RICE UNIVERSITY

By

Zhuang Wang

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

T. S. Eugene Ng
T. S. Eugene Ng (Nov 28, 2023 10:36 CST)

T. S. Eugene Ng (Chair)

Santiago Segarra
Santiago Segarra (Aug 18, 2023 12:01 EDT)

Santiago Segarra

Anshumali Shrivastava

Edward Knightly

Edward Knightly

HOUSTON, TEXAS

August 2023

Abstract

Deep neural networks (DNNs) have achieved unparalleled performance in numerous fields, including computer vision, natural language processing, and recommendation systems. However, the computational complexity of DNNs poses challenges due to escalating training data and model sizes. As GPU cloud network bandwidth has not kept up with GPU computational capacity as well as the burgeoning size of training data and models, scaling DNN training becomes problematic. This thesis offers a comprehensive solution by enhancing communications in the *data plane* for model training and in the *management plane* for fault tolerance. The primary tenet is that communication bottlenecks in distributed deep learning (DDL) can be alleviated by utilizing current hardware resources within a training system, complemented by intelligent traffic and resource scheduling algorithms.

Zen: Addressing data-plane communication issues when tensors have high sparsity, Zen introduces a provably optimal communication scheme for sparse tensor synchronization to minimize the communication time in DDL, bridging the existing gap. By comprehensively analyzing sparse tensor characteristics in mainstream DNN models and systematically exploring the design space of communication schemes for sparse tensors for the first time, we reveal the optimal communication scheme and realize it using a novel hierarchical hashing algorithm, capitalizing on GPU's parallel computing capabilities to minimize the operation overheads.

Espresso and Cupcake: For DNN models with minimal sparsity, we introduce Espresso and Cupcake, systems that harness gradient compression (GC) algorithms to optimize data-plane communications. Espresso employs a decision tree abstraction to pinpoint near-optimal compression strategies determining how to apply GC to each gradient tensor. It also leverages a compression decision algorithm to analyze

the intricate interactions among tensors and optimally offloads compression operations from GPUs to CPUs. Recognizing inefficiencies in applying GC algorithms tensor by tensor, Cupcake uses a fusion approach that merges multiple tensors for a single compression operation and finds the provably optimal fusion strategy to maximize their training throughput by reducing the amount of communicated data and minimizing the compression overhead simultaneously. Espresso has been partially deployed at ByteDance GPU clusters as a compression module.

Gemini: On the management plane, Gemini substantially reduces recovery overhead for DDL fault tolerance. Traditional model checkpointing, reliant on remote persistent storage to regularly store checkpoints, often leads to prolonged recovery times due to large checkpoint sizes and bandwidth limitations. Gemini utilizes a hierarchical storage system, consisting of local CPU memory, remote CPU memory, and remote persistent storage, to store checkpoints. It employs an optimized checkpoint placement strategy to maximize the probability of failure recovery from checkpoints in CPU memory, ensuring rapid failure recovery. The system also integrates a communication scheduling algorithm, allowing for minimal interference between checkpointing and model training. Notably, Gemini has been adopted by Amazon Web Services (AWS) to bolster fault tolerance in extensive language model training.

Acknowledgments

I would like to express my deepest gratitude to the following individuals who have played a crucial role in the completion of this Ph.D. thesis. Their support, guidance, and encouragement have been invaluable throughout this challenging yet rewarding four-year journey.

First and foremost, I am profoundly grateful to my advisor, Prof. T. S. Eugene Ng, for his unwavering support, mentorship, and insightful guidance. Prof. Ng gave me a huge amount of freedom and trust to explore interesting and fundamental research topics in machine learning systems during my graduate studies. He taught me how to identify the right problems to work on, how to become an independent researcher, and how to behave professionally as a scholar. His expertise and commitment to excellence have had a significant impact on my research tastes and my academic growth.

I extend my heartfelt thanks to the members of my thesis committee, Prof. Edward Knightly, Prof. Anshumali Shrivastava, and Prof. Santiago Segarra, for their expertise, time, and thoughtful critiques. Prof. Knightly's scientific insight and intuition have been extremely important in shaping this thesis, even though he is not an MLSys insider. Prof. Shrivastava's excellent insights on machine learning algorithms have inspired me a lot to rethink ML problems from a system perspective. Prof. Segarra's insightful suggestions and constructive feedback have greatly enhanced the overall quality of this thesis.

During my graduate studies, I had the opportunity to interact with many excellent research collaborators at both ByteDance and AWS: Yibo Zhu, Haibin Lin, Yida Wang, Zhen Jia, Xinwei Fu, Shuai Zheng, and Zhen Zhang. I would like to thank them for their collaboration and the stimulating discussions that have contributed significantly to the development of my research. Their collective insights and diverse

perspectives have been instrumental in broadening the scope and depth of my work.

I also want to extend my gratitude to all members of the BOLD lab and my friends at Rice University, including Sushovan Das, Weitao Wang, Xinyu Crystal Wu, Hariharan Sezhiyan, Jiarong Xing, Yiming Qiu, Hongyi Liu, Patrick Kon, Qiao Kang, Zhaozhuo Xu, Zichang Liu, Zhi Yan, Yilong Ju, and Yunxi Liu. Thank you all for your companionship, encouragement, inspiration, and the wonderful memories we shared throughout this journey. I want to give a special acknowledgment to Zhaozhuo Xu for helping me start my research in machine learning systems and patiently introducing the basic background in machine learning algorithms.

Last but not least, my sincere thanks go to my family for always supporting and believing in me. I am deeply indebted to my dear wife, Luming Yang, for her unconditional love, understanding, and encouragement throughout my graduate years. This thesis would not be possible without your continual support. I also appreciate the company of my cute corgi, Cupcake.

Contents

Illustrations

Tables

Chapter 1

Introduction

Machine learning algorithms, especially Deep Neural Networks (DNNs), have recently achieved record-breaking performance in a wide range of domains, such as computer vision [87, 196, 190, 129], speech recognition [65], natural language processing (NLP) [67, 104, 124, 204], and recommendation systems [54, 210]. The current success of DNNs is collectively brought by three factors: models, data, and hardware.

The evolution of model architectures has catalyzed the integration of deep learning into various applications. Recurrent Neural Networks (RNNs) [182, 185] have significantly propelled the accuracy of tasks like speech recognition and video tagging. Convolutional Neural Networks (CNNs) [82, 205] excel in image recognition and analysis as well as object detection and segmentation. The domain of NLP has undergone revolutionary shifts with the advent of BERT [67] and Transformer [204]. These breakthroughs have led to the emergence of Large Language Models (LLMs) with trillions of parameters [58]. Remarkably, these LLMs exhibit capabilities comparable to, and sometimes surpassing, human performance in tasks like language translation, code generation, and medical diagnosis [158].

Underpinning the remarkable achievements of DNNs is the pivotal role of data. As DNNs become more intricate, the demand for larger training datasets intensifies. For example, the ImageNet database [64] marked the inception of deep learning in computer vision. Similarly, the dataset released in Netflix challenge [40] ushered in a new era of research in recommendation systems. Reinforcement learning's success in gaming owes much to vast data from simulated environments and self-play. LLMs, such as GPT-3 [44], also rely on vast text databases sourced from the internet, encompassing books, web content, Wikipedia, articles, and more. The training dataset for GPT-3 is approximately 570GB in size.

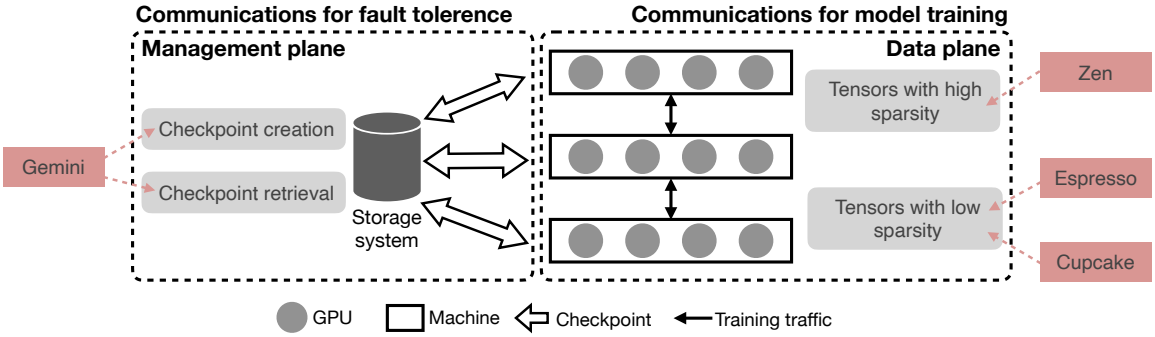


Figure 1.1 : The thesis overview.

The pursuit of larger training data and model sizes demands substantial computing resources, posing pressure on hardware. In the wake of the plateauing of Moore's Law and Dennard Scaling, hardware specialization has become integral for efficient DNN training. GPUs have been emerging as the driving force behind accelerated DNN training, sparking the deep learning explosion. Notably, GPU architectures have evolved significantly, with computing power surging by over 100× in the past eight years [1, 7].

Distributed deep learning (DDL) systems. These three pillars—models, data, and hardware—constitute the foundation of deep learning. Yet, the synthesis of these elements necessitates distributed deep learning (DDL) systems [156, 25, 50, 134] for scaling up DNN training efficiently. Given the magnitude of data and model complexity, single GPU-based DNN training can span months, if not years [142, 57]. DDL systems offer the potential to accelerate training with data parallelism [111, 94, 187, 25, 112] by leveraging multiple GPUs to concurrently process training data. Furthermore, due to the sheer number of parameters, large DNN models cannot be accommodated by a single GPU's limited memory, necessitating model sharding across multiple GPUs. DDL systems provide sophisticated abstractions for model sharding, embracing diverse parallelism strategies such as tensor model parallelism [189], pipelining parallelism [142], and ZeRO [166].

A distributed deep learning system typically consists of two components: 1) a *data plane* that involves a set of GPU machines for model training, and 2) a *man-*

agement plane that involves a storage system storing checkpoints of model states for fault tolerance, as shown in Figure 1.1. In the data plane, each GPU machine has one or multiple GPUs. Either the training data or model or both are partitioned among GPUs according to the applied parallelism strategies. The *training traffic*, i.e., the traffic for model computation, such as gradient synchronization in data parallelism [111, 112] and parameter fetching in ZeRO [166], is communicated among GPUs. Due to the large number of GPUs participating in a distributed training workload and its extended training time, especially for large language model (LLM) training, both software failures [92, 152] and hardware failures [203, 79, 198] can frequently occur. In the management plane, the GPU machines are required to periodically checkpoint the model states and transmit the *checkpoint traffic* to a storage system [139, 145]. When failures occur, GPU machines retrieve the saved checkpoint from the storage system for failure recovery, instead of restarting training from the very beginning.

Communications hinder scaling up distributed deep learning. Both the training traffic in the data plane and the checkpoint traffic in the management plane can become performance bottlenecks to scaling up DDL.

• In the data plane, there exists an exacerbating tension between computation and communication. The recent innovations of hardware accelerators [118, 146] and domain-specific software optimization [51, 238, 55] have dramatically reduced the computation time of DNN training. For example, the single-GPU iteration time of ResNet50 has seen a 22× decrease in the last seven years [194]. This trend leads to more frequent gradient synchronization in DDL and puts higher pressure on the network. However, it is difficult for GPU cloud network deployments to match this pace; network bandwidth has grown only by roughly 10× in the same period [183, 241, 128, 150, 146]. The communication time for gradient synchronization can account for more than 70% of distributed training [38].

• In the management plane, both storing and retrieving checkpoints can cause a significant waste of GPU computation resources when failures occur. To achieve continued performance improvements, DNN models, especially large language models,

have witnessed a 360× increase in the number of parameters over the last three years, from 1.5 billion in GPT-2 [164] to 540 billion in PaLM [58]. Because checkpoints are periodically saved for failure recovery, they are practically stored in a remote persistent storage system. Unfortunately, the network bandwidth connecting the GPU machines and the remote persistent storage system grows much slower (less than 10×) than the checkpoint size increases (more than 300×). It can take hours to store each checkpoint, leading to a significant loss of training progress, and to retrieve checkpoints for failure recovery, during which all GPUs have to remain idle. Training large models inevitably suffers from frequent failures due to the number of involved accelerators (e.g., tens of thousands of GPUs) and the length of training time (in months), and these failures can dramatically slow down the training progress by up to 43% [121].

1.1 Thesis Statement

The goal of this thesis is to recognize and tackle communication obstacles in both the data plane and the management plane of DDL to enhance its scalability.

Thesis Statement. This thesis demonstrates the feasibility of mitigating both data- and management-plane communication bottlenecks in distributed deep learning by *utilizing current hardware resources* within a training system, complemented by *intelligent traffic and resource scheduling algorithms*.

In particular, this thesis makes contributions to the following aspects of deep learning systems:

- A scalable system that achieves near-optimal communication time for gradient synchronization when tensors have high sparsity in distributed deep learning.
- Scalable systems that achieve near-optimal scalability of compression-enabled distributed deep learning when tensors only have low sparsity.
- A distributed system that provides efficient and scalable fault tolerance services with minimized failure recovery overhead to distributed deep learning.

A near-optimal scalable system for sparse tensor synchronization in DDL (Zen [218] in Chapter 3). Communication for gradient synchronization is a well-known performance bottleneck for the scalability of DDL [142, 172, 94, 74, 235]. The prevalence of sparsity has been widely observed in DNN training, e.g., over 98% of the gradients in a tensor can be zeros [77]. Transiting only non-zero gradients, known as *sparse tensors*, can greatly reduce traffic volume and communication time for gradient synchronization. However, the performance of existing communication schemes for sparse tensor synchronization is far from optimal and the optimal scheme is still missing. To bridge this gap, we first analyze the characteristics of sparse tensors in popular DNN models to understand the fundamentals of sparsity. We then systematically explore the design space of communication schemes for sparse tensors for the first time and find the provably optimal one that minimizes communication time. The key challenge lies in achieving balanced communications among GPUs and to date, no such scheme exists. We realize the optimal scheme with Zen, which includes a novel hashing algorithm that achieves well-balanced communications among GPUs without information loss. Zen incurs negligible hash operation overheads by using parallel computing on GPUs.

Near-optimal scalable systems for compression-enabled DDL (Espresso [214] in Chapters 4 and Cupcake [216] in Chapter 5). Zen achieves near-optimal communications for sparse tensor synchronization, and it can noticeably reduce the communication time when tensors have high sparsity in DDL. When tensors only have low sparsity, many gradient compression (GC) algorithms are proposed to shrink the communicated data size by compressing gradient tensors with different techniques, such as quantization [186, 100, 42] and sparsification [192, 27, 115]. These algorithms reduce communication time and theoretically increase training throughput. However, they only achieve moderate performance improvement or even harm the training throughput in practice because applying GC algorithms to DDL incurs additional computation overheads. We observe that the training throughput of compression-enabled DDL is determined by the *compression strategy*, including whether to compress each tensor, the type of compute resources (e.g., CPUs or GPUs) for com-

pression, the communication schemes for compressed tensor, and so on. We propose Espresso to unleash the benefits of GC algorithms by finding the near-optimal compression strategy. It first designs a decision tree abstraction to express any compression strategies and develops empirical models to timeline tensor computation, communication, and compression to enable Espresso to derive the intricate interactions among tensors. It then designs a compression decision algorithm that analyzes tensor interactions to eliminate and prioritize strategies and optimally offloads compression from GPUs to CPUs. Espresso has been partially deployed at the ByteDance GPU cluster as a compression module of BytePS [94]. We also observe that existing compression-enabled DDL systems apply GC algorithms in a layer-wise fashion, i.e., tensor by tensor. Unfortunately, this fashion can cause non-negligible compression overheads due to the fixed overheads to launch and execute kernels in CUDA [33], regardless of tensor sizes. To further improve the system efficiency of DDL, we propose Cupcake, a compression optimizer that applies GC algorithms in a *fusion fashion*, i.e., fuse multiple tensors for one compression operation. Cupcake determines the provably optimal fusion strategy to maximize training throughput by reducing the amount of communicated data and minimizing the compression overhead simultaneously.

A distributed system for fast failure recovery in DDL (Gemini [213] in Chapter 6). Because of the tremendous checkpoint size and the low network bandwidth from GPU machines to the remote persistent storage system, DDL training jobs typically checkpoint their model states every few hours [184, 20]. Therefore, the wasted time caused by a software or hardware failure can be several hours. Considering the large number of involved GPUs and the extended training time, significant GPU resources are wasted to tackle frequent failures in DDL. For example, about 178,000 GPU hours were wasted due to various training failures according to the report from OPT-175B training [233]. To minimize the failure recovery overheads, we propose Gemini, a distributed system that optimizes both the lost training progress and the stall time for training recovery by checkpointing model states to the CPU memory. Because the availability of checkpoints stored in CPU memory cannot be guaranteed when failures occur, we develop a provably near-optimal checkpoint place-

ment strategy to maximize the probability of failure recovery from checkpoints in CPU memory. Furthermore, since the communication traffic for training and checkpointing to CPU memory share the same network, checkpoint traffic might interfere with training traffic and harm training throughput. We then propose a communication scheduling algorithm that pipelines checkpoint traffic across GPU machines to minimize, if not eliminate, its interference with model training. Gemini reduces the wasted time for each failure from several hours to a few minutes by automatically checkpointing the model states at the optimal frequency and incurs no overhead on the training throughput of DDL. It is being deployed at Amazon Web Services (AWS) to provide fault tolerance to large model training involving thousands of GPUs.

1.2 Thesis Contributions

This section summarizes the main contributions of the thesis building on top of the materials from our past papers about Zen [218], Espresso [214], Cupcake [216], and Gemini [213].

Chapter 3 introduces Zen, a scalable system that minimizes communication time in sparse tensor synchronization. We make the following contributions:

- We comprehensively analyze the characteristics of sparse tensors in popular DNN models to understand the fundamentals of sparsity.
- We systematically explore the design space of schemes for sparse tensor synchronization for the first time.
- We find the provably optimal schemes for sparse tensor synchronization from the design space.
- We formalize a new problem for how to realize the optimal scheme.
- We propose a novel hierarchical hashing algorithm that approximately realizes the optimal scheme with theoretical guarantees and leverages parallel computing on GPUs to minimize overheads of hash operations.

Chapter 4 introduces Espresso and Chapter 5 introduces Cupcake. The main contributions of these two chapters are:

- We fundamentally analyze the challenges of efficiently applying gradient compression algorithms to DDL.
- We advocate leveraging different types of compute resources in training systems to perform gradient compression simultaneously.
- We design a decision tree abstraction to holistically describe the search space of compression strategies for any compression-enabled DDL.
- We devise a compression decision algorithm that selects a near-optimal strategy in seconds.
- We devise an algorithm that can find the provably optimal fusion strategy to maximize the training throughput of compression-enabled DDT jobs in seconds.
- We build compression-enable systems with Espresso and Cupcake, respectively. Espresso has been partially deployed at the ByteDance GPU cluster as a compression module of BytePS [94].

Chapter 6 introduces Gemini, a scalable system that provides efficient fault tolerance to large model training. The main contributions are as follows.

- We propose the first system that takes advantage of CPU memory for checkpoints to achieve efficient failure recovery in large model training, regardless of the underlying parallelism strategy.
- We design a provably optimal checkpoint placement strategy that maximizes the probability of failure recovery from CPU memory.
- We propose a communication scheduling algorithm that pipelines checkpoint traffic across GPU machines to minimize its interference with model training.

- We implement Gemini atop DeepSpeed [134]. Compared to existing solutions, Gemini achieves a faster failure recovery by more than 13× without incurring overhead on training throughput. It is being deployed at AWS to provide fault tolerance to large model training.

Chapter 2

Background

2.1 Basics of Deep Neural Networks

In the realm of artificial intelligence [181, 222] and machine learning [242, 96, 132], Deep Neural Networks (DNNs) have emerged as a transformative technology, reshaping the landscape of computer vision [87, 190], natural language processing [67, 164], speech recognition [76, 230], and numerous other domains [173, 86]. At their core, DNNs are a class of machine learning models inspired by the structure and functioning of the human brain [195, 135]. They have the remarkable ability to learn and make sense of complex patterns and features within data, rendering them a cornerstone of modern AI applications.

The foundation of DNNs lies in *neurons*, which are the basic building blocks of these networks. These neurons mimic the neurons in the human brain and carry out computations on incoming data. Essentially, a neuron functions as a mathematical function that accepts one or more inputs, multiplies them by values of *parameters*, and then adds them. Subsequently, the result is subjected to a non-linear function to generate the neuron's output. In a neural network, neurons are organized into *layers*: an input layer, one or more hidden layers, and an output layer, as shown in Figure 2.1. These layers are interconnected, forming a network that can process information in a hierarchical manner. DNNs typically comprise numerous hidden layers, often ranging from dozens to hundreds.

The data is propagated through the network layer by layer in *forward propagation*, with each neuron in a layer receiving inputs from the previous layer, computing its output, and passing it to the next layer. One of the key reasons for the effectiveness of DNNs is their capacity to learn from data. This learning process is achieved through

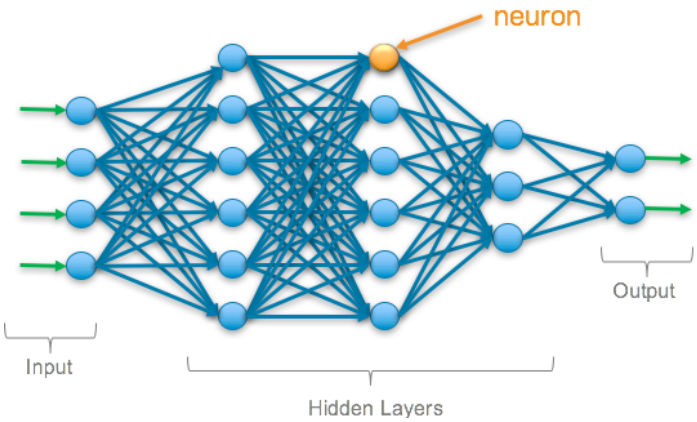


Figure 2.1 : The illustration of neural networks. DNNs typically comprise tens of or even hundreds of hidden layers. The figure is created by [2].

a technique called *backpropagation*. It calculates the loss function based on the output of a DNN model via forward propagation and the ground truth. It then uses the loss value to compute the *gradient* of each parameter. The gradients computed for each layer during backpropagation form a *gradient tensor*. Finally, it uses gradient tensors to update the parameters in each layer with a certain optimizer, such as SGD [243] or Adam [103]. Training a DNN model is a process to refine the model parameters with the above steps iteratively until its convergence.

2.2 Distributed Deep Learning (DDL)

Data parallelism. DNNs demand larger training datasets to achieve higher model accuracy. However, it can take weeks or even months to finish today's DNN workloads with a single GPU [142, 57]. A common strategy to accelerate DNN training is data parallelism, which uses multiple GPUs to digest the datasets simultaneously [111, 94, 187, 25, 112]. In data parallelism, each GPU has a replica of the DNN model; the training dataset is divided into multiple partitions and each GPU takes one partition. Each GPU consumes a mini-batch of training data from its own partition at the beginning of an iteration. It then independently performs forward propagation and backpropagation to generate gradient tensors. Since GPUs have different inputs, their generated gradient tensors will also be different. They need to synchronize gradient

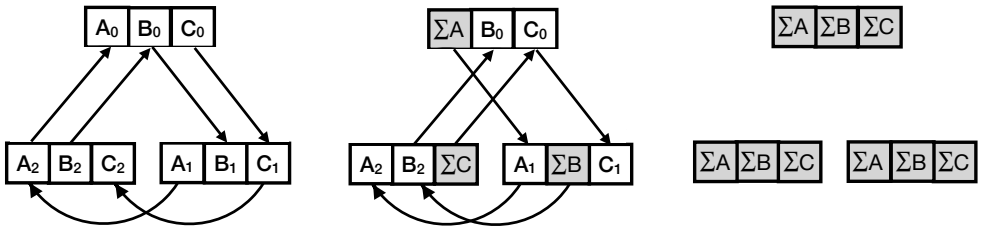
tensors from all GPUs to ensure model consistency synchronously or asynchronously. Synchronous data parallelism, where all GPUs communicate the gradient tensors and wait for the aggregated results prior to the next iteration, is the de facto standard used by DDL frameworks [94, 112, 187, 25]; asynchronous data parallelism, where GPUs do not wait for aggregation to complete, can hurt the model accuracy [49]. We focus on synchronous data parallelism in this thesis because of its wide adoption.

Model parallelism. DNN models have demonstrated remarkable capabilities in a wide range of tasks with their architectures growing in both depth and complexity. These expansive models often have billions of parameters, e.g., 540 billion in PaLM [58], but unfortunately, training such models on a single GPU is impractical due to memory limitations. Model parallelism distributes a model, instead of its training dataset, across GPUs and it divides the model into partitions, each residing on a different computational unit. These partitions can be organized in various ways, depending on the architecture of the model and the applied parallelism strategies. For example, pipeline parallelism [142, 89] breaks the layers of a DNN model network into stages, with each stage assigned to a different GPU; tensor model parallelism partitions individual layers of a model over multiple GPUs [144]. Each GPU is responsible for computing a portion of the model's forward propagation and backpropagation. In contrast, Zero Redundancy Optimizer (ZeRO) [166] shards each individual layer across all GPUs, which perform the same computation (but with different inputs) by communicating parameters on-demand before computations. In addition, it is common practice to train large DNN models with a combination of these model parallelism strategies [144, 235, 58, 184].

2.3 Data-plane Communications in DDL

The data plane in DDL involves a set of GPUs that need to synchronize their gradient tensors in each iteration, leading to data-plane communications. Allreduce [157] and Parameter Server (PS) [111] are two widely adopted gradient synchronization strategies to support data-plane communications in DDL.

Allreduce. It aggregates the gradients from every GPU in a collective fashion in



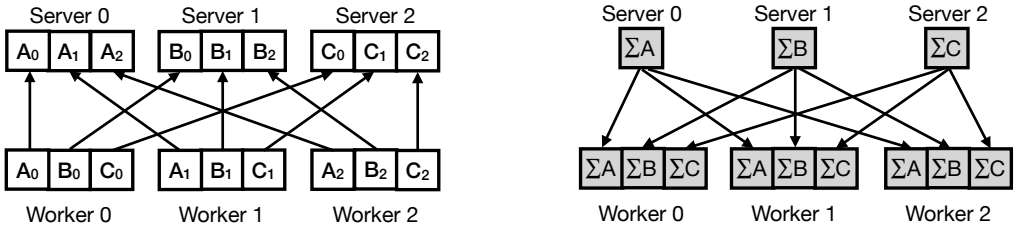
(a) Reduce-scatter (b) All-gather (c) Result

Figure 2.2 : The communication workflow of Allreduce [94].

each iteration before GPUs update their parameters locally. Ring-Allreduce is the most popular Allreduce algorithm [157] and its example with three GPUs is shown in Figure 2.2. It is dissected into two communication operations: reduce-scatter and all-gather. Suppose there are n GPUs involved in DDL and their ranks are indexed from 0 to $n-1$. Figure 2.2a shows that the reduce-scatter operation evenly partitions a tensor into n parts. It then uses n rings with different starting and ending points to reduce the n parts, respectively. For the i^{th} part, its starting point is GPU i and its ending point is GPU $(n+i-1) \mod n$. It takes $n-1$ steps to reduce each part of the n GPUs and the reduced results of the i^{th} part are in GPU $i-1$ after reduce-scatter. Next, the all-gather operation broadcasts the reduced part to all other $n-1$ GPUs using a ring structure that takes another $n-1$ steps, as shown in Figure 2.2b. After that, all GPUs have identical data that have been all-reduced, as shown in Figure 2.2c.

Parameter Servers (PS). The PS architecture [50, 94, 111] consists of two roles: *workers* and *servers*. It has three steps for the gradient synchronization of a tensor: 1) workers evenly partition the tensor across multiple servers for traffic load balance and then *Push* them to servers, as illustrated in Figure 2.3a; 2) servers *aggregate* the gradients across all workers; and 3) workers *Pull* the aggregated tensor from servers to update the DNN model, as illustrated in Figuer 2.3b.

Because of the layered structure and a layer-by-layer computation pattern in DNNs [39], the wait-free back-propagation mechanism (WFBP) [231, 94] is proposed to overlap communication with computation in data parallelism. A gradient tensor



(a) Push (b) Pull

Figure 2.3 : The communication workflow of Parameter Server [111].

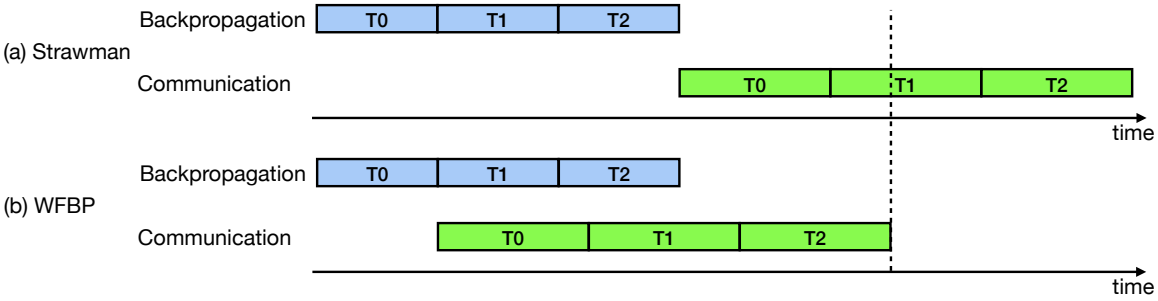


Figure 2.4 : Gradient computation and communication timeline in DDL. In (a), the gradient tensor communications need to wait for the completion of all of their computations. In (b), the communication of a tensor can begin once its computation finishes.

can begin its synchronization once it is ready during backpropagation, rather than waiting for the completion of all gradient tensors, as illustrated in Figure 2.4. WFBP is widely adopted by DDL frameworks [187, 94, 112, 50, 25] because it can significantly reduce the communication time for gradient synchronization and improve training throughput. However, there still exists an exacerbating tension between computation and communication in DDL.

Gradient synchronization communication is the performance bottleneck. The recent advancements in ML hardware accelerators [146] and specialized software stacks [51, 238, 179] have significantly improved the single-GPU training speed. For instance, the single-GPU iteration time of ResNet50 has seen a 22× decrease in the last seven years [194]. Faster training speeds necessitate more frequent gradient synchronization, thereby augmenting the network's requirements. Nonetheless, network upgrades have not kept up with the pace of computation-related advancements. The network bandwidth in GPU clouds has only seen a roughly 10× increase in the same

period [128, 150, 146]. This imbalance between the fast-growing computing capability and the slower-growing communication bandwidth reduces the chance of overlapping communication with computation, and results in poor scalability of DDL. According to recent literature [38], the communication time for gradient synchronization can account for more than 70% of the total time for training BERT [67] with Allreduce and more than 60% for training Transformer [204] with PS across 16 AWS EC2 instances, each with 8 NVIDIA V100 GPUs, in a 100Gbps network. Similar findings have also been reported in other studies [74, 142, 214].

2.4 Management-plane Communications in DDL

The role of the management plane in DDL is to resume training from failures that disrupt training jobs. Both saving and retrieving checkpoints result in management-plane communications and their importance has grown substantially in response to the advancements in distributed training.

Trends in distributed training. Many large language models (LLMs) [177, 144, 165, 191, 58, 88, 200] have been developed recently to continuously push the state of the art forward because of their great potential towards artificial general intelligence (AGI). Table 2.1 lists the statistics of several recent LLMs released in the last three years. There are three observations for the training of LLMs: 1) The model size keeps increasing, 30x larger from 2020 to 2022. For example, Turing-NLG [177] released in 2020 has 17.2 billion parameters, but MT-NLG [191] released in 2022 increases the number of parameters by 30× and achieves 530 billion parameters. 2) The number of accelerators involved in LLM training keeps increasing, from hundreds to thousands. For example, 256 NVIDIA V100 GPUs are used for the training of Turing-NLG, but 4480 NVIDIA A100 GPUs are used for the training of MT-NLG. These two trends are still moving forward because continued improvements have been observed from scaling the model sizes of LLMs [58]. 3) It can take months to finish model training.

Frequent failures in distributed training. Developers have observed many failures during LLL training due to the large number of GPUs and the long training time. For example, Meta used 992 NVIDIA A100 GPUs to train OPT-175B. The

Model	Parameters	Accelerators	Training time	Checkpoint size	Year
Turing-NLG [177]	17.2B	256 V100	—	206 GB	2020
GPT-3 [44]	175B	—	—	2.1 TB	2020
OPT-175B [233]	175B	992 A100	2 months	2.1 TB	2021
Gopher [165]	280B	4096 TPU v3	1.3 months	3.4 TB	2021
MT-NLG [191]	530B	4480 A100	3 months	6.4 TB	2022
PaLM [58]	540B	6144 TPU v4	2 months	6.5 TB	2022

Table 2.1 : The statistics of recent large language models. The optimizer is Adam [103]. — indicates no data is found from public resources.

training process encountered around 110 failures over a period of two months and about 178,000 GPU hours were wasted due to various training failures [233]. Similar symptoms have also been reported during training BLOOM [5]. Considering the three trends in distributed training, it is expected that future distributed training will experience even more frequent failures.

Checkpoints for failure recovery. Frequent failures can result in a significant waste of computation resources. The *model states*, i.e., the learnable parameters and the optimizer states, are maintained in GPU memory during model training. In case of a failure, the model states learned so far can be lost, leading to a loss of training progress and wastage of computation resources [139]. One commonly used approach to handle failures is model checkpointing. It has two phases: 1) during training, the training system regularly stores training snapshots by dumping model states into a storage system [71, 139, 5], incurring *checkpoint saving communications*; and 2) upon the occurrence of a failure, the training system retrieves the latest checkpoint from the storage to resume training on GPU machines, causing *checkpoint retrieval communications*. Both checkpoint saving and retrieval communications are management-plane communications in DDL. Furthermore, machine learning practitioners can restart the training process from a certain checkpoint should the model fail to converge in later iterations or exit unexpectedly due to failures. The state-of-the-art distributed training adopts a synchronized method to guarantee model quality [233, 144, 228], making

it infeasible to only drop the training progress of the failed machines upon a failure to proceed training without waiting for the failure recovery. Instead, it requires all machines to roll back to the same checkpoint for failure recovery.

Failure recovery overheads with checkpoints. The overhead associated with failure recovery in DDL is primarily influenced by the checkpoint frequency and the duration required to retrieve the checkpoint from the storage. In the event of a failure, the training process can resume from the most recent checkpoint, but any progress made during the time period between that checkpoint and the time point of the failure is lost. A higher checkpoint frequency is desirable because it reduces the duration of this lost progress. However, the frequency is constrained by the time it takes to save the model states to the storage system because a new checkpoint cannot commence until the previous one is completed. Meanwhile, all GPUs must remain idle while the checkpoint is being retrieved from storage. In other words, the overhead of failure recovery is determined by the communication time required for the transmission of checkpoints to storage and their subsequent retrieval from storage.

Checkpoint communication is the performance bottleneck. The burgeoning number of parameters in DNN models has led to a substantial increase in checkpoint sizes in distributed training. As listed in Table 2.1, over the past three years, we have witnessed a remarkable 360-fold expansion in checkpoint sizes, soaring from 206GB for Turing-NLG [177] to 6.4TB for MT-NLG [58]. These checkpoints are periodically stored in a remote storage system [71, 184], such as HDFS [43] and S3 [153], and are both saved and retrieved via the network connecting GPU machines and the remote storage system. Unfortunately, the growth in bandwidth of this network has lagged far behind, increasing by less than a factor of 10 during the same period. This disconnect has significant consequences, notably in significantly prolonging the time required for checkpoint saving communications and checkpoint retrieval communications and exacerbating the overhead associated with failure recovery. To put it into perspective, it takes 42 minutes to checkpoint the model states of MT-NLG to a remote storage system with a network bandwidth of 20Gbps, and the average recovery overhead exceeds 1.7 hours for each failure [213]. Considering thousands of GPUs involved in

training and hundreds of failures experienced during training, the total computation resource waste is significant, and the training time slowdown can be up to 43% [121].

Chapter 3

Zen: Near-Optimal Communications for Sparse and Distributed DNN Training

The ever-growing size of DNN models and training datasets enhances the remarkable achievements of DNNs but at the cost of substantial computing resources. Completing DNN workloads on a single GPU can take weeks or even months. Distributed DNN training with multiple GPUs is commonly used to accelerate the training process. Nonetheless, communication for gradient synchronization among GPUs poses the main challenge in system efficiency [216, 214, 235], as discussed in Section 2.3. In this chapter, we will introduce a gradient synchronization system called Zen that optimizes communication time and improves training throughput by fully leveraging sparsity in gradient tensors.

3.1 Introduction

There exists an exacerbating tension between computation and communication in distributed deep learning (DDL). Recent hardware developments have greatly improved the computation efficiency of DNN training. For instance, the training efficiency of BERT [67] has been doubling every year in the past three years [125]. These advancements increase the frequency of gradient synchronization in distributed training and shift more burdens to the network, but the network upgrades have not kept up with computation improvements [128, 150, 146].

Recently, practitioners in the deep learning community have observed the prevalence of sparsity in DNN training [48, 127, 47, 217, 75, 229]. The gradients computed for each DNN layer during training form a *tensor*. Over 98% of the gradients in a tensor can be zeros [77] and these tensors can dominate the size of DNN models. We can

represent non-zero gradients with a sparse format and denote the tensor with a sparse format as a *sparse tensor*. This observation provides a great opportunity to reduce communication time if sparse tensors are transmitted for gradient synchronization.

Previous works, such as AGsparse [112], SparCML [172], and OmniReduce [74], have recognized this potential. They use various sparse formats and communication schemes for sparse tensor synchronization. However, these approaches do not fully consider the fundamental characteristics of sparsity in DNN models for their designs, resulting in suboptimal performance of communication time for gradient synchronization. The root cause is that these previous works lack an understanding of the optimal scheme for sparsity. To advance the state-of-the-art, it is essential to first revisit the fundamentals of sparsity in DNN models.

In this chapter, we comprehensively analyze the characteristics of sparse tensors. We profile the sparse tensors from popular DNN models [83, 98, 120, 67] across GPUs and iterations to gain insights into how they are related to different inputs for training. Additionally, we investigate the changes in sparse tensors before and after aggregation with varying numbers of GPUs. We also examine the locations of non-zero gradients in tensors and inspect their distributions.

We next systematically explore the design space of communication schemes to synchronize sparse tensors. To construct different schemes, we discuss four elementary dimensions that consider the communication, aggregation, partition, and balance aspects of a scheme. All existing schemes [112, 172, 74, 111] can be described by these four dimensions. We find that there exists an optimal scheme for synchronizing sparse tensors. However, the challenge lies in achieving balanced communications among GPUs and to date, no such scheme exists.

We develop a system called Zen that approximately realizes the optimal scheme within the design space described by the four dimensions. One class of natural approach is sparsity-aware tensor partitioning, but it is inefficient due to data dependency. In contrast, Zen eliminates data dependency and achieves high efficiency by using hashing algorithms. However, a challenge with hashing algorithms is the significant information loss of gradients, resulting in reduced model accuracy. To address

this challenge, we propose a hierarchical hashing algorithm that guarantees balanced communications without information loss. In addition, it can fully leverage parallel computing on GPUs to minimize the incurred computation overheads for hash operations.

We summarize our contributions as follows: 1) we conduct a comprehensive analysis of the characteristics of sparse tensors; 2) we explore the design space for communication schemes for sparse tensors synchronization, and we find the optimal scheme; 3) we propose Zen that approximately realizes the optimal scheme to achieve near-optimal communication time; and 4) we evaluate Zen and show that it achieves up to 5.09× speedup in communication time and up to 2.48× speedup in training throughput compared to the state-of-the-art methods [74, 172].

3.2 Analysis of Communication Schemes for Sparse Tensor Synchronization

3.2.1 Gradient Sparsity in DNN Training

The synchronization of gradient tensors from different GPUs is commonly required in distributed training. For example, in data parallelism [63, 187, 94, 112], the training dataset is partitioned among all GPUs. Since GPUs have different inputs, their generated gradient tensors will also be different and they need to synchronize gradient tensors from all GPUs to ensure model consistency. In tensor model parallelism [189], individual layers of a DNN model are partitioned over multiple GPUs. These GPUs synchronize the gradient tensors during backward propagation for the gradient computation of subsequent layers. In addition, it is common practice to train large DNN models [144, 235, 58, 184, 141] with a mix of data parallelism and other parallelism strategies, such as pipeline parallelism [142, 89], tensor model parallelism [189], and ZeRO [166, 236]. These training workloads must synchronize gradient tensors across GPUs as well.

