COMPSO: Optimizing Gradient Compression for Distributed Training with Second-Order Optimizers

Baixi Sun
Indiana University
Bloomington, IN, USA

Weijin Liu
Stevens Institute of Technology
Hoboken, NJ, USA

J. Gregory Pauloski
University of Chicago
Chicago, IL, USA

Jiannan Tian
Indiana University
Bloomington, IN, USA

Jinda Jia
Indiana University
Bloomington, IN, USA

Daoce Wang
Indiana University
Bloomington, IN, USA

Boyuan Zhang
Indiana University
Bloomington, IN, USA

Mingkai Zheng
Rutgers University
New Brunswick, NJ, USA

Sheng Di
Argonne National Laboratory
Lemont, IL, USA

Sian Jin
Temple University
Philadelphia, PA, USA

Zhao Zhang
Rutgers University
New Brunswick, NJ, USA

Xiaodong Yu
Stevens Institute of Technology
Hoboken, NJ, USA

Kamil A. Iskra
Argonne National Laboratory
Lemont, IL, USA

Pete Beckman
Northwestern University
Argonne National Laboratory
Lemont, IL, USA

Guangming Tan
University of Chinese Academy of Sciences
Beijing, China

Dingwen Tao*
University of Chinese Academy of Sciences
Beijing, China

Abstract

Second-order optimization methods have been developed to enhance convergence and generalization in deep neural network (DNN) training compared to first-order methods like Stochastic Gradient Descent (SGD). However, these methods face challenges in distributed settings due to high communication overhead. Gradient compression, a technique commonly used to accelerate communication for first-order approaches, often results in low communication reduction ratios, decreased model accuracy, and/or high compression overhead when applied to second-order methods. To address these limitations, we introduce a novel gradient compression method for second-order optimizers called *COMPSO*. This method effectively reduces communication costs while preserving the advantages of second-order optimization. COMPSO employs stochastic rounding to maintain accuracy and filters out minor gradients to improve compression ratios. Additionally, we develop GPU optimizations to minimize compression overhead and performance modeling to ensure end-to-end performance gains across various systems. Evaluation of COMPSO on different DNN models shows that it achieves a compression ratio of 22.1×, reduces communication time by 14.2×, and improves overall performance by 1.9×, all without any drop in model accuracy.

CCS Concepts: • **Theory of computation → Data compression**; • **Computing methodologies → Parallel algorithms**.

Keywords: Deep learning, distributed training, second-order optimization, K-FAC, data compression.

ACM Reference Format:
Baixi Sun, Weijin Liu, J. Gregory Pauloski, Jiannan Tian, Jinda Jia, Daoce Wang, Boyuan Zhang, Mingkai Zheng, Sheng Di, Sian Jin, Zhao Zhang, Xiaodong Yu, Kamil A. Iskra, Pete Beckman, Guangming Tan, and Dingwen Tao. 2025. COMPSO: Optimizing Gradient Compression for Distributed Training with Second-Order Optimizers. In *The 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3710848.3710852

1 Introduction

Today's Deep Neural Networks (DNNs) are increasingly required to process larger volumes of data due to their robustness and generality [3, 6, 9, 10, 26, 56]. On one hand, this leads to more iterations needed for training. On the other hand, the high cost of GPU hours necessitates faster training methods. As a result, sophisticated optimizers are drawing more attention, in addition to parallelism mechanisms for training. Conventional first-order optimizers, such as Stochastic Gradient Descent (SGD) [17] and ADAptive Moment estimation (ADAM) [27], have been extensively studied.

Recent advancements in second-order optimizers have proven effective, as they can achieve convergence with fewer

[*] Corresponding author: Dingwen Tao.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710852

iterations compared to first-order methods. Notable second-order optimizers include Kronecker-Factored Approximate Curvature (KFAC) [35], Broyden–Fletcher–Goldfarb –Shanno (BFGS) [15], Shampoo [18], and Generalized Gauss-Newton (GGT) [1]. Among these, the pioneering KFAC optimizer has set a paradigm [34, 36] by efficiently decomposing the Fisher Information Matrix (FIM) into invertible factors.

However, the intensive computation required for inverting the factors limits the effectiveness of the KFAC optimizer compared to first-order optimizers. To address this issue, existing approaches have designed distributed KFAC methods that parallelize the computation. These approaches divide the computational workload by layers among GPUs [35, 42], which necessitates substantial communication to synchronize the KFAC gradient across all GPUs. As shown in Figure 1, broadcast communication within a distributed KFAC framework constitutes at least 30% of the total end-to-end time and increases with model size and GPU count, making communication a performance bottleneck.

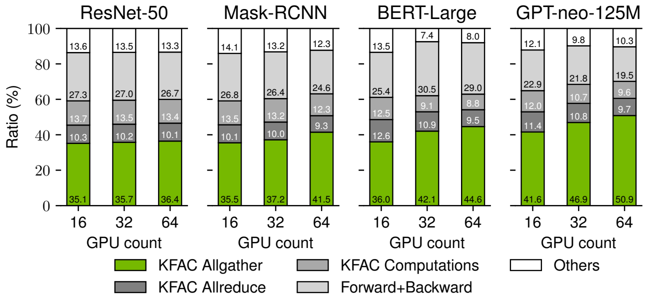


Figure 1. Time breakdown of distributed KFAC training on ResNet50, Mask R-CNN, BERT-large, and GPT-neo models with 16, 32, and 64 compute nodes (four A100 GPUs per node).

Compression is a practical approach to reducing the cost of gradient communication in distributed training. Gradient compression methods (e.g., sparsification [7, 12] and quantization [21, 38]) are widely used in Stochastic Gradient Descent (SGD)-based DNN training. These methods compress gradients in a lossy manner to reduce the size of communication data while keeping the introduced error within a predefined threshold to ensure convergence.

Error-bounded lossy compression, originally applied to large-scale scientific data, can be applied to reduce communication data [48, 60–63]. However, simply adapting existing first-order gradient compression methods to second-order optimizers like KFAC is ineffective because: ① Preserving convergence can conflict with achieving a high compression ratio. ② Lack of consideration for different system setups can limit end-to-end performance gains; and ③ Ignoring architecture-specific optimizations (e.g., GPU considerations) leads to high compression overhead, limiting overall performance gains.

To address these issues, we design COMPSO to enhance the distributed training performance with second-order optimizers through a compression method and a combination of system and algorithm co-design. To the best of our knowledge, this work is the first to explore these techniques applied to KFAC gradients and the co-design of the compression algorithm. Our contributions are summarized as follows:

- We analyze the impact of different quantization methods on KFAC convergence and select the most suitable method to develop our new compression algorithm.
- We design a novel gradient compression algorithm for KFAC, which includes an error-bounded scheme and an adaptive mechanism. These mechanisms incorporate various strategies throughout training iterations.
- We develop a performance model that guarantees end-to-end performance improvement and formulates an adaptive compression scheme for layers of varying sizes.
- We implement GPU optimizations to minimize compression and decompression overheads.
- We evaluate COMPSO with KFAC on two GPU clusters with different network configurations. Compared to the KFAC no-compression baseline, COMPSO improves communication efficiency by 14.2× and overall training speedup by 1.9× by achieving a compression ratio of 22.1× for the gradient. Additionally, compared to SGD with state-of-the-art compression, COMPSO with KFAC achieves 1.8 × overall speedup, reducing training time from 60 hours to 33 hours.

The remainder of this paper is organized as follows. §1 provides an overview of second-order optimization methods and gradient compression techniques. §2 discusses the research challenges associated with applying gradient compression to second-order optimizers. In §4, we detail the design and optimizations of our proposed compression method, COMPSO. §5 presents the results of our evaluation. §6 reviews related work in the field. Finally, §7 summarizes our findings and explores potential directions for future research.

2 Background

In this section, we introduce KFAC algorithm, distributed KFAC mechanism, gradient quantization methods, compression tools, and CUDA architecture.

