Near-Lossless Gradient Compression for Data-Parallel Distributed DNN Training

Xue Li
Alibaba Group
youli.lx@alibaba-inc.com

Cheng Guo
Tsinghua University
aeromarisa@gmail.com

Kun Qian
Alibaba Group
kunqian.qk@alibaba-inc.com

Menghao Zhang
Unaffiliated
zhangmenghao0503@gmail.com

Mengyu Yang
Unaffiliated
yangmengyu0107@gmail.com

Mingwei Xu
Tsinghua University
xumw@tsinghua.edu.cn

ABSTRACT

Data parallelism has become a cornerstone in scaling up the training of deep neural networks (DNNs). However, the communication overhead associated with synchronizing gradients across multiple nodes has emerged as a significant bottleneck, adversely affecting training efficiency and leading to a surge in large-scale distributed model training costs. By leveraging insights into the statistical characteristics of gradients, we present GComp, a near-lossless gradient compression scheme designed to reduce the communication burden during data-parallel training significantly. GComp develops an optimized Huffman encoding/decoding strategy to compress gradient exponents effectively. Additionally, it introduces an innovative multi-level quantization method for mantissa, complemented by a pruning strategy that eliminates zero-valued gradients. These integrated approaches significantly reduce the volume of data for synchronization, while virtually not affecting the DNN model's training accuracy. We conduct comprehensive evaluations of GComp, demonstrating that our method can decrease the communication volume by as much as 67.1%, and enhance training speed by up to 1.9×.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

KEYWORDS

Gradient Compression, Distributed DNN Training

ACM Reference Format:
Xue Li, Cheng Guo, Kun Qian, Menghao Zhang, Mengyu Yang, and Mingwei Xu. 2024. Near-Lossless Gradient Compression for Data-Parallel Distributed DNN Training. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3698038.3698541

1 INTRODUCTION

The great success of generative AI models boosts the significant demand for distributed deep neural network (DNN) training, which has emerged as a crucial service for major cloud providers such as Amazon Web Services [5], Microsoft Azure [43], Google Cloud Platform [19], and Alibaba Cloud [9]. Efficiently training DNN models requires the collaboration of dozens to thousands of GPUs. To make this large-scale DNN training efficient on large datasets, data parallelism (DP) is a fundamental approach. DP entails substantial gradient synchronization at the end of each iteration, introducing a large amount of communication overhead.

As shown in Figure 1, as the number of DP groups expands, the training speed for extremely large-scale models like Llama [61] (trained utilizing DeepSpeed [53] on Alibaba Cloud PAI [10]) and GPT-3 [50] (trained utilizing Megatron [57] on PAI) scales sub-linearly, which can result in a performance degradation of up to 70%. Furthermore, this sub-linear scalability in DP is also ubiquitous in the training of typical DNN models (e.g., ResNet [21] and Bert [12]). Our in-depth analysis confirms that this degradation is mainly attributed to communication among GPUs. In cloud environments, especially when compared to dedicated physical clusters, the multi-tenant nature of resource sharing can lead to performance degradation due to competition for network resources. In this context, optimizing communication becomes even more critical, thus emerges as one of the main challenges faced by cloud service providers in delivering efficient DNN training services.

To mitigate the side-effect introduced by gradient synchronization, the most straightforward optimizing approach is to reduce the size of the transmitted message, through

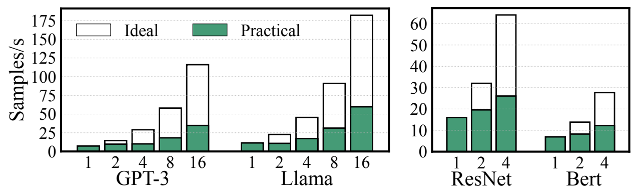


Figure 1: Training speedup of GPT-3, Llama, ResNet and Bert with increasing DP groups (more GPUs used), tested on Alibaba Cloud PAI.

the compression of gradients. Existing gradient compression efforts depend on either sparsification [4, 7, 14, 55, 60], which omits a certain portion of the gradients, or quantization [3, 37, 41, 59, 66], which reduces the size of each gradient value. Despite being effective at reducing message sizes, these efforts are lossy compression. They compromise the integrity of gradient information, introduce non-negligible errors in parameter updates, and affect the overall convergence of models. By surveying our engineers in charge of training DNN models for crucial online recommendation services, we found that directly employing state-of-the-art lossy compression algorithms regresses the model's accuracy (loss rate) to the state it was several months prior, causing a significant waste of training resources. On the other side of the spectrum, lossless compression of floating-point numbers is a classical research and engineering topic, where many typical solutions (e.g., ZSTD [15] and Zlib [1]) are proposed. These methods guarantee absolute precision for arbitrary inputs. However, experiments show that they yield only a limited compression ratio and require substantial computation resources in gradient compression.

We emphasize that, by leveraging the statistical characteristics of gradients throughout DNN training, a delicate compression mechanism can be constructed to achieve a sweet point: efficiently compressing gradient message sizes while preserving near-lossless parameter updates. Specifically, we identify three key observations (**OB**) from DNN model training.

OB-1: During training, the gradients tend to cluster around zero, with a notable concentration in their exponent part. Remarkably, the 15 most frequent exponent values represent almost 95% of all occurrences.

OB-2: There exists a substantial disparity between the magnitude of gradients and their corresponding parameters. Generally, gradients are 3 to 5 orders of magnitude less than the corresponding parameters. Moreover, this gap becomes even larger as training progresses.

OB-3: A significant fraction of the gradients is zero, indicating that up to 50% of the gradient values do not contribute to the parameter update process.

Based on the above observations, we propose a near-lossless gradient compression scheme, GComp, that comprises both exponent compression and mantissa compression. For exponent compression, inspired by **OB-1**, we develop an enhanced Huffman encoding strategy tailored to efficiently compress the most common exponents and handle rare exponents with full transmission. We also optimize the decoding procedure by utilizing a lookup table.

In mantissa compression, guided by **OB-2**, we recognize that the lower segments of mantissas in gradients have a minimal impact on the significant digits of corresponding parameters and, therefore, can be truncated during transmission to save bandwidth. Through a meticulous analysis of various optimizers used in DNN training, we establish a quantization metric for mantissas that ensures the compression-induced deviation remains within an acceptable range. We further implement the multi-level quantization to reduce the complexity introduced by customized quantization for each mantissa. Leveraging **OB-3**, our approach eliminates the transmission of non-contributory gradients, thereby further reducing the communication overhead.

We implement GComp and evaluate its effectiveness using widely-used representative DNN models (ResNet [21] and Bert [12]), trained on PAI [10], a machine learning platform provided by Alibaba Cloud. The results indicate that GComp can cut the communication volume by as much as 67.1%, and improve end-to-end training efficiency by up to 1.9×. Moreover, it maintains the convergence trajectory of the model's loss function closely aligned with that observed in scenarios without compression. In comparison with existing lossy compression techniques, GComp decreases the average deviation in the model loss function's value by 73.6% ∼ 93.1% compared to those observed with a conventional quantization-based method, and by 77.4% ∼ 96.8% compared to those associated with a sparsification-based method.

This paper makes the following contributions:

- We identify and outline three key observations regarding gradients and parameters during data-parallel distributed DNN training, which are crucial for the optimization of gradient transmission (Section 3).
- We introduce GComp, an innovative gradient compression approach that uses a lossless method for efficiently compressing the exponents of gradients (Section 4) and a near-lossless method for compressing their mantissas (Section 5).
- We efficiently implement GComp and integrate it within the AllReduce operation of Gloo [16], thereby facilitating its application in distributed DNN training frameworks (Section 6).
- Through comprehensive evaluations, we demonstrate that GComp can offer a speed increase of up to 1.9× for end-to-end DNN model training while maintaining near-lossless outcome in terms of model convergence (Section 7).

2 BACKGROUND

In this section, we provide an overview of the background of gradient compression in DNN training, and discuss the limitations of existing compression techniques.

2.1 Data Parallelism in DNN Training

DNNs have become the cornerstone of modern machine learning applications, with their ability to model complex relationships in massive data. Nowadays, extremely large-scale DNNs such as GPT-3 [50], which contain billions of parameters, are being trained on large datasets to achieve state-of-the-art performance. The scale of data and computational intensity required for training large DNNs necessitates the use of distributed training frameworks. PyTorch [51], as well as frameworks built on top of it such as Megatron [57] and DeepSpeed [53], provide the necessary infrastructure to facilitate this process effectively.

Distributed Data Parallelism [25, 32, 38, 53] is a foundational and prevalent approach for distributed training, facilitating the division of workload across multiple workers. This method has each DP group—whether it is an individual worker or a group of workers coordinated via tensor parallelism [45, 57] or pipeline parallelism [17, 23, 33, 44, 46, 53]—holding a complete replica of the model to perform computations on its assigned data. The central aim of this paradigm is to harness the collective power of disparate training data processed by various workers to accelerate the model parameters' updates.

2.2 Gradient Aggregation via AllReduce

AllReduce is a crucial communication operation in distributed training frameworks that employ data parallelism. It is facilitated by collective communication libraries such as NCCL [48], Gloo [16], and MPI [49]. This operation works by performing an element-wise reduction operation (e.g., sum, prod, max, min) across all working nodes, followed by a broadcast of the resultant value to all workers.

For example, consider a training iteration using the Megatron framework, where a *global-batch* of samples is processed. This global batch is subdivided into several *micro-batches* that are evenly distributed among various DP groups. Throughout an iteration, each DP group is responsible for its respective set of accumulated gradients, conducting both forward and backward passes on the micro-batches allocated to it. Once all backward passes are complete, the AllReduce operation is employed to synchronize the gradients across DP groups, ensuring consistent updates to the model parameters. However, as the scale of the model increases, the communication overhead associated with AllReduce grows significantly, leading to a bottleneck in the training process. For instance, when training the GPT-3 22B model on 16 nodes with the Megatron framework[1], where each node corresponds to a single DP group and comprising 8 GPUs cooperating through tensor parallelism, it has been observed that the communication overhead from AllReduce can represent up to 30.7% of total training time.

2.3 Floating-point Numerical Formats

Within the context of this paper, we focus on the IEEE 754 standard [26] for floating-point numbers, which is the predominant format for encoding real numbers in computer systems, and is employed by main DNN training frameworks. This norm encompasses various precision such as FP16, FP32, FP64, and FP128, each tailored to convey floating-point numerals with distinct levels of accuracy and magnitudes. Specifically, FP32 and FP16 formats are extensively utilized during the training of machine learning models, to represent the parameters and gradients. Represented as a combination of three components, these floating-point numbers include:

- Sign bit (*sign*): A single bit to indicate the polarity (positive or negative) of the floating-point value.
- Exponent (*exp*): A fixed number of bits allocated to express the numerical range of the number.
- Mantissa (*mant*): A fixed number of bits that represent the precision of the value.