High sparsity in gradient tensors has been widely observed in DNN training [48, 127, 47, 217]. Because the training of DNN models may focus on updating a subset of parameters instead of all of them [75, 229, 48], some of the gradient tensors in DNN

Model	Dataset	MLP Size	Embedding Size	Batch Size	Density
LSTM [129]	One Billion Word	20M	406M	128	1.13%
DeepFM [83]	Criteo	68M	214M	1024	2.80%
NMT [120]	IWSLT 2014 De-En	31M	112M	64	2.47%
BERT [67]	SQuAD v1.1	86M	23M	4	1.06%

Table 3.1 : DNN models and their training statistics. Density is the average density of embedding gradient tensors on one GPU.

models are naturally sparse, with most of the gradients being zeros. Table 3.1 lists the statistics of four widely deployed DNN models for both recommendation systems and language processing. Each model contains two parts: multilayer perceptron (MLP) and embedding table. There are two major observations from Table 3.1. Firstly, the embedding tables comprise a large portion of model parameters. Secondly, these embedding tables show a significant level of sparsity in their gradient tensors. We define the *density* of a gradient tensor as the percentage of its non-zero gradient values and then provide the average density for the sparse gradient tensors in the four models. As shown in the table, gradient tensors can have only 1.06% non-zero gradients in DNN training.

If the notable sparsity can be leveraged, it can significantly reduce the traffic volume for gradient synchronization and shorten the communication time in distributed training. Previous works [102, 74, 172] have recognized this potential, but the fundamental implications of sparsity are not yet understood.

3.2.2 Characteristics of Sparse Tensors

In this section, we will analyze the characteristics of sparse tensors in DNN models. The original gradient tensor is in a dense format, in which the gradients of all the parameters in a DNN layer are stored.

Definition 3.1 (Dense tensor)**.** *We define the original gradient tensor in a DNN layer as a dense tensor.*

When there are many parameters having zero gradients, we can also represent a gradient tensor in a sparse format. A typical realization of the sparse format

is coordinate lists (COO) that store a list of non-zero gradients and a list of the corresponding indices [74, 227].

Definition 3.2 (Sparse tensor)**.** *We define a gradient tensor in a sparse format as a sparse tensor.*

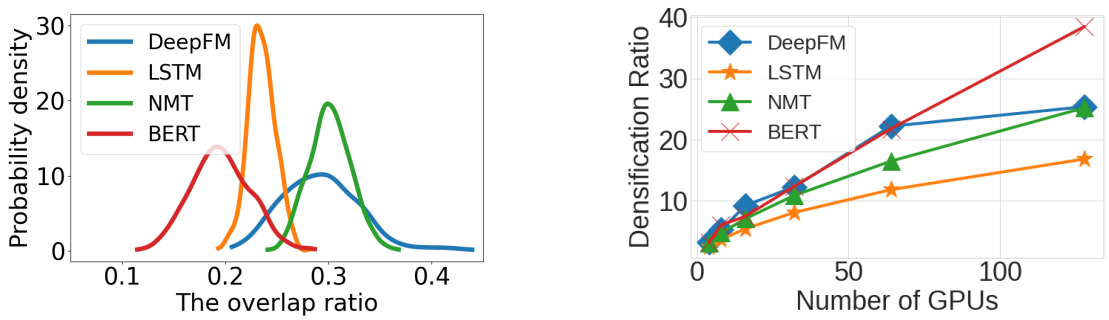
We assume that the size of a dense tensor G is M and its density is d_G. The network bandwidth is B and the training involves n machines. For simplicity, we assume each machine has only one GPU in this section.

C1: The overlap of sparse tensors varies. Similar to dense tensors, sparse tensors need to be aggregated during synchronization. When aggregating dense tensors, the indices of gradients from different GPUs are identical. However, due to the different batches as the input for training on different GPUs, the set of indices for non-zero gradients in a sparse tensor is unknown a priori. They can have overlaps, while how much they can overlap depends on many factors, such as the DNN model, the training dataset, and the batches. We define the overlap ratio following [206] to quantify this overlaps between two sparse tensors.

Definition 3.3 (The overlap ratio)**.** *Given two sparse tensors and their sets of indices for non-zero gradients are I_1 and I_2, respectively, their overlap ratio is defined as $\frac{|I_1 \cap I_2|}{\min\{|I_1|,|I_2|\}}$, where $|\cdot|$ is the cardinality of a set.*

Figure 3.1a shows the probability density function (PDF) of the overlap ratios for four DNN models. We can see that the overlap ratio in a model is approximately normally distributed and it is in a wide range. In addition, different models have different distributions of overlap ratios.

C2: The tensor size after aggregation varies. When aggregating dense tensors, the tensor sizes before and after aggregation remain the same. However, when aggregating sparse tensors, the unknown overlaps of sparse tensors lead to varying tensor sizes after aggregation. Because the aggregation involves sparse tensors from multiple GPUs, we denote d_G^n as the density after the aggregation of tensors from n GPUs. We observe that sparse tensors get denser after aggregation. Here we define the densification ratio to quantify this characteristic.



(a) (b)

Figure 3.1 : The characteristics of sparse tensors in DNN models. (a) shows that the overlap ratio of sparse tensors varies; (b) shows that tensors have higher density after aggregation.

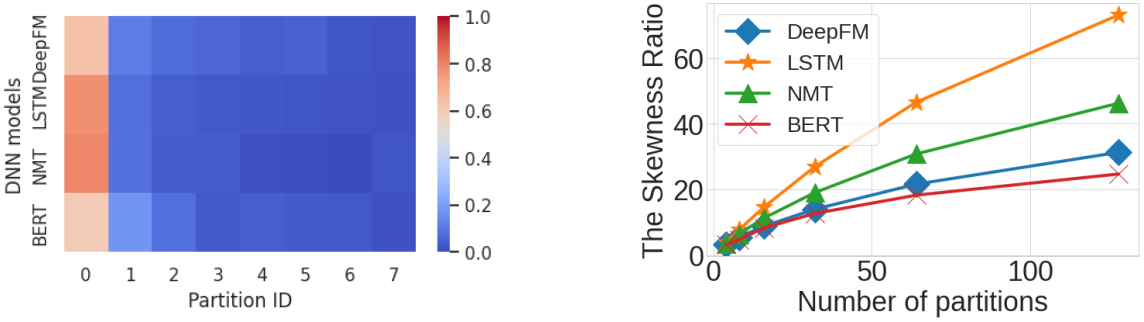
Definition 3.4 (The densification ratio)**.** *Given a dense tensor G, its densification ratio is define as* $\gamma_G^n = \frac{d_G^n}{d_G}$.

Figure 3.1b presents the average densification ratio γ_G^n with respect to the number of GPUs for the four DNN models studied in this section. The densification ratio increases with the number of GPUs, demonstrating that tensors have higher density after aggregation. We can also see that the densification ratio is smaller than the number of GPUs, i.e., $\gamma_G^n < n$. It suggests that the indices of non-zero gradients in sparse tensors from different GPUs are partially overlapped.

C3: The distribution of non-zero gradients is skewed. When evenly splitting a dense tensor into multiple partitions, we observe that most of the non-zero gradients are in one of them. For example, with eight partitions, over 60% of the non-zero gradients are in the first partition in the four DNN models. Figure 3.2a shows the percentage heatmap of the non-zero gradients in each partition for tensors from the embedding table. Here we define the skewness ratio to quantify the skewed distribution of non-zero gradients.

Definition 3.5 (The skewness ratio)**.** *Given a dense tensor G and we evenly split G into n disjoint partitions, denoted as $\{G_1, \cdots, G_n\}$, then the skew ratio of G with n partitions is defined as* $s_G^n = \frac{\max_{i\in[n]}\{d_{G_i}\}}{d_G}$.

Figure 3.2b presents the skewness ratios of gradient tensors from the embedding



(a) The heatmap of non-zero gradients distribution.

(b) The skewness ratio.

Figure 3.2 : The distribution of non-zero gradients is skewed.

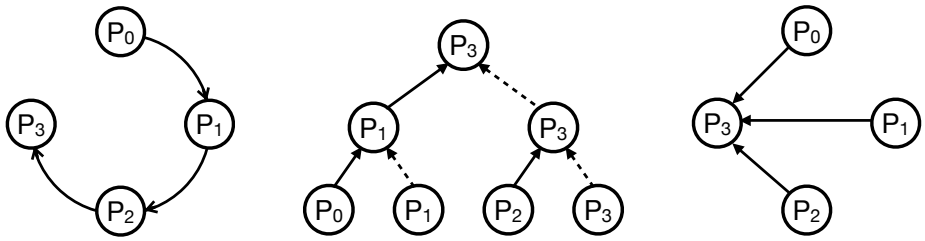
table in the DNN models studied in this section. They are significant in all four models. For example, when we evenly split the gradient tensor from the embedding table in LSTM into 128 partitions, the skewness ratio is over 70. It indicates that more than half of the non-zero gradients are in the same partition. Another observation is that the skewness ratio consistently increases with the number of partitions. It suggests that the distribution of non-zero gradients gets "skewer" with more partitions.

3.2.3 Elementary Dimensions for Synchronization

Communication schemes for synchronizations of dense tensors have been extensively studied [199, 187, 94, 111]. In this section, we will explore the design space to construct communication schemes to synchronize sparse tensors for the first time.

Given a tensor G in each GPU, the outcome of its synchronization is that gradients with the same indices are aggregated and all GPUs have identical aggregated results. We will discuss four dimensions that construct a communication scheme for the synchronizations of sparse tensors.

Communication dimension. There are typically three communication patterns for synchronization: 1) `Ring`, 2) `Hierarchy`, and 3) `Point-to-point`. They are illustrated in Figure 3.3 with an example in which there are four GPUs and GPU P_3 aggregates the data from all GPUs. In Ring, all GPUs form a ring structure. P_0 first sends its data to P_1, which then passes the data along with its own data to P_2 and so



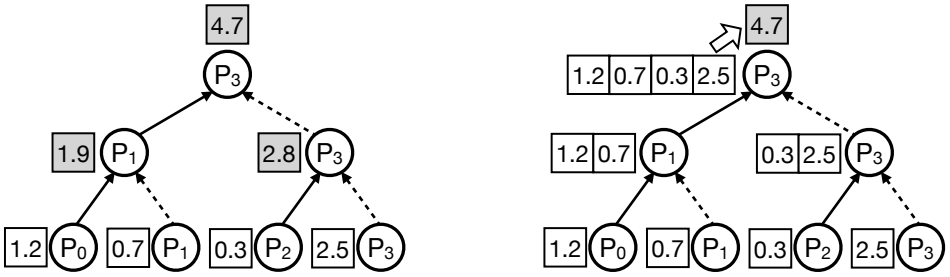
(a) Ring. (b) Hierarchy. (c) Point-to-point.

Figure 3.3 : An illustration of three communication patterns with four GPUs. GPU P_3 aggregates the data from all GPUs.

on until P_3 receives all the data. In Hierarchy, all GPUs form a hierarchical structure and P_3 is the root. There are two stages in Figure 3.3b. In the first stage, P_0 sends its data to P_1 and P_2 sends its data to P_3. In the second stage, P_1 sends the data from both its own and P_0 to P_3. In Point-to-point communication, the other three GPUs directly send data to P_3.

Aggregation dimension. A communication pattern can have multiple communication stages and thus there are two options for aggregation: 1) `Incremental aggregation`, i..e, aggregate the tensors at each communication stage; and 2) `One-shot aggregation`, i.e., aggregate tensors from all GPUs after the last communication stage. In the example illustrated in Figure 3.3, Ring has three stages and Hierarchy has two stages. Although each GPU has one tensor for synchronization, it can host multiple tensors at each stage. Figure 3.4 displays an example with Hierarchy as the communication pattern. When P_1 receives a tensor from P_0, it has two tensors due to its own tensor. P_1 can aggregate the two tensors and send the aggregated result to P_3, as shown in Figure 3.4a; it can also just send the concatenated tensor to P_3, as shown in Figure 3.4b. The two aggregation patterns for Point-to-point are identical because they only have one stage.

Partition dimension. There are two partition patterns to ensure that all GPUs have the same aggregated results after synchronization: 1) `Centralization`, in which each tensor is communicated and aggregated as a whole; and 2) `Parallelism`, in which each tensor is decomposed into multiple partitions and each partition is communicated



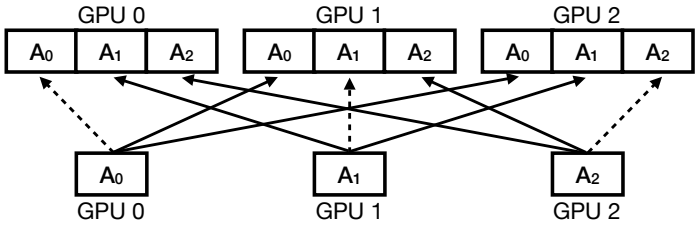
(a) Incremental aggregation. (b) One-shot aggregation.

Figure 3.4 : An Illustration of two aggregation patterns with Hierarchy. The gradients on each GPU are from the same parameter and 4.7 is the final aggregated result.

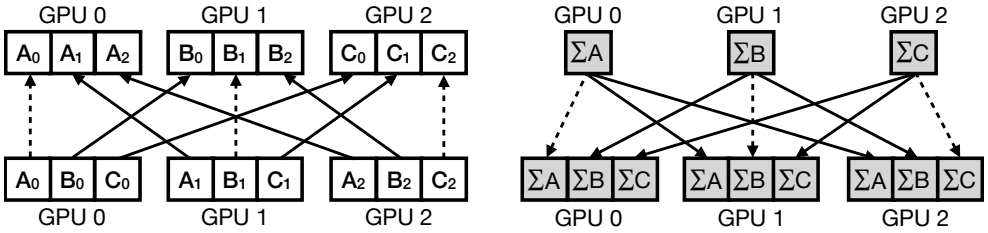
and aggregated separately. Figure 3.5 compares the two partition patterns with Point-to-point as the communication pattern. With Centralization, as shown in Figure 3.5a, each GPU sends its tensor as a whole to other GPUs. With Parallelism, as shown in Figure 3.5b, each GPU first decomposes its tensor into three partitions and it requires two steps for synchronization. The first step aggregates the same partition from different GPUs in different places and the second step ensures that all GPUs have the aggregated results of all partitions.

Balance dimension. With Parallelism, tensors are partitioned and there are two patterns in terms of the traffic volume received at each GPU: 1) `Balanced communication`, in which GPUs receive the same amount of data; and 2) `Imbalanced communication`, in which the traffic volumes received at different GPUs are greatly different. Figure 3.6 compares the two balance patterns among three GPUs with Point-to-point. There are 15 gradients in the tensor and six of them are non-zero. As shown in Figure 3.6a, four non-zero gradients are in the middle partition and they are sent to GPU 1. The traffic volume received at GPU 1 is 4× that received at GPU 0 and GPU 2. In Figure 3.6b, each GPU sends two non-zero gradients to other GPUs and the volume among them is well-balanced.

The four dimensions can describe the design space of communication schemes to synchronize sparse tensors. Table 3.2 classifies existing schemes [112, 172, 74, 111] based on their dimensions and data formats to represent sparse tensors.



(a) Centralization



(b) Parallelism

Figure 3.5 : An illustration of the two partition patterns with Point-to-point. In (a), each tensor is communicated as a whole and each GPU receives all the tensors. In (b), each tensor is split into three partitions; the same partition from different GPUs is sent to the same place, and the aggregated results are then sent back to all GPUs.

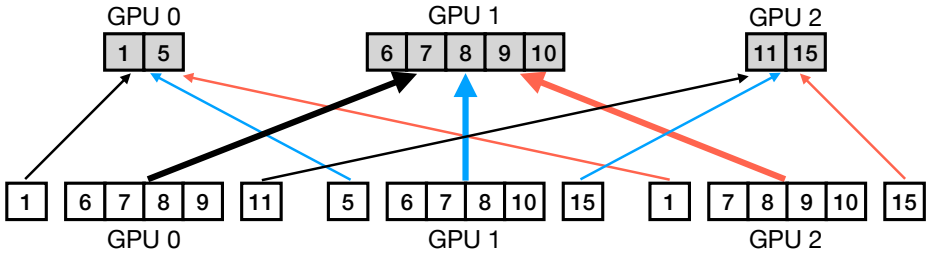
3.2.4 The Optimal Communication Schemes

In this section, we will next analyze the optimal schemes within the design space described by the four dimensions in terms of the theoretical communication time to synchronize sparse tensors.

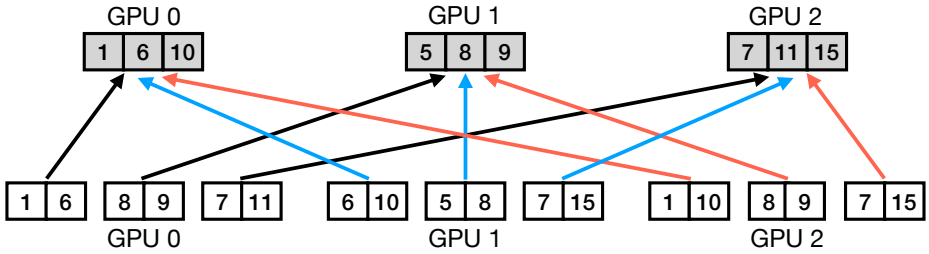
Theorem 3.1 (Optimal schemes)**.** *When choosing a communication scheme to minimize communication time:*

1. *If sparse tensors exhibit little to no overlap, the scheme with Hierarchy, Incremental aggregation, and Centralization is optimal; but this case is unlikely in reality.*
2. *If sparse tensors are partially or fully overlapped, the optimal one is the scheme with Point-to-point, One-shot aggregation, Parallelism, and Balanced communication; this case is very likely in distributed DNN training.*

Proof of Theorem 3.1.1. We prove it with three lemmas. When any two sparse tensors have no overlaps, the minimum traffic volume each GPU has to receive is



(a) Imbalanced communication



(b) Balanced communication

Figure 3.6 : An illustration of the two balance patterns with Point-to-point and Parallelism. The numbers are the indices of non-zero gradients. Each GPU has six non-zero gradients. In (a), four gradients from each GPU are sent to GPU 1. In (b), each GPU sends two gradients to other GPUs, and communications among them are well-balanced. However, it is non-trivial to achieve such balanced communications.

all the tensors from other GPUs. Any communication scheme with Centralization achieves the optimal communication time. Therefore, we have the following lemma.

Lemma 3.1. *When sparse tensors have no overlap, any communication scheme with Centralization can achieve this minimum with any communication pattern.*

When sparse tensors are overlapped, we have the following lemmas.

Lemma 3.2. *When sparse tensors overlap, the scheme with Hierarchy, Incremental aggregation, and Centralization outperforms other schemes with Centralization.*

Proof. Let n denote the number of GPUs and $I_0, I_1, \ldots, I_{n-1}$ are the set of indices for non-zero gradients in each GPU, respectively. C is the overlap of all the sparse tensors, i.e., $C = \bigcap I_i$. If a communication scheme adopts point-to-point communication or one-shot aggregation, each GPU has to receive C for $n-1$ times. Then we consider

Incremental aggregation and the communication pattern is Ring or Hierarchy. With Ring, the tensor from each GPU is aggregated at each stage and then forwarded to the next GPU. Consequently, this tensor is received by every GPU. Because C is the common overlap, each GPU also has to receive C for $n-1$ times. When the communication pattern is Hierarchy, each GPU receives data from all the other GPUs with its own hierarchical structure that has $\log n + 1$ levels, as shown in Figure 3.3b. Because the root GPU is in each level, it has to receive C for $\log n$ times. It suggests that the traffic volume in the scheme with Hierarchy, Incremental aggregation, and Centralization is less than that in other schemes with Centralization. Let C' denote the overlap of a subset of the sparse tensors, we can have a similar conclusion that a subset of GPUs has to receive C' multiple times. In other words, each GPU still has to receive the overlaps multiple times. □

Lemma 3.2 implies if we fix the choice for partition pattern to Centralization, the scheme with Hierarchy, Incremental aggregation, and Centralization is the best.

Lemma 3.3. *When sparse tensors exhibit little to no overlap, the scheme with Hierarchy, Incremental aggregation, and Centralization outperforms schemes with Parallelism.*

Proof. For schemes with Parallelism, the aggregation operation in their first step aims to reduce the traffic volume in the second step when sparse tensors overlap. But when sparse tensors exhibit little to no overlap, or minimal overlap at best, the first step has no benefits and the communication time in the second step equals the communication time of schemes with Centralization. It implies that schemes with Parallelism have longer communication time than schemes with Centralization. □

Lemmas 3.1-3.3 imply Theorem 3.1.1.

Proof of Theorem 3.1.2. Unless otherwise specified, we assume the sparse format is COO. Communication schemes with Parallelism can suffer from imbalanced communications due to the skewed distribution of non-zero gradients. Given a dense tensor G, a straightforward parallel communication scheme first evenly splits it into

multiple chunks and then extracts the non-zero gradients from each chunk. As discussed in Section 3.2.2, the distribution of non-zero gradients in a gradient tensor is skewed. One chunk can have much more non-zero gradients than other chunks. Consequently, one GPU has to receive most of the non-zero gradients from all the GPUs in the first step, leading to imbalanced communications among GPUs. After aggregation, the number of non-zero gradients in one GPU can be still much more than that in other GPUs. Therefore, communications in the second step are also imbalanced. It is important to note that an alternative to first extracting non-zero gradients from the gradient tensor and then evenly splitting the sparse tensor into multiple partitions is not a viable option. This is because the gradients for the same parameter from different GPUs can be sent to different places, causing incomplete aggregations.

Lemma 3.4. *The scheme that adopts Point-to-point, One-shot aggregation, Parallelism, and Balanced communication outperforms other schemes with Parallelism.*

Proof. There are three communication patterns: Ring, Hierarchy, and Point-to-point. We first consider communication schemes with Point-to-point and Parallelism, namely, the PS architecture. Given a gradient tensor G with the density of d_G and there are n servers, we first analyze the communication time of push and pull operations separately. We then discuss the communication time of different PS schemes.

• **Push.** Because the skewness ratio is s_G^n, the largest density in the n partitions is $s_G^n d_G$. The size of the sparse tensor extracted from this partition is $2s_G^n d_G M/n$. As a result, the communication time of push in sparse PS is $2(n-1)s_G^n d_G M/n/B$.

• **Pull.** After aggregation, the largest density in the n partitions becomes $s_G^n d_G^n$. In existing implementations of the PS architecture, the communication time of Pull is $2(n-1)s_G^n d_G^n M/n/B$ because each server needs to broadcast its aggregated results to all the workers with point-to-point communications [94, 111]. In theory, there are other ways to implement Pull in the PS architecture. For example, each server can perform a `broadcast` collective operation. The performance of `broadcast` with different algorithms is analyzed in [36] and its communication time for Pull can be expressed as $2bd_G^n M/B$, where b is the number of rounds in an algorithm. For example,

$b = \lceil \log n \rceil$ when it uses Binomial Tree Algorithm and $b = \frac{2(n-1)}{n}$ when it uses Scatter-AllGather Algorithm [81, 36].

• **Sparse PS.** Combining the communication time of push and pull with point-to-point communications, it overall communication time is $2(n-1)(d_G + d_G^n)s_G^n M/n/B$.

• **Sparse PS with the broadcast.** When considering `broadcast` for Pull, the overall communication time becomes $2(n-1)s_G^n d_G M/n/B + 2bd_G^n M/B$. We denote this case as sparse PS with the broadcast.

For simplicity, we call the scheme with Point-to-point, One-shot aggregation, Parallelism, and Balanced communication as Balanced Parallelism.

• **Balanced Parallelism.** In Balanced Parallelism, the skewness ratio s_G^n is always 1. We replace the s_G^n in the communication time of sparse PS as 1 and have the communication time for Balanced Parallelism: $2(n-1)(d_G + d_G^n)M/n/B$.

• **Balanced Parallelism is optimal among schemes with Parallelism.** It is clear that Balanced Parallelism is much better than sparse PS when the skewness ratio is large DNN models. The performance ratio of PS with broadcast to Balanced Parallelism is $\frac{s_G^n}{1+\gamma_G^n} + \frac{n}{n-1}\frac{b\gamma_G^n}{1+\gamma_G^n} > \frac{s_G^n + b\gamma_G^n}{1+\gamma_G^n}$. Because both s_G^n and b are greater than 1, the ratio is also greater than 1. Hence, Balanced Parallelism always outperforms sparse PS and sparse PS with broadcast in terms of communication time. □

Lemma 3.4 implies that if we fix the choice for partition pattern to Parallelism, the scheme with Point-to-point, One-shot aggregation, Parallelism, and Balanced communication is always the best.

Lemma 3.5. *When sparse tensors are partially or fully overlapped, the scheme with Point-to-point, One-shot aggregation, Parallelism, and Balanced communication outperforms the scheme with Hierarchy, Incremental aggregation, and Centralization because the latter cannot fully leverage the overlaps among sparse tensors to minimize the traffic volume.*

Proof. Because One-shot aggregation cannot leverage the overlaps among sparse tensors, the performance of communication schemes with One-shot aggregation is worse than those with Incremental aggregation. Therefore, we only consider Incremental

aggregation for schemes with Ring communication or Hierarchy communication. We consider the best case for them, i.e., the skewness ratio is 1 after tensor partition with Parallelism. In addition, we only need to compare the first step because they have the same communication time in the second step. The communication time of the first step in Balanced Parallelism is $2(n-1)d_GM/n/B$.

• **Schemes with Ring and Incremental Aggregation.** They have $n-1$ communication stages. The tensor density in the i_{th} stage is d_G^i. Note that $d_G^1 = d_G$. Therefore, the communication time is $2\sum_{i=1}^{n-1} d_G^i M/n/B$. Because tensors can get denser after aggregation, we have $d_G^i \leq d_G^j$ when $i < j$ and $\sum_{i=1}^{n-1} d_G^i \geq (n-1)d_G$. As a result, the communication time of schemes with Ring and Incremental aggregation is no less than that of Balanced Parallelism.

• **Schemes with Hierarchy and Incremental aggregation.** They have $\log n$ communication stages. Because each partition has a hierarchical structure, the total traffic volume in the i_{th} stage is $\frac{d_G^{2^{i-1}}}{2^{i-1}}Mn$ and the total traffic volume in all the $\log n$ stages is $V = \sum_{i=1}^{\log n} \frac{d_G^{2^{i-1}}}{2^{i-1}}Mn$. Because $d_G^{2^{i-1}} \geq d_G$, we have $V \geq \sum_{i=1}^{\log n} \frac{d_G}{2^{i-1}}Mn = 2(n-1)d_GM$. Therefore, the traffic volume received at each GPU is no less than $2(n-1)d_GM/n$ and the communication time is no less than that of Balanced Parallelism. □

In summary, Balanced Parallelism outperforms other PS schemes and the performance of other schemes with Parallelism cannot be better than Balanced Parallelism. Lemmas 3.2, 3.4, and 3.5 imply Theorem 3.1.2.

3.2.5 Numerical Comparison

As listed in Table 3.2, there are several communication schemes proposed to support the synchronizations of sparse tensors [112, 172, 74, 111]. In this section, we will compare their performance from an algorithmic perspective.

AGsparse. It adopts One-shot aggregation, Centralization, and separately collects non-zero gradients and the corresponding indices [112]. It cannot leverage the overlaps among sparse tensors to reduce the traffic volume. Note that there are different implementations for AGsparse with different communication patterns [199].

Schemes	Communication	Aggregation	Partition	Balance
AGsparse [112]	Ring, Hierarchy, Point-to-point	One-shot	Centralization	N/A
SparCML [172]	Hierarchy	Incremental	Centralization	N/A
Sparse PS [111]	Point-to-point	One-shot	Parallelism	Imbalanced
OmniReduce [74]	Point-to-point	One-shot	Parallelism	Imbalanced
Balanced Parallelism	Point-to-point	One-shot	Parallelism	Balanced

Table 3.2 : Comparison of different communication schemes for sparse tensors based on their dimensions.

SparCML. It adopts Hierarchical, Incremental aggregation, and Centralization [172]. According to Lemma 3.5, it cannot leverage the overlaps among sparse tensors to reduce the traffic volume. The performance of both AGsparse and SparCML depends on the overlaps. The fewer overlaps, the closer their performance is to the optimal. However, as shown in Figure 3.1, sparse tensors across GPUs in DNN models have significant overlaps.

Sparse PS. Parameter Servers (PS) architecture [111, 102] is a communication scheme that adopts Point-to-point, One-shot aggregation, and Parallelism. It has two roles: workers and servers. For synchronizations of sparse tensors, workers push sparse tensors to servers. After aggregation, workers pull aggregated tensors from servers to update model parameters. We call this PS architecture for sparse tensors as *Sparse PS* to distinguish it from the PS architecture for dense tensors. Because Sparse PS evenly partitions tensors, it suffers from imbalanced communications as discussed in Section 3.2.4.

OmniReduce. It also adopts Point-to-point, One-shot aggregation, and Parallelism [74]. OmniReduce consists of workers and aggregators. It splits a gradient tensor into blocks of gradients and only sends non-zero blocks, i.e., blocks with at least one non-zero gradient, to aggregators for aggregations. Compared to Sparse PS, OmniReduce does not need to transmit indices for non-zero gradients and it has a lower traffic volume for communications. However, it also requires multiple aggregators for better scalability, just like multiple servers in PS. It evenly partitions tensors

and its performance also suffers from imbalanced communications.

Figure 3.7 numerically compares the performance of these communication schemes to synchronize sparse tensors in NMT. The sparse data formats are as each scheme proposed, i.e., OmniReduce uses tensor block; AGsparse, SparCML, and Sparse PS use COO. We only consider their theoretical communication time and ignore other overheads, such as the computation time for aggregations and the sparse tensor encoding and decoding overheads. Their communication times are normalized to *Dense*, which is the synchronization time for dense tensors [*].

The communication time of AGsparse linearly increases with the number of GPUs. It performs worse than Dense with more than 40 GPUs because it does not leverage the overlaps among sparse tensors to reduce the traffic volume. Sparse PS is worse than AGsparse due to the skewed distribution of non-zero gradients and imbalanced communications among servers. It is even worse than Dense because it has to transmit both non-zero gradients and the corresponding indices. OmniReduce outperforms Dense with a small number of GPUs. However, its performance improvement is very marginal with more than 64 GPUs. Due to the skewed distribution of non-zero gradients, most of the non-zero gradients are in one partition, leading to imbalanced communications. In addition, tensors become denser after aggregation. When splitting this partition into tensor blocks (e.g., each block has 256 gradients [74]), most of them are non-zero blocks. Therefore, almost all gradients in this partition are sent to one aggregator and it becomes the communication bottleneck. SparCML is worse than Dense with a large number of GPUs due to the duplicated indices and their gradients received at each GPU.

Now, let us consider a hypothetical scheme suggested by Theorem 3.1 called Balanced Parallelism.

Balanced Parallelism. It adopts Point-to-point, One-shot aggregation, Parallelism, and Balanced communication. In Figure 3.7, we assume its sparse data format is COO for a fair comparison. Balanced Parallelism greatly outperforms existing com-

[*]We use Ring-Allreduce as an example [187]. It adopts Ring, incremental aggregation, Parallelism, and Balanced communication.

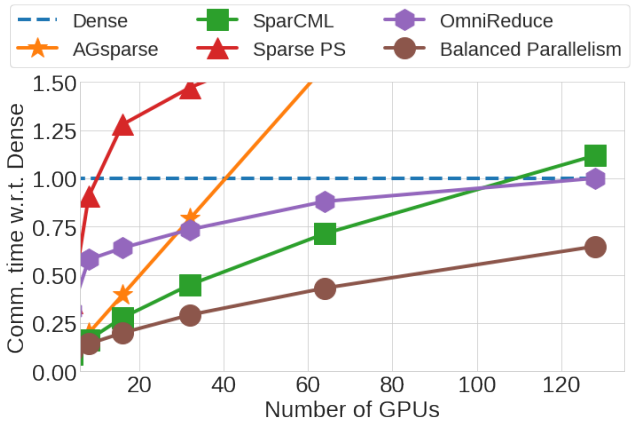


Figure 3.7 : Comparison of different schemes for synchronizing sparse tensors in NMT [120]. The sparse data formats are as each scheme proposed, i.e., OmniReduce uses tensor block; AGsparse, SparCML, and Sparse PS use COO. For a fair comparison, the data format of Balanced Parallelism is COO.

munication schemes. For example, existing schemes cannot reduce communication time compared to Dense with 128 GPUs, but the communication time of Balanced Parallelism is still 36% lower than Dense.

Takeaways. The sparsity in DNN models offers a great opportunity to optimize communications in distributed training, but no existing schemes can fully unleash this potential. Regardless of the sparse data formats, Balanced Parallelism is optimal for the common case (see Theorem 3.1) and thus substantially outperforms existing schemes such as SparCML [172] and OmniReduce [74]. However, no solution that realizes Balanced Parallelism exists to date. We will show how this gap is closed in the next section.

3.3 Zen

We propose Zen to leverage the sparsity in DNN models to minimize the synchronization time in distributed training. We first formulate the problem for achieving Balanced Parallelism and discuss two strawman solutions. We then develop a hierarchical hashing algorithm to approximately address the problem. When synchronizing

sparse tensors, a communication scheme with COO has to transmit non-zero gradients and their indices, which double the traffic volume. Therefore, we also design a new data format for the indices to minimize the traffic volume.

3.3.1 Problem Formulation

Balanced Parallelism has the same communication pattern, aggregation pattern, and partition pattern as Sparse PS, but its communications are always well-balanced. Therefore, we borrow the concepts of workers and servers from Sparse PS to Balanced Parallelism. We also call its two communication operations as Push and Pull, respectively.

Suppose there are n workers and n servers in Balanced Parallelism. $I_i \subset \mathbb{N}_+$ is the set of indices of non-zero gradients generated by worker i. We define the problem to achieve Balanced Parallelism as follows.

Problem 3.1. *Let I denote the union of $\{I_1, I_2, \cdots, I_n\}$. We would like to have a mapping $f : I \to [n]$ such that:*

1. *For every $i \in [n]$ and $j \in [n]$, the cardinality of set $\{idx \in I_i | f(idx) = j\}$ is equal to $|I_i|/n$.*
2. *For every $j \in [n]$, the cardinality of set $\{idx \in I | f(idx) = j\}$ is equal to $|I|/n$.*

Here we elaborate more on the two requirements for the mapping f accordingly as below:

1. **Load balance in Push.** For every worker, mapping f needs to decompose its non-zero gradients evenly into n partitions. Therefore, workers can transmit the same amount of non-zero gradients to each server.
2. **Load balance in Pull.** Each of the servers should have the same number of non-zero gradients after aggregation. It also implies that the same index from different workers should be sent to the same server.

Problem 3.1 assumes the same number of workers and servers, but it is easy to generalize this problem to cases in which the number of workers and servers are different.

3.3.2 Strawman Solutions

Because exactly solving Problem 3.1 is challenging, we will explore the opportunities for approximate solutions, including both data-dependent and data-independent solutions. Before discussing the solutions, we first define the imbalance ratio to measure the performance of algorithms to approximately solve Problem 3.1.

Definition 3.6 (The imbalance ratio)**.** *Given a mapping f that decompose I_i into n partitions, which are denoted as $\{I_i^1, \cdots, I_i^n\}$, the imbalance ratio of Push with f is* $\max_{i,j\in[n]}\{n|I_i^j|/|I_i|\}$.

Let I denote the union of $\{I_1, I_2, \cdots, I_n\}$ and the sets of indices at the n servers after gradient aggregation are $\{\mathbb{I}_1, \mathbb{I}_2, \cdots, \mathbb{I}_n\}$. The imbalance ratio of Pull with f is $\max_{i\in[n]}\{n|\mathbb{I}_i|/|I|\}$.

Based on Definition 3.6, the imbalance ratio of push in Sparse PS equals the skewness ratio, and in Balanced Parallelism is 1. Our goal is to minimize the imbalance ratio for any distributions of non-zero gradients in distributed training.

Data-dependent solutions. Due to different sets of indices on different workers, data-dependent solutions need to analyze their overall distribution and calculate one mapping (see Problem 3.1) for all workers, inevitably incurring non-negligible computation overheads. Therefore, we cannot afford to apply a data-dependent algorithm and obtain a mapping f for every iteration. A possible approach is to compute f periodically and maintain it for the next iterations.

However, this approach can still lead to high imbalance ratios due to the varying distributions of the indices across iterations. One strawman following this approach is to sort the index set I, evenly partition it into n parts, and use the boundary indices as the thresholds to partition the index sets in the next iterations. When we compute the thresholds every 1000 iteration for NMT model with $n = 16$ and apply these thresholds to the following iterations, the imbalance ratio of push fluctuates between 1.4 and 5.1, causing imbalanced communications among servers. Moreover, the imbalanced communications introduced by data-dependent solutions make it hard to estimate the iteration time. Many resource scheduling mechanisms for GPU clusters assume predictable and stable iteration times for allocating resources to DNN

Algorithm 3.1 A strawman solution with hashing

Input: G is a dense tensor and $I \subset \mathbb{N}_+$ is a set of indices of its non-zero gradients. $n \in \mathbb{N}_+$ is the number of partitions. $r \in \mathbb{N}_+$ is the memory size for each partition. $h : \mathbb{N}_+ \to [nr]$ is an universal hash function.

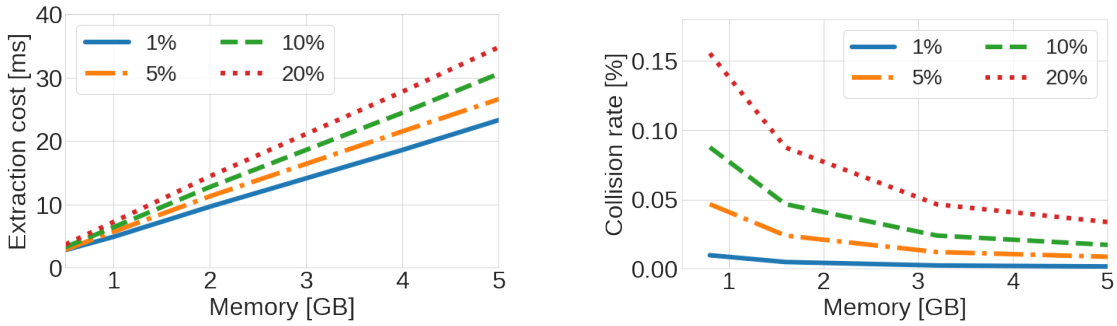
Output: The partitioned sparse tensors.

Function `Main`(I, G, $h(\cdot)$)**:**
  Allocate memory $x \leftarrow \mathbf{0}^{n \times r}$ **foreach** $idx \in I$ **in parallel do**
    $p \leftarrow \lfloor h(idx)/r \rfloor$ $q \leftarrow h(idx) \mod r$ $x[p][q] \leftarrow idx$
  end
  $output = []$ **for** $i \leftarrow 0$ **to** $n - 1$ **do**
    $indices =$ `nonzero`$(x[i])$ $values = G[indices]$ $output.append((indices, values))$
  end
  return $output$;

training jobs [143, 163, 159, 122]. It is cumbersome to schedule GPU resources with fluctuating communication time.

Strawman data-independent solution. Hashing algorithms have been widely applied to address load imbalance problems in various domains, such as distributed system [99, 193] and distributed database [72, 116]. We discuss a straightforward hashing algorithm here to approximately solve Problem 3.1. The pseudocode is illustrated in Algorithm 3.1. Note that we must leverage multiple threads in GPUs to perform hash functions to reduce computation overhead.

Given a dense tensor G, the set of indices of its non-zero gradients is I. Algorithm 3.1 first allocates a memory x with shape $n \times r$, where n represents the number of partitions and r is the memory size for each partition. For every $idx \in I$, it uses a given universal hash function [46] $h : \mathbb{N}_+ \to [nr]$ to generate the hash value $h(idx)$, where nr is the range of hash function h. Next, it writes the idx to the $(h(idx) \mod r)$th location in partition $\lfloor h(idx)/r \rfloor$. The hashing operation is performed in parallel to minimize the computation overhead [217]. After that, it extracts the non-zero indices from the memory of each partition and uses them to look up the corresponding gradients from G. Finally, it returns a sparse tensor for each partition and pushes them to the corresponding servers.



(a) Memory size vs. cost to extract indices.

(b) Memory size vs. hash collision rate.

Figure 3.8 : Larger memory size reduces the hash collision, but it leads to higher extraction overhead.

Although a universal hash function can naturally provide an approximation to Problem 3.1, it is a lossy operation. Two indices can be hashed to the same location, but only one index can be written into the memory and the other is overwritten, causing the information loss of gradients.

