PipeDream-2BW trains large models up to 3.2× faster than GPipe at low batch sizes, due to a lack of periodic pipeline flushes, and lower memory footprint (allowing more inputs to be pushed into the pipeline). PipeDream cannot train these large models. PipeDream-2BW's lower memory footprint does come with tradeoffs, however – PipeDream-2BW accumulates weight gradients over multiple microbatches, increasing the minimum batch size that PipeDream-2BW supports. Thus, for models that only support very small batch sizes, PipeDream-2BW, PipeDream-Flush, and GPipe, which perform gradient accumulation within the pipeline, may not be viable.

PipeMare [175] uses asynchronous pipeline parallelism to provide high throughput (no pipeline flushes) with asynchronous weight update semantics. PipeMare offers two theoretically-motivated techniques to ensure good statistical efficiency. In contrast, PipeDream-2BW and all the baselines we compare against in the chapter (traditional data parallel training, PipeDream, GPipe), use synchronous execution where the weights used for the forward pass computation are the same as those used during the backward pass. PipeDream-2BW's double buffered weight updates use a 1-stale gradient update that is similar to the vanilla weight update. In our evaluation, we show that we do not require hyperparameter tuning to generate comparable results to synchronous execution.

Memory-Saving Optimizations. A rich line of work attempts to decrease the memory footprint of DNN training. Gist [89] employs lossless and lossy layer-specific encoding schemes to compress stashed activations. Systems such as Checkmate [90] systematically determine when activation recomputation [53, 77] should be performed. DeepSpeed [140] partitions optimizer state over data-parallel replicas instead of replicating it, using a technique called ZeRO. Such orthogonal optimizations can be combined and incorporated in PipeDream-2BW.

Planning Algorithms. PipeDream, DAPPLE [71], and FlexFlow [96] use planning algorithms to partition operator graphs over multiple accelerators to maximize throughput. Unfortunately, these planners do not exploit the repetitive nature of modern transformer-based models. For example, PipeDream's planner explores $O(n^3m^2)$ configurations (assuming n layers in the model and m workers). Furthermore, these planners do not consider the effect of memory-saving optimizations, which are critical for training large models efficiently (e.g., always applying activation recomputation can make the system 1.33$\times$ slower). PipeDream-2BW's planner, on the other hand, performs an exhaustive search of a *much reduced* search space since it only considers *parallel pipelines* (all possible (w, p) pairs with m workers is $O(m^2)$). Given this small number of explored configurations, Bagpipe's planner takes a fraction of a second with a closed-form cost model; PipeDream's partitioning algorithm with the same cost model takes about 30 minutes for large models.

3.6 Summary

In this work, we proposed and implemented PipeDream-2BW, a system for memory-efficient pipeline-parallel training that achieves high throughput, low memory footprint, and data parallelism-like semantics through a novel weight update double buffering strategy (2BW). PipeDream-2BW uses a planner to partition a model's operator graph over training resources in a memory-aware way. PipeDream-2BW accelerates the training of models with billions of parameters by up to 20$\times$ compared to model-parallel baselines, and by up to 3.2$\times$ compared to GPipe, on commodity hardware.

Chapter 4

PTD-P Parallelism: Training Models on Thousands of GPUs

4.1 Introduction

Transformer-based language models [164, 135, 136, 66, 113, 176, 138] in Natural Language Processing (NLP) have driven rapid progress in recent years as computation at scale has become more available and datasets have become larger. Recent work [45, 153] has shown large language models to be effective zero- or few-shot learners, with high accuracy on many NLP tasks and datasets. These large language models have a number of exciting downstream applications such as client feedback summarization, automatic dialogue generation, semantic search, and code autocompletion [1, 15, 7]. As a result, the number of parameters in state-of-the-art deep neural network (DNN) models for NLP have grown at an exponential rate (Figure 4.1). Training such models, however, is challenging for two reasons: (a) it is no longer possible to fit the parameters of these models in the main memory of even the largest GPU (NVIDIA recently released 80GB-A100 cards), and (b) even if we are able to fit the model in a single GPU (e.g., by swapping parameters between host and device memory [143]), the high number of compute operations required can result in unrealistically long training times (e.g., training GPT-3 with 175 billion parameters [45] would require about 288 years with a single V100 NVIDIA GPU). This calls for parallelism. Data-parallel scale-out usually works well, but suffers from two limitations: a) beyond a point, the per-GPU batch size becomes too small, reducing GPU utilization and increasing communication cost, and b) the maximum number of devices that can be used is the batch size, limiting the number of accelerators that can be used.

Various model parallelism techniques have been proposed to address these two challenges. For example, recent work [152, 153] has shown how tensor (intra-layer) model parallelism, where matrix multiplications within each transformer layer are split over multiple GPUs, can be used to

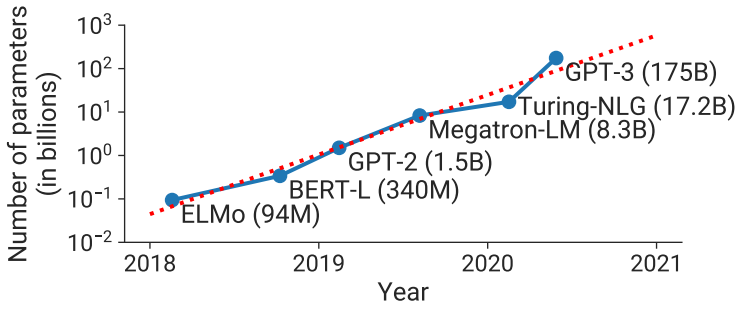


Figure 4.1: Trend of sizes of state-of-the-art Natural Language Processing (NLP) models with time. The number of floating-point operations to train these models is increasing at an exponential rate.

overcome these limitations. Although this approach works well for models of sizes up to 20 billion parameters on NVIDIA DGX A100 servers (with 8 80GB-A100 GPUs), it breaks down for larger models. Larger models need to be split across multiple multi-GPU servers, which leads to two problems: (a) the all-reduce communication required for tensor parallelism needs to go through inter-server links, which are slower than the high-bandwidth NVLink [22] available within a multi-GPU server, (b) a high degree of model parallelism can create small matrix multiplications (GEMMs), potentially decreasing GPU utilization.

Pipeline (model) parallelism [125, 86, 127, 175, 99, 71], as introduced in the previous chapters of this dissertation, is another technique to support the training of large models, where layers of a model are striped over multiple GPUs. A batch is split into smaller microbatches, and execution is pipelined across these microbatches. Layers can be assigned to workers in various ways, and various schedules for the forward and backward passes of inputs can be used. The layer assignment and scheduling strategy results in different performance tradeoffs. Regardless of schedule, to preserve strict optimizer semantics, optimizer steps need to be synchronized across devices, leading to a *pipeline flush* at the end of every batch, where microbatches are allowed to complete execution (and no new microbatches are injected). As much as 50% of time can be spent flushing the pipeline depending on the number of microbatches injected into the pipeline. The larger the ratio of number of microbatches to the pipeline size, the smaller the time spent in the pipeline flush. Therefore, to achieve high efficiency, a larger batch size is often necessary. In this chapter, we also introduce a new pipeline schedule that improves efficiency at small batch sizes.

Users can thus train their large models using various techniques, each with different tradeoffs. Moreover, these techniques can be *combined*. However, combining these techniques leads to non-trivial interactions, which need to be reasoned through carefully for good performance. In this chapter, we address the following question:

> *How should parallelism techniques be combined to maximize the training throughput of large models given a batch size while retaining strict optimizer semantics?*

In particular, we show how to combine pipeline, tensor, and data parallelism, a technique we call *PTD-P*, to train large language models with good computational performance (52% of peak device throughput) on 1000s of GPUs, which is a much larger scale compared to the scales considered in Chapters 2 and 3. Our method leverages the combination of pipeline parallelism across multi-GPU servers, tensor parallelism within a multi-GPU server, and data parallelism, to practically train models with a trillion parameters with graceful scaling in an optimized cluster environment with high-bandwidth links between GPUs on the same server and across servers. We can use similar ideas to train larger models as well, given more training resources. In our experiments, we demonstrate close to linear scaling to 3072 A100 GPUs, with an achieved end-to-end training throughput of 163 teraFLOP/s per GPU (including communication, data processing, and optimization), and an aggregate throughput of 502 petaFLOP/s, on a GPT model [45] with a trillion parameters using mixed precision. This throughput facilitates practical training times: we estimate end-to-end training of this model to take $\sim$ 3 months. We believe this is the fastest training throughput achieved for this size of model: past systems [153, 125] cannot train such large models since they do not combine pipeline and tensor parallelism. We also compared to ZeRO [140], and found that our approach outperforms ZeRO-3 by 70% for models with 175 and 530 billion parameters due to less cross-node communication. These models are too large to fit on a multi-GPU server.

Achieving this throughput at scale required innovation and careful engineering along multiple axes: efficient kernel implementations that allowed most of the computation to be compute-bound as opposed to memory-bound, smart partitioning of computation graphs over the devices to reduce the number of bytes sent over network links while also limiting device idle periods, domain-specific communication optimization, and fast hardware (state-of-the-art GPUs and high-bandwidth links between GPUs on the same and different servers). We are hopeful that our open-sourced software (available at `https://github.com/nvidia/megatron-lm`) will enable other groups to train large NLP models efficiently at scale.

In addition, we studied the interaction between the various components affecting throughput, both empirically and analytically when possible. Based on these studies, we offer the following guiding principles on how to configure distributed training:

- Different forms of parallelism interact in non-trivial ways: the parallelization strategy has an impact on the amount of communication, the compute efficiency with which kernels are executed, as well as the idle time workers spend waiting for computation due to pipeline flushes (pipeline bubbles). For example, in our experiments, we found that sub-optimal combinations of tensor and pipeline model parallelism can lead to up to $2\times$ lower throughput, even with high-bandwidth network links between servers; tensor model parallelism is effective within a multi-GPU server, but pipeline parallelism must be used for larger models. Moreover, the combination of these parallelization strategies is *necessary* to train models with hundreds of billions to a trillion parameters; these parallelization strategies in isolation are insufficient.

- The schedule used for pipeline parallelism has an impact on the amount of communication, the pipeline bubble size, and memory used to store activations. We propose a novel interleaved schedule that can improve throughput by as much as 10% compared to previously-proposed schedules [86, 127] with comparable memory footprint.
- Values of hyperparameters such as microbatch size have an impact on the memory footprint, the arithmetic efficiency of kernels executed on the worker, and the pipeline bubble size. In our experiments, the optimal value of the microbatch size is problem-dependent and can increase throughput by 15%.
- At scale, distributed training is communication-intensive. When training a trillion-parameter model on 3072 GPUs, our implementation used an effective bisection bandwidth of 892 GB/s for pipeline-parallel communication, and 13 TB/s for data-parallel communication. Using slower inter-node interconnects or more communication-intensive partitionings would hinder scaling performance.

We should note that we do not automatically explore the search space of parallelization strategies (such as FlexFlow [96], PipeDream [125], Tarnawski et al. [159], and DAPPLE [71]), but instead suggest heuristics (in §4.3) that we found work well in practice. Automating this process is interesting future work.

4.2 Modes of Parallelism

In this section, we discuss the parallelism techniques introduced in §2.2 in more detail. These parallelism modes help facilitate the *efficient* training of large models that do not fit in the memory of a single GPU *at scale*. In this chapter, we combine pipeline model parallelism and tensor model parallelism (combination shown in Figure 4.2) with data parallelism. We call this PTD-P for short.

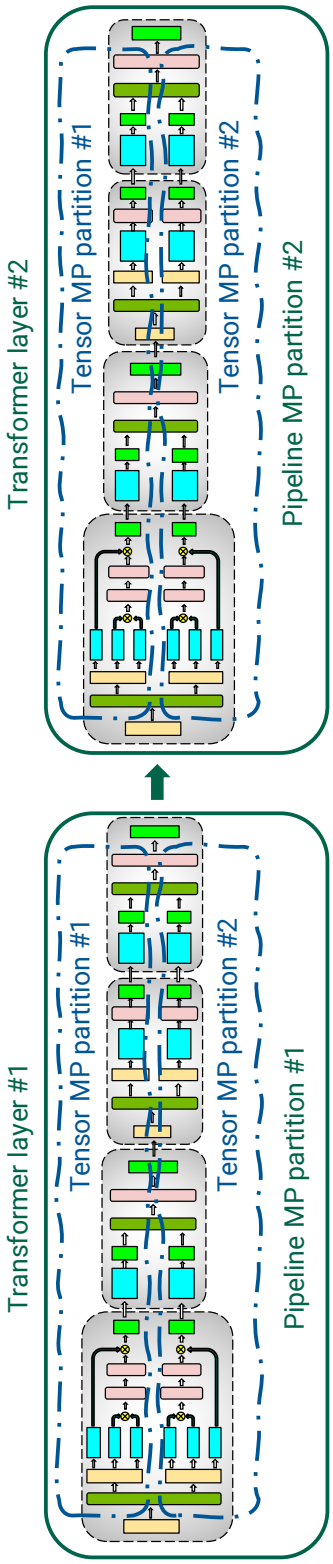


Figure 4.2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.

4.2.1 Data Parallelism

With data parallelism [173, 109], each worker has a copy of the full model, the input dataset is sharded, and workers aggregate their gradients periodically to ensure that all workers see a consistent version of the weights. For large models which do not fit on a single worker, data parallelism can be used on smaller model shards.

4.2.2 Pipeline (Model) Parallelism

With pipeline (model) parallelism[1], the layers of a model are sharded across multiple devices. When used on models with the same transformer block repeated, each device can be assigned an equal number of transformer layers. In this chapter, we do not consider more asymmetric model architectures, where assignment of layers to pipeline stages is harder; we defer to Chapter 2 and related work [96, 159] to solve this problem.

A batch is split into smaller microbatches; execution is then pipelined across microbatches. Pipelining schemes need to ensure that inputs see consistent weight versions across forward and backward passes for well-defined synchronous weight update semantics. Specifically, naïve pipelining can lead to an input seeing weight updates in the backward pass not seen in the forward pass.

To retain strict optimizer semantics *exactly*, we introduce periodic pipeline flushes so that optimizer steps are synchronized across devices. At the start and end of every batch, devices are idle. We call this idle time the *pipeline bubble*, and want to make it as small as possible. Asynchronous and bounded staleness approaches such as PipeMare [175, 99], PipeDream (Chapter 2), and PipeDream-2BW (Chapter 3) do away with flushes completely, but relax weight update semantics. We do not consider the combination of such pipelining schemes with data and tensor model parallelism in this chapter, and instead defer this to future work.

There are several possible ways of scheduling forward and backward microbatches across devices; each approach offers different tradeoffs between pipeline bubble size, communication, and memory footprint. We discuss two such approaches in this section.

Default Schedule

GPipe [86] proposes a schedule where the forward passes for all microbatches in a batch are first executed, followed by backward passes for all microbatches (shown in Figure 4.3). We can quantify the size of GPipe's pipeline bubble (t_{pb}). We denote the number of microbatches in a batch as m, the number of pipeline stages (number of devices used for pipeline parallelism) as p, the ideal time per iteration as t_{id} (assuming ideal scaling), and the time to execute a single microbatch's forward and backward pass as t_f and t_b. In this schedule, the pipeline bubble consists of $p - 1$ forward

[1] We drop the "model" in "pipeline model parallelism" in most places for consistency with other chapters in this dissertation, but we do want to note that pipeline parallelism is an augmented form of model parallelism.

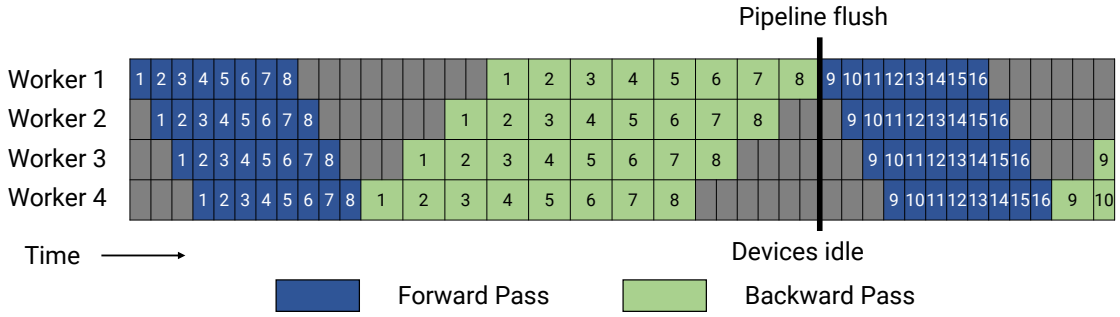


Figure 4.3: GPipe pipeline schedule with forward passes (blue) for all microbatches (represented by numbers) followed by backward passes (green). The gray area represents the pipeline bubble. For simplicity, we assume that the backward pass takes twice as long as the forward pass. The efficiency of the pipeline schedule does not depend on this factor. Each batch in this example consists of 8 microbatches, and the numbers in each blue or green box are unique identifiers given to the corresponding microbatch (in particular, the first batch consists of microbatches $1-8$, and so on). The optimizer is stepped and weight parameters updated at the pipeline flush to ensure strict optimizer semantics, leading to idle devices and a pipeline bubble.

passes at the start of a batch, and $p-1$ backward passes at the end. The total amount of time spent in the pipeline bubble is then $t_{pb} = (p-1) \cdot (t_f + t_b)$. The ideal processing time for the batch is $t_{id} = m \cdot (t_f + t_b)$. Therefore, the fraction of ideal computation time spent in the pipeline bubble is:

$$\text{Bubble time fraction (pipeline bubble size)} = \frac{t_{pb}}{t_{id}} = \frac{p-1}{m}.$$

For the bubble time fraction to be small, we thus need $m \gg p$. However, for such large m, this approach has a high memory footprint as it requires stashed intermediate activations (or just input activations for each pipeline stage when using activation recomputation) to be kept in memory for all m microbatches through the lifetime of a training iteration.

Instead, we use the PipeDream-Flush schedule from the previous chapter. In this schedule, we first enter a warm-up phase where workers perform differing numbers of forward passes as shown in Figure 4.4 (top). This schedule limits the number of in-flight microbatches (the number of microbatches for which the backward pass is outstanding and activations need to be maintained) to the depth of the pipeline, instead of the number of microbatches in a batch. After the warm-up phase, each worker then enters a steady state, where workers perform one forward pass followed by one backward pass (1F1B for short). Finally, at the end of a batch, we complete backward passes for all remaining in-flight microbatches. The time spent in the bubble is the same for this new schedule, but the number of outstanding forward passes is at most the number of pipeline stages for the PipeDream-Flush schedule. As a result, this schedule requires activations to be stashed for p or fewer microbatches (compared to m microbatches for the GPipe schedule). Consequently, when $m \gg p$,

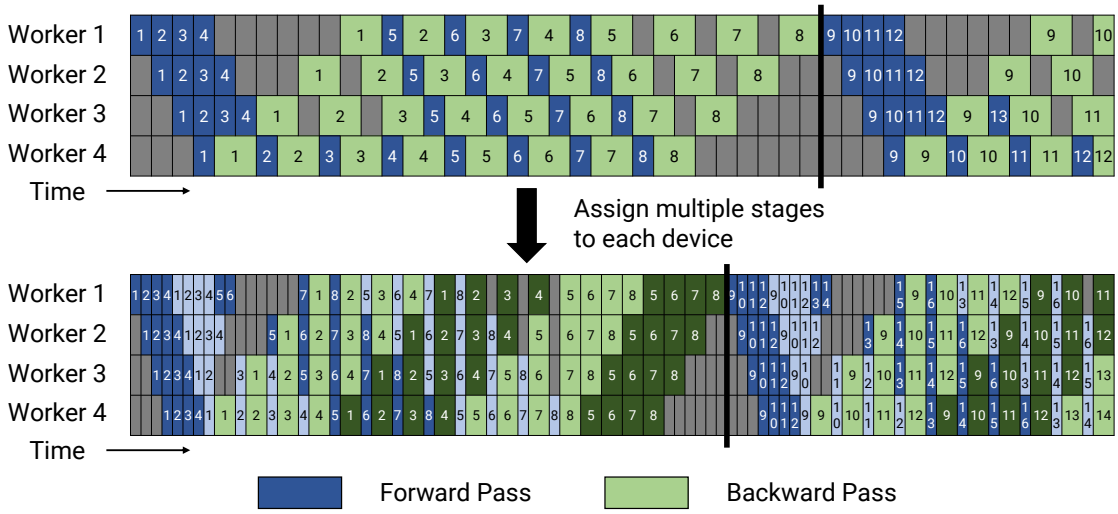


Figure 4.4: Default and interleaved 1F1B pipeline schedules. The top figure shows the default non-interleaved 1F1B schedule. The bottom figure shows the interleaved 1F1B schedule, where each device is assigned multiple chunks (in this case, 2). Dark colors show the first chunk and light colors show the second chunk. The size of the pipeline bubble is smaller (the pipeline flush happens sooner in the interleaved timeline).

PipeDream-Flush is much more memory-efficient than GPipe.

Schedule with Interleaved Stages

To reduce the size of the pipeline bubble, each device can perform computation for multiple subsets of layers (called a model chunk), instead of a single contiguous set of layers. For example, if each device had 4 layers before (i.e., device 1 had layers $1-4$, device 2 had layers $5-8$, and so on), we could have each device perform computation for two model chunks (each with 2 layers), i.e., device 1 has layers $1, 2, 9, 10$; device 2 has layers $3, 4, 11, 12$; and so on. With this scheme, each device in the pipeline is assigned multiple pipeline stages (each pipeline stage has less computation compared to before).

As before, we can use an "all-forward, all-backward" version of this schedule, but this has a high memory footprint (proportional to m). Instead, we developed an interleaved schedule that adapts the more memory-efficient 1F1B schedule from before. This new schedule is shown in Figure 4.4, and requires the number of microbatches in a batch to be an integer multiple of the degree of pipeline parallelism (number of devices in the pipeline). For example, with 4 devices, the number of microbatches in a batch must be a multiple of 4.

As shown in Figure 4.4, the pipeline flush for the same batch size happens sooner in the new schedule. If each device has v stages (or model chunks), then the forward and backward time for a microbatch for each stage or chunk will now be t_f/v and t_b/v. The pipeline bubble time thus

reduces to $t_{pb}^{\text{int.}} = \frac{(p-1)\cdot(t_f+t_b)}{v}$, and the bubble time fraction is then:

$$\text{Bubble time fraction (pipeline bubble size)} = \frac{t_{pb}^{\text{int.}}}{t_{id}} = \frac{1}{v}\cdot\frac{p-1}{m}.$$

This means that the new schedule reduces the bubble time by v. This reduced pipeline bubble size, however, does not come for free: this schedule requires extra communication. Quantitatively, the amount of communication also increases by v. In the next section, we discuss how we can utilize the 8 InfiniBand networking cards in a multi-GPU server (e.g., a DGX A100 node) to reduce the impact of this extra communication.

4.2.3 Tensor Model Parallelism

With tensor model parallelism, individual layers of the model are partitioned over multiple devices. We use the particular partitioning strategy used by Megatron [153] for transformer layers, the bedrock of language models. We can apply similar ideas to other types of models, like CNNs, as well. We briefly outline this strategy, illustrated in Figure 4.5, below.

A transformer layer consists of a self-attention block followed by a two-layer multi-layer perceptron (MLP). Further details of the transformer layer can be found in Vaswani et al [164].

The MLP block consists of two GEMMs and a GeLU non-linearity:

$$Y = \text{GeLU}(XA),\ \ Z = \text{Dropout}(YB).$$

We can split A along its columns $A = [A_1, A_2]$. This partitioning allows the GeLU non-linearity to be independently applied to the output of each partitioned GEMM:

$$[Y_1, Y_2] = [\text{GeLU}(XA_1), \text{GeLU}(XA_2)].$$

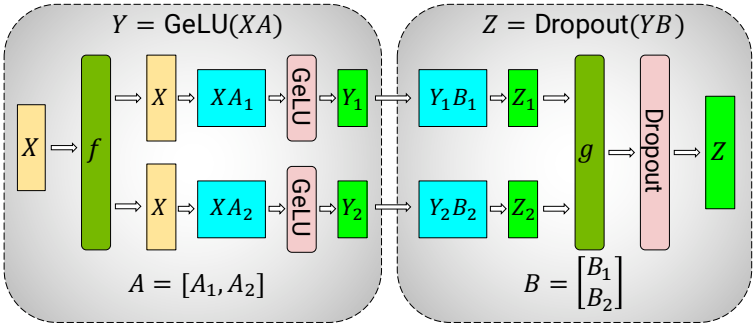
This is advantageous as it removes the need for synchronization (needed if A is split along its rows since GeLU is non-linear).

