SSFusion: Tensor Fusion with Selective Sparsification for Efficient Distributed DNN Training

Zhangqiang Ming[1,2], Rui Wang[1], Yuchong Hu[1], Yuanhao Shu[2], Wenxiang Zhou[1], Xinjue Zheng[1], Dan Feng[1]
[1]*Huazhong University of Science and Technology, Wuhan 430074, China*
[2]*Innovation Research Institute of Cethik Group Co., Ltd., Hangzhou 311100, China*
{zqming, ruiwang, yuchonghu, wxzhoucs, zhengxinjue, dfeng}@hust.edu.cn, yuanhaoshu@cethik.com

***Abstract*—Distributed deep neural networks (DNN) training systems deployed across multiple workers have been widely used to accelerate the training of large models and datasets, while the communication overhead for synchronizing data (i.e., gradient tensor) among workers often becomes a performance bottleneck. To improve communication efficiency, two key techniques are often adopted: i) tensor fusion, which merges multiple tensors to transmit them together to reduce the communication startup overhead, and ii) gradient sparsification compression, which transmits only the largest gradient elements to reduce communication traffic. Recent studies focus on combining tensor fusion and sparsification: a straightforward way is to perform sparsification on each tensor before fusion (*per-tensor sparsification*), but it incurs significant sparsification overhead; alternatively, state-of-the-art works first merge multiple tensors (i.e., fusion) and then perform sparsification on them to reduce sparsification overhead (*multi-tensor sparsification*). However, we observe that multi-tensor sparsification causes many tensors to be missing, leading to a significant decrease in convergence accuracy.**

In this paper, we revisit per-tensor sparsification for no tensor missing and propose a new selective sparsification mechanism tailored for fused tensors. This mechanism strategically selects a subset of fused tensors: the selected ones undergo individual sparsification, while the others remain unsparsified. This design achieves both low sparsification overhead (via selective sparsification) and high convergence performance (without tensor missing), which we term SSFusion. On top of SSFusion, we propose an efficient sparsification offloading scheme that offloads GPU-based gradient sparsification to the CPU to further speed up sparsification, and an interleaved communication scheme that improves communication efficiency through fusion separation. Evaluations on cloud and local clusters show that SSFusion improves the training throughput by 27.5%-106.7% over state-of-the-art solutions, while maintaining approximately the same convergence accuracy as the non-sparsification baseline.

***Index Terms*—Distributed DNN Training, Communication Bottleneck, Tensor Fusion, Gradient Sparsification**

I. Introduction

Modern deep neural network (DNN) models have been widely used across various domains [1]–[4], with rapid growth in both model size and training data. For instance, GPT-3 [3] has 175 billion parameters and was trained on roughly 45 TB of text from sources including Wikipedia and books. Due to the large data volumes and complex model architectures, training these tasks on a single GPU is often very time-consuming. To speed up training of large-scale DNNs, mainstream systems typically distribute datasets and models across multiple GPU nodes [5]–[10]. In a typical data-parallel distributed training setup, data samples are partitioned across multiple GPU nodes. Each *worker* (or training node) processes a subset of the input dataset, known as a *mini-batch*. In each iteration, all workers compute gradient *tensors* on their local mini-batches and then exchange them through synchronization mechanisms [8], [11] before updating the model parameters.

Despite their widespread use, modern distributed training systems are hindered by a significant communication bottleneck: frequent and heavy data synchronization across all nodes can consume up to 90% of total training time [1], [2], [6], severely limiting training performance [6], [12]. Many communication optimization techniques have been proposed [5], [6], [8], [12]–[20] to alleviate the communication problem, where there are two key techniques: (i) *tensor fusion* is one of the most effective methods [8], [13]–[18], which combines multiple gradient tensors before gradient synchronization communication to reduce communication startup overhead; (ii) *gradient sparsification compression* is another effective communication optimization technique that selects only a subset of gradient elements with large absolute values rather than the entire gradient for gradient synchronization to reduce communication traffic [12], [19]–[25]. Both of the above two techniques have been widely applied in industry, such as Horovod [5], PyTorch-DDP [7], BytePS [26], and DeepSpeed [27].

Recently, several studies have combined tensor fusion and gradient sparsification techniques (referred to as *tensor fusion with sparsification*) in high-performance distributed DNN training systems [17], [18], [20], [28], [29], leveraging the advantages of both approaches to improve communication efficiency. A simple combination method [5], [7], [30], [31] applies sparsification to each tensor *individually* before fusion, which we refer to as *per-tensor sparsification*, but this method incurs significant sparsification overhead due to every DNN layer (i.e., tensor) needs sparsification, which impairs the training throughput [28], [32].

Alternatively, state-of-the-art combination schemes [17], [20], [28], [33], which perform a *jointly* sparsification on multiple merged tensors created by fusion, are proposed to reduce sparsification overhead which we refer to as *multi-tensor sparsification*. However, we observe that compared to per-tensor sparsification that maintains all tensors after sparsification, multi-tensor sparsification will miss many tensors that are only composed of small gradient elements and thus are not selected by sparsification, which significantly impairs the training convergence performance (see §III-A). Therefore,

it is intuitive to revisit per-tensor sparsification for its high convergence accuracy (without tensor missing), and the goal is to reduce its sparsification overhead.

To this end, we observe two insights that imply the relation between sparsification overhead and small tensors: i) small tensors still have non-negligible sparsification latency; ii) small tensors account for a high proportion in number but only occupy a small proportion of the total size of all tensors (see §III-B). These limitations motivate our proposed method, *tensor fusion with selective sparsification*, which only selects large tensors to perform individual sparsification while leaving small tensors unsparsified during tensor fusion (see §III-C).

In this paper, we propose SSFusion, which implements **s**elective **s**parsification of tensor **fusion** to achieve low sparsification overhead while maintaining high convergence performance. Furthermore, we propose two optimization techniques to further improve the training performance of SSFusion. Our main contributions are as follows:

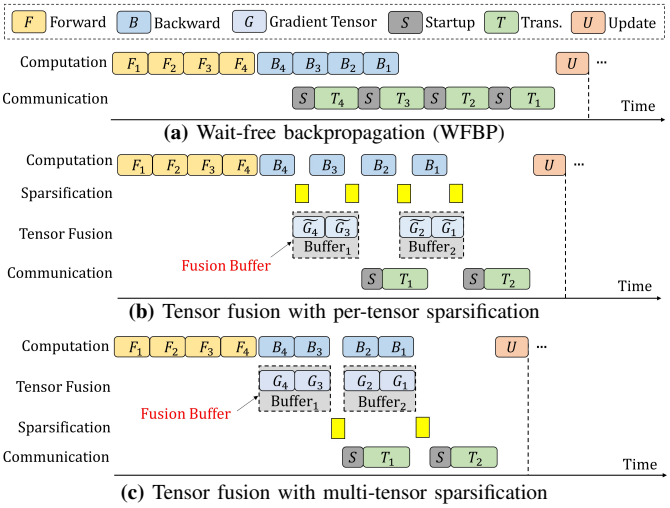
- Through measurement analysis, we first observe the limitation of multi-tensor sparsification methods in terms of degraded convergence performance, then observe two insights that small tensors do not have to be selected for sparsification to reduce sparsification overhead, and are motivated to propose SSFusion, which performs selective sparsification before tensor fusion (§III).
- We design SSFusion with three key components: i) overlap-maximized selection and density-aware sparsification schemes to determine which small tensors to exclude from sparsification for minimal transmission time; ii) an efficient sparsification offloading scheme to further speed up sparsification; iii) an interleaved communication scheme to boost sparse communication efficiency (§IV).
- We conduct experiments on both cloud and local clusters using nine popular DNN models. Results show that, compared with state-of-the-art tensor fusion schemes, SSFusion improves training throughput by 27.5%–106.7% while maintaining convergence performance comparable to the non-sparsification baseline (§V). SSFusion is open-sourced at https://github.com/zqming-cs/SSFusion.

II. Background and Related Work

A. Basics of Distributed DNN Training

A DNN model consists of multiple *layers*, each with numerous *parameters*. To achieve *convergence*, training iterates over the dataset for multiple *epochs*, each containing several *iterations*. In each iteration, the model computes a *gradient tensor* for each layer to update its parameters.

Data-parallel distributed DNN training systems accelerate large-scale DNN training by deploying multiple GPU *workers* [6], [12], [19]. In a typical setup, the training dataset is partitioned into *mini-batches*, which are then assigned to different workers in each iteration. Each worker independently processes a mini-batch from its local data partition, performing *forward* and *backward* propagation to compute gradient tensors, and subsequently *synchronizes* them with other workers via collective communication operations [34], [35].



(a) Wait-free backpropagation (WFBP)

(b) Tensor fusion with per-tensor sparsification

(c) Tensor fusion with multi-tensor sparsification

Fig. 1: Comparison of three distributed DNN training systems with four tensors for gradient synchronization communication.

B. Communication Bottlenecks of Distributed DNN Training

Following prior works [13], [17], [28], we model the communication cost of each gradient tensor as:

$$T = \alpha + \beta D, \tag{1}$$

where α represents the network latency (or startup time) per tensor, β is the transfer time per byte, and D denotes the tensor size. Consequently, the communication bottleneck mainly arises from *frequent communication startups* and *large data transfers* (i.e., the gray and green regions in Fig. 1(a)).

Frequent communication startups. To improve pipeline efficiency, traditional gradient synchronization schemes [5], [36], [37] typically adopt *wait-free backpropagation* (**WFBP**), which overlaps gradient communication with backward computation by exploiting the layer-wise structure of DNN models [17]. As shown in Fig. 1(a), the backward computation times of the four gradient layers B_4–B_1 can be overlapped with both communication startup and transmission times (S and T_4–T_1). However, many modern DNNs consist of numerous small layers, causing frequent communication startups and significant overhead [13], [17], [28], [38], [39].

Huge transmission traffic. Modern DNN models have grown from millions to billions of parameters [2], [3], leading to substantial data transfers between workers. Such communication can consume up to 90% of the total training time [6], [12], [28], [40]–[42], making it a major performance bottleneck in distributed DNN training systems [6], [12].

