CGX: Adaptive System Support for Communication-Efficient Deep Learning

Ilia Markov
ilia.markov@ist.ac.at
Institute of Science and Technology Austria
Klosterneuburg, Austria

Hamidreza Ramezanikebrya*
hamid@ece.ubc.ca
University of British Columbia
Vancouver, Canada

Dan Alistarh
dan.alistarh@ist.ac.at
Institute of Science and Technology Austria
Klosterneuburg, Austria

Abstract

The ability to scale out training workloads has been one of the key performance enablers of deep learning. The main scaling approach is data-parallel GPU-based training, which has been boosted by hardware and software support for highly efficient point-to-point communication, and in particular via hardware bandwidth over-provisioning. Overprovisioning comes at a cost: there is an order of magnitude price difference between "cloud-grade" servers with such support, relative to their popular "consumer-grade" counterparts, although single server-grade and consumer-grade GPUs can have similar computational envelopes.

In this paper, we show that the costly hardware overprovisioning approach can be supplanted via algorithmic and system design, and propose a framework called CGX, which provides efficient software support for compressed communication in ML applications, for both multi-GPU single-node training, as well as larger-scale multi-node training. CGX is based on two technical advances: *At the system level*, it relies on a re-developed communication stack for ML frameworks, which provides flexible, highly-efficient support for compressed communication. *At the application level*, it provides *seamless, parameter-free* integration with popular frameworks, so that end-users do not have to modify training recipes, nor significant training code. This is complemented by a *layer-wise adaptive compression* technique which dynamically balances compression gains with accuracy preservation. CGX integrates with popular ML frameworks, providing up to 3X speedups for multi-GPU nodes based on commodity hardware, and order-of-magnitude improvements in the multi-node setting, with negligible impact on accuracy.

CCS Concepts: • **Computing methodologies** → **Distributed algorithms**; *Neural networks.*

Keywords: Distributed Systems, Deep Learning, Gradients compression

ACM Reference Format:
Ilia Markov, Hamidreza Ramezanikebrya, and Dan Alistarh. 2022. CGX: Adaptive System Support for Communication-Efficient Deep Learning. In *23rd International Middleware Conference (Middleware'22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3528535.3565248

*Work performed during an internship at Institute of Science and Technology Austria.

Middleware '22, November 7–11, 2022, Quebec, QC, Canada

ACM ISBN 978-1-4503-9340-9/22/11.
https://doi.org/10.1145/3528535.3565248

1 Introduction

Deep learning has made significant leaps in terms of accuracy and performance, enabled by the ability to scale out workloads. Yet, distributed scalability of deep neural network (DNN) training still presents non-trivial challenges, and the last decade has seen a tremendous amount of work on distributed paradigms, algorithms, and implementations to address them [2, 9, 27, 33, 43]. Specifically, two key scaling challenges behind are reducing the *synchronization costs* among computing nodes [26, 27, 34, 43], and minimizing the *communication costs* which arise naturally due to the high bandwidth requirements of all-to-all transmission of model updates (gradients) between nodes. In this paper, we focus mainly on mitigating the *bandwidth cost* of gradient transmission in DNN training, which is an increasingly common bottleneck, correlated to the soaring parameter counts of modern machine learning models.

There are two main strategies for removing bandwidth bottlenecks. The *industrial approach* has been to employ *bandwidth over-provisioning*: for instance, the inter-GPU bandwidth for NVIDIA-enabled cloud-grade multi-GPU servers has increased by more than 30X between 2015 (Kepler generation) and the post-2018 Ampere generation, and has been complemented by a customized GPU-centric communication library, called NCCL, which leverages hardware support. Yet, bandwidth over-provisioning comes at significant hardware and development costs, reflected in the monetary cost borne by end-users: there is an almost order-of-magnitude cost difference between *cloud-grade*, overprovisioned multi-GPU servers such as NVIDIA DGX systems [19] and *commodity* workstations, built using consumer-grade GPUs (e.g. NVIDIA GeForce/RTX series). The latter have become extremely popular, due to lower costs and comparable single-GPU performance [20, 29, 30]; however, as we show, there are major performance gaps between the two in terms of scalability.

The alternative *algorithmic approach* builds on the fact that stochastic gradient descent (SGD), the standard algorithm for neural network training, can converge with *compressed* gradients. Several elegant lossy compression methods, such as gradient quantization [5, 46, 53], sparsification [14, 49], and gradient decomposition [51, 52], allow the theoretical bandwidth cost to be reduced by *up to two orders of magnitude* without accuracy loss. Despite their promise, realising these gains in practice runs into a number of significant challenges.

The first challenge is that of **parametrization and integration**: approaches such as gradient sparsification or decomposition often require non-trivial parameter and implementation changes to the training process, e.g. [36, 45, 51], to support compression. This would require practitioners to revisit their entire training setup, and tune additional hyper-parameters, in order to achieve compression while recovering accuracy. A second challenge is that of

efficient system support for communication-compression, as it often requires significant changes to lower levels of the software stack, such as supporting compressed or sparse data types. Despite research in this direction [6, 18, 55], the question of general and efficient system support for communication-compression is still open: currently, only one such approach, PowerSGD decomposition [51], is supported natively by one popular framework, PyTorch [42].

Contributions. In this paper, we introduce a communication framework called CGX, which addresses these challenges, and allows for *parameter-free, seamless integration* of communication-compression into data-parallel DNN training workflows, with up to order-of-magnitude speedups for data-parallel DNN training.

At the application level, CGX starts from an investigation of the feasibility of *parameter-free compression*: specifically, we implement and test all existing algorithmic approaches, and identify a variant of quantization-based compression that converges to *full accuracy* for many popular models, under *fixed, universal settings of parameters*, without modifying to the original training recipes. At the system level, we investigate how gradient compression can be seamlessly and efficiently integrated with modern ML frameworks. Specifically, we revisit the entire communication stack of modern ML frameworks with compression in mind, from a new point-to-point communication mechanism which supports compressed types, to compression-aware reductions, and finally a communication engine which interfaces with ML frameworks, supporting compression at the tensor/layer level.

The existence of a parameter-free compression technique which recovers accuracy, combined with the ability of CGX to customize the compression level per layer motivates a new *layer-wise adaptive compression problem*. The idea is that we can to customize the way model gradients are compressed in *layer-wise* fashion, so that the overall compression error is close to a given accurate baseline, but maximizing the bandwidth gains: for instance, one can apply more aggressive compression to layers that are larger, but less "sensitive" in terms of accuracy. While prior work has already considered techniques which adapt the degree of compression during training, e.g. [3, 37], this is the first instance of this problem to jointly considers both error and compression constraints at the fine-grained *per-layer* level. Our experimental results show that our layer-wise adaptive compression can bring significant additional gains.

To justify our design choices, we contrast our design against the first implementation of quantized collectives in NCCL, which we call QNCCL, which we contribute as a separate artefact, showing clear performance and usability improvements in favor of the CGX design. In addition, CGX does not require significant user-code or training pipeline changes, as we provide turn-key integrations with popular ML frameworks such as Pytorch and Tensorflow.

Experimental Validation. From the practical perspective, our work is motivated by the experimental data in Figure 1, showing that *bandwidth congestion* is the key scalability bottleneck on single-node, multi-GPU commodity servers, which have emerged as a popular training approach [20, 29, 30]. The same phenomenon occurs generally in multi-node data-parallel training settings, for a wide range of current and emerging training workloads, from image classification using classical convolutional neural networks (CNNs), to Transformer-based models for both language modeling [10, 50] and image classification [13, 41].

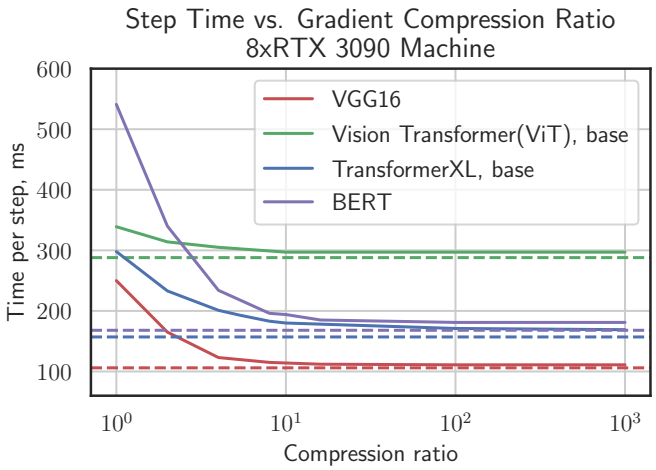


Figure 1. Compression vs. average step time for different models, when using all GPUs on an 8x RTX-3090 machine (Table 2). Dotted lines denote the throughput at perfect scalability for each model. Throughput nears ideal as we decrease transmission size, suggesting that bandwidth is the main bottleneck. See Section 2.1 for details.

