Accelerating Model Training in Multi-cluster Environments with Consumer-grade GPUs

Hwijoon Lim
KAIST
Daejeon, Republic of Korea
hwijoon.lim@kaist.ac.kr

Juncheol Ye
KAIST
Daejeon, Republic of Korea
juncheol@kaist.ac.kr

Sangeetha Abdu Jyothi
UC Irvine & VMware Research
Irvine, California, USA
sangeetha.aj@uci.edu

Dongsu Han
KAIST
Daejeon, Republic of Korea
dhan.ee@kaist.ac.kr

Abstract

Rapid advances in machine learning necessitate significant computing power and memory for training, which is accessible only to large corporations today. Small-scale players like academics often only have consumer-grade GPU clusters locally and can afford cloud GPU instances to a limited extent. However, training performance significantly degrades in this multi-cluster setting. In this paper, we identify unique opportunities to accelerate training and propose StellaTrain, a holistic framework that achieves near-optimal training speeds in multi-cloud environments. StellaTrain dynamically adapts a combination of acceleration techniques to minimize time-to-accuracy in model training. StellaTrain introduces novel acceleration techniques such as cache-aware gradient compression and a CPU-based sparse optimizer to maximize GPU utilization and optimize the training pipeline. With the optimized pipeline, StellaTrain holistically determines the training configurations to optimize the total training time. We show that StellaTrain achieves up to 104× speedup over PyTorch DDP in inter-cluster settings by adapting training configurations to fluctuating dynamic network bandwidth. StellaTrain demonstrates that we can cope with the scarce network bandwidth through systematic optimization, achieving up to 257.3× and 78.1× speed-ups on the network bandwidths of 100 Mbps and 500 Mbps, respectively. Finally, StellaTrain enables efficient co-training using on-premises and cloud clusters to reduce costs by 64.5% in conjunction with a reduced training time of 28.9%.

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Keywords

System for Machine Learning, Distributed Training, Cloud Computing, Consumer-grade GPU

ACM SIGCOMM '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0614-1/24/08
https://doi.org/10.1145/3651890.3672228

ACM Reference Format:
Hwijoon Lim, Juncheol Ye, Sangeetha Abdu Jyothi, and Dongsu Han. 2024. Accelerating Model Training in Multi-cluster Environments with Consumer-grade GPUs. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3651890.3672228

1 Introduction

The field of machine learning (ML) has seen incredible growth, driven by the development of increasingly complex models [7, 12, 51] and large datasets [10]. Since training these models requires massive amounts of computing power and memory, distributed training across multiple nodes and GPUs has gained traction [26]. However, large-scale distributed learning is expensive, requiring datacenter-grade GPUs [30] that cost more than $10,000 per unit, high-speed interconnects between GPUs (e.g, NVLink or NVSwitch [31]), and GPU-dedicated networks reaching 800 Gbps [45].

Many AI researchers leverage consumer-grade GPUs in lab-scale settings for training due to their cost-effectiveness [11]. For instance, RTX 4090 delivers 73% of the training speed compared to a datacenter-grade A100 GPU at only 1/5 price. However, distributed training with consumer-grade GPUs is extremely slow, as gradient exchange is often bottlenecked by scarce network bandwidth, eventually leading to GPU underutilization [2]. Such bottlenecks are even worse in hybrid cluster settings when researchers augment the local lab resources with limited cloud GPU instances, and train a single model collaboratively across clusters separated by the Wide Area Network (WAN) with constrained and highly variable bandwidth [20] (Figure 1).

Existing approaches to accelerate training, such as gradient compression [3, 4, 14, 19, 38, 43, 48] and pipelining [23, 33, 41, 46, 53], operate in a datacenter environment with 100+ Gbps GPU-to-GPU connectivity. These solutions require inter-node bandwidths of at least 25 Gbps [41, 45, 46] to achieve high efficiency. However, in a typical WAN environment where the bandwidth is orders of magnitude lower, these methods face substantial challenges. First, in systems that rely on synchronous updates, existing pipelines are ineffective even with compression because of the lengthened gradient transfer time, leading to GPU pipeline stalls. Second, systems that rely on asynchronous updates [8, 49, 52, 54] result in an excessive degree of staleness, significantly slowing down model convergence.

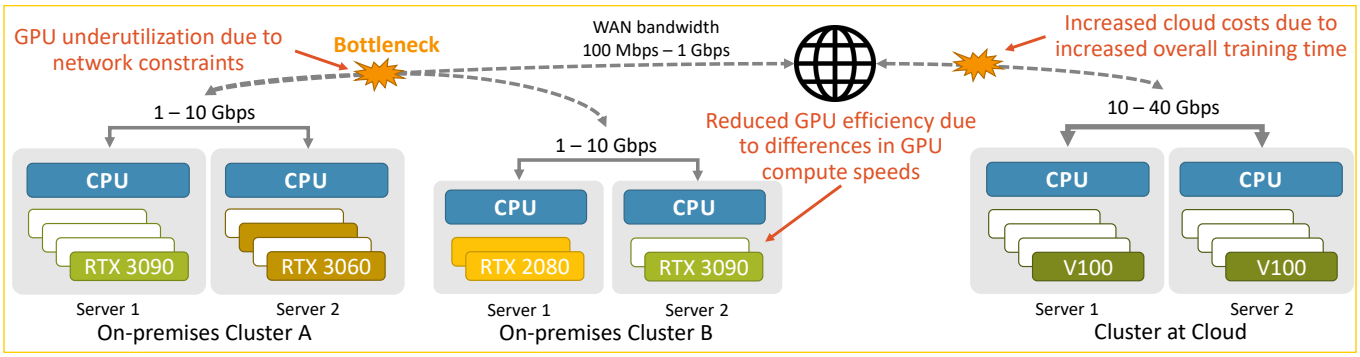


Figure 1: A multi-cluster environment with two on-premises lab clusters and a cloud cluster.

In this paper, we present StellaTrain, the first framework for distributed training that minimizes the time-to-accuracy of model training in multi-cluster environments separated by a WAN. It is the first to achieve near-optimal training speeds in multi-cloud environments. We introduce two key enablers to achieve such high training speeds. First, StellaTrain employs gradient compression to effectively use the network in low-bandwidth environments and exploits the resulting sparsity of gradients to devise computationally efficient compression and optimization.

This delivers a 128× speedup in optimization at 99% compression, allowing it to be offloaded to the CPU, further streamlining the training pipeline. Second, StellaTrain introduces layer-wise partial staleness, in which some layers immediately receive the gradient update, but for other layers, it is delayed by one iteration. This ensures that gradient updates are performed synchronously with minimum staleness and that the transfer of compressed gradient is fully interleaved with computation.

However, introducing partial staleness and compression simultaneously invites new challenges. In contrast to existing systems that rely on synchronous updates, the use of partial staleness makes the convergence speed more sensitive to compression rate and training batch size. This means that blindly optimizing for GPU utilization may not minimize Time-To-Accuracy in our environment. In addition, the optimal values of hyperparameters, such as batch size and compression rate, change dynamically with the changing WAN bandwidth, and hence, these parameters must be adapted on the fly. Reduced WAN bandwidth, for example, requires a higher compression rate and/or larger batch sizes, but these adjustments may affect convergence speed, requiring careful real-time optimization. To this end, we reassess the impact of various hyperparameters under staleness on the two key determinants of TTA—convergence speed and iteration speed. Finally, to find the optimal batch size and compression rate given the current bandwidth, StellaTrain employs Bayesian optimization and the Nelder–Mead method, which effectively locates an optimal point out of a large search space.

Our evaluation shows that StellaTrain can effectively minimize Time-To-Accuracy (TTA) with consumer-grade GPUs spread over multiple clusters. Our implementation demonstrates that StellaTrain successfully adapts the training strategy to variable network conditions [13] and reduces TTA by up to 104× compared to PyTorch DDP [26] in environments with variable WAN bandwidth.

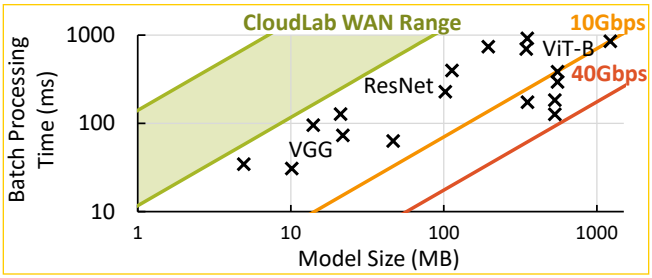


Figure 2: Model size vs. batch processing time. The bandwidth lines show the ratio of model size to batch processing times. Most models require large bandwidths for gradient exchange, falling outside of the WAN bandwidth range of CloudLab [13].

We verify that StellaTrain reduces TTA by up to 257.3× and 78.1× compared to PyTorch DDP in 100 Mbps and 500 Mbps inter-cluster settings, respectively. Finally, we show that StellaTrain can reduce cloud costs by 64.5% while also reducing the training time by 28.9% by leveraging a combination of the public cloud and on-premises cluster in a multi-cluster setup.

2 Motivation

2.1 DL training in lab-scale clusters

Compared to data center GPUs, consumer-grade GPUs are slower but are more cost-effective. For example, NVIDIA DGX A100 [29], facilitates data transfer at a rate of 2.4 TB/s over NVSwitch for intra-node communications and up to 250 GB/s through InfiniBand for inter-node communications, thus significantly accelerating training at scale. However, it costs $14,999 per unit, nearly 10× the price of an NVIDIA GeForce RTX 3090 server, which relies on slower communication channels—PCIe for intra-GPU and Ethernet for inter-node communications. Despite the limited connectivity, servers with consumer-grade GPUs provide nearly half the performance [6] of an A100 at one-tenth of the price.