C. Two Key Communication Optimization Techniques

To address the communication bottlenecks in distributed DNN training systems, two key communication optimization techniques have been proposed.

Tensor fusion. To alleviate the impact of the frequent communication startups, *tensor fusion* has been widely studied as a representative communication scheduling technique [5], [14], [17], [28], [31], [38], [43]. It merges multiple gradient tensors into a single buffer, allowing them to be transmitted

synchronously in a single communication, thereby reducing the startup overhead. In practice, tensor fusion is an effective technique that has been widely adopted in mainstream distributed DNN training systems [5], [7], [9], [26]. As illustrated in Fig. 1(b), pairs of gradient tensors (e.g., G_4 and G_3; G_2 and G_1) are merged into fusion buffers (e.g., $Buffer_1$ for G_4 and G_3, and $Buffer_2$ for G_2 and G_1) after the backward pass, and the tensors in each buffer are then transmitted together (e.g., T_1 for $Buffer_1$ and T_2 for $Buffer_2$). Note that, compared to Fig. 1(a), the startup overhead is lower in Fig. 1(b).

Gradient sparsification compression. To mitigate the substantial transmission traffic in distributed training systems, *sparsification* is a key technique that exploits the sparsity of gradient tensors by selecting only a subset of their elements (i.e., those with the largest absolute values) for communication [20], [22]–[25], [44]–[52], while maintaining nearly the same convergence accuracy [21], [53], [54]. The sparsification-based compression approach has been widely adopted in industry [26], [27], [34]. In this paper, we only focus on gradient sparsification, which is distinct from quantization techniques [55]–[57] that reduce numerical precision.

D. Recent Communication Optimization Schemes by Combining Tensor Fusion with Sparsification

To further enhance communication efficiency, recent studies [6], [17], [18], [20], [28], [32], [33] have focused on combining tensor fusion with gradient sparsification techniques to leverage the benefits of both.

Tensor fusion with per-tensor sparsification. In several typical distributed training frameworks (e.g., Horovod [5], PyTorch-DDP [7], MXNet [30], Accelerating [31], etc.), tensor fusion straightforwardly takes place after sparsification and is orthogonal to sparsification. In this process, each tensor is first sparsified independently and then merged into a fusion buffer for communication, which we call *per-tensor sparsification*.

As illustrated in Fig. 1(b), G_4 and G_3 are first sparsified to $\widetilde{G}_4$ and $\widetilde{G}_3$, and then merged to $Buffer_1$, whose transmission time T_1 and startup time S are less than that in Fig. 1(a). Similarly, G_2 and G_1 are first sparsified to $\widetilde{G}_2$ and $\widetilde{G}_1$, respectively, and then merged to $Buffer_2$, whose T_2 and S are also less than that in Fig. 1(a).

Tensor fusion with multi-tensor sparsification. Per-tensor sparsification incurs significant sparsification overhead since every DNN layer (i.e., tensor) needs sparsification. To reduce sparsification overhead, recent studies [17], [20], [28], [33] adopt *multi-tensor sparsification* strategy, which first applies tensor fusion to merge multiple tensors, and then performs sparsification on the fused tensors.

As illustrated in Fig. 1(c), G_4 and G_3 are first merged into $Buffer_1$ and then are sparsified overall, while G_2 and G_1 are merged into $Buffer_2$ and sparsified overall. Thus, compared with Fig. 1(b), the overall iteration time in Fig. 1(c) is further reduced, owing to the decrease in sparsification time.

E. State-of-the-arts and Their Limitations

We review six state-of-the-art training optimization methods and their limitations.

HiPress [6]: HiPress improves tensor communication efficiency by bulk synchronization and selects tensors for sparsification based on sparsification benefits. Specifically, its selection relies solely on a theoretical cost model that estimates the communication time and sparsification time of each tensor.

Espresso [12]: Espresso selects near-optimal sparsification strategies and offloads sparsification from the GPU to the CPU to mitigate contention. Specifically, it uses a decision tree to analyze interactions among tensors and make sparsification and offloading decisions for each tensor, aiming to maximize training throughput.

OMGS [17]: OMGS is a typical tensor fusion scheme with multi-tensor sparsification, which explores the optimal tensor fusion for backward computation, sparsification, and communication by formulating an optimization problem to maximize the overlap between communication and computation.

OkTopk [20]: OkTopk first merges all gradient tensors into a single buffer for threshold-based sparsification and then uses a sparse all-reduce algorithm for synchronization. However, its single-buffer tensor fusion scheme often fails to overlap with backward computation, resulting in inefficient pipelining.

Cupcake [28]: Cupcake fuses multiple tensors for one sparsification operation to reduce sparsification overhead, and determines the provable fusion strategy by balancing the sparsification overhead and the communication overhead to improve the training throughput.

SparDL [33]: SparDL combines the reduce-scatter phase with multiple block-wise sparsification processes in the fusion buffer to address the sparse gradient accumulation dilemma, but it does not consider the overlap efficiency between computations and communications.

Limitations. Despite various optimizations, all the above methods still have limitations: some incur significant sparsification overhead due to per-tensor sparsification, others suffer from convergence degradation caused by multi-tensor sparsification (see §III-A), and still others exhibit inefficient pipelining between computation and communication.

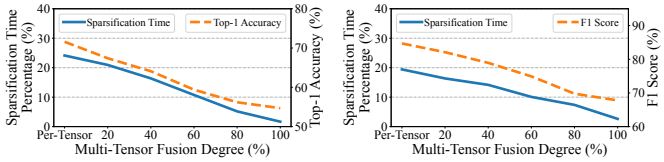
III. Observations and Motivation

We first conduct measurement analysis to provide observations illustrating the limitation of multi-tensor sparsification (§III-A) and two insights that help reduce the sparsification overhead for per-tensor sparsification (§III-B). Finally, we leverage the insights to motivate our main idea (§III-C).

Measurement setting. We evaluate three representative training tasks (detailed in §V-A) on a local cluster: ResNet-152 [58] on Cifar-100 [59], BERT-large [2] on SQuAD [60], and GPT2-large [4] on WikiText-2 [61]. We adopt DGC [44] as the sparsification method, setting the density to 0.01.

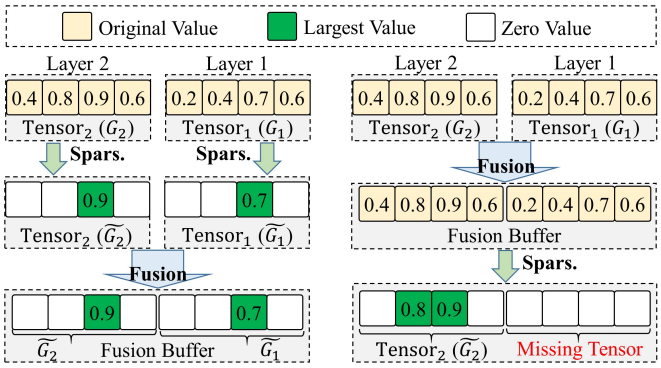
A. Limitation of Multi-Tensor Sparsification

Observation: compared to per-tensor sparsification, multi-tensor sparsification significantly trades off convergence performance for reduced sparsification overhead. We present our first observation based on the measurement setup. To characterize varying extents of multi-tensor sparsification, we



(a) ResNet-152 on Cifar-100 **(b)** BERT-large on SQuAD

Fig. 2: The sparsification time percentage and convergence accuracy of two training tasks under different multi-tensor fusion degrees.



(a) Per-tensor sparsification **(b)** Multi-tensor sparsification

Fig. 3: Example of the tensor fusion with per-tensor and multi-tensor sparsification, density = 25%.

introduce a new metric, the *multi-tensor fusion degree*, where $fusion\ degree = 100\%$ indicates that all tensors are fused into a single buffer, $fusion\ degree = 20\%$ means that only 20% of the tensors are fused. Fig. 2 shows the sparsification time percentage and convergence performance of per-tensor and multi-tensor sparsification on two training tasks.

Specifically, for ResNet-152 on Cifar-100, Fig. 2(a) shows that per-tensor sparsification's sparsification time accounts for 24.1% of total training time, with a Top-1 accuracy of 71.6%. In contrast, multi-tensor sparsification's sparsification time percentage decreases from 20.9% to 1.7% as fusion degree increases from 20% to 100%, while Top-1 accuracy drops correspondingly from 67.4% to 54.7%.

Similarly, for BERT-large, Fig. 2(b) demonstrates that per-tensor sparsification has a sparsification time percentage of 19.4% with an F1 score of 84.7%. Multi-tensor sparsification, however, reduces this percentage to as low as 2.6% (at 100% fusion degree), but its F1 score drops significantly to 66.8%.

Thus, tensor fusion with per-tensor sparsification trades sparsification overhead for convergence accuracy, whereas multi-tensor sparsification does the opposite.

Analysis: multi-tensor sparsification's convergence performance degradation is caused by tensor missing, while per-tensor sparsification always maintains all tensors. We find that the reason for the decreased convergence performance of multi-tensor sparsification is *tensor missing*, in which complete tensors are randomly excluded during sparsification. Fig. 3 shows the process of tensor fusion with per-tensor and multi-tensor sparsification. The DNN model includes two-layer tensors (G_2 and G_1) with a total of eight elements; each tensor has four elements (for simplicity, we consider only the absolute values of all gradient tensor elements). Specifically,

- Per-tensor sparsification: Fig. 3(a) shows that G_2 and G_1 are first independently sparsified with density=0.25. Then, the sparsified $\widetilde{G}_2$ and $\widetilde{G}_1$ are merged into the fusion buffer.
- Multi-tensor sparsification: Fig. 3(b) shows that G_2 and G_1 are first merged, and sparsification is then performed on the tensors of the entire fusion buffer.

We observe in Fig. 3 that the elements of G_2 have larger magnitudes (i.e., absolute values) than those of G_1, which is common in real systems where different tensors have varying distributions [13], [50], [62]. In this case, Fig. 3(a) shows that under per-tensor sparsification, each tensor retains at least one element for gradient synchronization; in contrast, Fig. 3(b) shows that under multi-tensor sparsification, G_1 is not selected at all, which we call *tensor missing*.