We validate our system experimentally in both single-node and multi-node settings, across all of the above standard training tasks. We compare servers using commodity NVIDIA GPUs (RTX series) against cloud-grade NVIDIA servers from the Volta and Ampere architectures. (See Table 2 for details.) First, we find that, once communication bottlenecks are eliminated from "commodity" machines using CGX, they can match or *outperform* cloud-grade server with similar peak performance. Importantly, this can be done with negligible accuracy loss.

For example, we find that, on a commodity 8x RTX 3090 server, CGX can almost *triple* training throughput, reaching up to 90% of the ideal scaling, matching or even outperforming a bandwidth-overprovisioned (and more expensive) DGX-1 system. Our second application is to *multi-node training*, where we show up to 10x performance gains, enabled in part by our new solution to the adaptive layer-wise compression problem, without accuracy loss or additional parameters.

Our findings imply that hardware bandwidth overprovisioning may not be required for scalability in DNN training, and that highly-customized, hyperparameter-heavy compression techniques are not always necessary to remove bottlenecks. This should be immediately useful to users aiming to scale such workloads on commodity or multi-node hardware, but also more broadly for hardware/software co-design for distributed deep learning.

2 Motivation and Prior Work

2.1 A Motivating Experiment

The standard computational unit for DNN training is the multi-GPU node, usually in instances with 4–16 GPUs. End-users often rely on consumer-grade GPUs for training, whereas traditionally cloud services mainly employ cloud-grade GPUs, with some notable exceptions, e.g. [20, 29, 30]. We begin by briefly examining the scalability differences between cloud and commodity GPU servers. As

Table 1
Server-grade (first 2) vs. consumer-grade NVIDIA GPUs. Throughput obtained using the NVIDIA Deep Learning Examples benchmark [40]. TDP stands for Thermal design power.

GPU type	Arch.	SM	TensorCores	GPU Direct	GPU RAM, GB	TDP	ResNet50	Transformer-XL
V100	Volta	80	640	Yes	16	250 Watt	1226 imgs./s	37K tokens/s
A6000	Ampere	84	336	Yes	48	3000 Watt	566 imgs./s	39K tokens/s
RTX 3090	Ampere	82	328	No	24	350 Watt	850 imgs./s	39K tokens/s
RTX 2080 TI	Turing	68	544	No	10	250 Watt	484 imgs./s	13K tokens/s

Table 2
Systems characteristics of workstations used in evaluation.

System	GPUs	Inter-GPU link	Inter-GPU bandwidth	GPU RAM	RAM	CPUs
DGX-1	8xV100	NVLink	100 GBps	128 GB	512 GB	64
A6000	8xA6000	NVLink	100 GBps	384 GB	1008 GB	128
RTX-3090	8xRTX3090	None (bus)	15 GBps	192 GB	512 GB	128
RTX-2080	8xRTX2080 TI	None (bus)	15 GBps	96 GB	256 GB	72

we illustrate in Figures 5b and 5c, the maximum effective throughput of a cloud-grade 8-GPU DGX-1 server is $> 2\times$ higher than that of a comparable commodity 8xRTX-3090 GPU server, when using the same state-of-the-art software configuration (specifically, Horovod [47] on top of the NCCL communication library).

This gap is surprising, considering that the single-GPU performance is similar (see Table 1). To examine the specific impact of *gradient transmission / bandwidth cost*, we implemented a synthetic benchmark that reduces bandwidth cost by artificially compressing transmission. Specifically, assuming a buffer of size N to be transmitted, e.g. a layer's gradient, and a target compression ratio $\gamma \geq 1$, we only transmit the first $k = N/\gamma$ elements. The results for the 8x RTX-3090 machine, using all 8 GPUs, are shown in Figure 1, where the compression ratio is varied on the X axis, and we examine its impact on the time to complete an optimization step, shown on the Y axis. The dotted line represents the time per step in the case of ideal (linear) scaling of single-GPU times. We consider Transformer [10] and BERT-based models [12] for language modelling tasks, as well as VGG-16 [48] and Vision Transformer (ViT) models for classification on ImageNet.

We therefore observe that *bandwidth cost appears to be the main scalability bottleneck on this machine*. Moreover, recent models (Transformer-XL and ViT) benefit more from compression relative to the classic ResNet50 model, which has fewer parameters. Second, *there are limits to how much compression is required for scalability*, which depend on the model characteristics. An order of magnitude compression appears to be sufficient for significant timing improvements, although Transformer-based architectures can still benefit from compression of up to two orders of magnitude.

Discussion. The reason for this poor scalability is the lack of efficient communication support. Specifically, GPU-to-GPU transmissions on commodity hardware have significantly lower bandwidth, and higher latency, relative to their cloud counterparts. Specifically, in software , the NVIDIA GPUDirect technology should allow GPUs on the same machine to communicate directly, without the need for extra memory copies. Commodity GPUs, such as the RTX 3090, do not support this technology. At the same time, the hardware communication support for NVIDIA GPUs, i.e. NVLink and NVSwitch components, is also not available or severely restricted for commodity GPUs [1, 24].

2.2 Data-Parallel DNN Training

Distribution Strategies and Costs. Training a DNN essentially minimizes a loss function, related to the error of the model on the dataset, via a sequence of optimization steps, each acting on some data samples. To preserve computational efficiency, it is common to perform a *batched* version of this process, by which several samples are processed in a single optimization step, and the sum of gradients is applied.

Data-parallelism is arguably the standard way to scale DNN training, and can be viewed as a variant of batch SGD in which sample gradients are generated in parallel over compute nodes. Specifically, the dataset is partitioned over nodes, each of which maintains a copy of the model, and computes gradients over samples in parallel. Periodically, these gradients are aggregated (e.g., averaged) and the resulting update is applied to all local models.

Several techniques have been proposed to address the synchronization and communication costs inherent to this lock-step averaging procedure. Here, we focus on *communication/bandwidth cost*, and assume that synchronization preserves the synchronous ordering of gradient iterations, although our techniques are also compatible with other scheduling strategies, e.g. [26, 27, 43, 56].

Batch Scaling. An orthogonal scaling approach is increasing the batch size at each node. This requires careful hyper-parameter tuning for accuracy preservation, e.g. [21, 57], although recipes for large batch scaling are known for many popular models. We consider scalability in both 1) *the large-batch setting*, where we adopt the best-known hyperparameter recipes to preserve accuracy, and 2) *the small-batch setting*, corresponding to datasets or models for which large-batch scaling parameters are unavailable or unknown.

2.3 Communication Compression Methods

The basic idea behind communication-compression methods is to reduce the bandwidth overhead of the gradient exchange at each step by performing lossy compression. Our presentation assumes that a generic mechanism allowing for all-to-all communication among the nodes is available. (We discuss our implementation choices in Sections 3 and 4.) Roughly, existing schemes can be classified as follows.

Gradient Quantization. This approach works by reducing the bit-width of the transmitted updates [46]. One of the first compression approaches [5] observed that *stochastic* quantization of the gradient values is sufficient to guarantee convergence. Their

method, called QSGD, is a codebook compression method which quantizes each component of the gradient via randomized rounding to a uniformly distributed grid. Formally, for any non-zero vector $\vec{v}$, given a codebook size s and $\vec{v} \in \mathbb{R}^d$, $Q_s(v_i) = \|\vec{v}\|_2 \cdot sign(v_i) \cdot q(v_i, s)$. The stochastic quantization function $q(v_i, s)$ essentially maps the component's value v_i to an integer quantization level, as follows. Let $0 \leq \ell \leq s-1$ be an integer such that $|v_i|/\|\vec{v}\| \in [\ell/s, (\ell+1)/s]$. That is, ℓ is the lower endpoint of the quantization interval corresponding to the normalized value of v_i. Then,

$$q(v_i, s) = \begin{cases} \ell/s, \text{with probability } 1 - p(|v_i|/\|\vec{v}\|, s), \\ (\ell+1)/s, \text{otherwise} \end{cases}$$

where $p(a, s) = as - \ell$ for any $a \in [0, 1]$. The trade-off is between the higher compression due to using a lower codebook size s, and the increased variance of the gradient estimator, which in turn affects convergence speed. This idea inspired a range of related work [16, 35, 44] reducing the variance of the compression by improved quantizers. We discuss these schemes further in Section 4.

Gradient Sparsification. These methods, e.g. [14, 28, 36, 49], capitalize on the intuition that many gradient values may be skipped from transmission. The standard approach to sparsification is *magnitude thesholding*, effectively selecting the top K gradient components for transmission, where K is a hyper-parameter. Then, error correction is applied to feed the thresholded gradient components back into the next round's gradient. Variants of this procedure can achieve more than 100× gradient compression while still recovering accuracy [36]. However, this comes at the price of model-specific hyper-parameter tuning, which may be unreasonable in a deployment setting.

