A Generic, High-Performance, Compression-Aware Framework for Data Parallel DNN Training

Hao Wu, Shiyi Wang, Youhui Bai, *Member, IEEE,* Cheng Li, *Member, IEEE,* Quan Zhou, Jun Yi, Feng Yan, *Member, IEEE,* Ruichuan Chen, Yinlong Xu, *Member, IEEE,*

Abstract—
Gradient compression is a promising approach to alleviating the communication bottleneck in data parallel deep neural network (DNN) training by significantly reducing the data volume of gradients for synchronization. While gradient compression is being actively adopted by the industry (e.g., Facebook and AWS), our study reveals that there are two critical but often overlooked challenges: 1) inefficient coordination between compression and communication during gradient synchronization incurs substantial overheads, and 2) developing, optimizing, and integrating gradient compression algorithms into DNN systems imposes heavy burdens on DNN practitioners, and ad-hoc compression implementations often yield surprisingly poor system performance. In this paper, we propose a compression-aware gradient synchronization architecture, `CaSync`, which relies on flexible composition of basic computing and communication primitives. It is general and compatible with any gradient compression algorithms and gradient synchronization strategies and enables high-performance computation-communication pipelining. We further introduce a gradient compression toolkit, `CompLL`, to enable efficient development and automated integration of on-GPU compression algorithms into DNN systems with little programming burden. Lastly, we build a compression-aware DNN training framework `HiPress` with `CaSync` and `CompLL`. `HiPress` is open-sourced and runs on mainstream DNN systems such as MXNet, TensorFlow, and PyTorch. Evaluation via a 16-node cluster with 128 NVIDIA V100 GPUs and a 100Gbps network shows that `HiPress` improves the training speed over current compression-enabled systems (e.g., BytePS-onebit, Ring-DGC and PyTorch-PowerSGD) by 9.8%-69.5% across six popular DNN models.

Index Terms—Data parallel DNN training, gradient compression

1 Introduction

To efficiently train large DNN models over the continuously growing datasets, it has been a norm to employ data parallel DNN training to explore massive parallelism in an increasingly large cluster of GPU nodes [20], [55], [57], [78], [102]. In a typical data parallel setting, each node iterates over its data partition in parallel, and exchanges a large volume of gradients with other nodes per iteration via a gradient synchronization strategy like Parameter Server (PS) [33], [38] or Ring-allreduce [9].

However, in recent years, the fast-growing computing capability, driven by the booming of GPU architecture innovations [60] and domain-specific compiler techniques [16], [19], [70], [72], tends to result in more frequent and heavier gradient synchronization during data parallel DNN training. This trend puts high pressure on the slower-growing bandwidth and reduces the chance of pipelining computation and communication during training. We have found that, even with the latest highly-optimized BytePS [33] and Ring-allreduce [78] synchronization strategies, the communication time for gradient synchronization still accounts for 63.6% and 76.8% of the total time for training the Bert-large and Transformer models across 16 AWS EC2 instances, each with 8 NVIDIA V100 GPUs, in a 100Gbps network. Thus, *there is a fundamental tension between gradient communication and computation in data parallel DNN training* [75].

Gradient compression algorithms have a great potential to relieve or even eliminate the above tension, since they can substantially reduce the data volume being synchronized with a negligible impact on training accuracy and convergence [5], [44], [84], [94], [96]. This practice of gradient compression is being adopted by the industry. In fact, the efforts from Facebook and AWS to bring gradient compression to mainstream DNN systems have begun since June 2020 [6], [51], [53]. However, our experiment shows that the actual training speedups of compression-enabled DNN systems are far behind their expectations. For instance, applying gradient compression to the aforementioned Transformer training achieves only a 1.3× speedup, 38.1% lower than the expected performance. The gap becomes even larger in a lower-bandwidth network. This surprising observation

- *Hao Wu, Shiyi Wang, Youhui Bai (corresponding author), Cheng Li (corresponding author), Quan Zhou and Yinlong Xu are with the Department of Computer Science and Technology, University of Science and Technology of China, Hefei, China. Cheng Li and Yinlong Xu are also with the Anhui Province Key Laboratory of High Performance Computing, and Institute of Artificial Intelligence, Hefei Comprehensive National Science Center.*
  E-mail: {mark14, wsy0111, byh0912, zzqq2199}@mail.ustc.edu.cn, {chengli7, ylxu}@ustc.edu.cn
- *Feng Yan is with the Computer Science Department and Electrical and Computer Engineering Department of University of Houston.*
  E-mail: fyan5@central.uh.edu
- *Ruichuan Chen is a Distinguished Member of Technical Staff at Nokia Bell Labs.*
  E-mail: ruichuan.chen@nokia-bell-labs.com

Manuscript received April 19, 2005; revised August 26, 2015.

drives us to rethink gradient compression from the system perspective.

To fully unleash the benefits of gradient compression, only an efficient compression algorithm is not sufficient. The compressed gradients are not directly aggregatable, and they are not compatible with common optimizations (such as gradient partitioning and batching) used in the conventional gradient synchronization strategies. In the current compression-enabled DNN system designs, the computational overhead introduced by gradient compression is often overlooked and could be greatly amplified along the gradient synchronization path. Therefore, the first challenge we have to address is how to amortize the extra computational overhead along the communication steps during gradient synchronization, whereby the computation and communication may have data dependencies. This requires us to revisit the original design choices across existing gradient synchronization strategies to identify the right granularity of combining and coordinating various gradient compression and communication operators. Second, a sophisticated systematic support for compression awareness is generally lacking. Without such a support, DNN practitioners cannot live up to the full promise of gradient compression to accelerate DNN training. The adoption of gradient compression also becomes difficult because substantial system expertise and manual efforts are required for developing, optimizing, and integrating individual compression algorithm into DNN systems.

In this paper, we address these systems challenges to bridge the gap between gradient compression and synchronization in data parallel DNN training. We first propose a general, composable gradient synchronization architecture, called `CaSync`, which enables a compression-aware gradient synchronization with a composition of decoupled communication, aggregation, and compression primitives. This fine-grained composition allows us to strike a balance between 1) the effective pipelining of computational and communication tasks to hide communication overhead behind compression-related computation and vice versa, and 2) the efficient bulky execution of smaller tasks. Furthermore, `CaSync` employs a selective compression and partitioning mechanism to decide whether to compress each gradient and how to partition large gradients (before compression) to optimally leverage pipelining and parallel processing. It is worth mentioning that our `CaSync` architecture is intentionally designed to be general and not tie to specific gradient compression algorithms and synchronization strategies (e.g., PS or Ring-allreduce) so that its benefits are applicable to existing and potentially future compression algorithms and synchronization strategies.

Second, we advocate that the on-GPU compression is the preferred approach for gradient compression considering GPU has much higher bandwidth and processor density than CPU, and gradients are produced in GPU directly. This creates new opportunities to further optimize the compression-communication pipeline during gradient synchronization. However, developing and optimizing gradient compression algorithms on GPU is non-trivial and usually requires significant system expertise and manual efforts. To relieve the burden on DNN practitioners, we design and develop a gradient compression toolkit named `CompLL`, which facilitates the compression algorithm development and its integration on GPU. `CompLL` provides a unified API abstraction and exposes a library of highly-optimized common operators that can be used to construct sophisticated gradient compression algorithms. `CompLL` also offers a domain specific language to allow practitioners to specify their algorithm logic, which is then converted into efficient low-level GPU implementation and automatically integrated into DNN systems with little human intervention.

For easy adoption, we build a compression-aware data parallel DNN training framework called `HiPress`, with both `CaSync` and `CompLL`. `HiPress` is compatible with mainstream DNN systems (i.e., MXNet, TensorFlow, and PyTorch), and we have open-sourced it at [2].

We evaluate `HiPress` extensively. First, we use `CompLL` in `HiPress` to construct six state-of-the-art compression algorithms (i.e., onebit [76], TBQ [84], TernGrad [94], DGC [44], GradDrop [5] and PowerSGD [87]) with only 22 lines of `CompLL` code on average, and they achieve significant performance speedups over open-source counterparts. We train six widely-used DNN models across the computer vision and natural language processing fields using a 16-node cluster on AWS EC2 with 128 NVIDIA V100 GPUs and 100Gbps network links. Experimental results show that `HiPress` achieves speed improvements of 17.3%-110.5% and 9.8%-69.5% compared with non-compression systems (including the latest BytePS) and current compression-enabled systems (e.g., BytePS-onebit, Ring-DGC and PyTorch-PowerSGD), respectively. The results in a lower-end 16-node cluster with 32 1080Ti GPUs and 56Gbps network show a similar trend. Lastly, `HiPress` does not sacrifice the convergence and accuracy claims of exercised algorithms.

2 Background and Motivation

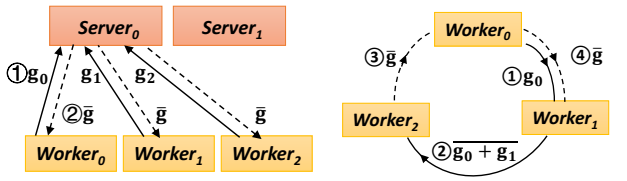
2.1 Data Parallel DNN Training

A DNN model typically consists of multiple neural network layers, each of which contains a large number of parameters. Training a DNN model needs to iterate over a dataset many times (i.e., *epochs*) towards convergence [97]. Each epoch is further split into *iterations*. Data parallel DNN training enables each training node to consume data from its own partition of the training dataset. In each iteration, training nodes independently run forward and backward propagation to generate *gradients*, which are then synchronized with other nodes to update the global model parameters collectively. This group coordination can be done synchronously or asynchronously. The former case often acts as a distributed barrier for convergence guarantees [106], while the latter case eliminates the negative impact of stragglers at the cost of possibly not converging. We focus on synchronous coordination because of its wide adoption [1], [15], [35], [69].

2.2 Gradient Synchronization

Parameter Server (PS) [38], [68] and AllReduce [9], [78] are two widely-adopted gradient synchronization strategies.

Parameter Server. In Figure 1a, each node acts as a *server* or a *worker* [38]. Model parameters and gradients are often partitioned across multiple servers for load balancing. When



(a) Parameter Server (PS) (b) Ring-allreduce

Fig. 1: Gradient synchronization strategies. For Parameter Server (PS), we only show interactions between $Server_0$ and $workers$ for clarity.

local training completes, each worker pushes gradients to servers (①), which are then aggregated and updated to model parameters. Afterward, each worker pulls the updated results from servers to trigger the next iteration (②).

AllReduce. This strategy uses collective communication primitives. One representative example is Ring-allreduce [9], where all nodes are *workers* and form a logical ring. In Figure 1b, it takes $N-1$ communication steps along the ring to aggregate gradients (①-②) and another $N-1$ steps to disseminate the updated gradient (③-④), where N is the number of workers. Furthermore, Ring-allreduce can batch gradients which are then partitioned again for load balancing. Following this, at each synchronization step, each worker simultaneously sends a partition to its successor and receives another partition from its predecessor to best utilize its bi-directional network bandwidth [67].

2.3 Computation and Communication Tension

Modern DNN systems pipeline computation and communication for better performance, e.g., via running the gradient communication and DNN backward computation of two DNN layers in parallel to hide the former overhead behind the latter when possible. However, there exists a fundamental tension between computation and communication [75].

The recent DNN accelerator booming [60] and domain-specific compiler advancement [16], [19], [70], [72] have significantly improved the single-node training speed. Such fast-advancing computing capabilities typically lead to more frequent gradient synchronization and thus put high pressure on the network infrastructure. However, the network upgrade does not keep up the pace of the computation-related advancements [46], [48], [64], [95]. The imbalance between the fast-advancing computing capability and the slower-advancing communication capability increasingly reduces the chance of pipelining the gradient communication and computation.

A few software approaches have been recently proposed to optimize the computation-communication pipeline, ranging from priority-based gradient scheduling and partitioning [69] to advanced synchronization architectures [33], [78]. However, as shown in Table 1, the latest highly-optimized BytePS [33] and Ring-allreduce [78] only achieve scaling efficiencies[1] of 0.71 and 0.47, when training two popular DNN models (Bert-large and Transformer) in a cluster of

1. Scaling efficiency is defined as $\frac{\text{actual\_performance}}{N \times \text{single\_GPU\_performance}}$, where N is the total number of GPUs, with 1 being the best (i.e., linear scaling).

TABLE 1: Training performance of Bert-large and Transformer with 16 AWS p3dn.24xlarge instances (8 V100 GPUs each), 100Gbps, BytePS 0.2.5, Horovod 0.19.2, fp32 precision.