Tensor missing prevents certain layer gradients from being synchronized across nodes, causing parameters to miss part of the global updates and thus degrading convergence. A recent study [63] also observes that the sparsification with all layer tensors fused yields lower convergence accuracy than the one without fusion between layer tensors, because, as stated in [63], the former fails to transmit certain tensors. This observation in [63] is consistent with our analysis, confirming that the convergence degradation is due to missing tensors.

To validate the finding in Fig. 3, Table I shows the *tensor missing rate* under per-tensor and multi-tensor sparsification on two training models, defined as the ratio of the number of tensors missed due to fusion to the total number of gradient tensors. As shown, both models exhibit a tensor missing rate of 0 under per-tensor sparsification. In contrast, for multi-tensor sparsification, the tensor missing rates range from 9.8% to 75.6% and 8.6% to 81.2% across the two models as the fusion degree increases from 20% to 100%.

Due to multi-tensor sparsification's significantly reduced high accuracy, it is natural to *revisit per-tensor sparsification for its high accuracy.* Thus, our main goal is to directly reduce the sparsification overhead for per-tensor sparsification.

TABLE I: The tensor missing rate under tensor fusion with per-tensor and multi-tensor sparsification for two modern DNN models.

Modern DNN Model	Per-Tensor	Multi-Tensor Fusion Degree(%)				
		20	40	60	80	100
ResNet-152 [58]	0	9.8%	22.8%	47.0%	68.2%	75.6%
BERT-large [64]	0	8.6%	24.0%	42.5%	66.5%	81.2%

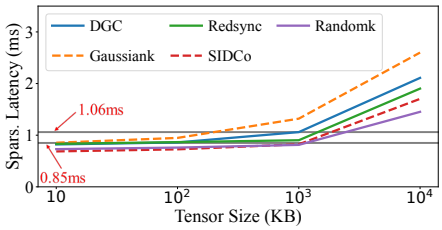


Fig. 4: The comparison of sparsification latencies across different sparsification algorithms.

B. Insights

To achieve the main goal (sparsification efficiency) stated above in §III-A, we investigate how the tensor size impacts the sparsification overhead and how the tensors of different sizes are distributed. We observe the following two insights:

TABLE II: The number and size proportion of small and large tensors in modern DNN models.

Modern DNN Model	Small Tensors ($\leq$ 200 KB)		Large Tensors ($>$ 200 KB)	
	Number(%)	Total Size(%)	Number(%)	Total Size(%)
ResNet-152	72.38	2.37	27.62	97.63
ViT-large	62.76	0.11	37.24	99.89
BERT-large	62.66	0.10	37.34	99.90
GPT2-large	66.51	0.08	33.49	99.92

Insight #1: Small tensors still have non-negligible sparsification latency relative to large tensors. To study the sparsification overhead impacted by different tensor sizes, we give an observation in Fig. 4, which shows the sparsification latencies with five representative sparsification algorithms (DGC [44], Gaussiank [22], Redsync [23], SIDCo [24], and Randomk [65]). We find that sparsification latency does not scale linearly with tensor size; for small tensors, it instead approaches a roughly constant value. For example, under the DGC algorithm, tensors smaller than 100 KB have a latency of about 0.85 ms, while those between 100 KB and 1 MB incur latencies ranging from 0.85 ms to 1.06 ms. Such latencies make sparsification relatively expensive for small tensors.

Insight #2: Small tensors account for a high proportion in number, but only occupy a small proportion of the total size of all tensors. Table II shows the number and size proportion of small and large tensors. We define a gradient tensor with fewer than 102,400 elements as a small tensor (approximately 200 KB); otherwise, it is a large tensor. This threshold is a heuristic to motivate our idea; the actual cutoffs are determined via online profiling in our design (§IV-B). We can see that small tensors account for 72.38%, 62.76%, 62.66%, and 66.51% of the total number of tensors in the four models, but only 2.37%, 0.11%, 0.10%, and 0.08% of their total size.

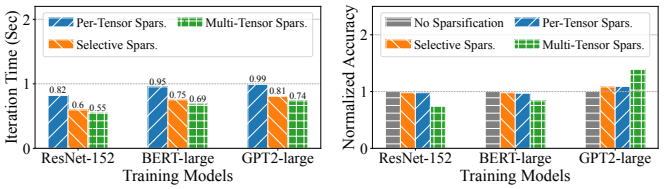
The above two insights imply that *not selecting small tensors that have a high number proportion and non-negligible sparsification latency for sparsification can significantly reduce sparsification overhead, while leading to little impact on data transmission due to the small total size of small tensors.*

C. Motivation

As stated in §III-A, the limitation of multi-tensor sparsification makes us revisit per-tensor sparsification for no tensor missing; as stated in §III-B, per-tensor sparsification's high sparsification overhead can be reduced by not sparsifying some small tensors. This motivates us to only select large tensors to perform individual sparsification while leaving small tensors unsparsified, which is called *tensor fusion with selective sparsification*, so as to reduce sparsification overhead while avoiding tensor missing for high accuracy.

Main idea: Therefore, we propose SSFusion, a selective sparsification tensor **fusion** mechanism, whose main idea is to selectively sparsify certain large tensors while excluding small tensors from individual sparsification during tensor fusion, instead of the existing mechanism that sparsifies multiple tensors together (multi-tensor sparsification).

To verify the effectiveness of the main idea, Fig. 5 compares per-tensor, multi-tensor, and selective sparsification (where only tensors larger than 200 KB are selected for sparsification as in Insight #2) in terms of iteration time and convergence accuracy. The non-sparsification baseline is used as the normalized convergence accuracy (i.e., set to 1).



(a) Iteration training time **(b)** Normalized convergence accuracy

Fig. 5: Iteration training time and normalized convergence accuracy of tensor fusion with different sparsification schemes.

We can see that selective sparsification outperforms per-tensor sparsification in iteration time and multi-tensor sparsification in convergence accuracy. As shown in Fig. 5, selective sparsification achieves iteration times of 0.6s, 0.75s, and 0.81s for the three models, which are close to those of multi-tensor sparsification and lower than those of per-tensor sparsification (0.82s, 0.95s, and 0.99s). In terms of convergence, selective sparsification improves accuracy over multi-tensor sparsification by 32.4%, 19.3%, and 28.5%, while remaining comparable to both the baseline and per-tensor sparsification. For GPT2-large on WikiText-2, lower perplexity indicates higher accuracy.

Analysis. While existing approaches [6], [12] also employ selective sparsification to reduce sparsification overhead, SSFusion fundamentally differs in its selection objective. Specifically, (i) HiPress [6] relies on a *static, theoretical* cost model of communication and sparsification times to estimate the benefit for each tensor, without considering overall *system-level* performance optimization; (ii) Espresso [12] analyzes *tensor-level* interactions to maximize training throughput using a decision tree and offloads sparsification to the CPU to mitigate GPU contention, but it does not support tensor fusion, which incurs significant communication startup overhead; and (iii) To our knowledge, SSFusion is the first to systematically select tensors for optimizing *fusion-level* pipelining performance at the system level. Despite SSFusion 's promise, a more detailed elaboration of its fusion-level pipelining design remains unexplored and will be presented in §IV.

IV. SSFusion Design

In this section, we design SSFusion based on the main idea in §III-C with the design goals as follows:

- **Optimizing selective sparsification:** SSFusion optimizes its selective sparsification under tensor fusion via maximizing the overlap between the computation and communication time as well as adjusting sparsification density (§IV-B).
- **Accelerating sparsification process:** SSFusion extends to SSFusion-O that employs an efficient sparsification offloading scheme to speed up the sparsification (§IV-C).
- **Improving communication efficiency:** SSFusion also extends to SSFusion-I that employs an interleaved communication scheme to combine All-Reduce and All-Gather to improve sparse communication efficiency (§IV-D).

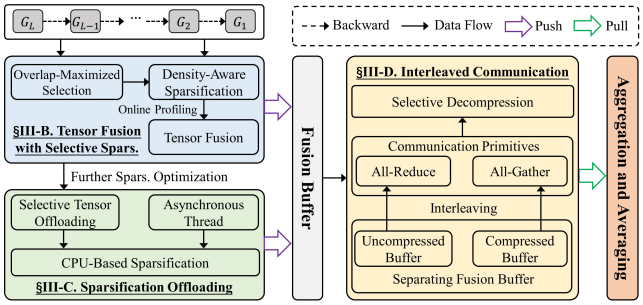


Fig. 6: Overview of SSFusion framework.

A. System Overview

We outline the system design as follows. As shown in Fig. 6, the framework comprises three major modules: a tensor fusion and selective sparsification module (i.e., basic SSFusion in §IV-B), a sparsification offloading module (i.e., SSFusion-O in §IV-C), and an interleaved communication module (i.e., SSFusion-I in §IV-D). Based on our framework, users can efficiently sparsify and synchronize gradients. First, selected gradient tensors are sparsified and merged into a fusion buffer during the backward pass. Second, the sparsification of large tensors in the fusion buffer is selectively offloaded to the CPU to overlap with backward computation. Third, the fusion buffer is separated into an unsparsified one and a sparsified one to enable interleaved All-Reduce and All-Gather communication.

Various time estimation. To specify the above design modules, we describe how various time costs of the design processes are estimated as follows.

Computation time T_b^l: T_b^l is the backward computation time of l-th layer tensor. We collect execution traces of DNN training for 100 iterations without tensor fusion or sparsification to capture each tensor's computation start and end times, and then average the computation time across iterations.

Communication time T_{cm}^l: T_{cm}^l is the communication time of l-th layer tensor. As stated in §II-B, we can estimate the communication time via Equation (1) based on the startup overhead (α) and per-byte transfer cost (β). We measure α and β offline based on the system configurations, such as tensor sizes, communication schemes, the number of GPUs, and network bandwidth.

