每个张量设置决策树，统计时间，逐步安排每个张量的压缩策略，gpu和cpu都可做压缩

Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies

Zhuang Wang
Rice University

Haibin Lin
ByteDance Inc.

Yibo Zhu
ByteDance Inc.

T. S. Eugene Ng
Rice University

Abstract

Gradient compression (GC) is a promising approach to addressing the communication bottleneck in distributed deep learning (DDL). It saves the communication time, but also incurs additional computation overheads. 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 compression, the communication schemes for compressed tensor, and so on. However, it is challenging to find the optimal compression strategy for applying GC to DDL because of the intricate interactions among tensors. To fully unleash the benefits of GC, two questions must be addressed: 1) How to express any compression strategies and the corresponding interactions among tensors of any DDL training job? 2) How to quickly select a near-optimal compression strategy?

In this paper, we propose Espresso to answer these questions. 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. Experimental evaluations show that Espresso can improve the training throughput over the start-of-the-art compression-enabled system by up to 77% for representative DDL training jobs. Moreover, the computational time needed to select the compression strategy is measured in milliseconds, and the selected strategy is only a few percent from optimal.

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567505

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → *Machine learning*.

Keywords: Distributed Systems, Systems for Machine Learning, DNN Training, Gradient Compression

ACM Reference Format:
Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567505

1 Introduction

Deep Neural Networks (DNNs) have brought remarkable success to domains such as computer vision [24, 41, 61, 64] and natural language processing (NLP) [20, 31, 39, 66]. Because today's training jobs with a single GPU typically take days and even weeks [17, 45], data-parallel distributed deep learning (DDL) has become the norm to accelerate the training with multiple GPUs [3, 27, 34, 35, 59].

However, there exists an exacerbating tension between computation and communication in DDL. The recent innovations of hardware accelerators [37, 46] and domain-specific software optimization [15, 16, 76] 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 [63]. 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 [40, 46, 49, 57, 79].

The growing concern of communication bottlenecks in DDL has motivated numerous works, such as wait-free backpropagation mechanism [35, 74], priority-based scheduling [23, 25, 50], and optimized aggregation algorithms [17, 27, 59]. However, even with the latest highly-optimized BytePS [27] 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 [51] and BERT-base [20] with 64 NVIDIA V100 GPUs in 8 machines connected by a 100Gbps Ethernet network.

Gradient compression (GC) algorithms [5, 6, 29, 36, 58, 62, 70, 71] 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 [26, 62, 72]. 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 the 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 paper, 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 the first question. The abstraction can express any possible compression options of any tensors 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 computation, 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 GPUs and CPUs 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 [9]) 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.

2 Background

2.1 Communication in DDL

In data-parallel distributed deep learning (DDL), each GPU has a replica of the DNN model. The training dataset is divided into multiple partitions and each GPU takes one partition. Training is performed in multiple iterations. At the beginning of an iteration, each GPU consumes a mini-batch of training data from its own partition. It then independently performs *forward propagation* and *backward propagation* to generate gradient tensors, which can be aggregated synchronously or asynchronously among GPUs. Synchronous data-parallel DDL, 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 [3, 27, 35, 59]; asynchronous data-parallel DDL, where GPUs do not wait for aggregation to complete, can hurt the model accuracy [13]. We focus on synchronous data-parallel DDL because of its wide adoption.

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 1) is widely applied in DDL frameworks [14, 27, 35, 59] because the intra-machine network is usually faster than the inter-machine network. There are three *phases* for gradient synchronization in hierarchical 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 [35, 59].

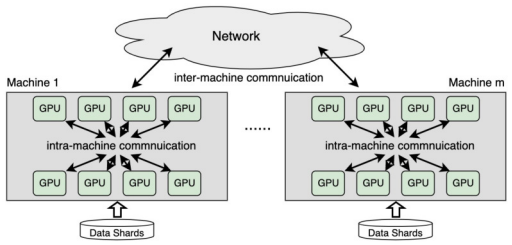


Figure 1. Hierarchical communication in DDL.

2.2 Computation and Communication Tension

Because of the layered structure and a layer-by-layer computation pattern in DNN models [10], the wait-free backpropagation mechanism (WFBP) [14, 27, 35, 59, 74] is widely adopted to overlap communication with computation in DDL to reduce the iteration time.

However, there still exists an exacerbating tension between computation and communication. The recent advancements in ML hardware accelerators [46] and specialized software stacks [15, 55, 76] 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 [63]. Faster training speed leads to more frequent gradient synchronization and higher demands on the network. Unfortunately, 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 [40, 46, 49]. This imbalance between the fast-growing computing capability and the slower-growing communication bandwidth reduces the chance to overlap communication with computation, and results in poor scalability of DDL.