	System configurations	Scaling efficiency	Communication ratio
Transformer	Ring-allreduce w/o compression	0.47	76.8%
	Ring-allreduce w/ DGC compression	0.61 (29.8%↑)	70.3% (8.5%↓)
Bert-large	BytePS w/o compression	0.71	63.6%
	BytePS w/ onebit compression	0.76 (7.0%↑)	60.9% (4.2%↓)

16 nodes on AWS EC2 with 128 NVIDIA V100 GPUs and a 100Gbps network. The communication time accounts for up to 76.8% of the total training time for training these two models, with a significant portion not being hidden behind DNN computation. This highlights the fundamental tension between gradient computation and communication persists in data parallel DNN training, even with state-of-the-art approaches and recent bandwidth advancements.

2.4 Gradient Compression

Gradient compression is a general approach for reducing the transmitted data volume during gradient synchronization [44], [87], [94], and has a great potential to alleviate the aforementioned communication bottleneck. Indeed, it is being adopted by the industry, and a number of recent efforts from Facebook and AWS have started to integrate gradient compression into modern DNN systems since June 2020 [6], [51], [53].

The gradient compression algorithms generally fall within the *sparsification*, *quantization* and *low-rank* categories. Sparsification leverages the sparsity of gradients and filters out insignificant elements in the gradient matrix [5], [35], [44], and quantization decreases the precision of gradients [84], [94], [96]. For instance, a 1-bit quantization enabled by onebit algorithm [76] could reduce the transmitted data volume by 96.9%. Unlike them, the low-rank methods decompose the original gradient matrix into multiple smaller matrices [18], [87], [89], [103]. For example, PowerSGD [87] aims to approximate the gradient with size NxM by multiplying two matrices with size NxR and RxM respectively. Obviously, the compression ratio of PowerSGD is $\frac{(N+M)\times R}{N\times M}$. Many of these algorithms either theoretically prove or empirically validate that adopting them does not affect model convergence and imposes only a negligible impact on accuracy, i.e., a compression-enabled DNN training converges to approximately the same accuracy through the same number of iterations compared with a non-compression training [44], [84], [94].

2.5 System Challenges and Opportunities

Surprisingly, our study reveals that, without proper system support, the gradient compression's benefits are diluted significantly at the best, and could even negatively affect the overall DNN training throughput at the worst.

One important reason for this surprising observation is that gradient compression requires non-negligible com-

putational overhead. Alongside the gradient synchronization path, an `encode` operator must precede sending fully or partially aggregated gradients, and a `decode` operator must follow when receiving compressed gradients. There could be up to $3N-2$ extra operators for each gradient synchronized across N workers. These extra operators are needed because it is impossible to directly aggregate over compressed gradients for most of the sparsification and quantization algorithms, due to the existence of metadata (in sparsification-based algorithms) or the potential overflow of operating low-precision numbers (in quantization-based algorithms).

The accumulated compression-related computational cost during gradient synchronization can significantly dilute its benefits of reducing the transmitted data volume. To demonstrate this, we train Bert-large with the onebit compression [76] developed by AWS and integrated into MXNet with BytePS.[2] Table 1 shows that BytePS-onebit achieves a very limited improvement over BytePS. As another example, the DGC compression [44] with 0.1% compression rate (where it is integrated into TensorFlow with the Ring-allreduce synchronization strategy) achieves only a 1.3× training speedup for the Transformer model. We discover that such limited improvements are mainly due to the co-design of BytePS and Ring-allreduce with the compression algorithms, whereby the compression logic is separated and scattered across gradient synchronization. Such a co-design also makes it difficult to verify the correctness of the implemented algorithms as well as to generalize to other gradient compression algorithms and synchronization strategies. To enable a general approach, it is important to separate the design of compression algorithms from that of synchronization strategies.

The first challenge to address the aforementioned issues lies in designing a general approach to amortize the extra computational overhead brought by gradient compression (e.g., `encode` and `decode` operators) along the communication steps during gradient synchronization. This is difficult due to non-trivial factors including, for instance, the data dependencies between gradient computation and communication, the communication topology such as a bipartite graph for PS and a ring for Ring-allreduce, the compression speed and ratio of different compression algorithms, to name a few. To address this challenge, the key is to identify the right granularity of combining and coordinating various gradient compression and communication operators.

Take Ring-allreduce as an example. It coordinates the communication of all training nodes by running a global, atomic, bulk synchronization operation to complete $2(N-1)$ point-to-point communication steps for batched gradients. While this design is bandwidth-optimal [67], such a *coarse-grained* approach fails to hide the compression-related overhead behind the communication overhead. Unlike Ring-allreduce, the PS synchronization strategy (including the latest BytePS) exchanges gradients via micro point-to-point communication steps. While such a *fine-grained* approach facilitates a better computation-communication pipelining to hide compression-related computational overhead, it incurs a larger number of communication steps and in turn a proportionally growing extra computational overhead.

The second challenge is to provide systematic support for developing, optimizing, and integrating gradient compression algorithms into DNN systems. Without this support, the real-world adoption of gradient compression algorithms requires significant system expertise and manual efforts to perform various ad-hoc development and optimization, which is particularly challenging for DNN practitioners. Thus it is quite difficult, if not impossible, for gradient compression to live up to its full promise of accelerating DNN training.

To provide general system support for various algorithms, one critical question to answer is where to perform their computation, e.g., on CPU or GPU? We observe that compression algorithms typically need to scan large gradient matrices multiple times to filter out insignificant gradients, decrease the precision of gradients, or decompose them into multiple sub-matrices with reduced sizes. Therefore, they are extremely memory-intensive and require massive parallelism to achieve fast compression (and decompression). We believe the on-GPU gradient compression is the preferred approach considering GPU's high memory bandwidth and many-core architecture. Furthermore, given that gradients produced by DNN computations are inherently in the GPU memory, the on-GPU compression can greatly alleviate the bandwidth tension of the PCIe bus between GPU and host. As an example, for the onebit compression algorithm [76], its CPU implementation runs 35.6× slower than the GPU-oriented counterpart (our implementation); using the same experimental setup as Table 1, BytePS with the on-CPU onebit introduces 95.2% training overhead than its on-GPU counterpart. Despite of on-GPU advantages, developing, optimizing and integrating on-GPU compression algorithms puts heavy burden on DNN practitioners, and doing it well requires extensive system expertise and the understanding of lower-level GPU hardware and CUDA programming details.

In summary, the above two challenges motivate us to rethink the abstraction for both gradient compression algorithms and compression-aware synchronization strategies, as well as to identify the common design patterns to support easy development, optimization, and integration of compression algorithms in DNN systems for real-world use.

3 Compression-Aware Synchronization

We propose `CaSync`, a compression-aware gradient synchronization architecture that provides a *general* support for gradient compression algorithms and synchronization strategies.

3.1 Composable, Pipelined Synchronization

As motivated in Section 2.5, a proper granularity of abstraction for gradient compression algorithms and synchronization strategies is the key to achieve a general yet high-performance gradient synchronization. To identify the right granularity, we employ a composable approach which first decouples all gradient synchronization primitives in a fine-grained manner, and then combines and coordinates them

2. The open-source onebit was implemented only on CPU [13]. For a fair comparison, we have implemented and integrated a highly-optimized on-GPU onebit into BytePS.

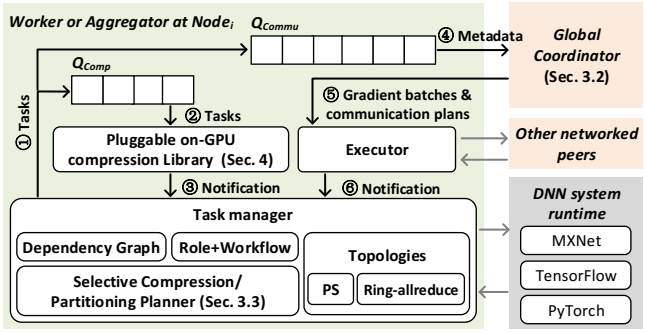


Fig. 2: The CaSync architecture design, where the DNN system runtime is omitted.

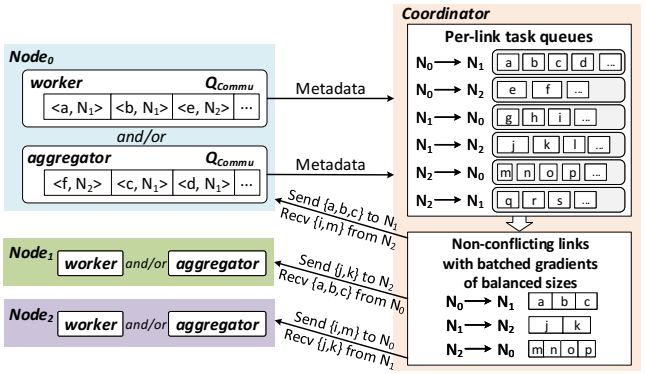


Fig. 3: The workflow of the compression-aware, coordinated bulk synchronization.

according to their data dependencies and order constraints to build an efficient computation-communication pipeline.

We first decouple the communication topology from gradient synchronization strategies. We represent the topology as a directed graph, where the vertex set contains training nodes and the edge set specifies the connections between these nodes. In gradient synchronization, there are fundamentally two node roles, namely, *worker* and *aggregator* (with potentially other roles serving for optimizations only). A worker produces gradients from its local DNN computation and initiates the gradient synchronization process. An aggregator aggregates gradients and then relays the aggregate result to workers or other aggregators. Take PS and Ring-allreduce as two examples of gradient synchronization strategies. As shown in Figure 1, for PS, we build bipartite connections between servers (i.e., aggregators) and workers; for Ring-allreduce, each node serves both roles and the clockwise connections are built between these nodes.

We then split the gradient synchronization process into five general primitives, namely, `encode`, `decode`, `merge`, `send` and `recv`. Specifically, '`encode`' and '`decode`' are two computing primitives for compressing and decompressing gradients, respectively. '`merge`' is another computing primitive for aggregating multiple gradients into one. '`send`' and '`recv`' are two communication primitives for sending and receiving gradients to and from other nodes, respectively. With these general primitives, we can conveniently specify a compression-aware workflow at each worker and aggregator, which defines proper data dependencies or order constraints between these primitives. For instance, '`encode`' precedes '`send`' at the worker because of the data dependency that the worker has to compress a gradient before sending it.

Figure 2 shows an overview of the CaSync design. With the aforementioned abstraction, we are able to design a holistic gradient synchronization architecture for both workers and aggregators. Each worker or aggregator employs a *task manager* to schedule and execute computing and communication tasks. Specifically, according to the node role, the task manager consults the specified workflow to select which series of computing and communication primitives to execute during gradient synchronization. Afterwards, according to the communication topology (e.g., a PS bipartite graph or a ring), the task manager informs the communication primitives where to send and receive compressed gradients.

The above fine-grained abstraction creates opportunities to pipeline computing and communication tasks for improved performance. As shown in Figure 2, at Step ①, the task manager pushes tasks into two task queues: Q_{comp} for computing tasks, and Q_{commu} for communication tasks. Tasks in Q_{comp} and Q_{commu} are executed in an asynchronous manner for efficient use of computing and networking resources. However, as tasks are spread in two independent task queues and are executed asynchronously, there is a high risk that the data dependencies between tasks are violated. Therefore, one challenge here is how to preserve data dependencies and order constraints when executing tasks from Q_{comp} and Q_{commu} asynchronously.

To ensure the proper order, the task manager maintains a dependency graph to manage data dependencies between tasks at runtime. For instance, for a compressed gradient, its '`recv`' task must first write to a memory buffer and only then it can be read by the '`decode`' task. Upon the completion of a computing task from Q_{comp} (step ②), it notifies the task manager to clear the following tasks' pending dependencies, and then promotes the execution of any task if all its pending dependencies are cleared (step ③). In doing so, the asynchronous execution of gradient synchronization is driven by the dependency graph among tasks. Note that, the step ④-⑥ correspond to a coordinated, compression-aware bulk synchronization mechanism in the next section.

3.2 Compression-aware Bulk Synchronization

While the above composable, pipelined synchronization can already improve the training performance significantly, it does not explore the opportunities brought by bulk synchronization — an important feature that is supported by most modern DNN systems. Instead of computing and communicating each gradient at a time, bulk synchronization handles gradients in a batched manner to better take advantage of parallelism and reduce the execution overhead [78]. Here, we extend the conventional bulk synchronization to be compression-aware, and additionally introduce batch compression to compress gradients in a batched manner. Our main goal is to reduce the compression and communication overheads. Compression-aware bulk synchronization is particularly important for small gradients as their compression

and communication overheads are difficult to be compensated by the size reduction brought by compression.