Sparsification time T_{gpu}: T_{gpu}^l is the GPU-based sparsification time of l-th layer tensor. We estimate the sparsification time for tensors of various sizes on different compute resources (CPU or GPU) by profiling all possible tensor sizes from model training traces. Each operation is repeated multiple times, and the averaged results are used in the cost model.

B. SSFusion: Tensor Fusion with Selective Sparsification

Despite the promise of selective sparsification, determining which tensors to sparsify individually under tensor fusion for optimal training efficiency remains challenging. To address this gap, we first analyze the overlap between computation and communication in tensor fusion with per-tensor sparsification.

Analysis of computation-communication overlap in tensor fusion with per-tensor sparsification. As described in §III-C, we perform per-tensor sparsification to avoid tensor missing, with each tensor being sparsified individually before fusion. However, frequent sparsification significantly increases the computation time of each fusion buffer, which results in incomplete overlap between the current buffer's computation and the previous buffer's communication.

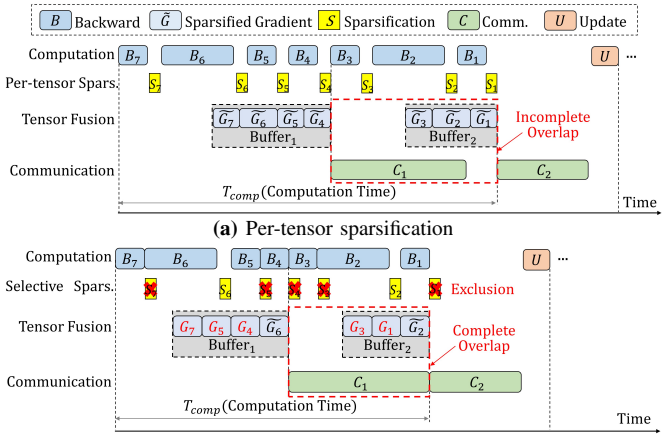


Fig. 7: The process of (a) per-tensor sparsification and (b) selective sparsification for SSFusion.

As shown in Fig. 7(a), tensor fusion with per-tensor sparsification increases the computation time T_{comp} in both $Buffer_1$ and $Buffer_2$, resulting in incomplete overlap with communication and thereby delaying training.

Design idea. Therefore, we think a good selective sparsification should aim to maximize the overlap between the computation during the current fusion buffer and the communication of the previous one, to achieve optimal training performance.

This motivates selective sparsification to exclude some small tensors from sparsification in the fusion buffer until the previous buffer's communication time slightly exceeds or matches the current buffer's computation time.

Overlap-maximized selection. Based on the design idea, we design an overlap-maximized selective sparsification scheme with the following two steps:

Step-1: Initialization. We first fix the size of the i-th fusion buffer (denoted by N_i, $1 \leq i \leq M$) and the number of fusion buffers (denoted by M) according to `FindOptFusion` function in a state-of-the-art work Cupcake [28], which is proved to ensure that training time is minimized. We then flatten the l-th layer's tensor G_l ($1 \leq l \leq L$) into a vector and apply a sparsification algorithm [44] to select elements whose absolute values exceed a given threshold (i.e., $density = 0.01$), where L is the number of all layer gradients of the DNN model. Next, we merge the sparsified gradient $\widetilde{G}_l$ into $Buffer_i$. When $Buffer_i$ is full (i.e.,it has N_i tensors), we start the next merging of $Buffer_{i+1}$.

Step-2: Optimization. To optimize selective sparsification, we start from the smallest gradient tensors, and progressively exclude sparsification of small gradient tensors in the fusion buffer until the overlap between the current buffer's computation time and the previous buffer's communication time is maximized.

Specifically, for N_i tensors in the i-th buffer, the process involves three steps: (i) we first sort these tensors by ascending size; (ii) we calculate this buffer's computation time (sum of backward and sparsification time) and the previous buffer's communication time; (iii) we progressively exclude the sparsification of the l'-th tensor ($1 \leq l' \leq N_i$) in the i-th buffer until the following equation becomes satisfied,

$$T_{cm}^{i-1} \geq \sum_{j=1}^{N_i} \left(T_b^j + T_{gpu}^j\right), 2 \leq i \leq M, \tag{2}$$

where T_{cm}^{i-1} denotes the communication time of the previous fusion buffer, $\sum_{j=1}^{N_i} \left(T_b^j + T_{gpu}^j\right)$ is the computation time including the backward computation time (T_b^j) and the GPU-based sparsification time (T_{gpu}^j) of all tensors in the $Buffer_i$. The above process achieves maximum overlap because if we proceed to the next tensor sparsification, the computation time will decrease, while the computation time remains unchanged, resulting in overall decreased overlap.

Fig. 7(b) illustrates the selective sparsification process in the fusion buffer. We (i) sort the tensors in each fusion buffer from smallest to largest, for example, the tensors in $Buffer_2$ are sorted as $[G_3, G_1, G_2]$ and the tensors in $Buffer_1$ are sorted as $[G_7, G_5, G_4, G_6]$, (ii) perform selective sparsification from the last buffer to the first one by calculating the $Buffer_2$'s computation time T_{comp}^2, and $Buffer_1$'s communication time T_{cm}^1, and (iii) gradually exclude the sparsification of small tensors G_3 and G_1 in $Buffer_2$ until T_{cm}^1 is greater than or equal to T_{comp}^2 via Equation (2).

Density-aware sparsification. Although excluding small tensors from sparsification in the fusion buffer can reduce sparsification overhead, this choice still prolongs transmission time. To address this, we deliberately decrease the sparsification density of large tensors to compensate for the negative impact of non-sparsifying small tensors. To this end, we propose a density-aware sparsification method that dynamically adjusts the sparsification density of large tensors to ensure that the compensation works.

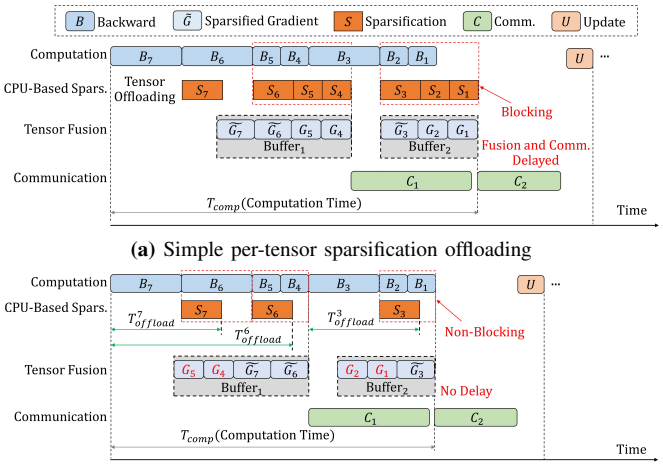
Specifically, the process involves four steps: (i) it first computes the number of gradient elements; (ii) it then computes the number of elements selected via selective sparsification; (iii) it calculates the difference between the above two numbers (i.e., the negative impact of non-sparsifying small tensors); (iv) it adjusts the sparsification density of large tensors until the difference approaches zero.

To avoid over-sparsification [6], we set the lower bound of the density to 0.001, which is confirmed to ensure high convergence accuracy [6], [44]. This bound ensures that no tensor is overly sparsified, thereby preserving convergence and making the method robust across different models.

Online profiling. We perform online profiling during the first five warm-up iterations to identify tensors for sparsification by sorting all tensors in terms of their size. We progressively exclude small tensors, and then record their names in the configuration. In subsequent iterations, tensor selection directly determines the large tensors for the sparsification via the configuration without sorting, incurring negligible overhead in the total training time (see Experiment 8 in §V-B).

TABLE III: CPU-based and GPU-based sparsification times with the DGC [44] sparsification algorithm for different tensor sizes.

Methods	Sparsification Time (ms)			
	10 KB	100 KB	1000 KB	10,000 KB
GPU-based Spars.	0.85	0.89	1.06	1.82
CPU-based Spars.	1.49	1.81	3.22	8.74



(a) Simple per-tensor sparsification offloading

(b) Sparsification offloading in SSFusion

Fig. 8: Comparison of two sparsification offloading schemes, with sparsification of G_5, G_4, G_2, and G_1 excluded in (b).

C. SSFusion-O: *Sparsification Offloading in* SSFusion

Issue of simple per-tensor sparsification offloading. While GPU-based sparsification is generally faster than CPU-based sparsification [12], the former competes with training for GPU resources [66], whereas the latter does not. To mitigate this contention, recent studies have offloaded sparsification to CPUs [12]. However, we observe that CPU-based sparsification is much slower, with the gap widening as the tensor size increases. As Table III shows, when tensor sizes range from 10 KB to 10,000 KB, CPU sparsification time increases by 75.3% to 380.2% compared to GPU sparsification.

However, simply offloading each tensor's sparsification to the CPU during tensor fusion can block the sparsification of subsequent tensors, delaying their fusion and thus reducing overall training efficiency. As illustrated in Fig. 8(a), CPU-based sparsification of G_6 blocks that of G_5 and G_4, which delays the fusion and communication of $Buffer_1$. Similarly, the sparsification of the G_3 also blocks that of G_2 and G_1, delaying the fusion and communication of $Buffer_2$.

Insight into sparsification offloading benefit from SSFusion. Fortunately, SSFusion provides an opportunity to offload sparsification from the GPU to the CPU. This is because SSFusion selectively skips the sparsification of some small tensors, thereby reducing contention among offloading operations for multiple adjacent tensors.

Design details. Guided by the above insight, we design SSFusion-O, an efficient sparsification offloading scheme for SSFusion, which selectively offloads the sparsification of large tensors to the CPU without compromising tensor fusion

efficiency. Specifically, SSFusion-O realizes this objective through three key steps:

Step-1: Selecting tensors for sparsification offloading. First, we only select those tensors that can produce overlaps between CPU sparsification and GPU backpropagation after their sparsification offloading, as these overlaps indicate the beneficial potential for pipelining. Specifically, we denote the time interval from the start of backpropagation of the first tensor in the i-th fusion buffer to the completion of $G_{l'}$'s CPU-based sparsification by $T^{l'}_{offload}$ $(1 \leq l' < N_i)$, and ensure that $T^{l'}_{offload}$ does not exceed the total computation time of all N_i tensors in the buffer, i.e., $\sum_{j=1}^{N_i} T_b^j$, where T_b^j denotes the backward computation time of the j-th tensor (§IV-A).