One possible approach to reducing the information loss is to increase the memory size, but it leads to a dilemma between the information loss and the incurred computation overhead. After writing the indices into the memory, the algorithm needs to extract the indices, which are the non-zero values in the memory. We profile the performance of the built-in `nonzero()` API in PyTorch 1.12 on NVIDIA A100 GPUs to extract the non-zero gradients of a tensor. We set the tensor size as 214M parameters, which equals the embedding table size in DeepFM, and the performance is illustrated in Figure 3.8a. The extraction cost is 19.2ms when the memory size is 4GB and the density is 1%; it increases to 29.1ms when the density is 20%. This extraction cost is unacceptable as the communication time of the dense tensors in DeepFM is only around 150ms with 128 GPUs and a 100Gbps network. However, reducing the memory size can cause non-negligible information loss. For example, when the memory size is 0.85GB, which equals the tensor size, the extraction cost is 5.8ms for the density of 20%, but 15.8% gradients are lost due to hash collision, as shown in Figure 3.8b. We will show in Section 3.4.2 that the information loss can harm model accuracy.

3.3.3 A Hierarchical Hashing Algorithm

We develop a hierarchical hashing algorithm that ensures no information loss to achieve balanced communications in distributed training with negligible extraction overhead. Before introducing our algorithm, we first discuss the possible approaches.

Approach with one hash function. As discussed in Section 3.3.2, balancing communications with only one hash function can lead to significant information loss due to hash collision. One approach to address this information loss issue is to check hash collision and write all the colliding indices into a separate memory chain [168]. The hashing operation is performed in parallel, but it has to use an atomic operation to serially write the indices into the separate memory chain (denoted as *serial memory*). Otherwise, the indices in the serial memory can still be overwritten. Unfortunately, we observe that serial writing is costly when the hash collision rate is high because only one thread can operate at one time.

Approach with multiple hash functions. Another approach is to rehash a colliding index with a new hash function to another location. There is a chance that this new location is available. However, this approach can cause incomplete aggregations. Because different GPUs have different sets of indices for non-zero gradients, their sequences of indices being hashed are also different. Therefore, the location of a particular index can be different across GPUs. For example, two indices idx_1 and idx_2, where $idx_1 < id_2$, are hashed to the same location with the first hash function. GPU 1 has idx_2; GPU 2 has both idx_1 and idx_2. In GPU 1, the location of idx_2 is determined by the first hash function, but in GPU 2, the location of idx_2 is determined by the second hash function because the location hashed by the first one has been occupied by idx_1. Subsequently, partitioning the memory will lead to the same index assigned to different partitions at different GPUs, resulting in incomplete aggregations.

Our approach. We propose a hierarchical algorithm with multiple hash functions to guarantee complete aggregations. The first-level hash function determines the partition that an index belongs to and guarantees that an index will belong to the same partition across all GPUs. The second-level hash functions determine its locations in

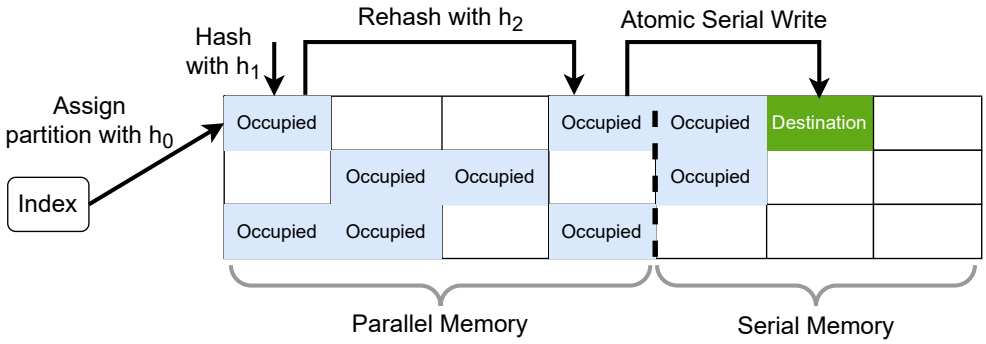


Figure 3.9 : Demonstration of the hierarchical hashing algorithm with $n = 3$, $k = 2$, $r_1 = 4$, and $r_2 = 3$. We perform parallel hashing on the indices. For each index, we use the hash function h_0 to assign its partition. Next, we use the hash function h_1 to assign it the first location. However, because this location is occupied, we rehash it with function h_2 to the fourth location. As it is also occupied we serially write the index into the serial memory with an atomic operation.

this partition. However, hash collision still exists even with multiple hash functions. We observe that the collision rate is less than 1% with four hash functions and the overhead to write the colliding indices into the serial memory becomes acceptable. Therefore, our approach also uses serial memory to achieve no information loss after multiple hashing.

We illustrate the hierarchical hashing algorithm in Figure 3.9. The pseudocode of the hierarchical hashing algorithm is shown in Algorithm 3.2. Given a dense tensor G and the indices of its non-zero gradients I, it allocates a memory x with shape $n \times (r_1 + r_2)$, where n is the number of partitions, r_1 is the memory size for parallel hashing operations, and r_2 is the serial memory size. It performs a hashing operation for every $idx \in I$ in parallel (Lines 4-17). A universal hash function $h_0 : \mathbb{N}_+ \to [n]$ is used to locate idx to partition $p = h_0(idx)$ (Line 5). The algorithm also needs k universal hash functions $H = \{h_1, \cdots, h_k\}$ with $h_i : \mathbb{N}_+ \to [r_1]$. After determining the partition p, the algorithm attempts to find an available destination $x[p][h_1(idx)]$ with h_1. If this location is available, idx is written into it. Otherwise, the algorithm rehashes idx with h_2 to find a new location. It rehashes an index for at most k rounds

Algorithm 3.2 Hierarchical Hashing Algorithm

Input: G is a dense tensor and $I \subset \mathbb{N}_+$ is a set of indices of its non-zero gradients. $n \in \mathbb{N}_+$ is the number of partitions. Each partition has memory size $r_1 + r_2$, where $r_1 \in \mathbb{N}_+$ and $r_2 \in \mathbb{N}_+$ are the memory sizes for parallel and serial operations, respectively. $h_0 : \mathbb{N}_+ \to [n]$ is a universal hash function. $H = \{h_1, \cdots, h_k\}$ is a set of universal hash functions where $h_i : \mathbb{N}_+ \to [r]$.

Output: The partitioned sparse tensors.

```
9  Function hierarchical_hash(I, G, h_0, H):
10     Allocate memory x ← 0^{n×(r_1+r_2)}
11     Allocate atomic counters c ← r_1^n
12     foreach idx ∈ I in parallel do
13         p ← h_0(idx)
14         for i ← 1 to k + 1 do
15             q ← h_i(idx)  if i = k + 1 then
16                 q ← atomicAdd(c[p], 1)
17                 x[p][q] ← idx
18             end
19             if x[p][q] == 0 then
20                 x[p][q] ← idx  break
21             end
22         end
23     end
24     output = []  for i ← 0 to n − 1 do
25         indices = nonzero(x[i])
26         values = G[indices]
27         output.append((indices, values))
28     end
29     return output
```

and uses h_i as the hash function for round i until it finds an available destination (Lines 6-16). The algorithm writes it to the serial memory allocated to partition p after k rehashes (Lines 8-11). Serial writing is an atomic operation (Lines 9-10) to ensure no information loss Once all indices are written into the memory, it extracts sparse tensors from the memory (Lines 19-23).

We next analyze the imbalance ratio of Algorithm 3.2 and highlight its properties.

Guaranteed imbalance ratio. The imbalance ratio of Algorithm 3.2 is guaranteed by the universal hash function h_0 because it determines the partition of each index.

Theorem 3.2 (Load Balance of Algorithm 3.2)**.** *Given a dense tensor G with $|G|$ parameters. Algorithm 3.2 provides a mapping $f : I \to [n]$ such that*

1. *With probability at least $1 - o(1)$, its imbalance ratio of Push is at most $1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G}})$.*
2. *With probability at least $1 - o(1)$, the imbalance ratio of Pull is at most $1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G^n}})$.*

Proof. The imbalance ratio of Algorithm 3.2 is only determined by $h_0 : \mathbb{N}_+ \to [n]$, while the hash function set H focuses on exact memory write.

The number of indices in I_i is $|G|d_G$, where d_G is the density of G. Since h_0 is data-independent, part 1 in Problem 3.1 can be formulated as: given $|G|d_G$ balls, we would like to toss them into n bins with the universal hash function h_0. Taking the results from [37], the maximum load of the bins is at most $\frac{|G|d_G}{n} + \Theta(\sqrt{|G|d_G \log n/n})$ with probability at least $1 - o(1)$. Recall the definition of the imbalance ratio of push in Definition 3.6:

$$Push_{h_0}^n = \max_{i,j \in [n]} \frac{n|I_i^j|}{|I_i|}.$$

Because $\max\{|I_i^j|\} \leq \frac{|G|d_G}{n} + \Theta(\sqrt{|G|d_G \log n/n})$, we have

$$\begin{aligned} Push_{h_0}^n &\leq \frac{|G|d_G + \Theta(\sqrt{|G|d_G n \log n})}{|G|d_G} \\ &= 1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G}}), \end{aligned}$$

with probability at least $1 - o(1)$. Thus, we finish the proof of the first part.

Since h_0 is data-independent, part 2 in Problem 3.1 can be formulated as: given $|I| = |G|d_G^n$ balls, we would like to toss them into n bins with the universal hash function h_0. The maximum load of the bins is at most $\frac{|G|d_G^n}{n} + \Theta(\sqrt{|G|d_G^n \log n/n})$

with probability at least $1 - o(1)$. Recall the definition of the imbalance ratio of pull in Definition 3.6:

$$Pull^n_{h_0} = \max_{i \in [n]} \frac{n|I_i^{'}|}{|I|}.$$

Because $\max\{|I_i^{'}|\} \leq \frac{|G|d_G^n}{n} + \Theta(\sqrt{|G|d_G^n \log n/n})$, we have

$$\begin{aligned} Pull^n_{h_0} &\leq \frac{|G|d_G^n + \Theta(\sqrt{|G|d_G^n n \log n})}{|G|d_G^n} \\ &= 1 + \Theta(\sqrt{\frac{n \log n}{|G|d_G^n}}), \end{aligned}$$

with probability at least $1 - o(1)$. Thus, we finish the proof of the second part. □

Because $n \log n$ is orders of magnitude smaller than $|G|d_G$ and $|G|d_G^n$, Algorithm 3.2 performs a very good approximation to the exact solution of Problem 3.1 for both push and pull operations, achieving load-balanced communications among workers and servers. As shown in Section 3.4.3, its practical imbalance ratio is always less than 1.1 for the four DNN models we study in the section. Note that we impose no assumptions on data distributions and only just use the property of universal hashing defined on positive integers. Hence, Algorithm 3.2 obtains a general theoretical guarantee for different distributions of non-zero gradients in DNN training.

No information loss. One may be concerned that two indices can be hashed to the same available location at the same time, leading to information loss. Fortunately, the probability of this special case is negligible with the probability less than 10^{-5} in our implementation on GPUs. Zen can use a write-and-read mechanism to check this collision. After memory writing, a thread reads the value stored in the memory. If the value is not what it writes, this thread takes a rehash.

Negligible extraction overhead. Thanks to multiple hashing functions and the serial memory, Algorithm 3.2 can achieve no information loss with very small memory size. The incurred overhead to extract the indices from the memory after hashing (Line 20 in Algorithm 3.2) becomes negligible.

Strength in parallelizable computing. Because the computations for different indices are independent of each other, it enables Algorithm 3.2 to use multiple threads

to hash and rehash them. Although the indices written in the memory are in a random order, there is no need to sort them because their orders have no effect on the aggregated results.

Hash consistency among workers. Algorithm 3.2 determines the partition p of each index with h_0. To ensure that the same index from different workers can be sent to the same server, Zen allocates the same h_0 to all the workers.

Zen can support different optimizer, such as SGD [175], Adam [56], and AdaGrad [70]. The computation of an optimizer has two steps: gradient synchronization and parameter update. Zen decouples the two steps following BytePS [94]. It aims to minimize the communication time of gradient synchronization. After sparse tensors are synchronized, all GPUs have the same aggregated gradients, with which they can update the parameters of the DNN model individually.

3.3.4 Minimizing the Indices Overhead with Hash Bitmap

There are several sparse formats to represent sparse tensors for their synchronization. Unfortunately, none of them can minimize the overhead incurred by the indices for non-zero gradients. We assume the data type of gradients is FP32.

• **COO.** It is efficient with a low tensor density [217, 115]. However, it doubles the traffic volume and becomes inefficient for a high density. As shown in Figure 3.1b, tensors get denser after aggregation. For example, the average tensor density of BERT increases from 1.06% to 40.8% after the aggregation of sparse tensors from 128 GPUs. Theoretically, transmitting sparse tensors in Pull can reduce the traffic volume by 2.5× compared to transmitting dense tensors, but the reduction shrinks to 1.2× due to the indices for non-zero gradients.

• **Tensor block.** It is used in OmniReduce [74]. A dense tensor is split into blocks of gradients and only non-zero blocks are transmitted. However, it is also inefficient when the tensor density is high. When splitting a tensor with high density into tensor blocks (e.g., each block has 256 gradients), most of them have at least one non-zero gradient and become non-zero tensor blocks. The communication scheme has to transit almost all the gradients.

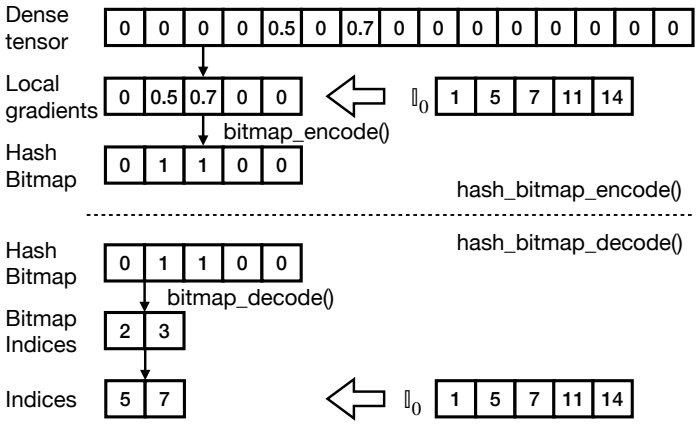


Figure 3.10 : An illustration of the hash bitmap.

• **Bitmap.** It only needs one bit to indicate whether a gradient is zero or not. Unfortunately, a straightforward bitmap still incurs non-negligible traffic volume to identify non-zero gradients. When the dense tensor G is evenly partitioned, the indices of non-zero gradients in each server are in a sub-range of $[1, |G|]$, where $|G|$ is the number of gradients in G. For example, when $|G| = 15$ and there are three servers, the index range in Server i is $[5i + 1, 5(i + 1)]$. The extra bitmap size required to represent the indices of non-zero gradients in each server is $|G|/n/32$ when the data type of gradients is FP32. The total bitmap size received by each worker is constantly $|G|/32$. When the tensor size of G is 856MB, which equals the embedding table size in DeepFM, the total bitmap size is 27MB. Although Algorithm 3.2 enables balanced communications, the non-zero gradients in each server are randomly distributed in the whole range. If we still use a bitmap to represent the indices, the extra bitmap size in each server is $|G|/32$ and the total bitmap size received at each worker becomes $n|G|/32$, which linearly increases with the number of servers. When there are 16 servers, the total bitmap size is 428MB.

Hash Bitmap. We develop a *hash bitmap* for Zen to minimize the overhead to represent indices for non-zero gradients in Pull. Given a dense tensor G and h_0 in Algorithm 3.2, the set of indices $\mathbb{I}_i = \{idx \in [1, |G|] \mid h_0(idx) = i\}$ in each worker that should be pushed to Server i is determined, though it is not in a continuous range. Since $\mathbb{I}_i$ and $\mathbb{I}_j$ are disjoint when $i \neq j$, it provides an opportunity to construct the

Algorithm 3.3 The hash bitmap

Input: G is a dense tensor. $\mathbb{I}_i = \{idx \in [1, |G|] \mid h_0(idx) = i\}$, where h_0 is defined in Algorithm 3.2.

30 **Function** `hash_bitmap_encode`(G, $\mathbb{I}_i$)**:**
31 $local\_gradients = G[\mathbb{I}_i]$
32 $hash\_bitmap =$ `bitmap_encode`($local\_gradients$)
33 **return** $hash\_bitmap$
34 **Function** `hash_bitmap_decode`($\mathbb{I}_i$, $hash\_bitmap$)**:**
35 $bitmap\_indices =$ `bitmap_decode(hash_bitmap)`
36 $indices = \mathbb{I}_i[bitmap\_indices]$
37 **return** $indices$

bitmap based on $\mathbb{I}_i$, rather than the whole range.

Figure 3.10 illustrates how the hash bitmap works for $\mathbb{I}_0$ with $|G| = 15$ and three servers. The indices for the two non-zero gradients are $\{5, 7\}$. `hash_bitmap_encode()` is used to encode the indices. Given a dense tensor G, it first extracts the local gradients according to the indices in $\mathbb{I}_0$. It then encodes the local gradients into a bitmap. Because the second and the third gradients are non-zero, the second bit and the third bit in the hash bitmap are 1 and the other bits are 0. `hash_bitmap_decode()` is used to decode the hash bitmap to a set of indices. It first decodes a hash bitmap to the bitmap indices, which are the indices of 1. For example, because the second and the third bits in the hash bitmap are 1, the bitmap indices are $\{2, 3\}$. It then uses the bitmap indices as the indices to extract the corresponding values in $\mathbb{I}_0$ as the global indices for non-zero gradients. In this example, the values are $\{5, 7\}$, which are exactly the indices for the two non-zero gradients. The pseudocode is shown in Algorithm 3.3.

The function `hash_bitmap_encode()` is invoked at each server, which then broadcasts the hash bitmap to all the workers. After each worker receives the hash bitmaps from all the servers, it invokes `hash_bitmap_decode()` to decode the hash bitmaps to the indices with the corresponding $\mathbb{I}_i$. Note that $\mathbb{I}_i$ is computed and sorted offline and it remains unchanged for the same h_0 in both servers and workers.

Theorem 3.3. *In Pull of Zen, the total hash bitmap size received at each worker*

from all servers is constantly $|G|/32$.

Proof. Suppose there are n servers. The set of indices that should be pushed to Server i is $\mathbb{I}_i = \{idx \in [1, |G|] \mid h_0(idx) = i\}$. With `hash_bitmap_encode()`, the size of the hash bitmap encoded at Server i is $|\mathbb{I}_i|/32$. Because each worker needs to receive the hash bitmap from all the servers, the total size is $\sum_{i=0}^{n-1} |\mathbb{I}_i|/32 = |G|/32$. □

Zen still uses COO to represent sparse tensors in Push due to the low tensor density. The benefit of replacing COO with hash bitmap is limited.

3.4 Evaluation

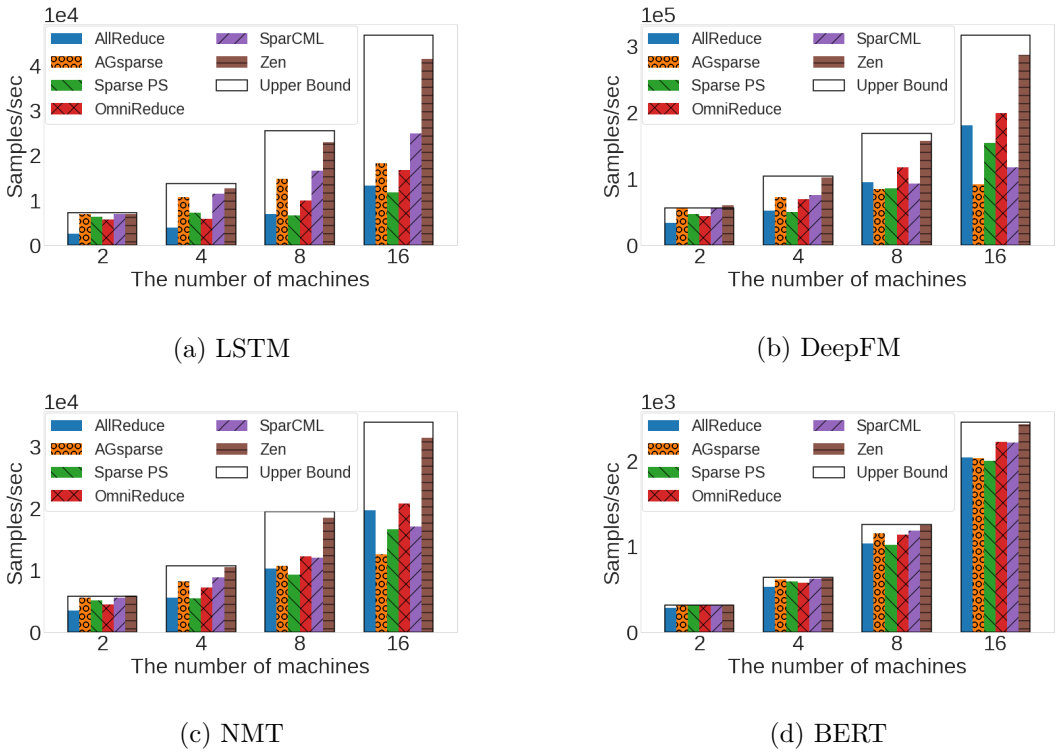
3.4.1 Experimental Setup

Testbeds. There are two testbeds in the evaluation. The first testbed contains 16 GPU machines equipped with NVLink and the machines are connected by a 25Gbps network with TCP/IP. Each machine has 8 NVIDIA V100 GPUs (32 GB GPU memory) and 2 CPUs/48 cores (Intel Xeon 8260 at 2.40GHz). The second testbed contains 16 GPU machines equipped with NVLink and the machines are connected by a 100Gbps network with RDMA. Each machine has 8 NVIDIA A100 GPUs (40 GB GPU memory) and 4 CPUs/256 cores (AMD EPYC 7742 at 2.25GHz). Each machine runs Ubuntu 20.04 and the software environment includes CUDA-11.0, PyTorch-1.8.0, Horovod-0.22.1, NCCL-2.7.8, and CuPy-11.0.

Workloads. We use four popular DNN models for both recommendation systems (DeepFM [83]) and language processing (LSTM [98], NMT [120], and BERT [67]). Table 3.1 lists their training datasets and their batch sizes. The per-GPU batch size is kept constant as the number of GPUs increases.

Baselines. We compare Zen with AGsparse, SparCML, Sparse PS, and OmniReduce. We use `AllReduce` in Horovod [187] as Dense for the synchronization of dense tensors. We also provide the upper bound on the training throughput of DNN models (Upper Bound). This is obtained by assuming the lower bound [†] of the communication time can be achieved by leveraging the sparsity in DNN models.

[†]For any GPU involved in training, it must receive non-zero gradients in G from all other GPUs



(a) LSTM

(b) DeepFM

(c) NMT

(d) BERT

Figure 3.11 : Training throughput of DNN models with 25Gbps TCP/IP networks.

Implementation. The hash function we use in Algorithm 3.2 is MurmurHash[32]. We only need to set the seeds for MurmurHash to generate different hash functions. At the beginning of training, Zen generates a set of random seeds and then broadcasts the seeds to all the GPUs to ensure that they have the same set of hash functions in Algorithm 3.2. Because NVLink is equipped in GPU machines, Zen communicates dense tensors within machines with ReduceScatter/AllGather [81, 199].

3.4.2 End-to-end Experiments

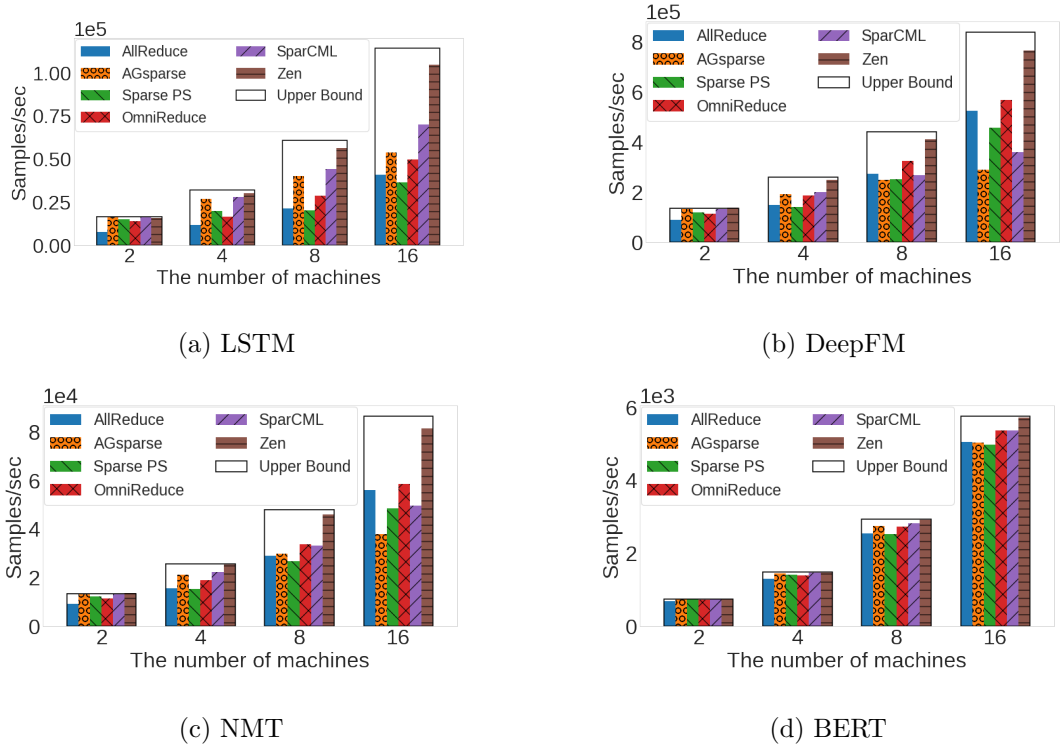
In this section, we present the end-to-end DDT efficiency of Zen on the four models and compare it with the baselines. We set $k = 3$, $r_1 = 2|G|d_G$, and $r_2 = r_1/10$ for Algorithm 3.2.

during synchronization. The overall density of sparse tensors from other $n-1$ GPUs after aggregation is d_G^{n-1}. The lower bound of communication time is $d_G^{n-1}M/B$, without transmitting the indices.

Training throughput improvement. Figure 3.11 shows the results with a 25Gbps TCP/IP network. Zen outperforms all baselines by processing more samples in a second. In LSTM model, the best baseline is SparCML. Zen achieves up to 1.67× speedup over SparCML and 3.1× speedup over AllReduce in the end-to-end training throughput. In both DeepFM and NMT with 16 machines, the best baseline is OmniReduce. Zen achieves 1.44× speedup and 1.51× speedup over OmniReduce for DeepFM and NMT, respectively. The performance of Zen for BERT is very close to the upper bound; it achieves 1.13× speedup over AllReduce and 1.07× speedup over OmniReduce. As we increase the number of machines, the benefits of Zen over SparCML and OmniReduce are enlarged, indicating Zen's great scalability. When we increase the network bandwidth from 25Gbps to 100Gbps, Zen still has great end-to-end speedups in DDT, as shown in Figure 3.12. Specifically, in LSTM model, Zen is up to 1.50× faster than SparCML, which is the best baseline. In DeepFM and NMT, Zen is up to 1.46× and 1.45× faster than AllReduce and up to 1.34× and 1.39× faster than the best baseline OmniReduce. These results demonstrate that Zen can fully leverage sparsity in DNN models to optimize their training efficiency.

Communication improvement. The performance of Zen is driven by the reduction in communication time. Figure 3.13 shows the speedups of different communication schemes compared to AllReduce with 16 machines and a 25Gbps network. The speedup of OmniReduce is up to 1.58×. We also observe the performance of AGsparse, Sparse PS, and SparCML can be even worse than AllReduce in some cases. They use COO as the sparse format. The communication time of AGsparse linearly increases with the number of machines. Sparse PS has severe imbalanced communications and it has to transmit both gradients and indices. With a high density, the sparse tensor size with COO is larger than the dense tensor size. With SparCML, the overlaps among sparse tensors can be received multiple times at each GPU. In contrast, Zen achieves 6.77× speedup for LSTM and 3.0× speedup for BERT. It outperforms SparCML and OmniReduce by up to 1.82× and 4.09×, respectively. Zen also achieves 2.10× speedup for DeepFM and 1.97× speedup for NMT.

Model accuracy. We will demonstrate that Zen can preserve the iteration-wise



(a) LSTM

(b) DeepFM

(c) NMT

(d) BERT

Figure 3.12 : Training throughput of DNN models with 100Gbps RDMA networks.

accuracy of DNN models as AllReduce because there is no information loss in Algorithm 3.2. We also take the strawman data-independent solution (refer to Section 3.3.2) as a baseline. It can achieve balanced communications at the cost of information loss due to hash collision, which is related to the used auxiliary memory size. Given a dense tensor $|G|$ with the density d_G, the memory size used in Zen is $2|G|d_G$. We vary the memory size in the strawman to evaluate the impacts on accuracy with different information loss rates. Figure 3.14 displays the test accuracy of DeepFM with different schemes. Zen and AllReduce have the same iteration-wise accuracy during training. When the memory size is $2|G|$ in the strawman (denoted as 2|G| Strawman in Figure 3.14), the information loss rate is around 9% and the training accuracy has a noticeable drop compared to training with AllReduce. Increasing the memory size to $8|G|$ (denoted as 8|G| Strawman) can reduce the rate to around 1% and it almost has no harm on the accuracy. However, it can lead to costly overhead to extract the indices from the memory. The communication time

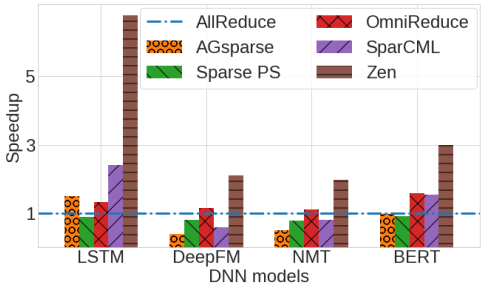


Figure 3.13 : Communication speedups for embedding layers in four DNN models compared to AllReduce.

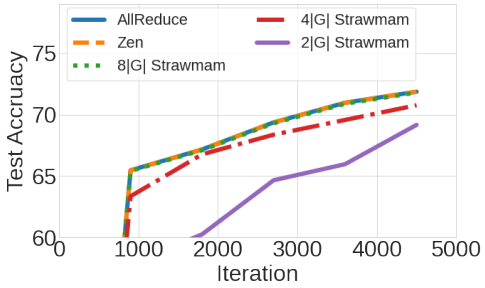


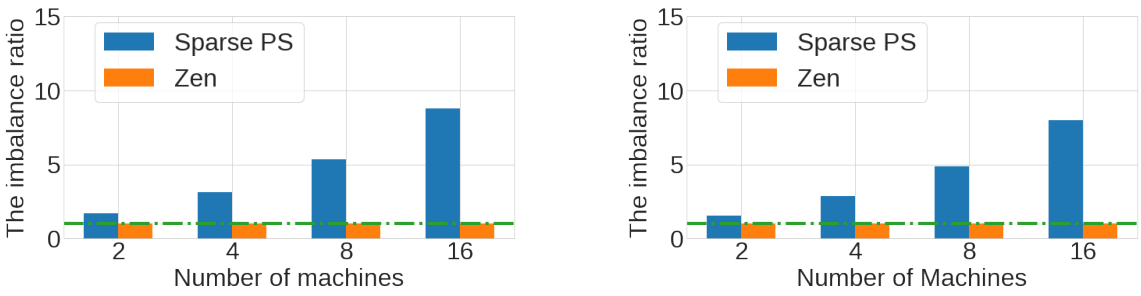
Figure 3.14 : DeepFM Accuracy with different schemes. We set the strawman with different memory sizes.

is around 60ms, but the extraction overhead is around 45ms with A100 GPUs. In contrast, the extraction overhead in Zen after hashing is only around 2ms thanks to the small memory size.

3.4.3 Microbenchmarks

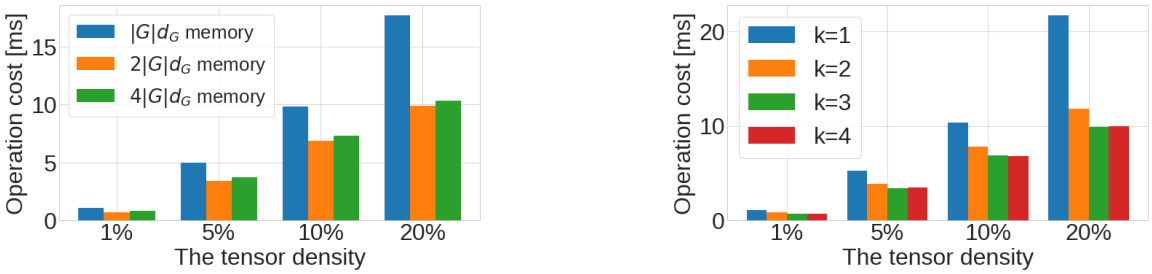
Guaranteed load balance. Figure 3.15 presents the imbalance ratio of DeepFM using Sparse PS and Zen, respectively. Figure 3.15a shows that the imbalance ratio of Push with Sparse PS is severe due to the skewed distribution of non-zero gradients and it gets worse with more machines. It leads to imbalanced communications among servers and one of the servers has to receive most the of non-zero gradients. Similarly, the imbalance ratio of Pull with Sparse PS is also very high. In contrast, applying Zen can significantly reduce the imbalance ratio in both Push and Pull. Zen can always keep the imbalance ratio smaller than 1.1, regardless of the number of machines, as shown in Figure 3.15b. This conclusion is consistent across different DNN models. It demonstrates that Zen guarantees well-balanced communications in DDT.

A study on parameters for Algorithm 3.2. We simulate a tensor with size 214M (same as the embedding gradients in DeepFM) and vary its density to perform a study on both r_1 and k in Algorithm 3.2. We first study parameter r_1. We set $r_2 = r_1/10$ and $k = 3$. As shown in Figure 3.16a, when we increase r_1 from $|G|d_G$ to $2|G|d_G$, there is a notable reduction in the operation cost because the larger memory



(a) Push (b) Pull

Figure 3.15 : The imbalance ratio of DeepFM in Pull and Push.



(a) Different memory sizes. (b) Different number of rehash.

Figure 3.16 : The computation overhead of Algorithm 3.2.

size increases the probability of successful parallel writing and reduces the workload of serial writing. But when we further increase r_1 from $2|G|d_G$ to $4|G|d_G$, it leads to a higher computation time. There are two reasons behind this effect. Firstly, the workload of serial writing is already low with $2|G|d_G$ memory. Further increasing r_1 only marginally helps reduce the computation overhead. Secondly, a larger memory size can increase the computation overhead to extract the indices (see Algorithm 3.2) and thus degrades the overall performance. Figure 3.16b shows the computation costs versus k when we use $2|G|d_G$ memory. Increasing k from 1 to 3 can help reduce the operation cost as it alleviates the serial writing workload, but $k = 3$ and $k = 4$ have very similar operation costs.

The hash bitmap. We demonstrate the effectiveness of the hash bitmap to represent the indices of non-zero gradients. Figure 3.17 shows the tensor data size with different sparse formats. The sizes are normalized to the dense tensor and there are

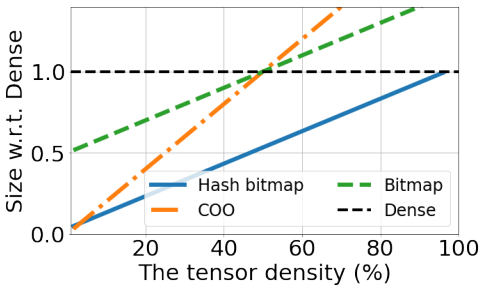


Figure 3.17 : The effectiveness of the hash bitmap.

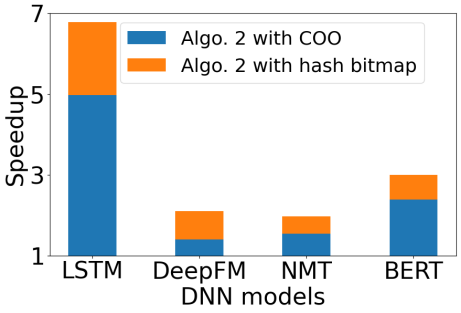


Figure 3.18 : The performance breakdown of Zen.

16 servers. The tensor density is the total density of all servers after aggregation. The gap between the hash bitmap and COO increases with the tensor density. It also significantly outperforms the bitmap. In addition, the hash bitmap can still reduce the traffic volume with a density of 95% compared to the dense tensor, but the bitmap and COO cannot save the volume when the density is greater than 50%. The performance of tensor blocks varies with the distribution of non-zero gradients. For some sparse tensors in the four DNN models we study, it can transmit higher traffic volume than COO since a non-zero block has more zero gradients than non-zero gradients.

Zen's performance breakdown. We also break down the performance of Zen by Algorithm 3.2 and the hash bitmap format. Figure 3.18 illustrates the speedup breakdown over AllReduce with 16 machines and a 25Gbps network. It can be seen that the primary performance benefits of Zen come from Algorithm 3.2, with the hash bitmap format providing noticeable additional benefits. For example, when the data format is COO after applying Algorithm 3.2, the speedup is 4.97× and 2.39× for LSTM and BERT, respectively. Replacing COO with the hash bitmap can further improve the speedups by 36% and 26%.

Chapter 4

Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies

Zen utilizes the sparsity in gradient tensors to minimize communication time for gradient synchronization and it can greatly reduce the communication time and improve training throughput when tensors have high sparsity. When tensors only have low sparsity in the training of some DNN models, gradient compression (GC) algorithms are applied to shrink the communicated data size by compressing gradient tensors. Although these algorithms reduce communication time and thus theoretically increase training throughput, they present practical challenges due to additional computation overheads when applying GC algorithms to DDL. In this chapter, we will propose a gradient compression framework named Espresso that optimizes the benefits of GC algorithms for compression-enabled DDL.

4.1 Introduction

The growing concern of communication bottlenecks in DDL due to the exacerbating tension between computation and communication in DDL has motivated numerous works, such as wait-free back-propagation mechanism [231, 112], priority-based scheduling [160, 91, 85], and optimized aggregation algorithms [94, 57, 187]. However, even with the latest highly-optimized BytePS [94] which incorporates these state-of-the-art approaches, communications for gradient synchronization still account for 42% and 49% of the total training time of GPT2 [164] and BERT-base [67] with 64 NVIDIA V100 GPUs in 8 machines connected by a 100Gbps Ethernet network.

Gradient compression (GC) algorithms [192, 27, 115, 219, 220, 186, 100, 28] have

a great potential to address the communication bottlenecks in DDL by saving up to 99.9% of the gradient exchange while preserving the training accuracy and convergence [223, 192, 93]. However, the training speedups of DDL with GC are only modest because of the costly compression operations. For example, applying GC to the aforementioned GPT2 training only achieves a 1.15× speedup. This motivates us to revisit GC from the system perspective to fully unleash its benefits for DDL.

Applying GC to a DNN model can reduce the communication time, but it also incurs additional compression overheads. The training throughput of compression-enabled DDL is determined by the *compression strategy*, which refers to the compression-related decisions for each tensor in a DNN model, such as whether to compress, the type of compute resources (e.g., CPUs or GPUs) for compression, and the communication schemes for compressed tensors. DDL typically involves both communications inside a machine and across machines. Therefore, another decision is whether to apply GC to intra- or inter-machine communication or both.

Unfortunately, it is very challenging to make these decisions because of the intricate interactions among tensors. Therefore, the first research question we have to answer to unleash the benefits of GC is how to express any possible compression strategies and the corresponding interactions among tensors for any DDL training job. Because of the extremely large search space, even if all the strategies and interactions are available, the time to find the optimal one can be prohibitive. Hence, the second research question is how to analyze the interactions among tensors to quickly select a near-optimal compression strategy.

In this chapter, we propose Espresso to answer these two questions in order to maximize the benefits of GC. We make the following contributions.

• We develop a decision tree abstraction for the compression strategy and empirical models for the time of tensor computation, communication, and compression to answer the first question. The abstraction can express any possible compression options of any tensor regardless of different tensor sizes and GC algorithms. Based on the abstraction, Espresso can express any compression strategies of any DDL training jobs. The empirical models enable Espresso to derive the timeline of tensor computa-

tion, communication, and compression of all tensors in a DNN model, and thus their intricate interactions with any compression strategy.

• We propose a compression decision algorithm for quickly selecting a near-optimal compression strategy to answer the second question. Espresso analyzes the interactions among tensors to eliminate a large number of suboptimal compression strategies. Based on the analysis, Espresso proposes a prioritization method for applying GC to tensors to maximize the benefits, and considers the overlapping time among tensor computation, communication, and compression to make compression decisions for each tensor. Because of different performance trade-offs of GPUs and CPUs for GC, Espresso finds a provably optimal solution to offload compression from GPUs to CPUs to minimize the resource contentions with tensor computation.