Renggli et al. [45] proposed efficient sparse collectives, and observed that sparsification methods can be promising in cases where there is high natural redundancy–such as fully-connected or embedding layers–but may be a poor choice for general compression due to the need for hyper-parametrization. Our investigation confirmed their finding.

Gradient Decomposition. This approach treats the gradients as multidimensional tensors, and decomposes the gradient matrix $G \in \mathbb{R}^{m \times n}$ into 2 rank-r matrices $P \in \mathbb{R}^{m \times r}$ and $Q \in \mathbb{R}^{r \times n}$, with r much smaller than m and n. ATOMO [52] uses singular value decomposition (SVD) to find the matrices P and Q. However, in the case of large models, the SVD of gradient matrices becomes too compute-intensive to be used during training. PowerSGD [51] uses a generalized power iteration algorithm to calculate the matrices P and Q, and is the fastest currently-known factorization method. To recover accuracy, it applies a combination of error correction techniques. Their results show that these methods can be highly useful in the case of CNNs, yielding high compression ratios (up to 100×). However, in our experience, recovering accuracy in e.g. Transformers training requires careful tuning, and higher rank values, resulting in lower performance.

Adapting Compression during Training. The idea of adapting the degree of compression during different stages of DNN training has been considered by [7, 8, 23, 37]. However, we emphasize the fact that all these references in practice *globally adapt* the amount of gradient compression for the entire model to preserve end accuracy, whereas we investigate mechanisms which adapt compression at the per-layer level. Moreover, to achieve high compression, some existing methods require hyperparameter tuning [8]. A work[3] that supports per-layer compression parameters has a very limited choice of compression parameters (namely picks out of two parameters), requires additional hyperparameter tuning and focuses on specific architectures. By contrast, we adapt compression parameters automatically both across layers, and across training iterations.

Efficient Software Support. There has already been significant work on providing system support for compression. Two main challenges are: 1) the introduction of additional hyper-parameters in the training process, and 2) the fact that, since most compression methods are *not associative*, they are not directly supported by standard collective implementations and require algorithm-specific re-implementations. Grubic et al. [22] showed that CNNs can withstand 8-bit gradient compression, and provided a simple MPI-based implementation of quantization, while Dutta et al. [15] examined the implementation gap, showing that frameworks should support both global and per-layer compression. Renggli et al. [45] and Fei et al. [17] provided efficient support for sparse reductions, while the GRACE framework [55], Bagua [18] and HiPress [6] frameworks provided efficient implementations of communication-compression methods. We compare against these frameworks in Section 6.

We differ from this prior work in two major directions. At the application level, we focus on *seamless, parameter-free integration* with existing data-parallel training pipelines: thus, we investigate compression techniques which allow accuracy recovery *without additional hyper-parameter tuning*. This is not the case with prior frameworks, which leave the choice of compression parameters to the user. Second, at the system level, we seek to maximize speedup by rewriting components of the communication stack to support compression, provide an adaptive layer-wise compression solution which maximizes speedup.

Recent work by [4] investigated the practical potential of gradient compression methods in cloud-grade settings. They provide analytical and empirical evidence suggesting that gradient compression methods can only provide marginal speedups in distributed data-parallel training of DNNs in such bandwidth-overprovisioned settings.

However, the generality of their results is restricted by the following factors: 1) they only consider a limited subset of compression methods and possible implementations: for instance, their compressed implementations strictly follow the NCCL API, which, as we illustrate via our QNCCL implementaiton, means that the compression methods were used inefficiently and with accuracy loss; 2) they focus on cloud-grade bandwidth-overprovisioned systems, and therefore their findings do not apply to the popular setting of commodity servers. These two factors, as well as additional implementation differences, explain the difference between their conclusions and the ones from this work.

3 Goals and Challenges

The results in Section 2.1 suggest that bandwidth can be a key bottleneck when attempting to scale DNN training on commodity GPUs, while the discussion in Section 2.3 outlines non-trivial trade-offs when implementing these techniques for general models. We therefore outline our key goals:

1. **Accuracy Recovery:** Similar to MLPerf [39], we set our accuracy loss threshold at $< 1\%$ relative to the main metric of the full-precision baseline (e.g. Top-1 classification accuracy),

Table 3
Compression approaches. Stateful here means that approach requires maintaining of a state of error compensating techniques.

	Compression rate with recovery	Tunable Parameters	Properties	Computational Overhead
Quantization	~ 8x	Bits, bucket size	Non-associative, stateless	≤ 3%
Sparsification (TopK)	~ 100x	Sparsity, momentum	Non-associative, stateful, not overlapping with compute	10%
Decomposition (PowerSGD)	~ 100x	Rank, warm-up	Associative, stateful, incompatible with mixed precision	20%

although in most of the tasks we present the accuracy loss is practically negligible.

2. **Hyperparameter-Freedom:** Second, we wish to enable scalable data-parallel DNN training in the absence of any model or task information, recovering accuracy under *standard (uncompressed) hyper-parameters.*
3. **Eliminating Bandwidth Bottlenecks:** Third, we aim to mitigate or even completely eliminate bandwidth constraints. Since not all target models are equally communication-bottlenecked, this allows us some flexibility with respect to how much compression to apply depending on the model and application.
4. **Simple Interface:** Finally, the integration with the underlying training framework should be seamless.

State of the art. We executed implementations of the compression methods described in Section 2.3 on a range of modern tasks and models. Our findings are summarized in Table 3, and discussed in detail below.

We found that no existing approach fully satisfies all the above requirements. For instance, *quantization-based methods* are known recover accuracy on CNNs when using 8-bit compression [22], meeting Goals 1 and 2. However, this amount of compression is not sufficient to remove the bandwidth bottlenecks for modern Transformer-class models (Goal 3); moreover, the parameters of [22] do not allow full accuracy recovery on Transformers.

Second, examining *gradient sparsification* methods, we notice that they can ensure high compression (Goal 3); however, they require complex hyperparameter tuning for accuracy recovery in the high-compression regime [36], breaking either Goal 1 or Goal 2. Conversely, as also noted by [45], these methods can recover accuracy under medium density (e.g. 20%), but in that case their performance is similar to quantization approaches. This family of methods has the additional cost of having to maintain state (the error buffer) and being less amenable to computation-communication overlap, since the selection operation is applied over the entire gradient.

Finally, *decomposition* methods have been shown to yield compression ratios of up to 100× in the case of CNNs, attaining Goal 3. Moreover, with careful tuning of hyper-parameters, PowerSGD is able to recover accuracy for CNNs under generic rank-decomposition values. In addition, this method is *associative*, lending itself to seamless implementation via MPI or NCCL (Goal 4). Unfortunately, however, we found that this method can require high rank values for stable training, especially on Transformers, where there is almost no speedup, and that it is not compatible with reduced-precision (FP16) training, which is used by virtually all frameworks.

4 CGX System Design

ML Frameworks under the Hood. A typical DNN training framework has three parts, as described in Figure 2:

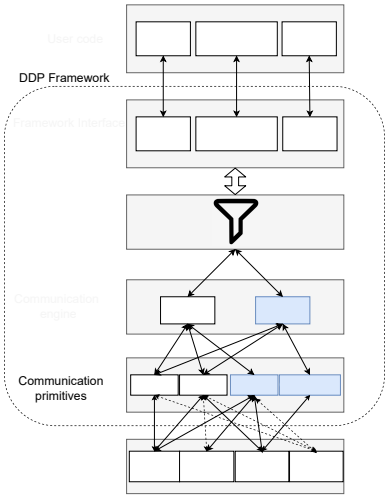


Figure 2. Abstract architecture of a Distributed Data Parallel (DDP) framework. CGX components are in blue, and arrows stand for procedure calls. Dashed arrows represent hardware interactions, e.g. P2P transport is supported via GPU NVLinks.

1. **Framework interface** (in Python) with high-level API is called by User code. It may also include a frontend that unifies the input from the learning framework.
2. **Background thread** collects inputs, groups them into blocks based on query type and input properties, schedules the reduction for each block.
3. **Communication engine** performing the query (Allreduce, Broadcast, Allgather). At this stage, the framework typically calls an existing communication library, such as NCCL (QNCCL), Gloo, or an MPI implementation.

A key issue when implementing most compression methods such as quantization or sparsification is that their operations are *non-associative*, and so the aggregation function (sum) must be performed at the lowest level in the above diagram. This means that we cannot integrate the compression into higher levels without a bespoke implementation, which in turn may lead to performance and implementation costs.