Step-2: Copying gradient tensor from GPU to CPU memory. To enable CPU-based sparsification, we asynchronously copy tensors from GPU to CPU memory without disrupting the training process. Specifically, we first pre-allocate host CPU memory to hold the tensor. The gradient tensor is then copied to CPU memory in a pipelined manner via CUDA streams, using PyTorch's `copy_()` function with the `non_blocking` flag enabled to avoid interfering with training.

Step 3: Performing CPU-based asynchronous sparsification. To prevent resource contention between sparsification and backpropagation, CPU-based sparsification runs on a separate asynchronous thread, independent of GPU backpropagation. This decoupling avoids training stalls caused by sparsification. Once sparsification completes, the sparsified tensor is asynchronously copied back to GPU memory.

Fig. 8(b) illustrates the sparsification offloading process of SSFusion-O: First, we sequentially select large tensors to be sparsified in each fusion buffer, e.g., G_3 in $Buffer_2$, G_7 and G_6 in $Buffer_1$. We then compute their CPU sparsification timestamps $T^3_{offload}$, $T^6_{offload}$ and $T^7_{offload}$. If $T^3_{offload}$ precedes $Buffer_2$'s computation completion, G_3 can be copied to CPU memory for asynchronous sparsification without delaying training. The same procedure is applied to G_6 and G_7.

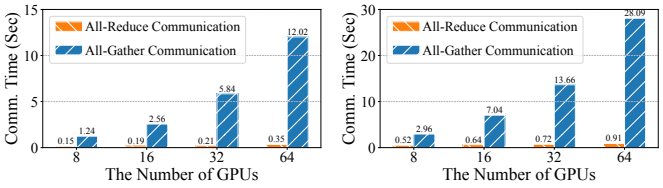
D. SSFusion-I*: Interleaved Communication in* SSFusion

Issue of common communication primitive for sparsification. Existing state-of-the-art sparsification schemes [17], [20], [28], [42] typically rely solely on the All-Gather communication primitive to synchronize sparsified tensors. However, we observe that SSFusion, similar to these schemes, incurs substantial All-Gather communication overhead, which grows markedly with the number of training nodes.

As illustrated in Fig. 9, we measure the per-iteration communication time for synchronizing tensors of identical size in two training models across multiple GPU clusters. The results show that All-Gather communication time increases linearly with the number of GPUs, rising from 1.24s to 12.02s and from 2.96s to 28.09s as the GPU count grows from 8 to 64 in the two models.

We analyze the communication complexity of the All-Gather primitive [5], [67], denoted as T_{AG}, which is given by:

$$T_{AG} = (P-1)\alpha + (P-1)D\beta, \tag{3}$$



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

Fig. 9: Comparison of communication time of each iteration using All-Gather and All-Reduce.

where P is the number of training workers, α is the network latency per tensor, β is the transfer time per byte, and D is the tensor size, assuming a ring-based All-Gather. The formula shows that All-Gather overhead grows linearly with P. Thus, relying exclusively on All-Gather to synchronize tensors in the fusion buffer would impose considerable communication costs.

Insight into communication benefit from SSFusion. Another typical communication primitive is All-Reduce, whose communication cost is significantly lower than that of All-Gather. As shown in Fig. 9, when the number of GPUs scales from 8 to 64, the All-Reduce communication time increases only marginally—from 0.15s to 0.35s and 0.52s to 0.91s for the two training models, respectively.

The communication complexity of the All-Reduce primitive, denoted as T_{AR}, is given by:

$$T_{AR} = 2(P-1)\alpha + 2\frac{P-1}{P}D\beta, \tag{4}$$

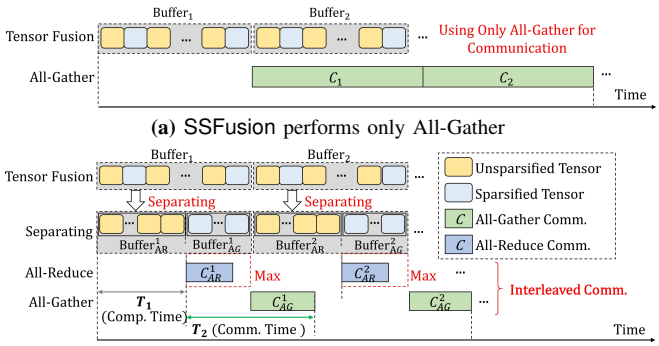
where All-Reduce only increases the communication startup overhead. Similarly, the analysis assumes the ring-based All-Reduce algorithm. Unfortunately, the underlying implementation of All-Reduce does not support sparse gradient tensors and imposes constraints that input tensors must have identical data types and shapes [34], [42], [68].

Interestingly, SSFusion's selective sparsification mechanism enables the fusion buffer to accommodate both sparsified (sparse) and unsparsified (dense) tensors concurrently. Leveraging this capability, sparsified tensors can be processed via All-Gather, while unsparsified tensors can be handled using All-Reduce, thus enhancing communication efficiency.

Design idea. Building on these insights, SSFusion introduces an interleaved communication framework, SSFusion-I. It first separates each fusion buffer into an unsparsified buffer and a sparsified buffer according to selective sparsification (§IV-B) and then interleaves All-Reduce for dense tensors with All-Gather for sparse tensors, exploiting their complementary communication efficiencies.

Separating fusion buffers. To achieve optimal interleaved communication, we separate each fusion buffer into an unsparsified buffer ($Buffer_{AR}$) and a sparsified buffer ($Buffer_{AG}$) by minimizing the sum of computation time (T_1) and communication time (T_2) for each buffer. Specifically,

Step-1: Assigning each tensor to $Buffer_{AR}$ or $Buffer_{AG}$. When merging the l'-th tensor $G_{l'}$ $(1 \leq l' \leq N_i)$ into $Buffer_i$ $(1 \leq i \leq M)$, we first check whether it is sparsified. If it is unsparsified, we assign it to $Buffer^i_{AR}$; otherwise, it is assigned to $Buffer^i_{AG}$.



(a) SSFusion performs only All-Gather

(b) SSFusion performs interleaved All-Reduce and All-Gather

Fig. 10: Example of SSFusion's training timeline using only All-Gather vs. interleaved All-Reduce and All-Gather.

Step-2: Calculating T_1 and T_2. Every time assigning the tensor $G_{l'}$, we calculate T_1 and T_2 of the buffer the tensor resides.

- Computation time (T_1): T_1 is defined as the sum of backward time (T_b^j) and the GPU-based sparsification time (T_{gpu}^j) for all tensors before $G_{l'}$ in $Buffer_i$, i.e., $T_1 = \sum_{j=1}^{l'} \left(T_b^j + T_{gpu}^j\right)$.
- Communication time (T_2): T_2 consists of the All-Reduce time T_{AR}^i for $Buffer_{AR}^i$ and the All-Gather time T_{AG}^i for $Buffer_{AG}^i$. T_{AG}^i begins after either T_{AR}^i completes or after the computations of all tensors in the separated $Buffer_{AG}^i$ finish. Accordingly, $T_2 = \text{Max}\left(T_{AR}^i, \sum_{j=l'+1}^{N_i}\left(T_b^j + T_{gpu}^j\right)\right) + T_{AG}^i$.

Step-3: Minimizing T_1+T_2. We sequentially assign each tensor in the fusion buffer until finding $G_{l'}$ that minimizes T_1+T_2. The resulting separated buffers, $Buffer_{AR}^i$ and $Buffer_{AG}^i$, are then employed to enable optimal interleaved communication.

Interleaved All-Reduce and All-Gather communication. The process involves three steps: (i) For each fusion buffer, we obtain the separated buffers (denoted as $Buffer_{AR}$ and $Buffer_{AG}$) that minimize the sum of time costs T_1+T_2; (ii) For each fusion buffer, we repeatedly perform an interleaving process of an All-Reduce communication on $Buffer_{AR}$ and an All-Gather communication on $Buffer_{AG}$; (iii) Once the interleaved communications are completed, we decompress the synchronized result in $Buffer_{AG}$, while the result from $Buffer_{AR}$ remains unchanged.

As illustrated in Fig. 10(a), $Buffer_1$ and $Buffer_2$ contain both sparsified and unsparsified tensors. When SSFusion uses only All-Gather for communication on each fusion buffer, it incurs significant communication overhead. In contrast, Fig. 10(b) shows that SSFusion-I first determines the optimal separation of $Buffer_1$ into $Buffer_{AR}^1$ for All-Reduce and $Buffer_{AG}^1$ for All-Gather by minimizing the T_1+T_2. Interleaving All-Reduce and All-Gather then significantly reduces communication time compared to Fig. 10(a).

E. Discussion

Generalizability of SSFusion-O and SSFusion-I. For SSFusion-O, its performance benefits can be *fully* leveraged under any CPU-to-GPU power ratio. This is because for any given ratio, tensors are offloaded from GPU to CPU only if their CPU sparsification can overlap with GPU backpropagation (Step-1 in §IV-C), reflecting the potential benefits of pipelining.

For SSFusion-I, its performance benefits are expected to be more pronounced than those of SSFusion without interleaved communication in slower Ethernet networks. This is because the former transfers only part of the tensors via All-Gather, whereas the latter transfers all tensors via All-Gather. Since All-Gather typically incurs higher communication overhead than All-Reduce (§IV-D), reducing the number of tensors transmitted via All-Gather leads to better communication performance.

GPU-CPU data transfer overhead. Although GPU–CPU data transfers can introduce PCIe I/O overhead, this overhead is limited in our system for two reasons: (i) we use a high-bandwidth PCIe Gen4 interface (§V-A), which substantially mitigates the transfer bottleneck, and (ii) we selectively offload only large tensors whose CPU sparsification can overlap with GPU backpropagation, instead of transferring all tensors (§IV-C). Nevertheless, in more general settings or on less advanced infrastructures, PCIe I/O overhead may still become a bottleneck, which we plan to address in future work.

