SAFusion: Efficient Tensor Fusion with Sparsification Ahead for High-Performance Distributed DNN Training

Zhangqiang Ming
Huazhong University of Science and Technology
Wuhan, Hubei, China
zqming@hust.edu.cn

Yuchong Hu*
Huazhong University of Science and Technology
Wuhan, Hubei, China
Shenzhen Research Institute of Huazhong University of Science and Technology
Shenzhen, Guangdong, China
yuchonghu@hust.edu.cn

Xinjue Zheng
Huazhong University of Science and Technology
Wuhan, Hubei, China
zhengxinjue@hust.edu.cn

Wenxiang Zhou
Huazhong University of Science and Technology
Wuhan, Hubei, China
wxzhoucs@hust.edu.cn

Dan Feng
Huazhong University of Science and Technology
Wuhan, Hubei, China
dfeng@hust.edu.cn

ABSTRACT

Distributed deep neural networks (DNN) training systems deployed across workers have been widely used in various domains, while the communication overhead among workers for synchronizing gradient tensors often becomes the performance bottleneck. To optimize communication efficiency, state-of-the-art studies often apply both two techniques: i) gradient sparsification compression, which truncates the gradient to its largest elements to reduce the communication traffic, and ii) tensor fusion, which merges multiple gradient tensors within a fusion buffer to transmit them together to reduce the communication startup overhead. However, we find that existing studies often apply gradient sparsification *after* tensor fusion (we call *sparsification-behind tensor fusion*), which leads to a fact that a lot of fused gradient tensors are missed after the sparsification, thus impairing the convergence performance.

In this paper, we propose a new tensor fusion mechanism for high-performance distributed DNN training, called SAFusion, which performs sparsification on each of gradient tensors *before* merging them during tensor fusion, instead of sparsification-behind tensor fusion, so as to avoid gradient tensor missing and thus improve the convergence performance. Further, we find that SAFusion will face two limitations: the long synchronization waiting time across different workers and many waiting periods within each worker during tensor fusion, so we propose two efficient (inter-worker and intra-worker) tensor fusion schemes to address the two limitations. We prototype SAFusion, and experiments on local and cloud clusters show that SAFusion improves the training throughput by 49.71%-144.42% over training without tensor fusion, and by 18.96%-60.89% over state-of-the-art tensor fusion schemes, while maintaining approximately the same convergence performance as the non-compression baseline.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Distributed computing methodologies**.

KEYWORDS

High Performance, Distributed DNN Training, Gradient Sparsification Compression, Tensor Fusion

ACM Reference Format:
Zhangqiang Ming, Yuchong Hu, Xinjue Zheng, Wenxiang Zhou, and Dan Feng. 2025. SAFusion: Efficient Tensor Fusion with Sparsification Ahead for High-Performance Distributed DNN Training. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3731545.3731581

1 INTRODUCTION

State-of-the-art deep neural network (DNN) models have been largely used in AI applications [11, 14, 40, 56], with a rapid growth in the number of model parameters and training datasets. For example, in natural language processing, BERT-large [14] has 340 million parameters, and GPT-3 [11] has 175 billion parameters. Since training for these tasks with a single GPU typically takes days or even weeks, high-performance DNN training systems are commonly deployed across distributed GPUs to accelerate the training process of large DNN models [10, 25, 30, 42, 47]. In a typical setting, multi-GPU training jobs process different data samples, and each GPU node (or *worker*) has a subset of the input data (called *mini-batch*). For each

*Corresponding author: Yuchong Hu, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China.

HPDC '25, July 20–23, 2025, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07…$15.00
https://doi.org/10.1145/3731545.3731581

training iteration, each worker first computes the gradients based on its mini-batch and exchanges them through synchronization strategies [19, 25] (i.e., Parameter Server or Ring-AllReduce), before updating their parameters at the end of the iteration.

Despite being widely used, distributed DNN training systems have to face a performance problem: frequent and heavy gradient synchronization among all training workers in each iteration makes the gradient communication time often take up a significant portion of the overall training time (as high as 90%) [10, 14, 56], which becomes the main performance bottleneck [10, 57]. *Gradient sparsification* is one of the major compression algorithms to alleviate the communication bottleneck by selecting a subset of the gradient tensor for reducing gradient communication traffic [17, 29, 31, 34, 37, 45, 48, 53, 57], which has been adopted by the industry, e.g., Uber, ByteDance, and Microsoft [1–3].

Besides gradient sparsification, *tensor fusion* is another communication optimization technique and has been applied in both academics [25, 32, 33, 47, 49, 51, 67, 68] and industry (e.g., PyTorch-DDP [30], Horovod [47], DeepSpeed [2], and BytePS [1]). The tensor fusion technique merges multiple gradient tensors into a buffer (called *fusion buffer*) before performing gradient synchronization to reduce the communication startup overhead. Recently, state-of-the-art studies have begun to combine the above two techniques (i.e., tensor fusion and sparsification compression) in high-performance distributed DNN training systems [29, 51, 58, 67], so as to embrace both of their advantages in terms of communication efficiency.

However, we first find that the above existing works often perform an overall gradient sparsification compression *after* tensor fusion by default (see §2.3), which we call *sparsification-behind tensor fusion*, and then we observe a fact that sparsification-behind tensor fusion will miss many gradient tensors and thus impair the convergence performance significantly (see §2.4). Fortunately, we also observe that if we just perform gradient sparsification *before* tensor fusion (see §2.4), then no gradient tensor missing occurs and the convergence performance can be maintained. This motivates our main idea of performing tensor fusion with sparsification ahead, called *sparsification-ahead tensor fusion*.

In this paper, we propose SAFusion based on the above main idea, which performs **s**parsification **a**head of tensor **fusion**, so as to avoid gradient tensor missing for high-performance distributed DNN training. Further, we find that SAFusion gives rise to two limitations, including the long synchronization waiting time among different workers and numerous waiting periods within each worker during tensor fusion. Correspondingly, we propose two efficient (inter-worker and intra-worker) tensor fusion schemes to enhance the performance of SAFusion. Our contributions include:

- We conduct measurement analysis that first shows the challenge of existing schemes based on sparsification-behind tensor fusion, and motivates our main idea of sparsification-ahead tensor fusion. We propose SAFusion which performs sparsification before tensor fusion to address the above challenge (§2).
- We design the architecture of SAFusion that performs sparsification before tensor fusion. Atop SAFusion, we propose an inter-worker gradient-aligned fusion scheme that avoids long gradient synchronization waiting time across all workers, and an intra-worker adaptive fusion buffer sizing scheme that reduces multiple waiting periods within each worker (§3).
- We implement a distributed DNN training framework with SAFusion and state-of-the-art tensor fusion schemes (OkTopk [29], OMGS [51], and Cupcake [58]). We also integrate some state-of-the-art sparsification compression algorithms to support different sparsification compression schemes (§4). SAFusion is open-sourced at https://github.com/YuchongHu/SAFusion.
- We perform both local and cloud cluster experiments with seven popular DNN models. Experimental results show that compared to training without tensor fusion and state-of-the-art tensor fusion schemes, SAFusion increases the training throughput by 49.71%-144.42% and 18.96%-60.89%, respectively, while maintaining almost the same convergence performance as the baseline without sparsification compression (§5).

2 BACKGROUND AND MOTIVATION

2.1 DNN Training Systems

Basic. A DNN model contains multiple neural network *layers*, each with many model *parameters*. To reach the model *convergence* (i.e., the model training process reaches a stable state), DNN training iterates over a dataset many times (i.e., *epochs*), each of which contains multiple *iterations*. In each iteration, the model runs *forward* and *backward* propagation to generate a *gradient* for each layer so as to update the model parameters. Note that a gradient is often represented in the form of a *tensor*, so the *gradient tensor* specifically refers to the tensor-valued representation of the gradient.

Distributed DNN training systems. Distributed DNN training systems are deployed across multiple GPUs (or *workers*) to accelerate the training process of large DNN models, often in a *data-parallel* way [10, 37, 57]. Specifically, the dataset is split into *mini-batches* and then assigned to different workers in each iteration. Each worker processes a mini-batch of training data from its own partition, independently performing forward and backward propagation to generate gradient of each layer. Each worker's gradient is communicated with the other workers using a synchronous collective operation, such as all-reduce [6] or all-gather to update the model's parameters.

2.2 Communication Performance Bottlenecks

Gradient communication is usually composed of transmission and startup overhead [50, 58] (i.e., the gray and green parts in Figure 1(a)), so the performance bottleneck is mainly caused by *huge transmission traffic* and *frequent communication startups*.

Huge transmission traffic. Modern deep learning models scale rapidly from millions to billions of parameters [11, 14], which makes a huge transmission traffic between workers during gradient synchronization. Consequently, the huge transmission traffic causes the transmission time to take up a large portion (as high as 90% [10, 39, 46, 57, 58, 61]) of the overall training time, making the transmission be one of the major performance bottlenecks of distributed DNN training systems [10, 57].