The *batch compression* in CaSync batches a sequence of compression-related tasks from Q_{comp} and schedules them together to GPU for compression. This allows a single callback function for a batch of gradients and thus also reduces the CPU-GPU coordination overhead. This is feasible as modern DNN systems often employ the operation fusing technique to produce multiple gradients at once [7].

The *bulk communication* in CaSync parallelizes the network transmission across training nodes to amortize the communication overheads across gradients. However, deciding the appropriate bulk granularity for communication is challenging. As discussed in Section 2.5, there are pros and cons for both fine-grained and coarse-grained granularity, and a proper balance needs to be struck. Our high-level design is that we slice the gradient synchronization process into monotonically increasing time slots, and select a group of network-idle nodes to join each time slot. In a slot, to avoid bandwidth contention, each selected node sends its gradients to only one other node via its uplink and receives gradients from its downlink. Note that, the transmitted gradients in a time slot may correspond to different communication steps (see Figure 1) in the gradient synchronization process. Together, the goal of this design is to enable the adaptive granularity of communication and the optimized node coordination during gradient synchronization.

Specifically, we introduce a global coordinator to adapt the communication of all gradients indiscriminately (compressed or not) and determine an optimal, coordinated communication plan. The plan should fulfill two goals: 1) maximize the utilization of network bandwidth between pairs of nodes, and 2) balance the size of transmitted gradients. The design of the global coordinator is shown in Figure 3. Each node (e.g., $Node_0$ or N_0) can serve as a worker or an aggregator or both, and it periodically sends the metadata ⟨gradient_name, gradient_size, destination_node⟩ of the tasks in its communication task queue Q_{commu} to the global *coordinator*, e.g., 'gradients a, b, c, and d to node N_1' and 'gradients e and f to node N_2' (gradient sizes omitted for clarity). Upon arrival, the coordinator places these tasks into their respective per-link task queues. Afterward, the coordinator looks up these queues and selects a set of non-conflicting links between nodes (e.g., 3 of 6 links are selected). The coordinator then batches the gradients that need to be transmitted over each selected link with balanced batch sizes, amortizing the communication overhead across gradients. The size of each batch is decided based on a specified timeout or a size threshold, whichever is met first.

Finally, the coordinator broadcasts the information of these gradient batches and coordinated communication plans to the relevant nodes (step ⑤ in Figure 2), so that the executor on each node can execute these plans in a coordinated manner and notify its task manager to clear the dependencies of the tasks in each batch accordingly (step ⑥). Altogether, our compression-aware, coordinated bulk synchronization enables both efficient batch compression on GPU and efficient communication of small gradients.

TABLE 2: Notation in selective compression and partitioning.

Notation	Interpretation
m	Gradient size in bytes
K	Number of gradient partitions
N	Number of workers or aggregators
r	Compression rate
$T_{enc}(m)$	Time for compressing an m-byte gradient
$T_{dec}(m)$	Time for decompressing an m-byte gradient
$T_{send}(m)$	Time for transmitting an m-byte gradient

TABLE 3: Synchronization parameters and their values.

	Algorithm	α	β	γ
CaSync-Ring	Spars. Quant.	$2(N-1)$	N	N
	Low-rank		1	1
CaSync-PS	Spars. Quant.	$2N$	$K+1$	$N+1$
	Low-rank		1	1

3.3 Selective Compression and Partitioning

Reducing data volume being transmitted does not always offset the compression-related overhead even with optimized synchronization strategies. It is more complicated when large gradients require partitioning to leverage parallelism before compression. Therefore, we design a *selective compression and partitioning* mechanism with a cost model to analyze the time cost of synchronizing gradients with and without compression, and then make a selective decision to avoid over-compression penalties and further leverage parallelism and load balancing. The cost model is simple yet unified, and is applicable to different CaSync synchronization strategies.

There are a few parameters used in the cost analysis as defined in Table 2. Here, the compression rate r, as well as the compression cost $T_{enc}(m)$ and decompression cost $T_{dec}(m)$, are specific to compression algorithms and can be easily profiled. Moreover, $T_{send}(m)$ denotes the network transmission time for an m-byte gradient. We omit merge operators as they are compression-irrelevant.

We first analyze the original time to synchronize an m-byte gradient with K partitions but without compression, denoted as $T^{orig}_{sync}(m, K)$. Here, we use PS and Ring-allreduce designed within CaSync as examples, denoted as CaSync-PS and CaSync-Ring. For simplicity, let N be the number of their respective workers or aggregators. We assume the common practice used in real world where all nodes are homogeneous [68], [69]. Also, the number of gradient partitions, K, is between 1 and N for both strategies, with a discussion of larger K values later. We calculate $T^{orig}_{sync}(m, K)$ as follows:

$$T^{orig}_{sync}(m, K) = \alpha \times T_{send}(\frac{m}{K}). \quad (1)$$

Here, α denotes the total number of serial communication steps for synchronizing a gradient, and its value depends on the given synchronization strategy. As shown in Table 3, the α value of CaSync-Ring is $2(N-1)$, since it takes $N-1$ steps for gradient aggregation and another $N-1$ steps to disseminate the updated gradient (see Figure 1b), and all K

gradient partitions are synchronized in parallel. Similarly, the α value of CaSync-PS is $2N$, where the communication of gradient partitions is well coordinated so that no network links used are conflicting, i.e., all aggregators run in parallel and each takes N steps to receive gradient partitions from N workers and another N steps to return results (see Figure 1a).

Next, we calculate the time, $T^{cpr}_{sync}(m, K)$, to synchronize an m-byte gradient with K partitions and compression:

$$T^{cpr}_{sync}(m,K) = \alpha \times T_{send}(r \times \frac{m}{K}) + \beta \times T_{enc}(\frac{m}{K}) + \gamma \times T_{dec}(r \times \frac{m}{K}). \quad (2)$$

Here, the α value remains the same, but the communication cost is reduced to $T_{send}(r \times \frac{m}{K})$ because one needs to send only the compressed gradient partition of the reduced size $r \times \frac{m}{K}$. This, however, comes with an extra compression-related computational cost. We denote the number of encode and decode operators that do not overlap with gradient transmission as β and γ, whose values are described in Table 3. When using CaSync-Ring to globally synchronize quantized and sparsified gradients, its first aggregation phase requires $N-1$ encode and $N-1$ decode operators, and they are non-overlapping because a node can compress a gradient partition only after it has decompressed and aggregated the partition received from its predecessor (i.e., data dependencies). Its second dissemination phase requires only one encode and $N-1$ decode operators. However, all decode operators except the last one can overlap with gradient transmission. Therefore, $\beta = (N-1)+1 = N$ and also $\gamma = (N-1)+1 = N$.

However, the application of CaSync-Ring to gradients compressed by low-rank methods behaves differently and leads β and γ to be 1. This is due to the unique characteristics of this kind of gradient compression algorithm; compressed gradients can be directly aggregated along the synchronization path, and thus only one encode operator and one decode operator are needed to compress the target gradient at the first step and decompress the aggregated compressed sub-matrices at the final step, respectively. We omit the analysis for CaSync-PS due to space limit. Note that, our cost model can be relaxed to split a gradient into beyond N partitions to leverage the compression-communication pipeline enabled by CaSync further. To do so, we simply adapt the calculation of $T^{cpr}_{sync}(m, K)$ by grouping K partitions into $\lceil \frac{K}{N} \rceil$ batches.

Based on the comparison of $T^{orig}_{sync}(m, K)$ and $T^{cpr}_{sync}(m, K)$, we decide whether it is beneficial to enable compression for a gradient. If so, we also compute the optimal number of partitions for the best performance. This is feasible because: 1) all parameters in Table 2 can be easily obtained or profiled via GPU and network measurements, where we launch the GPU kernels and peer-to-peer communication tasks with respect to different gradient sizes to fit the compression and network cost curves, respectively; 2) the values of α, β and γ in Table 3 needed to analyze $T^{cpr}_{sync}(m, K)$ are determined once a DNN system with its CaSync synchronization strategy is given, and 3) the expressions 1 and 2 are convex functions which make it straightforward to identify the best setting for each gradient. It is worth mentioning that, our cost model assumes a homogeneous environment where all GPUs and network links have the same capacities, and the profiling results are obtained without considering the variance or interference of network and GPUs. We leave the exploration of the impacts of dynamics on the profiling accuracy of our cost model as future work.

```
1 void encode(float* input, uint8* output, params);
2 void decode(uint8* input, float* output, params);
```

Fig. 4: Unified compression-related API abstraction.

Note that most, if not all, gradient compression algorithms (including the six state-of-the-art ones we evaluate) are *layer-wised*. We impose a strict partition-compress-batch order which is applied to each DNN layer independently, and thus it does not affect the accuracy and convergence of original compression algorithms. For few non-layer-wised compression algorithms, we simply turn off the selective compression and partitioning, thus incurring no negative impacts on accuracy and convergence of these algorithms.

4 Compression Library and Language

As discussed in Section 2.5, on-GPU compression can greatly accelerate compression-related computation, alleviate the bandwidth tension between GPU and host, and create new opportunities to further optimize the gradient synchronization process. However, developing and optimizing gradient compression algorithms on GPU is non-trivial, and integrating them into DNN systems usually requires substantial system expertise and manual efforts. Thus, we design a toolkit CompLL, which allows practitioners to easily develop highly-optimized compression algorithms using GPU capability. The CompLL-generated code is then consumed by CaSync, thus enabling an automated integration of compression algorithms with CaSync into DNN systems.

4.1 Unified API Abstraction

CompLL provides a unified API abstraction for implementing gradient compression algorithms. As shown in Figure 4, CompLL has two simple APIs: encode and decode, as well as a few algorithm-specific parameters (e.g., compression rate for sparsification, bitwidth or precision for quantization and rank size for low-rank). The encode API takes as input a gradient matrix and generates a compressed gradient as output. In particular, we use uint8 as the type of the output matrix, because we can then cast one or multiple uint8 to any type in CUDA. On the other hand, the decode API unfolds a compressed gradient into its original form.

4.2 Common Operator Library

By studying the state-of-the-art compression algorithms, we observe that they can generally be specified using a few common operators [5], [12], [18], [44], [76], [84], [85], [87], [89], [94], [96], [103]. For instance, these algorithms all need to scan the elements of a gradient. Alongside scanning, they all need to perform operations such as filtering or reducing the scanned elements to produce compressed gradients. With this observation, we generalize a library of common operators that can be used to construct gradient

TABLE 4: List of common operators. G is a gradient matrix, and o is the rank size required by low-rank methods.

Operator	Interpretation
sort(G, *udf*)	Sort elements in G w.r.t the order given by the user-defined function *udf*
filter(G, *udf*)	Select elements from G via *udf*
map(G, *udf*)	Return H where $H[i] = udf(G[i])$
reduce(G, *udf*)	Return a reduced value of G via *udf*
random(a, b)	Return a random int/float in [a,b)
concat(a,$\cdots$)	Concatenate values together into a vector
extract(G')	Extract metadata from the compressed G'
decomp(G, o)	Return P, Q such that $PQ^\top \approx G$, where $G \in \mathbb{R}^{n\times m}$, $P \in \mathbb{R}^{n\times o}$, and $Q \in \mathbb{R}^{m\times o}$
matmul(G_0, G_1)	Matrix multiplication of G_0 and G_1

compression algorithms, as listed in Table 4. For instance, the `reduce(G, maxAbs)` operator with a user-defined function `maxAbs` computes the maximum absolute value of the gradient matrix G, and the `decomp(G, o)` operator decomposes G into two smaller matrices with their sizes properly calculated. We have carefully optimized these common operators regarding memory access and bank conflicts in GPU [27], so that any algorithm implementation based on these operators can automatically inherent our GPU optimizations (see details in Section 5).

4.3 Code Synthesis and Domain-specific Language

We provide two ways for practitioners to implement algorithms using `CompLL`. They can invoke our common operator library directly in their algorithm implementation. This, however, requires them to be familiar with the low-level CUDA programming. To further relieve the burden, we design a simple, C-like domain-specific language (DSL) for practitioners to easily implement their algorithms with the unified API abstraction filled with common operators, without worrying about hardware-oriented implementation and optimization. Specifically, our DSL supports basic data types such as `uint1`, `uint2`, `uint4`, `uint8`, `int32`, `float`, and `array`, as well as simple numerical computations and function calls to the common operators. Though not supported, our practice shows that it is often unnecessary to include loops in the DSL code as the iterative processing semantics have already been covered by the implementation of common operators.

