CITY UNIVERSITY OF HONG KONG
香港城市大學

Efficient Scheduling of Distributed Deep Neural Network Workloads
分佈式深度神經網絡任務的高效調度

Submitted to
Department of Computer Science
電腦科學學系
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
哲學博士學位

by

LI Jiamin
李嘉敏

February 2024
二零二四年二月

Abstract

Deep Neural Networks (DNNs) have become the cornerstone for a myriad of AI applications. However, the growing complexity and size of DNN models, along with the increasing scale of datasets, have precipitated a surge in computational resource requirements, elevating the costs of training and inference. To tackle these issues, DNNs are increasingly deployed across extensive GPU clusters. Yet, the design of systems to host distributed DNN workloads encounters significant challenges across the infrastructure, framework, and algorithmic layers.

This Ph.D. thesis contributes the realm of distributed DNN systems by addressing challenges on multiple fronts in a bottom-up approach. At the infrastructure layer, we design Lyra, that addresses the problem of separate training and inference clusters by introducing capacity loaning and elastic scaling. This novel cluster scheduler significantly reduces queuing times and completion times of DNN training jobs, improving cluster resource utilization. Moving to the model framework layer, Lina addresses the challenges of distributed training and inference of sparsely activated models, specifically Mixture-of-Experts (MoE) language models, by identifying and alleviating communication bottlenecks. This results in substantial reductions in training step time and inference latency. Lastly, the thesis introduces Adaptive Gating in MoE, a flexible training strategy that reduces the computation costs of each token based on its linguistic complexity. This algorithmic approach achieves less training FLOPS and time while maintaining the same inference quality.

The collective advancements presented in this thesis signify a small but meaningful advancement in the scalability and efficiency of systems underpinning distributed DNN workloads.

City University of Hong Kong

Qualifying Panel and Examination Panel

Surname: LI

First Name: Jiamin

Degree: Doctor of Philosophy

Department: Department of Computer Science

The Qualifying Panel of the above student is composed of:v

Supervisor

Prof. WANG Cong — Department of Computer Science, City University of Hong Kong

External Co-supervisor

Prof. XU Hong — Department of Computer Science and Engineering, The Chinese University of Hong Kong

Qualifying Panel Members

Prof. WANG Jianping — Department of Computer Science, City University of Hong Kong

Prof. LI Minming — Department of Computer Science, City University of Hong Kong

Prof. XUE Chun, Jason — Department of Computer Science, City University of Hong Kong

This thesis has been examined and approved by the following examiners:

Prof. WANG Cong — Department of Computer Science, City University of Hong Kong

Prof. LI Zhenjiang — Department of Computer Science, City University of Hong Kong

Prof. WANG Shiqi — Department of Computer Science, City University of Hong Kong

Prof. WU Chuan — Department of Computer Science, The University of Hong Kong

Contents

List of Tables

List of Figures

Acknowledgments

First, I would like to express my deep gratitude to my advisor, Prof. Hong Xu, for leading me into the field of computer systems and networking, engaging in countless inspiring discussions, and providing me with precious guidance, insightful comments, and kind encouragement. I am also equally thankful to my co-advisor, Prof. Cong Wang for his continuous support. His wise words, "Hard work will pay off down the road," have been a constant reminder of the importance of perseverance and dedication in academic endeavors.

I want to express my gratitude to my mentors during my internship at ByteDance: Dr. Yibo Zhu and Dr. Chuanxiong Guo. Their expertise and willingness to share their knowledge have played a pivotal role in the development of my research work. I feel fortunate to have been mentored by Dr. Peng Cheng at Microsoft Research Asia. My sincere thanks go out to Prof. Aditya Akella for hosting me as a visiting researcher at The University of Texas at Austin and providing invaluable feedback on my work. My appreciation also extends to my qualifying and examination panel members: Prof. Chun Xue, Jason, Prof. Jianping Wang, Prof. Zhenjiang Li, Prof. Shiqi Wang and Prof. Chuan Wu, for their constructive suggestions and insights.

I would like to express my sincere thanks to my dedicated collaborator, Dr. Yimin Jiang, whose support was instrumental in bringing my research ideas to fruition. I also appreciate the suggestions offered by my fellows at City University of Hong Kong, Dr. Xiaorui Wu and Dr. Qiang Su, in the challenging times of my Ph.D. study. I want to thank my senior fellow, Dr. Libin Liu, and my colleagues at ByteDance, Zherui Liu and Shuguang Wang, for providing all the technical support and resources for my research.

Special thanks go to my dear friends, Kun Yang, Lin Jin, and Ruoying Ma, for their companionship since the beginning of my undergraduate studies and

for sharing both my joy and sorrow. I am also grateful to Dr. Yiding Wang for helping me shape my career decisions during the final year of my study.

In particular, I want to thank Dr. Fangzhou Wang, who has been the most supportive and caring friend in my life. Our weekly dinners have consistently served as a pillar of strength. Without his unwavering encouragement, I could not complete my Ph.D. study.

Finally, I would like to thank my parents, my aunt, and my cousin for their endless love and patience.

Chapter 1

Introduction

1.1 Distributed Deep Neural Networks

In the ever-evolving landscape of artificial intelligence (AI), deep learning stands as a pivotal innovation, poised to revolutionize industries, enhance decision-making, and augment our everyday experiences. The rapid expansion of deep neural networks (DNN) across various fields—including natural language processing [5], computer vision [148], reinforcement learning [68], and recommendation systems [60]—has sparked an imperative for efficient, scalable, and reliable systems to sustain these sophisticated algorithms.

Over the last decade, we have witnessed a trend towards larger datasets [142], more complex DNNs [41], and increasingly powerful hardware [105] to boost computational prowess, leading to unprecedented accuracy levels. Distributed computation has become essential for scaling DNN workloads effectively, with systems explicitly engineered for these tasks now at the forefront of modern cloud computing.

However, this rapid advancement has not been without its challenges, which present themselves across the different strata of DNN systems. At the summit, the algorithmic layer is tasked with the development of DNN architectures, the

refinement of training algorithms, and the efficient handling of datasets. Following closely is the framework layer, which provides the essential APIs via platforms such as PyTorch [116] and TensorFlow [2]. This enables the building, training, and deployment of models, and is further extended to key communication libraries like NCCL [102] and inference engines like TensorRT [141]. The infrastructure layer underpins these efforts, offering a specialized platform for DNN execution, complete with resource allocation, task scheduling, GPU virtualization [33], and the orchestration of storage and network systems [1, 43]. The bedrock of this technological edifice is the hardware layer, which includes not just the GPU chips, but also the critical compilers and runtime libraries that accompany them [27, 104].

Given the extensive transformation of system design due to DNN workloads, it is an arduous task to encapsulate all facets of progress within a single thesis or to design an all-encompassing system that addresses every emerging issue. Consequently, this thesis narrows its focus to three distinct yet pivotal contributions that address specific challenges within the infrastructure, framework and algorithmic layers.

1.2 Summary of Contributions

In the thesis, we describe three DNN system designs that improve the scalability and efficiency of distributed DNN workloads by leveraging domain-specific knowledge of DNN training and inference. Lyra is a scheduler that operates in the infrastructure. Lina improves the framework layer by optimizing communication costs. Adaptive gating tries to further elevate the model efficiency from the algorithmic perspective.

Lyra. Organizations often build separate training and inference clusters for deep learning, and use separate schedulers to manage them. This leads to

problems for both: inference clusters have low utilization when the traffic load is low; training jobs often experience long queuing due to a lack of resources. We introduce Lyra, a new cluster scheduler to address these problems. Lyra introduces capacity loaning to loan idle inference servers for training jobs. It further exploits elastic scaling that scales a training job's resource allocation to better utilize loaned servers. Capacity loaning and elastic scaling create new challenges to cluster management. When the loaned servers need to be returned, we need to minimize job preemptions; when more GPUs become available, we need to allocate them to elastic jobs and minimize the job completion time (JCT). Lyra addresses these combinatorial problems with principled heuristics. It introduces the notion of server preemption cost, which it greedily reduces during server reclaiming. It further relies on the JCT reduction value defined for each additional worker of an elastic job to solve the scheduling problem as a multiple-choice knapsack problem. Prototype implementation on a 64-GPU testbed and large-scale simulation with 15-day traces of over 50,000 production jobs show that Lyra brings 1.53x and 1.48x reductions in average queuing time and JCT, and improves cluster usage by up to 25%.

Lina. Scaling model parameters improves model quality at the price of high computation overhead. Sparsely activated models, usually in the form of Mixture of Experts (MoE) architecture, have sub-linear scaling of computation cost with model size, thus providing opportunities to train and serve a larger model at lower cost than their dense counterparts. However, distributed MoE training and inference is inefficient, mainly due to the interleaved all-to-all communication during model computation. This work makes two main contributions. First, we systematically analyze all-to-all overhead in distributed MoE and present the main causes for it to be the bottleneck in training and inference, respectively. Second, we design and build Lina to address the all-to-all bottleneck head-on. Lina opportunistically prioritizes all-to-all over the concurrent allreduce

whenever feasible using tensor partitioning, so all-to-all and training step time is improved. Lina further exploits the inherent pattern of expert selection to dynamically schedule resources during inference, so that the transfer size and bandwidth of all-to-all across devices are balanced amid the highly skewed expert popularity in practice. Experiments on an A100 GPU testbed show that Lina reduces the training step time by up to 1.73x and reduces the 95%ile inference time by an average of 1.63x over the state-of-the-art systems.

Adaptive gating in MoE. Large language models, such as OpenAI's ChatGPT, have demonstrated exceptional language understanding capabilities in various NLP tasks. Sparsely activated MoE has emerged as a promising solution for scaling models while maintaining a constant number of computational operations. Existing MoE model adopts a fixed gating network where each token is computed by the same number of experts. However, this approach contradicts our intuition that the tokens in each sequence vary in terms of their linguistic complexity and, consequently, require different computational costs. Little is discussed in prior research on the trade-off between computation per token and model performance. This work introduces adaptive gating in MoE, a flexible training strategy that allows tokens to be processed by a variable number of experts based on expert probability distribution. The proposed framework preserves sparsity while improving training efficiency. Additionally, curriculum learning is leveraged to further reduce training time. Extensive experiments on diverse NLP tasks show that adaptive gating reduces at most 22.5% training time while maintaining inference quality. Moreover, we conduct a comprehensive analysis of the routing decisions and present our insights when adaptive gating is used.

1.3 Thesis Organization

Chapter 2 provides more background on distributed DNN workloads. Chapter 3 proposes Lyra, an elastic cluster scheduler for distributed DNN training workloads. Chapter 4 studies sparsely-activated Mixture-of-Expert models. It proposes Lina, a communication scheduler for distributed MoE models and improves training and inference efficiency. Chapter 5 explores sparsity and adaptivity in the scope of MoE-based language models. It presents adaptive gating in MoE models and provides in-depth empirical analysis of the trade-off between computation cost and model performance. Section 5.5 concludes the thesis and discusses future research directions.

Chapter 2

Background & Literature Review

2.1 Distributed DNN Training and Inference

In the domain of DNN, training and inference represent two pivotal workloads in the DNN lifecycle. Over recent years, the computational demands of DNN models have surged exponentially, especially in cases like Large Language Models (LLMs), which now employ billions of parameters. To cope with this growing computational burden, distributed computation has become indispensable. Nevertheless, the efficiency of distributed DNN is far from optimal, hampered by several factors such as communication overhead, synchronization requirements, and memory costs. Additionally, DNN training and inference possess unique characteristics that require careful consideration. Researchers have been hard at work proposing various solutions to mitigate these challenges and enhance the overall efficiency of both DNN training and inference. In this section, we delve into state-of-the-art solutions for enabling distributed DNN, followed by recent developments aimed at addressing the communication and memory bottlenecks in distributed DNN.

2.1.1 Parallelism Strategies

Determining how to distribute DNN models across multiple GPU devices, a problem known as parallelism strategy, is anything but trivial. Numerous solutions have emerged in recent years to tackle this issue. With advancements in model architecture and hardware specifications, two primary parallelism strategies have become standard practice.

The first strategy is data parallelism, which stands as the most widely adopted approach in distributed DNN. It involves partitioning the input data into multiple segments, each assigned to a worker. During training, each worker computes gradients of the model parameters based on its data partition and then synchronizes these gradients with other workers.

The second strategy, model parallelism, is designed for situations where the entire model cannot fit within a single device. This approach divides the model parameters into multiple partitions, with each partition assigned to a worker. There are two methods for partitioning the model. The first approach partitions the model vertically, i.e., layer-wise, where each worker is responsible for computing the output of a subset of layers. This is commonly referred to as pipeline parallelism. GPipe [57] and PipeDream [99] are among the first that introduces pipeline parallelism and designs efficient training mechanisms by exploiting micro-batches. The second approach partitions the model horizontally, splitting the parameter tensor into multiple chunks. It is first introduced by Nvidia and named as tensor parallelism in [135]. Each worker is tasked with computing a slice of the model parameters, and the outputs are synchronized at the end of each step.

More recently, to address the computational demands of LLMs, hybrid parallelism has gained popularity. This approach integrates various forms of parallelism, combining data parallelism and model parallelism (both layer-wise

and parameter-wise) to provide a comprehensive solution for LLM execution. FlexFlow [64] and Megatron-LM [135] are two notable works that employ hybrid parallelism to train LLMs.

In the pursuit of elevating the efficiency of distributed DNN, researchers have ventured into harnessing lower-level parallelism, specifically at the operator and kernel levels. This approach allows for a more fine-grained optimization of DNN workloads. Notably, two innovative systems have emerged to address this challenge: Pathways [11] by Google and Alpa [172] proposed by Zheng et al. Both solutions advocate a unified controller runtime architecture that combines the principles of "single program multiple data" (SPMD) and "multiple program multiple data" (MPMD) models. AlpaServe [83] is a dedicated work for DNN inference. It reveals a fundamental trade-off between the overhead introduced by model parallelism and the opportunity to exploit statistical multiplexing to reduce serving latency in the presence of bursty workloads.

SPMD focuses on harnessing intra-operator parallelism, optimizing the parallel execution of operations within a single DNN model. This approach seeks to maximize computational efficiency by concurrently processing individual operations or kernels, thereby reducing processing time. MPMD explores the domain of inter-operator parallelism, aiming to enhance efficiency by parallelizing the execution of multiple operations or kernels across the DNN model. By orchestrating the parallel execution of distinct operations, MPMD minimizes idle time and promotes overall throughput.

2.2 Bottlenecks in Distributed DNN

Intitially, performance bottlenecks in distributed DNN training and inference were primarily attributed to the communication overhead among workers. However, as the scale of DNN models continues to grow, the memory cost of storing

model parameters has become a significant concern. We thus discuss the existing work on addressing the communication and memory bottlenecks in distributed DNN respectively.

2.2.1 Communication Overhead

TicTac [48], ByteScheduler [112] and BytePS [65] are three leading work in efficiently scheduling communication oeprations in distributed DNN training. All three are designed for data parallel training. TicTac reduces the iteration time by identifying and enforcing parameter transfers in the order in which the parameters are consumed by the underlying computational model, thereby guaranteeing near-optimal overlap of communication and computation. ByteScheduler proposes to partition and rearrange the tensor transmissions to achieve theoretically-optimal training efficiency. BytePS leverages spare CPU and bandwidth resources in the cluster to accelerate distributed DNN training tasks running on GPUs.

Another set of contributions—BlueConnect [24], Blink [149], and TACCL [129]—concentrates on designing more efficient collective communication algorithms as alternatives to the widely adopted communication library NCCL provided by Nvidia. BlueConnect decomposes a single all-reduce operation into a large number of parallelizable reduce-scatter and all-gather operations to exploit the trade-off between latency and bandwidth, and adapt to a variety of network configurations. Blink identifies the heterogeneity of network resources and designs a communication library for inter-GPU parameter exchange that achieves near-optimal link utilization. TACCL is a tool that enables algorithm designers to guide a synthesizer into automatically generating algorithms for collective communication based on a given hardware configuration and communication collective.

Optimizing communication costs with network topology in mind, three notable works—PLink [88], SYNDICATE [91], and TopoOpt [151]—take different

routes to achieve their goals. PLink introduces an optimized communication library that probes the physical network then generates and executes a fitted hierarchical aggregation plan to take advantage of such locality, and evolves the plan to adapt to changing network conditions. SYNDICATE proposes a novel abstraction to break large communication work as smaller pieces as part of execution planning and perform joint optimization of scheduling and execution planning by rethinking the interfaces in the networking systems stacks used for DNN training. TOPOOPT creates dedicated partitions for each training task using reconfigurable optical switches and patch panels, and jointly optimizes the topology and parallelization strategy within each partition.

2.2.2 Memory Consumption

When LLM becomes one of the mainstream of DNN tasks, memory gradually becomes one of the bottleneck as well. With billions of parameters, naively placing the model on multiple devices still result in a large memory footprint. To address this issue, researchers have proposed some solutions to reduce the memory consumption of distributed DNN training.

The most representative work is ZeRO-offload by Microsoft [126]. It is a memory optimization technique that partitions the model into multiple shards and places each shard on a different device. The model parameters are then split into two categories: optimizer states and non-optimizer states. The optimizer states are replicated across all devices, while the non-optimizer states are only stored on a single device. This approach significantly reduces the memory footprint of the model, as the non-optimizer states are typically much larger than the optimizer states. However, this approach introduces small communication overhead, as the non-optimizer states must be transferred to other devices during training. ZeRO-offload then becomes a standard practice in distributed DNN training, and is adopted by many other systems, such as DeepSpeed [30].

Another line of literature tries to make the best use of GPU memory by letting multiple training tasks concurrently execute on the same device. Zico [85], Wavelet [150] and Salus [165] are three representative works in this direction. Zico and Wavelet exploits the cyclic pattern of memory consumption in DNN training to schedule multiple training tasks on the same device. Salus is a distributed execution engine that enables two GPU sharing primitives: fast job switching and memory sharing to dynamically allocates GPU memory to each task based on their memory demands.

2.3 GPU Cluster Scheduling

GPUs have evolved to become the cornerstone of system infrastructure for hosting Deep Neural Network (DNN) workloads. As the demand for GPU-accelerated tasks has grown exponentially, large-scale GPU clusters have been developed to provide the requisite computational resources. However, the scale of these GPU clusters has brought about new challenges in cluster management. Both academia and industry practitioners have responded with a variety of cluster scheduling systems designed to address these challenges. This section provides a concise overview of the state-of-the-art GPU cluster scheduling systems.

The early entrants into the realm of GPU cluster scheduling included Optimus [111], Tiresias [46], and Gandiva [158]. Optimus introduced a resource-performance model that predicted DNN model convergence, relying on this knowledge to estimate training times and schedule tasks. Tiresias adopted an information-agnostic scheduling approach, relaxing the stringent placement requirements for training tasks that were not sensitive to locality. Meanwhile, Gandiva introduced an introspective scheduling policy that leveraged intra-job predictability for DNN tasks.

Two notable works that focused on online adjustments to resource alloca-

tion for DNN tasks to enhance overall cluster performance are AFS [59] and Pollux [120]. AFS prioritized tasks with greater throughput gains when allocating additional resources to them. Pollux took a step further by proposing a co-optimization strategy for model convergence and cluster efficiency. This was achieved by tuning job hyperparameters and resource allocation simultaneously. Shockwave [173], inspired Pollux, incorporated fairness considerations into scheduling by extending market theory to dynamic settings and employing stochastic dynamic programming.

While most cluster schedulers primarily considered GPU resources, making the assumption of sufficient CPU and memory resources, Synergy [96] and Muri [171] stood out by addressing multi-resource allocation problems. Synergy used optimistic profiling to infer the sensitivity of DNNs to different resources and made multi-resource workload-aware assignments across a set of tasks. Muri, on the other hand, packed tasks along multiple resource types in the time dimension.

Beyond resource allocation, other schedulers focused on different aspects of cluster efficiency. For instance, AntMan [159], building on the work of Gandiva, enabled the colocation of resource-guaranteed and best-effort tasks. Gavel [100] delved into the performance heterogeneity of tasks in heterogeneous clusters. HiveD [170] aimed to improve resource isolation in multi-tenant GPU clusters by leveraging virtual private clusters.

A distinct line of research analyzed GPU cluster workloads. Philly [61], for instance, examined the granularity of GPU resources and the gang scheduling requirements of DNN tasks, although these observations have become somewhat outdated due to advancements in GPU hardware and the advent of elastic DNN training. Additionally, PAI [154] openedourced a workload trace from Alibaba and highlighted several primary challenges in GPU clusters, including low GPU utilization, long queueing delays, hard-to-schedule tasks with demanding re-

quirements for high-end GPUs, load imbalances across heterogeneous machines, and potential CPU bottlenecks.

We highlight the diverse and evolving landscape of GPU cluster scheduling systems, each tailored to address specific challenges and aspects of cluster efficiency and management.

2.4 Adaptive and Sparse Computation

Adaptive and sparse computation has become a popular approach to keep scaling the DNN models, among which Mixture-of-Experts (MoE) is one of the most representative ones. Switch Transformer [37] and GShard [79] promote the usage of MoE in DNN models.GLaM [34] has shown empirically that MoE architecture effectively enlarges the model capacity while keeps low computation costs. Extensive work from both theory-side and system-side has been proposed to advance this particular area. We therefore follow such progress and discuss the recent developments of MoE-based DNN models from both perspectives.

2.4.1 Algorithms

[177] analyzes the trade-off between MoE training stability and quality and the fine-tuning performance comparison with dense models. It then provides suggestions how to determine the expert size, expert capacity and routing algorithms to achieve optimal performance. To handle imbalanced token-to-expert assignment, [174] proposes to have experts selecting the top-k tokens instead of letting tokens select the top-k experts. [80] introduces new balanced assignment of experts that formulates token-to-expert allocation as a linear assignment problem, allowing an optimal assignment in which each expert receives an equal number of tokens. [178] adopts stochastic routing. [127] proposes to modify the feedforward layer to hash to different sets of weights

depending on the current token, over all tokens in the sequence in a deterministic approach.

2.4.2 Systems

FastMoE is the first MoE framework. It provides simple API and designs dedicated computation kernels to accelerate the training process. FasterMoE [50] and SmartMoE [166] are two follow-up contributions. FasterMoE proposes a dynamic shadowing approach to cope with load imbalance, and a smart fine-grained schedule that splits different operations and executes them concurrently. SmartMoE designs a dedicated approach to search for the best parallelism strategy for MoE models. It finds optimization opportunities in an enlarged space of hybrid parallelism, considering the workload of data-sensitive models

DeepSpeed is one of the leading framework that supports converting dense DNN models to sparse MoE models [124]. It also introduces a pyramid-like MoE architecture where expert size grows along with the depth of the model. Tutel [58] identifies the inefficiency of static execution in MoE models and proposes a highly scalable stack design and implementation for MoE with dynamically adaptive parallelism and pipelining. Tutel is later integrated into DeepSpeed.

SE-MoE [133] introduces acceleration designs for both training and inference. In training, it proposes Elastic MoE training with 2D prefetch and Fusion communication over Hierarchical storage, so as to enjoy efficient parallelisms in various types. For scalable inference in a single node, especially when the model size is larger than GPU memory, SE-MoE forms the CPU-GPU memory jointly into a ring of sections to load the model, and executes the computation tasks across the memory sections in a round-robin manner for efficient inference.

Chapter 3

Lyra: Elastic Scheduling for Deep Learning Clusters

3.1 Introduction

Recently, Deep Neural Networks (DNNs) have seen wild successes in many applications [78]. Hyperscale online service providers have adopted DNN, and build large-scale GPU clusters to accelerate DNN workloads for both training and inference. GPU cluster scheduling is a fundamental and critical task to utilize the expensive infrastructure efficiently, by optimizing job resource allocation and task placement.

It is common practice today to separately build and manage two types of GPU clusters, one for training and one for inference. This is because, for the same model, inference requires less computation and GPU memory than training and is less likely to utilize the numerous cores of training GPU [110, 26, 101]. Inference clusters usually use weaker GPUs, like Nvidia T4, with a fraction of the resources of the training GPUs, such as Nvidia V100 and A100.

This separation creates problems for both sides (Section 3.2). Our observations are based on experiences of operating production clusters with O(10k) GPUs for

training and even more for inference. Specifically, inference cluster utilization is usually low (<40%) for an extended period of time due to the diurnal traffic pattern. At the same time, training jobs experience long queuing time before they can start, with an average of over 3,000s and 95%ile of almost 10,000s as seen from a 15-day trace with over 50,000 jobs. The long queuing time is due to both the high cluster utilization and the GPU resource fragmentation.

To address these problems, we propose *capacity loaning* to allow the inference cluster to loan the idle GPU servers during low-traffic periods to run training jobs, and reclaim them back when inference workloads increase again (Section 3.2.1). Capacity loaning mitigates both the utilization problem for inference and queuing problem for training. It is feasible for training jobs that do not have strict requirements on GPU type. Then to ensure the on-loan servers are rapidly utilized by training jobs when they become available, we draw inspiration from *elastic scaling* [53, 117, 106] (Section 3.2.2). Elastic scaling enables a running job to scale out or scale in to better utilize the dynamically changing resource pool. It also helps reduce queuing delay since an elastic job can start with a small number of workers first and increase its workers when more resources become available.

Capacity loading and elastic scaling create new degrees of freedom for cluster scheduling. As we navigate the new design space, we meet several new challenges that must be addressed before we can reap the benefits.

First, though loaning decisions can be solely made by the inference cluster scheduler to ensure inference workloads are not affected, reclaiming is more intricate. When the inference cluster needs to reclaim on-loan resources, the training scheduler has to preempt all running jobs on those servers. Given the high overhead of preemption and prolonged running time to the jobs, the scheduler must carefully select the servers in order to minimize the total preemptions.

Second, job scheduling is inherently more complicated with elastic scaling.

Resource allocation has to consider a mix of inelastic jobs with fixed demand and elastic jobs with *variable* demand. We show that classical scheduling policies such as shortest job first (SJF) no longer work well with elasticity, and finding the JCT-optimal solution for merely two jobs is difficult. Given the allocation results, the scheduler still needs to determine the worker-server placement to minimize fragmentation, where servers are now heterogeneous with different GPUs because of capacity loaning.

Our key insight in solving these challenges is to prioritize the minimum resources needed by a job over its elastic demand, and to prioritize the dedicated training servers over the on-loan inference servers. This makes sense because the minimum demand of an elastic job is equivalent to an inelastic job to which not allocating resources is detrimental, but the elastic part can be fulfilled later without stalling the job.

Our solution, therefore, exhibits a two-phase structure following the above insight. For reclaiming, we first kill the elastic workers running on on-loan servers since stopping them does not lead to any job-level preemption. When preemption becomes inevitable, we characterize the problem as a knapsack problem with dependent item values [97] and develop an efficient heuristic to solve it (Section 3.4).

For resource allocation, we first allocate for both inelastic jobs and elastic jobs' base demand, with the aim of launching as many jobs as possible. We then scale out the scheduled elastic jobs if resources permit. The first phase can be solved using SJF to reduce queuing time and the second phase is formulated as a multiple-choice knapsack problem [138] to minimize running time, which in practice can often be solved using dynamic programming (Section 3.5.2). We then tackle the placement problem by placing the inelastic jobs on training servers, and elastic jobs on on-loan servers as much as feasible. Jobs are ordered based on the best-fit-decreasing policy to address the bin packing nature [25] and

minimize fragmentation (Section 3.5.3).

Putting everything together, we design (Section 3.3–Section 3.5), implement (Section 3.6), and evaluate (Section 3.7) Lyra, a new cluster scheduler that realizes capacity loaning with elastic scaling. Lyra has an orchestrator that manages capacity loaning by executing instructions from the inference scheduler on when and how much to loan or reclaim, and by deciding which on-loan servers to return for reclaiming. Then a job scheduler periodically determines allocation and placement, and scales new and existing elastic jobs in response to resource and job dynamics. To be pragmatic, Lyra considers elastic scaling only for large DNNs whose training throughput scales well in our experiments.

The results of Lyra are promising (Section 3.7). We build a high-fidelity simulator, and replay a 15-day job trace collected from 3,544 training GPUs and 4,160 inference GPUs. We find that compared to a FIFO scheduler, Lyra can reduce the average and 95%ile JCT by up to 1.48x and 1.47x, respectively, and improve GPU usage by 25%. In terms of job scheduling, Lyra also outperforms state-of-the-art Pollux by 1.28x and 1.27x in median JCT and 95%ile JCT when elastic jobs occupy 36% training resources.

We summarize our contributions as follows.

- With production traces, we report the problem of separate management of training and inference clusters, i.e. low utilization in the inference cluster and long queuing time in the training cluster.
- We propose cluster-level capacity loaning and job-level elastic scaling, two new control knobs for cluster scheduling to address the above problems.
- We study the resulting cluster scheduling problems, develop a key insight to prioritize the minimum resources needed by each job to address elasticity, and use a principled approach to characterize and solve each problem.
- We design and implement Lyra, a novel cluster scheduler that integrates

our solutions. Lyra works with existing resource management frameworks and is ready for deployment. Evaluation using testbed experiments and large-scale simulations validates its effectiveness.