Frequent communication startups. Another performance bottleneck in distributed training systems is a number of communication startup overhead caused by the traditional gradient synchronization [51]. Specifically, the traditional gradient synchronization scheme often leverages *wait-free backpropagation* (WFBP) [9, 47, 66] that

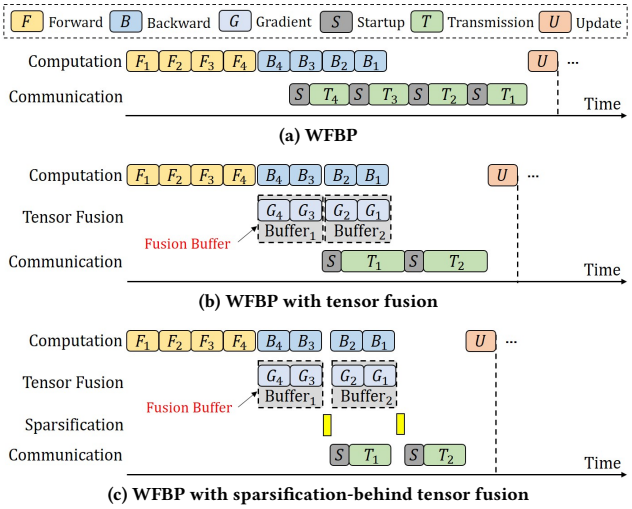


Figure 1: Three examples of distributed DNN training with four tensors for gradient synchronization.

can make backward propagation overlapped with the gradient communication by exploiting the layer-wise structure of DNN models. As shown in Figure 1(a), the backpropagation time of four layers (i.e., B_4, B_3, B_2 and B_1) can be overlapped with the transmission and startup time of the four layers (i.e., T_4, T_3, T_2, T_1 and S). However, in many DNN models, there exist a large number of small layers that lead to frequent communication startups, thus incurring significant communication startup overhead [24, 49–51, 58].

2.3 Communication Optimization Techniques

To address the above communication performance bottlenecks in distributed training systems, two major communication optimization techniques have been proposed as follows.

Gradient sparsification compression. To reduce the transmission traffic in distributed DNN training systems, *gradient sparsification* compression is a key approach to leverage the sparsity of the gradient tensor by only selecting a subset of its all *elements* (in terms of their absolute values) for gradient communication [7, 18, 31, 43, 45, 48, 52, 55, 60, 62, 69], while maintaining almost the same convergence accuracy [8, 53, 59]. The sparsification compression approach has been widely adopted by the industry, e.g., Uber, ByteDance and Microsoft [1–3, 10, 57]. However, the sparsification method often incurs significant computation overhead due to its additional compression, so many state-of-the-art studies [17, 29, 31, 34, 45, 48] have focused on the *threshold-based* sparsification, which only transmits each gradient's elements that are greater than a given threshold and is very GPU-friendly with a time complexity of only O(N) [29, 45]. In this paper, we use threshold-based sparsification as the compression baseline.

Tensor fusion. To alleviate the impact of the frequent communication startups, *tensor fusion*, a typical communication scheduling technique, has been widely studied [23, 44, 47, 50, 51, 58]. It merges multiple gradient tensors into a buffer (called *fusion buffer*), such that they can perform gradient synchronization together (instead of individually) to have only one startup (instead of multiple startups), thus reducing the communication startup overhead. In practice,

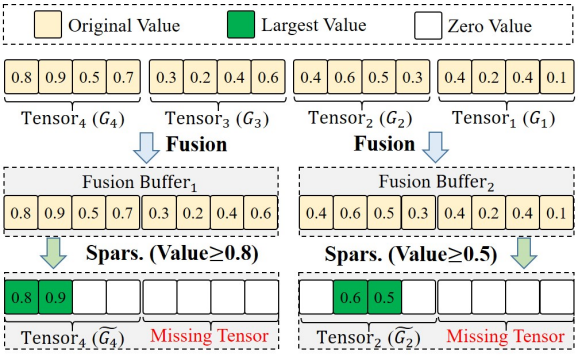


Figure 2: Example of the traditional sparsification-behind tensor fusion (density = 25%). Note that some fused gradient tensors (e.g., G_3 and G_4) are missing after sparsification.

tensor fusion is a promising technique that has been widely used in mainstream distributed DNN training systems such as PyTorch-DDP [30], Horovod [47], DeepSpeed [42], and BytePS [1]. Note that we define the process where some gradient tensors are merged into the fusion buffer as the *fusion phase*.

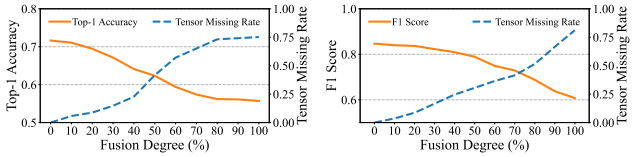
As shown in Figure 1(b), every two gradient tensors (e.g., G_4 and G_3; G_2 and G_1) are merged into the fusion buffer (e.g., $Buffer_1$ for G_4 and G_3, and $Buffer_2$ for G_2 and G_1) after the backward propagation and then the buffer's tensors are transmitted together (e.g., transmission time T_1 for $Buffer_1$ and transmission time T_2 for $Buffer_2$). Note that compared with Figure 1(a), the communication time is reduced due to tensor fusion in Figure 1(b).

Combining tensor fusion with sparsification. To further improve the communication efficiency, recent state-of-the-art studies (e.g., OkTopk [29], OMGS [51], and Cupcake [58]) focus on combining tensor fusion and gradient sparsification compression, which first performs tensor fusion that merges multiple gradient tensors and then performs sparsification compression that compresses all the fused gradient tensors in the fusion buffer, which we call *sparsification-behind tensor fusion*. As shown in Figure 1(c), the gradient tensors G_4 and G_3 that are ready in the backward propagation are first merged to the fusion buffer $Buffer_1$ and then are sparsified overall, whose transmission time T_1 is less than that in Figure 1(b). Similarly, T_2 in Figure 1(c) is also less than that in Figure 1(b). Thus, compared with Figure 1(b), the transmission time is further reduced in Figure 1(c). In this paper, we will give the challenge of the above sparsification-behind tensor fusion (see §2.4).

2.4 Challenge and Motivation

We conduct measurement analysis to give two observations that show: i) the challenge of existing sparsification-behind tensor fusion schemes and ii) the motivation of our proposed sparsification-ahead tensor fusion.

Measurement setting. We evaluate four typical training tasks (specified in §5.1) on a local cluster with 8 NVIDIA V100 GPUs connected by a 100Gbps InfiniBand, including ResNet-152 [21] and VGG-19 [54] on Cifar-100 [27], and BERT-base [14] on SQuAD [41]. The performance metrics are expressed as: Top-1 accuracy on ResNet-152 and VGG-19, and F1 score on BERT-base. We use



(a) ResNet-152 [21] on Cifar-100 [27] **(b) BERT-base [14] on SQuAD [41]**

Figure 3: The tensor missing rate and convergence accuracy for two training tasks with different fusion degree.

DGC [31] as the sparsification compression approach with density = 0.01, while Baseline indicates the non-compression scheme.

Observation #1 (Challenge): Sparsification-behind tensor fusion causes gradient tensor missing, which leads to a low convergence accuracy. First, we find that gradient tensors are often missing atop sparsification-behind tensor fusion. Specifically, we analyze the process of Figure 1(c) in details, as shown in Figure 2, the DNN model includes four gradient tensors (from G_4 to G_1) with a total of sixteen elements, each of which has four elements (to describe the gradient tensor easily, we only take the absolute values of all elements). If we perform sparsification behind tensor fusion, G_4 and G_3 are first merged into $Buffer_1$, G_2 and G_1 are merged into $Buffer_2$, and then we perform sparsification (where we select gradient elements with $value \geq 0.8$ and $value \geq 0.5$, respectively.) on two fused tensors in $Buffer_1$ and $Buffer_2$, individually. Clearly, for $Buffer_1$, the sparsified elements ($\tilde{G_4}$) are only concentrated in G_4, no elements of G_3 are selected; similarly, for $Buffer_2$, the sparsified elements ($\tilde{G_2}$) are only concentrated in G_2, no elements of G_1 are selected. Thus, the two tensors G_3 and G_1 are actually missing.

To verify the above finding, we give the first observation based on our measurement setting. To show different extent of tensor fusion, we give a new definition "fusion degree" where $fusion\ degree$ = 100% means all the tensors are fused together in a single buffer and $fusion\ degree$ = 0% means no tensors are fused by putting every tensor into an individual buffer. Figure 3 shows that when $fusion\ degree$ = 0%, no tensor is missing; as the fusion degree increases (tensors continue to be fused), the number of missing tensors is increasing significantly, and when $fusion\ degree$ = 100%, there are almost 75% of the tensors are missing (or we say the *tensor missing rate* = 75%).

Figure 3 also shows the impact of the tensor missing rate on the convergence performance. We define the *tensor missing rate* of the model as the number of missing tensors caused by sparsification-behind tensor fusion divided by the total number of all gradient tensors of the model. For example, Figure 3(a) shows that, for ResNet-152 on Cifar-100, when the fusion degree from 0% to 100%, the tensor missing rate ranges from 0% to 75%, and the Top-1 accuracy ranges from 0.72 to 0.56. Similarly, Figure 3(b) shows that, for BERT-base on SQuAD, when the fusion degree ranges from 0% to 100%, the tensor missing rate ranges from 0% to 81%, and the F1 score ranges from 0.84 to 0.61. Observation #1 implies that tensor missing after compression will lead to a degradation of the convergence performance, as confirmed in existing studies [16, 28, 56].

Observation #2 (Motivation): Tensor fusion with sparsification ahead can avoid gradient tensor missing. Observation #1 indicates that sparsification-behind tensor fusion causes tensor missing and convergence performance degradation. It's natural to ask what if we can perform sparsification *before* merging them during tensor fusion (called *tensor fusion with sparsification ahead*), instead of sparsification-behind tensor fusion?

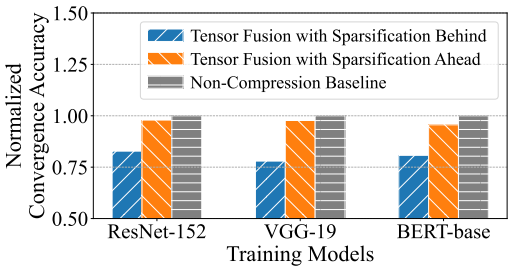


Figure 4: Comparison between sparsification-behind (traditional) tensor fusion and sparsification-ahead tensor fusion in terms of baseline-based normalized convergence performance.