To show how DSL works, we use it to implement the classic TernGrad compression [94] as an example in Figure 5. Line 1-3 specify `bitwidth` as the algorithm parameter to determine compression rate. Line 5-8 specify a user-defined function `floatToUint` to compress a float number into a `bitwidth`-sized integer. The TernGrad's logic to implement our `encode` API begins at line 9, and takes the original `gradient` as input and outputs the `compressed` gradient. Through line 11-14, the algorithm metadata which is essential for decompression is generated. At line 15, we pass the user-defined function `floatToUint` to the common operator `map` to generate the compressed gradient matrix `Q`. Finally, at line 16, we use the common operator `concat` to combine all metadata and `Q` into the output `compressed` gradient. We omit the implementation of the TernGrad's decompression code in the interest of space.

```
1  param EncodeParams{
2      uint8 bitwidth; // assume bitwidth = 2 for clarity
3  }
4  float min, max, gap;
5  uint2 floatToUint(float elem) {
6      float r = (elem − min) / gap;
7      return floor(r + random<float>(0, 1));
8  }
9  void encode(float* gradient, uint8* compressed, \
10             EncodeParams params) {
11     min = reduce(gradient, smaller);
12     max = reduce(gradient, greater);
13     gap = (max − min) / ((1 << params.bitwidth) − 1)
           ;
14     uint8 tail = gradient.size % (1 << params.
           bitwidth);
15     uint2* Q = map(gradient, floatToUint);
16     compressed = concat(params.bitwidth, tail, \
17         min, max, Q);
18 }
```

Fig. 5: TernGrad's compression logic specified using the API, common operators and DSL of `CompLL`.

TABLE 5: Comparison of implementation and integration costs (measured in lines of code) between open-source (OSS) and `CompLL`-based compression algorithms.

Algorithm	OSS		`CompLL`			
	logic	integration	logic	udf	# common operators	integration
onebit	80	445	21	9	4	0
TBQ	100	384	13	18	3	0
TernGrad	170	513	23	7	5	0
DGC	1298	1869	29	15	6	0
GradDrop	N/A	N/A	29	21	6	0
PowerSGD	307	211	15	10	2	0

Next, `CompLL`'s code generator parses the gradient compression algorithm specified in our DSL, traverses its abstract syntax tree, and automatically generates the CUDA implementation. When encountering a function call to common operators, `CompLL` directly substitutes it with our highly-optimized CUDA implementation and then converts the specified parameters into their desired formats. For other operations such as numerical computations, `CompLL` declares specified variables and copies the necessary numerical computation code accordingly, as our DSL supports a subset of C's syntax. For a variable of type (such as `uint1`) which is not supported in CUDA, `CompLL` uses a byte to store it and uses bit operations to extract the actual value. If it is an array of variables of unsupported type, `CompLL` uses consecutive bits of one or more bytes to represent this array compactly, with the minimal zero padding to ensure the total number of bits is a multiple of 8.

4.4 Case Studies and Discussions

To demonstrate the easy algorithm development enabled by `CompLL`, we use it to implement six state-of-the-art compression algorithms: onebit [76], TBQ [84], and TernGrad [94] are quantization algorithms; DGC [44] and GradDrop [5] are sparsification ones; PowerSGD [87] is a low-rank algorithm. Onebit, TBQ, TernGrad, DGC and PowerSGD have open-source (OSS) implementations.

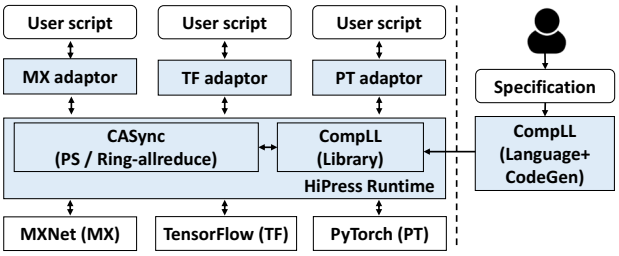


Fig. 6: The overview of HiPress. The shadow boxes are the new components introduced by HiPress.

Auto-generated code. Table 5 summarizes the comparison between the open-source and CompLL-based implementations of these algorithms. The open-source implementations need a lot more code to implement these algorithms, and spend substantial effort to integrate them into DNN systems. In contrast, with CompLL, we use only 3 to 6 common operators to implement these algorithms with fewer than 21 lines of code for user-defined functions and fewer than 29 lines of code for algorithm logic. The algorithm is then translated into GPU code via our code generator and integrated into DNN systems by CompLL without manual efforts. We leave the detailed performance evaluations in Section 6.4 to compare the encode and decode operations between CompLL's auto-generated implementations and three open-source (OSS) baselines.

Expressiveness and extensibility. Beside the classic algorithms listed in Table 5, we exercise more gradient compression algorithms and find that they all can be easily specified and auto-generated by CompLL. For instance, AdaComp [14] needs map, reduce, filter, concat and extract common operators, while 3LC [43] needs reduce, map, concat, filter and extract. As an example, it requires only 69 lines of CompLL's DSL code to express the encode function of 3LC, whose zero-run encoding logic is specified by partitioning the target gradient and applying map and filter over each partition. For future algorithms possibly requiring new operators, CompLL is open and allows registering them into the common operator library for enjoying our automated code generation and integration into DNN systems.

5 HiPress Framework

We incorporate the aforementioned coherent design into an open-source framework HiPress [2] for compression-aware data parallel DNN training. HiPress has 7.5k and 3.3k lines of code in C/C++ and Python, respectively, and is composed of the following main components, as shown in Figure 6.

5.1 Major System Components

CaSync. We implement CaSync using Horovod [78], a popular gradient synchronization library used by almost all mainstream DNN systems. CaSync currently supports both PS and Ring-allreduce. We leverage the MPI_all_to_all [56] and NCCL_all_to_all [62] primitives to execute the *bulk communication step* introduced in Section 3.2.

We deploy the global *coordinator* on one of the training nodes. Though being a centralized component, its load is always light and the coordination overhead is negligible due to the following reasons: (1) only the gradient metadata is exchanged, and (2) the coordination of one gradient batch runs asynchronously with the bulk synchronization of the previous batches, thus its cost can be always hidden (confirmed in our experiments).

The selective compression and partitioning planer is a standalone component for producing per-gradient compression and partitioning plans. It obtains the variables defined in Section 3.3 from the training scripts (including the synchronization strategy and cluster configurations), the network, and GPU-measurements via the first training iteration. The produced plans are executed by CaSync at runtime.

CompLL. We implement decompose common operator using LAPACK [37] and remaining operators using Thrust [61], the CUDA C++ template library, with the following optimizations. (1) CompLL reuses gradients produced by DNN computation and only allocates buffers for the much smaller compressed gradients to avoid the GPU memory contention. (2) CompLL uses fast share memory rather than global memory, and eliminates bank conflicts [27] by making each thread access disjoint memory banks when possible. We also fuse the decode and merge operators for better performance. To avoid computing resource contention, we allocate CUDA cores for compression-related computation, while letting DNN computation kernels make the full use of Tensor cores. Therefore, these two types of computation can run concurrently with no interference.

Local aggregation. For multiple GPUs per node, we first aggregate the original gradients among local GPUs, and then synchronize the compressed gradients across nodes. This is because the bandwidth of intra-node connection links (e.g., PCIe, NVLink) is often orders of magnitude higher than the inter-node links. Local aggregation reduces the number of gradients exchanged across nodes for better performance.

5.2 Training Integration and Execution Scheduling

HiPress integrates CaSync and CompLL-generated library into three modern DNN systems TensorFlow, MXNet, and PyTorch. First, CaSync is integrated via Horovod. CompLL creates wrapper functions for encode and decode primitives to obtain pointers to gradients and the algorithm-specific arguments from the training context. CompLL then invokes the CompLL-generated code. Second, we create adaptors to make training workflows compression-enabled by instrumenting the original training scripts with function calls to CaSync. The major challenge we identify here is the mismatch between the existing execution models with coarse-grained dependency tracking mechanisms in DNN systems and the new computation-compression-communication pipeline HiPress introduces. To illustrate the problem and our system adaptations, we consider the following two cases.

MXNet and TensorFlow. The two systems have their own task manager to overlap the gradient synchronization and

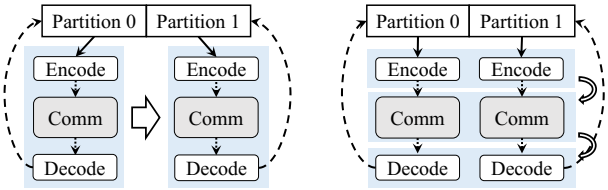


Fig. 7: Ordering constraint optimization. Solid-line/dashed-line arrows represent read/write dependencies, respectively, and other arrows represent causal dependencies.

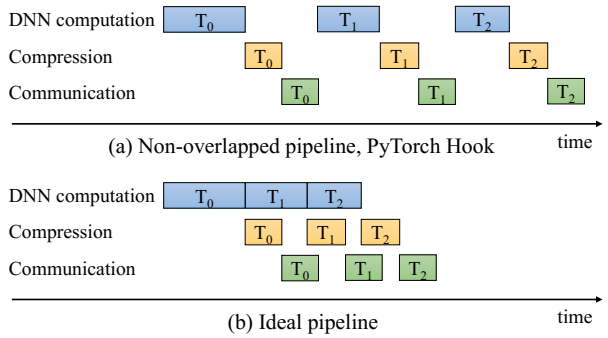


Fig. 8: The new DNN computation, compression, and communication pipeline in HiPress.

DNN computation. We add a new task queue and a dedicated CPU thread to schedule *encode* and *decode* operators on GPU for further running compression-related computation and DNN computation in parallel, as we observe that the former computation is often memory-intensive. However, the asynchronous execution engines open a challenge, especially when considering gradient partitioning in the context of compression. As described in Section 3.3, our CaSync synchronization may partition gradients before compression. To avoid expensive memory copying, instead of allocating temporary spaces for partitioned gradients, we use offsets pointing to the original gradient in GPU to mimic partitioning. Without a proper design, however, we could impose unnecessary ordering constraints on the synchronization of all partitions. This is because the dependency tracking in existing DNN systems is coarse-grained and not aligned with our memory copying avoidance.

Figure 7 illustrates this problem. On the left side, the synchronization-related operations (i.e., encode, comm (communication) and decode) execute sequentially for a partition. Partition 0's decode has a write dependency on the shared gradient, while the Partition 1's encode imposes a read dependency on the same gradient despite of different partitions. This leads to a situation where the synchronization of the two partitions is serialized (unnecessarily). Instead, on the right side, we divide the synchronization into stages, group operations in one stage, and then pass them to the next stage. This meets proper ordering constraints while allowing for pipelining the synchronization across different partitions of the same gradient.

PyTorch. Though the asynchronous execution is also supported, PyTorch does not rely on a task manager, instead, it executes gradient synchronization in hook [52]. However, directly calling CaSync in hook would lead to performance degradation as it precludes the opportunity of overlapping computation and communication, as shown in Figure 8a. This is because the main process can launch the next DNN computation operator as long as the operators in hook, such as encode, bulk communication, and decode operators, complete their execution.

To address the hook issue, we implement a task manager inspired by the execution engine of MXNet and TensorFlow to enable the new computation-compression-communication pipeline. This engine employs multiple queues to manage different tasks, each assigned a dedicated thread for task launching. In addition, we apply the above fine-grained dependency tracking optimization that works with gradient partitioning to PyTorch. Finally, the new pipeline allows the DNN computation to overlap with both compression and communication, as depicted in Figure 8b.

5.3 Choosing CaSync-PS or Ring?

Here, we discuss the performance behaviors of the two compression-enabled gradient synchronization methods offered by CaSync across the three categories of compression algorithms. First, for the low-rank algorithms, CaSync-PS has the same performance as CaSync-Ring, due to the following reasons. On one hand, according to Table 3, CaSync-PS takes the same number of transmission, compression and aggregation steps as CaSync-Ring. This is as expected since existing studies [33], [35] including ours prove that their non-compression baselines are semantically equivalent. Furthermore, when compression is enabled, at every intermediate step, compressed gradients are directly aggregatable, and there is no need to perform *encode* and *decode* operators. On the other hand, we have implemented CaSync-PS efficiently, which follows the common practice of Ring-allreduce to make use of high performance collective communication primitives. Second, regarding the quantization algorithms, the performance of CaSync-PS is also similar to that of CaSync-Ring, due to the same reasons above.