2.1 Kronecker-Factored Approximate Curvature

Kronecker-Factored Approximate Curvature (KFAC), a second-order optimizer, accelerates the convergence by approximating the inversion of the Fisher Information Matrix (FIM) and achieves fewer iterations to convergence compared to first-order optimizers (e.g., SGD) [35, 41, 42, 44, 45]. This approximation comprises *two* main steps: Kronecker-product decomposition to approximate the FIM (Equation 1) and eigendecomposition (Equation 2). ① The first step utilizes the Kronecker product of the activation and SGD-gradient covariance matrices, formulated as

$$\hat{F}_l = A_{l-1} \otimes G_l = a_{l-1} a_{l-1}^{\top} \otimes g_l g_l^{\top}, \tag{1}$$

where A and G represent the covariance matrices, a_{l-1} denotes the activation data from layer $l-1$, and g is the SGD-gradient. ② The second step involves performing eigendecomposition on matrices A_{l-1} and G_l, significantly reducing computational overhead by diminishing the sizes of these matrices. The two processes approximate the FIM inversion, a preconditioner multiplied by the SGD gradient. This yields the KFAC gradient, detailed below,

$$\underbrace{\left(\left(\hat{F} + \gamma I\right)^{-1}\right)}_{\text{PRECONDITIONER}} \underbrace{\left(\sum_{i=1}^{n} \triangledown L_i \left(y, f\left(x, w_t\right)\right)\right)}_{\text{SGD GRADIENT}} = Q_G \left(\frac{Q_G^{\top} \sum_{i=1}^{n} \triangledown L_i (y, f(x, w_t) Q_A}{v_G v_A^{\top} + \gamma}\right) Q_A^{\top}, \tag{2}$$

where Q_A, Q_G and v_A, v_G are eigendecompistion results of A and G.

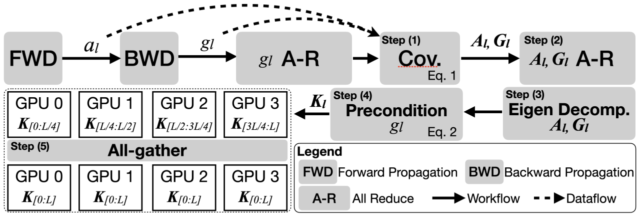


Figure 2. Work and data-flow of distributed KFAC optimizer. a_l and g_l are activation and gradient data of layer l, respectively. K and L are the KFAC gradient and the number of model layers.

2.2 Distributred KFAC

Distributed KFAC optimizers utilize layer-wise parallelism across GPUs to compute A and G and perform eigendecomposition efficiently. The workflow includes five main steps for each neural network layer: ① covariance computation of activation and gradient, respectively; ② local covariances communication using the all-reduce operation; ③ eigendecomposition on covariance matrices; ④ compute the preconditioned gradient; and ⑤ all-gather the preconditioned gradient to each worker (e.g., GPU), as shown in Figure 2. The computation workload in steps 3 and 4 are evenly split across multiple GPUs, i.e., each GPU computes a subset of all layers' KFAC gradient. Thus, in step 5, each GPU sends its computed results to all other GPUs using Allgather. Note that some KFAC implementations use broadcast instead of Allgather to overlap communication with computation.

Training with second-order optimizers follows a data-parallel fashion [42–45]. This contrasts with pipeline-parallel methods (e.g., PipeFisher [41]), which are the outcome of restricted GPU memory capacity (e.g., 16-GB P100). The data-parallel trend also coincides with the introduction of large-memory GPUs (e.g., 40-GB A100, 141-GB H200), initially for accommodating large models. These large-memory GPUs can benefit the memory-intensive KFAC optimization, ultimately speeding up the training process.

With the data-parallel requirement initially met, KAISA, a distributed KFAC approach, is proposed to optimize KFAC workflow [44]. The contributions of KAISA are threefold. ① KAISA minimizes intermediate data memory overhead by performing an Allgather of each layer's computation results immediately upon completion instead of waiting and buffering all layers. ② KAISA overlaps computation-communication overlap across layers on each GPU. ③ KAISA employs an alternate implicit inversion method for FIM to further optimize the process. In this work, we focus on KAISA as our design basis.

2.3 Gradient Quantization

Quantization represents FP32 values using fewer bits [2, 13, 24, 25]. Gradient quantization typically consists of two steps: normalization and rounding. ① An n-bit quantization considers the data range and encloses all the integers, which are transformed from the input value v, within $[-2^n, 2^n]$. To constrain all the input numbers, it is done by

$$v' = 2^n \cdot v / \max(|v_{\max}|, |v_{\min}|), \tag{3}$$

② Intuitive rounding to the nearest integer (RN) and stochastic rounding (SR) are jointly used. SR is defined as

$$v_{\text{int}} = \begin{cases} \lceil v' \rceil & \text{with probability } p \\ \lfloor v' \rfloor & \text{otherwise} \end{cases} \quad \text{where } p = \frac{v' - \lfloor v' \rfloor}{\lceil v' \rceil - \lfloor v' \rfloor}. \tag{4}$$

2.4 Representative Compression Methods

We use three representative algorithms: ① QSGD [2], a gradient compression algorithm for SGD; ② SZ [29], an error-controllable lossy compressor for FP32 data; and ③ CocktailSGD [53], a combination of sparsification and SR-based quantization. QSGD includes SR-based quantization and Elias Encoding. SZ includes prediction, RN-based quantization, and Huffman encoding [23]. SZ uses the surroundings to predict a data value and quantizes the prediction error; with the quantized error properly encoded, the data is reduced in size. Sparsification of CocktailSGD leverages the distribution of SGD gradients, recognizing that smaller values occur more frequently. It selects the most frequent values and represents the SGD gradient in a sparse format. Lossless encoding boosts the compression ratio (CR) to surpass the 1-bit quantization limit [1] if repeated patterns are presented.

3 Motivation and Challenges

As mentioned in §1, the communication-to-total-time ratio exceeds 30% in distributed KFAC, even considering the computation-communication overlap. Besides upgrading network bandwidth (which is rare and unexpectable), compression practically reduces the communication data size to communicate and, consequently, the communication overhead.

Existing SGD gradient compression algorithms compress the gradient vector in a lossy manner and have been validated

[1] This is equivalent to 32× in CR for FP32 gradients.

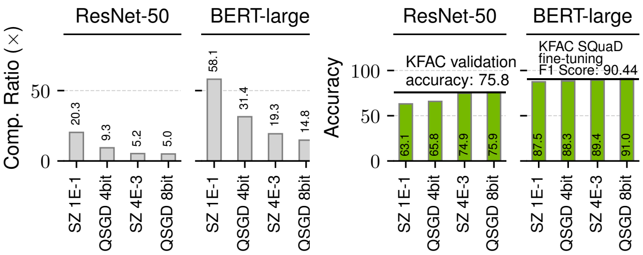


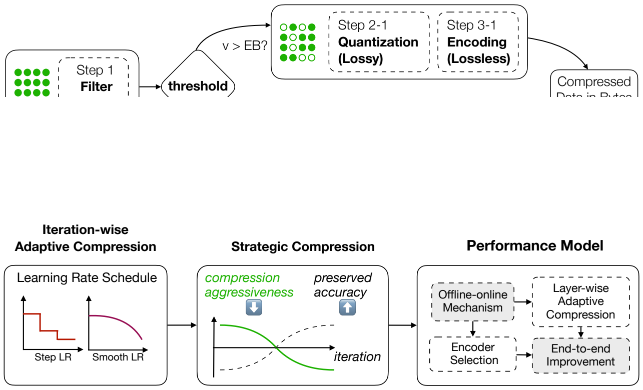
Figure 3. Compression ratio (left) and validation accuracy (right) of different solutions on ResNet50 and BERT-large, and the accuracy benchmark result with KFAC is 75.8 and 90.44, respectively.

to converge with compressed SGD gradients [2, 30, 33, 53, 55]. However, the compression ratio (CR) and impact on accuracy/convergence from lossy compression on KFAC gra ents has not been explored. We apply two state-of-the-a compression algorithms to the second-order gradients KFAC: QSGD and SZ (introduced in §2.4). Note that QSG and SZ only differ in their (lossy) quantization method, a in the remainder of the paper, we tune SZ's error bounds match the accuracy for 4/8-bit QSGD.

However, directly applying them results in either a lo compression ratio (CR) or impacted validation accuracy well as convergence, as illustrated in Figure 3 with ResN 50 [20] on ImageNet [9] and BERT-large [10] on Wiki [3 Using a lower error bound for SZ (4E-3) or more bits f QSGD (8-bit) results in higher accuracy (reaching 75.8 a 90.44) but limited compression ratios (8× to 20×) On t other hand, high error bound will cause compressors su as SZ to have higher compression error, resulting in lo validation accuracy, without our optimization (will be c tailed in §4.2-4.3) Furthermore, QSGD 4-bit failed to prese accuracy on KFAC, contrasting to the behavior on SGD[because 1 KFAC gradients have a *larger range* than SGD g dients, resulting in more scattered quantized values with the scaled range and degraded encoder performance; and KFAC gradients make more accurate and aggressive ste toward convergence [32], making them *more sensitive* the introduced errors. These necessitate a new compressi scheme to address these issues. Additionally, existing SGD gradient-compression algorithms lack architecture-specific optimizations, particularly for GPU. Effectively utilizing their parallelism and memory hierarchy is crucial. Consequently, designing GPU optimizations to reduce the (de)compression overhead is essential, ensuring that communication speedup contributes to the overall gain.