TABLE IV: Tasks, models and datasets used for evaluation.

Training Tasks	Models	Model Size	Datasets
Image Classification	ViT-large [64]	1.2 GB	ImageNet [69]
	ResNet-152 [58]	230 MB	ImageNet [69]
	ResNet-152 [58]	230 MB	Cifar-100 [59]
	VGG-19 [70]	548 MB	Cifar-100 [59]
Natural Language Processing	GPT2-large [4]	3.2 GB	WikiText-103 [71]
	LSTM [61]	328 MB	WikiText-2 [71]
	BERT-large [2]	1.32 GB	SQuAD [60]
	BERT-base [2]	420 MB	SQuAD [60]
	GPT2-XL [4]	6 GB	WikiText-103 [71]
	BLOOM-3B [72]	12 GB	WikiText-103 [71]

V. Evaluation

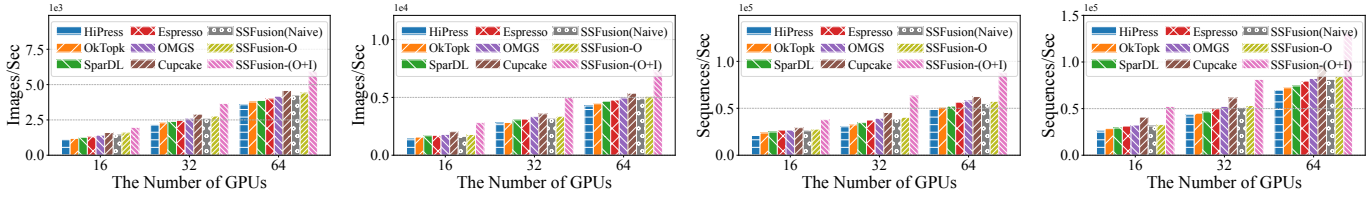
A. Experiment Setup

Testbed. We carried out our experiments on two testbeds: 1) a cloud cluster with 64 GPUs across 16 nodes, connected with 200 Gbps InfiniBand. Each node has two 32-core EPYC 7543 CPUs and 4 NVIDIA A100 (80 GB); 2) a local cluster of 8 nodes, each of which is equipped with one NVIDIA V100S (32 GB), two 12-core Intel XEON Silver 4214 CPUs, and all nodes are connected by a 100 Gbps InfiniBand.

Each machine runs Ubuntu 20.04 and includes CUDA 12.0, NCCL-2.8.3, OpenMPI-4.0.3, Horovod-0.28.1, and PyTorch-2.3.0. Within each node, GPUs are connected to host memory and to each other via PCIe Gen4.

Workloads. Our evaluation encompasses nine widely-used DNN models, comprising three image classification models and four natural language processing models, which are listed in Table IV. We set batch sizes and learning rates across different models similar to the studies [2], [6], [12], [17]. Following prior work [28], we apply DGC [44] with 1% sparsification density (i.e., compression rate) to the training.

Comparative approaches. To establish a comprehensive benchmark, we compare SSFusion with state-of-the-art sparsification schemes such as HiPress [6] and Espresso [12], as well as



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

Fig. 11: Experiment 1: Throughput of different DNN training tasks on a cloud cluster with 200 Gbps cross-machine Infiniband.

state-of-the-art tensor fusion approaches with sparsification, including OkTopk [20], OMGS [17], SparDL [33], and Cupcake [28], which are detailed in §II-D.

SSFusion settings. For **§V-B Experiment 1**, we compare three versions of SSFusion: SSFusion(Naive), which performs the basic selective sparsification (§IV-B); SSFusion-O, which extends SSFusion(Naive) with sparsification offloading (§IV-C); SSFusion-(O+I), which extends SSFusion-O with interleaved communication (§IV-D). For Experiments 2–7, we use SSFusion to denote SSFusion-(O+I) for short.

B. Experiments

Experiment 1 (Training throughput): We evaluate the training throughput of SSFusion and other state-of-the-art schemes on real-world training tasks listed in Table IV, conducted on a cloud cluster. Fig. 11 shows that the training throughput achieved by SSFusion-(O+I) on the four training tasks is much higher than that of the other schemes. We see that:

Image classification: Fig. 11(a) and Fig. 11(b) for ResNet-152 and ViT-large on ImageNet, respectively, show that the training throughput of SSFusion-(O+I) outperforms other state-of-the-art methods by 28.7%-74.2% and 35.8%-89.5%, respectively, versus different GPU cluster sizes.

Natural language processing: For BERT-large on SQuAD, Fig. 11(c) shows that SSFusion-(O+I) improves training throughput by 29.1%-106.7% compared to state-of-the-arts, versus different GPU cluster sizes. Fig. 11(d) for GPT2-large on WikiText-103, shows that SSFusion-(O+I) improves training throughput by 27.5%-84.1%, versus different GPU cluster sizes.

Effectiveness of various optimizations: We evaluate the individual performance gains of the various optimization schemes introduced in the three versions of SSFusion: SSFusion(Naive), SSFusion-O, and SSFusion-(O+I).

As shown in Fig. 11, SSFusion(Naive) outperforms HiPress by 14.8%-26.2% on four training tasks. This means that SSFusion(Naive) reduces sparsification overhead and maximizes communication-computation overlap through tensor fusion with selective sparsification. Compared to SSFusion(Naive), SSFusion-O improves the training throughput by 8.2%-12.7%. This is because SSFusion-O further reduces sparsification overhead by offloading large tensors in SSFusion(Naive) to the CPU. Compared to SSFusion-O, SSFusion-(O+I) improves the training throughput by 54.5%-64.0% on four training tasks. This implies that SSFusion-(O+I)'s interleaved communication scheme avoids the inefficiency of sparse communication caused by using only All-Gather.

Analysis: We elaborate on SSFusion-(O+I)'s benefits over other state-of-the-arts and analyze the reasons behind.

(1) SSFusion-(O+I) and SSFusion-O improve training throughput by 66.9%–106.7% and 22.6%–40.1% over HiPress, respectively. HiPress selects tensors for sparsification based on their sparsification benefits and synchronizes them in batches, but its layer-wise sparsification and poor communication-computation overlap limit training efficiency.

(2) SSFusion-(O+I) achieves 51.7%-89.3% higher training throughput than OkTopk and SparDL. OkTopk merges all tensors into a single fusion buffer and adds operations to balance gradients across workers, causing poor communication-computation overlap and high latency. SparDL mitigates the sparse gradient accumulation dilemma by partitioning and packing, but incurs additional computational overhead.

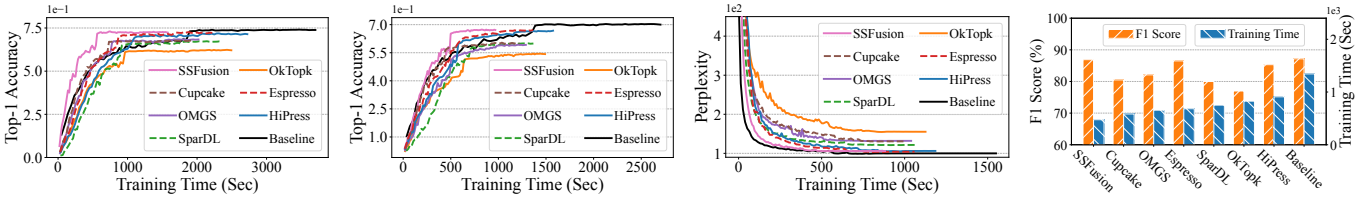
(3) The training throughput of SSFusion-(O+I) and SSFusion-O outperforms Espresso by 41.5%–68.7% and 8.9%–21.4%, respectively. While Espresso reduces sparsification overhead by selectively offloading tensors to the CPU for sparsification via decision algorithms, its lack of tensor fusion support leads to inefficient communication.

(4) Training throughput of SSFusion-(O+I) is 27.5%-65.4% higher than that of OMGS and Cupcake. Although both OMGS and Cupcake employ tensor fusion with multi-tensor sparsification to reduce sparsification overhead, their use of All-Gather alone still limits training throughput.

(5) SSFusion-(O+I) achieves the best performance. It employs selective sparsification and sparsification offloading to reduce sparsification overhead and maximize communication-computation overlap, while interleaved communication further improves communication efficiency. SSFusion-(O+I) scales effectively across clusters with different numbers of GPUs.

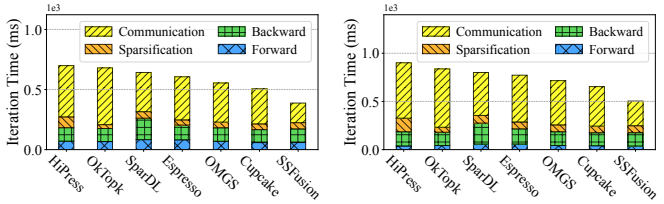
Experiment 2 (End-to-end convergence performance): We evaluate the end-to-end convergence performance of four training tasks on a local cluster. To ensure fairness, all methods are trained with the same number of epochs and sparsification density. As shown in Fig. 12, SSFusion achieves convergence accuracy close to Baseline (non-sparsification) and outperforms other multi-tensor sparsification schemes. We see that:

Image classification: Fig. 12(a) and 12(b) show that the Top-1 accuracy of SSFusion achieves 72.3% and 67.1%, which are slightly lower than that of Baseline (i.e., 73.6% and 70.0%), but higher than other multi-tensor sparsification methods. Specifically, OkTopk, SparDL, OMGS, and Cupcake achieve 62.1%, 67.5%, 68.3%, and 68.1% on ResNet-152, and 54.3%, 61.2%, 60.2%, and 60.8% on VGG-19.