For example, a 32-bit FP32 floating-point number, N_{32}, can be represented as:

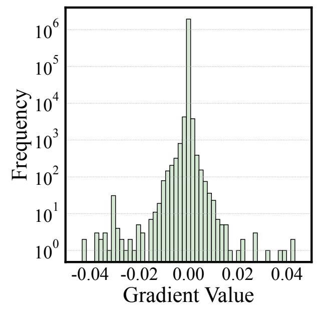
$$N_{32} = (-1)^{sign} \times 2^{(exp-127)} \times (1 + \frac{mant}{2^{23}}), \tag{1}$$

where the sign is represented by 1 bit, the exponent by 8 bits, and the mantissa by 23 bits. Different formats of floating-point numbers adhere to this structure, with variations solely in the number of bits used for the exponent and mantissa.

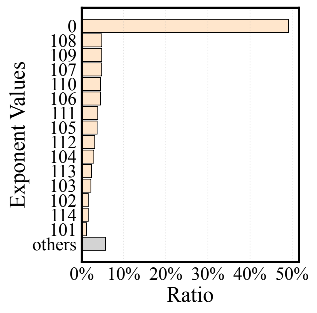
2.4 Limitations of Existing Solutions

Limitations of Lossy Gradient Compression Solutions. Compressing the volume of the gradients synchronized during the training process is an effective strategy to alleviate the communication overhead, and consequently, enhance the training efficiency. Current gradient compression techniques can be broadly categorized into two classes: quantization-based and sparsification-based methods. Quantization-based methods [4, 7, 14, 55, 60], reduce the precision of the gradients by quantizing them to a lower bit-width, e.g., from FP32 to FP16, thereby decreasing the volume of data that needs to be communicated. Sparsification-based methods [3, 37, 41, 59, 66], on the other hand, reduce the communication overhead by selectively transmitting a subset of gradients and

[1]Each of the 16 nodes represents a separate instance equipped with eight NVIDIA A100 GPUs.

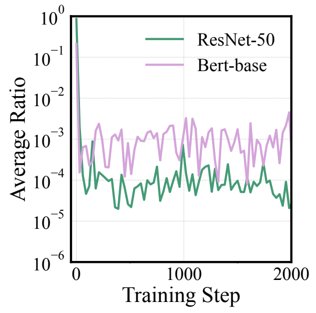


(a) Gradient distribution.

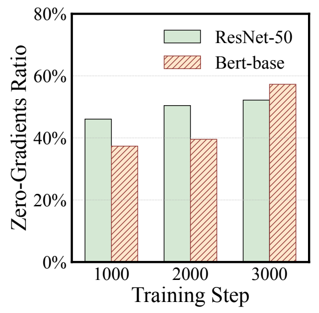


(b) Exponent distribution.

Figure 2: Distribution of gradient values and gradient exponents during the training of a ResNet-50 model using FP32.



(a) The ratio of gradient to its corresponding parameter.



(b) Proportion of zero-valued gradients during training.

Figure 3: The change in the average ratio of gradients to parameters during the training process of ResNet-50 and Bert-base (a). The proportion of zero-valued gradients in the transmitted gradients during the training process of ResNet-50 and Bert-base (b).

discarding the rest. This can be accomplished through various strategies, such as transmitting only the gradients with the largest magnitudes (*Top-k*) or choosing a random subset of gradients (*Random-k*). Another approach involves sparsely conducting the AllReduce operation to decrease the volume of data communicated across the network. However, these two kinds of methods have a common limitation: they are lossy compression and lead to a degradation in the model's convergence rate and final accuracy, which is undesirable in practice for training large-scale DNNs.

Limitations of Lossless Compression Solutions. On the other hand, a large amount of lossless compression solutions are proposed to compress data without sacrificing accuracy [1, 2, 15, 18, 31, 34, 36, 40, 42, 52]. However, experiments show that employing existing lossless compression methods can decrease transmitted message size but introduce unacceptable compress and decompress overhead (details in Section 7.2). The root cause is that these existing solutions target compressing data with the general statistical data pattern. However, gradients generated during data-parallel training have specific data distribution features. Without considering these features, it is hard to achieve accurate and efficient gradient compression simultaneously.

3 INSIGHTS AND GCOMP OVERVIEW

In this section, we present the motivation behind our work, which is driven by the characteristics of gradients during DNN training, and the need for efficient compression of these gradients. We then leverage these insights to propose the design of a near-lossless compression scheme for gradients.

3.1 Gradient Characteristics in Training

Takeaway-1: *The distribution of gradient values during training is observed to be concentrating near zero.*

Gradients represent the direction of updates for parameters in each iteration of DNN training. As training proceeds, the model gradually converges, meaning its parameters become stabilized. Consequently, the magnitudes of the gradients tend to decrease towards zero. In the context of floating-point representation, this phenomenon indicates that the exponent parts of gradients gradually converge to several specific values.

We have performed a statistical analysis examining both the distribution of gradient magnitudes and the distribution of their exponent parts during the training of a ResNet-50 [21] model using 32-bit floating-point precision as an example. The findings are illustrated in Figures 2. Our analysis reveals that the distribution of gradient exponents is relatively narrow. Out of 256 possible values for an 8-bit exponent, the 15 most frequent values encompass nearly 95% of the total occurrences. Notice that this characteristic is ubiquitous in all DNN trainings.

Takeaway-2: *As the training progresses, the relative magnitude of gradients diminishes significantly in comparison to the corresponding parameters.*

Gradients typically are 3 to 5 orders of magnitude smaller than the parameters they are dedicated to updating. While the gradients in a DNN model progressively migrate towards smaller values as training progresses, the parameters maintain a relatively consistent magnitude. Therefore, it results in a steadily diminishing ratio of gradient compared with the corresponding parameter.

We also have statistically analyzed the ratio of gradients to the corresponding model parameters' magnitudes during the training process of ResNet-50 and Bert-base. As shown in Figure 3a, when the training progresses, the gradients become far smaller than the corresponding parameters. During the training, the optimizers could refine parameter values by integrating gradients into these parameters. This integration involves aligning the operands according to their exponent

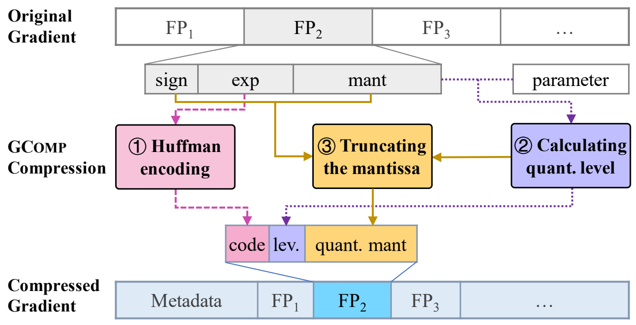


Figure 4: Overall overview of the GComp scheme for gradient compression.

values, followed by the addition of their mantissa values. Particularly, when the magnitude of a gradient is significantly smaller than that of its corresponding parameter, lower bits in the gradient's mantissa hardly affect the parameter.

Takeaway-3: *Zero-valued gradients are widespread throughout the training process, and they do not contribute to the change of model parameters.*

As the training advances, it is common for a certain proportion of gradients to turn to zero. To illustrate this effect, we conducted a study where we trained two different models, ResNet-50 and Bert-base [12], and recorded the percentage of zero-valued gradients transmitted by a single worker. The findings, presented in Figure 3b, reveal a substantial presence of zero gradients in the data transmitted across the communication at various training steps. Since these zero gradients do not contribute to the updating of parameters, they can be effectively pruned.

3.2 GComp Overview

Leveraging the insights from the characteristics of gradients during training, we propose GComp, a near-lossless gradient compression scheme aiming to reduce the communication volume while maintaining the training accuracy of DNN models. The design of GComp is outlined in Figure 4.

Takeaway-1 highlights a concentrated frequency distribution, signifying that the exponent set carries low information entropy. If a uniform encoding length, such as the 8-bit representation in FP32, is used for all exponent values, significant redundancy exists in the bit representation. Thus, we propose the enhanced Huffman coding, an entropy-based encoding scheme which leverages the uneven frequency distribution to compress the exponent values. We further optimize both the encoding and decoding processes to enhance its compression efficacy and processing speed, as detailed in Section 4.

Takeaway-2 reveals that truncating lower bits in a gradient's mantissa does not affect the final result when the gradient's magnitude is considerably smaller than that of the parameter. This observation creates a chance to quantize mantissas without compromising the parameter update process. Thus, we propose a multi-level quantization scheme to compress the gradient mantissa based on an analysis of gradients' impact on parameters within a wide range of optimizers used in DNN models.

Furthermore, *Takeaway-3* reveals the prevalence of zero-valued gradients, which do not contribute to the model parameters' updates. Therefore, directly excluding these gradients can further compress the transmitted message, to diminish the communication volume. The details of quantization and pruning techniques are provided in Section 5.

4 EXPONENT COMPRESSION

Inspired by the observation that exponent values exhibit a non-uniform distribution, we utilize an optimized Huffman encoding and decoding algorithm, tailored to minimize the number of bits necessary to represent exponent values of gradients, without any loss of information.

4.1 Design Considerations

Considering that most exponents are within a narrow range of values, the initial idea is to construct a hash table mapping 8-bit exponents to more concise (e.g., 4-bit) encodes for compression. However, two main issues arise when turning this straightforward idea into a tangible solution, leading us toward the use of an optimized Huffman encoding method.

Issue 1: The hash table can be further optimized for skewed distributions. Even when we concentrate on those frequently occurring exponents, such as the top 15 values in Figure 2b, the distribution remains markedly uneven. For instance, around 50% of the exponents are zero, and the second most common value, 108, occurs with a frequency that is 4.2 times greater than that of the fifteenth most common value, 101. Considering this skewed distribution, a simple hash table that assigns fixed-size codes to each value would fall short of achieving an optimal compression ratio. When values that appear more frequently are represented by shorter codes, the overall performance improves. Huffman encoding [24] is a widely used technique that embodies this principle by allocating variable-length codes to input symbols based on their frequencies. This approach, which assigns shorter codes to more frequent symbols, aligns with the frequency distribution of exponents, thereby improving compression efficiency.

Issue 2: Standard Huffman encoding still has limitations in our specific context. While Huffman encoding is adept at handling diverse inputs, its direct application to exponent compression presents certain limitations. The primary concern is the generation of excessively long codes for infrequently occurring exponents, which compromises compression efficiency. Consequently, it has become essential to refine the Huffman encoding approach specifically for exponent compression.