To illustrate, we trained real-world DNN models on BytePS-0.2.5 [27], 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 [21, 75], 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 1. To put this into context, the training time of BERT-base is about 1200 GPU hours under ideal linear scaling [47], but in practice, it will take 2350 GPUs 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 [8].

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.

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 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.

2.3 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 [5, 36, 62] and it can save up to 99.9% of the gradient exchange while maintaining model accuracy [36]. Quantization decreases the precision of gradients; gradients in single-precision floating-point format (FP32) are mapped to fewer bits, such as 8 bits [19], 2 bits [71], and even 1 bit [11, 29, 58] to reduce the communicated traffic volume by up to 96.9%. 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 [26, 36, 58, 62, 72]. 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 [7, 44, 78]. However, the scalability improvement of DDL via GC has been still poor.

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 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 [36], to the aforementioned GPT2 training and a representative 1-bit quantization algorithm, EFSignSGD [29], to BERT-base training. The compression rate of DGC is 1%, i.e., only 1% of gradients are exchanged during synchronization. Tensors are compressed

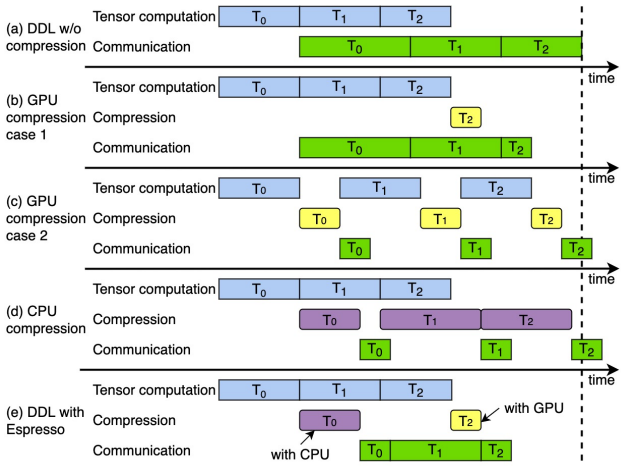


Figure 2. A DDL example with different compression strategies. (a) is the baseline; (b) reduces the iteration time, but it is not optimal; (c) and (d) harm the performance; (e) is our solution and achieves optimal performance. The communication and compression overheads depend on the interactions among tensors. The decompression operations are omitted.

with GPUs [9] or CPUs [78] in separate experiments. Compression with GPUs is typically faster than compression with CPUs [9], but it competes for the GPU resources with training [4]. As shown in Table 1, GC only achieves up to 20% training speedup, which is on par with the findings in prior works [4, 9, 73]. In fact, GC can harm performance in some situations. To illustrate, we apply DGC with 1% compression rate to the training of LSTM [41] on 64 V100 GPUs with PCIe 3.0 ×16 as the intra-machine network and 25Gbps inter-machine Ethernet.[1] As listed in Table 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.

3.1 Root Reasons of the Challenges

The choice of compression strategies determines the iteration time of compression-enabled DDL. Figure 2 is an example that shows the timelines of tensor computation, communication, and compression of DDL with different compression strategies. Figures 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 2(b) compresses T_2 with GPUs and it reduces the iteration time. Figures 2(c) and (d) compress the three tensors with GPUs and CPUs, respectively, but unfortunately, they both harm the performance of DDL. Figures 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 time of different tensors can vary. For example, in Figure 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 2(a), T_1's communication partially overlaps with T_2's tensor computation. However, in Figure 2(c), after compression, T_1's communication can completely overlap with T_2's tensor computation. Furthermore, in Figure 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 2(b), T_2's GPU compression fully overlaps with T_1's communication. In Figure 2(d), T_1's CPU compression partially overlaps with T_2's tensor computation. In Figure 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 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 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

[1] NVLink 2.0 gives every GPU in total 1.2Tbps GPU-GPU bandwidth, but PCIe 3.0 ×16 only provides ~100Gbps bandwidth [27]. PCIe-only GPU machines are common in GPU clusters that have 25Gbps Ethernet [2, 22, 27, 53].

all-gather

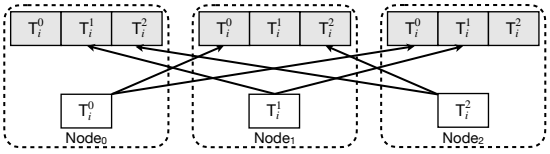


Figure 3. An indivisible scheme. T_i^j is the tensor T_i on Node j. Each node retrieves tensors from other nodes.