Owing to their cost-effectiveness, many academic researchers and ML practitioners employ consumer-grade GPUs in their on-premises lab settings. To verify the model validity and get quick feedback, researchers prefer to have GPU resources available at all

Strategy	Iter. Speed	Conv. Speed	TTA
Gradient compression (r)	☺	☹	?
Batch size (b)	☺	?	?
Staleness (s)	☺	☹	?
Pipelining	☺	No Effect	☺
Sparse optimizer	☺	No Effect	☺
Cache-aware compression	☺	No Effect	☺

Table 1: Impact of acceleration strategies on training metrics. Each technique may positively (marked ☺) or negatively (marked ☹) affect performance. The performance of components marked with ? will vary according to the choice of parameters.

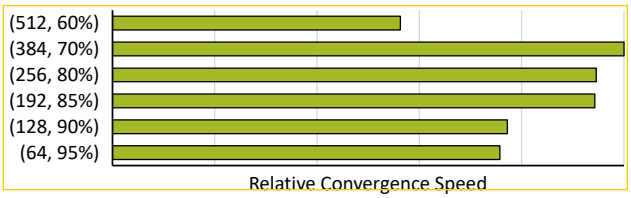


Figure 3: Convergence speed varies across various configurations of (batch size, compression rate), even if they require the same bandwidth.

times, rather than waiting in a job queue on a shared cluster. Using a public cloud is not a viable option due to its significant cost (e.g., $23,924/month for 8 A100 GPUs [39]).

As ML practitioners scale the models and work with larger datasets, the computing requirements may surpass what is feasible on on-premises GPU clusters, resulting in very long turnaround times for each training epoch. Thus, they may want to augment their computing power with remote resources to accelerate training. For example, collaborating academic groups can pool their resources to improve the overall performance of large-scale training jobs. One such model is CloudLab clusters [13], which have a wide variety of GPUs scattered over a wide-area network for shared academic use. Alternatively, academics may want to use public cloud resources alongside their lab resources.

The core problem in training under such an environment is that the transmission of gradients under the relatively low and fluctuating bandwidth conditions of the WAN significantly hampers the performance of high-end GPUs. Cutting-edge models require the transfer of tens of gigabytes of gradient data, as illustrated in Figure 2. The size of the model divided by the processing time of a training batch indicates the required bandwidth represented as crosses. The bandwidth requirements of most models fall outside the range of WAN bandwidth measured between CloudLab clusters [13] in Utah and Wisconsin, which fluctuates between 50 Mbps and 600 Mbps with an average of 193 Mbps. In our experiment with 1 Gbps inter-node connection, training a ResNet50 model leads to scenarios where clusters utilize only 17% of utilization in the cloud during the entire training duration. This inefficiency results in extended training time and wasted resources. StellaTrain ensures the full utilization of all available GPUs across multi-clusters, effectively eliminating idle wait time. Consequently, StellaTrain can deliver as much training performance in the multi-cluster as a purely public cloud-based setup. As we show in our evaluation, this reduces training time and cloud resource usage, resulting in cost savings of 64.5% and 45.1% for FP32 and FP16 training, respectively.

2.2 Need for Direction Optimization of TTA

Different acceleration strategies have varying impacts on TTA, which is a product of iteration speed and convergence speed. Some strategies, such as pipelining, improve the iteration speed without any adverse impact on the convergence speed. However, the impact of other solutions, such as gradient compression and staleness, can be double-edged—they improve the iteration speed but reduce the convergence speed. Table 1 summarizes how the strategies of StellaTrain impact one or both factors that determine the TTA: the iteration time and the convergence speed. Among the strategies, the bottom three in the table—pipelining, sparse optimizer, and cache-aware gradient compression—are all system optimizations that improve GPU utilization and thus improve the iteration speed without affecting the convergence speed.

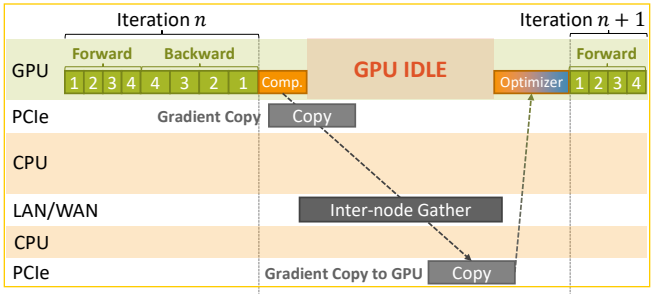
The top three strategies have tunable parameters, depending on which the overall impact is determined. Hence, these strategies need to be employed cautiously. Determining the optimal configuration of the techniques above is challenging, even in isolation, because it depends on the variable network condition and the model complexity/size. Moreover, when two or more acceleration techniques are employed simultaneously, tuning one technique could inadvertently hurt convergence speed; hence, joint optimization of parameters is necessary. For example, two configurations that achieved the same iteration time and TTA without staleness may exhibit different convergence speeds with stale gradient updates (delay of one iteration). Figure 3 shows the convergence speed for different combinations of batch size and compression rate. Although they are all equivalent in terms of iteration time, their convergence speeds, and thus their TTA, are different.

Due to the interdependence of these strategies, predicting the impact of configuration changes in the multi-strategy system is significantly more complex. Hence, a holistic approach for minimizing TTA is crucial in the multi-cluster environment separated by the WAN. Finally, since the network bandwidth between clusters is scarce and subject to frequent fluctuations [37], the optimal configuration can vary with time based on network conditions.

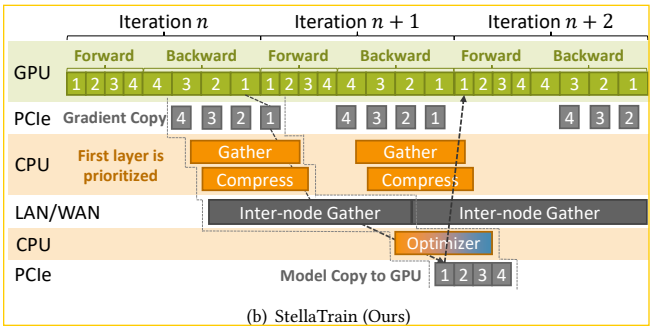
3 Optimizing Training Pipeline

StellaTrain revisits CPU offloading and model staleness to streamline the pipeline with the following observations:

- Direct data exchange between GPUs [32] is not supported in consumer-grade GPUs. Thus, CPUs play a pivotal role in gradient transfer. The presence of CPUs on the path presents an opportunity for offloading tasks, such as compression and model optimization. However, optimization and compression are an order of magnitude slower with CPUs than with GPUs. Thus, there is a critical need to enhance CPU-based optimization.



(a) GPU-based Compression (e.g., BytePS-Compress [55])



(b) StellaTrain (Ours)

Figure 4: Comparison of training pipelines.

- In a WAN environment where the bandwidth is often orders of magnitude lower than datacenter networks, transferring gradients in time for the next iteration is impossible. Such synchronous update inevitably results in underutilization of GPUs even with gradient compression, as shown in Figure 4(a).

StellaTrain strategically offloads compression and optimization to the CPU and adopts carefully designed partial staleness in the gradient update, creating a pipeline that fully utilizes the GPUs, as shown in Figure 4(b). The CPU-based sparse optimizer (§3.1) and gradient sparsification/compression (§3.2) combined delivers a 128× speed-up by leveraging sparsity in computation and cache-awareness. StellaTrain schedules gradient transfers by prioritizing layers that appear first in the pipeline and leverages partially stale updates for the remaining layers (§3.3).

Benefits. As a whole, our pipeline design offers three key benefits. First, the staleness allows us to efficiently overlap GPU-based computations (both forward and backward passes) with CPU-based operations (such as compression and optimization) and communication (both within and across nodes). Second, the CPU offload design leverages gradients already residing in the CPU memory for gradient transfer. Third, it frees up GPU resources for the more computationally intensive forward and backward passes. Consumer-grade GPUs, limited by memory, can only utilize small batch sizes. This causes the optimization and compression phases to consume a relatively larger segment of the processing time. They account for more than 43% of the total processing time. By offloading these compute-intensive tasks to the CPU, StellaTrain not only makes better use of available resources, but also avoids a major slowdown in the overall training process, offering up to 7.6× acceleration.

3.1 CPU-based Sparse Optimizer

The optimization step involves large matrix operations between the parameter and gradient tensors. These operations are slow on CPUs because of the inherently limited computational parallelism. For example, optimizing a ViT-B model with a CPU takes 17.2× more time than with a GPU (RTX 2080 Ti). To accelerate optimization at the CPU, we introduce *sparse optimizer* that takes advantage of two key characteristics:

- **Independence of each model parameter:** Each model parameter is updated independently, a feature that holds true for most standard optimization algorithms, including the Stochastic Gradient Descent (SGD) [36]. This independent update procedure can be generally represented by the equation $x \leftarrow x + \lambda g$, where λ stands for the learning rate, and g signifies the gradient of the parameter.
- **Significantly fewer sparsified gradients:** In contexts where compression (sparsification) is utilized, the number of elements in the sparsified gradients is greatly reduced compared to the original set of parameters.