Table 1: An example of the native and optimized Huffman encoding, where the 10101111 in bold stands for the special code and the subsequent 8-bit sequence is the raw representation of the exponents.

Source	Native Huffman	Optimized Huffman
0	011	011
1	0011	0011
...	...	...
32	101011110100101010	**10101111**00100000
...	...	...
138	110111011100110110	**10101111**10001010
...	...	...
255	1001	1001

4.2 Huffman Encoding

Native Huffman Encoding. The process of Huffman encoding involves constructing a binary tree, where each leaf node represents an input symbol, and the path from the root to the leaf determines the symbol's code. This binary tree is developed iteratively by merging the two least frequent symbols into a single composite node until only a single root node remains, thereby representing all symbols. The specific code for each symbol is obtained by tracing the path from the root to its corresponding leaf node. On the other hand, the decoding process works by navigating the tree from the root to the appropriate leaf node as directed by the incoming code sequence. Given the 8-bit exponent representation in the FP32 format, which permits exponent values to span from 0 to 255, the initial two columns of Table 1 demonstrate the raw exponent values alongside their corresponding Huffman-encoded representations, with the encoded values shown in binary format.

Limitations. Three main limitations exist in the native Huffman encoding:

(1) In certain corner cases, the length of the generated Huffman codes can extend to as long as $(2^8 - 1)$ bits in the worst-case scenario of a completely skewed tree, markedly surpassing the original 8-bit representation. This discrepancy inflates the average code length in compression. To address this issue, GComp designs a truncated encoding and full transmission method, as detailed in Section 4.3.

(2) The optimal Huffman encoding for exponents may change as the training progresses; thus, GComp dynamically updates the encode table (Section 4.4), ensuring that compression efficiency is maintained.

(3) The Huffman decoding process can be inefficient; therefore, GComp streamlines it by utilizing a lookup-table-based method (Section 4.5).

4.3 Full Transmission of Long Codes

During the encoding process, if the length of the code generated exceeds a predefined threshold, it will be replaced by a special code to illustrate that the original exponent value should be transmitted in full. The updated codebook, as illustrated in the third column of Table 1, with the inclusion of the special code (**10101111** in this example) and the original value, can result in a shorter overall code length, in the case where the actual distribution of exponent values deviates from the initial assumptions.

Suppose the bit length of the source value is n_s, and the threshold for the code length is n_θ, all codes with lengths exceeding n_θ will be replaced by the special code. The special code is generated by truncating the first encountered overly long code to n_θ bits, which is proven to be a legitimate Huffman code because of its prefix-free property. Since the full transmission of the original value is required, the actual code length transmitted will be:

$$n_f = n_\theta + n_s. \tag{2}$$

Analysis. Let c_i denote the i-th value to be encoded, and n_i represent its corresponding native Huffman encoding length, we divide the set of source values $S = \{c_i\}$ into three subsets:

$$\begin{cases} S_1 = \{c_i \mid 1 \le n_i \le n_\theta\}, \\ S_2 = \{c_i \mid n_\theta < n_i \le n_f\}, \\ S_3 = \{c_i \mid n_f < n_i \le 2^{n_s} - 1\}. \end{cases} \tag{3}$$

Among these subsets, S_1 represents the range where the exponential distribution's probability is concentrated, resulting in shorter Huffman codes. In contrast, subsets S_2 and S_3 correspond to the less probable values, yielding longer native Huffman codes that necessitate full transmission. Employing native Huffman encoding without adjustments may lead to a misalignment between the actual and the assumed distributions for code table construction, potentially causing an increase in the occurrence probability and, consequently, a surge in the actual encoding length for elements within S_3.

Let p_1, p_2, and p_3 represent the occurrence probabilities of values in the three subsets, n_1, n_2, and n_3 denote the probability-weighted average encoding lengths of the native Huffman codes for the subsets, where $n_i = -logp_i$ for $i \in \{1, 2, 3\}$. The average encoding length is then given by:

$$n_{\text{avg}} = -p_1 logp_1 - p_2 logp_2 - p_3 logp_3. \tag{4}$$

If there is a deviation from the source distribution used to establish the encoding scheme, resulting in an increase in the probability p_3 of subset S_3 by Δp, and a corresponding decrease in p_1 by Δp, the average encoding length using native Huffman encoding can be expressed as the cross entropy:

$$n_{\text{nat}} = -(p_1 - \Delta p) logp_1 - p_2 logp_2 - (p_3 + \Delta p) logp_3. \tag{5}$$

The average length when utilizing the modified Huffman encoding with full transmission is as follows:

$$n_{\text{opt}} = -(p_1 - \Delta p) log p_1 + (p_2 + p_3 + \Delta p) n_{\text{f}}. \tag{6}$$

Supposing $p_2 = r p_3$, with r being a constant, the difference between the two average encoding lengths becomes:

$$\begin{aligned} n_{\text{opt}} - n_{\text{nat}} &= (p_2 + p_3 + \Delta p) n_f + p_2 \log p_2 + (p_3 + \Delta p) \log p_3 \\ &= [(r + 1 + \frac{\Delta p}{p_3})(n_f - n_3) + r \log r] p_3. \end{aligned} \tag{7}$$

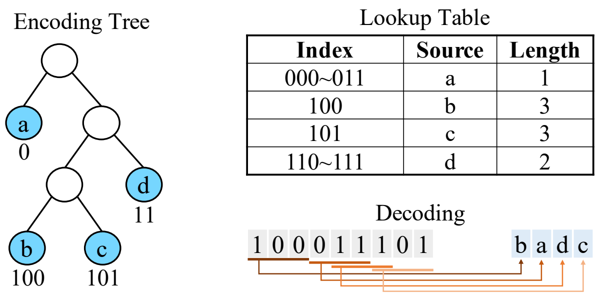
Given that $n_f < n_3$, an increase in Δp will ultimately render $n_{\text{opt}} - n_{\text{nat}}$ negative. This indicates that our full transmission strategy results in a shorter average encoding length compared to native Huffman encoding under such conditions. Full transmission effectively provides a superior upper bound for the data length during actual compression. In an ideal scenario, where the actual distribution of source characters closely aligns with the code table ($\Delta p = 0$), employing full transmission slightly raises the average encoding length. However, due to the very low probability of encountering overly long codes in this ideal case, the impact is negligible.

4.4 Dynamic Adjustment of the Huffman Encoding Table

Within GComp, the Huffman encoding table is constructed by leveraging prior knowledge about the distribution of exponent values before training commences. In addition, to accommodate the changing distribution, this table is periodically updated, e.g., after every 50 iterations. This regular adjustment is essential for accurately representing the evolving distribution of exponent values in the transmitted gradients. The update of the Huffman encoding table takes extra network bandwidth. In order to eliminate this side-effect, we update this table with an **asynchronous interaction**: GComp calculates and updates the Huffman encoding table only during the forward and backward phases in the iteration, when the network resource is idle. Experimental results in Section 7.3 demonstrate that this dynamic adjustment strategy significantly enhances the compression efficiency of exponent values.

4.5 Decoding based on Lookup Table

The Huffman decoding process typically involves sequentially reading bits from the encoded segment and traversing the encoding tree from the root to the leaves, which greatly affects the decoding efficiency. Fortunately, our optimized encoding algorithm imposes an upper limit on the code length, presenting GComp an ideal opportunity to construct a lookup table for decoding. This approach can improve the decoding efficiency significantly, particularly in the case of an unbalanced encoding tree.



Index	Source	Length
000~011	a	1
100	b	3
101	c	3
110~111	d	2

Figure 5: Encoding tree and lookup table with $n_\theta = 3$.

Assuming the maximum code length in our scheme is n_θ, a lookup table of size 2^{n_θ} is required. The indexes in this table represent the codes, while the values contain two elements: the corresponding source symbols and the actual length of the codes. For each code, it is first extended to n_θ bits, utilizing all possible codes within this range as indexes, and assigning the corresponding source along with the code's original length prior to padding. For instance, if $n_\theta = 3$ and the source symbol a is represented by the code 0, as depicted in Figure 5, then the index range from 000 to 011 would all point to the source symbol a with a length of 1. Figure 5 illustrates the method of creating a lookup table with $n_\theta = 3$.

Employing the lookup table allows for constant-time decoding of a single source symbol. Initially, a binary string of length n_θ is extracted from the current position in the encoded segment and utilized as an index to identify the corresponding source symbol in the lookup table. Subsequently, the current position pointer advances by the actual length of the code, marking the starting point for decoding the next code. This lookup-table-driven decoding process, requiring merely one lookup per code, is significantly more efficient than the traditional encoding tree traversal technique.

5 MANTISSA COMPRESSION

This section presents the mantissa compression technique employed in GComp, aimed at decreasing the number of bits needed to represent the mantissa portion of gradients without compromising the accuracy of model parameter updates. Our approach adopts a multi-level quantization strategy, anchored in an analysis of the impact of gradient precision on the updating process of various optimizers (Section 5.1). It also incorporates a zero-value gradient pruning mechanism to further minimize the size of data transmissions (Section 5.2).

5.1 Multi-Level Quantization

DNN training is inherently an iterative process. It involves the repetitive adjustment of model parameters guided by gradients of the loss function with respect to those parameters. These gradients are obtained through backpropagation

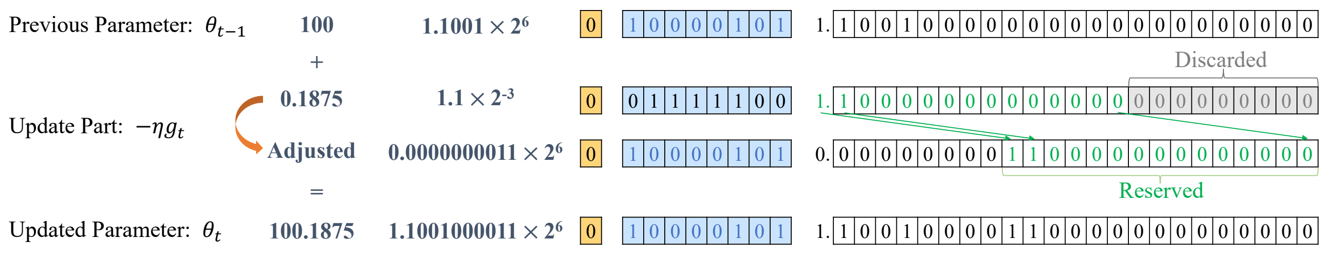

Previous Parameter: θ_{t-1}
100
1.1001×2^6
+
Discarded
Update Part: $-\eta g_t$
0.1875
1.1×2^{-3}
Adjusted
0.0000000011×2^6
Reserved
=
Updated Parameter: θ_t
100.1875
1.1001000011×2^6

Figure 6: The addition step ($\theta_t = \theta_{t-1} - \eta g_t$) for parameter updating in the SGD optimizer, utilizing FP32 format.

of the loss across the network, subsequently utilized by the optimizer to update the model parameters.