Finally, contrary to quantization and low-rank algorithms, the two methods can deliver different performance numbers for sparsification ones, mainly because of the data inflation when aggregating gradients. In more detail, upon receiving compressed sparse gradients, often containing top K important elements, the *worker* or *aggregator* merges them to form updated gradients, which will contain top K' elements, where $K' >= K$. This inflation would lead to increases in transmission latency and bandwidth consumption. However, this impacts CaSync-PS less heavily than CaSync-Ring due to the following reasons. CaSync-PS is implemented by the combined use of gather and broadcast, and the inflation only affects the performance of broadcast. As the gradients are partitioned, and balanced across *workers*, the impacts are not heavy. Dislike this, for CaSync-Ring, inflation happens at every step in its aggregating phase (in total, N-1 steps). To make it comply with the Ring-allreduce semantics, which requires gradients to be same sized during all steps, we have to add an extra encode operator following the merger operator to re-select top K elements. However, this incurs additional computation overhead, compared to CaSync-PS. Note that we

TABLE 6: Statistics of trained models.

Name	Total size	Max gradient	# Gradients
VGG19 [83]	548.05MB	392MB	38
ResNet50 [29]	97.46MB	9MB	155
UGATIT [34]	2558.75MB	1024MB	148
UGATIT-light [34]	511.25MB	128MB	148
Bert-base [21]	420.02MB	89.42MB	207
Bert-large [21]	1282.60MB	119.23MB	399
LSTM [50]	327.97MB	190.42MB	10
Transformer [86]	234.08MB	65.84MB	185

TABLE 7: Compression and partitioning plans of `CompLL-onebit`. In each tuple, the first value decides whether to compress a gradient and the second value indicates the number of partitions.

Gradient size	`CaSync-PS`		`CaSync-Ring`	
	4 Nodes	16 Nodes	4 Nodes	16 Nodes
4MB	<yes, 2>	<yes, 1>	<yes, 1>	<no, 16>
16MB	<yes, 4>	<yes, 6>	<yes, 4>	<yes, 5>
392MB	<yes, 12>	<yes, 16>	<yes, 4>	<yes, 16>

can only observe visible performance differences between `CaSync-PS` and `CaSync-Ring` when the sparsity rates are considerably low. We leave the further exploration and system optimizations in the future work.

6 Evaluation

Our evaluation answers the following main questions:

- Can `HiPress` significantly improve the performance of DNN data parallel training jobs over the baselines?
- What are the performance implications of synchronization optimizations and the auto-generated compression code?
- What are the effects of compression rate and network bandwidth?
- Can `CompLL`-generated compression algorithms outperform their open-source ones?
- Can `CompLL`-generated compression algorithms achieve the same training accuracy as their original versions?

6.1 Experimental Setup

Machine configurations. We conduct experiments in both AWS EC2 and local clusters to evaluate `HiPress` with both high-end and low-end machines. We use 16 p3dn.24xlarge EC2 instances with 128 GPUs. Each instance has 96 vCPU, 8 NVIDIA Tesla V100 GPUs (32GB memory, connected by NVLink), and is connected by a 100Gbps network. We also replicate the same experiments in our local cluster with 16 nodes and 32 GPUs. Each local node has two 16-core Intel E5-2620 processors, 2 NVIDIA 1080 Ti GPUs (connected via a PCIe switch), and is connected by a 56Gbps Infiniband network. EC2 instances and local nodes run Ubuntu 16.04 and CentOS 7.6, respectively, with the remaining software being identical, such as CUDA 11.1, OpenMPI 4.0.3, NCCL 2.10.3, MXNet 1.5.1, TensorFlow 1.15.5, PyTorch 1.10.0, Horovod 0.20.3 and BytePS 0.2.5.

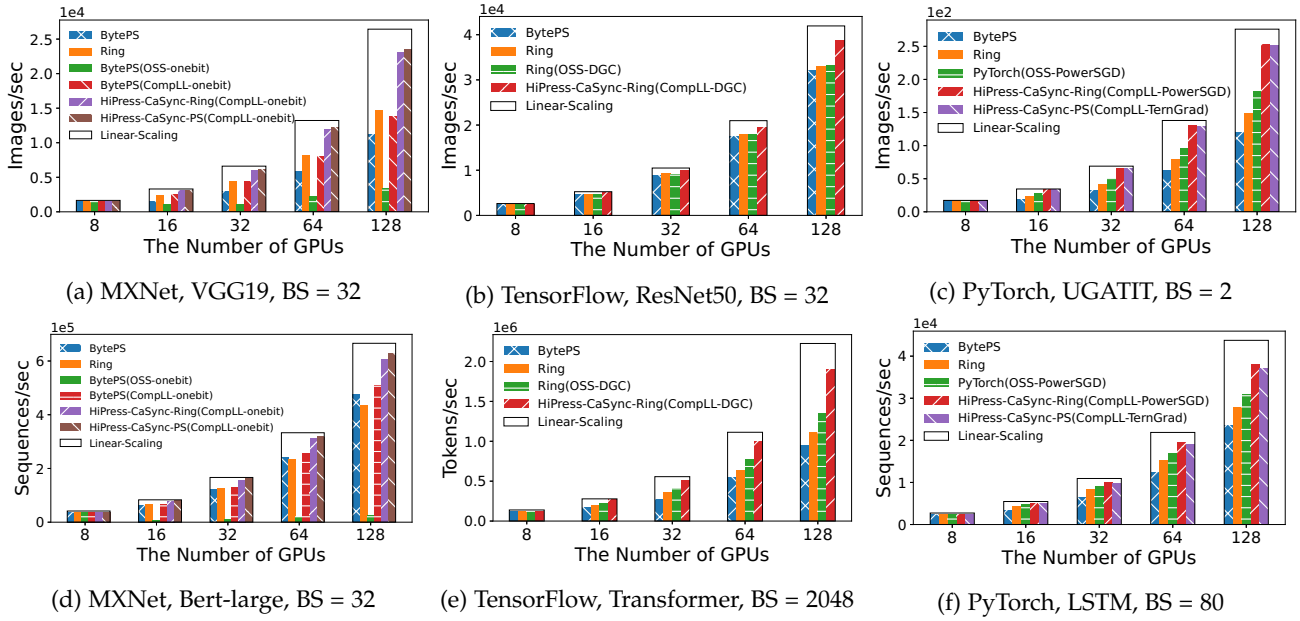
Baselines. We use TensorFlow (TF), MXNet, PyTorch with `BytePS` and Ring-allreduce (`Ring`) as no-compression baselines. In the interest of space, we only demonstrate the end-to-end performance with four out of six generated compression algorithms, namely, onebit, DGC, TernGrad and PowerSGD, with different DNN systems. We use the recently developed `BytePS(OSS-onebit)` [6], [13], `Ring(OSS-DGC)` [66] and `PyTorch(OSS-PowerSGD)` [54] from industry as compression-enabled baselines with open-source (OSS) quantization, sparsification and low-rank algorithms respectively. Note that for a fair comparison, we use our highly optimized on-GPU implementation instead of the original on-CPU implementation for onebit algorithm denoted by `CompLL-onebit`.

Models and datasets. Following the literature [33], [75], we choose six widely-used DNN models with three computer vision (ResNet50, VGG19 and UGATIT) and three natural language processing (Bert, Transformer and standard-LSTM). We train ResNet50 and VGG19 with the ImageNet dataset [74], and the remaining models with the selfie2anime [73], RTE [11], WMT17 [77] and wikitext-2 [49] dataset, respectively. We additionally deploy Bert and UGATIT under their light mode with fewer parameters to meet the GPU memory constraint in our local cluster, denoted as Bert-base and UGATIT-light, respectively. The model details are summarized in Table 6.

Metrics. We measure the total number of samples processed per second as the training throughput, the latency breakdown of the key steps in the computation-synchronization pipeline, speedups of `encode` and `decode`, and the training accuracy and convergence speed.

System configurations. We tune the configurations of baselines for their best performance, e.g., co-locating aggregators and workers for BytePS and `CaSync-PS`. We deploy all systems with RDMA enabled except BytePS on EC2. This is because BytePS does not support the Elastic Fabric Adapter (EFA) used by EC2 instances at the moment. We keep the per-GPU batch size constant as the number of GPUs are scaled up (*weak scaling*). We set batch sizes across different models by following literature [21], [36], [50], [86], instead of setting them to the largest value that a single GPU can sustain, since a too large batch size may lead to convergence problems [47], [75]. For all four compression algorithms, we inherit the parameter settings from their original papers.

Table 7 shows the optimal thresholds for compressing a gradient and the optimal partition numbers, produced by `CaSync` based on `CompLL-onebit` algorithm. According to two synchronization strategies `CaSync` currently supports and their cluster deployment configurations, we set the value of α, β and γ for `CaSync-PS` as $2(N-1)$, K and N, respectively. This assignment is slightly different from the numbers in Table 3. This is because the evaluated `CaSync-PS` in Section 6 co-locates aggregators and workers, and the local workers do not need to send its gradients to the co-located aggregator via network activities. For `CaSync-Ring` against sparsification and quantization algorithms, we set three parameters as $2(N-1)$, N, and N, while for `CaSync-Ring` against the PowerSGD algorithm, we set three parameters as $2(N-1)$, 1, and 1 respectively. The optimal thresholds of selective compression and partition sizes are produced by our cost analysis model. With 16 nodes, `CaSync` suggests compressing gradients larger than 4MB and splitting the largest VGG gradient into 16 partitions before compression for AWS EC2 platform.



(a) MXNet, VGG19, BS = 32 (b) TensorFlow, ResNet50, BS = 32 (c) PyTorch, UGATIT, BS = 2

(d) MXNet, Bert-large, BS = 32 (e) TensorFlow, Transformer, BS = 2048 (f) PyTorch, LSTM, BS = 80

Fig. 9: Throughput comparison of image processing and natural language processing models atop MXNet, TensorFlow, and PyTorch. AWS EC2 V100 instances. 100Gbps cross-node RDMA network. BS stands for per-GPU batch size.

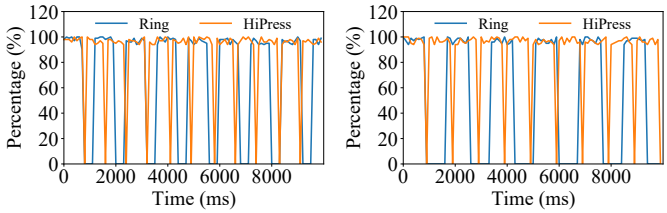
In addition, for the baseline synchronization strategies like BytePS and Horovod (Ring-allreduce enabled), we use their default settings. These baselines are able to tune the number of partitions for maximizing the overall performance [33], [63].

6.2 End-to-End Performance

6.2.1 AWS EC2 Results

Figure 9 compares the end-to-end training throughput of `HiPress` and baselines with `MXNet`, `TensorFlow` and `PyTorch` as the underlying DNN system, respectively, using a total of 128 GPUs.

Atop MXNet. We demonstrate the throughput comparison results using MXNet in Figure 9a and 9d. For the VGG19 model, `Ring` outperforms `BytePS` by 31.3-50.3% across all cluster sizes. When not using RDMA, `Ring` still outperforms `BytePS` by 19.3-36.6%. These results are not consistent with the `BytePS` paper, but valid. This is due to we use a newer version of NCCL library that both `BytePS` and `Ring` relies on, and we also disable intra-node aggregation in `Ring`, which leads to better performance. For the Bert-large model, `BytePS` outperforms `Ring` by up to 8.9% across all cluster sizes. However, enabling the onebit compression algorithm within `BytePS` does not bring expected speedups. With the native on-CPU onebit implementation, `BytePS(OSS-onebit)` performs the worst, and runs far slower than all other system configurations. When migrating compression-related computation from CPU to GPU, `BytePS(CompLL-onebit)` catches up with non-compression baselines, but brings only limited improvements over the best-performed one, e.g., only up to 7.3% improvement over `BytePS`. Such surprising result verifies the importance of implementing fast compression algorithms on GPU and designing a compression-aware synchronization strategy to release the full potential of compression algorithms.



(a) Bert-large (b) UGATIT

Fig. 10: GPU utilization of `Ring` and `HiPress` when training Bert-large and UGATIT. The configurations of `HiPress` are the same as those used in Figure 9d and 9c.

Unlike limited speedups brought by the latest synchronization strategies and open-source versions of compression algorithms, `HiPress` significantly improves the training throughput over all baselines across all cases. E.g., with 128 GPUs, for VGG19 and Bert-large, `HiPress-CaSync-PS(CompLL-onebit)` outperforms `BytePS`, `Ring` and `BytePS(CompLL-onebit)` by 110.5% and 32.3%, 60.4% and 44.1%, 69.5% and 23.3%, respectively. `HiPress-CaSync-Ring(CompLL-onebit)` performs similarly to `HiPress-CaSync-PS(CompLL-onebit)`, and also significantly outperforms all baselines. One important observation is that the improvements of `HiPress` become larger when the number of GPUs increases. This implies that when the cluster size expands, the communication overhead of the communication-intensive models increases, and thus `HiPress` becomes even more beneficial.