Therefore, we compare the normalized convergence accuracy between tensor fusion with sparsification behind and tensor fusion with sparsification ahead as shown in Figure 4. We follow the literature [50, 51, 58] for setting the fusion buffers. We consider the non-compression baseline as the normalized convergence accuracy (i.e., its value is 1). We can see that for ResNet-152, VGG-19, and BERT-base, the convergence accuracy of the tensor fusion with sparsification ahead is 19.4%, 26.8%, and 17.6% higher than that of the tensor fusion with sparsification behind, respectively, and is very close to that of the non-compression baseline.

Analysis. We think that the reason why sparsification-ahead tensor fusion outperforms sparsification-behind tensor fusion is that the latter misses multiple tensors (as stated in Observation #1), while the former only misses some elements of tensors instead of tensors.

Specifically, since the distributions of the different gradient tensors are very different, some gradients have larger magnitudes (absolute values of the elements) and other gradients have smaller magnitudes [20, 48, 69]. As shown in Figure 2, the gradient tensors G_4 and G_2 have larger magnitudes, while G_3 and G_1 have smaller magnitudes. Thus, the gradient elements selected by the sparsification-behind tensor fusion scheme tend to be concentrated in the gradient tensors with larger magnitudes (G_4 and G_2), while the gradient elements of the tensors with smaller magnitudes (G_3 and G_1) are not selected, leading to tensor missing.

On the contrary, the sparsification-ahead tensor fusion scheme applies sparsification independently to each tensor before merging. This ensures that elements are selected from both large- and small-magnitude tensors (without elements selected in sparsification-behind tensor fusion). As shown in Figure 5, the gradients G_4, G_3, G_2, and G_1 are sparsified individually before merging, and each gradient retains at least one element, thus avoiding tensor missing.

Main idea: Observation #2 motivates us to recombine tensor fusion with gradient sparsification compression to improve the convergence performance. Therefore, we propose SAFusion, a sparsification ahead tensor fusion mechanism, whose main idea is to perform gradient sparsification on each gradient before tensor fusion, instead of the existing mechanism that performs an overall gradient sparsification for the entire fused gradients after tensor fusion.

While SAFusion can help address the challenge of degraded convergence performance, how to design SAFusion in details for high communication performance still remains unexplored, which will be specified in §3.

Table 1: Frequently used variables and notations

Notation	Description
K	The number of workers in the cluster.
T	The number of iterations for the entire training.
L	The number of all layer gradients of the DNN model.
M	The number of fusion phases (or fusion buffers) within one iteration.
$\tilde{G}_l$	The sparsified gradient of l-th DNN layer.
N_i	The number of merged layer gradients in i-th buffer.
T_b	Backward computation time of all layer gradients.
T_p^i	Backward computation time of merged layer gradients in i-th buffer.
T_{cm}^l	Communication time of l-th layer gradient.
T_{cm}^i	Communication time of sparsified gradients in i-buffer.
T_{cp}^l	Compression time of l-th layer gradient.
T_{cp}^i	Compression time of sparsified gradients in i-th buffer.

3 SAFUSION DESIGN

In this section, we design SAFusion based on the main idea in §2.4 with the design goals as follows:

- **Improving convergence performance:** SAFusion avoids gradient tensor missing via a sparsification-ahead tensor fusion mechanism to improve the convergence performance (§3.1).
- **Reducing inter-worker synchronization waiting:** SAFusion aligns the number of sparsified gradients among fusion buffers of different workers to reduce the long inter-worker gradient synchronization waiting (§3.2).
- **Improving intra-worker pipeline efficiency:** SAFusion adjusts each worker's fusion buffer size according to the historical record of the fusion buffer to improve the intra-worker training pipeline efficiency (§3.3).

3.1 Sparsification-Ahead Tensor Fusion

Based on the main idea in §2.4, our sparsification-ahead tensor fusion mechanism is composed of the following three steps. We first perform the threshold-based gradient sparsification compression (Step-1), then merge sparsified gradients into the fixed-size fusion buffer (Step-2), and finally pull fused gradients for synchronization (Step-3). Note that for ease of presentation, we list the frequently used variables and notation throughout this paper in Table 1. We specify the above three steps as follows.

Step-1: Compressing gradients via threshold-based sparsification. To realize sparsification-ahead tensor fusion for different workers, SAFusion i) first flattens the l-th layer's gradient tensor G_l ($1 \le l \le L$) as a one-dimensional vector; ii) then compresses G_l by selecting its elements whose absolute values are larger than a defined threshold τ, i.e., $|G_l| \ge \tau$, where the threshold-based sparsification is state-of-the-art (see §2.3); iii) finally obtains the sparsified gradient tensor $\tilde{G} = \left[\tilde{G}_1, \tilde{G}_2, ..., \tilde{G}_L\right]$. This means that some elements of each tensor are selected for tensor fusion instead of missing the entire gradient tensor. For example, Figure 5 shows that the gradient tensors G_1, G_2, G_3, and G_4 are first compressed by sparsification into $\tilde{G}_1$, $\tilde{G}_2$, $\tilde{G}_3$, and $\tilde{G}_4$ (where SAFusion selects the gradient elements in G_1, G_2, G_3, and G_4 with $value \ge 0.9$, $value \ge 0.6$, $value \ge 0.6$, and $value \ge 0.4$, respectively).

Step-2: Merging multiple sparsified gradients into the fusion buffer. After sparsification, SAFusion merges the sparsified gradient tensors into the fusion buffer. Specifically, SAFusion first adds a new fusion *status* for each sparsified gradient to the DNN model parameters, and this status is initially set to 0 to show that each sparsified gradient has not been merged, and then set to 1 when it has been merged. Then, SAFusion sequentially obtains the ready sparsified gradient $\tilde{G}_l$. If the merge status of $\tilde{G}_l$ is 0, SAFusion merges $\tilde{G}_l$ into the i-th fusion Buffer$_i$ ($1 \le i \le M$), while setting its status to 1 to indicate that they have been merged. Finally, SAFusion performs the ALLGATHER operation on Buffer$_i$, while starting the new sparsification-ahead tensor fusion in the $(i+1)$-th fusion phase. Figure 5 shows the sparsified tensors $\tilde{G}_1$ and $\tilde{G}_2$ are merged into $Buffer_1$, $\tilde{G}_3$ and $\tilde{G}_4$ are merged into $Buffer_2$.

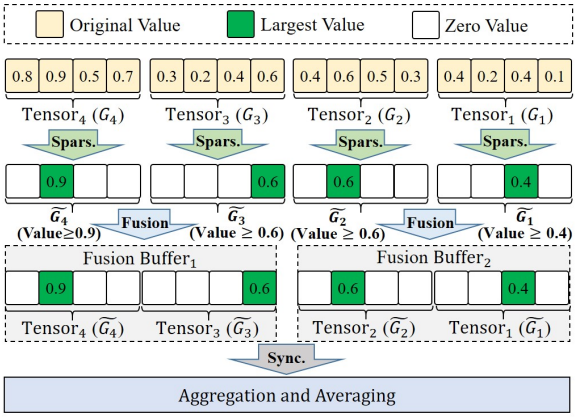


Figure 5: SAFusion **merges multiple sparsified gradient tensors into the fusion buffer for gradient synchronization (density = 25%).**

Step-3: Synchronizing the gradient tensors in the fusion buffer. After completing the buffer-based transmission, SAFusion needs to pull all sparsified gradients from the fusion buffer for gradient synchronization. Specifically, each worker gets the sparsified gradients from other workers and the basic buffer information (such as sparsified gradient length, shape, indices, values, etc) from other workers. First, SAFusion splits the fusion buffer into multiple independent sparsified gradients based on the lengths of all the sparsified gradients in the fusion buffer. Then, SAFusion decompresses the sparsified gradients in the fusion buffer to the original gradient size and shapes according to the indices and values of the sparsification compression. Finally, every worker obtains an aggregate and averaging of decompressed gradient on all workers to achieve synchronization, typically $\bar{G}_l = \frac{1}{K}\sum_{k=1}^{K} G_l^k$, where G_l^k is the gradient of the l-th layer of the k-th worker. For example, Figure 5 shows that all the sparsified gradient tensors in two fusion buffer are synchronized with the other workers, and then aggregated and averaged for updating parameters.

3.2 Aligned Inter-Worker Gradient Fusion

Limitation of SAFusion: Long inter-worker gradient synchronization waiting. We find that there exist performance limitation of SAFusion when it comes to tensor fusion cross workers (called *inter-worker fusion*). To analyze the limitation, we break down SAFusion's inter-worker fusion and gradient synchronization processes. As shown in Figure 6, we define the runtime in which the sparsified gradients are merged into the fusion buffer during backward propagation as the *tensor fusion phase*, and the sparsified gradient

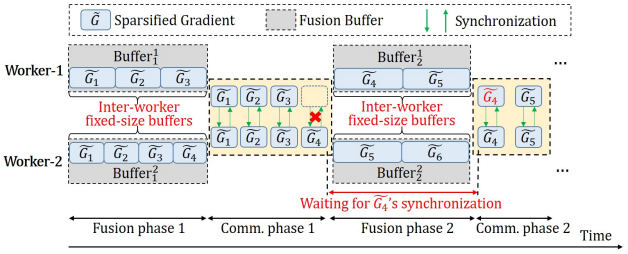


Figure 6: SAFusion may have long synchronization waiting. Note that the gradient $\tilde{G}_4$ of Worker-1 has to wait for fusion phase 2 before performing synchronization.