3.2 Motivation

We start by presenting our motivation of Lyra.

3.2.1 Why Capacity Loaning?

Large GPU clusters are built to accommodate inference and training workloads with distinct requirements. Customer-facing inference jobs are latency-sensitive [26, 110]. Training jobs are much more resource-heavy and run for extended periods of time. Thus they emphasize completion times instead. Operators usually deploy separate clusters with different GPUs for training and inference, and manage them independently to minimize interference. Our production environment, for example, mainly uses Tesla V100 in the training cluster and T4 in the inference cluster. Job traces show that this practice leads to low utilization of inference resources and sub-optimal performance for training jobs.

Inefficient inference cluster utilization. Similar to other web services [86], the inference cluster is overprovisioned in order to handle the peak traffic. Inevitably, its resources are often underutilized due to the dynamic inference requests generated by customers.

We plot the GPU utilization in one of our inference clusters with 5-minute intervals for one week's time in Figure 3.1. Utilization is defined as the fraction of GPUs occupied by at least one inference job. We observe a clear diurnal pattern: peak traffic lasts about four hours at night, and demand trough occurs before dawn. The peak-to-trough ratio is $\sim$2.2 within a day, and the average utilization is $\sim$65%, both implying that there are abundant resources to be exploited in the

inference cluster for extended periods of time.

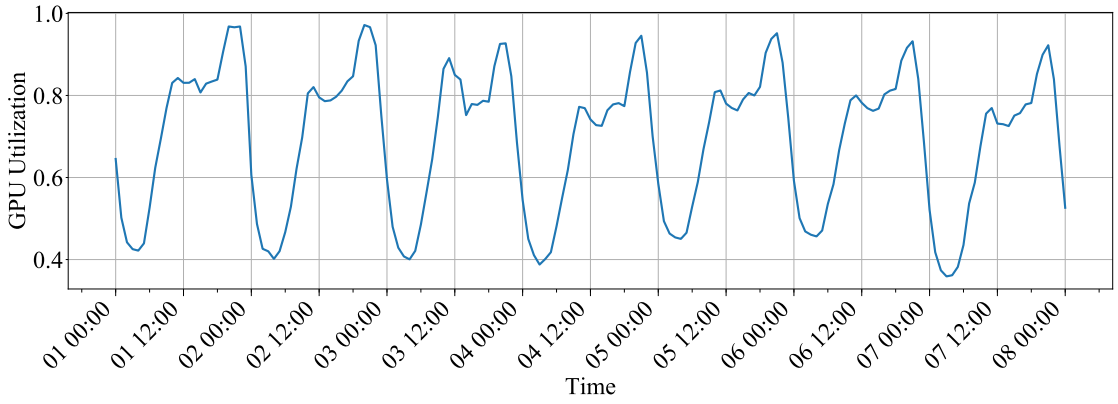


Figure 3.1: Inference cluster GPU utilization, i.e. fraction of GPUs serving at least one request in our inference cluster. The measurement spans one week from Oct 1 to Oct 7, 2020. The cluster has about 4,000 GPUs. The utilization changes from 42% in bottom hours to 95% in peak hours.

Long queuing time for training jobs. Turning to the training cluster, a salient observation we make is that many training jobs experience long queuing before they can be dispatched with enough resources. Figure 3.2 depicts the hourly queuing job ratio in our training cluster for the same week as in Figure 3.1. A significant fraction of jobs (as high as 100%) still has to wait for resources from time to time. The average queuing time is longer than 3,000 seconds and certainly non-negligible.

The long queuing time is not only due to lack of resources. In fact, the average GPU utilization across the same period of time is 82%, which means there are often idle GPUs. The dynamic training demand certainly also contributes to the long queuing time. In addition, training demand does not exhibit clear patterns for prediction.

Capacity loaning. We propose to exploit the unused inference resources to run training jobs temporarily, i.e. loaning inference capacity for training. It mitigates both problems above at the same time: The inference cluster is better utilized, and training jobs have more resources to help reduce queuing time. The on-loan capacity can be reclaimed dynamically in case the inference traffic spikes to

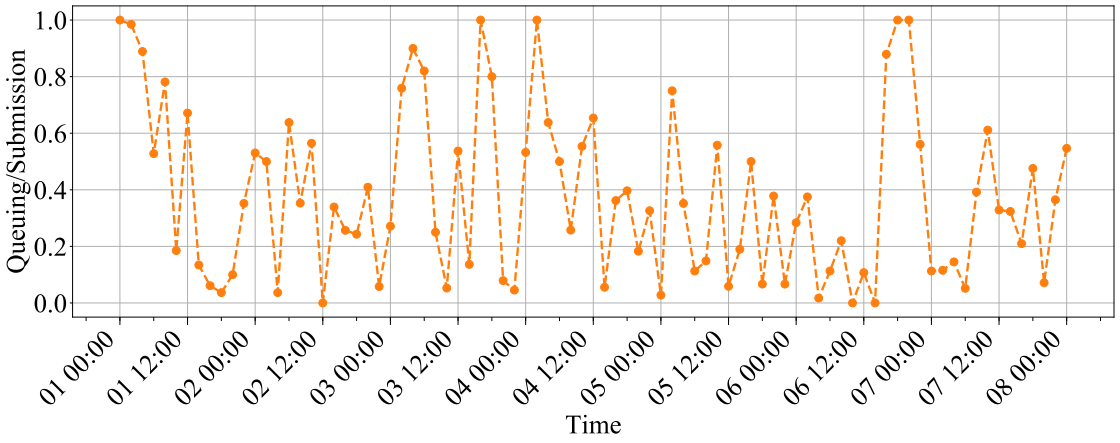


Figure 3.2: The fraction of queuing jobs among all the newly-submitted jobs in each hour in our training cluster for one week. A job suffers queuing time when the scheduler fails to satisfy its resource demand on the first try. If the ratio is high, it means that most of the jobs submitted in that hour are queued. The cluster has ∼3,500 GPUs, and the average utilization is 82%.

ensure the quality of service.

Though training jobs typically request specific GPUs, we find that up to 21% of jobs in our production traces do not do so. These *fungible* workloads can work with any GPU types in different execution runs. Lyra can launch these jobs on the loaned inference servers rather than waiting for training servers. To ensure feasibility, we may need to adjust the local batch size of the training job so that the models and the intermediate data can fit into the smaller inference GPU memory. We increase the number of workers so that the global batch size does not change to ensure the same model performance. This is straightforward since we know the GPU memory differences.

Another more aggressive way to exploit the loaned servers is to run a training job on heterogeneous GPUs, i.e. run on both training and inference GPUs (e.g. V100 and T4) at the same time. Heterogeneous training further improves scheduling flexibility with more potential gains. However, it requires delicate systems and algorithm support to work well, since the workers have to adopt different hyperparameter settings and inherently progress at different paces [109,

18, 99, 163]. Given that heterogeneous training remains an active research topic, our production training system only provides experimental support for it at the moment. Lyra's design does not depend on it, and we evaluate its effect in Section 3.7.2 when it is enabled for a small fraction of our jobs with a performance penalty.

3.2.2 Elastic Scaling for the Full Potential

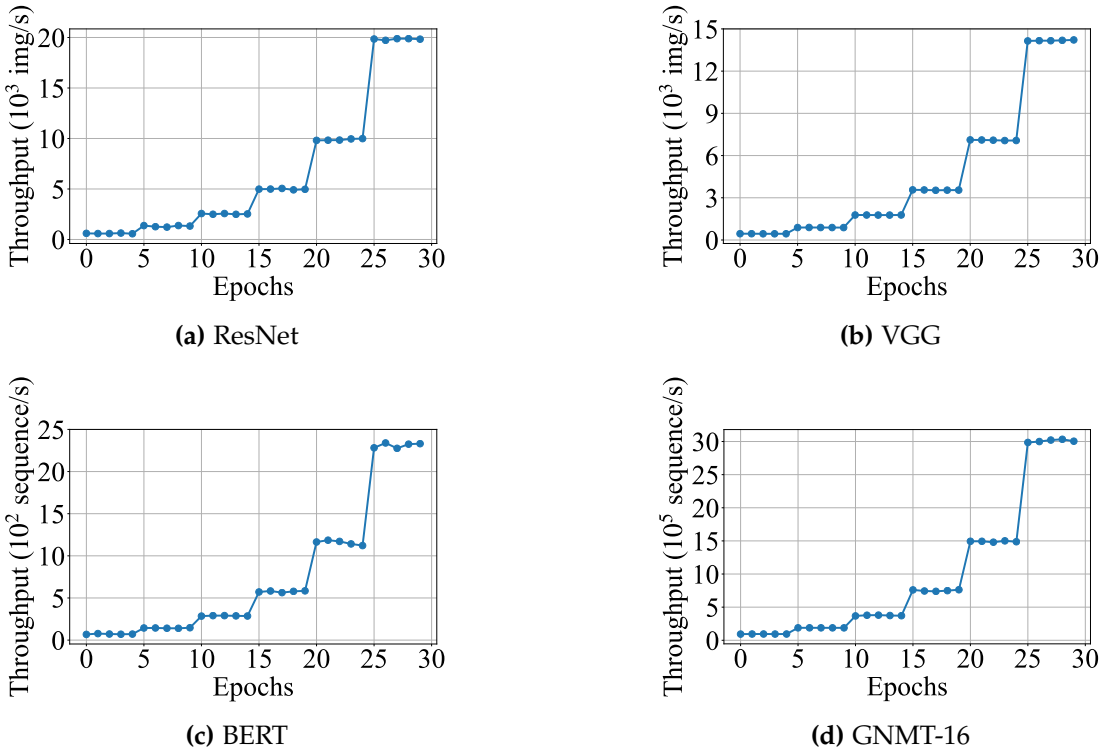
To better cope with the constantly changing cluster capacity and further exploit the loaned inference resources, Lyra considers *elastic scaling*. Recently, elastic scaling has been introduced into ML frameworks [53, 117, 106] where a job can take a variable number of workers according to resource availability. One can even adjust the number of workers on-the-fly when the job is running.

Elastic scaling can greatly facilitate capacity loaning. With additional resources, training jobs can dynamically scale out to use more workers with more inference GPUs to accelerate training (provided they are running on inference GPUs already). When the cluster experiences high loads, some jobs could scale in to free some servers. In addition, when vacating the inference servers so they can be returned, the scaling-in operation reduces the need to completely preempt the jobs which incurs high overheads with checkpointing, re-launching containers, etc.

An acute reader might be wondering about the feasibility and benefit of elastic scaling in general. Indeed, besides the scalability issue of distributed training systems [113, 66, 161, 59], when we change the number of workers on-the-fly, the training hyperparameters may have to be updated as well. This can be fairly complex: for example, simply fixing the local batch size and linearly increasing the global batch size may impede the model convergence [44].

Thus in Lyra, elastic scaling is only adopted for jobs that scale well to the number of workers without updating the local batch size or without updating

the global batch size. Existing studies [63, 94, 23, 92, 164, 82, 7, 135] show that certain models like ResNet [51] and BERT [32] satisfy this requirement. We also find that, as shown in Figure 3.3, ResNet-50 [51], VGG16 [137], BERT [32], and GNMT-16 [157] all enjoy good throughput scalability and are well-suited for elastic scaling. Our traces reveal that the large jobs (∼5% of all jobs) from these model families account for 36% of training cluster resources with an average running time of 14.2 hours, suggesting ample potential gains using Lyra. For these jobs, Lyra also restricts itself to *limited elasticity* where the worker number varies within a range.



(a) ResNet **(b)** VGG **(c)** BERT **(d)** GNMT-16

Figure 3.3: Throughput of four elastic training jobs using Tesla V100 GPUs. The workers are doubled every five epochs, starting from 1 worker. In our testbed, each server hosts 8 GPUs connected by NVLink. Servers use 100G InfiniBand interconnects. Each worker container uses 2 GPUs.

3.2.3 Existing Cluster Schedulers

Much prior work exists on GPU cluster scheduling amid the proliferation of DL workloads. Lyra differs in two aspects.

First, capacity loaning represents a new angle to cluster scheduling that few have studied. Though the shared infrastructure is exploited by recent systems [147, 87, 143, 140, 176, 154], their focus is to schedule different types of workloads in a single cluster. Lyra instead focuses on virtually loaning resources between two different clusters. Specifically, it considers the problem of how to reclaim the transient on-loan resources while minimizing its negative impact on training jobs running on them (Section 3.4), which has not been considered before. Further, Lyra takes advantage of the elasticity of training jobs to better utilize the dynamic cluster resources.

Second, some recent studies also considered scheduling elastic jobs. Gandiva [158] adopts an opportunistic approach to grow or shrink the number of GPUs used by a job without considering cluster-wide efficiency. AFS [59] greedily prioritizes jobs with the highest marginal throughput gain per GPU. Pollux [120] co-optimizes resource allocation and training hyperparameters to achieve high resource efficiency.

Compared to them, Lyra exploits the interplay between elastic scaling and capacity loaning to further improve the performance. In terms of technical approach, Lyra preserves the problem nature of scheduling elastic jobs and treats it as a variant of the knapsack problem, enabling it to make globally good allocation decisions and outperform greedy local heuristics in prior work. Though Lyra does not consider tuning hyperparameters, it can be readily integrated into Lyra (Section 3.7.4).

3.3 Design Overview

In this section, we describe Lyra's overall architecture and the key design questions we need to address.

Overall architecture. Lyra is a GPU cluster scheduler that exploits capacity loaning with elastic job scheduling. It runs on top of a cluster resource manager such as YARN [146] and Kubernetes [75] to execute its decisions.

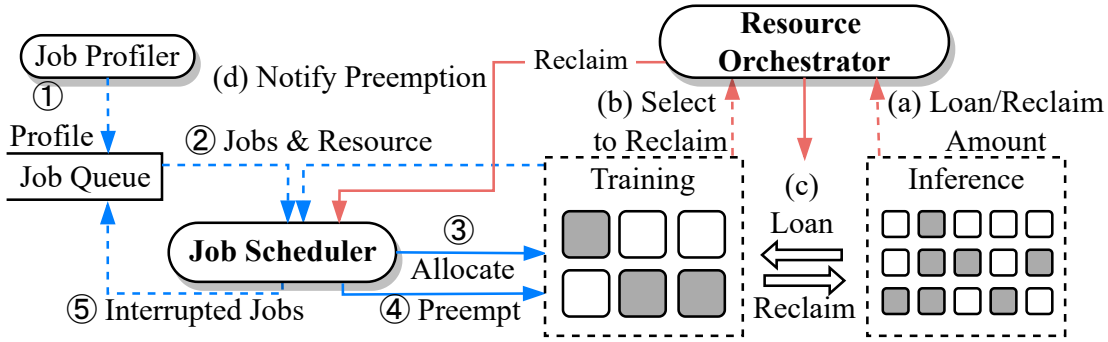


Figure 3.4: Lyra system architecture. Solid lines indicate control flow and dashed ones data flow. Red lines represent capacity loaning workflow, while blue ones elastic scaling workflow. Each square represents a GPU server; the gray ones are in use.

Figure 3.4 presents Lyra's architecture. At the cluster level, the *resource orchestrator* receives instructions from the inference cluster about the number of servers to loan or reclaim (a), determines which servers shall be returned for reclaiming (b), and commands the underlying resource manager to move the selected servers virtually across management boundaries (c). When the orchestrator reclaims on-loan servers, it may need to preempt the training jobs running on them (d). Job preemption is executed via the job scheduler.

At the job level, jobs are submitted to the queues. The job profiler estimates the workload ① after jobs are enqueued. The *job scheduler* ② periodically collects job status and resource usage of the training cluster. Then it ③ computes the resource allocation and placement decisions for each job. Meanwhile, it gets preemption instructions from the orchestrator, interrupts the running jobs

④, and puts them back into the job queues ⑤. Job launching, scaling and interruption actions are again executed by the resource manager. Job scheduler works periodically in a much smaller interval than the orchestrator in order to better handle job dynamics.

Since Lyra mainly deals with the training cluster and does not interfere with inference cluster scheduling, we use "jobs" to simply refer to training jobs hereafter without ambiguity. The basic unit of capacity loaning is a physical server. This is to prevent training jobs from interfering with the inference jobs on the same server.

Key questions. Lyra's design is centered around two key questions.

- **Server reclaiming.** Which servers should be returned so that the number of preempted jobs is minimized, when some on-loan servers need to be reclaimed?
- **Job scheduling.** How should we determine resource allocation across jobs, and how do we place a job's workers on servers, when some jobs are elastic and some servers are loaned from the inference cluster?

We now present how we address them with Lyra's detailed design in Section 3.4 and Section 3.5, respectively.

3.4 Capacity Loaning and Reclaiming

Lyra moves resources dynamically across inference and training clusters to improve utilization and training performance.

Assumptions. We presume that the inference cluster scheduler dynamically estimates the capacity needed to meet the latency, GPU utilization [76], or other performance targets, based on the predicted inference traffic [47, 132, 26]. Inference workloads are able to grow or shrink their containers along with the

incoming traffic. Inference scheduler informs Lyra's resource orchestrator of (1) the amount of resources available for loaning when traffic is low, and (2) the amount of resources to be reclaimed from training in busy hours if any. That is, the inference cluster scheduler *autonomously* determines when and which servers to lend, and when and how many servers to ask back, based on its own policy. The inference performance is *not* affected by capacity loaning.

The key question for the training scheduler is the reclaiming mechanism as mentioned in Section 3.3, i.e. which on-loan servers should be returned given the number of servers needed by the inference scheduler. This matters because reclaiming a server entails preempting all its running jobs immediately. A job with checkpointing would incur overheads to save and load the checkpoint when resuming training later. If the job does not perform checkpointing [95], which is common in practice in our environment, its entire progress is lost and training has to restart from the very beginning. Clearly, both are undesirable and we strive to minimize preemptions by strategically picking the servers to return.

We propose a solution to reduce the negative impact on the training jobs hosted by the on-loan servers.

Minimizing preemptions. Vacating an on-loan server means its jobs are preempted in a cluster with no elastic jobs. We start with how Lyra minimizes inevitable preemption under this case. and will explain how elastic scaling plays its part in minimizing preemptions in Section 3.5.3.

Denote the number of servers that need to be returned at this point as N_R. Our problem is to pick N_R on-loan servers—which host inelastic jobs' workers—in order to minimize preemptions. More concretely, we choose to minimize the number of preempted jobs so fewer users are affected. This implies when reclaiming a server, we prefer the one with a big job to the one with a few small ones.

The problem closely resembles the classic knapsack problem (i.e. the 0-1

knapsack problem): The number of servers to reclaim N_R can be considered as the capacity of the knapsack; each server consumes one unit capacity, and the number of running jobs is each server's preemption cost (i.e. value). However, the server's preemption cost actually has inter-dependencies that make the problem more difficult.

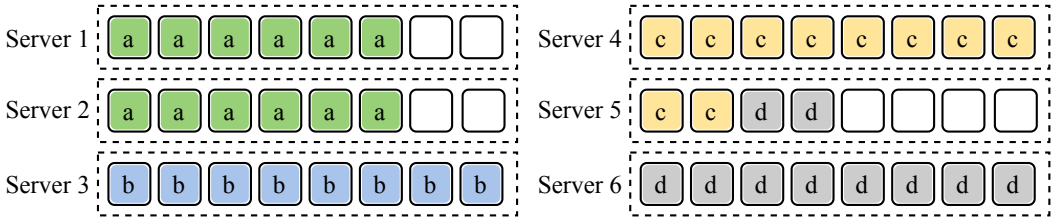


Figure 3.5: A reclaiming example. Each server has 8 GPUs. GPUs in-use are indicated by the job ID inside each square.

Server	# running jobs	sum of job's GPU fraction	sum of job's server fraction
1	1	0.5	0.5
2	1	0.5	0.5
3	1	1	1
4	1	0.8	0.5
5	2	0.4	1
6	1	0.8	0.5

Table 3.1: Different definitions of server preemption cost for the reclaiming example in Figure 3.5.

Consider an example as depicted in Figure 3.5. Table 3.1 shows each server's preemption cost as the number of its running jobs (second column). Suppose we need to reclaim two servers. Servers 1 and 2 are obviously the optimal choice with one preemption. Yet the corresponding knapsack problem would select any two 1-cost servers such as 3 and 4 which lead to more preemptions. The issue here is that in our problem the costs of servers are coupled when they host the same job(s), whereas in the 0-1 knapsack problem the cost is independent of each other. Reclaiming server 1 for instance results in an idle server 2 whose cost

becomes 0 instead of 1.

Knapsack problem with dependent item values is known to be NP-hard [97]. When N_R is one server, selecting the one with the fewest preemptions is simply by iterating all the on-loan servers. Given an N_R larger than a single server, we propose to resolve the dependency by treating it as part of the server preemption cost. One possible way is to define server preemption cost as the sum of the GPU fractions of each job on the server. For instance, server 4's cost would be 0.8 as it hosts 80% of job c's GPUs, and server 5's cost is 0.4 (0.2+0.2) as shown in Table 3.1. One can immediately see that this does not work well as it does not capture the job count. It causes server 5 to be selected with the least cost, which actually leads to two preemptions. Thus we choose to define server preemption cost as the sum of the server fractions of each job as shown also in Table 3.1. Server 4's cost would thus be 0.5 as it hosts job c and the workload of job c is shared by two servers as shown in Figure 3.5. This way server 5's cost is 0.5+0.5=1, i.e. the highest.

Once the preemption cost of each server is computed, the orchestrator selects the servers using the following heuristic: it iteratively picks the server with the lowest preemption cost, preempts its jobs by removing them from all their servers, and updates the cost of these servers correspondingly, until N_R servers are vacated. In case of a tie in the preemption cost, we additionally consider the collateral damage incurred if the server is reclaimed.

3.5 Job Scheduling

Lyra schedules jobs—both inelastic and elastic—to reduce overall JCT by utilizing resources as efficiently as possible. We start by explaining the challenge due to elasticity (Section 3.5.1). Then we present the solutions to the two facets of our scheduling problem. The first is *resource allocation* with both elastic and inelastic

jobs, i.e. how to determine the number of workers each job gets (Section 3.5.2); the second is *placement*, i.e. which servers to place a job's workers on (Section 3.5.3).

Throughout this section, we assume that training throughput scales linearly with the number of workers within the scaling range, i.e. job's running time is inversely proportional to resources allocated, as discussed in Section 3.2.2.

3.5.1 Challenge of Elasticity

Job elasticity presents a unique challenge to resource allocation. Conventional schedulers either deal with jobs with fixed demands, or ones that can arbitrarily scale [59, 120]. However, for jobs with *limited elasticity* [106], the question of how to arbitrate resources so as to minimize average JCT is intricate.

Let us consider a simple example as shown in Table 3.2. There are two elastic jobs with different minimum running times when allocated their maximum demand. Assume the cluster has eight workers in total. Table 3.3 shows three common allocation strategies and the corresponding JCT performance. In solution 1, we favor job A by giving it the maximum demand; in solution 2, we favor B instead; and in solution 3 we equally allocate resources to them. All three strategies lead to different JCTs and the difference between the worst and best is 24%, demonstrating that inefficient allocation can lead to poor JCT performance.

Job	w^{min}	w^{max}	Min. running time
A	2	6	50
B	2	6	20

Table 3.2: Two elastic jobs and their demand information. Jobs complete in min. running time when allocated with w^{max} workers.

Classic algorithms are not optimal. One may be wondering if the classic shortest (or smallest) job first strategies would work here. At least in the example of Table 3.2, the optimal allocation is indeed to first satisfy job B, which has the shortest running time. Yet, we can construct a counter example as depicted in

Solution	Initial allocation		JCT		Average JCT
	A	B	A	B	
1	6	2	50	53.33	51.67
2	2	6	63.33	20	41.67
3	4	4	60	30	45

Table 3.3: Possible resource allocation results for the two jobs when they share a cluster that can host 8 workers. Only the initial allocation is shown; once the first job finishes, the other is immediately allocated more resources as much as possible. Three solutions lead to very different JCTs.

Job	w^{min}	w^{max}	Min. running time	JCT when favored	Avg. JCT
A	2	3	100	A: 100, B: 24	62
B	2	6	20	A: 106.67, B: 20	63.33

Table 3.4: A counter example with two elastic jobs, where prioritizing A with longer running time is actually better for JCT.

Table 3.4 to show that this does not always work. We slightly modify job A to have a maximum demand of 3, and minimum running time of 100; the other setup is identical to Table 3.2. In this case, if we satisfy B first, the average JCT (63.33) is actually worse than satisfying A's demand first (62).

Intuitively, shortest job first, or SJF, is designed for fixed job running times with the intuition that each job should be given the least queuing time, which is the only variable in computing JCT. In our case, job running time itself varies along with the resource allocated, which in turn affects the overall JCT and makes the problem more complex.

More specifically, the above examples reveal two characteristics of elastic job's running time that SJF cannot handle. (1) Elastic scaling complicates the job sorting decision of SJF. Since job running time varies with the resources allocated, it is no longer apparent that we simply sort them based on their minimum running time. As shown already, doing so does not lead to an optimal result. (2) The resource efficiency of each job is different. In Table 3.4, job A has a larger

workload (i.e. product of maximum demand and minimum running time) than B, implying that the running time improvement of A is larger than that of B if both are given the same number of workers. Even though the resource allocation difference is merely one when we prioritize different jobs, job A's running time contributes to a 6.67-second JCT reduction while job B's only increases by 4 seconds.

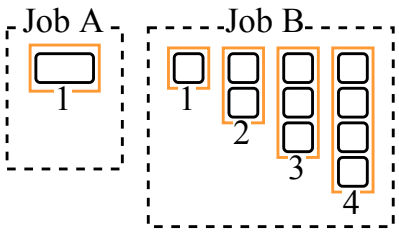
In the simplest two-job case, we can analyze the outcome of different allocation strategies. The complete theoretical analysis is omitted here for brevity and can be provided upon request. Allocation in the general case is undoubtedly more complicated with more elastic jobs plus inelastic jobs, as the optimal strategy requires enumerating the exponentially many possible resource allocations. Our quest in the following is, therefore, to find a good heuristic for the problem.

3.5.2 Two-Phase Resource Allocation

Intuition: Prioritize inelastic workload. To ease the challenge of elasticity, our insight is that an elastic job has two types of demand: a *base demand* that is inelastic in nature, i.e. the minimum demand, and a *flexible demand* that is elastic. They should be treated separately: The base demand essentially corresponds to an inelastic job whose allocation strategy is binary, and not allocating resources to it incurs more queuing delay to the job. In contrast, the flexible demand can be unfulfilled without serious impact since the job is still making progress with base demand.

Therefore, we treat the *inelastic* workload, including elastic jobs' base demands and inelastic jobs, as the first class citizen. We schedule them first with all available resources to minimize the average JCT. This also avoids starvation. Then in phase two, we consider the flexible demand of elastic jobs to fully utilize the remaining resources from phase one.

Setup and assumptions. We focus on solving the offline setting myopically



Group	Item	Weight	JCT Reduction Value
A	1	2	50
B	1	1	20
	2	2	30
	3	3	36
	4	4	40

Figure 3.6: Item weights and JCT reduction values for jobs in Table 3.4. Here, we assume job A needs 2 GPU per worker and job B 1 GPU per worker.

where the set of jobs and resources are given, and cope with the job dynamics and cluster capacity change by periodically performing scheduling in high frequency. This is common in the literature [169, 38]. Our scheduling solution is non-preemptive to minimize disruptions to training; preemption only happens during reclaiming when it becomes inevitable as in Section 3.4. Thus at a scheduling epoch, the set of available resources refers to idle GPUs and GPUs being used by flexible workers for resizing (including on-loan GPUs), and the set of jobs includes those waiting in the queue and running elastic jobs (only flexible workers). The on-loan inference GPUs are normalized relative to training GPUs when calculating the resource capacity.

We rely on job's running time information (minimum running time for elastic jobs), which can be predicted with profiling and ML methods [56, 175].

Two-phase heuristic design. We now elaborate our heuristic. The problem in phase one is how to minimize average JCT for jobs with fixed demands and known running times, for which we adopt the shortest job first (SJF) policy [35] which is a sensible and commonly used solution. As long as there are idle GPUs and pending jobs, we schedule job j^* with the smallest running time. If the demand of j^* exceeds the remaining capacity, we remove it from the pool and continue.

Phase two is more interesting. We must determine the number of additional GPUs elastic jobs get to maximize the JCT reduction. Elastic jobs here include

those already running. It turns out this problem can be transformed into *multiple-choice knapsack problem* [138]: The knapsack's capacity is the number of remaining GPUs. An elastic job j is a group with $w_j^{max} - w_j^{min}$ items, each representing a possible allocation for j's flexible demand. An item's weight is the number of GPUs in this allocation, and its value is its JCT reduction over the job's maximum running time. Figure 3.6 illustrates this transformation with the two-job example in Table 3.4. The problem is to pack the items into the knapsack so that the total value is maximized, with the constraint of taking exactly one or zero items from each group.

The multiple-choice knapsack problem, similar to the classical knapsack, is NP-hard and often solved by dynamic programming which runs in pseudo-polynomial time [138]. With a moderate number of GPUs and jobs, dynamic programming can usually solve the instance efficiently. We find that the longest solution time in our evaluation is 0.02s with 354 items and 245 GPUs which is much shorter than a typical job's time use.