Atop TensorFlow. We evaluate the integration with TensorFlow using the ResNet50 and Transformer models. In Figure 9b, the non-compression `BytePS` and `Ring` perform similarly for ResNet50. In contrast, for Transformer, `Ring` outperforms `BytePS` by up to 30.9% and 23.5%, when switching on/off RDMA. Transformer's scaling efficiency is

significantly lower than that of ResNet50, since it is more communic-ation-intensive and exchanges more gradients than ResNet50.

Note that BytePS(OSS-onebit) cannot be directly applied to TensorFlow, since it is tightly coupled with MXNet. Thus, we exercise DGC, integrated into Ring-allreduce and TensorFlow. To compare with Ring(OSS-DGC), we configure HiPress with CaSync-Ring rather than CaSync-PS. For the Transformer model, Ring(OSS-DGC) outperforms BytePS and Ring by up to 42.8% and 22.1%, respectively, though brings almost no improvement for ResNet50. Because of the optimized compression-aware synchronization strategy design and the highly-efficient on-GPU DGC code generated by CompLL, HiPress-CaSync-Ring(CompLL-DGC) outperforms Ring(OSS-DGC) by up to 41.1%, and the non-compression baselines such as BytePS and Ring by up to 101.4%, for Transformer. Interestingly, even for ResNet50, HiPress improves its training speed by up to 20.7% over all baselines. This implies that when the cluster size expands, the communication cost of the computation-intensive models also increases, and can benefit from HiPress.

Atop PyTorch. Here, we exercise the UGATIT and LSTM models. Here, we choose PyTorch(OSS-PowerSGD) as the compression-enabled baseline, while BytePS and Ring as the non-compression ones. In Figure 9c and 9f, similar to the results of HiPress atop both MXNet and TensorFlow, HiPress over PyTorch with CaSync-PS the CompLL-TernGrad algorithm obtains a speedup up to 2.1 × compared to BytePS and Ring, for UGATIT and LSTM. Such consistent results verify that HiPress is a general and high performance compression-aware data parallel framework.

Though the two models are communication-intensive, the open-source PyTorch(OSS-PowerSGD) only outperforms the best-performed Ring baseline by up to 23.3% and 11.6% respectively. This is mainly due to the sequential execution introduced by the hook mechanism (see Section 5.2) and the lack of local aggregation in the native communication library of PyTorch. In contrast, HiPress-CaSync-Ring(CompLL-PowerSGD) further brings 22.9%-39.3% and 9.8%-22.7% improvement for the two DNN models across all setups with different numbers of GPUs in-use, compared to PyTorch(OSS-PowerSGD), thanks to its fine-grained task management, local aggregation optimization and high performance auto-generated PowerSGD operators.

GPU utilization. Figure 10 compares the GPU resources used for the DNN-related computation of the non-compression baseline Ring and the best-performed HiPress configurations (Figure 9d and 9c). Here, we use nsight instead of nvidia-smi to measure the GPU utilization of training jobs, since the latter does not distinguish the GPU resources used for the DNN computation and gradient synchronization. For the Bert-large and UGATIT model, both Ring and HiPress can use nearly 100% GPU computing resources at the peak. However, the overall GPU usage of Ring is more sparse than HiPress. This is because Ring's GPU utilization drops to zero during gradient transmission, which is time-consuming in data parallel training. However, within HiPress, the fast compression-aware gradient synchronization eliminates the communication bottleneck, which leads the system to spend more time doing useful work.

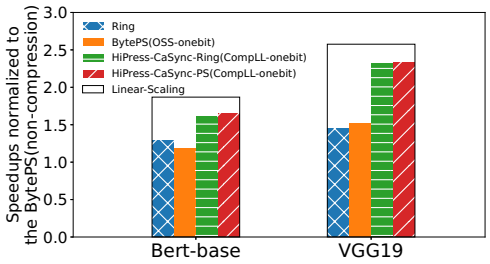


Fig. 11: Training speedup normalized to BytePS atop MXNet system, in a 16-node local cluster connected via 56Gbps Infiniband network with RDMA enabled.

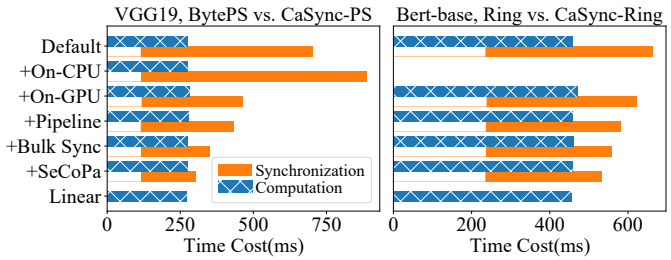
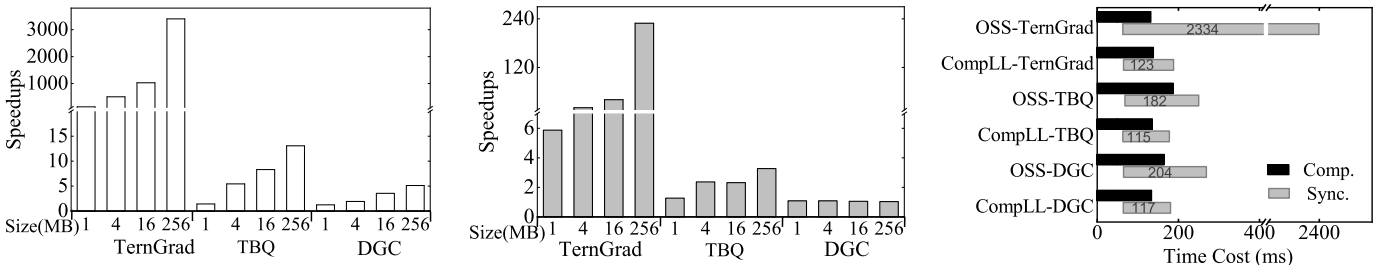


Fig. 12: Impacts of enabling synchronization optimizations on the time cost of computation and synchronization.

6.2.2 *Local Cluster Results.*

In addition to the high-end servers, low-end clusters with earlier generations of GPUs than V100 and commodity networks slower than the 100Gbps links in AWS have been still in widespread use [8], [23], [41], [42]. To test the applicability of HiPress to low-end servers, we replicate all above experiments in our local cluster with low-end GPUs and RDMA-enabled 56Gbps network links. Similar to the performance improvements of HiPress in the high-end AWS cluster, in our local cluster tests, the combinations of two CaSync synchronization strategies and various CompLL-generated algorithms in HiPress significantly outperform all baselines, and HiPress's performance advantages become more obvious with more GPUs. In the interest of space, we only show the performance speedups of all system configurations when training Bert-base and VGG19 over MXNet, normalized to the non-compression baseline BytePS, see Figure 11. We use the onebit algorithm to reduce the transmitted data volume like in Figure 9a and 9d. Due to the GPU memory constraint, we run Bert-base, a light variant Bert with fewer parameters. With 16 nodes and 32 GPUs, for both communication-intensive models, surprisingly, the state-of-the-art compression-enabled baseline BytePS(CompLL-onebit) runs even 8.5% slower than the non-compression Ring. By contrast, HiPress outperforms the non-compression baselines (i.e., BytePS and Ring) and the compression-enabled baseline BytePS(CompLL-onebit) by up to 133.1% and 53.3%, respectively. Thus, HiPress could benefit training jobs with diverse software/hardware configurations, as long as the communication is the bottleneck.



(a) Speed of *encode* (b) Speed of *decode* (c) Training speed using `HiPress`

Fig. 13: Speeds of `CompLL-TBQ`, TernGrad, and DGC normalized to open-source baselines and training performance effects. The subfigure (c) represents the timeline diagram of the DNN computation (Comp.) and gradient synchronization (Sync.) pipeline. The pipeline is shorter, and the performance is better.

6.3 Effectiveness of Various Optimizations

Next, we evaluate the individual performance gains of various synchronization and compression optimizations we introduced. We report the latency breakdown when enabling optimization one by one for training VGG19 and Bert-base across 16 local nodes in Figure 12 (the AWS results look similar and thus are not shown here). We use `HiPress(CompLL-onebit)` as an example with the same setup as Figure 11 (results using other algorithms look similar). We synchronize gradients of VGG19 via `CaSync-PS`, and Bert-base via `CaSync-Ring`. *Default* are baselines where the state-of-the-art `BytePS` or `Ring` is used without compression.

`CompLL` auto-generation. Compared to *Default*, surprisingly, directly using the open-source on-CPU onebit (denoted as on-CPU) results in 32.2% more gradient synchronization cost for `BytePS` on VGG19. This is because the overhead of on-CPU compression operators largely exceeds the communication savings. However, this does not apply to Bert-base since `Ring` uses GPU and does not work with on-CPU compression. In contrast, our `CompLL-onebit` (denoted as on-GPU) reduces the synchronization cost by 41.2% and 10.0% for VGG19 and Bert-base, respectively. We also observe that on-GPU `CompLL-onebit` imposes negligible negative impact on the local DNN computation time, even though they share GPU.

Pipelining. Compared to *on-GPU*, *pipelining* compression and communication in `CaSync` further improves the synchronization performance of VGG19 and Bert-base by 7.8% and 10.6% respectively. This is because: (1) the conventional Ring-allreduce precludes pipeline, and (2) although BytePS enables pipelining, it incurs multiple extra memory copies, which are eliminated by `CompLL`'s memory-centric optimizations.

Bulk synchronization. Our compression-aware bulk synchronization in `CaSync` achieves 26.1% and 6.6% further synchronization performance improvements for VGG19 and Bert-base, respectively. This is because our bulk synchronization approach improves the network utilization, promotes parallel compression, and reduces the overhead of small tasks. The improvement on VGG19 is higher than Bert-base because BytePS does not coordinate data transmission while Ring-allreduce does.

Selective compression and partitioning. Judicious compression and partition decisions (denoted as SeCoPa) further reduces the synchronization cost of VGG19 and Bert-base by 19.9% and 7.4%, respectively. Bert-base benefits more from *selective compression* since 62.7% of its gradients are below 16KB, where the over-compressing penalties are eliminated. VGG19 contains a few large gradients (the largest is 392MB), and thus fine-grained partitioning leads to significant performance boosts. When all the four optimizations are stacked up, `HiPress` pushes the scaling efficiency of training VGG19 and Bert-base up to 0.90, which is 133.1% and 28.6% higher than the two *Default* baselines, respectively.

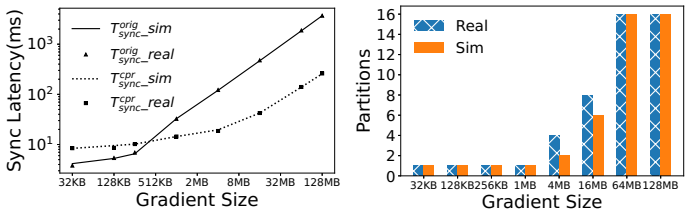
6.4 Importance of Compression Speed

Figure 13a and 13b compare the speed of the *encode* and *decode* operations between our auto-generated implementations and open-source baselines within our local cluster. Our implementations achieve constantly lower latency for compressing or decompressing gradients than baselines. For instance, the *encode* of `CompLL-TBQ` runs over 12× faster than `OSS-TBQ`, which needs to take 38.17ms to compress a $256MB$ gradient. `CompLL-TernGrad` achieves up to an impressive 3400 (229) × speedup for compressing (decompressing) gradients compared to the open-sourced CPU implementation. Even though `OSS-DGC` is highly tuned, our auto-generated `CompLL-DGC` still significantly outperforms the *encode* of `OSS-DGC` by up to 5.1×.

Faster compression/decompression speed has positive impacts on the computation-synchronization pipeline and thus on the overall end-to-end training performance. Figure 13c summarizes the pipeline breakdown latencies for training VGG19 across 16 machines using both open-source version and our auto-generated version of TBQ, TernGrad and DGC within `HiPress`. Other experimental setup is the same as Figure 11. Our auto-generated TernGrad, TBQ, and DGC reduces the synchronization cost by 94.7%, 36.8% and 42.6%, respectively, compared to their open-sourced counterparts. In addition, our faster TBQ and DGC implementation also alleviates the tension between compression and training computation at workers, e.g., they reduce the local computation time by 27.6% to 18.4%. All these benefits are reflected on the overall training speedup by up to 13.4×.