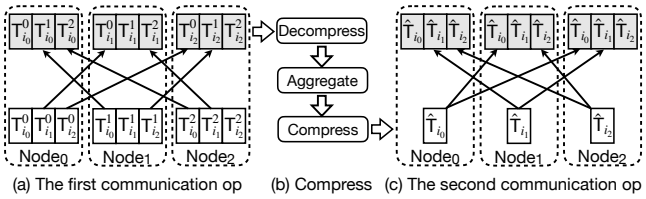


Figure 4. A divisible scheme. In (a), T_i^j is partitioned into 3 parts, i.e., $T_{i_0}^j$, $T_{i_1}^j$, and $T_{i_2}^j$. The first communication operation (op) is a shuffle. After Node j receives the j_{th} part from other nodes, it decompresses and aggregates them. It then compresses the aggregated tensor and obtains $\hat{T}_{i_j}$ for the second communication op.

为什么要分两步

scheme has two communication operations, as shown in Figure 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 performs decompression, aggregation, and the second compression operation.[2] 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 5(a), GC with a divisible scheme outperforms GC with an indivisible scheme. However, in Figure 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 the interactions among tensors.

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 1. It has intra- and inter-machine communications, and both can become the performance bottleneck.

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 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 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 to determine the right schemes make the decision of the compression choices even harder.

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 2 and Figure 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 (§4.4.1). 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.

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

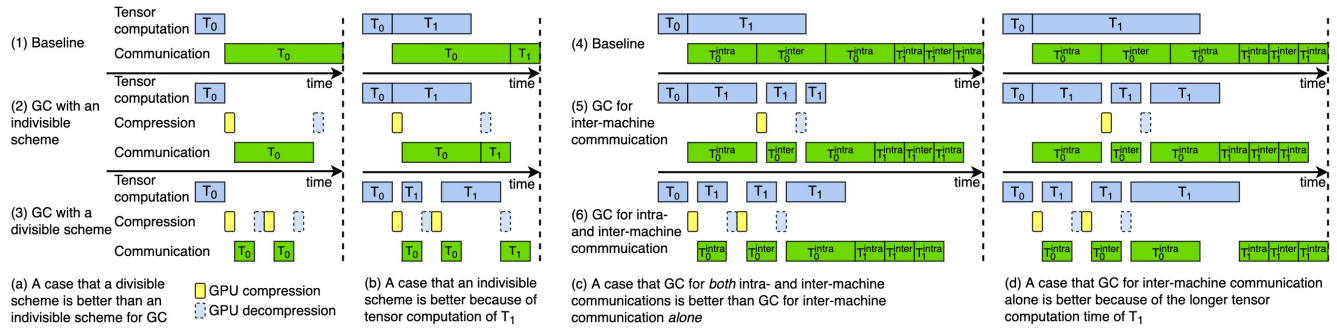


Figure 5. (a) and (b) show that the choice of communication schemes depends on the interactions among tensors. Only T_0 is compressed. (c) and (d) show that the decision to apply GC to inter-machine communication alone or to both intra- and inter-machine communications also depends on the interactions among tensors. T_i^{intra} and T_i^{inter} are T_i's intra- and inter-machine communications.

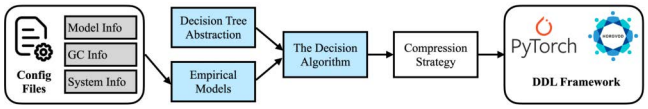


Figure 6. Espresso Overview.

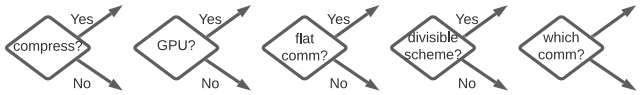


Figure 7. The decision tasks. *compress?* is for Dimension 1, *GPU?* is for Dimension 2, and the other three decision tasks are for Dimension 3. The options of Dimension 4 is illustrated in Figure 8.

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.

4 The Design of Espresso

4.1 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 types 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 model, the GC, and the training system setting, as illustrated in Figure 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.

4.2 The Decision Tree Abstraction

4.2.1 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 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.

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

Table 2. The collective routines for synchronization.

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

Espresso must determine the set of tensors that should be compressed to maximize the benefits of GC.

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 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 [4, 9, 73]?? 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 2 lists the common collective routines used in DDL for GC [1, 35, 65]. Because tensors can be communicated without GC, Table 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 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 3.

Dimension 4: the compression choice. It determines where to perform compression and decompression operations. For flat communication, it has two *communication 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 8.

4.2.2 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 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 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 8 are the invalid paths ruled out by these pruning rules.