Four challenges lie ahead in developing a framework for accelerating communication in distributed KFAC training:

1. CR can be limited by the bitrate requirement to maintain validation accuracy (§ 2.3) This issue is even more pronounced for KFAC gradients, as evidenced in Figure 3.
2. Generalizing the characteristics of a system and modeling it is non-trivial, considering significant variations in network bandwidth, GPU compute capability, etc. Furthermore, applying compression complicates the modeling.
3. The mechanism of distributed KFAC misfits the circumstance when the model is split into cross-GPU layers, as the gradients vary in data sizes and range across layers.
4. Additionally, implementing compression algorithms eff



(b) System components.

Figure 4. Overview of the proposed communication-efficient second-order training framework, COMPSO.

4 COMPSO Design

In this section, we present the design of COMPSO to address the identified challenges. We first characterize the accuracy impact of different quantization approaches for various error distributions. Building on this characterization, we detail the design of our new compression algorithm, which features an iteration-wise adaptive mechanism incorporating different compression strategies across training iterations. We then devise a performance model considering many factors, including communication bandwidth, compressor cost, compression gain, etc. Furthermore, based on our performance model, we introduce a layer-wise adaptive compression mechanism for aggregated layers of varying sizes. Lastly, we describe our GPU optimizations.

4.1 Overview of COMPSO

COMPSO includes 1 a validation-accuracy-preserving high-ratio compression algorithm that hybridizes filter and stochastic rounding and adapts error bounds based on iterations, 2 a performance model, thereby ensuring end-to-end performance gain and integration with the layer-wise adaptive compression mechanism. and 3 GPU optimizations for different data sizes and value ranges by efficiently utilizing CUDA parallelism and multi-level memory hierarchy. The design is shown in Figure 4.

Specifically, the hybrid compression algorithm consists of a filter that maps gradients within a predefined threshold to zero and represents the indices with a bitmap, an error-bounded quantizer that maps FP32 values into integers, and a lossless compressor that compresses the bitmap and quantized values. It is important to note that, unlike fixed-rate gradient quantization methods (e.g., 4/8-bit), our fine-grained error-bounded quantization can balance the compression ratio and validation accuracy. The adaptive compression mechanism enables flexibility for different KFAC gradient compression scenarios, as it determines the *aggressive-conservative* compression strategy across iterations. The performance model considers the system setup and algorithm capabilities and utilizes the micro-benchmark results to estimate the end-to-end performance without a complete run of end-to-end training. Our performance model helps design future compressors for distributed training communication on various systems. GPU optimizations are implemented in two directions: enhancing memory parallelism and improving efficiency. We reduce frequent memory access by combining block reduction with warp-level shuffle techniques. High-performance parallelism is achieved by tailoring data sizes and ranges in a fine-grained manner.

4.2 Rounding Method Analysis

This section discusses the relationship between lossy compression error and validation accuracy (with the same number of iterations to convergence) using KFAC. First, the error is defined as the difference between the de-quantized value and the original value. Specifically, two established lossy compressors with the two rounding schemes (§ 2.3) are studied: ① SZ, which uses RN for quantization, and ② QSGD, which uses SN for quantization. Figure 3 shows that QSGD achieves a lower or comparable compression ratio but maintains better validation accuracy than SZ. We studied several metrics, including the L2 norm, peak signal-to-noise ratio (PSNR), and error distribution, to understand the reason. We find the (error distribution)-accuracy consistency is significant, which becomes a key point throughout this paper.

First, we analyze the distributions of RN- and SR-caused errors on KFAC gradients when training ResNet-50. We visualized the error distribution across all layers for every 50 iterations (out of 1563, BERT-large) and five epochs out of 100 (ResNet-50). Each model was trained five times, and we observed a similar distribution shape across both model layers and throughout epochs/iterations. As shown in Figure 5, while RN results in a uniform error distribution deterministically, SR results in a triangular distribution due to its probabilistic nature. We conduct extended experiments by compressing synthetic uniformly and normally distributed data. We observe that RN and SR error distribution is consistent with KFAC gradients.

To investigate if the non-deterministic feature preserves accuracy, we test another non-deterministic rounding, $P_{0.5}$ (i.e., mode-2 SR in [8]), which rounds up/down with equal probability, resulting in a uniform error distribution. This method significantly impacts accuracy at the same bit level as SR. For example, with 8-bit quantization on ResNet-50, convergence accuracy drops to 74.5%, compared to 75.8% without compression. In contrast, QSGD-8bit (SR) maintains 76.0% in accuracy.

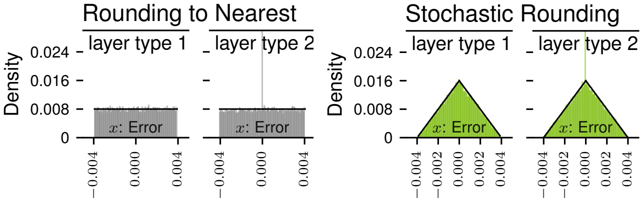


Figure 5. The distribution of KFAC gradient compression error with error bound of 4E-3 using RN (left) and SR (right) on all layers every 50 iterations. The error distributions are similar across iterations.

We conduct extensive experiments on four DNN training benchmarks using SZ and QSGD (will be detailed in §5.1), in addition to results mentioned in §3 Figure 3. The results demonstrate that QSGD better preserves accuracy than SZ (e.g., QSGD 8-bit achieves higher accuracy under a similar ratio with SZ 4E-3). This outcome is attributed to SZ, leading to a uniform error distribution, and QSGD, resulting in a triangular error distribution. Furthermore, Figure 3 shows that SZ with an error bound of 4E-3 achieves higher accuracy than 1E-1, underscoring that a smaller error bound is more conducive to accuracy preservation.

From the theoretical and experimental analysis presented above, we draw three key insights concerning validation accuracy: ① a triangular error distribution is more effective at preserving accuracy than a uniform distribution (SR versus RN and $P_{0.5}$), ② within the same class of error distribution, a smaller error bound is more beneficial (§3), and ③ whether the quantization is deterministic (RN) or non-deterministic ($P_{0.5}$) has no significant impact on accuracy or convergence. Accordingly, stochastic rounding (SR) is the superior KFAC gradient quantization technique. Thus, we focus our compression algorithm on this method.

4.3 Novel Gradient Compression Algorithm

As discussed in §3, direct quantization methods like SR preserve validation accuracy but achieve a limited compression ratio. Thus, we set to develop a new SR-based compression algorithm to ① maintain validation accuracy and ② compress the FP32 values more.

The workflow of our compression algorithm is presented on the left in Figure 4 and the algorithm is describes in Algorithm 1. First, the algorithm employs a filter to convert a subset of KFAC gradient FP32 values into a bitmap representation. Specifically, the filter selects values based on a predefined error bound, i.e., values that are less than the error bound and marks them as one in the bitmap (remaining values will be marked as zero). We use a bitmap to record the subset of KFAC gradients that are processed by the filter. After that, we apply an encoder to compress this bitmap losslessly, achieving a high compression ratio; for the remaining values that are higher than the error bound, we apply SR to preserve the validation accuracy and encoding to further improve the compression ratio. Notably, the error bound of SR (eb_q) is defined separately from eb_f. The eb_q is determined empirically to minimize the accuracy impact. For the lossless encoder, we select the best-fit GPU encoders from existing implementations, which balances the high compression ratio and (de-)compression throughput (to be detailed in §4.4).

Additionally, we design an *iteration-wise* adaptive compression mechanism that enables more aggressive compression across training iterations. Specifically, we divide the number of iterations required for convergence into multiple stages. In distributed training using KFAC optimizers, the early iterations are typically unstable, while the later iterations trend closer to convergence. This occurs because, ① in DNN training, the learning rate decreases across iterations, making early iterations less sensitive to the error introduced in KFAC gradients than later ones; and ② the covariance matrices A and G are computed as the running averages during training, becoming more stable as more training samples processed. Consequently, we use larger error bounds combining filter and SR in the early iterations and smaller error bounds applying SR only during the later iterations as the learning rate changes.

