Gradient Compression Supercharged High-Performance Data Parallel DNN Training

Youhui Bai[1], Cheng Li[1,4], Quan Zhou[1], Jun Yi[2], Ping Gong[1], Feng Yan[2], Ruichuan Chen[3], Yinlong Xu[1,4]
[1]University of Science and Technology of China [2]University of Nevada, Reno [3]Nokia Bell Labs
[4]Anhui Province Key Laboratory of High Performance Computing

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 first propose a compression-aware gradient synchronization architecture, `CaSync`, which relies on a 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 100Gbps network shows that `HiPress` improves the training speed over current compression-enabled systems (e.g., BytePS-onebit and Ring-DGC) by 17.2%-69.5% across six popular DNN models.

CCS Concepts: • **Computer systems organization → Distributed architectures**; **Neural networks**.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483553

Keywords: DNN training, gradient compression

ACM Reference Format:
Youhui Bai[1], Cheng Li[1,4], Quan Zhou[1], Jun Yi[2], Ping Gong[1], Feng Yan[2], Ruichuan Chen[3], Yinlong Xu[1,4]. 2021. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3477132.3483553

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 [18, 45, 47, 64, 81]. In a typical setting, each node iterates over its own 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) [30, 34] or Ring-allreduce [7].

However, in recent years, the fast-growing computing capability, driven by the booming of GPU architecture innovations [49] and domain-specific compiler techniques [14, 17, 57, 58], 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 [30] and Ring-allreduce [64] 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* [61].

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 [4, 37, 67, 74, 76]. 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 [5, 20]. 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 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.

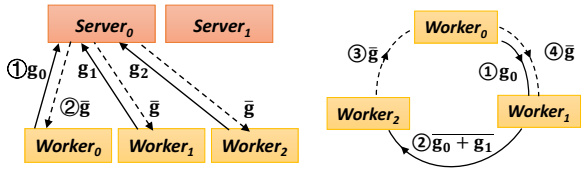
Second, we advocate that the on-GPU compression is the preferred approach for gradient compression considering GPU has much higher bandwidth and processor density than CPU, and gradients are produced in GPU directly. This creates new opportunities to further optimize the compression-communication pipeline 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 use `CompLL` in `HiPress` to construct five state-of-the-art compression algorithms (i.e., onebit [62], TBQ [67], TernGrad [74], DGC [37] and GradDrop [4]) with only 23 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 17.2%-69.5% compared with non-compression systems (including the latest BytePS) and current compression-enabled systems (e.g., BytePS-onebit and Ring-DGC), 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 [77]. 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 collectively update the global model parameters. This group coordination can be done synchronously or asynchronously. The former case often acts as a distributed



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

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

barrier for convergence guarantees [84], while the latter case eliminates the negative impact of stragglers at the cost of possibly not converging. We focus on synchronous gradient coordination because of its wide adoption [1, 13, 32, 56].

2.2 Gradient Synchronization

Parameter Server (PS) [34, 55] and AllReduce [7, 16, 42, 64] are two widely-adopted gradient synchronization strategies.

Parameter Server. In Figure 1a, each node acts as a *server* or a *worker* [34]. DNN model parameters and gradients are often partitioned across multiple servers for load balancing. When local training completes, each worker pushes gradients to servers (①), which are then aggregated and updated to model parameters. Afterwards, 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 [7], where all nodes are *workers* and they form a logical ring. As shown 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 [54].

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 [61].

The recent DNN accelerator booming [49] and domain-specific compiler advancement [14, 17, 57, 58] 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 [39, 41, 51, 75]. 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.

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%↓)

A few software approaches have been recently proposed to optimize the computation-communication pipeline, ranging from priority-based gradient scheduling and partitioning [56] to advanced synchronization architectures [30, 64]. However, as shown in Table 1, the latest highly-optimized BytePS [30] and Ring-allreduce [64] 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 16 nodes on AWS EC2 with 128 NVIDIA V100 GPUs and 100Gbps network. The communication time accounts for up to 76.8% of the total training time for training these two models, with significant portion not being hidden behind DNN computation. This indicates that the fundamental tension between gradient computation and communication persists in data parallel DNN training, even with the 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 [37, 74], 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 [5, 20].