• We implement a fully featured system for Espresso. We implement both GPU and CPU compression libraries. We also implement communication libraries to support different communication schemes in both intra- and inter-machine communications. Experimental evaluations demonstrate that with 64 GPUs, Espresso can improve the training throughput by up to 269% compared with BytePS. It also outperforms the state-of-the-art compression-enabled system (i.e., HiPress [38]) by up to 77% across representative DNN training jobs. Moreover, the computational time needed by Espresso to select the compression strategy is measured in milliseconds, and the performance difference between the selected strategy and the optimal strategy is only a few percent.

4.2 Background

4.2.1 Computation and Communication Tension in DDL

Because DDL typically employs multiple machines and each machine has multiple GPUs, it involves both intra-machine and inter-machine communication. *Hierarchical communication* (as shown in Figure 4.1) is widely applied in DDL frameworks [94, 187, 112, 50] because the intra-machine network is usually faster than the inter-machine network. There are three *phases* for gradient synchronization in hi-

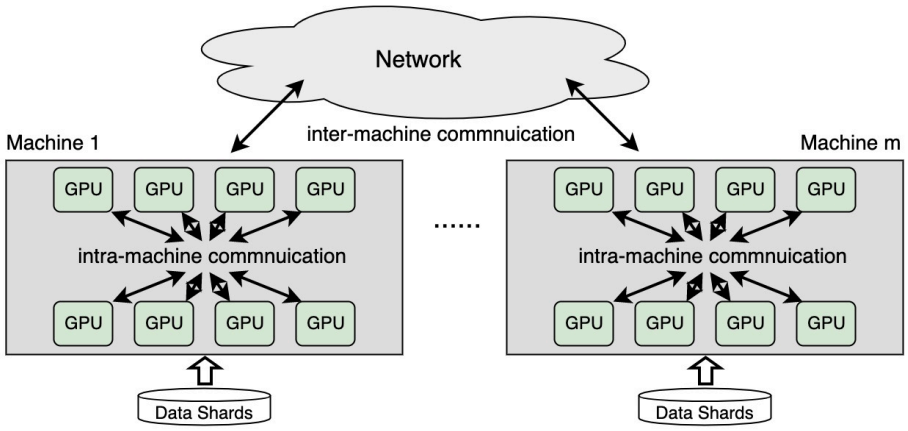


Figure 4.1 : Hierarchical communication in DDL.

erarchical communication: 1) the gradients are first aggregated among GPUs within one machine; 2) they are then aggregated across machines; and 3) the aggregated gradients are communicated within one machine again to ensure that all GPUs have the same synchronized results. *Flat communication*, i.e., all GPUs join the same collective operation and have only one communication phase, is also supported in some frameworks [187, 112].

As discussed in Section 2.3, there exists an exacerbating tension between computation and communication in DDL. Single precision (FP32) is a common floating point format representing the weights and gradients in deep learning. When gradients are communicated in FP32 for synchronization, it can lead to costly communication time and thus poor scalability in DDT. To illustrate, we trained real-world DNN models on BytePS-0.2.5 [94], a highly-optimized DDL framework, with 64 NVIDIA V100 GPUs (8 GPUs per machine) and a 100Gbps inter-machine Ethernet network. We measure the scaling factor [234, 74], which is defined as $\frac{T_n}{nT}$, where T is the training throughput of a single device and T_n is the throughput of DDL with n devices. BytePS only achieves the scaling factors of 0.58 and 0.51 for the training of two representative and popular DNN models, GPT2 and BERT-base, with NVLink 2.0 for GPU-to-GPU interconnection, as shown in Table 4.1. To put this into context, the training time of

Model	Networks	FP32	GC with GPU	GC with CPU
GPT2	NVLink, 100Gbps	0.58	0.67 (+15%)	0.64 (+10%)
BERT-base	NVLink, 100Gbps	0.51	0.55 (+8%)	0.61 (+20%)
LSTM	PCIe, 25Gbps	0.46	0.43 (−6%)	0.42 (−9%)

Table 4.1 : The scaling factors of three popular DNN models with 64 GPUs (8 GPUs per machine) and hierarchical communication. FP32 is the training without GC.

BERT-base is about 1200 GPU hours under ideal linear scaling [147], but in practice, it will take 2350 GPU hours with 64 GPUs due to the communication time caused by gradient synchronization. Thus, DNN practitioners have to spend nearly twice the amount of money on training because the cost linearly increases with the required GPU hours [30].

When network bandwidth in GPU clouds has not kept pace with the improvements in computation, an alternative is to shrink the communicated traffic volume by applying gradient compression.

4.2.2 Gradient Compression

Many gradient compression (GC) algorithms have been proposed in the machine learning community. *Sparsification* and *Quantization* are the two main types of GC algorithms. Sparsification selects a subset of the original stochastic gradients for synchronization [192, 27, 115] and it can save up to 99.9% of the gradient exchange while maintaining model accuracy [115]. Quantization decreases the precision of gradients; gradients in single-precision floating-point format (FP32) are mapped to fewer bits, such as 8 bits [66], 2 bits [220], and even 1 bit [186, 100, 42] to reduce the communicated traffic volume by up to 96.9%. There are other types of gradient compression algorithms, such as low-rank decomposition [207, 211] and FFT-based compression [212]. Such compression algorithms have been theoretically proven and/or empirically validated to preserve the convergence of model training and impose negligible impact on model accuracy when combined with error-feedback mechanisms [186, 223, 192, 115, 93]. The industry is adopting GC because of its

great potential to alleviate the communication bottleneck in DDL. The efforts from Meta, AWS, and ByteDance to bring GC to mainstream DNN systems have begun recently [131, 29, 240]. However, the scalability improvement of DDL via GC has been still poor.

4.3 Challenges of Applying GC to DDL

We first define some key terms.

• **Tensor computation** is the computation of a tensor during backward propagation.

• **Communication time** is the wall-clock time for communication. It is denoted as τ_{comm}.

• **Communication overhead** is the communication time that cannot overlap with the tensor computation of any tensors. It is denoted as o_{comm}.

• **Compression time** is the wall-clock time to perform compression and decompression operations on devices, e.g., GPUs or CPUs. It is denoted as τ_{comp}

• **Compression overhead** is the compression time that cannot overlap with either tensor computation or communication of any tensors. It is denoted as o_{comp}.

Although GC can reduce τ_{comm}, its compression overheads can dramatically dilute the benefits gained from the reduced communication time. To demonstrate this, we apply a popular sparsification algorithm, DGC [115], to the aforementioned GPT2 training and a representative 1-bit quantization algorithm, EFSignSGD [100], to BERT-base training. The compression rate of DGC is 1%, i.e., only 1% of gradients are exchanged during synchronization. Tensors are compressed with GPUs [38] or CPUs [240] in separate experiments. Compression with GPUs is typically faster than compression with CPUs [38], but it competes for the GPU resources with training [26]. As shown in Table 4.1, GC only achieves up to 20% training speedup, which is on par with the findings in prior works [227, 38, 26]. In fact, GC can harm performance in some situations. To illustrate, we apply DGC with 1% compression rate to the training of LSTM [129] on 64 V100 GPUs with PCIe 3.0 ×16 as the intra-machine

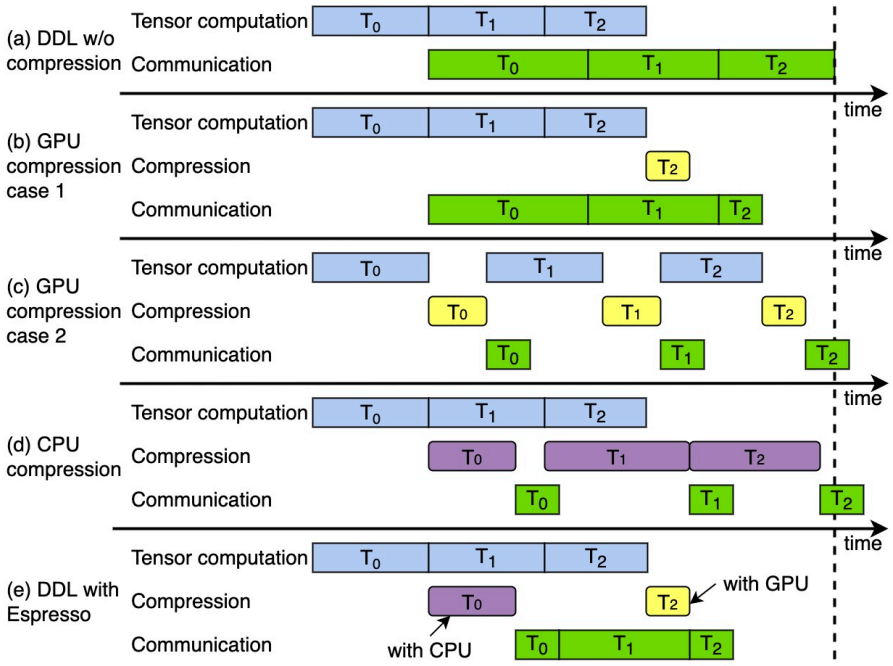


Figure 4.2 : A DDL example with different compression strategies. (a) is the baseline; (b) reduces the iteration time, but it is not optimal; (c) and (d) harm the performance; (e) is our solution and achieves optimal performance. The communication and compression overheads depend on the interactions among tensors. The decompression operations are omitted.

network and 25Gbps inter-machine Ethernet.* As listed in Table 4.1, GC slows down training by up to 9%.

In the following, we will explain the root reasons why it is challenging to obtain large benefits from GC for DDL.

*NVLink 2.0 gives every GPU in total 1.2Tbps GPU-GPU bandwidth, but PCIe 3.0 ×16 only provides ∼100Gbps bandwidth [94]. PCIe-only GPU machines are common in GPU clusters that have 25Gbps Ethernet [94, 6, 80, 169].

4.3.1 Root Reasons of the Challenges

The choice of compression strategies determines the iteration time of compression-enabled DDL. Figure 4.2 is an example that shows the timelines of tensor computation, communication, and compression of DDL with different compression strategies. Figures 4.2(a) is the baseline without GC and it illustrates the tensor computation time (blue boxes) and communication time (green boxes) of all tensors, i.e., T_0, T_1, and T_2. Figure 4.2(b) compresses T_2 with GPUs and it reduces the iteration time. Figures 4.2(c) and (d) compress the three tensors with GPUs and CPUs, respectively, but unfortunately, they both harm the performance of DDL. Figures 4.2(e) shows the optimal compression strategy with Espresso.

It is challenging to find the optimal compression strategy. Applying GC to DDL is essentially to reduce the communication overheads at the cost of the compression overheads. The optimal compression strategy maximizes the difference between the reduced communication overheads and the incurred compression overheads. There are three root reasons for the challenges.

Reason #1. *It is hard to quantify the communication and compression overheads because of the intricate interactions among tensors.*

Communication may or may not overlap with tensor computation. The overlapping times of different tensors vary. For example, in Figure 4.2(a), T_0's o_{comm} is zero because its communication is fully overlapped with tensor computation, but T_2's o_{comm} is its communication time because it has no overlap with tensor computation. Moreover, the overlapping time of one tensor can vary under different compression strategies. For example, in Figure 4.2(a), T_1's communication partially overlaps with T_2's tensor computation. However, in Figure 4.2(c), after compression, T_1's communication can completely overlap with T_2's tensor computation. Furthermore, in Figure 4.2(d), T_1's communication has no overlap with the computation of other tensors. Hence, it is difficult to quantify the communication overhead of each tensor.

Compression may or may not overlap with tensor computation and communication. How much τ_{comp} can be overlapped highly depends on the strategy. For instance, in Figure 4.2(b), T_2's GPU compression fully overlaps with T_1's communi-

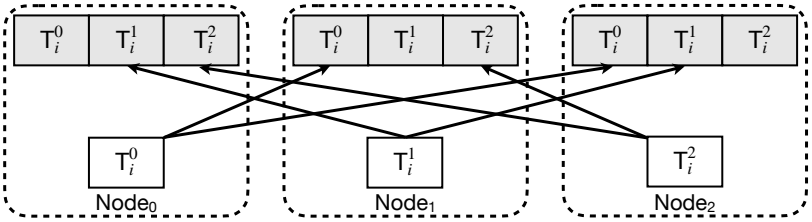


Figure 4.3 : An indivisible scheme. T_i^j is the tensor T_i on Node j. Each node retrieves tensors from other nodes.

cation. In Figure 4.2(d), T_1's CPU compression partially overlaps with T_2's tensor computation. In Figure 4.2(c), the three GPU compressions are fully exposed. Hence, it is difficult to quantify the compression overhead.

Only considering τ_{comm} and τ_{comp} for the decision of compression strategies can harm the performance. Figure 4.2(c) maximizes the difference between the reduced communication time and the compression time by compressing the three tensors. However, because GPU compression competes for compute resources with tensor computation, it delays training and prolongs the iteration time instead. Hence, we must consider o_{comm} and o_{comp} to determine compression strategies for compression-enabled DDL.

Reason #2. *It is hard to choose the right communication schemes for compressed tensors because of Reason #1.*

There are two types of communication schemes for compressed tensors: *indivisible schemes* and *divisible schemes*. We first consider the case that there are N machines in DDL and each machine has a single GPU. An indivisible scheme has only one communication operation, as shown in Figure 4.3. Once a tensor is compressed, each node (e.g., GPU or CPU) broadcasts its compressed tensor to other nodes. After communication, each node decompresses these compressed tensors and aggregates them. In contrast, a divisible scheme has two communication operations, as shown in Figure 4.4. Tensors are first compressed and partitioned into n parts, where $1 \leq n \leq N$. The j_{th} node receives the j_{th} part from other nodes. It then

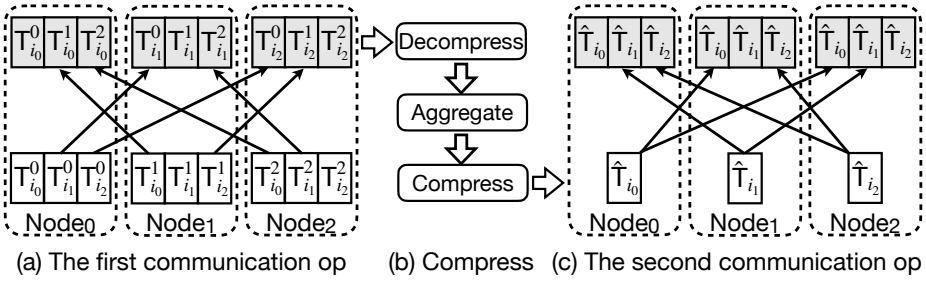


Figure 4.4 : A divisible scheme. In (a), T_i^j is partitioned into 3 parts, i.e., $T_{i_0}^j$, $T_{i_1}^j$, and $T_{i_2}^j$. The first communication operation (op) is a shuffle. After Node j receives the j_{th} part from other nodes, it decompresses and aggregates them. It then compresses the aggregated tensor and obtains $\hat{T}_{i_j}$ for the second communication op.

performs decompression, aggregation, and the second compression operation.[†] After that, it broadcasts the compressed tensor to other nodes. After communication, each node decompresses these compressed tensors and aggregates them.

It is hard to decide between indivisible and divisible schemes for GC. Compared to indivisible schemes, divisible schemes have lower communication time and higher compression time due to the two compression and decompression operations. As shown in Figures 4.5(a), GC with a divisible scheme outperforms GC with an indivisible scheme. However, in Figure 4.5(b), T_0's communication overlaps with T_1's tensor computation and an indivisible scheme outperforms a divisible scheme for GC. Thus, the decision of communication schemes depends on tensor interactions.

Reason #3. *It is hard to determine whether to apply GC to intra- or inter-machine communication or both to alleviate communication bottleneck because of Reasons #1 and #2.*

DDL can involve both intra- and inter-machine communications. We now consider the case that there are N machines and each machine has k GPU, where $k > 1$, as shown in Figure 4.1. It has intra- and inter-machine communications, and both can become the performance bottleneck.

[†]In some cases it can skip these three operations to begin the next communication directly.

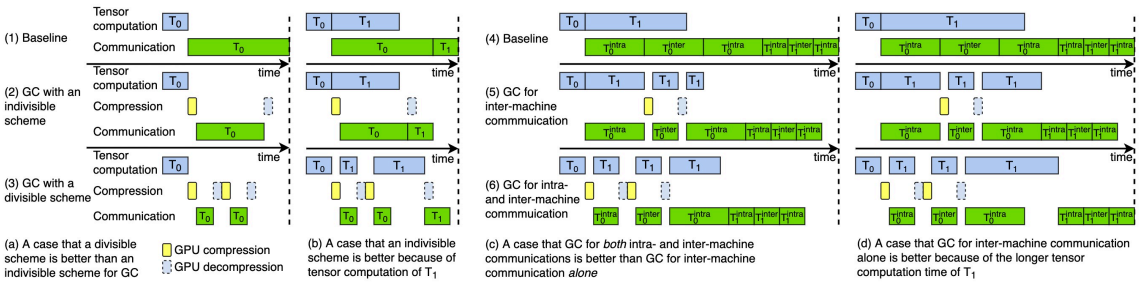


Figure 4.5 : (a) and (b) show that the choice of communication schemes depends on the interactions among tensors. Only T_0 is compressed. (c) and (d) show that the decision to apply GC to inter-machine communication alone or to both intra- and inter-machine communications also depends on the interactions among tensors. T_i^{intra} and T_i^{inter} are T_i's intra- and inter-machine communications.

Whether to apply GC to intra- or inter-machine communication or both depends on the interactions among tensors. If a tensor is only compressed for inter-machine communication, intra-machine communication can still be a performance issue. Figure 4.5(c) shows that applying GC to intra-machine communication can further reduce the iteration time. However, if T_1 has a longer computation time, it can overlap more time with T_0's communication, as shown in Figure 4.5(d). In this case, applying GC to both intra- and inter-machine communications leads to worse performance than applying it to inter-machine communication alone.

This decision also depends on the chosen communication schemes. Because both intra- and inter-machine communications need to choose from indivisible or divisible schemes, the difficulties in determining the right schemes make the decision of the compression choices even harder.

4.3.2 Research Questions

In light of the three root reasons, there are two research questions to answer for applying GC to DDL.

Question #1: how to express any possible compression strategies and interactions among tensors for DDL regardless of different distributions of

computation and communication time of tensors in different DNN models, different intra- and inter-machine bandwidth, and different GC algorithms?

Applying GC to a tensor must answer the following fundamental questions: Does it need compression? If so, what type of compute resources to use for its compression? After compression, what communication schemes should the compressed tensor use? If it has multiple communication phases, where to compress and decompress this tensor? The search space is huge when holistically considering these decisions. Moreover, there are typically a large number of tensors in a DNN model and the compression decisions of one tensor can impact the choices of other tensors because of their intricate interactions. The compression strategy determines the interactions among tensors, which determine the training throughput of compression-enabled DDL. Therefore, it is crucial to express any strategies and the corresponding interactions among tensors to avoid missing the opportunity to maximize the training throughput.

Question #2: how to analyze the interactions among tensor computation, communication, and compression, as well as the different performance trade-offs of GPUs and CPUs for GC, to determine a near-optimal compression strategy for DDL and to do so quickly?

It is important to derive the training timeline, as shown in Figure 4.2 and Figure 4.5, to analyze the interactions for a given compression strategy and its effect on training throughput because the timeline reveals the iteration time. Even if all compression strategies are at hand, the time complexity to derive their timelines and find the optimal strategy is exponential (§4.7.1). The searching time can be much longer than the training time, which is unacceptable. Moreover, the optimal strategy is specific to each situation depending on the DNN model, intra- and inter-machine bandwidth, GC algorithm, etc., and thus cannot be reused across situations. A successful solution to this question must develop new insights on the interactions among tensors and the different performance trade-offs with different types of compute resources for GC that can eliminate suboptimal strategies from consideration.

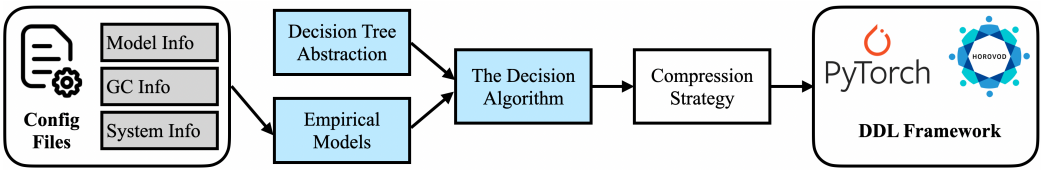


Figure 4.6 : Espresso Overview.

4.4 Espresso Overview

To maximize the training throughput of compression-enabled DDL, the core idea of Espresso is to select a near-optimal compression strategy from an extremely large search space with the following two techniques.

A decision tree abstraction to describe any compression options of any tensors as well as empirical models for the time of tensor computation, communication, and compression to express any compression strategies and interactions among tensors. The abstraction can express any type of compute resources for compression, communication schemes, and different choices to apply GC to intra- and inter-machine communications. It can also serve as the building block to describe any compression strategies of a compression-enabled DDL. The empirical models enable Espresso to derive the timeline of tensor computation, communication, and compression of all tensors, and thus their intricate interactions with any compression strategy.

An algorithm for selecting a near-optimal compression strategy with four properties. The algorithm 1) rules out tensors that certainly bring no benefits to DDL with GC based on the analysis of interactions among tensors; 2) uses a prioritization method for applying GC to tensors to maximize the benefits with the minimum number of tensors for compression; 3) determines the compression options with the compression and communication overheads based on the analysis of interactions, rather than with the wall-clock time; and 4) finds a provably optimal solution to offload compression from GPUs to CPUs.

The input of Espresso is three configuration files for the information of the DNN

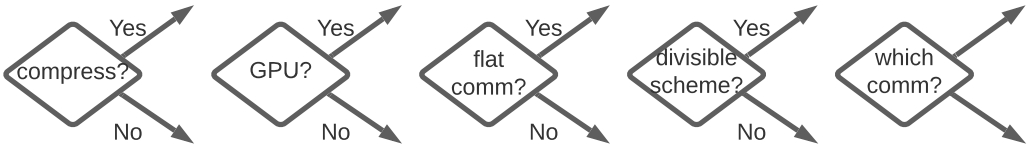


Figure 4.7 : The decision tasks. *compress?* is for Dimension 1, *GPU?* is for Dimension 2, and the other three decision tasks are for Dimension 3. The options of Dimension 4 are illustrated in Figure 4.8.

model, the GC, and the training system setting, as illustrated in Figure 4.6. The model information contains the tensor sizes and their tensor computation time. Espresso implements a GC library and the GC information specifies the GC algorithm and the compression ratio. The training system information gives the number of GPU machines, the number of GPUs in each machine, and the network bandwidth for both intra- and inter-machine communications. Espresso takes the three types of information to construct the empirical models. Based on the decision tree abstraction and the empirical models, Espresso selects a near-optimal compression strategy offline for the training job with the decision algorithm. After that, it applies the compression strategy to the DDL framework to execute the compression option for each tensor at run-time whenever their gradients are ready for communication.

4.5 The Decision Tree Abstraction

4.5.1 The dimensions of the search space

There are four dimensions that Espresso must consider to describe the search space of compression options for each tensor. The decision tasks for these dimensions are shown in Figure 4.7.

Dimension 1: compression or no compression. Because GC can incur non-negligible compression time and even harm performance, there is no need to compress all tensors. Espresso must determine the set of tensors that should be compressed to maximize the benefits of GC.

Routines	Uncompressed tensors	Compressed tensors
Indivisible schemes	Allreduce	Allgather
Divisible schemes	Reduce-scatter/Allgather	Alltoall/Allgather
	Reduce/Broadcast	Gather/Broadcast

Table 4.2 : The collective routines for synchronization.

Dimension 2: GPU or CPU for compression. Both GPUs and CPUs can be used for GC to minimize the compression overhead. Espresso must determine the set of tensors in a DNN model for GPU and CPU compression, respectively. Task Comp and Task Decomp, as listed in Table 4.3, are the action tasks to decide between GPUs and CPUs for compression and decompression operations, respectively.

Dimension 3: the communication schemes. Compressed tensors cannot use Allreduce for synchronization because their aggregation operations are not associative [26, 38, 227]. Both indivisible and divisible communication schemes can be used, while each can have more than one choice of collective routines, i.e., one collective communication operation or an operation pair. Table 4.2 lists the common collective routines used in DDL for GC [199, 3, 112]. Because tensors can be communicated without GC, Table 4.2 lists the collective routines for uncompressed tensors as well. We distinguish the two communication operations in a divisible scheme as its first and second steps. In addition, flat and hierarchical communications lead to a different number of communication phases for gradient synchronization. Therefore, this dimension requires Espresso to consider three sub-dimensions: flat or hierarchical communication, indivisible or divisible schemes, and specific collective routines for each communication phase. The decision tasks of the three sub-dimensions are shown in Figure 4.7 as *flat comm?*, *divisible scheme?*, and *which comm?*. Because both uncompressed and compressed tensors have indivisible and divisible schemes, and division schemes have two collective operations, *which comm?* then has six action tasks, as listed in Table 4.3.

Dimension 4: the compression choice. It determines where to perform compression and decompression operations. For flat communication, it has two *commu-*

Action Tasks	Description	Search space
Comp	Compression operation	{CPU, GPU}
Decomp	Decompression operation	{CPU, GPU}
Comm	Indivisible scheme for UT	{Allreduce}
Comm1	The first step of a DS for UT	{Reduce-scatter, Reduce}
Comm2	The second step of a DS for UT	{Allgather, Broadcast}
Comm_{comp}	Indivisible scheme for CT	{Allgather}
Comm1_{comp}	The first step of a DS for CT	{Alltoall, Gather}
Comm2_{comp}	The second step of a DS for CT	{Allgather, Broadcast}

Table 4.3 : The eight action tasks. UT denotes uncompressed tensors, CT denotes compressed tensors, and DS denotes divisible schemes.

nication patterns because it can choose from an indivisible or a divisible scheme. For hierarchical communication, it can choose from a divisible or an indivisible scheme for its inter-machine communication. Although it can also choose from a division scheme or two indivisible schemes for its two intra-machine communications, the former is better than the latter due to the less amount of traffic volume. Therefore, Espresso only considers division schemes for intra-machine communications in hierarchical communication. Tensors can be compressed as long as they need communication and compressed tensors can be decompressed after any communication operation. All the options for this dimension, i.e., the possible positions of Task Comp and Task Decomp in each communication pattern, are illustrated in Figure 4.8.

4.5.2 Constructing the tree

A *compression option* is a series of decision tasks that determine all the communication and compression operations of a tensor for its synchronization. These operations have orders and dependencies. There are eight action tasks (as listed in Table 4.3), but not all of them can have direct connections, i.e., a task is performed right after another. The valid connections of action tasks are omitted due to space limitations.

Tree construction. Based on the four dimensions and the valid connections of the eight action tasks, Espresso can express any possible compression options of any

tensor with a decision tree, as shown in Figure 4.8. Because the choices of GPUs or CPUs for Task Comp and Task Decomp do not impact communication tasks, we use one arrow to represent their two choices for simplicity.

There are three *pruning rules* to construct the tree. The first rule is that the following action tasks of an action task must be its valid connections. The second rule is that the communication tasks must match the correct steps. For example, Comm1 and Comm1$_{comp}$ are only valid as the first steps of divisible schemes. The third rule is that the choices of communication tasks in the first and second steps must pair. For example, if Comm1 is Alltoall, then Comm2 in this divisible scheme must be Allgather. Each path from Start to End is a valid compression option. The red crosses in Figure 4.8 are the invalid paths ruled out by these pruning rules.

There are five sub-trees illustrated in Figure 4.8 to abstract parts of the tree. Sub-tree T_1 and T_2 describe the process of the second intra-machine step with the input tensor uncompressed and compressed, respectively. Sub-tree T_3 and T_4 describe the process of inter-machine communication plus the second intra-machine step with the input tensor uncompressed and compressed, respectively. Sub-tree T_5 describes the process of the second inter-machine step plus the second intra-machine step with the input tensor uncompressed.

Expressiveness and extensibility. Because all the valid connections between decision tasks have been considered, this decision tree abstraction can cover any possible compression options. It is easy for Espresso to extend the search space by considering new communication schemes for GC [172, 74] and other types of compute resources [97, 208]. In addition, it allows users to manually add constraints to prune the decision tree to rule out undesirable compression options for their applications. For example, users can limit the number of compression operations for each tensor to avoid the accuracy loss of training models.

Compression strategies. Let $\mathcal{T} = \{T_i\}$ denote the set of tensors in a DNN model and the number of tensors in $\mathcal{T}$ is $|\mathcal{T}| = N$. $\mathcal{C}$ is the set of possible compression options. $S = \{c_j\}$ is a compression strategy for the DNN model, where $c_j \in \mathcal{C}$ is the compression option for tensor T_j.

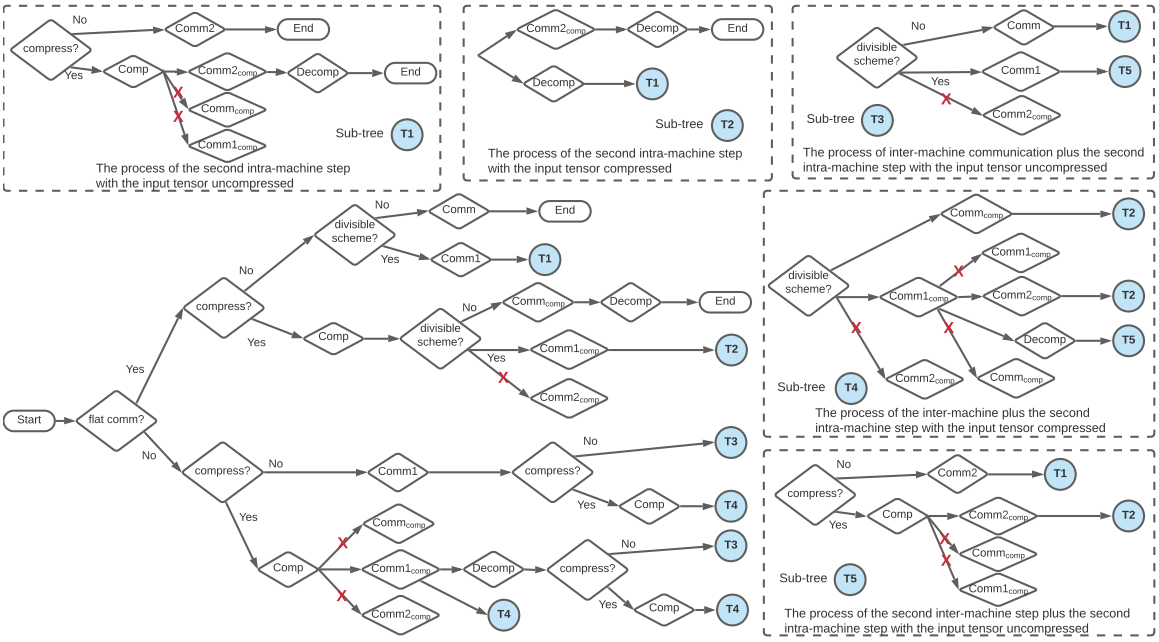


Figure 4.8 : The decision tree abstraction for the compression options. Each diamond in the tree is a decision task.

4.6 Empirical interactions among tensors

The decision tree abstraction can express any compression strategies, but it is incapable of describing the intricate interactions among tensors, which determine the choice of compression strategies for different DDL training jobs. To describe the interactions, Espresso proposes different methods to empirically model the time of tensor computation, communication, and compression, respectively.

Tensor computation. Espresso needs the computation time of each tensor. It collects execution traces of DNN training jobs without GC for 100 iterations to capture the starting and ending time of the computation of each tensor during backward propagation. Espresso then averages the computation time. It also collects information on tensor sizes.

Communication time. Espresso needs the communication time of tensors with and without GC. Given a tensor, Espresso predicts its communication time with different communication schemes and network bandwidth. The cost models follow

the model analysis in the literature [199, 148]. These communication models account for different tensor sizes, communication schemes, the number of machines and GPUs, and network bandwidth.

Compression time. Espresso also predicts the compression time of tensors with different sizes and different types of compute resources. Based on the information collected from execution traces, it can have all the possible tensor sizes as the input of compression and decompression operations. For any GC algorithm, Espresso profiles the computational time of these operations on GPUs and CPUs, respectively. It runs compression and decompression operations with different tensor sizes 100 times and then averages the results.

Empirical measurement. Espresso requires the applied GC algorithm to have deterministic compression time given a tensor size and deterministic compression ratio. To the best of our knowledge, all existing GC algorithms satisfy these requirements. It models the tensor computation for each DNN training job without GC and models the compression time for each GC algorithm. The communication time is independent of the used DNN model and the applied GC algorithm. We observe that both the measured tensor computation time and the compression time remain almost constant across runs [234, 194]. The normalized standard deviation of the measurements is less than 5%.

Expressing interactions. Given these empirical models and a compression strategy, Espresso can derive the timeline of tensor computation, communication, and compression of all tensors in a DNN model. Several timeline examples are shown in Figure 4.2. It can obtain the overlapping time of tensors and thus their interactions based on the timeline.

In the next section, we will introduce how Espresso exploits the timeline and analyzes the interactions among tensors to obtain a near-optimal compression strategy for compression-enabled DDL.

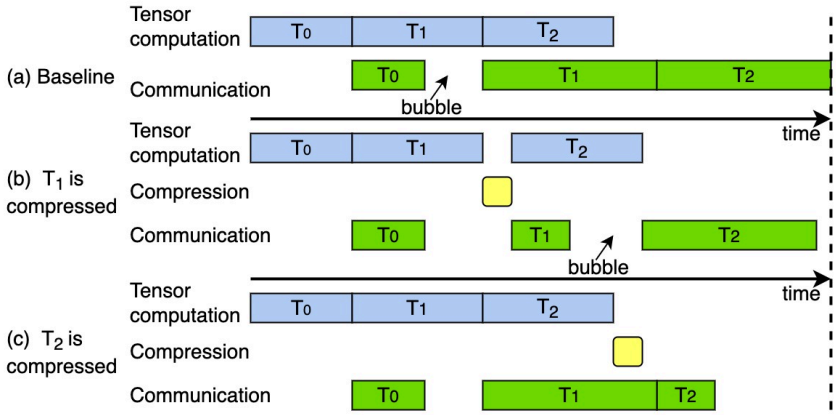


Figure 4.9 : (a) shows that tensors communicated before bubbles need no compression. T_1 and T_2 have the same size. In (b), T_1 is compressed and a new bubble is formed. In (c), T_2 is compressed and it reduces more iteration time than compressing T_1.

4.7 Espresso's Decision Algorithm

4.7.1 The optimization problem

We define the optimization problem as follows to search for the optimal compression strategy for a DDL training job.

Problem 4.1. *Given a DDL training job and a compression algorithm, how to maximize its training throughput with an optimal compression strategy?*

Let $F(S)$ be the iteration time with compression strategy S. The objective is to minimize $F(S)$ with the optimal compression strategy. The difficulty of the problem results from the overlapping time among tensor computation, communication, and compression. Given a compression strategy, Espresso can obtain the overlapping time of each tensor with other tensors. However, both CPU and GPU compression delay communications and change the overlapping time accordingly. Naively, we can enumerate possible combinations to find the optimal solution. This is not acceptable because the time complexity is $O(|\mathcal{C}|^N)$, where N could be a few hundred and $|\mathcal{C}|$ is 4341 based on the decision tree abstraction in Figure 4.8.

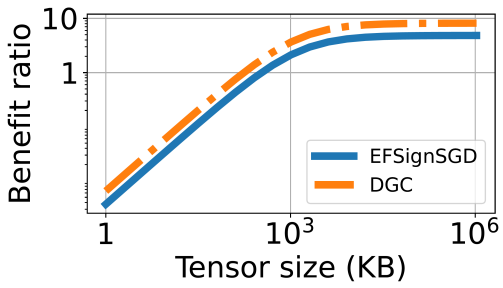


Figure 4.10 : The benefit ratio of GPU compression.

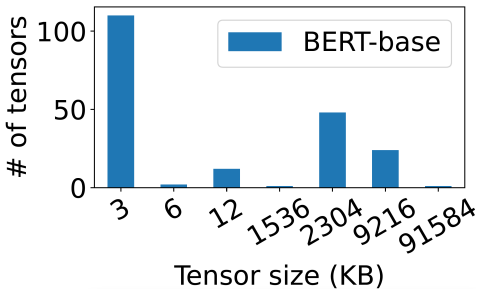


Figure 4.11 : Number of tensors with the same sizes.

4.7.2 Espresso's GPU compression

To quickly determine a near-optimal compression strategy for DDL, Espresso first considers GPU resources for GC and then offloads compression to CPUs to minimize the contention with tensor computation. There are three properties for the design of Espresso's GPU compression decision algorithm.

Property #1. The communication timeline of a DNN model can have *bubbles*, i.e., the gaps between communications of adjacent tensors. In Figure 4.9(a), there is a bubble between the communications of T_0 and T_1 because T_1 is not ready for communication when T_0's communication completes. There is no benefit to compressing tensors communicated before bubbles because reducing their communication time only widens the gaps, rather than shifts communications of tensors after bubbles to an earlier time. Compressing these tensors even harms the performance of DDL because of the resource contentions with tensor computation. We observe that half of the tensors are communicated before bubbles in the training of LSTM with 8 NVLink-based GPU machines in a 100Gbps network. Moreover, compressing particular tensors can also lead to new bubbles being formed due to the reduced communication time. For example, Figure 4.9(b) shows that a new bubble appears when T_2 is compressed. Therefore, Espresso rules out uncompressed tensors communicated before bubbles for GC whenever the bubbles appear.

Property #2. There are two insights for the compression order of tensors. The first one is that compressing larger tensors can bring more benefits to DDL because

GC incurs a constant overhead to launch GPU kernels for compression [215, 187]. Figure 4.10 shows the ratio of the reduced communication time to the incurred compression time with 64 GPUs and NVLink. The ratio increases with tensor sizes and it indicates that GPU compression is more efficient for larger tensors. The second one is that compressing tensors closer to the output layer, i.e., the last layer during backward propagation, can bring more benefits. For example, in Figure 4.9(c), T_1 and T_2 have the same size. Compressing T_2 can reduce more iteration time than compression T_1 for two reasons: 1) T_2's compression overlaps more with communication and has no contention with tensor computation, and 2) compressing T_2 can reduce more communication overhead because its communication overlaps less with tensor computation. Based on these two insights, Espresso applies GC to tensors in the descending order of their sizes, and prioritizes tensors closer to the output layer when they have the same size.

Property #3. Espresso considers the communication and compression overheads to determine the compression options. As discussed in Section 4.3.1, only considering the communication and compression time for the decisions can harm the performance because they can overlap with other operations. Given a tensor, Espresso enumerates the possible compression options and expresses the corresponding interactions among tensors. It then chooses the one which minimizes the iteration time as the compression option.

Algorithm 4.1 shows Espresso's GPU compression decision algorithm to determine the compression option of each tensor in a DNN model. It first sorts and groups tensors with Lines 2-3 (Property #2) and then rules out uncompressed tensors communicated before bubbles with `Remove()` (Property #1). Given a tensor T_{idx}, `GetBestOption()` enumerates the possible GPU compression options for this tensor and keeps the options of other tensors unchanged (Lines 16-20). Then there are $|\mathcal{C}_{gpu}| + 1$ strategy candidates (one of them is no compression). Espresso can derive the iteration time of each candidate with the empirical models introduced in Section 4.6. Line 21 accounts for the interactions among tensors and selects the best candidate with the minimum iteration time (Property #3). After determining the

Algorithm 4.1 Espresso with GPU compression

Input: S is a compression strategy and $S[i]$ is the compression option for T_i. It is initialized with no compression for all tensors. $\mathcal{C}_{gpu}$ is the set of compression options with GPUs only. G_{m_i} is a group of tensors with size m_i.

Output: S

38 **Function** `Main()`:
39 sort all tensors in descending order of their sizes and group them based on their sizes to have $\mathcal{G} = \{G_{m_1}, G_{m_2}, \cdots, G_{m_n}\}$, where $m_1 > \cdots > m_n$
40 sort tensors in each group of $\mathcal{G}$ in ascending order of their distances to the output layer of the DNN model
41 Remove($S, \mathcal{G}$)
42 **for** $i \leftarrow 1$ **to** n **do**
43 **foreach** $T_j \in G_{m_i}$ **do**
 `// S is updated after GetBestOption()`
44 S = GetBestOption(S, j)
45 Remove($S, \mathcal{G}$)
46 **end**
47 **end**
48 **return** S
49 **Function** `Remove(`S`,` $\mathcal{G}$`)`:
50 derive the communication timeline with compression strategy S and detect the communication bubbles; remove uncompressed tensors from $\mathcal{G}$ communicated before bubbles
51 **Function** `GetBestOption(`S`,` idx`)`:
52 candidates = [S]
53 **foreach** $c_i \in \mathcal{C}_{gpu}$ **do**
54 $S_i = S.copy()$
55 $S_i[idx] = c_i$
56 candidates.add(S_i)
57 **end**
 `//` $F(S)$ `is the iteration time with` S
58 **return** $\arg\min\{F(S_j) \mid S_j \in \text{candidates}\}$

compression option of one tensor, Espresso checks if new bubbles appear and rules out uncompressed tensors communicated before them again in Line 8 (Property #1).