Two popular learning rate (LR) schedulers can adjust the learning rate: StepLR and SmoothLR. StepLR decays the LR at predefined steps by multiplying the base LR by a decay factor. SmoothLR decays LR by multiplying a factor by the base LR at each iteration after the warmup. For StepLR, we use both filter and SR with a large error bound before the first learning rate decreases. For SmoothLR, we divide the training process into z stages, where z is an empirically tunable parameter.

We empirically validate the design in §5, demonstrating that our approach does not significantly impact model accuracy and achieves a higher compression ratio compared to not using this mechanism. Our design differs from previous sparsification approaches, such as Ok-topk, which maintains a fixed error bound across all iterations; we adaptively vary the error bound based on the learning rate.

Unlike existing SGD gradient quantization methods at a rigidly fixed rate (i.e., 8/4/2/1-bit), our fine-grained algorithm features tunable error bounds. This is accomplished by packing bits into bytes based on the specified error bound. For instance, with an error bound set at 1e-2 to maintain validation accuracy, our method requires a maximum of 100 quantization bins, corresponding to a 7-bit representation. Each 7-bit group is then packed into bytes. In contrast, other quantization methods like QSGD necessitate 256 quantization bins for an 8-bit representation. Consequently, our approach yields a higher compression ratio by 14%.

Algorithm 1: Proposed filter and SR-based compression algorithm with adaptive iteration-wise compression.

```
Inputs  : $G$: KFAC gradient values; $eb_f$: filter error bound; $eb_q$: SR error
          bound; $LR$: learning rate schedule; $T$: total iterations; $z$: number of
          stages; $\alpha$: error bound decay factor;
Outputs : $C$: Compressed representation of $G$
// Initialize variables
$B \leftarrow \emptyset$
$C \leftarrow \emptyset$
$stage\_length \leftarrow \lceil T/z \rceil$ // Length of each stage
for $t \leftarrow 0$ to $T$ do
    // Adjust error bounds based on stage
    if $LRS == StepLR$ then
        if $t < first\_LR\_drop$ then
            $eb_f, eb_q \leftarrow$ loose bounds;
        end
        else
            $eb_f, eb_q \leftarrow$ tight bounds;
        end
    end
    if $LRS == SmoothLR$ then
        // Determine the current stage
        $current\_stage \leftarrow \lfloor t/stage\_length \rfloor$
        if $current\_stage == 0$ then
            $eb_f, eb_q \leftarrow$ loose bounds;
        end
        else
            $eb_f, eb_q \leftarrow eb_f \times \alpha, eb_q \times \alpha$;
        end
    end
    // Filter Branch: Generate Bitmap
    foreach $g \in G$ do
        if $|g| < eb_f$ then
            $B[g] \leftarrow 1$;
        else
            $B[g] \leftarrow 0$;
        end
    end
    Compress $B$ losslessly and append to $C$;
    // SR Branch: Quantize and Compress
    foreach $g \in G$ where $B[g] == 0$ do
        Quantize $g$ into using SR with $eb_q$;
        Pack quantized values into bytes and append to $C$;
    end
end
// Return final compressed representation
return $C$;
```

Our algorithm is tailored for KFAC rather than SGD due to differences in communication patterns and sensitivity to compression errors. While SGD relies on ring AllReduce, which has the error propagation issue, KFAC uses AllGather, avoiding this issue. KFAC's faster convergence also heightens its sensitivity to compression errors, requiring a balanced design for compression and convergence.

4.4 Performance Model for Optimal Compression

To secure end-to-end performance gain, we develop a performance model that incorporates impacting factors and *offline-online* mechanism. The overall performance includes communication and the incurred (de-)compression overhead. We first define the modeling space and introduce the notations below. For system $[x]$:

- L_o and L_c [MB] are the sizes of the original and compressed KFAC gradients, respectively.

- $\hat{C}_o^{[x]}$ and $\hat{C}_c^{[x]}$ [MB/s] are the reference communication throughput for the original data (of size s) and the compressed data (of size c), respectively. They are from the prebuilt lookup table for each system.
- $\hat{T}_{o,1..k}^{[x]}$ and $\hat{T}_{c,1..k}^{[x]}$ [MB/s] are the compression and decompression throughputs for the original data (of size s) and the compressed data (of size c), respectively. They are averaged from the first k iterations.
- $\hat{r}_{1..k}^{[x]}$ [%] is the ratio of the communication time to the total iteration time without compression, and this is averaged from the first k iterations.

Modeling the communication speedup with compression requires the communication throughput before/after compression. However, without online measures, we cannot know the compressed data size for each layer. Considering runtime profiling may incur overhead, we employ a mixed offline-online strategy to gain knowledge. Specifically, we benchmark communication *offline* on *each* system with synthetic data, forming a deterministic lookup table that maps communication throughput $\hat{C}^{[x]}$ to different message sizes and the GPU count. Thus, it becomes a reference for online queries for the varying data sizes (original or compressed).

Table-querying required L_o and L_c are measured with real data. Notably, COMPSO includes an encoder that is selected from a vector of candidates (detailed in § 5.2), and they achieve varying L_c. To select the best-fit encoder, we need to measure L_c and overall compressor throughputs ($\hat{T}_{o,1..k}^{[x]}$ and $\hat{T}_{c,1..k}^{[x]}$) on real data (i.e., KFAC gradients) *online*. We use the encoder with smaller L_c and low overall compression overhead. The gradient data feature is unknown before runtime, so we are set to profile only k training iteration. We also found that warmup training iterations can be representative across the training. Thus, we choose the first k warmup iterations, with negligible performance impact, to determine the stabilized compression-decompression throughput ($\hat{T}_{o,1..k}^{[x]}$ and $\hat{T}_{c,1..k}^{[x]}$) and the portion of communication ($\hat{r}_{1..k}^{[x]}$) for each system $[x]$. The communication speedup is formulated as

$$s = \underbrace{\left(\sum_{i}^{i+m} \frac{L_o}{\hat{C}_o^{[x]}} \right)}_{\text{EST. TIME WITH ORIGINAL DATA SIZE}} \div \underbrace{\left(\frac{L_c}{\hat{C}_c^{[x]}} + \frac{\sum_i^{i+m} L_o}{\hat{T}_{o,1..k,i..i+m}^{[x]}} + \frac{L_c}{\hat{T}_{c,1..k}^{[x]}} \right)}_{\text{EST. TIME FOR COMP.+DECOMP. AND COMMUNICATING COMPRESSED DATA}} \tag{5}$$

where m is the layer-aggregation factor, determined by our layer-wise compression and layer-aggregation mechanism on each GPU. Specifically, DNN models feature size-varying KFAC gradients for each layer, some of which are small and lead to GPU resources underutilization. We aggregate multiple layers before compression is employed to improve and stabilize the performance. We find the m such that the end-to-end speedup $\left(\left(1-\hat{r}_{1..k}^{[x]}\right)+\hat{r}_{1..k}^{[x]}/s\right)^{-1}$ is high. For example, with 50% of the communication to total iteration time ratio and 10× communication speedup considering compression, the end-to-end performance gain is 1.8×.

4.5 GPU Implementation and Optimizations

KFAC gradients are computed and buffered in GPU global memory during training. Hence, compressing on the GPU is essential to avoid the GPU-to-CPU data-transfer overhead, which can be unacceptable. Existing vanilla implementations of the GPU compressor, such as QSGD and CocktailSGD, degrade the system performance (to be detailed in §5.3), motivating us to develop a GPU-centric compression pipeline.

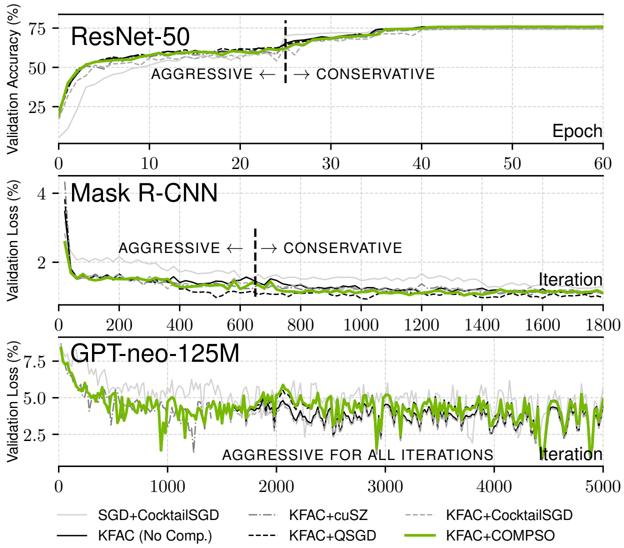
As outlined in §4.3 and Fig. 4a, the compressor consists of a filter, a quantizer, and an encoder. Given the compression is done in $O(n)$ time (if sequential), the arithmetic intensity is $O(1)$ relative to input size n, implying that the compression is essentially memory-bound with lightweight computation. The related optimizations to decrease traffic throughout the memory hierarchy are twofold. 1) We fuse the three kernels into one to decrease the memory traffic to *global* memory and improve performance. This approach allows the context to persist in local buffers (e.g., shared memory) and increases the data reuse in the memory-bound compression process. 2) In addition, we implement the fine-grained range computation (i.e., finding the extrema of a layer) in a parallel reduction manner using *block reduction* and *warp-level shuffle*. Specifically, the update frequency of global extrema in global memory can be much lowered after the local extrema are found by block reduction. Backtracing the block reduction, considering the one order of magnitude higher latency to access share memory than the warp-wide (SIMD-32) register file, warp-level shuffle is employed to decrease the block-wide local extrema update in the shared memory after each warp finds the even-finer local extrema.