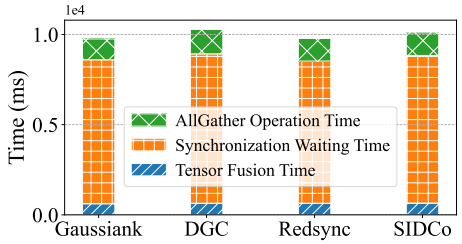


Figure 7: The breakdown of the training time in SAFusion with fixed buffer size under different sparsification algorithms [4].

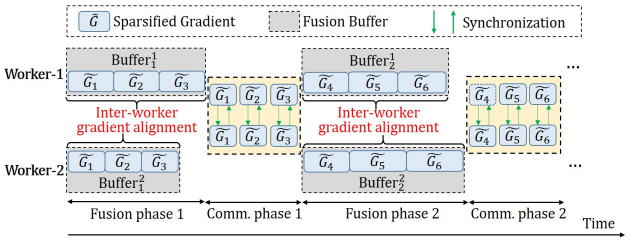


Figure 8: Example of aligned inter-worker tensor fusion.

transmission and synchronization runtime as the *communication phase*. We have two observations in Figure 6 as follows:

- Fixed-size inter-worker buffer. As with existing sparsification-behind tensor fusion schemes [44, 47, 51, 58], SAFusion sets the size of all workers' buffers at the same fusion phase (called *inter-worker buffer*) to be the same for different workers by default in the same fusion phase; that is, the inter-worker buffers are of fixed size. Figure 6 shows SAFusion has fixed inter-worker buffer size (e.g., Worker-1's $Buffer_1^1$ size = Worker-2's $Buffer_1^2$ size, where $Buffer_i^k$ is the i-th fusion buffer of the k-th worker) in the same fusion phase.
- Varied-size sparsified gradients of different workers. As stated in §3.1, SAFusion adopts threshold-based sparsification schemes, but the threshold is often estimated by sampling [29, 31, 34, 45, 64] and thus different workers may have thresholds of different size, such that the selected sparsified gradient (larger than the estimated threshold) across workers are also varied-size. Figure 6 shows the compressed gradients $\tilde{G}_1, \tilde{G}_2$ and G_3 in $Buffer_1^1$ are larger than the sparsified gradients $\tilde{G}_1, \tilde{G}_2$ and $\tilde{G}_3$ in $Buffer_1^2$ at the fusion phase 1 when using threshold-based sparsification.

Based on the above two observations, we see that SAFusion's fixed-size inter-worker buffer is *inconsistent* with its inter-workers varied-size sparsified gradients, which causes different workers to have different numbers of sparsified gradients at the same fusion phase, thus some gradients may have to wait a long time for synchronization. As shown in Figure 6, although the two inter-worker buffers $Buffer_1^1$ and $Buffer_1^2$ are fixed-size, the former has three sparsified gradients $\tilde{G}_1, \tilde{G}_2$, and $\tilde{G}_3$ while the latter has four sparsified gradients $\tilde{G}_1, \tilde{G}_2, \tilde{G}_3$, and $\tilde{G}_4$, since the former has larger sparsified gradients while the latter has smaller ones due to their different estimated thresholds. In this case, the gradient $\tilde{G}_4$ of Worker-1 only arrives in $Buffer_2^1$ during the fusion phase 2, causing $\tilde{G}_4$ of Worker-2 to have to wait for the next fusion phase to complete its synchronization.

To show the seriousness of long inter-worker synchronization waiting caused by SAFusion, we break down the training time on ResNet-152 with four sparsification algorithms (Gaussiank [48], DGC [31], Redsync [17], and SIDCo [34]) into three parts using Horovod's `Timeline` [4, 22] command in Figure 7. We set a uniform compression rate (i.e., density=0.01), and use Horovod's default fixed buffer size [47]. Here, the *tensor fusion time* and *synchronization waiting time* indicate the time taken to copy sparsified gradient into and out of the fixed-size buffer, respectively, called `MEMCPY_IN_FUSION_BUFFER` and `MEMCPY_OUT_FUSION_BUFFER` in `Timeline`. The *allgather operation time* indicates the time taken to do the actual gradient tensor operation on GPU using NCCL or MPI, called `ALLREDUCE_OR_ALLGATHER` in `Timeline`. We can see that the training time for all four methods is mainly concentrated in the synchronization waiting for the sparsified gradients out the fixed-size buffer, and the proportion even exceeds 80%.

Design idea: To address the above performance limitation that rises due to different number of sparsified gradients of inter-worker fusion buffers during the same fusion phase, we *align* the same number of variable-size sparsified gradient tensors of the inter-worker buffers during the same fusion phase.

Design details: Specifically, we first define $Buffer_i^k$ as the i-th fusion buffer of the k-th worker, where $1 \le i \le M$ and $1 \le k \le K$. Then, we generate inter-worker fusion buffers by aligning the number of variable-size sparsified gradients as follows.

Step-1: SAFusion aligns the number of gradients of each fusion buffer as $\lceil L/M \rceil$, meaning that the number of gradients in the i-th fusion $N_i = \lceil L/M \rceil$.

Step-2: SAFusion traverses the `named_parameters` of the DNN model in reverse to obtain information on the model's original gradients, including the name, length, shape, buffer number, and merge status, and use sparsification to compress the gradient of l-th layer ($1 \le l \le L$) during backward pass;

Step-3: In each worker, SAFusion sequentially selects the l-th sparsified gradient to merge into the buffer $Buffer_i^k$ and number it's buffer as i;

Step-4: SAFusion counts the number of sparsified gradients that are merged in $Buffer_i^k$, denoted as n. When $n = N_i$, SAFusion finishes the i-th fusion phase for all workers and generate the $(i+1)$-th fusion buffer. SAFusion stores the buffer number corresponding to each gradient generated in the initialization phase in memory to guide the tensor fusion in subsequent iterations.

Figure 8 shows an example of the aligned inter-worker gradient tensor fusion scheme for optimizing SAFusion, where the number of

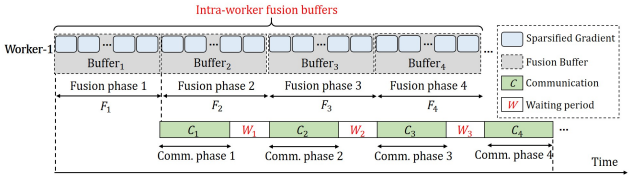


Figure 9: A general case of intra-worker pipeline for SAFusion.

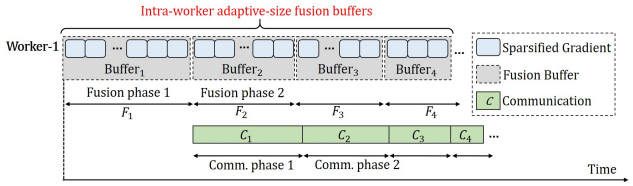


Figure 10: Example of intra-worker tensor fusion with adaptive buffer size.

sparsified gradients is L=6 (i.e., $\tilde{G}_1$, $\tilde{G}_2$,...,$\tilde{G}_6$). The number of fusion buffers is M=2 (i.e., $Buffer_1^1$ and $Buffer_2^1$). Thus, the number of gradients in the first and second fusions is $N_1 = N_2 = \lceil L/M \rceil = 3$. We can see that although the size of sparsified gradients becomes variable in the fusion phase 1 among different workers (i.e., $\tilde{G}_1$, $\tilde{G}_2$, and $\tilde{G}_3$ of Worker-2 are smaller than $\tilde{G}_1$, $\tilde{G}_2$, and $\tilde{G}_3$ of Worker-1), Worker-1 and Worker-2 align the number of merged gradients in $Buffer_1^1$ and $Buffer_2^1$ (i.e., $N_1 = N_2 = 3$), which avoids the problem of long gradient synchronization waiting time across all workers in Figure 6.

3.3 Adaptive Intra-Worker Buffer Sizing

Limitation of SAFusion: Multiple intra-worker waiting periods. We also find that SAFusion has performance limitation in the intra-worker training pipeline. Since SAFusion performs sparsification before tensor fusion, so we introduce a new definition *sparse tensor fusion time* (i.e., F_1, F_2, F_3, and F_4 in Figure 9), which refers to the time including the layer-wise gradient sparsification compression time and the tensor fusion time during the backpropagation. Figure 9 shows two observations as follows:

- Greater sparse tensor fusion time than communication time. SAFusion reduces the communication time of gradient tensors but also increases the sparse tensor fusion time of each worker due to the non-negligible compression overhead, which leads to the sparse tensor fusion time being greater than the communication time [10, 29, 57, 58]. Figure 9 shows a general case of intra-worker pipeline for SAFusion, we can see that the sparse tensor fusion time F_2 of $Buffer_2$ is greater than the communication time C_1 of $Buffer_1$, so it is necessary to wait for a period W_1 before starting the communication C_2 of $Buffer_2$.
- Multiple intra-worker waiting periods within each iteration. SAFusion makes the sparse tensor fusion time within an iteration always greater than the communication time, which causes multiple waiting periods for neighboring fusion buffers. As shown in Figure 9, we can see that the sparse tensor fusion time (e.g., F_2, F_3, and F_4) is always greater than the sparsified gradient communication time (e.g., C_1, C_2, and C_3) in multiple fusion phases, resulting in many additional intra-worker waiting periods (e.g., W_1, W_2, and W_3).

Design idea: The reason for the communication waiting periods is that there is *incomplete* overlap between the sparse tensor fusion time and the previous buffer's communication time (e.g., $F_2 > C_1$), which motivates us to adjust the fusion buffer size at different fusion phases for each worker to maximize the overlap between sparse tensor fusion time and the previous communication time. The core idea is to *adjust* the communication time of the previous fusion buffer increase until it begins to only slightly greater than or equal to the sparse tensor fusion time in the current fusion buffer.

Design details: Based on the design idea, we design an intra-worker adaptive buffer sizing scheme with two following steps.