4.1 The CGX Communication Engine

To efficiently support compression, we implemented our own communication engine (blue component on Figure 2), with primitives which support non-associative compression operators. Broadly, there are two approaches to do this. The first is a *native* one, by

which one can implement compression-aware Allreduce using communication libraries. Alternatively, one can modify or extend existing communication libraries, such as NCCL, to support compression operators.

The native approach requires deeper integration, but has the advantage that compression is performed "closer" to training, which means that the compression engine has information about the model layers, and their gradients and thus has a richer, more flexible API. The disadvantage is that it has to explicitly interface with the training framework, and users may have to adjust their training pipeline.

The second *low-level* approach is to directly perform compression and de-compression at the primitive/transport level, independently of the user's code and training pipeline. In this case, the framework can only operate with the raw data buffers provided by the upper layers. This loses information about the data it operates with, e.g., layer names, which could be useful for compression operators, but is easier to interface with, and may have lower overheads.

4.1.1 Framework Integration. To investigate this non-trivial dichotomy, we implemented *both* variants. Specifically, our main framework, called CGX, integrates natively with the user's code, and can interface both via Horovod [47], a popular distribution wrapper that works with all major ML frameworks, but also separately via framework-specific extensions, such as PyTorch Distributed Data Parallel (DDP). Separately, as an instance of the "low-level" approach, we re-implemented the NCCL communication library to support quantized reduction operations. We call this separate implementation QNCCL, and contrast it to our main approach.

The Native CGX Framework. The main version of CGX uses the Horovod wrapper [47] to interface with popular ML frameworks. Specifically, we implemented a communication engine with Allreduce methods supporting compression operators. Next, we added layer filters that split model gradients into logical subsets, which the framework may handle differently: some accuracy-critical subsets are communicated in full precision, while other subsets are compressed and reduced in lower-precision. Empirically, it is known that layers like batch/layer normalization and bias layers are sensitive to gradient compression, while being small. Therefore, we communicate them uncompressed. As a bonus, this avoids calling compression operators for multiple small inputs. At the filtering level, the framework also performs packing or splitting of levels into the units of communications, so called fused buffers. Typical size is around 64MB. The communication engine then performs reduction with these units not the layers. But it keeps the information of offsets of the layers within the fused buffers because this information will be used for layer-wise compression.

Further, CGX performs compression *per-layer*, and not as a blob of concatenated tensors. This provides the flexibility of exploring heterogeneous compression parameters and avoids mixing gradient values from different layers, which may have different value distributions, leading to large quantization error. We found that such filters can be applied "at line rate" without loss of performance, as most of the computation can be overlapped with the transmission of other layers. CGX's API allows users to choose the compression parameters for specific layers or filter out the group of layers.

Torch DDP Integration. Our compression/communication engine is portable: to illustrate this, we also integrate it separately with the Torch DDP pipeline [42]. In this case, CGX acts as a Torch extension that implements an additional Torch DDP backend, as a supplement to the built-in NCCL, MPI and Gloo backends. Thus, users only need to import the extension and change the backend at initialization.

We integrated our functionality into the communication engine of the Data Parallel framework. At this level, we no longer have access to the buffer structure, therefore we can not explicitly filter layers. Nevertheless, the user can provide the layout of the model layers (e.g. gradient sizes and shapes). Using this information, we can obtain the offsets of the layers in each buffer provided by `torch.distributed`.

4.1.2 Choosing a Reduction Scheme. The "hottest" operation in distributed data-parallel training is Allreduce, corresponding to the logical gradient averaging. To support non-associative compression operators, we need to choose the reduction algorithm together with the compression operator, to maximize performance and minimize the compression error due to iterative compression-decompression. We considered the following reduction schemes.

Scatter-Reduce-Allgather (SRA) works in two rounds: a process first divides its vector of dimension d into N subarray "chunks;" each node receives its chunk of the initial vector from all other nodes and aggregates it (Scatter-Reduce). Second, it broadcasts the aggregated chunk (Allgather). The bandwidth cost is $O(d(N-1))$, the latency term is 2α, corresponding to the two rounds. **Ring-Allreduce** is the bandwidth-optimal algorithm, implemented in most libraries (e.g. NCCL, Gloo). Similar to SRA, it divides the initial vector into chunks, and communication is done in a ring-shaped topology. In the first phase, each node sends a chunk to its "right" neighbor and receives a chunk from its left neighbor. It then sums the received chunk with its local result and sends the result forward, repeating $N-1$ times. In the second phase, nodes broadcast (Allgather) the resulting chunks on the ring. The bandwidth cost is $O(d(N-1)/N)$, with latency $2\alpha(N-1)$, assuming communication can not be itself parallelized. **Tree-Allreduce** can be seen as a hierarchical parameter-server. Communication is done in $2\log N$ rounds and two phases. The nodes build a tree-like topology, and send their vectors up to the root, summing them along the path, and then propagate back the result. Communication complexity is $O(2d \times \log(N))$, while latency is $2\alpha \log N$.

Discussion. We examined the practicality of these reductions, and found **Scatter-Reduce-Allgather (SRA)** to show the best performance. It also has the key algorithmic advantage of *lower compression error*, due to fewer compression/decompression steps. Thus, we mainly employed this algorithm inside CGX. Table 4 illustrates CGX throughput under different reduction schemes, for different tasks, on an 8-GPU server. (See Section 6 for the full setup.)

Table 4
Throughput of different reduction schemes (items per second).

	ResNet-50	Transformer-XL	ViT
SRA	**2900**	**260k**	**1918**
Ring	2830	236k	1883
Tree	2770	202k	1756

4.1.3 Default Compression Approach. Our framework implements several compression approaches; yet, based on the discussion in Section 2.3, we use *gradient quantization* as our main method.

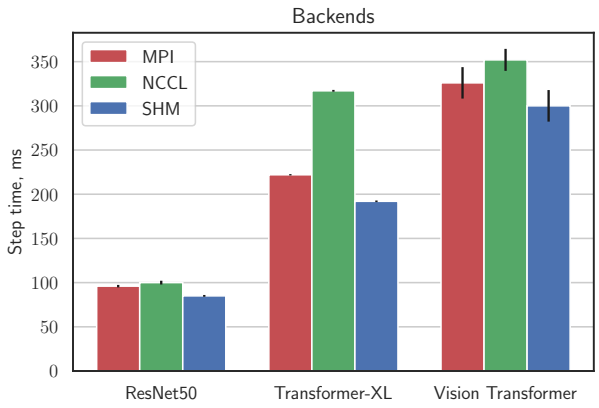
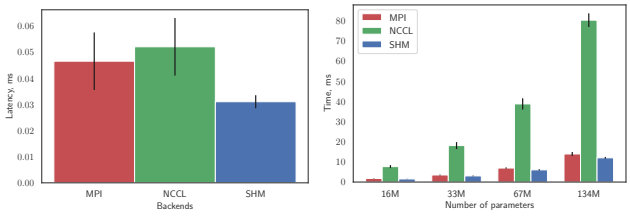


Figure 3. Training step times for different communication backends in CGX Communication engine on a single node, 8 RTX3090 GPUs. Lower is better.



(a) Communication latency. **(b)** Communication time.

Figure 4. Comparison of point-to-point communication using different backends.

The rationale behind our choice is the following. First, as suggested by Figure 1, quantization compression by 8-10x should provide sufficient bandwidth reduction to overcome most of the communication bottleneck. Moreover, it can do so *in a generic, parameter-free way*: an independent contribution of our work is that we identify *general parameter values providing 8-10x compression without accuracy loss on all the model classes and tasks we tried.* We investigate additional performance improvements customized per-layer compression, which can provide an additional performance boost.

4.2 Communication Backend

The key question at the lower level of the stack is how to implement the point-to-point communication primitives. Here, existing options are GPU-aware MPI implementations, NCCL, or Facebook Gloo. (For instance, GRACE [55] supports all three options.) To maximize performance, instead of relying on existing implementation, we developed a set of new point-to-point communication primitives, that are based on data transfers through UNIX shared memory. We call this communication backend SHM.

SHM works by registering a UNIX shared memory buffer for each pair of GPUs within a node and mapping it to GPU memory. On send, we move the input buffer to the shared segment and synchronize with the recipient using CUDA IPC primitives. At SHM communicator initialization, we allocate an 2 auxiliary buffers for each one directional point-to-point communication. The size of the buffer is the size of the fused buffer, i.e. 64 MB. It means that for 8 GPU communication (e.g. for SRA all All-to-All communication) we allocate $128 * 7 = 896MB$ on each GPU. SHM is only supported for a single server, while CGX can use both MPI- and NCCL-based backends in multi-server setups. Moreover, we support heterogeneous communication where the intra node communication uses SHM, MPI, or NCCL as the backend, or performs NCCL-allreduce without compression, while the inter-node communication uses MPI or NCCL. The difference in performance is illustrated in Figures 3 and 4. The speedup is justified by the lower synchronization between compression and communication, and the memory transfers via the GPU Communication Engine. In Figure 4a we show timings for point-to-point communication of small buffers, whereas the Figure 4b shows the communication time dependency on large buffers sizes. The figures demonstrate that SHM significantly outperforms other backends. Thus, unless otherwise stated, we use SHM for intra-node communication in all our experiments.