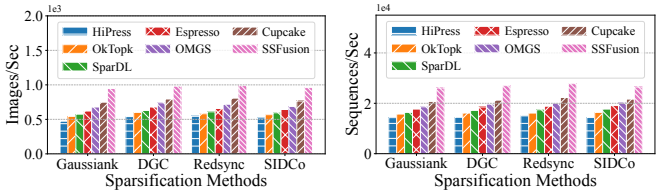
(a) ResNet-152 on Cifar-100 **(b)** VGG-19 on Cifar-100 **(c)** LSTM on WikiText-2 **(d)** BERT-base on SQuAD

Fig. 12: Experiment 2: Convergence performance of different DNN training tasks on a local cluster with 100 Gbps cross-machine Infiniband.



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

Fig. 13: Experiment 3: Training time breakdown for different tensor fusion schemes on two training tasks.



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

Fig. 14: Experiment 4: Training performance comparison using different sparsification methods.

Natural language processing: Fig. 12(c) shows that SSFusion achieves a Perplexity of 106, close to the Baseline(102) and lower than OkTopk (159), SparDL (121), OMGS (131), and Cupcake (130). Note that the smaller the Perplexity, the higher the prediction accuracy of LSTM. Fig. 12(d) shows SSFusion achieves an F1 score of 86.5%, close to the Baseline (87.2%), while OkTopk, SparDL, OMGS, and Cupcake achieve 76.8%, 79.8%, 82.9%, and 80.5%, respectively.

Analysis: The reasons behind this can be analyzed and are summarized as follows:

(1) Existing multi-tensor sparsification methods, such as OkTopk, SparDL, OMGS, and Cupcake, apply sparsification to all fused tensors in the fusion buffer, which leads to missing tensors and degrades convergence accuracy.

(2) HiPress and Espresso achieve the same high convergence accuracy as SSFusion because both employ layer-wise gradient sparsification, which does not impact the convergence accuracy of their original algorithms. SSFusion performs sparsification on each tensor independently before fusion, avoiding missing tensors and ensuring convergence accuracy.

(3) Compared to other state-of-the-art approaches, SSFusion also reduces end-to-end training time by 12.5%–74.1% by reducing sparsification overhead through tensor fusion with selective sparsification and by improving communication efficiency via interleaved communication.

Experiment 3 (Training time breakdown): To better illustrate how SSFusion improves training efficiency, we break down the iteration time into four components: forward, backward, sparsification, and communication. Fig. 13(a) and 13(b) show the iteration time breakdown for ResNet-152 and BERT-large on a cloud cluster, respectively. SSFusion reduces communication time by 61.7% and 55.4% compared with HiPress, by 65.5% and 57.8% compared with OkTopk, by 33.4% and 40.0% compared with SparDL, by 54.5% and 47.3% compared with Espresso, by 42.5% and 44.2% compared with OMGS, and by 33.0% and 37.6% compared with Cupcake.

Despite using bulk synchronization to improve training efficiency, HiPress suffers from an unoptimized communication–computation pipeline, leading to low communication efficiency. OkTopk merges all tensors and communicates them together, which limits pipelining efficiency. Although SparDL uses the spar-reduce-scatter algorithm to reduce communication time compared to OkTopk, its partitioning and packaging increase backpropagation cost. Without tensor fusion, Espresso suffers from significant communication startup overhead. OMGS and Cupcake overlap computation and communication by tuning the fusion buffer size, but remain inefficient due to their reliance on All-Gather. In contrast, SSFusion significantly reduces communication time by interleaving communication.

In addition, SSFusion reduces sparsification time by 41.9% and 48.3% on the two training tasks, respectively, compared to HiPress. This value is very close to those of Espresso, OMGS, and Cupcake. This is because SSFusion's tensor fusion with selective sparsification and offloading can minimize the sparsification overhead.

Experiment 4 (Impact of different sparsification methods): We have completed the performance evaluation on the sparsification method DGC in Experiment 1 and Experiment 3 and proved that our SSFusion outperforms the other state-of-the-art methods. Here, we will compare several sparsification methods (e.g., Gaussiank [22], DGC [44], RedSync [23], and SIDCo [24]) and evaluate their performance across state-of-the-art tensor fusion schemes on a cloud cluster.

Fig. 14 shows the training throughput of HiPress, OkTopk, SparDL, Espresso, OMGS, Cupcake, and SSFusion on ResNet-152 and BERT-large using different sparsification methods with the same settings as in Experiment 1. Overall, SSFusion achieves higher training throughput than other schemes. Specifically, for ResNet-152 on ImageNet, SSFusion achieves throughput improvements of 26.5%–102.3%, 23.4%–81.7%, 22.6%–79.8%, and 25.3%–83.9% over the other schemes across the four sparsification methods. For BERT-large on SQuAD, SSFusion increases training throughput by 28.8%-86.4%, 27.1%-87.1%, 24.6%-82.0%, and 22.3%-84.6%, respectively,

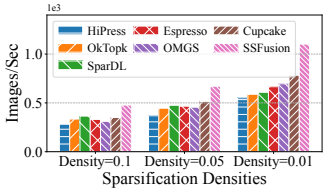


Fig. 15: Experiment 5: Training performance comparison using different sparsification densities.

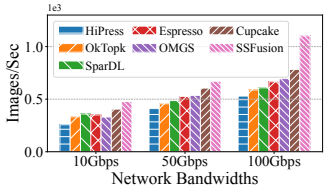


Fig. 16: Experiment 6: Training performance comparison using different bandwidths.

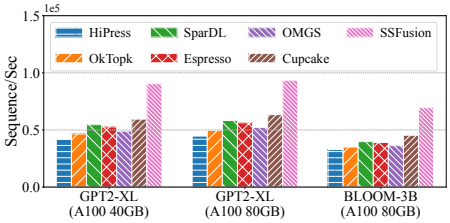


Fig. 17: Experiment 7: Training throughput of two large-scale training models on a cloud cluster with 64 NVIDIA A100 (40GB/80GB) GPUs.

across the four sparsification methods.

Experiment 5 (Impact of sparsification density): Fig. 15 compares the training throughput of HiPress, OkTopk, SparDL, Espresso, OMGS, Cupcake, and SSFusion on ResNet-152 using different sparsification densities (i.e., density=0.1, 0.05, and 0.01) with the same settings as in Experiment 1. Compared to other state-of-the-art tensor fusion schemes, SSFusion achieves throughput improvements of 36.3%-70.1%, 34.0%-82.0%, and 25.5%-98.5% when the density is reduced from 0.1 to 0.05 and 0.01, respectively.

We observe that at larger sparsification densities (e.g., density=0.1), OkTopk and SparDL have higher training throughput than Espresso and OMGS. The increase in density leads to the transmission of more sparsified gradients, resulting in the sparse gradient accumulation dilemma. OkTopk and SparDL alleviate the above problems by partitioning and adjusting sparsified tensors, but the additional computation still limits training efficiency. Nevertheless, SSFusion's training throughput remains higher than that of other state-of-the-arts.

Experiment 6 (Impact of network bandwidths): We compare the training throughput of SSFusion against other state-of-the-art methods on a cloud cluster with network bandwidths of 10 Gbps, 50 Gbps, and 100 Gbps.

Fig. 16 shows that SSFusion achieves higher training throughput than other state-of-the-art schemes under different bandwidth networks. Specifically, SSFusion improves throughput by 31.5%–86.1%, 22.7%–63.3%, and 25.5%–98.5% as the network bandwidth increases from 10 Gbps to 50 Gbps and 100 Gbps, respectively, compared to other tensor fusion schemes. This indicates that SSFusion can effectively increase training throughput in high communication pressure (e.g., 10 Gbps network).

Experiment 7 (Scalability). In Experiment 1, we evaluate SSFusion and other state-of-the-art schemes on four models using cloud clusters with 16, 32, and 64 GPUs. Results show that SSFusion consistently achieves higher training throughput, with the performance gap widening as the cluster size increases.

TABLE V: Experiment 8: The ratio of SSFusion's online profiling time to the total training time across four DNN models.

Models	ResNet-152	ViT-large	BERT-large	GPT2-large
Overhead Ratio (%)	0.04%	0.06%	0.09%	0.08%

This scalability stems from SSFusion's low sparsification overhead and efficient communication.

As shown in Fig. 17, we evaluate SSFusion and other schemes on two large-scale models, GPT2-XL and BLOOM-3B, using a cloud cluster with 64 A100 (40 GB/80 GB) GPUs. The results are consistent with Experiment 1: SSFusion achieves higher training throughput than state-of-the-art methods. Specifically, for GPT2-XL on A100 80 GB, SSFusion improves throughput by 47.5%-109.7% over HiPress, OkTopk, SparDL, Espresso, OMGS, and Cupcake. For BLOOM-3B on A100 80 GB, the improvement reaches 53.6%-117.0%. These results demonstrate that SSFusion maintains its performance advantages on large-scale models.

We compare the training throughput of SSFusion by training GPT2-XL on a cloud cluster with A100 40GB GPUs against that with A100 80GB GPUs. Note that GPT2-XL's memory usage (38.2GB) nearly reaches the memory limit in A100-40GB-GPU clusters. Fig. 17 shows that SSFusion on A100 40GB GPUs achieves 97% of the throughput on A100 80GB GPUs. This indicates that SSFusion still maintains a performance benefit under high memory pressure.

Experiment 8 (Online profiling overhead). We also evaluate the ratio of SSFusion's online profiling time to total training time. As shown in Table §V, this ratio is negligible across the four training models. Specifically, the online profiling time ratios for ResNet-152, ViT-large, BERT-large, and GPT2-large are 0.04%, 0.06%, 0.09%, and 0.08%, respectively. These results indicate that SSFusion's online profiling time has a negligible impact on actual training performance.

VI. Conclusion

SSFusion proposes a new selective sparsification tensor fusion mechanism for distributed DNN training that addresses the limitations of high sparsification overhead or low convergence accuracy caused by existing tensor fusion schemes. SSFusion also proposes an efficient sparsification offloading scheme to further speed up sparsification, and an interleaved communication scheme to improve sparse communication efficiency. SSFusion improves the training throughput by 27.5%-106.7% compared to the state-of-the-arts, while maintaining almost the same convergence accuracy as the non-sparsification baseline.