Step-1: Adjusting intra-worker fusion buffers' sizes by making the communication time of the previous buffer begin to be larger than or equal to the sparse tensor fusion time. Specifically, we merge multiple sparsified gradients into the fusion buffer during gradient backward and compute the sparse tensor fusion time and communication time. The i-th fusion buffer size is increasingly adjusted until the following equation begins to be satisfied.

$$T_{cm}^{i-1} \geq T_{cp}^{i} + T_{b}^{i}, 2 \leq i \leq M, \tag{1}$$

where T_{cm}^{i-1} is the communication time of the previous fusion buffer Buffer_{i-1}, $T_{cp}^{i} + T_{b}^{i}$ is the sparse tensor fusion time including the compression time of the Buffer_i, and T_{b}^{i} is the sum of tensor fusion time during the backward computation of all gradients in the Buffer_i.

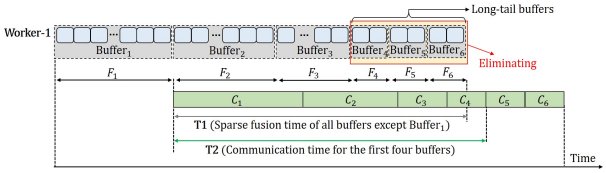
Note that above adjustment does not consider the first fusion buffer (i.e., $i = 1$), so we simply adjust the first buffer size to satisfy that the sum of its fusion time and communication time is less than the layer-wise gradient communication time, such that this adjustment is better than layer-wise gradient communication time. Thus, we have the following equation

$$T_b^1 + T_{cm}^1 < \sum_{j=1}^{N_1} T_{cm}^{l-j}, \tag{2}$$

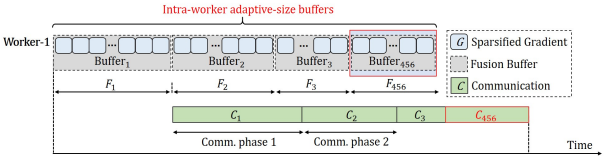
where T_{cm}^{l} is the communication time of the l-th layer gradient, and $\sum_{j=1}^{N_1} T_{cm}^{l-j}$ is the sum of layer-wise gradient communication time [9, 47, 66].

Figure 10 shows the above merging process as follows: **(i)** First, we merge some sparsified gradients to the first fusion buffer $Buffer_1$ base on Equation (2). **(ii)** We calculate the $Buffer_1$'s communication time as C_1, and Buffer_2's sparse tensor fusion time as F_2. We increase Buffer_2 until the sparse tensor fusion time F_2 becomes larger than or equal to C_1. **(iii)** Repeating the above process, we sequentially obtain intra-worker adaptive-size fusion buffers: $Buffer_2$, $Buffer_3$, and $Buffer_4$. Clearly, compared to Figure 9, Figure 10 has no waiting periods.

Step-2: Eliminate long-tailed buffers. Although we generate some adaptive-size fusion buffers in Step-1 to reduce multiple intra-worker communication waiting periods, we find that when the communication time of the buffers is long, the total communication time of the fusion buffers that have previously completed the fusion may exceed the sum of the sparse tensor fusion time of all the gradients. As a result, communication of the subsequent buffers cannot be overlapped with the sparse tensor fusion time, which is



(a) Process of eliminating long-tailed buffers



(b) Results of eliminating long-tailed buffers

Figure 11: Example of SAFusion eliminating long-tailed buffers.

called *long-tailed buffers*. However, the scheme in Step-1 does not eliminate them.

To address this problem, we design a scheme to eliminate long-tailed buffers at the later fusion phases. Specifically, we compute the total communication time of the first M' buffers and the total sparse tensor fusion time of all M buffers except Buffer_1 during the tensor fusion process; when the former is larger than the latter, we merge all the subsequent sparsified gradients into the tail starting from the M'-th buffer. That is, the number of long-tailed buffers is eliminated only when

$$\sum_{i=1}^{M'} T_{cm}^{i} > \sum_{j=2}^{M} (T_{b}^{j} + T_{cp}^{j}), \tag{3}$$

where $M' < M$, we merge the M'-th buffer and subsequent small buffers into a single large buffer to eliminate long-tailed buffers.

For example, as shown in Figure 11(a), the sum of the sparse tensor fusion time of all buffers is less than the sum of their communication time. When the communication C_4 of $Buffer_4$ is finished, the total communication time of the first four buffers (**T2**) is greater than the sparse tensor fusion time of all buffers except $Buffer_1$ (**T1**). This implies that the communication time C_5 and C_6 do not be overlapped with the sparse tensor fusion time of any buffer, so we merge the long-tailed buffers ($Buffer_4$, $Buffer_5$, and $Buffer_6$) into a single $Buffer_{456}$ as shown in Figure 11(b). We further eliminate long-tailed buffers to improve the efficiency of the computation-communication pipeline.

3.4 Discussion

Independent inter- and intra-worker tensor fusion: The inter-worker gradient alignment scheme (§3.2) and the intra-worker adaptive buffer sizing scheme (§3.3) are orthogonal. Indeed, the intra-worker adaptive buffer sizing scheme operates independently within each worker. It adjusts the number of gradients to be merged in each fusion phase based on the size of the previous buffer. In contrast, the inter-worker gradient alignment scheme performs a global alignment across all workers, ensuring that they merge the same number of gradients in each fusion phase. Therefore, the local adaptive adjustment of intra-worker buffer sizing does not interfere with the global alignment of the inter-worker tensor fusion scheme.

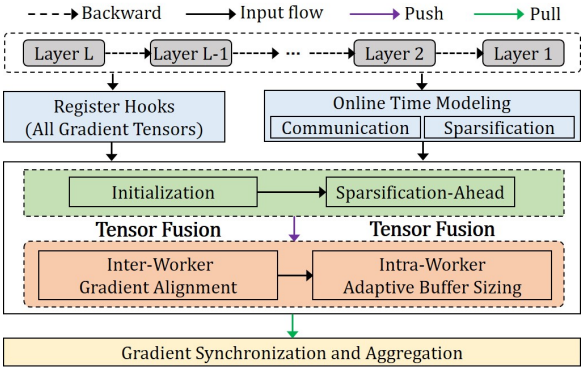


Figure 12: The workflow of SAFusion generator.

Compression overhead: SAFusion enables layer-wise sparsification and merges multiple sparsified gradient tensors into the fusion buffer to significantly reduce the transmission traffic and communication startup time, but as the size of the DNN increases, the compression overhead of SAFusion will also increase slightly (§5.2 shows the results: 4.5% to 6.2%). In the future, we will trade off the compression overhead against the communication overhead, and further reduce the compression overhead by not compressing some small tensors but merging them directly into the fusion buffer.

Training scalability: SAFusion considers both tensor fusion and sparsification. Due to the differences in the stochastic gradients generated by different workers in each iteration, the number of gradient elements aggregated in the communication after sparsification becomes larger than the number actually selected by each worker [12, 63], which has difficulties with large scaling owing to gradient build-up in the fusion buffer. Nevertheless, SAFusion online profiles the efficient tensor fusion scheme based on different experimental setups. Thus, we evaluate the convergence performance and training throughput of SAFusion on different training platforms and different numbers of training GPU nodes in §5.2.

4 IMPLEMENTATIONS

System overview. We leveraged the PyTorch framework and implemented the prototype of SAFusion based on the Horovod [3] using NCCL as the communication library. In our system of SAFusion, each worker contains a `generator` module for generating an efficient sparsification-ahead fusion buffer, a `controller` module for controlling a series of operations such as sparsified gradient pushing, pulling, and communication in the fusion buffer, and a sparsification compression module for performing layer-wise gradient sparsification during the backward pass. The workflow for each worker can be divided into two threads. The `computation_thread` is responsible for forward and backward propagation of the DNN model, executing fusion buffer `generator` and `controller` functions when initializing the register `hook` and backward of the call `hook`, respectively, and interacts with the `communication_thread`. The `communication_thread` receives synchronization tasks from the `computation_thread` and performs the communication operations of each worker based on the NCCL.

Generator module. Figure 12 shows the workflow of the `generator` module, which is responsible for generating the fusion buffers. The

Table 2: Tasks, models and datasets used for evaluation.

Tasks	Models	Model Size	Datasets
Image Classification	ViT-large [15]	1.13 GB	ImageNet [13]
	ResNet-152 [21]	230 MB	ImageNet [13]
	ResNet-152 [21]	230 MB	Cifar-100 [27]
	VGG-19 [54]	548 MB	Cifar-100 [27]
Natural Language Processing	GPT2-large [40]	3.16 GB	WikiText-103 [36]
	LSTM [35]	328 MB	WikiText-2 [36]
	BERT-large [14]	1.32 GB	SQuAD [41]
	BERT-base [14]	420 MB	SQuAD [41]

`generator` first compresses the gradient tensors generated by backward propagation and initializes the queue `group` to represent the fusion buffers. The sparsified gradients are taken as inputs to be merged into the buffer sequentially. `Generator` aligns the number of merged gradient tensors in `group` across different workers. Then, `generator` adaptively adjusts intra-worker buffer size by maximizing the overlap of sparse tensor fusion time with communication time, and further eliminates long-tailed buffers. The `generator` gets the gradient communication time and compression time by online profiling. The communication and compression time are fitted based on the piecewise `interp1d` function.

Controller module. The `controller` controls the push and pull operations of the fusion buffer. It first maintains a list that records the mapping between the gradient tensor `Name` and its corresponding buffer `ID` in the optimal tensor fusion. Upon completion of the gradient calculation for a given gradient tensor, the registered `hook` function is invoked and the gradient is sparsified, then the sparsified gradient is pushed into the corresponding buffer by the mapping list's contents. The sparsified gradients in the buffer are not transmitted until they have all been merged. Following the completion of communication, the `controller` partitions the gradients from the aggregated buffer based on their respective sizes, decompresses them to restore their original shapes, and updates the model parameters accordingly.