4.3 Implementation Details

Efficient Quantization. The quantization algorithm sketched in Section 2.3 has the following downside: when applied to the entire gradient vector it leads to convergence degradation, due to scaling issues. A common way to address this is to split the vector into subarrays, called buckets, and apply compression independently to each bucket [5]. This approach increases the compressed size of the vector because we have to keep scaling meta-information for each bucket and slows downs the compression, but helps to recover full accuracy. The bucket size has an impact on both performance and accuracy recovery: larger buckets lead to faster and higher compression, but higher per-element error. Therefore, one has to pick the bucket size appropriate for the chosen bits-width empirically. We found out that 4 bits and 128 bucket size always recovers full accuracy, has reasonable speedup, and can be efficiently implemented, so we use this as a compression baseline in all our experiments.

To achieve low compression overheads, we applied the following optimizations: we use an efficient parallel bucket norm computation algorithm, and, for elementwise compression/decompression, we perform cache-friendly vectorized memory load/stores. Quantization overhead amounts to 1-3% of computational cost in our benchmarks.

Improved Scheduling. CGX also aims to improve the latency term. For this, we perform fine-grained scheduling of gradient synchronization, which is known to lead to improved performance for Parameter Servers [27]. The scheduling of the communication is task-based, where each task are layer gradients that we want to synchronise every iteration. In the background thread we collect the tasks until the total size of collected gradients reaches user-defined size B or user-defined cycle time C expires. Then the concatenated group of gradients is synchronised. The constants B and C are autotuned; we leveraged parts of this implementation from Horovod and `torch.distributed`. As part of scheduling optimization, CGX supports user-defined filtering of layers and cross-barrier training. Filtering of small layer modules such as biases or batch norm not only improves convergence, but positively affects performance. Such filtering removes the need of extra compression kernels calls without notable increase of communication costs. Cross-barrier

optimization does not provide significant performance in a single node setup, confirming the observations in [27].

4.4 The QNCCL Library

The role of the QNNCL implementation is to contrast our design choices relative to a direct re-implementation of communication compression in the popular NCCL library. To build this low-level variant, we started from vanilla NCCL and replaced Allreduce with implementations that compress every piece of data before its transfer. Basically, in DDP stack(Figure. 2) we replace NCCL with our version that supports compression. We leverage the NCCL communication optimizations, to avoid costs for additional GPU calls. However, in this case, we lack information about the internal structure of the buffer, and have to apply compression parameters uniformly over the entire model. In this case, we also have limitations in terms of the GPU resources imposed by NCCL itself, which lead to additional compression overheads. We examine the performance trade-offs of this approach in the experimental section.

5 Layer-wise Adaptive Quantization

One key optimization supported by CGX is *varying compression parameters at the per-layer level*. This is especially well-suited to models such as Transformers which have heterogeneous layer sizes, e.g. due to large embeddings. Synchronization of such layers can be quite expensive, and, since they come early in the model, cannot be overlapped with computation. Yet, these massive layers can support highly-compressed communication. Thus, we investigate *automatic* mechanisms to pick per-layer compression levels.

We focus on the trade-off between two parameters for each layer: the *magnitude of the compression error* and *compressed size of the layer*. Our adaptive algorithm tries to balance these constraints in order to maximize speedup while recovering convergence. We periodically collect gradient statistics and then re-assign bit-widths and bucket-size to each layer. Specifically, we want to minimize the compressed size of the model gradients, while minimizing the ℓ_2-norm of the compression error, which is linked to convergence [28].

Problem Definition. We formalize this problem as identifying per-layer bit-widths $b_1, b_2, \ldots, b_L$ for the L layers minimizing the *bandwidth objective* $\sum_{\ell=1}^{L} b_\ell \cdot size(L_\ell)$ across all the b_is, *subject to* the fact that compression error cannot not exceed a maximum threshold $\alpha \cdot E_4$. Here, $\alpha > 0$ is a fixed parameter, and E_4 is the error when we compress all layers to 4 bits, for which we know that full recovery occurs.

We emphasize that this formulation is different from the (global) adaptive compression problems considered by prior work [3, 7, 8, 23, 37], as they usually consider the problem of adapting the global degree of compression to the various stages of the training process, as opposed to optimization of the fine-grained layer-wise bit-width adaptation we consider.

This constrained optimization problem can be approached via standard solvers, and in fact our first approach has been to use Bayesian optimization. However, we found that this requires instance-specific tuning, and adds hyper-parameters. We therefore investigate problem-specific heuristics.

A straightforward such approach is to simply sort layers by the ratio of gradient magnitude over the layer size. We then assign the lowest bit-width to the first layers in this order, and the highest to the last layers, interpolating linearly in the middle. Experimentally, this approach recovers accuracy and improves over static assignment, but the performance gains are minor.

Algorithm 1 KMEANS-based adaptive compression

Input: Model Layers L_i, accumulated gradients G_i, possible bit-widths $B = \{\beta_1, \beta_2, \ldots, \beta_k\}$
Output: Bit-width assignments $b_\ell \in B$ for each layer ℓ
Initialisation: Compute 2D-representation for each layer ℓ by computing points $(size(L_\ell), norm(G_\ell))$.
1: Obtain (centroids, clusters) = kmeans over data into k clusters
2: Sort centroids based on $norm(C_i) - size(C_i)$ and assign them
3: Assign points (layers) corresponding to each centroid to the corresponding bit width b_ℓ.

This observation inspires a *clustering-based* approach, by which we collect layers with similar sensitivity to gradient compression into groups, and assign bit-widths correspondingly. We use a 2D-clustering algorithm [38], where the dimensions are the size of the layer, and the ℓ_2-norm of the top values of the accumulated gradient. We perform clustering to obtain "sensitivity groups," each with its own centroid, and then sort the centroids by their gradient norms. Finally, we linearly map bit-widths and bucket sizes to the layers. The exact procedure is described in Algorithm 1. We investigate its practical performance in Section 6.3.

6 Experimental Validation

6.1 Experimental Setting

Infrastructure. Our evaluation uses commodity workstations based on RTX2080 and RTX3090 consumer-grade GPUs, and a cloud-grade EC2 `p3.16xlarge` machine, with 8 V100 GPUs, equivalent to a DGX-1 server. Please see Table 2 for complete system characteristics. In brief, the 8 GPUs are split into two groups, each assigned to a NUMA node, which are bridged via QPI. Bandwidth measurements via [32] show that inter-GPU bandwidth varies from 13 to 16 GBps depending on location. At the same time, we have 1GBps Allreduce bandwidth for reasonable buffer sizes. Results for RTX2080 are similar, with 1.5GBps Allreduce bandwidth.

The V100/DGX-1 machine forms a so-called *Backbone Ring* inside a *Hypercube Mesh* [31], in which GPUs are connected via NVLINK. The DGX-1 has GPU-to-GPU bandwidth of up to 100 GBps, leading to the same Allreduce bandwidth our workloads. Performance on our setup is identical to a branded DGX-1 measured via NVIDIA's benchmarks [40].

Environment and Tasks. Most experiments were run using the PyTorch version of the NVIDIA Training Examples benchmark [40]. For state-of-the-art model implementations we used the Pytorch Image Models [54] and the Huggingface Transformers repositories [25]. For the experiments on V100 machine we used the official NGC PyTorch 20.06-py3 Docker image. We used CUDA 11.1.1, NCCL 2.8.4, and cudnn/8.0.5. We examine three different DNN learning tasks: 1) image classification on ImageNet [11] ; 2) language modeling on WikiText-103; 3) question-answering on the SQUAD dataset.

Baselines. We use the non-compressed original training recipes as a baseline. We *do not* modify any of the training hyper-parameters. In distributed training, we use either Horovod-NCCL or PyTorch-DDP with NCCL backend. In all our experiments, NCCL showed

Table 5
Validation results for training with the baseline and CGX optimizations, respectively. ResNet50, VGG and ViT numbers are Top-1% accuracies, Transformer-XL and GPT-2 show perplexity, while BERT shows F1-score.

	ResNet50	VGG16	ViT-base	Transformer-XL-base	GPT-2	BERT
Baseline	75.8 ± 0.2	69.1 ± 0.1	79.2	22.81 ± 0.1	14.1 ± 0.1	93.12 ± 0.05
CGX	75.9 ± 0.2	68.9 ± 0.1	78.6	22.9 ± 0.1	13.9 ± 0.1	93.06 ± 0.05

Table 6
Training throughput with CGX, PowerSGD, and GRACE on single machine with 8 RTX3090 GPUs. (Transformer-XL/PowerSGD did not converge, so we only provide throughput numbers.)