Given that layer sizes vary, a fine-grain mapping of layer (data) and thread block is required. First, the shared memory buffer is padded to ensure that only gradients for one layer are processed. This also ensures that the determination of the data range (for normalization) is not mixed when aggregating multiple layers. Second, the varying layer size can result in workload imbalance, which can increase latency during training. At the same time, layer features (e.g., size) can be stable across iterations. i.e., given a list of layers whose indices are $0 \ldots N$, the occurrence of layer$[i]$ tends to be stable. Thus, a pre-determined layer-block *hashmap* can be built during the initialization of the KFAC optimizer and reused for the rest of the iterations.

5 Experimental Evaluation

Platforms. We evaluate COMPSO on two platforms: ① A 16-node cluster, each node equipped with two AMD EPYC 7742 processors, 256 GB of RAM, and four 40GB NV-Link connected NVIDIA A100 GPUs. These nodes are interconnected using Slingshot10 with a maximum bandwidth of 100 Gbps. ② A 64-node cluster, each node equipped with the same GPU configuration, one AMD EPYC Milan 7543P processor, and 512 GB of RAM. These nodes are interconnected using Slingshot11 with a maximum bandwidth of 200 Gbps.

Baselines. We compare COMPSO with three state-of-the-art compression approaches: QSGD [2], cuSZ(A GPU version of the SZ algorithm) [52], and CocktailSGD [53]. QSGD and SZ algorithms are introduced in §2.4. As CocktailSGD is a fine-tuning solution rather than training from scratch, we focus more on its compression performance in ratio and throughput and performance gain in training than on validation accuracy. For COMPSO, we fix the tunable parameter aggregation factor to be 4 for all cases according to our performance model. In addition, we utilize the state-of-the-art distributed KFAC optimizer, KAISA [44], to enhance memory efficiency for large-batch training.



(a) Convergence Curve Comparison between SGD+CocktailSGD, KFAC without compression, KFAC+cuSZ,iKFAC+QSGD, KFAC+CocktailSGD, KFAC+COMPSO on ResNet-50, Mask R-CNN, and GPT-neo-125M.

Method	Error Control	ResNet -50	Mask R-CNN	GPT-neo -125M
SGD+CocktailSGD	20% sparsity +8-bit quant.	74.08%	1.32	5.67
KFAC (No Comp.)	(n/a)	75.80%	1.12	4.44
KFAC+cuSZ	4E-3, relative to value range	74.88%	1.07	(failed)
KFAC+QSGD	8-bit quant.	75.96%	0.98	4.98
KFAC+CocktailSGD	20% sparsity +8-bit quant.	74.47%	1.13	4.31
KFAC+COMPSO	iteration-wise adaptive	75.90%	1.11	4.88
		Accuracy	Loss	Loss

(b) Validation Metric Value for {ResNet-50, Mask R-CNN, GPT-neo-125M}.

Figure 6. Convergence Evaluation using four compressors on KFAC on three models, comparing to KFAC without compression baseline. Additionally, SGD with CocktailSGD compressor.

DNN models, datasets, and benchmark. We evaluate the convergence and performance of COMPSO using four representative and widely-used models, two CNN-based and two transformer-based: ResNet-50 [20], Mask R-CNN [19], BERT-large [10], and a GPT-3 style model, GPT-neo-125M [11]. The first three are from NVIDIA [39] and GPT from EleutherAI [4]. This selection demonstrates the versatility and effectiveness of COMPSO for various DNNs. Specifically, we train ResNet-50 on ImageNet, Mask R-CNN on the Microsoft COCO [31], BERT-large-uncased on the `enwiki` [37] and Toronto BookCorpus datasets [5] , and GPT-neo-125M on the Pile [14]. Additionally, we use widely-received downstream task dataset and benchmark SQuAD v1.1 [47] to evaluate the BERT-large-uncased model quality.

5.1 Evaluation of Convergence

We present COMPSO's impact on convergence in Figure 6a, along with its auxiliary Figure 6b, and Table 1. We observe it to have minimal effect on KFAC convergence, on ResNet-50, Mask R-CNN, and GPT-neo-125M, using 64 GPUs on Platform 1. Figure 6a and its auxiliary Table 6b show the convergence and final iteration metric values averaged by multiple runs of ResNet-50, Mask R-CNN, and GPT-neo-125M, comparing with cuSZ, QSGD, and CocktailSGD. Table 1 show the SQuAD v1.1 BERT-large benchmark results that evaluate the model quality. Additionally, we apply CocktailSGD to SGD optimizers on the four models to demonstrate the effectiveness of KFAC over SGD with compressors. Experiments are configured to use the same number as the baseline (without compression) for the iterations to convergence, with the validation metrics as close as possible. This allows a comparative performance analysis as outlined in §5.4-5.2. As shown in Figure 6a, without compression, the SGD optimizer uses more iterations than the KFAC optimizer for convergence: 60 vs. 40 epochs, 1800 vs. 1000 iterations, and 5000 vs. 3000 iterations on ResNet-50, Mask R-CNN, and GPT-neo-125M, respectively. This results in KFAC's 1.3×, 1.2×, and 1.5× end-to-end speedup over SGD. For BERT-large, SGD-based optimizer (i.e., LAMB[57]) uses 1563 iterations to convergence, whereas KFAC-based optimizer uses 1000 iterations, 1.3× over SGD. Moreover, applying CocktailSGD to SGD results in preserved convergence compared to the case without compression. Therefore, SGD+CocktailSGD uses more iterations than KFAC and KFAC+compressors. Additionally, compared to SGD+CocktailSGD, KFAC+COMPSO achieves 15% to 50% end-to-end performance gain.

For the KFAC baseline and with compressors, ResNet-50 and Mask R-CNN employ StepLR for KFAC, with the first learning rate decrease occurring at epoch 25 and iteration 650, respectively. Therefore, we apply aggressive compression with an error bound 4E-3 prior to the first learning rate drop, then switch to conservative compression with an error bound 2E-3 for the remaining epochs/iterations. Table 6b shows that cuSZ's accuracy is significantly lower than QSGD, demonstrating SR's superiority over RN in preserving accuracy. COMPSO first aggresses in compression with error

Table 1. Comparing SQuAD result of KFAC with different compression methods. Underlined are the targets (without compression), and the shading denotes close to the target.

Approach	Equivalent Error Control	F1 Score	Exact Match
SGD+CocktailSGD	20% sparsity +8-bit quant.	90.48	83.80
KFAC (No Comp.)	(n/a)	90.44	83.78
KFAC+cuSZ	4E-3, relative to value range	89.41	82.40
KFAC+QSGD	8-bit quant.	91.01	84.10
KFAC+CocktailSGD	20% sparsity +8-bit quant.	90.31	83.39
KFAC+COMPSO	iteration-wise adaptive	90.27	83.37

bound 4E-3 for filtering and quantization, then conserves with SR-only mode at the same error bound, effectively maintaining accuracy on both models. Notably, cuSZ's accuracy significantly suffers when the error bound exceeds 1E-2, and QSGD's with less than 8-bit quantization.

For GPT-neo-125M with cosine LR, we apply aggressive compression for the first 5,000 iterations. COMPSO maintains a similar validation loss curve to training without compression. In the case of BERT-large, pre-training encompasses 1,000 iterations, segmented into four stages of 250 iterations each. BERT results are presented in Table 1. The F1 score and exact-match ratio (higher is better), indicate that QSGD 8-bit and CocktailSGD 8-bit SR quantization surpasses cuSZ 4E-3 RN quantization in accuracy preservation, while COMPSO has a minimal impact on model quality by refining the error bound from 4E-3 in stage 1 to 2E-3 in stage 4. This demonstrates that SR is more effective than RN, and COMPSO maintains good model quality using SR.