The gradient compression algorithms generally fall within the *sparsification* and *quantization* categories. Sparsification leverages the sparsity of gradients and filters out insignificant elements in the gradient matrix [4, 32, 37], while quantization decreases the precision of gradients [67, 74, 76]. For instance, a 1-bit quantization enabled by onebit algorithm [62] could reduce the transmitted data volume by 96.9%. Many

[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).

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 [37, 67, 74].

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 computational 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, 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 [62] 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 [37] 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 generalizable 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 [54], 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 individual 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 a 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 or to decrease the precision of gradients. 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 [62], 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.

[2]The open-source onebit was implemented only on CPU [11]. For a fair comparison, we have implemented and integrated a highly-optimized on-GPU onebit into BytePS.

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. In particular, CaSync employs a composable design to enable this general yet high-performance gradient synchronization.

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 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.

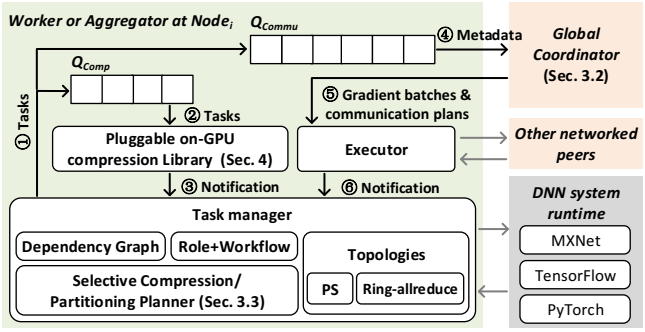


Figure 2. The CaSync architecture design, where the DNN system runtime is omitted.

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.

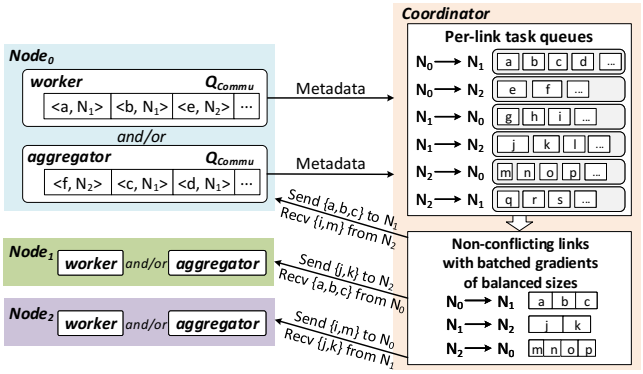


Figure 3. The workflow of the compression-aware, coordinated bulk synchronization.

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 [64]. 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 in GPU memory [6].

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. Afterwards, 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 in Figure 2). Altogether, our compression-aware, coordinated bulk synchronization enables both efficient batch compression on GPU and efficient communication of small gradients.

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 $T_{dec}(m)$,

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.

	α	β	γ
CaSync-Ring	$2(N-1)$	N	N
CaSync-PS	$2N$	$K+1$	$N+1$

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 [55, 56]. 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. Take CaSync-Ring as an example. 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, for CaSync-Ring, $\beta = (N-1)+1 = N$ and also $\gamma = (N-1)+1 = N$. 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.

Note that most, if not all, gradient compression algorithms (including the five 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

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

Figure 4. Unified compression-related API abstraction.

Table 4. List of common operators. G is a gradient matrix.

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)	Generate a random int/float in range of $[a,b)$
concat(a,$\cdots$)	Concatenate values together into a vector
extract(G')	Extract metadata from the compressed G'

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, and bitwidth or precision for quantization). 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 [4, 10, 37, 62, 67, 68, 74, 76]. 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 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. We have carefully optimized these common operators regarding memory access and bank conflicts in GPU [24], so that any algorithm implementation

```
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 }
```

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

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 [74] 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

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

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.

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 five state-of-the-art compression algorithms: onebit [62], TBQ [67], and TernGrad [74] are quantization algorithms; DGC [37] and GradDrop [4] are sparsification ones. Onebit, TBQ, TernGrad, and DGC have open-source (OSS) implementations.

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.