The rows of the second weight matrix B can then be split along its rows to remove the need for any communication between the GEMMs (shown in Figure 4.5a), as shown below:

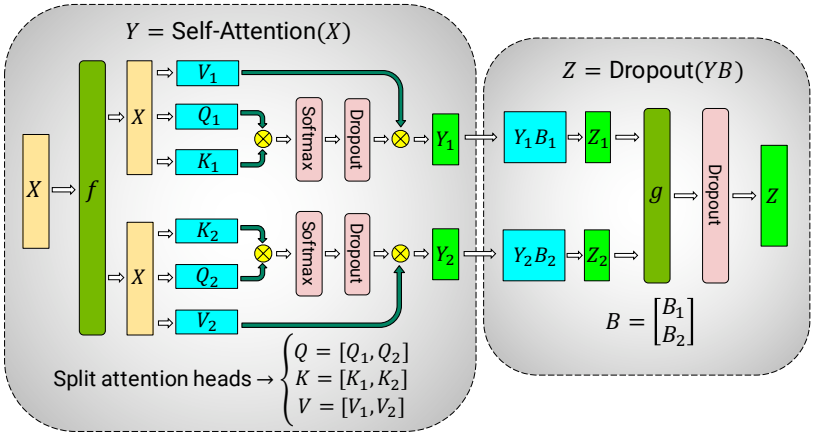
$$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix},\ Y = [Y_1, Y_2].$$

The output of the second GEMM is then reduced across the GPUs before the dropout layer.

We exploit the inherent parallelism in the multi-head attention operation to partition the self-attention block (shown in Figure 4.5b). The key (K), query (Q), and value (V) matrices can be partitioned in a column-parallel fashion. The output linear layer can then directly operate on the



(a) MLP.



(b) Self-Attention.

Figure 4.5: Blocks of transformer model partitioned with tensor model parallelism (figures borrowed from Megatron [153]). f and g are conjugate. f is the identity operator in the forward pass and all-reduce in the backward pass, while g is the reverse.

partitioned output of the attention operation (weight matrix partitioned across rows).

This approach splits GEMMs in the MLP and self-attention blocks across GPUs while requiring only two all-reduce operations in the forward pass (g operator) and two all-reduces in the backward pass (f operator). We implemented f and g in a few lines of code.

4.3 Performance Analysis of Parallelization Configurations

In this section, we consider the performance implications of combining pipeline and tensor model parallelism with data parallelism. Given a fixed budget of GPUs and batch size, one can use different degrees of the parallelism types in PTD-P to train models; each dimension exposes tradeoffs between memory footprint, device utilization, and amount of communication.

We discuss these tradeoffs in the rest of this section, and then show empirical results in §4.5.4.

We present analytical models where relevant for the pipeline bubble size. We qualitatively describe how communication time behaves and present cost models for amount of communication; however, we do not present direct cost models for communication time, which is harder to model for a hierarchical network topology where interconnects between GPUs on the same server have higher bandwidth than interconnects between servers. To the best of our knowledge, this is the first work to analyze the performance *interactions* of these parallelization dimensions.

4.3.1 Notation

We use the following notation in this section:

- (p, t, d): Parallelization dimensions. p for the pipeline-model-parallel size, t for the tensor-model-parallel size, and d for the data-parallel size.
- n: Number of GPUs. We require $p \cdot t \cdot d = n$.
- B: Global batch size (provided as input).
- b: Microbatch size.
- $m = \frac{1}{b} \cdot \frac{B}{d}$: Number of microbatches in a batch *per pipeline*.

4.3.2 Tensor and Pipeline Model Parallelism

Tensor and pipeline model parallelism can both be used to partition a model's parameters over multiple GPUs. As stated earlier, using pipeline parallelism with periodic flushes results in a pipeline bubble of size $(p-1)/m$. Let us assume that $d = 1$ (data-parallel size); consequently, $t \cdot p = n$. The pipeline bubble size in terms of t is:

$$\frac{p-1}{m} = \frac{n/t - 1}{m}.$$

As t increases, the pipeline bubble thus decreases for fixed B, b, and d ($m = B/(b \cdot d)$ is fixed).

The amount of communication performed between different GPUs is also affected by the values of p and t. Pipeline parallelism features cheaper point-to-point communication. Tensor model parallelism, on the other hand, uses all-reduce communication (two all-reduce operations each in the forward and backward pass, see §4.2.3). With pipeline parallelism, the total amount of communication that needs to be performed between every pair of consecutive devices (for either the forward or backward pass) per microbatch is bsh, where s is the sequence length and h is the hidden size. With tensor model parallelism, tensors of total size bsh need to be all-reduced among t model replicas twice each in the forward and backward pass for each layer, leading to a total communication of $8bsh\left(\frac{t-1}{t}\right)$ *per layer* per device for each microbatch. Each device typically has multiple layers; the total amount of tensor-parallel-communication is then $l^{\text{stage}} \cdot \left(8bsh\left(\frac{t-1}{t}\right)\right)$, where l^{stage} is the number of layers in a pipeline stage.

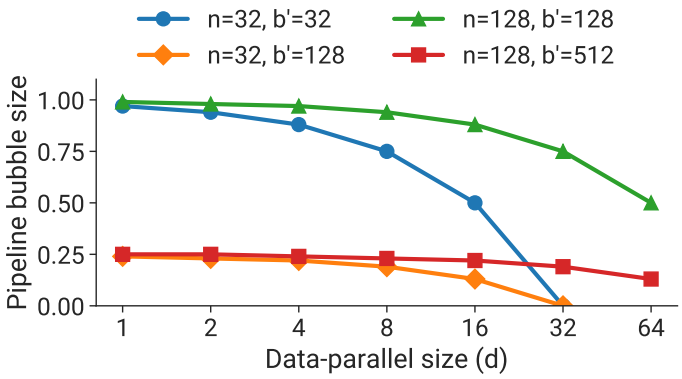


Figure 4.6: Fraction of time spent in a pipeline flush (pipeline bubble size) versus data-parallel size (d), for different numbers of GPUs (n) and ratio of batch size to microbatch size ($b' = B/b$).

Consequently, we see that tensor model parallelism increases the amount of communication between devices. Thus, when t is larger than the number of GPUs in a single node, the overhead of performing tensor model parallelism across slower inter-node links can be impractical. We see these results empirically in §4.5.4.

Takeaway #1: When considering different forms of model parallelism, tensor model parallelism should generally be used up to degree g when using g-GPU servers, and then pipeline parallelism can be used to scale up to larger models across servers.

4.3.3 Data and Model Parallelism

We also want to consider the interaction between data parallelism and the two types of model parallelism. In this section, we consider these interactions independently for simplicity.

Pipeline Parallelism

Let $t = 1$ (tensor-model-parallel size). The number of microbatches per pipeline is $m = B/(d \cdot b) = b'/d$, where $b' := B/b$. With total number of GPUs n, the number of pipeline stages is $p = n/(t \cdot d) = n/d$. The pipeline bubble size is:

$$\frac{p-1}{m} = \frac{n/d - 1}{b'/d} = \frac{n-d}{b'}.$$

As d becomes larger, $n-d$ becomes smaller, and thus the pipeline bubble becomes smaller. Figure 4.6 shows the behavior of the pipeline bubble size for various values of d, n, and b'. It might not be possible to increase d all the way to n for all models, since a model's full training memory footprint might be larger than the memory capacity of a single accelerator.

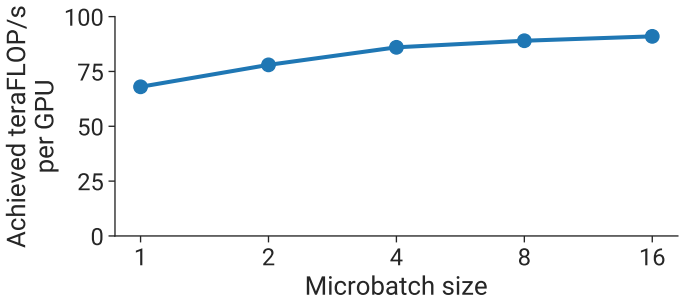


Figure 4.7: Per-GPU throughput versus microbatch size for a GPT model with a billion parameters (128 attention heads, hidden size of 4096, 4 transformer layers).

Overall throughput will thus increase if the all-reduce communication needed for data parallelism does not drastically increase with higher d, which should hold since the communication time for a ring-based implementation scales with $\frac{d-1}{d} = 1 - \frac{1}{d}$.

We can also analyze the impact of increasing the batch size B. For a given parallel configuration, as the batch size B increases, $b' = B/b$ increases, $(n-d)/b'$ decreases, consequently increasing throughput. All-reduce communication required by data parallelism also becomes more infrequent, further increasing throughput.

Data and Tensor Model Parallelism

With tensor model parallelism, all-reduce communication needs to be performed for *every* microbatch. This can be expensive across multi-GPU servers. On the other hand, data parallelism only needs to perform expensive all-reduce communication *once per batch*. Moreover, with tensor model parallelism, each model-parallel rank performs a subset of the computation in each model layer, and thus for insufficiently-large layers, modern GPUs might not perform these sub-matrix computations with peak efficiency.

Takeaway #2: When using data and model parallelism, a total model-parallel size of $M = t \cdot p$ should be used so that the model's parameters and intermediate metadata fit in GPU memory; data parallelism can be used to scale up training to more GPUs.

4.3.4 Microbatch Size

The choice of the microbatch size b also affects model-training throughput. For example, we see in Figure 4.7 that per-GPU throughput increases by up to $1.3\times$ with a larger microbatch size on a single GPU. We now want to determine the optimal microbatch size b given a parallel configuration (p, t, d) and batch size B. The amount of data-parallel communication will be the same regardless of the microbatch size. Given functions $t_f(b)$ and $t_b(b)$ that map the microbatch size to the forward

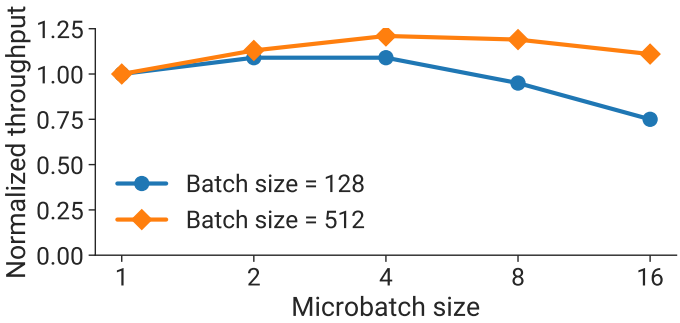


Figure 4.8: Behavior of normalized estimated throughput (time computed as $t = (b'/b + p - 1) \cdot (t_f(b) + t_b(b))$) with respect to the microbatch size b for the same GPT model from Figure 4.7.

and backward computation times for a single microbatch, the total time spent computing a batch, ignoring communication cost, is (as before, define b' as B/d):

$$(b'/b + p - 1) \cdot (t_f(b) + t_b(b)) \,. \tag{4.1}$$

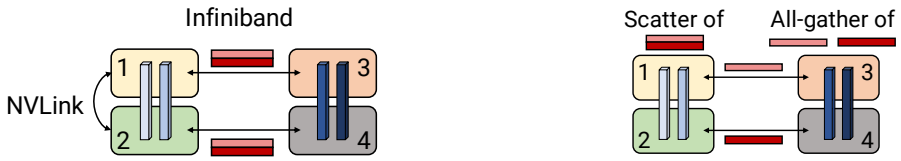
The microbatch size thus affects both the arithmetic intensity of operations as well as the pipeline bubble size (by affecting m). Figure 4.8 shows estimated throughput (equation (4.1) used to estimate processing time) for a GPT model with a billion parameters and $(p, t) = (8, 8)$. The optimal b for both batch sizes is 4.

Takeaway #3: The optimal microbatch size b depends on the throughput and memory footprint characteristics of the model, as well as the pipeline depth p, data-parallel size d, and batch size B.

4.3.5 Activation Recomputation

Activation recomputation [86, 53, 77, 90] is an optional technique that trades off an increase in the number of compute operations performed for additional memory footprint, by running the forward pass a second time just before the backward pass (and stashing only the input activations for a given pipeline stage, as opposed to the entire set of intermediate activations, which is much larger). Activation recomputation is required to train reasonably large models with pipeline parallelism to keep memory footprint acceptably low. Chapter 3 briefly looked at the performance ramifications of activation recomputation.

The number of activation checkpoints does not impact throughput, but impacts memory footprint. Let A^{input} be the size of the input activations of a layer, and $A^{\text{intermediate}}$ be the size of intermediate activations per layer. If a model stage has l layers, and if c is the number of checkpoints, the total memory footprint is going to be $c \cdot A^{\text{input}} + l/c \cdot A^{\text{intermediate}}$. The minimum value of this function is obtained when $c = \sqrt{l \cdot (A^{\text{intermediate}}/A^{\text{input}})}$. In practice, we measure $A^{\text{intermediate}}$ empirically. For most cases, checkpointing every 1 or 2 transformer layers is optimal.



(a) W/o scatter/gather optimization.

(b) With scatter/gather optimization.

Figure 4.9: Scatter/gather communication optimization. Light blue blocks are layers in the first pipeline stage, and dark blue blocks are layers in the second pipeline stage. Without the scatter/-gather optimization, the same tensor is sent redundantly over inter-node InfiniBand links. Instead, at the sender, we can scatter the tensor into smaller chunks, reducing the sizes of tensors sent over InfiniBand links. The final tensor can then be rematerialized at the receiver using a gather operation.

Other techniques such as activation partitioning [140] can also be used in conjunction with tensor model parallelsim to reduce the memory footprint due to activations further.

4.4 Implementation

We implemented PTD-P as an extension to the Megatron-LM codebase. Our implementation is built using PyTorch [134]. We use NCCL [18] for communication between devices. To obtain good performance, we implemented optimizations targeting both communication and computation, which we outline below.

4.4.1 Communication Optimizations

When using pipeline parallelism, we want to send and receive tensors in the forward and backward direction in parallel. Each DGX A100 is equipped with 8 InfiniBand (IB) networking cards. Unfortunately, sends and receives are point-to-point, and only happen between a pair of GPUs on two servers, making it hard to leverage all 8 cards for a single communication call within the pipeline.

However, we can leverage the fact that we use both tensor model parallelism and pipeline parallelism to reduce the overhead of cross-node communication. In particular, we note that the output of each transformer layer is replicated (after g in MLP block, see Figure 4.5a) across the tensor-parallel ranks. As a result, ranks in two consecutive pipeline stages that are performing tensor model parallelism send and receive the exact same set of tensors (Figure 4.9a).

For large enough models, we use a tensor-model-parallel size of 8. This means we are sending the same set of tensors 8 times between corresponding GPUs on adjacent multi-GPU servers. To reduce this redundancy, we can instead split the tensor on the send side into equal-sized chunks, and then only send one chunk to the corresponding rank on the next node using the rank's own InfiniBand card (e.g., rank 1 sends to rank 3 and rank 2 sends to rank 4 in Figure 4.9). With 8

tensor-model-parallel ranks, each chunk would be one-eighth smaller. Then, on the receive side, we can perform an all-gather over NVLink, which is much faster than the InfiniBand interconnect, to re-materialize the full tensor. This is shown in Figure 4.9b. We call this the *scatter/gather communication optimization*. This optimization helps better leverage the multiple IB cards on the DGX A100 servers, and makes more communication-intensive schedules such as the interleaved one feasible.

Quantitatively, with the scatter-gather communication optimization, the total amount of communication that needs to be performed between every pair of consecutive stages is reduced to $\frac{bsh}{t}$, where t is the tensor-model-parallel size, s is the sequence length, and h is the hidden size ($t = 8$ in our experiments).

4.4.2 Computation Optimizations

We implemented three model-specific optimizations to the computation graph to attain high performance. First, we changed the data layout in the transformer layer to avoid memory-intensive transpose operations, and to enable the use of strided batched GEMM kernels. Specifically, we changed the data layout from $[b, s, a, h]$ to $[s, b, a, h]$, where b, s, a, and h are batch, sequence, attention-head, and hidden-size dimensions, respectively. Second, we generated fused kernels for a sequence of element-wise operations (bias + GeLU and bias + dropout + add) using PyTorch JIT [25]. Third, we created two custom kernels to enable the fusion of scale, mask, and softmax (reduction) operations: one to support general masking (used in models such as BERT) and another to support implicit causal masking (used in auto-regressive models such as GPT). We quantify the effect of these optimizations in the next section.

4.5 Evaluation

In this section, we seek to answer the following questions:

- How well does PTD-P perform? Does it result in realistic end-to-end training times?
- How well does pipeline parallelism scale for a given model and batch size? How much impact does the interleaved schedule have on performance?
- How do different parallelization dimensions interact with each other? What is the impact of hyperparameters such as microbatch size?
- What is the impact of the scatter-gather communication optimization? What types of limits do we put on hardware when running training iterations at scale?

All of our results are run with mixed precision on the Selene supercomputer [21]. Each cluster node has 8 NVIDIA 80-GB A100 GPUs [17], connected to each other by NVLink and NVSwitch [22].

Each node has eight NVIDIA Mellanox 200Gbps HDR Infiniband HCAs for application communication, with an additional two HCAs per node for dedicated storage. The nodes are connected in a three-level (leaf, spine, core) fat-tree topology with 850 switches. This topology allows efficient all-reduce communication (dominant communication pattern in deep learning training). The cluster uses an all-NVME shared parallel filesystem for high-performance data access and storage. The peak device throughput of an A100 GPU with 16-bit precision is 312 teraFLOP/s. For most of our results, we report throughput per GPU. Aggregate throughput can be computed by multiplying with the number of GPUs used.

For our experiments, we use GPT models of appropriate sizes. In particular, for any given microbenchmark, the model needs to fit on the number of model-parallel GPUs used in the experiment. We use standard model architectures such as GPT-3 [45] when appropriate.

4.5.1 End-to-End Performance

Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7	24	2304	24	1	1	32	512	137	44%	4.4
3.6	32	3072	30	2	1	64	512	138	44%	8.8
7.5	32	4096	36	4	1	128	512	142	46%	18.2
18.4	48	6144	40	8	1	256	1024	135	43%	34.6
39.1	64	8192	48	8	2	512	1536	138	44%	70.8
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0

Table 4.1: Weak-scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

We consider the end-to-end performance of our system on GPT models ranging from a billion to a trillion parameters, using tensor, pipeline, and data parallelism (degrees picked using heuristics described in §4.3). In particular, we use the interleaved pipeline schedule with the scatter/gather optimization enabled.

We consider a language model with l transformer layers, hidden size h, sequence length s, vocabulary size V, and training batch size B.

A $A_{m\times k} \times X_{k\times n}$ matrix multiplication requires $2m \times k \times n$ FLOPs (factor of 2 needed to account for multiplies and adds).

A transformer layer consists of an attention block followed by a 2-layer feed-forward network. For the attention block, the main FLOP contributors are the key, query, and value transformation ($6Bsh^2$ operations), attention matrix computation ($2Bs^2h$ operations), attention over values ($2Bs^2h$ operations), and post-attention linear projection ($2Bsh^2$ operations). The feed-forward network increases the hidden size to $4h$ and then reduces it back to h; this requires $16Bsh^2$ FLOPs. Summing these together, each transformer layer results in $24Bsh^2 + 4Bs^2h$ FLOPs for the forward pass. The backward pass requires double the number of FLOPs since we need to calculate the gradients with respect to both input and weight tensors. In addition, we are using activation recomputation, which requires an additional forward pass before the backward pass. As a result, the total number of FLOPs per transformer layer is $4 \times \left(24Bsh^2 + 4Bs^2h\right) = 96Bsh^2\left(1 + \frac{s}{6h}\right)$.

The other main contributor to the FLOP count is the logit layer in the language model head, which transforms features of dimension h to the vocabulary dimension V. The required FLOPs for this operation is $2BshV$ in the forward pass and $4BshV$ in the backward pass, resulting in $6BshV$ FLOPs in total.

For a transformer model with l transformer layers, the number of floating-point operations is:

$$F = 96Bslh^2\left(1 + \frac{s}{6h} + \frac{V}{16lh}\right). \tag{4.2}$$

This is a lower bound for the true FLOP count but should be close to the actual value. We count a FLOP as a floating-point operation regardless of precision. We also note that equation 4.2 assumes activation recomputation and takes into account the floating-point operations associated with the extra forward pass.

The number of parameters in a model, P, can be computed as:

$$P = 12lh^2\left(1 + \frac{13}{12h} + \frac{V+s}{12lh}\right). \tag{4.3}$$

All models use a vocabulary size (V) of 51,200 (multiple of 1024) and a sequence length (s) of 2048. As the model size increases, we also increase the number of GPUs (n).

Table 4.1 shows the model configurations along with the achieved FLOP/s (both per GPU and

Scheme	Number of parameters (billion)	Model-parallel size	Batch size	Number of GPUs	Microbatch size	Achieved teraFlOP/s per GPU	Training time for 300B tokens (days)
ZeRO-3 without Model Parallelism	174.6	1	1536	384	4	144	90
				768	2	88	74
				1536	1	44	74
	529.6	1	2560*	640	4	138	169
			2240	1120	2	98	137
				2240	1	48	140
PTD Parallelism	174.6	96	1536	384	1	153	84
				768	1	149	43
				1536	1	141	23
	529.6	280	2240	560	1	171	156
				1120	1	167	80
				2240	1	159	42

Table 4.2: Comparison of PTD Parallelism to ZeRO-3 (without model paralllelism). The 530-billion-parameter GPT model did not fit on 560 GPUs when using a microbatch size of 4 with ZeRO-3, so we increased the number of GPUs used to 640 and global batch size to 2560 to provide a throughput estimate (relevant row marked in table with a *).

aggregate over all GPUs). We see super-linear scaling to 3072 A100 GPUs (384 DGX A100 nodes), since GPU utilization improves as the models get larger (larger matrix multiplications) without significant increase in the communication time relative to computation time. Note that throughput is measured for end-to-end training, i.e., includes all operations including data loading, optimizer steps, communication, and logging. We achieve 52% of peak device throughput for the largest model, and 44% of peak device throughput for the smallest model.

Training Time Estimates. Given these throughputs, we can estimate the total amount of time needed for end-to-end training on T tokens. Training requires $I = T/\left(B \cdot s\right)$ iterations. Using the value of F from equation 4.2 and empirical end-to-end throughputs from Table 4.1 (X), we can estimate total training time. We note that for the configurations in Table 4.1, we have $6h \gg s$, $16lh \gg (V + s)$, and $12lh \gg V$. Combining these observations with equations 4.3 and 4.2:

$$\text{End-to-end training time} \approx \frac{8TP}{nX}. \tag{4.4}$$

Let us consider the GPT-3 model with P =175 billion parameters as an example. This model was trained on $T = 300$ billion tokens. On $n = 1024$ A100 GPUs using batch-size 1536, we achieve $X = 140$ teraFLOP/s per GPU. As a result, the time required to train this model is 34 days. For the 1 trillion parameter model, we assume that 450 billion tokens are needed for end-to-end training. With 3072 A100 GPUs, we can achieve a per-GPU throughput of 163 teraFLOP/s, and training time of 84 days. We believe these training times (using a reasonable number of GPUs) are practical.

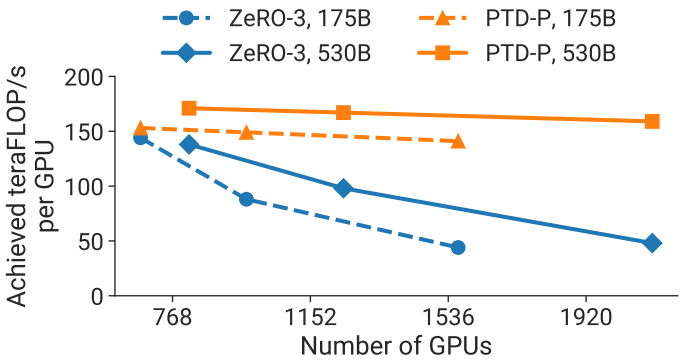


Figure 4.10: Throughput per GPU of PTD-P and ZeRO-3 for two different GPT models (the 175B GPT-3 model is shown with dotted lines, and the 530B model is shown with solid lines). Global batch sizes are fixed and ZeRO-3 is used without any model parallelism.

4.5.2 Comparison to ZeRO-3

We compare PTD-P to ZeRO-3 [140, 141] in Table 4.2 and Figure 4.10 for the standard GPT-3 model architecture, as well as the 530-billion-parameter model from Table 4.1. The results provide a point of comparison to a method that does not use model parallelism. We integrated ZeRO into our codebase using the `DeepSpeed` Python library [6]. We keep the global batch size the same as we increase the number of GPUs. With fewer GPUs and a microbatch size of 4, PTD-P results in 6% and 24% higher throughput for the 175- and 530-billion-parameter models respectively. As we increase the number of GPUs, PTD-P scales more gracefully than ZeRO-3 in isolation (see Figure 4.10). For example, by doubling the number of GPUs (keeping the batch size the same), PTD-P outperforms ZeRO-3 by 70% for both models due to less cross-node communication. We note that we have only considered ZeRO-3 without tensor parallelism. ZeRO-3 can be combined with model parallelism to potentially improve its scaling behavior.