4.7.3 Espresso's CPU offloading

Espresso offloads compression from GPUs to CPUs to further improve the training throughput of DDL after Algorithm 4.1. Tensors with no compression are ruled out for CPU offloading and the set of the left tensors is denoted as $\mathcal{T}_{gpu}$, which can have hundreds of tensors. The time complexity with brute force for CPU offloading is $O(2^{|\mathcal{T}_{gpu}|})$. Tensors in $\mathcal{T}_{gpu}$ can have the same compression option, i.e., they take the same compression choice and communication schemes. Espresso takes a greedy algorithm to find a provably optimal compression strategy for CPU offloading based on an interesting observation.

Lemma 4.1. *Suppose G is a set of tensors with the same size and same compression option from $\mathcal{T}_{gpu}$. Suppose also q tensors in G must be offloaded to CPUs for compression. The best solution is to offload the q tensors farthest from the output layer.*

The intuition of Lemma 4.1 is that offloading tensors to CPUs earlier can overlap more CPU compression with communication and tensor computation, and thus reduce the CPU compression overheads. Therefore, if tensors in $\mathcal{T}_{gpu}$ can be grouped like G in Lemma 4.1, there is no need to evaluate all possible combinations because Lemma 4.1 restricts the choices of tensors for CPU offloading in each group.

Algorithm 4.2. Espresso first groups $\mathcal{T}_{gpu}$ to have $\mathcal{G}^{gpu} = \{G_1^{gpu}, G_2^{gpu}, \cdots, G_d^{gpu}\}$, where G_i^{gpu} is a set of tensors with the same size and the same compression option. The tensors in G_i^{gpu} are sorted in the descending order of their distances to the output layer. Denote $U = \{u_1, u_2, \cdots, u_d\}$, where u_i is the number of tensors in G_i^{gpu} for CPU offloading and $0 \leq u_i \leq |G_i^{gpu}|$. $\mathcal{U}$ is the set of all possible U. For each $U \in \mathcal{U}$, Espresso considers a compression strategy that offloads the compression of the first u_i tensors in G_i^{gpu} to CPUs, and derives its iteration time. It traverses $\mathcal{U}$ to search for the best U with the minimum iteration time.

Theorem 4.1. *Algorithm 4.2 can find the best CPU offloading solution in $O(\prod(|G_i^{gpu}|+1))$ given $\mathcal{T}_{gpu}$.*

Proof. We first prove that given a $U = \{u_1, u_2, \cdots, u_d\}$, the best CPU offloading is to offload the first u_i tensors in G_i^{gpu} to CPUs. Without loss of generality, we assume in

Model	Dataset	Batch size	Model size
VGG16 [190]	ImageNet [64]	32 images	528 MB
ResNet101 [87]	ImageNet [64]	32 images	170 MB
UGATIT [101]	selfie2anime [180]	2 images	2559 MB
BERT-base [67]	SQuAD [167]	1024 tokens	420 MB
GPT2 [164]	WikiText-2 [130]	80 tokens	475 MB
LSTM [129]	WikiText-2 [130]	80 tokens	328 MB

Table 4.4 : Characteristics of the benchmark DNN models.

the best CPU offloading, the u_j offloaded tensors in G_j^{gpu} are not the first u_j tensors, which contradicts the conclusion in Lemma 4.1. Then the assumption does not hold.

Because the number of tensors in G_i^{gpu} is $|G_i^{gpu}|$, there are $|G_i^{gpu}|+1$ options for the number of tensors for CPU offloading, from 0 tensors to $|G_i^{gpu}|$ tensors. Therefore, the number of possible U in $\mathcal{U}$ is $O(\prod(|G_i^{gpu}|+1))$. For each U, its best offloading solution is determined. Therefore, Espresso only needs to traverse the $\prod(|G_i^{gpu}|+1)$ possibilities to find the best CPU offloading. □

4.8 Evaluation

4.8.1 Experimental Setup

Testbeds. Two testbeds are used: 1) 8 GPU machines with NVLink and a 100Gbps network with TCP/IP, and 2) 8 PCIe-only GPU machines with a 25Gbps network. Each machine has 8 NVIDIA Tesla V100 GPUs (32 GB GPU memory) and 2 CPUs/48 cores (Intel Xeon 8260 at 2.40GHz). Each machine runs Debian 10 and the software environment includes CUDA-11.0, PyTorch-1.8.0, BytePS-0.2.5, and NCCL-2.7.8.

Workloads. We use six popular real-world DNN models including three computer vision (CV) models (VGG16, ResNet101, and UGATIT) and three natural language processing (NLP) models (BERT-base, GPT2, and LSTM) by following the literature [94, 74, 183]. We set the batch sizes of these models by also following the literature [105, 129, 38, 74, 183]. Specifically, the per-GPU batch size is kept con-

stant as the number of GPUs increases, and the batch sizes are modest because large batch sizes are known to cause convergence problems [196, 183]. The details of the models, datasets, and batch sizes are shown in Table 4.4.

Compression algorithms. We use three representative compression algorithms: Randomk [192] and DGC [115] for sparsification (99% sparsity), and EFSignSGD [100] for quantization. Error-feedback [100, 115] is applied on both GPU and CPU compression to preserve the model accuracy.

Baselines. We use BytePS [94] as the training baseline without GC (FP32). We use HiPress [38] and HiTopKComm [188] as the two baselines with GPU compression, and BytePS-Compress [240] as the baseline with CPU compression. These baselines explore narrower search spaces in comparison to Espresso.

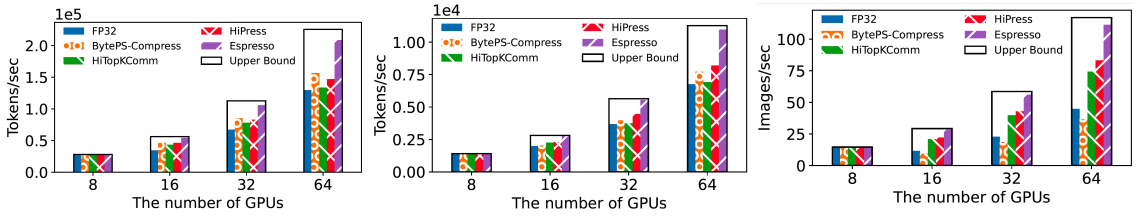
Performance metrics. We use trained images per second as the performance metric for CV models and tokens per second for NLP models. We measure the computational time of Espresso and the training accuracy of DNN models. We also provide the upper bound on the training throughput of compression-enabled DDL (Upper Bound). This is obtained by assuming GC has no compression overhead and has no impact on tensor computation.

Implementation. We implement a GPU compression library shared by HiPress, HiTopKComm, and Espresso as well as a CPU compression library shared by BytePS-Compress and Espresso. We also implement a communication library to support different communication schemes in both intra- and inter-machine communications shared by all baselines and Espresso. These libraries consist of 5.1K and 3.0K lines of code in C++ and Python. Espresso's decision algorithm is implemented with 1.1K lines of code in Python.

4.8.2 End-to-End Experiments with NVLink-based GPU machines

Figure 4.12 shows the training throughput of three DNN models with Espresso and baselines. The performance bottleneck is inter-machine communication.

As shown in Figure 4.12a, the compression baselines bring very limited speedups over FP32 for BERT-base. For example, HiTopKComm and HiPress only outperform



(a) BERT-base+Randomk (b) GPT2+EFSignSGD (c) UGATIT+DGC

Figure 4.12 : Throughput of DNN models with NVLink-based GPU machines and 100Gbps cross-machine Ethernet.

FP32 by up to 4% and 13%, respectively. It is because there are a large number of tensors in BERT-base, while none of the baselines consider the interactions among tensors. Their compression strategies lead to costly compression overheads. Espresso significantly improves the performance over all baselines. For example, with 64 GPUs, it outperforms BytePS-Compress, HiTopKComm, and HiPress by 31%, 54%, and 40%, respectively. For GPT2, it outperforms BytePS-Compress and HiPress by 42% and 33% with 64 GPUs, as shown in Figure 4.12b.

UGATIT is very communication-intensive because of its large model size. When the number of GPUs is 64, the performance improvement with HiPress and HiTopKComm is 86% and 66%, respectively, as shown in Figure 4.12c. BytePS-Compress even harms the performance by 18% due to the costly computational overhead for CPU compression. Espresso leverages both GPUs and CPUs for compression. It outperforms FP32, BytePS-Compress, HiTopKComm, and HiPress by 149%, 205%, 50%, and 35%, respectively. One important observation is that the improvements of Espresso become larger from 8 GPUs to 64 GPUs. This implies that when DDL scales out, the computational overhead caused by compression also increases, and Espresso becomes more beneficial.

4.8.3 Computational time of Espresso

Table 4.5 lists the computational time of Espresso to select compression strategies for the training of different DNN models with 8 NVLink-based GPU machines (the results are similar with PCIe-only GPU machines). The time increases with the

	VGG16	ResNet101	UGATIT	BERT-base	GPT2	LSTM
# of Tensors	32	314	148	207	148	10
Espresso	17ms	179ms	84ms	125ms	99ms	1ms
Brute force	> 24h	> 24h	> 24h	> 24h	> 24h	> 24h

Table 4.5 : The time to select compression strategies. # of tensors is the number of tensors in DNN models.

	VGG16	ResNet101	UGATIT	BERT-base	GPT2	LSTM
# of Tensors	11	42	32	54	34	5
Espresso	1ms	30ms	12ms	44ms	18ms	1ms
Brute force	1ms	> 24h	1.9h	> 24h	7.6h	1ms

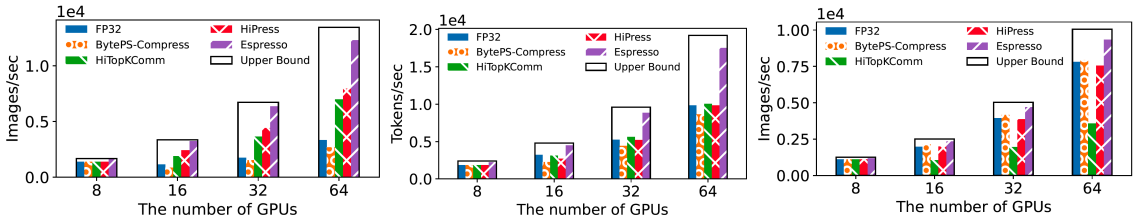
Table 4.6 : The time to find the best CPU offloading solutions. # of tensors is the number of tensors for offloading.

number of tensors in DNN models, but even for ResNet101 with 314 tensors, the computational time is still within one iteration time. In contrast, brute force takes a very long time because it has to traverse all the possibilities. Even though LSTM only has 10 tensors, the search time is still unacceptable.

Table 4.6 shows the computational time of Espresso to find the best CPU offloading solution. After Espresso's GPU compression decision algorithm, the number of tensors for CPU offloading has been significantly reduced. Brute force can quickly find the best solution for VGG16 and LSTM, but it takes a long time for other models. Espresso can still quickly find the best solution. For example, there are 54 tensors in BERT-base for CPU offloading, but they only have a few different tensor sizes, as shown in Figure 4.11. Espresso only needs to consider a few thousand choices to find the best CPU offloading.

4.8.4 End-to-End Experiments with PCIe-only GPU machines

The performance bottlenecks could be both inter- and intra-machine communications in this setup. Figure 4.13b shows that the three compression baselines bring almost



(a) VGG16+Randomk (b) LSTM+EFSignSGD (c) ResNet101+DGC

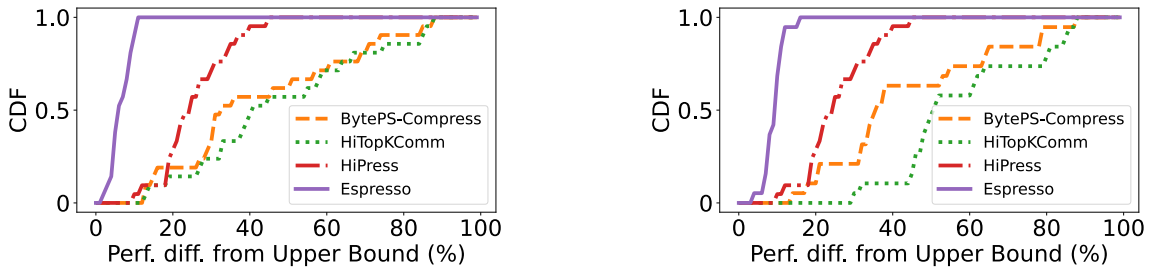
Figure 4.13 : Throughput of DNN models with PCIe-only GPU machines and 25Gbps cross-machine Ethernet.

no improvement for LSTM model with GC. For example, HiPress only outperforms FP32 by up to 2%, and BytePS-Compress even harms the performance by 12% with 64 GPUs. It is because they only compress tensors to reduce inter-machine communication and cannot effectively alleviate the intra-machine communication bottleneck. Moreover, they also incur costly compression overhead. Espresso compresses tensors to reduce both inter- and intra-machine communications when necessary and always has the best performance across all cases. For example, with 64 GPUs, it outperforms BytePS-Compress, HiTopKComm, and HiPress by 101%, 73%, 77%, respectively. For VGG16 model with 64 GPUs, the speedups of Espresso over FP32, BytePS-Compress, and HiPress are 269%, 357%, 55%, respectively.

We observe that ResNet101 is not communication-intensive and it achieves the scaling factor of 0.70 even with FP32. Figure 4.13c shows applying GC to ResNet101 with the compression baselines harms its performance. HiTopKComm reduces its training throughput by up to 54% because it compresses all tensors, causing exorbitant compression overhead. HiPress also has high over-compression penalties and it degrades the performance by 4% with 64 GPUs. In contrast, Espresso outperforms FP32, BytePS-Compress, and HiPress by up to 20%, 18%, and 24%, respectively.

4.8.5 Espresso's compression strategies are near-optimal

We have performed experiments for all combinations of GC algorithms (i.e. Randomk, DGC, EFSignSGD), DNN models (i.e. VGG16, ResNet101, UGATIT, BERT-



(a) NVLink-based machines

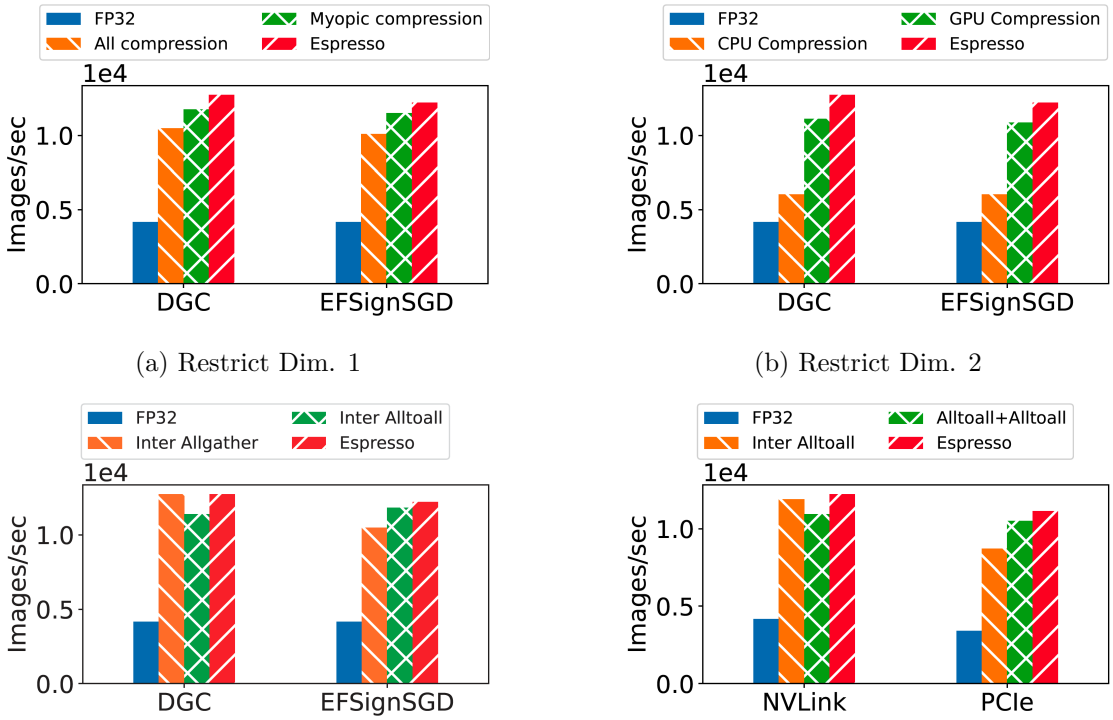
(b) PCIe-only machines

Figure 4.14 : The performance differences between compression frameworks and Upper Bound with 64 GPUs.

base, GPT2, LSTM), varying the number of GPUs from 8 to 64, over both NVLink and PCIe, across all schemes (i.e. FP32, HiPress, BytePS-Compress, HiTopKComm, Espresso). We present a summary of all the results for the 64-GPU scenario. Specifically, we present the cumulative distribution of the performance differences of each scheme from the Upper Bound. Figure 4.14a displays the distributions of performance differences for all the training with NVLink-based machines and 64 GPUs. The performance differences between Espresso and Upper Bound is always less than 10%. To call out a few specific data points, the performance differences for the training of GPT2 with EFSignSGD, UGATIT with DGC, and BERT-base with Randomk are only 3%, 5%, and 7%, respectively. Note that the differences between Espresso's compression strategy and the optimal strategy can be even smaller because Upper Bound is by definition higher than the training throughput of the optimal strategy. Figure 4.14b shows the distributions for all the training with PCIe-only machines and 64 GPUs and Espresso similarly outperforms other baselines.

4.8.6 Importance of the Entire Search Space

To evaluate the importance of considering all four dimensions, we cripple one of the dimensions and then select the compression strategy with the remaining three dimensions. We cripple Dimension 1 with two restricted mechanisms: **All compression:** It compresses all tensors. **Myopic compression:** It does not consider interactions



(a) Restrict Dim. 1

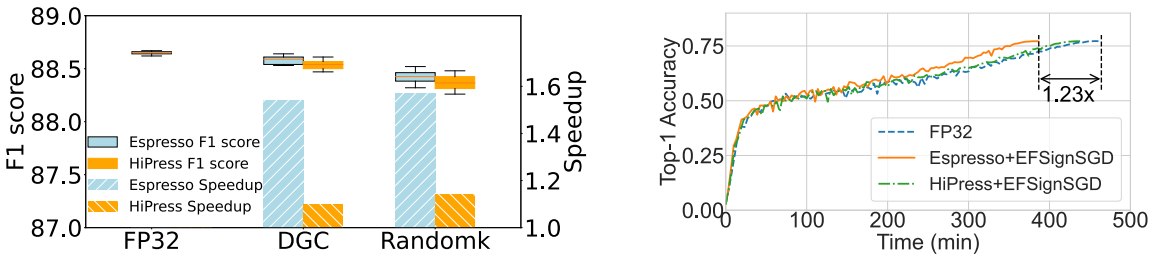
(b) Restrict Dim. 2

(c) Restrict Dim. 3

(d) Restrict Dim. 4

Figure 4.15 : Considering all four dimensions is always better than considering only three dimensions.

among tensors when applying GC to tensors. We cripple Dimension 2 with two restricted mechanisms: **GPU compression:** It only compresses tensors with GPUs. **CPU compression:** It only compresses tensors with CPUs. We cripple Dimension 3 with two restricted mechanisms: **Inter Allgather:** It compresses tensors for inter-machine communication and uses Allgather for compressed tensors. **Inter Alltoall:** It compresses tensors for inter-machine communication. The communication scheme is Alltoall/Allgather. We cripple Dimension 4 with **Inter Alltoall** and another restricted mechanism **Alltoall+Alltoall:** It first compresses tensors for the first intra-machine communication and the communication scheme is Alltoall. It then decompresses and compresses tensors again for inter-machine communication. It uses Alltoall/Allgather for inter-machine communication and Allgather for the second intra-machine communication.



(a) BERT (b) ResNet101

Figure 4.16 : Model accuracy of BERT-base (F1 score) and ResNet101 (Top-1 accuracy).

Figure 4.15 shows the scaling factors of VGG16 with 64 GPUs. NVLink-based GPU machines are used in (a), (b), and (c), and EFSignSGD is used in (d). The compression strategies determined by Espresso always outperforms the compression strategies selected from the cripple search space. Moreover, Figure 4.15(c) verifies that different types of GC algorithms need different communication schemes, and Figure 4.15(d) verifies that different intra- and inter-machine bandwidth need different compression choices.

4.8.7 Convergence validation

It has been theoretically proven and empirically validated that GC can preserve the training accuracy and convergence [186, 223, 192, 115, 93, 74, 38]. In this section, we reaffirm these conclusions and demonstrate that Espresso can preserve training accuracy and convergence.

We conduct a test following the methodology in [74] to fine-tune BERT-base for the question-answering task on SQuAD [167] for two epochs and repeat the experiments ten times. The number of GPUs is 64 on 8 NVLink-based GPU machines. Figure 4.16a shows that Espresso with DGC can achieve around 1.55× speedup over no compression (i.e. FP32) and it has almost the same F1 score as no compression. We also train ResNet101 for 120 epochs on ImageNet [64] from scratch and apply EFSignSGD to the model training. As shown in Figure 4.16b, the speedup of Espresso over no compression (i.e. FP32) is 1.23×. The achieved Top-1 accuracy with Espresso is 77.10%, which is very close to the no-compression accuracy of 77.18%.

Chapter 5

Cupcake: A Compression Optimizer for Scalable and Communication-Efficient Distributed Training

Espresso unleashes the benefits of GC algorithms by finding the near-optimal compression strategy. Like other existing compression-enabled DDL systems [38, 226], it applies GC algorithms in a layer-wise fashion, i.e., tensor by tensor. We observe that there is still improvement room to further reduce the compression overhead caused by this layer-wise fashion due to the fixed overheads to launch and execute kernels in CUDA [33]. In this chapter, we propose Cupcake, a compression optimizer that applies GC algorithms in a *fusion fashion*, i.e., fuse multiple tensors for one compression operation. Cupcake determines the provably optimal fusion strategy to maximize training throughput by reducing the amount of communicated data and minimizing the compression overhead simultaneously.

5.1 Introduction

As discussed in Chapter 4, gradient compression (GC) is a promising approach to alleviate the communication bottleneck in DDL by significantly reducing the amount of communicated data. Nonetheless, it is challenging to achieve the desired speedup when applying GC to DDL jobs [38, 26]. In this chapter, we first analyze the practical difficulty when applying GC to DDL with the *layer-wise* fashion, which is the state-of-the-art approach used in existing compression-enabled DDL systems [38, 226, 214]. The layer-wise fashion compresses tensors one by one when they are ready for communication and it can greatly shorten the communication time for gradient synchronization thanks to the reduced amount of traffic volume. However, we observed that the end-to-end training speedups of DDL with the layer-wise compression are only

modest, and even worse than training without GC in many cases due to the incurred prohibitive overhead of compression operations [226, 217].

We then propose Cupcake to maximize the training throughput of compression-enabled DDL by reducing the amount of communicated data and minimizing the compression overhead simultaneously. Cupcake is a general compression optimizer to enable GC algorithms to unleash their benefits to accelerate DDL. Essentially, GC reduces the communication time of DDL at the cost of the compression overheads. Because of the fixed overheads to launch and execute kernels in CUDA [34], the compression overhead is non-negligible even for small tensor sizes. Fortunately, we observe that this overhead remains constant when the tensor size is smaller than a threshold (e.g., 4 MB in our testbed) and it then linearly increases with the tensor size. This observation motivates us to fuse multiple tensors for one compression operation.

Fusing tensors for compression leads to a trade-off between the reduced compression overhead and the communication overhead, i.e., the communication time that cannot overlap with computation. Because communication overlaps with computation in DDL [231, 187, 94, 112], gradient tensors can begin their communications whenever they are ready in the layer-wise fashion. However, in a fusion fashion, a tensor has to wait for its following tensors for a unified compression operation and communication operation. Therefore, fusion can delay communications, shrink the overlapping time, and thus worsen the iteration time. To address this challenge, Cupcake determines the provably optimal fusion strategy for applying GC to DDL by balancing the compression overhead and the communication overhead. It can maximize the training throughput of compression-enabled DDL jobs, regardless of different training models, GC algorithms, and training system configurations, such as the number of GPUs and the network bandwidth.

Our evaluations in both computer vision and NLP demonstrate that GC algorithms applied with Cupcake can greatly improve the training throughput of DDL. Specifically, Cupcake enables GC algorithms to achieve up to 2.03× speedup in training throughput over training without GC, and up to 1.79× speedup over the state-of-the-start solutions that compress tensors in a layer-wise fashion.

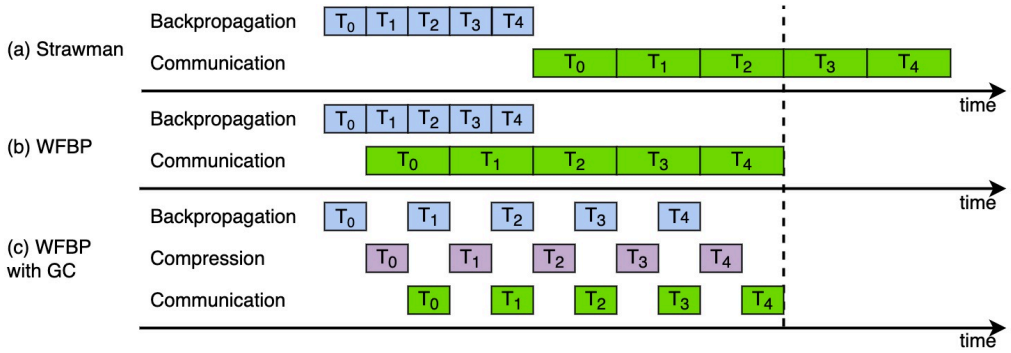


Figure 5.1 : An example of DDL with five tensors for gradient synchronization. (a) is the strawman in which communications have to wait for the completion of backpropagation. (b) uses WFBP to overlap communication with backpropagation to reduce the iteration time. In (c), every tensor is compressed, but it does not reduce the iteration time compared to (b) due to the compression overheads. Forward propagation and decoding are omitted.

5.2 The Practical Performance of GC with Layer-wise Compression

5.2.1 Overlapping Communication with Computation

Because of the layered structure and a layer-by-layer computation pattern in DNN models, the wait-free backpropagation mechanism (WFBP) [231, 187, 94, 112, 50] is widely adopted to overlap communication with computation in DDL. As illustrated in Figures 5.1(a) and 5.1(b), WFBP can significantly reduce the iteration time compared to the strawman solution, in which communication cannot begin until the completion of backpropagation. Existing distributed ML frameworks, such as PyTorch [156], Tensorflow [25], and Horovod [187], apply GC to DDL in a *layer-wise fashion*, i.e., tensor by tensor, to overlap communication with computation because of WFBP.

5.2.2 Empirical Measurements

Because applying GC to DDL requires computation resources, it competes for GPU resources with backpropagation and delays tensor computation, as shown in Figure 5.1(c). Although GC algorithms can reduce the communication time of DDL, the incurred compression overheads can dramatically dilute the benefits gained from the reduced communication time.

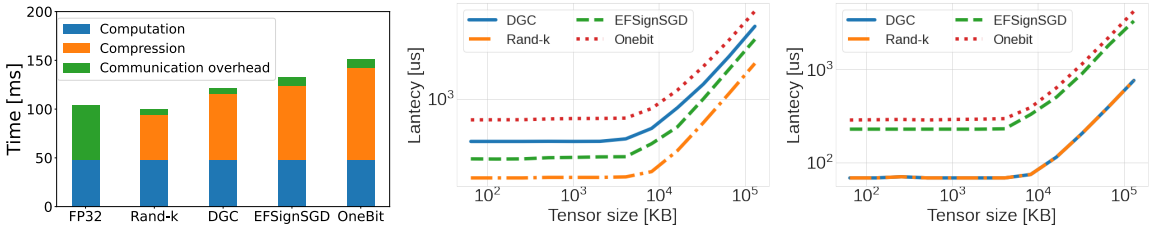
To demonstrate, we empirically measure the training throughput of compression-enabled DDL with several popular GC algorithms, including both sparsification and quantization. The experiments are conducted on a server equipped with 8 GPUs (NVIDIA Tesla V100 with 32 GB memory), two 20-core/40-thread processors (Intel Xeon Gold 6230 2.1GHz), and PCIe 3.0×16. GRACE [226] is used to support compression-enabled DDL and GC algorithms are applied in a layer-wise fashion. The training model is ResNet50 [87] over CIFAR10 [107] and the batch size is 32.

Two sparsification algorithms, DGC [115] and Rand-k [192], are evaluated and the gradient sparsity is 99%, i.e., only 1% of gradients are exchanged during synchronization. Two 1-bit quantization algorithms, EFSignSGD [100] and OneBit [186], are also evaluated.

Figure 5.2a shows the breakdown of the iteration time of the training. The *communication overhead* refers to the communication time that cannot overlap with backpropagation and compression of any tensors. FP32 is the training baseline without GC. We observe that the performance improvement with these GC algorithms is modest. Some algorithms, such as DGC, EFSignSGD, and OneBit, even surprisingly lead to a longer iteration time. This observation is on par with the findings in prior works [226, 183, 113, 84, 26].

5.2.3 The Root Cause of the Poor Performance

When a gradient tensor is ready for synchronization in DDL without compression, it is communicated and then the aggregated results are used to update the training model. However, there are two additional operations when applying GC to DDL: encoding (encode a tensor before communication to reduce the traffic volume) and



(a) Breakdown of iteration time (b) Encoding overhead (c) Decoding overhead

Figure 5.2 : The compression overheads with different compression algorithms. The data in (a) are collected from the training of ResNet50; the data in (b) and (c) are collected from a microbenchmark. Both encoding and decoding overheads are non-negligible.

decoding (decode the received compressed tensor for model updates) [*]. These two operations can incur non-negligible computation overhead.

Figure 5.2b and 5.2c display the encoding and decoding latencies with four representative GC algorithms, i.e., DGC [115] and Rand-k [192] for sparsification, EFSignSGD [100] and Onebit [186] for quantization. Both encoding and decoding latencies are non-negligible, even for tensors with small sizes. For instance, the encoding latencies of DGC, EFSignSGD, and Onebit are all greater than 0.25 ms, regardless of the tensor sizes.

DNN models typically have a large number of tensors for gradient synchronization [87, 67]. The layer-wise compression invokes encoding and decoding operations for each tensor and leads to prohibitive compression overheads. We take training ResNet50 over CIFAR10 with 8 GPUs in our testbed as a concrete example to compare the overall compression overhead against the communication improvement. In our measurement, the iteration time of the single-GPU training is around 48 ms. Without any compression, the communication overhead in each iteration is about 56 ms. Both sparsification and 1-bit quantization algorithms can reduce the communication overhead to less than 10 ms thanks to the much smaller communicated traffic

[*]It may require more decoding operations after communication when multiple encoded tensors are received on each GPU.

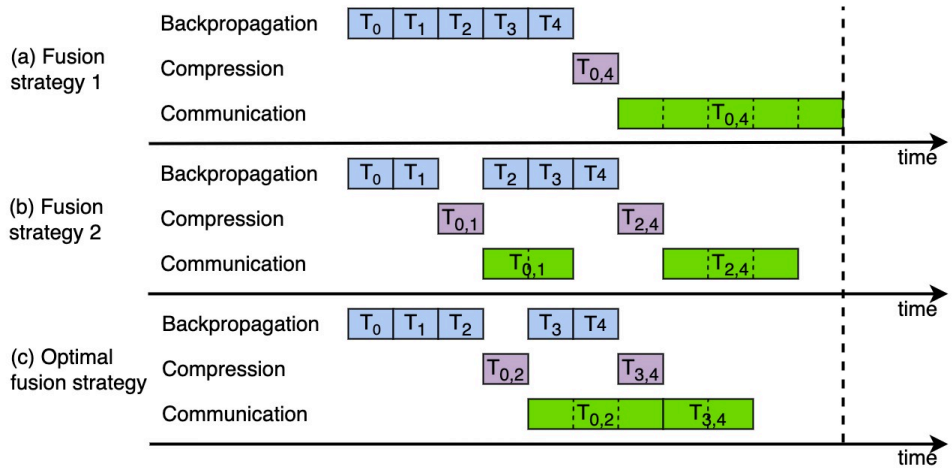


Figure 5.3 : Cupcakefuses multiple tensors for one compression operation and one communication operation to minimize the iteration time. It is challenging to find the optimal fusion strategy given a DDL job and a GC algorithm because fusing tensors leads to a trade-off between the reduced compression overhead and the overlapping time between communication and backpropagation.

volume. However, the overall compression overheads of DGC and EFSignSGD are both larger than 60 ms, which is even higher than the communication overhead in the baseline. The costly compression overheads result in the poor practical performance of DDL with GC.

5.2.4 An Opportunity and a Challenge

The compression overhead of GC algorithms with a layer-wise fashion becomes the new efficiency bottleneck in DDL. We observe that there are some fixed overheads to launch and execute kernels in CUDA [34]. Figure 5.2 shows that the encoding and decoding latencies of GC algorithms keep constant when the tensor size is smaller than a threshold (e.g., 4 MB in our testbed). They then almost linearly increase with the tensor sizes. This observation indicates that fusing multiple tensors for one compression operation can potentially reduce the overall compression overhead and

thus the iteration time. Suppose there are ten tensors with a size of 128 KB for gradient synchronization and the encoding latency of DGC for a 128 KB tensor is 0.4 ms. If we can fuse these ten tensors for one encoding operation, the overall encoding latency is still 0.4 ms, which is significantly lower than the latency incurred by ten encoding operations for the ten tensors separately.

However, fusing tensors for GC algorithms raises a new challenge: what is the optimal fusion strategy to minimize the iteration time? There is a trade-off between the compression overhead and the communication overhead because fusing tensors to reduce the compression overhead has to delay communications and thus shrink the overlapping time between communication and computation (including both backpropagation and compression). For example, an extreme case of tensor fusion is applying a GC algorithm to an entire training model with only one compression operation. However, in this case, communication cannot begin until the completion of backpropagation, resulting in suboptimal communication overhead, as shown in Figure 5.3(a). Another extreme case is to apply the layer-wise fashion to compress tensors one by one to minimize the communication overhead, but it leads to prohibitive compression overheads, as discussed in Section 5.2.3.

There are numerous fusion strategies to apply a GC algorithm to a DDL job and three strategies are illustrated in Figure 5.3. It is challenging to find the optimal one because it depends on many factors, such as the applied GC algorithms, the tensor size and the computation time of the DNN model, the number of GPUs, and network bandwidth. We must jointly consider backpropagation, compression, and communication overheads to search for the optimal strategy to maximize the training throughput of compression-enabled DDL jobs.

5.3 Cupcake

In this section, we first formulate the tensor fusion problem given a DDL job and a GC algorithm. We then design an algorithm to provably find the optimal fusion strategy to minimize the iteration time.

5.3.1 Problem Formulation

The core idea of Cupcakeis to fuse multiple tensors for one compression operation, instead of applying GC to DDL in a layer-wise fashion. It can reduce the compression overhead and meanwhile overlap communication with computation to reduce the communication overhead.

Given a training model with N tensors, the set of tensors is $\mathcal{T} = \{T_0, \ldots, T_{N-1}\}$. For example, Figure 5.1 and Figure 5.3 display a training model with five tenors. Cupcakepartitions the model into y groups and determines a fusion strategy $\mathcal{X}_y = \{\mathrm{x}_0, \ldots, \mathrm{x}_{y-1}\}$, where x_i is a group of consecutive tensors that are compressed and communicated together. Cupcakeperforms an encoding operation and a communication operation for each tensor group in each iteration. After encoding, the fused tensor x_i is communicated and synchronized. After communication, the encoded x_i is decoded and aggregated to update the training model.

Let A denote the computation time for forward propagation in an iteration and $B(T_i)$ denote the computation time of T_i in backpropagation. x_i is the total tensor size of x_i. $h(\mathrm{x}_i)$ is the time to compress x_i and $g(\mathrm{x}_i)$ is the corresponding communication time. $P(\mathcal{X}_y)$ is the total overlapping time, i.e., the total communication time that overlaps with the compression and backpropagation of any tensors. Given a fusion strategy $\mathcal{X}_y = \{\mathrm{x}_0, \ldots, \mathrm{x}_{y-1}\}$, the iteration time is

$$f(\mathcal{X}_y) = A + \sum_{b=0}^{N-1} B(T_i) + \sum_{i=0}^{y-1} h(\mathrm{x}_i) + \sum_{i=0}^{y-1} g(\mathrm{x}_i) - P(\mathcal{X}_y). \tag{5.1}$$

A and $B(T_i)$ can be profiled offline for a training model and they are constant across iterations [234, 194]. Following the literature [199, 3, 74, 172], we model the communication time of x_i as $g(\mathrm{x}_i) = \alpha_g + \beta_g x_i$, where α_g is the latency (or startup time) per tensor and β_g is the transfer time per byte after encoding. Based on the measurement in Figure 5.2, we model the compression time of x_i as $h(\mathrm{x}_i) = \alpha_h + \beta_h x_i$, where α_h is the fixed overhead to launch and execute kernels in CUDA and β_g is the compression time per byte. Cupcakemeasures α_g, β_g, α_h, and β_h offline based on the system configurations, such as the GPU computation capacity, the number of GPUs, and the network bandwidth.

Given a fusion strategy, Cupcakecan calculate its iteration time by deriving the timelines of its backpropagation, compression, and communication [214], as shown in Figure 5.3. Unfortunately, it is still challenging to formulate $f(\mathcal{X}_y)$ due to $P(\mathcal{X}_y)$, which is determined by the strategy and the intricate interactions among backpropagation, compression, and communications of all the tensors.

Instead of deriving the expression of $P(\mathcal{X}_y)$, we formulate the tensor fusion problem in a recursive way to minimize the iteration time of a DDL job with a given GC algorithm. Let $F(M, i)$ be the iteration time of the optimal fusion strategy from T_i to T_{N-1}, given the fusion strategy for tensors from T_0 to T_{i-1} is represented by M. We then have the following recurrence relation

$$\begin{cases} F(\{\}, 0) = \min\limits_{1 \le i \le N} F(\{\text{fuse}(0, i-1)\}, i), & (5.2) \\ F(M, i) = \min\limits_{i+1 \le j \le N} F(M + \text{fuse}(i, j-1), j), & (5.3) \end{cases}$$

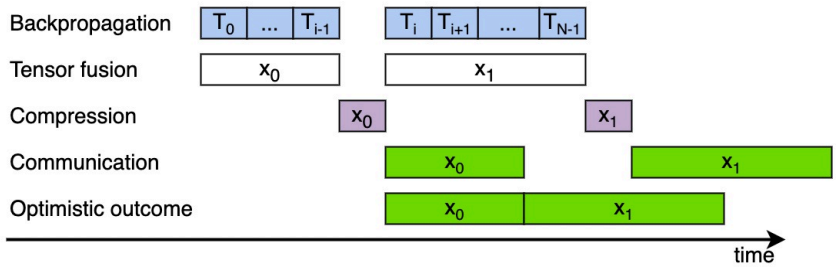
where $\text{fuse}(i, j)$ fuses tensors from T_i to T_j as one group. Cupcakefirst considers the form of x_0, i.e., $\text{fuse}(0, i-1)$, where $1 \le i \le N$, in Equation (2). It then recursively computes $F(M, i)$ to find the optimal fusion strategy for the entire model. For simplicity, let $h(i, j)$ denote the compression time of a fused tensor from T_i to T_j, $g(i, j)$ denote its communication time, and $B(i, j)$ denote its backpropagation time.