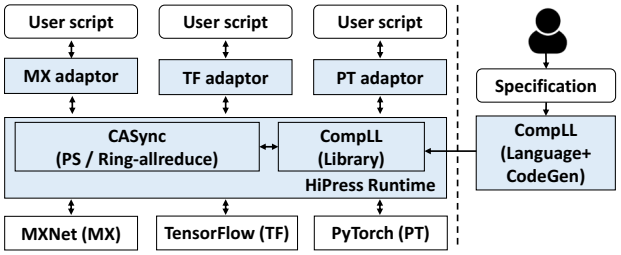


Figure 6. The overview of HiPress. The shadow boxes are the new components introduced by HiPress.

Compression performance. We compare the `encode` and `decode` operations between CompLL's auto-generated implementations and the three open-source (OSS) baselines. CompLL constantly achieves much faster speed than the baselines. For instance, the `encode` of `CompLL-TBQ` runs over 12× faster than the `OSS-TBQ`'s GPU implementation which takes 38.2ms to compress a 256MB gradient. Even though the `OSS-DGC`'s GPU implementation is manually optimized, our auto-generated `CompLL-DGC` still outperforms the `encode` of `OSS-DGC` by up to 5.1×. CompLL's auto-generated code outperforms the CPU implementation even further. For instance, `CompLL-onebit` runs up to 35.6× faster than the `encode` of `OSS-onebit`'s CPU implementation. We omit the results of `decode` operation where CompLL achieves a similar speedup.

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 [12] needs `map`, `reduce`, `filter`, `concat` and `extract` common operators, while 3LC [36] 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.

CaSync. We implement CaSync using Horovod [64], 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` [46] primitive to execute the *bulk communication step* introduced in Section 3.2. We offer another alternative primitive called

nccl_bulk_SendRecv, by composing the NCCL send and recv point-to-point communication operators.

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 common operators using Thrust [50], 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 [24] by making each thread access disjoint memory banks when possible. We also fuse the decode and merge operators for better performance.

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.

DNN systems integration. 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. Third, we add a task queue and a dedicated CPU thread to the execution engine of MXNet and TensorFlow to schedule *encode* and *decode* operators on GPU. PyTorch does not have such an execution engine, thus we implement one to enable the above function.

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 achieve the same training accuracy as their original versions?

Table 6. Statistics of trained models.

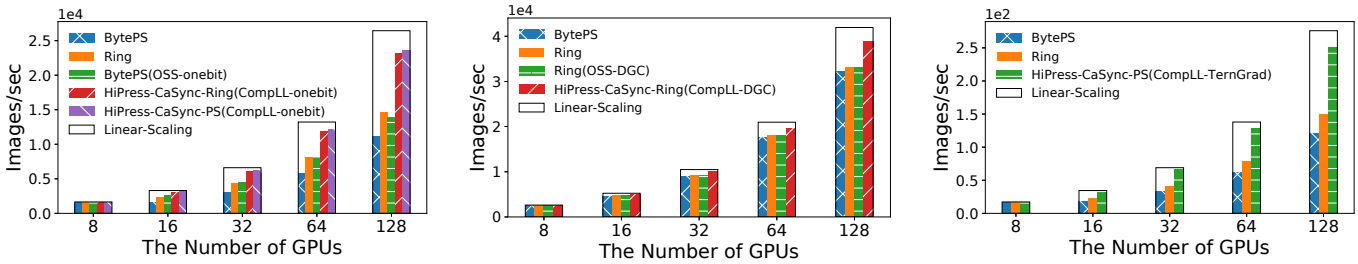
Name	Total size	Max gradient	# Gradients
VGG19 [66]	548.05MB	392MB	38
ResNet50 [26]	97.46MB	9MB	155
UGATIT [31]	2558.75MB	1024MB	148
UGATIT-light [31]	511.25MB	128MB	148
Bert-base [19]	420.02MB	89.42MB	207
Bert-large [19]	1282.60MB	119.23MB	399
LSTM [44]	327.97MB	190.42MB	10
Transformer [69]	234.08MB	65.84MB	185

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 10.1, OpenMPI 3.1.2, NCCL 2.8.4, MXNet 1.5.1, TensorFlow 1.15.5, PyTorch 1.5.0, Horovod 0.19.2 and BytePS 0.2.5.