Existing optimizers [46] that deal with compressed gradients ignore these characteristics and naïvely perform decompression, i.e., generating the dense gradient tensor by filling zero values for sparsified elements, before applying the optimization. As a result, regardless of how sparse the gradients are, the optimization time stays the same.

In contrast, our *sparse optimizer* incurs computation linearly proportional to the number of non-zero gradients, delivering substantial benefit in our environment where gradients are heavily compressed. The optimizer directly performs optimization on sparsified gradients without decompression into a dense matrix. Leveraging the independence of each model parameter, it applies updates element-by-element, exclusively to parameters associated with non-zero gradients, avoiding the computational redundancy of updating parameters with zero gradients. Consequently, we observe a substantial reduction in total computation by a factor of $\frac{1}{1-r}$, where r refers to the compression rate. For example, 99% compression (transferring 1% of gradients) results in 1% of computation.

Behaviorally, a minor difference exists only for momentum-based optimizers that update the parameters using a moving average, even when the current gradient is zero. Since *sparse optimizer* in StellaTrain does not update the parameter when its gradient value is zero, it may lead to a small error. However, our empirical analysis indicates the resulting error is negligible and has a minor effect on the speed of convergence.

3.2 CPU-based Gradient Sparsification

Although the sparse optimizer significantly reduces computational demands during optimization, it does not reflect the same efficiency in reducing the optimization time, i.e., a 99% compression of gradients does not result in a 99% reduction in the optimization time. This is because applying the sparse gradient in the optimization phase results in random access to the model parameters. This does not work well with the CPU's mechanism of fetching data from DRAM in cache-line size blocks (64 bytes or 16 four-byte elements) [21]. Hence, the benefits of sparse optimization can be fully harnessed

only when combined with a more cache-friendly approach in gradient compression.

Cache-aware sparsification. To reap the full benefit of sparse optimization, we design a novel compression scheme that takes into account the cache-line size for determining the top elements. We extend the threshold-v [14], which is known to be faster than the Top-k method [3] that requires partial sorting. Unlike the vanilla version, which selects the top k elements using a threshold, StellaTrain selects the top $\frac{k}{16}$ cache-aligned blocks based on the highest cumulative magnitude within each block and then selects all 16 elements within these chosen blocks for update. This strategy ensures that every element in the CPU cache is fully utilized, thereby increasing the cache hit ratio and reducing CPU pipeline stalls. This cache-aware threshold-v scheme itself enables up to 7.6× faster sparse optimization than the vanilla version, while also accelerating the compression itself by up to 3.3×. This scheme exhibits only a marginal slowdown in model convergence, as shown in Section 5.

StellaTrain leverages an automated feedback loop to adjust the threshold, v. First, StellaTrain filters blocks of gradients, each equivalent to the size of the cache line, so that the sum of the magnitudes of the gradients within a block exceeds the threshold, v. Next, the system checks if the number of filtered blocks is greater than or smaller than k_{block}, the target value. Note that $k_{\text{block}} = \frac{k}{16}$, since the cache-line blocks consist of 16 elements. When the number of filtered blocks is greater than k_{block}, StellaTrain increases v; conversely, if the number of filtered blocks is less than k_{block}, StellaTrain reduces v. StellaTrain uses an Additive Increase Multiplicative Decrease (AIMD) based estimator to update the threshold value v. The multiplicative decrease phase ensures that the system backs off quickly when the total load exceeds the threshold. To further improve efficiency, the compressed gradients are stored in COO (Coordinate) format—instead of storing the entire gradient matrix, we only store the non-zero values along with their corresponding indices.

The cache-aware threshold-v scheme offers several performance advantages. First, it minimizes the number of blocks fetched to the CPU cache. For a given target k, cache-aware threshold-v will fetch at most $\lceil \frac{k}{16} \rceil$ blocks, while the vanilla threshold-v can potentially fetch up to k in the worst case. Second, when a block is accessed, the cache-aware threshold-v updates all parameters within the block, resulting in a near-perfect cache hit rate. In contrast, the vanilla threshold-v may update only a single parameter within a block of 16, leading to a cache hit rate as low as 6.25% in the worst case. This results in a net speed-up of 128× in CPU-side optimization.

3.3 Efficient Pipeline Management

StellaTrain carefully schedules the tasks in the CPU pipeline to minimize training stalls and improve training efficiency.

Priority-based task scheduling. Traditional DNN training scheduling pipelines use a first-come, first-served approach (FCFS), where tasks are processed in the order they arrive, without considering that the gradients of the initial layers (those closer to the input) are computed last, but the updated parameters of the initial layers are needed first in the next iteration. This results in suboptimal use of computational resources, leaving GPUs underutilized while waiting for the necessary gradients from the initial layers. In addition, this delay in the initial layers can propagate through subsequent layers, bloating the entire iteration time.

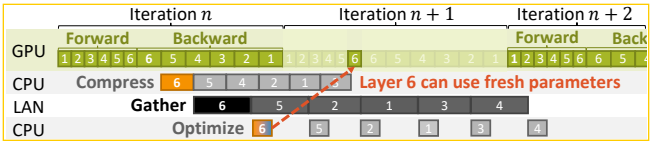


Figure 5: StellaTrain allows instant updates for some layers using fresh parameters from the previous iteration.

To address this, StellaTrain adopts a *priority-based scheduling scheme* based on the Earliest Deadline First (EDF) algorithm. Unlike the FCFS approach, StellaTrain gives higher priority to tasks associated with earlier training iterations and initial layers (Figure 4(b)). Prioritizing the initial layers effectively prevents the GPU pipeline from stalling and maximizes GPU utilization by focusing on reducing delays in the critical path. However, currently executing lower-priority tasks are allowed to continue without pre-emption to ensure efficient resource usage.

Although prior work has employed priority-based scheduling for network transfers between workers [16, 22], StellaTrain is the first to leverage layer-wise prioritization for intra-node pipelining, to the best of our knowledge. Prioritization emerges as a unique opportunity in the intra-node context with StellaTrain due to the strategic combination of bounded staleness and CPU offloading.

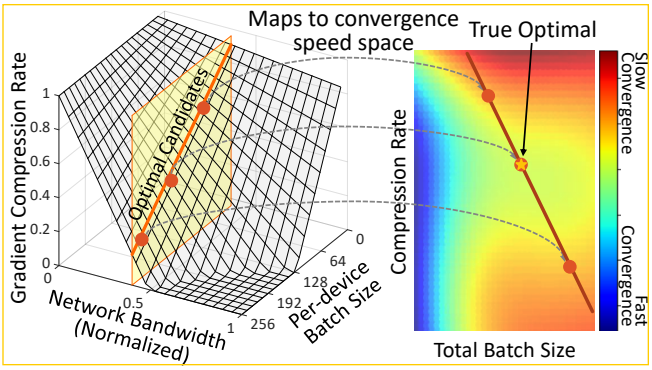
Partially stale update. Despite the priority-based task scheduling, there are instances where the parameters of the later layers (those closer to the output) are updated before those of the initial layers due to the unfinished backward pass on initial layers. In such cases, we efficiently schedule tasks for the later layers ahead of those for the initial layers without waiting for the initial layers to finish the backward pass. For example, in Figure 5, tasks associated with Layer 6 are scheduled earlier than any other layers, as the backward pass on other layers has not finished.

This provides an opportunity to leverage freshly updated parameters for some layers from the immediately preceding iteration to proceed with the next iteration. For example, in Figure 5, new parameters of Layer 6 are ready before the forward pass of Layer 6 in iteration $n+1$. We find that 15% of the layers can benefit from such instantly updated parameters on the ViT-B model.

Recognizing this, StellaTrain introduces partial staleness—a maximum staleness limit of 1 applies to all layers while also allowing immediate update of some layers without any staleness. If the parameters for a particular layer are updated before the next iteration's forward pass begins, StellaTrain employs the freshly updated parameter, potentially improving the convergence speed. Note that this approach of training models with inconsistent staleness still provides convergence guarantees [8].

4 Holistic minimization of TTA

With its optimized pipeline, StellaTrain performs co-optimization to determine training configurations that minimize TTA. For this, it employs a centralized controller that optimizes GPU configurations across heterogeneous clusters, as shown in Figure 7. Unlike hyperparameters in the ML context—such as learning rate, optimizer, and learning rate scheduling—which are usually determined through a



(a) Configuration space for iteration speed (b) Convergence speed

Figure 6: Co-optimizing strategies: Mapping between (a) Configuration space of iteration speed and (b) Configuration space of convergence speed.

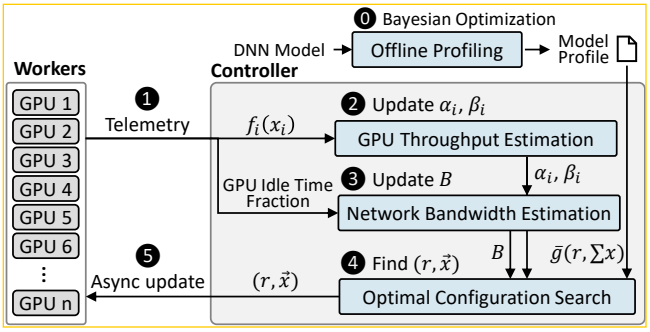


Figure 7: StellaTrain finds the best setting of (r, x) that minimizes TTA using GPU training throughput and network bandwidth estimation based on the telemetry.

random search or a rule of thumb [42], configurations to optimize TTA can be determined systematically.