5.3.2 The Optimal Fusion Strategy

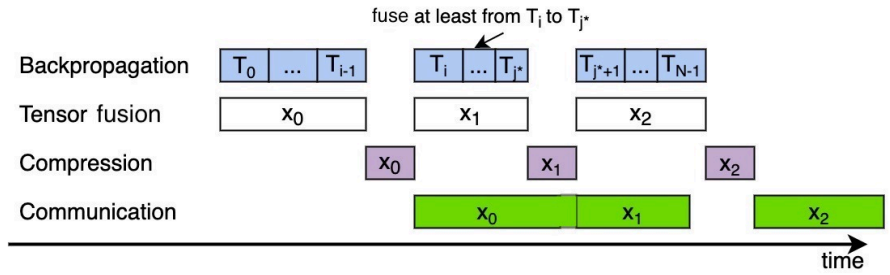
For any tensor in a DNN model except T_0, it can be either fused into the current group or form a new one. Therefore, there are 2^{N-1} possible fusion strategies. It indicates that the time complexity to address the tensor fusion problem with brute force is exponential. Because DNN models typically have hundreds of tensors, it is impractical to enumerate all possible strategies. Moreover, the optimal strategy is specific to each situation because different DDL jobs have different characteristics, such as different tensor numbers, different tensor sizes, and different system configurations.

We first introduce two pruning techniques based on two insights to enable Cupcaketo find the optimal fusion strategy efficiently.

Insight #1: It is not necessary for Cupcaketo examine all the N cases for the formation of x_0. When there are too many tensors in x_0, it can delay



(a) Cupcake prunes a strategy if its optimistic outcome is greater than the current optimal iteration time when determining x_0.



(b) Cupcake prunes strategies in which x_1 fuses tensors from T_i to T_j, where $j < j^*$, when x_0 is determined.

Figure 5.4 : Examples of the two pruning techniques.

communication and lead to a long iteration time. Given a case of x_0, we can calculate the *optimistic outcome* of the iteration time as follows.

$$\begin{aligned} &F(\{\text{fuse}(0, i-1)\}, i) \geq \\ &\max\{B(0, N-1) + h(0, i-1) + h(i, N-1), \\ &\quad B(0, i-1) + h(0, i-1) + g(0, i-1) + g(i, N-1)\}. \end{aligned} \tag{5.4}$$

The optimistic outcome considers two cases. The first case is that there is no communication overhead. The second case is that except tensors in x_0, all the other tensors are fused as one group for compression and communication, and x_1's communication begins right after the completion of x_0's communication, as shown in Figure 5.4a. The compression time of x_1 is perfectly overlapped with communication. If the optimistic outcome of a case is already greater than the iteration time of the best fusion strategy

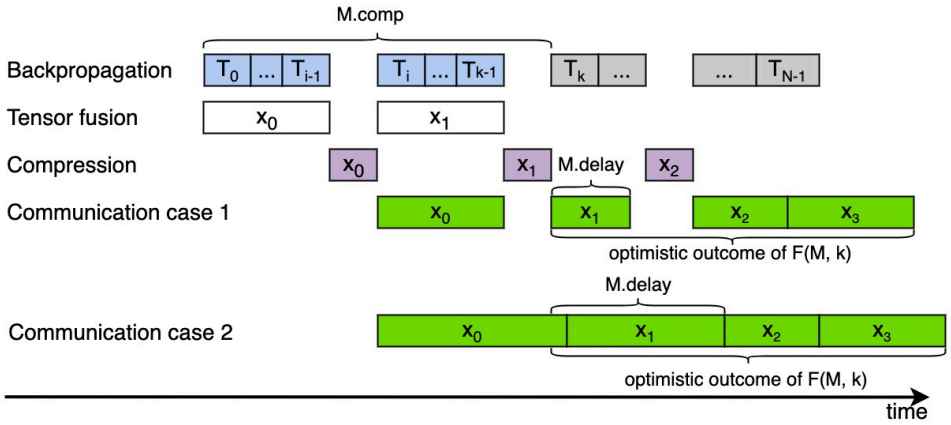


Figure 5.5 : A general case for the two pruning techniques given M, which is the set of fused tensors from T_0 to T_{k-1}. *M.comp* and *M.delay* can be derived based on the timelines of backpropagation, compression, and communication of tensors in M.

found so far, then this case, i.e., the recursive computation for $F(\{\text{fuse}(0, i-1)\}, i)$, can be pruned.

Insight #2: It is safe to fuse more tensors in a group based on the progress of communication of the previous group. Suppose x_0 is fused from T_0 to T_{i-1}. We apply Equation (5.3) recursively to enumerate cases for x_1, which is formed by fusing tensors from T_i to T_j. The backpropagation time and the compression time of x_1 can overlap with x_0's communication, as shown in Figure 5.4b. The smallest j can be calculated with

$$j^* = arg\,max_j\{B(i,j) + h(i,j) \leq g(0, i-1)\}. \tag{5.5}$$

Note that the less number of tensors in x_1 means more tensors in x_2 and DDL has to communicate more tensors after the completion of backpropagation. Fusing from T_i to T_j, where $j < j^*$, is no better than fusing from T_i to T_{j^*} because it shrinks the overlapping time between communication and computation. Therefore, Cupcakecan prune the strategies in which x_1 fuses tensors from T_i to T_j where $j < j^*$ and only examine $j \geq j^*$.

Algorithm 5.1 Optimal Fusion Strategy

Input: N is the number of tensors in a DNN model. $global\_opt\_fuse = \{\}$. $global\_opt\_time = \infty$

Output: The optimal fusion strategy $global\_opt\_fuse$.

```
59 Function Main():
60     for k ← 1 to N do
61         FindOptFusion({fuse(0, k-1)}, k)
62     end
63     return global_opt_fuse
64 Function FindOptFusion(M, k):
65     local_opt_fuse ← {fuse(k, N − 1)}
       // f() is defined in Equation (5.1)
66     local_opt_time ← f(M + fuse(k, N − 1))
67     j* ← k
68     for i ← k to N − 1 do
69         if B(k, i) + h(k, i) ≤ M.delay then
70             j* ← i
71         else
72             break
73         end
74     end
75     M.comp = B(0, k − 1) + Σ_{x∈M} h(x)
76     for i ← j* to N − 1 do
77         base ← B(k, N − 1) + h(k, i) + h(i + 1, N − 1)
78         cases ← max{B(k, i) + h(k, i), M.delay} + g(k, i) + g(i + 1, N − 1)
79         optim_outcome ← M.comp + max(base, cases)
80         if optim_outcome > global_opt_time then
81             continue
82         end
83         first_fuse ← fuse(k, i)
84         rest_fuse ← FindOptFusion(M + first_fuse, i + 1)
85         cur_fuse ← first_fuse + rest_fuse
86         cur_fuse_time = f(M + cur_fuse)
87         if cur_fuse_time < local_opt_fuse_time then
88             local_opt_fuse ← cur_fuse
89             local_opt_time ← cur_fuse_time
90         end
91         if cur_fuse_time < global_opt_time then
92             global_opt_fuse ← M + cur_fuse
93             global_opt_time ← cur_fuse_time
94         end
95     end
96     return local_opt_fuse
```

Fusion algorithm. Based on the two insights of examining possible fusion strategies,

we design Algorithm 5.1 that finds the optimal fusion strategy given a DDL job and a GC algorithm. `Main()` checks the N cases of x_0 and it invokes `FindOptFusion()` to recursively search for the optimal strategy (lines 1-5). `FindOptFusion()` takes two inputs M and k: $M = \{x_0, \ldots, x_{a-1}\}$, which is the fusion strategy for tensors from T_0 to T_{k-1}, and T_k is the first tensor to be fused for x_a.

Algorithm 5.1 uses $global\_opt\_fuse$ to store the best fusion strategy found so far and $local\_opt\_fuse$ to store the local best strategy from T_k to T_{N-1}. Given M, it first applies the second insight to fuse the first group beginning from T_k (Lines 9-16). The example illustrated in Figure 5.4b only considers the case that x_{a-1}'s communication begins right after its compression, but it is likely that its communication can be delayed by communication of its previous group. The algorithm replaces $g(0, i-1)$ in Expression (5.5) with $M.delay$, which denotes the difference between the completion time of compression and communication of x_{a-1}. The two cases of $M.delay$ are illustrated in Figure 5.5. We also denote $M.comp$ as the time duration from the beginning of backpropagation to the completion of x_{a-1}'s compression, as shown in Figure 5.5. Given M, both $M.delay$ and $M.comp$ can be calculated based on the timelines of backpropagation, compression, and communication, regardless of the strategy to fuse the remaining tensors.

Algorithm 5.1 finds j^* based on $M.delay$ (Lines 9-16) to skip the enumerations of tensors from T_k to T_{j^*-1}. It then uses the first insight to calculate the optimistic outcome (Lines 19-21). Similarly, the example shown in Figure 5.4a only considers the case that x_a's communication begins right after its compression. However, it is also likely that its communication can be delayed by x_{a-1}'s communication, as shown in Communication case 2 in Figure 5.5. The algorithm considers both cases and calculates the optimistic outcome. Algorithm 5.1 prunes the search if the optimistic outcome is already greater than the iteration time of the current best strategy. Suppose x_a is fused from T_k to T_i, `FindOptFusion()` recursively applies itself to find the local optimal fusion strategy from T_{i+1} to T_{N-1} (Lines 25-28). It updates $local\_opt\_fuse$ and $global\_opt\_fuse$ if the current strategy is better (Lines 29-36).

In practice, Algorithm 5.1 can use a heuristic to bootstrap $global\_opt\_fuse$ with

a relatively good fusion strategy. For example, it can partition a DNN model into multiple groups (e.g., two groups) with the same number of tensors.

Time complexity. The complexity of Algorithm 5.1 is $O(2^N)$ because it has to enumerate all fusion strategies in the worst case. Fortunately, the two pruning techniques can prune most of them and enable Cupcaketo find the optimal one quickly, as we will show in Section 5.4.3.

Theorem 5.1. *Algorithm 5.1 finds the optimal fusion strategy that minimizes the iteration time of a DDL job given a GC algorithm.*

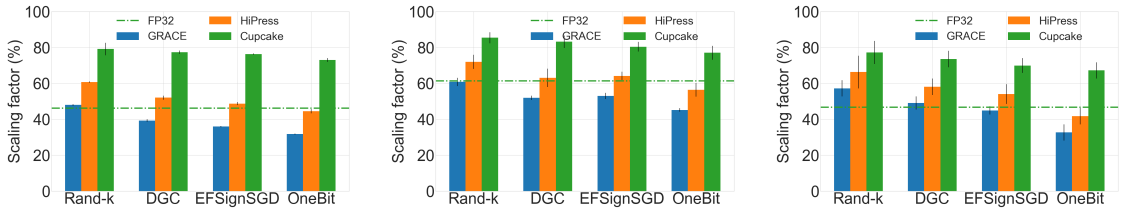
Proof. Algorithm 5.1 recursively invokes `FindOptFusion`(M, k). Let $n = N - k$, which is the number of tensors this function considers. We use induction on n to prove that the function finds the optimal fusion strategy from T_k to T_{N-1} given M.

Base case. When $n = 1$, the function only needs to examine one tensor and thus only one fusion strategy, which is the optimal one.

Inductive step. Assume that for any $1 \leq n \leq p$, `FindOptFusion`(M, k) returns the optimal fusion strategy from T_k to T_{N-1} given M. Consider $n = p+1$. Algorithm 5.1 divides the problem into $p + 1$ cases, where case i $(0 \leq i \leq p)$ fuses the first group from T_k to T_{k+i}.

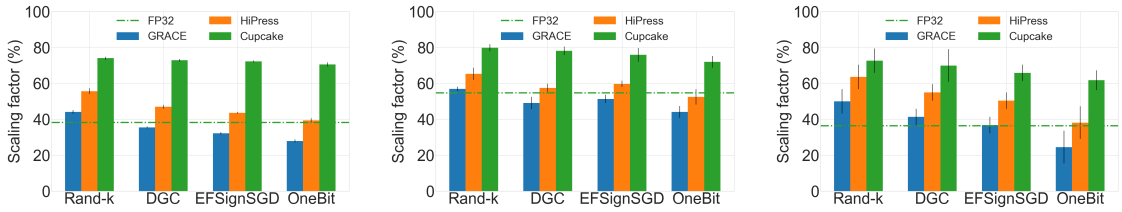
We first consider case i where $0 \leq i \leq p - 1$. The function invokes `FindOptFusion`$(M + fuse(k, k + i), k + i + 1)$, in which the number of tensors considered is $p - i \leq p$. Hence, it outputs the optimal strategy for case i based on the assumption. We then consider case $i = p$ and the first group is fused from T_k to T_{N-1}. This is the only fusion strategy and thus the optimal one. Because these cases are exclusive and cover the entire search space, Algorithm 5.1 finds the optimal fusion strategy for $n = p + 1$ by searching for the optimal one from these cases.

Algorithm 5.1 applies two pruning techniques to quickly find the optimal fusion strategy. The first one prunes the cases whose lower bounds are no better than the optimal found so far and it has no impact on the optimality. The second one prunes the cases whose first groups are fused from T_k to T_j, where $j < j^*$. Because they cannot advance communication to an earlier point than fusing from T_k to T_{j^*}, pruning these cases does not affect the optimality. □



(a) ResNet50 (b) ResNet101 (c) BERT-base

Figure 5.6 : The scaling factors of three DNN models running on a server with 8 GPUs connected by PCIe 3.0 ×16.



(a) ResNet50 (b) ResNet101 (c) BERT-base

Figure 5.7 : The scaling factors of three DNN models running on 64 GPUs in 8 servers connected by a 25Gbps network.

5.4 Evaluation

In this section, we will first show the performance improvement of Cupcakefor GC algorithms with sparsification and quantization. We then evaluate Time-to-Accuracy to demonstrate that Cupcakecan preserve the accuracy of these applied GC algorithms. At last, we show that Cupcakecan find the optimal fusion strategy quickly.

Setup. Two testbed setups are used for the evaluations. The first setup is the same as that described in Section 5.2. The server has 8 GPUs and they are connected by PCIe 3.0 ×16. The second setup has 8 GPU machines connected to a 25Gbps network. Each machine has 8 NVIDIA Tesla V100 GPUs (32 GB GPU memory) connected by NVLink and 48-core/96-thread processors (Intel Xeon 8260 at 2.40GHz). The server has an Ubuntu 18.04.4 LTS system and the software environment includes PyTorch-1.8.1, Horovod-0.22.1, CUDA-11.1, and NCCL-2.9.9.

Workloads. We validate the performance of Cupcakeon two types of machine learn-

ing tasks: computer vision and natural language processing (NLP). The models include ResNet50 over CIFAR10 [107] and ResNet101 [87] over ImageNet-1K [64]; BERT-base [67] over SQuAD [167]. These models are widely used as standard benchmarks to evaluate the scalability of DDL. The batch sizes for ResNet50 and ResNet101 are 32 and for BERT-base are 1024 samples.

Compression algorithms. We use four representative GC algorithms: Rand-k [192] and DGC [115] for sparsification with 99% sparsity, and EFSignSGD [100] and OneBit [186] for quantization. Error-feedback [100, 115] is applied to GC algorithms to preserve the model accuracy.

Baselines. We use Horovod [187] as the training baseline without GC (FP32). We use GRACE [226] and HiPress [38] as the two layer-wise baselines for applying GC to DDL. GRACE applies GC to all tensors in a model and HiPress only compresses tensors greater than a threshold, which is determined by the tensor size, network bandwidth, and compression overhead.

Metrics. Suppose the training speed with n GPUs is T_n. The scaling factor [234] is defined as $\frac{T_n}{nT_1}$. We use the scaling factor, Top-1 accuracy, and F1 score as evaluation metrics. The results for scaling factors are reported with an average of 100 iterations. We also report the standard deviation using the error bar because the training speed varies at times.

Allgather for communications. Allreduce is used for communications in FP32 [187, 156]. Existing frameworks' implementation of Allreduce requires tensors to be aligned and support element-wise aggregations. However, compressed tensors typically do not satisfy these requirements. For example, compressed tensors using Rand-k have different indices for selected elements, while those using Onebit cannot support addition. In contrast, the implementation of Allgather [199] has no such restrictions. It gathers tensors from all GPUs and allows for customized aggregation operations for compressed tensors. Therefore, we chose to use Allgather in our implementation to communicate compressed tensors [226, 214].

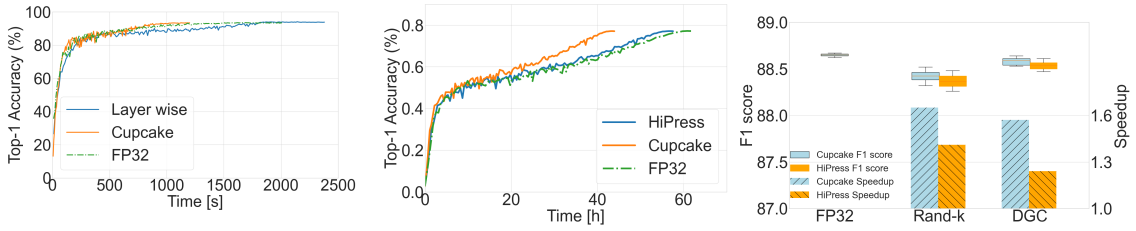
5.4.1 Training Speed Improvement

Figure 5.6 shows the scaling factors of the three DNN models running on a server with 8 GPUs connected by PCIe 3.0 ×16. The four compression algorithms are applied with Cupcakeand the two layer-wise baselines, respectively.

We can see from Figure 5.6 that applying GC in a layer-wise fashion can even harm the performance of DDL due to the costly compression overhead. The scaling factors of both ResNet50 and ResNet101 with GRACE compression are lower than those without any compression in most cases. HiPress outperforms GRACE because it avoids encoding small tensors and incurs less compression overhead. However, its improvement in the training throughput is just modest compared to training without GC. Applying Rand-k, DGC, and EFSignSGD to the training of BERT with HiPress can improve the training speed, but it still harms the training performance when the GC algorithm is OneBit.

In contrast, Cupcakesignificantly improves the training speed of DDL with GC algorithms for the three DNN models compared to FP32. For the training of ResNet50, it outperforms FP32 by up to 72% (apply Rand-k). It also outperforms GRACE and HiPress by up to 130% and 64% (apply OneBit), respectively. For ResNet101, Cupcakeoutperforms FP32, GRACE, and HiPress by up to 39%, 70%, and 37%, respectively. For BERT-base, it outperforms FP32, GRACE, and HiPress by up to 65%, 106%, and 61%, respectively.

Figure 5.7 shows the scaling factors of the three DNN models running on 8 servers (each has 8 GPUs) connected by a 25Gbps network. Because intra-machine communications are supported by NVLink, which can provide every GPU in total 1.2Tbps GPU-GPU bandwidth [94], the performance bottleneck is inter-machine communications. Therefore, tensors are not compressed for intra-machine communications and GC is applied for inter-machine communications only. Figure 5.7 shows that the speedups of Cupcakeover FP32 are up to 93%, 46%, and 103% for the training of ResNet50, ResNet101, and BERT-base, respectively. It also outperforms HiPress by up to 79%, 37%, and 58% for the training of the three models, respectively.



(a) ResNet50 (b) ResNet101 (c) BERT-base

Figure 5.8 : Cupcakeachieves almost the same model accuracy as no compression. DGC and EFSignSGD are applied to the training of ResNet50 over CIFAR10 and ResNet101 over Imagenet-1K, respectively. Both Rand-k and DGC are applied to the training of BERT-base over SQuAD.

5.4.2 Time-to-Accuracy Improvement

Because HiPress is always better than GRACE in terms of the training throughput, we compare Cupcaketo HiPress in this section. We train ResNet50 over CIFAR10 until convergence on a server with 8 GPUs connected by PCIe 3.0 ×16. The applied GC algorithm is DGC. As shown in Figures 5.8a, Cupcakecan achieve around 1.68× speedup over no compression (i.e. FP32), and 1.30× speedup over HiPress. The achieved Top-1 accuracy with Cupcakeis 93.2% (with HiPress is 93.1%), which is very close to the no-compression accuracy of 93.6%. We also train ResNet101 for 120 epochs on ImageNet-1K from scratch and apply EFSignSGD to the model training. Figure 5.8b shows that Cupcakeoutperforms no compression and HiPress by 1.32× and 1.25×, respectively. The achieved Top-1 accuracy with Cupcake, HiPress, and no compression is 76.7%, 76.6%, and 77.1%, respectively. In addition, we conduct a test following the methodology in [74] to fine-tune BERT-base for the question-answering task on SQuAD [167] for two epochs and repeat the experiments ten times. Figure 5.8c shows that Cupcakewith DGC can achieve around 1.65× speedup over no compression and it has almost the same F1 score as no compression.

	ResNet50	ResNet101	BERT-base
# of tensors	161	314	207
Algorithm 5.1	2.8 s	6.6 s	4.2 s
Only Pruning 1	15 s	68 s	32 s
Only Pruning 2	2.2 h	9.4 h	> 24 h
No Pruning	> 24 h	> 24 h	> 24 h

Table 5.1 : Running time of Algorithm 5.1.

5.4.3 Effectiveness of Cupcake

Computation time. We first measure the computation time of Algorithm 5.1 with the two pruning techniques. Note that the number of tensors in a DNN model, their sizes, and the computation time of backpropagation are measured in advance. The cost model of the communication time is determined by the network bandwidth. We also profile the encoding and decoding overheads of a GC algorithm, as shown in Figure 5.2, to model the compression time.

Table 5.1 shows that it only takes several seconds for Algorithm 5.1 to find the optimal fusion strategy for the three DNN models when training them on a server with 8 GPUs connected by PCIe 3.0 ×16. For example, the computation time is only a few seconds even for ResNet101 which has 314 tensors. However, the search cannot finish after running for 24 hours without the two pruning techniques, i.e., searching for the optimal strategy with brute force.

Compared to strawman solutions. We also compare Cupcake with the following two strawman solutions for tensor fusion.

- Bucket Fusion [112, 187]. It stores tensors in a buffer and fuses tensors in the buffer when their total size exceeds a threshold. We set different thresholds from 2 MB to 64 MB and use the best performance as its performance.
- Evenly Split. It evenly splits consecutive tensors into multiple groups for fusion and each group has the same number of tensors. We set the number of groups from 2 to 32 and use the best performance as its performance.

We apply DGC with three fusion strategies, Cupcake, Bucket Fusion, and Evenly Split, to three DNN models when training them on a server with 8 GPUs. Figure 5.9

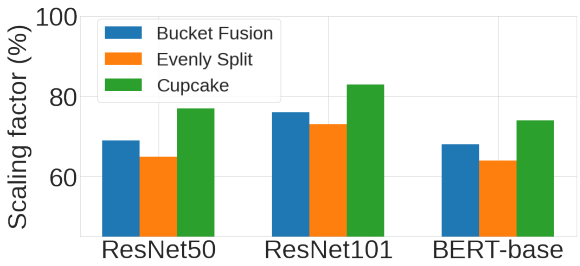


Figure 5.9 : The scaling factors of three DNN models running on a server with 8 GPUs. The GC algorithm is DGC.

displays their scaling factors. Both Bucket Fusion and Evenly Split outperform the layer-wise baselines thanks to the reduced compression overheads. Cupcake outperforms them by up to 1.12× and 1.18×, respectively. Cupcake searches for the optimal fusion strategy from the whole search space. We observe that the number of tensors and the size of the fused tensor vary a lot across groups in the optimal strategy for the three evaluated DNN models. However, the two strawman solutions limit the search space and constrain that each group has to have the same number of tensors or the same fused tensor size, leading to suboptimal fusion strategies.

Chapter 6

Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints

As discussed in Chapter 2, effective communication in both the data plane and the management plane holds a pivotal role in the endeavor to scale up DDL. In Chapters 3-5, we present systems that are designed to optimize data-plane communications to enhance the scalability of DDL. In this chapter, our focus shifts towards optimizing management-plane communications. Recall that DDL, particularly in the context of large language models, can involve a substantial number of GPUs and span several months for completion. Hundreds of software or hardware failures might occur during the training process. Unfortunately, existing solutions have fallen short of efficiently handling training failures, resulting in significant GPU resource wastage and dramatically slowing down the training progress. In response to these challenges, we introduce Gemini that enables fast failure recovery in the management plane for DDL and reduces the recovery overhead resulting from each failure from hours to minutes.

6.1 Introduction

Deep learning models have shown their ability to perform outstandingly on a spectrum of tasks including computer vision [87, 196], natural language processing [67, 204], etc. Recently, language models like ChatGPT [14] and GPT-4 [158] have drawn significant attention from both academia and industry with unprecedented performance as well as model size. PaLM [58] has 540 billion parameters, which is a 360× increase over GPT-2 [164] that was released three years earlier. This trend is still expediting because continued improvements have been observed from scaling the model sizes [58]. To train such a large model, failures are inevitably frequent because of the number

of involved accelerators (e.g., tens of thousands of GPUs) and the length of training time (in months). For example, OPT model training reports a failure frequency of twice a day [20]. The situation will get worse as the model size keeps growing.

Existing solutions cannot handle training failures efficiently. According to the report from OPT-175B training [233], about 178,000 GPU hours were wasted due to various training failures. As the failure frequency increases with the scale of the training, failures can dramatically slow down the training progress (up to 43% [121]). One major reason for such a significant overhead caused by failures is the inefficiency of checkpointing. Existing solutions rely on naïve checkpointing [71, 139, 5], which periodically saves the model states to a remote persistent storage system, for *failure recovery*, i.e., the process to fetch the latest checkpoint and resume training to the states right before a failure. Intuitively, a higher network bandwidth leads to shorter checkpoint retrieval times, and a higher checkpoint frequency reduces training progress loss in case of failures. However, existing solutions are restricted by the low bandwidth to remote persistent storage, resulting in significant failure recovery costs, i.e., taking up to tens of minutes to retrieve the checkpoint captured a few hours ago to resume the training. It is worth noting that the state-of-the-art large model training adopts a synchronized method to guarantee model quality [233, 144, 228], making it infeasible to only drop the training progress of the failed machine/device upon a failure to proceed without waiting for the failure recovery. Instead, it requires all machines/devices to roll back to the same checkpoint for failure recovery.

To reduce the prohibitively large failure recovery overhead, we present Gemini, a distributed training system that leverages the high bandwidth of CPU memory to achieve fast failure recovery in large model training via prompt checkpoint retrieval (in seconds) and high checkpoint frequency (ideally checkpoint for every training iteration). Gemini incorporates a hierarchical storage consisting of local CPU memory, remote CPU memory, and remote persistent storage, to store checkpoints. It leverages CPU memory to store checkpoints for failure recovery, and meanwhile stores checkpoints for other purposes in remote persistent storage. Gemini takes advantage of the optimized network connection for large-scale training to checkpoint model states in

the CPU memory of the compute cluster, which allows for a much higher frequency than existing solutions. It guarantees a 100% failure recovery and always fetches the available checkpoint from the fastest storage to minimize the recovery cost.

Checkpointing to CPU memory raises two questions that Gemini needs to address. First, how to maximize the probability of a successful failure recovery from CPU memory? The availability of checkpoints in CPU memory is not guaranteed upon a failure as the corresponding machines could be down. When the checkpoints are unavailable, in the worst case, the system has to resort to checkpoints stored in remote persistent storage, leading to significant failure recovery costs. The success rate of recovering a failure from checkpoints stored in CPU memory largely depends on how the checkpoints are placed among the CPU memory in different host machines. Gemini stores redundant checkpoints and proposes a placement strategy that maximizes the probability. We have proved that the strategy is optimal when the number of machines participating in training is divisible by the number of replicas and the strategy remains near-optimal with established bounds in other cases. Second, how to minimize the interference of checkpoint traffic with model training? Checkpointing model states to remote CPU memory shares the network resource with the regular training. Naïvely checkpointing to CPU memory will easily delay the training traffic which impacts the training throughput. Gemini designs a deliberate communication scheduling algorithm for interleaving these two types of traffic to minimize the interference on training throughput.

Gemini makes no assumptions about the underlying parallelism strategy [112, 166, 142, 189, 239] of the training system. It targets static and synchronous training with fixed computation resources, following the common practice for large model training in industrial settings [5, 214, 71, 58, 191, 216]. Elastic training [151, 224, 123] and asynchronous training [138, 232] are beyond the scope of this work. Gemini also makes no assumptions about the accelerator. In this chapter, we conducted experiments on NVIDIA GPUs, but the technique applies to other accelerators such as AWS Trainium [4], which remains for future work.

To sum up, this chapter makes the following contributions:

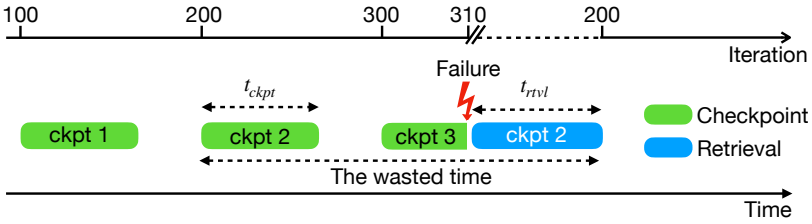


Figure 6.1 : An illustration of how failure recovery uses checkpoints. The checkpoint frequency f to the remote persistent storage is every 100 iterations (same as BLOOM [5]). A failure occurs at iteration 310 when the third checkpoint is incomplete. The failure recovery rolls back the model states to iteration 200 by retrieving the second checkpoint.

- To our knowledge, Gemini is the first system that takes advantage of CPU memory checkpointing to achieve efficient failure recovery in large model training.
- We design a provably near-optimal checkpoint placement strategy that maximizes the probability of a successful failure recovery from CPU memory.
- We propose a communication scheduling algorithm that pipelines checkpoint traffic across host machines to minimize its interference with model training.

We build Gemini atop DeepSpeed [170] and evaluate it with ZeRO-3 [166] on various large deep learning models using both Amazon EC2 p4d.24xlarge (NVIDIA A100 GPUs) and p3dn.24xlarge (NVIDIA V100 GPUs) instances. Compared to existing solutions [139, 5], Gemini reduces the checkpoint retrieval time by up to 250× and improves the checkpoint frequency by up to 8×. Hence, Gemini achieves a faster failure recovery by more than 13× without incurring overhead on training throughput.

6.2 Motivation

6.2.1 Failure Recovery in Model Training

We have noticed a significant waste of computation resources caused by large model training failures. The *model states*, i.e., the learnable parameters and the optimizer states, are resided in GPU memory during training. When a failure occurs, the model states must be rolled back to previous states by retrieving the latest checkpoint for failure recovery. For example in Figure 6.1, a failure occurs at iteration 310, but

the latest available checkpoint is at iteration 200. After the failure recovery, the training progress from iteration 200 to 310 is lost. Additionally, retrieving the latest checkpoint incurs overhead during the failure recovery process.

We define *wasted time* as the sum of the time spent on the lost training process before a failure and the time for retrieving the latest checkpoint during a failure recovery. As illustrated in Figure 6.1, the wasted time describes the timespan of a paused training process due to a failure, i.e., the time of computation resource wasted in terms of the training process. It is determined by three factors:

- *checkpoint time*, which is the time to finish a checkpoint of model states. We denote checkpoint time as t_{ckpt} in Figure 6.1.
- *checkpoint frequency*, which determines how frequently the training system checkpoints model states to storage system. We denote checkpoint frequency as f.
- *retrieval time*, which is the time to retrieve the latest complete checkpoint*. We denote retrieval time as t_{rtvl}, shown in Figure 6.1.

In this section, we use the average wasted time as the main metric to evaluate the performance of a checkpointing solution, because a failure may occur at any time and the wasted time varies. The best case is that a failure occurs right after the completion of a checkpoint and the wasted time is $t_{cpkt} + t_{rtvl}$. The worst case is that a failure occurs right before the completion of a checkpoint and the wasted time is $t_{cpkt} + 1/f + t_{rtvl}$. Assuming failures are evenly distributed between two consecutive checkpoints, the average wasted time (denoted as T_{wasted}) can be expressed as

$$T_{wasted} = t_{ckpt} + \frac{1}{2f} + t_{rtvl}. \tag{6.1}$$

In addition, we have the following constraint:

$$1/f \geq max(t_{ckpt}, T_{iter}), \tag{6.2}$$

where T_{iter} is the iteration time. One checkpoint cannot start until its previous checkpoint completes, and there is no need to have multiple checkpoints within one iteration as the model states are updated once every iteration.

*We exclude the overheads to fix failures and replace machines in the wasted time because they are not caused by checkpoints.

To reduce the wasted time, it is critical to reduce checkpoint time t_{ckpt} to enable a higher checkpoint frequency f, and the optimal frequency f is every iteration $1/T_{iter}$.

6.2.2 Limitations of Existing Solutions

Existing solutions fail to achieve high checkpoint frequency for failure recovery due to the remote persistent storage system usage. They checkpoint the model states at a particular frequency and persist checkpoints in a remote persistent storage system [139, 184]. In common practice, existing solutions checkpoint model states at a low frequency, e.g., every three hours in BLOOM training [5], to reduce the required storage capacity. A few hours of computation resources are wasted when a failure occurs. Considering thousands of GPUs involved in training and hundreds of failures experienced during training, the total computation resource waste is significant, and the training time slowdown can be up to 43% [121]. It is infeasible to arbitrarily increase the checkpoint frequency because checkpoint frequency is bottlenecked by the bandwidth of the remote persistent storage [71]. For example, it takes 42 minutes to checkpoint the model states of MT-NLG [191] to the remote persistent storage when the bandwidth is 20Gbps. According to Equation (6.1), the average wasted time for failure recovery is 105 minutes, which makes the training system less efficient.

6.2.3 The Opportunity and Challenges

Minimizing the wasted time for failure recovery is crucial for enhancing the system efficiency of distributed training, especially large model training. We next explore the opportunity to achieve this goal and discuss the identified challenges.

Checkpointing to CPU memory. The low bandwidth severely restricts the frequency of checkpointing to remote persistent storage. We observe that the CPU memory in GPU machines is sufficient to store a few checkpoints. Table 6.1 compares the GPU and CPU memory in popular GPU instances in public clouds for large model training, demonstrating that the CPU memory is much larger than the GPU memory. This observation provides a great opportunity for Gemini to store the latest checkpoint in CPU memory. Gemini can leverage the network connecting

Instance type	Cloud	GPU	GPU memory	CPU memory
p3dn.24xlarge [21]	AWS	8 V100	8 × 32 GB	768 GB
p4d.24xlarge [22]	AWS	8 A100	8 × 40 GB	1152 GB
ND40rs_v2 [17]	Azure	8 V100	8 × 32 GB	672 GB
ND96asr_v4 [18]	Azure	8 A100	8 × 40 GB	900 GB
n1-8-v100 [16]	GCP	8 V100	8 × 32 GB	624 GB
a2-highgpu-8g [16]	GCP	8 A100	8 × 40 GB	640 GB
DGX A100 [19]	NVIDIA	8 A100	8 × 80 GB	2 TB

Table 6.1 : The CPU memory size is much larger than the GPU memory size in one GPU machine provided in public GPU clouds.

GPU instances for checkpointing. Because this network is optimized for training, its bandwidth is much higher than the bandwidth of the remote persistent storage [22]. Therefore, Gemini can achieve a much higher checkpoint frequency for failure recovery than existing solutions.

One concern is that the CPU memory size is insufficient to store the history of checkpoints for purposes other than failure recovery, such as transfer learning [155] and model debugging [52, 71]. To address this concern, Gemini decouples checkpoints for different purposes. It only stores the checkpoints for failure recovery in CPU memory, while storing checkpoints for other purposes in remote persistent storage.

Challenges. Checkpointing to CPU memory allows for a much higher frequency than existing solutions, thereby reducing the wasted time. However, this approach also presents new challenges.

How to maximize the probability of failure recovery from checkpoints stored in CPU memory? Although checkpointing to CPU memory enables a high frequency, the availability of checkpoints in CPU memory cannot be guaranteed when failures occur. In the cases of unavailable checkpoints in CPU memory, we have to fall back to using the low-frequency checkpoints stored in the remote persistent storage for failure recovery, causing significant wasted time.

How to minimize the interference of checkpoint traffic with model training? When checkpointing to CPU memory, communication traffic for training and checkpointing have to share the same network. Without careful design, checkpoint

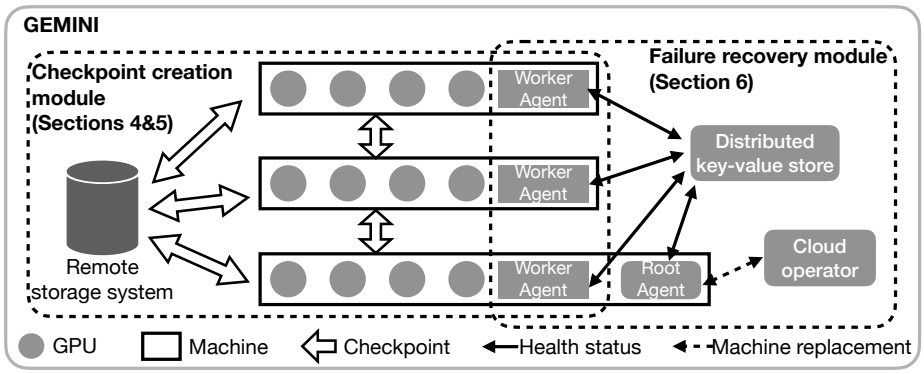


Figure 6.2 : The system architecture of Gemini. Gemini consists of checkpoint creation and failure recovery modules. In the checkpoint creation module, each worker agent controls checkpoint destinations and schedules checkpoint communications. In the failure recovery module, worker agents update machines' health statuses in the distributed key-value store. The root agent periodically checks the health statuses in the distributed key-value store, interacts with the cloud operator to replace failed machines as needed, and guides the checkpoint retrieval for failure recovery.

traffic can interfere with training traffic and harm training throughput. The interference overhead is non-negligible because it negatively impacts every iteration. This can significantly diminish the benefits gained from checkpointing to CPU memory.

6.3 Gemini Overview

We propose Gemini, which achieves a high checkpoint frequency, even every iteration, to optimize the failure recovery overhead in distributed training. It minimizes the wasted time by checkpointing to CPU memory and addresses the two aforementioned challenges. Figure 6.2 illustrates Gemini's architecture that consists of two modules: 1) a checkpoint creation module (Section 6.3.1); and 2) a failure recovery module (Section 6.3.2). The two modules cooperate to resume training once a failure occurs.

6.3.1 Checkpoint Creation Module

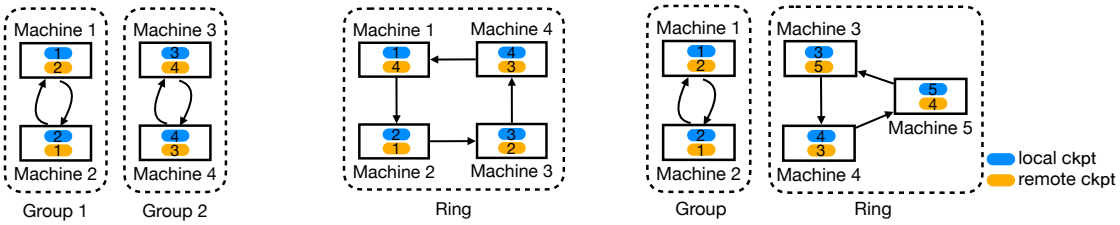
Gemini uses a decoupled and hierarchical storage design for checkpointing. In Gemini, the checkpoint creation module stores the checkpoints of each GPU machine to different destinations, including local CPU memory, remote CPU memory on other

machines, and remote persistent storage. The checkpoint creation module stores the checkpoints for failure recovery in local and remote CPU memory. These checkpoints are managed by Gemini's checkpoint creation module and are transparent to users. On the other hand, checkpoints for other purposes, such as transfer learning [155] and model debugging [71], are stored in remote persistent storage and managed by users. During failure recovery, checkpoints are first retrieved from local CPU memory and then remote CPU memory if unavailable in local CPU memory. If both local and remote CPU memory checkpoints are unavailable, Gemini retrieves checkpoints from remote persistent storage.

As illustrated in Figure 6.2, each training machine has a Gemini worker agent for checkpointing to CPU memory. Where to place checkpoints for failure recovery on CPU memory determines the failure recovery capacity. To maximize the probability of failure recovery from checkpoints in CPU memory, we propose a provably near-optimal checkpoint placement strategy for checkpointing to CPU memory (Section 6.4). Gemini determines the checkpoint placement strategy when training is initialized. During runtime, the Gemini worker agent on each machine communicates checkpoints from GPU memory to CPU memory based on the placement strategy and checkpoint frequency. To minimize or even eliminate the interference of checkpoint traffic with model training, we propose a traffic scheduling algorithm that pipelines checkpoint traffic and interleaves it with training traffic (Section 6.5).

6.3.2 Failure Recovery Module

Gemini's failure recovery module has four components: a group of Gemini worker agents, a Gemini root agent, a distributed key-value store, and a cloud operator. Worker agents monitor their own machine's health status and update it in the distributed key-value store [78, 73, 15]. The unique root agent runs on a regular training machine with a worker agent. The training machine with the root agent running is called the root machine. The root agent periodically checks the health status of each training machine from the distributed key-value store. The cloud operator manages the training computation resources and replaces failed machines with healthy ones as



(a) Group placement. (b) Ring placement. (c) Mixed placement.