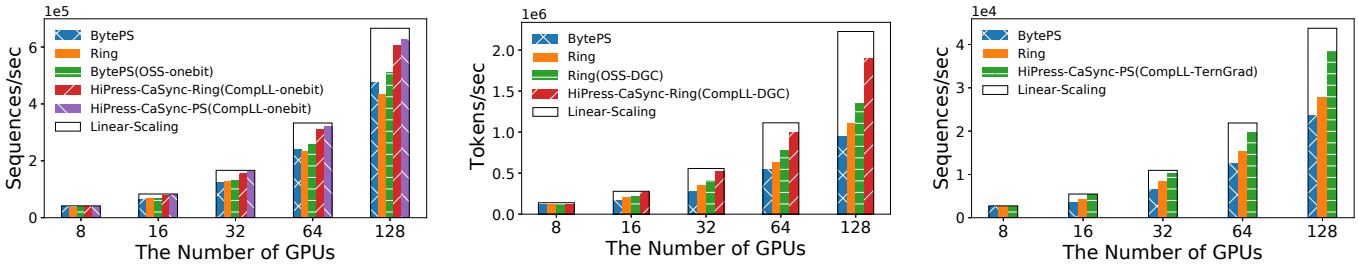
Baselines. We use TensorFlow (TF), MXNet, PyTorch with BytePS and Ring-allreduce (Ring) as no-compression baselines. In the interest of space, we only demonstrate the end-to-end performance with three out of five generated compression algorithms, namely, onebit, DGC and TernGrad, with each being evaluated within one DNN system. We use the recently developed BytePS(OSS-onebit) [5, 11] and Ring(OSS-DGC) [53] from industry as compression-enabled baselines with open-source (OSS) quantization and sparsification algorithms. Note that for a fair comparison, we use our highly optimized on-GPU implementation instead of the original on-CPU implementation for OSS-onebit.

Models and datasets. Following the literature [30, 61], 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 [60], and the remaining models with the selfie2anime [59], RTE [9],



(a) MXNet, VGG19, batch=32 images **(b)** TF, ResNet50, batch=32 images **(c)** PyTorch, UGATIT, batch=2 images

Figure 7. Throughput of computer vision models. AWS EC2 V100 instances. 100Gbps cross-node RDMA network.



(a) MXNet, Bert-large, batch=32 sequences **(b)** TF, Transformer, batch=2048 tokens **(c)** PyTorch, LSTM, batch=80 sequences

Figure 8. Throughput of natural language processing models. AWS EC2 V100 instances. 100Gbps cross-node RDMA network.

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>

WMT17 [63] and wikitext-2 [43] 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, and the training accuracy and convergence speed.

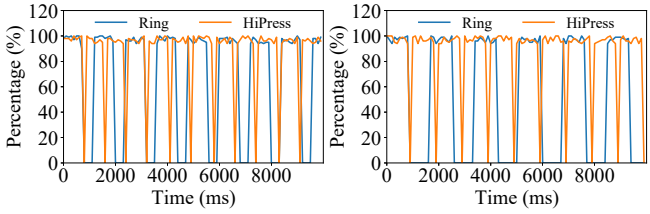
System configurations. We tune the configurations of baselines for their best performance, e.g., co-locating aggregators and workers for BytePS and `CaSync-PS`. We deploy all systems with RDMA enabled except BytePS on EC2. This is because BytePS does not support the Elastic Fabric Adapter (EFA) used 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 [19, 33, 44, 69], 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 [40, 61]. For all three 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`, we set three parameters as $2(N-1)$, N, and N respectively. The optimal thresholds of selective compression and partition sizes are produced by our cost analysis model. With 16 nodes, `CaSync` suggests to compress gradients larger than 4MB and to split the largest VGG gradient into 16 partitions before compression for AWS EC2 platform.

6.2 End-to-End Performance

6.2.1 AWS EC2 Results. Figure 7 and Figure 8 compare 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 7a and 8a. 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



(a) Bert-large **(b)** UGATIT

Figure 9. GPU utilization of Ring and HiPress when training Bert-large and UGATIT. The configurations of HiPress are the same as those used in Figure 8a and 7c.