4.5.3 Pipeline Parallelism

We now evaluate the weak-scaling performance of pipeline parallelism in isolation, and also compare the performance of the non-interleaved schedule to the interleaved schedule.

Weak Scaling

We evaluate the scaling of the default non-interleaved pipeline-parallel schedule using a weak scaling setup, a GPT model with 128 attention heads and a hidden size of 20480, and a microbatch size of 1. As we increase the number of pipeline stages, we also increase the size of the model by proportionally increasing the number of layers in the model, e.g., with a pipeline-parallel size of 1, we use a model with 3 transformer layers and 15 billion parameters, and with a pipeline-parallel

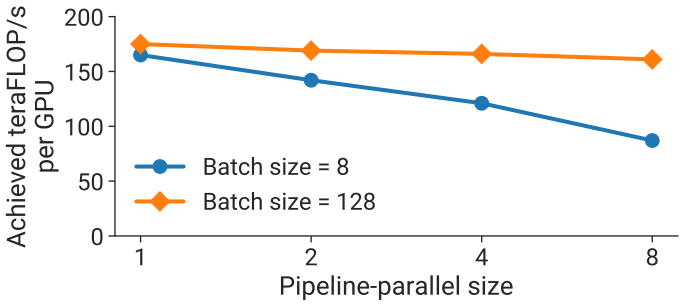


Figure 4.11: Throughput per GPU of pipeline parallelism using two different batch sizes in a weak-scaling experiment setup (model size increases with the pipeline-parallel size).

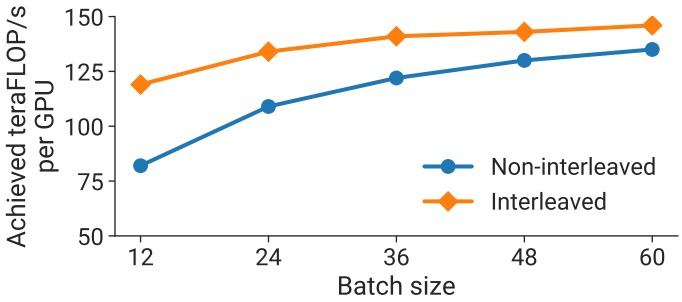


Figure 4.12: Throughput per GPU of interleaved and non-interleaved schedules for a GPT model (175 billion parameters) on 96 GPUs.

size of 8, we use a model with 24 transformer layers and 121 billion parameters. We use a tensor-parallel size of 8 for all configurations, and vary the total number of A100 GPUs used from 8 to 64. Figure 4.11 shows throughput per GPU for two different batch sizes to illustrate the impact of the pipeline bubble, which behaves as $\frac{p-1}{m}$ (§4.2.2). As expected, the higher batch size scales better since the pipeline bubble is amortized over more microbatches.

Interleaved versus Non-Interleaved Schedule

Figure 4.12 shows the per-GPU-throughput for interleaved and non-interleaved schedules on the GPT-3 [45] model with 175 billion parameters (96 layers, 96 attention heads, hidden size of 12288). The interleaved schedule with the scatter/gather communication optimization has higher computational performance than the non-interleaved (default) schedule. This gap closes as the batch size increases due to two reasons:

1. As the batch size increases, the bubble size in the default schedule decreases.
2. The amount of point-to-point communication within the pipeline is proportional to the batch size, and consequently the non-interleaved schedule catches up as the batch size increases (the

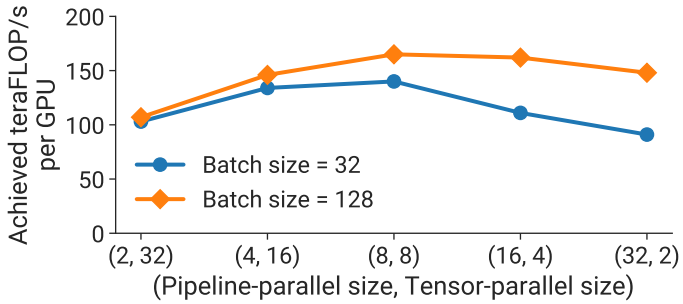


Figure 4.13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

interleaved schedule features more communication per sample).

Without the scatter/gather optimization, the default schedule performs better than the interleaved schedule at larger batch sizes (not shown).

4.5.4 Comparison of Parallel Configurations

In this sub-section, we show the various tradeoffs associated with combining different parallelization dimensions. In particular, we show the performance for parallel configurations using the same number of GPUs for a given model and multiple batch sizes.

Tensor versus Pipeline Parallelism

We evaluate the impact of pipeline and tensor model parallelism on performance for a given model and batch size. The empirical results in Figure 4.13 show the importance of using both tensor and pipeline model parallelism in conjunction to train a 161-billion-parameter GPT model (32 transformer layers to support pipeline-parallel size of 32, 128 attention heads, hidden size of 20480) with low communication overhead and high compute resource utilization. We observe that tensor model parallelism is best within a node (DGX A100 server) due to its multiple expensive all-reduce communication calls. Pipeline parallelism, on the other hand, features much less communication. However, with pipeline parallelism, significant time can be spent in the pipeline bubble: the total number of pipeline stages should thus be limited so that the number of microbatches in the pipeline is a reasonable multiple of the number of pipeline stages. Consequently, we see peak performance when the tensor-parallel size is equal to the number of GPUs in a single node (8 with DGX A100 nodes). This result indicates that neither tensor model parallelism (used by Megatron [153]) nor pipeline parallelism (used by PipeDream [127] and others) in isolation can match the performance of using both techniques in conjunction.

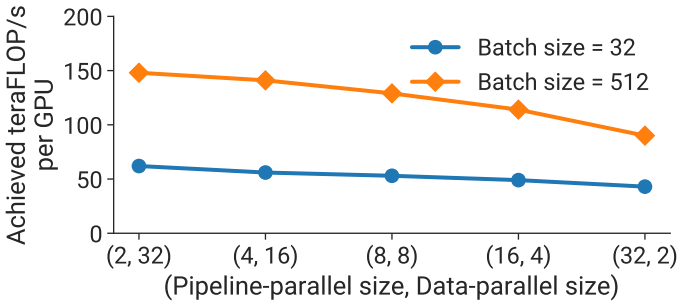


Figure 4.14: Throughput per GPU of various parallel configurations that combine data and pipeline parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

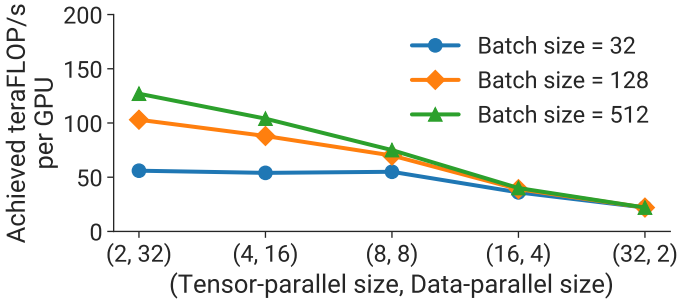


Figure 4.15: Throughput per GPU of various parallel configurations that combine data and tensor model parallelism using a GPT model with 5.9 billion parameters, three different batch sizes, microbatch size of 1, and 64 A100 GPUs.

Pipeline versus Data Parallelism

We evaluate the impact of data and pipeline parallelism on performance for a GPT model with 5.9 billion parameters (32 transformer layers, 32 attention heads, hidden size of 3840) in Figure 4.14. We use a smaller model than before since we want to show performance for models that fit when the model-parallel size is only 2. For simplicity, we keep the microbatch size equal to 1 in these experiments. We see that for each batch size, the throughput decreases as the pipeline-parallel size increases, matching our analytical model from §4.3.3. Pipeline parallelism should be used primarily to support the training of large models that do not fit on a single worker, and data parallelism should be used to scale up training.

Tensor versus Data Parallelism

We also evaluate the impact of data and tensor model parallelism on performance for the same GPT model with 5.9 billion parameters in Figure 4.15 (smaller model used for same reason as

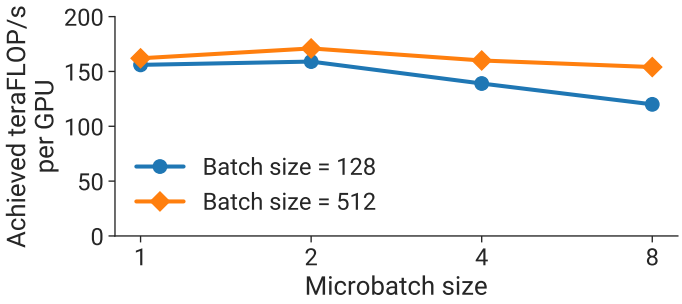


Figure 4.16: Throughput per GPU for different microbatch sizes on a GPT model with 91 billion parameters, for two different batch sizes using 64 A100 GPUs ((t, p) is (8, 8)).

above). As before, we keep the microbatch size equal to 1 initially. With larger batch sizes and a microbatch size of 1, data-parallel communication is infrequent; the all-to-all communication required in tensor model parallelism needs to be performed for *every* microbatch in a batch. This all-to-all communication with tensor model parallelism dominates end-to-end training time, especially when communication needs to be performed across multi-GPU nodes. Additionally, as the tensor-model-parallel size increases, we perform smaller matrix multiplications on every GPU, decreasing utilization on each GPU.

We should note that although data parallelism can lead to efficient scaling, we cannot use data parallelism in isolation for very large models with a limited training batch size because of:

- Insufficient memory capacity.
- Scaling limitations of data parallelism (e.g., GPT-3 was trained to convergence with a batch size of 1536. Data parallelism thus supports parallelization to only 1536 GPUs; however, roughly $10,000$ GPUs were used to train this model in a reasonable amount of time).

4.5.5 Microbatch Size

We evaluate the impact of the microbatch size on the performance of parallel configurations that combine pipeline and tensor model parallelism in Figure 4.16 for a model with 91 billion parameters ((t, p) is (8, 8)). We see that the best microbatch size is 2 for this model; the optimal microbatch size is different for other models (not shown in Figure) and *model-dependent*. For a given batch size, increasing the microbatch size decreases the number of microbatches in the pipeline (m), leading to a larger pipeline bubble; however, increasing the microbatch size can also improve GPU utilization by increasing the arithmetic intensity of executed kernels. These two factors are at odds with each other, which makes the choice of optimal microbatch size challenging. Our analytical model from

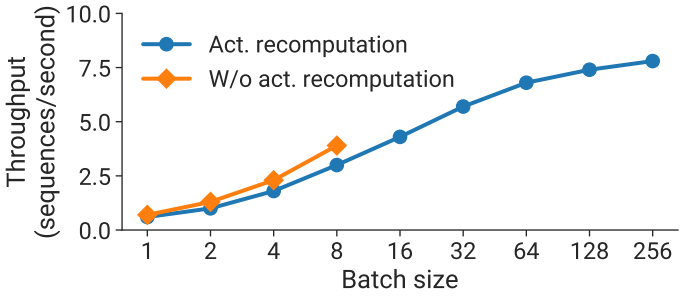


Figure 4.17: Throughput (in sequences per second) with and without activation recomputation for a GPT model with 145 billion parameters using 128 A100 GPUs ((t, p) is (8, 16)).

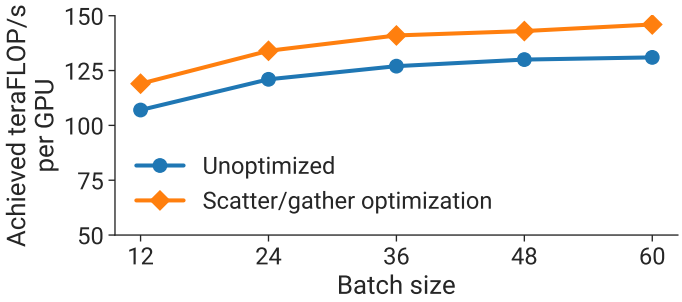


Figure 4.18: Throughput per GPU with and without the scatter/gather optimization for a GPT model with 175 billion parameters using 96 A100 GPUs and the interleaved schedule.

§4.3.3 reasonably approximates true performance, and can be used as a proxy to determine how to pick this hyperparameter value for various models and training configurations.

4.5.6 Activation Recomputation

Figure 4.17 shows throughput with and without activation recomputation for a GPT model with 145 billion parameters (80 transformer layers, 96 attention heads, hidden size of 12288) using 128 A100 GPUs, $(t, p) = (8, 16)$, and a range of batch sizes. For small batch sizes, activation recomputation leads to up to 33% lower throughput (in sequences per second) due to the extra forward pass that needs to be executed during the backward pass. However, activation recomputation is needed to support larger batch sizes. Throughput at large batch sizes with activation recomputation is up to $2\times$ higher than the best throughput achieved without activation recomputation (for a smaller batch size) due to a smaller pipeline bubble.

4.5.7 Scatter-Gather Communication Optimization

Figure 4.18 shows per-GPU-throughput with and without (unoptimized) the scatter/gather communication optimization for the GPT-3 model with 175 billion parameters. We see an improvement of up to 11% in throughput for communication-intensive schedules (large batch size with interleaving) by reducing the amount of communication over cross-node links.

4.5.8 Fused Operators

We also evaluate the performance impact of operator fusion described in §4.4.2. For the GPT-3 model (175 billion parameters), throughput increased by 19% with fusion (113 teraFLOP/s per GPU to 135 teraFLOP/s per GPU). For the larger GPT model with 530 billion parameters (model configuration in Figure 4.1), throughput increased by 11% (133 teraFLOP/s per GPU to 148 teraFLOP/s per GPU).

4.5.9 Inter-Node Communication Bandwidth

Our strong results are a byproduct of using an optimized software and hardware stack *together*. In particular, we take advantage of the high-bandwidth communication links between GPUs on the same server and across servers. On the trillion-parameter model with 3072 GPUs, we observed that the effective bisection bandwidth of point-to-point communication among pipeline stages is 892 GB/s, while the effective bisection bandwidth of all-reduce operations among data-parallel replicas is 12.9 TB/s. A less-optimized partitioning of operators across devices would lead to more inter-node communication, hampering scaling performance.

4.5.10 Checkpoint Loading and Saving

An important practical consideration for the training of large models is loading and saving model checkpoints, which are especially large for the models considered in this evaluation. For example, the trillion-parameter model has a checkpoint of size 13.8 terabytes. The initial load of checkpoints for the trillion-parameter model by all 384 nodes (3072 GPUs) reaches a peak read bandwidth of 1TB/s, the maximum read throughput possible from the parallel filesystem. Checkpoint saves reach 40% of peak write bandwidth (273 GB/s).

4.6 Related Work

In this section, we discuss other techniques to train models at scale.

Parallelism for Large Models. Pipeline model parallelism is a common technique used to train large models. Pipeline parallelism comes in a few flavors: the mode discussed in this chapter uses

flushes to ensure *strict* optimizer semantics. TeraPipe [110] exposes fine-grained pipeline parallelism across tokens in a single training sequence for auto-regressive models like GPT. PipeTransformer [82] elastically adjusts the degree of pipelining and data parallelism by freezing layers with "stable" weights, and instead dedicates resources to train the remaining "active" layers. HetPipe [133] uses a combination of pipeline and data parallelism on a set of heterogeneous accelerators. Pipeline parallelism can also be implemented with relaxed semantics: PipeDream-2BW [127] maintains two weight versions and guarantees 1-stale weight updates without expensive flushes, while PipeMare [175] and Kosson et al. [99] use asynchoronous pipeline parallelism. These techniques have improved throughput compared to the techniques with pipeline flushes considered in this chapter, but potentially at the cost of convergence rate or final accuracy. Moreover, pipeline parallelism in isolation can still only scale to a number of devices equal to the number of layers in the model, which is limiting for certain model architectures.

PipeDream [125] combined pipeline parallelism and data parallelism in a principled way to reduce cross-device communication. DeepSpeed [5] combined pipeline parallelism with tensor and data parallelism to train models with up to a trillion parameters, but with lower throughput than what was shown in this chapter (52% vs. 36% of peak) for a few reasons: operator fusion to keep most of the operator graph compute-bound, a more-efficient pipeline parallelism schedule to minimize the pipeline bubble size, fast hardware (A100 vs. V100 GPUs and high-bandwidth links between GPUs on the same and different servers), and scaling to more GPUs. We want to emphasize that this higher throughput makes estimated training times much more practical (about 3 months); an aggregate throughput of 37.6 petaFLOP/s would take about 40 months to train an equivalently-sized model. PTD-P can be used to scale to larger models as well, but would need more GPUs to keep training time practical.

Mesh-TensorFlow [152] proposes a language for easily specifying parallelization strategies that combine data and model parallelism. Switch Transformers [72] used Mesh-Tensorflow to train a sparsely activated expert-based model with 1.6 trillion parameters, with improved pre-training speed over the T5-11B model [138].

Sharded Data Parallelism. As part of performance optimizations for MLPerf 0.6 [117], sharded data parallelism [103, 174], where optimizer state is sharded over data-parallel workers, was introduced. This method has two advantages: (a) it does not introduce extra communication over vanilla data parallelism, and (b) it divides the optimizer's computation and memory cost across the data-parallel partitions. ZeRO [140, 141] extends this idea: weight parameters and gradients are sharded across data-parallel workers as well, and workers fetch relevant state from their "owning" workers before performing computations. This adds additional communication, which can be partially hidden by carefully overlapping computation and communication. However, this can become

harder if tensor parallelism is not used or the batch size is not large enough to hide the extra communication overhead (Figure 4.10). ZeRO-Infinity [141] uses NVMe to efficiently swap parameters, enabling the training of very large models on a small number of GPUs. We note that using a small number of GPUs for training a very large model results in unrealistic training times (e.g., thousands of years to converge).

Automatic Partitioning. FlexFlow [96], PipeDream [125], Tarnawski et al. [159], and DAPPLE [71] all auto-partition model training graphs over multiple devices with the help of cost models. However, each of these do not consider *all* the parallelism dimensions considered in this chapter: pipeline and tensor model parallelism, data parallelism, microbatch size, and the effect of memory-savings optimizations like activation recomputation on the training of models larger than the memory capacity of an accelerator. These added dimensions increase the search space that needs to be explored. Gholami et al. [75] show how communication costs for combinations of data and model parallelism can be modeled.

HPC for Model Training. Goyal et al. [76] and You et al. [178] both demonstrate the use of High Performance Computing techniques to train highly-accurate ImageNet models in minutes. However, the image classification models considered fit comfortably on a single accelerator, rendering model parallelism unnecessary, support very large batch sizes (> 32k) that allow scaling data parallelism to large worker counts with infrequent communication, and are composed of compact convolutional layers that are inherently amenable to data-parallel communication (Figure 2.1).

4.7 Discussion and Summary

In this chapter, we have shown how PTD-P (inter-node pipeline parallelism, intra-node tensor parallelism, and data parallelism) can be composed to achieve high aggregate throughput (502 petaFLOP/s) while training large models with a trillion parameters. This facilitates end-to-end training in reasonable times (estimated time of around 3 months for a trillion-parameter model). We discussed the various tradeoffs associated with each of these types of parallelism, and how the interactions between them need to be considered carefully when combined.

Even though the implementation and evaluation in this chapter is GPU-centric, many of these ideas translate to other types of accelerators as well. Concretely, the following are ideas that are accelerator-agnostic: a) the idea of smartly partitioning the model training graph to minimize the amount of communication while still keeping devices active, b) minimizing the number of memory-bound kernels with operator fusion and careful data layout, c) other domain-specific optimizations (e.g., scatter-gather optimization).

Part II

Scheduling at the Macroscale: Heterogeneity-Aware Job Placement on Private and Public Compute Resources

Chapter 5

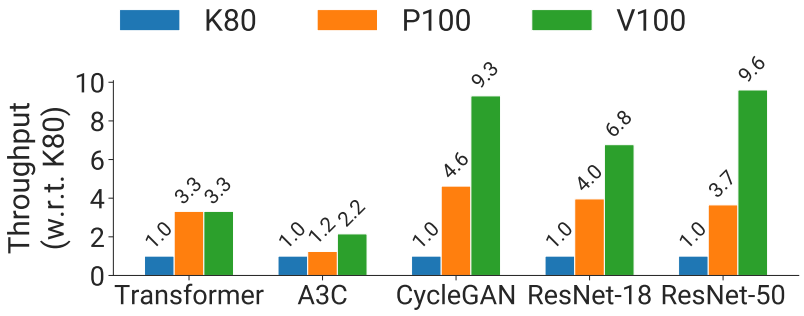
Gavel: A Framework for Heterogeneity-Aware Scheduling

5.1 Introduction

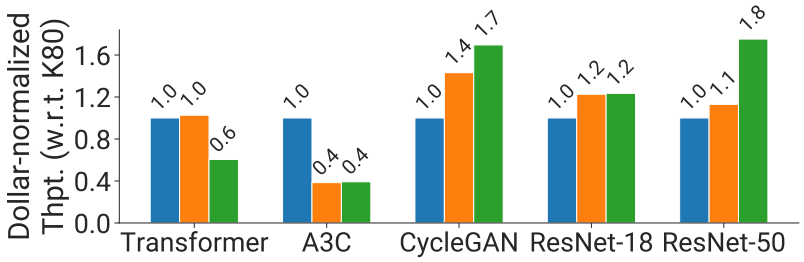
As Moore's law comes to an end, specialized accelerators such as GPUs, TPUs, FPGAs, and other domain-specific architectures have emerged as an alternative to more general-purpose CPUs. These accelerators have been deployed to great effect [97, 73] to train state-of-the-art deep neural network (DNN) models for many domains, including language, image and video [164, 40, 83, 84, 150].

Consequently, users today must choose from a wide variety of accelerators to train their DNN models. For example, public cloud users can rent several generations of NVIDIA GPUs and Google TPUs from cloud providers [2, 3, 4]. Even organizations with private clusters have accumulated different accelerator types over time [91]; anecdotally, our research group at Stanford has NVIDIA Titan V, Titan X, and P100 GPUs in its private cluster. Resources in these multi-tenant settings are typically arbitrated by a scheduler. GPU cluster schedulers such as Themis [114], Tiresias [79], AlloX [106], and Gandiva [172] thus need to decide how to allocate diverse resources to many users while implementing complex cluster-wide *scheduling policies*, optimizing objectives such as fairness or makespan. Unfortunately, choosing the most effective accelerator types in this context is difficult for three reasons:

Performance Heterogeneity. Commonly used models show heterogeneous performance behavior across accelerator types due to various architectural differences. For example, Figure 5.1a shows that a ResNet-50 model sees a nearly 10× speedup from an NVIDIA V100 GPU compared to a K80 GPU, while an A3C Deep Reinforcement Learning model only sees a 2× speedup. However, as shown in Figure 5.1b, the V100 is no longer the optimal choice for all models when we consider



(a) Throughput.



(b) Dollar-normalized.

Figure 5.1: Throughputs and dollar-normalized throughputs of training for various ML models. Dollar-normalized throughputs are computed by dividing the corresponding throughput by the relevant GCP on-demand price. The magnitude of speedup across GPU generations varies significantly across models.

the number of samples trained per dollar – for many models, the older P100 GPU is competitive or cheaper on a per-dollar basis. Some scheduling policies can also benefit from splitting a job between *multiple* resource types: for example, minimizing a job's cost subject to a latency SLO (e.g., complete a job in 10 hours) might involve using a cheaper accelerator to begin training and then switching to a faster, more expensive device to meet the SLO. Thus, for even simple *single-job* settings, the choice of accelerator type is non-trivial and depends on *both* the job and the policy. This gets more complicated in *multi-job* settings as granting all jobs their preferred accelerator simultaneously might not be possible. Existing schedulers like Gandiva, Tiresias, and Themis do not consider this heterogeneous performance behavior.

Generality across Policies. Cluster operators might want to implement different scheduling policies based on their business goals, such as optimizing for time to complete a set of batch jobs (makespan), fairness for ad-hoc jobs, or more sophisticated *hierarchical* policies that divide resources among high-level entities (e.g., departments) using one policy, and then individual jobs within the entity using another [91]. In data analytics clusters, many job schedulers have support for hierarchical allocation policies [11, 179, 12, 28] already. The two recently proposed GPU schedulers

that do consider heterogeneous resources, AlloX [106] and Gandiva$_{\text{fair}}$ [48], optimize for a single scheduling objective, and tightly couple their scheduling mechanism to that objective (e.g., max-min fairness). Thus, they cannot easily support the more sophisticated policies often used in practice.

Colocation and Placement Optimizations. To improve cluster utilization, existing GPU schedulers often deploy optimizations such as space sharing as in Gandiva [172], where multiple jobs can use the same accelerator concurrently, and placement sensitivity as in Themis and Tiresias [114, 79], which involves the careful placement of tasks in a distributed job to ensure good scaling performance. The performance benefits of these optimizations should be considered explicitly while optimizing for global scheduling objectives, since these optimizations are more effective when deployed in a heterogeneity-aware way. We show that explicit modeling for space sharing can improve objectives by 2.2× compared to Gandiva's ad-hoc approach.