Sparsification compression module. We also implement state-of-the-art gradient sparsification compression libraries, including DGC [31], Gaussiank [48], Redsync [17], and SIDCo [34] according to their papers and open source code. Following the literature [29, 37], since the gradient distribution remains almost unchanged over multiple iterations, we periodically estimate the threshold for sparsification compression and reuse it in some subsequent iterations. We also implement a decompress function, which restores the compressed elements into a tensor of the size of the original gradient, then uses the `scatter` function [38] to fill missing elements with zeros, and reshapes the tensor to the original form. To preserve the form of the original gradient for decompression, we add a specific parameter `ctx` to store the original gradient tensor's shape and number of elements.

5 EVALUATION

5.1 Experiment Setup

Testbed. We carried out our experiments on two testbeds: 1) a cloud cluster with 64 GPUs across 16 nodes, connected with 200Gbps InfiniBand. Each node has two 32-core EPYC 7543 CPUs and 4 NVIDIA A100 GPUs (80GB) with NVLink; 2) a local cluster of 8 nodes, each of which is equipped with one NVIDIA V100S GPU (32 GB), two 12-core Intel XEON Silver 4214 CPUs, and is connected by a 100Gbps InfiniBand. Both cloud and local clusters run Ubuntu 20.04, with software libraries including CUDA-12.0, OpenMPI-4.0.3, NCCL-2.8.3, PyTorch-1.13.1, and Horovod-0.28.1.

Workloads. Our evaluation encompasses seven widely-used DNN models, comprising three image classification models and four natural language processing models. We set batch sizes and learning rates across different models similar to the studies [10, 51, 57]. The per-GPU batch size is kept constant as the number of GPUs increases. The details of the models, datasets, and the number of gradients are listed in Table 2.

Comparative approaches. To establish a comprehensive benchmark, we compare SAFusion to WFBP [3] and the state-of-the-art tensor fusion approaches, including OkTopk [29], OMGS [51], and Cupcake [58], which are detailed in §6.2.

Sparsification algorithms. We use four state-of-the-art gradient sparsification compression algorithms with density = 0.01, including DGC[31], Gaussiank[48], RedSync[17], and SIDCo[34]. We also apply error feedback techniques[26, 31] to these sparsification algorithms to ensure the preservation of model accuracy.

SAFusion settings. We also compare the basic version of SAFusion, called SAF(Naive), which performs only sparsification-ahead tensor fusion (in §3.1). We optimize the SAF(Naive), called SAF-Inter, which avoids long synchronization waiting via inter-worker gradient alignment fusion scheme (in §3.2). We further optimize the SAF-Inter, called SAF-(Inter+Intra), which reduces multiple waiting periods via intra-worker adaptive buffer sizing scheme (in §3.3).

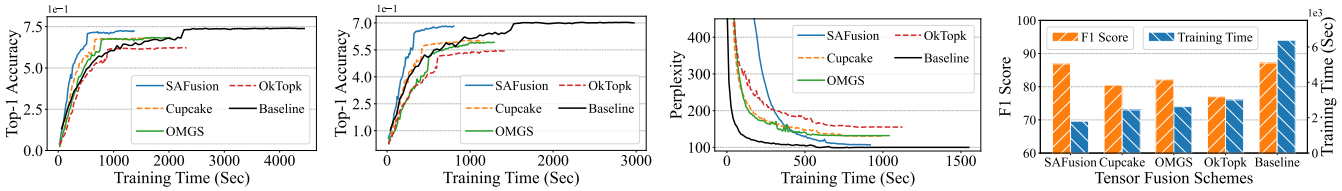
5.2 Experiments

Experiment 1 (End-to-end convergence performance): We evaluate the convergence accuracy of SAFusion and state-of-the-arts using real-world training tasks listed in Table 2 on a local cluster with 8 PCIe-based NVIDIA V100 GPUs connected by a 100Gbps Infiniband. To provide a fair comparison of all methods, we have the same settings for SAFusion and other state-of-the-art schemes.

Figure 13 shows that the convergence performance achieved by SAFusion is very close to the Baseline (non-compression) and higher than the other state-of-the-arts on four different training tasks. Note that we train the same number of epochs and compression rate for different tensor fusion methods on each training task to obtain training time and convergence accuracy. We see that:

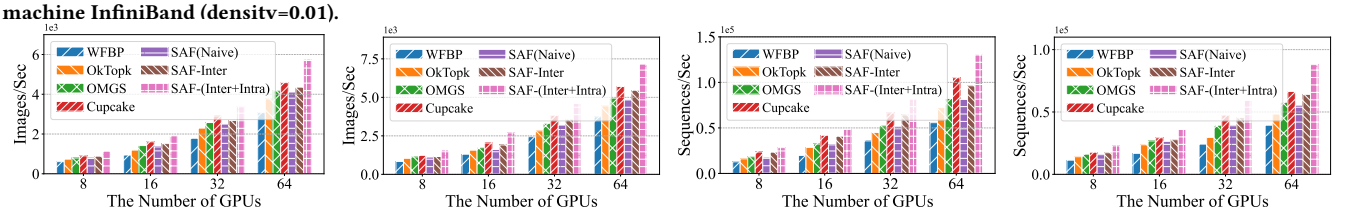
Image classification: Figure 13(a) and (b) for ResNet-152 and VGG-19 on Cifar-100, respectively, show that the Top-1 accuracy of SAFusion achieves 72.32% and 68.28%, which are slightly lower than that of Baseline (i.e., 73.61% and 70.02%), but larger than other state-of-the-art tensor fusion methods. We can see that OkTopk, OMGS, and Cupcake only achieve 62.09%, 68.32%, and 68.08% Top-1 accuracy on ResNet-152, respectively, and 54.26%, 59.18%, and 60.01% Top-1 accuracy on VGG-19.

Natural language processing: Figure 13(c) on WikiText-2 for LSTM shows that SAFusion maintains the same convergence performance as the non-compression Baseline. For example, the Perplexity of SAFusion is 109, which is very close to the Baseline of 104 and lower than OkTopk (155), OMGS (132), and Cupcake (129), which indicates



(a) ResNet-152 on Cifar-100 (b) VGG-19 on Cifar-100 (c) LSTM on WikiText-2 (d) BERT-large on SQuAD

Figure 13: Experiment 1: Training time and convergence accuracy of DNN training tasks with PCIe-only GPU machines and 100Gbps cross-machine InfiniBand (densitv=0.01).



(a) ResNet-152 on ImageNet (b) ViT-large on ImageNet (c) GPT2-large on WikiText-103 (d) BERT-large on SQuAD

Figure 14: Experiment 2: Throughput of DNN training tasks with NVLink-based GPU machines and 200Gbps cross-machine Infiniband.

that SAFusion improves convergence performance by preserving more tensors of the LSTM. Note that the smaller the Perplexity, the higher prediction accuracy of LSTM training Wikitext-2. Figure 13(d) shows that the F1 score (86%) for SAFusion is close to the same as the non-compression Baseline, while the convergence F1 scores of OkTopk, OMGS, and Cupcake are only 76.98%, 82.09%, and 80.5%, respectively.

Analysis: Overall, SAFusion achieves convergence accuracy close to the non-compression Baseline, which confirms our main idea of performing sparsification compression before tensor fusion to avoid missing gradient tensors, thus improving the convergence performance. In addition, we can see that SAFusion also has a shorter training time than other state-of-the-art tensor fusion schemes.

For example, the training time of SAFusion is less than other state-of-the-art tensor fusion schemes by 26.5%-68.8% and 38.9%-84.2% in Figure 13(a) and (b), respectively. SAFusion reduces the training time on LSTM and BERT-large by 9.1%-22.5% and 35.4%-67.1%, respectively. The reason is that SAFusion's inter-worker gradient alignment fusion and intra-worker adaptive buffer sizing scheme improves gradient synchronization and training pipeline efficiency, thus reducing training time.

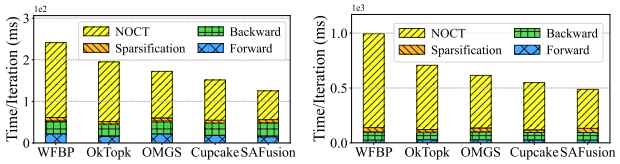
Experiment 2 (Training throughput): We evaluate the training throughput of SAFusion and other state-of-the-art tensor fusion schemes using real-world training tasks listed in Table 2 on a cloud cluster with 64 NVLink-based GPUs connected by 200Gbps cross-machine Infiniband. Figure 14 shows that the training throughput achieved by SAFusion (also called SAF-(Inter+Intra) here) on the four training tasks is much higher than that of the other state-of-the-art schemes. We see that:

Image classification: Figure 14(a) and Figure 14(b) for ResNet-152 and ViT-large on ImageNet, respectively, show that the training throughput of SAFusion outperforms other methods by 18.96%-104.80% and 20.78%-98.43%, respectively, versus different GPU cluster sizes. Unlike the limited training throughput of the latest open-sourced versions of state-of-the-arts, SAFusion improves the training throughput on multiple training tasks. For example, with 64 GPUs, for ResNet-152 on ImageNet, SAFusion outperforms WFBP, OkTopk, OMGS, and Cupcake by 87.41%, 52.89%, 37.43%, and 25.45%, respectively. For ViT-large on ImageNet, SAFusion outperforms WFBP, OkTopk, OMGS, and Cupcake by 91.41%, 60.89%, 44.43%, and 31.45%, respectively.