Case Study: SGD. Taking the widely-used Stochastic Gradient Descent (SGD) optimizer as an example, the parameter update method can be encapsulated by the formula [51]:

$$g_t = g_t + \lambda\theta_{t-1},$$
$$\theta_t = \theta_{t-1} - \eta g_t, \tag{8}$$

where g_t represents the gradient at the current step t, θ_t signifies the model parameters at step t, and θ_{t-1} is the parameters' values at the preceding step. The adjustment of parameters follows a specific updating rule involving η, the learning rate, and λ, a regularization parameter. The value of the learning rate η is typically below 1. With the magnitude of the gradient (g) significantly lesser than the parameter (θ), as analyzed in Section 3, the outcome of ηg_t is also substantially smaller in magnitude compared to the parameters. In operations involving floating-point arithmetic, especially during the addition or subtraction of two numbers where one is considerably smaller in absolute terms, the less significant bits of the mantissa in the lesser value are discarded. Figure 6 illustrates this phenomenon (utilizing the FP32 format), where the addition of two floating-point numbers results in the truncation of the less significant bits of the mantissa in the smaller number.

Specifically for the SGD optimizer, we define a **quantization metric** $\delta = \frac{\theta_{t-1}}{\eta g_t} - \frac{\lambda\theta_{t-1}}{g_t}$, which mirrors the gradient and parameters' relative size in the optimizer's context. This serves as a determination of the number of ineffective bits in the gradient during parameter updates. If $\delta > 2^n$, it indicates that in the process of updating parameters, at least n bits in the gradient's mantissa would naturally be discarded. Hence, pre-quantizing and truncating n bits from the gradient's mantissa have negligible impact on the accuracy of parameter updates[2]. GComp leverages this insight to introduce a parameter-aware quantization compression algorithm specifically tailored for efficiently compressing the mantissa components of gradients.

[2]When no carry-over occurs, there is no impact whatsoever. When carry-over occurs, the impact is limited to affecting only the last bit of the mantissa. Thus, in the context of FP32, this error is constrained to be within $2^{-22}\theta$.

Quantization Metric. The calculation of δ varies among different optimizers due to their unique computational methods for updating parameters with gradients. Building on the foundation set by SGD, we have examined the parameter update mechanisms of the widely-used optimizers in DNN training, refer to the implementations provided by PyTorch [51], as outlined in Table 2. We then derive the corresponding quantization metrics for them, also detailed in Table 2. Space limitations preclude the inclusion of further details in this section, but can be found in the Appendix.

Importantly, optimizers that use first/second-order moments require more complex operations rather than simply integrating the gradient into the parameter. They necessitate the ongoing management of their internal states, adjusting to the changing gradients. Truncating the mantissa may alter the gradient, subsequently leading to changes in the optimizer's internal state. This change, while a consequence of the quantization, introduces limited side-effects and does not typically pose a substantial issue in maintaining the model's training accuracy. For example, the maximum absolute deviation in gradient value is capped at 2^{-5} when truncating a 23-bit mantissa (in FP32) by 18 bits, aligned with the uppermost quantization level. This adjustment could be viewed as a damping coefficient (λ in Table 2) to the gradient, with the restriction set to 2^{i-23} when truncating the mantissa by i bits for a gradient in FP32 format. λ is a parameter that controls the effect of prior weight updates on the current update in momentum-based optimization algorithms during training.

Our experimental findings, detailed in Section 7.6, confirm that our quantization approach effectively maintains training accuracy, achieving nearly lossless fidelity and significantly outperforming the lossy compression methods.

Multi-Level Quantization. We introduce a multi-level quantization strategy to optimize mantissa compression, tailoring the level of compression based on the gradient's relative significance compared to its corresponding parameter's magnitude within the optimizer. To be specific, GComp categorizes quantization into four distinct levels. Each level is characterized by a particular number of truncation bits. These levels are represented by 2-bit binary codes in the encoded segment. Level 0 signifies no truncation, whereas higher levels correspond to a progressively greater count of truncation bits. For

Table 2: The quantization metrics for different optimizers.

Optimizer	Updating Method	Quantization Metric δ
SGD	$g_t = g_t + \lambda\theta_{t-1}$ ①; $\theta_t = \theta_{t-1} - \eta g_t$ ②.	$\frac{\theta_{t-1}}{\eta g_t} - \frac{\lambda\theta_{t-1}}{g_t}$
Momentum	$g_t = g_t + \lambda\theta_{t-1}$ ①; $b_t = \mu b_{t-1} + (1-\tau)g_t$ ②; $\theta_t = \theta_{t-1} - \eta b_t$ ③.	$\frac{\theta_{t-1} - \eta\mu b_{t-1}}{\eta(1-\tau)g_t} - \frac{\lambda\theta_{t-1}}{g_t}$
Nesterov Momentum	$g_t = g_t + \lambda\theta_{t-1}$ ①; $b_t = \mu b_{t-1} + g_t$ ②; $g_t = g_t + \mu b_t$ ③; $\theta_t = \theta_{t-1} - \eta g_t$ ④.	$\frac{(1-\eta\lambda(1+\mu))\theta_{t-1} - \eta\mu^2 b_{t-1}}{\eta(1+\mu)g_t}$
AdaGrad	$\tilde{\eta} = \eta/(1+(t-1)\lambda_{lr})$ ①; $g_t = g_t + \lambda\theta_{t-1}$ ②; $r_t = r_{t-1} + g_t^2$ ③; $\theta_t = \theta_{t-1} - \frac{\tilde{\eta} g_t}{\sqrt{r_t}+\epsilon}$ ④.	$\frac{\theta_{t-1}(\sqrt{r_t}+\epsilon)}{\tilde{\eta} g_t} - \frac{\lambda\theta_{t-1}}{g_t}$
RMSProp	$g_t = g_t + \lambda\theta_{t-1}$ ①; $v_t = \alpha v_{t-1} + (1-\alpha)g_t^2$ ②; $\theta_t = \theta_{t-1} - \frac{\eta g_t}{\sqrt{v_t}+\epsilon}$ ③.	$\frac{\theta_{t-1}(\sqrt{v_t}+\epsilon)}{\eta g_t} - \frac{\lambda\theta_{t-1}}{g_t}$
Adam	$g_t = g_t + \lambda\theta_{t-1}$ ①; $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$ ②; $v_t = \beta_2 v_t + (1-\beta_2)g_t^2$ ③; $\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$ ④; $\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$ ⑤; $\theta_t = \theta_{t-1} - \frac{\eta\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon}$ ⑥.	$\frac{\theta_{t-1}(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t) - \eta\beta_1 m_{t-1}}{\eta(1-\beta_1)g_t} - \frac{\lambda\theta_{t-1}}{g_t}$
AdamW	$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$ ①; $v_t = \beta_2 v_t + (1-\beta_2)g_t^2$ ②; $\widehat{m_t} = \frac{m_t}{1-\beta_1^t}$ ③; $\widehat{v_t} = \frac{v_t}{1-\beta_2^t}$ ④; $\theta_t = \theta_{t-1} - \eta(\frac{\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon} + \lambda\theta_{t-1})$ ⑤.	$\frac{\theta_{t-1}(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)(1-\eta\lambda) - \eta\beta_1 m_{t-1}}{\eta(1-\beta_1)g_t}$

instance, considering a 23-bit mantissa from a value in FP32 format, the four levels would correspond to truncating 0, 6, 12, and 18 bits, respectively.

5.2 Pruning of Zero-valued Gradients

In Section 3, we analyze the prevalence of zero-valued gradients in training, and GComp employs a pruning technique tailored for these gradients. Given that both the exponent and mantissa of a zero-valued floating-point number are zero, the mantissa can be safely discarded. To represent these zero-valued gradients, GComp only retains the Huffman encoding associated with the exponent part (that is, the encoding for the source symbol of zero). Notice that the floating-point number system includes denormalized (or subnormal) numbers, which possess a zero exponent alongside a non-zero mantissa to represent extremely small non-zero numerical values. For instance, the greatest absolute value of a denormalized number in the FP32 format is capped at 2^{-126}. Within GComp, all such denormalized numbers are treated as equivalent to zero since they rarely occur within the gradient values throughout DNN training, and this infinitesimal error resulting from zero-gradient pruning is deemed negligible.

6 IMPLEMENTATION

In this section, we detail the implementation of the compression algorithms introduced in Section 4 and Section 5. We start with an overview of the compression workflow, and then proceed to elucidate the integration of GComp with the AllReduce operation in the Gloo [16] library, which serves as a standard communication backend in PyTorch.

6.1 Compression/Decompression Process

Encoding a Single Value. The workflow for encoding a single value follows the procedure outlined in Figure 4. Initially, the original floating-point number is processed through a shifting operation that isolates the exponent and mantissa, with the sign bit incorporated into the mantissa. Following this, the value undergoes a transformation into its compressed form through the following steps:

(1) *Exponent encoding:* The exponent part of a gradient value is allocated a specific Huffman code, which is obtained from a Huffman encoding table. This table is constructed leveraging prior knowledge and undergoes periodic updates, to accommodate the distribution of exponent values in the gradients. After this Huffman code is established, it is placed into the segment allocated for it within the targeted buffer.

(2) *Quantization level calculation:* For the mantissa, a check is performed to determine if the exponent value is non-zero. If it is, the quantization metric is calculated as detailed in Section 5.1, which aids in establishing the appropriate quantization level for the mantissa. The resulting 2-bit quantization level is then placed into the quantization level segment of the target buffer.

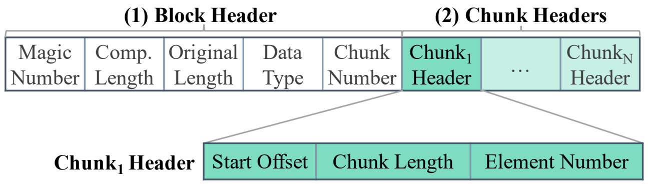


Figure 7: Metadata layout for a compressed data block.

(3) *Mantissa quantization and truncation:* With the quantization level ascertained, the mantissa undergoes truncation to align with the determined level. The truncated mantissa is then populated into the segment reserved for it.

Decoding a Single Value. The first step of the decompression process entails accessing the Huffman coding portion of the compressed data, followed by retrieving the original exponent value using the lookup table as described in Section 4. If the original exponent is not zero, the process continues with the reading of both the quantization level and the quantized mantissa segments. Thereafter, the quantized mantissa undergoes a bit-shift, determined by the quantization level, to reconstruct the mantissa and sign bit. These elements are then combined to form a complete floating-point number that represents the gradient value. If the original exponent is zero, the result is a direct output of a floating-point number valued at zero.