The co-optimization problem can be visualized as shown in Figure 6. In the three-dimensional space of gradient compression rate, network bandwidth, and per-device batch size in Figure 6a, the optimal operating points in terms of the iteration speed of a given model lie on an intersecting line between the gray surface (determined by Equation (6)) and a plane determined by the current network bandwidth. The goal of the co-optimization problem is to identify the optimal point such that the product of iteration speed and convergence speed (represented as color in Figure 6b) is maximized.

4.1 Optimization Cycle

StellaTrain implements a centralized controller that collects telemetry data from GPUs and determines the optimal configuration of $(r, \vec{x})$ to minimize TTA, directing GPUs to use the updated configuration, as illustrated in Figure 7. Before training, the controller loads a model profile obtained via offline profiling. During training, the controller periodically collects the telemetry data from each worker and estimates the performance of each GPU and the network bandwidth. Next, the controller determines the optimal compression rate and per-device batch size based on real-time telemetry data and the model profile. Finally, the controller applies the updated configurations to the workers asynchronously.

In detail, before training, the controller loads the model profile, $\bar{g}(r, \sum_i x_i)$, which is a map from compression rate and total batch size to convergence speed, obtained from offline Bayesian optimization (§ 4.3). During training, the controller collects telemetry data from multiple GPUs, which includes the current batch size x_i, GPU training throughput $f_i(x_i)$, and GPU idle rate. Upon receiving this telemetry (❶), the controller performs the following steps:

- ❷ *GPU Throughput Estimation*: Using x_i and $f_i(x_i)$, the controller updates the parameters of the GPU throughput model, α_i and β_i, for each GPU using the Nelder-Mead method.
- ❸ *Network Bandwidth Estimation*: Based on the idle rate of the GPU, it updates the estimated bandwidth of the network B. We use a feedback loop to estimate the network bandwidth B from the idle rate of the GPU. When the idle rate of the GPU is less than 5%, the controller increases the estimated bandwidth, as this indicates that the network bandwidth is overestimated. Otherwise, the controller reduces the estimated bandwidth.
- ❹ *Optimal Configuration Search*: Leveraging $f_i(x_i)$, $\bar{g}(r, \sum x)$, and B, the controller finds the best combination of $(r, \vec{x})$ that minimizes Equation (1), given the constraint Equation (6).

After finding the optimal $(r, \vec{x})$, the controller directs the GPUs to use the updated configuration from the designated iteration (❺). Thus, this procedure does not slow down the training.

4.2 Optimization Objective

StellaTrain minimizes a weighted sum of two objectives—TTA and iteration time variance. It employs the Nelder-Mead method, a simplex-based optimization technique for nonlinear optimization problems, to minimize the resulting objective function, shown in Equation (1).

$$\text{TTA} + \lambda \times \text{iteration time variance across GPUs} \quad (1)$$

Minimizing iteration time variance across GPUs optimizes for high utilization of GPU resources. The optimization problem is also subject to a bandwidth-based constraint on the compression rate r. Next, we describe our steps in deriving the equation of the two objectives and the bandwidth constraint.

Objective 1: Variance in iteration time. Given a model, the training throughput depends on the batch size. When the batch size is less than a certain threshold (α_i), GPUs are not fully utilized. With small batches, gradient transfers occur more frequently, often limiting the throughput. Thus, training throughput increases linearly with batch size up to a threshold until the GPUs are fully saturated. The throughput of GPU i given the batch size x_i can be modeled as:

$$f_i(x_i) = \min\left(\frac{\beta_i}{\alpha_i} x_i, \beta_i\right) \quad (2)$$

where α_i and β_i respectively denotes the batch size threshold to saturate GPU i and the training throughput of GPU i after saturation. Iteration time at GPU i then can be formulated as $\frac{x_i}{f_i(x_i)}$, obtained by dividing the batch size x_i assigned to GPU i by the throughput of GPU i. We provide the values of α_i and β_i for selected models and

	RTX 2080 Ti	RTX 3090	RTX 4090
ResNet152 [17]	(32, 118)	(48, 183)	(64, 270)
ViT-B-16 [12]	(20, 99)	(32, 148)	(28, 300)
Swin-B [27]	(16, 80)	(36, 112)	(40, 217)

Table 2: Tuple of (batch size threshold α to saturate GPU, training throughput β (image/s) after saturation).

GPUs in Table 2. Note that this saturation occurs with relatively small batch sizes compared to the maximum batch size that can be trained on a device without gradient accumulation. For instance, RTX 4090 can train ResNet50 with a batch size of up to 256, but training throughput saturates early at a batch size of 64. This allows StellaTrain to select the best per-device batch size that maximizes the convergence speed.

To maximize GPU utilization in a heterogeneous GPU cluster, StellaTrain allocates batch sizes proportional to each device's computational speed. It aims to minimize iteration time discrepancies across GPUs by reducing the variance in training time per iteration (variance of $\frac{x_i}{f_i(x_i)}$ across different GPUs i). Instead of directly targeting the variance, however, StellaTrain focuses on minimizing the relative standard deviation (also known as the coefficient of variation) $\frac{\sigma_f}{\mu_f}$ shown in Equation (3), which normalizes the standard deviation against the mean, thus diminishing the influence of the mean.

$$\frac{\sigma_f}{\mu_f} = \frac{\sqrt{\sum_i^n (\frac{x_i}{f_i(x_i)})^2 - (\sum_i^n \frac{x_i}{f_i(x_i)})^2}}{\sum_i^n \frac{x_i}{f_i(x_i)}} \tag{3}$$

Objective 2: TTA. StellaTrain aims to minimize the number of additional training epochs to achieve the same training accuracy while maximizing the training throughput. We first formulate the global iteration speed by dividing the total batch sizes $(\sum_i x_i)$ by the iteration time of the slowest GPU k, since the global iteration speed is determined by the slowest device.

$$\sum_i x_i \times \frac{f_k(x_k)}{x_k} \text{ s. t. } k = \underset{i}{\text{argmax}} \frac{x_i}{f_i(x_i)}, \tag{4}$$

Leveraging Equation (4) and the surrogate model for convergence speed $\bar{g}(r, \sum_i x_i)$ (detailed in Section 4.3), the objective function to minimize TTA can be expressed as:

$$\frac{1 + \bar{g}(r, \sum_i x_i)}{(\sum_i x_i) \times \frac{f_k(x_k)}{x_k}} \text{ s. t. } k = \underset{i}{\text{argmax}} \frac{x_i}{f_i(x_i)} \tag{5}$$

Bandwidth constraint. When the network bandwidth is insufficient to complete the gradient exchange within a single iteration time, the iteration time increases. To prevent such a slowdown, we impose the following constraint on the compression rate, r:

$$r \geq 1 - \frac{B}{m} \times \frac{x_i}{f_i(x_i)} \tag{6}$$

where B and m represent the link bandwidth and the model size, respectively. Note that r should be dynamically adjusted based on the change in the link bandwidth B and the time taken for the forward and backward pass $(\frac{x_i}{f_i(x_i)})$.

Algorithm 1 Modeling convergence speed with Bayesian Optimization based on compression rate and total batch size

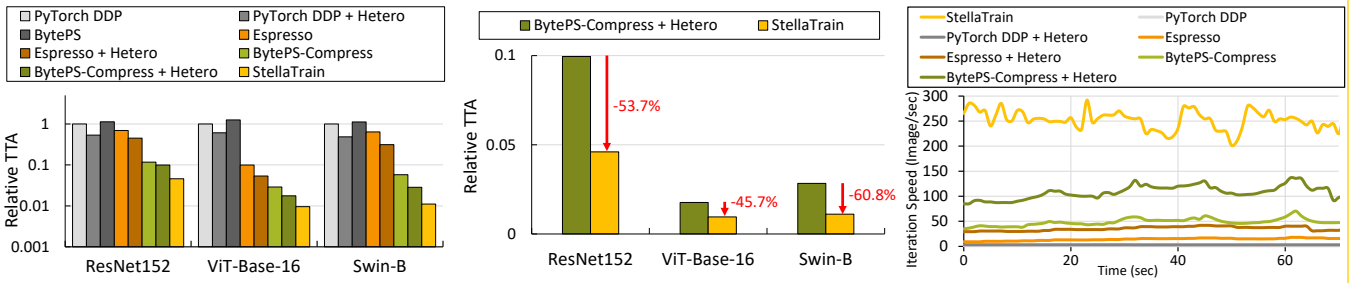
1: **function** TestRun(r, b, s)
// r: compression rate, b: batch size, s: staleness
2: $\mathcal{D}_{\text{sub}} \leftarrow$ sample($\mathcal{D}$, 0.5)
3: **for** (x, y) in get_batches($\mathcal{D}_{\text{sub}}$, b) **do**
4: $\mathcal{L} = \ell(\mathbf{y}, f(\mathbf{x}; \theta_{n-s}))$
5: $\theta_{n+1} \leftarrow \text{optimizer}(\theta_n, \text{compress}(\nabla_{\theta_{n-s}} \mathcal{L}, r))$
6: $\bar{\mathcal{L}} \leftarrow 0.99 \cdot \bar{\mathcal{L}} + (1 - 0.99) \cdot \mathcal{L}$
7: **end for**
8: **return** $\bar{\mathcal{L}}$
9: **end function**
10: Init Gaussian Process (GP)
11: $\bar{g}(r, x) \leftarrow$ surrogate model derived from the GP
12: $y_{\text{base}} \leftarrow$ TestRun(r, $\frac{x_{\min}+x_{\max}}{2}$, 0)
13: **for** $i = 1$ to N **do**
14: $r \leftarrow$ uniform random sampling in [0, 0.999]
15: Select x_{total} by optimizing the acquisition function (Expected Improvement) over $[x_{\min}, x_{\max}]$
16: $y \leftarrow$ TestRun(r, x_{total}, 1)
17: Update the Gaussian Process with $(r, x_{\text{total}}) \rightarrow y - y_{\text{base}}$
18: **end for**
19: **return** $\bar{g}(r, x)$