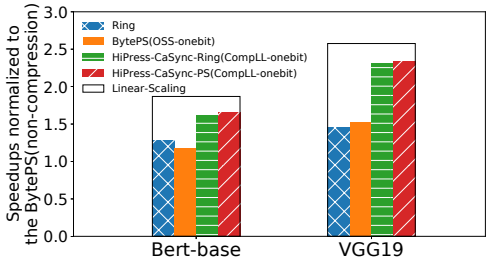


Figure 10. Training speedup normalized to BytePS atop MXNet system, in a 16-node local cluster connected via 56Gbps Infiniband network with RDMA enabled.

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, when enabling the onebit compression algorithm, BytePS(OSS-onebit) brings only limited improvements over the best-performed non-compression baselines, e.g., only up to 7.3% improvement over BytePS. Such surprising result verifies the importance of designing a compression-aware synchronization strategy to release the full potential of compression algorithms.

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(OSS-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 7b, 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 communication-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. Since PyTorch has no integrated open-sourced compression algorithms, we only compare with non-compression BytePS and Ring baselines. In Figure 7c and 8c, 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.

GPU utilization. Figure 9 compares the GPU resources used for the DNN-related computation of the non-compression baseline Ring and the best-performed HiPress configurations (Figure 8a and 7c). 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.

6.2.2 Local Cluster Results.

We also replicate all above experiments in our local cluster with low-end GPUs and

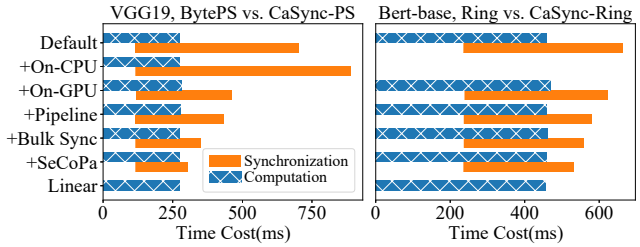
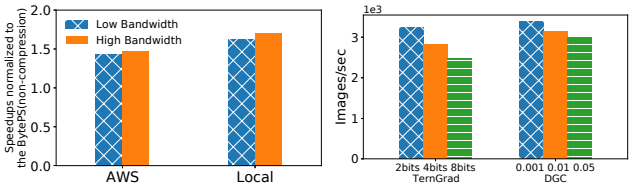


Figure 11. Impacts of enabling synchronization optimizations on the time cost of computation and synchronization.



(a) Diff. network bandwidth **(b)** Diff. compression rates

Figure 12. Training performance comparison using different network bandwidth and compression rates. Figure 12a and 12b use Bert-base and VGG19, respectively.

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 10. We use the onebit algorithm to reduce the transmitted data volume like in Figure 7a and 8a. 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(OSS-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(OSS-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.

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 11 (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 10 (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 Discussion of Other Factors

Impacts of network bandwidth. Figure 12a 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

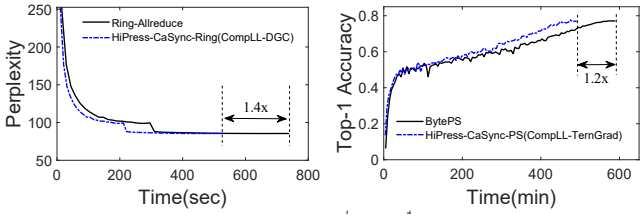


Figure 13. Convergence time of LSTM (left) and ResNet50 (right). The target perplexity for LSTM is 86.28 and the target accuracy for ResNet50 is 77.11% [21, 22].

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 12b, we compare the throughput of TernGrad and DGC algorithms generated by `CompLL` on VGG19 using `CaSync-PS` with the same setup as Figure 10. 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 13, 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.

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 [80], adopting Ring-allreduce [7, 8, 29], co-designing gradient synchronization with the physical topology [35, 38], and priority-based scheduling [25, 28, 56]. Blink generates optimal communication primitives [70], and BytePS uses spare CPU and bandwidth resources in the cluster and has already incorporated some of the above optimizations [30]. However, they are all compression-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 [82] synchronizes sufficient factors, which are compressed forms of gradients of fully connected layers in CV models. Parallax [32] 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[79] 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 [3], which can be employed by `HiPress` as an advanced feature.