In this chapter, we present Gavel, a new cluster scheduler designed for DNN training in both on-premise and cloud deployments, that effectively incorporates heterogeneity in both hardware accelerators and workloads to generalize a wide range of existing scheduling policies in a completely automated fashion. For example, Gavel can provide heterogeneity-aware versions of fair sharing / least attained service [79], FIFO, minimum makespan, minimum cost subject to SLOs, finish-time fairness [114], shortest job first, and hierarchical policies [179, 28].

Gavel's key observation is that many widely used scheduling policies, including hierarchical ones, can be expressed as *optimization problems* whose objective is a function of the jobs' achieved throughputs. For example, the least attained service policy involves maximizing the minimum scaled throughput across jobs, the minimize makespan policy involves minimizing the maximum duration (computed as the ratio of number of iterations to achieved throughput), and so on. Given the optimization problem for a scheduling policy, Gavel introduces a general way to transform the problem to make it heterogenity-, colocation- and placement-aware. In particular, Gavel changes the problem to search over a *heterogeneous allocation* for each job, the fraction of time spent in various resource configurations (e.g., 60% of time running alone on a V100 GPU and 40% of time space-sharing an A100 GPU with another job), and changes the throughput terms in the objective function to *effective throughput*, i.e. the average throughput of the job over the mix of resources in its allocation. Additional constraints need to be added to ensure that the returned allocation is valid. We show that Gavel's transformed optimization problems are efficient to execute even for clusters with hundreds of GPUs and jobs, and can support a wide range of policies. Many of these problems can be solved using a sequence of one or more linear programs.

Gavel's heterogeneity-aware allocations for each job need to be mapped to actual scheduling decisions (placement of jobs on specific resources in the cluster for a specified duration of time). To achieve this, Gavel uses a preemptive *round-based scheduling mechanism* to ensure that jobs receive resources in fractions similar to the computed target allocation. Gavel's scheduling mechanism needs

to be able to schedule both distributed training jobs, which request multiple accelerators at once, as well as combinations of jobs running concurrently on a given accelerator due to space sharing.

Gavel makes these scheduling decisions transparently: it specifies an API between the scheduler and applications that allow jobs written in existing deep learning frameworks like PyTorch [134] and TensorFlow [36] to be moved between resources with minimal code changes, and uses a mechanism similar to Quasar [63] to estimate performance measurements of colocated jobs, which are needed as inputs to Gavel's policies, when not available *a priori*.

By explicitly considering performance heterogeneity, Gavel improves various policy objectives (e.g., average job completion time or makespan): on a smaller physical cluster, it improves average JCT by 1.5×, and on a larger simulated cluster, it increases the maximum input load a cluster can support, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

Summary of Contributions. To summarize, our main contributions are:

- A systematic method to convert existing cluster scheduling policies into equivalent policies that consider heterogeneity and colocation; these equivalent optimization problems are practical for current DNN clusters.
- A round-based scheduling mechanism to ensure that the cluster realizes the allocations returned by these policies.
- Generalizations of many existing policies that improve corresponding objectives.

Gavel is open sourced at `https://github.com/stanford-futuredata/gavel`.

5.2 Background

In this section, we provide a brief overview of DNN training (§5.2.1), and discuss performance optimizations used in existing schedulers that Gavel can help deploy more effectively (§5.2.2).

5.2.1 Deep Neural Network (DNN) Training

DNN training proceeds in iterations. In each iteration, the DNN processes a collection of inputs (called a batch) and subsequently updates the model parameters using gradients derived from the input batch. Each batch is typically of similar size, which means model training throughput using short profiling runs (order of minutes). Gavel leverages this fact in its throughput estimator. Jobs are typically fairly long-running (on the order of hours to days), and can be distributed over many workers [34, 172].

Modern DNN schedulers leverage the fact that DNN training is iterative to suspend and resume training at iteration boundaries [79, 172]; this ensures that jobs can be time multiplexed over the existing physical resources. The latest model parameters need to be checkpointed to stable storage when a job is suspended to ensure training progress is not lost. In this work, we show how *time sharing* should be deployed to optimize various single- and multi-job objectives.

5.2.2 Performance Optimizations

Prior work has shown that GPUs can be severely under-utilized in multi-tenant clusters [91]; for example, average GPU utilization (measured as the percentage of GPU Streaming Multiprocessors active over time) was as low as 52% on a Microsoft cluster. Prior work has also shown the placement of tasks for a distributed training job can have significant impact on performance. Gavel can *optionally* deploy these optimizations systematically, as we show in §5.3.1.

Space Sharing. Smaller models often do not leverage the full computational capacity of modern GPUs. In such cases, concurrently executing multiple models on the same GPU using NVIDIA's Multi Process Service (MPS) or CUDA streams can help improve utilization [35, 130].

Placement Sensitivity. DNN models show heterogeneity in their distributed scaling behavior depending on the size of the tensors that need to be exchanged between workers during training: some models have compact weight representations and can scale well even when workers are not on the same server, while other models scale poorly when workers are spread over many servers. Existing schedulers like Tiresias use heuristics for placement sensitivity.

5.3 System Overview

Given a collection of jobs, Gavel arbitrates cluster resources (in the form of accelerators of different types) among the resident jobs, while optimizing for the desired cluster objective. This is accomplished in a two-step process: first, a *heterogeneity-aware policy* computes the fraction of time different jobs (and combinations) should run on different accelerator types to optimize the desired objective. These policies require as input the performance behavior (in terms of throughputs) for each job on each accelerator type, which can either be provided by the user, or can be measured on the fly by Gavel's throughput estimator. Allocations are intended to be respected only between allocation recomputation events; for example, if job 1 is much longer than job 2, the allocation will be recomputed once job 2 completes. Gavel can recompute its policy either when a *reset event* occurs (job arrives or completes, worker in the cluster fails), or at periodic intervals of time. Given the policy's output allocation, Gavel's *scheduling mechanism* grants jobs time on the different resources, and moves jobs between workers as necessary to ensure that the true fraction of time each job spends on

different resources closely resembles the optimal allocation returned by the policy. Gavel's workflow is shown in Figure 5.2.

Figure 5.2: Gavel overview. Jobs are written in frameworks like PyTorch or TensorFlow. Gavel's throughput estimator obtains performance measurements for each runnable job on each available accelerator type if necessary; its policy then computes an allocation that optimizes a user-specified objective such as fairness. Gavel's scheduling mechanism accepts this computed allocation as an input, and makes per-round placement decisions in proportions that faithfully mimic the computed allocation.

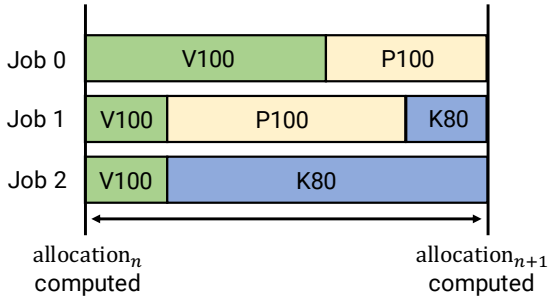


Figure 5.3: The *cumulative* time each job spends on accelerator types between allocation recomputations for allocation X^{example}.

5.3.1 Heterogeneity-Aware Policies

Gavel expresses scheduling policies as optimization problems for various objectives of interest, such as fairness or makespan, and allocations as matrices that specify the fraction of wall-clock time a job should spend on each accelerator type *between* allocation recomputations. A matrix X can represent allocations on a single accelerator type (homogeneous setting), on multiple accelerator types (heterogeneous setting), as well as with other optimizations. Consider X^{example}:

$$X^{\text{example}} = \begin{array}{c} \begin{array}{ccc} V100 & P100 & K80 \end{array} \\ \begin{pmatrix} 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.0 & 0.8 \end{pmatrix} \end{array} \begin{array}{l} \\ \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array}$$

According to this allocation specified over three jobs and three accelerator types, job 0 should spend 60% of the time this allocation is valid on a V100 GPU, and the remaining 40% of time on a P100 GPU. This is shown visually in Figure 5.3.

Gavel finds an optimal value for the matrix X given a policy expressed as an optimization problem. To construct the optimization problem for a given policy, Gavel requires a *throughput matrix* T with each job's throughput (in training iterations per second) on different accelerators. T_{mj} can be set to $-\infty$ if job m does not run on accelerator type j (for example, due to memory constraints).

Given T and X, we define the *effective throughput* of a model m as the time-weighted average throughput across accelerators and jobs. We denote this quantity $\text{throughput}_T(m, X)$ or simply $\text{throughput}(m, X)$ (dropping the T) for brevity. For allocations X without space sharing,

$$\text{throughput}(m, X) = \sum_{\substack{j \in \\ \text{accelerator types}}} T_{mj} \cdot X_{mj}$$

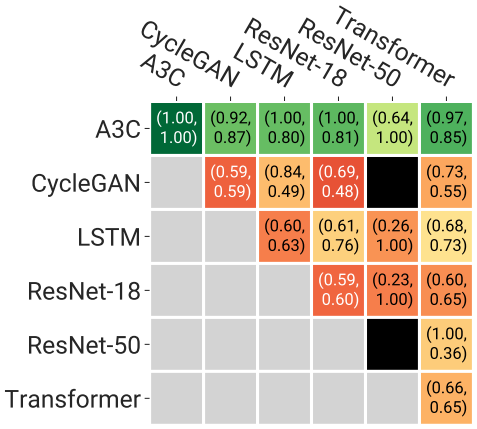


Figure 5.4: Performance of several DNN models when run concurrently on a single P100 GPU. The cell at row i and column j reports the normalized throughput (iterations/second) achieved by co-located models i and j. Throughputs are normalized with respect to the throughput achieved by each model when run in isolation. Black squares show jobs that cannot co-locate due to memory constraints.

Different cluster scheduling policies can be expressed as optimization problems for X while maximizing or minimizing an objective function. Constraints need to be specified to ensure that X is a valid allocation. A hypothetical policy that maximizes total effective throughput looks like:

$$\text{Maximize}_X \sum_{m \in \text{jobs}} \text{throughput}(m, X)$$

Subject to the constraints:

$$0 \leq X_{mj} \leq 1 \qquad \forall (m, j) \tag{5.1}$$

$$\textstyle\sum_j X_{mj} \leq 1 \qquad \forall m \tag{5.2}$$

$$\textstyle\sum_m X_{mj} \cdot \text{scale\_factor}_m \leq \text{num\_workers}_j \qquad \forall j \tag{5.3}$$

These constraints ensure that each job-worker allocation is non-negative and between 0 and 1 (equation 5.1), that the total allocation for a job does not exceed 1 (equation 5.2), and that the allocation does not oversubscribe workers (equation 5.3).

Space Sharing. Gavel's allocation matrices can also incorporate space sharing (SS). While previous work has used greedy algorithms for space sharing, we found that different pairs of DNN applications in practice have vastly different performance when colocated together, based on the resources they consume (Figure 5.4). When using space sharing, X needs to contain rows for each

viable combination of jobs, and T needs to have throughputs of the job combinations, like:

$$T = \begin{array}{c} \begin{array}{ccc} V100 & P100 & K80 \end{array} \\ \begin{pmatrix} 40.0 & 20.0 & 10.0 \\ 15.0 & 10.0 & 5.0 \\ (20.0, 7.5) & 0.0 & 0.0 \end{pmatrix} \end{array} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{jobs (0, 1)} \end{array}$$

The SS-aware allocation X dictates the fraction of time that each *job combination* should spend on each accelerator type.

We limit entries of T to combinations of at most 2 jobs; we found empirically that larger combinations rarely increase net throughput. Additionally, although the size of T grows quadratically with the number of jobs even with job combinations of size 2, we found that in practice we only need to consider combinations that actually perform well. We evaluate the scaling behavior of these SS-aware policies in §5.7.4.

Objectives in terms of throughput(m, X) remain the same; however, throughput(m, X) now needs to be computed to include the throughputs of co-located jobs:

$$\text{throughput}(m, X) = \sum_{\substack{j \in \\ \text{accelerator types}}} \sum_{k \in C_m} T_{kjm} \cdot X_{kjm}$$

The constraints need to be slighly modified as well to ensure that X is still a valid allocation:

$$\begin{aligned} 0 \leq X_{kj} \leq 1 \quad & \forall (k, j) \\ \textstyle\sum_{k \in C_m} \sum_j X_{kj} \leq 1 \quad & \forall m \\ \textstyle\sum_k X_{kj} \cdot \text{scale\_factor}_m \leq \text{num\_workers}_j \quad & \forall j \end{aligned}$$

C_m is the set of all job combinations that contain job m.

Placement Sensitivity. Similarly, Gavel's allocation matrices can also be extended to incorporate placement sensitivity. The observed throughput for distributed jobs depends on the location of tasks, as well as the model and accelerator type (slower workers are less likely to be communication-bound, which means consolidation of tasks is less effective). We can make our policies *placement-sensitive* by considering the performance of distributed jobs in: 1) a consolidated setting, where as many accelerators are on the same server as possible (for example, 8 GPUs per server if using 8-GPU servers), and 2) an unconsolidated setting, where accelerators are on independent servers. These are extreme points in the placement space, and are upper and lower bounds on performance. We can model this in our policies by having two different worker types (consolidated and unconsolidated) with corresponding throughput values in T and allocation values in X.

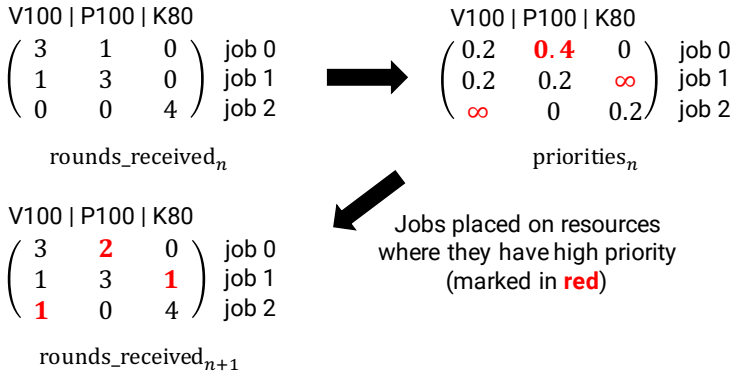


Figure 5.5: Priorities are used to move the received allocation towards the intended allocation (in this case, X^{example}). priorities$_n$ is computed as $X/\text{rounds\_received}_n$ (element-wise division).

5.3.2 Round-based Scheduling Mechanism

After computing the optimal allocation, Gavel's next step is to assign jobs (or job combinations, in the case of SS) to accelerator types while matching the optimal allocation as closely as possible. That is, to realize the allocation X^{example} above, the scheduling mechanism needs to make sure that in the time period where jobs 0, 1, and 2 are the only three runnable jobs in the cluster, jobs should receive resources according to their computed optimal time fractions.

To do this, the scheduler computes a priority score for every job and accelerator type combination. This priority score is high when a job has received a smaller time fraction on a particular accelerator type than specified in the optimal allocation. Scheduling is performed in rounds; in each round, the scheduler runs jobs in decreasing priority order, while ensuring that a given job is not scheduled on multiple sets of workers (or accelerators) in a given round. This is shown in Figure 5.5. Priorities are updated as rounds complete. We have found empirically that round durations of around 6 minutes allow Gavel to effectively approximate the ideal allocation (§5.7.5).

5.3.3 Throughput Estimator

To estimate the throughputs of concurrent jobs (e.g., in the case of space sharing), Gavel employs a throughput estimator, similar to those found in prior work such as Quasar [63]. Gavel's throughput estimator maps a new job to a set of pre-profiled reference jobs. The throughputs of the closest reference job can then be used as the initial performance estimate for the new job's combinations. For individual jobs, the throughput estimator is not needed, since throughputs can be estimated on the fly as jobs run on different resource types.

5.3.4 Limitations and Non-Goals

While Gavel exposes a flexible API that supports a variety of policies and objectives, we do not propose new scheduling policies or performance optimizations in this work. Instead, Gavel's main goal is to determine how best to share resources amongst many different users and jobs in a heterogeneity-aware way, while supporting many existing cluster-wide objectives. Gavel accomplishes these goals with a policy framework that easily allows policies to be made heterogeneity-, colocation-, and placement-aware (§5.4), a reusable scheduling mechanism (§5.5), and a narrow scheduler API that allows users to deploy their applications with minimal code changes (§5.6).

5.4 Scheduling Policies

In this section, we show how various scheduling policies such as max-min fairness (Least Attained Service or LAS) and multi-level fairness can be expressed as optimization problems in terms of effective throughput. We describe some properties of the resulting heterogeneity-aware allocations at the end of this section.

5.4.1 Max-Min Fairness as an Optimization Problem

The classical Least Attained Service (LAS) policy, used by Tiresias [79], implements max-min fairness across active users in the cluster, by round-robining resources across jobs according to the total number of accelerator hours consumed. This can be modified into a weighted max-min fairness policy with per-user weights w_m. On a homogeneous cluster, if a job m with weight w_m receives a fraction X_m (which is a scalar since there is only one resource type), LAS can be expressed as the following optimization problem:

$$\text{Maximize}_X \min_m \frac{1}{w_m} X_m.$$

We need to add a constraint to ensure that the cluster is not overprovisioned ($\sum_m X_m \leq 1$).

However, this vanilla LAS policy is not fair in a heterogeneous setting; jobs might see unequal reductions in throughput due to variations in performance across accelerator types. For example, giving one job a K80 and another job a V100 would equalize their number of resources, but could result in very low performance for the job with the K80.

To compute a more fair allocation, we can compute max-min fairness over the weighted normalized effective throughputs (defined in §5.3.1). Let X_m^{equal} be the allocation given to job m assuming it receives equal time share on each worker. For example, if the cluster had 1 V100 and 1 K80, $X_m^{\text{equal}} = [0.5, 0.5]$. X_m^{equal} scales the effective throughputs to make them comparable across jobs.

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})}.$$

Policy	Description
Makespan	Minimize time taken by batch of jobs.
LAS [79]	Max-min fairness by total compute time.
LAS w/ weights	Max-min fairness with weights.
Finish Time Fairness [114]	Maximize minimum job speedup.
FIFO	First in, first out.
Shortest Job First	Minimize time taken by shortest job.
Minimize cost	Minimize total cost in public cloud.
Minimize cost w/ SLOs	Minimize total cost subject to SLOs.
Hierarchical [179]	Multi-level policy: FIFO, fairness, etc.

Table 5.1: Policies that can be expressed in Gavel.

As specified in §5.3.1, additional constraints need to be specified to ensure that allocations are valid.

As an example, consider 3 jobs which benefit differently when moved from a K80 to a V100 GPU:

$$T = \begin{array}{c} \begin{array}{cc} V100 & K80 \end{array} \\ \begin{pmatrix} 40.0 & 10.0 \\ 12.0 & 4.0 \\ 100.0 & 50.0 \end{pmatrix} \end{array} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array}$$

Solving the above optimization problem with $w_m = 1$, and a cluster with 1 V100 and 1 K80 yields the following allocation:

$$X^{\text{het.}} = \begin{array}{c} \begin{array}{cc} V100 & K80 \end{array} \\ \begin{pmatrix} 0.45 & 0.0 \\ 0.45 & 0.09 \\ 0.09 & 0.91 \end{pmatrix} \end{array} \begin{array}{l} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{array}$$

Jobs receive about 10% higher throughput compared to an allocation where every user is given $1/n$ of the time on each accelerator (here, $n = 3$), also called an *isolated allocation* [74].

Objective functions for fairness policies need to be modified to take into account multi-resource jobs (scale_factor$_m > 1$), since these multi-resource jobs occupy a larger share of the cluster per unit time. An easy way to do this is to multiply the max-min objectives from before by scale_factor$_m$. Concretely, the LAS objective from before becomes:

$$\text{Maximize}_X \min_m \frac{1}{w_m} \frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})} \cdot \text{scale\_factor}_m.$$

5.4.2 Other Policies as Optimization Problems

We can express many other common cluster scheduling policies, some proposed by recent papers, using throughput(m, X); we list these policies in Table 5.1. Most of these policies can be expressed using a single linear program, with a few exceptions: the cost policies are formulated as a linear-fractional program [13], which can be reduced to a sequence of linear programs. These optimization problems yield corresponding heterogeneity-aware allocations. The optimal allocation can be computed using off-the-shelf solvers.

Minimize Makespan. The makespan minimization policy tries to complete all active jobs as soon as possible. Gandiva uses a version of this policy to finish higher-level tasks such as hyperparameter tuning and AutoML, which involve training a large number of variants of a model. If num_steps$_m$ is the number of iterations remaining to train model m, then the makespan is the maximum of the durations of all active jobs, where the duration of job m is the ratio of the number of iterations to throughput(m, X) (expressed in iterations / second). Overall, this can be framed as,

$$\text{Minimize}_X \max_m \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}.$$

Minimize Finish-Time Fairness (Themis). Themis [114] proposes a new metric called finish-time fairness (represented as ρ), which is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users using the cluster. This can be expressed in terms of throughput(m, X) as follows (num_steps$_m$ is the number of iterations remaining to train model m, t_m is the time elapsed since the start of training for model m, and t_m^{isolated} is the hypothetical time elapsed since the start of training if model m had $1/n$ of the cluster to itself),

$$\rho_T(m, X) = \frac{t_m + \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}}{t_m^{\text{isolated}} + \frac{\text{num\_steps}_m}{\text{throughput}(m, X^{\text{isolated}})}}.$$

The final optimization problem is then,

$$\text{Minimize}_X \max_m \rho_T(m, X).$$

FIFO. The First-In-First-Out (FIFO) policy schedules jobs in the order they arrive. In a heterogeneous regime, jobs should be placed on the fastest available accelerator type. Mathematically, we can write this as maximizing the throughput of job m relative to its throughput on the fastest type (throughput(m, X^{fastest})). Assuming that jobs are enumerated in order of their arrival time (m arrived before $m + 1$), a FIFO allocation can be computed with the following objective:

$$\text{Maximize}_X \sum_m \frac{\text{throughput}(m, X)}{\text{throughput}(m, X^{\text{fastest}})}(M - m).$$

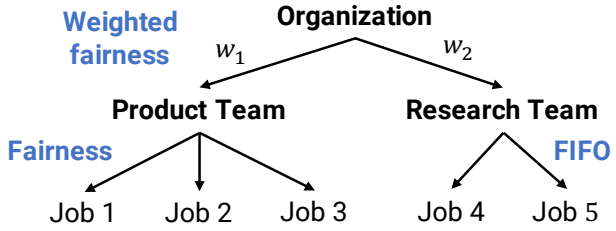


Figure 5.6: Example of a hierarchical policy. Weighted fairness across two entities (a product and research team), fairness across jobs within the product team, and FIFO within the research team.

where M is the total number of jobs.

Shortest Job First. The Shortest Job First (SJF) policy finds the allocation that minimizes the duration of the shortest job,

$$\text{Minimize}_X \min_m \frac{\text{num\_steps}_m}{\text{throughput}(m, X)}.$$

Minimizing Total Cost and Cost Subject to SLOs. We can also express policies for deployments that use elastic public cloud resources. Since cloud VMs are charged on a per-time basis, we can express policies that explicitly optimize for total cost, speed, or both. We show details of such policies in the next chapter.

5.4.3 Hierarchical Scheduling Policies

Modern cluster schedulers do not only deploy "single-level" policies. Hierarchical policies are common [11, 179, 28]: a large organization might share a single physical cluster among many sub-organizations (or entities) using a fairness policy. In turn, each entity can share resources among individual jobs according to a distinct per-entity policy, such as per-user fairness or FIFO. We give an example in Figure 5.6, where a research and product team share the same physical cluster. The research team runs ad-hoc experiments that can be executed in FIFO order, but the product team needs to ensure that all its jobs receive a fair share of the cluster.

Gavel can currently support fairness in the upper levels and fairness or FIFO in the lower levels, which matches the hierarchical policies supported by the Hadoop scheduler [11]. Determining how to extend this to other types of hierarchical policies (e.g., with finish time fairness) is future work.

Gavel solves hierarchical objectives using a procedure called water filling [42], which is used in other max-min fairness problems such as link allocation in networks [137]. At a high level, the water-filling algorithm increases the allocation given to all parties at an equal rate to respect max-min fairness, until a party saturates. The saturated party is then taken out, and the procedure

is repeated until all commodities are saturated. We adapt this procedure to our setting, solving a series of optimization problems iteratively: an LP that computes a fair allocation across entities while respecting each entity's internal policy, and an MILP that identifies *bottlenecked jobs*, i.e., jobs whose effective throughputs cannot be further improved without lowering other jobs' effective throughput.

We assume that each entity s is associated with a weight w_s; the jobs belonging to this entity receive a total cluster share proportional to this weight. We denote w_m^{job} to be the weight of job m, set such that $\sum_{m \in s} w_m^{\text{job}} = w_s$. Jobs are assigned priorities in accordance to the relevant entity's policy; for example, a fairness policy within an entity would assign each job a weight proportional to its individual weight within the entity, while for FIFO, the first job in the queue would initially receive the entire weight of the entity.