4.3 Modeling Convergence Speed

StellaTrain estimates the convergence speed $g(r, x)$, given the compression rate r and total batch size $x = \sum_i^n x_i$, which is used for TTA optimization in Equation (5). Ideally, we would like to create a detailed model which shows the estimate of the fraction of extra trainings epochs needed to attain equivalent model accuracy, as in Figure 6(b). To this end, we define $g(r, x)$ as the relative difference in loss after half an epoch of probe training with the total batch size x, comparing the training loss with applied compression and staleness against the loss with neither. However, profiling every potential setting of (r, x) is computationally expensive due to the large search space of possible configurations, where r can take any value in the range $[0, 1)$, and x can be up to the sum of the maximum batch sizes for all GPUs. To address this, StellaTrain employs Bayesian optimization (BO) to reduce the number of probe training runs (line 1-9 in Algorithm 1) while obtaining a $\bar{g}$ that closely approximates g.

BO builds a surrogate statistical model (e.g., Gaussian process) to approximate the expensive objective function $g(r, x)$ based on past evaluation points. An acquisition function uses this surrogate model to determine the next most promising point (r, x) to evaluate $g(r, x)$, iteratively updating the surrogate for N steps. This allows BO to find a good approximation $\bar{g}(r, x) \approx g(r, x)$ with many fewer expensive $g(r, x)$ evaluations than exhaustive search over (r, x).

We build a surrogate model $\bar{g}(r, x)$ that closely approximates $g(r, x)$ from offline profiling (Algorithm 1). We leverage Expected Improvement (EI) as the acquisition function, which is used to determine the next point to evaluate the surrogate model $\bar{g}(r, x)$, iteratively updating the Gaussian Process (GP) for N steps. Unlike the online, dynamic optimization of Section 4.1, this Bayesian optimization process is done offline in the profiling phase, once



(a) Speedup in End-to-End TTA (log-scale) (b) Speedup in End-to-End TTA (linear-scale) (c) Bandwidth Adaptation

Figure 8: Performance of StellaTrain in a multi-cluster environment.

per each model architecture before the actual training process begins. Specifically, we use a variant of EI that focuses on minimizing $g(r, x)$ for a given r, rather than finding the optimal pair (r, x) that minimizes $g(r, x)$. To achieve this, our custom EI randomly selects r and chooses x that maximizes the expected decrease in the loss, thereby ensuring uniform exploration of potential r values, which improves robustness to different r values (line 15-16 in Algorithm 1). We find that our approach with BO can achieve a high quality approximation of $g(r, x)$ after exhaustively exploring only $N = 30$ steps, which is only 0.003% of what would be required for exhaustively exploring the entire search space (i.e., 1000 r values (0 to 0.999) and 1021 x values (4 to 1024)).

5 Evaluation

We evaluate the performance of StellaTrain to answer the following questions:

- How much improvement in TTA does it deliver? (§5.1)
- What impact do the batch size adaptation and cache-aware compression have on the training pipeline? (§5.2)
- How efficient are the CPU-based schemes, cache-aware compression, and sparse optimizer? (§5.3)

Implementation. We implement StellaTrain in 4.7k lines of C++ and provide a model wrapper for compatibility with existing PyTorch training workflows. It leverages 16 worker threads per GPU in a thread pool model to maximize resource utilization. It avoids busy waiting for CUDA events and synchronization to save CPU cycles. We employ shared memory and ZeroMQ [5] for efficient inter-process and inter-node communication, respectively.

Settings. We compare the performance of StellaTrain with four baselines: PyTorch DDP [26] (gloo backend), BytePS [23], BytePS-Compress [55] and Espresso [46]. BytePS-Compress uses GPU-based gradient compression, which is built upon a network pipelining solution, BytePS. Espresso, the closest related work, adds adaptive gradient compression atop BytePS. We further implement heterogeneous GPU support for PyTorch DDP and BytePS-Compress. For the baselines, we use a batch size that fully fills the GPU memory.

We perform evaluations using three nodes in the on-premises cluster with 2 GPUs each and one node in the remote cluster with 4 GPUs (4 nodes and 10 GPUs total). Three nodes in the on-premises cluster have two NVIDIA RTX 4090, two RTX 4090, and two RTX 3090, respectively. The node in the remote cluster has four NVIDIA V100 GPUs. We emulate the bandwidth between the on-premises (Node 1, 2, 3) and remote (Node 4) clusters using the network bandwidth trace we measured between two CloudLab clusters [13] (Utah and Wisconsin), in order to maintain consistency across experiments. The average WAN bandwidth between the on-premises and the remote cluster is 115.7 Mbps.

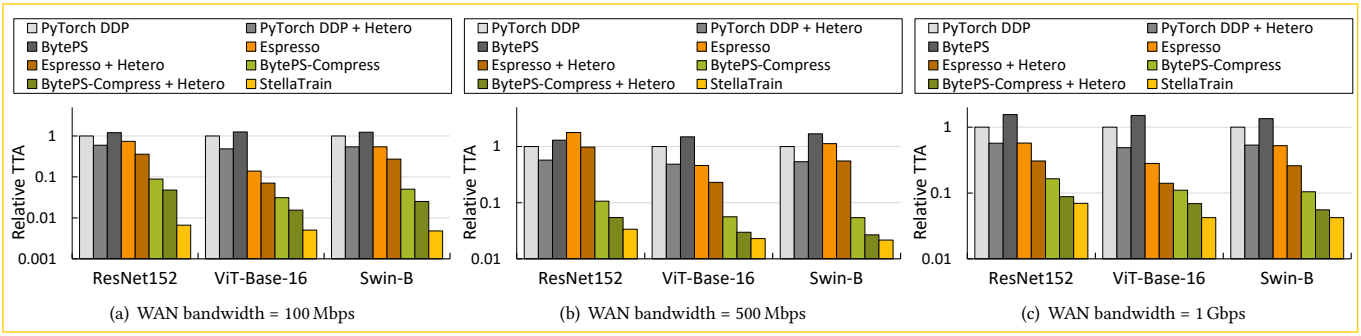
We also show evaluations with the specific link speeds of 100 Mbps, 500 Mbps, and 1 Gbps, which were selected after the WAN bandwidth observation above. In these evaluations, we use the same node setup except for Node 4, in which we replace the remote GPU node with an on-premises node with two RTX 2080 Ti. We use the suggested compression rate of 90%, 95%, and 99% for BytePS-Compress and Espresso for the bandwidths of 1 Gbps, 500 Mbps, and 100 Mbps, respectively.

We evaluate three different models: ResNet152 [17] (58 M parameters), ViT-Base-16 [12] (88 M parameters), and Swin-B [27] (139 M parameters). We train the model with the ImageNet-100 dataset (a subset of ImageNet [10] with 100 classes) up to 100 epochs and leverage SGD [36] optimizer to update parameters. We also fine-tune a pre-trained LLM, GPT-2 (123.6 M parameters).

5.1 End-to-end benefits

Figure 8(a) shows the end-to-end training time to reach the same training loss (or time-to-accuracy) in the multi-cloud scenario, normalized to that of PyTorch DDP. Compared to methods that do not involve compression, StellaTrain achieves higher TTA reduction by up to 104×. PyTorch DDP suffers from both high gradient exchange time under limited network bandwidth and under-utilization of GPUs due to stragglers, as RTX 3090 is up to 2.02× slower than RTX 4090 (Table 2). The improved version of PyTorch DDP that supports heterogeneous GPUs (denoted as PyTorch DDP + Hetero) trains the model up to 2.06× faster but is still up to 63.7× slower than StellaTrain.

Compared to systems that employ gradient compression (BytePS-Compress and Espresso), StellaTrain reduces TTA by up to 57.8×. Powered by GPU-based compression, BytePS-Compress achieves significant gain over non-compressed baselines as it eliminates the bottleneck in gradient transmission. However, all baselines are still slower than StellaTrain, even with heterogeneous GPU support. StellaTrain achieves 1.84×-2.55× better TTA compared to BytePS-Compress + Hetero, which is the best-performing baseline except for StellaTrain, as shown in Figure 8(b) in linear scale. This is because its gradient compression runs on GPU, consuming valuable



(a) WAN bandwidth = 100 Mbps (b) WAN bandwidth = 500 Mbps (c) WAN bandwidth = 1 Gbps

Figure 9: Speedup of End-to-End TTA with different WAN bandwidths (relative to PyTorch DDP TTA).

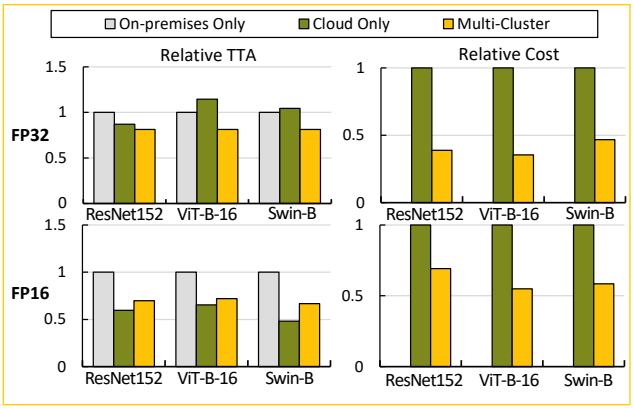


Figure 10: Cost savings of StellaTrain.