5.2 Communication Performance Gain

We present the communication speed up during aggressive compression iterations in Figure 7. The communication time excludes any compression-decompression overhead. Specifically, COMPSO achieves up to 14.5×/11.2× (11.0×/7.2× on average) on the two platforms, respectively. The speedup is limited by the accuracy-preserving settings for cuSZ at 4E-3 and QSGD with 8-bit due to their low compression ratios (CR). Compared to the baseline without compression, COMPSO achieves up to a 14.15× speedup on BERT-large using 64 GPUs, attributed to a high overall CR by its initial aggressive compression. With a slower network (e.g., Slingshot 10), the speedup is greater than with a faster network (e.g., Slingshot 11) and thus benefits more from a high CR. Furthermore, as GPU counts increase, the speedup is even greater due to the high compression ratio and layer aggregation based on our performance model. Specifically, COMPSO achieves average CR of 18.95×/ 23.52×/22.05×/18.41× for ResNet-50/Mask R-CNN/ BERT-large/GPT-neo-125M, respectively. In comparison, cuSZ achieves compression ratios of 6.04×/7.04×/15.98×/5.63×, while QSGD achieves 4.97×/5.73×/14.77×/4.87×. COMPSO outperforms CocktailSGD on Mask R-CNN and BERT-large in communication efficiency and end-to-end speedup because of the advantageous CR.

Table 2. Overall compression ratio (CR), compression throughput (C-GB/s), and decompression throughput (D-GB/s) on KFAC gradient data when training ResNet-50 (left) and BERT-large (right). The shading represents the optimal compressor compared to others.

ResNet-50			Encoder	BERT-large		
C-GB/s	CR	D-GB/s		C-GB/s	CR	D-GB/s
10.73	18.95	7.63	**ANS**	43.52	22.05	93.85
4.13	14.96	3.81	**Bitcomp**	108.16	14.04	34.29
2.31	11.21	2.42	**Cascaded**	10.34	10.70	16.66
0.21	20.72	0.09	**Deflate**	0.39	22.68	1.20
0.44	20.11	0.26	**Gdeflate**	0.39	22.53	2.53
0.22	13.52	0.24	**LZ4**	0.46	14.30	1.43
0.44	13.90	0.22	**Snappy**	0.48	14.65	2.23
0.13	21.57	0.13	**Zstd**	0.27	23.76	0.76

Moreover, the communication speedups by CocktailSGD are lower than COMPSO on ResNet-50 and GPT-neo-125M due to our proposed layer aggregation strategy, which enhances efficiency when both approaches have a similar compression ratio of ≈ 20×. For instance, while COMPSO achieves a compression ratio of 18.95×, slightly lower than CocktailSGD's 20×, our method still secures a more substantial communication speedup due to layer-wise adaptive compression. This leads to a higher overall performance enhancement on ResNet-50 and GPT-neo-125M.

Next, we analyze the performance by examining the compression ratios. Our strategy for the lossy compression component has already been determined but has yet to be decided for the encoder. Specifically, we consider eight encoders (lossless compressors) from NVIDIA nvCOMP[40]: ANS, Bitcomp, Cascaded, Deflate, Gdeflate, LZ4, Snappy, and Zstd. It is crucial to note that different data types affect the compression ratio and the throughput of compression/decompression. Therefore, we must select the appropriate encoder for each model during the sampled iterations after warmup. The throughput and overall compression ratios for ResNet-50 and BERT-large are presented in Table 2. To simplify the demonstration, we present the throughput and overall compression ratios using representative models, namely a CNN and a transformer-based language model. We observe that compressors incorporating entropy coding (e.g., ANS, Deflate, and Zstd) achieve higher compression ratios than those based on dictionary matching (e.g., LZ4, Snappy) or run-length coding (e.g., Cascaded). This is attributed to the gradient distribution's non-uniformity.

ANS stands out for its higher compression/decompression throughput, attributable to its fewer operations compared to other algorithms and its capability for parallel execution on GPUs via a block processing scheme, as discussed in [54]. For Bitcomp and GDeflate, we find limited widely acknowledged documentation. Our results indicate that Bitcomp delivers high throughput but a lower compression ratio. GDeflate, a variant of Deflate, achieves a high compression ratio through entropy coding but low throughput (similar to Deflate). In these cases, ANS is the overall best encoder.

Furthermore, our compression ratio (i.e., >22) significantly surpasses cuSZ and QSGD when the accuracy does not drop considerably (i.e., below 10), as shown in Figure 3. In addition, the compression ratios of COMPSO are slightly higher than CocktailSGD, which maintains a constant ratio of 20 (by fixing the sparsity at 20% and using 8-bit quantization bits). Specifically, CocktailSGD employs Top-k with random sampling for sparsification before quantization, while COMPSO uses a relative threshold to filter values. The advantage of our method is adaptively filtering values based on their range rather than consistently zeroing out 20% elements.

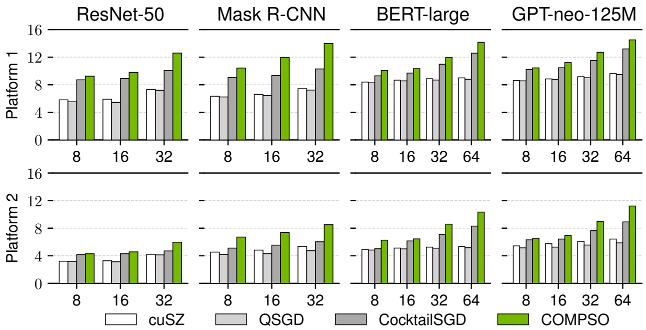


Figure 7. Comparison of communication speedup (y-axis) of cuSZ, QSGD, CocktailSGD, and COMPSO compressed KFAC gradients on ResNet-50 and BERT-large with different GPU counts (x-axis).

5.3 GPU Performance Gain

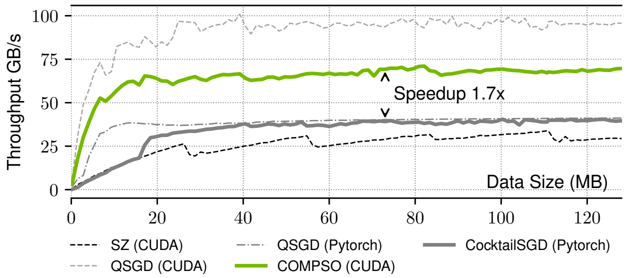


Figure 8. Comparison of GPU performance of cuSZ (SZ CUDA), QSGD, CocktailSGD, and COMPSO on A100 with various data sizes.

We also present a comparison of GPU performance between QSGD using PyTorch's `torch.cuda()` [46], the CUDA implementation of SZ (i.e., cuSZ [52]), QSGD, CocktailSGD with `torch.cuda()`, and COMPSO in Figure 8. Note that the compression throughput may vary with input data, so we report their average throughput across our tested datasets.

Our CUDA implementation of QSGD offers higher throughput than its PyTorch counterpart. This is because PyTorch launches multiple kernels for CUDA tensor operations[22], whereas our approach fuses kernels to reduce kernel launch overhead and minimize across-GPU memory data movement time. Furthermore, our implementation of QSGD exhibits

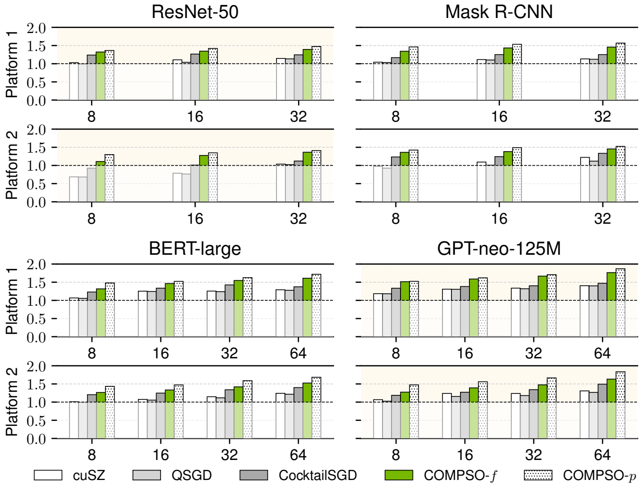


Figure 9. Overall performance gain of cuSZ, QSGD, CocktailSGD, and COMPSO scaled by GPU counts. The 1.0× speedup threshold is marked in each plot.

higher throughput compared to COMPSO. This is because QSGD performs fewer operations by omitting the filter despite a lower compression ratio. In addition, COMPSO is 1.7× faster than CocktailSGD due to the latter's relatively slow Top-k sparsification with random sampling and its implementation in PyTorch. As a result, COMPSO enhances communication efficiency and overall end-to-end performance, even in scenarios where CocktailSGD achieves a marginally higher compression ratio than COMPSO[2].