There are five sub-trees illustrated in Figure 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

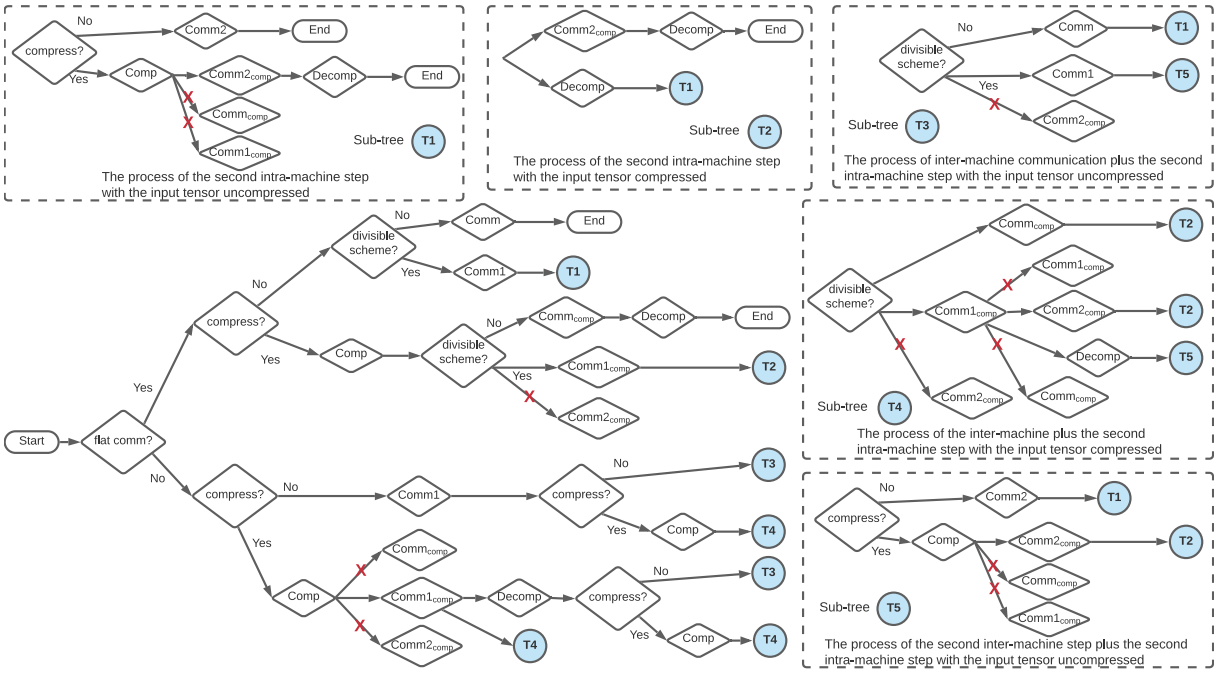


Figure 8. The decision tree abstraction for the compression options. Each diamond in the tree is a decision task.

by considering new communication schemes for GC [21, 54] and other types of compute resources [28, 67]. 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 $|\mathcal{T}| = N$. C is the set of possible compression options. $S = \{c_j\}$ is a compression strategy for the DNN model, where $c_j \in C$ is the compression option for tensor T_j.

4.3 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 the information of 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 [48, 65]. 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 its 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 keep almost constant across runs [63, 75]. 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

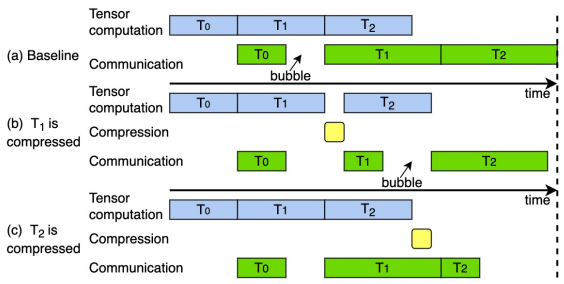


Figure 9. (a) shows that tensors communicated before bubbles need no compression. T_1 and T_2 have the same size. In (b), T_1 is compressed and a new bubble is formed. In (c), T_2 is compressed and it reduces more iteration time than compressing T_1.

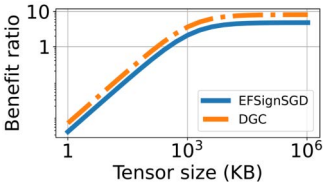


Figure 10. The benefit ratio of GPU compression.

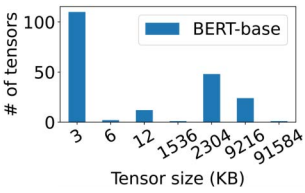


Figure 11. Number of tensors with the same sizes.

of tensor computation, communication, and compression of all tensors in a DNN model. Several timeline examples are shown in Figure 2. It can obtain the overlapping time of tensors and thus their interactions based on the timeline.

In the next section, Espresso will exploit the timeline and analyze the interactions among tensors to obtain a near-optimal compression strategy for compression-enabled DDL.

4.4 Espresso's Decision Algorithm

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

Problem. *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 $O(|C|^N)$, where N could be a few hundred and $|C|$ is 4341 based on the decision tree abstraction in Figure 8.