GPU resources for compression, and its use of synchronous update causes the GPU to wait for the gradient exchange. In addition, their inability to adjust to variable WAN bandwidth results in slower gradient exchanges and reduced iteration speed when the WAN bandwidth experiences degradation.

Figure 8(c) shows the iteration speed of training the ViT-Base-16 model trained with StellaTrain and other baselines over time, under varying WAN bandwidth. StellaTrain, unlike baselines lacking bandwidth adaptation, dynamically adapts its compression rate and batch size to the available bandwidth, thereby maintaining stable and high iteration speeds close to the optimal.

Cloud cost reduction. Figure 10 demonstrates that StellaTrain reduces cloud costs by 64.5% and 45.1% for FP32 and FP16 training, respectively, through efficient co-training between cloud and on-premises clusters. This is achieved through efficient co-training across cloud and on-premises clusters, allowing users to augment their GPU resources in the cloud as needed but at a reduced cost. We examine TTA and cloud expenses across three configurations: 1) 6 RTX GPUs (on-premises only), 2) 8 V100 GPUs (cloud-only), and 3) a combination of 6 RTX GPUs and 4 V100 GPUs using StellaTrain for co-training.

Training only with on-premises resources eliminates cloud costs, but limits training throughput, limiting scalability and TTA reduction potential. Conversely, cloud-based training with 8 V100 GPUs increases throughput (in FP16 training[1]) but incur significant costs. Co-training with StellaTrain, using both on-premises and cloud resources, enhances throughput by 40.7% for FP32 while requiring fewer cloud GPUs. This leads to a total cost reduction of 64.5% for FP32 and 45.1% for FP16 training.

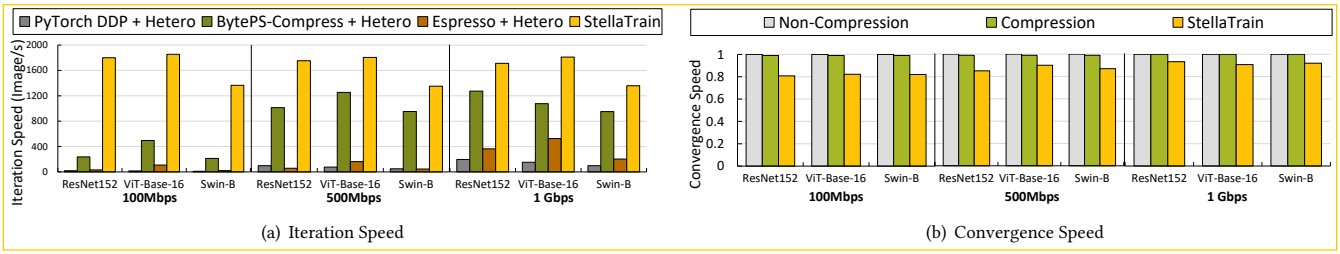
TTA with different WAN bandwidths. Figure 9 shows the TTA of StellaTrain and other baselines on multiple different preset bandwidths. As in the case with variable bandwidth, StellaTrain achieves higher TTA reduction by 127.8-257.3×, 26.8-78.1×, and 14.3-35.8× in 100 Mbps, 500 Mbps, and 1 Gbps scenarios, respectively, over non-compressed baselines. Compared to systems with gradient compression and heterogeneous GPU support (BytePS-Compress and Espresso + Hetero), StellaTrain still reduces TTA by 3.07-56.4×, 1.23-26.3×, and 1.31-6.15× in 100 Mbps, 500 Mbps, and 1 Gbps scenarios, respectively.

However, even in the static bandwidth environment, BytesPS-Compress with heterogeneous GPU support, the fastest baseline except for StellaTrain, is still up to 6.12× slower than StellaTrain. In particular, in the 100 Mbps environment, it wastes valuable GPU resources for compression and needs to wait for the gradient exchange, causing GPU stalls. Espresso performs better than non-compressed baselines but still struggles in low-bandwidth scenarios. As Espresso's logic deciding how, where, or whether to compress is highly optimized for higher bandwidth environment (≥25 Gbps), Espresso scheduler makes inefficient decisions—it exchanges most of the layers uncompressed or only applies gradient quantization to FP16. (e.g., only compresses 7 out of 330 layers for Swin-B). This ends up transferring more data than fixed compression.

Iteration and convergence speed. To identify the factors contributing to performance gains, we compare the iteration speed as well as the convergence speed of each framework in Figure 11. StellaTrain achieves the best iteration speed (Figure 11(a)) among all systems due to its optimized pipeline. The iteration speed of StellaTrain outruns that of GPU-based compression scheme by up to 785%, 72.8%, and 68.1% in 100 Mbps, 500 Mbps, and 1 Gbps, respectively.

Unlike common belief, Figure 11(b) show that the convergence speed is not heavily penalized even with high compression rate

[1]Note that FP32 training in V100 is even slower than in RTX 2080 Ti. Datacenter-grade GPUs (V100, A100) are optimized for FP16 training and are only faster than consumer-grade GPUs in such cases.



(a) Iteration Speed (b) Convergence Speed

Figure 11: Iteration Speed and Convergence Speed with different WAN bandwidths (relative to PyTorch DDP).

and staleness. While other compression-based schemes (BytePS-Compress, Espresso) feature high convergence speed, we empirically demonstrate that the overhead of waiting for the gradient exchange is actually greater than the penalty in convergence speed due to staleness.

Fine-tuning LLMs. We show that StellaTrain enables efficient fine-tuning of LLM models in a multicluster environment. We compare the TTA of fine-tuning pre-trained GPT-2 model (123.6 M parameters) when trained with PyTorch DDP and StellaTrain. We fine-tune pre-trained GPT-2 model with WikiText-103 [28] dataset for 1 epoch. When trained with 1 Gbps network, we find StellaTrain is 17.2× and 8.71× faster than training with PyTorch DDP and PyTorch DDP with heterogeneous GPU support. The performance gain is comparable to our evaluation on image-based training.

Using StellaTrain, users can now efficiently train LLMs across multiple nodes. Without StellaTrain, the best one can do within our 8 GPU cluster is using only one node with 2 RTX4090. On a single machine with two RTX 4090s, LLM fine-tuning results in a throughput of 17.8 iterations/second. On our distributed cluster with 8 GPUs (detailed in settings), we achieve a significant speed-up with a throughput of 27.2 iterations/second with StellaTrain (a 53% speedup compared to the single-node setup). On the same cluster, PyTorch-DDP can achieve only 1.58 iterations/second, i.e., 11.24× slowdown compared to a single-node setup.

For fine-tuning LLMs over large-scale distributed datasets, StellaTrain can reduce the WAN bandwidth usage by avoiding the need for moving the entire dataset. Instead, it exchanges gradients over the limited-bandwidth WAN.

5.2 Component-wise benefits

Dynamic batch size adaptation. Figure 14(a) shows how StellaTrain selects the optimal compression ratio and total batch size under varying bandwidth for the ViT-Base-16 model, evaluated on two nodes with two RTX 4090 and two RTX 2080 Ti each. StellaTrain selects different batch size and compression rate depending on the observed bandwidth; when the available bandwidth is 1 Gbps, StellaTrain selects a batch size of 137 with a 97% compression rate, and when the available bandwidth is 100 Mbps, StellaTrain choose a batch size and compression rate of 240 and 99.5%, respectively.

With Swin-B, StellaTrain achieves 43.3% and 8.1% faster TTA than baselines with no adaptation and compression-ratio only adaptation, respectively, as shown in Figure 14(b). The baseline without any adaptation suffers from both slower iteration speed and convergence speed coming from congestion and sub-optimal batch size selection. Adapting compression rate also makes a sub-optimal choice of batch size, risking a very high compression ratio when bandwidth is scarce, slowing down the convergence.

Figure 14(c) shows how StellaTrain adapts the batch size of each GPU in response to fluctuations in bandwidth in an evaluation with 4 GPUs (two RTX 4090 and two RTX 2080 Ti). The dashed line represents a trace of the fluctuating WAN bandwidth measured between CloudLab clusters, while the solid line indicates the batch size selected by StellaTrain for different types of GPUs. StellaTrain dynamically adjusts the batch size across the GPUs to minimize the TTA, selecting larger batch sizes for faster GPUs and smaller batch sizes for slower GPUs to prevent stragglers.

Cache-aware compression and Sparse optimizer. Cache-aware compression and the sparse optimizer are essential for maximizing GPU utilization during compressed gradient exchange. Using PyTorch Top-k compression instead of cache-aware compression decreases iteration speed by 11.3% due to the higher computational demands. Replacing the sparse optimizer with the default SGD optimizer in PyTorch further slows down iteration speed by 17.3%. As both processes are computationally heavy, when both cache-aware compression and the sparse optimizer are disabled, the iteration speed diminishes significantly, showing a 44.3% reduction.