In each iteration, we solve the following modified LP (assuming $\text{scale\_factor}_m = 1$ for simplicity):

$$\text{Maximize}_X \min_{\{m : w_m^{\text{job}} > 0\}} \frac{1}{w_m^{\text{job}}} \left(\frac{\text{throughput}(m, X)}{\text{throughput}(m, X_m^{\text{equal}})} - t_m \right).$$

t_m is the normalized effective throughput of job m in the previous iteration ($t_m := 0$ in the first iteration). The above objective can be appropriately modified for $\text{scale\_factor}_m > 1$. Bottlenecked jobs are given priority 0 and no longer considered in future iterations. Priorities are redistributed among non-bottlenecked jobs according to the entity's policy at the end of every iteration. For instance, in the example shown in Figure 5.6, if job 4 is bottlenecked, then its weight is reassigned to job 5 in accordance to the FIFO policy, while if job 2 is bottlenecked, its weight is distributed equally between jobs 1 and 3 in accordance with the entity's fairness policy. The LP then solves the max-min problem on the resources remaining while ensuring each job's throughput does not drop compared to the previous iteration's allocation X^{prev}, expressed as $\text{throughput}(m, X) \geq \text{throughput}(m, X^{\text{prev}})$ for all m. Iterations continue until all jobs are bottlenecked. To make this procedure more concrete, consider an example with 4 identical jobs: job 1 with a weight of 3.0, and jobs 2 to 4 with a weight of 1.0; and 4 identical GPUs. In the first iteration, job 1 is assigned resources such that its throughput is 1.0, and jobs 2, 3, and 4 are assigned resources such that their throughput is 0.33 to respect weights. Job 1 is a bottleneck; the throughput of the remaining jobs can still be increased. In the next iteration, jobs 2 to 4 are given full-GPU allocations.

The final allocation satisfies both inter-entity and intra-entity policies. We note that the above water-filling procedure can also be used for single-level fairness policies such as the one described in §5.4.1 to improve the throughput of non-bottelenecked jobs.

Identifying bottleneck jobs in fairness policy. Solving a max-min fairness policy, such as LAS or hierarchical fairness, results in an allocation that satisfies fairness metrics but may underutilize resources in scenarios where the bottlenecked job's throughput is matched by other jobs without using all available resources. Identifying bottleneck jobs after an iteration of a fairness policy computation

can be done by solving a mixed-integer linear program. The binary integer variable z_m is set to 1 when job m's scaled effective throughput can be improved without causing any other job's scaled effective throughput to drop below the minimum computed in the previous iteration of the policy's LP. We identify all jobs which are stuck as $\{m : z_m = 0\}$ by computing an allocation that maximizes the sum of all z_m:

$$\text{Maximize}_X \sum_{\{m:p_m>0\}} z_m$$

Subject to:

$$z_m = \begin{cases} 1 & \text{if throughput}(m, X) > \text{throughput}(m, X^{\text{prev}}) \\ 0 & \text{otherwise} \end{cases}$$

The conditional constraint on z_m can be expressed as two linear inequalities:

$$\text{throughput}(m, X^{\text{prev}}) < \text{throughput}(m, X) + Y(1 - z_m)$$
$$\text{throughput}(m, X^{\text{prev}}) \geq \text{throughput}(m, X) - Y z_m$$

Y here is a sufficiently large number such that it is not an active constraint, such as the maximum throughput of the job.

5.4.4 Properties of Gavel's Policies

Existing scheduling schemes have been analyzed in terms of properties like sharing incentive, Pareto efficiency, and strategy proofness [74]. We formalize Gavel's heterogeneity-aware policies in the context of these properties as well.

Homogeneous Clusters. For homogeneous clusters, Gavel's heterogeneity-aware policies are equivalent to the baseline policies (throughput$(m, X) = X_m \cdot T_m$), since the heterogeneity-aware optimization problems reduce to the original optimization problems with one accelerator type.

Sharing Incentive. For heterogeneous clusters, the policy's objective metric (maximize least job share in LAS, completion time of first job in FIFO, or makespan) is at least as good as it would be under a policy that naïvely splits all resources equally among all runnable jobs. This is because the allocation corresponding to giving each user $1/n$ of each resource is a feasible solution, so Gavel's solution will be at least as good. All Gavel policies thus have *sharing incentive* [74], which encourages users to use the shared cluster rather than a static private share.

Colocation. Solutions with colocation are always at least as good as without colocation.

Pareto Efficiency. Allocations of max-min fairness policies with water filling are Pareto efficient: that is, the allocation for a particular job cannot be increased without decreasing the allocation for another job. This follows directly from the water filling procedure.

Note that some of Gavel's policies may not satisfy other desirable properties. For example, Sun et al. [158] showed that no fair-sharing policy can simultaneously satisfy Pareto efficiency, sharing incentive, and strategy proofness in a setting with interchangeable resources. If users manipulate their throughputs, then they can possibly obtain larger shares of the cluster (e.g., jobs can be placed on a faster accelerator type) for certain objectives. Exploring how to make Gavel's policies strategy-proof is interesting future work.

5.5 Scheduling Mechanism

Gavel's scheduling mechanism schedules training iterations of runnable jobs on the available workers (with possibly different accelerators), such that for each schedulable job (or combination), the fraction of wall-clock time spent on each accelerator type is approximately equal to the computed optimal allocation X^{opt}. This is challenging for two reasons:

1. Jobs can run on multiple accelerators. Moreover, since distributed training can be communication intensive [57, 125], jobs should be placed on accelerators "close" to each other (for example, on accelerators on the same server, or on accelerators in servers in the same rack).
2. Combinations of up to two jobs can run on a set of accelerators in order to improve resource utilization (space sharing). Each distinct job can have $\leq$ one job combination running in a given round to prevent work duplication.

Gavel makes its scheduling decisions in *rounds*. This is similar in spirit to Tiresias's [79] priority discretization. However, Gavel's scheduling mechanism differs from Tiresias's in three ways:

1. Gavel needs to schedule jobs on different accelerator types: it needs to decide which job should be active in any round *and* which accelerator type to use.
2. Gavel needs to grant resources to jobs while respecting an *arbitrary allocation*.
3. Gavel's round-based scheduler grants time to jobs while ensuring that multiple job combinations sharing a job do not run in the same round; Tiresias does not consider job combinations and does not need to deal with this.

Gavel's scheduler tries to place work on all available workers for a specific duration (this time period is configurable; we use 6 minutes in our experiments). We call the work handed to each worker in a given round a *micro-task*. Without rounds, jobs that request many accelerators can

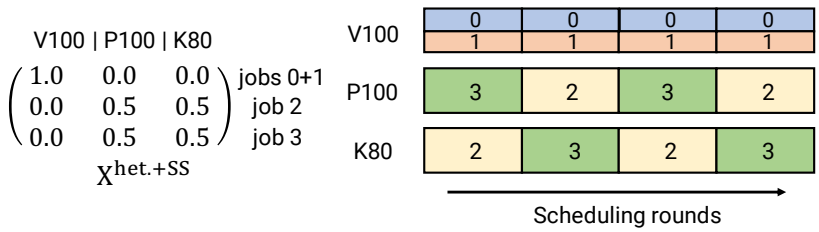


Figure 5.7: Round-based scheduling mechanism in action to achieve an allocation $X^{\text{het.+SS}}$. Space sharing is shown with vertically split boxes. Each round is denoted by a box.

suffer from starvation. For example, consider a cluster with 8 total accelerators and 4 available. The scheduler can handle a 8-accelerator job waiting for resources in one of two ways:

1. Wait for 8 accelerators to become available; 4 accelerators will be unused until the full quota of 8 accelerators becomes available.

2. Keep the 8-accelerator job in the queue, and give 4 accelerators to another job that requests a fewer number of resources.

However, this situation can repeat itself, leading to starvation [179]. Scheduling is thus performed in rounds to limit resource under-utilization, simplify scheduling logic, and ensure that jobs with large scale factors do not experience prolonged starvation.

Since the number of active, *schedulable* jobs might far exceed the total number of workers, Gavel first determines the job combinations that should run in the upcoming round. To do this, Gavel maintains the time t_{mj} spent by a job (or combination) m on accelerator type j, which is updated as jobs run on different accelerator types. Given t_{mj}, Gavel's scheduler can then compute the fraction of total wall-clock time spent by each job (or combination) m on each accelerator type j as $f_{mj} = t_{mj}/(\sum_{m'} t_{m'j})$. The matrix of priorities is then just the element-wise division of X^{opt} by f.

Algorithm. In every round, we want to move f_{mj} closer to X^{opt}_{mj}. This can be achieved by giving high-priority jobs time on accelerator type j.

This problem can be solved exactly if jobs only request single accelerators and if space sharing is not deployed by finding the num_workers$_j$ jobs with highest priority (for example, using a heap). However, jobs submitted to Gavel can be distributed, and space sharing can be used to improve resource utilization. Solving this problem exactly with these added requirements makes the problem similar to a multiple-choice knapsack problem [155], which is NP-hard.

To overcome these challenges, we observe that it is acceptable to make greedy sub-optimal *scheduling* decisions occasionally in any given round, since we can recover from these sub-optimal decisions in subsequent rounds: our goal is to ensure that the average allocation each job receives

Algorithm 2 Algorithm for Gavel's Scheduling Mechanism.

function SCHEDULE_JOBS
    active_combinations ← all active job combinations
    num_workers_rem. ← number of total workers
    while num_workers_rem.g > 0 **do**
        $j \leftarrow$ job combination with highest priority
        Remove j from active_combinations
        if j.scale_factor $>$ num_workers_rem. **then**
            continue
        for all j' that conflict (share a job k) with j **do**
            Remove j' from active_combinations
        num_workers_rem. $-= j$.scale_factor

over multiple rounds resemble the computed allocation (the allocations returned by policies are optimal, which follows from how policies in Gavel are expressed as optimization problems). We study the impact of this design choice in §5.7.5. A job (combination) not run in a particular round will have increased priority in subsequent rounds until it receives accelerator time, while a job that runs in a particular round will have decreased priority. This ensures that jobs do not suffer from starvation if they have a non-zero optimal allocation.

Gavel uses a greedy algorithm to pick the highest-priority job combinations that fit in the provided resource budget. The algorithm maintains a set of eligible job combinations that can be scheduled in the upcoming scheduling round. The scheduling mechanism then tries to add job combinations with highest priority into a job_combinations_to_schedule set. Once a job combination is added to this set, all *conflicting* job combinations are removed from the set of eligible combinations to ensure that a given job is not run more than once in a given scheduling round. Job combinations that cannot fit in the current round due to space limitations (required number of accelerators unavailable) are also removed from the set of eligible combinations. This procedure is detailed in Algorithm 2. Gavel's scheduling mechanism is decoupled from its policies, ensuring that the same scheduling mechanism can be used for many different policies. Figure 5.7 shows Gavel's scheduling mechanism in action.

Once Gavel has decided what jobs (and combinations) should run in a given round on different accelerator types, Gavel must decide how to *place* these jobs. Gavel's scheduler places jobs in decreasing order of the number of requested workers, and tries to give jobs accelerators on the same physical server to minimize fragmentation.

5.6 Implementation

We implemented a prototype of Gavel in approximately 9000 lines of Python code, and implemented a simulator in about 500 LOC. We used `cvxpy` [67] to implement Gavel's heterogeneity-aware policies, and `gRPC` [9] to communicate control messages between the scheduler and workers.

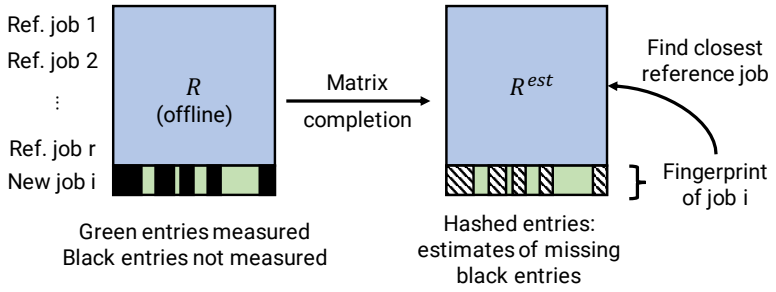


Figure 5.8: Gavel's throughput estimator. Profiling is combined with matrix completion to obtain a fingerprint for every new job. The fingerprint is then used to find the closest reference job.

Interface between Scheduler and Applications. Gavel currently supports user applications written in PyTorch [134]; support for TensorFlow [36] is left for future work. The scheduler and user applications then interact through a narrow API. Gavel ships with a Python library that users can import into their code. This library provides an implementation for a wrapper around existing framework-provided data iterators (`GavelIterator`). `GavelIterator` ensures that each task in a distributed job runs for the same number of iterations, and synchronizes the conclusion of rounds between the scheduler and workers. `GavelIterator` is instantiated with arguments `train_loader` (base data loader), `load_checkpoint`, `save_checkpoint`, and a configuration object. `load_checkpoint` is a pointer to a function that loads all necessary parameters and metadata from a checkpoint at the start of a round, and `save_checkpoint` is a pointer to a function that creates a checkpoint at the end of a round; these need to call appropriate framework methods (< 5 LOC).

`GavelIterator` contacts the scheduler near the end of a round to see if the same job will run in the next round on the same worker. We call this a *lease renewal*. If the lease is not renewed, the iterator calls `save_checkpoint`. The scheduler can then launch another job on the worker.

Throughput Estimation. Gavel uses a similar technique to Quasar [63] to estimate colocated throughputs when using the optional space-sharing optimization (if they are not available a priori), mixing profiling with matrix completion. Matrix completion enables sparse low rank matrices to be reconstructed with low error [122, 46]. With matrix completion, Gavel is able to extrapolate measurements obtained through direct profiling on separate workers dedicated to profiling, and determine the job's most similar pre-profiled reference job. The throughput estimator can then use the reference job's throughput measurements as an initial throughput estimate. Gavel's throughput estimator is diagrammed in Figure 5.8.

5.7 Evaluation

In this section, we seek to answer the following questions:

Model	Task	Dataset / Application	Batch size(s)
ResNet-50 [84, 10]	Image Classification	ImageNet [64]	16, 32, 64, 128
ResNet-18 [84, 112]	Image Classification	CIFAR-10 [101]	16, 32, 64, 128, 256
A3C [123, 78]	Deep RL	Pong	4
LSTM [27]	Language Modeling	Wikitext-2 [119]	5, 10, 20, 40, 80
Transformer [164, 87]	Language Translation	Multi30k [69] (de-en)	16, 32, 64, 128, 256
CycleGAN [181, 111]	Image-to-Image Translation	monet2photo [181]	1
Recoder [124] (Autoencoder)	Recommendation	ML-20M [81]	512, 1024, 2048, 4096, 8192

Table 5.2: Models used in the evaluation.

- Do Gavel's heterogeneity-aware policies improve objective metrics in a physical cluster (§5.7.2) and in simulations of larger clusters (§5.7.3)?
- How do Gavel's policies scale? (§5.7.4)
- How well does Gavel's scheduling mechanism realize Gavel's heterogeneity-aware allocations? (§5.7.5)
- Is Gavel able to accurately estimate the throughputs of co-located jobs when using space sharing? (§5.7.6)

5.7.1 Experiment Setup

We run experiments on both a physical and simulated cluster.

Clusters. We run physical cluster experiments on a cluster with 8 V100s, 16 P100s, and 24 K80s. Simulated cluster experiments are run on a cluster with 36 GPUs of each type.

Traces. We run physical and simulated experiments on two types of traces: one where all jobs are available at the start of the trace and jobs are *not* subsequently added ("static"), and another where jobs are continuously added to the cluster ("continuous"). For the continuous trace, job arrival times are generated according to a Poisson arrival process with an inter-arrival rate λ. For the simulated experiments, we vary λ to show the extra load each heterogeneity-aware policy is able to sustain in steady state. We run 3 seeds for every λ, and show standard deviations. For the physical cluster

Trace	System	Objective	Physical	Simulation
Continuous	Gavel	Average JCT	3.4 hrs	3.7 hrs
Continuous	LAS	Average JCT	5.1 hrs	5.4 hrs
Static	Gavel	Makespan	17.7 hrs	17.6 hrs
Static	Gandiva	Makespan	21.3 hrs	22.1 hrs

Table 5.3: Comparison of end objective between physical experiment and simulation for two different traces. For the continuous trace, we measure the average JCT of 25 jobs in a steady-state cluster. For the static trace, we measure the total time needed to complete 100 jobs submitted at the start of the run. The heterogeneity-aware policies improve target objectives, and results on the physical cluster are in agreement with results on simulated cluster ($< 8\%$).

experiments, we use a single λ that keeps the cluster well-utilized in steady state. The online traces used in the simulated experiments have a variable number of jobs (at least 5000) and span 20-30 days. We measure the completion times of jobs with ID 4000 to 5000 to study steady state behavior (new jobs continue to be added until jobs of interest complete). Job types are uniformly sampled from the job table with 26 distinct job (or model) types, shown in Table 5.2. The online traces used in the physical experiments span a day and have 100 jobs.

The duration of each job *on a V100 GPU* is sampled from an exponential distribution: jobs have duration 10^x minutes, where x is drawn uniformly from $[1.5, 3]$ with 80% probability, and from $[3, 4]$ with 20% probability. Given the job's observed throughput on the V100 GPU, the number of training steps is then inferred by multiplying the throughput (in steps/sec) by the duration. This matches the process used by Gandiva [172]. For the simulated experiments, we show results in two regimes: one where all jobs use a single worker ("continuous-single"), and another where 70% of jobs request a single worker, another 25% request between 2 and 4 workers, and the remaining 5% request 8 workers, as observed in published traces from Microsoft [34] ("continuous-multiple").

Metrics. For fairness and FIFO policies, our target metric is average job completion time of steady-state jobs, which is the same metric used by related work [115, 79]. We also show finish time fairness (FTF) for policies that explicitly optimize for FTF. For makespan policies, our target metric is the time needed to complete a job batch. For cost-related policies, the metric is cost (in dollars), and the percentage of jobs that violate time SLOs.

5.7.2 End-to-End Results on Physical Cluster

For our physical cluster experiments, we run a heterogeneity-aware and a heterogeneity-agnostic fairness policy on a continuous trace, and a heterogeneity-aware makespan policy against a baseline that uses Gandiva's ad-hoc space sharing on a static trace. Results are shown in Table 5.3. Gavel's heterogeneity-aware policies improved average job completion time by **1.5**$\times$ and makespan by **1.2**$\times$.

Model	Overhead without lease renewals	Overhead with lease renewals
ResNet-18	0.94%	0.17%
ResNet-50	1.58%	0.25%
A3C	0.22%	0%
LSTM	2.91%	0.47%
Transformer	0.77%	0.11%
CycleGAN	0.77%	0.11%

Table 5.4: Overhead of using preemptive scheduling in Gavel, with and without lease renewals, and with a round duration of 6 minutes.

For the makespan objective, we do not run Gavel with space sharing; in theory, space sharing would additionally reduce makespan.

We also compare the real performance to simulations and observe that for both policies, the difference between metrics in simulation and on the physical cluster is small ($< 8\%$), indicating that our simulator has high fidelity.

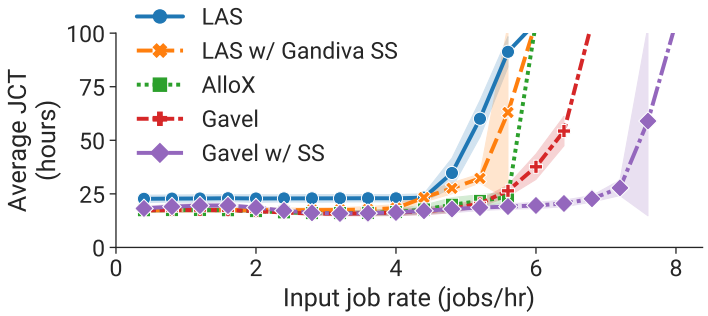
Table 5.4 shows the overhead of using Gavel's preemptive scheduler with a round duration of 6 minutes, with and without lease renewals. Allocations and worker assignments can be computed asynchronously. The only synchronous overhead is the loading and saving of checkpoints, which is dependent on the size of the model. Lease renewals decrease this overhead by allowing jobs to run on the same worker for extra rounds. The overhead of preemption, even without lease renewals and with a short round duration, is low ($< 3\%$).

5.7.3 End-to-End Results in Simulation

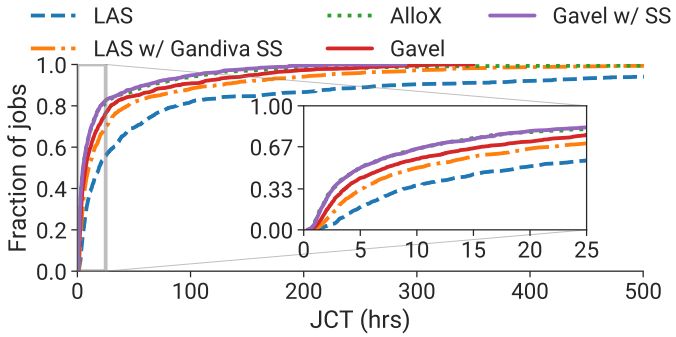
We use a larger simulated cluster to evaluate the efficacy of Gavel's heterogeneity-aware policies across a range of objectives, and compare with heterogeneity-agnostic versions from previous work using a round duration of 6 minutes. As appropriate, we compare to other baselines like AlloX. Magnitudes of speedups are higher for these experiments compared to the physical cluster experiments since the simulated traces show job behavior over weeks, while the physical cluster traces are only a day long; consequently, queue buildups are less extreme for the physical cluster experiments.

Least Attained Service (LAS). Figures 5.9 and 5.10 compare the vanilla LAS policy with its heterogeneity-aware variants. We compare with two other baselines: a modified LAS policy that uses Gandiva's ad-hoc space sharing, and an AlloX policy that explicitly optimizes average job completion time (but only for single-worker jobs). We make three observations.

First, the heterogeneity-aware policies support higher load on the *same* cluster, reduce average JCT by **3.5**$\times$ for the continuous-single trace, and by **2.2**$\times$ for the continuous-multiple trace (graph can be read by comparing average JCT value for a given input job rate or x-intercept) at high load

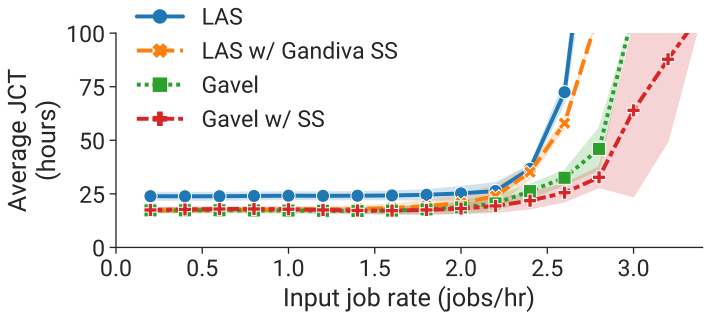


(a) Average job completion time vs. cluster load.

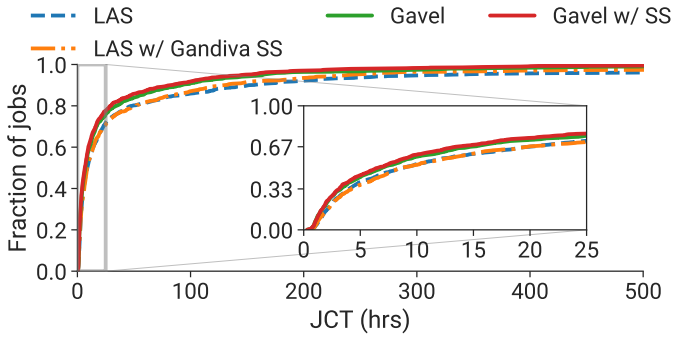


(b) CDF of job completion times (input job rate = 5.6 jobs/hr).

Figure 5.9: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-single trace. Each input job rate is run with 3 seeds.

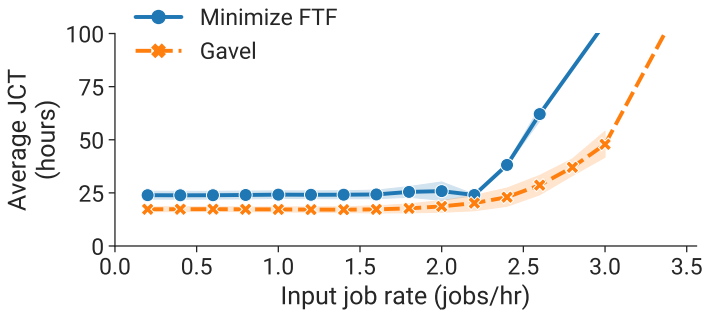


(a) Average job completion time vs. cluster load.