4.4.2 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 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 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 [59, 69]. Figure 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 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 3.1, only considering the communication and compression time for the decisions can harm the performance because they can be overlapped with other

Algorithm 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. 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

1 **Function** `Main()`:
2 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 $m_1 > \cdots > m_n$
3 sort tensors in each group of $\mathcal{G}$ in ascending order of their distances to the output layer of the DNN model
4 Remove(S, $\mathcal{G}$)
5 **for** $i \leftarrow 1$ **to** n **do**
6 **foreach** $T_j \in G_{m_i}$ **do**
`// S is updated after GetBestOption()`
7 S = GetBestOption(S, j)
8 Remove(S, $\mathcal{G}$)
9 **end**
10 **end**
11 **return** S
12 **Function** `Remove`(S, $\mathcal{G}$):
13 derive the communication timeline with compression strategy S and detect the communication bubbles; remove uncompressed tensors from $\mathcal{G}$ communicated before bubbles
14 **Function** `GetBestOption`(S, idx):
15 candidates = [S]
16 **foreach** $c_i \in C_{gpu}$ **do**
17 $S_i = S.copy()$
18 $S_i[idx] = c_i$
19 candidates.add(S_i)
20 **end**
`// F(S) is the iteration time with S`
21 **return** $\arg\min\{F(S_j) \mid S_j \in \text{candidates}\}$

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 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 $|C_{gpu}| + 1$ strategy candidates (one of them is no compression). Espresso can derive the iteration time of each candidate with the empirical models introduced in Section 4.3. Line 21 accounts for the interactions among tensors and selects the best candidate with the minimum iteration time (Property #3). After determining the 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).

4.4.3 Espresso's CPU offloading.

Espresso offloads compression from GPUs to CPUs to further improve the training throughput of DDL after Algorithm 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 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 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 1, there is no need to evaluate all possible combinations because Lemma 1 restricts the choices of tensors for CPU offloading in each group.

Algorithm 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 $0 \le u_i \le |G_i^{gpu}|$. $\mathcal{U}$ 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 1. *Algorithm 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 that in 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 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. □

5 Evaluation

5.1 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 models (VGG16, ResNet101 and UGATIT) and three NLP models (BERT-base, GPT2, and LSTM) by following the literature [21, 27, 57]. We set the batch sizes of these models by also following the literature [9, 21, 32, 42, 57]. Specifically, the per-GPU batch size is kept constant as the number of GPUs increases, and the batch sizes are modest because large batch sizes are known to cause convergence problems [57, 64]. The details of the models, datasets, and batch sizes are shown in Table 4.

Compression algorithms. We use three representative compression algorithms: Randomk [62] and DGC [36] for sparsification (1% compression rate), and EFSignSGD [29] for quantization. Error-feedback [29, 36] is applied on both GPU and CPU compression to preserve the model accuracy.

Baselines. We use BytePS [27] as the training baseline without GC (FP32). We use HiPress [9] and HiTopKComm [60] as the two baselines with GPU compression, and BytePS-Compress [78] as the baseline with CPU compression. These baselines explore narrower search spaces in comparison to Espresso and we will discuss the details in Section 6.

Performance metrics. We use trained images per second as the metric for computer vision models and tokens per second for NLP models. We measure the computational time of Espresso and 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 time 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.

5.2 End-to-End Experiments

5.2.1 DDL with NVLink-based GPU machines. Figure 12 shows the training throughput of three DNN models with Espresso and baselines. The performance bottleneck is inter-machine communication.

Model	Dataset	Batch size	Model size
VGG16 [61]	ImageNet [18]	32 images	528 MB
ResNet101 [24]	ImageNet [18]	32 images	170 MB
UGATIT [30]	selfie2anime [56]	2 images	2559 MB
BERT-base [20]	SQuAD [52]	1024 tokens	420 MB
GPT2 [51]	WikiText-2 [43]	80 tokens	475 MB
LSTM [41]	WikiText-2 [43]	80 tokens	328 MB

Table 4. Characteristics of the benchmark DNN models.