6.5 Discussion of Other Factors

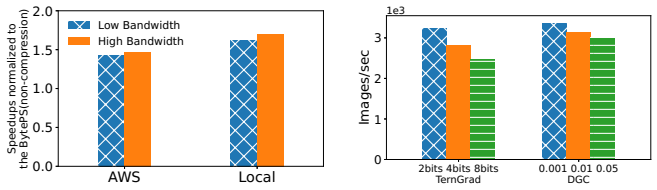
Cost model accuracy. Figure 14a compares the gradient synchronization latencies with or without compression sim-



(a) Latency comparison with $K = 1$.

(b) Comparison of the recommended partition number K.

Fig. 14: The accuracy investigation of the cost model presented in Section 3.3. The underlying compression algorithm is `CompLL-TBQ`. Figures 14a and 14b compare gradient synchronization latencies and optimal partition numbers simulated by the cost model or profiled. The parameters to drive the cost model are: $N = 16$, $r = 1/16$, and m varies from 32 KB to 128 MB.



(a) Diff. network bandwidth

(b) Diff. compression rates

Fig. 15: Training performance comparison using different network bandwidth and compression rates. Figure 15a and 15b use Bert-base and VGG19, respectively.

ulated by the cost model against the numbers measured in real experiments. To do so, we first profile the time costs of $T^{orig}_{sync}\_real$ and $T^{cpr}_{sync}\_real$ by synchronizing m-sized gradients via `CaSync-PS` and `CaSync-PS (CompLL-TBQ)` in our local 16-node cluster, connected via 56Gbps RDMA, with one GPU on each physical server. The compression rate r is $1/16$, while we assume gradients are not partitioned (K is 1) for simplicity. Then, we simulate latency numbers, denoted as $T^{orig}_{sync}\_sim$ and $T^{cpr}_{sync}\_sim$, by feeding Equations 1 and 2 with profiled T_{enc}, T_{dec} and T_{send} values, following the instructions presented in Section 3.3. The results highlight that our cost model can accurately predict the gradient synchronization time costs, and reveal that compression brings performance benefits only for gradients equal to or beyond 1MB, when partitioning is not enabled. Below that threshold, the original synchronization strategy performs sufficiently good and the compression-related cost becomes non-negligible.

Figure 14b further investigates the accuracy of our cost model for suggesting the optimal number of partitions in conjunction with compression. Our cost model suggests not to partition gradients equal to or below 1MB when compression is enabled, while using 16 partitions for 64MB and 128MB gradients, consistent with the real results produced by profiling. For 4MB and 16MB gradients, our cost model behaves slightly aggressive, and suggests a more fine-grained partitioning. This gap leads to a performance loss of less than 5% in the synchronization time cost, but we still observe visible gains, compared to the non-compression baseline.

Pipeline validation. Here, we explore if the compression-related cost can be hidden in the new pipeline we introduced in Section 5.2. To this end, we use Nsight to produce execution traces for training the VGG19 model with the `CompLL-PowerSGD` compression enabled across two physical GPU servers for simplicity. We use the PyTorch-native TorchDDP [54] synchronization library as the baseline without pipeline optimization, and compare it against `HiPress` with `CaSync-PS`. The raw traces, and detailed configurations and analysis are presented in the repo [3]. To summarize, combining TorchDDP and `CompLL-PowerSGD` would lead communication, compression, and DNN computation to be serialized. For instance, a single training iteration within TorchDDP takes 904ms, roughly equal to the sum of the time cost of DNN computation (168ms), gradient communication (299ms), compression (148ms), and bubbles (266ms). In contrast, within `HiPress` DNN computation takes the same time cost, but gradient synchronization becomes 62% faster while compression-related computations are almost all overlapped with DNN computation and communication. We also replicate the same experiments with MXNet and obtain the same conclusion that the new pipeline within `HiPress` is able to amortize the compression-related costs and improve compression-enabled, small-gradient-dominating synchronization.

Impacts of network bandwidth. Figure 15a compares the performance of training Bert-base model using `HiPress` with identical GPU configurations but two different networks. For EC2 instances, we use 100Gbps and 25Gbps as the high and low bandwidth networks, while 56Gbps and 10Gbps for local nodes. `HiPress-CaSync-PS(CompLL-onebit)` delivers similar speedups when using different networks in both 16-node EC2 and local clusters (`CaSync-Ring` has similar trends). Thus, `HiPress` can achieve near-optimal performance without expensive investment on high-end/specialized networks.

Impacts of compression rate. In Figure 15b, we compare the throughput of TernGrad and DGC algorithms generated by `CompLL` on VGG19 using `CaSync-PS` with the same setup as Figure 11. For TernGrad, when increasing `bitwidth` from 2 to 4 and 8-bit, the speedup achieved by `HiPress` decreases by 12.8% and 23.6%, respectively. As the compression cost remains the same with different precisions, the performance drops are mainly due to the increasing data communication volumes. Varying the compression rate of DGC from 0.1% to 1% and 5% also results in a performance drop of 6.7% and 11.3% respectively, due to the increasing compression and data communication cost. This implies that `CaSync` still enables fast compression-aware gradient synchronization even with lower gradient size reduction.

Convergence validation. We conduct the convergence validation experiments in our local cluster with 16 nodes, 32 1080Ti GPUs and 56Gbps RDMA network. We report the convergence results in Figure 16, which shows that `HiPress-CaSync-Ring(CompLL-DGC)` and `HiPress-CaSync-PS(CompLL-Ter-nGrad)` converge to almost the same perplexity or accuracy for LSTM and ResNet50 as no-compression baselines but with up to 28.6% less time.

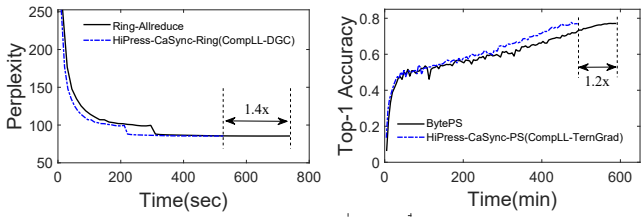


Fig. 16: Convergence time of LSTM (left) and ResNet50 (right). The target perplexity for LSTM is 86.28 and the target accuracy for ResNet50 is 77.11% [24], [25].

7 Related Work

Other than gradients compression, there are other approaches aiming at addressing the communication bottleneck in data parallel training, such as using RDMA [101], adopting Ring-allreduce [9], [10], [32], co-designing gradient synchronization with the physical topology [40], [45], and priority-based scheduling [28], [31], [69]. Blink generates optimal communication primitives [88], and BytePS uses spare CPU and bandwidth resources in the cluster and has already incorporated some of the above optimizations [33]. However, they are all compress-ion-agnostic approaches, and some of them rely on high-end networks. In contrast, HiPress enables fast compression-aware data parallel training via software innovations, and can be combined with most existing techniques.

Some recent works optimize specific gradient compression. Poseidon [104] synchronizes sufficient factors, which are compressed forms of gradients of fully connected layers in CV models. Parallax [35] focuses its optimization on sparse gradients, and shows superior performance when training NLP models where sparse gradients dominate. We significantly differ from these works by targeting at general gradient compression algorithms for any DNN models. Grace [99] studies the impacts of gradient compression algorithms, but it does not study nor address the system challenges for alleviating the tension between performance gains and programming overheads. Accordion dynamically sets compression rates to balance accuracy and performance [4], which can be employed by HiPress as an advanced feature.

There are some work share the same goal as CaSync to accelerate the speed of synchronizing compressed gradients [22], [59], [71], [81], [82]. However, almost all of them focus on the sparsification algorithms. In contrast, CaSync is more general and covers algorithms ranging from sparsification, quantization, to low rank, due to its decoupled design from the compression algorithm logics. Furthermore, most of these proposals are complementary to CaSync and their specific optimizations can be easily incorporated into CaSync for better performance.

Similar to HiPress, a few recent approaches aim to address the system challenges when incorporating gradients compression into distributed DNN training. For instance, ByteComp [92] and OMGS-SGD [80] formulate some optimization problems to improve the new compression-enabled training pipeline, e.g., merging gradient sparsification, tuning parameters for selective compression strategies, offloading compression to CPU, etc. DeepReduce [100] adopts sparse tensors organized by either keys or values, and applies different compression strategies to them. EmbRace [39] focuses on NLP models with large embedding tables. However, most of them neglect the development and integration effort of various compression algorithms and the cross-layer optimization opportunities, which have been, however, managed by HiPress. Furthermore, some optimization directions are orthogonal with HiPress and can be used to further extend HiPress's applicability.

Model Parallelism [17], [79] and *Pipeline Parallelism* [58] are often combined with *Data Parallelism* for large-scale deployment [90], [91], which can benefit from HiPress. Although HiPress focuses on Bulk Synchronous Parallel (BSP) in this paper given its wide adoption [35], [57]. HiPress is expected to work with other synchronization methods such as ASP [26] and SSP [30], [93], [98]. Finally, some components in HiPress are inspired by other works, such as dependency graph is inspired by Daydream [105], and fine-grained task management is inspired by MonoTasks [65].

8 Conclusion

Driven by CaSync and CompLL HiPress addresses the fundamental tensions imposed by gradient compression. CaSync innovates a general, composable, and adaptive gradient synchronization architecture that is compression-aware. CompLL facilitates an easy development of highly-optimized on-GPU gradient compression and an automated integration into modern DNN systems with minimal manual efforts. HiPress is open-sourced, and achieves a scaling efficiency of up to 0.92 and a training speed improvement up to 110.5% over the state-of-the-art baselines across six popular DNN models in a cluster of 16 nodes with 128 NVIDIA V100 GPUs and 100Gbps network.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the Open Fund of PDL under Grant No.: WDZC20215250115, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006, the University Synergy Innovation Program of Anhui Province under Grant No.: GXXT-2022-045, and National Science Foundation under Grant No.: CAREER-2048044. We also thank the technical support from Shanghai HPC-NOW Technologies Co., Ltd.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of OSDI*, volume 16, pages 265–283, 2016.

[2] USTC ADSL. Code of HiPress. https://gitlab.com/hipress/hipress, 2021. [Online; accessed Feb-2023].

[3] USTC ADSL. Overlapping Profiling Results of HiPress. https://gitlab.com/hipress/hipress-overlapping-profiling-results, 2023. [Online; accessed Feb-2023].

[4] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. Accordion: Adaptive gradient communication via critical learning regime identification, 2020.

[5] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint arXiv:1704.05021*, 2017.

[6] Amazon. Gradient Compression in AWS. https://docs.google.com/presentation/d/1Dt1Sh2ixVF8Or_Q3lzUM81F4Thj5LT8Xw6QjU1e6iwQ/edit#slide=id.p, 2021. [Online; accessed Feb-2023].

[7] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015.

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

[9] Baidu. Bringing HPC Techniques to Deep Learning. https://github.com/baidu-research/baidu-allreduce, 2017. [Online; accessed Feb-2023].

[10] Baidu. PaddlePaddle GitHub Source Code. https://github.com/PaddlePaddle/Paddle, 2021. [Online; accessed Feb-2023].

[11] Luisa Bentivogli, Bernardo Magnini, Ido Dagan, Hoa Trang Dang, and Danilo Giampiccolo. The fifth PASCAL recognizing textual entailment challenge. In *Proceedings of the Second Text Analysis Conference, TAC 2009, Gaithersburg, Maryland, USA, November 16-17, 2009*. NIST, 2009.

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

[13] BytePS. Open-source Implementation of onebit algorithm. https://github.com/bytedance/byteps/blob/master/byteps/common/compressor/impl/onebit.cc, 2021. [Online; accessed Feb-2023].

[14] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. Adacomp : Adaptive residual gradient compression for data-parallel distributed training. 12 2017.

[15] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. In *Proceedings of International Conference on Learning Representations Workshop Track*, 2016.

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

[17] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO, October 2014. USENIX Association.

[18] Minsik Cho, Vinod Muthusamy, Brad Nemanich, and Ruchir Puri. Gradzip: Gradient compression using alternating matrix factorization for large-scale deep learning. In *NeurIPS*. 2019.

[19] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel ngraph: An intermediate representation, compiler, and executor for deep learning, 2018.

[20] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Proceedings of Advances in neural information processing systems*, pages 1223–1231, 2012.

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

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

[23] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine tuning large-scale models on commodity gpu servers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 489–501, 2023.

[24] Gluon. gluoncv Homepage. https://cv.gluon.ai/model_zoo/classification.html, 2021. [Online; accessed Feb-2023].