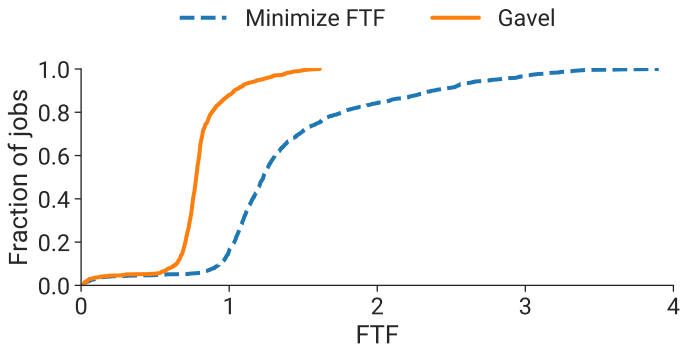


(b) CDF of job completion times (input job rate = 2.6 jobs/hr).

Figure 5.10: Comparison of heterogeneity-agnostic least attained service (LAS) policy to a heterogeneity-aware LAS policy (Gavel), in simulation on the continuous-multiple trace. Each input job rate is run with 3 seeds; shaded regions show the standard deviation.



(a) Average job completion time vs. cluster load.



(b) CDF of finish time fairness metric (input job rate = 2.6 jobs/hr).

Figure 5.11: Comparison of a heterogeneity-agnostic policy that optimizes for finish time fairness ("Minimize FTF") to a heterogeneity-aware one (Gavel), in simulation with the continuous-multiple trace. Each input job rate is run with 3 seeds.

(5.6 jobs/hr for continuous-single, 2.6 jobs/hr for continuous-multiple). Second, the heterogeneity-aware LAS policy supports higher load than AlloX, since AlloX can give short jobs preferential treatment in the interest of optimizing average JCT, leading to long jobs experiencing starvation (long tail in JCT CDF). At moderate load, AlloX represents a best-case scenario since it explicitly optimizes for average JCT on a heterogeneous cluster. Gavel is able to essentially match this best case scenario, while also supporting other objectives. Third, Gandiva-style packing, which randomly explores job combinations until a combination that improves performance is found, is ineffective compared to Gavel's principled packing (**2.2**× better average JCT for both traces at high load).

Finish Time Fairness (FTF). We compare the heterogeneity-aware version of Finish Time Fairness (FTF) to its heterogeneity-agnostic counterpart in Figure 5.11. The heterogeneity-aware policy reduces average JCTs by **3**× and improves average FTF by **2.8**×. FTF is the ratio of the time taken to finish a job using a given allocation and the time taken to finish the job using $1/n$ of the cluster (X^{isolated}), assuming n users use the cluster. Lower FTF means jobs take less time with the provided

allocation compared to X^{isolated}.

Makespan. Gavel's heterogeneity-aware makespan policy reduces makespan by **2.5**× compared to a FIFO baseline, and by **1.4**× compared to a baseline that uses Gandiva's ad-hoc space sharing. Makespan is reduced by a further **8%** when using space sharing with a high number of jobs.

FIFO. The heterogeneity-aware versions of FIFO allow the cluster to support average input job rate. At high load, the heterogeneity-aware version without space sharing reduces average JCT by **2.7**×, and the heterogeneity-aware version with space sharing reduces average JCT by **3.8**× at high load. Space sharing is less effective for distributed jobs: it reduces average JCT by **1.1**× with distributed jobs, compared to **1.4**× for the continuous-single trace.

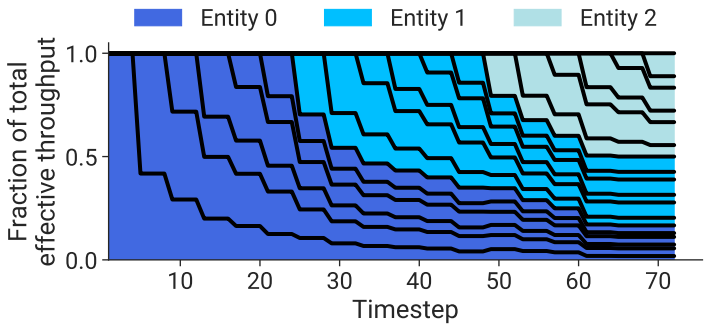
LAS with Priorities. We also run an experiment with the LAS policies where 20% of jobs have higher priority. At high load, Gavel reduces the average JCT of high-priority jobs by **1.5**× and the average JCT of low-priority jobs by **2.7**×.

Cost. We simulate each of the cost policies on a 500-job workload comprised of ResNet-50 and A3C jobs. As we observe in Figure 5.1b, the ResNet-50 job has the best cost-normalized throughput on the V100 while the A3C job has the best cost-normalized throughput on the K80. Job durations are chosen from $\{0.5, 1, 2, 4, 8\}$ days, and job SLOs are chosen from $\{1.2\times, 2\times, 10\times\}$ job duration.

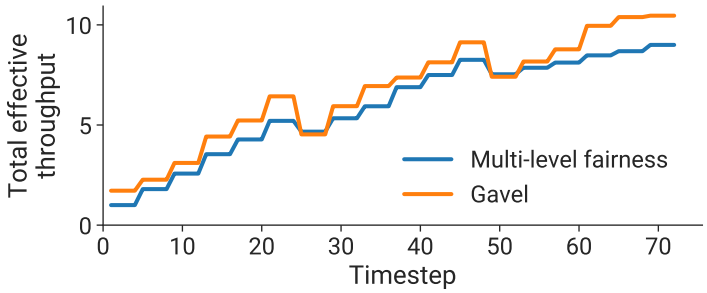
The policy that minimizes cost reduces the total cost compared to the policy that maximizes throughput by a factor of roughly **1.4**×. However, approximately **35%** of jobs violate their SLO as this policy prioritizes cheaper but slower GPUs; in particular, the A3C jobs are scheduled on K80 GPUs which results in violations for tight SLOs. In comparison, the policy that includes SLOs as well eliminates all violations for a small increase in cost (a cost reduction of **1.2**× compared to the baseline policy), by ensuring that A3C jobs with tight SLOs are run on instances with V100 GPUs.

Multi-level Hierarchical Policies. Figure 5.12 shows the behavior of a multi-level fairness policy as new jobs belonging to multiple entities are added to a heterogeneous cluster with equal numbers of K80, P100, and V100 GPUs. Resources are granted to jobs in a way that respects both the higher-level and lower-level policies: in Figure 5.12a, fairness is enforced both within and across entities (as can be seen by the widths of the colored bands, which represents cross-entity fairness, and the widths of bands within a color, which represents fairness across jobs within an entity), and allocations are adjusted as new jobs come in. Figure 5.13 shows results with a fairness+FIFO policy; later jobs in each entity 0 do not receive any GPU time to respect the per-entity FIFO policy.

The multi-level fairness policy can also be implemented in a heterogeneity-agnostic manner by statically partitioning resources across users while respecting per-entity and per-user weights. While



(a) Fraction of total throughput for each job with time.



(b) Total throughput vs. time.

Figure 5.12: Behavior of a multi-level fairness policy with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).

this results in a fair allocation as well, we observe that total effective throughput is about **17%** lower compared to the heterogeneity-aware policy (Figure 5.12b).

5.7.4 Scalability of Heterogeneity-Aware Policies

Figure 5.14 shows the scaling behavior of the heterogeneity-aware LAS and multi-level fairness policies with and without space sharing. We observe that even with 2048 active jobs, the hierarchical policy without space sharing can be run in < 10 minutes. With space sharing, the policy can be run with 512 jobs in < 10 minutes. The single-level LAS policy is much cheaper to compute in comparison. We note that allocations do not need to be recomputed every scheduling round – however, the longer the policy takes to run, the longer it takes for the new allocation to be acted upon (jobs can still be given heterogeneity-agnostic allocations in the interim, and consequently time on resources). We believe latencies of < 30 minutes for large clusters are still preferable to non-preemptive schedulers where jobs experience large queuing delays, or preemptive schedulers with heterogeneity-agnostic policies which lead to worse objective values, as shown above. We

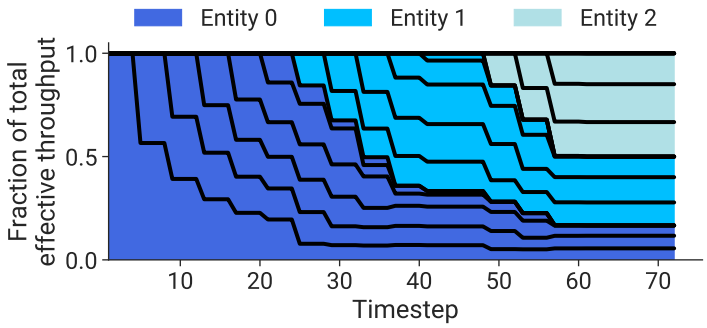


Figure 5.13: Behavior of a hierarchical policy (weighted fairness as top-level policy, FIFO as bottom-level policy) with time as jobs are added to a small cluster with 3 V100 GPUs, 3 P100 GPUs, and 3 K80 GPUs. Each line represents a separate job, and jobs are added every 4 timesteps. The first 6 jobs belong to entity 0 (weight of entity, $w_0 = 1$), the next 6 jobs belong to entity 1 ($w_1 = 2$), and the last 6 jobs belong to entity 2 ($w_2 = 3$).

believe approaches like POP [126] can make this process even more efficient, allowing scaling to larger clusters and more jobs.

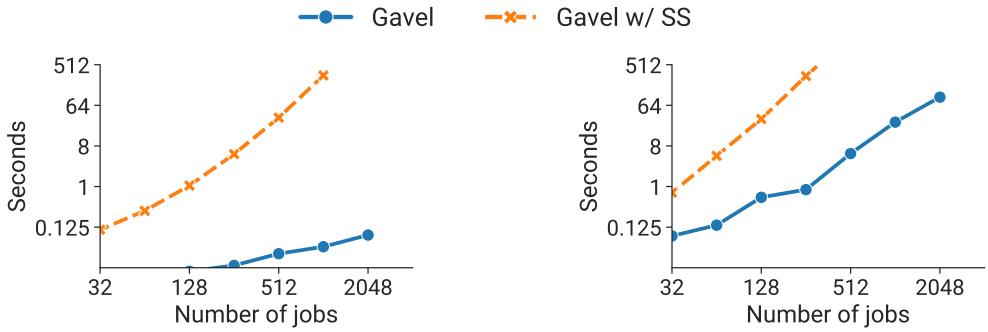
5.7.5 Efficacy of Scheduling Mechanism

Figure 5.15a shows the effect of the round length on average JCT for the heterogeneity-aware LAS policy with a single-GPU trace. We observed similar behavior on traces with multi-GPU jobs, as well as other policies. A smaller round length gives Gavel's scheduling mechanism more rounds to course correct, allowing the true allocation and computed optimal allocation to more closely match. We found that the time needed to load and save checkpoints for our target models is < 5 seconds, which means that a round length of 6 minutes gives a good tradeoff between fidelity with the optimal allocation and preemption overhead (preemption overhead shown in Table 5.4).

We compare this to an ideal baseline that allocates resources to jobs *exactly* according to their computed allocation. As shown in Figure 5.15b, Gavel's scheduling mechanism with a round duration of 6 minutes behaves almost identically to this ideal baseline with a single-GPU trace (behavior with a multi-GPU trace is similar). We note that the ideal baseline is impractical to use in practice, since jobs with different scale factors can complete at different times (leading to starvation), and preemptions can be often since allocations for some (job, accelerator type) pairs are small, leading to high overhead.

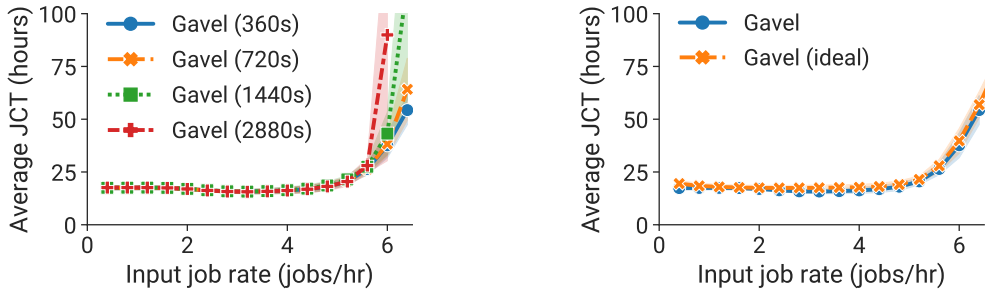
5.7.6 Impact of Throughput Estimation

Figure 5.16 shows the effect of Gavel's throughput estimator on average JCT when using the space sharing-aware LAS policy compared to the LAS policy without space sharing, and the LAS policy



(a) LAS. (b) Hierarchical.

Figure 5.14: Scaling of LAS and hierarchical policies with the number of active jobs on a heterogeneous cluster with an equal number of V100, P100, and K80 GPUs. The size of the cluster is increased as the number of active jobs is increased.



(a) Effect of round length. (b) Mechanism vs. ideal.

Figure 5.15: (a) Effect of round length on average JCT for the heterogeneity-aware LAS policy. (b) Comparison of scheduling mechanism to an ideal baseline that allocates resources to jobs *exactly* according to the computed allocation for the same policy.

with space sharing and oracle throughputs. The throughput estimator is able to determine missing throughputs in an online fashion accurately enough to observe a very small decrease in average JCT at high load (orange and blue lines).

5.8 Related Work and Discussion

In this section, we compare Gavel to related work.

Existing DNN Training Schedulers. Several recent papers have proposed schedulers targeting DNN training workloads.

Figure 5.16: Comparison of SS-aware LAS policy with estimated throughputs, compared to the SS-aware with oracle throughputs and LAS without space sharing on a heterogeneous 12-GPU cluster.

Gandiva [172] uses time and space sharing to reduce queuing delay and improve resource utilization, but does not specify an explicit scheduling policy and does not support configurable objectives. It uses a profiling-based methodology to determine whether to co-locate jobs on an accelerator. However, it does not incorporate model performance data (isolated or co-located performance) explicitly into its scheduling policy, resorting to random exploration of job combinations until a combination that improves performance is found.

Tiresias [79] and Themis [114] use different objectives to achieve multi-job fairness. However, both do not incorporate jobs' affinities for different accelerator types in their scheduling objectives, and have scheduling mechanisms strongly coupled with the target policy, making it hard to support other more sophisticated policies like multi-level fairness.

AlloX [106] and Gandiva$_{\text{fair}}$ [48] are recent DNN schedulers that do consider worker and model heterogeneity. However, both only work for single policies (average job completion time for AlloX, max-min fairness for Gandiva$_{\text{fair}}$). Moreover, Gandiva$_{\text{fair}}$ uses a second-price auction mechanism to improve the performance of a heterogeneity-agnostic max-min fairness scheme, but does not provide guarantees as to the optimality of the final allocation. On the other hand, Gavel formalizes each policy as an optimization problem, and can provide a guarantee that the returned solution is "optimal" according to the provided objective. Gavel is also able to support more sophisticated policies such as multi-level fairness.

Traditional Cluster Schedulers. Traditional schedulers such as Mesos, Borg, TetriSched, and YARN [85, 168, 161, 165] support workloads with fixed heterogeneous resource requests, but do not reason about the performance characteristics of jobs across accelerators. Mesos and YARN do not reason about interchangeable resource types that can run the same computation: for example, Mesos's DRF multi-resource sharing policy [74] decides how to give jobs allocations of distinct resource types, such as RAM and CPUs, but assumes that each job has declared which resources it needs to use and in what ratio.

The multi-interchangeable resource allocation (MIRA) problem [158] also introduces the notion of effective throughput, but does not demonstrate how this can be used to specify policies as optimization problems, does not consider performance optimizations like space sharing and placement sensitivity, and does not discuss how computed allocations can be realized on physical resources.

Omega [145], Apollo [44], and Hydra [61] are schedulers that take into account the fact that the target workload shows heterogeneity in the number and duration of constituent tasks. However, tasks largely take the same time on different CPUs, and heterogeneity in memory capacities only impacts the number and size of tasks that can be placed on a server. In our work, the compute devices themselves are interchangeable with sometimes large performance differences, and policies decide the time fractions of resources each job should receive while optimizing various end objectives.

Dynamic Performance Estimation. Gavel uses the approach proposed by Quasar [63] to estimate co-located job performance online (§5.6). In particular, Gavel uses a mix of profiling and matrix completion to compute a "fingerprint" against a set of reference models profiled offline. In this work, we show that the techniques used by Quasar can be successfully applied to this new setting.

Applicability to Other Settings. Even though Gavel was explicitly targeted at allocating heterogeneous resources for DNN training workloads, we believe that Gavel can be used for non-DNN workloads as well. Other workloads that are amenable to GPU execution, such as simulations, can be considered, even though performance estimates for these applications will be needed. We also believe the main technical insight presented in this chapter – formulating diverse scheduling policies as optimization problems – is broadly applicable, and can be used to more easily deploy policies on homogeneous deep learning clusters, and on CPU clusters as well.

5.9 Summary

In this chapter, we proposed Gavel, a heterogeneity-aware cluster scheduler that is able to optimize for many high-level metrics like fairness, makespan, and cost. Gavel demonstrates how existing policies can be expressed as optimization problems, and extends these policies to be heterogeneity-aware. Gavel then uses a decoupled round-based scheduling mechanism to ensure that the optimal allocation is realized. Gavel's heterogeneity-aware policies improve end objectives both on a physical and simulated cluster. It can support a higher average input job rate, while improving objectives such as average job completion time by 3.5×, makespan by 2.5×, and cost by 1.4×.

Chapter 6

Exploiting Dynamic Pricing for Training in the Public Cloud

6.1 Introduction

Cloud providers like AWS, GCP, and Azure provide an opportunity for users to rent instances of many different types, in multiple regions and availability zones. In addition to reserved and on-demand cloud markets for long-term and guaranteed instances, many cloud providers offer a market for accessing unclaimed machines at lower cost, often referred to as the *spot market*. These instances are priced independently and dynamically, according to instance-specific supply and demand. In this chapter, we explore the following question: *how much can a user benefit from a dynamic multi-cloud instance market?*

The primary challenge in taking advantage of spot pricing is that spot instances can be reclaimed or preempted at any time. Applications running on spot instances thus need to be easily stoppable; applications would then be restarted on another instance. DNN model training is a good example of an application suitable for spot instances; its iterative nature makes it conducive to preemption. DNN training is also compute-heavy and uses expensive instances with accelerators, and often uses a static read-only training data set that can be easily copied across clouds and availability zones. Using DNN training as a target workload, we focus on answering three important questions.

How should cloud instances be chosen? A DNN model can be trained in the cloud using many instance types, with different accelerators (e.g., GPU generations like the K80, P100, V100; dedicated ML chips like the TPU [97]) and varying prices. DNN models are extremely diverse with many operator types, and show widely different performance behavior across instance types. The most appropriate choice of instance type depends on the *model* as well as the *user's objective* (e.g.,

throughput, cost, or a combination of the two, such as minimizing cost subject to a performance SLO like "complete job X in 10 hours").

Furthermore, spot instances, which are a cheap alternative to on-demand instances, are dynamic:

- Instances are priced differently across regions, availability zones, and cloud providers. These prices change with time as supply and demand change.
- A spot instance may be preempted at any time.
- Instances with multiple accelerators may be in less demand compared to an instance with a single accelerator of the same type, and consequently cheaper on a per-accelerator basis.

All these factors influence the optimal instance choice.

How should higher-level objectives over multiple jobs be taken into account? Many organizations use public cloud instances to train models with the latest data on a repeated (e.g., daily) schedule. In such a use case, cost may not be the only objective to optimize for, e.g., some important jobs might have strict deadlines that must be met, even at a higher cost.

How can real systems realize these cost-saving opportunities? Leveraging the spot market comes with many practical challenges, including dealing with instance preemption, determining how to schedule jobs on instances while respecting the computed allocation, responding to price changes, and transparently allowing movement of jobs between instances without user intervention. We touch on these challenges in §6.5.

Summary of Contributions. We measured the cost benefits of leveraging the dynamic multi-cloud instance market using AWS, GCP, and Azure instance prices collected over a month. We highlight the following key takeaways:

- The optimal instance type for a given model is dependent on both the target objective (cost, speed, or both) and performance characteristics of the model, even when using statically-priced instances.
- The cost of moving model checkpoints between instances is cheap. Moving input datasets is more expensive, but can be amortized over many jobs.
- Jobs do not need to be preempted more frequently than once a day to leverage the benefits from spot instance price variations. We observe that cloud providers today change instance prices at a much coarser granularity than before [30, 151]; this affects how systems leveraging the dynamic spot market should be designed.

- Instances themselves are usually preempted fairly infrequently (on the order of hours). In such cases, recent systems such as Spotnik [169], which provides fine-grained resilience to transient instance failures for distributed training, are not needed.
- The cost of training a model can be reduced by up to 3.5× (in practice, thousands of dollars) by making use of all available sources of price variation, including by up to 1.4× when enabling movement of applications across instances mid-computation.

Code and pricing data are open sourced at `https://github.com/stanford-futuredata/training_on_a_dime`.

6.2 Background

In this section, we provide background on DNN training and instance pricing in the public cloud.

Deep Neural Network (DNN) Training. DNN training proceeds in iterations. In each iteration, the model processes a collection of training data inputs (called a batch), and subsequently updates its parameters using gradients derived from the batch. If training were interrupted, the model's parameters would need to be *checkpointed* to stable storage; state-of-the-art DNNs can have millions to billions of parameters. These model checkpoints then need to be loaded on the new worker to ensure that training progress is not lost. On-premise DNN schedulers leverage the fact that DNN training is iterative to suspend and resume training at iteration boundaries [79, 172].

Pricing in Public Clouds. Cloud providers allow compute instances to be rented by users at fine granularities. The standard way to rent instances from public cloud providers involves using *on-demand* instances, which are guaranteed to be available at all times. Instances are hosted in different *regions*; each region has multiple availability zones.

Using on-demand instances for long durations can be expensive. As a cheaper alternative, cloud providers offer spot or preemptible instances, which can be preempted with little warning. Cloud providers usually price these instances in one of two ways: either the spot price changes (capped at the on-demand price) as demand changes (AWS and Azure), or the instances are offered at a constant price and can only be run for 24 hours or less (GCP).

6.3 Quantitative Analysis of Cloud Pricing

In this section, we pose two questions in the context of training various DNN models on instances with accelerators in the public cloud:

1. How should users go about picking which instance and accelerator type to use?

Model	Throughput		Dollar-norm. Throughput	
	P100	V100	P100	V100
Transformer	**3.3**×	**3.3**×	**1.0**×	0.8×
A3C	1.2×	**2.2**×	0.4×	0.4×
CycleGAN	4.5×	**9.3**×	1.4×	**1.7**×
ResNet-18	4.0×	**6.8**×	**1.2**×	**1.2**×
ResNet-50	3.7×	**9.6**×	1.1×	**1.8**×

Table 6.1: Throughput and dollar-normalized throughput (using GCP on-demand prices) speedups with respect to a NVIDIA K80 GPU for various ML training workloads. The magnitude of speedup across GPU generations varies significantly across models, with later GPU generations (V100) faster. The V100 is no longer always optimal when considering dollar-normalized throughputs; dollar-normalized speedups are smaller across all models.

2. Can jobs leverage the fact that instance pricing is dynamic and changes across cloud providers, regions, availability zones, and over time, to achieve *better* allocations, as defined by the user's desired objective, by moving between instances (on the same or different cloud) over the course of training? Is this practical, given the overheads of moving model checkpoints and the associated input dataset?

6.3.1 Instance Type Choice for Various Models

Cloud providers like AWS, GCP, and Azure offer instances with various GPU types. Models use a diverse set of operators, leading to vastly different performance behavior on these hardware architectures. Table 6.1 shows the observed throughput speedups for various models and GPU types compared to a NVIDIA K80 GPU. While one of NVIDIA's more recent GPU offerings, the V100, outperforms other GPUs for every model type, the relative speedup compared to the older K80 GPU is *model-dependent*, and varies from 2.2× to 9.6×. However, instances with V100 GPUs also cost more than instances with K80 GPUs.

The cost effectiveness of instances for a particular model can be compared using the model's *cost-normalized throughput*. When normalizing by the GCP on-demand price (we use GCP since AWS does not offer P100 GPUs), we see that the K80 and P100 GPUs are superior compared to the V100 GPU for certain models, like A3C [78] and Transformer [87]. The best GPU for a given model on a cost basis can also change over time if using spot instances, which have dynamic pricing.

Moreover, users might have more nuanced deployments, where they have both cost and time budgets; in such situations, we may want to switch between instance types partway through training. For example, an optimal schedule may have a job spend 60% of training time on a cheap K80 GPU and the remaining 40% on a faster V100 GPU to minimize cost while still ensuring that the provided time budget is respected.

Model	Dataset Size (GB)	Model Size (GB)	Dataset Cost	Model Cost
ResNet-50	150	0.098	9.13%	0.006%
BERT-Base	17	0.408	0.98%	0.025%

Table 6.2: Dataset and model sizes for ResNet-50 and BERT-Base architectures, along with the compute cost and egress costs (as a fraction of compute cost) for a single dataset and model transfer. Each transfer is from a North American region to the Internet. Each model transfer is extremely cheap. Dataset transfers are more expensive, but need to be performed only once per (dataset, cloud provider) pair.