Data Layout. In communication between workers triggered by AllReduce, gradients are exchanged as data blocks, where each block could contain multiple data chunks in GComp. A comprehensive data block in GComp includes not only the compressed values but also a data segment for the metadata, which precedes the compressed values, to provide the necessary information for the decoding process. This metadata layout is illustrated in Figure 7 and encompasses two key components: (1) The block header, which holds a magic number to verify the data block's legitimacy, information about the total length of the compressed values (measured in bits), the total length of the original values (also in bits), the data type of the original values (e.g., FP32), and the count of data chunks within this block. (2) The data chunk headers, each presenting essential details such as the starting offset for the values in the particular chunk, the chunk's total length (in bits), and the total number of values the chunk contains.

6.2 Integration with AllReduce

We have implemented the compression and decompression algorithms of GComp by customizing the AllReduce operation within Gloo [16], which serves as a standard communication backend in PyTorch and Megatron. The AllReduce operation is a critical component whose implementation is similar among various libraries, including NCCL [48] and MPI [49], which suggests the potential for integrating GComp into these libraries with minimal adjustments.

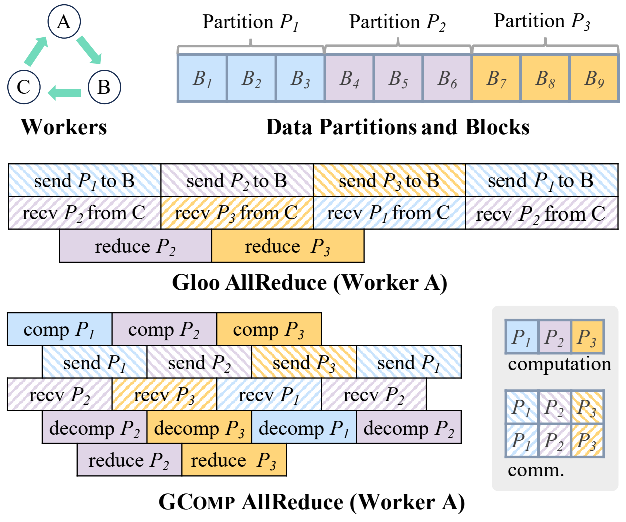


Figure 8: Integration of GComp into the AllReduce operation (with three workers).

We integrate GComp into the ring-based AllReduce pattern in Gloo. This pattern divides the entire processed tensor into multiple partitions, with the number equating to the count of workers. As illustrated in Figure 8, in the case where three workers participate in the operation, each worker—for example, worker A—first engages in a send/receive interaction for a partition with its adjacent workers in the ring and then undertakes the reduction operation on the received data. This process is iterated $(N-1)$ times, where N represents the number of workers. Then, every worker synchronizes an ultimately reduced partition to the following worker in the ring, repeating this step $(N-1)$ times as well.

Within GComp, each tensor partition involved in the AllReduce operation is split into multiple data blocks, which are then further divided into chunks and processed in parallel by multiple threads. This concurrent compression and decompression process is integrated into the AllReduce operation, as shown in Figure 8. Although adopting this method incurs additional computational overhead, the parallel and pipelined execution ensures that the gains in reducing communication overhead far exceed these extra computational costs. Consequently, it results in a notable decrease in the total execution time for the AllReduce operation, achieving a reduction of as much as 53%, as detailed in Section 7.2.

7 EVALUATION

In this section, we first evaluate GComp's compression efficiency focusing on the compression rate and performance improvements of AllReduce. We then conduct micro-benchmarks to assess the individual components of GComp. We also present an end-to-end comparison of training performance and analyze the impact of GComp on model convergence against established baselines.

7.1 Experimental Setup

Platform. All experiments are conducted on PAI (Platform for AI) [10], a machine learning platform provided by Alibaba Cloud. For the evaluation, a total of 4 instances are utilized. Each instance is equipped with a single NVIDIA A100 (40GB) GPU, 12 CPU cores, and 94GB of CPU memory. Each instance serves as an individual DP group in the distributed training setup, and instances are interconnected using 10Gbps Ethernet NICs. Except for the scalability tests, all other experiments are conducted using all four instances.

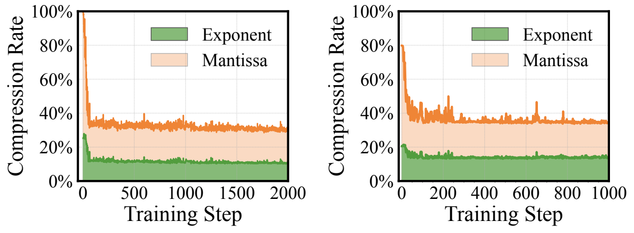
Workloads. We have selected two classical DNN models: ResNet-50 [22], comprising 23.6 million parameters, is trained on the ImageNet1K dataset [54], a standard benchmark for image classification tasks; Bert-base [12], which has 110 million parameters, is trained on the SST dataset [58], tailored for sentiment analysis. These models were chosen due to their wide usage in the field and their varied structures and parameter sizes. For these models, parameters, gradients, and the intermediate values during training are all represented in the FP32 format.

Frameworks. Our implementation of GComp integrates compression and decompression modules into the AllReduce of the Gloo [16] library. To evaluate the efficacy of our approach in practical settings, we benchmark the training performance using PyTorch [51], a prominent framework for distributed DNN training, through its data parallel architecture. Gloo is recognized as one of the default communication backends in PyTorch. It is noteworthy that other advanced frameworks for distributed training, such as Megatron [57] and DeepSpeed [53], which are built upon the PyTorch framework, can also benefit from incorporating GComp for similar enhancements in performance.

7.2 Compression Efficiency

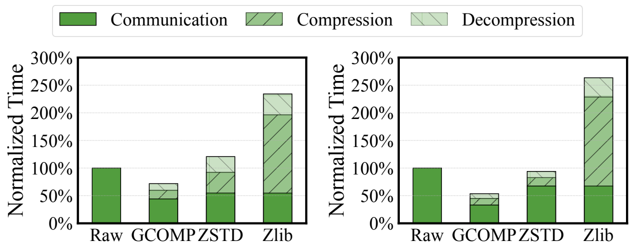
Compression Rate. To evaluate the compression efficiency of GComp, we first measure the achieved compression rate, calculated as the ratio of the combined compressed size (including both exponent and mantissa portions) to the original gradient size. This evaluation is conducted using ResNet-50 and Bert-base models over 2000 and 1000 training iterations, respectively. The findings are depicted in Figure 9, showing that the compression methods applied to both exponent and mantissa demonstrate effective compression, as the gradient volume quickly decreases with ongoing training. Taken together, GComp reaches an average compression rate of 32.9% for ResNet-50 and 36.5% for Bert-base. This translates to a reduction of over 60% in the volume of transmitted gradients.

AllReduce Performance. Figure 10 presents the performance improvements of the AllReduce operation achieved by integrating GComp, depicted as normalized time, which represents the ratio of the operation's duration with GComp integration to its duration without any compression. The results



(a) ResNet-50. (b) Bert-base.

Figure 9: Compression rate changes during training.



(a) ResNet-50. (b) Bert-base.

Figure 10: The normalized time for the AllReduce with different compression methods.

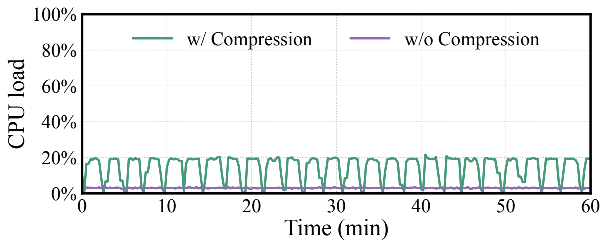
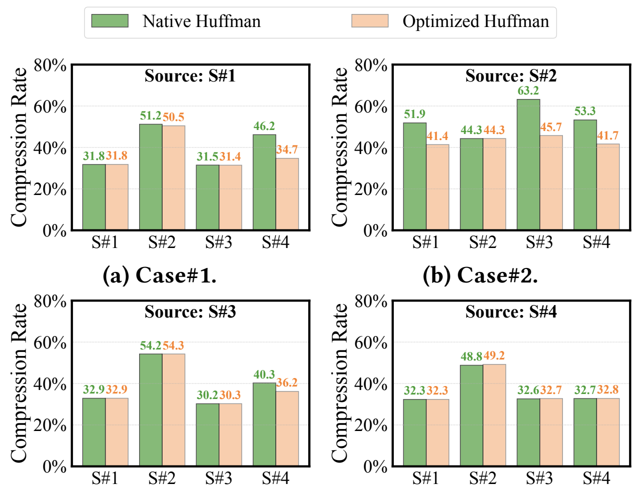


Figure 11: Monitoring CPU load during training.

clearly demonstrate that GComp offers a substantial boost in AllReduce's efficiency, reducing its average execution time to merely 61.6% for ResNet-50 and 47.0% for the Bert-base model, in comparison with the baseline duration. This reduction significantly mitigates the communication overhead during DNN training.

We also use various representative lossless compression methods as comparisons. As shown in Figure 10, although these lossless compression methods (ZSTD and Zlib) could decrease the communication time, their complex encoding/decoding phases overwhelm the gains and result in worse end-to-end AllReduce performance. Some lossless compression methods may even lead to 10× increased consumed time owing to the slow compression procedure (not included in the figure). Therefore, we do not include lossless compression methods as alternatives in other evaluations.

CPU Load. Figure 11 illustrates the CPU load monitoring during the training of Bert-base, both with and without GComp enabled. These results indicate that while GComp imposes some additional load on the CPU, the CPU usage remains below 21.6%. In comparison, when no compression is



(a) Case#1. (b) Case#2.

(c) Case#3. (d) Case#4.

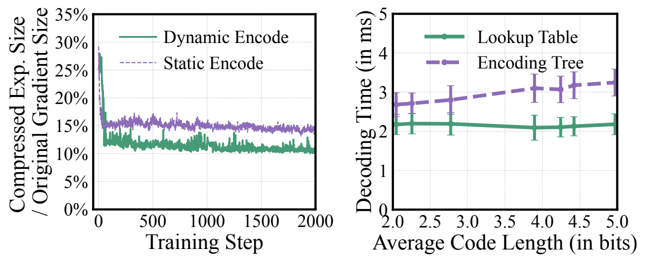
Figure 12: Compression rate of the exponent part when using the optimized Huffman code and the native Huffman code, each subfigure using a different information source for building the Huffman codebook.

applied, the CPU load stays under 3.6%, suggesting that CPUs are largely idle during training. Therefore, the associated overhead from GComp is deemed acceptable.