[25] Gluon. gluonnlp Homepage. https://nlp.gluon.ai/model_zoo/language_model/index.html, 2021. [Online; accessed Feb-2023].

[26] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. Taming momentum in a distributed asynchronous environment. *CoRR*, abs/1907.11612, 2019.

[27] Mark Harris. Bank conflict in GPU. https://devblogs.nvidia.com/using-shared-memory-cuda-cc/, 2013. [Online; accessed Feb-2023].

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

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

[30] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Proceedings of Advances in neural information processing systems*, pages 1223–1231, 2013.

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

[32] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[33] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous gpu/cpu clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[34] Junho Kim, Minjae Kim, Hyeonwoo Kang, and Kwang Hee Lee. U-gat-it: Unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation. In *International Conference on Learning Representations*, 2020.

[35] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 43. ACM, 2019.

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

[37] LAPACK and INTEL Math Kernel Library teams. Linear Algebra PACKage. https://netlib.org/lapack/explore-html/index.html, 2022. [Online; accessed Feb-2023].

[38] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of OSDI*, volume 14, pages 583–598, 2014.

[39] Shengwei Li, Zhiquan Lai, Dongsheng Li, Xiangyu Ye, and Yabo Duan. Embrace: Accelerating sparse communication for distributed training of nlp neural networks. *arXiv preprint arXiv:2110.09132*, 2021.

[40] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 279–291, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Youjie Li, Amar Phanishayee, Derek Murray, and Nam Sung Kim. Doing more with less: Training large dnn models on commodity servers for the masses. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 119–127, 2021.

[42] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *arXiv preprint arXiv:2202.01306*, 2022.

[43] Hyeontaek Lim, David Andersen, and Michael Kaminsky. 3lc: Lightweight and effective traffic compression for distributed machine learning. 02 2018.

[44] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. *arXiv preprint arXiv:1712.01887*, 2017.

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

[46] MARVELL. MARVELL White Paper for 25Gb Ethernet. https://www.marvell.com/content/dam/marvell/en/public-collateral/ethernet-adaptersandcontrollers/marvell-ethernet-adapters-fastlinq-25gb-ethernet-white-paper.pdf, 2021. [Online; accessed Feb-2023].

[47] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*, 2018.

[48] Mellanox. Mellanox Corporate Update. https://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf, 2021. [Online; accessed Feb-2023].

[49] Stephen Merity. The wikitext long term dependency language modeling dataset. https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/, 2016. [Online; accessed Feb-2023].

[50] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[51] Meta. Gradient Compression in Meta. https://github.com/pytorch/pytorch/issues/39272, 2021. [Online; accessed Feb-2023].

[52] Meta. Hook of PyTorch. https://pytorch.org/docs/stable/generated/torch.Tensor.register_hook.html, 2021. [Online; accessed Feb-2023].

[53] Meta. Gradient Compression in PyTorch. https://pytorch.org/docs/stable/ddp_comm_hooks.html, 2022. [Online; accessed Feb-2023].

[54] Meta. PyTorch PowerSGD Communication Hook. https://pytorch.org/docs/stable/ddp_comm_hooks.html#powersgd-communication-hook, 2022. [Online; accessed Feb-2023].

[55] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. Imagenet/resnet-50 training in 224 seconds. *ArXiv*, abs/1811.05233, 2018.

[56] MPICH. MPI_Alltoall. https://www.mpich.org/static/docs/latest/www3/MPI_Alltoall.html, 2021. [Online; accessed Feb-2023].

[57] msalvaris. Distributed training of deep learning models on Azure. https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/ai/training-deep-learning, 2021. [Online; accessed Feb-2023].

[58] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Truong Thao Nguyen, Mohamed Wahib, and Ryousei Takano. Topology-aware sparse allreduce for large-scale deep learning. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2019.

[60] NVIDIA. A Timeline of Innovation for NVIDIA. https://www.nvidia.com/en-us/about-nvidia/corporate-timeline/, 2021. [Online; accessed Feb-2023].

[61] NVIDIA. The API reference guide for Thrust, the CUDA C++ template library. https://docs.nvidia.com/cuda/thrust/index.html, 2021. [Online; accessed Feb-2023].

[62] NVIDIA. NCCL_Alltoall. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/p2p.html#all-to-all, 2022. [Online; accessed Feb-2023].

[63] NVIDIA. NCCL Allreduce Source Code. https://github.com/NVIDIA/nccl/blob/master/src/collectives/device/all_reduce.h, 2023. [Online; accessed Feb-2023].

[64] OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute/, 2021. [Online; accessed Feb-2023].

[65] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 184–200, New York, NY, USA, 2017. Association for Computing Machinery.

[66] Yuechao Pan. Deep gradient compression implementation in the common layer using CUDA. https://github.com/horovod/horovod/pull/453, 2018. [Online; accessed Feb-2023].

[67] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[68] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 3:1–3:14, New York, NY, USA, 2018. ACM.

[69] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (ACM SOSP 2019), Huntsville, Ontario, Canada, October 27-30, 2019*, 2019.

[70] PyTorch. PyTorch TVM. https://github.com/pytorch/tvm, 2021. [Online; accessed Feb-2023].

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

[72] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph lowering compiler techniques for neural networks, 2019.

[73] Arnaud ROUGETET. selfie2anime in Kaggle. https://www.kaggle.com/arnaud58/selfie2anime, 2019. [Online; accessed Feb-2023].

[74] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

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

[76] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[77] Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. The University of Edinburgh's neural MT systems for WMT17. In *Proceedings of the Second Conference on Machine Translation*, pages 389–399, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[78] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[79] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

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

[81] Shaohuai Shi, Qiang Wang, Kaiyong Zhao, Zhenheng Tang, Yuxin Wang, Xiang Huang, and Xiaowen 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)*, pages 2238–2247. IEEE, 2019.

[82] Shaohuai Shi, Xianhao Zhou, Shutao Song, Xingyao Wang, Zilin Zhu, Xue Huang, Xinan Jiang, Feihu Zhou, Zhenyu Guo, Liqiang Xie, et al. Towards scalable distributed training of deep learning on public cloud clusters. *Proceedings of Machine Learning and Systems*, 3:401–412, 2021.

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

[84] Nikko Strom. Scalable distributed dnn training using commodity gpu cloud computing. In *Proceedings of Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[85] Jun Sun, Tianyi Chen, Georgios Giannakis, and Zaiyue Yang. Communication-efficient distributed learning via lazily aggregated quantized gradients. In *Proceedings of Advances in Neural Information Processing Systems*, pages 3365–3375, 2019.

[86] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[87] Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. Powersgd: Practical low-rank gradient compression for distributed optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

[88] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Conference on Machine Learning and Systems (MLSys 2020)*, March 2020.

[89] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. Atomo: Communication-efficient learning via atomic sparsification. *Advances in Neural Information Processing Systems*, 31, 2018.

[90] Minjie Wang, Chien-Chin Huang, and Jinyang Li. Unifying data, model and hybrid parallelism in deep learning via tensor tiling. *CoRR*, abs/1805.04170, 2018.

[91] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[92] Zhuang Wang, Haibin Lin, Yibo Zhu, and T. S. Eugene Ng. Byte-Comp: Revisiting Gradient Compression in Distributed Training. (arXiv:2205.14465), June 2022.

[93] Jinliang Wei, Wei Dai, Aurick Qiao, Qirong Ho, Henggang Cui, Gregory R Ganger, Phillip B Gibbons, Garth A Gibson, and Eric P Xing. Managed communication and consistency for fast data-parallel iterative analytics. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 381–394. ACM, 2015.

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

[95] Wikipedia. List of NVIDIA Graphics Processing Units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units, 2021. [Online; accessed Feb-2023].

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

[97] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[98] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 1335–1344, New York, NY, USA, 2015. Association for Computing Machinery.

[99] Hang Xu, Chen-Yu Ho, Ahmed M. Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. GRACE: A Compressed Communication Framework for Distributed Machine Learning. In *Proceedings of ICDCS'21*, Jul 2021.

[100] Hang Xu, Kelly Kostopoulou, Aritra Dutta, Xin Li, Alexandros Ntoulas, and Panos Kalnis. Deepreduce: A sparse-tensor communication framework for federated deep learning. *Advances in Neural Information Processing Systems*, 34:21150–21163, 2021.

[101] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–14, 2019.

[102] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019.

[103] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander Schwing, Murali Annavaram, and Salman Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed cnn training. *Advances in Neural Information Processing Systems*, 31, 2018.

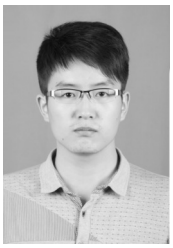
[104] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters. In *Proceedings of USENIX Annual Technical Conference 2017(USENIX ATC 17)*, pages 181–193, 2017.

[105] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 337–352. USENIX Association, July 2020.

[106] Martin A. Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2*, page 2595–2603, Red Hook, NY, USA, 2010.

Hao Wu received the BS degree from the Department of Computer Science from University of Science and Technology of China (USTC), Hefei, China in 2020. He is currently working toward the MS degree in Advanced Data Systems Laboratory at USTC. His research interests include distributed AI Systems.

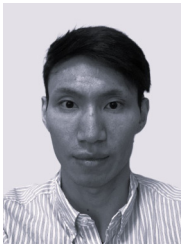
Shiyi Wang received the BS degree from the School of Information and Communication Engineering of Xi'an Jiaotong University, Xi'an, China. He is currently toward the MS degree in Advanced Data Systems Laboratory at University of Science and Technology of China. His research interests include distributed AI system.



Youhui Bai received the PhD degree from the Department of Computer Science from University of Science and Technology of China (USTC), Hefei, China, in 2021. He is currently working together with Cheng Li in Advanced Data Systems Laboratory at USTC. His research interests include distributed machine learning systems, graph processing and storage systems.

Cheng Li received the PhD degree from the Saarland University/Max Planck Institute for Software Systems, Germany, in 2016. He has been a pre-tenure professor with the Department of Computer Science and Technology, University of Science and Technology of China, since Fall 2017. His research interests include various topics related to improving performance, consistency, fault tolerance and availability of distributed systems. His work has been published at various prestigious venues, including SOSP, OSDI, ASPLOS, FAST, etc. He is the recipient of the ACM ChinaSys Rising Star Award 2021.

Quan Zhou received the BS degree from the Department of Computer Science from University of Science and Technology of China (USTC), Hefei, China, in 2018. He is currently working toward the PhD degree in Advanced Data Systems Laboratory at USTC. His research interests include distributed AI systems, graph processing and storage systems.



Jun Yi Jun Yi received the bachelor and master degrees from Southwest Jiaotong University in 2011 and 2014, respectively, and the PhD degree in computer science and engineering from the University of Nevada Reno in 2022. His research interests include large scale distributed systems, machine learning, graph neural network and cloud computing. He is a member of the IEEE.

Feng Yan received the Ph.D. degree in Computer Science from the College of William and Mary in 2016. He worked at Microsoft Research (2014-2015) and HP Labs (2013-2014). He is currently an Associate Professor of Computer Science and Electrical and Computer Engineering at the University of Houston. His research bridges the fields of big data, machine learning, and systems. He is the recipient of the Best Student Paper Award of IEEE CLOUD 2018, the Best Paper Award of CLOUD 2019, and the Best Student Paper Award of ITNG 2021, the NSF CAREER Award, the NSF CRII Award, the Outstanding Service Award of IEEE ACSOS, the Regents' Rising Researcher Award, and the CSE Best Researcher Award.

Ruichuan Chen received the Ph.D. degree in computer science from Peking University, Beijing, China, in 2009. He is currently a Distinguished Member of Technical Staff at Nokia Bell Labs, Stuttgart, Germany. Before that, he was a postdoctoral researcher at the Max Planck Institute for Software Systems, Kaiserslautern, Germany. His current research centers around cloud computing, machine learning systems, decentralized systems, and privacy-preserving technologies. His work has been published at various prestigious venues, including SOSP, OSDI, SIGCOMM, NSDI, among others, and has led to real-world adoptions.

Yinlong Xu received the BS degree in mathematics from Peking University, Beijing, China, in 1983, and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC), Hefei, China, in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC. He served the Department of Computer Science and Technology, USTC as an assistant professor, a lecturer, and an associate professor. He is currently leading a group in doing some networking, storage and high performance computing research. His current research interests include storage system, file system, social network, and high performance I/O. He was a recipient of the Excellent PhD Advisor Award of the Chinese Academy of Sciences in 2006 and Baosteel Excellent Teacher Award in 2014.