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

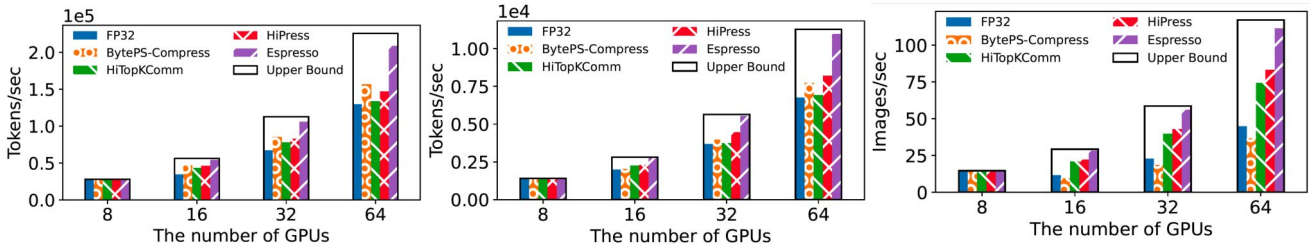
	VGG16	ResNet101	UGATIT	BERT-base	GPT2	LSTM
# of Tensors	11	42	32	54	34	5
Espresso	1ms	30ms	12ms	44ms	18ms	1ms
Brute force	1ms	> 24h	1.9h	> 24h	7.6h	1ms

Table 6. The time to find best CPU offloading solutions. # of tensors is the number of tensors for offloading.

As shown in Figure 12a, the compression baselines bring very limited speedups over FP32 for BERT-base. For example, HiTopKComm and HiPress only outperform 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 12b.

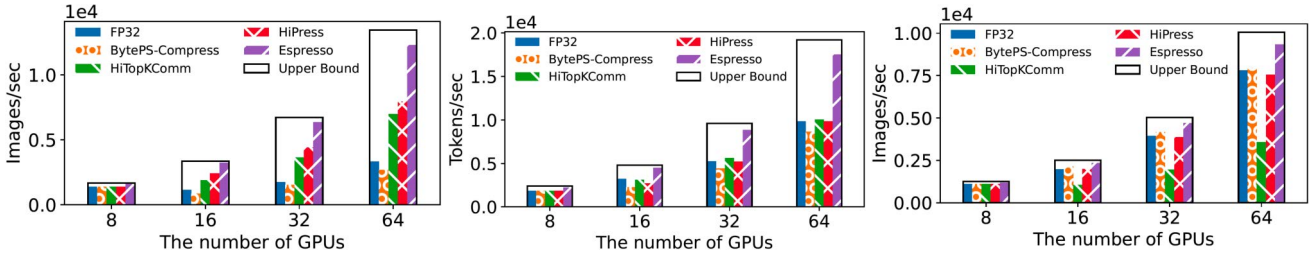
UGATIT is very communication-intensive because of its large model size. When the number of GPUs is 64, the performance improvement with HiPress and HiTopKComm is 86% and 66%, respectively, as shown in Figure 12c. BytePS-Compress even harms the performance by 18% due to the costly computational overhead 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.

5.2.2 Computational time of Espresso. Table 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 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



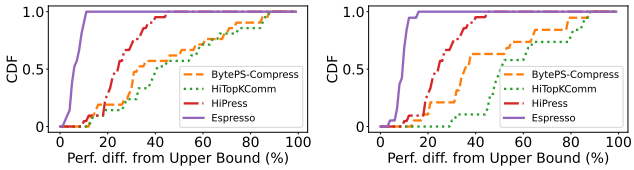
(a) BERT-base+Randomk **(b)** GPT2+EFSignSGD **(c)** UGATIT+DGC

Figure 12. Throughput of DNN models with NVLink-based GPU machines and 100Gbps cross-machine Ethernet.



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

Figure 13. Throughput of DNN models with PCIe-only GPU machines and 25Gbps cross-machine Ethernet.



(a) NVLink-based machines **(b)** PCIe-only machines

Figure 14. The performance differences between compression frameworks and Upper Bound with 64 GPUs.

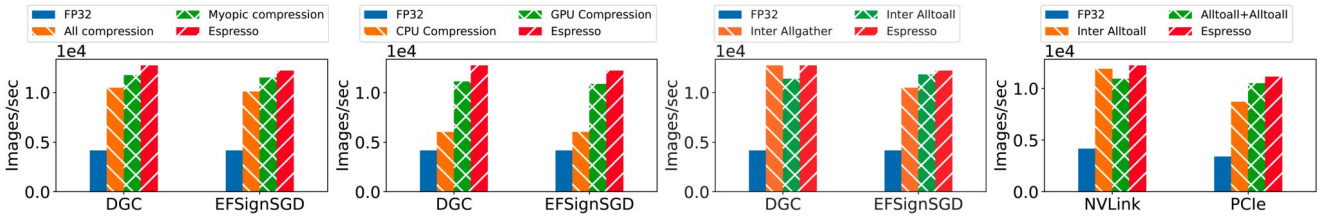
traverse all the possibilities. Even though LSTM only has 10 tensors, the searching time is still unacceptable.

Table 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 11. Espresso only needs to consider a few thousand choices to find the best CPU offloading.