Natural language processing: For GPT2-large on WikiText-103, Figure 14(c) shows that SAFusion improves training throughput by 20.27%-144.42% versus different GPU cluster sizes. Specifically, SAFusion outperforms WFBP, OkTopk, OMGS, and Cupcake by 94.89%-144.42%, 33.27%-45.66%, 28.20%-58.28%, and 20.27%-30.26%, respectively. Figure 14(d) for BERT-large on SQuAD, shows that SAFusion improves training throughput by 23.29%-128.44% versus different GPU cluster sizes. Specifically, SAFusion outperforms WFBP, OkTopk, OMGS, and Cupcake by 49.71%-128.44%, 30.70%-52.14%, 28.91%-46.21%, and 23.29%-37.22%, respectively.

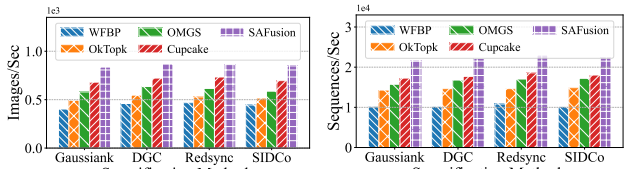
Effectiveness of various optimizations: We evaluate the individual performance gains of the various fusion optimizations introduced for SAFusion, including SAF(Naive), SAF-Inter, and SAF-(Inter+Intra). As shown in Figure 14, SAF(Naive) outperforms the existing WFBP by up to 43.05%, 34.0%, 58.50%, and 60.78% on four training tasks, respectively. Further, we evaluate the case where only inter-worker gradient alignment fusion scheme is used, SAF-Inter (in §3.2). Compared to SAF(Naive), SAF-Inter improves the training throughput by up to 12.39%, 18.21%, 30.78%, and 16.81%, respectively. This is because SAF-Inter avoids the long inter-worker synchronization waiting caused by SAF(Naive). In addition, we also evaluate the intra-worker adaptive buffer sizing scheme, SAF-(Inter+Intra) (in §3.3). As shown in Figure 14, compared to SAF-Inter on four training tasks, SAF-(Inter+Intra) improves the training throughput by up to 34.73%, 46.42%, 35.29%, and 37.68%, respectively. This implies that SAF-(Inter+Intra) further reduces multiple intra-worker waiting periods, thus improving the training efficiency.

Analysis. We can see that WFBP has the lowest training throughput for different GPU cluster sizes. This is because the layer-wise gradient communication of WFBP incurs a large startup overhead,



(a) ResNet-152 on ImageNet (b) BERT-large on SQuAD

Figure 15: Experiment 3: Training time breakdown for different tensor fusion schemes on two training tasks.



(a) ResNet-152 on ImageNet (b) BERT-large on SQuAD

Figure 16: Experiment 4: Training performance comparison using different sparsification compression methods (density=0.01).

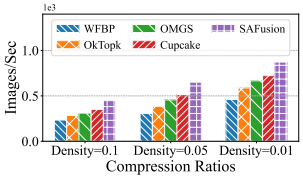


Figure 17: Experiment 5: Training performance comparison using different compression ratios.

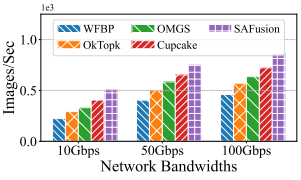


Figure 18: Experiment 6: Training performance comparison using different bandwidths.

which limits the training throughput. OkTopk realizes global compression that merges the gradients of all layers into a single buffer, but waiting on the global gradient prevents opportunities for communication and computation to overlap, which can negatively affect training efficiency.

Although OMGS and Cupcake theoretically derive the optimal solution for tensor fusion, they still face the problem of limited training throughput when scaling to real-world environments with different GPU clusters. On the contrary, SAFusion uses online profiling to generate an effective inter- and intra-worker tensor fusion scheme that scales to clusters with different numbers of GPUs and still achieves higher training throughput than other state-of-the-art tensor fusion schemes.

Experiment 3 (Training time breakdown): To better understand the details of how SAFusion improves the training efficiency, we break down the training time into four parts: forward, sparsification compression, backward, and non-overlapping communication time (NOCT). Note that a smaller NOCT indicates a larger overlap between communication and backpropagation, resulting in a higher training pipeline efficiency. Figure 15 shows the breakdown of training time on ResNet-152 and BERT-large during an iteration. We find that SAFusion reduces the NOCT by 160.67% and 132.89% compared to WFBP, by 107.90% and 64.02% compared to OkTopk, by 54.83% and 34.68% compared to OMGS, and by 36.59% and 20.31% compared to Cupcake. This implies that our SAFusion maximizes the overlap between communication and backpropagation while minimizing communication startup overhead through effective tensor fusion. Surprisingly, we can see that the compression time of SAFusion increases by only 4.5%-6.2% compared to OMGS and Cupcake. This is because SAFusion uses effective threshold-based sparsification algorithm that does not further increase compression overhead.

Experiment 4 (Impact of different sparsification methods): We have completed the performance evaluation on the sparsification compression method DGC in Experiment 1 and Experiment 3 and proved that our SAFusion outperforms the other state-oft-the-arts. Here, we will compare other sparsification compression methods (e.g.,Gaussiank[48], DGC[31], RedSync[17], and SIDCo[34]), and evaluate their performance on various state-of-the-art schemes. Figure 16 shows the training throughput of WFBP, OkTopk, OMGS, Cupcake and SAFusion on ResNet-152 and BERT-large using different sparsification compression methods with the same settings as in Experiment 1 in an 8-nodes local GPU cluster. We can see that the overall training throughput of SAFusion is higher than the other state-of-the-art schemes under all four sparsification methods. Specifically, for ResNet-152 on ImageNet, compared to WFBP, OkTopk, OMGS, and Cupcake, SAFusion achieves throughput improvements of 14.01%-84.56%, 12.22%-77.18%, 9.34%-68.47%, and 10.22%-72.39%, respectively. For BERT-large on SQuAD, SAFusion also increases the training throughput by 10.16%-88.24%, 13.92%-90.37%, 10.75%-89.63%, and 11.01%-90.58%, respectively.

Experiment 5 (Impact of compression ratios): Figure 17 compares the training throughput of WFBP, OkTopk, OMGS, Cupcake and SAFusion on ResNet-152 using different compression ratios (i.e., density=0.1, 0.05, and 0.01) with the same settings as in Experiment 1. For ResNet-152 on ImageNet, compared to other state-of-the-art schemes, SAFusion achieves throughput improvements of 27.70%-78.63%, 26.21%-83.08%, and 14.22%-77.18% when the density is reduced from 0.1 to 0.05 and 0.01, respectively.

We observe that at larger densities, SAFusion improves training throughput more compared to other state-of-the-art schemes. The reason is that transmitting more gradients increases the opportunities for communication and computational overlap. When the density is reduced to a certain level it often makes the network bandwidth not be the bottleneck. Nevertheless, our SAFusion still achieves higher training throughput by its effective inter- and intra-worker tensor fusion scheme. Note that in practice, smaller densities may impair convergence.

Experiment 6 (Impact of network bandwidths): Figure 18 compares the throughput of WFBP, OkTopk, OMGS, Cupcake and SAFusion training ResNet-152 using the same GPU configuration but two different training tasks in an 8-nodes local GPU cluster. We use 10Gbps, 50Gbps, and 100Gbps under the low and high bandwidth networks, respectively. We can see that the training throughput of SAFusion is higher than other state-of-the-art schemes under both low and high-bandwidth networks for both training tasks. Specifically, for ResNet-152 on ImageNet, compared to other four state-of-the-art schemes, SAFusion achieves throughput improvements of 12.18%-85.94%, 9.21%-75.63%, and 11.22%-77.18% when the bandwidth is increased from 10Gbps to 50Gbps and 100Gbps, respectively. Note that SAFusion can effectively increase training throughput even in a low-bandwidth network environment. The reason may be that as the bandwidth decreases, the communication time of the gradient grows, thus increasing the communication-computation overlap.

6 RELATED WORK

6.1 Gradient Sparsification Compression

Gradient sparsification reduces the communication traffic by transmitting a subset of gradient elements and proves to be one of the most effective compression algorithm [45]. Recently, many state-of-the-art sparsification compression methods have been proposed to improve the training performance [7, 18, 31, 37, 43, 45, 48, 52, 55, 60, 62, 69]. DGC [31] samples only 0.1% to 1% of the elements of gradient tensor to estimate the compression threshold and uses it to select elements from the entire gradient, but the threshold is often inaccurate due to the randomness of the sampling. Gaussiank [48] regards the gradient tensor at each iteration as a normal distribution and estimates the threshold by exploiting the percent point function, but the estimated threshold is biased because Gaussiank does not fit an exact normal distribution. Redsync [17] estimates the threshold by moving the ratio between the maximum and mean elements of the gradient, but the threshold estimated in this way is often inaccurate. Since the difference between the gradient tensors of adjacent iterations is usually small, SIDCo [34] is similar to Gaussiank and periodically estimates the threshold by fitting some sparsity-inducing distributions, and selects the gradient elements that are larger than the estimated threshold.

6.2 Tensor Fusion

Some recent works merge multiple gradient tensors into the buffer for a single transmission operation to reduce multiple communication startup overheads and thus improve training efficiency [29, 33, 47, 49, 51, 58, 65, 67, 68]. SyncEA [65] and OkTopk [29] merge all layers' gradient tensors into single buffer to be synchronized by a once all-reduce operation at the end of each iteration. However, the above single buffer tensor fusion scheme often cannot overlap with the backward pass, leading to inefficient pipelining. Thus, state-of-the-arts focus on multiple fusion buffer schemes as shown in Figure 1(b). Horovod [47] merges gradient tensors in intervals of 5ms with merged buffer size upper-bounded by 64MB, as well as Horovod Autotune [5], which automatically tunes the interval and buffer upper bound for better performance. MG-WFBP [49] also pre-defines the size of the buffer by analyzing the minimum training iteration time and performs the communication task when the buffer is full to reduce the startup overhead. OMGS [51] explores the optimal tensor fusion on backward, compression, and communication by formulating an optimization problem for pipelining. Cupcake [58] uses a tensor fusion strategy to minimize the incurred compression overheads. However, these tensor fusion mechanisms occur before sparsification compression (called sparsification-behind tensor fusion), which may lead to tensor missing, and thus compromise convergence performance [16]. In contrast, SAFusion proposes a sparsification-ahead tensor fusion mechanism to address the challenges of sparsification-behind tensor fusion and proposes an efficient (inter-worker and intra-worker) fusion scheme for optimizing communication performance.