5.4 End-to-End Training Performance Gain

We evaluate the overall speedup of COMPSO in two ways, ① with the fixed-to-4 aggregation factor (denoted COMPSO-f) and ② with our performance model of dynamic aggregation factor (denoted COMPSO-p). As illustrated in Figure 9, COMPSO achieves up to a 1.9× (1.3× average) overall performance gain when training the four models on our platforms, compared to training without compression. The average training time before COMPSO of ResNet-50, Mask R-CNN, BERT-large, and GPT-neo-125M on the two platforms using 8 GPUs is 5, 1, 54, and 1 hours, respectively. [3] This reduces the training time for the four models to 3.5, 0.7, 36, 0.7 hours, respectively. Compared to cuSZ, and QSGD COMPSO demonstrates superior performance due to its greater communication reduction and higher compression throughput thanks to the communication speedup, our GPU optimizations (discussed in §5.2-5.3). Moreover, with the increasing GPU amount, COMPSO's performance gain over CocktailSGD increases from 10% to 40%, mainly attributed to our efficient aggregation strategy, GPU optimizations, and performance model.

[2] The local-then-global block reduction with warp shuffle boosts GPU performance when CocktailSGD achieves a slightly higher compression ratio.
[3] Fully pre-training GPT-neo-125M requires >120 hours on 8 A100 GPUs. We demonstrate COMPSO's effectiveness using 5000 iterations (1 hour).

Note that without our performance model (i.e., COMPSO-*f*), COMPSO achieves up to 1.8× and 1.6× speedup (1.4× and 1.3× on average) on the two platforms, respectively. With the performance model enabled (i.e., COMPSO-*p*), COMPSO achieves up to 1.9× and 1.8× speedup (1.5× and 1.4× on average) on the platforms, respectively. This highlights the importance of dynamically adjusting the aggregation factor using our performance model, as discussed in §4.4. Specifically, a fixed layer aggregation factor with varying layer sizes can result in the aggregated size being either too small or too large for optimal end-to-end speedup. Additionally, the performance model's computational overhead is minimal compared to the total training time - for instance, 2 minutes out of 42 minutes (5%) of training for KFAC+COMPSO on GPT-neo-125M. Compared to SGD+CocktailSGD, KFAC+COMPSO achieves up to 2.5× (1.8× in average) speedup. Specifically, this reduces the training time from 6, 1.2, 60, and 1.3 hours to 4.6, 0.8, 33, and 0.5 hours for ResNet-50, Mask R-CNN, BERT-large, and GPT-neo-125M, respectively.

6 Related Work

There are several ways to conduct SGD gradient quantization: sparsification and their combination. Additionally, there are other KFAC optimizations.

Quantization methods. Two primary quantization methods reduce FP32 data to fewer bits: ① rounding to the nearest (RN) and ② stochastic rounding (SR). For instance, 3LC [30] and TernGrad [55] use RN to quantize original values into one-bit and two-bits, respectively. QSGD [2] and QSDP [33] employ SR for quantizing normalized values into a pre-defined number of bits. However, using quantization alone either has limited reduction on required bits or significantly affects convergence. Thus, the error feedback (EF) mechanism is proposed to compensate for the quantization error (recovered value minus original value). These methods store errors locally and add them back in the subsequent training steps [16, 30], necessitate additional GPU memory or lead to CPU-GPU memory copy overhead. Our work does not use error feedback to facilitate large batch training with data parallelism without risking out-of-memory errors.

Sparsification. Sparsification analyzes the SGD gradient value distribution to obtain value frequency and select top frequent values in sparse format to represent full gradients, such as Top-k [51], GaussianK [49], and OK-topK [28]. Successive works either employ sparsification with different granularity (e.g., row/column-wise [59]) or utilize momentum techniques to preserve convergence while increasing sparsity [50, 58]. These approaches need to rigidly control sparsity to preserve convergence, limiting the improvement. In contrast, we apply error-bounded filtering in the selected iterations based on the learning rate change, providing better communication message size reduction.

Combining both. Recent works in SGD gradient compression combine various approaches, with CocktailSGD [53] being one of the most notable. This method integrates random sampling, Top-k selection, and RN-based quantization. The distinction between COMPSO and these methods is threefold: ① We focus on KFAC (second-order) gradient compression instead of SGD (first-order) gradient, where KFAC has more accurate directions and aggressive steps toward convergence [32], thus more sensitive to compression error. COMPSO introduces an error-bounded compression algorithm that combines filter and SR to preserve convergence. ② We design an adaptive compression strategy that preserves accuracy (iteration-wise) and enhances compression throughput (layer-wise). ③ We optimize GPU performance carefully to enhance end-to-end performance on HPC systems, which often have greater communication bandwidth and are thus more sensitive to the compressor throughput.

Other KFAC optimizations. PipeFisher [41] introduced a mechanism for pipeline parallelism in KFAC, aiming to minimize GPU idle time by integrating the KFAC computations into the idle periods of pipeline parallelism. PipeFisher operates under the assumption that models trained with KFAC exceed the memory space of a single GPU, as evidenced by P100 and V100 GPUs (16 GB memory). However, the effectiveness of PipeFisher might be limited. Firstly, modern GPUs like Nvidia A100 and H100 offer substantial memory capacities ranging from 40GB to 94GB, which is more than sufficient for large models validated as effective with KFAC. Secondly, models that have demonstrated faster convergence with KFAC can easily fit within the memory capacities of A100 GPUs. Thus, pipeline parallelism might not be essential for KFAC. In this work, we align with previous studies that focus primarily on data parallelism for KFAC.

7 Conclusion and Future Work

We introduced COMPSO, a novel framework for distributed KFAC optimizers. This framework features a new KFAC gradient compression algorithm with iteration- and layer-wise adaptive compression strategies, as well as GPU optimizations aimed at enhancing end-to-end performance. Experimental evaluation shows that COMPSO achieves up to 14.2× communication speedup and 3.1× in overall performance.

Future work will focus on: ① Precisely optimizing filter thresholds and quantization error bounds, moving beyond empirical settings; ② Exploring compression techniques for intermediate data in KFAC, specifically the factor matrices A and G, to further enhance overall efficiency.

Acknowledgment

The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. This work was also supported by the National Science Foundation (Grant Nos. 2312673, 2247060, 2303064, 2326494, and 2326495). Dingwen Tao and Guangming Tan were supported by Ant Group and the National Natural Science Foundation of China (Grant Nos. 62032023 and T2125013). Dingwen Tao contributed to this work while he was at Indiana University.

References

[1] Naman Agarwal, Brian Bullins, and Elad Hazan. 2016. Second-order stochastic optimization in linear time. *stat* 1050 (2016), 15.

[2] 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).

[3] Oana Balmau. 2022. Characterizing I/O in machine learning with mlperf storage. *ACM SIGMOD Record* 51, 3 (2022), 47–48.

[4] Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[5] Aligning Books. 2015. Movies: Towards Story-like Visual Explanations by Watching Movies and Reading Books—Yukun Zhu. In *Ryan Kiros, Richard Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba and Sanja Fidler—Proceedings of the IEEE international conference on computer vision*. 19–27.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

[7] Yi Cai, Yujun Lin, Lixue Xia, Xiaoming Chen, Song Han, Yu Wang, and Huazhong Yang. 2018. Long live time: improving lifetime for training-in-memory engines by structured gradient sparsification. In *Proceedings of the 55th Annual Design Automation Conference*. 1–6.

[8] Matteo Croci, Massimiliano Fasi, Nicholas J Higham, Theo Mary, and Mantas Mikaitis. 2022. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science* 9, 3 (2022), 211631.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, Miami FL USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[10] 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).

[11] EleutherAI. [n. d.]. EleutherAI/gpt-neo-125m · Hugging Face. https://huggingface.co/EleutherAI/gpt-neo-125m.