3.5.3 Worker Placement

Given the allocation results, i.e. number of workers each job gets, we still need to determine the placement of each worker to complete scheduling. Our goal is to reduce fragmentation. The primary concern is the mix of inelastic and elastic jobs as well as the transient on-loan servers with different GPUs.

Our fundamental strategy is bin packing with best-fit decreasing (BFD) heuristic [108]. Jobs are sorted in decreasing order of their per-worker GPU demand as GPU is most likely the bottleneck resource for training. Starting from the largest job, we place each worker of the job into a non-empty server that best fits its demand; if none has sufficient remaining resources, we place it on a new server. If the job is elastic, we prefer to place it on inference servers in order to maximize the potential for scaling in during reclaiming and reduce job

preemptions. If it is inelastic, we prefer to place it on training servers. When placing elastic jobs, we also place their base and flexible demands on separate groups of inference servers so that during reclaiming (Section 3.4), Lyra can release the server group for flexible demands first without any preemption to see if this alone is sufficient.

3.6 Implementation

We have implemented a prototype of Lyra with about 3500 lines of Python. The prototype works with our existing YARN and Kubernetes deployment to move servers across clusters virtually, manage worker containers for training, and monitor the status of servers and workers.

We highlight key details of the implementation as follows.

Interface for capacity loaning. During loaning, resource orchestrator need to update the available resource of each cluster once the operation is decided. We create a *whitelist* API to facilitate capacity loaning operations. Both Lyra's scheduler and the inference scheduler maintain their own whitelist of servers under their control. During loaning, the orchestrator adds on-loan servers to Lyra scheduler's whitelist according to inference scheduler's instructions. In reclaiming, the orchestrator removes the selected servers from Lyra's whitelist after its scheduler confirms they no longer have running workers.

Inference resource usage predictor. We develop a simple NN model to predict the inference resource usage. The predictor is an LSTM model with a window size of 10 and two hidden layers. We apply Adam optimizer and use MSE to compute loss. We predict the resource usage of the next five minutes and compare the average resource usage to the ground truth. The average loss is 0.00048 over 1440 data points. With the predictor, Lyra can initiate reclaiming decisions in advance before the inference resource usage increases.

Data locality and resource isolation. Lyra performs capacity loaning only between clusters in the same datacenter to ensure the network bandwidth across servers is consistently high [8, 136]. Also, the basic unit of loaning is a physical server so co-location of inference and training jobs is not possible, and no additional isolation mechanisms are needed.

Enable elastic scaling. We enable elastic training in our environment with a few modifications to the ML frameworks. Chiefly, we embed a controller process to each elastic job that coordinates the worker join and departure. As presented before, an elastic training job has a base demand and a flexible demand. Base demand guarantees the gang scheduling of minimum requests and the flexible demand shortens the running time whenever possible while preserving loss convergence. Recent work [106, 159] developed more complete scaling solutions that our implementation could also utilize.

Handle heterogeneous GPU training. As discussed in Section 3.2, a small portion of training jobs can run on heterogeneous GPUs experimentally. When this feature is turned on, Lyra's job scheduler considers these jobs with the lowest priority on the remaining servers after all other jobs are scheduled. The actual scheduling logic for these jobs remains the same as we discussed in Section 3.5, except that if they are elastic, their base demands are placed on training servers, and flexible demands on inference servers whenever possible.

3.7 Evaluation

We evaluate Lyra using large-scale simulations and testbed experiments with traces from our production clusters.

The highlights of our findings are:

- In simulations, Lyra shows salient benefits with 1.53x and 1.48x reductions on average queuing time and JCT, respectively, compared to the baseline

FIFO scheduler. When working alone, capacity loaning has 1.39x and 1.31x reductions in average queuing time and JCT, and elastic scaling has 1.35x and 1.38x reductions in the same metrics.

- Compared to state-of-the-art scheduler Pollux [120], Lyra's scheduling algorithm brings 1.35x average queuing time and 1.42x average JCT reductions when both consider tuning the training hyperparameters. Lyra's reclaiming algorithm performs comparably to the optimal solution with only 1–3ms running time.

- In testbed, Lyra improves average job queuing time by 1.38x and average JCT by 1.22x over the Baseline without loaning or scaling. Preemption only happens to ∼9% of the jobs in reclaiming with an average 63-second overhead.

3.7.1 Setup

Traces. We rely on a 15-day job trace from one of our production training clusters with 3,544 GPUs (443 8-GPU servers). There are 50,390 training jobs, and job running time range from minutes to days. We also use a GPU utilization trace from the inference cluster for the same time period. Part of the traces has been shown in Figures 3.1 and 3.2 already.

Simulator. We built a discrete-event simulator for evaluating Lyra at scale using job traces from production. It simulates the cluster scale, hardware configuration, and all job events including arrival, completion, scaling, and preemption. Job's running time in the simulator is derived from actual training time in the traces. For elastic jobs, we compute its actual training time based on the traces which is inversely proportional to its resource allocation as discussed in Section 3.5. We also consider jobs with imperfect scalability in Section 3.7.2.

Testbed. Our testbed consists of four 8-GPU training servers and four 8-GPU

inference servers. Each training server uses Nvidia V100 GPUs with 32GB GPU memory and has 92 vCPU with 350 GB memory. Each inference server uses Nvidia T4 GPUs with 16GB GPU memory and has 92 vCPU and 210 GB memory. The resource management framework is YARN, and training data is stored in HDFS.

Headroom in inference cluster. To handle unexpected traffic surges in the inference cluster, we leave a headroom of 2% of the inference cluster capacity. These machines are never to be loaned. This is chosen based on our empirical observations. Lyra's resource orchestrator runs every five minutes with an overhead of less than one second; and we find that the median inference traffic burst within five minutes is $\sim$2% of the inference cluster capacity based on our GPU utilization trace (Section 3.2.1).

Training job types. Based on the resource requirements, training jobs can be:

- *Fungible*: 21% jobs can be executed on different GPU types in different runs, i.e. ideal for capacity loaning.
- *Elastic*: Jobs can take a variable number of workers that can be adjusted on-the-fly. They are ideal for elastic scaling.
- *Heterogeneous*: Jobs can run on different GPU types at runtime.

Scenarios. We consider various scenarios with different degrees of support for elastic scaling and heterogeneous training, both of which are not widely used today.

- *Basic*: Here fungible jobs are used for capacity loaning (21% of total training load), and elastic jobs are used for elastic scaling ($\sim$5% of all jobs accounting for 36% of total training resources). No heterogeneous training. This corresponds to the status quo in our environment (recall Section 3.2.1) and is the default scenario.

- *Advanced*: On top of *Basic*, 10% of jobs can use heterogeneous GPUs with non-ideal performance. The jobs are randomly selected and distributed evenly across 15 days. Specifically, heterogeneous training jobs only achieve at most 70% of the ideal results. We experimentally confirm such a performance gap which has also been reported by prior work [109, 18].

- *Heterogeneous*: Different from the *Advanced* scenario, we disable the 21% fungible training load and consider the 10% heterogeneous training non-ideal performance solely.

- *Ideal*: All jobs support scaling and heterogeneous training with ideal performance. For jobs without a pre-defined scaling range, we consider its requested demand to be the base demand, and its scaling range is twice that.

Schemes compared. We compare Lyra to the following schemes that represent the state-of-the-art and/or the most common solutions to each sub-problem of Lyra.

We first compare capacity loaning to a simple opportunistic scheme:

- *Opportunistic Scheduling*: We disable capacity loaning, and queue the 21% fungible training jobs to the inference cluster with a lower priority than inference jobs, so they can opportunistically use the idle servers.

We also consider two basic strategies for server reclaiming:

- *Random*: On-loan servers are randomly selected.

- *Smallest (Job) Count First (SCF)*: The top-k servers that host the smallest number of jobs are chosen.

We consider several solutions to elastic scheduling. Some are slightly modified to conform with our setup for elastic jobs.

- *Gandiva* [158]: Elastic scaling is also mentioned in Gandiva. It exploits elasticity by scaling out jobs to utilize the remaining resources on servers whenever they are under-utilized. We consider under-utilization to be the period when there are available resources but no pending jobs.

- *AFS* [59]: Starting from one GPU per job, it iteratively adds one more GPU to the job with the largest marginal throughput gain. We implement AFS by allocating base demand to each job first and allocating one more worker to the job with the largest throughput gain per GPU.

- *Pollux* [120]: Pollux computes the goodput of training jobs and applies genetic algorithms to find the resource allocation. It also adjusts batch size to maximize goodput and learning rate based on Adascale [67]. We adopt the model distribution listed by Pollux to capture the model goodput.

We notice that Pollux's idea of tuning the hyperparameters according to allocated resources is orthogonal to job scheduling. To compare with Pollux fairly, we integrate this idea into Lyra in Section 3.7.4:

- *Lyra+TunedJobs*: Use Lyra's job scheduler and adapt Pollux's job agent for job-level hyperparameter-tuning within the scaling range. Job agent adjusts model batch size and learning rate whenever job resource allocation changes.

Lastly, our baseline scheme is:

- *Baseline*: A FIFO cluster scheduler with no capacity loaning or elastic scaling.

Metrics. We consider queuing time and JCT to evaluate Lyra. We report Lyra's performance improvements using the following method:

$$\text{Reduction} = \frac{\text{Duration of a scheme compared}}{\text{Duration of Lyra}}$$

Both the average queuing time and the average JCT are the arithmetic mean.

3.7.2 Overall Performance in Simulation

We evaluate Lyra thoroughly with large-scale simulation. We provide its overall performance here. Analyses of its individual components are presented in Section 3.7.3 and Section 3.7.4.

Simulator calibration and fidelity. To first establish its fidelity, we evaluate our simulator against the prototype system in a testbed with a small trace.

We calibrate our simulator by comparing the scheduling logs between the testbed and the simulator. We carefully build several tiny training job traces (20 minutes – 2 hours) to cover all possible job and resource allocation status. We run the traces on the testbed and record the timestamp of every activity (e.g. job launching, start and end of training, scheduling decision). The same traces are replayed on the simulator. We compare the timestamp and decision of each activity, and find the first wrong decision or the first activity with a larger-than-two-seconds time difference. We resolve the time difference and replay the trace repeatedly until all activities on the simulator match with the testbed records.

We add a fixed overhead according to our testbed experiments (Section 3.7.5) whenever a job is preempted in simulation. The simulation results are similar to testbed results, with a difference of 6.2% and 3.4% in average and 95%ile JCT, and 3.5% and 4.4% in average and 95%ile queuing time. The small difference mainly stems from the overhead of placing and removing workers and moving resources between clusters which the simulator does not capture.

Cluster and workload. We use the full 15-day trace and the same cluster configuration as our production clusters.

Queuing time, JCT, and cluster usage. Table 3.5 records the performance of Lyra in different scenarios. Overall, queuing time and JCT are improved by 1.53x and

#	Scenario	Scheme	Queuing Time (s)			JCT (s)			GPU Usage		Preemption
			Mean	Median	95%ile	Mean	Median	95%ile	Training	Overall [1]	Ratio [2]
1	—	Baseline [3]	3072	55	8357	16610	791	82933	0.72	0.52	0
2	Basic		2010	26	3358	11236	568	56477	0.86	0.65	12.24%
3	Advanced	Lyra	1835	24	3238	10434	525	56553	0.86	0.68	7.35%
4	Heterogeneous		1944	27	3574	12113	604	57392	0.78	0.64	11.23%
5	Ideal		1157	22	3204	8891	422	41146	0.93	0.72	5.72%
6	Capacity Loaning (Basic)	Opportunity	2788	22	5256	14828	744	67843	0.74	0.63	19.35%
7		Random	2901	23	5478	14678	731	62923	0.76	0.64	20.89%
8		SCF	2783	24	4994	14923	695	62456	0.76	0.64	17.48%
9		Lyra	2212	23	3427	12947	662	57987	0.76	0.65	14.94%
10	Elastic Scaling (Basic)	Gandiva	3035	49	6632	15912	755	80567	0.79	NA	NA
11		AFS	2284	47	3488	15045	686	60883	0.95	NA	NA
12		Pollux	2791	58	5883	14534	721	72123	0.93	NA	NA
13		Lyra	2275	47	3475	12048	602	57597	0.92	NA	NA
14		Lyra+TunedJobs	2054	43	2749	10229	564	52458	0.91	NA	NA

(1) Overall GPU usage denotes the GPU utilization in both training and inference cluster. It is applied when the training cluster size is changing in capacity loaning.
(2) Preemption ratio is the ratio between the total number of preemptions and the total number of job submissions.
(3) No capacity loaning or elastic scaling is considered. We use the FIFO job scheduler in Baseline Section 3.7.1.

Table 3.5: Simulation results in different scenarios using different schemes.

1.48x when compared to Baseline in the Basic scenario (row 2). The overall cluster usage is improved by 25%. In the Advanced case with non-ideal heterogeneous training, queuing time and JCT are reduced by 1.67x and 1.59x over Baseline and by 1.10x and 1.08x over Lyra itself in the Basic scenario. In the Ideal case which represents the performance upper bound, the average combined usage of inference and training clusters is improved by 38.5% (to 72%) over Baseline. Compared to the Basic case, average queuing time and JCT in the Ideal case show additional 27% and 14% improvements by virtue of complete job flexibility and perfect performance scalability.

Since the training cluster's resource is dynamically changing, we depict the hourly combined cluster usage for 48 hours in Figure 3.7. The Baseline usage curve shows a clear diurnal pattern mostly attributable to the inference cluster. When capacity loaning is enabled, Lyra improves the usage and flattens the

curve; the most significant improvement is a 14% usage increase between Basic and Baseline. Notice the combined usage does not reach 100% since the inference cluster needs a 2% headroom to gracefully handle the latency SLA.

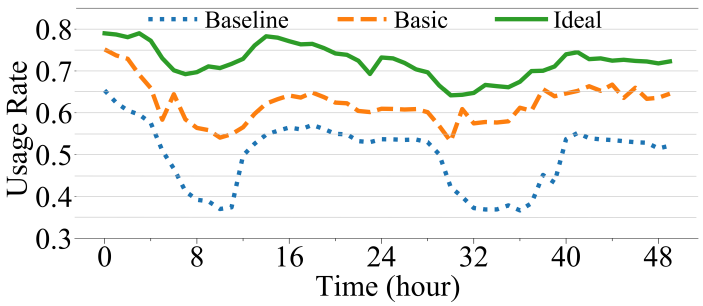


Figure 3.7: Overall resource usage rate of Baseline and Lyra in Basic and Ideal scenario.

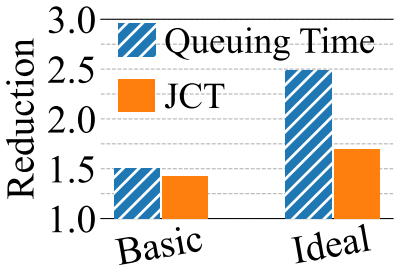


Figure 3.8: Average queuing time and JCT against Baseline in imperfect scalability.

How scaling helps capacity loaning? We now seek to understand how our two key ideas interact and complement each. Scaling helps capacity loaning, especially in reducing preemptions in reclaiming the on-loan servers. With elastic scaling disabled, Table 3.5 shows that preemption as percentage of running jobs increases from 12.24% (row 2) to 14.94% (row 9). We also observe that on average the flexible server group (hosting elastic workers only) alone satisfies 53.5% of reclaiming demand each time. With more aggressive flexibility (row 5), preemption is reduced to 5.72% and satisfies 83.5% of reclaiming demand each time.

In Section 3.5.3, we discussed how Lyra places elastic and inelastic jobs with on-loan servers in the cluster. In Table 3.6, we compare the placement performance in different scenarios without special treatment to elastic jobs, i.e. instead of grouping their flexible demand and placing them to on-loan servers as much as possible, the scheduler places them to training servers first just like inelastic jobs. The most significant difference is in preemption ratio. Without grouping the flexible demand, preemption ratio increases by up to 91% in Ideal (compared to Table 3.5 row 5). Preemptions also incur degradation to job runtime;

for example average queuing time and JCT in the Basic case increase by up to 11.1% and 15.2%.

Scenario	Avg. Queuing Time (s)	Avg. JCT (s)	Preemption Ratio
Basic	2231	13872	13.22%
Advanced	1944	12474	10.04%
Ideal	1273	9982	10.93%

Table 3.6: Performance without special placement of elastic jobs. Lyra naively places jobs based on the BFD heuristic.

Impact of imperfect scaling. Thus far we have assumed linear scalability of elastic jobs based on our empirical analysis in Section 3.2.2. Here we also evaluate Lyra when elastic jobs scale non-linearly with throughput loss. When one more worker is added to a job, we add a 20% loss to the throughput brought by this worker. Figure 3.8 presents Lyra's gains over Baseline with non-linear scaling. In Basic, average queuing time and JCT are 3.03% and 5.82% higher than those with linear scalability (Table 3.5 row 2). The degradation is mild because most training jobs are inelastic in Basic scenario and Lyra always satisfies their base demands. In Ideal, JCT is inflated by 10.54% to 9,828 seconds (compared to Table 3.5 row 5) due to the increase in job running time; the gain over Baseline is $\sim$1.7x.

Heterogeneous training. In Table 3.5, Lyra in the Heterogeneous scenario shows 1.58x and 1.37x reduction over Baseline in average queuing time and JCT. However, the preemption ratio is only 1% lower than Basic compared to 4.89% reduction in Advanced. We also manually enable heterogeneous training for more jobs in Figure 3.9. Intuitively, more jobs capable of heterogeneous training could bring more benefits to cluster efficiency. Job resource allocation could be more flexible. However, heterogeneous training leads to throughput loss and uses more resources to maintain the training progress than homogeneous training. Moreover, the availability of inference servers is subject to inference cluster traffic, and jobs may have to wait when few resources are available.

Therefore, the reduction of average queuing time approaches its asymptotic limit when 50% or more jobs support heterogeneous training.

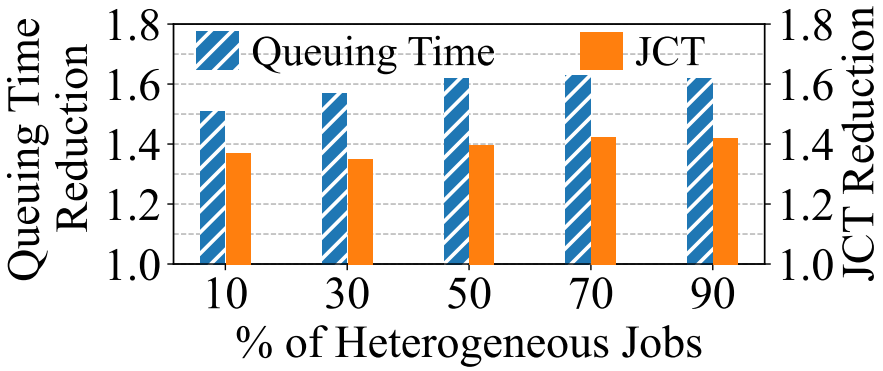


Figure 3.9: Average queuing time and JCT against Baseline in Heterogeneous.

Reproducibility of results. We also validate the reproducibility of the results. Here we compose ten 10-day training job traces based on the full 15-day trace in Section 3.7.1 using the bootstrapping technique. The cluster size remains the same. Figure 3.10 shows the results. Lyra's gains in queuing time and JCT are 1.45x and 1.44x in Basic, and 2.47x and 1.78x in Ideal. Lyra's performance is better when the training cluster has a long job queue. On weekends, training cluster is less busy. We notice that the gain in traces No.0 and No.4 is lower (10%) than others because two weekends are selected. On weekends, training cluster is less busy. Lyra's performance is better when the training cluster is busy and has a long job queue. Excluding these two traces, Lyra's improvement is statistically significant and consistent with results in Table 3.5 (rows 2 and 5). The average JCT improvement in Basic and Ideal shows a less than 4% gap with the performance improvement on the complete trace.

3.7.3 Deep-Dive: Capacity Loaning

We now dive into the two components of Lyra. We first focus on capacity loaning, aiming to understand its sources of gain and how our knapsack-based reclaiming

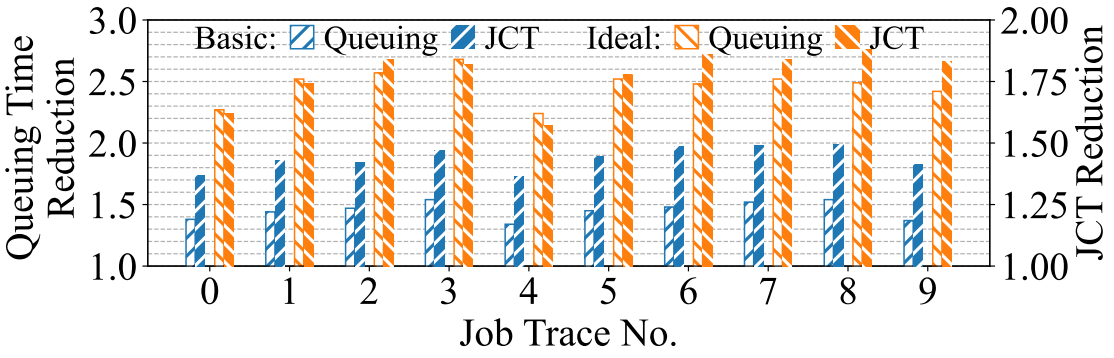


Figure 3.10: Average queuing time and JCT against the respective Baseline in Basic and Ideal for ten 10-day traces.

heuristic compares to other schemes. The results here are obtained without elastic scaling.

Scheme	Queuing Time (s)			JCT (s)		
	Mean	Median	95%ile	Mean	Median	95%ile
Baseline	4573	1283	23351	11547	2122	60170
Lyra	1119	274	7256	6887	1373	35776

Table 3.7: Queuing time and JCT of jobs running on on-loan servers.

Sources of gain. Table 3.5 (row 9) shows that loaning alone reduces average queuing time and JCT by 1.39x and 1.31x over Baseline. Loaning also improves the combined cluster usage from 52% to 66%. The JCT improvement mainly comes from the reduction in queuing time as jobs now can run on the loaned resources instead of waiting in the queue. Table 3.7 shows the statistics of queuing time and JCT for jobs running on the on-loan servers. The median and 95%ile queuing time is improved by 4.68x and 3.22x, respectively, compared to Baseline. The resource usage rate of on-loan servers throughout the experiment is consistently above 92% as depicted in Figure 3.11, which proves the effectiveness of resource loaning. We observe that JCT improvement of capacity loaning is not as significant as elastic scaling (Table 3.5 row 13). This is because (1) loaning

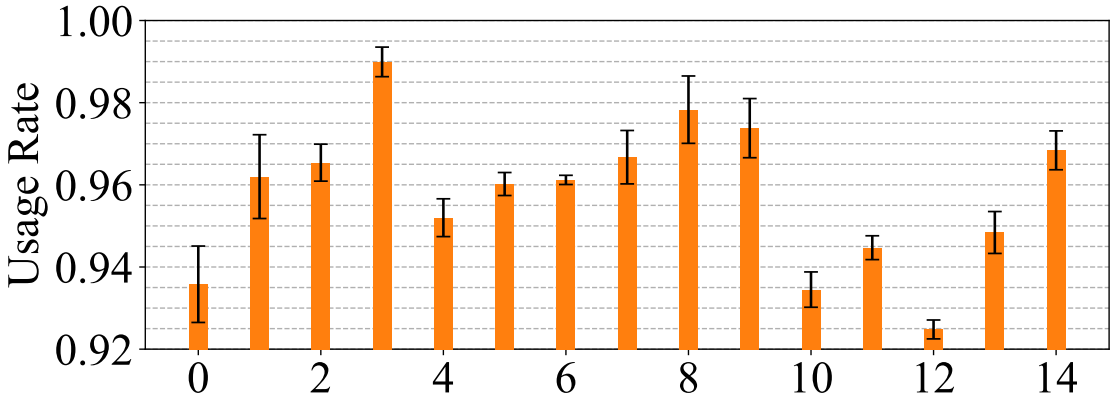


Figure 3.11: The daily average resource usage of on-loan servers (monitored every 5 minutes).

depends on idle inference resources and its gain is less stable, and (2) compared to scaling, loaning itself does not affect job running time.

Opportunistic scheduling. We then discuss why capacity loaning is more efficient than simple opportunistic scheduling. Table 3.5 row 6 shows the performance when the fungible jobs are scheduled opportunistically in the inference cluster. This does improve average queuing time and JCT over Baseline, but suffers 26.0% and 14.5% loss compared to Lyra (row 9). This is mainly because when fungible jobs are blindly put to inference servers, they suffer lower resource efficiency.

Reclaiming heuristic. We compare our reclaiming heuristic to Random and SCF. We use two metrics, the percentage of preempted jobs among running jobs, and collateral damage as the fraction of GPUs vacated in excess of the reclaiming demand. It is clear from Figure 3.12 Lyra outperforms others with and without elastic scaling. Without scaling, Lyra's knapsack-based heuristic reduces preemption and collateral damage by 1.51x, 1.68x and 1.36x, 1.59x over SCF and Random, respectively. With scaling, Lyra scales elastic jobs on the flexible server group first which further widens the gap. From Table 3.5, it is clear that reducing preemptions is beneficial: Lyra reduces the average queuing time and JCT by 1.26x, 1.15x and 1.31x, 1.13x over SCF and Random. We also

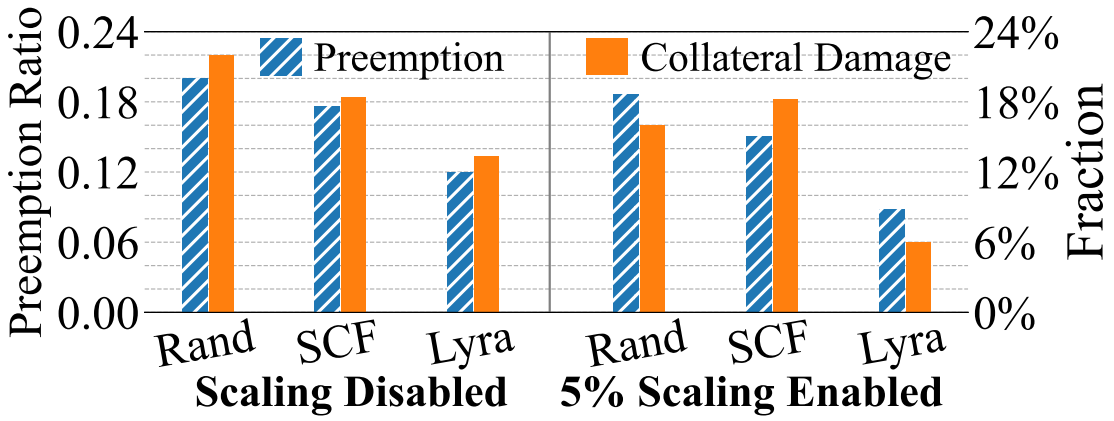


Figure 3.12: Preemption ratio and average collateral damage (defined in Section 3.7.3, Reclaiming heuristic).

run an exhaustive search to find the optimal reclaiming solution. Lyra results in the same number of preemptions as optimal when reclaiming fewer than 60 servers, and incurs 19% more preemptions otherwise. We compare the servers reclaimed by Lyra with the optimal solution. An average 84% of servers in the optimal solution are picked by Lyra's reclaiming decision. The average running time of the optimal solution, however, is 420k times that of Lyra.

Use of checkpointing. Checkpointing can effectively help a preempted job recover the training progress and resume from where they are interrupted. Since the scheduler cannot determine if a job has proper checkpointing or not, in our default setup we have made a conservative assumption that no jobs have checkpointing. Here we gradually increase the fraction of jobs with checkpointing enabled and present its impact on performance against the default case without checkpointing (Table 3.5 row 9). Figure 3.13 shows that prevalent checkpointing consistently improves Lyra: for example the preemption ratio is reduced to 0.26% and average JCT is reduced by 1.24x when 80% jobs have checkpoints.

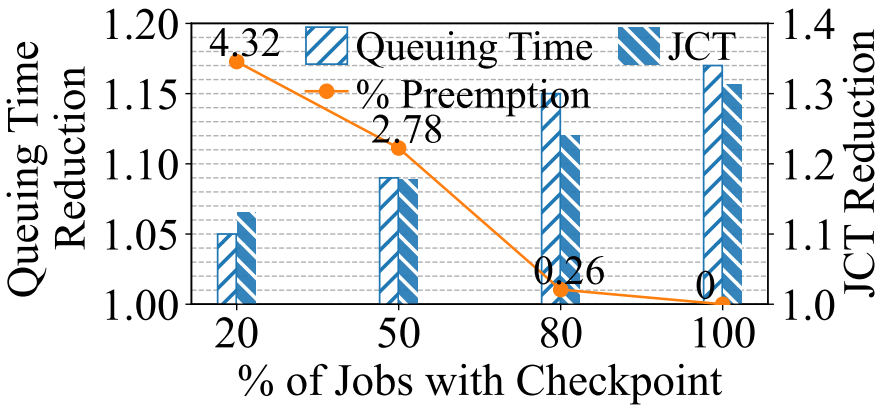


Figure 3.13: Average queuing time and JCT when jobs with checkpointing increase in Ideal.