7.3 Exponent Compression Benchmarks

Full Transmission of Long Codes. To demonstrate the robustness of the exponent compression strategy in the face of varying data distributions, we design cross-validation experiments targeting diverse gradient exponent distributions. We first randomly sample four distinct sets of gradient data from a training process of ResNet-50, each set corresponding to a single data block. These samples display marked variance in exponent value distributions and are designated as S#1 to S#4. We then utilize each of these four datasets as the source to establish the encoding scheme, perform cross-compression for all four datasets using the established codes, and measure the compression rate of our optimized method (with full transmission of long codes) against the native Huffman encoding. Figure 12 illustrates the experimental findings. In the majority of instances, GComp's compression rates are on par with those achieved using native Huffman encoding. Notably, in scenarios characterized by significant deviations in exponent distribution, the performance of GComp's strategy is superior to that of the native one. This underscores GComp's enhanced robustness, demonstrating its capability to withstand variations and distributional shifts in gradients throughout the training process.

Dynamic Encoding Table Adjustment. Figure 13a illustrates the effectiveness of dynamically adjusting the Huffman encoding table during the training. This involves updating



(a) Compressed exponent size. (b) Decoding time.

Figure 13: The compressed exponent size using different encoding methods (a). The decoding time of Huffman codes using different decoding methods (b).

the encoding scheme at intervals of every 50 iterations while training ResNet-50 for 2000 iterations. The performance is compared with that of a static encoding table, which is predetermined offline prior to training. To assess the efficiency, we measure the total compressed size of the exponent portion for all values and express it as a ratio relative to the original size of the gradients before compression. The results clearly indicate that dynamic adjustment brings a reduction of the average ratio from 14.5% down to 11.6%.

Decoding based on a Lookup Table. To validate the efficiency of the lookup-table-based decoder, we conduct an experiment where we implement both the lookup-table-based decoder and the encoding-tree-based algorithm. We then compare their decoding efficiencies across exponent segments encoded with Huffman coding. To evaluate the impact of code length on decoding efficiency, we select test cases with varying average code lengths from the training gradient data. The average decoding time of a single data block for each test case is measured for both decoding algorithms, and the results are presented in Figure 13b. These findings indicate that the lookup-table-based method consistently outperforms the encoding tree-based approach across all test cases. An advantage of the lookup-table-based method is that it requires only a single lookup operation for each code, regardless of code length. Therefore, its decoding time remains stable and does not increase as the average code length grows. Conversely, the tree-based method involves a bit-by-bit traversal of the code segment for decoding purposes. As a result, any increase in code length directly translates to a longer decoding time.

7.4 Mantissa Compression Benchmarks

Ablation Study. We conduct an ablation study to evaluate the effectiveness of our optimization strategies for mantissa compression, as shown in Figure 14. Our baseline involves the application of single-level quantization to the mantissa component of gradients, herein referred to as "Single-Quant". This approach corresponds to the highest truncation level within the multi-level quantization scheme (denoted

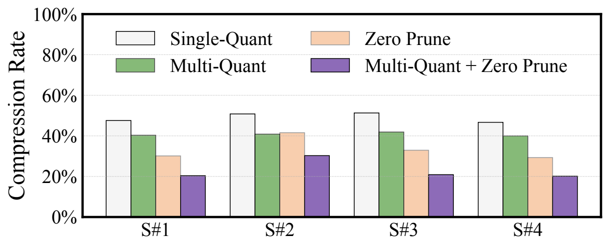


Figure 14: Compression rate of the mantissa part using different strategies.

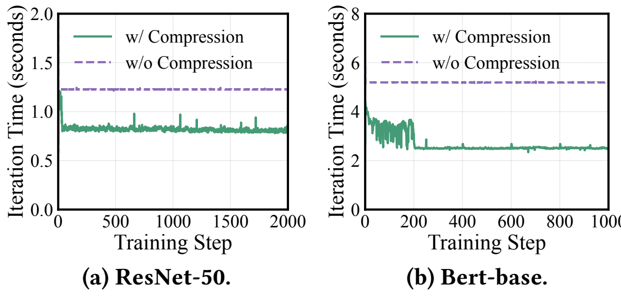


Figure 15: Iteration time with and without GComp.

as "Multi-Quant") of GComp, specifically entails truncating 18 bits from a 23-bit mantissa. We also consider the "Zero Prune" method, which exclusively employs a zero-valued gradient pruning technique. A combination of the two strategies, known as "Multi-Quant + Zero Prune", represents the full-scale mantissa compression approach of GComp. The evaluation utilizes four distinct datasets—S#1 to S#4—each characterized by a unique gradient distribution as observed in the previous experiment. The resulting data indicates that both multi-level quantization and zero-valued gradient pruning significantly enhance the compression efficiency. Moreover, integrating both strategies emerges as the superior method, achieving the most effective performance.

7.5 Training Efficiency

Iteration Time. Figure 15 showcases the training efficiency of ResNet-50 and Bert-base models, both with and without the integration of GComp. The metric we focus on here is the execution time required for each training iteration. The findings reveal that GComp significantly boosts the overall training efficiency for both models, resulting in an average speedup of 1.5× for ResNet-50 and 1.9× for Bert-base. This performance enhancement is due to GComp's effective reduction in communication overhead—specifically, the time consumed during the AllReduce operation—achieved through its compression and decompression mechanisms. The practical implications of these findings are substantial: by reducing the training duration with the same computational resource demands, GComp empowers a more efficient and cost-effective methodology for model training.

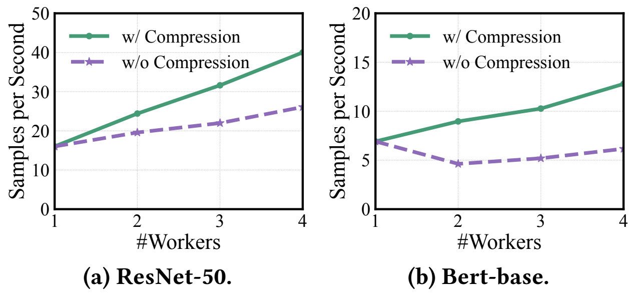


Figure 16: Training scalability with and without GComp (evaluation results).

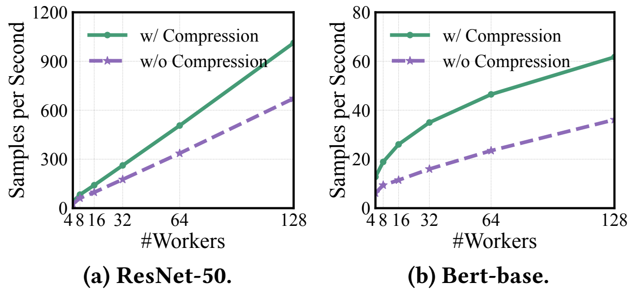


Figure 17: Training scalability with and without GComp (simulation results).

Scalability. We further evaluate the scalability of training these models with and without the integration of GComp. This evaluation spans from a single worker to four workers, each representing an individual instance on the cloud platform. Figure 16 depicts the number of samples trained per second for each model across varying worker configurations. This metric is commonly employed to measure the throughput of the training process and, consequently, its scalability. When utilizing only one worker, the training speeds of the models with GComp are same as those without GComp, as this scenario does not introduce any communication overhead. However, as the number of workers increases, the training speed without GComp encounters limitations due to escalating communication demands. In comparison, with GComp, the throughput increases by 53.4% and 107% under the training of ResNet and Bert, respectively.

To evaluate the scalability of GComp on larger-scale clusters, we conduct simulations using four workers to replicate the behaviors of additional workers. The maximum iteration time across all participating workers represents the actual iteration time. Figure 17 illustrates the simulated results, with the number of workers scaling from 4 to 128. The findings indicate that GComp significantly improves training speed for both models. Specifically, with GComp, the throughput increases by 35.3% ~ 53.4% for ResNet and 70.8% ~ 127% for Bert. These results underline GComp's effectiveness in mitigating communication overhead and enhancing distributed

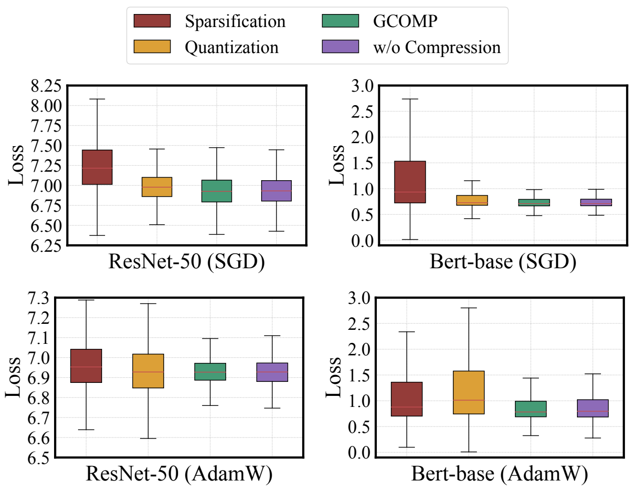


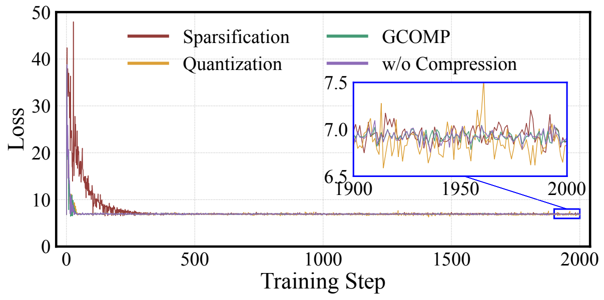
Figure 18: Distribution of loss deviations compared with the baseline training without any compression.

training efficiency. Additionally, it is important to note that, theoretically, the compression rate of GComp remains consistent across different scales. The total compression and decompression overhead for a single worker also exhibits consistent complexity. This is illustrated in the workflow shown in Figure 8, where the total data size for compression remains stable, even though the data partition number changes. This property allows GComp to be effectively utilized across clusters of various sizes.