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

[13] Hao Feng, Boyuan Zhang, Fanjiang Ye, Min Si, Ching-Hsiang Chu, Jiannan Tian, Chunxing Yin, Summer Deng, Yuchen Hao, Pavan Balaji, et al. 2024. Accelerating Communication in Deep Learning Recommendation Model Training with Dual-Level Adaptive Lossy Compression. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[14] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2020. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027 [cs.CL]

[15] Donald Goldfarb, Yi Ren, and Achraf Bahamou. 2020. Practical quasi-newton methods for training deep neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 2386–2396.

[16] Eduard Gorbunov, Dmitry Kovalev, Dmitry Makarenko, and Peter Richtárik. 2020. Linearly converging error compensated SGD. *Advances in Neural Information Processing Systems* 33 (2020), 20889–20900.

[17] Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, and Peter Richtárik. 2019. SGD: General analysis and improved rates. In *International conference on machine learning*. PMLR, 5200–5209.

[18] Vineet Gupta, Tomer Koren, and Yoram Singer. 2018. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*. PMLR, 1842–1850.

[19] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2018. Mask R-CNN. arXiv:1703.06870 [cs.CV]

[20] 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.

[21] Samuel Horváth, Dmitry Kovalev, Konstantin Mishchenko, Peter Richtárik, and Sebastian Stich. 2022. Stochastic distributed learning with gradient quantization and double-variance reduction. *Optimization Methods and Software* (2022), 1–16.

[22] Lihan Hu, Jing Li, and Peng Jiang. 2024. cuKE: An Efficient Code Generator for Score Function Computation in Knowledge Graph Embedding. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 903–914. https://doi.org/10.1109/IPDPS57955.2024.00085

[23] David A Huffman. 1952. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101.

[24] Jinda Jia, Cong Xie, Hanlin Lu, Daoce Wang, Hao Feng, Chengming Zhang, Baixi Sun, Haibin Lin, Zhi Zhang, Xin Liu, et al. 2024. SDP4Bit: Toward 4-bit Communication Quantization in Sharded Data Parallelism for LLM Training. *arXiv preprint arXiv:2410.15526* (2024).

[25] Haoyu Jin, Donglei Wu, Shuyu Zhang, Xiangyu Zou, Sian Jin, Dingwen Tao, Qing Liao, and Wen Xia. 2023. Design of a Quantization-Based DNN Delta Compression Framework for Model Snapshots and Federated Learning. *IEEE Transactions on Parallel and Distributed Systems* 34, 3 (2023), 923–937.

[26] Pritzel A. et al. Jumper J., Evans R. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596 (2021), 583–589.

[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[28] 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.

[29] Xin Liang, Kai Zhao, Sheng Di, Sihuan Li, Robert Underwood, Ali M. Gok, Jiannan Tian, Junjing Deng, Jon C. Calhoun, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2022. SZ3: A Modular Framework for Composing Prediction-Based Error-Bounded Lossy Compressors. *IEEE Transactions on Big Data* (2022), 1–14.

[30] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2019. 3lc: Lightweight and effective traffic compression for distributed machine learning. *Proceedings of Machine Learning and Systems* 1 (2019), 53–64.

[31] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

[32] Hong Liu, Zhiyuan Li, David Leo Wright Hall, Percy Liang, and Tengyu Ma. 2024. Sophia: A Scalable Stochastic Second-order Optimizer for Language Model Pre-training. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=3xHDeA8Noi

[33] Ilia Markov, Adrian Vladu, Qi Guo, and Dan Alistarh. 2023. Quantized Distributed Training of Large Models with Convergence Guarantees. *arXiv preprint arXiv:2302.02390* (2023).

[34] James Martens, Jimmy Ba, and Matt Johnson. 2018. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*.

[35] James Martens and Roger Grosse. 2015. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*. PMLR, 2408–2417.

[36] James Martens and Roger Grosse. 2015. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*. PMLR, 2408–2417.

[37] Meta. [n. d.]. Data dump torrents - Meta. https://meta.wikimedia.org/wiki/Data_dump_torrents#English_. (Accessed on 04/07/2023).

[38] Giorgi Nadiradze, Amirmojtaba Sabour, Peter Davies, Shigang Li, and Dan Alistarh. 2021. Asynchronous decentralized SGD with quantized and local updates. *Advances in Neural Information Processing Systems* 34 (2021), 6829–6842.

[39] NVIDIA. [n. d.]. NVIDIA/DeepLearningExamples: State-of-the-Art Deep Learning scripts organized by models - easy to train and deploy with reproducible accuracy and performance on enterprise-grade infrastructure. https://github.com/NVIDIA/DeepLearningExamples.

[40] NVIDIA. 2024. NVCOMP | NVIDIA Developer. https://developer.nvidia.com/nvcomp. (Accessed on 01/14/2024).

[41] Kazuki Osawa, Shigang Li, and Torsten Hoefler. 2023. PipeFisher: Efficient Training of Large Language Models Using Pipelining and Fisher Information Matrices. *Proceedings of Machine Learning and Systems* 5 (2023).

[42] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. 2019. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12359–12367.

[43] J Gregory Pauloski, Lei Huang, Weijia Xu, Kyle Chard, Ian T Foster, and Zhao Zhang. 2022. Deep Neural Network Training With Distributed K-FAC. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3616–3627.

[44] J Gregory Pauloski, Qi Huang, Lei Huang, Shivaram Venkataraman, Kyle Chard, Ian Foster, and Zhao Zhang. 2021. Kaisa: an adaptive second-order optimizer framework for deep neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[45] J Gregory Pauloski, Zhao Zhang, Lei Huang, Weijia Xu, and Ian T Foster. 2020. Convolutional neural network training with distributed K-FAC. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[46] PyTorch. 2024. TORCH.CUDA. https://pytorch.org/docs/stable/cuda.html.

[47] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[48] Bharath Ramesh, Qinghua Zhou, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar K Panda. 2022. Designing Efficient Pipelined Communication Schemes using Compression in MPI Libraries. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, IEEE, 95–99.

[49] Shaohuai Shi, Xiaowen Chu, Ka Chun Cheung, and Simon See. 2019. Understanding top-k sparsification in distributed deep learning. *arXiv preprint arXiv:1911.08772* (2019).

[50] Navjot Singh, Deepesh Data, Jemin George, and Suhas Diggavi. 2021. Squarm-sgd: Communication-efficient momentum sgd for decentralized optimization. *IEEE Journal on Selected Areas in Information Theory* 2, 3 (2021), 954–969.

[51] Nikko Ström. 2015. Scalable distributed DNN training using commodity GPU cloud computing. (2015).

[52] Jiannan Tian, Sheng Di, Kai Zhao, Cody Rivera, Megan Hickman Fulp, Robert Underwood, Sian Jin, Xin Liang, Jon Calhoun, Dingwen Tao, et al. 2020. Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 3–15.

[53] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. 2023. CocktailSGD: Fine-tuning foundation models over 500Mbps networks. In *International Conference on Machine Learning*. PMLR, 36058–36076.

[54] André Weißenberger and Bertil Schmidt. 2019. Massively parallel ans decoding on gpus. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[55] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. *Advances in neural information processing systems* 30 (2017).

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

[57] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Syx4wnEtvH

[58] Hao Yu, Rong Jin, and Sen Yang. 2019. On the Linear Speedup Analysis of Communication Efficient Momentum SGD for Distributed Non-Convex Optimization. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 7184–7193. https://proceedings.mlr.press/v97/yu19d.html

[59] Jiaqi Zhang, Keyou You, and Lihua Xie. 2021. Innovation Compression for Communication-efficient Distributed Optimization with Linear Convergence. arXiv:2105.06697 [math.OC]

[60] Qinghua Zhou, Quentin Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K DK Panda. 2022. Accelerating Broadcast Communication with GPU Compression for Deep Learning Workloads. In *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, IEEE, 22–31.

[61] Qinghua Zhou, Quentin Anthony, Lang Xu, Aamir Shafi, Mustafa Abduljabbar, Hari Subramoni, and Dhabaleswar K DK Panda. 2023. Accelerating distributed deep learning training with compression assisted allgather and reduce-scatter communication. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, 134–144.

[62] Q Zhou, C Chu, NS Kumar, Pouya Kousha, Seyedeh Mahdieh Ghazimirsaeed, Hari Subramoni, and Dhabaleswar K Panda. 2021. Designing high-performance mpi libraries with on-the-fly compression for modern gpu clusters. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 444–453.

[63] Qinghua Zhou, Pouya Kousha, Quentin Anthony, Kawthar Shafie Khorassani, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. 2022. Accelerating MPI all-to-all communication with online compression on modern GPU clusters. In *International Conference on High Performance Computing*. Springer, Springer, 3–25.