3.7.4 Deep-Dive: Job Scheduling

We evaluate job scheduling in more detail here. The results are obtained without capacity loaning in Basic scenario.

Scheme	%ile Queuing Time (s)				%ile JCT (s)			
	50	75	95	99	50	75	95	99
Baseline	55	1892	8357	14323	791	29163	82933	376513
Gandiva	49	1764	6632	11806	755	27244	80567	323626
AFS	58	1297	5883	11124	721	12304	72123	323513
Pollux	47	772	3488	9031	686	20143	60883	247435
Lyra	47	697	3475	8731	602	12072	57597	223815
Lyra+TunedJobs	43	566	2749	7112	564	9293	52458	194391

Table 3.8: 50%ile, 75%ile, 95%ile and 99%ile of queuing time and JCT (Basic).

Sources of gain. Table 3.8 shows the queuing time and JCT distributions of all schemes. Our key insight in solving the scheduling problem is to prioritize the inelastic workload (Section 3.5.2). Gandiva does not improve Baseline much due to its opportunistic nature: it only scales jobs in low-utilization periods. Both Lyra and AFS allocate the minimum demand to each job initially. From Table 3.8,

they have similar median queuing time. Though Pollux considers job's minimum demand and favors those with large goodput, it does not explicitly launch as many jobs as possible, thus incurring longer queuing time. Lyra outperforms Pollux by 1.23x and 1.69x in median and 95%ile queuing time.

Turning to JCT, we find from Table 3.8 that Pollux tends to prolong the large-and-long jobs by shrinking their resources towards the end of training to yield for newly-started jobs that make rapid progress with the same resources. Moreover, Pollux's performance heavily hinges upon the problem scale and the number of iterations allowed for its genetic algorithm. In a large cluster of over 3,500 GPUs with heavy workload, the preset 100 iterations are not sufficient to get an efficient allocation result. To keep the scheduling overhead acceptable, we set the number of iterations to 250 and Lyra still has 1.20x and 1.25x improvements in median and 95%ile JCT. AFS assumes unbounded elasticity and shows a higher resource usage. However, unlimited elasticity and greedy allocation implicitly favor jobs with better throughput at the cost of others. Its average JCT is 1.2x that of Lyra which balances the resources each job gets by making global allocation and considering limited elasticity.

Sensitivity analysis: Proportion of elastic jobs. We wish to analyze whether Lyra is sensitive to the proportion of elastic jobs in the mix. Figure 3.14 shows the performance comparison when elastic jobs grow from 20% to 100% of the population. All schemes show improvements as a result. Lyra delivers the largest gains in both queuing time and JCT compared to other schemes with more elastic jobs, demonstrating that its scheduler most efficiently exploits job elasticity. AFS also has good gains in queuing time as it initially allocates minimum demand to each job. Its JCT gains, however, are much lower due to the greedy heuristic in ordering the jobs for allocation. Pollux's queuing time performance is poor as queuing time is not considered in its design. Its JCTs are much better because it auto-tunes the hyperparameters for the best performance.

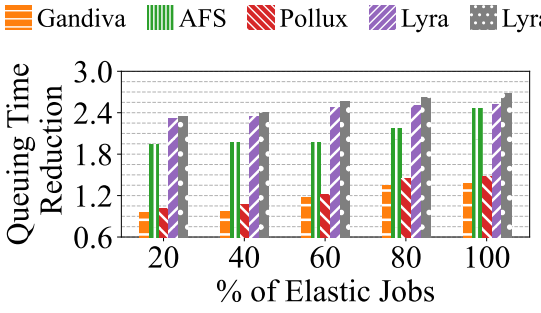


Figure 3.14: Queuing time reduction of Baseline as elastic jobs increase.

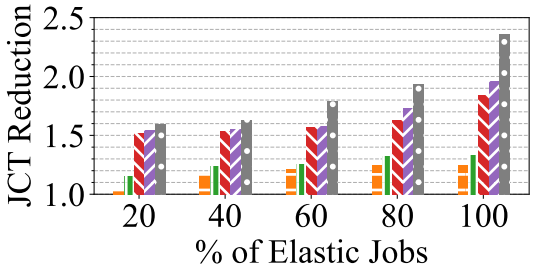


Figure 3.15: JCT reduction of Baseline as elastic jobs increase.

Sensitivity analysis: Error in running time estimation. Our second sensitivity analysis concerns the running time prediction which Lyra's scheduler relies on. Table 3.9 shows the performance under different estimation accuracy. Lyra improves queuing delay by 1.76x over Baseline even when there are 60% wrong predictions (each with at most 25% error). Its gain is consistent with less than 60% wrong predictions, which demonstrates its robustness.

% Wrong Prediction	Queuing Time Reduction	JCT Reduction
20%	2.21	1.52
40%	2.17	1.49
60%	1.76	1.38

Table 3.9: Queuing time and JCT reduction with incorrect running time estimation. The fraction of incorrect estimation varies from 0% to 60%. We assume each incorrect prediction has a random error margin within 25%.

Sensitivity analysis: Imperfect scaling of elastic jobs. In Figure 3.16, we plot Lyra's performance with non-linear scalability of training throughput, following the same setup discussed in Section 3.7.2. The average JCT improves by 1.86x when all the elastic jobs scale non-linearly. When the fraction of elastic jobs is less than 50%, non-linear scalability has less than 5% impact on JCT compared to linear scalability. Yet its impact on JCT grows (up to 9%) as elastic jobs become the primary workload, because they run slower due to non-linear scalability.

Meanwhile, the newly-arrived jobs have to wait longer for running jobs to vacate the resources, resulting in up to 7% increase in average queuing time.

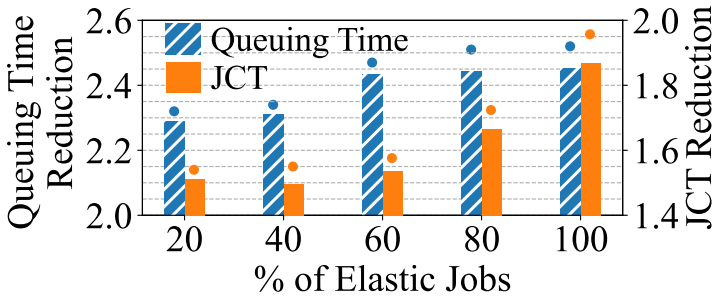


Figure 3.16: Lyra with non-linear scaling. Dots indicate the results with linear scaling.

Hyperparameter tuning. We study Lyra+TunedJobs now which adapts Pollux's job agent to tune jobs' hyperparameters as explained in Section 3.7.1. In the Basic scenario, Lyra+TunedJobs (row 14 in Table 3.5) enjoys additional 18% and 13% gains over Baseline in 95%ile and 99%ile JCT. This gain is more significant when all the jobs are elastic as seen in Figures 3.14–3.15.

More importantly, Lyra+TunedJobs allows for a fair comparison with Pollux as both have hyperparameter tuning. It outperforms Pollux by 1.32x and 1.37x in median and 95%ile JCT in Basic scenario (Table 3.8). Lyra's gain over Pollux is larger here which shows that Lyra's scheduling policy performs better in JCT. The main reason is that Lyra specifically optimizes JCT while Pollux optimizes goodput for resource efficiency. Thus JCT for some jobs is affected, especially near the end of training when the marginal gain of resources becomes smaller (i.e., goodput is lower) and resource allocation is decreased. Another side-effect of goodput-based scheduling is back-and-forth scaling as goodput varies as soon as the hyperparameter or allocation changes. We find the number of scaling operations in Pollux is 1.76x that of Lyra+TunedJobs in the Ideal scenario, and many are scaling-out followed immediately by scaling-in in the next interval.

3.7.5 Testbed Results

We use our prototype in testbed experiments to schedule jobs and YARN to run, scale, and preempt them.

Workload. We use a scaled-down version of the traces with 180 training jobs (10 elastic ones, similar to Basic scenario); jobs with (maximum) demand larger than 16 GPUs (50% cluster) are excluded. Job submission lasts for 8 hours and training time varies from 2 minutes to 2 hours. The inference trace is also scaled down according to the testbed capacity.

JCT and queuing time. Table 3.10 shows the statistics of queuing time and JCT. Lyra improves average and 95%ile queuing time by 1.38x and 1.36x over Baseline . In terms of JCT, Lyra improves the median and 95%ile by 19.9% and 11.7% over Baseline. The gains come from both capacity loaning and elastic scaling: the orchestrator performed 6 loaning and 8 reclaiming operations involving a total of 10 servers, and the scheduler issued 73 scaling operations. In capacity loaning, Lyra outperforms Random and SCF by 19% and 15% in average queuing time. In elastic scaling, Lyra's tail queuing time is 10% shorter than AFS. Its JCT gain is 1.19x over Baseline compared to 1.14x and 1.15x for AFS and Pollux. The results here show that Lyra is highly effective in reducing queuing time. The JCT improvements are relatively small due to the inference cluster's limited resources compared to job demand. We observe the inference cluster loaned at most three servers which is equivalent to one training server in computational capability, while it is common for a job to demand an entire training server in our trace.

Preemption. Figure 3.17 shows the total number of preemptions and the corresponding collateral damage in testbed. Lyra reduces preemption significantly by over 1.3x compared to Random and SCF reclaiming schemes (row group 2). We also measure the preemption overhead, including the time to save a checkpoint to the disk, terminate containers, launch new containers on different servers, and

Scenario	Scheme	Queuing Time (s)			JCT (s)			Preemption
		Mean	Median	95%ile	Mean	Median	95%ile	Ratio
Overall	Baseline	1532	772	1003	4078	2183	3096	0
	Lyra	1109	503	738	3335	1747	2731	18%
Capacity Loaning	Random	1527	658	993	3893	2046	3015	34%
	SCF	1473	614	864	3857	1994	3001	30%
	Lyra	1230	594	823	3748	1946	2864	22%
Elastic Scaling	Gandiva	1443	645	1002	3882	2015	2893	NA
	AFS	1338	534	882	3521	1836	2803	NA
	Pollux	1405	576	937	3552	1934	3004	NA
	Lyra	1318	546	798	3413	1791	2794	NA

Table 3.10: Testbed results using different schemes in Basic scenario.

load the checkpoint before training starts. The average overhead is 63 seconds, which is adopted in our large-scale simulation.

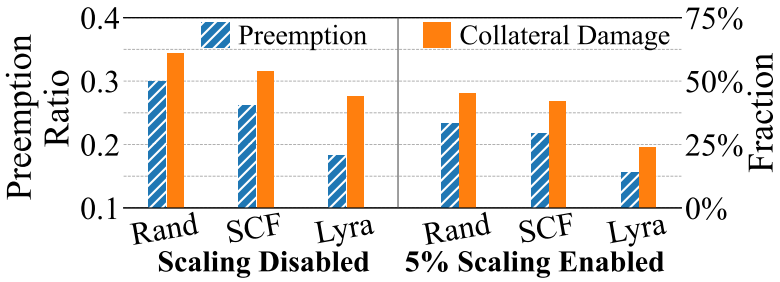


Figure 3.17: Preemption Ratio and average collateral damage comparison in testbed.

3.8 Discussion

Fine-grained resource sharing. Lyra uses physical machines as the basic unit of loaning and reclaiming. Our intention is to avoid interference between training and inference. This concern can be alleviated by improvements from the infrastructure (e.g. better isolation mechanisms). Then one may consider fine-grained sharing on the GPU level, which allows more sharing opportunities but also demands a more careful scheduling design because of the larger problem scale.

Strong and weak scaling. Strong and weak scaling is paramount for optimizing performance in distributed DNN. Strong scaling pertains to the ability of a distributed system to train a fixed global batch size with less time as more computation resources are added to the system. On the other hand, weak scaling addresses the system's capability to maintain a constant workload per computation unit while scaling both the global batch size and the number of resources proportionally. Effective utilization of strong scaling allows for faster training of DNN models for a given dataset size, while proficient weak scaling facilitates handling increasingly large datasets or more complex models with consistent performance. These concepts serve as fundamental pillars in optimizing the efficiency and scalability of distributed DNN training frameworks, essential for pushing the boundaries of machine learning capabilities in today's data-intensive applications.

Performance under scaling. We assume the elastic job's training throughput is linear in the allocated resources within the scaling range. In practice training throughput is likely to scale sub-linearly due to factors such as network communication and synchronization overhead. An improved approach may be to empirically profile the throughput and running time of the workloads as a non-linear function of resources. Lyra's scheduling algorithm still works with non-linear scaling which does not change the combinatorial nature of the problem; we provided simulation results in Section 3.7.2.

Heterogeneous GPU training. Training with heterogeneous GPUs is an active area of research and current mechanisms are primitive [18]. We observe that though adjusting the batch size can roughly synchronize the workers, it may prolong the training convergence in some cases. More effort is needed to improve training efficiency with heterogeneous GPUs and to automate hyperparameter adjustment [17, 100].

3.9 Related Work

GPU cluster schedulers. We have discussed Pollux, AFS, and Gandiva extensively in Section 3.2.3 and Section 3.7.2. Tiresias [46] applies least-attained-service to minimize average JCT. It does not consider elastic scaling. Optimus [111] predicts the training time by modeling the loss convergence speed and designs a heuristic to minimize average JCT. Predicting a DNN's convergence, however, is challenging as discussed in [51]. PAI [154] introduces a scheduler which reserves high-end GPUs for high-GPU tasks and packs low-GPU tasks on less advanced GPUs. These works all schedule jobs in a cluster with a fixed capacity.

Systems support for elastic scaling. There is emerging interest in exploiting resource elasticity in distributed training. Systems such as [74, 53, 117] extend various ML frameworks to support elasticity. [106] proposes an auto-scaling policy by considering both cost and scaling efficiency. AntMan [159] provides a scaling mechanism to micromanage computation and GPU memory during training, and a job scheduler for performance guarantees. They are complementary to Lyra as they provide practical solutions for scaling DNN jobs.

Dynamic resource allocation. Graphene [45] and PriorityMeister [176] dynamically adjust resource allocation to fit job's time-varying demand and utilize resources more efficiently. In Lyra, we consider scaling for jobs that can work with a range of resources, which are taken as constraints to the scheduling problem. Lyra schedules jobs with an extra dimension of how much resource should a job get and its impact on cluster performance.

Chapter 4

Lina: Accelerating Distributed MoE Training and Inference

4.1 Introduction

Recent advances in deep learning have shown that a model's quality typically improves with more parameters [15, 32, 145, 37, 69]. Many new frontiers in Computer Vision (CV) and Natural Language Processing (NLP) have been explored using large dense models [130, 34, 115]. While effective in terms of model quality, the computation cost of model training and serving is extremely high. ChatGPT [16], an impressive chatbot released by OpenAI, is estimated to spend 3 million dollars per month to serve user requests. Wider adoption and development of these models are impeded by the exorbitant compute cost.

Following the basic idea of curbing the computation cost of massive models, *sparsely activated* models have recently been introduced [12, 130, 37, 84]. The *Mixture-of-Experts* (MoE) structure is now one of the most popular ways to implement sparse activation [130, 12, 13, 178]. For each input, instead of using all parameters, an MoE model selects just a few of them, i.e. *experts*, for processing. This leads to sub-linear scaling of FLOPs needed with model size. Recent

literature [6, 177, 34, 115, 62, 73] has proven the potential of MoE models. For instance, Google develops a family of language models named GLaM using MoE [34]. Compared to GPT-3 with 175 billion parameters, the largest GLaM has 1.2 trillion parameters while only consuming 1/3 of the energy for training. Meanwhile, GLaM still achieves better zero-shot and one-shot performance than GPT-3. Microsoft reports that their MoE-based language models achieve a 5x training cost reduction compared to a dense model with the same model quality [115].

Given the uptake of MoE, there have been several systems for efficient MoE training and inference, including Google's Mesh TensorFlow [131], Meta's FairScale [9], Microsoft's DeepSpeed [30] and Tutel [144], etc. They provide APIs for users to replace the conventional dense layers with MoE layers with minimal code changes. They adopt both data parallelism and expert parallelism to accelerate the training and inference. That is, each device (e.g. GPU) is assigned with a unique expert, and uses all-to-all to receive inputs from other devices and then sends the gradients back to them accordingly. During training, allreduce is then used to aggregate non-expert gradients in the backward pass.

We focus on the efficiency of distributed MoE training and inference in this work. As some [124, 79, 50] has shown, the all-to-all operation is the main bottleneck. All-to-all blocks the subsequent computation operations and needs to be invoked two times in the forward pass and another two in the backward pass for each MoE layer. Interestingly, the main causes for all-to-all being the bottleneck are different in training and inference. In training, all-to-all and allreduce often contend for network bandwidth when they overlap in the backward pass, leading to a prolonged blocking period to the computation. Inference, on the other hand, presents a highly-skewed expert popularity driven by real-world requests. Devices with popular experts have to handle much more data than others. Not only does it delay the launch of all-to-all, but it also causes

imbalanced transfer size and bandwidth utilization across the devices, both of which are detrimental.

We are thus motivated to systematically tackle the all-to-all bottleneck. Our solution is Lina, a system that accelerates both MoE training and inference.

In training, we prioritize all-to-all over allreduce in order to improve its bandwidth. Existing MoE systems launch separate CUDA streams for the expert-parallel and data-parallel process groups which correspond to all-to-all for expert and allreduce for non-expert parameters, respectively. As there is no coordination between these streams, all-to-all and allreduce can overlap and fair-share the network bandwidth. Unlike allreduce, all-to-all is blocking and cannot be made parallel with the computation process. Thus, prioritizing all-to-all in the backward pass and avoid concurrent allreduce is crucial to reducing the blocking period.

To efficiently prioritize all-to-all, we adopt tensor partitioning which breaks down a tensor into smaller chunks, each of which forms a micro-op. With micro-ops, simple priority scheduling can be applied to guarantee full bandwidth for all-to-all while allowing allreduce micro-ops to make progress when all-to-all is not present. In addition, micro-ops allow the expert computation to be pipelined with all-to-all.

In inference, we dynamically schedule the resources for each expert in order to balance the workload of each device, thereby alleviating the imbalanced all-to-all transfer size and bandwidth. Intuitively popular experts should be given more resources while the rest may be served with less resources. The key challenge here is to efficiently and accurately obtain the expert popularity *before* the selection is actually done by the gating network, for every batch of input at each MoE layer, so scheduling benefit can be maximized with minimal overheads. Fortunately, we find the experts selected by each token across the layers demonstrate clear patterns, which allow us to estimate the expert distribution of the upcoming

layer based on the past selection results from the preceding layers. We adopt a two-phase scheduling approach that fine-tunes the estimation based allocation only when the actual expert popularity deviates too far.

We build Lina based on DeepSpeed MoE [30] and PyTorch, and evaluate it on a cluster with up to 16 Ampere A100 GPUs with 40GB memory and 100Gbps InfiniBand. Results show that Lina accelerates all-to-all by at least 2.21x, and achieves on average 1.57x speedup in overall training step time compared to state-of-the-art system DeepSpeed. The median and 95%ile inference time is reduced by 1.45x and 1.63x.

Our contributions can be summarized as follows:

- We present an in-depth empirical analysis of distributed MoE to show the main causes for all-to-all to be the performance bottleneck in training and inference.
- We propose to prioritize all-to-all over allreduce in order to improve its bandwidth and reduce its blocking period in distributed training. Lina's scheduler incorporates tensor partitioning and pipelining to perform micro-op scheduling.
- We examine the pattern in expert selection of MoE layer and propose to estimate the expert popularity to conduct resource scheduling in advance during inference. Lina adopts a two-phase scheduling scheme to minimize the overhead.
- We implement a concrete prototype system and conduct comprehensive testbed experiments to demonstrate the benefits of our design in a realistic GPU cluster setting.

4.2 Background and Motivation

We start with an introduction on MoE and a widely-adopted distributed system for MoE model in Section 4.2.1. Then, we motivate our idea by analyzing the performance bottleneck (i.e. all-to-all) in distributed MoE training and inference in Section 4.2.2.

4.2.1 A Primer on MoE

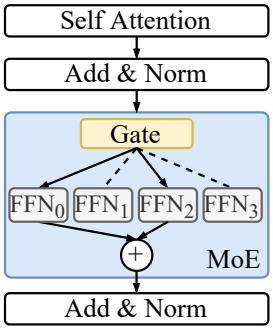
Mixture-of-Experts (MoE) has been adapted to different types of DNN models , and exhibits great potential in improving the performance of language models in particular. GShard [79] and Switch Transformer [37] are two seminal works on scaling Transformer-based language models with MoE layers. We focus on MoE in Transformer-based models in this work.

Transformer-based models normally use an MoE layer to replace the feed-forward network (FFN) layer. An MoE layer consists of multiple FFNs each serving as an *expert*, and a gating network (Figure 4.1a). Every expert is a fully-connected two-layer network using ReLU activation but with different parameters. The gating network takes in the embedding vector of each token and multiplies them with its trainable matrix. Based on the results, it dispatches the token to a small number of experts (usually one or two). The final output of the MoE layer is the weighted sum of outputs from the selected expert(s). The sparsity nature of MoE improves the model scaling in size without increasing the training cost and naturally leads to a dynamically-changing model graph.

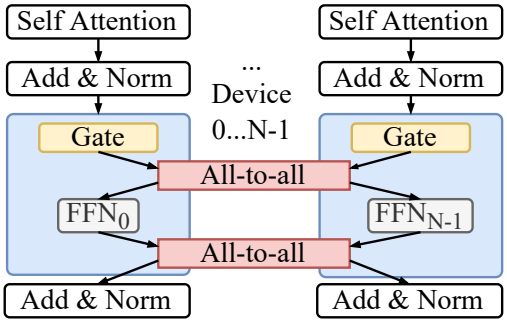
Load balancing loss. In MoE training, an auxiliary loss is introduced to evaluate the token distribution among the experts [37]. The objective is to achieve a uniform distribution of tokens across the experts, thereby preventing an excessive concentration of tokens in a single expert. By minimizing this loss term, we encourage but do not enforce the gating network to produce a perfectly balanced

token distribution.

The standard practice is to calculate the auxiliary loss of each MoE layer and sum them with the training loss using an appropriate weight. Previous research has demonstrated the effectiveness of this approach [19, 84, 133]. However, it should be noted that achieving a perfectly balanced distribution, where the auxiliary loss converges to zero, is challenging [174, 22]. During MoE inference, the trained gating network is utilized to dispatch tokens to the experts based on their respective embeddings. This process is solely driven by the characteristics of the token embeddings.



(a) There are four experts and the gate selects two experts.



(b) Distributed MoE. Data parallelism and expert parallelism are used.

Figure 4.1: MoE layer in Transformer-based models.

Hybrid parallelism in distributed MoE. Training and serving MoE models in a distributed manner are necessary due to the tremendous compute requirement of large-scale language models [11]. For efficiency, both data parallelism and MoE-specific *expert parallelism* (as a form of model parallelism) are applied [79, 37]. Existing MoE systems [79, 37, 131, 9, 30, 144] allocate one unique compute device (e.g., GPU) for each expert in expert parallelism. An all-to-all communication is then needed to send tokens to their experts selected by the gating network, and another all-to-all is needed to send tokens back to the device they belong to in data parallelism to finish the rest of the forward pass as shown in Figure 4.1b.

4.2.2 Bottleneck Analysis

Much prior work has identified that the introduction of all-to-all in MoE causes performance inefficiency in Transformer-based models [144, 124, 177]. We extract the completion time of all-to-all operations in both training and inference in our GPU cluster as shown in Table 4.1. All our experiments in this section use the same testbed and settings. Overall, all-to-all takes an average of 34.1% — a significant fraction of the step time. Interestingly, though the bottleneck brought by all-to-all is universal in both MoE training and inference, the causes differ. In the following, we motivate our work by analyzing how all-to-all affects the efficiency of training and inference, respectively.

# Experts	Model	Training (ms)		Inference (ms)	
GPUs	#Layers & Params	All-to-all	Ratio	All-to-all	Ratio
4	12L + 117M	259	36.7%	73	27.4%
	24L + 233M	589	35.4%	103	26.2%
	36L + 349M	979	38.2%	153	28.3%
16	12L + 419M	333	39.5%	102	32.5%
	24L + 838M	715	37.6%	177	31.7%
	36L + 1.2B	1145	36.8%	243	27.4%

Table 4.1: The completion time of all-to-all and its ratio in training and inference task of Transformer-XL [29] in different number of experts per layer. Training and inference have the same batch size here. Each FFN layer is replaced with MoE and the number of experts is equal to the number of GPUs similar to the common practice [37]. A100 GPUs with 40GB memory and 100Gb/s InfiniBand are used. We use the MoE implementation in DeepSpeed.

Synchronous all-to-all with large data transfer. The common characteristic shared by MoE training and inference is all-to-all's large data transfer. All-to-all is an irreplaceable synchronous component to handle the data exchange among devices in MoE layer. Each MoE layer has two all-to-all operations to send the tokens to the experts and then restore the position of tokens, as introduced

in Section 4.2.1. The data transfers in the two all-to-all operations have the same size because the expert's FFN architecture ensures that its input data size is the same as the output data size. Figure 4.2 shows an empirical timeline view of the forward pass of MoE model in our cluster. All-to-all takes 74.9% of the end-to-end running time of one MoE layer. Expert FFN computation and the combine operation follow when all-to-all operation completes. MoE training and inference suffer from such inefficiency consistently. GPU is mostly idle during this period: We use the PyTorch Profiler [119] to profile the GPU activities for 20 steps in each experiment in Table 4.1, and find that the average GPU SM efficiency during all-to-all is 3.7%. Besides, the data transfer size grows linearly with the number of experts. Figure 4.3 presents the empirical evidence of all-to-all's transfer size as the number of experts grows from 2 to 16 (128). With the increasing number of experts, the time taken by all-to-all grows from 33.4% to 44.5% of the step time.

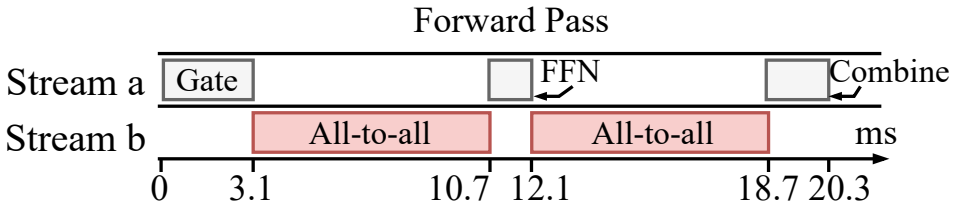


Figure 4.2: Timeline of forward pass an MoE layer. We simplify the presentation by bundling GPU kernels here: The computation kernels are grouped by their roles in the MoE layer into Gate, FFN and Combine. The Combine operation involves reshaping the tensors and computing the weighted output. The timeline is taken from a sample run of the 419M-parameter model in Table 4.1.

Problem in training: Prolonged all-to-all with allreduce. The unique challenge in MoE training is that applying the hybrid parallelism creates a particularly severe impact to all-to-all in backward pass. Non-MoE layers in *data parallelism* need *allreduce* to aggregate the gradients, while *expert parallelism* requires all-to-all to exchange tokens to compute expert gradients. Since the two operations control their own process groups independently, two dedicated CUDA streams

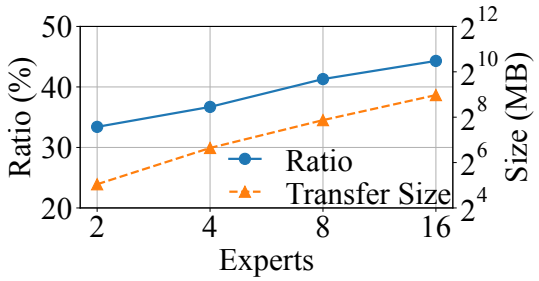


Figure 4.3: The proportion of all-to-all's completion time over training step time when the number of experts grows. Dashed line plots the data size in one all-to-all operation.

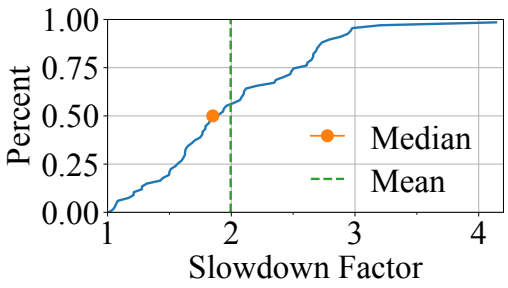


Figure 4.4: CDF of how much all-to-all is prolonged when it overlaps with allreduce operation. We mark the median and average slowdown factors.

are launched concurrently. This is demonstrated in Figure 4.5 with the timeline of backward pass in a sample run of MoE training. As the two operations overlap, they contend for the network bandwidth and their completion times are severely prolonged. To make matters worse, we find that the slowdown factor varies significantly. We collect the completion times of 1,500 all-to-all operations in backward pass on our testbed and plot the CDF of the slowdown factor they endure with allreduce in Figure 4.4. Observe that the median slowdown is over 1.83x and the worst is 4.14x.