Model Parallelism [15, 65] and *Pipeline Parallelism* [48] are often combined with *Data Parallelism* for large-scale deployment [71, 72], which can benefit from `HiPress`. Although `HiPress` focuses on Bulk Synchronous Parallel (BSP) in this paper given its wide adoption [32, 47]. `HiPress` is expected to work with other synchronization methods such as ASP [23] and SSP [27, 73, 78]. Finally, some components in `HiPress` are inspired by other works, such as dependency graph is inspired by Daydream [83], and fine-grained task management is inspired by MonoTasks [52].

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 and our shepherd, Lidong Zhou, for their insightful comments. We also thank Ruohui Wang for his initial exploration, as well as Lintao Zhang and Youshan Miao for their valuable suggestions. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 61802358 and 61772486, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006, and the National Science Foundation under Grant No.: CAREER-2048044, IIS-1838024 (using resources provided by Amazon Web Services as part of the NSF BIGDATA program), and CCF-1756013.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.. In *Proceedings of OSDI*, Vol. 16. 265–283.

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

[3] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2020. Accordion: Adaptive Gradient Communication via Critical Learning Regime Identification. arXiv:2010.16248 [cs.LG]

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

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

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

[7] Baidu. 2017. Bringing HPC Techniques to Deep Learning. https://github.com/baidu-research/baidu-allreduce. [Online; accessed Sept-2021].

[8] Baidu. 2021. PaddlePaddle GitHub Source Code. https://github.com/PaddlePaddle/Paddle. [Online; accessed Sept-2021].

[9] Luisa Bentivogli, Bernardo Magnini, Ido Dagan, Hoa Trang Dang, and Danilo Giampiccolo. 2009. 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. https://tac.nist.gov/publications/2009/additional.papers/RTE5_overview.proceedings.pdf

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

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

[12] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2017. AdaComp : Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. (12 2017).

[13] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. In *Proceedings of International Conference on Learning Representations Workshop Track*. https://arxiv.org/abs/1604.00981

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

[15] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. 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)*. USENIX Association, Broomfield, CO, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

[16] Henggang Cui, Hao Zhang, Gregory R. Ganger, Phillip B. Gibbons, and Eric P. Xing. 2016. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-Specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.1145/2901318.2901323

[17] 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. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. arXiv:1801.08058 [cs.DC]

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

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

[20] Facebook. 2021. Gradient Compression in Facebook. https://github.com/pytorch/pytorch/issues/39272. [Online; accessed Sept-2021].

[21] Gluon. 2021. gluoncv Homepage. https://cv.gluon.ai/model_zoo/classification.html. [Online; accessed Sept-2021].

[22] Gluon. 2021. gluonnlp Homepage. https://nlp.gluon.ai/model_zoo/language_model/index.html. [Online; accessed Sept-2021].

[23] Ido Hakimi, Saar Barkai, Moshe Gabel, and Assaf Schuster. 2019. Taming Momentum in a Distributed Asynchronous Environment. *CoRR* abs/1907.11612 (2019). arXiv:1907.11612 http://arxiv.org/abs/1907.11612

[24] Mark Harris. 2013. Bank conflict in GPU. https://devblogs.nvidia.com/using-shared-memory-cuda-cc/. [Online; accessed Sept-2021].

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

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

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

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

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

[30] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[31] Junho Kim, Minjae Kim, Hyeonwoo Kang, and Kwang Hee Lee. 2020. U-GAT-IT: Unsupervised Generative Attentional Networks with Adaptive Layer-Instance Normalization for Image-to-Image Translation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJlZ5ySKPH

[32] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware Data Parallel Training of Deep Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 43.

[33] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. CROSSBOW: scaling deep

learning with small batch sizes on multi-gpu servers. *arXiv preprint arXiv:1901.02244* (2019).

[34] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *Proceedings of OSDI*, Vol. 14. 583–598.

[35] Youjie Li, Iou-Jen Liu, Yifan Yuan, Deming Chen, Alexander Schwing, and Jian Huang. 2019. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 279–291. https://doi.org/10.1145/3307650.3322259

[36] Hyeontaek Lim, David Andersen, and Michael Kaminsky. 2018. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. (02 2018).