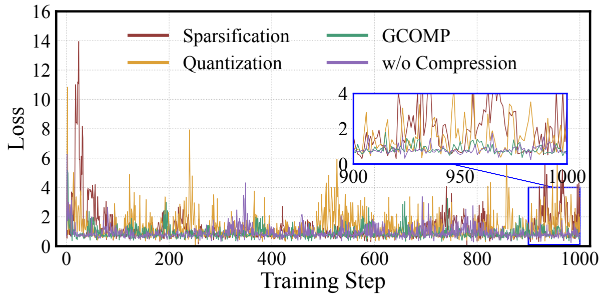
7.6 Impact on Model Convergence

The primary objective of training is to minimize a loss function. This function quantifies the discrepancy between the network's predicted outputs and the actual, true outputs. To assess the effect of GComp on model convergence, we track the values of the loss function throughout the training period with GComp integrated. We compare these results to those obtained without GComp, providing a conventional benchmark for assessment. Additionally, we explore two alternative compression techniques: (1) a quantization method that reduces the precision of gradient values by truncating 18 bits from the mantissa; and (2) a sparsification approach that performs the AllReduce operation in a sparse manner, carrying it out once every 8 iterations. For the optimizers, we select SGD and AdamW for comparative analysis. SGD represents the most fundamental approach, while AdamW stands out as the most sophisticated update mechanism, among the methods in Table 2.

Distribution of Loss Deviations. Figure 18 presents the distribution of the training loss as compared to the baseline without any compression, in the form of box plots. For both SGD-based and AdamW-based trainings, the data demonstrates that the variance in loss when GComp is employed is considerably tighter than with alternate compression methods. Compared to the quantization method, GComp decreases 73.6% ~ 93.1% of the average loss variance. Similarly, compared with the sparsification method, GComp decreases 77.4% ~ 96.8% of the average loss variance. As a consequence, the overall convergence trend remains largely consistent with the integration of GComp, which notably outperforms other compression strategies.



(a) ResNet-50.



(b) Bert-base.

Figure 19: The value of loss function during the ResNet-50 and Bert-base training using the AdamW optimiizer, with different compression strategies.

Evolution of Loss Values. Figure 19 shows the evolution of the loss values during the training of ResNet-50 and Bert-base, using different compression strategies with AdamW as the optimizer. It reveals that convergence trend observed with the application of GComp remarkably mirror those of the training procedure without GComp, outperforming both quantization-based and sparsification-based methods. This suggests that our compression strategy does not detrimentally affect model convergence.

End-to-end Training Time. To compare end-to-end training times for convergence, we gathered data on the durations required to reach specific convergence levels—defined as maintaining a loss below an established threshold (7.05 for ResNet-50 and 1.08 for Bert-base) for over 10 iterations. As shown in Figure 20, GComp achieves convergence more quickly than the baseline methods. In contrast, the quantization and sparsification techniques require longer time to converge compared to scenarios without compression, as they negatively impact training accuracy. These results

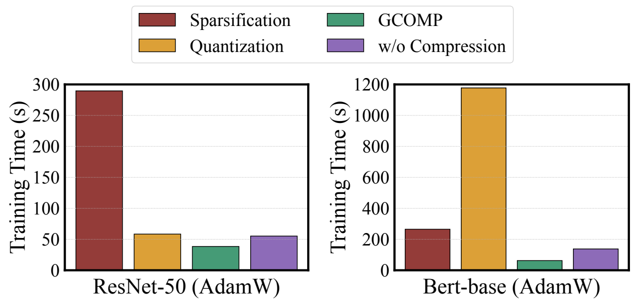


Figure 20: End-to-end training time to reach specific convergence levels.

demonstrate that GComp offers significant benefits in end-to-end DNN training.

8 DISCUSSION

Applicability and Limitations. As FP32 is the most prevalent format for training DNNs, the results presented in this paper are applicable to a wide range of real-world scenarios. For lower-precision floating-point formats such as FP16, BF16 [27] and TF32 [47], where the lengths of the exponent part or the mantissa part are much shorter than in FP32, the effectiveness of GComp's compression tends to decrease. Another limitation arises when the communication of DP is largely overlapped by computational operators [30, 38]; in such cases, the significance of communication compression is reduced since DP communication does not primarily appear on the critical path of a training iteration.

More Efficient Quantization Metrics. Current quantization metrics primarily focus on the relationship between the gradient and parameter, whose efficiency could be further improved. Actually, in many sophisticated optimizers, the gradient firstly updates the momentum, and then the parameter, suggesting that we could consider these three values as a whole to obtain more efficient quantization metrics. We leave the exploitation of more efficient quantization metrics as our future work.

9 RELATED WORK

DNN Training Frameworks. PyTorch [51] is widely acknowledged as a pivotal framework within the deep learning community for the training of neural networks. Several frameworks, such as Megatron [57] and DeepSpeed [53], are built upon PyTorch and introduce various optimizations to enhance the training efficiency of large DNN models. These optimizations encompass advancements in memory management, parallelism, and scalability, among other areas. While the current implementation of GComp is based on PyTorch and Gloo [16], it is crucial to emphasize that GComp maintains its applicability across a wide range of frameworks due to its reliance on the AllReduce operation, which is a common communication bottleneck in training processes.

Communication Optimizations in DNN Training. To mitigate communication overhead during DNN training, there have been numerous studies, include advanced collective communication algorithms [8, 11, 13, 28, 56], optimized communication architectures [20, 29, 68], and network-level optimizations [30, 39, 67]. GComp complements these strategies by focusing on gradient compression to minimize the data volume communicated. Consequently, it can be combined with these optimizations to enhance training efficiency even further.

Gradient Compression. Some studies have focused on gradient compression through methods such as quantization [4, 7, 14, 35, 55, 60], sparsification [3, 37, 41, 59, 66], low-rank decomposition [62, 63, 69], and other techniques [6, 64, 65]. THC [35] is the latest quantization-based compression method, which introduces additional negotiations among all nodes to optimize parameter updates. However, traditional compression approaches often require trade-offs between compression ratio and accuracy, which can be problematic in practice. GComp exploits the statistical characteristics of gradients throughput DNN training, minimizing the impact of compression on the training process while still achieving significant compression efficiency. GComp can be combined with the lossy compression techniques to further diminish the volume of communication.

Floating-Point Compression. Efforts directed toward general floating-point compression [2, 36, 40, 42], albeit not explicitly designed for gradients in DNN training, have also been made. These studies have developed strategies incorporating predictive schemes, adaptive encoding, and other techniques for compressing general floating-point data. GComp sets itself apart by leveraging the unique characteristics of DNN training to formulate a more efficient compression algorithm.

10 CONCLUSION

In this paper, we propose GComp, a near-lossless gradient compression scheme designed to address the significant communication overhead in data-parallel distributed training of DNNs. By exploiting the statistical characteristics of gradient distributions, GComp achieves near-lossless compression, which effectively reduces communication volume by as much as 67.1% and improves the training efficiency by up to 1.9×, showing it is a promising solution for accelerating and scaling DNN training.

ACKNOWLEDGMENTS

We thank our shepherd Prof. Anil Madhavapeddy and the anonymous reviewers for their valuable comments and helpful suggestions. This research is partially supported by the National Natural Science Foundation of China (Grants No. 62221003 and No. 62402025) and the Fundamental Research Funds for the Central Universities.

REFERENCES

[1] Mark Adler. 2024. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://www.zlib.net/.

[2] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

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

[4] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. QSGD: Communication-efficient SGD via gradient quantization and encoding. *Advances in neural information processing systems* 30 (2017).

[5] Amazon Web Services. 2024. Amazon Web Services. https://aws.amazon.com/.

[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*. 359–375.

[7] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Animashree Anandkumar. 2018. signSGD: Compressed optimisation for non-convex problems. In *International Conference on Machine Learning*. PMLR, 560–569.

[8] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 62–75.

[9] Alibaba Cloud. 2024. Alibaba Cloud: Reliable and Secure Cloud Computing Services. https://www.alibabacloud.com/.

[10] Alibaba Cloud. 2024. Platform for AI. https://www.alibabacloud.com/en/product/machine-learning.

[11] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. 2023. Mscclang: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 502–514.

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

[13] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, et al. 2021. Accl: Architecting highly scalable distributed training systems with highly efficient collective communication library. *IEEE Micro* 41, 5 (2021), 85–92.

[14] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. IEEE, 1–8.

[15] Facebook. 2024. ZSTD. https://github.com/facebook/zstd.

[16] Facebook AI Research. 2019. Gloo: a collective communications library. https://github.com/facebookincubator/gloo.

[17] Shiqing Fan et al. 2021. DAPPLE: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 431–445.

[18] FSF & GNU. 2024. GNU Gzip. https://www.gnu.org/software/gzip/.

[19] Google. 2024. Google Cloud Platform. https://cloud.google.com/.

[20] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. 2020. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*. Springer, 41–59.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

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

[23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[24] David A Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[25] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic resource sharing for distributed deep learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[26] IEEE. 2019. IEEE Standard for Floating-Point Arithmetic. https://doi.org/10.1109/IEEESTD.2019.8766229 [Online; accessed 27-February-2024].

[27] Intel. 2018. BFLOAT16 – Hardware Numerics Definition. https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf [Online; accessed 5-October-2024].

[28] Abhinav Jangda et al. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.

[29] 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)*. 463–479.

[30] Ziheng Jiang et al. 2024. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[31] jseward. 2024. bzip2 and libbzip2. https://sourceware.org/bzip2/.

[32] Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. 2021. Amazon sagemaker model parallelism: A general and flexible framework for large model training. *arXiv preprint arXiv:2111.05972* (2021).

[33] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. 2023. BPIPE: memory-balanced pipeline parallelism for training large language models. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 682, 15 pages.

[34] Abraham Lempel and Jacob Ziv. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* IT-23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

[35] Minghao Li, Ran Ben Basat, Shay Vargaftik, ChonLam Lao, Kevin Xu, Michael Mitzenmacher, and Minlan Yu. 2024. THC: Accelerating Distributed Deep Learning Using Tensor Homomorphic Compression. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1191–1211. https://www.usenix.org/conference/nsdi24/presentation/li-minghao

[36] Ruiyuan Li, Zheng Li, Yi Wu, Chao Chen, and Yu Zheng. 2023. Elf: Erasing-Based Lossless Floating-Point Compression. *Proc. VLDB Endow.* 16, 7 (mar 2023), 1763–1776. https://doi.org/10.14778/3587136.3587149

[37] Shigang Li and Torsten Hoefler. 2022. Near-optimal sparse allreduce for distributed deep learning. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–149.

[38] 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. *Proceedings of the VLDB Endowment* (2020).

[39] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: High precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[40] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.

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

[42] Peter Lindstrom and Martin Isenburg. 2006. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics* 12, 5 (2006), 1245–1250.

[43] Microsoft Azure. 2024. Microsoft Azure. https://azure.microsoft.com/.

[44] Deepak Narayanan et al. 2019. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[45] Deepak Narayanan et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[46] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*. PMLR, 7937–7947.

[47] NVIDIA. 2020. TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x. https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/ [Online; accessed 27-April-2024].

[48] NVIDIA Corporation. 2019. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl.