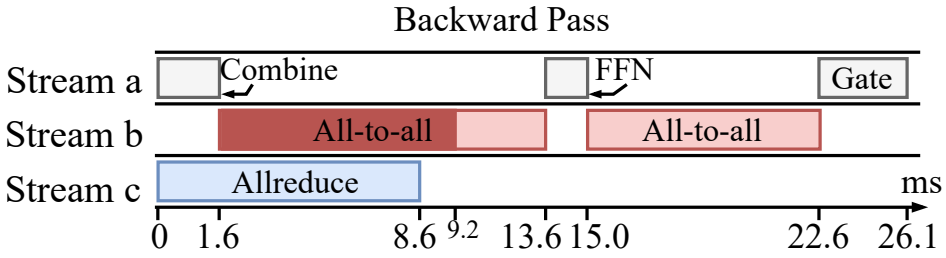
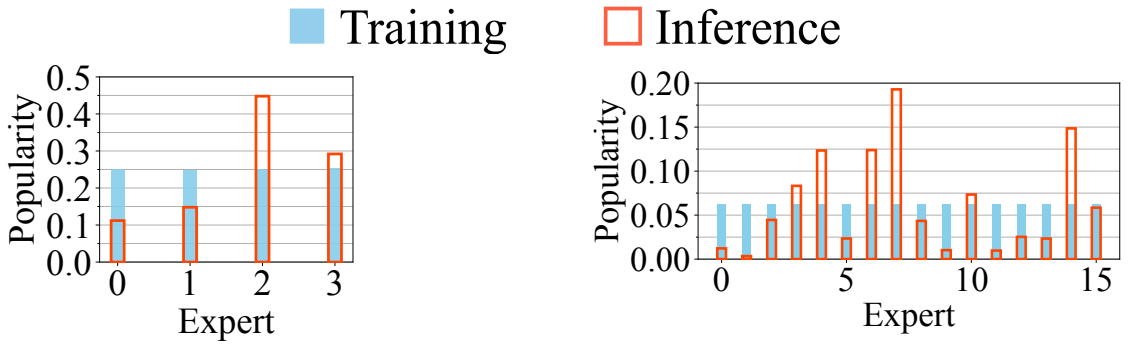


Figure 4.5: Timeline of backward propagating an MoE layer under hybrid parallelism. The first all-to-all is prolonged by the allreduce operation in Stream b. The shadowed part is its original completion time.

Problem in inference: Skewed expert popularity. The main cause of all-to-all being the bottleneck in MoE inference is the skewed expert popularity. The token-to-expert distribution in inference is purely workload-driven, and we

empirically find that the expert popularity is highly skewed in sharp contrast to training. We sample the expert popularity of the same MoE model in training and inference in Figure 4.6. In training, the distribution is nearly the same across all experts after hundreds of steps due to the use of load balancing loss. In inference, however, the most popular expert receives 4.02x and 5.56x tokens of the least popular ones in 4-expert and 16-expert inference tasks. With the same network and computation capacity, devices hosting popular experts take much longer to perform expert computation. In this experiment, the maximum idle time of the least popular expert is 29.4% of the inference time of that batch. Thus, within one batch, tokens to the less occupied experts have to wait for others to complete on the more popular experts, degrading the all-to-all performance significantly. Further, under uniform expert-device allocation, devices hosting popular experts have more tokens using their links for all-to-all, while the links of other devices are underutilized.



(a) 4-expert MoE.

(b) 16-expert MoE.

Figure 4.6: Sampled expert popularity. The distribution is computed as the ratio between the number of tokens received by the expert and total number of tokens in one batch. We use the Enwik8 test set [36] for evaluation.

4.3 Design Overview

Lina is designed to accelerate all-to-all in distributed MoE. It attacks both the bandwidth contention with allreduce in training, as well as the straggler with unbalanced all-to-all bandwidth in inference. We focus specifically on MoE implementations that leverage both data and expert parallelism.

MoE training. We aim to improve the *bandwidth* of all-to-all in order to reduce the blocking period of the computation operations. Our key idea here is to *prioritize all-to-all* so it does not fair-share bandwidth with concurrent allreduce (Section 4.4). This is achieved using tensor-partitioning. We partition all-to-all and allreduce tensors into small chunks, each of which then forms a *micro-op*. Lina schedules an allreduce micro-op only when there is no all-to-all waiting or ongoing so that all-to-all is guaranteed the full network bandwidth during its lifetime. Without prior information, tensor-partitioning and micro-ops can ensure that in most cases all-to-all can launch immediately and allreduce is not deferred excessively.

MoE inference. We propose to dynamically adjust the device allocation for experts based on the *expert popularity*, so that is not all-to-all is not delayed by the trailing tokens, and its bandwidth utilization across links is balanced (Section 4.5). We exploit the expert selection pattern across adjacent layers to estimate the expert popularity. Based upon the estimation, Lina performs scheduling at each layer to allocates proportionally more devices for popular experts and pack unpopular ones to fewer devices, and coordinate all-to-all correspondingly.

4.4 Prioritizing All-to-All Training

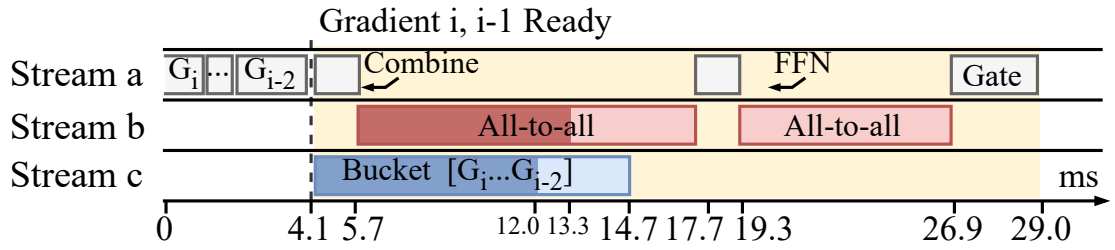
We have shown that all-to-all is slowed down significantly if it overlaps with allreduce in the backward pass in MoE training. Lina partitions the communica-

tion operations into small micro-ops and schedule them strategically in order to prioritize all-to-all without impeding allreduce and the computation process. We introduce the design challenges in Section 4.4.1. In Section 4.4.2, we present Lina's communication scheduler that uses tensor partitioning and pipelining to improve the training efficiency.

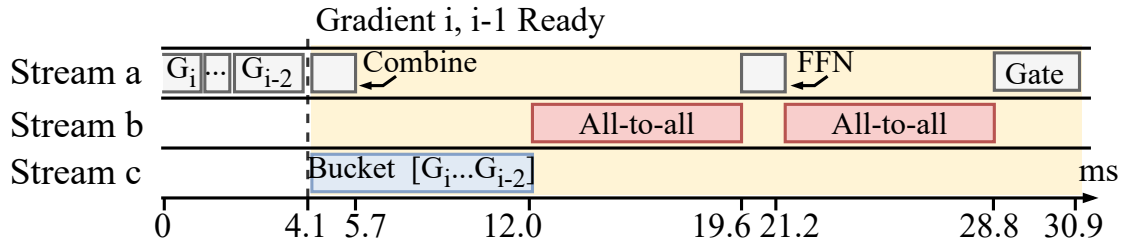
4.4.1 Design Challenge

Intuitively, Lina can prioritize all-to-all and avoid concurrent execution with allreduce with strict priority scheduling. All-to-all is always dispatched first if both are present in the queue, and subsequent operations have to wait until the running one finish to make sure allreduce does not share the bandwidth.

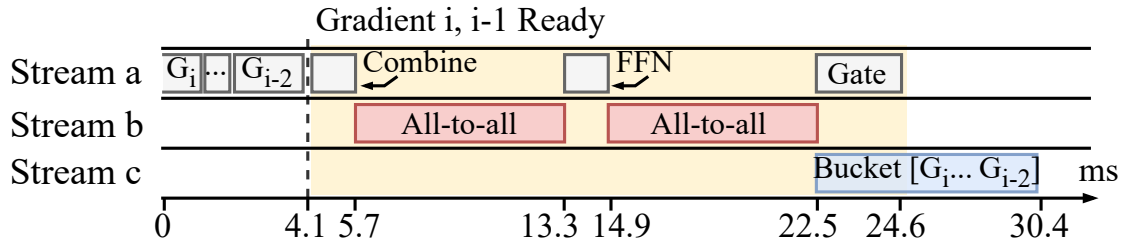
It turns out that simply prioritizing all-to-all is not as efficient as one may expect. For work-conservation, when an allreduce arrives first, it should be launched immediately. The problem is when an all-to-all arrives later, though ideally one would preempt the allreduce due to priority scheduling, this is not possible in current multi-GPU communication libraries such as NCCL [102]. The communication primitives are highly optimized and upon being called, their complete transmission strategies are settled and pushed to the CUDA streams. There is no control knob inside each primitive to adjust how it shares resources (e.g. CUDA cores, network bandwidth) with others. Thus, as the example in Figure 4.7b shows (based on testbed experiments), naively prioritizing all-to-all actually leads to a longer completion time for the first all-to-all and training step time compared to the baseline in Figure 4.7a.



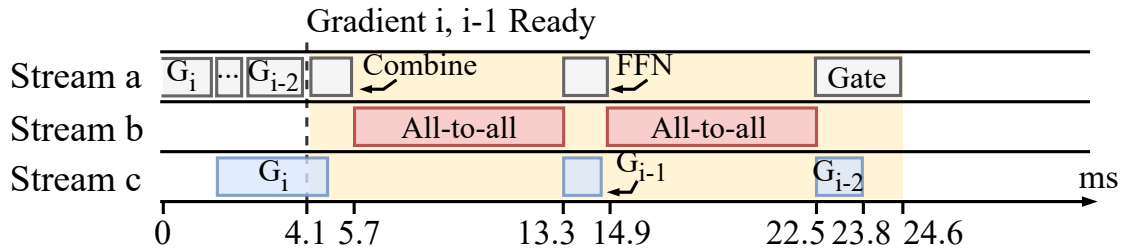
(a) Baseline. Shadowed all-to-all and allreduce are their completion times without concurrent operations. Computing the entire MoE layer's gradients ends at 29.0ms.



(b) Naively prioritizing all-to-all without concurrent transmission can lead to worse results; computing the MoE layer's gradients ends at 30.9ms. The completion time is profiled. Theoretically, the completion time should be the same as Figure 4.7a.



(c) Deferring allreduce to after the second all-to-all leads to better training efficiency; computing the MoE layer's gradients ends at 24.6ms.



(d) Scheduling results if the arrival time and running time of communication operations are known a priori. The allreduce completes much faster than (c).

Figure 4.7: Backward pass of MoE training. The yellow background is the period of computing the gradients of the MoE layer. Stream a is responsible for the computation process and streams b and c are for communication. This timeline is extracted from a real run of the 419M-parameter benchmark model in Table 4.1.

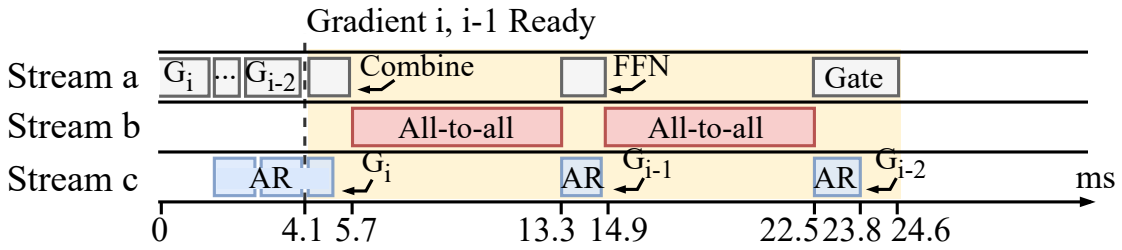
A potential solution is to obtain the arrival time and running time of the

upcoming all-to-all and allreduce, and orchestrate them accordingly to maximize the efficiency. Assuming we know that the allreduce for gradient i can complete before all-to-all and the completion time of gradient $i-1$'s allreduce is shorter than FFN computation. Then we can schedule gradient $i-1$'s allreduce to the gap between the two all-to-all operations at 13.3ms as depicted in Figure 4.7d. Obtaining the precise knowledge of arrival and running times is, however, a daunting task. ML frameworks such as PyTorch fuse gradients into buckets based on a user-defined bucket size to optimize allreduce efficiency. Yet in large Transformer-based models, gradient sizes are also large; since bucketing is done on the gradient boundary, the actual bucket size for allreduce varies wildly [118]. Moreover, the implementation details of allreduce make it difficult to acquire a reliable running time estimate as prior work has found out [24].

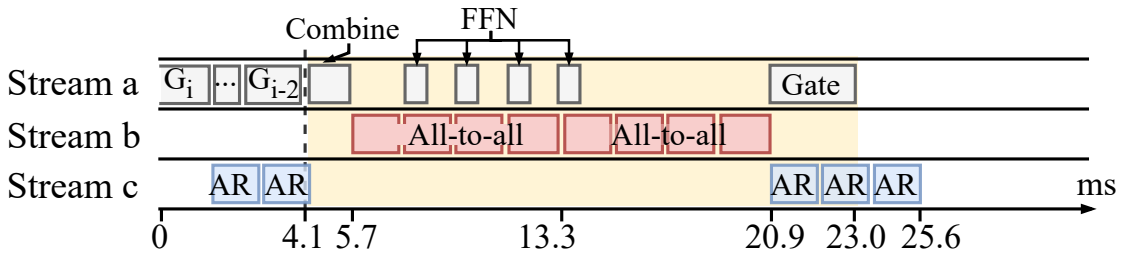
The other design choice is to blindly defer allreduce until an even number of all-to-all finish as there should be a larger gap between the backward pass of two MoE layers relative to FFN's backward computation. Figure 4.7c shows the best scheduling result based on the baseline in Figure 4.7a. In this case, allreduce can be launched when the second all-to-all finishes and completes before the first all-to-all of the next MoE layer (not shown in the figure). Yet, in other (worse) cases, allreduce may still block the all-to-all of the upcoming MoE layer if it takes relatively longer. In the extreme case, no allreduce can be launched until all four all-to-all operations of the current step finish. Since devices have to wait for allreduce before moving onto the optimization phase, this incurs more delay and is undesirable for wait-free backward pass [167].

4.4.2 Tensor Partitioning and Micro-Ops

To resolve the above challenges, we propose tensor partitioning that breaks down a communication operation into micro-ops, which can be easily prioritized with high efficiency.



(a) Prioritize all-to-all and partition allreduce tensors. Instead of bucketing gradients, we partition gradient i into three chunks when it is computed.



(b) Tensor partitioning for all-to-all and pipeline the FFN computation.

Figure 4.8: We show the scheduling results from Figure 4.7a with tensor partitioning. All-to-all and allreduce micro-ops are of the same size.

Tensor partitioning. Unlike tensor bucketing which fuses multiple gradients for an allreduce, Lina partitions each gradient tensor into equal-sized small chunks and executes individual allreduce *micro-ops* independently. This brings two advantages. First, it resolves the varying bucket size problem for allreduce since each micro-op is uniform in size now. Second, micro-ops naturally make better use of bandwidth [107] without causing too much delay to allreduce under priority scheduling. Consider the same setup from Figure 4.7a, in Figure 4.8a we partition gradients into five chunks. Before the first all-to-all arrives, Lina launches three allreduce micro-ops; after the first all-to-all ends, it starts another micro-op to opportunistically make use of the expert computation time. Compared to the scheduling result without micro-ops in Figure 4.7c, allreduce for gradient $i-2$ now completes 6.6ms or 21.7% faster without prolonging all-to-all. Tensor partitioning does incur overhead due to the partition and concatenation operations before and after an allreduce, but it is mild: the overall overhead

in Figure 4.8a's case is 764us. Section 4.7.2.2 has more details of the overhead analysis.

Pipelining micro-ops. Intuitively, we can also partition all-to-all which provides an opportunity to pipeline the expert FFN and further reduce the time that computation is blocked. Specifically, we can pipeline the expert computation and all-to-all micro-ops (Figure 4.8b). Since the FFN computation is in token granularity, the expert can start computing with a subset of the tokens after one all-to-all micro-op. With pipelining, we can eliminate the FFN time which is 1.6ms in this example.

Expert packing. Ideally, the FFN and all-to-all micro-ops should take a similar time so that both compute capacity and network bandwidth are fully utilized without any bubbles in the pipeline. However, we notice that a single FFN micro-op takes much less time than its corresponding all-to-all micro-op (Figure 4.8b). In Lina, we consider packing multiple experts on each device whenever possible to maximize the pipelining efficiency. Lina adopts the following approach: starting with one expert per device, it iteratively increases the number of experts per device in powers of two, until the FFN computation exceeds that of the all-to-all micro-op. In case of GPU memory shortage, we adopt DRAM-offloading [126] to transfer expert parameters that are not currently in use to host memory.

4.5 Scheduling Resources in Inference

Recall in Section 4.2.2, we have shown empirically that skewed expert popularity leads to unbalanced processing times across tokens of the same batch in MoE inference, which delays all-to-all and causes imbalanced bandwidth for it severely. The root cause lies in the data granularity mismatch between the expert and the attention layers in the model: an expert processes individual tokens, but the attention layer processes an entire sequence as a whole. Our design question is

thus: How can we ensure that each token within the same batch experiences the same end-to-end completion time no matter its expert selection result? We will first discuss the challenge of achieving this through dynamic resource scheduling (Section 4.5.1), and then present our design that exploits the unique token-level expert selection pattern to address the challenge in Section 4.5.2.

4.5.1 Design Challenge

To cope with skewed expert popularity, intuitively one must accordingly adjust the resource allocation for experts. This adjustment also needs to be done for each input sequence as the expert popularity distribution varies across sequences. An immediate question is: how can we know the expert popularity distribution, before the input is processed by the gating network?

This question is challenging for two reasons. First, even for a given batch of input, expert popularity varies across MoE layers of the model. We collect the expert popularity of different MoE layers for 1000 batches of input requests. Table 4.2 shows the top-4 popular experts of two 12-expert inference tasks: text generation and translation. Observe that each MoE layer of the same task (model) has completely different popular experts. This also suggests that dynamic resource scheduling has to be done before each MoE layer in order to be effective. Moreover, scheduling resources according to the actual expert selection results, as some might be thinking, incurs delay in collecting information, making scheduling decisions, and coordinating the all-to-all amongst all experts with respect to the new expert-device mapping, all of which are blocking operations and are performed at each layer. This is far from optimal (as will be shown in Section 4.7.3.1). Thus, we need to know as much as possible the expert popularity *before* the gating network selects experts in each layer, so these overheads can be largely overlapped with MoE computation.

4.5.2 Popularity based Scheduling

Lina tackles the design challenge by exploiting the token-level expert selection pattern which we empirically establish now. Building upon this, we design a resource scheduler that replicates popular experts on proportionally more devices in order to better balance the workload.

Pattern in expert selection. Experts in MoE models are trained to specialize in different types of input. We find that a token's expert selection demonstrates a pattern across the MoE layers. Tokens that have selected the same expert in layer i tend to select the same expert again in layer $i+1$. We trace the expert selection of sampled tokens. For each group of tokens that have selected the same expert in layer i, we calculate the ratio of them that in the next layer also select one of the same top-k experts ranked locally among the same group. Figure 4.9 plots this ratio averaged over token groups in two 12-layer MoE models. We see 41.94% tokens exhibit this pattern when k is 1 and 54.59% when k is 2, and deeper layers see more tokens with this pattern.

Model& Dataset	Layer	Top-4			
Transformer-XL & Enwik8 (Text generation)	3	9	4	5	10
	4	5	7	8	10
	8	9	2	3	13
	12	4	5	15	8
BERT-Large & WMT En-De (Translation)	6	7	6	10	1
	8	10	6	2	15
	10	9	4	11	8
	12	1	8	10	14

Table 4.2: Top-4 popular experts in sampled MoE layer of two MoE models.

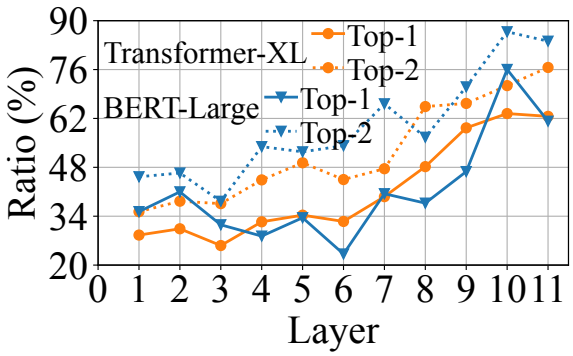


Figure 4.9: Ratio of tokens that select one of the top-k experts in layer $i+1$ given that they have selected the same expert in layer i.

This observation makes intuition sense. The gating network has a simple architecture, and their routing or expert selection decision is made (largely) based on relatively simple features, such as the parts of speech of a word (noun, verb,

etc.), and the meaning of the word (number, time, etc.) [80]. These features are fixed for each token. Meanwhile, experts focus on the local syntax information of each token rather than the cross-dependency within a sequence. For all these reasons, similar tokens naturally tend to be processed by the same or similar experts in each layer.

Estimating expert popularity. Though this pattern may not be sufficient to predict a particular token's expert selection accurately, it provides enough clues for us to estimate the overall expert popularity for a given batch. Specifically, Lina's estimation approach works as follows. In the profiling stage, we collect the expert selection results of all tokens when the load balancing loss is minimized and becomes stable. We then group tokens that select the same experts from layer $i-l$ to layer i, which represent a unique sample path of experts used. For each sample path j, we compute the expert popularity distribution Ψ_j^{i+1} for layer $i+1$. Here l is the path length to control the accuracy-cost tradeoff in profiling: a larger path length leads to more accurate estimation for layer $i+1$ at the expense of higher data collection and computation costs.

Then based on the profiled distributions $\{\Psi\}$, Lina can estimate the next layer's expert selection distribution for each sample path of experts traversed by a token in inference (starting from the l-th layer of the model). In each layer i, for a sample path j, we pick the top-k expert(s) of the subsequent layer from Ψ_j^{i+1} and use their probabilities $\{P_j^{i+1}(e)\}$ to represent expert popularity for resource scheduling, where e denotes an expert. The reason why we only consider top-k experts is that they demand the most resources, and the remaining experts have low popularity (Figure 4.9). Note that this estimation happens before any MoE layer computation takes place.

Two-phase scheduling. During inference, Lina dynamically conducts layer-wise resource scheduling in two phases.

The first phase happens right after the expert popularity estimation at each

MoE layer, when Lina relies on the estimation to replicate popular experts on more devices and pack unpopular ones onto fewer devices. Specifically, the total number of devices for expert e is determined by:

$$n_e = N \times \sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t, \tag{4.1}$$

That is, for the current batch of input with N_t tokens, using estimation from each token t's sample path $j(t)$ up to layer i, the overall popularity of expert e is estimated as $\sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t$ for layer $i+1$ accounting for all tokens. This requires the same proportion of devices assuming the expert parallelism degree is 1 (i.e. the number of devices equals the number of experts). For experts with the estimation n_e, we adopt the first-fit-decreasing heuristic to pack them into the empty devices so the total devices used are minimized. It is possible that some experts, being extremely unpopular (for this batch), are not amongst the top-k list of any tokens and thus do not have their n_e estimation. They are assigned evenly to the remaining free devices if any; otherwise are randomly assigned to a device.

In phase two, Lina fine-tunes the estimation-based scheduling decision after the gating network selects the actual experts. It checks if the selection result deviates significantly from the estimation, by comparing the overall top-2k experts. If the two lists are identical, no fine-tuning is needed and inference continues. Otherwise, the scheduler re-computes the resource allocation with the actual expert popularity now available following the same logic in phase 1. The fine-tuning phase does incur delay to collect the gating outputs and check against the estimation, which is necessary to deal with inaccurate estimation that turns out to be much more detrimental to performance, if left unchecked (Section 4.7.3).

4.6 Implementation

We implement Lina on DeepSpeed MoE and PyTorch using C++ and Python. PyTorch 1.10, CUDA 11, and NCCL 2.10 are used. We modify PyTorch's implementation of distributed training to support Lina in DeepSpeed. The implementation has ∼7500 LoC.

4.6.1 Training

Lina's communication scheduler for training is deployed on all devices and runs a single thread. Since the communication scheduling is purely local in scope, no coordination is needed across the scheduler instances on different devices.

Communication scheduler. Each scheduler instance maintains a priority queue to schedule the micro-ops. The micro-op size is passed in as a hyperparameter. Lina uses the built-in APIs `chunk` and `cat` in LibTorch to partition the data in the token dimension. We avoid putting chunks from different gradients into the same micro-op to simplify the subsequent concatenation operation. Moreover, the scheduler stops launching allreduce micro-ops if the combining computation in backward pass, since this implies all-to-all is imminent. We pipeline all-to-all micro-op in the MoE layer. FFN is ready to start right after each all-to-all micro-op.

Expert packing coordinator. We embed a packing controller in the MoE model and it runs a single thread. Expert packing is dynamically adjusted after 10 training steps. In the forward pass, the controller records the completion times of all-to-all and FFN micro-ops. When FFN micro-ops are shorter than all-to-all, the controller starts to pack experts. First, we initialize the new process groups. Second, the controller inserts a one-time synchronous all-to-all to exchange expert parameters between packed devices that would be invoked at the upcoming iteration. Finally, multi-stream parallel execution is adopted for both forward

and backward passes when more than one expert are hosted on a device.

4.6.2 Inference

Resource scheduler. The inference scheduler runs on a dedicated thread on device 0 of the cluster and manages resource scheduling. Each device saves the weights of all experts in their host DRAM and the collected layer-wise expert popularity distribution using multiple `unordered_map`, one for each layer. If GPU memory is in shortage, a device only loads one expert and the profiled distribution of one layer at a time.

In phase one of scheduling, all relevant communication happens by piggybacking the information on the regular all-to-all to reduce overheads. For each MoE layer, each device appends the popularity estimation to the first all-to-all for device 0. The scheduler computes the new expert-device mapping and instructs each device which expert and how many to host via the second all-to-all. We also include necessary information to coordinate all-to-all of the next layer, including the list of devices with the same expert, and how many tokens each replica should handle to balance the load. Devices then swap in the expert weights for the next layer. All these procedures are pipelined with model computation.

In phase two, each device updates the actual expert popularity in a separate NCCL `send` to the scheduler. If no fine-tuning is required, the scheduler broadcasts a resume signal. This only creates a negligible overhead as the transfer size is tiny. Otherwise, Lina broadcasts the fine-tuned expert-device mapping. The model computation is blocked during phase two until the scheduler's command is received.

All-to-all coordination. In inference, Lina uses all-to-all with an unequal split. That is, the transfer size to each device in all-to-all does not need to be the same. Using unequal split all-to-all can save the overhead of initializing multiple process groups. A placeholder data pointer is passed to all-to-all if no tokens are

directed to a certain device.

Expert packing. Expert computation is sequential on devices hosting multiple experts. Each device loads the experts one at a time to perform computation and move on to the next packed expert. In this manner, Lina avoids placing extra strain on the GPU memory. The second all-to-all is launched when the computation for all packed experts is completed. We set a maximum number of experts per device to control the overhead of swapping the weights.

4.7 Evaluation

We present the testbed evaluation results here.

4.7.1 Setup

Testbed setup. Our testbed has four worker nodes. Each node has 4 Ampere A100 GPUs with 40GB memory and is equipped with 100Gbps InfiniBand.

MoE models. We convert three common Transformer-based dense language models to MoE ones for training.

- Transformer-XL [29]: a 24-layer encoder model.
- BERT2GPT2 [156]: a 12-layer encoder-decoder model.
- GPT-2 [121]: a 12-layer decoder model.

Besides, we consider two inference tasks.

- Transformer-XL [29]: The inference task is text generation with Enwik8 [36] test set.
- BERT-Large [32]: a 12-layer decoder model. The inference task is translation using WMT En-De [155] test set.

All FFN layers in these models are converted to MoE layers. We vary the number of experts in an MoE layer from 2, 4, 8, to 16. We adopt top-2 gating in training and top-1 gating in inference following [37], i.e. $k = 2$ in training and $k = 1$ in inference

Metrics. We consider four metrics to evaluate Lina.

- Training step time: Time to complete one step of training.
- Inference time: Time to complete one batch of inference.
- All-to-all time: The completion time of all-to-all.
- MoE layer time: Time to complete one MoE layer of computation and communication.

In collecting these metrics we use PyTorch Profiler to obtain CUDA kernel execution time and GPU activities. Training results are averaged over 50 steps after a 10-step warm-up period. Inference results are averaged over the test set. Since the optimization introduced by Lina does not affect the precision of model parameters, model accuracy is unaffected and we omit its evaluation.

Training configurations. Lina's micro-op communication scheduler adopts a tensor partition size of 30MB, which can minimize the period blocked by all-to-all in most cases. Expert packing is launched at the 10-th step of each training task and is adjusted every four steps.

Inference configurations. Lina's resource scheduler runs on device 0. The path length l in popularity estimation is 3; the maximum number of experts packed on a device is 4.

Baselines. We use the vanilla DeepSpeed [30] as the Baseline. We also provide a comparison to the open-source version of Tutel [144], which performs similarly with DeepSpeed. We enable hierarchical all-to-all for both Lina and DeepSpeed and disable Random Token Dropping [162] introduced by DeepSpeed.