6.3.2 Leveraging Dynamic Pricing to Reduce Costs

We now consider the various costs incurred when dynamically moving training jobs between instances within the same cloud provider or even across cloud providers.

Cost of Data Movement between Clouds

Moving workloads between instances is only economical if the cost of the associated data transfer is less than the compute cost reduction from switching to the new instance.

Table 6.2 lists the dataset and model sizes for two commonly benchmarked models (ResNet-50 [84] and BERT-Base [66]), as well as egress costs as a fraction of the cost of training these models for 160 hours on V100 spot instances. We use ImageNet [64] as the ResNet-50 dataset and English Wikipedia [32] as the BERT-Base dataset. The compute cost is measured as the cost of 160 V100-hours using spot instances. We use AWS prices for these measurements but find similar results on GCP and Azure. We approximate the cost of a single model transfer by computing the cost of 10,000 model transfers and dividing by 10,000. Ingress into each cloud is free, and does not need to be accounted for.

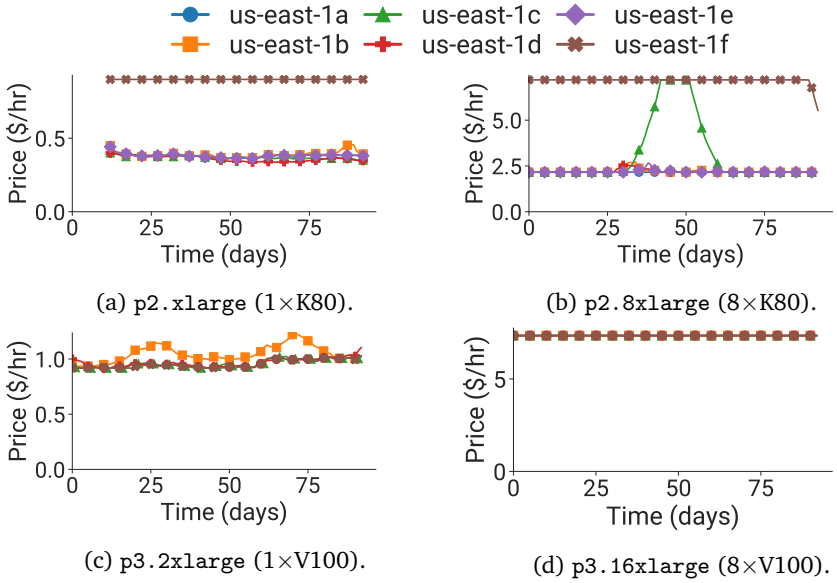
We observe that we can feasibly perform hundreds of transfers for each model before reaching even 10% of the compute cost, since the cost of transferring a single model checkpoint is cheap (on the order of cents). Furthermore, while a single dataset transfer is far more expensive than transferring a model checkpoint, the dataset need only be transferred once to each cloud during training and can be amortized over many jobs that use the same dataset. This transfer cost is zero if the user already has a copy of the input dataset available on all target clouds.

Volatility in Spot Instance Pricing for Compute

We collected spot instance prices for AWS and Azure over a month in February 2020; we were able to collect 3 months of backfilled data for AWS. We only include the most interesting graphs in this section; more graphs from our analysis are available at `https://github.com/stanford-futuredata/training_on_a_dime`.

Cloud Provider	Region	GPU Type K80	P100	V100
Amazon (AWS)	`us-east-1`	2.7×	N/A	3.3×
Google (GCP)	`us-west-1`	3.4×	3.4×	3.3×
Microsoft (Azure)	`us-east-1`	7.3×	8.0×	5.1×

Table 6.3: Best-case cost reduction moving from on-demand instances to spot instances with a single GPU on each cloud. The best-case cost reduction varies widely with cloud provider; however, as we show later in Figure 6.2, availability also varies with cloud provider and instance type.



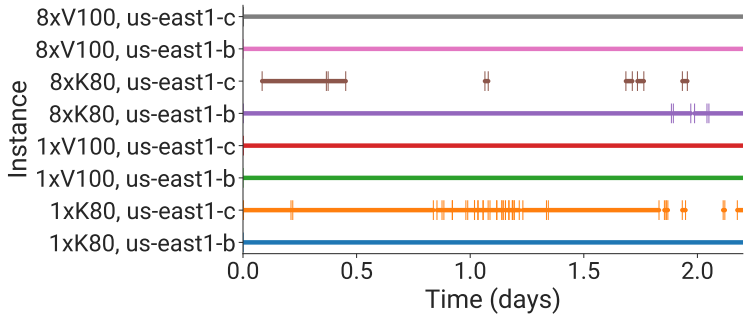
(a) `p2.xlarge` (1×K80). (b) `p2.8xlarge` (8×K80).

(c) `p3.2xlarge` (1×V100). (d) `p3.16xlarge` (8×V100).

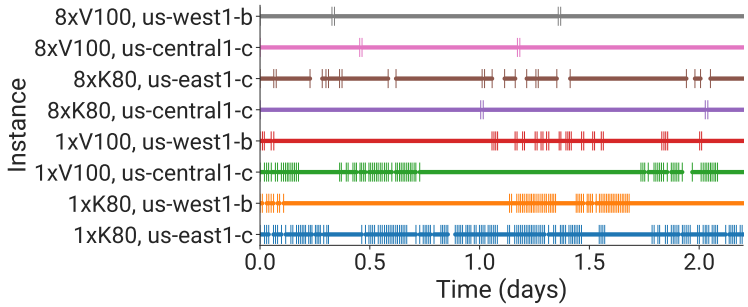
Figure 6.1: Per-hour price of AWS spot instances with various GPU accelerators in the `us-east-1` region. Prices can change with time and across availability zones, and are often capped at the on-demand price (`p2.xlarge`, `us-east-1f`). Some instances (`p3.16xlarge`) exhibit no price variation.

Cost Reduction from Spot Instances. Table 6.3 shows the best-case cost reduction observed when moving from an on-demand instance to a spot instance in the same region, for different clouds. Cost reductions vary from 2.7× to 8×.

Variation of Spot Price with Time. The price of spot instances can change with time as demand changes. Figure 6.1 shows the variation in spot prices for various instances with GPUs in the AWS `us-east-1` region. We observe that price changes across regions are not highly correlated with each other, with some regions capped at the on-demand price. The cheapest availability zone in a region can change with time. We also observe that some instances show extremely stable pricing (`p3.16xlarge`).



(a) AWS.

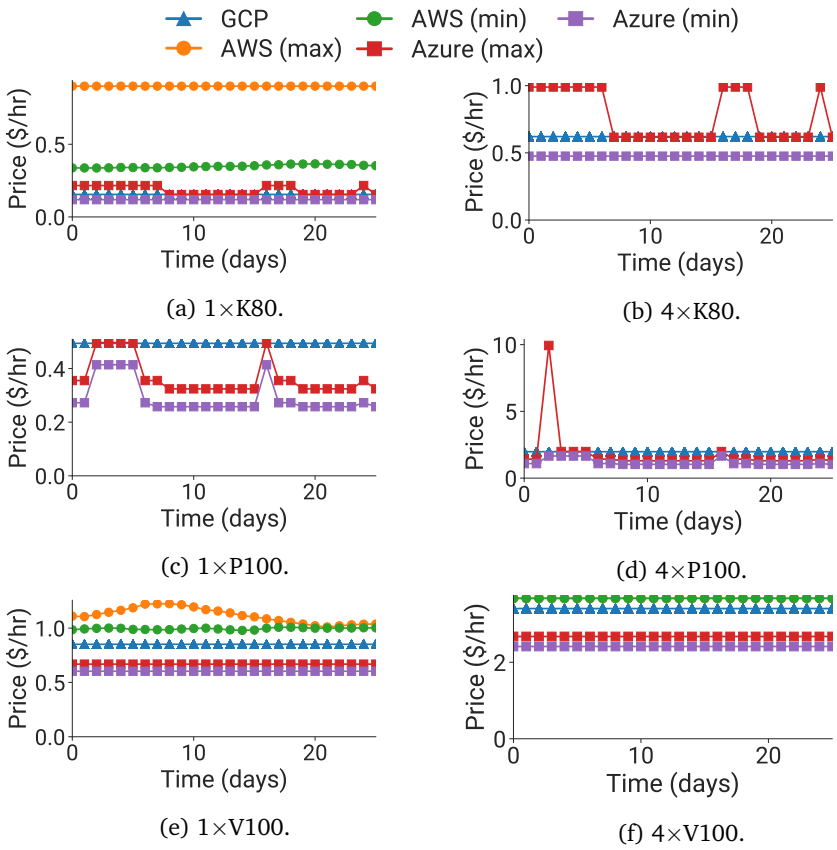


(b) GCP.

Figure 6.2: Availability of AWS and GCP preemptible instances. Vertical lines at the start of a horizontal line show the time at which the request was granted, and vertical lines at the end of a horizontal line show the time at which the instance was preempted. The frequency of preemption changes with both availability zone and instance type. GCP preempts instances at least every day.

Availability. GCP adopts an alternate pricing model for preemptible instances: prices stay constant, but instances might be preempted when demand exceeds supply. Figure 6.2 shows timelines of availability for instances with GPUs on AWS and GCP. Instances on AWS are more reliably available for longer (not capped at 24 hours). Instances in some regions were preempted more often than others (greater frequency of vertical lines); 8×GPU instances were preempted less frequently on GCP. Preemption is preceded by a 2-minute warning which can be used to checkpoint the model. For most regions and instance types on AWS, preemption is relatively infrequent (order of hours instead of minutes).

Instance Prices across Clouds. Figure 6.3 shows the price of the cheapest and most expensive instances with different numbers of accelerators across clouds. The cheapest cloud provider changes with instance type. In some cases (not shown), GCP is the cheapest option, but jobs are preempted after at most 24 hours.



(a) 1×K80.

(b) 4×K80.

(c) 1×P100.

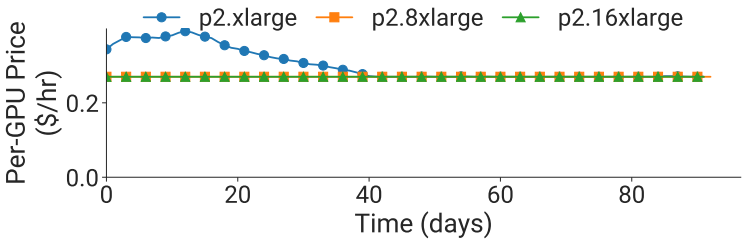
(d) 4×P100.

(e) 1×V100.

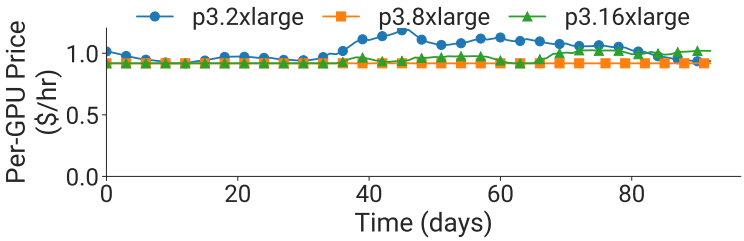
(f) 4×V100.

Figure 6.3: Minimum and maximum spot price over all availability zones and regions in the US for various cloud providers. GCP uses a static pricing model. Instance types have different relative orderings, and at any given time, the ordering can change (e.g., as in Figure 6.3d).

Per-GPU Price for Multi-GPU Instances. We also studied the variation of price on a per-GPU basis across instances with different numbers of the same GPU type (e.g., AWS has 1×, 8×, and 16×K80 instances). As shown in Figure 6.4, we found that on a per-GPU basis, instances with a larger number of GPUs have more stable pricing. However, a user may need to pack multiple jobs onto the larger instance (or run a single multi-GPU job) to fully utilize it.



(a) K80.



(b) V100.

Figure 6.4: Normalized cost on a per-GPU basis for instances with K80 and V100 GPUs. Instances with K80 GPUs have 1, 8, and 16 GPUs, while instances with V100 GPUs have 1, 4, and 8 GPUs. We found that instances with a greater number of GPUs generally exhibit more stable pricing.

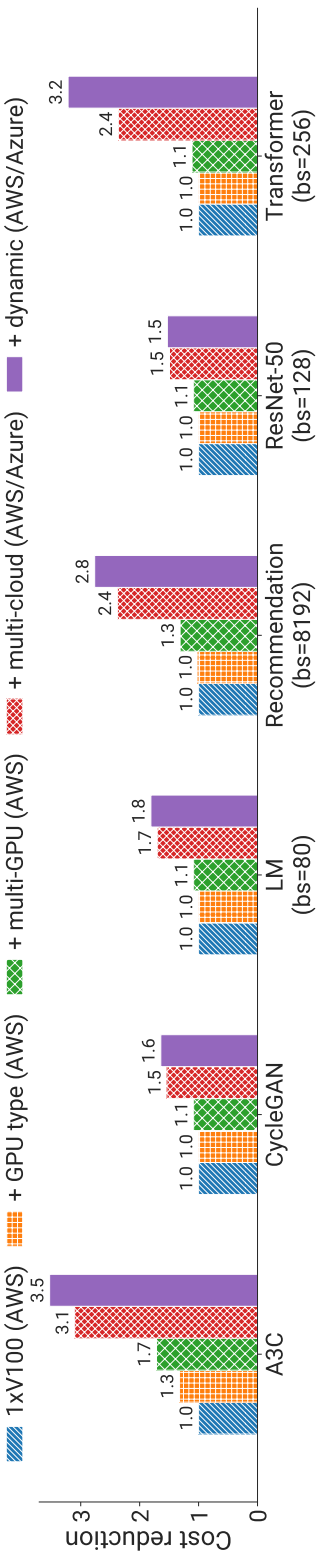


Figure 6.5: Average cost reduction to run the same number of training iterations (4 V100-days of computation), while cumulatively adding more sources of price variation. **1×V100** uses the cheapest 1×V100 instance within the `us-east-1` AWS region, **GPU type** chooses the GPU with highest cost-normalized throughput, **multi-GPU** picks instances with multiple GPUs if they are cheaper on a per-GPU basis; all these strategies use AWS instances only. The **multi-cloud** strategy picks the cheapest instance across AWS and Azure at the start of training, and then sticks with this choice throughout training. **Dynamic** continually picks the cheapest instance across AWS and Azure through training as prices change. Costs reduce as sources of price variation are added.

Figure 6.6: Average cost reduction from allowing *dynamic* switching of instance type, cloud, and availability zone during training, while varying job duration. Longer jobs are able to make use of greater variability in prices over longer horizons, consequently leading to larger cost reductions. The right two bars in Figure 6.5 shows the impact of dynamic switching for jobs with a duration of 4 V100-days.

End-to-End Cost Reduction

We show the net reduction in *compute* cost of training a single ML model using all these sources of price variation in Figure 6.5. Each ML training job takes 4 days to complete, and we show price reductions for single-GPU jobs for simplicity. All strategies before *multi-cloud* use AWS instances with GPUs in the `us-east-1` region; *multi-cloud* and *dynamic* use the cheapest instance available across AWS and Azure. *GPU type* chooses the GPU with best cost-normalized throughput (instead of 1×V100 instances) when the job starts and then sticks with that choice throughout, *multi-GPU* picks instances with multiple accelerators if they are cheaper on a per-GPU basis, and *dynamic* adapts the choice of instance through training as prices change. All results assume that datasets are available on each cloud (dataset movement cost is 0).

We can reduce costs by up to 3.5× compared to the baseline of using the cheapest 1×V100 instance. The effectiveness of each strategy depends on the GPU type where the model has the highest cost-normalized throughput (Table 6.1), which can change with time depending on the pricing behavior of these instance types across AWS and Azure. For example, ResNet-50 [84] is always cheapest on V100 instances, which show stable pricing; consequently, cost reductions are minimal. We note that the movement of checkpoints is extremely cheap (cents / transfer) and the number of transfers is small, since prices change only daily and not every price change leads to an instance switch.

Impact of Job Duration on Effectiveness of Dynamic Scheduling. We further study the impact of job duration on cost savings when using dynamic scheduling, where jobs can be moved between instances as training proceeds and the initial instance choice is not locked in through the duration of training. In Figure 6.6, we show the cost reduction of switching instances across GPU types, availability zones, and clouds during training as job duration changes compared to using the best option across cloud providers at the start of training and sticking with this choice (red and purple

bars in Figure 6.5). We see a cost reduction of up to 1.4× for long-duration jobs that can take advantage of pricing over longer horizons. Long-duration training jobs are common as models become larger. For example, the recently released GPT-3 model [45] requires about **100 V100-years** of total training computation.

Cost reductions vary across models since cost-normalized throughputs for different models can change with time, e.g., the Transformer model switches between the Azure K80 and P100 instances. Cost reductions are small for short-duration jobs since instance pricing is stable over the short term ($\leq$ 2 days). The number of switches between instances needed for these cost savings is small ($\leq$ 3). We note that even though we only looked at single-GPU jobs in this section, the cost savings are valid even for multi-GPU jobs. In particular, the durations of distributed jobs which use many GPUs is still often on the order of weeks to months [45].

6.4 Higher-Level Objectives

When training a collection of ML models, users might want to allocate resources while optimizing for higher-level objectives. For example, users might want to minimize cost alone, or minimize cost subject to performance SLOs (e.g., complete training in the next 12 hours), or minimize the time needed to complete a collection of training jobs with a given cost budget.

Representing Allocations and Throughputs. As we noted earlier, optimizing more complex objectives might result in allocations where jobs move dynamically between instance types. As in the previous chapter, allocations can be specified as the fraction of wall clock time a training job should spend on each instance type (represented as X), and scheduling policies can be expressed as optimization problems involving X that try to maximize or minimize an appropriate objective function. Objective functions can again be written in terms of *effective throughput*, the time-weighted average throughput across instance types; given the relative performance of each job on each instance type (T), the effective throughput of a model m, throughput$_T(m, X)$, is simply $\sum_j T_{mj} \cdot X_{mj}$.

6.4.1 Baseline: Maximizing Total Throughput

Maximizing the total effective throughput achieved by a collection of jobs can be achieved by solving the following optimization problem:

$$\text{Maximize}_X \sum_m \text{throughput}_T(m, X).$$

We add the following constraints to ensure that each job is not over-allocated, and worker quotas are not exceeded.

$$\begin{aligned} \textstyle\sum_j X_{mj} &\leq 1 \qquad \forall m \\ \textstyle\sum_m X_{mj} &\leq \text{quota}_j \quad \forall j \end{aligned}$$

6.4.2 Minimizing Total Cost

The above policy can be extended to incorporate cost. To minimize training cost, one can optimize:

$$\text{Maximize}_X \sum_m \frac{\text{throughput}_T(m, X)}{\text{cost}(m, X)}.$$

Here, cost(m, X) is effective cost, computed as $\sum_j c_j \cdot X_{mj}$, where c_j is the per-hour cost of instance type j. The numerator in each objective term represents the effective throughput in samples per unit time, the denominator represents the effective cost in dollars per unit time, and the resulting fraction is the effective normalized throughput in samples per dollar. As before, constraints are needed to ensure that a job is not over-allocated resources, and worker quotas are not exceeded.

6.4.3 Objectives with Both Throughput and Cost

Jobs can have time SLOs as well, e.g., certain high-priority jobs might need to complete by a certain cutoff time. To satisfy these SLOs, we can add additional constraints given SLO_m for each model m (models without SLOs can have SLO_m set to ∞):

$$\text{throughput}_T(m, X) \geq \text{num\_iterations}_m / \text{SLO}_m.$$

Similarly, one could also formulate policies with a minimize makespan (time taken to complete all jobs in a collection) objective, while keeping the cost within a prescribed cost budget B. The objective here would be:

$$\text{Minimize}_X M.$$

M is the makespan. In addition to the constraints above that ensure that each job is not-allocated and worker quotas are not exceeded, we need constraints that ensure that every job completes within this makespan M, while also staying within the cost budget B,

$$\begin{aligned} \frac{\text{num\_iterations}_m}{M} &\leq \text{throughput}_T(m, X) \quad \forall m \\ M \cdot (\textstyle\sum_m \text{cost}_T(m, X)) &\leq B. \end{aligned}$$

This can be solved by binary searching for the smallest M which results in a feasible solution.

6.5 System Design Considerations & Discussion

In this section, we discuss important design considerations that real systems need to address to be able to deliver these cost reductions in a transparent way. We also highlight some open questions that we think are worth reflecting on.

Scheduling of Applications on Physical Instances. Given a theoretical allocation computed from a policy, how should resources be allocated to applications, considering quotas on instances and applications that span multiple accelerators? In multi-cloud settings, how should datasets be streamed between clouds when not already available? How should instance preemptions be handled?

API between the Scheduler and Applications. An application can be moved either when the scheduler decides to take advantage of a pricing change, or when a spot instance is preempted by the cloud provider. How can we enable the movement of applications between clouds, regions, and availability zones seamlessly without user involvement?

These questions are especially pertinent with distributed training where state, such as IP addresses of participating workers, needs to be reset when preemptions occur. Fortunately, both forced and voluntary preemptions are relatively infrequent (as can be seen in Figure 6.2 and §6.3.2), meaning the cost of reconfiguration can be easily amortized away without using sophisticated failover mechanisms like those proposed in Spotnik [169]. Recent work [132] has demonstrated how state in the Horovod communication library [149] can be reset with minimal user intervention when using elastic resources; similar techniques can be used for other communication libraries as well.

Instance Preemption. Spot instances are preempted at different rates (Figure 6.2). How should one model the preemptions of instances? This is important since users might be willing to pay more for a more reliable instance. Can we estimate the mean time to failure to decide which instance types to use?

Spot Instance Pricing. Our measurements raise the following questions about how spot instances are priced: Why do availability zones in the same region show different pricing? Why do instance preemptions happen even when the instantaneous spot price is lower than the on-demand price?

Market Movement. What happens if all cloud users exploit the cost inefficiencies described in this chapter, and use regions and availability zones with cheaper and / or more stable pricing? Can this help with price smoothing, with each of the different AZs showing more similar pricing as demand equalizes? In other words, will drastic changes in demand based on the movement of applications to cheaper regions and availability zones cause prices to shift?

Incentivizing Easier and More Efficient Multi-Cloud Deployments. In times of high demand, cloud providers can preempt spot instances. In such cases, it might make sense for a user to take their computation to a different cloud provider – this not only could give the user a better experience, but can also improve the experience of all other users by reducing demand and consequently the likelihood of preemption. An auction system where cloud providers can bid for a small fraction of another cloud provider's jobs could solve this problem – the original cloud can receive a small commission for forwarding the job to another cloud while also partially alleviating demand, the bidding cloud receives additional business that it might not have otherwise received, and users receive better service.

ML Inference. Even though we only considered ML training as a target application in this chapter, we believe ML inference is an interesting target application as well. ML inference, however, introduces different challenges: in particular, instances need to be provisioned keeping system load in mind, since system load has downstream ramifications on other metrics of interest like application latency. Unlike training, where users mostly care about just throughput and consequently total time needed to train a model end-to-end, inference applications have a number of performance-related metrics of interest, such as average latency, tail latency, throughput, and throughput subject to latency constraints. Each of these performance metrics can be combined with cost. How does one optimize for these different objectives? Additionally, serverless offerings such as AWS Lambda and Google Cloud Functions [29, 33] can be used in the inference context; however, these do not come with accelerators attached. Can inference on cheap CPU cores for short durations compete with more expensive but faster accelerators?

Packing Multiple Applications onto a Single Accelerator. Concurrently executing multiple models on the same GPU using NVIDIA's Multi Process Service (MPS), CUDA streams, or new features like Multi-Instance GPU (MIG) on the just released A100 GPU can help improve utilization [91, 35, 130, 17]. Can this be used to further reduce cost and improve resource utilization for end users?

Performance Modeling of Applications. Instead of relying on timing runs for each application on each instance type, can we learn a performance model that predicts runtimes of applications? Can we use this in settings where multiple applications are packed onto a single instance?

Other Applications. What other applications are long-lived and amenable to such optimizations? For example, are physical simulations a good fit? How can one get around the fact that performance in other applications might be less predictable, making optimization more challenging?

6.6 Related Work

Existing work has looked at two ways to minimize cloud costs: performance modeling for instance sizing, and leveraging the spot market. However, no prior work considers *both*; prior work also does not specify how objectives over multiple jobs can be specified and acted upon in this setting.

Minimizing Costs in the Cloud. Existing systems, such as LLOOVIA [68, 70] and other resource provisioning systems [157], have taken advantage of multi-cloud to minimize costs, but have focused on on-demand and reserved cloud markets. AWS offers EC2 Fleet [31], a service that can launch multiple on-demand and spot instances within a maximum budget. Other systems have proposed using spot instances for DNN training. DeepSpotCloud [107] takes advantage of price differences within availability zones and regions. HotSpot [151] and Stratus [56] are cost-aware schedulers that move CPU jobs between spot instances to take advantage of dynamic pricing. However, all of these systems use *pre-specified* instance types, do not account for application performance heterogeneity across instance types, and cannot determine the optimal instance type for a given job / objective.