	ResNet50	Transformer-XL-base	BERT
Baseline	1900	170k	17.5k
CGX	**2900**	**260k**	**38.7k**
PowerSGD	2600	220k*	38.3k
Grace	1000	30k	14.3k

better performance than OpenMPI or Gloo, so we use it as the default backend. For a fair comparison, we use the CGX extension depending on the baseline framework: for Horovod-NCCL, we use our Horovod extension, and for PyTorch-DDP we apply our Torch distributed backend extension. We also compare our results against ideal linear scalability on the same machine, calculated by training speed on a single device multiplied by the number of devices. We use step time and throughput (items/sec) as the performance metrics. For all performance experiments, we validated that the hyper-parameters used are sufficient to recover training accuracy, across 3 runs with different seeds. All the reported speed numbers are averaged over 300 training iterations after a warm-up of 10 iterations. Unless specifically stated, we *do not* employ the adaptive compression algorithm.

6.2 Experimental results

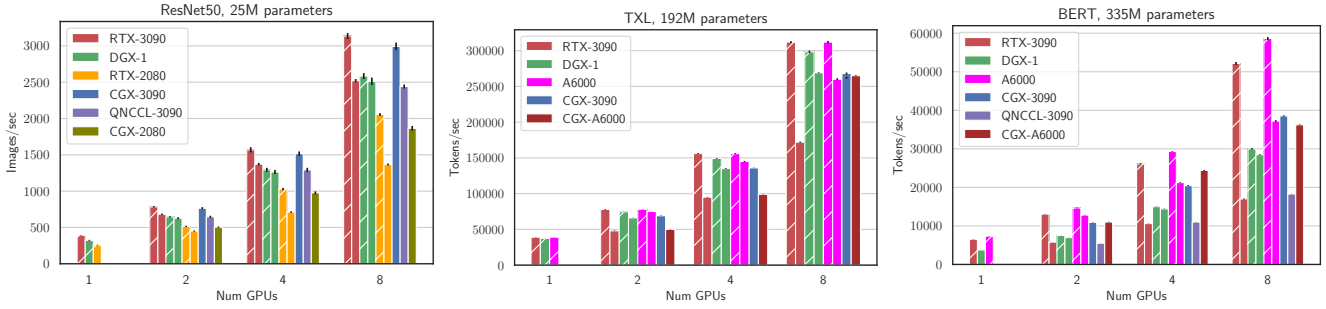
6.2.1 Accuracy Recovery. We first examine the model accuracies using standard hyper-parameters in end-to-end training experiments. The gradient bit-width used for these experiments is 4 bits. The bucket size was 1024 for CNNs, and 128 for Transformer models, chosen empirically. As stated, we reduce small layers (biases, batch and layer normalization layers) in full precision. The results of training on the RTX3090 machine with 8 GPUs are presented in the Table 5, with the corresponding accuracy parameters. All CGX accuracy results are within the standard 1% error tolerance [39]; in most cases, accuracy is within seed random variability.

Following the original recipes, ResNet50, VGG16, and the Vision Transformer (base model) were trained on ImageNet with total batch sizes 256, 256, 576 respectively. ViT was trained in mixed precision level 1 (activations at FP16, weights, and gradients in full precision). The Transformer-XL (base model) experiment was run on WikiText-103 dataset with batch size 256 and second level mixed precision (model, activations, and gradients cast FP16). The GPT-2 model was trained on WikiText-2, batch size 24, level 2 mixed precision. For question-answering we used BERT model on the SQUAD-v1 dataset with batch size 3 per GPU and FP32 training.

Unless otherwise stated, we focus on following model/task combinations: Transformer-XL on WikiText-103, ResNet50 on ImageNet, and ViT on ImageNet. The parameters are identical to the ones provided above. All experiments were run on 8 GPUs.

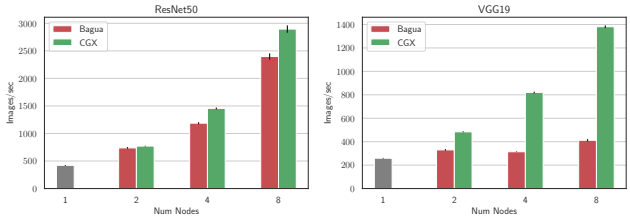
6.2.2 Comparison with other algorithmic approaches.
PowerSGD Compression. We follow the implementation of [51], and set the rank to 4 for CNNs and use rank 8 for Transformers, implying up to 100x compression. PowerSGD can not be used in conjunction with FP16 training, as it can lead to divergence in our experiments, so we compare at FP32. But with full-precision gradients training PowerSGD can not achieve baseline accuracy at Transformers pre-training (we tried ranks up to 32). As Table 6 shows CGX has superior performance on single node over PowerSGD in spite of lower compression. This is because 1) higher compression shows diminishing returns, 2) CGX has lower compression overhead (Table 3), and 3) CGX implements faster reductions.
Sparsification. We implemented the TopK [14] algorithm as part of CGX framework. Usage of the sparcification compression there faces following issues. In order to converge under standard paramters, sparcification must be applied upon entire model, not layer-wise which is impossible due to specifics of the communication frameworks (`torch.distributed`, `Horovod`). In our experiments we did not manage to make topK with error feedback converge with similar to QSGD compression rate. Moreover, topK with higher compression rates did not show any speedup in comparison to QSGD due to compression saturation on our workstation (see Figure 1) and higher topK overhead (see Table 3, we used [37] topK mechanism).

6.2.3 Comparison with other systems.
GRACE Comparison. We adapted our benchmarks to also compare to GRACE [55], which also implements quantization and sparsity compression techniques. We used the same uniform 4-bit compression variant for frameworks, as this recovers accuracy. We used NCCL as the communication backend for GRACE, as it provided the best performance in our setting. We found (see Table 6) that CGX outperforms GRACE by more than 3x on average. Our profiling suggests that this occurs because GRACE uses a less effective reduction scheme (NCCL-Allgather vs. optimized Allreduce), less efficient compression (e.g., no bucketing) and transmission (even with 4 bits compression, GRACE communicates in INT8). We also tried GRACE with very high-sparsity TopK compression (0.001), and performance did not improve significantly. This suggests that GRACE's implementation has additional bottlenecks in terms of communication latency.
NCCL and QNCCL Comparison. As shown in Figure 1, NCCL has poor scaling on commodity machines, especially from 4 to 8 GPUs, where communication cost is highest. CGX can give > 2x speedup relative to NCCL, reaching 80-90% of linear scaling. This enables the consumer-grade RTX3090 GPU to match or even surpass the throughput of a DGX-1 server. We found that QNCCL partly alleviates the scaling problems of NCCL, and only improves throughput by a limited margin, as it does not benefit from the bespoke communication backend integrated in CGX. An orthogonal issue for QNCCL is the fact that it has higher accuracy degradation: since compression cannot be performed layer-wise (as QNNCL does not have layer information), it cannot perform layer-wise compression. We have been able to recover accuracy within 1%



(a) ResNet50, ImageNet **(b)** Transformer-XL, WikiText103 **(c)** BERT, SQUAD

Figure 5. Throughput for ResNet50/ImageNet, Transformer-XL (TXL) on WikiText, and BERT on SQUAD. Higher is better. Hatched bars represent ideal scaling. CGX leads to self-speedups of > 2×, and scalability of 80% to 90%. Hatched bars represent ideal scaling.



(a) ResNet50 **(b)** VGG19

Figure 6. Scaling throughput in multi-node environment for image classification tasks. Bagua vs CGX.

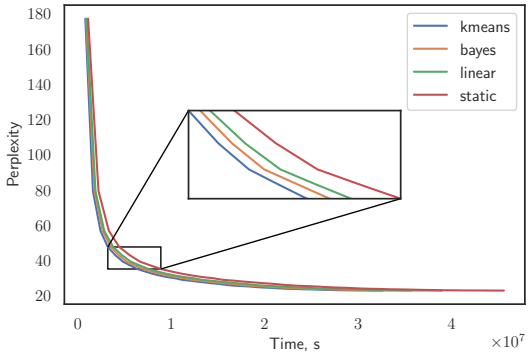


Figure 7. Transformer-XL training with adaptive schemes.

with QNCCL at 4bit compression by reducing bucket size to 128 for all models, but this comes with a further performance reduction.