Figure 6.3 : Illustrations of the mixed checkpoint placement strategy.

needed.

If the root agent detects a training machine failure, the root agent takes corresponding actions based on failure types (Section 6.6). For example, when a training machine replacement is needed, the root agent interacts with the cloud operator to complete the machine replacement and guides the replaced machine where to retrieve its checkpoints.

Worker agents also periodically check the root machine's health status in the distributed key-value store. A root machine failure is detected when the root machine's health status has not been updated for a predefined time threshold. In the case of a root machine failure, one alive worker machine is promoted as the root machine, and one new worker machine is initialized to replace the failed one. Gemini relies on the leader election method in the distributed key-value store [149, 108] for the root machine selection.

6.4 Checkpoint Placement to CPU Memory

To reduce the wasted time, Gemini writes checkpoints to CPU memory to achieve high frequencies. However, the checkpoints stored in CPU memory may become invalid for recovery when some GPU machines are disconnected from training. In such cases, Gemini has to fetch from remote persistent storage to perform recovery, leading to significant wasted time. Adding more checkpoint replicas reduces the possibility of unavailable checkpoints in CPU memory, but it also increases CPU memory usage and network bandwidth competition with training traffic. In addition to the number of replicas, our research has revealed that the checkpoint placement strategy, i.e., where

Algorithm 6.1 Mixed checkpoint placement strategy

Input: N is the number of GPU machines and m is the number of checkpoint replicas.

Output: The group list $\mathcal{G}$ and the strategy.

```
Function placement_strategy(N, m):
    G = []
    g = ⌊N/m⌋
    for i ← 0 to g − 1 do
        G = []
        for j ← 1 to m do
            G.add(m × i + j)
        end
        G.add(G)
    end
    strategy = "group"
    if N is not divisible by m then
        strategy = "mixed"
        // add remaining machines to the last group
        for j ← g × m + 1 to N do
            G[−1].add(j)
        end
    end
    return G, strategy
```

to store the checkpoint replicas, also affects the possibility, as shown in Figure 6.3. Hence, we aim to identify the best placement strategy that maximizes the probability of failure recovery from CPU memory, given a specific number of replicas. This problem can be formulated as follows.

Problem 6.1. *Given N machines and m checkpoint replicas, what is the optimal placement strategy to distribute the m replicas among the N machines to maximize the probability of failure recovery from CPU memory?*

We design a mixed placement strategy described in Algorithm 6.1 to solve Problem 6.1. The inputs of the algorithm are the number of machines N, and the number of replicas m. The output is the machine group assignment and the specific strategy. If the number of machines N is divisible by the number of replicas m, we will apply a

group placement strategy for all machines participating in training. The N machines are divided into N/m groups and each group has m machines. During training, each machine broadcasts its checkpoints to the $m-1$ machines in the same group. It also writes one checkpoint to its own CPU memory as a local replica, which is one tier in Gemini's hierarchical checkpoint solution. Otherwise, when N is not divisible by m, we split the N machines into $\lfloor N/m \rfloor$ groups and apply the group placement strategy to the first $\lfloor N/m \rfloor - 1$ groups. For the last $N - m(\lfloor N/m \rfloor - 1)$ machines, we apply a ring placement strategy, in which each machine writes the checkpoints from GPU memory to its local CPU memory and also sends checkpoints to the consecutive $m-1$ machines in the ring from its left hand. Regardless of the placement strategy employed, Gemini copies the checkpoint from GPU memory to the local CPU memory and treats it as a local replica. It has two advantages: 1) it can mitigate the network bandwidth contention with training traffic; and 2) for certain failure types, e.g., software failures (refer to Section 6.6.1), Gemini can directly resume training from the local replica to accelerate checkpoint retrieval. We pivot the group placement strategy because it exhibits a greater likelihood of recovering from CPU memory compared to the ring placement strategy with the same number of replicas. We also have Theorem 6.1 for the performance of the mixed placement strategy.

Theorem 6.1. *To address Problem 6.1 for checkpoint placement:*

1. *When N is divisible by m, the mixed placement strategy (equals group placement strategy) is the optimal placement strategy.*
2. *When N is not divisible by m, the mixed placement strategy minimizes the checkpoint communication time. Its failure recovery probability from CPU memory is near-optimal and the gap is bounded by $(2m-3)/\binom{N}{m}$.*

Proof. We first introduce two observations for checkpoint placements. (1) The optimal strategy requires m machines to store the m checkpoint copies of each machine to maximize the recovery probability. If there are only m' machines to store the m copies, where $m' < m$, it is equivalent to the strategy with only m' copies for recovering failures from CPU memory. (2) The optimal strategy requires Machine i to store one copy of its own machine checkpoint to minimize the checkpointing time. If so,

each machine only needs to send out $m-1$ checkpoint copies. Otherwise, it has to send out m copies and leads to a higher checkpointing time.

The checkpointing communication time with Group strategy is minimized because each machine sends out and receives $m-1$ checkpoint copies, no matter whether N is dividable by m or not. We next analyze the probability that Gemini can recover failures from CPU memory.

Suppose there are k machines disconnected at the same time. Gemini can certainly recover failures from CPU memory when $k < m$ because there are m copies in m instances. We mainly discuss the case that $k = m$ here because the failure rate with $k+1$ machines disconnected simultaneously is much lower than with k machines disconnected simultaneously in practice.

For Machine i, we denote the set of machines that store its machine checkpoints as s_i and it has $\binom{N-1}{m-1}$ possible combinations because one checkpoint replica is stored locally. Then a strategy can be expressed as $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$. Because it is possible that $s_i = s_j$ when $i \neq j$, we define $\mathcal{S}' = unique(\mathcal{S})$ and $n = |\mathcal{S}'|$. The union of these n sets in $\mathcal{S}'$ must cover all the N machines because each machine stores a local checkpoint.

We denote the set of the m disconnected machines as s_d and it has $\binom{N}{m}$ possible combinations. Note that $k = m$ in our analysis. Gemini cannot recover training from CPU memory when s_d is an element in $\mathcal{S}'$. If so, all the m copies of a machine checkpoint get lost and the model checkpoints stored in CPU memory become incomplete and invalid for failure recovery. The probability that s_d is an element in $\mathcal{S}'$ is $n/\binom{N}{m}$, which linearly increases with n.

Probability upper bound. The upper bound of the probability that training can be recovered from checkpoints stored in CPU memory is $1 - \lceil \frac{N}{m} \rceil / \binom{N}{m}$ because $n \geq \lceil N/m \rceil$. If n $< \lceil N/m \rceil$, the size of the union of the n sets is at most $nm < N$. It contradicts the requirement that the n sets must cover the N machines.

When N is divisible by m. Group placement strategy can achieve the upper bound. Machines in the same group have the same set of machines to store their checkpoints. Because there are N/m groups, the number of unique sets in $\mathcal{S}$ is N/m.

The probability that s_d is an element in $\mathcal{S}'$ is $\frac{N}{m}/\binom{N}{m}$, which is the lower bound. Therefore, we can conclude Group placement strategy is optimal for Problem 6.1 when N is divisible by m.

When N is not divisible by m. For simplicity, we rewrite $N = pm + q$, where $1 \leq q \leq m-1$. For the first $\lfloor N/m \rfloor - 1 = p - 1$ groups, machines in the same group have the same set of machines to store their checkpoints. For the last group, each machine has a distinct set of machines to store its checkpoints and there are $m + q$ unique sets. Therefore, the total number of unique sets in $\mathcal{S}$ is $m + q + p - 1$. Since the lower bound of the number of sets is $\lceil N/m \rceil = p + 1$, the gap is

$$\begin{aligned} gap &= (m + q + p - 1) - (p + 1) \\ &= m + q - 2 < 2m - 3, \end{aligned} \tag{6.3}$$

since $1 \leq q \leq m-1$. It suggests that the gap between the upper bound and probability with the mixed placement strategy is bounded by $(2m-3)/\binom{N}{m}$. Because $N \gg m$ and m is practically very small, the probability is very close to the upper bound. □

Figure 6.3a illustrates an example of the group placement strategy with $N = 4$ and $m = 2$. There are two groups and each group has two machines. Each machine has a local checkpoint, i.e., its local machine checkpoint, and a remote checkpoint, i.e., the checkpoint from the other machine in the same group. Figure 6.3b illustrates an example of the ring placement strategy with $N = 4$ and $m = 2$, in which all machines form a ring structure for checkpointing to CPU memory. Assume two machines fail at the same time. With the group placement strategy, training can recover failures from CPU memory except Machines 1 and 2, or Machines 3 and 4 fail simultaneously (a total of two possible cases). However, with the ring placement strategy, the concurrent failures of any two consecutive machines (four possible cases in total) will result in the loss of both replicas of a checkpoint stored in CPU memory. Consequently, the probability that training has to fetch remote persistent storage for failure recovery with the group placement strategy is 50% lower than that with the ring placement strategy. Figure 6.3c also illustrates an example of the mixed placement strategy with

$N = 5$ and $m = 2$, in which the first two machines form a group and the last three machines form a ring.

With the group placement strategy, we calculate the probability that Gemini can recover failures from CPU memory using Corollary 6.1.

Corollary 6.1. *When N is divisible by m and k machines are disconnected simultaneously, the probability that Gemini can recover failures from CPU memory is*

$$\begin{cases} \Pr(N,m,k) = 1, & \text{if } k < m \\ \Pr(N,m,k) \geq \max\{0, 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}\}, & \text{if } m \leq k \leq N \end{cases} \tag{6.4}$$

Proof. Gemini can certainly recover failures when $k < m$ because there are available checkpoint replicas in at least one of the machines. We then consider $m \leq k \leq N$.

With Algorithm 6.1 there are N/m groups in $\mathcal{G}$ after the group placement policy. When k machines fail at the same time, if there exist m failed machines forming a group that is an element of $\mathcal{G}$, it indicates that the checkpoints stored in CPU memory become incomplete and training has to recover from the remote persistent storage.

We first consider the case $m \leq k < 2m$. The number of combinations to choose k machines from N machines is $\binom{N}{k}$. The number of combinations that lead to incomplete checkpoints in CPU memory is $\frac{N}{m}\binom{N-m}{k-m}$. Therefore, the probability probability that Gemini can recover failures from CPU memory is

$$Pr(N,m,k) = 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}, \text{if m} \leq \text{k} < \text{2m}. \tag{6.5}$$

We then consider the case $k \geq 2m$. When we use the same method for $m \leq k < 2m$ to count the number of combinations, some combinations are counted more than once and the total number of combinations is less than $\frac{N}{m}\binom{N-m}{k-m}$. Therefore, the probability probability that Gemini can recover failures from CPU memory is

$$\Pr(N,m,k) > \max\{0, 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}\}, \text{if } k \geq m. \tag{6.6}$$

We then have Corollary 6.1 by combining the two cases together. □

According to Corollary 6.1, when the number of machines N is 16, the number of replicas m is 2, and the failure machine k is 2, the probability is 93.3% and it increases with N. It means that with two checkpoint replicas, Gemini can resume training from CPU memory in most cases.

6.5 Minimizing Training Interference

Frequently writing checkpoints to remote CPU memory might hinder overall training performance due to potential network bandwidth competition with training traffic. Our primary objective is to minimize the wasted time without compromising training performance. In this section, we will explain how Gemini mitigates the interference caused by frequent checkpointing. We begin by examining the possibility of minimizing the impact of checkpointing to CPU memory on model training (Section 6.5.1), then discussing the challenges and the approaches we took to overcome them (Section 6.5.2). Finally, we elaborate on the specific algorithm and mechanism we used in Gemini (Section 6.5.3 & 6.5.4).

6.5.1 Traffic Interleaving

Modern distributed training, such as large model training, relies on collective communication operations for synchronization. For example, in ZeRO [166], each GPU needs to fetch the parameters of each layer from other GPUs before its computation in both forward and backward passes. These communication operations can block computation when the parameters of a layer are not ready but the computation of the previous layer has been completed. We denote the communication traffic for model computation, including gradient synchronization and parameter fetching, as *training traffic*. An example of training traffic during model computation is shown in Figure 6.4a. When checkpointing to remote CPU memory, its traffic, denoted as *checkpoint traffic*, shares the same network as training traffic, resulting in potential network resource contention that may delay training traffic and hinder computations. When performing checkpointing at the start of subsequent iterations, it blocks the training process and incurs non-negligible overheads for model training, as illustrated

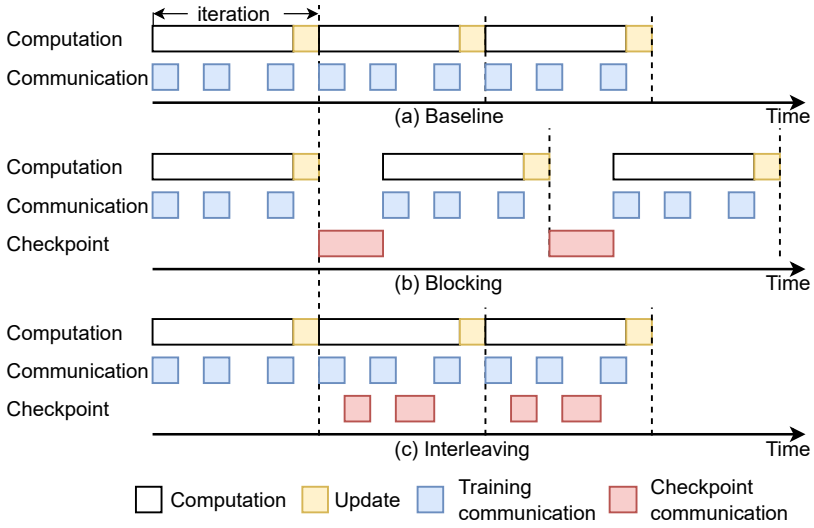


Figure 6.4 : Interleaving training communications and checkpoint communications can minimize the interference.

in Figure 6.4b. This significantly negates the benefits gained from the reduced wasted time by checkpointing to CPU memory. Hence, Gemini must carefully orchestrate the checkpoint traffic to minimize its interference with training throughput. Fortunately, we observe that the network has idle timespans overlapped with computation and this naturally occurs in large model training. This observation provides a great opportunity for Gemini to insert checkpoint traffic in these idle timespans and overlap checkpoint communications with computation, as shown in Figure 6.4c.

6.5.2 Difficulties and Approaches

Gemini needs to write checkpoints from local GPU memory to CPU memory on remote machines. It first uses GPU-to-GPU communications to send checkpoints between machines for interleaving checkpoint traffic with training traffic, which also uses direct GPU-to-GPU communications [178, 110] among machines in large model training [94, 235, 162]. After that, it transmits the checkpoints from GPU memory on remote machines to their CPU memory with GPU-to-CPU copy. This design allows scheduling training traffic and checkpoint traffic in the application layer without relying on the network layer. Gemini orchestrates both types of traffic by leveraging existing inter-GPU communication libraries, such as NCCL [3], in a unified manner.

However, this design raises two practical difficulties.

Difficulty: Extra GPU memory consumption. Naïvely sending a whole checkpoint from a local GPU to a remote GPU consumes a significant amount of GPU memory, which may trigger GPU out-of-memory (OOM) and crash the training process, as shown in Figure 6.5b. The checkpoint size is huge in large model training. For example, the checkpoint size of GPT2-100B [235] on each GPU is 9.4GB. Furthermore, most GPU memory has already been used to store model parameters, gradients, and intermediate results. Therefore, a remote GPU is unlikely to accommodate a whole checkpoint during large model training.

Approach: Partitioning checkpoint. Although a whole checkpoint with several GBs is too large for a remote GPU, we observe that each GPU usually has a few hundred of memory available during training based on our profiling results. Gemini first reserves a small GPU memory buffer for checkpoint communications, then partitions a whole checkpoint into small chunks and transfers the small chunks separately. The remote GPU moves the received chunk to CPU memory once a communication completes making the buffer available for the next communication. Figure 6.5c illustrates the process of partitioning checkpoint.

Difficulty: Local GPU-to-CPU copy overhead. Checkpointing to remote CPU memory includes a procedure of GPU-to-CPU copy on the receiver side. The sender cannot transit new checkpoint chunks until the GPU-to-CPU copy is complete, causing communication bubbles in the GPU-to-GPU communication timeline, as shown in Figure 6.5c. Since the GPU-to-CPU memory copy bandwidth is comparable to the inter-machine GPU-to-GPU network bandwidth [†], the bubble time could be close to the inter-machine GPU-to-GPU checkpoint communication time, which may exacerbate the interference with model training.

Approach: Pipelining checkpoint transmission. Gemini uses a pipeline mechanism to allow checkpoint communications to fully leverage the network idle timespans. It splits the reserved GPU memory buffer into multiple sub-buffers and partitions the checkpoints into chunks that fit into these sub-buffers. Gemini alternatively uses

[†]We measured both bandwidths in p4d.24xlarge instances in AWS and both are around 400Gbps.

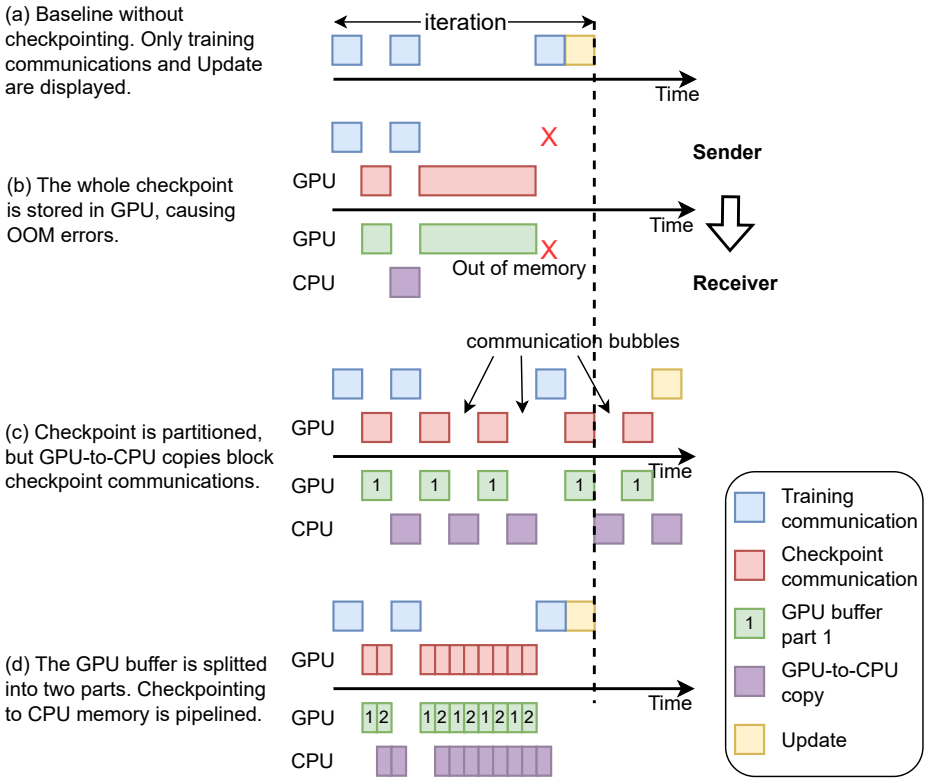


Figure 6.5 : Different schemes for interleaving training and checkpoint traffic.

these sub-buffers for transferring checkpoint chunks. When copying a chunk from GPU to CPU memory, Gemini can simultaneously receive a new checkpoint chunk using GPU-to-GPU communication in a separate sub-buffer. Figure 6.5d illustrates an example with two sub-buffers. Inter-machine GPU-to-GPU communication overlaps with local GPU-to-CPU memory copy and the idle timespans are fully utilized for checkpoint traffic.

6.5.3 Checkpoint Partition Algorithm

Gemini uses a checkpoint partition algorithm illustrated in Algorithm 6.2 to partition checkpoints for transmission pipelining. Given the set of profiled network idle timespans $\mathcal{T} = \{t_1, t_2, \ldots, t_d\}$ (discussed in Section 6.5.4), Algorithm 6.2 generates a scheduling of checkpoint partitions. Suppose there are p GPU buffers in Gemini and the size of each buffer is R/p, where R is the total reserved GPU memory size. Suppose there are m checkpoint replicas, and $m-1$ replicas are sent to the remote CPU

Algorithm 6.2 Checkpoint Partition Algorithm

Input: $\mathcal{T} = \{t_1, t_2, \ldots, t_d\}$ is the set of idle timespans. C is the size of a checkpoint and $m-1$ is the number of checkpoints for communications. There are p buffer parts and the size of each part is R/p. B is the network bandwidth. $\mu \in (0, 1)$ is a coefficient for the variance of idle spans across iterations. $f(s)$ is the communication time for a checkpoint chunk with size s

Output: The checkpoint partitions.

```
115 Function checkpoint_partition():
116     t[d] = +∞
117     partitions = []
118     cpkt_id = 0
119     remain_size = C
120     foreach t ∈ T do
121         remain_span = μ × t   while remain_span > 0 do
122             if remain_span ≥ f(R/p) then
123                 size = R/p
124             else
125                 size = max{0, (remain_span − α)B}
126             end
127             size = min{remain_size, size}   if size > 0 then
128                 remain_size = remain_size − size
129                 remain_span = remain_span − f(remain_size)
130                 partitions.add(size)
131             end
132             if remain_size == 0 then
133                 if cpkt_id < m − 1 then
134                     cpkt_id = cpkt_id + 1
135                     remain_size = C
136                 else
137                     return partitions
138                 end
139             end
140         end
141     end
142     return partitions
```

memory while one is stored locally. Suppose the time length of sending a partition of size s to a receiver is $f(s) = \alpha + s/B$, where α is the startup time for transmission and B is the network bandwidth [235, 36, 199].

Algorithm 6.2 uses a coefficient $\mu \in (0, 1)$ to consider the variance of the profiled timespans across iterations (Line 7). Because the size of each buffer is R/p, the maximum checkpoint chunk size is also R/p. The algorithm checks how many chunks it can insert in each idle timespan with multiple rounds. In each round, it compares $f(R/p)$ with the remaining idle timespan ($remain\_span$). If $remain\_span$ is greater, it sets $size$ to the maximum chunk size R/p (Lines 9-10); otherwise, it sets the size to the amount of traffic volume that can be transmitted during $remain\_span$ (Line 11). It then compares $size$ with the remaining checkpoint size ($remain\_size$) and takes the smaller one as the chunk size (Line 14). It accordingly updates $remain\_span$ and $remain\_size$ for the next round (Lines 15-19). When $remain\_size$ equals zero, the algorithm finishes the partition of one checkpoint. If there are multiple checkpoint replicas for a higher failure recovery rate from CPU memory, the algorithm resets $remaining\_size$ as the checkpoint size and determines the partition for the new checkpoint again (Lines 21-23). The algorithm returns $partitions$ after all the checkpoints are partitioned.

Our evaluation in Section 6.7 shows that for all the evaluated models, Algorithm 6.2 allows Gemini to fully utilize the network idle timespans and enables it to perform checkpointing at the frequency of every iteration without interfering with training.

Finish checkpointing within an iteration. However, it is still possible that the total time required for checkpointing cannot be fit in the available network idle timespans. In such a scenario, Gemini places the unfinished checkpoint traffic in the last idle timespan, as Algorithm 6.2 sets the interval of the last idle timespan as positive infinity (Line 2). Although checkpoint communications hinder the update operation and prolong the iteration time in this case, Gemini can reduce the checkpoint frequency to amortize the incurred overhead.

Move checkpoints from GPU to local CPU. Each machine also needs to copy

its checkpoint from GPU memory to its local CPU memory according to our placement strategy discussed in Section 6.4. This checkpoint copy incurs no traffic across machines. Gemini also partitions this replica and overlaps its GPU-to-CPU copy with communications for training traffic. In this way, there is no interference between the local GPU-to-CPU copy of its own checkpoint and other checkpoints.

6.5.4 Online Profiling

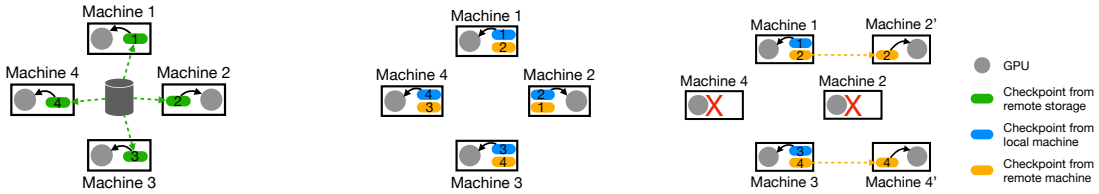
Gemini adopts online profiling for the first several iterations of training, e.g., 20 iterations in our implementation, without checkpointing in order to capture the network idle timespans during model training. It timestamps the start and the end time of all communication operations in an iteration to derive the timeline of communication traffic. Gemini then obtains the average time interval of each idle timespan for subsequent checkpoint traffic scheduling. We observed that the profiled timeline remains almost constant across iterations, which is consistent with previous studies [217, 214, 234]. The normalized standard deviation of the measurements is less than 10%. Gemini uses these idle timespan intervals to determine the checkpoint partitions in each idle timespan according to Algorithm 6.2 described in Section 6.5.3.

6.6 Resuming Training from Failures

Gemini achieves high-frequency checkpoints with the mixed checkpoint placement and the traffic interleaving algorithm. In this section, we will explain how Gemini uses the checkpoints to resume training when failures occur. We first define our failure classification and then describe how Gemini resumes training accordingly.

6.6.1 Failure Types

There are various failures that can occur during the training of large models [92, 152, 197, 203] and these failures have different root causes and consequences. We categorize these failures into two types from the perspective of recovery: software failures and hardware failures, following the literature [92, 152, 197, 203, 79, 202].



(a) Existing solutions for failure recovery. All checkpoints are always retrieved from the remote persistent storage.

(b) Gemini for software failures. Checkpoints are at local and the retrieval time is negligible.

(c) Gemini for failures with two machines replaced. The newly added machines retrieve checkpoints from alive machines.

Figure 6.6 : Illustrations of different mechanisms to recover training with four machines from different failures.

Software failures are caused by bugs in software or errors in data. Software failures can be fixed by restarting the training process without replacing hardware.

Hardware failures are caused by hardware issues, such as GPU malfunctions and network failures. For example, bit corruptions induced by radiation can cause double bit error, leading to data corruptions [203, 92]. The network links and switches that connect GPU machines can fail [79, 198], disconnecting them from training. These failures can occur in a single machine or multiple machines simultaneously. The training cluster typically detects problematic machines and then replaces them with healthy ones before resuming training.

6.6.2 Failure Recovery Mechanisms

Existing checkpointing solutions [71, 233, 184] make no distinction between *software failures* and *hardware failures*. As shown in Figure 6.6a, they always retrieve the checkpoints from the remote persistent storage regardless of the failure type, resulting in costly wasted time. In this subsection, we will present the recovery mechanisms of Gemini for both types of failures, respectively.

Software failures recovery. Recovering from software failure does not require fetching checkpoints from other machines, and the training configurations (e.g., the rank ID of the machine) remain the same. When a software failure occurs, the training process is interrupted, but the hardware remains healthy and all checkpoints stored in CPU memory are still accessible. Because each machine stores a replica of its own

checkpoint, all machines can directly recover training from their local checkpoints, as shown in Figure 6.6b.

Hardware failures recovery. When hardware failures occur, the training system needs to replace the failed machines. The root agent in Gemini interacts with the cloud operator (e.g., Auto Scaling Group platform in AWS) to replace the faulty machines with healthy ones. When recovering training from hardware failures, there are two cases: 1) there are still healthy machines in each checkpoint placement group assigned by Algorithm 6.1, and 2) there is at least one checkpoint placement group in which all machines fail simultaneously. We will next discuss these two cases.

Case 1: As each checkpoint placement group still has healthy machines maintaining checkpoint replicas, Gemini can fetch the checkpoint replica from them for newly added machines and then recover the training progress. Figure 6.6c illustrates an example with four machines and two machines, Machine 2 and Machine 4, just failed simultaneously. The root agent replaces the two failed machines with two healthy ones. The two newly added machines replace their positions, reuse their machine rank IDs, and retrieve their checkpoints from alive machines. Because a checkpoint replica of Machine 2 was stored in Machine 1, Machine $2'$ (the one that replaced Machine 2) retrieves the checkpoint from Machine 1 for failure recovery. Machine $4'$ also retrieves the checkpoint from Machine 4. The machines that have no failures can directly restart training from their local checkpoints.

Case 2: In this case, machines must retrieve checkpoints from the remote persistent storage to ensure all machines recover training consistently. Although part of the model checkpoints are still accessible in the alive GPU machines, they are not consistent with the ones in the remote persistent storage because they are stored from different iteration numbers. In practice, the majority of failures during large model training are software failures or hardware failures with one machine replaced; it is rare to have two or more machine failures at the same time [20, 5]. Even with multiple machine failures simultaneously, Gemini can still recover failures from CPU memory in most cases thanks to the checkpoint placement strategy, as we will discuss in Section 6.7.2.

Standby machines. In case of hardware failures, the cloud operator is expected to provide healthy machines to replace faulty ones immediately. However, this replacement operation heavily depends on the availability of healthy machines in the GPU cloud and it can take a non-deterministic duration to successfully reserve new machines for the current training workload. In order to minimize the waiting time resulting from machine replacement, the training cluster can pre-allocate a few standby machines. When a machine suffers from hardware failures, a standby machine can immediately become active to replace the failed one for failure recovery. After that, the root agent returns the failed one and requests another standby machine. Gemini allows users to specify different numbers of standby machines according to their training workloads and the availability of healthy machines in GPU clouds.

Failure detection. The cloud operators typically provide tools to detect training failures and locate the failed machines. For example, Amazon SageMaker [24] has tools for failure type detection and failure machine localization. Gemini relies on these tools to detect failures in large model training. In addition, the worker agents and the root agent in Gemini also periodically send heartbeat signals to the distributed key-value store for failure detection.

6.7 Evaluation

In this section, we will demonstrate the effectiveness of Gemini for failure recovery in large model training. Specifically, we will address the following research questions:

- **Failure recovery performance:** Can Gemini reduce the wasted time without harming training throughput? (Section 6.7.2)
- **Scalability:** How does Gemini perform under different failure frequencies and training scales? (Section 6.7.3)
- **Effectiveness of traffic interleaving:** How does our traffic interleaving algorithm affect the training throughput? (Section 6.7.4)

Model size	Hidden size	Intermediate	#Layers	#AH
GPT-2 10B	2560	10240	46	40
GPT-2 20B	5120	20480	64	40
GPT-2 40B	5120	20480	128	40
RoBERTa 40B	5120	20480	128	40
BERT 40B	5120	20480	128	40
GPT-2 100B	8192	32768	124	64
RoBERTa 100B	8192	32768	124	64
BERT 100B	8192	32768	124	64

Table 6.2 : Configurations of different language models. AH is short for attention heads. GPT-2 10B means GPT with 10 billion parameters. The same naming convention applies to other models.

6.7.1 Implementation and Experimental Methodology

Implementation. We implement Gemini on top of DeepSpeed [170] and use ZeRO-3 setting [166]. we adopt etcd [15] as the distributed key-value store implementation to coordinate failure recovery. On the cloud provider side, we rely on Amazon EC2 Auto Scaling Groups (ASG) [8] to manage GPU machines. When failures are detected by ASG, the faulty machines are replaced with healthy ones. Such service is also available in Google Cloud [10] and Microsoft Azure [9]. Gemini reserves 128MB GPU memory for checkpoint communications. There are two CPU memory buffers to store the checkpoints: one for the completed checkpoint and the other for the ongoing one. When a failure occurs, the root agent notifies all alive agents to serialize the latest complete checkpoints with `torch.save()`, allowing PyTorch to load the saved checkpoints for failure recovery with `torch.load()`.

Setups. We conduct all experiments on AWS EC2 platform. Unless otherwise specified, we use 16 p4d.24xlarge instances for evaluations. Each instance has 1152GB CPU memory and it has 8 NVIDIA A100 (40GB) GPUs, which are interconnected via NVSwitch. p4d.24xlarge instances are connected through a 400Gbps elastic fabric adaptor (EFA) network. We adopt FSx [13] as the remote persistent storage and the aggregated bandwidth is 20Gbps. We also evaluate Gemini with p3dn.24xlarge instances, which have 8 NVIDIA V100 (32GB) GPUs and are con-

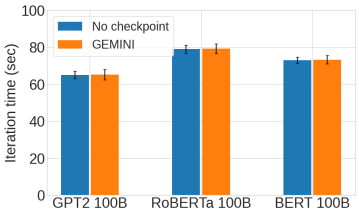


Figure 6.7 : The iteration time of three large models without checkpoints and with Gemini.

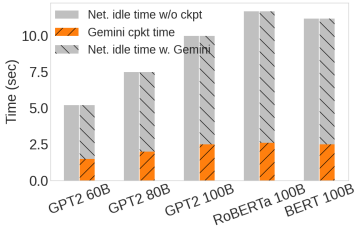


Figure 6.8 : The network idle time of three large models without checkpoints and with Gemini.

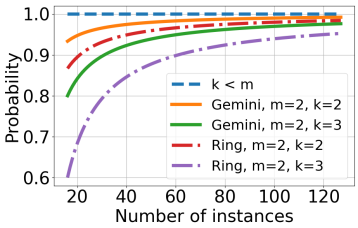


Figure 6.9 : The probability that Gemini can recover failures from checkpoints in CPU memory.

nected to a 100Gbps EFA network. The used software versions are CUDA-11.6, DeepSpeed-v0.7.3, PyTorch-1.13, nccl-v2.14.3, and etcd-v3.5.

Workloads. We evaluate Gemini with popular and representative large deep learning models, including GPT-2 [164], BERT [67], and RoBERTa [117]. We vary the number of layers, hidden sizes, and intermediate sizes in these models [235, 166]. Table 6.2 summarizes the detailed model configurations. We use the sequence length 512 and the vocabulary size 50265 for the evaluation. We set the micro-batch size to 8 with mixed-precision and we enable the activation recomputation [106, 144] in the evaluation. The optimizer used is Adam [103]. The training dataset is Wikipedia-en corpus [130].

Baselines. We adopt two baselines, *Strawman* and *HighFreq*, for the evaluations. Strawman uses the checkpoint frequency following the setup in training BLOOM [5] and it checkpoints model states every three hours. HighFreq aims to fully saturate the bandwidth capacity of the remote persistent storage and it represents the best we can do with remote storage-based solutions. HighFreq first profiles both the checkpoint time t_{ckpt} and the iteration time T_{iter}; it then checkpoints the model states every $\lceil t_{ckpt}/T_{iter} \rceil$ iterations. Both baselines store the checkpoints in the remote persistent storage, while the difference is the checkpoint frequency. Note that Gemini also checkpoints to the remote persistent storage every three hours in addition to checkpointing to CPU memory.

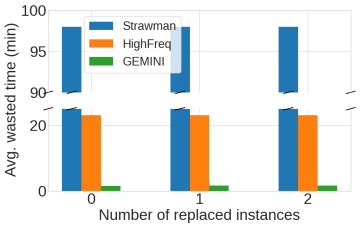


Figure 6.10 : The average wasted time of GPT-2 100B with different numbers of replaced instances.

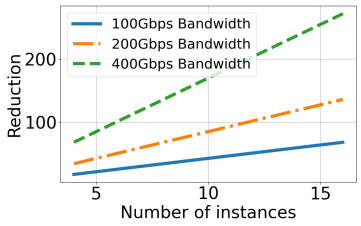


Figure 6.11 : The checkpoint time reduction of Gemini over baselines under different bandwidth.

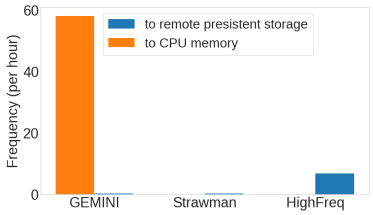


Figure 6.12 : Gemini achieves a much higher checkpoint frequency than the two baselines.

6.7.2 Training Efficiency

In this subsection, we evaluate Gemini on both p4d.24xlarge and p3dn.24xlarge instances. We first use 16 p4d.24xlarge instances to demonstrate the performance advantages of Gemini over the baselines on large-scale model training. The largest model size we can train is 100B given the machine scale and the GPU memory size. Further increasing the model size causes GPU out-of-memory errors.

Training time. We examined Gemini's impacts on the training throughput by benchmarking GPT-2 100B, RoBERTa 100B, and BERT 100B. We carried out 50 training iterations with Gemini, which performed checkpointing for every iteration, and an equal number of iterations without checkpointing using vanilla DeepSpeed. Figure 6.7 shows the iteration times for both settings across the three models. We can find that Gemini does not affect the training iteration times. This is because the network idle time during training is adequate to accommodate the checkpoint traffic. Figure 6.8 confirms that there is still available network idle time even after Gemini inserts all the checkpoint traffic. It indicates that Gemini can achieve per iteration checkpointing without incurring extra overhead to the training throughput thanks to the traffic interleaving algorithm.

Since Gemini has negligible overhead for all the large models we evaluated, we use the GPT-2 100B model as the representative in the following part for brevity. RoBERTa and BERT have similar results and will not affect our conclusions.

Wasted time. We next evaluate the wasted time when a failure occurs. We first

analyze the probability that Gemini can recover failures from CPU memory. Given the checkpoint replica number m, the probability is determined by the number of instances k that need to be replaced simultaneously (failures occurred on those instances). When $k < m$, Gemini can always recover training from CPU memory. When $k \geq m$, we can calculate the probability according to Corollary 6.1. Figure 6.9 plots the probability that Gemini can recover failures from CPU memory under different settings. The probability increases with the number of instances N. Suppose there are two checkpoint replicas, i.e., $m = 2$. When $N = 16$ and $k = 2$, Gemini has a probability of 93.3%; when $k = 3$, it still has a probability of 80.0%. We also consider the ring strategy, in which instance i stores its model states in itself and instance $(i + 1)$ mod N. When $N = 16$ and $k = 3$, Ring's probability is 25.0% lower than that of Gemini. According to OPT-175B [233] observation, there are 1.5% instances that fail every day. Even for a thousand-scale training cluster, the possibility of two instances having failures at the same time is very limited. Therefore, Gemini with $m = 2$ can recover failures from CPU memory for most cases.

We next calculate the average wasted time based on the measured iteration time, checkpoint time, and retrieval time according to Expression (6.1). Figure 6.10 shows the average wasted time for training of GPT-2 100B on 16 p4d.24xlarge instances with different numbers of replaced instances. The average wasted time of both Strawman and HighFreq is deterministic because the checkpoints are always retrieved from the remote persistent storage when failures occur. In contrast, the average wasted time of Gemini varies. When there is no instance replaced, e.g., due to software failures, the checkpoints are already at the local CPU memory. The average wasted time in this case is 1.5× the iteration time ($1.5T_{iter}$). When there is only one instance replaced or two instances are replaced but training can be recovered from the CPU memory, the extra overhead for failure recovery is to retrieve checkpoints from other instances and the retrieval time is less than three seconds. In these cases, Gemini can reduce the average wasted time by more than 13× compared to HighFreq. However, when two instances are replaced and training cannot be recovered from the CPU memory, of which the possibility is 6.7% with 16 instances according to Figure 6.9, Gemini

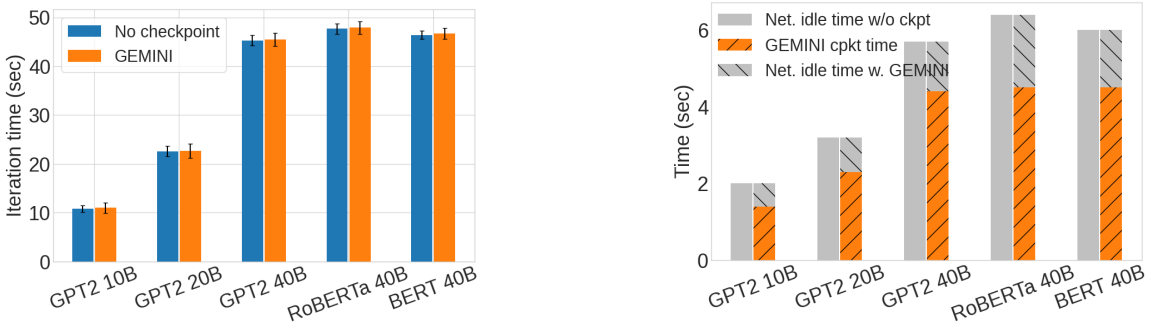
degrades to Strawman.