[49] Open MPI Project. 2019. Open MPI: A High Performance Message Passing Library. https://www.open-mpi.org/.

[50] OpenAI. 2020. Language Models are Few-Shot Learners. https://openai.com/blog/gpt-3-apps.

[51] Adam Paszke et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[52] Python. 2024. lzma — Compression using the LZMA algorithm. https://docs.python.org/3/library/lzma.html.

[53] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[54] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[55] 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*.

[56] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.

[57] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[58] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1631–1642. https://www.aclweb.org/anthology/D13-1170

[59] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. 2018. Sparsified SGD with memory. *Advances in neural information processing systems* 31 (2018).

[60] Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, and Yuxiong He. 2021. 1-bit adam: Communication efficient large-scale training with adam's convergence speed. In *International Conference on Machine Learning*. PMLR, 10118–10129.

[61] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

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

[63] Hongyi Wang, Scott Sievert, Shengchao Liu, Zachary Charles, Dimitris Papailiopoulos, and Stephen Wright. 2018. Atomo: Communication-efficient learning via atomic sparsification. *Advances in neural information processing systems* 31 (2018).

[64] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Ng. 2022. Bytecomp: Revisiting gradient compression in distributed training. *arXiv preprint arXiv:2205.14465* (2022).

[65] 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*. 867–882.

[66] Jianqiao Wangni, Jialei Wang, Ji Liu, and Tong Zhang. 2018. Gradient sparsification for communication-efficient distributed optimization. *Advances in Neural Information Processing Systems* 31 (2018).

[67] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN tuning for high-speed datacenter networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 384–397.

[68] Mingran Yang, Alex Baban, Valery Kugel, Jeff Libby, Scott Mackie, Swamy Sadashivaiah Renu Kananda, Chang-Hong Wu, and Manya Ghobadi. 2022. Using trio: juniper networks' programmable chipset-for emerging in-network applications. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 633–648.

[69] Lin Zhang, Longteng Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2023. Evaluation and optimization of gradient compression for distributed deep learning. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 361–371.

APPENDIX

A OPTIMIZER QUANTIZATION METRICS

A.1 Momentum

The Momentum optimizer updates the parameters as follows:

$$\begin{aligned} g_t &= g_t + \lambda\theta_{t-1}, \\ b_t &= \mu b_{t-1} + (1-\tau)g_t, \\ \theta_t &= \theta_{t-1} - \eta b_t \end{aligned} \tag{9}$$

This simplifies to: $\theta_t = \theta_{t-1} - \eta(\mu b_{t-1} + (1-\tau)(g_t + \lambda\theta_{t-1})) = \theta_{t-1} - \eta\mu b_{t-1} - \eta(1-\tau)\lambda\theta_{t-1} - \eta(1-\tau)g_t$. If $\frac{\theta_{t-1}-\eta\mu b_{t-1}-\eta(1-\tau)\lambda\theta_{t-1})}{\eta(1-\tau)g_t} > 2^n$, discarding the last n bits of $\eta(1-\tau)g_t$ hardly affect the value of θ_t. Therefore, we define the quantization metric as: $\delta = \frac{\theta_{t-1}-\eta\mu b_{t-1}-\eta(1-\tau)\lambda\theta_{t-1}}{\eta(1-\tau)g_t} = \frac{\theta_{t-1}-\eta\mu b_{t-1}}{\eta(1-\tau)g_t} - \frac{\lambda\theta_{t-1}}{g_t}$.

A.2 Nesterov Momentum

For the Nesterov Momentum optimizer, we have:

$$\begin{aligned} g_t &= g_t + \lambda\theta_{t-1}, \\ b_t &= \mu b_{t-1} + g_t, \\ g_t &= g_t + \mu b_t, \\ \theta_t &= \theta_{t-1} - \eta g_t \end{aligned} \tag{10}$$

As a result, $\theta_t = \theta_{t-1} - \eta(g_t + \lambda\theta_{t-1} + \mu(\mu b_{t-1} + (g_t + \lambda\theta_{t-1}))) = \theta_{t-1} - \eta\lambda\theta_{t-1} - \eta\mu^2 b_{t-1} - \eta\mu\lambda\theta_{t-1} - \eta(1+\mu)g_t$.
Similarly, if $\frac{\theta_{t-1}-\eta\lambda\theta_{t-1}-\eta\mu^2 b_{t-1}-\eta\mu\lambda\theta_{t-1}}{\eta(1+\mu)g_t} > 2^n$, discarding the last n bits of $\eta(1+\mu)g_t$ has a negligible effect on the value of θ_t. Therefore, we define the quantization metric as: $\delta = \frac{\theta_{t-1}-\eta\lambda\theta_{t-1}-\eta\mu^2 b_{t-1}-\eta\mu\lambda\theta_{t-1}}{\eta(1+\mu)g_t} = \frac{(1-\eta\lambda(1+\mu))\theta_{t-1}-\eta\mu^2 b_{t-1}}{\eta(1+\mu)g_t}$.

A.3 AdaGrad

The AdaGrad optimizer is defined by:

$$\begin{aligned} \tilde{\eta} &= \frac{\eta}{(1+(t-1)\lambda_{lr})}, \\ g_t &= g_t + \lambda\theta_{t-1}, \\ r_t &= r_{t-1} + g_t^2, \\ \theta_t &= \theta_{t-1} - \frac{\tilde{\eta} g_t}{\sqrt{r_t} + \epsilon} \end{aligned} \tag{11}$$

This leads to the expression: $\theta_t = \theta_{t-1} - \frac{\tilde{\eta}(g_t+\lambda\theta_{t-1})}{\sqrt{r_t}+\epsilon} = \theta_{t-1} - \frac{\tilde{\eta} g_t}{\sqrt{r_t}+\epsilon} - \frac{\tilde{\eta}\lambda\theta_{t-1}}{\sqrt{r_t}+\epsilon}$. If the condition $\frac{\theta_{t-1}(\sqrt{r_t}+\epsilon)-\tilde{\eta}\lambda\theta_{t-1}}{\tilde{\eta} g_t} > 2^n$ holds, then discarding the last n bits of $\tilde{\eta} g_t$ has a negligible effect on the value of θ_t. To quantify this, we define the quantization metric: $\delta = \frac{\theta_{t-1}(\sqrt{r_t}+\epsilon)-\tilde{\eta}\lambda\theta_{t-1}}{\tilde{\eta} g_t} = \frac{\theta_{t-1}(\sqrt{r_t}+\epsilon)}{\tilde{\eta} g_t} - \frac{\lambda\theta_{t-1}}{g_t}$.

A.4 RMSProp

The update equations for RMSProp are given by:

$$\begin{aligned} g_t &= g_t + \lambda\theta_{t-1}, \\ v_t &= \alpha v_{t-1} + (1-\alpha)g_t^2, \\ \theta_t &= \theta_{t-1} - \frac{\eta g_t}{\sqrt{v_t} + \epsilon} \end{aligned} \tag{12}$$

This leads to: $\theta_t = \theta_{t-1} - \frac{\eta(g_t+\lambda\theta_{t-1})}{\sqrt{v_t}+\epsilon} = \theta_{t-1} - \frac{\eta g_t}{\sqrt{v_t}+\epsilon} - \frac{\eta\lambda\theta_{t-1}}{\sqrt{v_t}+\epsilon}$. Similarly, if $\frac{\theta_{t-1}(\sqrt{v_t}+\epsilon)-\eta\lambda\theta_{t-1}}{\eta g_t} > 2^n$, we could discard the last n bits of ηg_t. Thus, we define the quantization metric as: $\delta = \frac{\theta_{t-1}(\sqrt{v_t}+\epsilon)-\eta\lambda\theta_{t-1}}{\eta g_t} = \frac{\theta_{t-1}(\sqrt{v_t}+\epsilon)}{\eta g_t} - \frac{\lambda\theta_{t-1}}{g_t}$.

A.5 Adam

The Adam optimizer updates its parameters as follows:

$$\begin{aligned} g_t &= g_t + \lambda\theta_{t-1}, \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t^2, \\ \widehat{m_t} &= \frac{m_t}{1-\beta_1^t}, \\ \widehat{v_t} &= \frac{v_t}{1-\beta_2^t}, \\ \theta_t &= \theta_{t-1} - \frac{\eta\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \end{aligned} \tag{13}$$

Consequently, we can express θ_t as $\theta_t = \theta_{t-1} - \frac{\eta m_t}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)} = \theta_{t-1} - \frac{\eta\beta_1 m_{t-1}}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)} - \frac{\eta(1-\beta_1)g_t}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)} - \frac{\eta(1-\beta_1)\lambda\theta_{t-1}}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)}$.
If $\frac{\theta_{t-1}(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)-\eta\beta_1 m_{t-1}-\eta(1-\beta_1)\lambda\theta_{t-1}}{\eta(1-\beta_1)g_t} > 2^n$, the last n bits of $\eta(1-\beta_1)g_t$ could be discarded, thus the metric is defined as: $\delta = \frac{\theta_{t-1}(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)-\eta\beta_1 m_{t-1}-\eta(1-\beta_1)\lambda\theta_{t-1}}{\eta(1-\beta_1)g_t}$
$= \frac{\theta_{t-1}(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)-\eta\beta_1 m_{t-1}}{\eta(1-\beta_1)g_t} - \frac{\lambda\theta_{t-1}}{g_t}$.

A.6 AdamW

For the AdamW optimizer, we have:

$$\begin{aligned} \theta_t &= \theta_{t-1} - \eta\lambda\theta_{t-1}, \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)g_t^2, \\ \widehat{m_t} &= \frac{m_t}{1-\beta_1^t}, \\ \widehat{v_t} &= \frac{v_t}{1-\beta_2^t}, \\ \theta_t &= \theta_t - \frac{\eta\widehat{m_t}}{\sqrt{\widehat{v_t}} + \epsilon} \end{aligned} \tag{14}$$

This results in: $\theta_t = \theta_{t-1} - \eta\lambda\theta_{t-1} - \frac{\eta m_t}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)} = \theta_{t-1} - \eta\lambda\theta_{t-1} - \frac{\eta\beta_1 m_{t-1}}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)} - \frac{\eta(1-\beta_1)g_t}{(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)}-$.
Similarly, if $\frac{\theta_{t-1}(1-\eta\lambda)(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)-\eta\beta_1 m_{t-1}}{\eta(1-\beta_1)g_t} > 2^n$, the last n bits of $\eta(1-\beta_1)g_t$ can be discarded safely. To quantify this, we define the metric: $\delta = \frac{\theta_{t-1}(1-\eta\lambda)(\sqrt{\widehat{v_t}}+\epsilon)(1-\beta_1^t)-\eta\beta_1 m_{t-1}}{\eta(1-\beta_1)g_t}$.