5.3 Deep dive

Sparse optimizer. Figure 12 shows the time taken by optimizers to update the model parameters. We compare the dense optimizer from PyTorch with the CPU-based sparse optimizer in StellaTrain (§3.1). While the dense optimizer is fast on GPUs, it becomes significantly slower on a CPU, requiring up to 172 ms, which makes the CPU-based optimization infeasible. In contrast, our sparse optimizer, with cache awareness turned off, can achieve an optimization time of 10.2 ms (at 99% gradient compression), which is 16.8× the reduction. The cache-aware compression further reduces this time, achieving 1.34 ms at 99% gradient compression, which amounts to 128x speed up. Note that for the dense optimizer, the time remains constant regardless of the compression ratio, as the total computation is fixed to the parameter size.

Cache-aware compression. The cache-aware threshold-v compression method accelerates compression itself as shown in Figure 13 (a). It achieves up to 3.35× faster compression than its non-cache-aware counterpart, requiring only 33.5 ms to compress the gradient for the entire model. Remarkably, cache-aware threshold-v is 16% faster than vanilla threshold-v, even on GPUs. In Figure 13

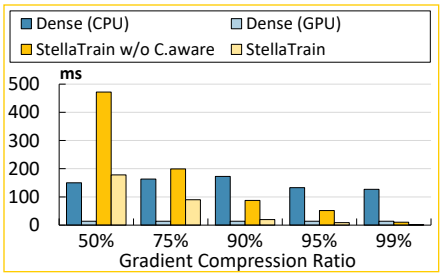
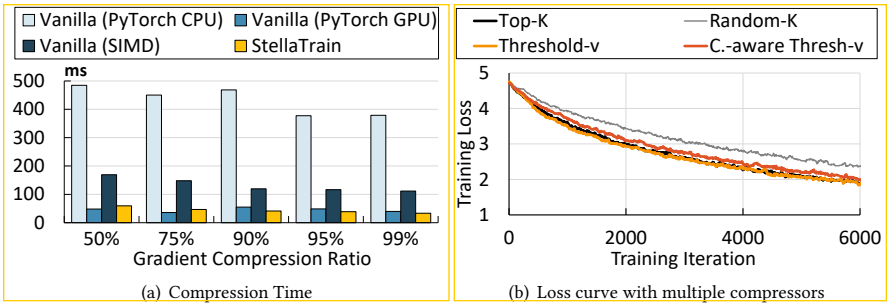


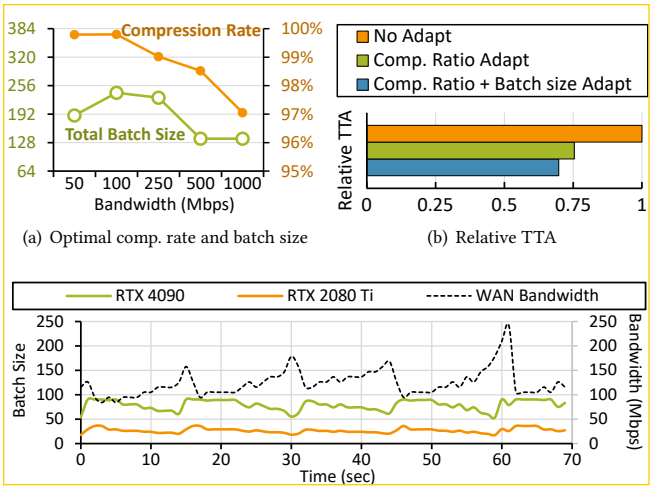
Figure 12: Optimization time of StellaTrain vs. dense optimizers.



(a) Compression Time

(b) Loss curve with multiple compressors

Figure 13: Performance of the cache-aware threshold-v compression.



(a) Optimal comp. rate and batch size

(b) Relative TTA

(c) Batch size adaption under bandwidth fluctuation between CloudLab clusters

Figure 14: Compression rate and batch size adaptation.

(b), we compare model convergence speed and observe that cache-aware threshold-v is comparable to vanilla threshold-v and outperforms random-k, a commonly used scheme for lightweight compression.

Priority-based scheduling. We evaluate the impact of gradient prioritization by comparing the baseline First-Come-First-Serve (FCFS) scheduling of gradient updates with priority-based scheduling in StellaTrain using the ResNet152 model. We observe that the baseline FCFS is 9.6% slower compared to priority-based scheduling in StellaTrain. Since model updates from initial layers are prioritized in StellaTrain, the forward pass of the subsequent iteration can begin sooner in StellaTrain. As model size increases, the benefits of prioritization will become more prominent since the initial layers have to wait longer in FCFS.

Staleness. Applying bounded staleness is important in accelerating the convergence speed. Eliminating staleness requires the subsequent iteration to wait for gradient exchange, which stalls the training pipeline. Our evaluation with Swin-B shows that eliminating staleness slows down the iteration speed by 22%. While bounded staleness affects the convergence speed, it doesn't influence the eventual convergence of the model. Our results indicate that the introduction of staleness can slow down the convergence speed by at most 18%. However, this slowdown is still less significant than the slowdown observed when staleness is completely eliminated (22% - 33.4%). Thus, introducing bounded staleness is beneficial in reducing the TTA.

In our adaptive staleness approach, about 15% of the layers are updated without staleness, providing a balance between maintaining fast iteration speed and effective convergence.

6 Related work and Discussion

Federated learning. StellaTrain is designed for environments where each GPU computes gradients very fast. This is in contrast to Federated Learning (FL), where asynchronous parameter update is applied to accommodate highly variable and inherently slow computation speeds. Applying existing FL solutions in such a setup would rather increase model staleness excessively, severely slowing down the convergence speed. The main focus of FL is rather on handling non-IID data across different clients, which is not a concern for StellaTrain as it operates with IID training data.

Benefits of co-optimization. Figure 15 shows the trade-off space of acceleration strategies in terms of the two determinants of TTA. The vanilla distributed training scheme, PyTorch DDP [26], has a high convergence speed but low iteration speed due to high network load. Compression strategies [3, 4, 14, 19, 43] reduce the network load considerably and, thereby, improve iteration speed. However, compression could affect the convergence speed significantly if it is not carefully tuned, particularly at low bandwidths. Network adaptive compression schemes, such as DC2 [1] and Kimad [47], can mitigate this effect. By combining pipelining with compression, Espresso [46] can further improve both convergence and iteration speeds, but it requires careful tuning. StellaTrain significantly improves the iteration time while also maintaining high convergence speed, as demonstrated in our evaluations. The co-optimization of multiple acceleration strategies enables us to unlock the high-performance region in the design trade-off space.

The performance of various schemes in Figure 15 can be visualized as the flexibility that each scheme offers to explore the optimal plane in Figure 6. Espresso searches for the optimal point on the two-dimensional plane of compression ratio and network bandwidth in Figure 6. StellaTrain, on the other hand, enables us to search the three-dimensional space alongside leveraging fixed staleness, priority-based task scheduling, and CPU-based optimizations.

Supporting larger models. The design of StellaTrain focuses on supporting data parallelism in a multi-cluster environment, separated by a WAN. Although our current implementation and evaluation are limited to training models that fit within the GPU memory, we believe it can be easily extended to support the training

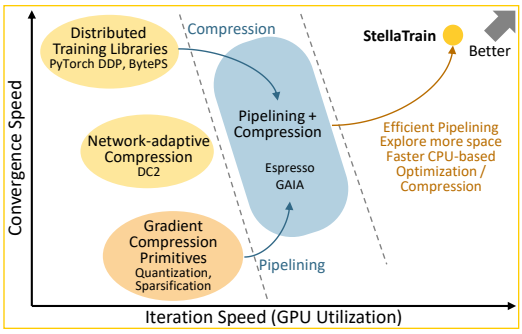


Figure 15: Multi-cluster acceleration strategies in the trade-off space of the two determinants of TTA.

of LLMs with hundreds of billions of parameters (e.g. GPT-3 [7], OPT-175B [51]) with negligible additional overhead.

There are two main ways to enable this. The most straightforward approach is to apply existing offloading techniques to train larger models within a single GPU. StellaTrain then treats each individual GPU machine as a cluster of its own without any modification. With parameter offloading [35] and activation checkpointing, StellaTrain can save GPU memory by maintaining only the active layer's parameters on the GPU and offloading the rest to the CPU memory. Implementing such techniques only requires an extra parameter upload step before the backward pass and is unlikely to significantly impact the iteration speed, given that the current PCIe upload bandwidth utilization is less than 20%. The second approach is to take a hierarchical design. This takes advantage of the high-speed connectivity within a cluster to train larger models with model parallelism and apply data parallelism across clusters. Further research is required into streamlined multi-tiered pipelining to enable hierarchial gradient exchange across model/data parallelism boundaries.

Lack of formal proof of convergence. In StellaTrain, we combine multiple acceleration strategies and empirically demonstrate that co-optimization does not compromise convergence. Through multiple experiments, we show that the convergence speed is within 82% of the baseline. We sacrifice slightly on convergence speed to significantly improve the iteration speed and, thereby, TTA. We leave the formal proof of convergence as future work.

Note that the proof of convergence for most acceleration strategies used in ML was developed after the techniques gained popularity and widespread practical adoption. For example, gradient sparsification/compression was shown to be practically feasible and highly performant as early as 2015 [3, 44], leading to extensive real-world usage, before a theoretical justification for convergence was established in 2018 [4]. Similarly, the theoretical grounds for staleness were established between 2011 and 2015 [18, 25, 34, 50] and are still a topic of recent studies [8], while the technique has been widely used in practice for scalable training for more than a decade since 2009 [9, 24, 56]. Our contribution is to empirically demonstrate the benefit of combining the two, calling for contributions in the theoretical realm.