Bagua and HiPress Comparison. Bagua [18] and HiPress-CaSync [6] are distributed training frameworks, which also support some generic forms of gradient quantization. In multinode experiments on 4x EC2 p3.8xlarge instances with 4 V100 GPUs each, we observed that Bagua and HiPress have similar performance to CGX on the smaller ResNet50 model, and that they are up to 10% slower on the larger VGG19 model. This is since all frameworks use the same NCCL backend for inter-node communication, but CGX uses a faster pattern (SRA vs Ring or Tree for NCCL) than HiPress and has better compression rate than Bagua (which only supports 8 bit quantization). Moreover, HiPress only supports 2 bit quantization, e.g. [49, 53]), which does not converge under standard parameters for Transformed-based models.

HiPress unfortunately does not support the newer commodity RTX-3090 GPUs, so we could only compare with Bagua on the Genesis Cloud 8xRTX3090 instance. The results of the comparison are presented in Figure 6, showing that CGX provides clearly superior performance, especially for the VGG19 model.

6.2.4 Comparison with Hardware Bandwidth Overprovisioning.

We now turn to Figure 5 where we first observe that, although in terms of single-GPU performance the RTX3090 is comparable to the V100/DGX-1, it has poor multi-GPU scaling for large models when using the standard NCCL setup (< 50% of linear scaling). The older 2080 GPUs have lower throughput both due to both lower memory, limiting maximum batch size, as well as lower computational power (Fig. 5a). Thus, we mainly focus on 3090 GPUs.

If we compare the maximum achievable performance (ideal scaling), CGX achieves similar results to the bandwidth overprovisioning approach, on both the DGX and the A6000 machines. In other words, CGX allows us to get bandwidth-overprovisioning performance via a "middleware" approach, achieving our stated goals. The remaining percentage gaps from perfectly linear scaling are because of 1) latency costs, 2) inefficiencies in our implementation, and 3) remaining communication costs, especially in early layers, which cannot be overlapped with computation. To measure this, we artificially removed the bandwidth bottleneck, by sending only a small number of elements per layer. The results in Table 7 show that CGX is close to ideal bandwidth reduction.

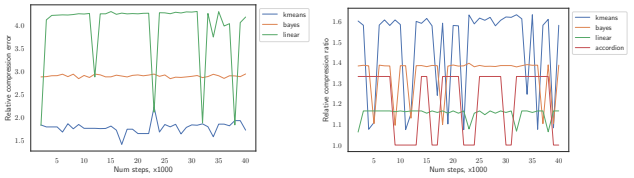
Table 7

Ideal performance (% of linear scaling) achievable via bandwidth-overprovisioning for different workloads, relative to CGX.

	ResNet50	VGG16	TXL	BERT	ViT
Ideal Perf.	92 %	91 %	95 %	88 %	95 %
CGX Perf.	90%	84 %	87 %	75 %	93 %

6.3 Layer-wise Adaptive Compression

So far, we have provided results for our version of 4bit quantization, which always recovers accuracy. We now examine additional performance savings due to adaptive compression. Across all models, the automated procedure in Section 4.1.1 identifies large layers with low-performance sensitivity (e.g. fully-connected or embedding



(a) Compression error. **(b)** Compression ratio.

Figure 8. Comparison of adaptive compression approaches. Error and size compression are shown relative to uniform static assignment of compression parameters to 4 bits.

Table 8

Comparison of adaptive methods. Speedups and compression rates are relative to static bits-width assignment (4 bits). Experiments are run with Transformer-XL base model on 8 RTX3090 GPUs (single node) and 4 nodes with 4xRTX3090 GPUs each (multi-node). Accordion is applied to QSGD with 3 and 4 as compression bounds.

	Compression	Speedup 1-Node	Speedup Multi-Node
KMEANS	**1.47**	**5%**	**40%**
Bayes	1.34	3%	30%
Linear	1.15	2%	13%
Accordion	1.21	3%	15%

layers) for lower bit-widths, and has similar total compression error to uniform compression. We illustrate this on Transformer-XL, the model with the most non-uniform layer sizes. We conducted single-node experiments on an 8xRTX3090 machine, and multi-node on four 4xRTX3090 machines. As before, the baseline is 4-bits static compression, which was shown to recover full accuracy. Figure 7 represents perplexity against time for different selection mechanisms. Figures 8a and 8b represent compression error and compression ratio relative to static assignment. Table 8 shows that Bayesian optimization shows stable compression error, and good *average* compression. Yet, the kmeans-based method shows the lowest quantization error, best average compression, and highest speedup, as it tends to compress large layers more. Specifically, this can lead to additional improvements in the order of 5% on a single node and up to 40% in multinode setting, without accuracy loss. This approach can still be improved by taking into account *runtime speedups* instead of absolute compression.

Among existing adaptive schemes, AdaComp [7] and Accordion [3] are the only ones which can be adapted to our setting. AdaComp suggests an adaptive scheme for sparsification, with possible further quantization of communicated elements. Accordion adapts gradient compression parameters based on identifying critical learning regimes.

For comparison, we execute the Transformer-XL model on a language modelling (LM) task. We first applied AdaComp only for sparsification: however, unfortunately the compression assignment provided by AdaComp did not converge to reasonable accuracy on this task.

Second, we adapted Accordion to our framework with QSGD compression, using Accordion to choose bit-width parameters based on its critical regimes detection approach. We used Accordion with hyperparameter $\eta = 0.5$, as suggested by the authors, and updated the compression parameter every 1k steps of training. As the lower and higher compression levels, we checked (2, 4) and (3, 4). The first pair resulted in significantly lower final accuracy relative to the baseline. The second pair (3,4) recovered the final accuracy, but the compression ratio was inferior to all the other adaptive schemes we investigated, and considerably below our proposed clustering scheme. Please see Figure 8b and Table 8 for an illustration. The table represents the speedups of different adaptive methods relative to the regular static compression. For instance, our adaptive scheme resulted in 17% additional multi-node speedup compared to Accordion.

6.4 Practical Implications

Multi-node experiments. Next, we examine performance on multi-node training in the cloud. We used 4 4xRTX3090 Genesis instances with 10GBps intra-node bandwidth and 5 GBps inter-node bandwidth. Table 9 shows that CGX provides up to 10x speedup over the uncompressed baseline.

Table 9

Items per second when training with the NCCL and CGX optimizations, respectively, on 4 machines with 4 RTX3090 GPUs each.

	ResNet50	ViT-base	Transformer-XL-base	BERT
Baseline	564	34	32k	1.4k
CGX	2.3k	235	85k	12k

Implications for Cloud Training. Several cloud services provide servers with commodity GPUs [20, 29, 30]. We therefore compare a standard AWS EC2 4xV100 GPU instance (p3.8xlarge) instance with a 4xRTX 3090 Genesis Cloud instance [20]. We execute the same training benchmark, with and without CGX. The numbers in Table 10 show that CGX allows us to obtain almost *twice* higher throughput (training tokens/second) per dollar on the more affordable Genesis instance, for a standard language modelling task (SQuAD) task using an industry-standard BERT model.

Table 10

Comparison of training performance for different cloud services (AWS and Genesis) with and without CGX. The training task is BERT-QA and achieves full accuracy.

Instance	Throughput (1K tok./sec)	Price per hour ($)	Tokens/second per $
Genesis + NCCL	4737	6.8	696
AWS + NCCL	**14407**	12.2	1181
Genesis + CGX	**14171**	6.8	**2083**

7 Conclusions

We proposed an algorithms & systems approach to remove the bandwidth bottlenecks from DNN training, supplanting the need for dedicated hardware support, and significantly improving performance in both single node (commodity) settings and, more generally, in multi-node cloud settings. Future work may extend our results to model-parallel or hybrid synchronization setups, e.g. [34, 58]; moreover, the idea of adaptive layer-wise compression should be extensible to other compression methods, for instance to choose ranks accurately for gradient decomposition methods, or layer-wise sparsities based on actual transmission speedups.

Acknowledgments

The authors sincerely thank Nikoli Dryden, Tal Ben-Nun, Torsten Hoefler and Bapi Chatterjee for useful discussions throughout the development of this project.

References

[1] 2021. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. Retrieved September 30, 2022 from https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf

[2] 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 the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA, November 2 - 4, 2016) *(OSDI'16)*. USENIX Association, USA, 265–283.

[3] Saurabh Agarwal, Hongyi Wang, Kangwook Lee, Shivaram Venkataraman, and Dimitris Papailiopoulos. 2021. Adaptive Gradient Communication via Critical Learning Regime Identification. In *Proceedings of Machine Learning and Systems* (Virtual event, USA, April 5 - 9, 2021), Vol. 3. 55–80.

[4] Saurabh Agarwal, Hongyi Wang, Shivaram Venkataraman, and Dimitris Papailiopoulos. [n.d.]. In *Proceedings of Machine Learning and Systems*.