5.2.3 DDL with PCIe-only GPU machines. The performance bottlenecks could be both inter- and intra-machine communications in this setup. Figure 13b shows that the three compression baselines bring almost 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 with FP32. Figure 13c shows applying GC to ResNet101 with the compression baselines can harm its performance. HiTopKComm reduces its training throughput by up to 54% because it compresses all the tensors and leads to exorbitant compression overhead. HiPress also has high over-compression penalties and it degrades the performance by 4% with 64 GPUs. In contrast, Espresso still outperforms FP32, BytePS-Compress, and HiPress by up to 20%, 18%, and 24%, respectively.

5.2.4 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-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). Due to space limitations, we present a summary of all the results for the 64-GPU scenario. Specifically, we present the



(a) Restrict Dim. 1 **(b)** Restrict Dim. 2 **(c)** Restrict Dim. 3 **(d)** Restrict Dim. 4

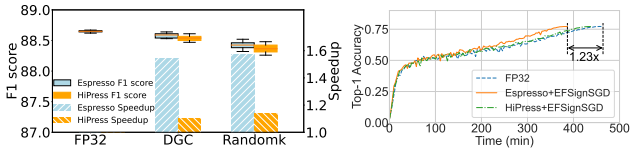
Figure 15. Considering all four dimensions is always better than considering only three dimensions.

cumulative distribution of the performance differences of each scheme from the Upper Bound. Figure 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 14b shows the distributions for all the training with PCIe-only machines and 64 GPUs and Espresso similarly out-performs other baselines.

5.3 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 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.

Figure 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



(a) BERT **(b)** ResNet101

Figure 16. Model accuracy of BERT-base (F1 score) and ResNet101 (Top-1 accuracy).

determined by Espresso always outperforms the compression strategies selected from the cripple search space. Moreover, Figure 15(c) verifies that different types of GC algorithms need different communication schemes, and Figure 15(d) verifies that different intra- and inter-machine bandwidth need different compression choices.

5.4 Convergence validation

It has been theoretically proven and empirically validated that GC can preserve the training accuracy and convergence [9, 21, 26, 36, 58, 62, 72]. In this section, we reaffirm these conclusions and demonstrate that Espresso can preserve the training accuracy and convergence.

We conduct a test following the methodology in [21] to fine-tune BERT-base for the question answering task on SQuAD [52] for two epochs and repeat the experiments ten times. The number of GPUs is 64 on 8 NVLink-based GPU machines. Figure 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 [18] from scratch and apply EFSignSGD to the model training. As shown in Figure 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%.

6 Related Work

GRACE [73] 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 are recently proposed to support compression-enabled DDL.

HiTopKComm [60] designs a new communication scheme for GC, but it compresses all tensors with GPUs and leads to prohibitive compression overhead. HiPress [9] 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 [78] 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 address both intra- and inter-machine communication bottlenecks. OmniReduce [21] introduces block gradient sparsification, which is a new type of GC algorithm, but Espresso focuses on how to efficiently apply GC to DDL.

Other than GC [5, 6, 11, 12, 29, 36, 58, 70, 71, 77], there are other approaches that aim at improving the training throughput of DDL. SwitchML [57] and ATP [33] exploit programmable switches for gradient aggregations. Other communication schemes have been proposed to more efficiently aggregate gradients. For example, BytePS [27] uses spare CPU and bandwidth resources in GPU clouds to optimize both intra- and inter-machine communications. Blink [68] generates optimal communication primitives for intra-machine communication with NVLink. PLink [38] 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 [50], P3 [25], and TicTac [23] schedule communications of tensors closer to the output layer with higher priority. These approaches are compression-agnostic; since Espresso supports compression-enabled DDL, it can be integrated with most of them.

7 Conclusion

Espresso is a general framework to enable DDL to achieve near-optimal training speed with GC. It holistically considers all the dimensions when making decisions for how to apply GC to DDL. Espresso can express any compression strategies and analyze the intricate interactions among tensors to quickly select a near-optimal compression strategy for any DDL training job. It outperforms the state-of-the-art compression-enabled systems by up to 77% across six popular DNN models and preserve model accuracy.

Acknowledgment

This work was done during Zhuang Wang's internship at ByteDance Inc. We would like to thank our shepherd Thaleia Doudali and the anonymous reviewers for providing valuable feedback. Zhuang Wang and T. S. Eugene Ng are partially supported by the NSF under CNS-2214272 and CNS-1815525.

References

[1] 2021. NVIDIA NCCL. https://developer.nvidia.com/NCCL.

[2] 2022. 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)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi22/presentation/weng

[3] 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. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[4] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2021. On the Utility of Gradient Compression in Distributed Training Systems. *arXiv preprint arXiv:2103.00543* (2021).

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

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

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

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

[9] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. *To appear in the proceedings of the 28th ACM Symposium on Operating Systems Principles* (2021).