4.7.2 Training

We start with Lina's training performance. Note that Lina is evaluated when the expert packing decision is stabilized; all settings here use 2 experts per device as the best strategy except Transformer-XL with 16 experts, which uses 4 experts per device. The number of GPUs is equal to the number of experts per layer in both Baseline and Lina.

4.7.2.1 Overall Performance

Training step time. Figure 4.10 shows Lina's speedup in step time over Baseline and Tutel. All other aspects of the models stay the same (e.g. sequence length, hidden states dimension, etc.). Compared to Baseline (DeepSpeed), step time is reduced by an average of 1.37x and 1.47x for the 4- and 16-expert cases, respectively, and by an average of 1.71x and 1.73x for 2- and 8-expert models, respectively. The 2- and 8-expert cases see more significant gains as Lina's packs two experts per device as mentioned before. The 2-expert case thus boils down to pure data parallelism without any all-to-all; the 8-expert models avoid inter-node all-to-all as our servers have 4 GPUs each. Lina's speedup over Tutel is slightly smaller than that of DeepSpeed. Thus in the following we only use DeepSpeed as the Baseline.

MoE layer time. We specifically seek to understand Lina's gain in MoE layers in both the forward and backward pass. As Figures 4.11 and 4.12 show, similar to step time, the gain in the 2- and 8-expert cases is the largest. The forward and backward pass of MoE layers in the 2-expert case are accelerated by 1.84x and 2.41x, and in the 8-expert case by 1.89x and 2.32x, respectively. Since backward pass in Baseline suffers from the interference of allreduce while the forward pass does not, the improvement in the backward pass is more significant. Average GPU utilization in the MoE layer for 16-expert cases is improved by at least 16%

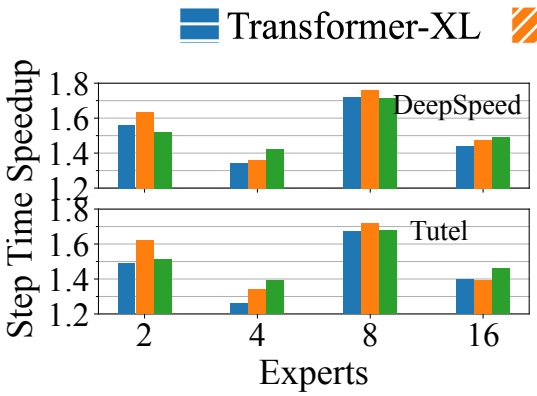


Figure 4.10: Speedup of training step time against two Baselines.

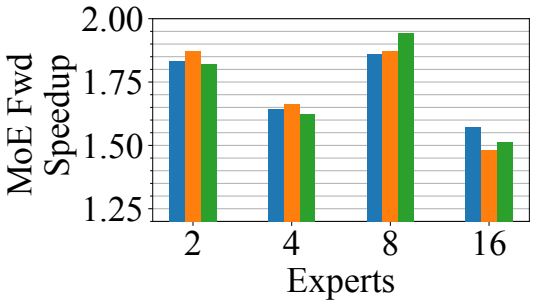


Figure 4.11: Speedup of MoE layer's forward pass completion time.

as the period blocked by all-to-all is minimized with Lina.

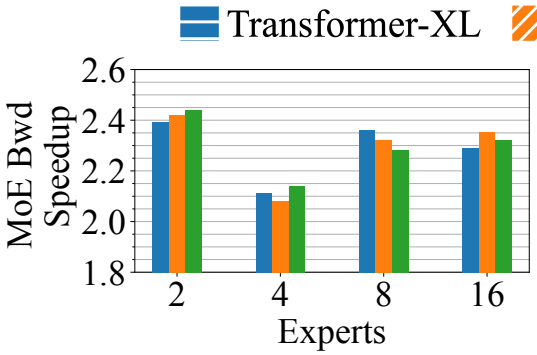


Figure 4.12: Speedup of MoE layer's backward pass completion time.

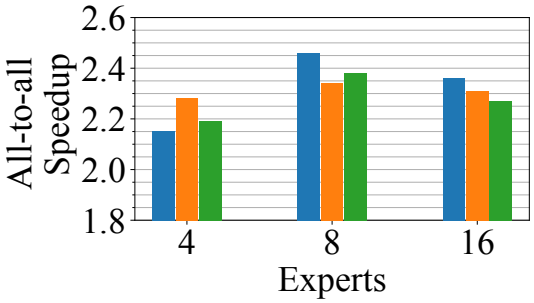


Figure 4.13: Speedup of all-to-all time in forward and backward pass.

GPU utilization and memory usage. We measure the average GPU utilization GPU memory usage (Table 4.3). We observe an average of 17.6% improvement in GPU utilization due to the efficient scheduling of Lina. Expert packing would lead to usage increase in GPU memory. The peak memory of BERT2GPT2 is increased by 19.5% while Transformer-XL and GPT-2 use up all the memory and apply DRAM-offloading to store the packed expert parameters.

All-to-all time. We then zoom in on all-to-all time in backward pass, where Lina prioritizes all-to-all and avoids concurrent execution with allreduce. Expert packing also reduces the all-to-all transfer size. Figure 4.13 shows an average

Expert	Model	Average GPU Utilization(%)		GPU Memory Peak Usage(%)		
		Baseline	Lina	Baseline	Lina	DRAM-offloading
16	Transformer-XL	66.2	83.4	72.1	100	✓
	GPT2	62.3	78.2	83.8	100	✓
	BERT2GPT2	63.5	82.5	74.3	94.2	✗

Table 4.3: GPU utilization and peak memory usage of 16-expert MoE models. GPU Memory Peak Usage is the ratio between the maximum usage and the total device memory. DRAM-offloading indicates if it is applied.

speedup of 2.21x, 2.39x, and 2.31x in 4-, 8-, and 16-expert cases in all-to-all time, respectively.

We also examine the pipelining efficiency between all-to-all and expert computation in Lina. We define the pipelining efficiency to be the fraction of non-idle time in the computation CUDA stream during the all-to-all duration. We calculate the pipelining efficiency of Lina before and after adopting expert packing in Table 4.4. The average improvement is 2.43x in 16-expert case, which also demonstrates the benefits of expert packing. The expert FFN micro-op time is thus closer to the all-to-all time. We find that two experts per device can achieve the best pipelining efficiency in most cases, justifying our settings mentioned before.

Expert	Model	Pipelining Efficiency		
		w/o Packing	w/ Packing (Experts per Device)	
16	Transformer-XL	33%	86%	4
	GPT-2	36%	85%	2
	BERTGPT2	34%	79%	2

Table 4.4: Pipelining efficiency comparison with and without expert packing.

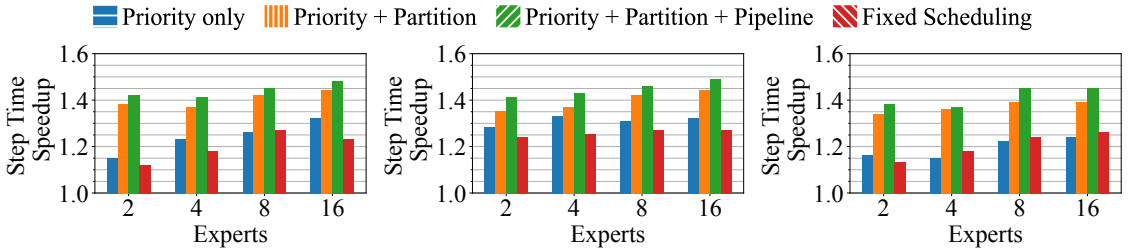
4.7.2.2 Communication Scheduler

We now present an in-depth analysis of Lina's priority-based micro-op scheduler, aiming to understand the benefit of each design choice. For fairness all

experiments here are obtained without expert packing in Lina, i.e. one expert per device.

Tensor partitioning and pipelining. To justify our design, we incrementally add the key design choices to Baseline and see their corresponding gain: first priority scheduling, then tensor partitioning, and lastly pipelining. Besides, we consider a fixed scheduling strategy where allreduce is always scheduled between pairs of all-to-all operations (i.e. two MoE layers) with tensor fusion enabled in PyTorch's `DistributedDataParallel` by default (same as Baseline).

Figure 4.14 shows the step time comparison. We make several interesting observations here. First, using priority brings about 10%–30% gain over Baseline in most cases, with an average of 24%. Priority scheduling in general presents more benefit when more devices and nodes are used in training. The main reason is that all-to-all's slowdown due to sharing bandwidth with allreduce is more severe as training scales out. Second, tensor partitioning significantly improves the benefit of prioritizing all-to-all: step time is reduced over Baseline by 1.36x, 1.36x, 1.41x and 1.42x in 2-, 4-, 8-, and 16-expert cases, respectively on average. On the other hand, pipelining's gain is limited as expected, since expert computation takes much less time than all-to-all without expert packing (recall section 4.4.2). Overall, all three design choices can effectively reduce all-to-all's completion time.



(a) Transformer-XL. **(b)** GPT-2. **(c)** BERT2GPT2.

Figure 4.14: Training step time speedup over Baseline with different design choices of the communication scheduler.

We also observe that the relative benefit of priority scheduling and tensor partitioning is model-specific: GPT-2 enjoys much more gain from priority compared to tensor partitioning while the other two models do not exhibit such clear pattern. This is likely due to the degree of overlapping of all-to-all and allreduce: most allreduce can fit in between all-to-all operations in GPT-2, and as a result using priority scheduling alone is very beneficial.

Finally, the fixed scheduling strategy leads to the smallest gains in almost all cases. This is because (1) all-to-all still has to fair-share bandwidth with allreduce, and (2) tensors are not partitioned which aggravates the impact of allreduce. This demonstrates again the effectiveness of our design in prioritizing all-to-all with smaller tensors instead of using fixed heuristics that cannot opportunistically maximize efficiency.

Partition size. We also evaluate the impact of partition size on the communication scheduler. Figure 4.15 shows the step time of 16-expert models when we gradually increase the partition size from 10MB to 100MB. We find that a partition size beyond 50MB slows down Transformer-XL and BERT2GPT2 compared with 30 MB. As long as the period blocked by all-to-all is minimized, step time would be minimum. Therefore, for each model and setting, there are multiple optimal partition sizes. Ideally, the scheduler can more precisely control the operations with a smaller partition size. In practice, small partitions (below 10MB) may cause heavy transmission overhead in each micro-op and degrade the overall performance [112]. **Overhead analysis.** We provide a brief analysis of the overhead incurred by Lina's communication scheduler. First, the preprocessing and postprocessing, including tensor partitioning and concatenation, take an average 1.02% of the step time. Second, we measure the transmission overhead of micro-ops. We sum up running times of all the communication micro-ops and compare against those without partitioning in Baseline. The average completion time is lengthened by 1.7%.

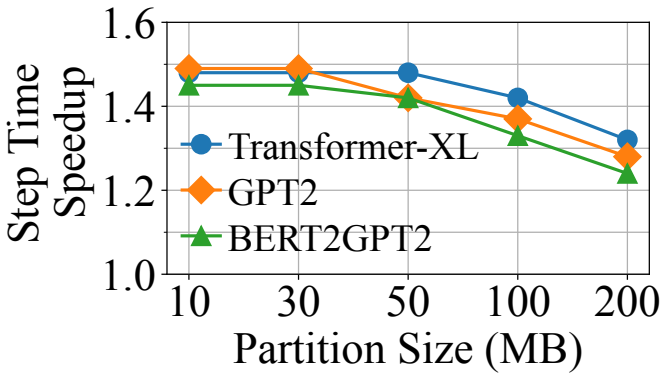


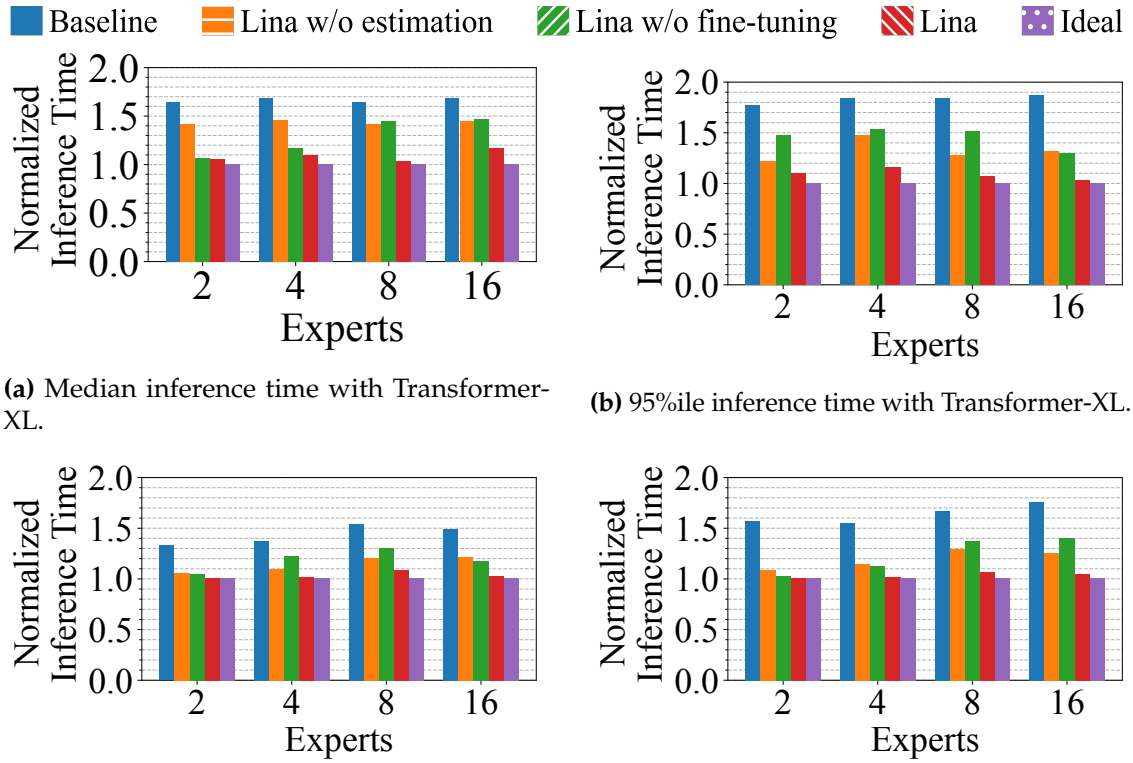
Figure 4.15: Partition size increases from 10MB to 200MB in 16-expert models.

4.7.3 Inference

We then evaluate Lina's inference performance. Each experiment is repeated five times: two of which measure the end-to-end inference time, and the rest profile the different components with Profiler and collect statistics for overhead and estimation accuracy. This way the inference time is not affected by the profiling overhead.

4.7.3.1 Resource Scheduler

Inference time. Figure 4.16 shows the median and 95% inference time of Baseline and Lina. We also present the ideal inference time with a perfectly balanced load across devices in all MoE layers. This is obviously challenging to achieve with real-world requests. Thus we modify the gating network to constantly output a balanced expert selection to obtain this benchmark. We normalize all results to the Ideal value. Lina's resource scheduler effectively balances the load among devices and achieve inference time close to Ideal. Compared to Baseline, median inference time is reduced by 1.54x and 1.45x for the 4- and 16-expert Transformer-XL, and by 1.36x and 1.46x for the 4- and 16-expert BERT-Large, respectively. The 95%ile inference time is reduced by 1.82x for 16-expert Transformer-XL and 1.68x for 16-expert BERT-Large. The reduction on tail inference time increases



(a) Median inference time with Transformer-XL.

(b) 95%ile inference time with Transformer-XL.

(c) Median inference time with BERT-Large.

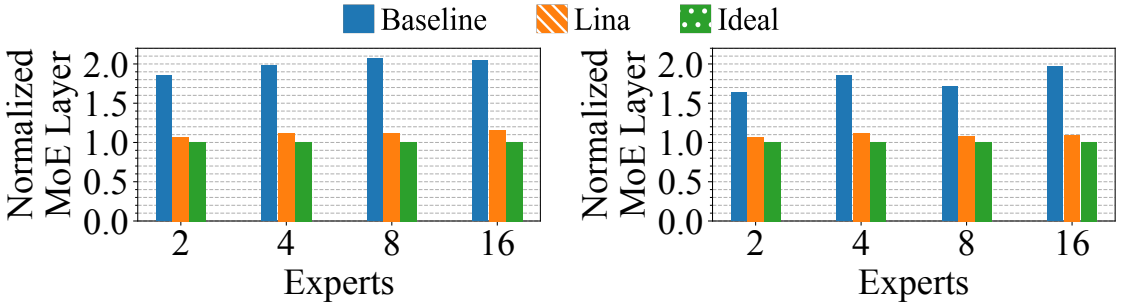
(d) 95%ile inference time with BERT-Large.

Figure 4.16: Median and tail inference time. We normalize the inference time with the ideal result. The median and tail inference time is the same in Ideal.

with more experts in a layer, because a wider MoE layer is more likely to present more skewed expert popularity, giving more room for Lina to optimize. Lastly, Lina's gap to Ideal can be explained for two reasons other than the overheads. First, Lina cannot perfectly balance load: the least popular experts are randomly placed for example. Second, Lina starts to schedule from the fourth layer.

MoE layer and all-to-all time. With Lina, MoE layer time includes gate computation, phase two of scheduling, two all-to-all, and expert computation; phase one of the scheduling is largely overlapped with computation as explained in section 4.6.2. The 95%ile MoE layer time is reduced by 1.87x and 1.77x in 8- and 16-expert Transformer-XL over Baseline and by 1.58x and 1.81x in 8- and 16-expert BERT-Large as in Figure 4.17.

We also extract all-to-all time, which is a direct indicator of whether Lina bal-



(a) Transformer-XL. **(b)** BERT-Large.

Figure 4.17: 95%ile completion time of MoE layer.

ances load across devices effectively. We present the tail all-to-all time reduction of different layers in Figure 4.18. The average and maximum improvements are 1.96x and 2.50x over Baseline. These results confirm that Lina effectively balances the load of each device and all-to-all transfer size of each link.

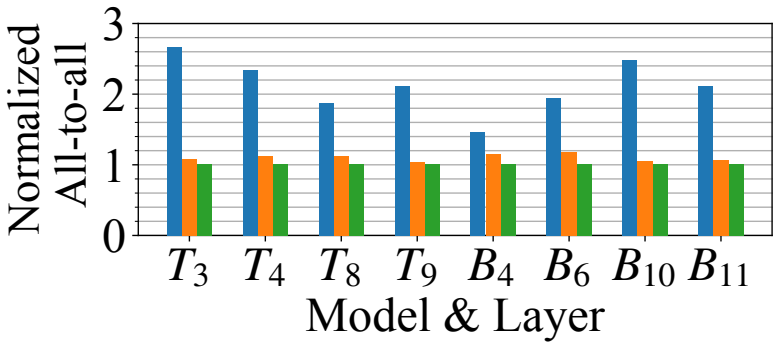


Figure 4.18: All-to-all time in 16-expert MoE. *T* is Transformer-XL and *B* is BERT-Large.

Two-phase scheduling. We then evaluate the effectiveness of our resource scheduler's design. We consider separately Lina without estimation and without fine-tuning in order to understand their individual gains. Lina w/o estimation refers to scheduling using the actual routing decision computed by the gating network.

In Figure 4.16, we present the comparison of inference time for all schemes. Without estimation the median inference time is worsened by 24.0% and 18.6% for 16-expert Transformer-XL and BERT-Large in Lina. The scheduler works

after the gating network and blocks all-to-all with the following computation until it completes. Thus the scheduling overhead manifests at each MoE layer, overweighing the additional gains brought by accurate popularity information. The tail inference time is less affected compared to the median, but still suffers without estimation.

Without fine-tuning, tail inference time is prolonged by 26.7% and 33.1% for 16-expert Transformer-XL and BERT-Large. This suggests that fine-tuning also plays an indispensable role when the estimation shows a large difference from the actual routing decision. For example, if the top-1 expert in the actual routing decision is estimated as an unpopular one packed with others, the Moe layer time would even be worse than Baseline. More discussions are presented in Section 4.7.3.2. The importance of fine-tuning depends heavily on estimation accuracy and number of expert in MoE layer.

Overhead analysis. We dissect the overhead of the resource scheduler, by considering the scheduling times of phase one and phase two separately. The scheduling time for both phases averages at $\sim$6.2ms since they share the same logic and coordination workflow. Yet, the overhead of phase one is largely overlapped with model computation. Though overhead of phase two with re-scheduling is more salient, it only kicks in for 23% of the cases on average, and is smaller than the idle time incurred by skewed expert popularity. The overhead of phase two without fine-tuning is merely 1.45ms.

4.7.3.2 Popularity Estimation

We now analyze Lina's popularity estimation method.

Estimation accuracy. We first examine the estimation accuracy across MoE layers. We resort to the same definition used in Lina's phase two scheduling: if the top-2 (recall $k = 1$ in inference) estimated experts are identical to the actual routing decision, we consider the estimation accurate. Figure 4.19 shows the accuracy for

every MoE layer in two inference tasks. Overall, estimation accuracy is 58.41% and 54.16% for Transformer-XL and BERT-Large, respectively. The estimation accuracy is higher in the latter layers of the model, which is consistent with our observation in Figure 4.9. We also compare the complete popularity rankings given by the estimation to better understand its accuracy. Using 1000 random batches of the Transformer-XL model, we observe that errors usually happen at experts with a similar popularity. An average of 3.67 experts out of the estimation are incorrectly ordered. Therefore, the fine-tuning only requires little adjustment to the experts packed together. The effectiveness of Lina's estimation can be justified.

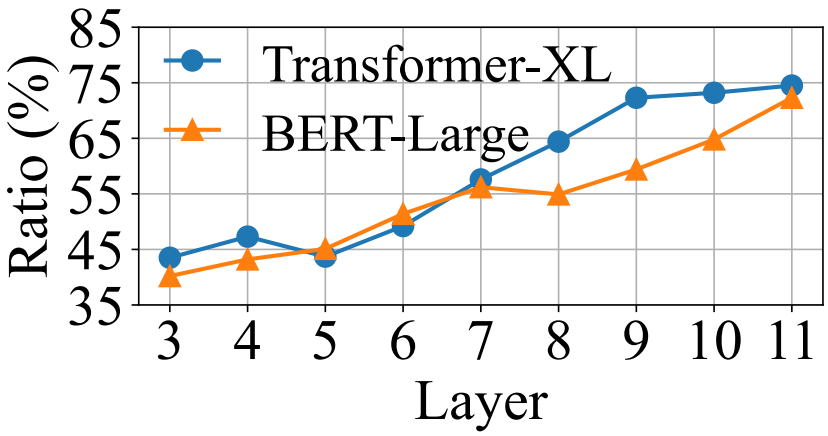


Figure 4.19: Estimation accuracy of 16-expert MoE.

Sample path length. We also investigate the impact of sample path length l. The longer the sample path of expert selection is for making an estimation, the more accurate the result is. Table 4.5 shows Lina's performance degrades with $l = 1$, in terms of inference time, estimation accuracy, and the occurrence of phase two fine-tuning, compared with the default length of 3. Longer paths can elevate the estimation accuracy and further reduce the number of times of Lina's fine-tuning. However, due to the problem of a slower start, the reduction of inference time is not as noteworthy as the estimation accuracy. For a path length of 6, Lina shows

a similar median and tail result as the performance with a path length of 3.

Model	Path Length	Norm. Inference Time Median	95%ile	Fine-tuning (%)	Estimation Accuracy (%)
Transformer-XL	1	1.41	1.32	76.5	31.6
	3	1.16	1.04	25.7	60.4
	6	1.19	1.11	22.5	71.4
BERT-Large	1	1.34	1.35	71.3	28.3
	3	1.07	1.04	32.2	63.5
	6	1.09	1.11	27.1	66.0

Table 4.5: Lina's performance using different path lengths during estimation. Both models have 16 experts per layer. Inference time is normalized to Ideal.

Generalizability. We proceed to evaluate how well Lina's popularity estimation approach can be generalized to different tasks. Table 4.6 shows the estimation accuracy of four tasks with different datasets. The 95%ile inference time can achieve at most 1.04x of the Ideal inference time and the estimation accuracy is at least 62.3%. Lina's estimation method relies on the patterns obtained from training stage. Therefore, it is tailored to each specific task and proves to be an effective approach to capturing the expert popularity prior.

Task	Dataset	Model	Norm. 95%ile Inference Time	Estimation Accuracy
Sentiment Analysis	IMDB Reviews [90]	BERT	1.08	64.4%
	Twitter [98]		1.11	62.3%
Translation (English)	WMT French [155]	T5 [123]	1.04	68.8%
	WMT Russian [155]		1.08	62.5%

Table 4.6: Lina's performance on different tasks and datasets. Inference time is normalized to Ideal. The path length is set to 3.

4.8 Discussion

Parallelism in training. With the increasing scale of language models, the adoption of pipeline and tensor parallelisms has become essential [135]. Pipeline

parallelism involves the use of blocking send and receive operations to transmit intermediate activations, while tensor parallelism utilizes blocking all-reduce operations to combine tensor partitions. Extensive research has been conducted on coordinating communication operations for dense models [172, 64, 144]. Lina focuses on sparsely-activated MoE with data and expert parallelism, which are orthogonal to existing work.

Estimation of expert popularity. The current estimation approach used by Lina relies on data collection during the training stage and does not achieve high-fidelity estimation result. The reason is that inference datasets do not have exactly same distributions and load balancing loss is disabled during inference stage, which differs from training stage. Besides, our estimation method is relatively naive, intending to prove the feasibility of expert selection estimation. While fine-tuning can assist in improving efficient expert placement decisions, an estimation method with improved accuracy and confidence would further reduce inference time. One potential approach is to leverage machine learning techniques to train a compact yet powerful model that can predict the expert selected by each token in every MoE layer ahead of time, when the requests are received.

4.9 Related Work

Existing MoE systems. Recent literature has proposed MoE-specific optimization techniques. DeepSpeed [124] enables distributed training for MoE models and leverages flexible combinations of parallelism strategies. It also introduces a novel MoE architecture called Pyramid-Residual MoE. PR-MoE applies experts only where they are most effective. Tutel [144] extends DeepSpeed and proposes an adaptive parallelism switching strategy specialized at MoE training tasks. It also includes a hierarchical all-to-all design to cope with the inter- and intra-node

GPU topology for better efficiency. It is complementary with Lina.

FasterMoE [50] proposes a roofline performance model to analyze the end-to-end performance of MoE training systems. Guided by this model, they propose a dynamic shadowing approach that pulls popular expert parameters instead of sending tokens to the experts. They also design a topology-aware expert selection strategy that relieves network congestion by sending tokens to experts with lower latency.

Communication acceleration in distributed training. Our community has proposed several communication schedulers for generic distributed training [48, 112, 10, 20, 24, 149]. The objective is to better overlap the communication and computation operations in the backward pass and prioritize the communication of former layers over latter layers in the model. In Lina, we leverage the domain-specific insight that all-to-all should be prioritized over allreduce in MoE training, which is different from prior work. BytePS [65] proposes to reduce the communication traffic by utilizing the heterogeneous GPU/CPU resources in a training cluster. These acceleration techniques can be integrated into distributed MoE. Lina can also benefit from this idea, since more available bandwidth can be left to all-to-all operations.

Chapter 5

Adaptive Gating in MoE-based Language Models

5.1 Introduction

The field of natural language processing (NLP) has undergone a remarkable revolution driven by the rapid advancements in language models [16, 142, 42, 3]. They exhibit so-called "emergent" capabilities for a wide variety of applications [153]. However, as demands for these applications continue to grow, scalability of these models poses an increasingly challenging hurdle due to constraints in computational resources, memory capacity, interconnect bandwidth, etc. [114].

Sparsely-activated MoE is a promising paradigm to address the scalability issue while maintaining a constant number of computation FLOPs [79, 37]. MoE utilizes an ensemble of experts to collectively tackle the learning task. Each input activates a subset of experts, resulting in a dynamically-changing and sparse computation graph. This method effectively distributes the computation among experts, increases model capacity and improves training efficiency [34, 124]. Very recently, there has been quite some prior work on improving the performance of

Transformers using MoE [124, 177, 19, 40].

Despite MoE's benefit in scalability, it suffers from suboptimal training efficiency. In particular, we focus on the gating mechanism that selects the experts for each token in this work. Existing MoE models adopt a fixed top-2 gating in training while employing top-1 gating during inference for shorter response times. Top-2 gating entails twice the computational cost per token and doubles the data transfer size of all-to-all operations compared to top-1. Yet, it remains unclear whether top-2 gating actually leads to performance gains that could justify the additional overheads. Therefore, a comprehensive analysis of the trade-off between training efficiency and model performance is increasingly crucial. More practically, how to construct an MoE language model that effectively balances training efficiency and performance, is of great interest and imminent value.

Towards this end, we present our first attempt to empirically characterize and improve the efficiency of the gating mechanism in MoE. We observe that across various models and tasks, a large number of tokens display simple linguistic characteristics or a single dominant feature, which allows them to be effectively processed using just the top-1 expert. This observation suggests that the current top-2 gating strategy incurs unnecessary computation costs for a significant number of tokens.