[5] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in Neural Information Processing Systems* (Long Beach, CA, USA, December 4 - 7, 2017), Vol. 30. 1709–1720.

[6] Youhui Bai, Cheng Li, Quan Zhou, Jun Yi, Ping Gong, Feng Yan, Ruichuan Chen, and Yinlong Xu. 2021. Gradient Compression Supercharged High-Performance Data Parallel DNN Training. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany, October 26-29, 2021) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 359–375. https://doi.org/10.1145/3477132.3483553

[7] Chia Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. 2018. ADaComP: Adaptive residual gradient compression for data-parallel distributed training. In *32nd AAAI Conference on Artificial Intelligence, AAAI 2018* (New Orleans, USA, February 2–7, 2018). 2827–2835.

[8] Mengqiang Chen, Zijie Yan, Jiangtao Ren, and Weigang Wu. 2020. Standard Deviation Based Adaptive Gradient Compression For Distributed Deep Learning. In *Proceedings of 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)* (Melbourne, Australia, May 11-14 2020). 529–538.

[9] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO, USA, October 6-8, 2014), Vol. 14. 571–582.

[10] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. (2019). arXiv:arXiv:1901.02860

[11] 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* (Miami, FL, June 20 - 25, 2009). IEEE, 248–255.

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

[13] 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).

[14] Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments* (Salt Lake City, UT, USA, November 14 2016). IEEE Press, 1–8.

[15] 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* (New York, New York, February 7-12, 2020), Vol. 34. 3817–3824.

[16] Fartash Faghri, Iman Tabrizian, Ilia Markov, Dan Alistarh, Daniel M Roy, and Ali Ramezani-Kebrya. 2020. Adaptive gradient quantization for data-parallel sgd. *Advances in neural information processing systems* 33 (2020), 3174–3185.

[17] 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 35th ACM SIGCOMM 2021 Conference* (Virtual Event,USA, August 23 - 27, 2021) *(SIGCOMM '21)*. 676–691.

[18] Shaoduo Gan, Jiawei Jiang, Binhang Yuan, Ce Zhang, Xiangru Lian, Rui Wang, Jianbin Chang, Chengjun Liu, Hongmei Shi, Shengzhuo Zhang, Xianghong Li, Tengxu Sun, Sen Yang, and Ji Liu. 2021. Bagua: Scaling up Distributed Learning with System Relaxations. *Proc. VLDB Endow.* 15, 4 (dec 2021), 804–813. https://doi.org/10.14778/3503585.3503590

[19] Nitin A. Gawande, Joshua B. Landwehr, Jeff A. Daily, Nathan R. Tallent, Abhinav Vishnu, and Darren J. Kerbyson. 2017. Scaling Deep Learning Workloads: NVIDIA DGX-1/Pascal and Intel Knights Landing. (2017), 399–408. https://doi.org/10.1109/IPDPSW.2017.36

[20] Genesis. 2021. *Genesis GPU Cloud Offering*. Retrieved September 30, 2022 from https:/genesiscloud.com

[21] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. (2017). arXiv:arXiv:1706.02677

[22] Demjan Grubic, Leo K Tam, Dan Alistarh, and Ce Zhang. 2018. Synchronous multi-gpu deep learning with low-precision communication: An experimental study. In *Proceedings of the 21st International Conference on Extending Database Technology* (Vienna, Austria, March 26-29, 2018). OpenProceedings, 145–156.

[23] Jinrong Guo, Wantao Liu, Wang Wang, Jizhong Han, Ruixuan Li, Yijun Lu, and Songlin Hu. 2020. Accelerating Distributed

Deep Learning By Adaptive Gradient Quantization. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (Barcelona, Spain, May 4 - 8, 2020). 1603–1607. https://doi.org/10.1109/ICASSP40776.2020.9054164

[24] William Harmon. 2021. *Dual NVIDIA GeForce RTX 3090 NVLink Performance Review.* Retrieved September 30, 2022 from https://www.servethehome.com/dual-nvidia-geforce-rtx-3090-nvlink-performance-review-asus-zotac/

[25] Inc Huggingface. 2022. *Huggingface Transformers Repository.* Retrieved April 30, 2022 from https://huggingface.co/models

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

[27] 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)* (Virtual Event, November 4–6, 2020). USENIX Association, 463–479. https://www.usenix.org/conference/osdi20/presentation/jiang

[28] Sai Praneeth Karimireddy, Quentin Rebjock, Sebastian Stich, and Martin Jaggi. 2019. Error feedback fixes signsgd and other gradient compression schemes. In *International Conference on Machine Learning* (Long Beach, CA, USA, Jun 10 – 15, 2019). PMLR, 3252–3261.

[29] LambdaLabs. 2021. *LambdaLabs GPU Cloud Offering.* Retrieved September 30, 2022 from https://lambdalabs.com/cloud

[30] LeaderGPU. 2021. *LeaderGPU Cloud Offering.* Retrieved September 30, 2022 from https://www.leadergpu.com/

[31] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 94–110. https://doi.org/10.1109/TPDS.2019.2928289

[32] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. 2018. Tartan: Evaluating Modern GPU Interconnect via a Multi-GPU Benchmark Suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)* (Raleigh, NC, USA, 30 September - 02 October 2018). 191–202. https://doi.org/10.1109/IISWC.2018.8573483

[33] 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 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)* (Broomfield, CO, USA, October 6-8, 2014). 583–598.

[34] Shijian Li, Oren Mangoubi, Lijie Xu, and Tian Guo. 2021. Sync-Switch: Hybrid Parameter Synchronization for Distributed Deep Learning. (2021). arXiv:arXiv:2104.08364

[35] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2018. 3lc: Lightweight and effective traffic compression for distributed machine learning. (2018). arXiv:arXiv:1802.07389

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

[37] Ahmed M Abdelmoniem, Ahmed Elzanaty, Mohamed-Slim Alouini, and Marco Canini. 2021. An efficient statistical-based gradient compression technique for distributed training systems. In *Proceedings of Machine Learning and Systems* (Virtual event, USA, April 5 - 9, 2021), Vol. 3. 297–322.

[38] J. B. MacQueen. 1967. Some Methods for Classification and Analysis of MultiVariate Observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability* (Berkeley, CA, USA, June 21-July 18 1965), Vol. 1. University of California Press, 281–297.

[39] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. 2020. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro* 40, 2 (2020), 8–16.

[40] Nvidia. 2020. *NVIDIA Deep Learning Examples for Tensor Cores.* Retrieved April 30, 2022 from https://github.com/NVIDIA/DeepLearningExamples

[41] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. 2018. Image transformer. In *International Conference on Machine Learning* (Stockholm, Sweden, July 10-15, 2018). PMLR, 4055–4064.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[43] 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* (Ontario, Canada, October 27 - 30, 2019). 16–29.

[44] Ali Ramezani-Kebrya, Fartash Faghri, Ilya Markov, Vitalii Aksenov, Dan Alistarh, and Daniel M Roy. 2021. NUQSGD: Provably Communication-efficient Data-parallel SGD via Nonuniform Quantization. *Journal of Machine Learning Research* 22, 114 (2021), 1–43.

[45] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA, November 17–22, 2019).

[46] 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* (Singapore, September 14-18, 2014).

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

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

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

(Dresden, Germany, September 6-10, 2015).

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[51] Thijs Vogels, Sai Praneeth Karinireddy, and Martin Jaggi. 2019. PowerSGD: Practical low-rank gradient compression for distributed optimization. *Advances In Neural Information Processing Systems 32 (Nips 2019)* 32 (2019).

[52] Hongyi Wang, Scott Sievert, Zachary Charles, Shengchao Liu, Stephen Wright, and Dimitris Papailiopoulos. 2018. ATOMO: Communication-efficient learning via atomic sparsification. (2018). arXiv:arXiv:1806.04090

[53] 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. (2017). arXiv:arXiv preprint arXiv:1705.07878

[54] Ross Wightman. 2019. PyTorch Image Models. https://github.com/rwightman/pytorch-image-models. https://doi.org/10.5281/zenodo.4414861

[55] 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* (Virtual event, July 7- 10, 2020).

[56] Xiaodong Yi, Ziyue Luo, Chen Meng, Mengdi Wang, Guoping Long, Chuan Wu, Jun Yang, and Wei Lin. 2020. Fast Training of Deep Learning Models over Multiple GPUs. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands, December 7 - 11, 2020). 105–118. https://doi.org/10.1145/3423211.3425675

[57] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2019. Large batch optimization for deep learning: Training bert in 76 minutes. (2019). arXiv:arXiv:1904.00962

[58] Qihua Zhou, Song Guo, Zhihao Qu, Peng Li, Li Li, Minyi Guo, and Kun Wang. 2020. Petrel: Heterogeneity-aware distributed deep learning via hybrid synchronization. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1030–1043.