Acknowledgments

We thank all the anonymous reviewers for their constructive comments. This work was supported in part by the National Key Research and Development Program of China (No. 2022YFB4501300), the National Natural Science Foundation of China (No. 62272185), and the Key Laboratory of Information Storage System, Ministry of Education of China.

AI-Generated Content Acknowledgement

The authors confirm that no generative AI tools or large language models were used in the preparation of this paper.

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 6000–6010.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, 2020, pp. 1877–1901.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[5] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[6] Y. Bai, C. Li, Q. Zhou, J. Yi, P. Gong, F. Yan, R. Chen, and Y. Xu, "Gradient compression supercharged high-performance data parallel dnn training," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 359–375.

[7] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania *et al.*, "Pytorch distributed: experiences on accelerating data parallel training," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3005–3018, 2020.

[8] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "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*, 2020, pp. 463–479.

[9] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.

[10] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu, "Communication-efficient decentralized machine learning over heterogeneous networks," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 384–395.

[11] A. Gibiansky, "Bringing hpc techniques to deep learning," *Baidu Research, Tech. Rep.*, 2017.

[12] Z. Wang, H. Lin, Y. Zhu, and T. S. E. Ng, "Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23, 2023, p. 867–882.

[13] S. Shi, X. Chu, and B. Li, "Mg-wfbp: Efficient data communication for distributed synchronous sgd algorithms," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 172–180.

[14] T. Liu, T. Miao, Q. Wu, Z. Li, G. He, J. Wu, S. Zhang, X. Yang, G. Tyson, and G. Xie, "Modeling and optimizing the scaling performance in distributed deep learning training," in *Proceedings of the ACM Web Conference 2022*, 2022, pp. 1764–1773.

[15] L. Zhang, S. Shi, and B. Li, "Accelerating distributed k-fac with efficient collective communication and scheduling," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[16] Y. Liu, B. Jiang, S. Zhao, T. Lin, X. Wang, and C. Zhou, "Libra: Contention-aware gpu thread allocation for data parallel training in high speed networks," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[17] S. Shi, Q. Wang, X. Chu, B. Li, Y. Qin, R. Liu, and X. Zhao, "Communication-efficient distributed deep learning with merged gradient sparsification on gpus," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 406–415.

[18] L. Zhang, S. Shi, X. Chu, W. Wang, B. Li, and C. Liu, "Dear: Accelerating distributed deep learning with fine-grained all-reduce pipelining," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2023, pp. 142–153.

[19] Z. Ming, Y. Hu, W. Zhou, X. Zheng, C. Yao, and D. Feng, "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*, 2024, pp. 135–147.

[20] S. Li and T. Hoefler, "Near-optimal sparse allreduce for distributed deep learning," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 135–149.

[21] S. Shi, K. Zhao, Q. Wang, Z. Tang, and X. Chu, "A convergence analysis of distributed sgd with communication-efficient gradient sparsification," in *IJCAI*, 2019, pp. 3411–3417.

[22] S. Shi, X. Chu, Cheung, and S. See, "Understanding top-k sparsification in distributed deep learning," *arXiv preprint arXiv:1911.08772*, 2019.

[23] J. Fang, H. Fu, G. Yang, and C.-J. Hsieh, "Redsync: reducing synchronization bandwidth for distributed deep learning training system," *Journal of Parallel and Distributed Computing*, pp. 30–39, 2019.

[24] A. M Abdelmoniem, A. Elzanaty, M.-S. Alouini, and M. Canini, "An efficient statistical-based gradient compression technique for distributed training systems," in *Proceedings of Machine Learning and Systems*, 2021, pp. 297–322.

[25] A. Sahu, A. Dutta, A. M Abdelmoniem, and P. Kalnis, "Rethinking gradient sparsification as total error minimization," in *Advances in Neural Information Processing Systems*, 2021, pp. 8133–8146.

[26] "Byteps," 2025, https://github.com/bytedance/byteps.

[27] "Deepspeed," 2025, https://https://github.com/microsoft/DeepSpeed.

[28] Z. Wang, X. C. Wu, Z. Xu, and T. E. Ng, "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), 2023.

[29] Z. Ming, Y. Hu, X. Zheng, W. Zhou, and D. Feng, "Safusion: Efficient tensor fusion with sparsification ahead for high-performance distributed dnn training," in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, 2025, pp. 1–14.

[30] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[31] J. Romero, J. Yin, N. Laanait, B. Xie, M. T. Young, S. Treichler, V. Starchenko, A. Borisevich, A. Sergeev, and M. Matheson, "Accelerating collective communication in data parallel training across deep learning frameworks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 1027–1040.

[32] S. Shi, X. Chu, and B. Li, "Exploiting simultaneous communications to accelerate data parallel distributed deep learning," in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[33] M. Zhao, Y. Yin, Y. Mao, Q. Liu, L. Chen, and Y. Gao, "Spardl: Distributed deep learning training with efficient sparse communication," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 1752–1764.

[34] "Horovod," 2025, https://github.com/horovod/horovod.

[35] "Nvidia nccl," 2025, https://developer.nvidia.com/nccl.

[36] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "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*, 2017, pp. 193–205.

[37] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *USENIX Annual Technical Conference*, vol. 1, 2017, pp. 1–2.

[38] S. Shi, X. Chu, and B. Li, "Mg-wfbp: Merging gradients wisely for efficient communication in distributed deep learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 8, pp. 1903–1917, 2021.

[39] A. Jangda, J. Huang, G. Liu, A. H. N. Sabet, S. Maleki, Y. Miao, M. Musuvathi, T. Mytkowicz, and O. Saarikivi, "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*, 2022, pp. 402–416.

[40] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 16–29.

[41] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 785–808.

[42] H. Xu, C.-Y. Ho, A. M. Abdelmoniem, A. Dutta, E. H. Bergou, K. Karatsenidis, M. Canini, and P. Kalnis, "Grace: A compressed

communication framework for distributed machine learning," in *2021 IEEE 41st international conference on distributed computing systems (ICDCS)*. IEEE, 2021, pp. 561–572.

[43] H. Hu, C. Jiang, Y. Zhong, Y. Peng, C. Wu, Y. Zhu, H. Lin, and C. Guo, "dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 623–637, 2022.

[44] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *International Conference on Learning Representations*, 2018.

[45] J. Fei, C.-Y. Ho, A. N. Sahu, M. Canini, and A. Sapio, "Efficient sparse collective communication and its application to accelerate distributed deep learning," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 676–691.

[46] C. Renggli, S. Ashkboos, M. Aghagolzadeh, D. Alistarh, and T. Hoefler, "Sparcml: High-performance sparse communication for machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–15.

[47] S. Shi, Q. Wang, K. Zhao, Z. Tang, Y. Wang, X. Huang, and X. Chu, "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, 2019, pp. 2238–2247.

[48] H. Xu, K. Kostopoulou, A. Dutta, X. Li, A. Ntoulas, and P. Kalnis, "Deepreduce: A sparse-tensor communication framework for federated deep learning," in *Advances in Neural Information Processing Systems*, 2021, pp. 21 150–21 163.

[49] D. Wu, W. Yang, C. Deng, X. Zou, S. Li, and W. Xia, "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, 2023, pp. 605–613.

[50] Z. Zhang and C. Wang, "Mipd: An adaptive gradient sparsification framework for distributed dnns training," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 3053–3066, 2022.

[51] Z. Tang, S. Shi, B. Li, and X. Chu, "Gossipfl: A decentralized federated learning framework with sparsified and adaptive communication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 909–922, 2022.

[52] J. Gui, Y. Song, Z. Wang, C. He, and Q. Huang, "Sk-gradient: Efficient communication for distributed machine learning with data sketch," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2372–2385.

[53] J. Wangni, J. Wang, J. Liu, and T. Zhang, "Gradient sparsification for communication-efficient distributed optimization," in *Advances in Neural Information Processing Systems*, 2018, pp. 1306–1316.

[54] D. Alistarh, T. Hoefler, M. Johansson, N. Konstantinov, S. Khirirat, and C. Renggli, "The convergence of sparsified gradient methods," in *Advances in Neural Information Processing Systems*, 2018, p. 5977–5987.

[55] G. Yan, S.-L. Huang, T. Lan, and L. Song, "Dq-sgd: Dynamic quantization in sgd for communication-efficient distributed learning," in *2021 IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS)*. IEEE, 2021, pp. 136–144.

[56] W. Wen, C. Xu, F. Yan, C. Wu, Y. Wang, Y. Chen, and H. Li, "Terngrad: Ternary gradients to reduce communication in distributed deep learning," in *Advances in Neural Information Processing Systems*, 2017, pp. 1508–1518.

[57] J. Wu, W. Huang, J. Huang, and T. Zhang, "Error compensated quantized sgd and its applications to large-scale distributed optimization," in *International Conference on Machine Learning*. PMLR, 2018, pp. 5325–5333.

[58] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[59] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," *Master's thesis, University of Tront*, 2009.

[60] P. Rajpurkar and P. Liang, "Know what you don't know: Unanswerable questions for squad," *arXiv preprint arXiv:1806.03822*, 2018.

[61] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," *arXiv preprint arXiv:1708.02182*, 2017.

[62] X. Glorot and Y. Bengio, "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, 2010, pp. 249–256.

[63] A. Dutta, E. H. Bergou, A. M. Abdelmoniem, C.-Y. Ho, A. N. Sahu, M. Canini, and P. Kalnis, "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*, 2020, pp. 3817–3824.

[64] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[65] S. U. Stich, J.-B. Cordonnier, and M. Jaggi, "Sparsified sgd with memory," in *Advances in Neural Information Processing Systems*, 2018, pp. 4452–4463.

[66] S. Agarwal, H. Wang, S. Venkataraman, and D. Papailiopoulos, "On the utility of gradient compression in distributed training systems," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 652–672, 2022.

[67] R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*. Springer, 2004, pp. 1–9.

[68] J. Peng, Z. Li, S. Shi, and B. Li, "Sparse gradient communication with alltoall for accelerating distributed deep learning," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 148–157.

[69] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[70] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[71] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[72] T. Le Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," 2023.