Selecting Instance Types. Existing work has looked at picking the right instance type for different classes of applications. Ernest [166] and CherryPick [38] try to predict the runtime performance of various applications on instance types available in the cloud, but do not consider spot pricing of instances, and do not specify how these performance models can be used downstream to optimize for various higher-level objectives.

6.7 Summary

In this chapter, we analyzed the impact of the dynamic pricing market in public clouds on the cost of performing ML training. We found that moving jobs between instances is cheap, that jobs can be preempted fairly rarely (once a day) to leverage the benefits from price variations, that jobs themselves are preempted fairly rarely by the cloud provider, and that the cost of end-to-end training for a given model can be reduced by up to 3.5$\times$ by exploiting the different sources of price variation. We also showed how one can write policies that optimize combinations of speed and cost for collections of jobs. We believe this is is an exciting area of future work, with applications to many other domains besides ML training.

Chapter 7

Conclusions

7.1 Contributions

In this dissertation, we have shown that ML training is heterogeneous along both the workload (in terms of the target model) and hardware dimensions. Consequently, using the same optimization strategy in a model- and hardware-agnostic manner can result in sub-optimal performance. We have shown that careful automated scheduling of computation on possibly heterogeneous resources is useful in two broad problem contexts: distributed model training for single jobs and resource allocation across one or more jobs in both private clusters and the public cloud.

7.1.1 Distributed Model Training

In applying pipelining to accelerate distributed model training, we made the following contributions:

- We discussed the challenges associated with using pipeline parallelism for distributed model training: operator partitioning to load balance computation across pipeline stages and minimize communication; scheduling forward and backward passes of different inputs to minimize memory footprint, maximize throughput, and not compromise convergence speed of training; and state management when necessary.
- We proposed new strategies for pipeline parallelism, and demonstrate the settings in which these strategies are advantageous compared to previously proposed forms of parallelism. Each of these strategies expose tradeoffs along the throughput, memory footprint, and weight update semantics dimensions (Table 7.1), and consequently are optimal in different problem settings. For example, PipeDream-Flush from Chapter 3 or the interleaved schedule from Chapter 4 would not be suitable to train a small model like VGG-16 (with training footprint

smaller than the memory capacity of a single GPU), since idle time would negate the benefits of reducing the amount of communication between workers.

- Pipeline parallelism can be composed with other forms of parallelism, such as data and tensor model parallelism. These parallelism modes interact in *non-trivial* ways. We demonstrated the performance characteristics of these combinations, both empirically and analytically. A careful combination of data parallelism with pipeline and tensor model parallelism can perform training iterations of a model with up to a trillion parameters using 3000+ GPUs with high efficiency (52% of theoretical peak device throughput). We were able to show that careful combinations of pipeline and data parallelism are also useful at smaller scales (speedups of up to 5$\times$ using just 16 GPUs).
- The best parallelization configuration can be picked in an automated way using an optimizer. A carefully picked combination of data and pipeline parallelism can be up to 5$\times$ faster than data parallelism alone by reducing the amount of communication that needs to be performed across workers, while still keeping workers active without idling. Depending on the problem setup, different partitioning algorithms can be used. For example, transformer models have repetitive structures, thus allowing the partitioning algorithm in Chapter 3 to be much simpler with far reduced asymptotic and empirical running time compared to the partitioning algorithm in Chapter 2 (the partitioning algorithm in Chapter 2 makes fewer assumptions of the model architecture, e.g., operators can be different, model architecture can feature branching, etc.).

Pipelining Scheme	Percentage of Ideal Time Idle	Memory Footprint (Weight, Activations)	Weight Update Equation
GPipe [86]	$\frac{p-1}{m}$	(1, m)	$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$
PipeDream (Chapter 2)	0	(p, p)	$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W_1^{(t-p+1)}, \ldots, W_p^{(t)})$
PipeDream-2BW (Chapter 3)	0	(2, p)	$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t-1)})$
PipeDream-Flush (Chapter 3)	$\frac{p-1}{m}$	(1, p)	$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$
Interleaved (Chapter 4)	$\frac{1}{v} \cdot \frac{p-1}{m}$	(1, p)	$W^{(t+1)} = W^{(t)} - \nu \cdot \nabla f(W^{(t)})$

Table 7.1: Comparison of various pipelining approaches discussed in this dissertation along three dimensions: percentage of ideal computation time spent in idle periods (pipeline bubble size), memory footprint (number of weight versions and number of stashed activation versions), and weight update semantics. Lower idle time and memory footprint are better. p is the pipeline-parallel size, m is the number of microbatches injected into the pipeline (typically $m \gg p$), and v is the number of virtual stages in the interleaved schedule ($v = 1$ if interleaving is not used). The interleaved schedule reduces the pipeline bubble size by a factor of v, but also increases the amount of in-pipeline communication by the same factor v. Vanilla PipeDream is the only pipelining scheme with no gradient accumulation within the pipeline (minimum supported batch size of b, where b is the microbatch size used); the other pipelining schemes use gradient accumulation within the pipeline (minimum supported batch size of $b \cdot p$).

7.1.2 Resource Allocation

We also were able to make a number of existing cluster scheduling policies heterogeneity-aware.

- We observed that the objectives of many popular policies (e.g., fairness, makespan, cost) can be expressed as a function of each job's observed throughput. Consequently, these policies can be formulated as optimization problems; the optimal value returned from solving the corresponding optimization problem gives the theoretically optimal allocation. Allocations represent the time fractions each job should spend on the available resource types.
- Each optimization problem formulation can be extended to be heterogeneity aware by using a concept called effective throughput, the time average of the raw throughputs each job observes on the heterogeneous compute resources. The effective throughput captures the effect of giving resources to various jobs in specific ratios prescribed by the allocation. The concept of effective throughput also makes it possible to apply performance optimizations such as space sharing in a heterogeneity-aware way with only small modifications to the allocation format (and consequently changes to the constraints in the optimization problem and the way effective throughput is computed). Our resulting heterogeneity-aware policies make it possible to automate the process of allocating different types of GUs to training jobs with different performance characteristics.
- A round-based scheduling mechanism can then ensure that each active job in the cluster obtains its theoretically-optimal allocation. Each round is of configurable duration. Every round, the scheduler decides what types of resources each job should receive (if any), while trying to match the "received" allocation with the optimal allocation that is being matched. The round-based scheduling mechanism also allows policies that deploy space sharing to be realized.
- Through this careful scheduling of jobs on resources (e.g., jobs that are slow on an older GPU type are never given time on that resource type), we showed that objectives such as average job completion time can be improved by 3.5$\times$ on clusters with various types of NVIDIA GPUs. The same cluster can also handle 50% higher input load with these heterogeneity-aware policies.
- This policy framework can also be used in settings where we are trying to optimize cost. In particular, these policies can integrate dynamic pricing and availability information from spot instances to further reduce costs.

7.2 Broad Takeaways

This dissertation tried to demonstrate the usefulness of *profile-driven automated optimization* in accelerating machine learning training. Machine learning computations are extremely regular: the

same computation kernels are repeated in a highly iterative fashion, with little to no data-dependent optimization. This makes profiles extremely easy to collect (e.g., by timing a couple of hundred iterations). In this dissertation, we used such profiles to determine how operators in a distributed training job should be placed on various training resources, and also how individual jobs should be placed on different types of training resources based on their affinity with the available hardware types. The optimizers we used to solve these problems were diverse: we used dynamic programming to decide how to execute distributed training more efficiently (how do we partition a model training graph among n GPUs to maximize training throughput?), and linear programs to decide how to allocate heterogeneous resources to different types of training jobs while optimizing various objectives (how do we time- and space-share heterogeneous resources among training jobs with certain performance characteristics to optimize a specific objective?). The profiles were also collected at different granularities. For distributed model training, we collected per-operator profiles (computation times, intermediate tensor sizes, parameter sizes for each operator in the model). For cluster scheduling, we collected per-job profiles (end-to-end iteration time for models on different types of resources).

However, profile-driven optimization becomes harder to apply when computation is less *regular*. For example, we did not target sparse models in this work. Determining the right optimization algorithms for data-dependent executions is an interesting area of future study.

7.3 Future Directions

We conclude with some directions for future work related to the ideas presented in this dissertation.

Model Inference. This dissertation largely focused on the macro- and micro- scheduling challenges associated with *training* modern deep neural network models. However, once trained, these models need to be deployed in end applications. Executing model inference efficiently, however, presents unique challenges:

- Users want to optimize for latency-related objectives (e.g., average latency, tail latency), which are more diverse than just throughput. These objectives also have implicit dependencies on throughput (e.g., if a system processes inputs slower than the rate at which they come in, then latency will also increase due to an increase in queuing delay).
- Inference systems need to respond to inputs coming in from real users, as opposed to training systems which operate on training data available a priori (usually stored as a full training dataset on disk).
- Inference is an online workload (unlike training, which is offline).

Consequently, parallelizing and allocating resources for inference workloads is challenging: the optimal parallel strategy might change as input distributions change (e.g., more inputs come in

during the day compared to the night), and decisions need to be made on the order of seconds (Gavel, on the other hand, was able to solve optimization problems that took minutes since training jobs run for hours to days).

More Scheduling Problems at the Micro Scale. This dissertation considered a narrow set of micro-scheduling optimizations (efficient parallelization given a budget of training resources). However, as noted in Chapter 1, various other such optimizations are possible (e.g., low-level code generation for each hardware architecture, graph substitutions). Considering all of these in a single unified scheduling framework could further improve resource utilization and reduce training times.

Unified Scheduling and Optimization. As the demand for compute resources grows, deciding how to share (possibly heterogeneous) resources efficiently among many users is a pressing problem. Current approaches to resource scheduling typically decouple resource allocation from micro-scheduling (local optimization) decisions. For example, deciding how to parallelize a distributed job is typically made *after* the job has been granted a set of resources from the cluster scheduler. What happens if we can make these decisions jointly instead? Could we distribute a computation using heterogeneous resources when the cluster is busy, reducing demand on faster resource types? Could we optionally decide to use architecture-specific optimizations depending on the allocated hardware (e.g., older hardware might not efficiently support irregular access patterns)?

Efficient Automated Scheduling Across More Dimensions. Considering all possible parallelization dimensions for a single training job, or all possible combinations of micro- and macro-schedules for a collection of jobs using shared resources, leads to large search spaces. Computing allocations in these unified problem settings is thus more computationally expensive. Approaches like POP [126] hint at possible solutions (e.g., by breaking up the original allocation problem into smaller sub-problems with a subset of the jobs and resources) for certain problem structures, but further work is needed to make such unified scheduling truly practical.

Bibliography

[1] Applications of GPT-3. https://openai.com/blog/gpt-3-apps/.

[2] AWS Accelerator Offerings. https://aws.amazon.com/ec2/instance-types/.

[3] Cloud GPUs on GCP. https://cloud.google.com/gpu.

[4] Cloud TPUs on GCP. https://cloud.google.com/tpu.

[5] DeepSpeed: Extreme-Scale Model Training for Everyone. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/.

[6] DeepSpeed Repository. https://www.deepspeed.ai/.

[7] GitHub Copilot. https://copilot.github.com/.

[8] Gloo. https://github.com/facebookincubator/gloo.

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

[10] ImageNet Training in PyTorch. https://github.com/pytorch/examples/tree/master/imagenet.

[11] Implementing Core Scheduler Functionality in Resource Manager (V1) for Hadoop. https://issues.apache.org/jira/browse/HADOOP-3445.

[12] Job Scheduling in Spark. https://spark.apache.org/docs/latest/job-scheduling.html#scheduling-within-an-application.

[13] Linear-fractional Optimization. http://www.seas.ucla.edu/~vandenbe/ee236a/lectures/lfp.pdf.

[14] Megatron Repository. https://github.com/nvidia/megatron-lm.

[15] Microsoft Translates Spoken Text to Code. https://techcrunch.com/2021/05/25/microsoft-uses-gpt-3-to-let-you-code-in-natural-language/.

[16] MLPerf. https://www.mlperf.org/.

[17] NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[18] NVIDIA Collective Communication Library (NCCL). https://developer.nvidia.com/nccl.

[19] NVIDIA Deep Learning Examples, BERT. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/README.md#results.

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

[21] NVIDIA Selene Supercomputer. https://www.top500.org/system/179842/.

[22] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/.

[23] OpenWebText Dataset. https://github.com/jcpeterson/openwebtext.

[24] PyTorch DDP. https://pytorch.org/docs/stable/_modules/torch/nn/parallel/distributed.html.

[25] PyTorch JIT. https://pytorch.org/docs/stable/jit.html.

[26] VGG-16 Target Accuracy using Caffe Model. https://gist.github.com/ksimonyan/211839e770f7b538e2d8#gistcomment-1403727.

[27] Word-level Language Modeling RNN. https://github.com/pytorch/examples/tree/master/word_language_model.

[28] YARN – The Capacity Scheduler. https://blog.cloudera.com/yarn-capacity-scheduler/.

[29] AWS Lambda. https://aws.amazon.com/lambda/, 2020.

[30] AWS Spot Pricing Model. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/, 2020.

[31] EC2 Fleet. https://docs.amazonaws.cn/en_us/AWSEC2/latest/UserGuide/ec2-fleet.html, 2020.

[32] English Wikipedia. https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2, 2020.

[33] Google Cloud Functions. https://cloud.google.com/functions, 2020.

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

[35] NVIDIA Multi-Process Service. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`, 2020.

[36] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[37] Alexander Aiken and Alexandru Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In *European Symposium on Programming*, pages 221–235. Springer, 1988.

[38] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[39] Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software Pipelining. *ACM Computing Surveys (CSUR)*, 27(3):367–432, 1995.

[40] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[41] Baidu Inc. Bringing HPC Techniques to Deep Learning, 2017.

[42] Dimitri P Bertsekas and Robert G Gallager. Data Networks. 1987.

[43] Léon Bottou and Olivier Bousquet. The Tradeoffs of Large Scale Learning. In *Advances in Neural Information Processing Systems*, pages 161–168, 2008.

[44] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[45] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, and et al. Language Models are Few-Shot Learners. *arXiv preprint arXiv:2005.14165*, 2020.

[46] Emmanuel J Candes and Yaniv Plan. Matrix Completion with Noise. *Proceedings of the IEEE*, 98(6):925–936, 2010.

[47] Liang-Fang Chao, Andrea S LaPaugh, and EH-M Sha. Rotation Scheduling: A Loop Pipelining Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(3):229–239, 1997.

[48] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing Efficiency and Fairness in Heterogeneous GPU Clusters for Deep Learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[49] David L Chen and William B Dolan. Collecting Highly Parallel Data for Paraphrase Evaluation. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 190–200. Association for Computational Linguistics, 2011.

[50] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting Distributed Synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

[51] 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. *arXiv preprint arXiv:1512.01274*, 2015.

[52] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 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.

[53] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174*, 2016.

[54] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined Back-Propagation for Context-dependent Deep Neural Networks. In *Interspeech*, 2012.

[55] 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.

[56] Andrew Chung, Jun Woo Park, and Gregory R Ganger. Stratus: Cost-Aware Container Scheduling in the Public Cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 121–134, 2018.

[57] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of DAWNBench, A Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review*, 53(1):14–25, 2019.

[58] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. *NeurIPS ML Systems Workshop*, 2017.

[59] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX Annual Technical Conference*, pages 37–48, 2014.

[60] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[61] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. Hydra: A Federated Resource Manager for Data-Center Scale Analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, 2019.

[62] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[63] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 127–144, 2014.

[64] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[65] Michael Denkowski and Alon Lavie. Meteor Universal: Language Specific Translation Evaluation for Any Target Language. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 376–380, 2014.

[66] 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.

[67] Steven Diamond and Stephen Boyd. CVXPY: A Python-Embedded Modeling Language for Convex Optimization. *The Journal of Machine Learning Research*, 17(1):2909–2913, 2016.

[68] José Luis Díaz, Joaquín Entrialgo, Manuel García, Javier García, and Daniel Fernando García. Optimal Allocation of Virtual Machines in Multi-Cloud Environments with Reserved and On-demand Pricing. *Future Generation Computer Systems*, 71:129–144, 2017.

[69] Desmond Elliott, Stella Frank, Khalil Sima'an, and Lucia Specia. Multi30K: Multilingual English-German Image Descriptions. In *Proceedings of the 5th Workshop on Vision and Language*, pages 70–74. Association for Computational Linguistics, 2016.

[70] Joaquín Entrialgo, José Luis Díaz, Javier García, Manuel García, and Daniel F García. Cost Minimization of Virtual Machine Allocation in Public Clouds Considering Multiple Applications. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*, pages 147–161, 2017.

[71] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[72] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[73] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2018.

[74] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 24–24, 2011.

[75] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated Model, Batch, and Domain Parallelism in Training Neural Networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pages 77–86, 2018.

[76] 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.

[77] Andreas Griewank and Andrea Walther. Revolve: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[78] David Griffis. RL A3C PyTorch. `https://github.com/dgriff777/rl_a3c_pytorch`.

[79] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[80] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv preprint arXiv:1806.03377*, 2018.

[81] F Maxwell Harper and Joseph A Konstan. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TIIS)*, 5(4):19, 2016.

[82] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. PipeTransformer: Automated Elastic Pipelining for Distributed Training of Transformers. *arXiv preprint arXiv:2102.03161*, 2021.

[83] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2961–2969, 2017.

[84] 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.

[85] 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 *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 22–22, 2011.

[86] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[87] Yu-Hsiang Huang. Attention is All You Need: A PyTorch Implementation. `https://github.com/jadore801120/attention-is-all-you-need-pytorch`, 2018.

[88] Zhouyuan Huo, Bin Gu, Qian Yang, and Heng Huang. Decoupled Parallel Backpropagation with Convergence Guarantee. *arXiv preprint arXiv:1804.10574*, 2018.

[89] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient Data Encoding for Deep Neural Network Training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2018.

[90] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497–511. 2020.

[91] 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 *USENIX Annual Technical Conference, USENIX ATC 2019*, pages 947–960, 2019.

[92] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[93] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[94] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML '18)*, 2018.

[95] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[96] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*, 2018.

[97] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[98] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[99] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Köster. Pipelined Backpropagation at Scale: Training Large Models without Batches. *Proceedings of Machine Learning and Systems*, 2021.

[100] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997*, 2014.

[101] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 Dataset. `http://www.cs.toronto.edu/kriz/cifar.html`, 2014.

[102] 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.

[103] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale MLPerf-0.6 Models on Google TPU-v3 Pods. *arXiv preprint arXiv:1909.09756*, 2019.

[104] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. RACE: Large-scale ReAding Comprehension Dataset From Examinations. *arXiv preprint arXiv:1704.04683*, 2017.

[105] Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 318–328, 1988.

[106] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. AlloX: Compute Allocation in Hybrid Clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[107] Kyungyong Lee and Myungjun Son. DeepSpotCloud: Leveraging Cross-Region GPU Spot Instances for Deep Learning. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 98–105, 2017.

[108] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, volume 1, page 3, 2014.

[109] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv preprint arXiv:2006.15704*, 2020.

[110] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. *arXiv preprint arXiv:2102.07988*, 2021.

[111] Erik Linder-Norén. PyTorch-GAN. `https://github.com/eriklindernoren/PyTorch-GAN#cyclegan`.

[112] Kuang Liu. Train CIFAR-10 with PyTorch. `https://github.com/kuangliu/pytorch-cifar`.

[113] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019.

[114] 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 20)*, pages 289–304, 2020.

[115] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[116] Dominic Masters and Carlo Luschi. Revisiting Small Batch Training for Deep Neural Networks. *arXiv preprint arXiv:1804.07612*, 2018.

[117] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. MLPerf Training Benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[118] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and Optimizing LSTM Language Models. *arXiv preprint arXiv:1708.02182*, 2017.

[119] 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*, 2017.

[120] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[121] Azalia Mirhoseini, Hieu Pham, Quoc Le, Mohammad Norouzi, Samy Bengio, Benoit Steiner, Yuefeng Zhou, Naveen Kumar, Rasmus Larsen, and Jeff Dean. Device Placement Optimization with Reinforcement Learning. *arXiv preprint arXiv:1706.04972*, 2017.

[122] Andriy Mnih and Ruslan R Salakhutdinov. Probabilistic Matrix Factorization. In *Advances in Neural Information Processing Systems*, pages 1257–1264, 2008.

[123] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[124] Abdallah Moussawi. Towards Large Scale Training of Autoencoders for Collaborative Filtering. In *Proceedings of Late-Breaking Results Track Part of the Twelfth ACM Conference on Recommender Systems*, RecSys'18, Vancouver, BC, Canada, 2018.

[125] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[126] Deepak Narayanan, Fiodar Kazhamiaka, Firas Abuzaid, Peter Kraft, and Matei Zaharia. Don't Give Up on Large Optimization Problems; POP Them! *arXiv preprint arXiv:2104.06513*, 2021.

[127] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. In *International Conference on Machine Learning*, pages 7937–7947. PMLR, 2021.

[128] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training. In *Workshop on Distributed Infrastructure, Systems, Programming and AI (DISPA)*, 2020.

[129] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[130] Deepak Narayanan, Keshav Santhanam, Amar Phanishayee, and Matei Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In *NeurIPS Workshop on Systems for Machine Learning (December 2018)*, 2018.

[131] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient Large-Scale Language Model Training on GPU Clusters. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[132] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource Elasticity in Distributed Deep Learning. In *Proceedings of Machine Learning and Systems 2020*, pages 400–411. 2020.

[133] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[134] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[135] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training, 2018.

[136] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 1(8):9, 2019.

[137] Bozidar Radunovic and Jean-Yves Le Boudec. A Unified Framework for Max-Min and Min-Max Fairness with Applications. *IEEE/ACM Transactions on Networking*, 15(5):1073–1083, 2007.

[138] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv:1910.10683*, 2019.

[139] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[140] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

[141] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv preprint arXiv:2104.07857*, 2021.

[142] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[143] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv preprint arXiv:2101.06840*, 2021.

[144] 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.

[145] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[146] 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.

[147] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-Bit Stochastic Gradient Descent and its Application to Data-Parallel Distributed Training of Speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[148] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On Parallelizability of Stochastic Gradient Descent for Speech DNNs. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE SPS, May 2014.

[149] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[150] Mohammad Javad Shafiee, Brendan Chywl, Francis Li, and Alexander Wong. Fast YOLO: A Fast You Only Look Once System for Real-Time Embedded Object Detection in Video. *arXiv preprint arXiv:1709.05943*, 2017.

[151] Supreeth Shastri and David Irwin. HotSpot: Automated Server Hopping in Cloud Spot Markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 493–505, 2017.

[152] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*, 2018.

[153] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models using GPU Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[154] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[155] Prabhakant Sinha and Andris A Zoltners. The Multiple-Choice Knapsack Problem. *Operations Research*, 27(3):503–515, 1979.

[156] Evan R Sparks, Ameet Talwalkar, Daniel Haas, Michael J Franklin, Michael I Jordan, and Tim Kraska. Automating Model Search for Large Scale Machine Learning. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 368–380. ACM, 2015.

[157] Satish Narayana Srirama and Alireza Ostovar. Optimal Resource Provisioning for Scaling Enterprise Applications on the Cloud. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 262–271, 2014.

[158] Xiao Sun, Tan N Le, Mosharaf Chowdhury, and Zhenhua Liu. Fair Allocation of Heterogeneous and Interchangeable Resources. *ACM SIGMETRICS Performance Evaluation Review*, 46(2):21–23, 2019.

[159] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. Efficient Algorithms for Device Placement of DNN Graph Operators. In *Advances in Neural Information Processing Systems*, pages 15451–15463, 2020.

[160] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[161] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: Global Rescheduling with Adaptive Plan-Ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 35. ACM, 2016.

[162] Uber Technologies Inc. Meet Horovod: Uber's Open Source Distributed Deep Learning Framework for TensorFlow, 2017.

[163] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8), August 1990.

[164] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[165] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[166] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[167] 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.

[168] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18, 2015.

[169] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. Spotnik: Designing Distributed Machine Learning for Transient Cloud Resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[170] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. 2019. In the Proceedings of ICLR.

[171] 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.

[172] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[173] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[174] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Hongjun Choi, Blake Hechtman, and Shibo Wang. Automatic Cross-Replica Sharding of Weight Updates in Data-Parallel Training. *arXiv preprint arXiv:2004.13336*, 2020.

[175] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. *Proceedings of Machine Learning and Systems*, 2021.

[176] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR*, abs/1906.08237, 2019.

[177] Yang You, Igor Gitman, and Boris Ginsburg. Large Batch Training of Convolutional Networks. *arXiv preprint arXiv:1708.03888*, 2017.

[178] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, 2018.

[179] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, pages 265–278. ACM, 2010.

[180] 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.

[181] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2223–2232, 2017.