Optimizing collectives in multi-strategy space. Aggregation of compressed gradients is a challenge [40] because, at each worker, the set of gradients with the largest magnitude may be different. Currently, StellaTrain employs a simple approximation by choosing the top k/N gradients at each of the N workers so that at most k gradients are updated across all workers. OmniReduce [15] proposed transmission of only non-zero blocks in this setting. We can expand the design space by allowing the careful elimination of non-zero blocks. We leave the design of a compression-aware gradient aggregation scheme with greater flexibility as future work.

Handling errors in bandwidth estimation. StellaTrain determines the compression ratio and batch size per device based on the estimated network bandwidth. Errors in bandwidth estimation could potentially affect pipelining. If the real bandwidth is higher than the estimates, we underutilize the resources. On the other hand, if the real bandwidth is lower, it can have significant adverse effects on pipelining and, in turn, the convergence speed. Thus, overestimation of bandwidth is more harmful than underestimation for pipelining. Hence, we adopt a conservative approach to bandwidth estimation and adaptation in StellaTrain.

Managing WAN challenges. While this work focuses on adapting to bandwidth fluctuations for WAN training, we acknowledge that WAN environments can experience other transient conditions such as packet drops and timeouts, introducing additional challenges. StellaTrain relies on the underlying TCP and ZeroMQ transports to recover from packet loss and adjust transmission rates accordingly, ensuring that training accuracy is not impacted. However, extremely poor WAN conditions may warrant dynamic node management that allows new nodes to join or leave the training cluster based on their network status, which could further improve training efficiency and robustness. Finally, StellaTrain does not deal with fault tolerance and recovery from failures.

7 Conclusion

StellaTrain is the first framework that takes a holistic approach to accelerate model training in consumer-grade GPU clusters by co-optimizing several acceleration techniques. We investigate the design space of acceleration techniques and identify the key trade-off knobs for reducing the time-to-accuracy. In addition to leveraging well-known solutions, StellaTrain introduces novel strategies for training acceleration, including a cache-aware gradient compression scheme and a CPU-based sparse optimizer. Our evaluation demonstrates that StellaTrain can accelerate distributed training in multi-cluster settings with limited bandwidth by up to 104×, while also adapting seamlessly to bandwidth fluctuations. StellaTrain achieves up to 257.3× and 78.1× speedups on the network bandwidths of 100 Mbps and 500 Mbps. StellaTrain also provides significant speedup in fine-tuning LLMs. This work does not raise any ethical issues.

Acknowledgments

We thank our shepherd Dejan Kostic and the anonymous reviewers for providing helpful feedback and suggestions to improve our work. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00340099) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00418784).

References

[1] Ahmed M Abdelmoniem and Marco Canini. 2021. DC2: Delay-aware compression control for distributed machine learning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications* (Vancouver, BC, Canada). IEEE.

[2] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2022. On the utility of gradient compression in distributed training systems. *Proceedings of Machine Learning and Systems* 4 (2022).

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

[4] Dan Alistarh, Torsten Hoefler, Mikael Johansson, Nikola Konstantinov, Sarit Khirirat, and Cédric Renggli. 2018. The convergence of sparsified gradient methods. *Advances in Neural Information Processing Systems* 31 (2018).

[5] The ZeroMQ authors. 2023. ZeroMQ. https://zeromq.org/.

[6] BIZON. 2023. GPU Deep Learning Benchmarks 2023–2024. https://bizon-tech.com/gpu-benchmarks/NVIDIA-RTX-3090-vs-NVIDIA-A100-40-GB-(PCIe)/579vs592. [Accessed 01-02-2024].

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020).

[8] Yangrui Chen, Cong Xie, Meng Ma, Juncheng Gu, Yanghua Peng, Haibin Lin, Chuan Wu, and Yibo Zhu. 2022. SAPipe: Staleness-Aware Pipeline for Data Parallel DNN Training. In *Advances in Neural Information Processing Systems*.

[9] 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. *Advances in neural information processing systems* 25 (2012).

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

[11] Tim Dettmers. 2023. The Best GPUs for Deep Learning in 2023 — An In-depth Analysis. https://timdettmers.com/2023/01/30/which-gpu-for-deep-learning/. [Accessed 27-Jun-2023].

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

[13] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. https://www.flux.utah.edu/paper/duplyakin-atc19

[14] Aritra Dutta, El Houcine Bergou, Ahmed M Abdelmoniem, Chen-Yu Ho, Atal Narayan Sahu, Marco Canini, and Panos Kalnis. 2020. On the discrepancy between the theoretical analysis and practical implementations of compressed communication for distributed deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34.

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

[16] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. 2019. Tictac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems* 1 (2019).

[17] 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*.

[18] 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. *Advances in neural information processing systems* 26 (2013).

[19] Samuel Horváth, Chen-Yu Ho, Ludovit Horvath, Atal Narayan Sahu, Marco Canini, and Peter Richtárik. 2022. Natural compression for distributed deep learning. In *Mathematical and Scientific Machine Learning*. PMLR.

[20] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. 2017. Gaia: Geo-distributed machine learning approaching LAN speeds. In *NSDI*.

[21] Intel. 2023. 13th Generation Intel® Core™ Processors Datasheet.

[22] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based parameter propagation for distributed DNN training. *Proceedings of Machine Learning and Systems* 1 (2019).

[23] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. 2020. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*.

[24] 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 *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*.

[25] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. 2014. Communication efficient distributed machine learning with the parameter server. *Advances in Neural Information Processing Systems* 27 (2014).

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

[27] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*.

[28] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer Sentinel Mixture Models. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Byj72udxe

[29] NVIDIA. 2022. NVIDIA DGX A100 : The Universal System for AI Infrastructure – nvidia.com. https://www.nvidia.com/en-us/data-center/dgx-a100/. [Accessed 22-09-2023].

[30] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU. https://nvidia.com/en-us/data-center/h100. [Accessed 27-Jun-2023].

[31] NVIDIA. 2023. NVLink & NVSwitch for Advanced Multi-GPU Communication. https://nvidia.com/en-us/data-center/nvlink. [Accessed 27-Jun-2023].

[32] NVIDIA. 2024. GPUDirect. https://developer.nvidia.com/gpudirect.

[33] 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*.

[34] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems* 24 (2011).

[35] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[36] Herbert Robbins and Sutton Monro. 1951. A stochastic approximation method. *The annals of mathematical statistics* (1951).

[37] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, et al. 2020. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*.

[38] 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*.

[39] Amazon Web Services. 2023. Compute – Amazon EC2 Instance Types – AWS – aws.amazon.com. https://aws.amazon.com/en/ec2/instance-types/. [Accessed 21-09-2023].

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

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

[42] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[43] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. *Advances in Neural Information Processing Systems* 31 (2018).

[44] Nikko Ström. 2015. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. In *Interspeech 2015*. https://www.amazon.science/publications/scalable-distributed-dnn-training-using-commodity-gpu-cloud-computing

[45] Guanhua Wang, Heyang Qin, Sam Ade Jacobs, Connor Holmes, Samyam Rajbhandari, Olatunji Ruwase, Feng Yan, Lei Yang, and Yuxiong He. 2023. ZeRO++: Extremely Efficient Collective Communication for Giant Model Training. *arXiv preprint arXiv:2306.10209* (2023).

[46] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. 2023. Hi-Speed DNN Training with Espresso: Unleashing the Full Potential of Gradient Compression with Near-Optimal Usage Strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems*.

[47] Jihao Xin, Ivan Ilin, Shunkang Zhang, Marco Canini, and Peter Richtárik. 2023. Kimad: Adaptive Gradient Compression with Bandwidth Awareness. In *Proceedings of the 4th International Workshop on Distributed Machine Learning*.

[48] Hang Xu, Chen-Yu Ho, Ahmed M Abdelmoniem, Aritra Dutta, El Houcine Bergou, Konstantinos Karatsenidis, Marco Canini, and Panos Kalnis. 2020. *Compressed communication for distributed deep learning: Survey and quantitative evaluation*. Technical Report. KAUST.

[49] Hao Zhang, Zhiting Hu, Jinliang Wei, Pengtao Xie, Gunhee Kim, Qirong Ho, and Eric Xing. 2015. Poseidon: A system architecture for efficient gpu-based deep learning on multiple machines. *arXiv preprint arXiv:1512.06216* (2015).

[50] Ruiliang Zhang and James Kwok. 2014. Asynchronous distributed ADMM for consensus optimization. In *International conference on machine learning*. PMLR.

[51] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).

[52] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. 2013. Asynchronous stochastic gradient descent for DNN training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE.

[53] Zhenwei Zhang, Qiang Qi, Ruitao Shang, Li Chen, and Fei Xu. 2021. Prophet: Speeding up Distributed DNN Training with Predictable Communication Scheduling. In *50th International Conference on Parallel Processing*.

[54] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. 2017. Asynchronous stochastic gradient descent with delay compensation. In *International Conference on Machine Learning*. PMLR.

[55] Yuchen Zhong, Cong Xie, Shuai Zheng, and Haibin Lin. 2021. Compressed communication for distributed training: Adaptive methods and system. *arXiv preprint arXiv:2105.07829* (2021).

[56] Martin Zinkevich, John Langford, and Alex Smola. 2009. Slow learners are fast. *Advances in neural information processing systems* 22 (2009).