Motivated by this insight, we further introduce adaptive gating in MoE that enables tokens to be processed by a *flexible* number of experts depending on the gating decision. Our approach, in contrast to conventional MoE models, preserves the sparsity of MoE models while enhancing flexibility in token handling. We incorporate a threshold within the gating network to conduct adaptive token routing based on the distribution of expert probabilities. With adaptive gating, the majority of tokens use simple top-1 gating; top-2 gating is selectively applied only when necessary and beneficial, thus significantly reducing the computation

cost. However, the training efficiency cannot achieve the same improvement as the computation cost due to the fact that tokens with top-2 gating always incur a longer training step, thus becoming the bottleneck. Therefore, to enhance training efficiency even further, we leverage the idea of curriculum learning by strategically adjusting the order of training data samples.

We conduct extensive experiments on six NLP tasks with different encoder and decoder models. The results show that our approach can effectively reduce the end-to-end training time by at most 22.5%, while achieving comparable inference quality with top-2 gating MoE models. Moreover, we show that the tokens routed to two experts are coupled with the nature of each NLP task. For sentiment analysis, those are the tokens expressing neutral opinions; translation task pays attention to sentences with complex structure; Question and Answer connects key words in question and context and assign both with top-2 gating; summarization puts more effort in understanding the pronouns and finding tokens expressing central idea; top-2 routing decision changes along with the token to generated in text completion task and conversational tokens in dialogue response task use top-2 experts frequently. Empirically, we find that a small threshold value (i.e. 0.1, 0.2) in adaptive gating can lead to a similar performance as top-2 gating.

Our contributions are as follows:

- We propose adaptive gating in the MoE training scheme, which enables tokens to be processed by a flexible number of experts.
- We leverage curriculum learning to alleviate the training bottleneck caused by varying execution times of tokens.
- We conduct extensive experiments on various NLP tasks and datasets and present a thorough analysis of the gating decision of the tokens to prove the effectiveness and efficiency of adaptive gating.

5.2 Background

5.2.1 Mixture-of-Experts

Mixture-of-Experts (MoE) has been adopted in various deep neural network models [134, 21] and has shown great promise in enhancing the performance of language models. For example, GShard [79] and Switch Transformer [37] effectively scale Transformer-based language models with MoE layers.

In particular, these models typically employ an MoE layer to substitute the feed-forward network (FFN) layer. The MoE layer comprises multiple FFNs, each acting as an expert, along with a gating network. Each expert i is a fully-connected two-layer network utilizing ReLU activation and with its own set of parameters. For a given token x, the output of an expert can be defined as:

$$FFN_i(x) = ReLU(x \cdot W_0^i) \cdot W_1^i, \tag{5.1}$$

where W_0^i and W_1^i are the trainable weights of the two linear layers in expert i.

The gating network takes in the embedding vector of each token x and multiplies them with its trainable matrix W_G. The gate value for a specific token can be determined through:

$$R = softmax(x \cdot W_G). \tag{5.2}$$

This softmax activation R indicates the weight of each expert in processing the token. The gating network then dispatches this token to top-k experts with k highest activations. The final output of the MoE layer is:

$$y = \sum_{i \in E} R_i \cdot FFN_i(x), \tag{5.3}$$

that is, the weighted sum of outputs from selected expert(s) $E \subset \{FFN_1, FFN_2 ... FFN_N\}$. The sparse nature of MoE improves the model scaling in size without increasing

the training cost.

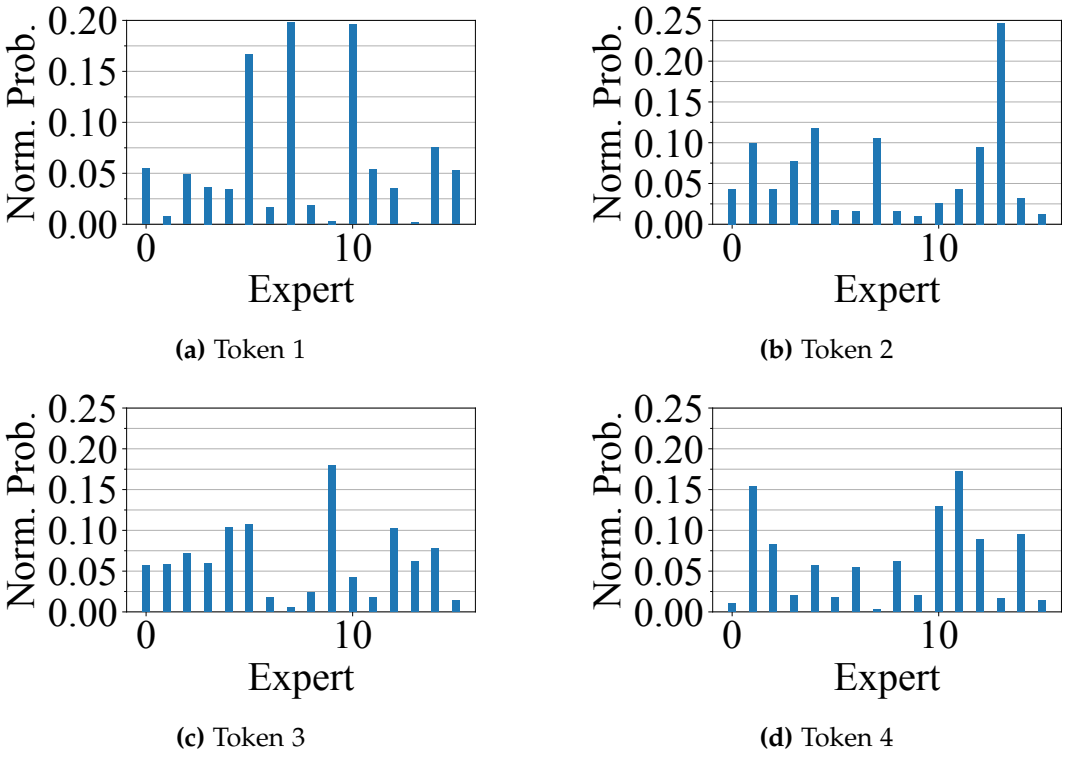
Related work. Several prior works have explored the efficient use of gating or expert selection in MoE. [4, 174, 49, 89] propose different approaches to encourage expert specialization. [28] adopt a pre-defined expert assignment for each input categories. [127, 178] propose to remove gating networks. [174] present a novel selection mechanism where experts selects token instead of token selecting experts. [49] introduce multiple routing policies to enhance specialization in multi-task scenario. [127] use deterministic hashing, while [178] use stochastic routing. However, it could lead to inconsistent inference results. Therefore, they employ a regularized loss to penalize the discrepancy of expert selection. All existing work adopts a fixed and equal computation capacity for each token and expert, while we look into the trade-off between computation costs and model performance with adaptive gating.

5.3 Design

We now discuss the design of adaptive gating in MoE for training.

5.3.1 Adaptive Gating in MoE

Observation. We first present our empirical findings from experiments with classical MoE models. Specifically, we extract the softmax activations and analyze the probability distribution of expert selection for each token in the gating network. Figures 5.1 depict the normalized activation values of four sampled tokens across 16 experts. We see that for tokens 1 and 4, their activations of the top-1 and top-2 expert are very close as shown in Figures 5.1a and 5.1d, while for tokens 2 and 3 a significant bias towards the top-1 expert exists as in Figures 5.1b and 5.1c. We find that these significantly-biased distribution accounts for at least 55% of all the tokens in our evaluation.



(a) Token 1

(b) Token 2

(c) Token 3

(d) Token 4

Figure 5.1: Normalized expert probability computed by top-2 gating network from four sampled tokens. Here we use the Sentiment analysis task list in Table 5.2.

Adaptive gating. Previous work has demonstrated that MoE experts specialize in different linguistic aspects. Building upon our empirical findings, one can see that many tokens can be effectively handled by a single expert during the training stage. To control the number of experts handling each token, we introduce a threshold parameter, denoted as T. If the activation value difference between the top-1 expert, denoted as i, and the top-2 expert, denoted as j, is within the threshold T, we consider the token as requiring both expert i and expert j for processing. Otherwise, we route the token only to the top-1 expert.

Load balancing loss. Adaptive gating uses a flexible number of experts to process each token. This flexibility, however, adds extra difficulty to the load balancing problem in training which aims to evenly distribute tokens among all experts. As it is still important to prevent the gating network from overly concentrating on a very small number of experts, in adaptive gating, we impose

the soft load balancing constraints on the top-1 gating decisions, while allowing top-2 gating decisions to be trained without any soft constraints. That is, the loss of each MoE layer i becomes:

$$L_i = E_i \sum_{e \in E} f_e^1 p_e, \tag{5.4}$$

where f_e^1 is the fraction of tokens dispatched to expert e among those processed by top-1 gating; p_e is the average gating probability to expert e over all tokens in the current batch, and E_i is the number of experts at layer i just as in classical MoE [37].

5.3.2 Batching

Challenge. While adaptive gating provides effective computational savings, Transformer MoE's model architecture poses a significant challenge to training efficiency. Specifically, there is a mismatch in the data processing granularity between the MoE experts and the Attention layer. The MoE experts operate on individual tokens, while the Attention layer requires input in the form of a complete sentence. As a result, although the processing time for a large portion of tokens is reduced by half in the MoE layer, we still need to wait until the remaining tokens (in the same data batch) complete their top-2 processing. Consequently, training step time cannot enjoy the same reduction as in computation. Table 5.1 shows the computation reduction as well as empirical MoE layer running time, both normalized to conventional top-2 gating. We use PyTorch Profiler to obtain the computation time of MoE layer. For simplicity, here we force a fixed percentage of tokens to be routed to only top-1 expert and measure the running time. The reduction in running time is clearly much smaller than the computation savings.

Curriculum learning. In adaptive gating, we propose to incorporate the concept of curriculum learning to address the aforementioned training efficiency

Gate	Norm. Computation	Norm. MoE Layer Running Time
Top-1	0.5	0.67
Adaptive (80% Top-1)	0.6x	0.76x
Adaptive (50% Top-1)	0.75x	0.92x
Adaptive (20% Top-1)	0.9x	0.97x

Table 5.1: We compare the computation savings and running time reduction of the MoE layer of varying degrees of top-1 gating against top-2 gating. The MoE layer running time is measured on our testbed Section 5.4.3. Tokens are randomly selected from the data batch. Here we also use the Sentiment analysis task list in Table 5.2. We show the results averaged from 40 runs.

challenge. Curriculum learning [14], as the name implies, is a paradigm where training examples are presented to a model in increasing order of complexity. It aims to enhance the learning efficiency and generalization performance of models. By carefully designing the curriculum, the model is exposed to easier examples at the initial stages, allowing it to build a solid foundation before tackling more challenging concepts. This gradual learning process has shown promising results in NLP [152].

Adjust training data order. Our intuition is that the number of experts required by each token can be an indicator of the token complexity. We can therefore reorder the training data in a way that prioritizes simpler sequences during model training. Additionally, we can group together training data with similar complexity levels to minimize the bottleneck effect caused by difficult tokens in need of top-2 experts.

To quantify the complexity of a training sample d, we define a complexity vector C:

$$C_d = [r_0^d, r_1^d, ...r_L^d], \tag{5.5}$$

where L is the number of MoE layers in the model, and r_i represents the ratio of tokens processed by top-2 experts to the sequence length (i.e., the total number of tokens in data sample d) in layer i.

To determine the order of the training data, we identify the data sample with the fewest tokens processed by top-2 experts, and calculate the cosine similarity using complexity vectors of the remaining data samples. Training data is then reordered based on this similarity value, starting from the most similar ones. This approach allows the model to gradually learn from simpler sequences and progressively handle more complex sequences.

5.4 Evaluation

We evaluate adaptive gating in MoE on six NLP tasks using various encoder and decoder models. We then analyze the gating decision to better understand the effectiveness of adaptive gating.

5.4.1 Tasks and Models

Table 5.2 summarizes the details.

Task	Dataset	Model	Architecture
Sentiment analysis	SST-2 [139]	BERT-Base [31]	12-layer encoder
Translation	WMT19 (De->En) [39]	FSMT [103]	6-layer encoder, 6-layer decoder
Question and Answer	SQuAD [125]	BERT-Base [31]	12-layer encoder
Summarization	CNN/Daily Mail [52, 128]	BART-Large [81]	12-layer encoder, 12-layer decoder
Text generation	wikitext [93]	GPT-2 [121]	24-layer decoder
Dialogue response	SODA [70]	DialoGPT-medium [168]	24-layer decoder

Table 5.2: Overall performance of adaptive MoE and compared baselines in different NLP tasks. All the models converge to the same loss value.

5.4.2 Baselines

We use the Transformer models from HuggingFace and convert the FFN layers to MoE layers [72]. We compare adaptive gating's training efficiency with the following three baselines and then evaluate the inference performance with top-1 gating MoE.

Dense models. Transformer with no MoE layers.

Top-2 gating MoE. MoE models with top-2 gating [79, 49] for training.

Top-1 gating MoE (Switch Transformer). Switch Transformer [37, 71, 160] uses top-1 gating to mitigate training instabilities.

5.4.3 Training Configurations

We use 8 A100 GPUs, each with 40 GB memory. Data and expert parallel is used for distributed training. We distribute the experts evenly among all the GPUs. In terms of hyperparameters and model architecture, we adopt the default configurations established in the existing models [156, 77].

Model architecture. BERT-Base has 12 attention heads per layer. The hidden size is 768 and the intermediate dimension is 3072. The Transformer model has 16 attention heads. The hidden size is 1024 and the intermediate dimension in encoder and decoder layers are 8192 and 4096, respectively. BART-Large has 16 attention heads. The hidden size is 1024 and the intermediate dimension is 4096. GPT-2 and DialoGPT-medium have 16 attention heads. The hidden size is 1024 and the intermediate dimension is 4096.

Hyperparameters. BERT-Base has a batch size of 24 and the learning rate is 0.00003. The maximum number of tokens for the translation model is 4096 with a learning rate of 0.0005. The maximum number of tokens allowed for BART-Large is set to 4096. The learning rate is 0.00001. The batch size of GPT-2 is 8 with a learning rate of 0.00015. For DialoGPT-medium, the batch size and learning rate are 64 and 0.0001.

MoE configurations. The parameter size of the FFN in each model is the same in Baseline and MoE models and we set the number of FFNs (i.e. experts) to 16 for all evaluated tasks. The coefficient of the load balancing loss is 0.01. No capacity constraints are enabled so no tokens would be dropped. The expert parameters are randomly initialized. We normalize the expert probability in adaptive gating

and set the threshold T to 0.1.

5.4.4 Overall Performance

We present the overall training and inference performance in Table 5.3.

Overall, adaptive gating achieves comparable performance to the baselines while significantly reducing the training time even compared to top-1 gating. This is because though top-1 gating maximizes the computation saving, it makes training more difficult to converge to the same loss value, eventually leading to slightly longer training time compared to top-2 gating in 4 out of 6 tasks we run. An in-depth analysis of how adaptive gating works in connection to each task is presented in Section 5.4.5.

Task	Scheme	Norm. Training Time	Computation FLOPs	Inference Performance
	Dense	0.88x	2.18G	0.912
Sentiment analysis	Top-2 Gating	1x	3.28G	0.918
	Top-1 Gating	0.99x	2.18G	0.902
(Accuracy)	Adaptive Gating	0.77x	2.30G	**0.919**
	Dense	0.87x	10.6G	40.9
En->De translation	Top-2 Gating	1x	15.9G	**41.1**
	Top-1 Gating	1.04x	10.6G	39.5
(BLEU Score)	Adaptive Gating	0.79x	11.5G	**41.1**
	Dense	0.84x	2.18G	75.7
Question and Answer	Top-2 Gating	1x	3.27G	**77.6**
	Top-1 Gating	1.07x	2.18G	75.5
(F1 Score)	Adaptive Gating	0.86x	2.36G	77.4
	Dense	0.89x	79G	42.3
Summarization	Top-2 Gating	1x	119G	**43.4**
	Top-1 Gating	1.06x	79G	40.8
(ROUGE-1)	Adaptive Gating	0.86x	87G	43.3
	Dense	0.84x	3.4T	16.3
Text completion	Top-2 Gating	1x	4.9T	**17.8**
	Top-1 Gating	1.14x	3.4T	16.5
(Perplexity)	Adaptive Gating	0.89x	3.73T	17.5
	Dense	0.82x	3.4T	12.5
Dialogue response	Top-2 Gating	1x	4.9T	**13.4**
	Top-1 Gating	0.93x	3.4T	12.6
(Perplexity)	Adaptive Gating	0.82x	3.76T	13.3

Table 5.3: Overall performance of adaptive gating and compared baselines in different NLP tasks. We normalize the training time with reference to the performance of top-2 gating MoE. All the schemes in the same task converge to the same loss.

Sentiment analysis. Adaptive gating in MoE outperforms both Dense models and top-2 gating MoE in all metrics. While the average computation FLOPs per

token is higher with adaptive gating compared to top-1 gating MoE, which represents the minimum possible FLOPs in the MoE structure, adaptive gating requires less training time and achieves superior accuracy during the inference stage. This is consistent across all the tasks. Notably, only 11.3% of the tokens in our evaluation receive two experts, which is the lowest among all tasks. Compared to top-2 gating, adaptive gating focuses on assigning more experts to tokens that represent neutral opinions, allowing for a more comprehensive decision-making process. Conversely, tokens expressing little or obvious sentiment are given less attention without degrading accuracy.

Translation. Adaptive gating delivers the same performance with top-2 gating while reducing training time and FLOPs per token by 25.6% and 38.2%, respectively. Notably, we observe that the gating network in adaptive gating exhibits a particular focus on the complexity of sentence structures. Even tokens that appear linguistically simple can involve two experts when they appear in sentences with intricate structures and grammar. Overall, 25.6% of all trained tokens are routed to two experts.

Question and Answer. The training time with adaptive gating is 85.7% that of top-2 gating. Although its inference performance is slightly lower, it still outperforms top-1 gating. Through our experiments (refer to Section 5.4.6), we discover that adaptive gating achieves the best results when the threshold is set to 0.2 for Question and Answer. The gating decision is influenced by both the context and the specific question being asked. For this task 16.4% tokens receive top-2 processing.

Summarization. Summarization is the most challenging task in our evaluation, as it involves processing long and information-rich articles. Adaptive gating takes 11.8% less training time than top-2 gating. However, its inference performance slightly lags behind. Particularly, in adaptive gating tokens selected for top-2 experts exhibit significant variations across different layers. We provide a more

detailed analysis of this observation in Section 5.4.5.

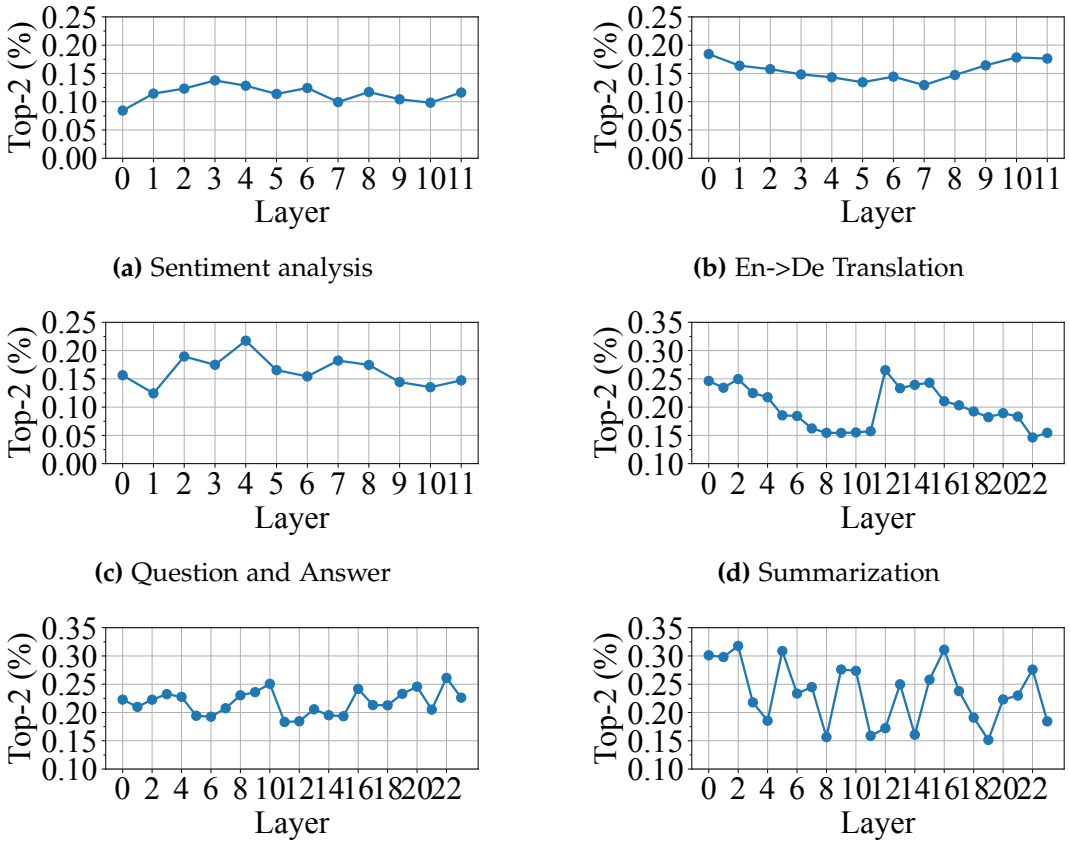
Text completion. We use a GPT-like decoder-only architecture for this task. Adaptive gating achieves similar performance as top-2 gating and Dense models while outperforming top-1 gating. When compared to top-2 gating, only 21.8% tokens rely on two experts, resulting in a reduction of 23.8% in average computation FLOPs per token. The selection of tokens utilizing two experts varies considerably due to the diverse nature of the input.

Dialogue response. Dialogue response requires more nuanced processing compared to simple text generation, as it involves generating responses in a targeted role based on narrative input and dialogue history. The sparsity introduced by MoE is advantageous for this task. All three MoE approaches outperform the Dense model. Among all the tasks evaluated, dialogue response exhibits the highest percentage, 23.4% of tokens routed to two experts, indicating the higher utilization of the top-2 gating mechanism among all the tasks. Upon evaluating the tokens, we observe that this task can be viewed as a combination of all the other evaluated tasks.

5.4.5 Analysis and Insights

While it is intuitive to understand that some minor tokens (e.g., “a”, “the”, “is”) only need top-1 expert to process, this does not fully explain how and why adaptive gating works in different NLP tasks. Thus we analyze how the tokens are processed in training with adaptive gating, and make quite a few interesting observations that can help better answer this question. In a broader sense, we believe our insights are also instrumental towards building better language models.

Note that when BPE tokenizer is used, we aggregate the result by mapping the tokens to the natural language word and perform analysis on the aggregated statistics.



(a) Sentiment analysis

(b) En->De Translation

(c) Question and Answer

(d) Summarization

(e) Text generation

(f) Dialogue response

Figure 5.2: Percentage of tokens computed by top-2 experts over all the tokens in each layer when using adaptive gating in MoE.

Sentiment analysis. Sentiment analysis exhibits the lowest percentage of top-2 gating among all tasks, and the percentage is stable across layers (Figure 5.2a). The top-2 gating mechanism focuses on two main types of input here. First, it frequently selects tokens that express a more neutral opinion since they are more difficult to classify (Table 5.4). Second, tokens associated with sarcastic statements, double negatives, or conflicting opinions are also commonly routed to two experts. Adaptive gating effectively identifies these tokens early on in the model as they are relatively easy to extract, which explains the stable percentage across layers. A special case is when the input does not explicitly convey any sentiment. Adaptive gating tends to initially route all tokens to either the top-1 or top-2 experts and gradually narrows down to more informative tokens. A

typical instance of this is "as a dentist's waiting room."

Translation. We focus on English-to-German translation only. We examine the top-2 gating results based on our understanding of the source sentences. The distribution of the top-2 gating percentages varies between the encoder and decoder layers, exhibiting a gradual decrease in the encoder layers and an increase in the decoder layers (Figure 5.2b). From sampled tokens and the adjusted data order in adaptive gating, we observe that tokens requiring two experts are usually within the same sentence. This observation leads us to infer that the complexity of sentence structure influences the gating results. In Table 5.4, we present one sentence containing multiple clauses that are frequently processed by the top-2 experts.

Question and Answer. The percentage of top-2 tokens in question and answer tasks fluctuates across layers (Figure 5.2c). First, adaptive gating pays extra attention to the question itself. Words listed in Table 5.4 are some common examples. These tokens often either specify the scope of the question or pose constraints to the answers. Second, in the context side, tokens routed to two experts are closely related to the question in the input as well. For example, asking a question about numbers and computations would result in top-2 gating on the numbers and the objects those numbers refer to.

Summarization. In summarization, the percentage of tokens using two experts decreases in both encoder and decoder layers (Figure 5.2d). Based on our analysis of sampled tokens, we identify two patterns for tokens that are likely to be routed to top-2 experts. First, tokens with multiple meanings that rely on both themselves and the surrounding context for their ultimate interpretation. They are often routed to two experts in the shallow layers. Second, pronoun tokens, as understanding their referents is crucial for accurate summarization, use two experts in the deeper layers. This pattern is particularly prevalent in this task. Additionally, certain key tokens (e.g. "in conclusion", "however", "in all")

that indicate the beginning the central idea or the main opinion of the context are often sent to two experts together with the following tokens.

Text completion. Text completion differs from the previous tasks as it is a decoder-only and auto-regressive task. The gating results in text completion are influenced by the current prediction being generated. The focus of tokens changes dynamically based on the current prediction. It is challenging to identify specific types of tokens that consistently receive two experts. When predicting a pronoun, for example, the focus shifts to the names of individuals. Similar patterns can be observed for numbers and dates. Additionally, we find that the percentage of tokens routed to two experts is linked to the length of the current sequence. Longer sequences have a higher percentage of top-2 gating.

Dialogue response. Dialogue response, compared to text completion, requires more understanding of the narrative input and the dialogue history. We find that lots of effort are put into processing dialogue history. First, one key distinction is that tokens with a conversational meaning occur much more frequently. These words lack informative content but serve to express human-like sentiments, such as gratitude and politeness. We infer that routing these tokens for two experts indicates that there is a difference between the conversational usage and written text and it is also critical to learn where and when these words should be used. Second, given the nature of the dialogue, many conversations are based on underlying assumptions and conditions. Related tokens are usually processed with two tokens to improve the understanding of the context. For instance, the dialogue example provided in Table 5.4 is built on top of a scenario assuming that "Johnathan tells his parents that he is gay" and asks the model to answer questions with this condition.

Task	Top-2 gating tokens
Sentiment analysis	realistic, thoroughly, handsome but unfulfilling, simply, is not the worst movie of the year, generic
Translation	I believe that anyone who has had the opportunity to visit Algeria during recent months or years can make a better assessment of what this terrible outbreak of terrorism means to the Algerian people and, indeed, I believe that it would be to our credit if we dealt with this issue in an urgent debate.
Question and Answer	Which entity, who else, after what, Up until, who was blamed, in terms of, after, Who's death caused this protest?
Summarization	Japanese actress Rinko Kikuchi walks Anjali Rao through the streets of Tokyo. She stunned global cinema audiences with her controversial and Oscar-nominated performance as a lonely deaf girl in the film "Babel". Rinko Kikuchi is one of Japan's hottest young actresses and models, recently working with Karl Lagerfeld as the new face of Channel. Despite her success, she remains an unconventional figure in Japan, at odds with the traditional demure image of the Japanese woman and forging a career on her own terms...
Text completion	Harris announced he would be stepping down as rabbi in 2011, and the synagogue hired Boris Dolin as his successor. Born and raised in Oregon, Dolin had worked at Temple Beth Israel as a teacher and youth group adviser from 1999 to 2001.
Dialogue response	exactly, definitely, hmm, um, well, I guess, [Narrative] Johnathan plans to tell his parents that he is gay. He feels anxious because he doesn't know they will react. He is worried that they will be disappointed or even angry with him.

Table 5.4: Examples of tokens using top-2 experts in different tasks. Underlined tokens use top-2 gating in a sequence.

5.4.6 Ablation Study

Threshold T in adaptive gating. We now conduct an ablation study on the threshold T introduced in adaptive gating. Increasing the threshold value results in a less sparse model, where more tokens are assigned to the top-2 gating mechanism, subsequently increasing the computational FLOPs. Table 5.5 shows the inference performance of different tasks when the threshold is increased from 0.05 to 0.5. When using a small threshold of 0.05, both the training time and inference performance closely resemble those of top-1 gating MoE. On the other

hand, setting the threshold to 0.4 does not always lead to the same performance as top-2 gating. Together with Table 5.3, we discover that threshold values of 0.1 and 0.2 often strike a favorable balance between training time and inference performance.