[10] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 499–514.

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

[12] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. Adacomp: Adaptive residual gradient compression for data-parallel distributed training. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

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

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

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

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

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

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

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

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

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

[22] Google. 2021. Google Cloud. https://cloud.google.com/compute/docs/gpus.

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

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

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

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

[27] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. 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)*. 463–479.

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

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

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

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

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

[33] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning.. In *NSDI*. 741–761.

[34] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[35] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[36] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. Deep gradient compression: Reducing the communication bandwidth for distributed training. *The International Conference on Learning Representations (ICLR)* (2017).

[37] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*. 41–54.

[38] Liang Luo, Peter West, Arvind Krishnamurthy, Luis Ceze, and Jacob Nelson. 2020. PLink: Discovering and Exploiting Datacenter Network Locality for Efficient Cloud-based Distributed Training. *Proc. of MLSys* (2020).

[39] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*. 55–60.

[40] Mellanox. 2022. Mellanox Corporate Update. https://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf.

[41] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[42] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[43] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[44] Meta. 2022. Gradient Compression in Pytorch. https://pytorch.org/docs/stable/ddp_comm_hooks.html.

[45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[46] NVIDIA. 2021. A Timeline of Innovation for NVIDIA. ttps://www.nvidia.com/en-us/about-nvidia/corporate-timeline/.

[47] NVIDIA. 2021. BERT training time. https://github.com/NVIDIA/DeepLearningExamples/blob/master/PyTorch/LanguageModeling/BERT/.

[48] NVIDIA. 2021. Broadcast in NCCL. https://github.com/NVIDIA/nccl-tests/blob/master/doc/PERFORMANCE.md.

[49] OpenAI. 2021. AI and Compute. https://openai.com/blog/ai-andcompute/.

[50] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[51] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[52] Pranav Rajpurkar, Robin Jia, and Percy Liang. 2018. Know what you don't know: Unanswerable questions for SQuAD. *arXiv preprint arXiv:1806.03822* (2018).

[53] Kiran Ranganath, Joshua D Suetterlein, Joseph B Manzano, Shuaiwen Leon Song, and Daniel Wong. 2021. 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*. 1–14.

[54] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. Sparcml: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[55] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele,

Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[56] Arnaud ROUGETET. 2022. selfie2anime in Kaggle. https://www.kaggle.com/arnaud58/selfie2anime.

[57] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[58] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 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*.

[59] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[60] Shaohuai Shi, Xianhao Zhou, Shutao Song, Xingyao Wang, Zilin Zhu, Xue Huang, Xinan Jiang, Feihu Zhou, Zhenyu Guo, Liqiang Xie, et al. 2021. Towards scalable distributed training of deep learning on public cloud clusters. *Proceedings of Machine Learning and Systems* 3 (2021).

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

[62] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. In *Advances in Neural Information Processing Systems*.

[63] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. 2019. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855* (2019).

[64] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[65] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005).

[66] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[67] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. 2016. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 3 (2016), 513–517.

[68] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2019. Blink: Fast and generic collectives for distributed ML. *arXiv preprint arXiv:1910.04940* (2019).

[69] Zhuang Wang, Xinyu Wu, and TS Ng. 2021. MergeComp: A Compression Scheduler for Scalable Communication-Efficient Distributed Training. *arXiv preprint arXiv:2103.15195* (2021).

[70] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1299–1309.

[71] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*. 1509–1519.

[72] Jiaxiang Wu, Weidong Huang, Junzhou Huang, and Tong Zhang. 2018. Error compensated quantized SGD and its applications to large-scale distributed optimization. *arXiv preprint arXiv:1806.08054* (2018).

[73] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2021. GRACE: A compressed communication framework for distributed machine learning. In *Proc. of 41st IEEE Int. Conf. Distributed Computing Systems (ICDCS)*.

[74] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. 2017. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (ATC)*. 181–193.

[75] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. 2020. Is Network the Bottleneck of Distributed Training?. In *Proceedings of the Workshop on Network Meets AI & ML*. 8–13.

[76] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.

[77] Shuai Zheng, Ziyue Huang, and James Kwok. 2019. Communication-efficient distributed blockwise momentum SGD with error-feedback. In *Advances in Neural Information Processing Systems*. 11450–11460.

[78] Yuchen Zhong, Cong Xie, Shuai Zheng, and Haibin Lin. 2021. Compressed Communication for Distributed Training: Adaptive Methods and System. *arXiv preprint arXiv:2105.07829* (2021).

[79] Xiang Zhou, Ryohei Urata, and Hong Liu. 2020. Beyond 1 Tb/s intra-data center interconnect technology: IM-DD OR coherent? *Journal of Lightwave Technology* 38, 2 (2020), 475–484.