7 CONCLUSION

SAFusion is an efficient tensor fusion for high-performance distributed DNN training that addresses the challenges of tensor missing caused by traditional tensor fusion. SAFusion proposes a sparsification-ahead tensor fusion, which performs sparsification on each of the gradient tensors before merging them during tensor fusion, instead of sparsification-behind tensor fusion, so as to avoid gradient tensor missing. Further, SAFusion designs an inter-worker gradient alignment fusion scheme that merges the same amount of sparsified gradients across workers to avoid long gradient synchronization waiting, and an intra-worker adaptive buffer sizing scheme that maximizes the overlap of backpropagation and communication time to reduce multiple waiting periods. For seven popular DNN training models on a local cluster and in a public cloud, SAFusion improves the training throughput by 49.71%-144.42% and 18.96%-60.89% compared to the training without tensor fusion and the state-of-the-art tensor fusion schemes, respectively, while maintaining almost the same convergence accuracy as the non-compression baseline.

We note that SAFusion performs layer-wise sparsification compression before tensor fusion, which can introduce significant compression overhead and increase the overall computation time in distributed training systems. For future work, it is worth exploring efficient gradient sparsification compression strategies for tensor fusion so that one can further improve the training efficiency.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers. This work was supported in part by the National Natural Science Foundation of China (NSFC) No.62272185, Shenzhen Science and Technology Program + JCYJ 20220530161006015, and Key Laboratory of Information Storage System Ministry of Education of China.

REFERENCES

[1] 2024. BytePS. https://github.com/bytedance/byteps.
[2] 2024. DeepSpeed. https://https://github.com/microsoft/DeepSpeed.
[3] 2024. Horovod. https://github.com/horovod/horovod.
[4] 2024. Horovod Analyze Performance. https://horovod.readthedocs.io/en/stable/tensor-fusion_include.html.
[5] 2024. Horovod Automated Performance Tuning. https://horovod.readthedocs.io/en/stable/autotune_include.html.
[6] 2024. NVIDIA NCCL. https://developer.nvidia.com/nccl.
[7] Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021* (2017).
[8] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. In *Advances in Neural Information Processing Systems*. 5977–5987.
[9] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 193–205.
[10] 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. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 359–375.
[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*. 1877–1901.
[12] Chia-Yu Chen, Jiamin Ni, Songtao Lu, Xiaodong Cui, Pin-Yu Chen, Xiao Sun, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Wei Zhang, et al. 2020. ScaleCom: Scalable Sparsified Gradient Compression for Communication-Efficient Distributed Training. In *Advances in Neural Information Processing Systems*. 13551–13563.
[13] 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.
[14] 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).

[15] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[16] Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. 2020. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 3817–3824.

[17] Jiarui Fang, Haohuan Fu, Guangwen Yang, and Cho-Jui Hsieh. 2019. RedSync: reducing synchronization bandwidth for distributed deep learning training system. *J. Parallel and Distrib. Comput.* (2019), 30–39.

[18] 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.

[19] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. *Baidu Research, Tech. Rep.* (2017).

[20] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.

[21] 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.

[22] Horovod. 2024. Tensor Fusion. https://github.com/horovod/horovod. Online accessed on Sept-2023.

[23] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. 2022. dPRO: A Generic Performance Diagnosis and Optimization Toolkit for Expediting Distributed DNN Training. *Proceedings of Machine Learning and Systems* 4 (2022), 623–637.

[24] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.

[25] 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 *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 463–479.

[26] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning*. PMLR, 3252–3261.

[27] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. *Master's thesis, University of Tront* (2009).

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[29] Shigang Li and Torsten Hoefler. 2022. Near-optimal sparse allreduce for distributed deep learning. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–149.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3005–3018.

[31] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *International Conference on Learning Representations*.

[32] Ting Liu, Tianhao Miao, Qinghua Wu, Zhenyu Li, Guangxin He, Jiaoren Wu, Shengzhuo Zhang, Xingwu Yang, Gareth Tyson, and Gaogang Xie. 2022. Modeling and optimizing the scaling performance in distributed deep learning training. In *Proceedings of the ACM Web Conference 2022*. 1764–1773.

[33] Yunzhuo Liu, Bo Jiang, Shizhen Zhao, Tao Lin, Xinbing Wang, and Chenghu Zhou. 2023. Libra: Contention-Aware GPU Thread Allocation for Data Parallel Training in High Speed Networks. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.

[34] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An efficient statistical-based gradient compression technique for distributed training systems. In *Proceedings of Machine Learning and Systems*. 297–322.

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

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

[37] Zhangqiang Ming, Yuchong Hu, Wenxiang Zhou, Xinjue Zheng, Chenxuan Yao, and Dan Feng. 2024. ADTopk: All-Dimension Top-k Compression for High-Performance Data-Parallel DNN Training. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 135–147.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 8026–8037.

[39] 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.

[40] 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.

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

[42] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. 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*. 3505–3506.

[43] 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.

[44] Joshua Romero, Junqi Yin, Nouamane Laanait, Bing Xie, M Todd Young, Sean Treichler, Vitalii Starchenko, Albina Borisevich, Alex Sergeev, and Michael Matheson. 2022. Accelerating collective communication in data parallel training across deep learning frameworks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 1027–1040.

[45] Atal Sahu, Aritra Dutta, Ahmed M Abdelmoniem, and Panos Kalnis. 2021. Rethinking gradient sparsification as total error minimization. In *Advances in Neural Information Processing Systems*. 8133–8146.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. 2021. Scaling distributed machine learning with {In-Network} aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 785–808.

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

[48] Shaohuai Shi, Xiaowen Chu, Cheung, and Simon See. 2019. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772* (2019).

[49] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2019. MG-WFBP: Efficient data communication for distributed synchronous SGD algorithms. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 172–180.

[50] Shaohuai Shi, Xiaowen Chu, and Bo Li. 2021. MG-WFBP: Merging gradients wisely for efficient communication in distributed deep learning. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 1903–1917.

[51] Shaohuai Shi, Qiang Wang, Xiaowen Chu, Bo Li, Yang Qin, Ruihao Liu, and Xinxiao Zhao. 2020. Communication-efficient distributed deep learning with merged gradient sparsification on GPUs. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 406–415.

[52] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen Chu. 2019. A distributed synchronous SGD algorithm with global top-k sparsification for low bandwidth networks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2238–2247.

[53] Shaohuai Shi, Kaiyong Zhao, Qiang Wang, Zhenheng Tang, and Xiaowen Chu. 2019. A convergence analysis of distributed SGD with communication-efficient gradient sparsification. In *IJCAI*. 3411–3417.

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

[55] Zhenheng Tang, Shaohuai Shi, Bo Li, and Xiaowen Chu. 2022. Gossipfl: A decentralized federated learning framework with sparsified and adaptive communication. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2022), 909–922.

[56] 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*. 6000–6010.

[57] 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 *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. 867–882.

[58] Zhuang Wang, Xinyu Crystal Wu, Zhaozhuo Xu, and TS Eugene Ng. 2023. Cupcake: A Compression Scheduler for Scalable Communication-Efficient Distributed Training. In *Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23)*. Proceedings of the Sixth Conference on Machine Learning and Systems (MLSys' 23).

[59] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems*. 1306–1316.

[60] Donglei Wu, Weihao Yang, Cai Deng, Xiangyu Zou, Shiyi Li, and Wen Xia. 2023. BIRD: A Lightweight and Adaptive Compressor for Communication-Efficient Distributed Learning Using Tensor-wise Bi-Random Sampling. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 605–613.

[61] 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 *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*. IEEE, 561–572.

[62] Hang Xu, Kelly Kostopoulou, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. 2021. Deepreduce: A sparse-tensor communication framework for federated deep learning. In *Advances in Neural Information Processing Systems*. 21150–21163.

[63] Daegun Yoon and Sangyoon Oh. 2023. MiCRO: Near-Zero Cost Gradient Sparsification for Scaling and Accelerating Distributed DNN Training. *arXiv preprint arXiv:2310.00967* (2023).

[64] Daegun Yoon and Sangyoon Oh. 2024. Preserving Near-Optimal Gradient Sparsification Cost for Scalable Distributed Deep Learning. *arXiv preprint arXiv:2402.13781* (2024).

[65] Yang You, Aydın Buluç, and James Demmel. 2017. Scaling deep learning on GPU and knights landing clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[66] 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 *USENIX Annual Technical Conference*, Vol. 1. 1–2.

[67] Lin Zhang, Shaohuai Shi, Xiaowen Chu, Wei Wang, Bo Li, and Chengjian Liu. 2023. DeAR: Accelerating Distributed Deep Learning with Fine-Grained All-Reduce Pipelining. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 142–153.

[68] Lin Zhang, Shaohuai Shi, and Bo Li. 2023. Accelerating Distributed K-FAC with Efficient Collective Communication and Scheduling. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.

[69] Zhaorui Zhang and Choli Wang. 2022. MIPD: An adaptive gradient sparsification framework for distributed DNNs training. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2022), 3053–3066.