Task	Norm. Training Time				Inference Performance			
	0.05	0.2	0.3	0.4	0.05	0.2	0.3	0.4
Sentiment analysis	1.02x	0.77x	0.92x	1.01x	0.912	**0.918**	0.917	0.918
Translation	0.88x	0.83x	0.83x	0.88x	40.2	**41.1**	40.8	41.1
Question and Answer	0.92x	0.87x	0.93x	0.96x	74.3	**77.6**	77.6	77.6
Summarization	0.98x	1.02x	1.05x	1.04x	40.8	42.3	**43.1**	43.1
Text generation	0.95x	0.93x	0.99x	1.01x	16.6	17.2	**17.4**	17.4
Dialogue response	0.93x	0.91x	1.01x	1.01x	12.2	12.8	13.2	**13.4**

Table 5.5: Overall performance when the threshold T changes. Training time is normalized with reference to top-2 gating MoE. We highlight the best one with the least training time.

Curriculum learning. Essentially, we disable the data order adjustment before each epoch and use the random data loader to feed the training set. We present the performance degradation compared to the full-set adaptive gating in Table 5.6. Since it is highly possible that there is at least one token that are routed to top-2 experts, the step time of each iteration cannot achieve the same level of reduction as the computation FLOPs. Consequently, the end-to-end training time is significantly inflated, with an average increase of 13.7%. Additionally, the idea of the curriculum also contributes to the improvement in inference performance. The maximum drop is 0.21 in Question and Answer task when the data is fed and trained in a random manner.

5.5 Limitation

Choice of k. Adaptive gating in MoE currently is limited to top-k gating, where k can be either 1 or 2. This is built on the common practice in extensive prior

Task	Training Time Inflation	Inference Performance
Sentiment analysis	22%	+0.00
Translation	14%	-0.14
Question and Answer	9%	-0.21
Summarization	14%	-0.14
Text completion	12%	-0.01
Dialogue response	11%	-0.19

Table 5.6: Overall performance comparison of adaptive gating when data batch is not adjusted.

work that top-2 gating shows a promissing resut in MoE. Further evaluation is necessary to validate the performance of a wider range of k values. Our experiments were conducted on a diverse set of NLP tasks and datasets, but it is essential to note that the effectiveness and efficiency of adaptive MoE may vary depending on the specific task characteristics. Different tasks may exhibit distinct patterns and complexities, which can impact the performance and generalizability of the proposed approach. Further investigation and evaluation on a wider range of tasks would provide a more comprehensive understanding of the limitations and applicability of adaptive MoE.

Conclusion

5.6 Conclusion

In this thesis, we have comprehensively reviewed the existing work on systems for distributed DNN workloads and we have presented three research projects that address the challenges in distributed DNN training and inference.

We have presented Lyra, an elastic GPU cluster scheduler for deep learning. The key idea is to exploit cluster-level elasticity by loaning idle inference servers for training, and job-level elasticity by scaling jobs to better utilize the dynamic resource pool. In designing and evaluating Lyra, we have addressed new challenges in cluster management, by introducing heuristics to reduce job preemption cost due to loan-reclaiming, and to minimize job completion time when elastic jobs are presented.

We then introduced Lina, a new system that accelerates all-to-all in distributed MoE. Through a systematic analysis, we build Lina upon two key ideas: first to prioritize all-to-all over allreduce using tensor partitioning and pipelining to improve its bandwidth in training, and second to dynamically balance the workload with token-level expert selection pattern in inference. We implemented Lina over DeepSpeed and performed extensive testbed evaluation using A100 GPUs and 100Gbps InfiniBand to show that Lina significantly improves training efficiency and inference time.

We demonstrate the effectiveness and flexibility of adaptive gating in MoE

models for a wide range of natural language processing tasks. By dynamically adjusting the number of experts based on token characteristics, we achieve improved training efficiency without compromising inference performance. Additionally, the integration of curriculum learning allows us to tackle the challenge of varying execution times, thereby reducing training costs. Our research sheds light on the trade-off between training efficiency and model performance in sparse and dynamic MoE networks, offering valuable insights for the development of more scalable and adaptable language models.

5.7 Future Work

Since LLM has become successful over the few years, the future work of this thesis would follow such trend. We plan to build efficient infrastructure tailored for LLM models.

Driven by the insight that today's Large Language Models are typically fine-tuned using diverse techniques, many of which employ parameter-efficient fine-tuning (PEFT), we are met with an intriguing opportunity. PEFT is a practice wherein a smaller, task-specific model is on top of a larger, pre-trained model, thereby reaping the benefits of the foundational model's extensive learning [122]. For instance, techniques like adapter layers [54] and LoRA [55] allow a model to preserve the parameters of the pre-trained model while learning task-specific patterns in the newly-added parameters. PEFT results in a multitude of fine-tuned LLMs sharing parameters from the same foundational models, presenting a unique set of benefits that can be leveraged to address the challenges previously outlined. Our goal is to design and build a scalable inference system for LLMs, which capitalizes on the advantages of PEFT. This system aims to address three critical aspects: efficiency, performance, and maintainability:

- The system aims to optimize efficiency in terms of both storage and com-

putational resources. By leveraging PEFT, we can exploit the opportunities to reduce storage size and improve compute efficiency through model sharing, as multiple fine-tuned models will share parameters from the same foundation models.

- The system aims to deliver high performance. PEFT allows models to be naturally segregated into components, enabling each component to scale individually. This modular nature allows the inference system to auto-scale model components at a finer granularity, thereby adapting to traffic changes quickly over time.

- The system aims to enhance maintainability by facilitating the efficient evolution of models over time. Foundation models typically have long development cycles and are costly to redeploy. In contrast, PEFT allows fine-tuned models to redeploy only a small amount of parameters during continual learning. This feature enables models to quickly adapt to changing data distributions over time, making the system more resilient and adaptable.

References

[1] *2023 State of Data + AI*. https://www.databricks.com/sites/default/files/2023-06/databricks-2023-state-of-data-report-06072023-v2_0.pdf.

[2] Martin 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: *Proc. USENIX OSDI*. 2016.

[3] *AI ACROSS GOOGLE: PaLM 2*. https://ai.google/discover/palm2/.

[4] Raquel Aoki, Frederick Tung, and Gabriel L Oliveira. "Heterogeneous multi-task learning with expert diversity". In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 19.6 (2022), pp. 3093–3102.

[5] EO Arkhangelskaya and Sergei Igorevich Nikolenko. "Deep learning for natural language processing: a survey". In: *Journal of Mathematical Sciences* (2023), pp. 1–50.

[6] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. "Efficient Large Scale Language Modeling with Mixtures of Experts". In: *arXiv preprint arXiv:2112.10684* (2021).

[7] Ammar Ahmad Awan, Khaled Hamidouche, Akshay Venkatesh, and Dhabaleswar K Panda. "Efficient large message broadcast using NCCL and CUDA-aware MPI for deep learning". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. 2016.

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. "PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications". In: *Proc. USENIX OSDI*. 2020.

[9] Mandeep Baines et al. *FairScale: A general purpose modular PyTorch library for high performance and large scale training*. https://github.com/facebookresearch/fairscale.

[10] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. "Preemptive all-reduce scheduling for expediting distributed DNN training". In: *IEEE INFOCOM*. 2020.

[11] Paul Barham et al. "Pathways: Asynchronous Distributed Dataflow for ML". In: *Proc. MLSys*. 2022.

[12] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. "Conditional computation in neural networks for faster models". In: *arXiv preprint arXiv:1511.06297* (2015).

[13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or propagating gradients through stochastic neurons for conditional computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[14] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. "Curriculum learning". In: *Proc. ICML*. 2009.

[15] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners". In: *Proc. NeurIPS* 33 (2020), pp. 1877–1901.

[16] *ChatGPT: Optimizing Language Models for Dialogue*. https://openai.com/blog/chatgpt/.

[17] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. "Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning". In: *Proc. ACM EuroSys*. 2020.

[18] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. "Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments". In: *Proc. ACM SoCC*. 2020.

[19] Tianlong Chen, Zhenyu Zhang, AJAY KUMAR JAISWAL, Shiwei Liu, and Zhangyang Wang. "Sparse MoE as the New Dropout: Scaling Dense and Self-Slimmable Transformers". In: *Proc. ICLR*. 2023.

[20] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. "Elastic parameter server load distribution in deep learning clusters". In: *Proc. ACM SoCC*. 2020.

[21] Zitian Chen, Yikang Shen, Mingyu Ding, Zhenfang Chen, Hengshuang Zhao, Erik G Learned-Miller, and Chuang Gan. "Mod-Squad: Designing Mixtures of Experts As Modular Multi-Task Learners". In: *Proc. IEEE/CVF CVPR*. 2023, pp. 11828–11837.

[22] Zewen Chi, Li Dong, Shaohan Huang, Damai Dai, Shuming Ma, Barun Patra, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, et al. "On the representation collapse of sparse mixture of experts". In: *Proc. NeurIPS* (2022).

[23] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. "Project adam: Building an efficient and scalable deep learning training system". In: *Proc. USENIX OSDI*. 2014.

[24] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. "BlueConnect: Decomposing All-Reduce for Deep Learning on Heterogeneous Network Hierarchy". In: *Proc. MLSys*. 2019.

[25] Edward G. Coffman Jr., János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. "Bin Packing Approximation Algorithms: Survey and Classification". In: *Handbook of Combinatorial Optimization*. Springer, 2013.

[26] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. "Clipper: A low-latency online prediction serving system". In: *Proc. USENIX NSDI*. 2017.

[27] *CUDA Toolkit*. https://developer.nvidia.com/cuda-toolkit.

[28] Yong Dai, Duyu Tang, Liangxin Liu, Minghuan Tan, Cong Zhou, Jingquan Wang, Zhangyin Feng, Fan Zhang, Xueyu Hu, and Shuming Shi. "One model, multiple modalities: A sparsely activated approach for text, sound, image, video and code". In: *arXiv preprint arXiv:2205.06126* (2022).

[29] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. "Transformer-XL: Attentive language models beyond a fixed-length context". In: *arXiv preprint arXiv:1901.02860* (2019).

[30] *DeepSpeed*. https://github.com/microsoft/DeepSpeed.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* (2018).

[32] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: (2018). arXiv: 1810.04805.

[33] Micah Dowty and Jeremy Sugerman. “GPU virtualization on VMware’s hosted I/O architecture”. In: *ACM SIGOPS Operating Systems Review* 43.3 (2009), pp. 73–82.

[34] Nan Du et al. “GLaM: Efficient Scaling of Language Models with Mixture-of-Experts”. In: *PMLR*. 2022.

[35] Samuel Eilon and IG Chowdhury. “Minimising waiting time variance in the single machine problem”. In: *Management Science* (1977).

[36] *Enwik8*. http://prize.hutter1.net/.

[37] William Fedus, Barret Zoph, and Noam Shazeer. “Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity”. In: *arXiv preprint arXiv:2101.03961* (2021).

[38] Dror G Feitelson and Larry Rudolph. “Metrics and benchmarking for parallel job scheduling”. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. 1998.

[39] Wikimedia Foundation. *ACL 2019 Fourth Conference on Machine Translation (WMT19), Shared Task: Machine Translation of News*. URL: http://www.statmt.org/wmt19/translation-task.html.

[40] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. “MegaBlocks: Efficient Sparse Training with Mixture-of-Experts”. In: *arXiv preprint arXiv:2211.15841* (2022).

[41] Andrea Gesmundo and Jeff Dean. “munet: Evolving pretrained deep neural networks into scalable auto-tuning multitask systems”. In: *arXiv preprint arXiv:2205.10937* (2022).

[42] *Google Bard*. https://bard.google.com/.

[43] *Google Falcon*. https://cloud.google.com/blog/topics/systems/introducing-falcon-a-reliable-low-latency-hardware-transport.

[44] 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". In: (2017). arXiv: 1706.02677.

[45] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters". In: *Proc. USENIX NSDI*. 2016.

[46] 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: *Proc. USENIX NSDI*. 2019.

[47] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *Proc. USENIX OSDI*. 2020.

[48] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. "TicTac: Accelerating Distributed Deep Learning with Communication Scheduling". In: *Proc. MLSys*. Ed. by A. Talwalkar, V. Smith, and M. Zaharia. 2019.

[49] Hussein Hazimeh, Zhe Zhao, Aakanksha Chowdhery, Maheswaran Sathiamoorthy, Yihua Chen, Rahul Mazumder, Lichan Hong, and Ed Chi. "Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning". In: *Proc. NeurIPS* (2021).

[50] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. "FasterMoE: modeling and optimizing training of large-scale dynamic pre-trained models". In: *Proc. ACM SIGPLAN PPoPP*. 2022, pp. 120–134.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proc. IEEE/CVF CVPR*. 2016.

[52] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. "Teaching Machines to Read and Comprehend". In: *Proc. ACM NeurIPS*. 2015.

[53] Horovod. *Elastic Horovod*. https://horovod.readthedocs.io/en/latest/elastic_include.html. 2021.

[54] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. "Parameter-efficient transfer learning for NLP". In: *Proc. ICML*. 2019.

[55] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "Lora: Low-rank adaptation of large language models". In: *arXiv preprint arXiv:2106.09685* (2021).

[56] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. "dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training". In: *Proc. MLSys*. 2022.

[57] 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: *Proc. NeurIPS* (2019).

[58] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. "Tutel: Adaptive mixture-of-experts at scale". In: *Proc. MLSys* (2023).

[59] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. "Elastic Resource Sharing for Distributed Deep Learning". In: *Proc. USENIX NSDI*. 2021.

[60] Folasade Olubusola Isinkaye, Yetunde O Folajimi, and Bolande Adefowoke Ojokoh. "Recommendation systems: Principles, methods and evaluation". In: *Egyptian informatics journal* 16.3 (2015), pp. 261–273.

[61] 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 ATC*. 2019.

[62] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Yong Li, Zhen Zheng, Xiaoyong Liu, Wei Lin, et al. "Whale: Scaling deep learning model training to the trillions". In: *arXiv preprint arXiv:2011.09208* (2020).

[63] 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". In: (2018). arXiv: 1807.11205.

[64] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks." In: *Proc. MLSys* (2019).

[65] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *Proc. USENIX OSDI*. 2020, pp. 463–479.

[66] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *Proc. USENIX OSDI*. 2020.

[67] Tyler B Johnson, Pulkit Agrawal, Haijie Gu, and Carlos Guestrin. "AdaScale SGD: A Scale-Invariant Algorithm for Distributed Training". In: (2019).

[68] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[69] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. "Scaling laws for neural language models". In: *arXiv preprint arXiv:2001.08361* (2020).

[70] Hyunwoo Kim et al. "SODA: Million-scale Dialogue Distillation with Social Commonsense Contextualization". In: *ArXiv* abs/2212.10465 (2022).

[71] Young Jin Kim, Ammar Ahmad Awan, Alexandre Muzio, Andres Felipe Cruz Salinas, Liyang Lu, Amr Hendy, Samyam Rajbhandari, Yuxiong He, and Hany Hassan Awadalla. "Scalable and efficient moe training for multitask multilingual models". In: *arXiv preprint arXiv:2109.10465* (2021).

[72] Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. "Sparse Upcycling: Training Mixture-of-Experts from Dense Checkpoints". In: *arXiv preprint arXiv:2212.05055* (2022).

[73] Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. "Sparse Upcycling: Training Mixture-of-Experts from Dense Checkpoints". In: *Proc. ICLR*. 2023.

[74] Kubernetes. *ElasticDL: A Kubernetes-native Deep Learning Framework*. https://github.com/sql-machine-learning/elasticdl. 2021.

[75] Kubernetes. *Kubernetes*. https://kubernetes.io/. 2021.

[76] Kubernetes. *Kubernetes Horizontal Pod Autoscaler*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. 2021.

[77] Yoohwan Kwon and Soo-Whan Chung. "MoLE: Mixture Of Language Experts For Multi-Lingual Automatic Speech Recognition". In: *ICASSP*. 2023.

[78] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), p. 436.

[79] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. "Gshard: Scaling giant models with conditional computation and automatic sharding". In: *arXiv preprint arXiv:2006.16668* (2020).

[80] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. "BASE Layers: Simplifying Training of Large, Sparse Models". In: *Proc. USENIX ICML*. 2021.

[81] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension". In: *arXiv preprint arXiv:1910.13461* (2019).

[82] 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: *Proc. USENIX OSDI*. 2014.

[83] Zhuohan Li et al. “AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving”. In: *Proc. USENIX OSDI*. 2023.

[84] Hanxue Liang, Zhiwen Fan, Rishov Sarkar, Ziyu Jiang, Tianlong Chen, Kai Zou, Yu Cheng, Cong Hao, and Zhangyang Wang. “M^{3}ViT: Mixture-of-Experts Vision Transformer for Efficient Multi-task Learning with Model-Accelerator Co-design”. In: *Proc. NeurIPS*. 2022.

[85] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. “Zico: Efficient GPU Memory Sharing for Concurrent DNN Training”. In: *USENIX ATC*. 2021.

[86] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. “Dynamic right-sizing for power-proportional data centers”. In: *Proc. IEEE INFOCOM*. 2011.

[87] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. “Heracles: Improving resource efficiency at scale”. In: *Proc. ACM ISCA*. 2015.

[88] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. “Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud”. In: *Proc. MLSys* (2020).

[89] Jiaqi Ma, Zhe Zhao, Xinyang Yi, Jilin Chen, Lichan Hong, and Ed H Chi. “Modeling task relationships in multi-task learning with multi-gate mixture-of-experts”. In: *Proc. ACM SIGKDD*. 2018.

[90] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. “Learning Word Vectors for Sentiment Analysis”. In: *Proc. ACL*. 2011.

[91] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. “Better Together: Jointly Optimizing {ML} Collective Scheduling

and Execution Planning using {SYNDICATE}". In: *Proc. USENIX NSDI*. 2023.

[92] 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". In: (2019). arXiv: 1910.01500.

[93] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. *Pointer Sentinel Mixture Models*. 2016. arXiv: 1609.07843.

[94] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. "Massively distributed SGD: ImageNet/ResNet-50 training in a flash". In: (2018). arXiv: 1811.05233.

[95] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. "CheckFreq: Frequent, Fine-Grained DNN Checkpointing". In: *Proc. USENIX FAST*. 2021.

[96] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. "Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters". In: *Proc. USENIX OSDI*. 2022.

[97] Davoud Mougouei, David MW Powers, and Asghar Moeini. "An integer linear programming model for binary knapsack problem with dependent item values". In: *Australasian Joint Conference on Artificial Intelligence*. 2017.

[98] Ibrahim Naji. "TSATC: Twitter Sentiment Analysis Training Corpus". In: *thinknook*. 2012.

[99] 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: *Proc. ACM SOSP*. 2019.

[100] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads". In: *Proc. USENIX OSDI*. 2020.

[101] 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*. 2018.

[102] *NCCL*. https://github.com/NVIDIA/nccl.

[103] Nathan Ng, Kyra Yee, Alexei Baevski, Myle Ott, Michael Auli, and Sergey Edunov. "Facebook FAIR's WMT19 News Translation Task Submission". In: *Proc. of WMT*. 2020.

[104] *NVIDIA cuDNN*. https://developer.nvidia.com/cudnn.

[105] *NVIDIA DGX H100*. https://www.nvidia.com/en-us/data-center/dgx-h100/.

[106] Andrew Or, Haoyu Zhang, and Michael Freedman. "Resource elasticity in distributed deep learning". In: *Proc. MLSys*. 2020.

[107] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. "The Case for Tiny Tasks in Compute Clusters". In: *Proc. USENIX HotOS*. 2013.

[108] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. *Heuristics for vector bin packing*. Microsoft Research Technical Report. Microsoft Research, 2011.

[109] 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: *Proc. USENIX ATC*. 2020.

[110] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications". In: (2018). arXiv: 1811.09886.

[111] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. "Optimus: an efficient dynamic resource scheduler for deep learning clusters". In: *Proc. ACM EuroSys*. 2018.

[112] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. "A Generic Communication Scheduler for Distributed DNN Training Acceleration". In: *Proc. ACM SOSP*. 2019.

[113] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. "A generic communication scheduler for distributed DNN training acceleration". In: *Proc. ACM SOSP*. 2019.

[114] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. "Efficiently scaling transformer inference". In: *Proc. MLSys* (2023).

[115] Andrey Proskurin. *DeepSpeed: Advancing MoE inference and training to power next-generation AI scale*. https://www.microsoft.com/en-us/research/blog/deepspeed-advancing-moe-inference-and-training-to-power-next-generation-ai-scale.

[116] *PyTorch*. https://pytorch.org.

[117] PyTorch. *PyTorch Elastic*. https://pytorch.org/elastic/0.2.0rc1/distributed.html#module-torchelastic.distributed.launch. 2021.

[118] *PyTorch Distributed Data Parallel*. https://pytorch.org/docs/stable/notes/ddp.html.

[119] *PyTorch Profiler*. https://pytorch.org/blog/pytorch-profiler-1.9-released/.

[120] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning". In: *Proc. USENIX OSDI*. 2021.

[121] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language Models are Unsupervised Multitask Learners". In: (2019).

[122] 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". In: *The Journal of Machine Learning Research* (2020).

[123] 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". In: *Journal of Machine Learning Research* (2020).

[124] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. "DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale". In: *Proc. ICML*. 2022.

[125] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: (2016). arXiv: 1606.05250.

[126] 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". In: *Proc. USENIX ATC*. 2021.

[127] Stephen Roller, Sainbayar Sukhbaatar, Jason Weston, et al. "Hash layers for large sparse models". In: *Proc. NeurIPS* (2021).

[128] Abigail See, Peter J. Liu, and Christopher D. Manning. "Get To The Point: Summarization with Pointer-Generator Networks". In: *Proc. ACL*. 2017.

[129] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. "{TACCL}: Guiding Collective Algorithm Synthesis using Communication Sketches". In: *Proc. USENIX NSDI*. 2023.

[130] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer". In: *arXiv preprint arXiv:1701.06538* (2017).

[131] Noam Shazeer et al. "Mesh-TensorFlow: Deep Learning for Supercomputers". In: *Proc. ACM NeurIPS*. 2018.

[132] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis". In: *Proc. ACM SOSP*. 2019.

[133] Liang Shen, Zhihua Wu, WeiBao Gong, Hongxiang Hao, Yangfan Bai, HuaChao Wu, Xinxuan Wu, Haoyi Xiong, Dianhai Yu, and Yanjun Ma. "SE-MoE: A Scalable and Efficient Mixture-of-Experts Distributed Training and Inference System". In: *arXiv preprint arXiv:2205.10034* (2022).

[134] Sheng Shen, Zhewei Yao, Chunyuan Li, Trevor Darrell, Kurt Keutzer, and Yuxiong He. "Scaling Vision-Language Models with Sparse Mixture of Experts". In: *arXiv preprint arXiv:2303.07226* (2023).

[135] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. "Megatron-lm: Training multi-billion parameter language models using model parallelism". In: (2019). arXiv: 1909.08053.

[136] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. "Singularity: Planet-Scale, Preemptible, Elastic Scheduling of AI Workloads". In: (2022). arXiv: 1403.1349.

[137] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: (2014). arXiv: 1409.1556.

[138] Prabhakant Sinha and Andris A Zoltners. *The multiple-choice knapsack problem*. Tech. rep. Operations, 1979.

[139] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. "Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank". In: *Proc. EMNLP*. 2013.

[140] Chunqiang Tang et al. "Twine: A Unified Cluster Management System for Shared Infrastructure". In: *Proc. USENIX OSDI*. 2020.

[141] *TensorRT*. https://github.com/NVIDIA/TensorRT.

[142] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. "Llama: Open and efficient foundation language models". In: *arXiv preprint arXiv:2302.13971* (2023).

[143] 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: *Proc. ACM EuroSys*. 2016.

[144] *Tutel*. https://github.com/microsoft/tutel.

[145] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need". In: *Proc. NeurIPS* 30 (2017).

[146] 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: *Proc. ACM SoCC*. 2013.

[147] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proc. ACM EuroSys*. 2015.

[148] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, et al. "Deep learning for computer vision: A brief review". In: *Computational intelligence and neuroscience* 2018 (2018).

[149] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. "Blink: Fast and Generic Collectives for Distributed ML". In: *Proc. MLSys*. 2020.

[150] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. "Wavelet: Efficient DNN training with tick-tock scheduling". In: *Proc. MLSys* (2021).

[151] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. "{TopoOpt}:

Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs". In: *Proc. USENIX NSDI*. 2023.

[152] Xin Wang, Yudong Chen, and Wenwu Zhu. "A survey on curriculum learning". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[153] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. "Emergent abilities of large language models". In: *arXiv preprint arXiv:2206.07682* (2022).

[154] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters". In: *Proc. USENIX NSDI*. 2022.

[155] *WMT 19*. https://github.com/facebookresearch/fairseq/blob/main/examples/wmt19/README.md.

[156] Thomas Wolf et al. "Transformers: State-of-the-Art Natural Language Processing". In: *Proc. EMNLP*. 2020.

[157] 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". In: (2016). arXiv: 1609.08144.

[158] 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: *Proc. USENIX OSDI*. 2018.

[159] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning". In: *Proc. USENIX OSDI*. 2020.

[160] Fuzhao Xue, Xiaoxin He, Xiaozhe Ren, Yuxuan Lou, and Yang You. "One student knows all experts know: From sparse to dense". In: *arXiv preprint arXiv:2201.10890* (2022).

[161] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul Chilimbi. "Performance modeling and scalability optimization of distributed deep learning systems". In: *Proc. ACM SIGKDD*. 2015.

[162] Zhewei Yao, Xiaoxia Wu, Conglong Li, Connor Holmes, Minjia Zhang, Cheng Li, and Yuxiong He. "Random-LTD: Random and Layerwise Token Dropping Brings Efficient Training for Large-scale Transformers". In: *arXiv preprint arXiv:2211.11586* (2022).

[163] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. "Optimizing distributed training deployment in heterogeneous GPU clusters". In: *Proc. CoNEXT*. 2020.

[164] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. "Large batch optimization for deep learning: Training bert in 76 minutes". In: (2019). arXiv: 1904.00962.

[165] Peifeng Yu and Mosharaf Chowdhury. "Salus: Fine-grained gpu sharing primitives for deep learning applications". In: *Proc. MLSys* (2020).

[166] Mingshu Zhai, Jiaao He, Zixuan Ma, Zan Zong, Runqing Zhang, and Jidong Zhai. "SmartMoE: Efficiently Training Sparsely-Activated Models through Combining Offline and Online Parallelization". In: *USENIX ATC*. 2023.

[167] 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: *Proc. USENIX ATC*. 2017.

[168] Yizhe Zhang, Siqi Sun, Michel Galley, Yen-Chun Chen, Chris Brockett, Xiang Gao, Jianfeng Gao, Jingjing Liu, and Bill Dolan. "DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation". In: *ACL, system demonstration*. 2020.

[169] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. "Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale". In: *Proc. VLDB Endow.* 2014.

[170] Hanyu Zhao et al. "HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees". In: *Proc. USENIX OSDI*. 2020.

[171] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. "Multi-resource interleaving for deep learning training". In: *Proc. ACM SIGCOMM*. 2022.

[172] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning". In: *Proc. USENIX OSDI*. 2022.

[173] Pengfei Zheng, Rui Pan, Tarannum Khan, Shivaram Venkataraman, and Aditya Akella. "Shockwave: Fair and Efficient Cluster Scheduling for Dynamic Adaptation in Machine Learning". In: *Proc. USENIX NSDI*. 2023.

[174] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. "Mixture-of-experts with expert choice routing". In: *Proc. NeurIPS* (2022).

[175] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. “Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training”. In: *Proc. USENIX ATC*. 2020.

[176] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. “Prioritymeister: Tail latency qos for shared networked storage”. In: *Proc. ACM SoCC*. 2014.

[177] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. “Designing Effective Sparse Expert Models”. In: *arXiv preprint arXiv:2202.08906* (2022).

[178] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. “Taming sparsely activated transformer with stochastic experts”. In: *arXiv preprint arXiv:2110.04260* (2021).

List of Publications

[1] **Jiamin Li**, Qiang Su, Yitao Yang, Yimin Jiang, Cong Wang, and Hong Xu. "Adaptive Gating in Mixture-of-Experts based Language Models". In: *arXiv preprint arXiv:2310.07188* (2023).

[2] **Jiamin Li**, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. "Accelerating Distributed MoE Training and Inference with Lina". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 945–959.

[3] **Jiamin Li**, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. "Lyra: Elastic scheduling for deep learning clusters". In: *Proceedings of the Eighteenth European Conference on Computer Systems (ACM EuroSys 23)*. 2023, pp. 835–850.

[4] Libin Liu, Hong Xu, Zhixiong Niu, Jingzong Li, Wei Zhang, Peng Wang, **Jiamin Li**, Jason Chun Xue, and Cong Wang. "ScaleFlux: Efficient Stateful Scaling in NFV". In: *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 33.12 (2022), pp. 4801–4817.

[5] Kaiwei Mo, Chen Chen, **Jiamin Li**, Hong Xu, and Chun Jason Xue. "Two-Dimensional Learning Rate Decay: Towards Accurate Federated Learning with Non-IID Data". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–7.

[6] Libin Liu, Chengxi Gao, Peng Wang, Hongming Huang, **Jiamin Li**, Hong Xu, and Wei Zhang. "Bottleneck-aware non-clairvoyant coflow scheduling with Fai". In: *IEEE Transactions on Cloud Computing (TCC)* (2021).