Checkpoint time. To showcase the advantage of Gemini in terms of checkpoint time, Figure 6.11 displays the checkpoint time reduction of Gemini over the baselines under different network bandwidths and different numbers of instances. Both baselines, Strawman and HighFreq, have the same checkpoint time and it stays almost the same as the number of machines increases from 4 to 16 because the aggregated bandwidth of the remote persistent storage is fixed. In contrast, Gemini's checkpoint time reduces with an increase in the number of instances in our testbed because it utilizes the aggregated network bandwidth among GPU machines to write checkpoints to the CPU memory. The checkpoint time reduction also increases with the network bandwidth connecting GPU instances. For example, with 16 p4d.24xlarge instances, the reduction is 65× with a 100Gbps network, and it increases to more than 250× with a 400Gbps network. It is very challenging for remote persistent storage to achieve comparable performance as Gemini. To match the checkpoint time of Gemini in our scenario, which involves 16 instances, persistent storage would need to achieve an aggregated bandwidth of 6.4Tbps theoretically.

Checkpoint frequency. Gemini checkpoints model states to CPU memory for every iteration. The iteration time of GPT-2 100B with 16 p4d.24xlarge is 62 seconds, but the checkpoint time with Gemini is less than 3 seconds. As shown in Figure 6.12, Gemini improves the checkpoint frequency over HighFreq by 8× and over Strawman by more than 170×. Note that the checkpoint frequency of Gemini is bounded by the iteration time and it can achieve an even higher frequency with the computation advancement of accelerators.

We then demonstrate that Gemini can also efficiently support other training models on p3dn.24xlarge instances. The largest model size we can train with this hardware setting is 40B. Further increasing the model size causes GPU out-of-memory errors in our testbed.

Model training on p3dn.24xlarge. Figure 6.13a illustrates that Gemini minimally affects the training throughput using 16 p3dn.24xlarge instances across various model sizes (10B, 20B, and 40B) and model architectures (GPT-2, RoBERTa, and BERT).



(a) The iteration time.

(b) The network idle time.

Figure 6.13 : Gemini is generalized to p3dn.24xlarge instances and other models.

The training efficiency aligns with the findings from 16 p4d.24xlarge instances. Figure 6.13b contrasts network idle times during model training without checkpoints and with Gemini, revealing that the network idle time is still sufficient to accommodate the checkpoint traffic.

6.7.3 System Scalability

In this subsection, we first report the overheads incurred by failures in Gemini and the baselines. We then use simulation to demonstrate that Gemini is scalable to scenarios with frequent failures and to support LLM training with thousands of instances.

Overheads incurred by failures. Besides the lost training progress, the checkpoint time, and the retrieval time, there are other overheads in Gemini to recover training from a failure. We train GPT-2 100B on 16 p4d.24xlarge instances and the training process is illustrated in Figure 6.14. Gemini checkpoints the model states to the CPU memory for every iteration. An instance failure is triggered during Iteration 4 and it takes 15 seconds for the root agent to detect this failure. The root agent then notifies all alive instances to serialize the checkpoints stored in CPU memory with `torch.save()`. We observe that this operation is time-consuming and it takes 162 seconds to finish the serialization of two checkpoint replicas, one is from local and the other is from another instance. We also measure the waiting time to successfully reserve a new p4d.24xlarge instance with ASG to estimate the extra instance-replacing overhead in case of hardware failures, which is around 4-7 minutes. Another noticeable

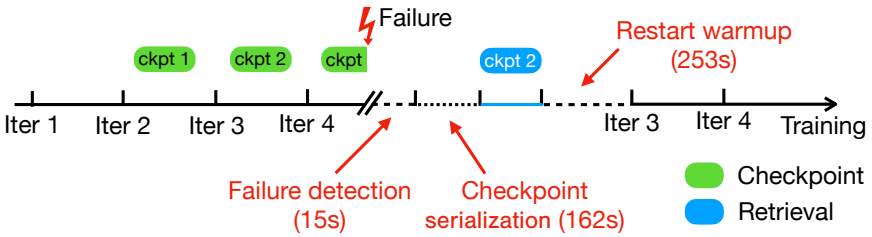
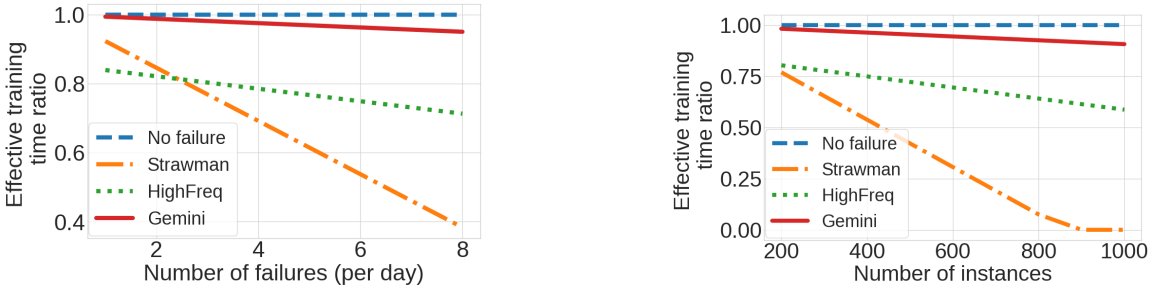


Figure 6.14 : The overhead of failure recovery for GPT-2 100B training with Gemini. A failure occurs during Iteration 4 and one instance is replaced.



(a) Different failure rates.

(b) Different instance numbers.

Figure 6.15 : The scalability of Gemini under simulation.

overhead is the restart warmup time and it takes more than four minutes before the training can proceed from Iteration 3. To sum up, in our testbed, the total overhead resulting from a failure that can be recovered from CPU memory is around 7 minutes for software failures and 12 minutes for hardware failures. Note that the instance-replacing overhead for hardware failures can be greatly reduced by standby machines.

The two baselines have no checkpoint serialization overhead when a failure occurs, but they have such overhead for every checkpoint to the remote persistent storage. Their checkpoint communications to the remote persistent storage are asynchronous to computation, but they need to serialize the checkpoints with `torch.save()`, which blocks training. HighFreq checkpoints the model states every nine iterations and the incurred overhead for each checkpoint serialization is around 81 seconds. Strawman also has this overhead, but it is negligible due to the low frequency.

Based on the incurred overhead by one failure, we can simulate the training performance of GPT-2 100B with different failure rates and numbers of instances. We consider software failures in the simulation because recovering training from hardware

failures has a similar overhead as from software failures if standby machines are used.

Scaling to frequent failures. To evaluate the impact of failure rates, we conducted simulations of training performance using 16 p4d.24xlarge instances and different checkpointing solutions. We measured the training performance using a metric called the effective training time ratio, which indicates the percentage of productive training progress achieved in a given period of time. Failures decrease this ratio due to the overheads for failure recovery. The effective training time ratios with different solutions are shown in Figure 6.15a. We found that even with 8 failures per day, Gemini remains highly efficient with a performance ratio close to the baseline with no failures. However, the costly overhead of checkpoint serialization, i.e. invoking `torch.save()`, in HighFreq significantly hurts its performance. Even without any failures, 14.5% time is spent on checkpoint serialization. On the other hand, Gemini only serializes checkpoints when failures occur. Strawman is worse than HighFreq due to its prohibitive wasted time.

Scaling to more instances. We also simulate the training performance with different numbers of instances involved in training. Following the training report of OPT-175B [233], we assume that 1.5% instances fail every day. The failure frequency increases with the number of instances. Figure 6.15b shows that with 1000 instances, the effective training time ratio of Gemini is still around 91%, which is 54% higher than HighFreq. Training with Strawman for failure recovery can hardly proceed because of the frequent failures and the prohibitive wasted time.

6.7.4 Effectiveness of Traffic Interleaving

In this subsection, we evaluate the effectiveness of Gemini's traffic interleaving algorithm. To understand the performance contributions of its two approaches, we report the iteration time of GPT-2 40B on 16 p3dn.24xlarge instances with the following schemes for checkpointing to CPU memory.

- **Baseline.** It is the model training without checkpointing.
- **Blocking.** It checkpoints the model states to CPU memory, but the checkpoint traffic blocks training traffic at the beginning of each iteration.

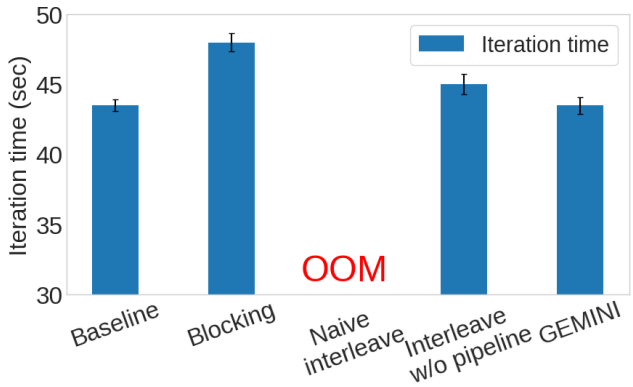


Figure 6.16 : The iteration time of GPT-2 40B with different schemes for checkpointing to CPU memory. OOM is short for out of memory.

- **Naïve interleave.** It partitions checkpoint traffic for interleaving, but each network idle timespan only has one checkpoint partition.
- **Interleave without pipeline.** Each idle timespan can have multiple partitions, but it only uses one GPU buffer for checkpoint communications. The buffer size is 128MB.
- **Gemini.** It uses four small sub-buffers for pipelining checkpoint communications and the size of each buffer is 32MB.

As shown in Figure 6.16, the iteration time with Blocking is 10.1% higher than the Baseline due to the extra checkpoint time. Naïve interleave can cause GPU out-of-memory (OOM) errors because it requires a large GPU memory buffer for checkpoint communications. For example, the largest idle time span profiled during training is 1.6s and the required memory buffer size is more than 2GB on each GPU. Interleave without pipeline can greatly reduce the required GPU memory buffer size and avoid OOM error, but communications have to wait for GPU-to-CPU copy. The total network idle time becomes insufficient to accommodate the checkpoint traffic in this case and it worsens the iteration time by 3.5%. In contrast, the iteration time with Gemini is almost the same as the Baseline because it can fully utilize the network idle time by pipelining checkpoint communications.

Chapter 7

Related Work

7.1 Related Work on Sparse Tensor Synchronization

Related work on communication schemes to support sparse tensors synchronization has already been discussed in Section 3.2.5.

Acceleration of dense tensor synchronization. ATP [109] and SwitchML [183] exploit programmable switches for the synchronization of dense tensors. BytePS [94] uses spare CPU and bandwidth resources in GPU clouds to optimize communications. Blink [209] generates optimal communication primitives for intra-machine communication with NVLink. PLink [119] designs a hierarchical aggregation scheme for DDL in public clouds, where the machine-to-machine bandwidth is non-uniform due to the hierarchical structure of data centers. ByteScheduler [160], P3 [91], and TicTac [85] schedule communications of tensors closer to the output layer with higher priority. These approaches disregard the values of gradients and communicate all gradients. In contrast, Zen leverages sparsity in DNN models and only transmits non-zero gradients to reduce the synchronization time.

Acceleration of sparse tensor synchronization. Parallax [102] utilizes Sparse PS and it cannot achieve balanced communications across GPU machines in gradient synchronization. Zen can achieve balanced communications by using a novel hierarchical hashing algorithm. Flare [62] and Libra [154] use programmable switches to accelerate sparse tensor communications, but they rely on specific hardware. Moreover, Zen analyzes the characteristics of sparse tensors and explores the design space for communication schemes to determine the optimal one, but prior approaches did not consider these factors.

Hash algorithms for load balancing. Previous works have attempted to achieve

load balancing using hashing [37, 45, 60, 41, 221]. They typically assign two partitions to a given index using two hash functions and then selecting the partition with more available memory as the final destination [136, 174, 61, 137]. This line of works utilizes the power of two choices [136, 174, 61, 137] in hashing. However, these methods require serial writing of indices to memory and cannot leverage the parallel computing power of GPUs. In contrast, Zen enables parallelizable computing on GPUs. While DRAGONN [217] introduces a hash-based algorithm for parallel writing of non-zero gradients to memory, it does not address the imbalanced communications in DDT and cannot handle hash collisions, resulting in information loss. In contrast, Zen achieves balanced communications and avoids information loss.

7.2 Related Work on Gradient Compression

Related work on gradient compression algorithms has already been discussed in Section 4.2.2.

Compression-enabled systems. GRACE [227] quantitatively evaluates the impacts of GC algorithms and observes that GC can incur non-negligible compression overhead, but it does not study or address the challenges of applying GC to DDL. Several frameworks have been recently proposed to support compression-enabled DDL. HiTopKComm [188] designs a new communication scheme for GC, but it compresses all tensors with GPUs and leads to prohibitive compression overhead. HiPress [38] proposes compression-aware synchronization to overlap compression with communication and a selective compression mechanism to decide whether to compress a tensor, but it only uses GPUs for compression and ignores the interactions among tensors. BytePS [240] also supports GC, but it only uses CPUs for compression and ignores the interactions among tensors as well. These frameworks only compress tensors for inter-machine communication. In contrast, Espresso uses both GPUs and CPUs for compression, analyses interactions among tensors to make compression decisions, and addresses both intra- and inter-machine communication bottlenecks. OmniReduce [74] introduces block gradient sparsification, which is a new type of GC algorithm, but Espresso focuses on how to efficiently apply GC to DDL.

Layer-wise compression. Recent work [226, 26] quantitatively evaluated the impacts of GC algorithms in a layer-wise fashion. They observe that GC can incur non-negligible compression overheads, but they have no solution to address the challenges of applying GC to DDT. HiPress [38] proposes a selective compression mechanism to determine whether to compress a tensor, but it still applies GC algorithms to a DDT job in a layer-wise fashion and incurs costly compression overheads. Cupcake uses a fusion fashion to minimize the incurred compression overheads.

Tensor fusion scheduling. Distributed deep learning frameworks batch multiple tensors for one communication operation to improve communication efficiency [187, 112, 50, 176]. However, this mechanism takes place after compression and is orthogonal to GC algorithms. PipeSwitch [39] fuses tensors to pipeline model transmission over the PCIe for fast context switching of deep learning applications. In contrast, Cupcake fuses tensors to improve compression efficiency.

7.3 Related Work on Fault Tolerance

Checkpointing in deep learning. Deep learning frameworks, such as PyTorch [156], TensorFlow [25], and MXNet [50], provide users with the interfaces to checkpoint model states during training for failure recovery. Unlike Gemini, it is the users' responsibility to decide how to checkpoint, such as the checkpoint frequency and storage location. To reduce checkpointing overheads, DeepFreeze [145] performs asynchronous checkpointing but stores checkpoints in remote persistent storage. CheckFreq [139] dynamically adjusts the checkpointing frequency, but the remote storage bandwidth limits the highest frequency. In contrast, Gemini stores checkpoints in CPU memory, enabling much higher frequencies than DeepFreeze and CheckFreq. Check-N-Run [71] compresses checkpoints with lossy schemes to reduce required storage, but this may harm model accuracy and incur compression overheads. Gemini stores the original checkpoints without impacting accuracy or incurring compression overheads. Gandiva [225] assumes healthy machines for checkpointing with an on-demand checkpoint mechanism for job migration. Because any machine involved in training can experience hardware failures, Gandiva's checkpoint mechanism cannot

handle this case in which checkpoints stored in failed machines will get lost. Furthermore, its on-demand checkpointing cannot tackle unexpected failures during large model training. In contrast, Gemini aims to recover training from both unexpected software and hardware failures.

Checkpointing in distributed systems. Diskless checkpointing [161] stores checkpoints in CPU memory. It requires processors to encode a checkpoint with parity and their checkpoints can be recalculated when a processor fails. However, encoding and decoding a checkpoint of large model training is extremely expensive. Instead, Gemini employs redundant checkpoints for failure recovery. FTC-Charm++ [237] stores two checkpoint copies on two processors for fault tolerance. However, it lacks an analysis of optimal checkpoint placements. Unlike traditional distributed systems, a key challenge in Gemini is to schedule checkpoint traffic to minimize its interference with model training. This differentiates Gemini from existing work on checkpointing for failure recovery in distributed systems.

Communication scheduling in distributed training. ByteScheduler [160], TicTac [85], and P3 [91] aim to improve the performance of training by scheduling the communication orders of tensors. These works primarily focus on accelerating training communication. They are orthogonal and complementary to Gemini because Gemini focuses on minimizing interference with training communication by scheduling checkpointing communications.

Failure recovery with spot instances. Bamboo [201] uses redundant computation to provide resilience and fast recovery for training large DNN models on preemptible instances. Gemini checkpoints to CPU memory and doesn't require redundant computation. Varuna [35] also enables large model training on preemptible instances, but it requires users to manage the checkpoints, such as the frequency and the storage, for failure recovery. In contrast, Gemini offers transparent checkpointing for failure recovery, eliminating the need for users to manage checkpoints.

Chapter 8

Conclusions and Future Directions

We conclude this thesis in this chapter by summarizing our contributions (Section 8.1) and suggesting potential future research directions (Section 8.2).

8.1 Conclusions

This thesis is dedicated to advancing the scalability of distributed deep learning through a strategic focus on optimizing communication across both the data plane and the management plane. The foundational principle underlying this research asserts the feasibility of alleviating communication bottlenecks in distributed deep learning by harnessing existing hardware resources within training systems, complemented by intelligent traffic and resource scheduling algorithms. We approach these communication challenges with a fresh perspective, grounded in *fundamental principles*.

Zen revolutionizes sparse tensor synchronization by finding the optimal synchronization scheme from first principles (Chapter 3). Departing from intuition-driven approaches that often yield suboptimal performance, this initiative systematically analyzes the design space of schemes for sparse tensor synchronization. By discerning the optimal schemes under varied scenarios, Zen ensures the determination of an optimal scheme tailored to the unique characteristics of sparse tensors in distributed deep learning. This first-principle analysis guarantees Zen's ability to deliver near-optimal performance in the communication time of gradient synchronization.

Espresso and **Cupcake** embark on a comprehensive reevaluation of compression strategies and granularity for gradient compression from first principles (Chapter 4 and Chapter 5). Addressing the limitations of existing approaches developed primarily from an algorithmic perspective, they critically rethink the design space for compression strategies and granularity in gradient compression operations from

a systematic perspective. By identifying and implementing near-optimal compression strategies and optimal fusion strategies, they significantly enhance the training throughput of compression-enabled distributed deep learning.

Gemini pioneers a redefined hierarchical storage system for checkpoint management in distributed deep learning (Chapter 6). It minimizes failure recovery overhead, particularly in large-scale model training, by distinguishing checkpoints for different purposes and strategically segregating them into different storage layers based on their storage bandwidth and capacity characteristics. This first-principle analysis enables Gemini to store checkpoints for failure recovery at extremely high frequency while preserving user flexibility in checkpoint management.

Collectively, these groundbreaking initiatives exemplify the paradigm shifts brought forth by this thesis, leveraging a fresh perspective and fundamental analysis to substantially enhance the system efficiency of distributed deep learning by optimizing both data-plane and management-plane communications. Notably, two techniques developed in this thesis have been adopted by modern distributed deep learning systems. Espresso has been incorporated into BytePS [94] as its gradient compression module and Gemini is serving as an in-memory checkpointing system in Amazon Web Services (AWS) to bolster fault tolerance in large model training.

8.2 Future Directions

This thesis represents a significant advancement in the development of distributed deep learning systems with optimized communications for popular training paradigms, such as data parallelism and static training, i.e., training jobs with fixed computation resources. In the following section, we will explore two potential future directions for further improving communications optimization for both model serving and elastic training with spot instances [31, 59, 133].

8.2.1 Communication Optimizations for Model Serving

The research presented in this thesis focuses on optimizing communications in model training. However, communications can also become a bottleneck in model serving,

especially in the context of large language models (LLMs). Unlike traditional DNN models like ResNet [87] and VGG [190] that can be fitted in a single GPU, modern large models such as OPT-175B and Mixture-of-Expert transformers model [68, 53] need to be partitioned across multiple GPUs. While existing work focuses on optimizing computation resource scheduling [114, 239] for serving, there is a lack of comprehensive analysis on serving traffic across GPUs that are typically connected by PCIe [11, 12] and how it affects the serving performance. Future work should conduct extensive measurements and evaluations on serving traffic under different types of accelerators, parallelism strategies, and network bandwidths to understand the traffic characterizations and optimize communications from the first principle to enhance serving throughput and latency.

Additionally, hallucination [171, 126] is a big issue in the context of LLMs where the generated information is plausible-sounding but inaccurate or fabricated. A vector database [23] is a promising solution to address hallucination by facilitating contextual understanding and supporting the incorporation of external knowledge to help improve the model's ability to generate accurate and contextually relevant information. Different from traditional databases, such as MySQL [69] and PostgreSQL [140] that excel in managing structured data with predefined relationships and retrieving data by indices, vector databases are tailored for handling high-dimensional vectors that are mathematical representations of objects and their queries often involve *similarity search*. However, it is challenging to design a communication-efficient and scalable vector database system for LLMs. A distributed vector database is a must because there are typically billions of entries that cannot be fitted into a single device. Unfortunately, under current designs [95], its communications cannot scale with the number of devices in the distributed vector database because each device has to process every query for similarity search with all entries in the database. Future work shall design a scalable vector database for LLM in terms of query traffic and achieve load-balanced query distribution among all the nodes.

8.2.2 Fault Tolerance for Training with Spot Instances

This thesis aims to optimize management-plane communications to minimize failure recovery overhead in static training using *on-demand instances* [21, 22], which are a type of cloud computing resource provided by cloud service providers on a pay-as-you-go basis. Recently, there has been a lot of interest in training DNN models using *spot instances* to reduce the monetary cost due to their much lower prices [201, 90]. However, compared to training with on-demand instances, this comes with the disadvantage of frequent preemptions, making efficient fault tolerance crucial to reduce the overhead of failure recovery.

Unfortunately, the design of Gemini cannot be directly applied to training with spot instances. This is because the number of available instances for training can change when some instances are preempted. As a result, the training system has to reconfigure the parallelism strategies. For instance, it has to update the number of pipeline stages in pipeline parallelism [142] or the partition number in ZeRO-3 [166]. Also, the checkpoints saved by each GPU cannot be used for failure recovery directly because the parallelism reconfiguration requires the checkpoints to be repartitioned to match the updated parallelism strategies. It would be interesting to research how to determine the optimal parallelism configuration after instance preemptions and efficiently repartition checkpoints for training with spot instances.

Another interesting future direction is to explore the possibility of using both on-demand and spot instances for training. Current solutions either solely rely on on-demand instances [213] or spot instances [201, 35, 90], but we believe that combining both options could be beneficial. The overhead for updating the parallelism configuration is non-negligible in the event of instance preemptions. One possible solution to avoid this overhead is to temporarily replace the preempted instances with on-demand instances, and then replace these expensive on-demand instances with cheaper spot instances once they become available again. However, this solution involves a trade-off between the reduced overhead of parallelism reconfiguration and the extra cost of using on-demand instances. Therefore, further research could investigate how to choose between parallelism reconfiguration and spot instance re-

placement with on-demand instances at runtime in order to minimize the monetary cost of training.

Bibliography

[1] NVIDIA P100. https://www.nvidia.com/en-us/data-center/tesla-p100/, 2016.

[2] Deep Neural Networks. https://srnghn.medium.com/deep-learning-overview-of-neurons-and-activation-functions-1d98286cf1e4, 2018.

[3] NVIDIA NCCL. https://developer.nvidia.com/NCCL, 2021.

[4] AWS Trainium. https://aws.amazon.com/machine-learning/trainium/, 2022.

[5] BLOOM Chronicles. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md, 2022.

[6] MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, April 2022. USENIX Association.

[7] NVIDIA H100. https://www.nvidia.com/en-us/data-center/h100/, 2022.

[8] Auto Scaling in AWS. https://docs.aws.amazon.com/autoscaling/, 2023.

[9] Auto Scaling in Azure. https://learn.microsoft.com/en-us/azure/app-service/manage-scale-up, 2023.

[10] Auto Scaling in Google Cloud. https://cloud.google.com/compute/docs/autoscaler, 2023.

[11] AWS A10 GPU. https://aws.amazon.com/ec2/instance-types/g5/, 2023.

[12] AWS AWS Inferentia2. https://aws.amazon.com/ec2/instance-types/inf2/, 2023.

[13] AWS FSx. https://aws.amazon.com/fsx/, 2023.

[14] ChatGPT. https://openai.com/blog/chatgpt, 2023.

[15] etcd. https://etcd.io/, 2023.

[16] GPU instances in Google Cloud Platform. https://cloud.google.com/compute/docs/gpus, 2023.

[17] ND40rs_v2 in Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/ndv2-series, 2023.

[18] ND96asr_v4 in Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, 2023.

[19] NVIDIA DGX A100. https://www.nvidia.com/en-gb/data-center/dgx-a100/, 2023.

[20] OPT-175B logbook. https://github.com/facebookresearch/metaseq/tree/main/projects/OPT/chronicles, 2023.

[21] P3dn.24xlarge in AWS. https://aws.amazon.com/ec2/instance-types/p3/, 2023.

[22] P4d.24xlarge in AWS. https://aws.amazon.com/ec2/instance-types/p4/, 2023.

[23] Pinecone. https://www.pinecone.io/learn/vector-database/, 2023.

[24] SageMaker. https://docs.aws.amazon.com/sagemaker/index.html, 2023.

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

[26] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. On the utility of gradient compression in distributed training systems. *arXiv preprint arXiv:2103.00543*, 2021.

[27] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. 2017.

[28] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[29] Amazon. Gradient Compression in MXNet. `https://mxnet.apache.org/versions/1.9.0/api/faq/gradient_compression.html`, 2021.

[30] Amazon. Amazon EC2 pricing on demand. `https://aws.amazon.com/ec2/pricing/on-demand/`, 2022.

[31] Amazon. AWS spot instances. `https://aws.amazon.com/ec2/spot/`, 2023.

[32] Austin Appleby. Murmurhash 2.0, 2008.

[33] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for NVIDIA GPGPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019.

[34] Yehia Arafa, Abdel-Hameed A Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for NVIDIA GPGPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019.

[35] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[36] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL? In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, 2018.

[37] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*, pages 593–602, 1994.

[38] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. Gradient compression supercharged high-performance data parallel DNN training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 359–375, 2021.

[39] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[40] James Bennett, Stan Lanning, et al. The Netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, 2007.

[41] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: The heavily loaded case. *SIAM Journal on Computing*,

35(6):1350–1385, 2006.

[42] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signSGD: Compressed optimisation for non-convex problems. *arXiv preprint arXiv:1802.04434*, 2018.

[43] Dhruba Borthakur et al. HDFS architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

[44] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[45] Zhiruo Cao, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 1, pages 332–341. IEEE, 2000.

[46] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 106–112, 1977.

[47] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. MONGOOSE: A learnable LSH framework for efficient neural network training. In *International Conference on Learning Representations (ICLR)*, 2021.

[48] Beidi Chen, Tharun Medini, James Farwell, Charlie Tai, Anshumali Shrivastava, et al. SLIDE: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. *Proceedings of Machine Learning and Systems*, 2:291–306, 2020.

[49] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. *arXiv preprint arXiv:1604.00981*, 2016.

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

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

[52] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On efficient constructions of checkpoints. In *International Conference on Machine Learning*, pages 1627–1636. PMLR, 2020.

[53] Zhifeng Chen, Dmitry Dima Lepikhin, HyoukJoong Lee, Maxim Krikun, Noam Shazeer, Orhan Firat, Yanping Huang, Yuanzhong Xu, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. 2020.

[54] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[55] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

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

[57] Minsik Cho, Ulrich Finkler, Mauricio Serrano, David Kung, and Hillery Hunter. BlueConnect: Decomposing All-Reduce for deep learning on heterogeneous network hierarchy. *IBM Journal of Research and Development*, 63(6):1–1, 2019.

[58] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[59] Google Cloud. Google Cloud Platform spot virtual machines. `https://cloud.google.com/spot-vms`, 2023.

[60] Artur Czumaj, Chris Riley, and Christian Scheideler. Perfectly balanced allocation. In *Approximation, Randomization, and Combinatorial Optimization.. Algorithms and Techniques*, pages 240–251. Springer, 2003.

[61] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, Eva Rotenberg, and Mikkel Thorup. The power of two choices with simple tabulation. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1631–1642. SIAM, 2016.

[62] Daniele De Sensi, Salvatore Di Girolamo, Saleh Ashkboos, Shigang Li, and Torsten Hoefler. Flare: Flexible in-network allreduce. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2021.

[63] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le,

and Andrew Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2012.

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

[65] Li Deng and John Platt. Ensemble deep learning for speech recognition. In *Proc. interspeech*, 2014.

[66] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

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

[68] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.

[69] Paul DuBois. *MySQL*. Addison-Wesley, 2013.

[70] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

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

[72] Carla Schlatter Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116, 1983.

[73] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 25–36, 2012.

[74] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 676–691, 2021.

[75] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2018.

[76] Santosh K Gaikwad, Bharti W Gawali, and Pravin Yannawar. A review on speech recognition technique. *International Journal of Computer Applications*, 10(3):16–24, 2010.

[77] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[78] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, pages 323–336, 2010.

[79] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[80] Google. Google Cloud. `https://cloud.google.com/compute/docs/gpus`, 2021.

[81] William Gropp. MPICH2: A new start for MPI implementations. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 7–7. Springer, 2002.

[82] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.

[83] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: a factorization-machine based neural network for CTR prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1725–1731, 2017.

[84] Vipul Gupta, Dhruv Choudhary, Ping Tak Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W. Mahoney. Fast distributed training of deep neural networks: Dynamic communication thresholding for model and data parallelism, 2020.

[85] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy H Campbell. TicTac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

[86] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

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

[88] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Jo-

hannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[89] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32:103–112, 2019.

[90] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[91] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *arXiv preprint arXiv:1905.03960*, 2019.

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

[93] Peng Jiang and Gagan Agrawal. A linear speedup analysis of distributed deep learning with sparse and quantized communication. In *Advances in Neural Information Processing Systems*, pages 2525–2536, 2018.

[94] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[95] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[96] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[97] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[98] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[99] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[100] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian U Stich, and Martin Jaggi. Error feedback fixes SignSGD and other gradient compression schemes. *arXiv preprint arXiv:1901.09847*, 2019.

[101] Junho Kim, Minjae Kim, Hyeonwoo Kang, and Kwanghee Lee. U-GAT-IT: Unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation. *arXiv preprint arXiv:1907.10830*, 2019.

[102] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[103] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[104] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

[105] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. CROSSBOW: scaling deep learning with small batch sizes on multi-GPU servers. *arXiv preprint arXiv:1901.02244*, 2019.

[106] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

[107] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[108] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[109] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. ATP: In-network aggregation for multi-tenant learning. In *NSDI*, pages 741–761, 2021.

[110] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch, and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[111] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[112] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[113] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, Zheng Qu, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, and Nam Sung Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–188. IEEE, 2018.

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

[115] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *The International Conference on Learning Representations (ICLR)*, 2017.

[116] Witold Litwin, Marie-Anna Neimat, and Donovan A Schneider. Lh*—a scalable, distributed data structure. *ACM Transactions on Database Systems (TODS)*, 21(4):480–525, 1996.

[117] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[118] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 41–54, 2018.

[119] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. PLink: Discovering and exploiting datacenter network locality for efficient cloud-based distributed training. *Proc. of MLSys*, 2020.

[120] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015.

[121] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems*, 3:637–651, 2021.

[122] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[123] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 937–954, 2020.

[124] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60, 2014.

[125] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. MLPerf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

[126] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. Sources of hallucination by large language models on inference tasks. *arXiv preprint arXiv:2305.14552*, 2023.

[127] Tharun Medini, Beidi Chen, and Anshumali Shrivastava. Solar: Sparse orthogonal learned and random embeddings. In *International Conference on Learning Representations*, 2020.

[128] Mellanox. Mellanox Corporate Update. https://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf, 2022.

[129] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182*, 2017.

[130] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[131] Meta. Gradient Compression in Pytorch. https://pytorch.org/docs/stable/ddp_comm_hooks.html, 2022.

[132] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach, Vol. I.* Tioga, Palo Alto, CA, 1983.

[133] Microsoft. Microsoft Azure spot virtual machines. https://azure.microsoft.com/en-us/products/virtual-machines/spot, 2021.

[134] Microsoft. DeepSpeed. https://github.com/microsoft/DeepSpeed, 2023.

[135] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Daniel Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, et al. Evolving deep neural networks. In *Artificial intelligence in the age of neural networks and brain computing*, pages 293–312. Elsevier, 2019.

[136] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[137] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[138] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[139] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, fine-grained DNN checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[140] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[141] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[142] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[143] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498, 2020.

[144] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[145] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181. IEEE, 2020.

[146] NVIDIA. A Timeline of Innovation for NVIDIA. ttps://www.nvidia.com/en-us/about-nvidia/corporate-timeline/, 2021.

[147] NVIDIA. BERT training time. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/, 2021.

[148] NVIDIA. Broadcast in NCCL. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md, 2021.

[149] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (ATC 14)*, pages 305–319, 2014.

[150] OpenAI. AI and Compute. https://openai.com/blog/ai-andcompute/, 2021.

[151] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2:400–411, 2020.

[152] George Ostrouchov, Don Maxwell, Rizwan A Ashraf, Christian Engelmann, Mallikarjun Shankar, and James H Rogers. GPU lifetimes on titan supercomputer: Survival analysis and reliability. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[153] Mayur R Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon S3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, 2008.

[154] Heng Pan, Penglai Cui, Zhenyu Li, Ru Jia, Penghao Zhang, Leilei Zhang, Ye Yang, Jiahao Wu, Jianbo Dong, Zheng Cao, Qiang Li, Hongqiang Harry Liu, Mathy Laurent, and Gaogang Xie. Enabling fast and flexible distributed deep learning with programmable switches. *arXiv preprint arXiv:2205.05243*, 2022.

[155] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[156] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[157] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[158] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with GPT-4. *arXiv preprint arXiv:2304.03277*, 2023.

[159] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[160] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[161] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on parallel and Distributed Systems*, 9(10):972–986, 1998.

[162] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.

[163] 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 *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[164] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[165] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[166] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: In-*

ternational Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–16. IEEE, 2020.

[167] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822*, 2018.

[168] M Ramakrishna, E Fu, and E Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int. Conf. on Computing and Information*, pages 1621–1636. Citeseer, 1994.

[169] Kiran Ranganath, Joshua D Suetterlein, Joseph B Manzano, Shuaiwen Leon Song, and Daniel Wong. MAPA: Multi-accelerator pattern allocation policy for multi-tenant GPU servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[170] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[171] Vipula Rawte, Amit Sheth, and Amitava Das. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*, 2023.

[172] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[173] Paul Resnick and Hal R Varian. Recommender systems. *Communications of the ACM*, 40(3):56–58, 1997.

[174] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.

[175] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[176] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. Accelerating collective communication in data parallel training across deep learning frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1027–1040, 2022.

[177] Corby Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. *Microsoft Blog*, 1(2), 2020.

[178] Davide Rossetti and S Team. GPUDirect: Integrating the GPU with a network interface. In *GPU Technology Conference*, page 185, 2015.

[179] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[180] Arnaud ROUGETET. selfie2anime in Kaggle. `https://www.kaggle.com/arnaud58/selfie2anime`, 2022.

[181] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[182] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[183] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[184] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[185] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[186] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[187] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[188] Shaohuai Shi, Xianhao Zhou, Shutao Song, Xingyao Wang, Zilin Zhu, Xue Huang, Xinan Jiang, Feihu Zhou, Zhenyu Guo, Liqiang Xie, et al. Towards scalable distributed training of deep learning on public cloud clusters. *Proceedings of Machine Learning and Systems*, 3, 2021.

[189] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[190] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[191] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[192] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*, 2018.

[193] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM computer communication review*, 31(4):149–160, 2001.

[194] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. Optimizing network performance for distributed DNN training on GPU clusters: ImageNet/AlexNet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855*, 2019.

[195] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[196] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[197] Amir Taherin, Tirthak Patel, Giorgis Georgakoudis, Ignacio Laguna, and Devesh Tiwari. Examining failures and repairs on supercomputers with multi-gpu compute nodes. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 305–313. IEEE, 2021.

[198] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: Active device and link failure localization in data center networks. In *NSDI*, pages 599–614, 2019.

[199] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1), 2005.

[200] Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

[201] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *NSDI*, 2023.

[202] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from GPU experience with the Titan supercomputer at Oak Ridge leadership computing facility. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[203] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[204] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[205] Andrea Vedaldi and Karel Lenc. MatConvNet: Convolutional neural networks for MATLAB. In *Proceedings of the 23rd ACM international conference on Multimedia*, pages 689–692, 2015.

[206] MK Vijaymeena and K Kavitha. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3(2):19–28, 2016.

[207] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. PowerSGD: Practical low-rank gradient compression for distributed optimization. *arXiv preprint arXiv:1905.13727*, 2019.

[208] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2016.

[209] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ML. *arXiv preprint arXiv:1910.04940*, 2019.

[210] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative deep learning for recommender systems. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1235–1244, 2015.

[211] Hongyi Wang, Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. ATOMO: Communication-efficient learning via atomic sparsification. *arXiv preprint arXiv:1806.04090*, 2018.

[212] Linnan Wang, Wei Wu, Junyu Zhang, Hang Liu, George Bosilca, Maurice Herlihy, and Rodrigo Fonseca. FFT-based gradient sparsification for the distributed training of deep neural networks. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 113–124, 2020.

[213] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pages 364–381, 2023.

[214] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Hi-speed DNN training with Espresso: Unleashing the full potential of gradient compression

with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*, pages 867–882, 2023.

[215] Zhuang Wang, Xinyu Wu, and T. S. Eugene Ng. MergeComp: A compression scheduler for scalable communication-efficient distributed training. *arXiv preprint arXiv:2103.15195*, 2021.

[216] Zhuang Wang, Xinyu Wu, Zhaozhuo Xu, and T. S. Eugene Ng. Cupcake: A compression scheduler for scalable communication-efficient distributed training. *Proceedings of Machine Learning and Systems (MLSys)*, 5, 2023.

[217] Zhuang Wang, Zhaozhuo Xu, Xinyu Wu, Anshumali Shrivastava, and T. S. Eugene Ng. DRAGONN: Distributed randomized approximate gradients of neural networks. In *International Conference on Machine Learning (ICML)*, pages 23274–23291. PMLR, 2022.

[218] Zhuang Wang, Zhaozhuo Xu, Xinyu Wu, Anshumali Shrivastava, and T. S. Eugene Ng. Zen: Near-optimal sparse tensor synchronization for distributed DNN training. *arXiv preprint arXiv:2005.14165*, 2023.

[219] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*, pages 1299–1309, 2018.

[220] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519, 2017.

[221] Udi Wieder et al. Hashing, load balancing and multiple choice. *Foundations and Trends® in Theoretical Computer Science*, 12(3–4):275–379, 2017.

[222] Patrick Henry Winston. *Artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[223] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. Error compensated quantized SGD and its applications to large-scale distributed optimization. *arXiv preprint arXiv:1806.08054*, 2018.

[224] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic deep learning in multi-tenant GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):144–158, 2021.

[225] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[226] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. Compressed communication for distributed deep learning: Survey and quantitative evaluation. Technical report, 2020.

[227] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A compressed communication framework for distributed machine learning. In *Proc. of 41st IEEE Int. Conf. Distributed Computing Systems (ICDCS)*, 2021.

[228] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. PipeMare: Asynchronous pipeline parallel DNN training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[229] Haoran You, Chaojian Li, Pengfei Xu, Yonggan Fu, Yue Wang, Xiaohan Chen, Richard G Baraniuk, Zhangyang Wang, and Yingyan Lin. Drawing early-bird tickets: Toward more efficient training of deep networks. In *International Conference on Learning Representations, 2019*, 2020.

[230] Dong Yu and Lin Deng. *Automatic speech recognition*, volume 1. Springer, 2016.

[231] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (ATC)*, pages 181–193, 2017.

[232] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. Asynchronous stochastic gradient descent for dnn training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663. IEEE, 2013.

[233] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[234] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

[235] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. MiCS: Near-linear scaling for training gigantic model on public. *Proceedings of the VLDB Endowment*, 16(1):37–50, 2022.

[236] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. PyTorch FSDP: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[237] Gengbin Zheng, Lixia Shi, and Laxmikant V Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In

2004 IEEE international conference on cluster computing, pages 93–103. IEEE, 2004.

[238] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, 2020.

[239] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[240] Yuchen Zhong, Cong Xie, Shuai Zheng, and Haibin Lin. Compressed communication for distributed training: Adaptive methods and system. *arXiv preprint arXiv:2105.07829*, 2021.

[241] Xiang Zhou, Ryohei Urata, and Hong Liu. Beyond 1 Tb/s intra-data center interconnect technology: IM-DD OR coherent? *Journal of Lightwave Technology*, 38(2):475–484, 2020.

[242] Zhi-Hua Zhou. *Machine learning*. Springer Nature, 2021.

[243] Martin Zinkevich, Markus Weimer, Alexander J Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, volume 4, page 4. Citeseer, 2010.