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

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

[39] MARVELL. 2021. 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. [Online; accessed Sept-2021].

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

[41] Mellanox. 2021. Mellanox Corporate Update. https://www.mellanox.com/related-docs/company/MLNX_Corporate_Deck.pdf. [Online; accessed Sept-2021].

[42] Mark F Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. 2006. Virtualization for high-performance computing. *ACM SIGOPS Operating Systems Review* 40, 2 (2006), 8–11.

[43] Stephen Merity. 2016. The wikitext long term dependency language modeling dataset. https://www.salesforce.com/products/einstein/ai-research/the-wikitext-dependency-language-modeling-dataset/. [Online; accessed Sept-2021].

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

[45] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. 2018. ImageNet/ResNet-50 Training in 224 Seconds. *ArXiv* abs/1811.05233 (2018).

[46] MPICH. 2021. MPI_Alltoall. https://www.mpich.org/static/docs/latest/www3/MPI_Alltoall.html. [Online; accessed Sept-2021].

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

[48] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3341301.3359646

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

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

[51] OpenAI. 2021. AI and Compute. https://openai.com/blog/ai-and-compute/. [Online; accessed Sept-2021].

[52] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 184–200. https://doi.org/10.1145/3132747.3132766

[53] Yuechao Pan. 2018. Deep gradient compression implementation in the common layer using CUDA. https://github.com/horovod/horovod/pull/453. [Online; accessed Sept-2021].

[54] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[55] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. ACM, New York, NY, USA, Article 3, 14 pages. https://doi.org/10.1145/3190508.3190517

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

[57] PyTorch. 2021. PyTorch TVM. https://github.com/pytorch/tvm. [Online; accessed Sept-2021].

[58] 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. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907 [cs.PL]

[59] Arnaud ROUGETET. 2019. selfie2anime in Kaggle. https://www.kaggle.com/arnaud58/selfie2anime. [Online; accessed Sept-2021].

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

[61] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. 2021. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio

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

[63] Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. 2017. The University of Edinburgh's Neural MT Systems for WMT17. In *Proceedings of the Second Conference on Machine Translation*. Association for Computational Linguistics, Copenhagen, Denmark, 389–399. https://doi.org/10.18653/v1/W17-4739

[64] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). arXiv:1802.05799 http://arxiv.org/abs/1802.05799

[65] 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. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Neural Information Processing Systems*.

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

[67] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Proceedings of Sixteenth Annual Conference of the International Speech Communication Association*.

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

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[70] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Jorgen Thelin, Nikhil Devanur, and Ion Stoica. 2020. Blink: Fast and Generic Collectives for Distributed ML. In *Conference on Machine Learning and Systems (MLSys 2020)*. https://www.microsoft.com/en-us/research/publication/blink-fast-and-generic-collectives-for-distributed-ml/

[71] Minjie Wang, Chien-Chin Huang, and Jinyang Li. 2018. Unifying Data, Model and Hybrid Parallelism in Deep Learning via Tensor Tiling. *CoRR* abs/1805.04170 (2018). arXiv:1805.04170 http://arxiv.org/abs/1805.04170

[72] Minjie Wang, Chien-chin Huang, and Jinyang Li. 2019. Supporting Very Large Models Using Automatic Dataflow Graph Partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/3302424.3303953

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

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

[75] Wikipedia. 2021. List of NVIDIA Graphics Processing Units. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units. [Online; accessed Sept-2021].

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

[77] 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. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[78] Eric P. Xing, Qirong Ho, Wei Dai, Jin-Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. 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* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 1335–1344. https://doi.org/10.1145/2783258.2783323

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

[80] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast Distributed Deep Learning over RDMA. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[81] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. *CoRR* abs/1906.08237 (2019). arXiv:1906.08237 http://arxiv.org/abs/1906.08237

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

[83] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 337–352. https://www.usenix.org/conference/atc20/presentation/zhu-hongyu

[84] Martin A. Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. 2010. Parallelized Stochastic Gradient Descent. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2* (Vancouver, British Columbia, Canada). Red Hook, NY, USA, 2595–2603.